DOI 10.12694/scpe.v14i4.932
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 14, Number 4, pp. 265–277. http://www.scpe.org

EVALUATING A FILE FRAGMENTATION SYSTEM FOR MULTI-PROVIDER CLOUD STORAGE

MASSIMO VILLARI, ANTONIO CELESTI, MARIA FAZIO, AND ANTONIO PULIAFITO *

Abstract. Currently, storage services represent a new way to do business in Cloud computing. This new trend is proved by the number of Cloud storage providers that are appearing on the market. In this work, we present an innovative approach useful for using different Cloud storage providers in a transparent way, avoiding both data lock-in and possible data privacy violation that can be caused by providers themselves. More specifically, we propose an approach enabling Cloud customers to rely on many Cloud storage providers. Differently from other solutions, with our approach only the customers have the full control of their data, and in addition, if a provider suddenly disappears and/or it is not available anymore, the customers will be able to continue accessing their data, reconstructing them from data fragments replicated in other Cloud storage providers. The paper shows how such an approach works. In particular, experiments, besides proving the goodness of our approach, also provide several guidelines regarding how to properly configure software systems in order to meet the customer's requirements (in terms of both QoS and costs).

Key words: Cloud Computing, Storage, Big Data, Reliability, Confidentiality.

1. Introduction. Nowadays, Cloud storage service is a very challenging topic, since it allows to store huge amount of data into different providers. The business behind the Cloud storage service is evident due the increasingly availability of storage providers (e.g., Dropbox, Google Drive, Copy, Amazon S3, SkyDrive, and so on). However, from the customer point of view, it is hard to choose the best offers, manage several accounts, and move data across multiple Cloud providers. Moreover, despite Cloud providers warranties, users' privacy could be compromised. From our point of view Cloud storage solutions lacks of a strong level of security and privacy [2], [3], [13]. In order to address such a problem, we propose to disseminate pieces of data among several Cloud providers that only the utilizer will be able to reconstruct.

In this work, we introduce an abstraction layer that works above heterogeneous Cloud storage providers. The benefits of the proposed strategies are multiple. Firstly, customers do not need to take care about a specific provider for data upload/download. They experience the storage service as a seamless service, where storage space is almost the sum of the storage spaces offered by the involved Cloud providers. Secondly, Cloud providers cannot have full access to the stored files, because each one is split in many chunks, that are stored into different Cloud providers. The technique we adopt in our approach that aims to avoid misusing of personal data is called *Data Obfuscation*.

Our solution is able to recover an rebuild the original file even if an error in a Cloud storage provider occurs (e.g., the operator fails). This is possible by means of the Redundant Residue Number System (RRNS) algorithm, that allows to split each file in several chunks that are called "residue-segments", including a redundancy code. Redundancy guarantees the recover of an original file when one or more residue-segments are missing. Before uploading data, the user selects the level of redundancy along with the Cloud providers involved in the storage service. Each residue-segment is BASE-64 encoded, attached within an XML wrapper and described through an XML metadata file, called *map-file*. Data and metadata are spread over different Cloud storage providers. The map-file tracks where the residue-segments are stored, in order to reconstruct the original file when the user requires to retrieve it. Following this approach, only the user can reconstruct the XML metadata combining the partial metadata coming from the two trusted providers, hence retrieve residue-segments, and rebuild the original files. This paper, extends our previous work [14], providing a more in-depth analisys about the proposed approach and further experimental results.

The paper is organized as follows. Section 2 describes related works, highlighting the lack of a resilient and confidential multi-provider Cloud storage service. Section 3 motivates this work according to the current trend on Cloud storage services. Section 4 briefly describe the RRNS algorithm on which our approach is based. Our solution for a reliable and confidential multi-provider Cloud storage service is described in Sect. 5. Experimental evaluations are discussed in Sect. 6. Finally, our conclusion are summarized in Sect. 7.

*DICIEMA, University of Messina,C.Da Di Dio 1, 98166, Messina, Italy (mvillari, acelesti, mfazio, apuliafito)@unime.it.

2. Related Work. Many works in literature deal with data reliability in data centers and in Cloud Infrastructure as a Service (IaaS). A well known solution is the Google File System (GFS), in which a file chunk replication mechanism is used [6]. Specifically, Google thought to make up a redundant storage of massive amounts of data on cheap and unreliable computers. The file chunk replication strategy is also at the basis of our solution.

In [1], the authors claim the improvement of file reliability by introducing redundancies into a large storage system exploiting different solutions, such as erasure correcting codes (used in RAID levels 5 and 6), introducing several data placement, failure detection and recovery disciplines inside data centers.

In [10], a data restore is accomplished using regenerating codes. In such a work, both redundancy and check controls are used to guarantee the possibility to repair data during file transfer over unreliable networks.

How to store pieces of file into Virtual Machines (VMs) is discussed in [11]. The authors introduce an enhanced distributed Cloud storage system. Nevertheless, the adopted protocol is rather complex and hard to be adapted in real scenarios. A similar technique is discussed in [15], where the authors present PRESIDIO, a framework able to detect similarity and reduce or eliminate redundancy when storing objects. The work in [7] discusses a way for optimizing the file partition in network storage environment. The model assessed by the authors shows how a partitioned network is able to maintain high availability. However, the approach is theoretical. In [9], a secure Cloud backup system is investigated. The authors study how to manage the Data Deletion (Assured) and the Version Control.

In [7], the authors describe a technique for optimizing the file partition considering a Network Storage Environment. They present a strategy to efficiently distribute files inside a cluster taking into account concepts of reliability, availability and serviceability. A file partitioning approach for Cloud computing is described in [4], where a smart procedure is used to optimize the placement of each data block according to its size. In [16], the authors face the problems that arise whenever a laptop is lost or stolen. The system guarantees that data cannot be accessed after an a priori configured time window and this can be a point of failure. The authors use XOR operations to split and merge files that have to be protected. The procedure is hard to be applied, because it requires to customize the kernel of the involved servers.

Data distribution [17] along with Data Migration [8] are topics quite relevant in the context of Cloud storage. The need to send big data over the Internet is important as well as the possibility to overcome data lock-in issues. Cloud operators are trying to prevent them for maintaining their business.

3. Motivations: the Current Trend on Cloud Storage. The arising requirement of Internet and, in particular, of Cloud Computing is the management of "Big Data". Big Data refers to a huge amount of data that users produce due the massive interconnection of smart devices and sensors over the Internet, which represent the basis for the development of Internet of Things (IoT) applications. Cloud Storage services represent the basic infrastructure for storing this produced data.

Cloud services are organized into three main levels: Infrastructure, Platform, or Software as a Service (i.e., IaaS, PaaS, or SaaS). Cloud providers can rely on these three levels in order to provide several storage functionalities. Amazon is the largest Cloud storage player and provides Storage as IaaS. Looking at the Amazon S3 storage service, it provides a simple web service able to store/retrieve any kind of data into/from the web. S3 provides the access to the scalable, reliable, secure, and fast Amazon storage infrastructure. S3 is widely used by many SaaS providers (e.g., Dropbox, Megauploader, Rapidshare, etc). In this paragraph, we discuss the trend of the number of objects stored in the S3 infrastructure during the last seven years.

The number of objects stored into the Amazon S3 has grown from roughly 700 Billion objects in one year to 2000 Billion objects at the time of when this paper is written. Figure 3.1 shows the market on data production and storage is vertiginously growing up in the last decade, according to the following equation:

$$NumObjs = 1,1335 * X^{3,5905} \tag{3.1}$$

Equation 3.1 allows to make the prevision on the storage demand of about 12000 Billion objects in the next five years (2018). Such a prevision is justified by the increasing interest of Information and Communication Technology (ICT) societies and end-users towards Cloud storage, with the purpose to reduce costs and satisfy their needs with a large plethora of opportunities. According to these considerations, we propose a new way to support emerging requirements for future Cloud storage.

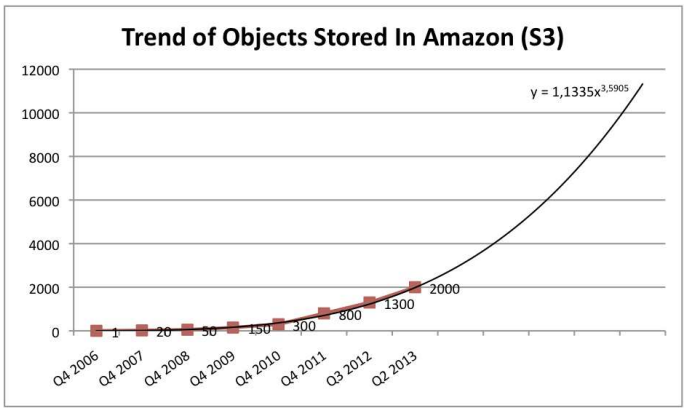
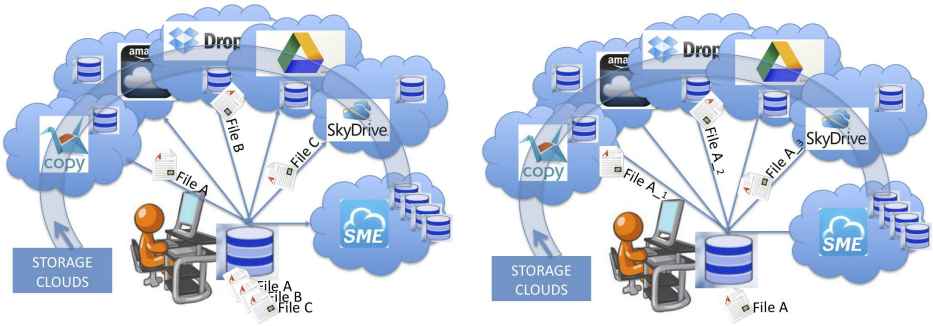


FIG. 3.1. *Amazon S3 growth in the last seven years and its future prevision for the next five years.*



(a) Classic management of data over multiple Cloud storage providers

(b) The new approach introduced in this work

FIG. 3.2. *Storage Cloud Services distributed over the Internet*

3.1. A New Approach to Store Data Into the Cloud. The usage of Cloud storage providers is characterized by the possibility for customers to subscribe many storage services even for free (e.g., Copy, DropBox, Drive,...) and to manually manage data upload and download. For example, a user holds files A, B and C in his local file system and decides to upload these files into the Cloud, as shown in Fig. 3.2(a). He/she has to choose where each file has to be uploaded (for example, file A in Copy, file B in Dropbox, and file C in SkyDrive), by using personal policies (such as type of file, size of file, and so on) or making a random choice. Thus, when he/she will require the file, he/she has to remember where the file has been uploaded. In addition, after upload, Cloud providers have the full control of the files they store, and this can be a potential threat for data confidentiality. We believe that a way for reducing such threats consists in using the Cloud storage in a different way than usual. In this paper, we describe an approach that consists in spreading chunks of each single file over different Cloud storage providers. Our approach introduces a software layer that abstracts heterogeneous Cloud storage providers and allows end-users to upload their files in an efficient way. Figure 3.2(b) shows how the proposed approach works. The original file A is split in three chunks, A_1, A_2 and A_3. The end-user makes a choice about the level of redundancy of each file, in order to overcome fails in data retrieval or data loss. The algorithm we implemented in our solution allows to manage a particular file redundancy in a smart way, as we will describe in Sect. 4.1. Of course, file splitting and merging processes are hidden to each Cloud operator, so that they cannot have any knowledge about data content. This prevents the possible fraudulent access to customer data by Cloud storage providers, that is a very hot topic on Cloud computing.

Moreover, pieces of file or chunks are wrapped into XML structure, in order to increase the portability of the system. Any Cloud storage provider sees the XML file with a body containing a chunk of the original file coded in BASE-64. In the example in Fig. 3.2(b), some chunks can be stored in Copy, others in SkyDrive, till Dropobox. Only end-users should be totally aware of what data are stored in each Cloud storage operator. Chunks distribution over the Cloud is described in a metadata map-file, an XML file that tracks where chunks are stored and allows to reconstruct the original file. The failure of the metadata map-file determines the loss of the whole file. To prevent this event and improve the reliability of the proposed solution, the map-file has to be stored in the Cloud, but information on chunk distribution must be spread over two or more further partial metadata map-files and deployed over two or more different independent trusted Cloud providers in order to carry out also medadata obfuscation. Since the trusted providers hold only partial metadata map-file, no one must be able, by itself, to reconstruct the whole metadata map-file of any particular user. In the following, we describe the mathematical concepts behind the redundancy algorithm used in our approach.

4. The Redundant Residue Number System. If you consider p prime, pairwise and positive integers $m_1, m_2, \cdots, m_p$ called *modulus* such as $M = \prod_{i=1}^{p} m_i$ and $m_i > m_{i-1}$ for each $i \in [2, p]$. Given $W \geq 0$, we can define $w_i = W \mod m_i$ the residue of W modulo m_i. The p-tuple $(w_1, w_2, \cdots, w_p)$ is named the *Residue Representation* of W with the given modulus and each tuple element w_i is known as the i^{th} residue digit of the representation. For every p-tuple $(w_1, w_2, \cdots, w_p)$, the corresponding W can be reconstructed by means of the Chinese Remainder Theorem:

$$W = \left(\sum_{i=1}^{p} w_i \frac{M}{m_i} b_i \right) \mod M \tag{4.1}$$

where b_i, $i \in [1, p]$ is such that $\left(b_i \frac{M}{m_i} \right) \mod m_i = 1$ (i.e. the multiplicative inverse of $\frac{M}{m_i}$ modulo m_i). We call *Residue Number System (RNS)*, with residue modulus $m_1, m_2, \cdots, m_p$, the number system representing integers in $[0, M)$ through the p-tuple $(w_1, w_2, \cdots, w_p)$. Considering $p + r$ modulus $m_1, \cdots, m_p, m_{p+1}, \cdots, m_{p+r}$ we have:

$$M = \prod_{i=1}^{p} m_i \tag{4.2}$$

and

$$M_R = \prod_{i=p+1}^{r} m_i \tag{4.3}$$

without loss of generality $m_i > m_{i-1}$ for each $i \in [2, p + r]$. We define *Redundant Residue Number System (RRNS)* of modulus $m_1, \cdots, m_{p+r}$, range M and redundancy M_R, the number system representing integers in $[0, M)$ by means of the $(p + r)$-tuple of their residue modulus $m_1, \cdots, m_{p+r}$. Although the above mentioned *RRNS* can provide representations to all integers in the range $[0, M \cdot M_R)$, the legitimate range of representation is limited to $[0, M)$, and the corresponding $(p+r)$-tuples are called *legitimate*. Integers in $[M, M \cdot M_R)$ together with the corresponding $(p+r)$-tuples are instead called *illegitimate*. Let now consider an *RRNS* whose range is M and redundancy M_R, where $(m_1, m_2, \cdots, m_p, m_{p+1}, \cdots, m_{p+r})$ is the $(p+r)$-tuple of modulus and $(w_1, w_2, \cdots, w_p, w_{p+1}, \cdots, w_{p+r})$ is the legitimate representation on an W integer in $[0, M)$. If an event maks unavailable d arbitrary digits in the representation, we have two new sets of elements $\{w'_1, w'_2, \cdots, w'_{p+r-d}\} \subseteq \{w_1, \cdots, w_{p+r}\}$ with the corresponding modulus $\{m'_1, m'_2, \cdots, m'_{p+r-d}\} \subseteq \{m_1, \cdots, m_{p+r}\}$. This status is also known as *erasures* of multiplicity d. If the condition $d \leq r$ in true, the *RNS* of modulus $\{m'_1, m'_2, \cdots, m'_{p+r-d}\}$ has range:

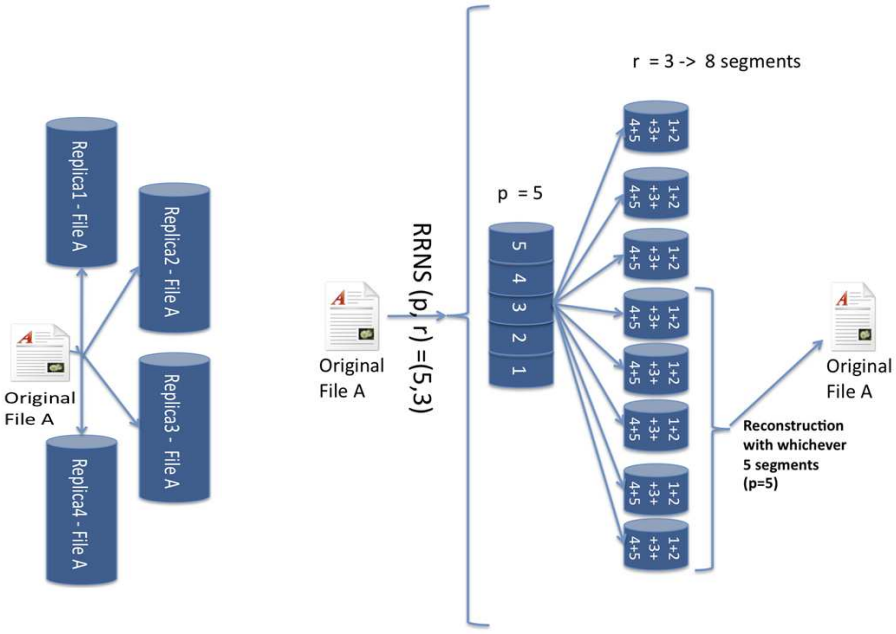
$$M' = \prod_{i=1}^{p+r-d} m'_i \leq M \tag{4.4}$$

since $W < M$, $(w_1, w_2, \cdots, w_p, w_{p+1}, \cdots, w_{p+r})$ is the unique representation of W in the latter *RNS*. Integer W can be reconstructed from the $p + r - d$-tuple $(w'_1, w'_2, \cdots, w'_p, w'_{p+1}, \cdots, w'_{p+r-d})$ by means of the Chinese Remainder Theorem (as in the case of equation 4.1):

$$W = \left(\sum_{i=1}^{p+r-d} w'_i \frac{M'}{m'_i} b'_i \right) \mod M' \tag{4.5}$$

where b_i is such that $\left(b'_i \frac{M'}{m'_i} \right) \mod m'_i = 1$ and $i \in [1, p + r - d]$. As a consequence, the above mentioned *RRNS* can tolerate erasures up to multiplicity r. It can be proved (see [12] for further details) that the same *RRNS* is able to detect any error up the multiplicity r and it allows to correct any error up the multiplicity $\lfloor \frac{r}{2} \rfloor$.

In the following, we are going to explain the overhead introduced by the RRNS encoding algorithm.



(a) Classic management of data redundancy.

(b) File management based on RRNS

FIG. 4.1. *Traditional approach versus the RRNS-based approach.*

4.1. RRNS Exploitation for Cloud storage. In this paper we present a new solution for storing files into the Cloud based on RRNS. It works by using encoding techniques and redundancy policies to offer a very reliable and more secure service using obfuscation. Of course, from the point of view of the storage overhead, the RRNS cause an increase in terms of file size. Here, we analyze the impact of the proposed solution in terms of space disk overhead, comparing our results with more traditional approaches.

The traditional approach used to increase fault tolerance in data storage is to replicate data, as shown in Fig. 4.1(a). Thus, if we need a 3 degree of redundancy for file A (that means we can recover A even if 3 files are lost), we need to deploy 4 replicas of A in different Cloud storage providers. The redundancy mechanism implemented in RRNS allows to reduce the storage consumption. Let consider the following two parameters: p

is the minimum number of modules necessary to reconstruct a file and r is the redundancy degree. Let consider a generic file A. We split A in p residue-segments. Moreover, to have 3 degree of redundancy, we set $r = 3$ and we can recover A even if 3 residue-segment are lost (Figure 4.1(b) shows the behavior of the system for p=5). According to formulations expressed in Sect. 4, a system able to tolerate chunks unavailability up to $d = 3$ can be codified with different configurations:

- p=1, r=3 → that is 4 residue-segments;
- p=3, r=3 → that is 6 residue-segments;
- p=5, r=3 → that is 8 residue-segments;
- p=7, r=3 → that is 10 residue-segments;
- p=9, r=3 → that is 12 residue-segments.

The difference among the above configurations depends on the number of Cloud storage operators available to store replicas/segments. By using RRNS, we can modulate the number of devices (i.e., Cloud storage providers) that have to be involved for storing data, that is 4 (as the traditional case), 6, 8, 10 and 12. This allows us to increase the overall reliability of the system if we consider many Cloud operators. In Fig. 4.1(b), the original file is split into 8 residue-segments. Here, the erasure r is equal to 3. The system is able to reconstruct the original file up to $d = 3$ residue-segments unavailability.

At the end of the encoding/uploading process, a single Cloud storage provider holding the whole file will not exist and this will lead to some direct consequences: even though there's not encryption on data, a self-contained file will not exist on any storage provider, leading to an increased confidentiality degree. This type of data access restriction is also know as *data obfuscation*. Thanks to the redundancy introduced by the RRNS, in case of temporary unavailability of one or more residue-segments (according to Eq. 4.5), the user's file might still be reconstructed from the owner. Indeed, only the owner knows the logical distribution of segments stored over the different Cloud storage providers, hence their potential reconstruction. Obviously, the introduced redundancy increases the resulting amount of data to be stored and transferred, but how previously discussed such an overhead is acceptable compared with the traditional approach consisting on managing whole replicas of files. In addition, if a particular Cloud storage provider is heavily overwhelmed from users' requests, having data spread over different Cloud operators might quicken the download task. In particular, instead of waiting for the transmission of a monolithic block from the overloaded Cloud provider, the client can download different residue-segments in parallel from different operators, allowing a more efficient bandwidth occupation. This same method is used by the *Torrent* protocol for increasing the speed of file downloading over the Internet. In the following, we formalize the overhead of our approach with respect to the traditional one.

4.2. Overhead Evaluation for Reliable Capabilities. Let us consider a file management based on RRNS(p,r) and x as the original file size of file A. The base64 encoding used to make the XML wrapping implies that a set of 6 bits is converted in an ASCII character (8 bit), with a total overhead of $\frac{8-6}{6} = \frac{1}{3}$ [5]. Considering that 4 is the compression rate due to the RRNS algorithm, the final size S_{file} of the files we have to upload over the Cloud can be calculated as following:

$$S_{file} = (\frac{p+r}{4}) * x + (\frac{p+r}{4}) * \frac{1}{3} * x \tag{4.6}$$

where the first term is related to the increased size of the file due to the RRNS algorithm and the second term is the increased size of the overhead for the base64 encoding due to the RRNS algorithm. From equation 4.6 we obtain:

$$S_{file} = (\frac{p+r}{4}) * 1,33 * x \tag{4.7}$$

Thus, the storage size of the file depends on both p and r. In Fig. 4.2, we have drawn the multiplicative factor of the file size for a file of 1MB, when $p = 5$. For example, when $r = 7$, the storage size required is about 4MB. A traditional redundancy approach, where multiple copies of the file are stored, implies a storage size of 7MB. Thus, on equal error tolerance, our approach reduces the storage size of about a factor $1,75$.

5. RRNS-Based Approch for a Reliable Cloud Storage. We underline that the two key-points on which our approach is based consist in guaranteeing data availability (resiliency) and increasing data confidentiality through the *obfuscation technique*. From the client point of view, different types of application front-end

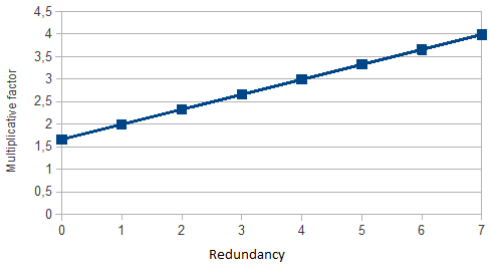


FIG. 4.2. *Multiplicative factor of file size with the redundancy*

can be developed: web application in form of SaaS, java applets, stand-alone desktop application, mobile application, and so on. Despite of any particular type of front-end application, here, we describe the main steps required to split a file and upload residue-segments over different Cloud operators and the main steps to recover residue-segments, downloading them, and reconstruct the original file. An application front-end should be able to receive as input one or more files belonging to a given user and uploading them over the several Cloud storage providers according to specific constraints. Thanks to the RRNS properties discussed in Sect. 4, each time the encoding process is applied to a file, it is split (as depicted in Fig. 5.1) on different residue-segments according to a given degree of redundancy. In order to guarantee a high level of flexibility, we use a XML wrapper for representing residue-segments adopting the BASE-64 encoding. BASE-64 encoding schemes are commonly used when there is the need to encode binary data that needs to be stored and transferred over media that is designed to deal with textual data. Although the XML container allows a strong level of environment independence, the BASE-64 encoding employed for encapsulating binary data within the *content node* involves a storage size overhead. In fact, the user data after the BASE-64 encoding will involve a storage requirement increment approximately of 33% as described in Sect. 4.2. Then, the XML residue-segments will be copied and stored on different Cloud storage providers by using the APIs they make available for developers.

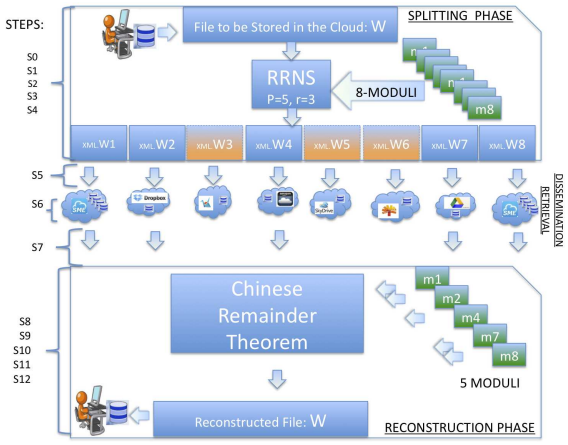


FIG. 5.1. *Representation of the RRNS encoding/decoding.*

After having introduced the general concepts regarding our idea, in the following we are going to discuss the a few implementation guidelines, analyzing how data is processed both at upload and download phases. Figure 5.1 depicts how these tasks are carried out: the top part represents the RRNS encoding/uploading process, while the underside one represents the downloading/decoding process. The main steps that have to be executed by a generic front-end application are schematized in Table 5.1. Whenever an end-user wants to store a file W, he/she specifies his/her requirements through an *init* file, in which he/she sets the parameters p and r and provides information on Cloud storage providers involved in data uploading (STEP S0). The

SPLITTING PHASE
S0: end-user selects the File W, the number of required fragments (p) and the redundancy degree (r).
S1: $W_{zip} = ZIP(W)$. The the File W is compressed by means of the zip algorithm
S2: $Wx = RRNS(W_{zip})$. This process generates $p + r$ residue-segments.
S3: Wx BASE-64 encoding.
S4: Wx XML encapsulation.
DISSEMINATION PHASE
S5: Upload of XML residue-segments. This step involves several Cloud storage operators. According to the particular type of front-end application, the upload task can be accomplished after S0.
S6: Storage of XML residue-segments into several Cloud storage providers. Each provider is not able to know the content of the whole file.
RETRIEVAL PHASE
S7: Download of the XML chunks.
RECONSTRUCTION PHASE
S8: Wx XML decapsulation.
S9: Wx BASE-64 decoding.
S10: $W_{zip} = RRNS(Wx)$. The Zipped file is reconstructed using the Chinese Reminder Theorem.
S11: $W = UNZIP(W_{zip})$. This compressed file is uncompressed with ZIP utility.
S12: end-user access the original file. According to the particular type of front-end application, S8, S9, and S10 steps can be accomplished after S8.

TABLE 5.1
Main steps that have to be accomplished by a front-end application.

application compress the W file with the ZIP utility in order to save space (STEP S1). The RRNS encoding is applied to the zipped file and generate a set of $p + r$ residue-segments (STEP S2). Each residue-segment is BASE-64 encoded (STEP S3) and attached within an XML wrapper (STEP S4). Each XML wrapper, in turn, will then be uploaded to a particular Cloud storage providers specified by the user (STEP S5). Retrieval and reconstruction of the file W are performed thorough vice versa activities (STEPS S6-S12).

5.1. XML Wrapper Details. In order to track the location of uploaded residue-segments, for each file, a metadata map-file is created. Although, how previously stated, how to obfuscate the metadata map-file spreading it over different Cloud providers is out of scope, here we provide a few details about the structure of such a file also presenting a simple example of metadata map-file obfuscation using the MD5 and two different trusted Cloud providers. The metadata map-file must be accessible only from the data owner and it allows to rebuild the original file. In the following is presented an example of possible metadata map-file:

```
<OWNER>ownerInfo</OWNER>
<SEGMENTS>...</SEGMENTS>
<FILE>
    [...]
    <CHUNK num="11">Path/to/the/StorageProviderX/
            94090e1381a1700fb8c34a0069bc6533.xml</CHUNK>
    <CHUNK num="5">Path/to/the/StorageProviderY/
            eaf2bcdcb47cd1eba2a4392857e66b33.xml</CHUNK>
    [...]
</FILE>
```

The first element of the file, *OWNER*, specifies owner information. The *SEGMENTS* element includes the number of necessary segments to reconstruct the file (that is the value of p). The *FILE* element contains a variable

number of *CHUNK* elements. The *CHUNK* tag has the attribute *num*, which refers to the residue-segment sequence number, its content represents a combination of the path on associated on the front-end application, the Cloud storage provider for that chunk, and the name of the XML file containing data. Information stored within the above XML document will allow to build up the original file during the decoding process. Depending on the number of available providers and the number of XML chunks, providers are in charge to store one or more chunks.

It is straightforward foreseeing that the metadata map-file represents a key point of the whole process: its accidental lost or unavailability leads to data loss, because retrieving chunks and rebuilding the file becomes impossible. Thus, keeping the map-file in the local file system of the end-user is not a strategic solution. To improve the reliability of the metadata map-file storing, the front-end application has to be designed to also store the map-file into several Cloud providers. To preserve data confidentiality, the map-file has to be slit in different partial medatada map-files and spread over different independent trused Cloud providers. This mechanism can be achieved using well known security techniques, in particular combining asymmetric and/or symmetric encryption with the MD5 message-digest algorithm. In order to clarify ideas, in the following we discuss a methodology to split the metadata map-file into two partial metadata map-files using the MD5. In this example, partial metadata map-files are called *servicelist* and *trusted*. The *servicelist* file is an XML document containing the list of storage providers on which a user holds an account for uploading/downloading files. The *trusted* file in an XML document used to associate a residue-segment number to the name of the related XML file containing the residue-segment data, and an unique identifier associated to the service provider on which that chunk is stored:

```
[...]
<OWNER>ownerInfo</OWNER>
<SEGMENTS>...</SEGMENTS>
<FILE>
[...]
<CHUNK num="11">
    <CHUNK_REF>94090e1381a1700fb8c34a0069bc6533.xml</CHUNK_REF>
    <UUID_REF>a72ebba5d9b695c39e6d2193c3cb8057</UUID_REF>
</CHUNK>
<CHUNK num="5">
    <CHUNK_REF>eaf2bcdcb47cd1eba2a4392857e66b33.xml</CHUNK_REF>
    <UUID_REF>9e296f8ea3992ef53ff93e9adbc80299</UUID_REF>
</CHUNK>
[...]
</FILE>
[...]
```

The first two elements of the file are identical to the ones in the map-file. The *CHUNK* tag within the *FILE* element has the attribute *num*, which refers to the fragment order and the child elements *CHUNK_REF* and *UUID_REF* respectively identify the name of the XML file containing the data associated to that chunk and a unique identifier associated to the storage service provider. The *UUID_REF* does not contain the actual service provider in order to obfuscate this information to the storage provider where the file will be uploaded. In fact, the unique identifier is obtained applying the MD5 to the couple (chunk , provider) in order to prevent brute force attacks aiming at found out the actual paths associated to the chunks. Uploading to different trusted Cloud providers the two partial metadata files instead of the actual whole metadata map-file guarantees only to the end-user the knowledge about the file partitioning. This task is highlighted in Fig. 5.2. Starting from *serviceList* and *trusted* files, we can obtain the metadata map-file necessary for reconstructing the original file. The metadata map-file obfuscation can be obtained also with other techniques considering several trused Cloud providers. Further details about metadata map-file obfuscation are out of scope. In the following, we specifically focus on evaluating how the RRNS algorithm works considering multiple Cloud storage providers.

6. Experimental Results. In order to evaluate our system, we conducted several experiments considering a real testbed composed of client stand-alone java desktop front-end application interacting with three different commercial Cloud storage providers, that are Google Drive, Dropbox and Copy. In our experiments, we used file with different sizes, different redundancy factors and we evaluated the time spent for the splitting activity,

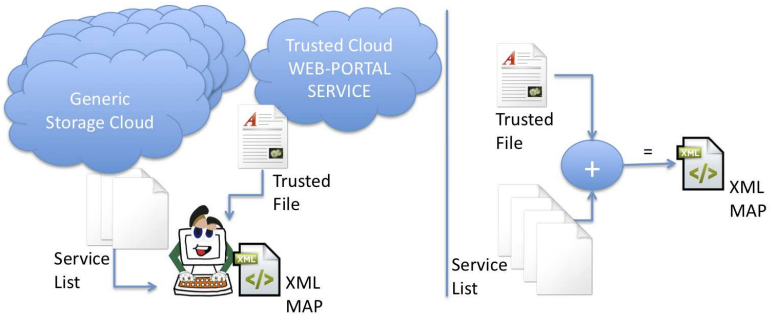
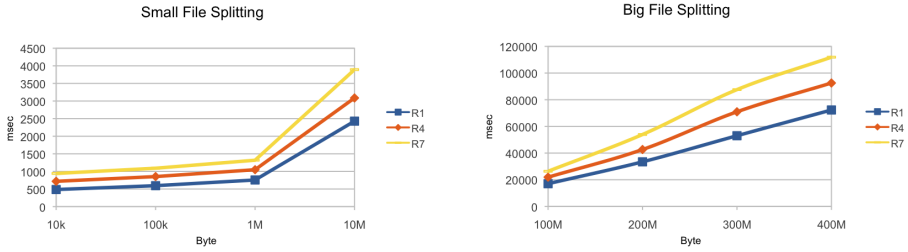


FIG. 5.2. *map file reconstruction.*

the time spent for uploading residue-segments, and the time spent for retrieving and re-composing the original file.

6.1. Testbed Setup. The local testbed was arranged at DICIEAMA GRID Laboratory at the University of Messina. The front-end application was deployed in a blade with the following hardware configuration: CPU Dual-Core AMD Opteron(tm) Processor 2218 HE, RAM 6GB, OS: ubuntu server 12.04.2 LTS 64 BIT. We considered two sets of file sizes defined *Small Files*, characterized by 10KB, 100KB, 1MB, 10MB, and *Big Files*, characterized by 100MB, 200MB, 300MB, and 400MB. We fix p=5 and the following values for r: [1, 4, 7]. According to Eq. 4.5, each file is split respectively into 6, 9, and 12 residue-segments, from now on called chunks. We store chunks balancing the workload over the three Cloud storage providers, so that in each one, we stored respectively 2, 3, and 4 pieces of file. Each experiment was repeated 30 times in order to consider mean values and confidence intervals at 95%. In the following, we will describe the performance of Splitting and Reconstruction phases along with Dissemination and Retrieval phases.



(a) Time needed for the Splitting phase with Small Files

(b) Time needed for the Splitting phase with Big Files

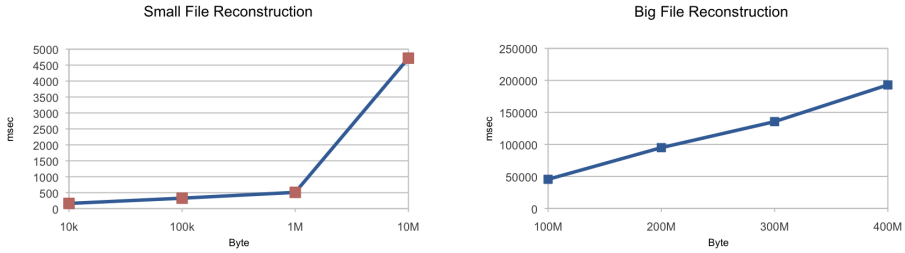
FIG. 6.1. *Time needed for the Splitting phase with Small and Big Files.*

6.2. Performance Analysis for Splitting and Reconstruction Phases. Here, we investigate the behavior of Splitting and Reconstruction phases described in Table 5.1. They are quite similar, but the Reconstruction phase takes into account a reduced number of chunks, i.e., $p = 5$, that are downloaded and reconstructed.

Figure 6.1 summarizes the time required to split each file. The x-axis reports the file size in bytes, whereas the y-axis reports the time in milli-seconds (msec). The graphs show the trend for both Small and Big files. For file sizes up to 1MB, the processing time is almost constant considering the different file sizes. Increasing the file size from 10 MB to 400MB, the time increasing is quite double at each step. Different values for r do not

significantly affect the performance, hence the user can improve the reliability of the storage service without a high degradation of performance.

Similar results, depicted Fig. 6.2, show the time spent for rebuilding Small and Big files. The time required to reconstruct a big file size, i.e, 400MB is about 200 seconds. The reconstruction phase is much more heavy respect to the split phase (110 seconds for a 400MB file with $r = 7$). Different values of r are not considered because the minimum number that is used for file reconstruction is $p = 5$.



(a) Time needed for the Reconstruction phase with Small Files

(b) Time needed for the Reconstruction phase with Big Files

FIG. 6.2. *Time needed for the Reconstruction phase with Small and Big Files.*

6.3. Performance Analysis for Dissemination and Retrieval Phases. Here, we investigate the behavior of Dissemination and Retrieval phases described in Table 5.1. Figure 6.3 shows the time spent to send

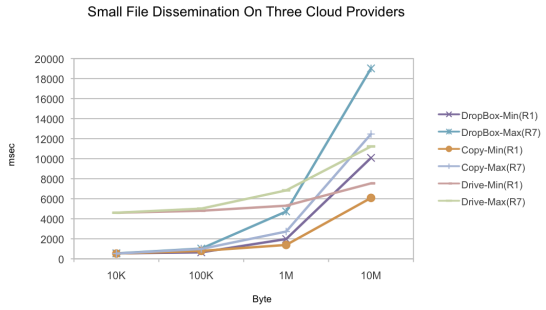


FIG. 6.3. *Time spent for the dissemination process to three Cloud providers.*

chunks in parallel to Google Drive, Copy, and Dropbox. We analyzed the transfer time of files up to sizes of 10MB with $r = 1$ and $r = 7$ respectively, i.e., the minimum and maxim values for r. The y-axis reports the time in milliseconds (msec), whereas the x-axis reports the file size in bytes. In these proofs, we prefer to limit the file size to 10 MB because in our testbed the upload bandwidth was much more low respect to the download one. This allowed us to repeat more times experiments in a reasonable time interval. For files of 10KB and 100KB, we experienced very similar transfer times and Google Drive results the slowest. For files of 10KB with r=1 the transfer times in Copy and Dropbox take respectively in average 523 msec and 543 msec. With Google Drive, the transfer time instead takes in average 4603 msec. We observed a similar trend considering r=4 and r=7 as well.

As the file size grows up over 1MB, results changed. Google Drive is the slowest provider again, but the transfer times increases considering different redundancy factors. Analyzing the results, we distinguished different behaviors between Copy and Dropbox: the former resulted the most efficient, instead the latter started

to degrade in performance. For files of 10MB, we observe an interesting behavior: Google Drive becomes more efficient than Copy and Dropbox, that experienced performance degradation. In fact, with r=7 the transfer times took 11223 msec, 12461 msec, and 19015 msec respectively with Google Drive, Copy and Dropbox. This means that for small file sizes ($< 100KB$) Copy and Dropbox are more efficient than Google Drive, but for big file ($> 10MB$) Drive is more efficient than Copy and Dropbox. In particular, Copy has a trend slightly worse than Google Drive, instead Dropbox results absolutely the worst. Figure 6.4 analyzes the download time of

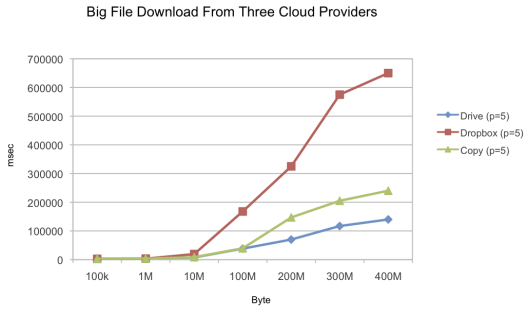


FIG. 6.4. *Time spent to download chunks from three different Cloud providers*

files with size from 100K up to 400 MB . For simplicity, we considered 5 segments ($p = 5$) stored on the same operator since performance evaluation is more complex mixing different operators. The picture highlights the better behavior of Google Drive respect to the others. In the worst case, the time for downloading 5 chunks for a file of 400 MB is more of 10 minutes (65000 msec). With the other providers, it is necessary to spend less than 5 minutes (respectively 140000 msec and 240000 msec). These results are reasonable enough for guarantying a good user experience in using the the our system.

This analysis allowed us to know the behavior of three of the major Cloud storage providers in order to understand a few useful information about how setup a system using different Cloud storage providers.

7. Conclusion and Future Works. In this paper, we discussed the data reliability and confidentiality problems considering a multi-provider Cloud storage service. By means of RRNS, our approach allows to split a file in p+r chunks, which are deployed over different Cloud storage operators. The advantage of our approach is twofold: on one hand, each single provider cannot access the whole file, and on the other hand if a provider is not available, files can be retrieved considering p pieces of files stored in other operators. Experiments highlighted how both file size and redundancy degree impact the performance of the proposed system considering Google Drive, Dropbox, and Copy. In future works, we aim to better investigate such an approach also considering data encryption.

REFERENCES

[1] Deepavali Bhagwat, Kristal Pollack, Darrell D. E. Long, Thomas Schwarz, Ethan L. Miller, and Jehan-Francois Paris. Providing high reliability in a minimum redundancy archival storage system. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, MASCOTS '06, pages 413–421, Washington, DC, USA, 2006. IEEE Computer Society.

[2] A. Celesti, M. Fazio, and M. Villari. Se clever: A secure message oriented middleware for cloud federation. In *IEEE Symposium on Computers and Communications (ISCC)*, 2013.

[3] A. Celesti, F. Tusa, M. Villari, and A. Puliafito. How the dataweb can support cloud federation: Service representation and secure data exchange. In *Second Symposium on Network Cloud Computing and Applications (NCCA)*, pages 73–79, 2012.

[4] Kai Fan, Libin Zhao, Xuemin Shen, Hui Li, and Yintang Yang. Smart-blocking file storage method in cloud computing. In *2012 1st IEEE International Conference on Communications in China (ICCC)*, pages 57–62, 2012.

[5] N. Freed and N. Borenstein. MIME: Multipurpose Internet Mail Extensions. Technical Report RFC2045, 1996.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system , 2013. http://www.cs.umd.edu/class/spring2011/cmsc818k/Lectures/gfs-hdfs.pdf.

[7] Wu Hai-Jia, Liu Peng, and Chen Wei-wei. The optimization theory of file partition in network storage environment. In *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, pages 30–33, 2010.

[8] P. Nahar, A. Joshi, and A. Saupp. Data migration using active cloud engine. In *Cloud Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on*, pages 1–4, 2012.

[9] A. Rahumed, H.C.H. Chen, Yang Tang, P.P.C. Lee, and J.C.-S. Lui. A secure cloud backup system with assured deletion and version control. In *Parallel Processing Workshops (ICPPW), 2011 40th International Conference on*, pages 160–167, 2011.

[10] K.W. Shum and Yuchong Hu. Functional-repair-by-transfer regenerating codes. In *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*, pages 1192–1196, 2012.

[11] S. Srivastava, V. Gupta, R. Yadav, and K. Kant. Enhanced distributed storage on the cloud. In *Computer and Communication Technology (ICCCT), 2012 Third International Conference on*, pages 321–325, 2012.

[12] N. S. Szabo and R. I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology.* Mc Graw-Hill, New York, 1967.

[13] G Vernik, A. Shulman-Peleg, S. Dippl, C. Formisano, M.C. Jaeger, E.K. Kolodner, and M. Villari. Data on-boarding in federated storage clouds. In *IEEE 6th International Conference on Cloud Computing*, 2013.

[14] Massimo Villari, Antonio Celesti, Francesco Tusa, and Antonio Puliafito. Data reliability in multi-provider cloud storage service with rrns. In Carlos Canal and Massimo Villari, editors, *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 83–93. Springer Berlin Heidelberg, 2013.

[15] Lawrence L. You, Kristal T. Pollack, Darrell D. E. Long, and K. Gopinath. Presidio: A framework for efficient archival data storage. *Trans. Storage*, 7(2):6:1–6:60, July 2011.

[16] Nan Zhang, Jiwu Jing, and Peng Liu. Cloud shredder: Removing the laptop on-road data disclosure threat in the cloud computing era. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 1592–1599, 2011.

[17] Yu Zhang, Weidong Liu, and Jiaxing Song. A novel solution of distributed file storage for cloud service. In *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, pages 26–31, 2012.

Edited by: Maria Fazio and Nik Bessis
Received: Nov 2, 2013
Accepted: Jan 10, 2014