InterCloud RAIDer: A Do-It-Yourself Multi-cloud Private Data Backup System

Chih Wei Ling and Anwitaman Datta*

Nanyang Technological University, Singapore
http://sands.sce.ntu.edu.sg/

Abstract. In this paper, we introduce InterCloud RAIDer, which realizes a multi-cloud private data backup system by composing (i) a data deduplication technique to reduce the overall storage overhead, (ii) erasure coding to achieve redundancy at low overhead, which is dispersed across multiple cloud services to realize fault-tolerance against individual service providers, specifically we use non-systematic instances of erasure codes to provide a basic level of privacy from individual cloud stores, and finally, (iii) a proof of data possession mechanism to detect misbehaving services - where we optimize the implementation by exploiting hash digests that are created in the prior deduplication phase. Apart from the uniqueness and non-triviality of putting these modules together, the system design also had to deal with artefacts and heterogeneity across different cloud storage services we used, namely Dropbox, Google drive and SkyDrive.

Keywords: backup, deduplication, erasure codes, proof-of-possession.

1 Introduction

Cloud based services have become an integral part for data storage and backup solutions used by many organizations as well as individuals. While the main commercial players generally provide robust and reliable service, guided both by the need to maintain good reputation as well as legal obligations, in this paper we explore an approach which enables the end-users to achieve better reliability and confidentiality while outsourcing their storage without being left to the mercy of the goodwill of any single service provider.

This is important, because, individual storage service providers may be compromised by hardware or software faults, e.g., Dropbox allowed password free access to data [20], Carbonite [21] and Microsoft lost customer data [22], or because a specific service provider may fall prey to hacking attack, e.g., Sony PlayStation Network [23] and LinkedIn [24], or a service may be compromised by malicious employees. Not to mention the issue of overzealous government agencies using or abusing legal instruments to access private data.

Likewise, when an user deletes his data from a service provider, there is no guarantee that all the copies that the service provider may have created (for

* This work was supported by A*Star TSRP grant 102 158 0038 for the pCloud project.

M. Chatterjee et al. (Eds.): ICDCN 2014, LNCS 8314, pp. 453–468, 2014.

fault-tolerance) are actually removed. Consequently, the exposure of data may persist beyond the period an user uses a specific service.

Storing only encrypted data can arguably deal with the confidentiality issue. However it does not prevent the problem of data loss. Creating a service which spreads the data across multiple clouds [25] is a logical way to achieve this.

Several prominent existing works embrace the latter idea, e.g., RACS (redundant array of cloud storage) [1] and DepSKY [2]. In order to keep the overhead of redundancy low, using erasure coding is a natural choice that is deployed by all such systems. Our work leverages the same broad ideas, but unlike these existing works, which demonstrate the concepts using services like Amazon S3, RackSpace, etc. that are aimed to cater to the businesses, we build a working system (InterCloud RAIDer[1]) using more 'plebeian' services such as Dropbox, Google Drive, SkyDrive, etc. which also cater to individual users, particularly by providing a stripped down service for free, which can then be augmented with more resources if needed, by paying. In doing so, we hope that InterCloud RAIDer can be readily utilized by individuals for personal use. A crucial challenge in realizing the system is to deal with the specific constraints posed by the limited and heterogenous APIs and functionalities these services provide.

In order to realize a wholesome system, we incorporate mechanisms for proof of data possession to determine that data stored with specific service providers are retained, and apply deduplication mechanisms in order to curtail the storage overhead. The constraints from the cloud service providers lead to peculiarities in achieving these functionalities, and our implementation experience reveals interesting insights pertaining to these. So to say, while the modular architecture is reasonably straightforward, the essence of this work is to instantiate these specific modules by taking into account the interplay and interference of the different design choices (e.g., to the best of our knowledge, deduplication is unique to our system w.r.to the above mentioned multi-cloud storage systems, and we will subsequently explore its synergy with proof of data possession mechanisms while discussing our design choices in building InterCloud RAIDer), and in presence of extrinsic artefacts as posed by the heterogeneous cloud service providers.

We use a non-systematic erasure code, which provides some natural obfuscation of the data being stored in the individual storage services. This approach is readily compatible with the deduplication mechanism, but does not provide the same kind of security as cryptography does. To keep the implementation simple, we did not add an encryption layer, but we will discuss how it could also be added if necessary while still retaining the other functionalities. Though the implementation is geared towards use by individuals for file backup, and uses commercial cloud services like Dropbox, the implementation can be easily generalized in two manner: (i) A reliable multi-cloud file system transparently providing a predefined set of interfaces, similar to Amazon S3 to be used by ap-

[1] The source code is available at `http://code.google.com/p/intercloudraider/`. The 'InterCloud' in the name refers to the use of multiple cloud services in our system, which are used to form a Redundant Array of Independent (virtual, cloud service based) Disks, i.e., RAID.

plications in a manner agnostic of the cloud-level distribution will allow diverse usage of the system. (ii) Mixing peer-to-peer or friend-to-friend 'crowdsourced' storage cloud [3] along with dedicated commercial cloud service providers, is expected to allow further robustness against various adversaries, while achieving agreeable quality of service. Such extensions comprise our future plans.

2 Preliminaries

InterCloud RAIDer is built on three techniques to achieve reliable and storage efficient data backup. The first is chunk-based deduplication [4–6, 10, 11], a method that breaks a file or data stream into a sequence of chunks and eliminates duplicate chunks by comparing the chunk hashes. The second is non-systematic erasure codes, which adds redundancy to stored file so that retrieving fully the file is possible even under the presence of faults. Using a non-systematic code provides some basic confidentiality of the data stored at different storage services. Encrypting the encoded data is an optional step (not implemented at present) that can be readily plugged in, in our system if stricter confidentiality guarantees are needed. The third is Provable Data Possession (PDP) [16–18], which provides the assurance to data owners that the data servers who claim to store the outsourced data from the owners, are actually storing the data. We next briefly summarize the specific instances of the respective techniques that we have employed in InterCloud RAIDer.

2.1 Chunk-Based Deduplication

Chunk-based deduplication has three main steps: chunking, indexing and deduplicating. Many chunking algorithms have been proposed and studied at length in the literature [5–9]. For an incoming data stream, the chunking step divides the stream into fixed or variable length chunks. We use the Two-Threshold Two-Divisor (TTTD) chunking algorithm [6] to subdivide the incoming data stream into a sequence of chunks. TTTD generates variable length chunks with smaller variation than other chunking algorithms and thus produces better deduplication ratio.

Then, a cryptographic hash function, such as MD5 [12] or SHA1 [13, 14], is used to hash the chunks at the indexing step. The chunks are identified by the hashes. By assuming that the collision probability is very low, chunks with the same hash and size are assumed to be identical. Since comparing hashes is faster than comparing the actual chunk contents, the deduplicating step can be executed efficiently. Only new chunks are stored and references are updated for duplicate chunks.

The naive and traditional way to implement deduplicating is to use full chunk index: a key-value index of all the stored chunks, where the key is a chunk's hash and the value holds metadata about that chunk, including where it is stored on disk. When an incoming chunk is to be stored, its hash is looked up in the full index, and the chunk is stored only if no entry is found for its hash. However,

there are more sophisticated deduplicating techniques, such as Data Domain Deduplication File System [10] or Sparse Indexing [11], that solve the chunk-lookup disk bottleneck problem caused by full chunk index. We implement full chunk index given its simplicity.

2.2 Non-systematic (Homomorphic Self-repairing) Erasure Codes

An erasure code is a mapping, which creates $n > k$ symbols out of k input symbols, such that any k' (where $k' \geq k$ and $k' < n$) symbols can be used to reconstruct the inputs. A systematic erasure code is an erasure code in which the input data is embedded in encoded data. More precisely, for a systematic linear code, the generator matrix G can be expressed as $G = [I_k|A]$, where I_k is the identity matrix of size k. A non-systematic erasure code does not contain the input data in its encoded output. If systematic coding is used, then a server storing a systematic piece will be able to retrieve this part of the original input. When using non-systematic codes, without access to enough (code parameter k') encoded pieces at a single service provider, it is information theoretically impossible to infer the original input. Note that a strong adversary which has controls multiple such apparently independent services will be able to access the data as well. An additional layer of encryption may be employed to mitigate such information leak. However, in the current implementation we assume that only individual storage services can be individually and independently compromised, and hence bypass this computationally expensive step by using non-systematic erasure codes 'to kill two birds with a single stone'. Specifically, we use homomorphic self-repairing codes (HSRC) [19] to encode the chunks. In the current implementation, we do not exploit the 'repairability' property of HSRC, and hence any other non-systematic erasure code may also be used instead - but we aim to exploit the repairability property in future extensions, which we will discuss later when concluding the paper.

We denote finite fields by $\mathbb{F}$ and fields with q elements by $\mathbb{F}_q$. Let $\mathbf{o}$ be an object of size M, that is, $\mathbf{o} \in \mathbb{F}_{q^M}$. We can write

$$\mathbf{o} = (\mathbf{o}_1, ..., \mathbf{o}_k), \mathbf{o}_i \in \mathbb{F}_{q^{M/k}} \quad . \tag{1}$$

Then a *homomorphic self-repairing code*, denoted by HSRC(n, k), is the code obtained by evaluating the polynomial

$$p(X) = \sum_{i=0}^{k-1} p_i X^{q^i} \in \mathbb{F}_{q^{M/k}}[X] \tag{2}$$

in n non-zero values $\alpha_1, ..., \alpha_n$ of $\mathbb{F}_{q^{M/k}}$ to get an n-dimensional codeword

$$(p(\alpha_1), ..., p(\alpha_n)), \tag{3}$$

where $p_i = \mathbf{o}_{i+1}$, $i = 0, ..., k-1$ and each $p(\alpha_i)$ is the ith encoded piece.

The resulting code does not have a deterministic value of k' - a compromise made n order to achieve good repairibility. This is a slightly tangential issue for

the current work, and we instead use a concrete example of the code in order to explain how it works. Let $f(X) = X^8+X^4+X^3+X^2+1 \in \mathbb{F}_2[X]$ be a primitive polynomial of degree 8 over $\mathbb{F}_2$ and α be a root of $f(X)$. Then a possible set to construct HSRC(7, 3) is the set $\{1, \alpha, \alpha^2, \alpha^{25} = 1+\alpha, \alpha^{26} = \alpha+\alpha^2, \alpha^{50} = 1+\alpha^2, \alpha^{198} = 1+\alpha+\alpha^2\}$ and the corresponding generator matrix is

$$\begin{pmatrix} 1 & \alpha & \alpha^2 & \alpha^{25} & \alpha^{26} & \alpha^{50} & \alpha^{198} \\ 1 & \alpha^2 & \alpha^4 & \alpha^{50} & \alpha^{52} & \alpha^{100} & \alpha^{141} \\ 1 & \alpha^4 & \alpha^8 & \alpha^{100} & \alpha^{104} & \alpha^{200} & \alpha^{27} \end{pmatrix} .$$

Suppose that we have two input strings of three bytes each: 00000100 00000010 00000001 and 00000010 00000101 00000000. By identifying the strings as the elements of $\mathbb{F}_{256}$, we can rewrite them as the vectors $(\alpha^2, \alpha, 1)$ and $(\alpha, \alpha^2+1, 0)$ respectively. By multiplying the input strings with the first and second columns of G_{HSRC}, we have

$$\begin{aligned} (\alpha^2, \alpha, 1) * (1,1,1)^T &= \alpha^2+\alpha+1 = (\alpha, \alpha^2+1, 0) * (1,1,1)^T \text{ , and} \\ (\alpha^2, \alpha, 1) * (\alpha, \alpha^2, \alpha^4)^T &= \alpha^4 = (\alpha, \alpha^2+1, 0) * (\alpha, \alpha^2, \alpha^4)^T \text{ ,} \end{aligned}$$

so we see that for two distinct inputs, some of the resulting encoded blocks may be identical - thus illustrating that individual storage providers cannot discern between the inputs.

2.3 Provable Data Possession (PDP)

While dispersal of redundant data across multiple cloud provides fault tolerance and privacy against individual cloud service providers, it is also essential to ensure that the integrity of the outsourced data in not compromised at each such individual services, e.g., by illegitimate modifications or deletion. Accessing all the data all the time to check its integrity is impractical. This has led to the study of several provable data possession (PDP) techniques in recent years. We build upon a scheme proposed in [16], and modify it to better suit the expected workload for InterCloud RAIDer.

In the original scheme, a data object O is divided into a sequence of fixed-size blocks. Then a pseudo-random permutation g_k indexed on some secret key k is applied on the indices of blocks and a predetermined number of verification tokens are computed based on the contents of the permutated blocks. In our modified scheme, the chunk hashes that uniquely identify the variable-length chunks resulting from deduplication can be naturally used as a replacement of fixed-size blocks in the original scheme. This saves us a large amount of computations during the setup phase, since we only compute the tokens based on chunk hashes which are only tens of bytes (depending on what cryptographic hashing scheme is used), rather than the whole contents of chunks (which may up to tens of KBs). We also remove the authenticated encryption (AE) scheme from the original scheme, because we are storing the verification tokens at local storage of data owner.

The general idea is as follows. Consider a data repository D which contains d variable length chunks $D[1]$, ..., $D[d]$ and a chunk hash log L which contains cryptographic hashes $h_1 = H_1(D[1])$, ..., $h_d = H_1(D[d])$ that are listed sequentially, where H_1 is an arbitrary cryptographic hash function. We chose MD5_HMAC [15] as our pseudo-random function f and AES as the building block of our index permutator g.

During the setup phase, as shown in Figure 1, the token generator generates in advance t possible verification tokens and each token covers r random chunk hashes. To produce the i^{th} token, where $1 \leq i \leq t$, we proceed as follows: First, generate two 16 bytes master keys W and Z by key generator. Store W and Z secretly at local storage of data owner. Then, generate a permutation key $k_i = f_W(i)$ and a challenge nonce $c_i = f_Z(i)$. Next, compute the set of indices $\{I_j \in [1, ..., d] : 1 \leq j \leq r\}$ where $I_j = g_{k_i}(j)$ by index permutator g. Finally, compute the token v_i using the formula:

$$v_i = H(c_i, h_{I_1}, ..., h_{I_r}) \quad , \tag{4}$$

where H is an arbitrary cryptographic hash and v_i is a stored locally secret.

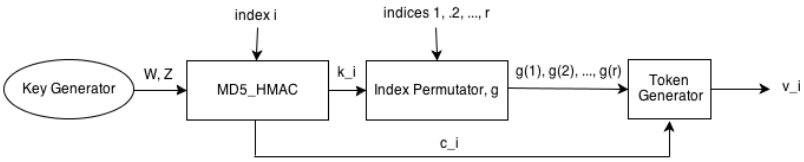


Fig. 1. A PDP Scheme

For the i^{th} proof of data possession verification, we proceed as follows: First, regenerate the token key k_i and c_i by using the master keys W and Z. Then, compute $\{I_j \in [1, ..., d] : 1 \leq j \leq r\}$ as in setup phase and send GET requests to online stores to retrieve the chunks that are identified by h_{I_j}. Next, compute $h'_{I_j} = H_1(D'[I_j])$ where $1 \leq j \leq r$ (the chunks may be modified or deleted). Finally, compute v'_i as:

$$v'_i = H\left(c_i, h'_{I_1}, ..., h'_{I_r}\right) \quad , \tag{5}$$

where H is the arbitrary cryptographic hash function using in setup phase. If $v \neq v'$ then *reject*.

Now, we consider the probability that the hashes of missing or corrupted chunks are not included in the i^{th} verification token. Assume that m is the number of chunks be deleted or modified. Then, the probability is given by

$$P = \left(1 - \frac{m}{d}\right)^r \quad . \tag{6}$$

3 Design Considerations

The techniques explored in this work are generic, but the target users for the implementation are individuals with a small or moderate amount of data to backup, ranging between a few GBs up to a few tens of GBs. At the bottom-end of this volume range, the free capacity provided by services like Dropbox ought to be adequate, but efficient usage of the storage capacity is imperative to maximize utilization of the free services, or keep the additional costs low.

3.1 Cloud Service Providers

In order to realize the backup service over multiple cloud services, the proposed system has to encompass cloud stores from distinct service providers. The cloud Service providers (CSP), however, may use different APIs for access control and resource manipulation. This situation is complicated by the fact that distinct and sometimes mutually incompatible authentication and authorization standards are implemented by different CSPs. For example, as of July 2013, Dropbox is using OAuth1.0 while Google Drive and Skydrive are supporting OAuth2.0 as the recommended authentication mechanism for all of their APIs. However, OAuth2.0 is not backward compatible with OAuth1.0.

In addition, the CSPs may provide distinct file operations to their clients. Dropbox and Google Drive both support retrieving partial file content when a download request is forwarded to them, that is, users can specify to download only a certain contiguous range of the target file at cloud store without downloading the whole file. Put simply, they allow random accesses. However, Skydrive doesn't have this feature - Skydrive only allows file level access. Random access is essential in order to realize an efficient PDP scheme. To address this, in InterCloud RAIDer we upload the encoded pieces of non-duplicated chunks separately rather than upload a data package that contains all the pieces to be stored on cloud stores. This implies that a lot of network requests have to be made for uploading those pieces one by one, which has strong implications on the system's performance (in terms of latency) and its practicality.

Moreover, different CSPs may have different speeds for access control and resource manipulation. For a given block size, the file access interfaces provided by different CSPs with request method GET, PUT or POST may operate under different network speeds. Hence, it is crucial to choose an average chunk size for the system prudently.

3.2 Fixed vs. Variable Length Chunks

In a deduplication system, either fixed-size or content-based chunking can be used to determine the chunk boundaries — thus creating fixed or variable length chunks. The main advantage of fixed-size chunking is its simplicity in implementation. However, inserting or deleting even a single byte at the beginning of the file will shift all the remaining boundaries following the modification point,

resulting in different chunks, knowing as boundary shifting problem. On an average, assuming writes at random locations, half the chunks would be different after every single byte modification to the file [6], and thereby reduce the overall deduplication ratio. Keeping updated the consequent changes in the associated metadata further accentuate the performance deterioration.

In contrast, content-based chunking has the advantage that the chunk sequence is more stable compared to fixed-size chunking under local modification [6]. Intuitively, given a file F, if we make a small modification on F to obtain F', stable under local modification means that after applying the chunking algorithm on both file versions, most of the chunks of F' are duplicates of the chunks of F. To support the dynamic nature of incremental backup system, content-based chunking is preferred over fixed-size chunking.

Average Chunk Size. Chunk size is one of the key design parameters. The choice of average chunk size is difficult because of its impact on the deduplication efficiency, network performance, and security. The smaller the chunks, the more duplicate chunks there will be and so the more efficient the deduplication technique employed in our system. On the other hand, the smaller the chunk size, there are more chunks to process during deduplication which means more times through the deduplication loop, and thus reduce the performance of deduplication. In addition, there will be more metadata of chunks to be maintained in local metadata log, and therefore increase the total cost of data maintenance.

As mentioned in Section 3.1, the APIs of different CSPs may operate under different network speeds. If we choose a smaller chunk size, there will be more number of chunks, and thus more PUT or GET requests have to be initiated to the CSPs for downloading or uploading, which will increase the total system operation time.

Suppose that we are using 4KB as the average chunk size and there is a 1TB of data on cloud store. Then there are approximately 2^{28} unique chunks. If we are using MD5 for indexing in deduplication, then we need 4GB of RAM for full chunk index. Moreover, as per Equation (6), if we have a larger total number of unique chunks, that is, $d = 2^{28}$, then we have to cover more random hashes in the verification tokens to provide better probability assurance provided that we want to give assurance based on amount of lost data rather than ratio between the affected chunks and total number of chunks. In both scenarios, using a smaller average chunk size increases the costs of deduplication and PDP scheme.

4 InterCloud RAIDer

InterCloud RAIDer stores some meta-information locally at the user client, while it outsources encoded data dispersed over multiple cloud service providers in a manner, such at none of the individual service providers can reconstruct the original files by using only data stored individually. The user client can also optionally store part of the encoded data. If the user wishes to access his outsourced data from another device, then he will have to carry the meta-information along.

One can imagine that the meta-information can be carried by the user on an USB storage device. Note that even if the device is lost, exposure of the meta-information is not adequate to gain access to the actual data, since the adversary will need the user's login credentials to gain access to the data stored at individual cloud services. However, in the current implementation, the onus of preserving a copy of the meta-information is still on the user, without which the backed-up data cannot be retrieved.

4.1 Architecture

The primary storage objects in the InterCloud RAIDer are the *files*, the *chunks*, the *encoded pieces* of chunks and their *metadata*. A file is a one-dimensional array of bytes. When a file data stream enters the system, simple metadata associated with the file, such as filename and size are generated and contained within a metadata structure (elaborated in next subsection). The data stream is then broken into a sequence of variable-length chunks by a chunking algorithm for deduplication. These chunks are identified by their contents. A cryptographic hash function is used to compute the hashes of the chunk contents and those hashes are used as the unique identifiers for chunks. Moreover, a file description that keeps track of the chunks mutual relation for constructing the retrieved file is generated. A chunk hash log is also constructed for the generated cryptographic hashes identifying the chunks.

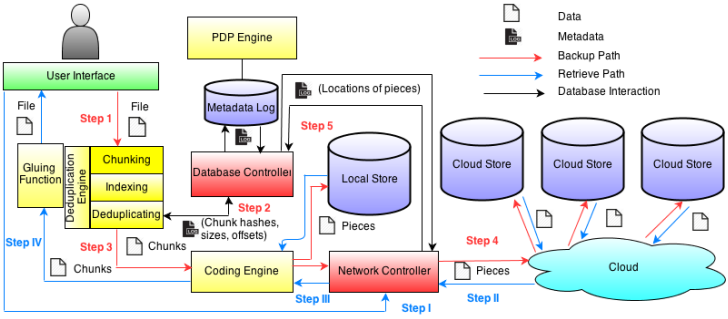


Fig. 2. InterCloud RAIDer Architecture

The software architecture is embodied in processes that execute on each user computer: a backup service interface, a deduplication engine, a coding engine, a database controller, a network controller and a PDP engine. The service interface accepts requests from users through a graphical user interface (GUI). A deduplication engine chunks and deduplicates a file object on backup request so that the identical chunks can be eliminated and thus achieve maximal storage efficiency. The coding engine adds redundancy to the unique chunks by encoding them with a non-systematic code. The network controller controls all the network requests to CSPs being used in our system. The PDP engine generates a

predefined number of tokens for post-verification to assure the data owners that their outsourced data are indeed retained at the CSPs. The database controller keeps track and manages all the meta-information induced by those components.

The service interface consists of operations on storage objects, namely: *backup*, *retrieve*, *delete*, and *verify*; which we elaborate later in this section.

The meta-data is stored locally at the users, as is optionally a part of the encoded data. The rest of the data is dispersed across multiple CSPs. The current implementation uses Dropbox, Google Drive and SkyDrive.

4.2 Metadata Log Layout

The database controller maintains a metadata log in the local storage of system users. The metadata log (illustrated in Figure 3) contains three major types of metadata: the backup log, the file description table and the chunk hash log.

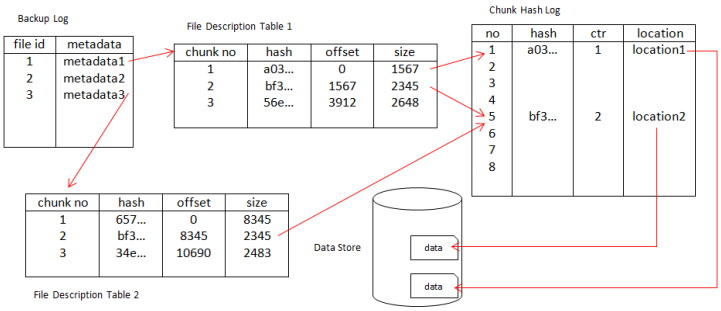


Fig. 3. Metadata Log Layout

The backup log contains a historical record of users' file backup requests. Each entry of the backup log represents the metadata of the backup file. In order to ensure that the backup files can be understood in the future, the metadata must contain a wealth of information such as filename, file size, format, original logical location at user computers, etc. Please note that every entry of the backup log is pointing to a file description table, which is generated by the database controller during deduplication.

The file description table contains 3 main fields: cryptographic hashes that uniquely identify the chunks, the offset of the chunks relative to the first byte of file data stream, and the chunk sizes. This table describes the arrangement of the variable length chunks that compose a file. For example, a file object could be divided into a sequence of variable-length chunks after chunking stage and then the chunks are hashed by a cryptographic hash function to a sequence of hashes in indexing stage. By listing the hash of every chunk in order, we provide a description of the file object. By using this description table, the file object can be reconstructed by fetching the constituting chunks. Depending on the number

of files to which a specific chunk belongs, a corresponding count needs to be maintained, which is essential to determine whether a specific chunk should be removed (garbage collection) when files are deleted by the user.

4.3 System Interaction and Data Flow

We now describe how the InterCloud RAIDer modules and CSPs interact to realize data deduplication, data storage, data retrieval, and data verification.

Backup and Retrieve: In Figure 2, we illustrate the backup process through these numbered steps.

1. An user makes a file backup request to the system. The database controller generates metadata about the file and insert it into a backup log.
2. When the file data stream enters the deduplication engine, the engine chunks the data stream using TTTD (described in Section 2.1). At the indexing step, the chunks are hashed using a collision resistant hash function. We chose MD5 for this purpose, but any other cryptographic hash function could also be used instead. The MD5 hashes are used as the identifiers of the chunks since the probability of collision is negligible. A file description table as described earlier in Section 4.2 is accordingly created. Since the target system users are data owners with a small or moderate amount of data to be outsourced, the anticipated total number of chunks is relatively small compared to organizations and enterprises. Consider, for example, a store has 1TB of data (which is at the very upper range of our current targeted system usage) and the average chunk size is 32KB. Then there are roughly 2^{25} unique chunks. Assuming that we are using MD5 for indexing, then we need 512MB of RAM for deduplication. Hence, we choose to use full chunk index for deduplication, namely, the deduplication engine compares the incoming chunk hashes with the hashes in chunk hash log that be read and stored in RAM one at a time, and the new chunk is stored only if no entry is found for its hash in chunk hash log. However, more sophisticated techniques such as Data Domain Deduplication File System or Sparse Indexing can also be applied in order to achieve better deduplication performance. The database controller creates a new entry for a non-duplicate chunk in chunk hash log, and alternatively increases the reading of counters provided there was an existing identical chunk.
3. The non-duplicate chunks are passed to the coding engine. We chose the (7,3)-HSRC with the Galois Field $\mathbb{F}_{256}$ as the underlying field for the convenience of the byte operations in computer hardware, since $\mathbb{F}_{256}$ is corresponding to one byte. We also apply zero-padding to the non-duplicated chunks and up to two bytes are zero padded to the chunks so that the resulting block sizes are multiple of 3 bytes. On the average, if the average chunk size is 32KB, then the additional expense induced by zero-padding is one byte per 32KB, which is negligible.

4. After HSRC encoding, the encoded pieces of chunks are ready for storing. Suppose that we have three CSPs A, B, and C. We use the following allocation scheme for storing the encoded pieces on the CSPs: the system uploads two encoded pieces with the corresponding points 1 and α to storage provider A, two encoded pieces with the corresponding points α^2 and α^{26} to storage provider B, and two encoded pieces with the corresponding points α^{25} and α^{50} to storage provider C, and (optionally) stores the encoded pieces with the point α^{198} at user's local storage. The network controller is responsible for the network operations: it goes through the OAuth1.0 or OAuth2.0 authorization and authentication process to retrieve an access token from respective CSPs, and then uses the tokens to initiates multiple PUT or POST network requests to the CSPs to store the encoded pieces.
5. After completing the storage, the database controller keeps track of the locations of the encoded pieces in the chunk hash log for future reference.

For file retrieve, it is just the reverse steps of file backup and are indicated using roman numerals in the architecture Figure 2.

1. First, the users use the system GUI to choose an entry from the backup log for retrieving. The network controller will contact database controller to access necessary information, such as file description table or locations of encoded pieces on cloud, for file retrieving.
2. Then, the network controller initiate the network operations for connecting to the CSPs. By using the file description table and chunk hash log, the network controller will send multiple GET requests to the CSPs and download the necessary encoded pieces for decoding a chunk. If some encoded pieces are stored locally, only two other pieces are needed from a single CSP, otherwise multiple CSPs need to be contacted.
3. After that, the downloaded pieces (and the local encoded piece) are decoded by the Gaussian Elimination and Back Substitution at coding engine.
4. The retrieved chunks are then glued together to reconstruct the original file according to the arrangement of chunks in the file description table by gluing function, before returning the retrieved file to the user.

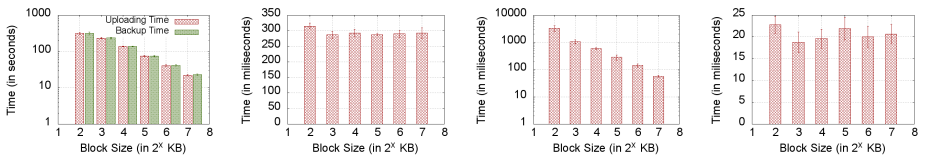
In addition, InterCloud RAIDer has several background processes - the most prominent one being the data integrity and retrievability checks (described in Section 2.3). There is also a garbage collection process, which needs to delete encoded blocks from all storage locations corresponding to files that are deleted by the user. Here, the main caution is to not inadvertently delete chunks that were duplicate across multiple files, and likewise, maintain the corresponding meta-data to determine when a chunk can indeed be garbage collected.

5 Experiments

We report the performance of InterCloud RAIDer from several aspects - looking at its overall performance, as well as micro-benchmarking the specific modules. As local user machine, an university PC running Windows 7 with 4GB RAM, and Intel CPU 6700 @ 2.66GHz was used for the experiments.

5.1 Impact of Chunk Size on Network Operations and Coding

By measuring the performance of InterCloud RAIDer that uses fixed-sized deduplication, we can narrow down the range of average chunk sizes to several candidates for optimality and then we will use these possible candidates to determine the performance of InterCloud RAIDer in Section 5.2. We have performed several series of experiments on a 1.12MB PDF file, 10.0MB DJVU file and 103.8MB RMVB file. Due to space limitation, we report only the case of 1.12MB PDF file, but the results scaled linearly with our experiments. While applying fixed-size chunking, the file is subdivided into 12KB, 24KB, 48KB, 96KB, 192KB and 384KB fixed-size chunks (the words *chunk* and *block* are used interchangeably) accordingly, and the chunks are passed to the remaining modules of InterCloud RAIDer for further processing, namely, encoding the chunks by (7,3)-HSRC and uploading the encoded pieces to cloud stores. After encoding, the sizes of the encoded pieces are 4KB, 8KB, 16KB, 32KB, 64KB and 128KB respectively. We measure the file access time, encoding time of (7,3)-HSRC, time of internal metadata manipulation, uploading time to CSPs, and total backup time of the 1.12MB PDF file against encoded piece sizes. We have plotted the results in Figure 4 with error bars by using 90% confidence interval.



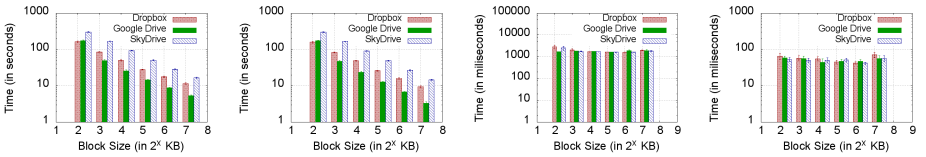
(a) Backup and Uploading Time (b) Encoding Time (c) Metadata Manipulation Time (d) File Access Time

Fig. 4. File Backup: Measurements of 1.12MB PDF File

From Figure 4(b), we see that the encoding time is consistent, since encoding is done byte wise, and only the absolute file size matters. Figures 4(a) and 4(c) show that the backup, uploading or metadata manipulation time decrease with increased chunk sizes. This is because fewer network requests are involved.

We also show in Figure 5 the file write time, decoding time of (7,3)-HSRC, downloading time from CSPs, and total retrieval time vs. encoded piece sizes. Note the huge discrepancy of network operation times across different service providers. Similar behavior was also observed for data upload (not shown here).

Figure 5(c) shows the decoding time of the 1.12MB file by using Gaussian Elimination and Back Substitution and it is more expensive than encoding. This is because the Gaussian Elimination has arithmetic complexity of $O\left(n^3\right)$. Similar to the trend of measurements in file backup, Figures 5(a) and 5(b) show the decreasing trend when the chunk size is increasing. For both series of experiments, the network operations like uploading and downloading always are the dominant factors in determining either the total backup or retrieve time.



(a) Retrieve Time (b) Downloading Time (c) Decoding Time (d) File Write Time

Fig. 5. File Retrieve: Measurements of 1.12MB PDF File

5.2 Deduplication in InterCloud RAIDer

While larger chunk size improves network operations, they have detrimental impact on deduplication as well as effectiveness of PDP. Based on the above experiments, we used variable length chunks with min/max thresholds configured to achieve mean sizes of 32KB and 64KB respectively, since they represent the middle of two extremes. We have chosen a real MS Word document with ten versions which were created by deleting, appending, or modifying the content from version to version. The modifications between two consecutive versions varied from 5% to 50% and the sizes of the versions ranged from 797KB to 1.71MB. This created a total data of 13.27MB. We then use the *backup* interface of InterCloud RAIDer to make incremental backup requests on the ten versions of the file and upload to cloud stores. The results are shown in the Table 1. Deduplication ratio reflects the factor of reduction in storage (and data exchange) needed w.r.to storing every chunk of every file instance.

As shown Table 1, smaller chunk size yields better deduplication ratio, 2.23 for 32KB vs. 1.93 for 64KB. However, if 32KB is used, the total time for incremental backup of ten versions of the MS Word document is 1.55 times higher than using 64KB average chunk size.

Table 1. Comparison of 32KB and 64KB Average Chunk Sizes for Deduplication

Average chunk size	32KB	64KB
Total chunking time (in sec) for ten versions	41.90	46.40
Total time (in sec) of full chunk index for ten versions	1.69	1.22
Total metadata manipulation time (in sec) for ten versions	19.16	8.37
Total time (in minutes) of incremental backups for ten versions	14.35	9.23
Total data (in MB) to be uploaded after deduplication	5.939	6.871
Deduplication Ratio	2.23	1.93

5.3 Performance of PDP

We assess the overhead of the PDP setup phase by measuring the time to compute a verification token, and then summing up the overhead of the individual

Table 2. Computation Time of Each Verification Token Covering r Hashes

r	64	128	256	512	1024
Time (in ms) for computing a token	28	43	63	171	291

tokens to get overall cost. Each token contained r random hashes (see Section 2.3), and we tried up several values of r. The result is shown in Table 2.

The computations in verification phase are just repetition of the setup phase. If there are 1%, 2% or 5% corrupted chunks in data repository and data owners are seeking to have 99.9% detection probability, then the downloading time for the necessary random hashes are 25 mins, 13 mins and 5 mins respectively. The time estimates are based on 48KB fixed-size chunks being downloaded from the CSPs. This gives a rough estimate about the network overhead in our PDP scheme.

6 Concluding Remarks

We presented a holistic system which incorporates mechanisms for deduplication, data dispersal over multiple clouds and proof of data possession to facilitate back-up of private data in multicloud environment. The system design had to take into account the dependencies and constraints arising from the conflicting needs of the different modules, as well as the artefacts of the cloud service provider. The performance of the current implementation is modest, and while a more optimized implementation of the individual algorithms and modules can help improve it to certain extent, the performance shortcomings also expose the need to design more scalable algorithms for the individual tasks to achieve a better performing multi-cloud storage system while not using well provisioned cloud services in contrast to what is inherently assumed in designing systems like RACS [1] and DepSky [2].

The current version of InterCloud RAIDer implementation does not leverage on the repairability property of HSRC since the cloud stores used did not have computational power. We intend to extend the current implementation to work with other cloud services, as well as with P2P assisted hybrid architectures (similar as [26]) where the issues of repairability will be explored.

References

1. Abu-Libdeh, H., Princehouse, L., Weatherspoon, H.: RACS: A Case for Cloud Storage Diversity. In: SOCC (2010)
2. Bessani, A., Correia, M., Quaresma, B., Andre, F., Sousa, P.: DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds. In: EuroSys 2011 (2011)
3. Sharma, R., Datta, A., Dell'Amico, M.: An empirical study of availability in friend-to-friend storage systems. In: P2P (2011)
4. Hewlett-Packard, "Understanding the HP Data Deduplication Strategy: Why One Size Doesn't Fit Everyone" (2008), `http://networkworld.com/documents/whitepaper/HPDataDeduper.pdf`

5. Muthitacharoen, A., Chen, B., Mazieres, D.: A Low-bandwidth Network File System. In: SOSP (2001)
6. Eshghi, K., Tang, H.K.: A Framework for Analyzing and Improving Content-Based Chunking Algorithms. HP Labs Tech. Rep. HPL-2005-30(R.1) (2005)
7. Manber, U.: Finding Similar Files in a Large File System. USENIX ATC (2004)
8. Forman, G., Eshghi, K., Chiocchetti, S.: Finding Similar Files in Large Document Repositories. In: KDD (2005)
9. Quilan, S., Dorward, S.: Venti: A New Approach to Archival Storage. In: FAST (2002)
10. Zhu, B., Li, K., Patterson, H.: Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In: FAST (2008)
11. Lilibridge, M., Eshghi, K., Bhagwat, D., Deolaikar, V., Trezise, G., Campbell, P.: Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In: FAST (2009)
12. Rivest, R.: The MD5 Message-Digest Algorithm. IETF, Request For Comments (RFC) 1321 (1992), http://tools.ietf.org/html/rfc1321
13. National Institute of Standards and Technology, "Secure Hash Standard", FIPS 180-1 (1995), http://www.itl.nist.gov/fipspubs/fip180-1.htm
14. National Institute of Standards and Technology, "Secure Hash Standard", FIPS 180-4 (2012), http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf
15. Bellare, M., Canetti, R., Krawczyk, H.: Keying Hash Functions for Message Authentication. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 1–15. Springer, Heidelberg (1996)
16. Ateniese, G., Pietro, R.D., Mancini, L.V., Tsudik, G.: Scalable and Efficient Provable Data Possession. In: SecureComm (2008)
17. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Song, D.: Provable Data Possession at Untrusted Stores. In: CCS (2007)
18. Juels, A., Kaliski, B.: PORs: Proofs of Retrievability for Large Files. In: CCS 2007 (2007)
19. Oggier, F., Datta, A.: Self-repairing Homomorphic Codes for Distributed Storage Systems. In: Infocom 2011 (2011)
20. http://techcrunch.com/2011/06/20/dropbox-security-bug-made-passwords-optional-for-four-hours/
21. http://blogs.computerworld.com/carbonite_loses_7500_customers_files
22. http://gigaom.com/2009/10/10/when-cloud-fails-t-mobile-microsoft-lose-sidekick-customer-data/
23. http://www.pcworld.com/article/226128/Sony_Makes_it_Official_PlayStation_Network_Hacked.html
24. http://news.cnet.com/8301-1009_3-57448465-83/linkedin-confirms-passwords-were-compromised/
25. http://broadcast.oreilly.com/2011/04/the-aws-outage-the-clouds-shining-moment.html
26. http://www.spacemonkey.com/