FAULT TOLERANT CONVERGENT KEY MANAGEMENT IN OBJECT-BASED STORAGE

A DISSERTATION

Submitted in partial fulfillment of the
requirements for the award of the degree
of

MASTER OF TECHNOLOGY
in
COMPUTER SCIENCE AND ENGINEERING

By

RONAK LAKHWANI

(13535043)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE – 247 667 (INDIA)

MAY, 2016

DECLARATION OF AUTHORSHIP

I declare that the work presented in this dissertation with title **“Fault tolerant convergent key management in Object-Based Storage”** towards the fulfillment of the requirement for the award of the degree of Master of Technology in Computer Science & Engineering submitted in the Dept. of Computer Science & Engineering, Indian Institute of Technology, Roorkee, India is authentic record of my own work carried out during the period from July 2015 to May 2016 under the supervision of **Dr. Sateesh K. Peddoju**, Assistant Professor, Dept. of CSE, IIT Roorkee.

The content of this dissertation has not been submitted by me for the award of any other degree of this or any other institute.

DATE: ……………….. SIGNED: …………………….

PLACE: ………………. (RONAK LAKHWANI)

CERTIFICATE

This is to certify that the statement made by the candidate is correct to the best of my knowledge and belief.

SIGNED: …………………….

DATE: ……………….. (**Dr. Sateesh K. Peddoju**)

Place: Roorkee

Assistant Professor

DEPT. OF CSE IIT Roorkee

ACKNOWLEDGEMENTS

FIRST and foremost, I would like to express my deep sense of gratitude and regards to my research supervisor **Dr. Sateesh K. Peddoju,** Assistant Professor, Dept. of Computer Science & Engineering, Indian Institute of Technology, Roorkee, for his trust in my work, his invaluable guidance, regular source of encouragement and assistance throughout the course of dissertation work. His wisdom, knowledge and commitment to the highest standards inspired and motivated me. He has been very generous in providing the necessary resources to carry out my research. He is an inspiring teacher, a great advisor and most importantly a nice person.

I would like to thank **Railtel Corporation of India Ltd. (Grant code: RCI-763(3)-ECD)** for providing financial assistance for procuring various hardware to setup cloud in High Performance Computing Lab.

On a personal note, I would like to say that I am indebted to my family for everything that they have given to me. I thank my parents, my sisters and my friends for providing constant support, encouragement and care.

Lastly, I thank almighty for the wisdom and perseverance that he bestowed upon me during my research.

RONAK LAKHWANI

ABSTRACT

An increasing amount of data is being stored in cloud-based storage services and this trend is expected to grow in the coming years. This creates a huge demand for systems and algorithms that are more efficient in the use of storage resources while being able to meet necessary cloud requirements of availability and scalability. A major source of inefficiency on cloud storage systems is data duplication, since bandwidth and storage resources are wasted to transfer and store data that is already present in the system.

Data deduplication is the technique to detect duplicate and store only one instance of the data. Different users of the system can put a reference to the data in case data belongs to multiple users. Duplicates can be detected at two levels i.e. File level and Block level. File level means the file is duplicated and Block level means file is broken up into pieces called blocks and then duplicates can be detected at block level. The way to detect duplicates is to hash the file or blocks and then detect duplicates based on hash value.

Much of the work is already being done in the area of data deduplication but none of them addresses the security aspect i.e. will the same data deduplication solution work when considering the data in the encrypted form. The problem with encrypted data is the same data after encryption through multiple users keys can produce different cipher texts and hence hash value solution to detect duplicates doesn’t work. The idea is to take the key (to encrypt) data from the data itself and then use that key to encrypt the data. The key is referred to as the convergent key and the encryption is referred to as the convergent encryption [10]. The drawback of this encryption scheme is that it is prone to brute force attacks and then can easily be compromised.

The proposed architecture addresses the problems of convergent encryption [10] and provides efficient key management such that system is fault tolerant in cases when user have limited access to storage servers. The solution segregates the key and data on multiple storage servers so that attackers cannot easily attack and hence security is not compromised.

Contents

List of Figures

List of Tables

1 INTRODUCTION

An increasing amount of data is being stored in cloud-based storage services and this trend is expected to grow in the coming years. According to the analysis report of IDC, the volume of data in the wild is expected to reach 40 trillion gigabytes in 2020[1]. This creates a huge demand for systems and algorithms that are more efficient in the use of storage resources while being able to meet necessary cloud requirements of availability and scalability. A major source of inefficiency on cloud storage systems is data duplication, since bandwidth and storage resources are wasted to transfer and store data that is already present in the system. Studies claims that up to 75% of all produced data is duplicated[2]. Today's cloud storage providers such as Dropbox[3], [4] Google drive[5] and Mozy[6] have been applying deduplication to save maintenance cost.

1.1 Motivation

Data deduplication is the technique to detect duplicate and store only one instance of the data. Different users of the system can put a reference to the data in case data belongs to multiple users. Duplicates can be detected at two levels i.e. File level and Block level. File level means the file is duplicated and Block level means file is broken up into pieces called blocks and then duplicates can be detected at block level. The way to detect duplicates is to hash the file or blocks and then detect duplicates based on hash value.

The technique has been successfully used in backup and archival storage systems ([7], [8], [9]), and its use is becoming popular for primary storage. Much of the work is already being done in the area of Data deduplication but none of them addresses the security aspect i.e. will the same data deduplication solution work when considering the data in the encrypted form. The problem with encrypted data is the same data after encryption through multiple users keys can produce different cipher texts and hence hash value solution to detect duplicates doesn't work. The idea is to take the key (to encrypt) data from the data itself and then use that key to encrypt the data. The key is referred to as the convergent key and the encryption is referred to as the convergent encryption [10]. The drawback of this encryption scheme is that it is prone to brute force attacks and then can easily be compromised.

The thesis concludes by presenting the fault tolerant and efficient key management architecture to store the encrypted data on top of cloud storage provider such that the entire system is not easily compromised.

1.2 Problem Statement

The Problem Statement representing this thesis is "Fault Tolerant Convergent Key management in Object-Based storage". The thesis proposes an architecture which tries to add fault tolerance and resilient features to the existing architecture. It also tries to manage the key storage servers such that key storage becomes independent with the number of users sharing the file/block.

1.3 Organization of Thesis

The entire report is comprises of five chapters including this chapter which states the introduction, motivations and problem statement. The rest of the thesis report is organized as follows:

Chapter 2 describes the background details of the components used in our proposed architecture such as Storage, Object Deduplication along with its types (on the basis of various parameters), Distributed Hash Table and Erasure Encoding.

Chapter 3 describes the related work done in the area of Encryption along with Object Deduplication. It describes the details about the encryption and the need for it. The Chapter also discusses about the System Requirements along with the current existing models. At last, a comparison is made between different existing approaches.

Chapter 4 discusses about the limitations of the current existing architectures. It then describes the approach to mitigate those issues along with the implementation details. At last, it shows the results achieved through experiments

Chapter 5 finally concludes and discuss about the future work of our framework.

2 Background

First section introduces and discusses the basic concepts of the types of storage services available by cloud providers and describe the use case for the type of storage service. It also emphasizes on the Object storage as deduplication uses Object storage to identify the duplicates by storing the extra metadata along with the Object itself.

Second section discusses the basic concept of Data deduplication. What is the motivation behind it and what are the components where it could be applied along with the pros and cons of each type of data deduplication? It also discusses the chunking algorithms, which basically divide the file into multiple blocks of fixed/variable size along with the discussion of the advantages and disadvantages of each type of chunking techniques. At the end of this section, it also discusses the type of hashing techniques available in order to detect duplicates along with the prerequisites for choosing the hash function.

Third section discusses the technique to implement dictionary in the distributed environment. Dictionary requires constant look up time and requires minimum keys to be rehashed due to failure of some of the nodes in the distributed environment. As it is very clear that Failure are the norms rather than the exceptions in the distributed environment. At the end of this section, it introduces consistent hashing algorithm which works very efficiently and rehashes a very few keys due to node failures in the distributed environment.

Fourth section discusses the erasure coding technique to segregate the data into multiple fragments. Erasure coding technique is used in our proposed architecture in order to bring the fault tolerant feature to the entire system so that clients are still able to access the data even if he/she has access to some limited servers.

2.1 Storage

A primary service offered by IaaS cloud providers is persistent storage, which can be of several types according to applications' needs and storage requirements. The most important categories of cloud storage offerings are summarized below.

1. **Block Storage:** Provides block storage capabilities through a low-level computer bus interface, such as SCSI or ATA, over the network. An operating system using cloud-based block storage will operate on the block volume just like it would in a normal disk. This type of storage is indicated for persisting volumes on virtual machines hosted on IaaS clouds. Example: Amazon Elastic Block Store (EBS).
2. **Object Storage:** Provides object storage capabilities through a simple get/put interface generally through a HTTP/REST protocol. This enables applications to store and retrieve objects from within the user-space, without the need to install and configure a file system. This type of storage is especially suitable for storing static/unstructured data, such as media files or virtual machines images. Example: Amazon Simple Storage Service (S3), Google Cloud Storage, Rackspace Cloud Files.
3. **Database Storage:** Provides database storage capability, generally to a non-relational database (NoSQL), through a simple query interface. This enables applications to use a database without the need for configuring a local database and dealing with database administration. This type of storage is useful for storing structured content or application's metadata, which is typically small in size. Example: Amazon DynamoDB, Google App Engine Datastore.

2.1.1 Object Storage

Object-based storage provides a simple and scalable model for cloud computing storage. While block and file-based storage are important for a wide-range of applications, a simpler object-based storage model is sufficient for storing many types of unstructured data, including media assets, application data and user-generated content. Moreover, the object-based storage interface is completely decoupled from the actual underlying storage technology, allowing data to be transparently stored by the cloud provider. This independence allows the provider to migrate data, to account for server failures, or even replace the storage infrastructure without needing to reconfigure clients. The basic advantages of Object storage are:

1. It is most suitable for unstructured data such as photos, videos, audios etc.
2. Users have the ability to store extra metadata along with the object.

3. In Object Storage, user can directly access the object using its object id.

Object Storage manages access to the files by storing the metadata per file basis. The advantage of storing the metadata could be files could be self descriptive and even by integrating access rights in metadata file, an authorized user can decipher an encrypted file only with his private key[11]. Several cloud providers currently offer an object-based storage service: Amazon Simple Storage Service(S3), Google Cloud Storage, Windows Azure Blob Storage, Rackspace Cloud Files, etc.

2.2 Data deduplication

Data deduplication comprises a set of compression techniques that improves space utilization in a storage system by reducing the amount of unnecessary redundant data. These techniques can also be used to avoid transfer of duplicated data between different hosts, thus improving bandwidth utilization. The thesis uses deduplication as a key technique to achieve storage and bandwidth savings in an object-based storage system.

Deduplication is generally done by identifying duplicate chunks of data and keeping just one copy of these repeated chunks. New occurrences of chunks that were previously stored are replaced by a reference to the already stored data piece. These references are typically of negligible size when compared to the size of the redundant data. The technique can be applied within a file or object (intra-deduplication) or between different files (inter-deduplication) [12].

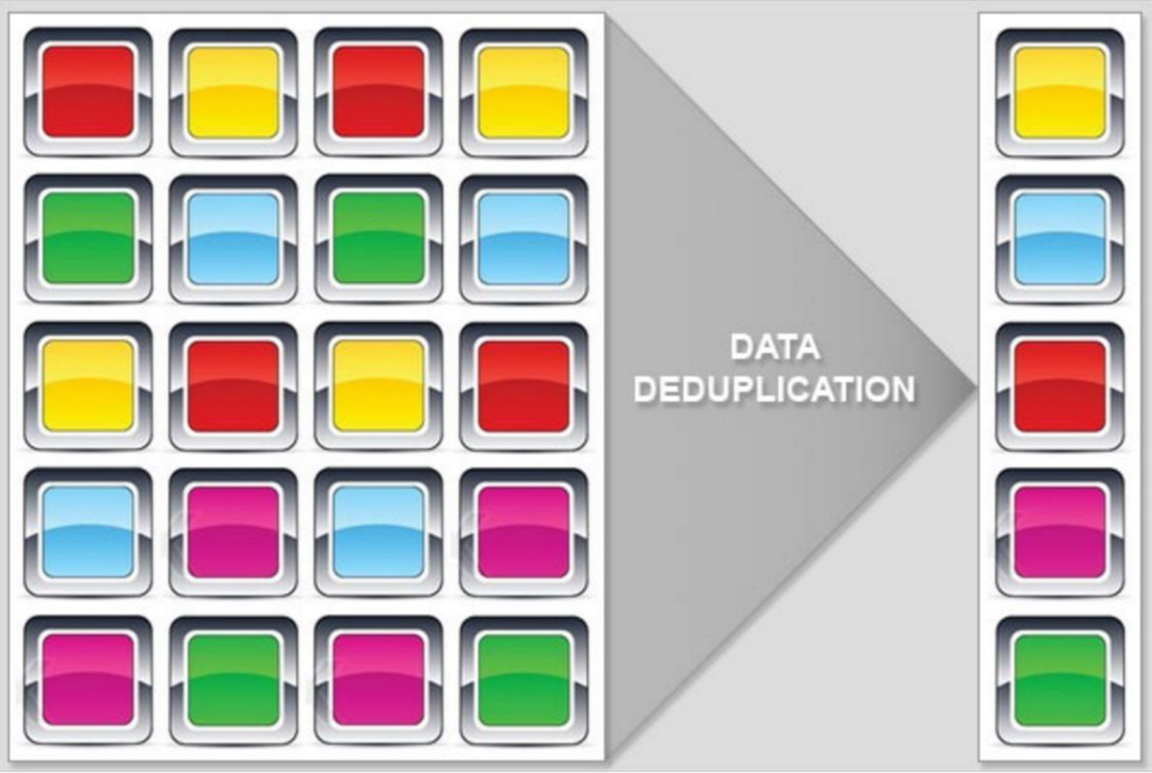


Figure 1: Showing how deduplication can save storage space[13]

Above Fig. 1 indicates the advantage of deduplication. The left hand side shows 20 objects, which are deduplicated, and hence instead of storing all the 20 objects, 5 objects can be stored and hence storage space can be saved. All the objects of the same color have same hash value and hence can be detected as duplicates by the cloud storage provider.

Duplicates in the storage servers are detected using hash. In hash-based deduplication, each data chunk is identified by a hash of its contents (chunk fingerprint). The deduplication is performed by comparing fingerprints of fresh data with fingerprints of previously stored data. When a match is found, it means the data is already stored and must not be stored again. In this case, the data piece is replaced by a metadata reference that indicates the fingerprint of the chunk that was previously stored. In order to ensure that any chunk will have a unique fingerprint, a secure cryptographic function, such as SHA-1 or SHA-2 must be used to hash chunks. This is due to the collision-resistance property of secure cryptographic functions that make it unfeasible for two data pieces to have exactly the same digests.

Since in this type of deduplication a data chunk is shared by many distinct files/objects, the server must track the files that reference a particular data chunk (back-references). This will enable the server to garbage collect chunks that are no longer referenced by any file. Reference tracking can be done by reference counting, where each data chunk has a counter indicating the number of files that currently reference that chunk.

2.2.1 Choice of Hash Function

A secure cryptographic hash function must be used to fingerprint data before it is inserted in the storage system. In order to increase deduplication efficiency, a single function must be used system-wide, in all clients and storage servers. Since the storage system will handle at least tens of terabytes and possibly petabytes of data, it is important that the chosen function is:

- **Collision-resistant:** It should be very difficult for two different data chunks to have the same fingerprint; otherwise this could cause data-loss since the system is based on the assumption that each piece of data has a unique fingerprint.
- **Preimage-resistant:** It should be very difficult to discover the fingerprint of a data chunk without having the actual chunk. Since the system doesn't authenticate data operations, a malicious user could fetch unauthorized pieces of data by guessing its fingerprint.
- **Second-preimage-resistant**: It should be very difficult to discover two data chunks with the same fingerprint. Since the system is based in the assumption that each data chunk has a unique fingerprint, a malicious user could store bogus data with a valid fingerprint that is the same as popular data chunk (such as a song or VM image), preventing users to store or retrieve the valid data piece.

2.2.2 Chunking

An important aspect in hash-based deduplication systems is the technique used to break data containers (such as file or object) into chunks. This will directly influence the deduplication efficiency of a system, since the more chunks in common, the less data needs to be stored. A simple approach is to use an entire data container as the unit to perform the digest calculation: whole-file hashing. Sub-file hashing or chunk-level deduplication splits a file into multiple chunks and performs a more fine-grained deduplication. The latter approach can use static or variable chunk sizes. Each of these file-splitting techniques are presented and discussed in the following subsections.

1. **Whole File Hashing (or File-level deduplication):** This method of hash-based deduplication calculates the fingerprint of the entire file to identify duplicated content. The major benefit of this approach is increased throughput, since only a single hash calculation needs to be done per file. This technique is particularly efficient when there are many repeated files in a dataset. For instance, a music hosting service could use whole-file hashing to identify repeated MP3 files, which would be very common for popular songs. The downside of this approach is that it may yield little benefits in datasets with low file redundancy. Additionally, a single bit change in the file will generate a different file hash, requiring the whole file to be re-stored or re-transmitted. For this reason, this method of hash-based deduplication may not be efficient for files that change often.
2. **Fixed Chunk Hashing (or Static chunking):** In this approach, the chunking algorithm splits the file into fixed-size chunks starting from the beginning of the file. A cryptographic hash function calculates the secure digest of each chunk, and the fingerprint is used to verify if the chunk is already present in the system. The benefits of this approach include ease of implementation and fast chunking performance. Moreover, the chunk size can be aligned with the disk block size to improve I/O performance. In comparison to whole file hashing, this approach has lower throughput since a hash-table must be consulted on each chunk to verify if it is already present in the system. Examples of storage systems that use static chunking are Venti[14] and Solaris ZFS. Another downside of this method is what is known as the boundary-shifting problem: if a single bit is appended to or removed from a file, the fingerprint of all chunks following the update will change. Thus, it is not resistant to changes in the deduplicated file.

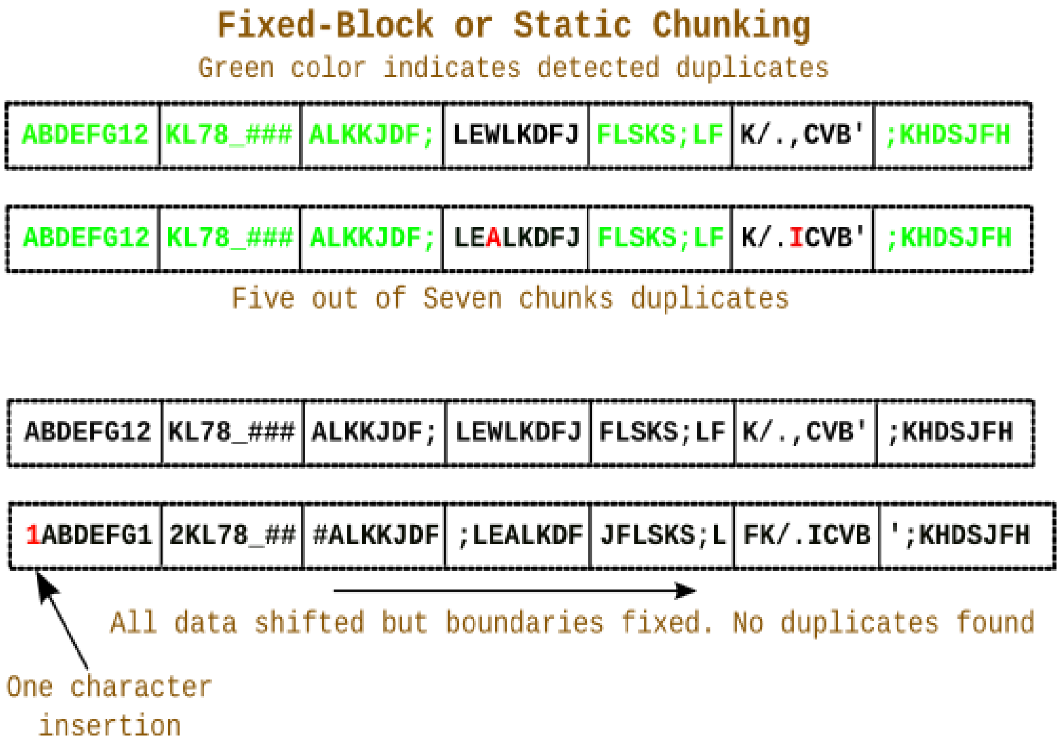


Figure 2: Fixed Chunking showing how one character insertion can be a problem[15]

Above Fig. 2 shows the Fixed or Static chunking. In the above part it is clearly visible that out of seven chunks, five of them are detected as duplicate and hence on the cloud storage provider, a total of 9 (7 +2) chunks are stored. In the below portion of the Fig. 2, it is shown that how fixed chunking can be a problem in case of scenarios where one byte of data is added to a random position. Due to such changes, all data preceding that change is shifted by one position and hence duplicates are not been detected by the cloud storage provider.

3. **Variable Chunk Hashing (or Content-defined chunking):** In variable chunking, a stream of bytes (such as a file or object) is subdivided by a rolling hash function that calculates chunk boundaries based on the file contents, rather than static locations across the file. For this reason, this method is also called content-defined chunking. After a variable-sized chunk boundary is defined, the chunk digest is calculated and used to verify if the chunk is already present in the system. One important property of content-defined chunking is that it has a high probability of generating the same chunks even if they are placed in different offsets within a file. This makes it resistant to the chunk boundary-shifting problem, because the

modification of a chunk does not change the fingerprint of remaining chunks. A popular method for variable chunking is Rabin fingerprinting which computes a modulo function across a rolling hash sliding window to define chunk boundaries. When the modulo is equal to a predetermined irreducible polynomial, a chunk boundary is determined. Variable chunk hashing is particularly efficient for deduplicating correlated data. Due to its resistance to the chunk boundary-shifting problem, it is also a good choice for dealing with dynamic data since changing a few bytes from the file does not change hash digests of other chunks. The major downside of this approach in comparison with the previous hash-based techniques is that it adds significant latency overhead to the deduplication process, due to the added step of calculating chunk boundaries. Variable chunk hashing is particularly efficient for deduplicating correlated data[29].

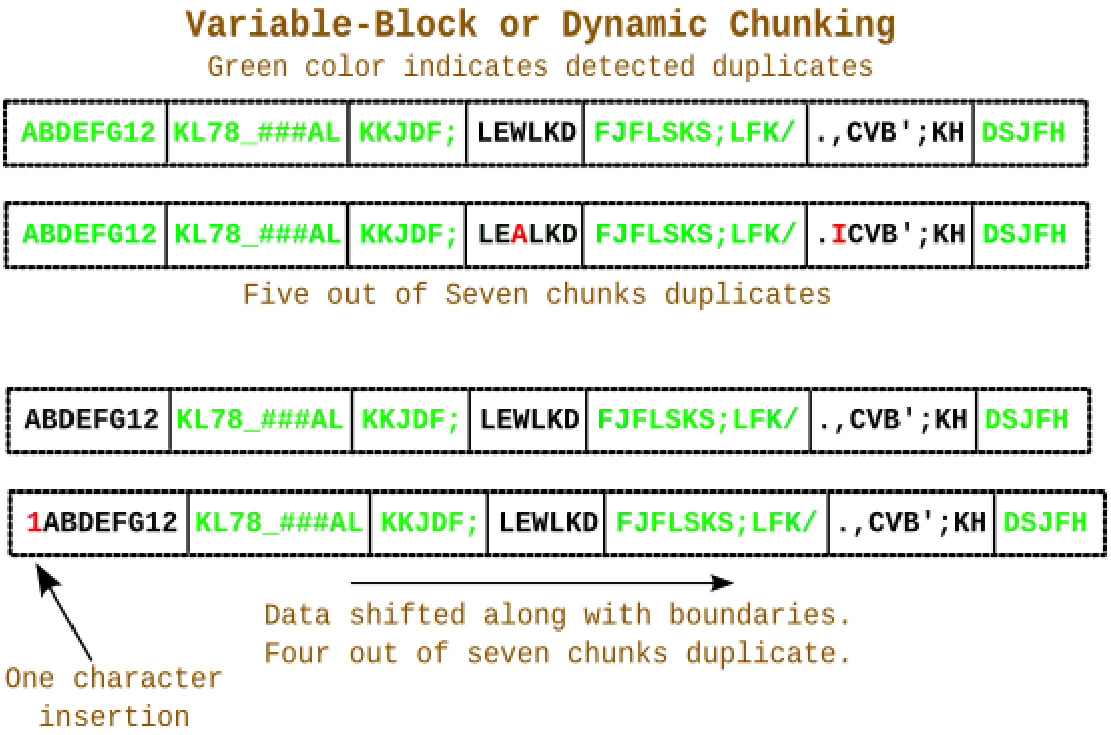


Figure 3: Variable Chunking showing how character insertion does not affect duplicate detection[15]

The above Fig. 3 shows the Variable size chunking. In the above part, it is shown that while uploading 14 blocks (of variable size) of data on the storage device, five of them are detected as duplicated and hence a total of 9 objects are stored on the storage disk. In the below part of the Fig. 3, it is shown that how variable size chunking is adaptive to the changes to the file and still 4 of the blocks are detected as duplicates.

2.2.3 Deduplication placement

The next dimension in the deduplication design space is related to where the deduplication process occurs: at the client (source) or at the deduplication appliance (target).

1. **Source Deduplication (Client-based):** In source deduplication, the client performs the deduplication procedure and exchanges metadata with the server to verify data duplication. For instance, when hash-based deduplication is used, the client calculates the fingerprint of each file chunk and sends this information to the server, to verify if that data piece exists on the appliance. Only unique data chunks – chunks that weren't previously stored - need to be transferred to the server. When the chunk already exists, the server updates the reference tracking data structures and the operation is completed. The major benefit of client-based deduplication is the potential network bandwidth savings when data is duplicated, since only small metadata and unique data needs to be transferred over the network. One drawback of client-based deduplication is that clients must spend computing and I/O cycles to perform deduplication, what may be a problem in devices with limited computation capacity or battery-constrained.

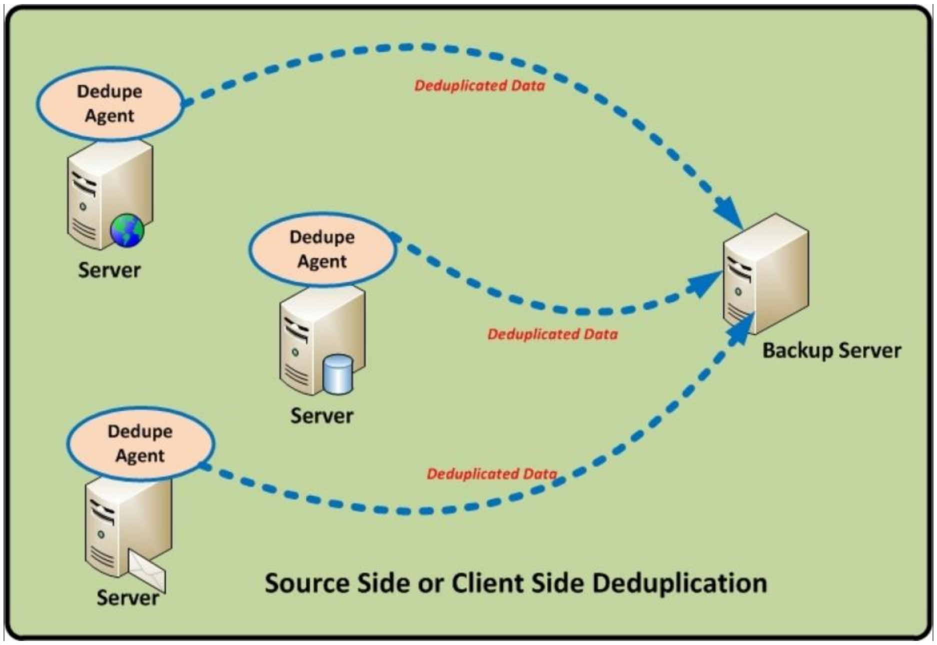


Figure 4: Source Side Deduplication[16]

Fig. 4 shows how Deduplication agents when trying to upload the deduplicated data are able to identify the duplicates. Backup server is only responsible for storing the object. Dedup Agents sends the hash to the backup server before sending the file and backup server replies to the Dedup agent about whether it has the same file/block on the storage disks or not. If it is found, then Dedup agents do not send that file/block to the backup server and hence end up saving a lot of bandwidth.

2. **Target Deduplication (Deduplication Appliance-based):** In target deduplication, the appliance performs the deduplication procedure. In this case, the appliance exposes a default storage interface and the client transfers the entire data that needs to be stored. In some cases, the client may even be unaware deduplication is being performed. When the deduplication appliance is located in another host and the communication is done over a network, this type of deduplication may be called server-based deduplication. Server-based deduplication is particularly beneficial for resource-constrained devices, since it consumes less computational resources from the client. Additionally, it is also a good choice to enable transparent deduplication on legacy applications or clients that cannot be upgraded, since

existing client code need not be modified to support target deduplication. A major drawback of server-side deduplication is that all data needs to be transferred over the network, so this approach provides no bandwidth savings. Examples of storage systems that use target deduplication is Venti[14].

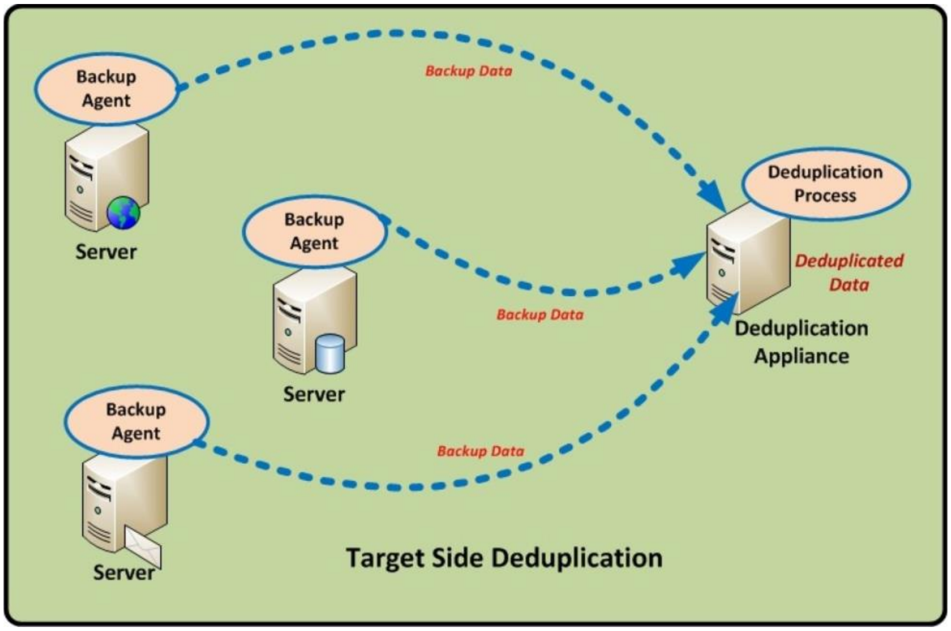


Figure 5: Target Side Deduplication[16]

Fig. 5 shows the Target Side Deduplication process. Dedup agents/ Backup agents upload the file to the Deduplication Appliance and then Appliance runs the process of deduplication. Target Side Deduplication is beneficial when clients do not have enough processing capability to run the deduplication process and hence the process can be made to run on the storage servers.

2.2.4 Deduplication Timing

With respect to timing, a deduplication system may perform the deduplication when the data is being inserted (synchronous), or post-process the data after it is already stored in the system (asynchronous).

1. **Synchronous (Inline deduplication):** When the deduplication procedure is performed during data insertion, before data is written to disk, it is called synchronous deduplication (also called inline, or in-band deduplication). This includes chunking the file or object, verifying if each data chunk is duplicated and generating metadata to represent the deduplicated file. The insert operation is

completed after the object metadata is generated and the unique chunks are written to disk. This approach matches well client-based deduplication, since the server is able to tell immediately if given data pieces are present in the system, avoiding transfer of duplicated data between the client and the server. One of the benefits of synchronous deduplication is that no additional storage space is required on deduplication appliances to stage objects for post-processing deduplication. Moreover, the system needs to maintain only one protocol for retrieving data, because it only deals with objects that were already deduplicated. However, these benefits come at the cost of increased latency during insertion operations, due to the overhead added by the deduplication process. However, part of this overhead can be mitigated by leveraging parallelism, for instance, by calculating fingerprints and transferring chunks in parallel.

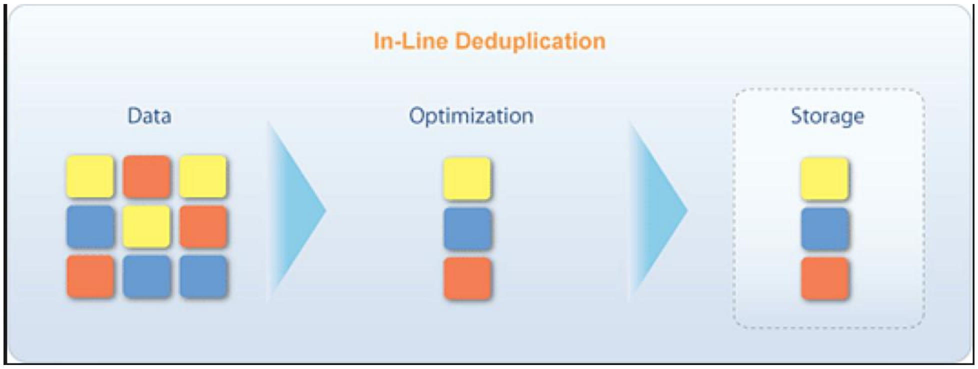


Figure 6: In-Line Deduplication[17]

Fig. 6 shows the process of In-Line Deduplication. All data when uploaded to the storage servers is first passed through the deduplication process before getting stored on the storage disks. Hence some part of the data is detected as duplicate and only the data, which is unique, is stored on the storage disks. This has the advantage that storage always have unique data at any point of time. The disadvantage to this deduplication strategy is client has to wait for the deduplication process to complete before getting its data uploaded on the storage servers.

2. **Asynchronous (Offline deduplication):** In order to avoid the extra step of deduplication in the write path, and its associated overheads, it is possible to perform deduplication in an asynchronous fashion. This method is also called offline or out-of-band deduplication. In this approach, the file/object is written to

the deduplication appliance as a whole to a special staging area. A background process deduplicates files that were recently inserted in the system, removing them from the staging area after the deduplication process is complete. The post-processing step is generally triggered during periods of low system load, as not to affect latency of other incoming requests. Since this deduplication method does not add latency overhead to the data insertion operation, it is a good choice for latency-critical applications, where the ingestion speed of data is a primary concern. The need to keep a staging area to store unprocessed data until it is deduplicated by a background process is one of the disadvantages of asynchronous deduplication. Another related disadvantage is to keep two protocols for reading data in the storage system: one for unprocessed data and another for deduplicated data.

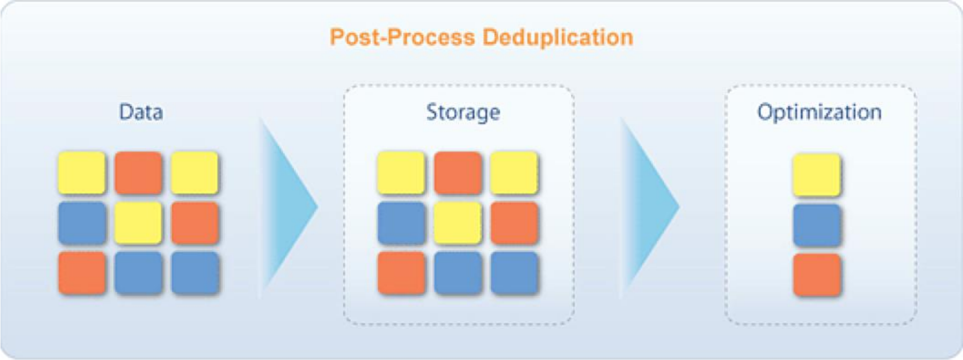


Figure 7: Offline Deduplication[17]

Fig. 7 shows the Offline Deduplication process. From the Fig. 7, it is clear that data is first uploaded on the storage servers and then some daemon running on the storage servers runs the deduplication algorithm to find the duplicates and then optimize the storage.

2.3 Distributed Hash Tables

Distributed hash tables (DHTs) provide an abstraction that maps keys onto nodes in a distributed environment. The most popular DHT protocols7 were conceived in the context of peer-to-peer (P2P) systems, which is a distributed architecture for resource sharing over a network where nodes (peers) act as servers and clients simultaneously. A fundamental operation in these systems is locating a peer responsible for keeping a particular resource. Previous solutions either involved maintaining a centralized lookup server (Napster 1999) or broadcasting a resource lookup message to all participants in the system (Gnutella 2000). While the former approach suffers from the single point of failure problem, both approaches present significant problems for scaling to a large number of nodes. DHTs were proposed in order to fill the gap of providing a scalable, decentralized solution for efficient resource lookup in a large distributed system. Several works have published solutions that leverage caching, bloom filters and chunk locality to mitigate the disk bottleneck problem[18], [19]. Some of these solutions can be integrated to further increase chunk lookup efficiency. Several distributed storage systems employ some form of distributed hash tables to efficiently distribute data across many nodes. The following hashing technique is used in order to implement distributed hash table in a distributed environment.

2.3.1 Consistent Hashing:

A straightforward method to map a set of keys into a set of nodes is a hash function. However, a change in the number of nodes causes nearly all keys to be remapped when a traditional hash function is used (such as: x -> (ax + b) mod p, where x is the key and p is the number of nodes in the system). When used in a distributed system this means that all resources that mapped to any node N, will now be mapped to node M, and possibly become inaccessible. Even though resources can be moved across nodes, this is an expensive operation which should be done as little as possible. In a dynamic distributed environment, where nodes come and go, it is essential that the hash function cause minimal disruption when the number of nodes changes. Consistent hashing is a hashing technique that remaps only O(K/N) keys (where K is the number of keys, and N the number of nodes) when the number of slots (nodes) changes. Moreover, the technique ensures that each node will be responsible for at most (1+∈)K/N keys with high probability, providing a degree of natural load balancing. Most distributed hash tables use some variant of consistent hashing to efficiently map keys to nodes. CEPH distributed file system[20] uses CRUSH[21] algorithm to identify the server storing the particular file. CRUSH[21], a variant of consistent hashing to dynamically map directory hierarchy to metadata servers based on current load.

Consistent hashing treats the output of a hashing function as a circular ring with cardinality m, where the largest hash value (2^m - 1) wraps around the smallest value (0). Each node is assigned a random m-bit identifier, where m should be large enough to make the probability of two entities having the same identifier negligible. This identifier defines the “position” of the node in the ring. The node responsible for storing a given key k is found by hashing k to yield its position in the ring and then walking the ring clockwise to find the node with identifier equal or larger to the item's position (successor). Thus, each node is responsible for the region in the ring between itself and its predecessor.

Fig. 8 shows an example consistent hashing ring for 3 nodes. Supposing that m = 3, id(A) = 2, id(B) = 4, id(C) = 6 and the task is to locate the node responsible for key k. Initially the value of k is hashed to find its location in the ring (indicated by the arrow), for instance, hash(k) = 1. The next node in the ring following 1 has identifier 2, so node A is responsible for storing k.

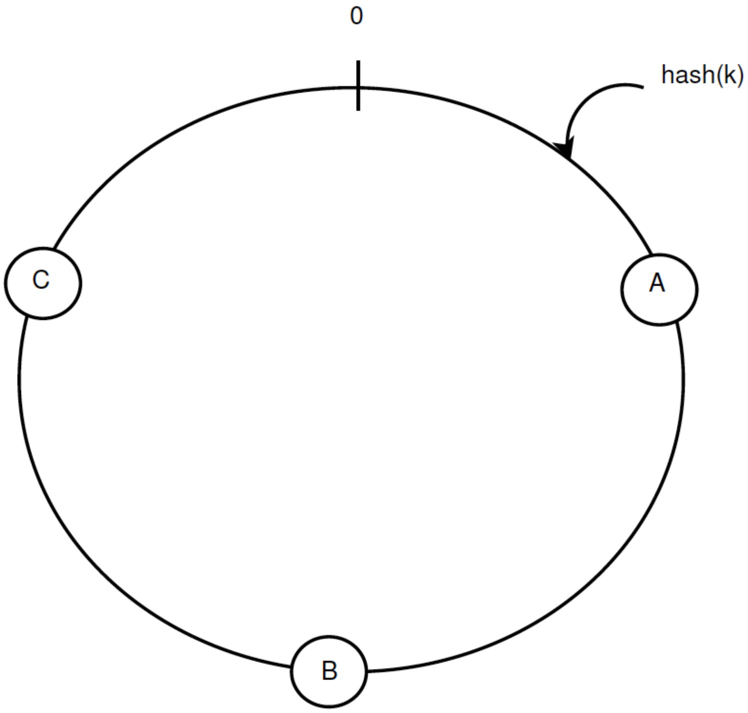


Figure 8: Consistent hashing ring for 3 nodes. A, B and C.

A major benefit of consistent hashing is that only the region a node is responsible for needs to be reassigned in case it leaves the system. Similarly, if a new node joins the system, it will gain responsibility over a fraction of its successor's previous region. So, there is no disruption in other regions of the ring.

In the example ring in Fig. 8, in case node A leaves the system, node B's region will grow and occupy the region that previously belonged to A. In this new configuration, the key k which previously mapped to node A will now map to node B.
Keys which previously mapped to node C will remain unchanged. In case a new node D is placed in the region between C and A, it will now become responsible for the region between C and D, which previously belonged to A. The remaining regions remain unchanged.

The basic algorithm can be enhanced by allowing each physical node to host multiple "virtual" nodes. In this approach, each virtual node has its own random identifier in the ring. This enhancement provides a more uniform coverage of the identifier space, thus improving load balance across system nodes. The downside of this optimization is that each physical node needs to store information (routing tables, etc.) for each of its virtual nodes.

2.4 Erasure Coding

Erasure coding is a forward error correction (FEC) technique, which transforms a message consisting of k parts into a bigger message with n parts such that the original message can be constructed from a subset of n parts. The use cases where erasure coding plays an important role is storage systems where Availability of users data is a primary concern and hence storage servers has to be available at all the time. As the data has to be available at all the time, failures will be there and hence there has to be some fault tolerant mechanism in order to ensure the 24 * 7 availability of the data. Fragmentation of data (composed of k parts) into n segments is referred to as **Encoding**.
The process to combine k parts together to form the original data is referred to as **Decoding**. Fig. 9 shows the basic flow diagram of Erasure Encoding. It has basically two components. Encoder & Decoder.

Encoder:

Encoder takes the original data and divides it into multiple fragments. This encoding process requires Encoder to add some redundant data to the original data in order to divide the data into n fragments.

Decoder:

Decoder takes at least k fragments (k <= n) and can get the original data back by decoding it.

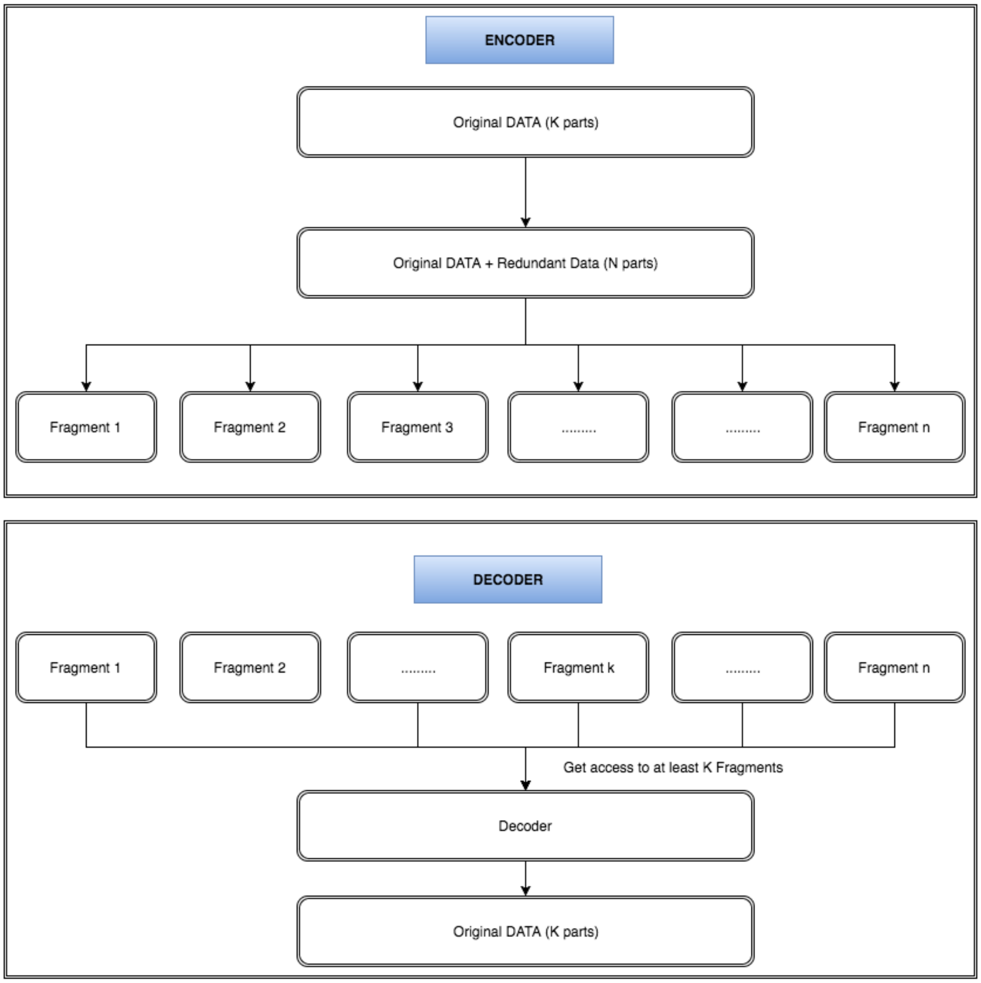


Figure 9: Erasure Coding Flow Diagram

Example

Lets say you have the data and you want to divide your data into 5 segments. Erasure coding first adds some redundant information to the data in order to divide it into 5 segments and the original data could be constructed if user is having access to at least 3 segments. Erasure Coding encoder takes the original data and adds the redundant data and divides the updated data into 5 fragments and gives it to the user. User can store these 5 segments on to multiple servers. User can get the original data after accessing at least 3 servers and get the fragments from them and hand it over to the Erasure encoding decoder. So by this mechanism, even if at any point of time, two of the servers are down due to power failure or network issues or any other issues, then user is still able to get back the original data. Below Fig. 10 shows the combined process.

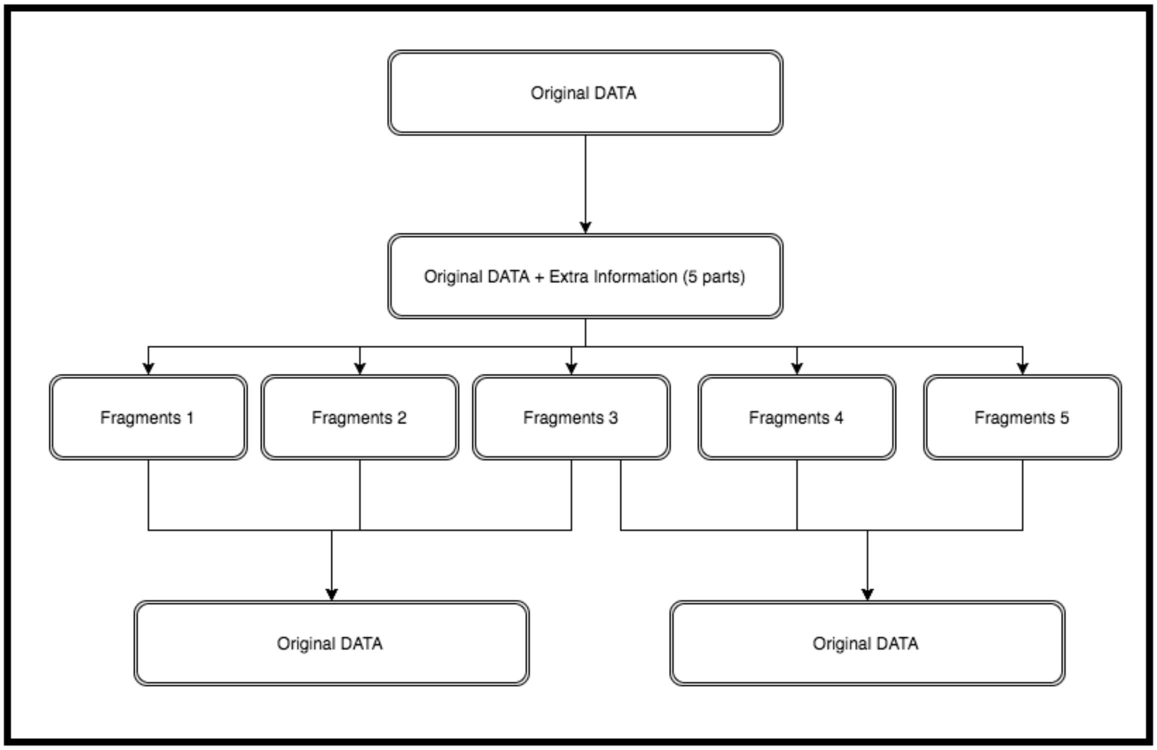


Figure 10: Erasure Coding example showing how original data is divided into 5 fragments and then can be constructed back from at least 3 fragments

2.4.1 Limitation

The basic limitation of Erasure Encoding is the overhead of adding redundant data to the original data. Although the additional storage added to the original data is max up to 40 percent of the size of the original data, it is an extra space the user has to bear in order to build fault tolerant system.

3 RELATED WORK

First section introduces and discusses what encryption is and what is the need of encryption and how it can be used to protect the confidentiality of the data?

Second section discusses about what are the system requirements that are to be kept in mind while designing the architecture for the cloud storage. What requirements could be compromised and what are the effects of those compromises to the cloud storage provider.

Third section discusses about the current approaches which are already available and are being used by some the cloud storage providers. This section also describes about what are the system requirements met from the current approaches along with their limitation.

Fourth section makes a comparison of all the current approaches and combine them into a table so that all the current approaches benefits along with the limitations are available at a single page.

3.1 Encryption

Encryption is the process of encoding messages or information in such a way that only authorized parties can read it. Encryption does not of itself prevent interception, but denies the message content to the interceptor. In an encryption scheme, the intended communication information or message, referred to as plaintext, is encrypted using an encryption algorithm, generating cipher text that can only be read if decrypted. For technical reasons, an encryption scheme usually uses a pseudo-random encryption key generated by an algorithm. It is in principle possible to decrypt the message without possessing the key, but, for a well-designed encryption scheme, large computational resources and skill are required. An authorized recipient can easily decrypt the message with the key provided by the originator to recipients, but not to unauthorized interceptors [22].

3.1.1 Need of Encryption

While encryption doesn’t magically convey security, it can still be used to protect a user's identity and privacy [23]. If we are ever being watched, inadvertently or not, we can hide our data by using properly implemented crypto systems. According to cryptographer and security and privacy specialist Bruce Schneier, “Encryption works best if it is ubiquitous and automatic. It should be enabled for everything by default, not a feature you only turn on when you’re doing something you consider worth protecting.”

3.2 System Requirements

The storage providers seeks minimum of these requirements in order to serve the clients. Hence the following requirements must be met in order to save the storage space along with providing necessary features such as Fault tolerance etc. While designing the system, it should be viewed from the attackers perspective as well such that there are no loopholes in the system and attacker cannot easily get into the system and decipher the data. Even if the attacker has access to some components of the system, it should be very difficult to get the deciphered data and system should be able to detect the attackers attempt. Our analysis considers two types of attacks. One is that the attacker has the encrypted data and tries to break the system. Another is attacker has access to the storage server somehow and then tries to get some extra information in order to get the deciphered data. In order to design storage systems, the system must have the following features:

1. **Deduplication:** Storage Providers should be able to detect the duplicates on the storage disks and store only one instance so that storage space can be utilized efficiently.
2. **Compromise resilience:** It should be very difficult for the attacker to get the deciphered data even he/she gets access to a few components of the system.
3. **Brute-force attack resilience:** Once attacker has the encrypted data, he/she can try to break the encryption or get the key by brute force attacks. The system should be designed in such a way such that such online brute force attacks are not possible and are detected by the system if someone tries to go for it. Such features could be added to the system by having some rate adaptive features so that client can access the storage server only after some epoch once client has already accessed it.

4. **Fault tolerance:** Clients should be able to access the contents that they uploaded on the storage servers at any point of time. Hence the storage servers should have 24 * 7 availability. As it is very clear from the fact that in case of distributed storage systems, Failures are the norms rather than the exceptions and hence there is a probability that the file requested by client is stored on the server, which is currently down. The file should be served to the client in any case and hence there should be some form of replication on the servers so that client has always access to the data even in case of node failures or network failures. Sometimes this may also be a case when client has access to the encrypted data, but does not have access to the key and hence client is not able to decipher the encrypted data. Our architecture mainly focusses on how to get efficient replication of the key such that system is fault tolerant.

3.3 Current Approaches

3.3.1 Client Specific Keys:

This scheme allows each of the clients to have the separate keys to encrypt their data. Each of the users will encrypt the file before getting it stored on the storage system using their own key. This scheme provides Compromise resilience against the untrusted users and brute force attack resilience against the attackers but the problem with this approach is duplicates are not detected and hence multiple instances of the same file are stored on the storage server.

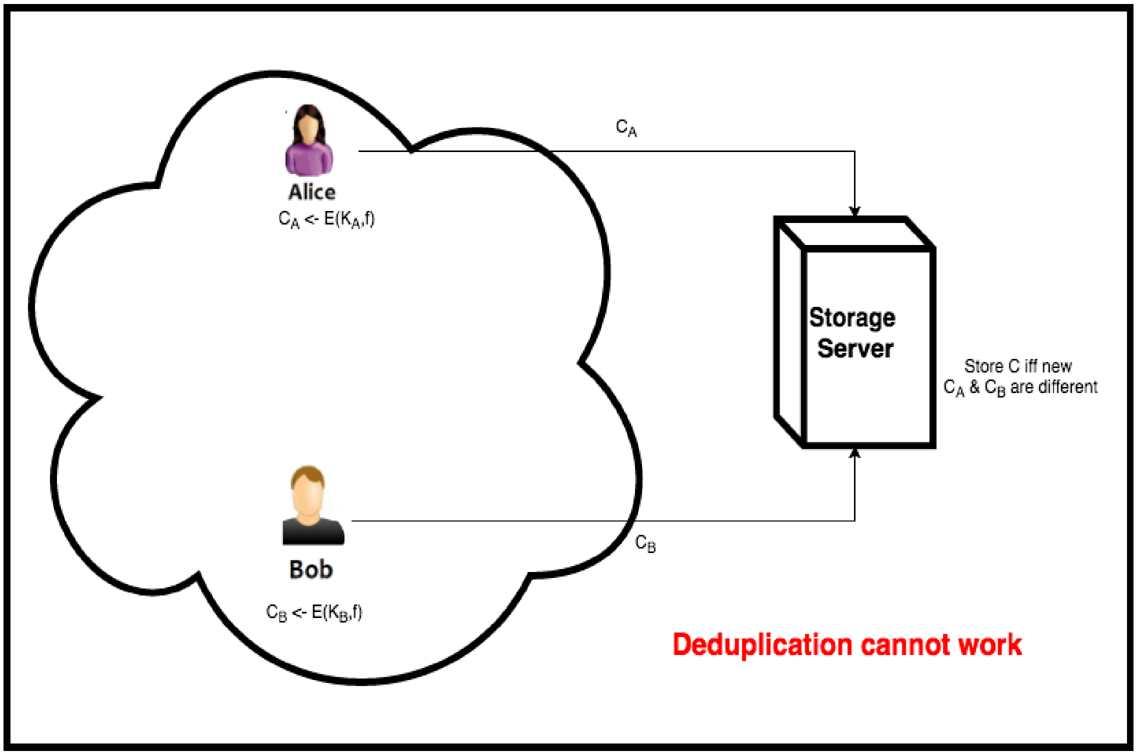


Figure 11: Client Specific keys

For example, lets have two users Alice and Bob. Both wants to store file f on the storage system S. Alice encrypts the file F using his own key K_A and gets the Cipher text C_A. Similarly Bob encrypts the file F using his own key K_B and gets the cipher text C_B. Now when at the time of applying Object deduplication on C_A and C_B, it identifies both of them as different and stores both of them. Fig. 11 illustrates this example.

3.3.2 Network Wide Key:

This scheme allows each of the clients to share the key to encrypt their data. Each of the users will encrypt the file before getting it stored on the storage system using the shared key. This scheme allows brute force attack resilience against the attackers and Object deduplication also works but the scheme is highly prune to Compromise resilience against the untrusted users.

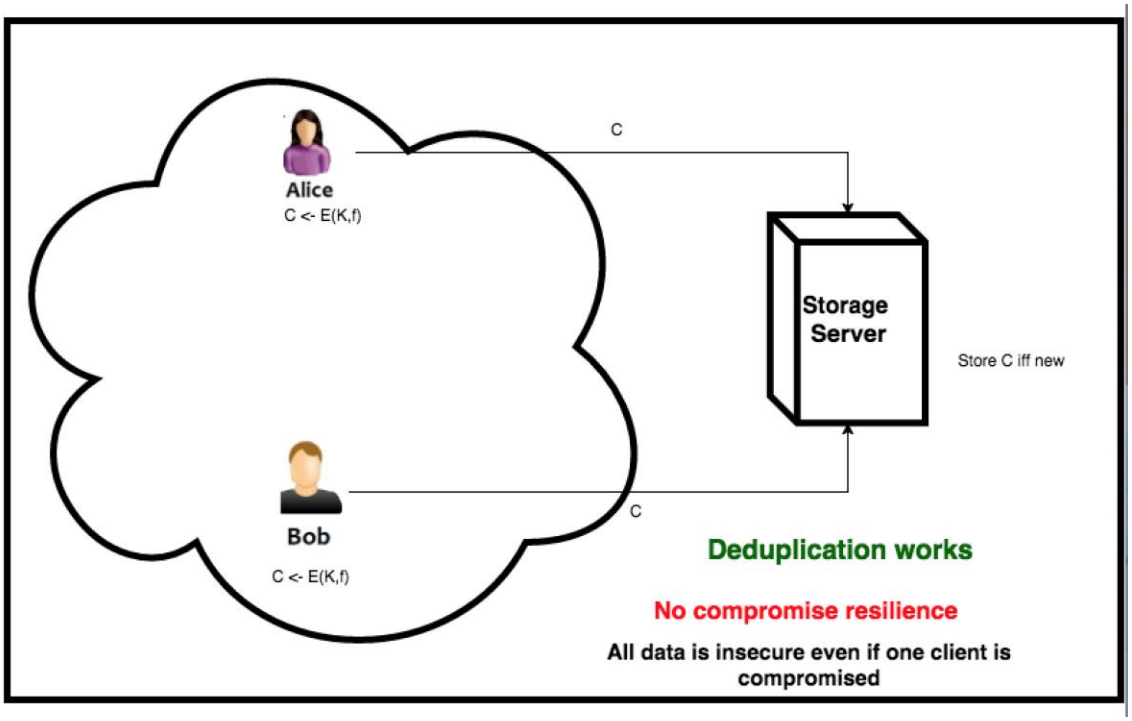


Figure 12: Network Wide Key

For example, lets have two users Alice and Bob. Both wants to store file f on the storage system S. Alice encrypts the file F using shared key K and gets the Cipher text C. Similarly Bob encrypts the file F using shared key K and gets the cipher text C. So when multiple users store C, it is detected as duplicate and hence only one instance is stored on the storage server. The bottleneck of this approach is if the key is compromised somehow to an untrusted user, then he/she can access the entire files and hence the system is prune to Compromise resilience. Fig. 12 illustrates this example.

3.3.3 Convergent Encryption

This scheme allows the users to have their own key. This encryption works in two phases and hence for each file, there are two objects stored on the storage server.

1. The encrypted file
2. Encrypted Convergent key

Explanation of convergent key is as follows. The idea is to get the key from the data itself and use that key to encrypt the file. The key, which is fetched from the file, is referred to as the convergent key. Convergent key can be obtained by computing the cryptographic hash value of the content of the data copy itself. Convergent key obtained for the same file from different users will be same and hence will produce the same encrypted file. Convergent key itself can be encrypted using users own secret key and hence encrypted convergent key for two users sharing the same file will be different and hence will not be able to detect as duplicates by the storage providers. As the keys are 48 bytes, this much storage overhead could be considered if the system is dealing with petabytes of data. Now let's consider how the encryption and decryption flow works through a block diagram. Fig. 13 & Fig. 14 illustrates the Encryption and Decryption process respectively.

Encryption:

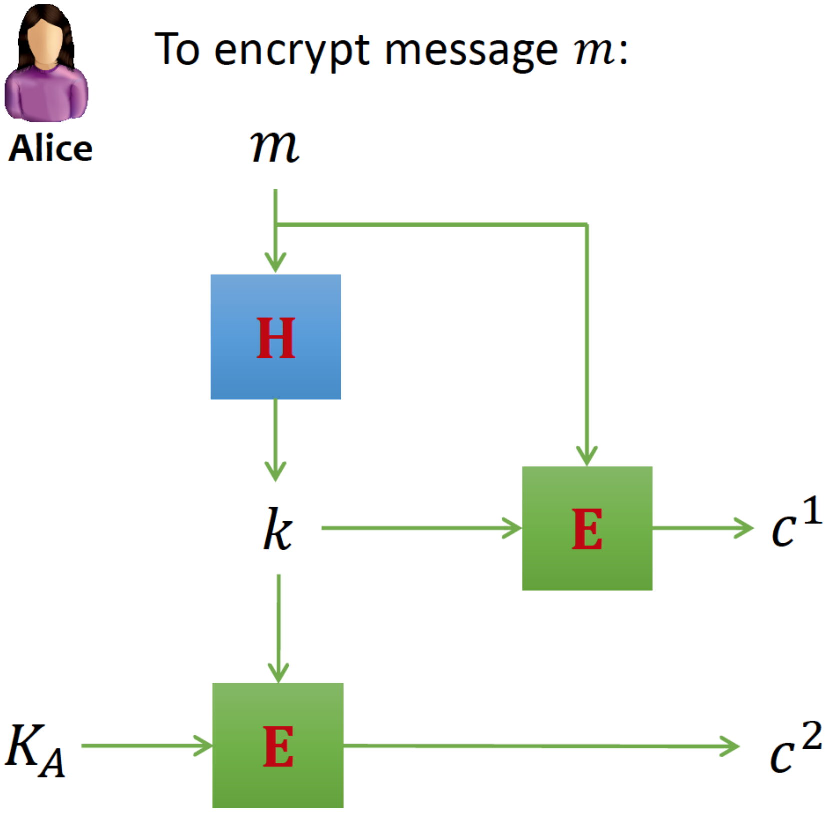


Figure 13: Convergent Encryption Flow Diagram

Decryption:

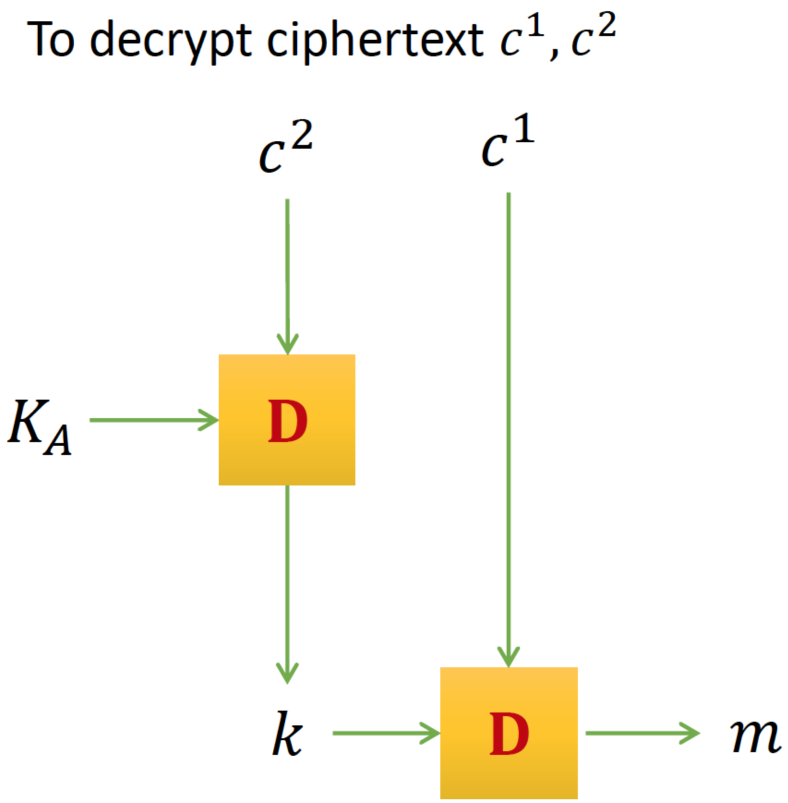


Figure 14: Convergent Decryption Flow Diagram

The advantage of this scheme is Object deduplication works and provides Compromise resilience against untrusted users. The disadvantage of this scheme is it is highly prune to brute force attacks, as messages are often predictable [24].

For example, lets have two users Alice and Bob. Both wants to store file f on the storage system S. Below are the steps done by Alice and Bob in order to encrypt and store file on storage server. Fig. 15 illustrates the computation done by Alice and Bob in case of Convergent Encryption. The number in green denotes the sequence of operations done by users i.e. Alice & Bob.

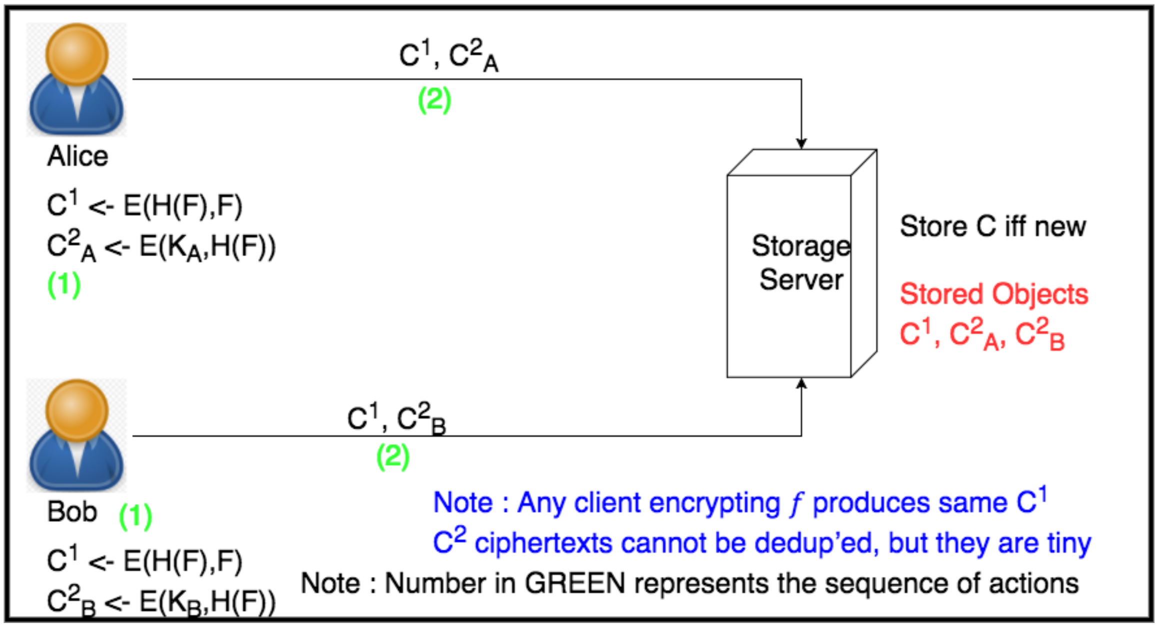


Figure 15: Convergent Encryption Example

Alice

- Computes the cryptographic hash of the file H(f)
- Use the cryptographic hash H(f) as the convergent key to encrypt the file itself and gets the cipher text C_1. (C_1 <- E(H(f),f))
- Alice also encrypts the convergent key using his own key K_A and gets C_2^A. **(C_2^A <- E(K_A,H(f)))**
- Stores C_1 and C_2^A on the storage server.

Bob

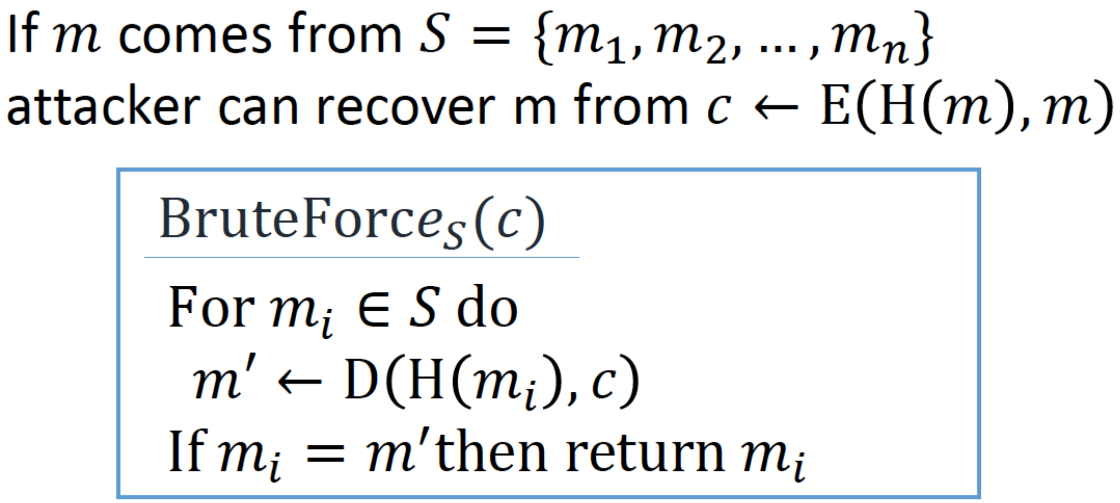
- Computes the cryptographic hash of the file H(f)
- Use the cryptographic hash H(f) as the convergent key to encrypt the file itself and gets the cipher text C_1. (C_1 <- E(H(f),f))
- Alice also encrypts the convergent key using his own key K_B and gets C_2^B. **(C_2^B <- E(K_B,H(f)))**
- Stores C_1 and C_2^B on the storage server.

Hence C_1 is detected as duplicate on the storage server and hence stored only once. Whereas C_2^A and C_2^B are different because they are encrypted using clients specific keys.

Storage server stores C_1, C_2^A and C_2^B. C_2^A and C_2^B are overheads but they are very tiny and hence could be considered.

3.3.3.1 Brute force attack

The dirty secret of convergent encryption is brute force attacks. The idea behind brute force attacks is that data that is being uploaded by the clients on the storage server is often predictable. Hence if the attacker has access to the encrypted data, then using the message predictability, he can apply the brute force attacks and get the deciphered data. Brute force attack algorithm could be written as follows:



If m comes from $S = \{m_1, m_2, \dots, m_n\}$
attacker can recover m from $c \leftarrow \mathrm{E}(\mathrm{H}(m), m)$

```
$\mathrm{BruteForce}_S(c)$
For $m_i \in S$ do
  $m' \leftarrow \mathrm{D}(\mathrm{H}(m_i), c)$
If $m_i = m'$ then return $m_i$
```

Figure 16: Brute force Attacks in Convergent Encryption

It could be seen from the above algorithm (Fig. 16) that the above algorithm runs in time proportional to |S| where |S| is the set of predicted deciphered data. Hence, Convergent encryption strategy works only when |S| is too large to exhaust. But the bottleneck is Real files are often predictable.

3.4 DUPLESS[25]

A secure deduplicated storage architecture resisting brute-force attacks and realizes it in a system referred to as the DUPLESS (Duplicateless Encryption for Simple Storage) [25],[26]. The architecture uses the basic idea of convergent encryption, but instead of fetching the key from the data itself, fetches the key from a key server via an oblivious Pseudo random function (PRF) protocol.

In the DUPLESS architecture, client first interacts with the key server (KS) in order to get the key. It then encrypts the data using this key and gets the ciphered data. Client also encrypts the key using the client's own secret key and hence uploads both the parts on the storage servers. Client transfers the hash of the file to the key server in order to get the key. Key server obliviously transforms the hash into some other hash using a PRF function. The main purpose of Key server is as follows:

1. On receiving the request (Request to get the key) from the client for the first time, it authenticates the client and associates a session id to it. The idea behind associating session id is that it can resist brute-force attacks. That is between two requests, it associates that there should be a minimum epoch duration in order for the request to be fulfilled.
2. Key server then calculates the key from the hash obtained from the client using Pseudo random function and returns the key to the client.

Below is the diagram (Fig. 17) illustrating the DUPLESS architecture and showing the files uploaded by two users Alice and Bob.

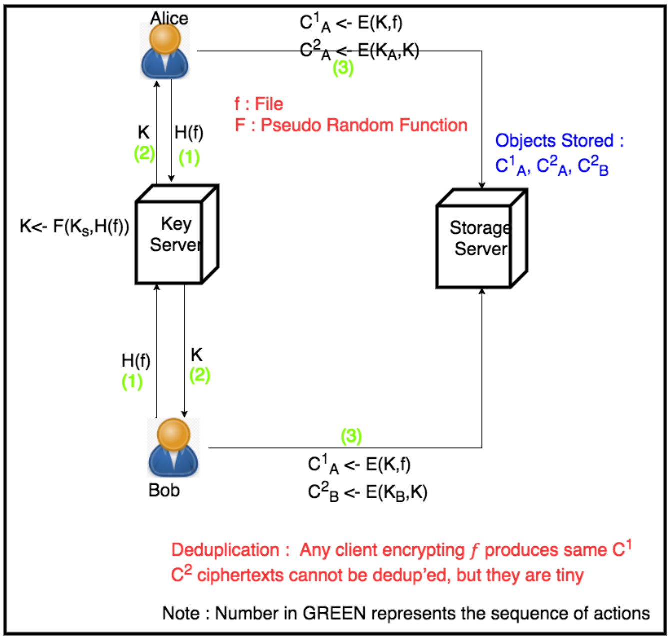

Alice
C^1_A <- E(K,f)
C^2_A <- E(K_A,K)
(3)
f : File
F : Pseudo Random Function
Objects Stored :
C^1_A, C^2_A, C^2_B
K
(2)
H(f)
(1)
K<- F(K_S,H(f))
Key Server
Storage Server
H(f)
(1)
K
(2)
(3)
Bob
C^1_A <- E(K,f)
C^2_B <- E(K_B,K)
Deduplication : Any client encrypting f produces same C^1
C^2 ciphertexts cannot be dedup'ed, but they are tiny
Note : Number in GREEN represents the sequence of actions

Figure 17: DUPLESS Architecture

Alice

- Computes the cryptographic hash of the file H(f)
- Sends the H(f) to the Key server and waits for the response from the Keyserver to get the key (K).
- Use the K as the convergent key to encrypt the file itself and gets the cipher text C_1. (C_1 <- E(K,f))
- Alice also encrypts the convergent key using his own key K_A and gets C_2^A. **(C_2^A <- E(K_A,H(f)))**
- Stores C_1 and C_2^A on the storage server.

Bob

- Computes the cryptographic hash of the file H(f)
- Sends the H(f) to the Key server and waits for the response from the Keyserver to get the key (K).
- Use the cryptographic hash H(f) as the convergent key to encrypt the file itself and gets the cipher text C_1. (C_1 <- E(H(f),f))
- Alice also encrypts the convergent key using his own key K_B and gets C_2^B. **(C_2^B <- E(K_B,H(f)))**
- Stores C_1 and C_2^B on the storage server.

Key Server (KS)

- Gets the request from the client containing H(f).
- If it is the first request, then it authenticates the client and then associates a session identifier to it. Sessions are maintained in order to restrict the brute-force attacks.
- Key server obliviously applies PRF to the hash value and calculates the key.
- Key is returned to the client.

Hence C_1 is detected as duplicate on the storage server and hence stored only once. Whereas C_2^A and C_2^B are different because they are encrypted using clients specific keys. Storage server stores C_1, C_2^A and C_2^B.

3.4.1 Limitations of DUPLESS:

- Lets say there are 100 users trying to upload the same file. Although the encrypted file generated by 100 users is same and detected as duplicate on the storage servers. But the problem is encrypted convergent key for all the 100 users is different and hence 100 keys are stored on the storage servers. The problem becomes much more worse when there are million of users sharing the same file. Lets say user tries to store 1TB of data with all unique blocks of size 4KB each, and that each convergent key is the hash value of SHA-256 which is used by dropbox for deduplication. Then the total size of keys will be 8GB. The number of keys is further multiplied by the number of users and hence key management becomes an issue as this approach does not appears to be scalable [27].
- If the attacker has somehow got an access to the storage server along with the client credentials, then he/she has both the key and the content stored on the storage server and hence could get the deciphered text.

3.5 Comparison

Below is the table mentioning all the features of all the above-mentioned architectures.

Table 1: Table showing comparison of different encryption techniques

	SYSTEMS			
Property	Client Specific Keys	Network Wide Key	Convergent Encryption	DupLESS
Deduplication	**N**	**Y**	**Y**	**Y**
Compromise Resilience	**Y**	**N**	**Y**	**Y**
Brute-force attack resilience	**Y**	**Y**	**N**	**Y**
Fault tolerance	**N**	**N**	**N**	**N**
Keys efficient & Reliable storage	**N**	**Y**	**N**	**N**

4 Proposed Architecture

None of the Architecture that have been discussed in the previous section provides Deduplication along with Compromise resilience, Brute-force attack resilience and ability to store the key and content separately and efficiently such that storage does not scale linearly with the number of users uploading the duplicated data. Client specific keys architecture provides Compromise resilience, Brute-force attack resilience but does not provide deduplication facility on the encrypted data. Network wide key architecture provides deduplication facility and Brute-force attack resilience but does not provide Compromise resilience. Convergent Encryption provides deduplication facility and Compromise resilience but does not provide Brute-force attack resilience. Dupless architecture addressed all the three basic issues i.e. Deduplication along with Compromise resilience, Brute-force attack resilience but has the limitation that the key storage scales linearly with the number of users sharing the duplicated file or block.

Limitations of Dupless:

1. Key storage requirements were linearly dependent on the number of users sharing the duplicated data.
2. Key and content are stored on the single storage server system and hence if there is a high probability that if attacker is able to compromise the storage servers somehow then he/she has both the key along with the encrypted file and hence security is compromised.
3. If the storage server is down, user has no access to any file.

Our proposed architecture addresses all the limitations of the Dupless architecture. Our architecture considers two types of storage servers. One for storing the encrypted data and the other for storing the fragments of the key. By separating the key and content, attacker has to gain access to both types of servers in order to get the decrypted file. Moreover, key is segregated into multiple fragments so that it becomes difficult for the attacker to get some fragments from which key could be constructed. Key is divided into multiple fragments through Erasure Encoding mechanism that has been discussed in previous section.

Dividing the key into multiple fragments and storing each fragment on different server has the following advantages:

1. Attacker has to gain access to a minimum of few key servers (configurable) in order to get the key and also gain access to encrypted file in order to get the decrypted file (decrypting it from the key).
2. Storing different fragments on different servers provides fault tolerant feature to the system. So user is able to access the files even if he/she is able to access k out of n key servers.

4.1 Architectural Diagram

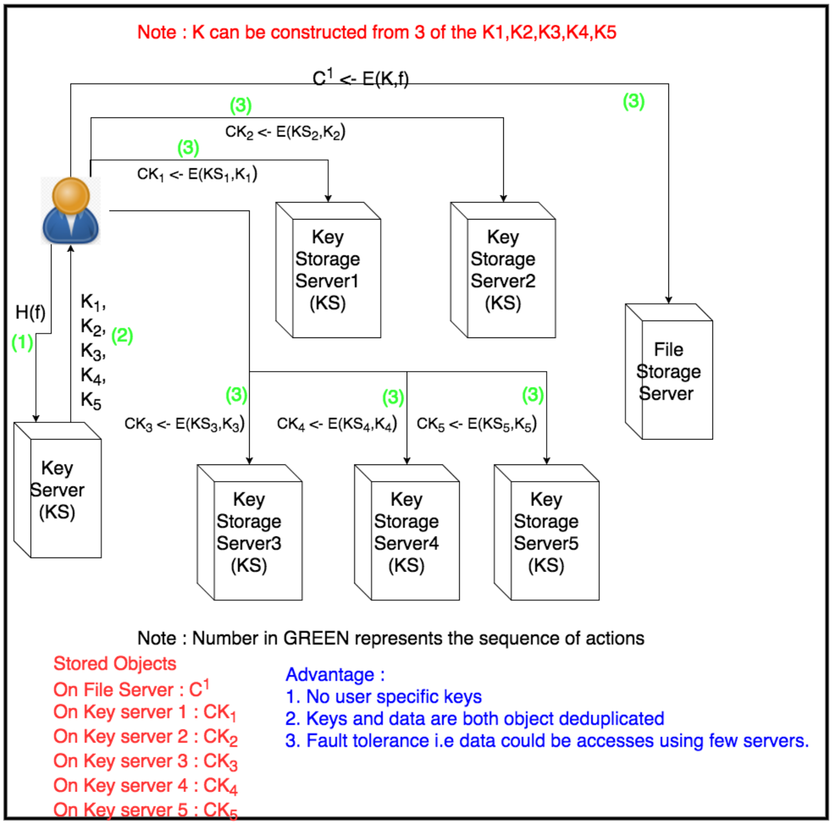


Figure 18: Proposed Architecture Diagram

Fig. 18 shows the architectural diagram of the proposed architecture. The user is trying to upload a file on the storage server and number in the green shows how the file is uploaded on the storage server.

Other than the storage servers, our proposed architecture have a key splitting server (KSS), which serves the purpose of generating the key from the hash value of the file and splitting it into multiple fragments. So a client would send Hash value H to the KSS and receives back multiple fragments of the key. Key is generated at KSS as K' -> F(K,H). After client receives those fragments, it would simply encrypt file with the key(got after combining those fragments) and store the encrypted file on the storage server and stores each fragment on different key storage server. The above approach might prove unsatisfying from a security perspective. The KSS could be figured out as a single point of failure, violating our goal of compromise resilience. So an attacker may obtain hashes of files after gaining access to the KSS, and can recover files with brute force attacks.

To eliminate such attacks, architecture employes an oblivious PRF (OPRF) protocol between the KSS and the client, which ensures that KSS learns nothing about client inputs or fragments that are output from the KSS and the client learns nothing about the key. Other than this, KSS also performs client integrity check by verifying its identity and checking it against some database in order to identify whether client is a legitimate user or not. KSS also performs rate-limiting strategies that limit client queries to slow down online brute-force attacks. Communication between client and KSS is implemented with low latency protocols for the sake of better performance, which is important because the critical path during encryption includes interaction with a KSS. In order for the KSS to handle a reasonably a high request volume, the protocol should be light weight.

Our implementation assumption is that each client has a unique certificate and the KSS verifies this certificate before proceeding for key generation. All communication to the KSS happens over HTTPS. KSS exposes an interface with two procedures. KSInit and KSReq. When the client makes the request to the KSS for the first time, it makes the KSInit request and KSS responds to the client and exchanges the following information such as oblivious PRF public key, hash function identifier (SHA-256), random session identifier, and a random session key. KSS also initiates a sequence number for the client initialized to zero. KSS maintains two mappings, one for mapping session identifier with keys and the other for mapping session identifier with sequence numbers. Each session lasts for a fixed time period (20 mins in our implementation) and mappings are removed after the session expires. Server implementation have three threads running on KSS out of which one thread is doing the task of cleaning the mappings.

To get the key fragments, client first updates the sequence number (N <- N + 1) and then computes a MAC tag using its session key as follows. T <- HMAC[H](K_s ,S || N || X) where X is blinded hash value of the file and sends the concatenation (S || N || X || T) to the KSS in a single UDP packet. The KSS fetches S, N, X and T and looks up KS and NS. It performs some checks such as $N > N_S$ and verifies correctness of the MAC T. If the packet is malformed or if some of the check fails, then the KSS drops the packet without any further action. If all the checks pass, the KSS sends the OPRF protocol response in a single UDP packet.

The client waits for time t_R after sending a KSReq packet before triggering timeout behavior. In our implementation, this involves retrying the same request twice more with time t_R between the tries, incrementing the sequence number each time. After three attempts, the client will try to initiate a new session, again timing out after tR units. If this step fails, the client believes the KS to be offline.

Thus a client to upload a file M on to the storage server will engage in the OPRF protocol to the KSS to compute fragments of the keys (K_1,K_2,K_3,K_4......K_n). Client constructs the key from the fragments (K <- Erasure_Encoding (K_1,K_2,K_3...K_k)) and then encrypts the file using the key. C <- E(K,M) and stores C on the storage server. Should there have been the other client uploading the same file, same cipher text is generated and hence storage server is able to detect it as the duplicate and hence store it only once. Moreover same key fragments are generated for the client trying to upload the same file and hence fragments are also detected as duplicates and hence are stored only once. Hence key fragments have no dependency on the number of users sharing the same file and have a single instance unlike dupless where keys storage scales linearly with the number of users sharing that file/block.

4.2 Implementation Details

I have created this architecture on top of dropbox, which uses deduplication to save storage space. It could be customized to use Google drive as well but the implementation till now is being done on top of dropbox. Any Distributed file system storage can be used as storage structure as long as it is adhering to the following requirements:

It should expose API for:

1. Getting the file contents and its metadata of the directory specified.
2. It should expose API to delete the file from the storage.
3. It should expose API to upload the file on the storage media.
4. It should expose API to download the file from the storage media.

Implementation has been done in python. There are 6 dropbox accounts used in order to store the contents and the keys. One of the dropbox accounts is used to store the contents of the files and the other 5 five dropbox accounts are used to store the fragments of the keys. Key Splitting server is also implemented in python.

Uploading flow:

1. User creates the hash of the file and sends the hash to the Key splitting server.
2. Key splitting server first authenticates the client and then obliviously creates the key by applying Pseudo random function (PRF) to the hash of the file sent by the client.
3. Key splitting server then uses this key and pass this to the Erasure Encoding module, which creates the 5 fragments of the key.
4. All the five fragments are sent to the client.
5. Client constructs the key back using any 3 fragments and encrypts the file using that key. The key is referred to as the convergent key.
6. Client sends the encrypted file to one of the designated dropbox account and sends each fragment to each one of the dropbox accounts.

7. User gets a confirmation that file has been uploaded to the server.

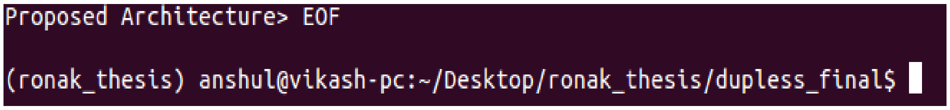
Downloading flow:

1. User requests for a file. File name is searched on the 3 key servers (dropbox accounts) that are randomly selected. If any of the dropbox account is inaccessible due to some issues, other two are tried in order to get 3 fragments from the key servers.
2. Client also fetches the encrypted file from the storage server via other thread.
3. Client then invokes the decoder of the Erasure encoding module to get the key from the fragments.
4. Client then decrypts the file from the key obtained.

4.3 Client Implementation

A command line tool has been designed in order to deal with the files to be uploaded on the dropbox storage server. A number of operations are available to the client to deal with the files such as

1. EOF command : to exit the command prompt. Fig. 19 shows the screenshot.

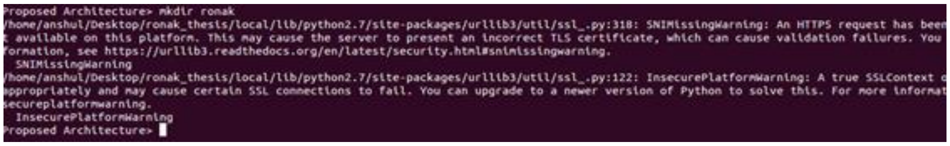


```
Proposed Architecture> EOF

(ronak_thesis) anshul@vikash-pc:~/Desktop/ronak_thesis/dupless_final$ 
```

Figure 19: EOF command screenshot

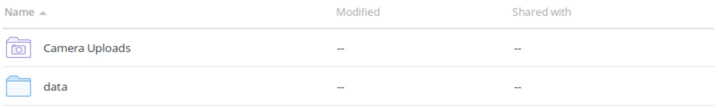
2. mdkir command : to create the directory on the dropbox into the current working directory. Fig. 20, Fig. 21, Fig. 22 shows the screenshot.



```
Proposed Architecture> mkdir ronak
/home/anshul/Desktop/ronak_thesis/local/lib/python2.7/site-packages/urllib3/util/ssl_.py:318: SNIMissingWarning: An HTTPS request has been
t available on this platform. This may cause the server to present an incorrect TLS certificate, which can cause validation failures. You
formation, see https://urllib3.readthedocs.org/en/latest/security.html#snimissingwarning.
  SNIMissingWarning
/home/anshul/Desktop/ronak_thesis/local/lib/python2.7/site-packages/urllib3/util/ssl_.py:122: InsecurePlatformWarning: A true SSLContext o
appropriately and may cause certain SSL connections to fail. You can upgrade to a newer version of Python to solve this. For more informat
secureplatformwarning.
  InsecurePlatformWarning
Proposed Architecture> 
```

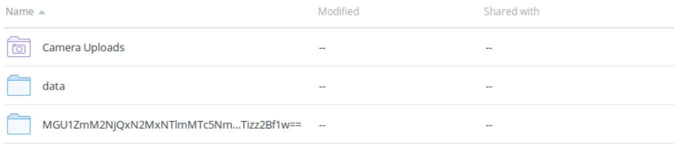
Figure 20: mkdir command screenshot

User can check into their dropbox account that this creates the folder into current working directory. I have attached the screenshot of the dropbox account before (Fig. 21) the command was run and after (Fig. 22) the command was run. Note that the folder name will be encoded and hence will not be shown as ronak into the dropbox account.



Name ▲	Modified	Shared with
Camera Uploads	--	--
data	--	--

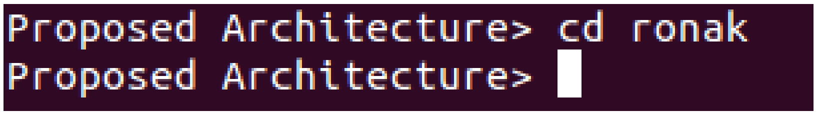
Figure 21: Dropbox screenshot before mkdir



Name ▲	Modified	Shared with
Camera Uploads	--	--
data	--	--
MGU1ZmM2NjQxN2MxNTlmMTc5Nm...Tizz2Bf1w==	--	--

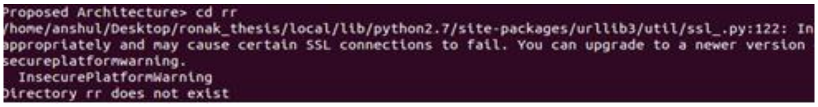
Figure 22: Dropbox screenshot after mkdir

3. cd <relative directory> command: to change the current working directory. cd command fails when you wish to change the current working directory to a directory, which actually do not exist. cd command only accepts the relative folder paths. Fig. 23 & Fig. 24 shows the screenshot.



```
Proposed Architecture> cd ronak
Proposed Architecture> █
```

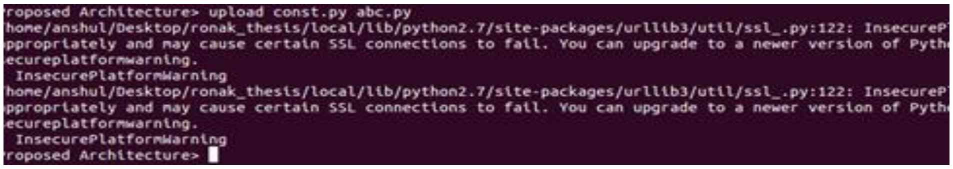
Figure 23: cd command success when directory exists



```
Proposed Architecture> cd rr
/home/anshul/Desktop/ronak_thesis/local/lib/python2.7/site-packages/urllib3/util/ssl_.py:122: In
appropriately and may cause certain SSL connections to fail. You can upgrade to a newer version
secureplatformwarning.
  InsecurePlatformWarning
Directory rr does not exist
```

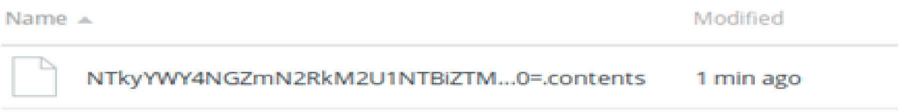
Figure 24: cd command failed when directory does not exist on the Storage servers(dropbox)

4. upload <source file name> <remote file name> command : to upload the file on the storage servers. After the successful completion run of this command, you can check the dropbox accounts to contain the content file in the current working directory. You can also check the key server dropbox accounts to contain the fragments of the key file. I have attached the screenshots of main dropbox account (Fig. 26) along with screenshot of one of the key storage server dropbox account (Fig. 27). Fig. 25 shows the command prompt screenshot.



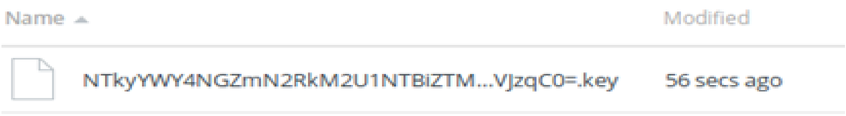
```
roposed Architecture> upload const.py abc.py
home/anshul/Desktop/ronak_thesis/local/lib/python2.7/site-packages/urllib3/util/ssl_.py:122: InsecureP
ppropriately and may cause certain SSL connections to fail. You can upgrade to a newer version of Pyth
ecureplatformwarning.
 InsecurePlatformWarning
home/anshul/Desktop/ronak_thesis/local/lib/python2.7/site-packages/urllib3/util/ssl_.py:122: InsecureP
ppropriately and may cause certain SSL connections to fail. You can upgrade to a newer version of Pyth
ecureplatformwarning.
 InsecurePlatformWarning
roposed Architecture> █
```

Figure 25: upload command screenshot



Name ▲	Modified
NTkyYWY4NGZmN2RkM2U1NTBiZTM...0=.contents	1 min ago

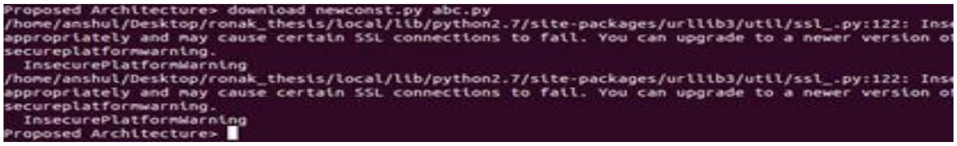
Figure 26: Storage server showing the uploaded content file



Name ▲	Modified
NTkyYWY4NGZmN2RkM2U1NTBiZTM...VJzqC0=.key	56 secs ago

Figure 27: Key storage server showing the fragment of the key file

5. download <source file name> <remote file name> command : to download the file from the current working directory. After the download you will find the decrypted file downloaded on you machine with the name given in the <source file name> attribute of the command. Fig. 28 shows the command prompt screenshot. Fig. 29 shows the folder screenshot with the downloaded file being highlighted.



```
Proposed Architecture> download newconst.py abc.py
/home/anshul/Desktop/ronak_thesis/local/lib/python2.7/site-packages/urllib3/util/ssl_.py:122: Ins
appropriately and may cause certain SSL connections to fail. You can upgrade to a newer version o
secureplatformwarning.
  InsecurePlatformWarning
/home/anshul/Desktop/ronak_thesis/local/lib/python2.7/site-packages/urllib3/util/ssl_.py:122: Ins
appropriately and may cause certain SSL connections to fail. You can upgrade to a newer version o
secureplatformwarning.
  InsecurePlatformWarning
Proposed Architecture> █
```

Figure 28: download command screenshot

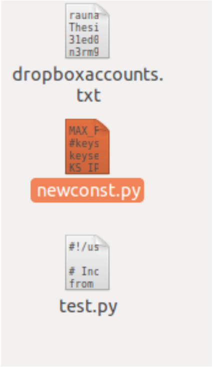


Figure 29: File downloaded in the directory after download command successfully executed

Note: You can press the tab key in order to see the list of available commands.

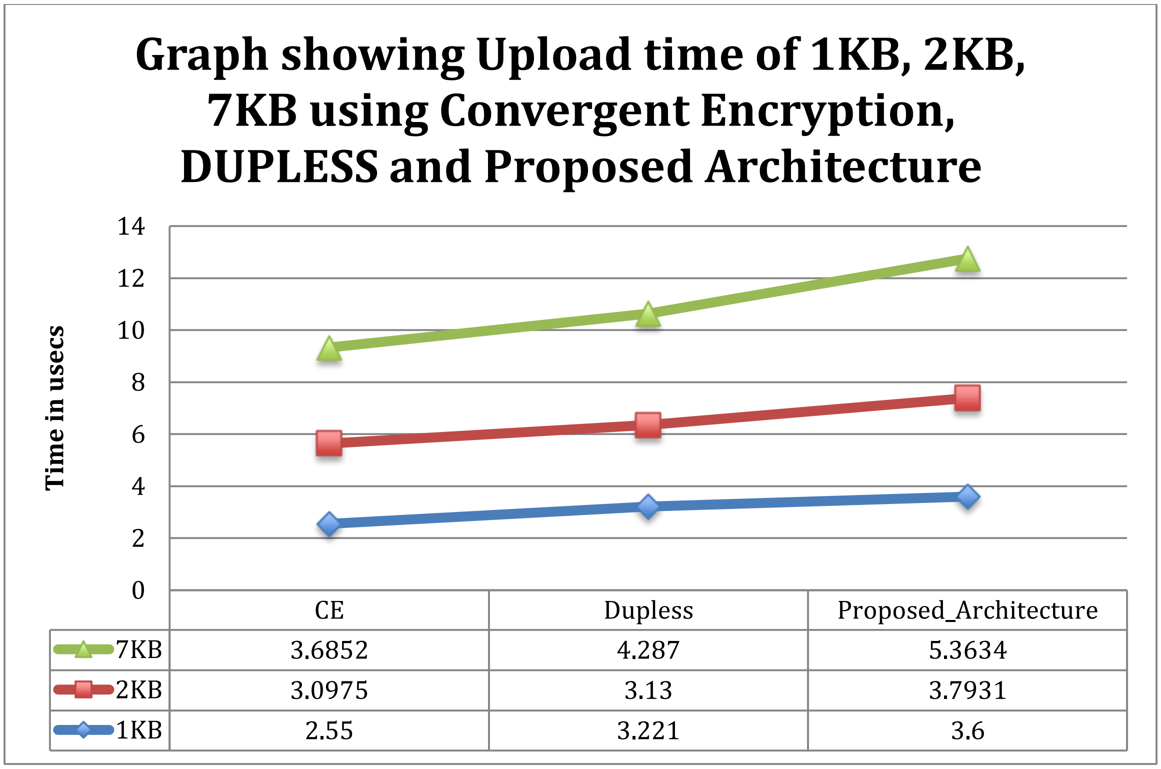
A client must have two types of credentials. One for the dropbox accounts and the other for the authentication to the Key Splitting server. KSS credentials can be put into SQL database uploaded on the KSS. A client must generate an APP on the dropbox and replace the APP_KEY and APP_SECRET in the const.py file. All the configuration details are mentioned in the const.py file.

Client must also generate self-signed certificate in order to be authenticated to the key splitting server. The client can generate the self-signed certificate using the openssl [28] module. Our implementation uses AES as the encryption scheme. Any Symmetric key based algorithm can be used to encrypt the files.

4.4 Results

The Key Splitting Server runs on the same machine as the client and the machine has the configuration : 2.3 GHz Intel Core i7 with 16GB RAM. The entire architecture has been tested on Ubuntu 14.04 and the code is uploaded on the gitlab. README contains all the steps in order to set up the environment.

We have taken three type of sample file size of 1KB, 2KB, 7KB. We have copied these files into separate folders and replicated it one more time. Hence each folder has 2 same files. We have taken out the results using the python timeit module to see the time taken by the scripts to upload the file on the server. We have compared the results with the convergent encryption and Dupless architecture. The script basically tries to upload all the same files on to the dropbox. The first file is basically uploaded and the next file is detected as duplicate. Although detection of duplicates is completely transparent to the client, but the dropbox documentation says that it performs the deduplication. Hence we have assumed that dropbox saved only one instance of the file.



	CE	Dupless	Proposed_Architecture
▲ 7KB	3.6852	4.287	5.3634
■ 2KB	3.0975	3.13	3.7931
◆ 1KB	2.55	3.221	3.6

Figure 30: Graph showing upload time of files using various architecture

Fig. 30 shows the upload time of various sizes of files on the storage server in the encrypted form using different architectures. From the table in the Fig. 30, it is clear that proposed architecture takes a bit long as it has to perform the erasure encoding to divide the key into fragments and upload these fragments on the key storage server. We have performed the experiments around 100 times and see that the proposed architecture takes 20% – 35% more time in comparison to Convergent Encryption to upload the file. But with this overhead involved, the system is fault tolerant enough to service the clients even if some of the key storage servers are down. Moreover, the system is resilient enough against the insider attacks where unauthorized users can gain access to the key and the data as now the data and key are separately stored.

Table 2: Table showing the number of bytes taken on the storage against the number of clients sharing the file/block for the file upload of 390 bytes

Size of Upload = 390 Bytes		
Number of Shares	**Convergent Encryption (Bytes)**	**Proposed_Architecture (Bytes)**
1	438	480
2	486	480
3	534	480
4	582	480
100	5190	480
1000	48390	480

Table 3: Table showing the number of bytes taken on the storage against the number of clients sharing the file/block for the file upload of 6482 bytes

Size of Upload = 6482 Bytes		
Number of Shares	**Convergent Encryption (Bytes)**	**Proposed_Architecture (Bytes)**
1	6529.92	6572
2	6577.92	6572
3	6625.92	6572
4	6673.92	6572
100	11281.92	6572
1000	54481.92	6572

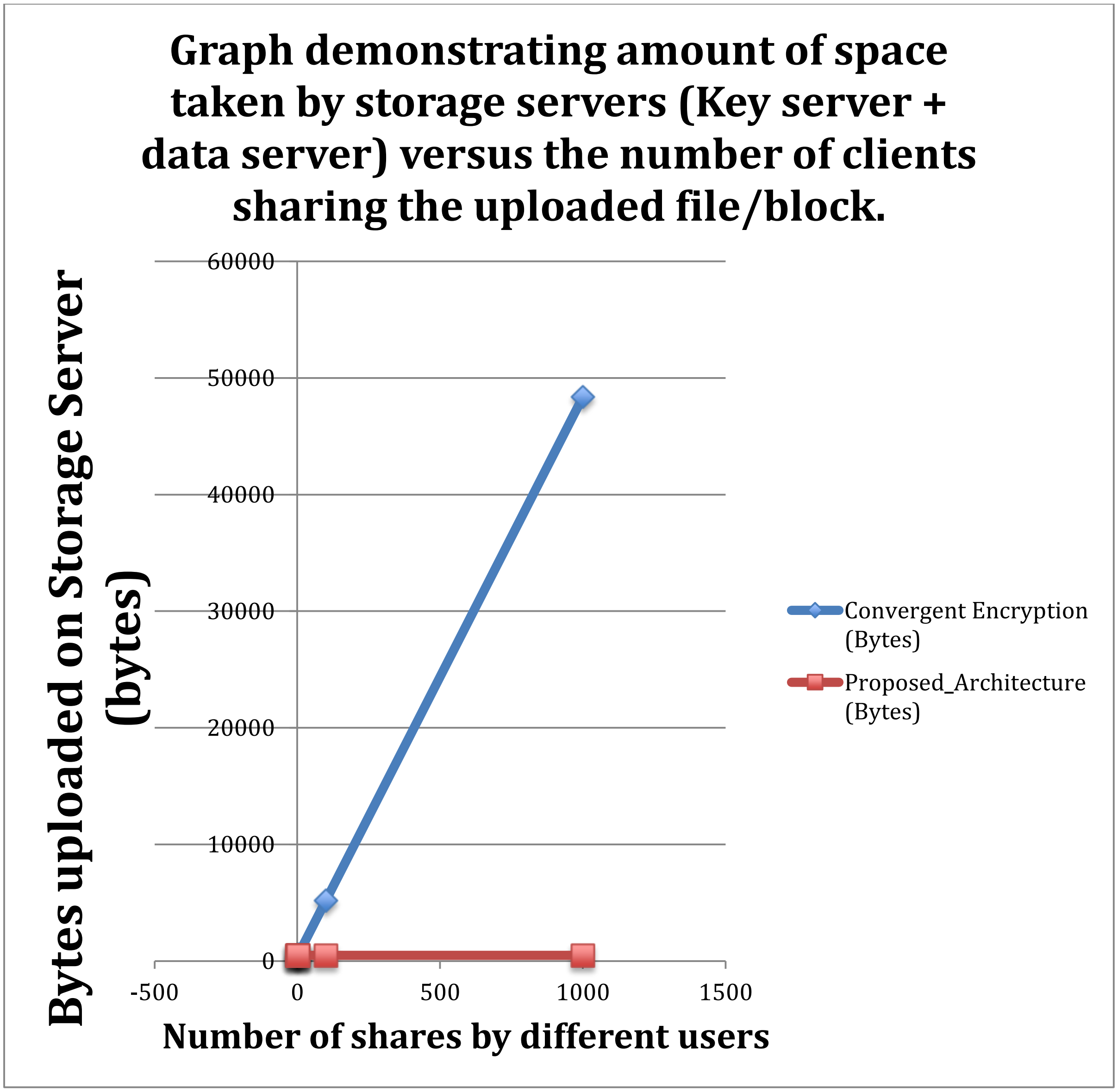


Figure 31: Graph representing the relationship between amount of space used vs the number of clients using the file/block for different architecture

Fig. 31 shows that in the proposed architecture, the storage size is not dependent on the number of clients sharing that storage whereas in the convergent encryption, it is linearly dependent on the number of clients sharing that file.

The current implementation of proposed architecture considers dividing the key into five fragments. Fault tolerance of the system with respect to the key servers is measured by calculating the probability of availability of at least three servers in order to construct the original key given the probability of failing of a given server. The current implementation considers each server to be same and have the same probability of failing.

Table 4: Table showing the Availability of the convergent key with respect to the probability of failing of a key server

Probability of server being up	Probability of server being down	Availability
0.1	0.9	0.00856
0.2	0.8	0.05792
0.3	0.7	0.16308
0.4	0.6	0.31744
0.5	0.5	0.5
0.6	0.4	0.68256
0.7	0.3	0.83692
0.8	0.2	0.94208
0.9	0.1	0.99144

The Availability of the convergent key is calculated as:

$$\sum_{k=3\ to\ 5} \binom{5}{k} p^k (1-p)^{5-k} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (1)$$

Fig. 32 shows the graph between the Availability and the the probability of failing of each server.

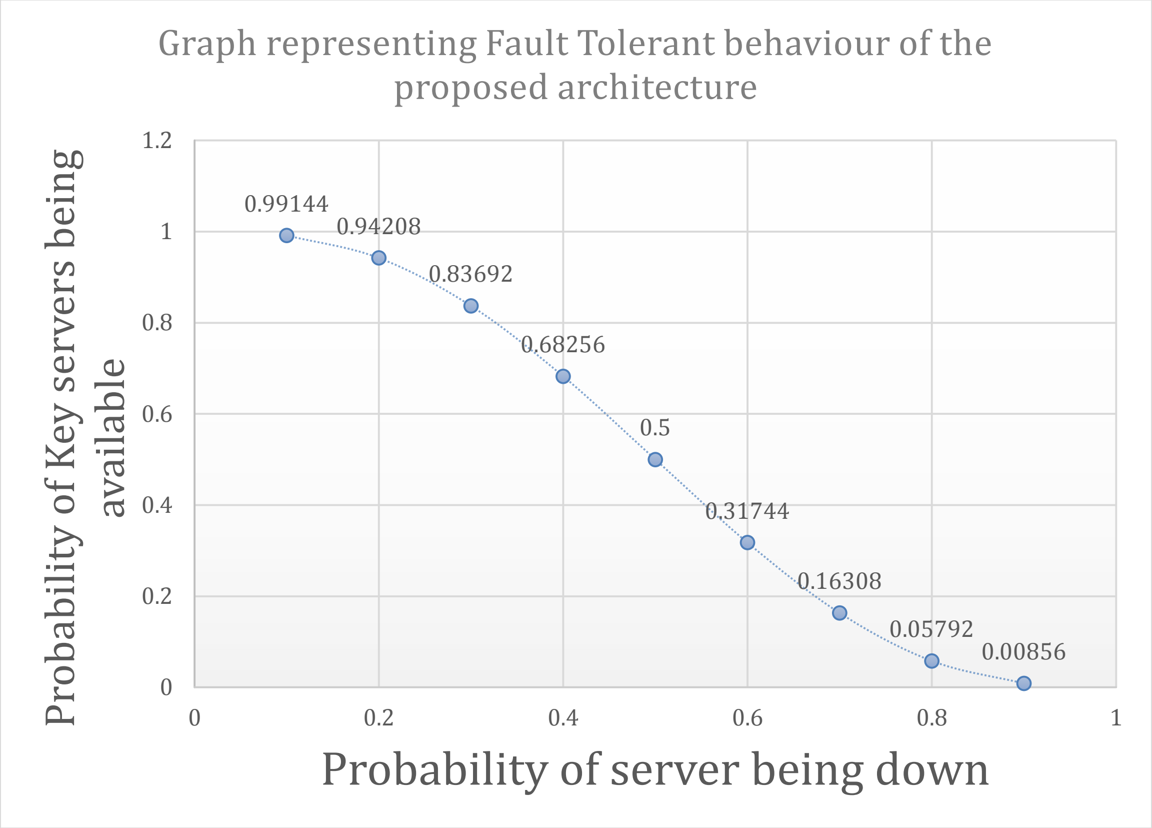


Figure 32: Graph representing Fault Tolerance behavior of the proposed architecture

5 CONCLUSION & FUTURE-WORK

5.1 CONCLUSION

The proposed architecture provides efficient fault tolerant system to the end user by segregating the key and the content stored on the storage servers. Moreover, the proposed architecture also provides key storage in such a way such that it is independent of the number of shares from different users. Hence a lot of space is saved in case when number of users sharing a file are very large. The architecture also provides resilience against the attacks in which attackers can access any one server to get the cipher text. As the key and content are stored separately, attacker needs to access four servers in order to get the try the possibility of deciphering the ciphered content.

5.2 FUTURE WORK

Our implementation relies on the fact that dropbox does the process of deduplication. Even the results that we mentioned in the results section are in accordance with the deduplication. Storage results could not be retrieved, as dropbox deduplication is transparent to the clients. Hence clients are actually not aware of the fact whether the data that they uploaded on the storage server is actually deduplicated or not.

Setting up own storage distributed file system and enabling deduplication on it could give better results. Even multiple types of data in various formats could be tried and see what block sizes fits best for which type of data. Currently we have used dropbox accounts to separate key and content but ideally our requirement was to segregate both of them on different storage file system. CEPH[20], an open source distributed file system can really be a good point to consider as it supports all types of storage. Moreover a variety of support is available in case of any issues in deployment.

Different erasure coding techniques can be tried in order to reduce the time taken by the key splitting server to split the key into multiple fragments. Some of the erasure encoding available are:

- Vandermonde Reed-Solomon encoding
- Cauchy Reed-Solomon encoding

- Flat-XOR based HD combination codes
- Intel Storage Acceleration Library (ISA-L) - SIMD accelerated Erasure Coding
- NTT Lab Japan’s Erasure Coding

6 Publications

“Fault tolerant optimized convergent key management system in Object Storage Based Cloud Environment” abstract has been accepted in the IEEE Cloud Computing for Emerging Markets (CCEM) 2016 conference to be held on 19 – 21 October 2016 in Bangalore, India.

7 References

[1] J. Gantz and D. Reinsel, The Digital Universe in 2020: Big Data, Bigger Digital Shadows, Biggest Growth in the Far East, Dec. 2012. [Online]. Available: http://www.emc.com/collateral/analystreports/idc-the-digital-universe-in-2020.pdf

[2] Vendor Spotlight Template (The Digital Universe Decade)
https://www.emc.com/collateral/analyst-reports/idc-digital-universe-are-you-ready.pdf

[3] D. Harnik, B. Pinkas, and A. Shulman-Peleg, ‘‘Side Channels in Cloud Services: Deduplication in Cloud Storage,’’ IEEE Security Privacy, vol. 8, no. 6, pp. 40-47, Nov./Dec. 2010.

[4] Dropbox, a file-storage and sharing service. http://www.dropbox.com/.

[5] Google Drive. http://drive.google.com.

[6] MOZY. Mozy, a file-storage and sharing service. http://mozy.com/.

[7] Bhagwat, D., K. Eshghi, D. D. Long, & M. Lillibridge (2009, September). Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09, pp. 1-9. IEEE.

[8] You, L. L., K. T. Pollack, & D. D. E. Long (2005). Deep Store: An Archival Storage System Architecture. In ICDE '05: Proceedings of the 21st International Conference on Data Engineering, Washington, DC, USA, pp. 804-8015. IEEE Computer Society.

[9] Cox, O. P., C. D. Murray, & B. D. Noble (2002). Pastiche: making backup cheap and easy. In In OSDI: Symposium on Operating Systems Design and Implementation, pp. 285-298.

[10] J.R. Douceur, A. Adya, W.J. Bolosky, D. Simon, and M. Theimer,‘‘Reclaiming Space from Duplicate Files in a Serverless Distributed File System,’’ in Proc. ICDCS, 2002, pp. 617-624.

[11] N. Kaaniche and M. Laurent, "A Secure Client Side Deduplication Scheme in Cloud Storage Environments," *2014 6th International Conference on New Technologies, Mobility and Security (NTMS)*, Dubai, 2014, pp. 1-7.

[12] Suresh, L. Padma, and Bijaya Ketan Panigrahi. *Proceedings of the International Conference on Soft Computing Systems*. Springer, India, 2015.

[13] StoneFly Z-Series DR Backup Appliance - StoneFly's iSCSI.com
http://www.stonefly.com/products/backup/drzseries/

[14] Sean Quinlan, Sean Dorward, Venti: A New Approach to Archival Storage, Proceedings of the Conference on File and Storage Technologies, p.89-101, January 28-30, 2002

[15] Rabin Karp for Variable Chunking · YADL/yadl Wiki https://github.com/YADL/yadl/wiki/Rabin-Karp-for-Variable-Chunking

[16] Deduplication Internals - Source Side & Target Side Deduplication: Part-4 | pibytes https://pibytes.wordpress.com/2013/03/09/deduplication-internals-source-side-target-side-deduplication-part-4/

[17] Post-process deduplication vs In-line deduplication https://www.starwindsoftware.com/post-process-deduplication-vs-in-line-deduplication

[18] Zhu, B., K. Li, & H. Patterson (2008). Avoiding the disk bottleneck in the data domain deduplication file system. In Proceedings of the 6th USENIX Conference on File and Storage Technologies, FAST'08, Berkeley, CA, USA, pp. 18:1-18:14. USENIX Association.

[19] Lillibridge, M., K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, & P. Camble (2009). Sparse indexing: large scale, inline deduplication using sampling and locality. In Proccedings of the 7th conference on File and storage technologies, Berkeley, CA, USA, pp. 111-123. USENIX Association.

[20] Weil, S. A., S. A. Brandt, E. L. Miller, D. D. E. Long, & C. Maltzahn (2006). Ceph: a scalable, high-performance distributed file system. In Proceedings of the 7th symposium on Operating systems design and implementation, OSDI '06, Berkeley, CA, USA, pp. 307-320. USENIX Association.

[21] Weil, S. A., S. A. Brandt, E. L. Miller, & C. Maltzahn (2006). Crush: controlled, scalable, decentralized placement of replicated data. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC '06, New York, NY, USA. ACM.

[22] Encryption Wikipedia https://en.wikipedia.org/wiki/Encryption

[23] Encryption 101: What It Is, How It Works, and Why We Need It - Security News - Trend Micro USA http://www.trendmicro.com/vinfo/us/security/news/online-privacy/encryption-101-what-it-is-how-it-works

[24] GOLDWASSER, S., AND MICALI, S. Probabilistic encryption. Journal of Computer and System Sciences 28, 2 (1984), 270–299.

[25] M. Bellare, S. Keelveedhi, and T. Ristenpart. Dupless: Server aided encryption for deduplicated storage. In USENIX Security Symposium, 2013.

[26] DupLESS: Server-Aided Encryption for Deduplicated Storage https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/bellare

[27] Jin Li, Xiaofeng Chen, Mingqiang Li, Jingwei Li, Patrick P.C. Lee, and Wenjing Lou "Secure Deduplication with efficient and Reliable Convergent Key Management" June 2014

[28] OpenSSL Project. [Online]. Available: http://www.openssl.org/

[29] Policroniades, Calicrates, and Ian Pratt. "Alternatives for Detecting Redundancy in Storage Systems Data." *USENIX Annual Technical Conference, General Track*. 2004.