Cloud Storage over Multiple Data Centers

Shuai MU, Maomeng SU, Pin GAO, Yongwei WU, Keqin LI, Albert ZOMAYA

0 Abstract

The increasing popularity of cloud storage services has led many companies to migrate their data into clouds for high availability whilst saving infrastructure constructions. Within the last few years, the architectures of most major cloud storage providers have evolved from single data center to multiple data centers, across different geo-locations. This architecture help improve the availability of consumers' data, as well as safety guarantee against disasters that can disable a whole datacenter. In this chapter we introduce the architecture evolution of typical cloud storage, and common techniques used in the design, such as replication strategies, data striping methods and consistency models. We also introduce a trend in the practice of cloud storage—cloud of clouds.

1 Introduction

Cloud storage has become a booming trend in the last few years. Individual developers, companies, organizations, and even governments have either taken steps or at least shown great interests in data migration from self-maintained infrastructure into cloud.

Cloud storage benefit consumers in many ways. A recent survey among over 600 cloud consumers has shown that primary reasons for most clients in turning to cloud are (1) to have highly reliable as well as available data storage services; (2) to reduce the capital cost of constructing their own datacenter and then maintaining it; and (3) to provide high-quality and stable network connectivity to their customers. The prosperity of the cloud market has also inspired many companies to provide cloud storage services of different quality to vast variety of companies.

Reliability and availability are the most important issues in designing a cloud storage system. In cloud storage, data reliability often refers to that data is not lost, and availability refers to data accessibility. To major cloud storage providers, accidents of data loss seldom happen. But there was accident that Microsoft once completely lost user data of T-Mobile cellphone users. Because data loss accidents seldom happen, data availability is often a more important concern to most cloud storage consumers. Almost all cloud storage providers have suffered from temporary failures, lasting

from hours to days. For example, Amazon failed to provide service in October, 2012, which caused many consumers such as Instgram to stop service.

Data replication is an effective way to improve reliability and availability. Limited by cost, cloud providers usually use commodity hardware to store consumers' data. Replicating data into different machines can tolerate hardware failures, and replicating data into multiple data centers can tolerate failures of a datacenter, caused by earthquakes, storms and even wars.

To reduce cost of replication, data are usually divided into stripes instead of the original copy. There are many coding methods for this, originating from traditional storage research and practice. Erasure coding, which is widely used in the RAID systems, provides suitable features for data striping requirements in cloud storage environment.

Data consistency is also an important issue in cloud storage. Different from traditional storage systems which usually provide a strong consistency model, cloud storage often offer weaker consistency model such as eventual consistency. Some also propose other reduced consistency models such as session consistency, and fork-join consistency.

Privacy and security are very essential to some consumers. Consumers are often concerned about how their data are visible to cloud providers, whether administrators can see their data transparently. For other consumers who are less sensitive to privacy, they are more concerned in access control to data, because all the traffic finally leads to charges.

In spite of all efforts of cloud storage providers, there comes up a new trend to build up an integration layer on top of current cloud storages, also named "cloud-of-clouds". A cloud-of-cloud system makes use of current cloud storage infrastructure, but still provides a uniformed user interface to top-level application developers. It also targets on the reliability, availability and security issues, but takes a different approach of using each cloud as building blocks. The advantage of this approach is that it can tolerate performance fluctuation of single cloud storage, and can avoid potential risks of a provider shutdown.

In the remainder of this chapter, we first look at cloud storage architecture at a high level in Section 2. Section 3 describes common strategies used in data replication. Section 4 gives a brief introduction on data striping. Section 5 introduces the consistency issue. Section 6 briefly highlights a new model of cloud-of-clouds. Section 7 discusses about privacy and security issues in storage systems. Section 8 summarizes and suggests future direction to cloud storage research and practice.

2 Cloud Storage in a Nutshell

In this section we give an overview of cloud storage architecture and its key components. Cloud storage are usually complex systems mixed with many mature and pioneer techniques. On a high level, cloud storage design and implementation consist of two parts: metadata and data. We will take look at the two parts in later sections.

2.1 Architecture

Early public cloud and most nowadays private cloud, are built in one datacenter, or several datacenters in nearby buildings. They are composed of hundreds or even thousands of commodity machines and storage devices, connected by high-speed networks. Besides large amounts of hardware, many other storage middleware such as distributed file systems are also necessity to provide storage service to consumers. Typical architecture of a cloud storage usually includes storage devices, distributed file system, metadata service, frontend, etc.

In practice, we find that the data models and library interfaces of different clouds are fairly similar; thus, we could support a minimal set to satisfy most users' needs. The data model shared by most services could be summarized as a "container-object" model, in which file objects are put into containers. Most services containers do not support nesting; i.e., users cannot create a sub-container in a container.

In the last decade, major cloud storage such as Amazon S3[1] and Windows Azure Storage[2] have upgrade their service from running in separate data centers to different data centers and different geographic regions. Compare to single datacenter structure, running service in multiple data centers require more resource management functions such as resource allocation, deployment, migration, etc.

An important feature of cloud storage is the ability to store and provide access to an immense amount of storage. Amazon S3 currently has a few hundred petabytes of raw storage in production, and it also has a few hundred more petabytes of raw storage based on customer demand. A modern cloud storage architecture could be divided into three layers: storage service, metadata service, and front-end layer. (see Figure 1)

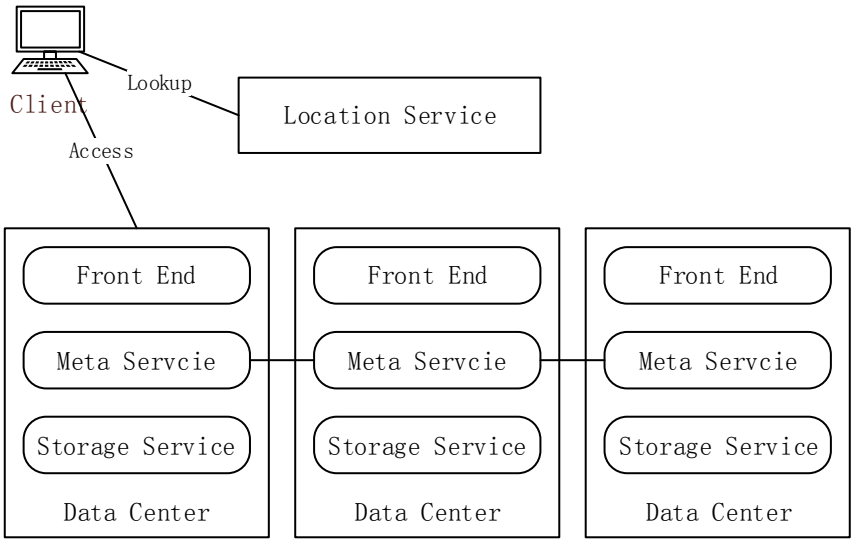

Figure 1 Cloud storage architecutre over mutiple data centers

- Metadata service – The metadata service is in charge of following functions: (a) handling high level interfaces and data structures; (b) managing a scalable namespace for consumers' objects; (c) storing object data into the storage service. Metadata service holds the responsibility to achieve scalability by partitioning all of the data objects within a datacenter. This layer consists of many metadata servers, each of which serves for a range of different objects. Also, it should provide load balance among all the metadata servers to meet the traffic of requests.

- Storage service – This storage service is in charge of storing the actual data into disks and distributing and replicating the data across many servers to keep data reliable within a datacenter. The storage service can be thought of as a distributed file system. It holds files, which are stored as large storage chunks. It also understands how to store them, how to replicate them, etc., but it does not understand higher level semantics of objects. The data is stored in the storage service, but it is accessed from the metadata service.

- Front-End (FE) layer – The front-end layer consists of a set of stateless servers that take incoming requests. Upon receiving a request, an FE looks up the account, authenticates and authorizes the request, then routes the request to a partition server in the metadata service. The system maintains a map that keeps track of the partition ranges and which metadata server is serving which partition. The FE servers cache the map and use it to determine which metadata server to forward each request to. The FE servers also file large objects directly from the storage service and cache frequently accessed data for efficiency.

2.2 Metadata Service

The metadata service contains three main architectural components, a layout manager, many meta servers, and a reliable lock service (see Figure 2). The architecture is similar to Bigtable[5].

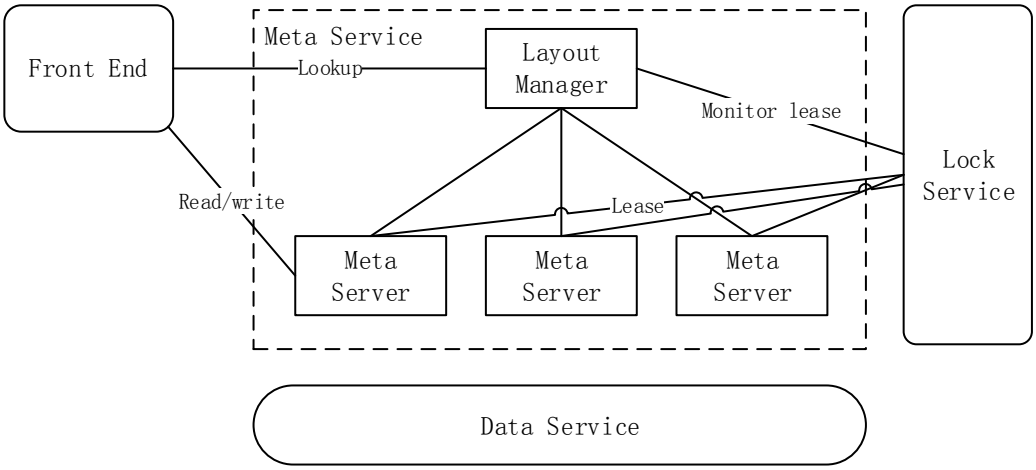

Figure 2 Meta Layer Architecture

2.2.1 Layout Manager

A layout manager (LM) acts as a leader of the meta service. It is responsible for dividing the whole metadata into ranges and assigning each meta server to serve several ranges and then keeping track of the information. The LM stores this assignment in a local map. The LM must ensure that each range is assigned only to one active meta server, and that two ranges do not overlap. It is also in charge of load balancing ranges among meta servers. Each datacenter may have multiple instances of the LM running, but usually they function as reliable replications of each other. For this they need a Lock Service to maintain a lease for leader election.

2.2.2 Meta Server

A meta server (MS) is responsible for organizing and storing a certain set of ranges of metadata, which is assigned by LM. It also serves requests to those ranges. The MS stores all metadata into files persistently on disks and maintains a memory cache for efficiency. Meta servers keep leases with the Lock Service, so that it is guaranteed that no two meta servers can serve the same range at the same time.

If a MS fails, LM will assign a new MS to serve all ranges served by the failed MS. Based on the load, LM may choose a few MS rather than one to serve the ranges. LM firstly assigns a range to a MS, and then updates its local map which specifies which MS is serving each range. When a MS gets a new assignment from LM, it firstly acquires for the lease from Lock Service, and then starts serving the new range.

2.2.3 Lock Service

Lock Service (LS) is used by both of layout manager and meta server. LS uses Paxos[16] protocol to do synchronous replication among several nodes to provide a reliable lock service. LM use LS for leader election; MS also maintains a lease with the LS to keep alive. Details of the LM leader election and the MS lease management are discussed here. We also do not go into the details of Paxos protocol. The architecture of lock service is similar to Chubby[18].

2.3 Storage Service

The two main architecture components of the storage service are the namenode and chunk server (see Figure 3). The storage service architecture is similar to GFS[4].

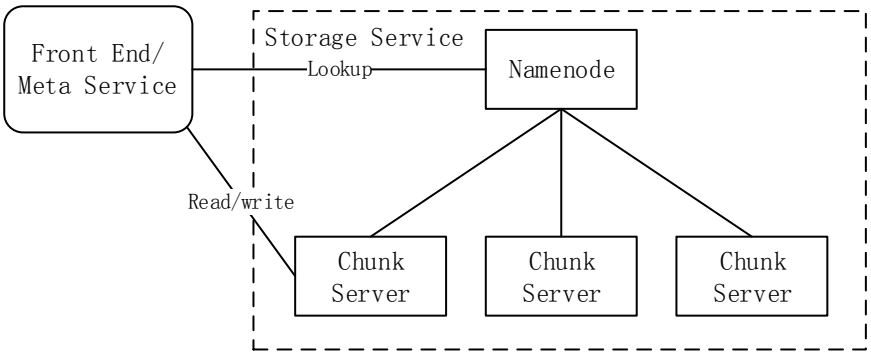

Figure 3 Storage service architecture

2.3.1 Namenode

The namenode can be considered as the leader of the storage service. It maintains file namespace, relationships between chunks and each file, and the chunk locations across the chunk servers. The namenode is off the critical path of client read and write requests. In addition, the namenode also monitors the health of the chunk servers periodically. Other functions of namenode include: lazy re-replication of chunks, garbage collection, and erasure code scheduling.

The namenode periodically checks the state of each chunk server. If the namenode finds that the replication number of a chunk is smaller than configuration, it will start a re-replication of the chunk. To achieve a balanced chunk replica placement, the namenode randomly chooses chunk server to store new chunk.

The namenode is not tracking any information about blocks. It remembers just files and chunks. The reason of this is that the total number of blocks is so huge that the namenode cannot efficiently store and index all of them. The only client of data service is the metadata service.

2.3.2 Chunk Servers

Each chunk server keeps the storage for many chunk replicas, which are assigned by the namenode. A chunk server machine has many large volume disks attached, to

which it has complete access. A chunk server only deals with chunks and blocks, and it does not care about file namespace in the namenode. Internally on a chunk server, every chunk on disk is a file, consisting of data blocks and their checksum. A chuck server also holds a map which specifies relationships between chunk and file. Each chunk server also keeps a view about the chunks it owns and the location of the peer replicas for a given chunk. This view is copied from namenode and is kept as a local cache by the chunk server. Under instructions from namenode, different chunk servers may talk to each other to replicate chunks, or to create new copies of an existing replica. When a chunk no longer stores any alive chunks, the namenode starts garbage collection to remove the dead chunks and free the space.

3 Replication Strategies

3.1 Introduction

Currently, more data-intensive applications are moving their large-scale datasets into cloud. To provide high availability and durability of storage services as well as improving performance and scalability of the whole system, data replication is adopted by many mature platforms [1],[4],[6],[2],[12] and research studies [7][8][9][10][14][30] in cloud computing and storage. Data replication is to keep several identical copies of a data object in different servers that may distribute across multiple racks, houses and region-scale or global-scale datacenters, which can tolerate different levels of failures such as facility outages or regional disasters [4][10][23][30]. Replication strategy is now an indispensable feature in multiple datacenters [1][6][2][7][8][9][12], which may be hundreds or thousands of miles away from each other, to completely replicate data objects of services, not only because wide-area disasters such as power outages or earthquakes may occur in one datacenter [10][23], but also because replication across geographically distributed datacenters can mostly reduce latency and improve the whole throughput of the services in the cloud [6][7][8][9][11].

Availability and durability is guaranteed as one data object is replicated on many servers across datacenters, thus in the presence of failing of a few number of components such as servers and network at any time [1][4][10][23] or natural disasters occurring in one datacenter, the durable service of cloud storage won't be influenced because applications can normally access their data through servers containing replicas in other datacenters. Moreover, as each data object is replicated over multiple datacenters, it enables different applications to be served from the fastest datacenter or the datacenter with the lowest working load in parallel [1][6][9][11][31], thus providing high performance and throughput of the overall cloud storage system.

Common replication strategies can be divided into two categories: asynchronous replication and synchronous replication. They own distinct features and have different

impacts on availability, performance and throughput of the whole system. Besides, the cloud storage service should provide the upper applications with a consistent view of the data replicas especially during faults [6][7][8][9][11], which requires that data copies among diverse datacenters should be consistent with each other. However, these two replication strategies bring in new challenges to replication synchronization, which finally will influence the consistency of data replicas over multiple datacenters.

Additionally, the placement of data replicas is also an important aspect of replication strategy in multiple datacenters as it highly determines the load distribution, storage capacity usage, energy consumption and access latency, and many current systems and studies [1][4][6][10][24][26] adopt different policies on the placement of data replicas in the multiple-datacenter design on different demands.

In this section, we will present the main aspects and features of asynchronous replication, synchronous replication and placement of replicas.

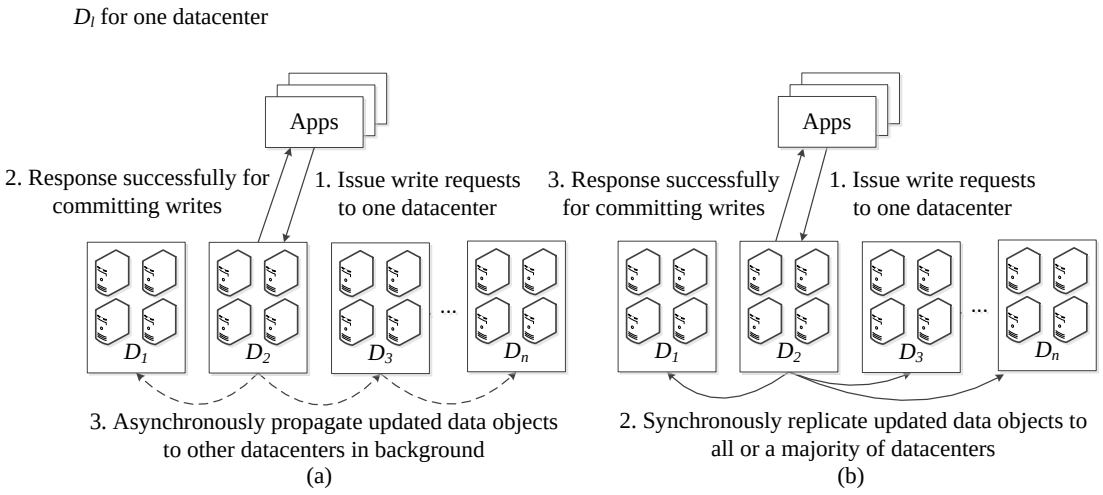

Figure 4 The working mechanism of asynchronous replication and synchronous replication over multiple datacenters. (a) for asynchronous replication and (b) for synchronous replication.

3.2 Asynchronous Replication

Figure 4(a) illustrates the working mechanism of asynchronous replication over multiple datacenters. As shown in Figure 4, the external applications issue write requests to one datacenter, which could be a fixed one configured previously or a random one chosen by applications, and get a successful response if the write requests completely commit in this datacenter. The updated data will be eventually propagated to other datacenters in background in an asynchronous manner [1][2][12]. Asynchronous replication is especially useful when the network latency between datacenters is at a high cost as applications only need to commit their write requests in one fast datacenter and don't have to wait for the data to be replicated in each

datacenter. Therefore, the overall performance and throughput for writes will be improved and systems with asynchronous replication can provide high scalability as they are decentralized. Now many prevailing systems such as Cassandra [12], Dynamo [1] and PNUTS [14] are using asynchronous replication.

However, asynchronous replication will bring a big challenge to consistency, since replicas may have conflicting changes with each other, that is, the view of all the replicas over multiple datacenters has the probability to be inconsistent at some time. Figure 5 presents a simple scenario that will cause inconsistency among replicas. Assume there are three datacenters *A*, *B* and *C*, and all of them hold data replica *d*. When a write request for *d* from application *P* is issued to *A* and successfully commits in *A*, *A* will response to *P* and then replicates the updated data d_1 to *B* and *C*. However, at the same time, another write request for *d* from application *Q* is issued to *C*. As *C* hasn't gotten to know the update of *d* in *A*, it normally accepts and processes this write request and then *d* in *C* turns into d_2 and will be replicated to *A* and *B*. As a result, there are now two different versions of the same data replica, and the system steps into an insistent state which means that a subsequent read may get two different data objects.

As there are also other factors such as server or network failure that will cause inconsistency in asynchronous replication over multiple datacenters [1][11], a few researches have been addressing this challenge of asynchronous replication. Eventual consistency [21] model is one scheme that is widely adopted by many studies and widespread distributed systems [1][12][13][14]. Eventual consistency model allows the whole system to be inconsistent temporarily but eventually, the conflicted data objects will merge into one singe data object and the view of the data replicas across multiple datacenters will become consistent at last. The process of merging conflicted data objects is critical in eventual consistency model and the merging decision can be made by the write timestamp [12][21], a chosen master [13][14] or even the applications [1].

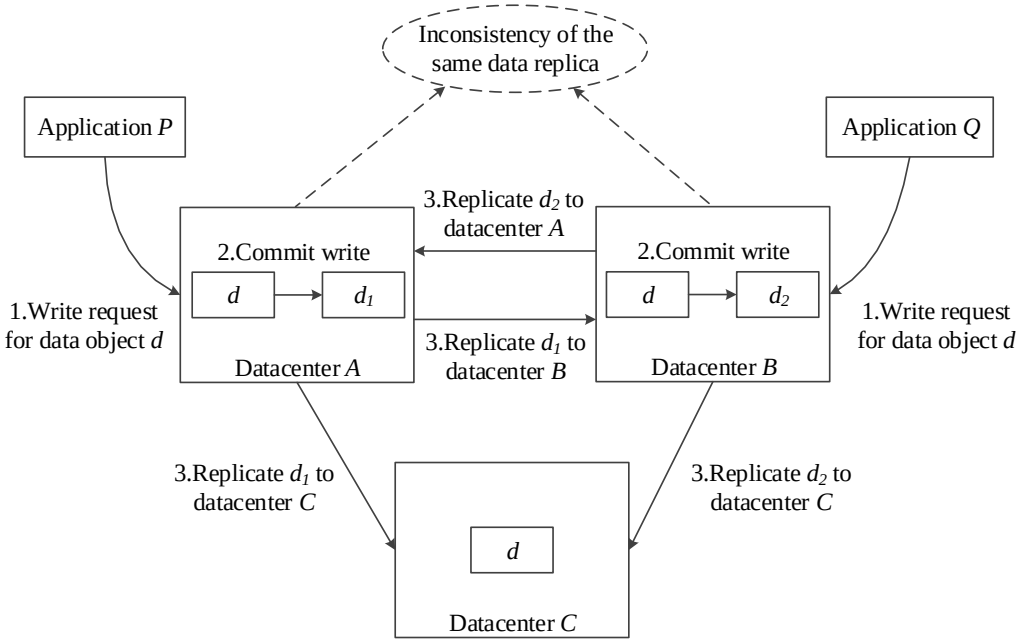

Figure 5 A scenario that causes inconsistent view of data replicas among datacenters under asynchronous replication.

Under asynchronous replication, a read request may get a stale data object from some datacenters, which will decline the performance of current reads and complicates application development. However, whether this circumstance is adverse depends on the applications. If applications such as search engine and shopping carts allow weaker consistency at reading or demand high quality of writing experience [1][12], asynchronous replication won't bring negative impacts to these applications.

3.3 Synchronous Replication

In contrast to asynchronous replication, synchronous replication requires that the updated data objects of write requests must be synchronously replicated to all or a majority of datacenters before applications get a successful response from the datacenter accepting the requests in the cloud, as presented in Figure 4(b). This synchronous replication mechanism can effectively guarantee a consistent view of cross-datacenter replicated data and it enables developers to build distributed systems that can provide strong consistency and a set of ACID semantics like transactions, which, compared with that in loosely consistent asynchronous replication, simplifies application building for the wide-area usage for the reason that applications can make use of serializable semantic properties while are free from write conflicts and system crashes [6][9][11][20][25].

The key point of synchronous replication is to keep states of replicas across different datacenters the same. A simple and intuitive way to realize this is to use synchronous master/slave mechanism [4][6][11]. The master waits for the writes to be fully committed in slaves before acknowledging to applications and is responsible for failure detection of the system. Another method to maintain consistent and up-to-date

replicas among datacenters is to use Paxos [16], which is a fault-tolerant and optimal consensus algorithm for RSM [15] in a decentralized way. Paxos works well when a majority of datacenters are alive and at current, many system services adopt Paxos [6][2][11][17][18][20] as their underlying synchronous replication algorithm.

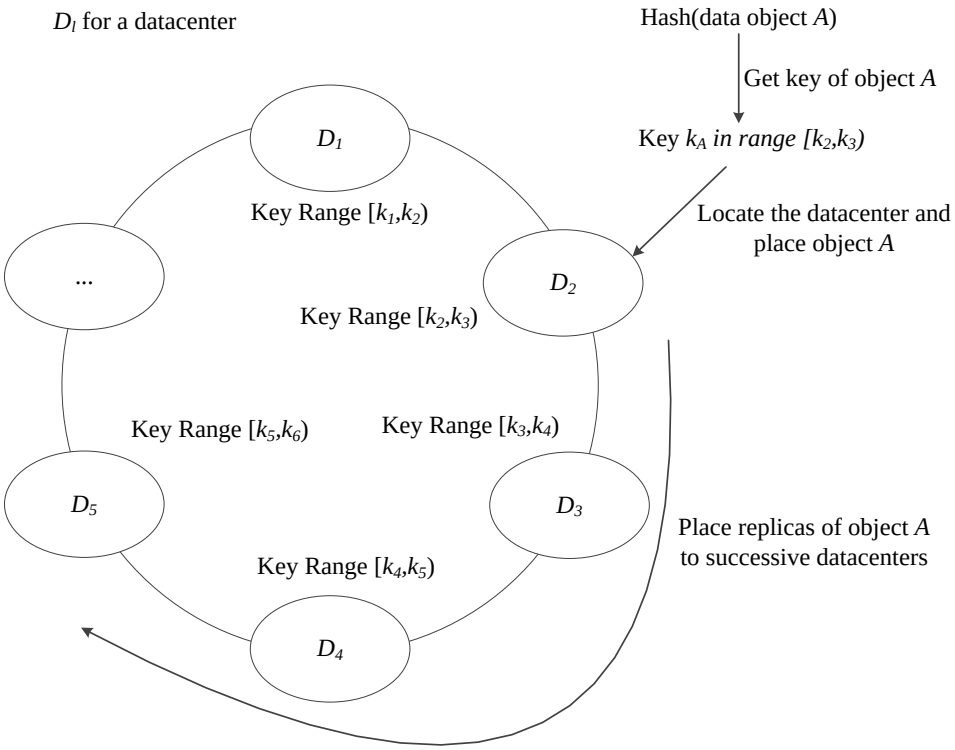

Figure 6 The decentralized method to place data replicas across multiple datacenters.

However, no matter which method is used, the overall throughput and performance of the services based on synchronous replication will be constrained when the communication latencies between datacenters are at high expense [7][9][11] and scalability is limited by strong consistency to certain extent. As a result, many researches put forward mechanisms to help improve the throughput and scalability of the whole system while not destroying the availability and consistency for applications. These mechanisms include reasonable partitioning of data [6][2][11], efficient use of concurrent control [7][11][31] and adopting combined consistent models [7][8][9][22][25].

3.4 Placement of Replicas

As cloud storage now holds enormous amount (usually petabytes) of data sets from large-scale applications, how to place data replicas across multiple datacenters also becomes a very important aspect in replication strategy as it is closely related to load balance, storage capacity usage, energy consumption and access latency of the whole system [6][10][19][24]. It is essential for both efficiently utilizing available datacenter resources and maximizing performance of the system.

Unbalanced replica placement will cause over-provisioning capacity and skewed utilization of some datacenters [26]. One way to address this issue is to choose a master or use partition layers to decide in which datacenter each data replica is placed [4][6][2]. This requires the master or partition layers to record the up-to-date load information of every datacenter so that they won't make unbalanced replica placement policies and can immediately decide to migrate data between datacenters to balance load. Another way is to use a decentralized method, as presented in Figure 6. We can form datacenters as a ring, each responsible for a range of keys. A data object can get its key through hash functions such as consistent hash and locate a datacenter according to its key. Then, replicas of this data object could be placed in this datacenter and its successive ones, similar to [1][12]. In this way, there is no need to maintain a master to keep information of each datacenter and if the hash functions could evenly distribute the keys, load balance can be achieved automatically.

Furthermore, as datacenters now consumes about 1.5% of the world's total energy and a big fraction of it does come from the consumption of storage in them [28][29], the number of datacenters to place the data replicas should also be considered carefully. If the number of datacenters to hold replicas increases, the storage capacity of the whole system will accordingly decease and the energy consumption improves [24][26][27] as those datacenters will contain large amounts of replicated data objects in storage. In addtition, placing data replicas in a high number of datacenters enables applications to survive wide-area disasters that will cause a few datacenter failures and thus, this can provide high availability for applications at the expense of storage capacity and energy consumption [6][2][7][11][25]. Moreover, when the number of datacenters to place replicas is large, applications can have a low access latency based on geographic locality, i.e., they can communicate with datacenters that are faster or have less working load [6][7][9]. Hence, system developers have to consider the trade-off between these features for the placement strategy of data replicas across multiple datacenters when they are building geographically distributed services for applications.

4 Data Striping Methods

4.1 Introduction

The main purpose of a storage system is to make data persistent, so reliability and availability should be top priority concern for storage systems. Actually, there are a variety of factors that may cause storage system unavailable. For example, if a server fails, the storage system is unable to provide storage services. Some physical damage to a hard disk will result in the loss of data stored. Therefore, it is indispensable for storage systems to introduce some techniques to make them reliable.

A lot of research work has been done in recent years to improve the availability and reliability of storage systems. The main idea is to generate some redundant

information of every data block and distribute them on different machines. When one server becomes outage, another server that holds the redundant data can replace the role of the broken server. During this time, the storage system can still provide storage service. When one data block is broken, then other redundant data blocks will restore the broken one. Thus, the availability and reliability is improved. Generally, redundant data can be presented in two ways: one is using full data backup mechanism, called full replication; the other is erasure code.

Full replication, also known as multi-copy method, is to store multiple replicates of data on separate disks, in order to make the data redundant. This method does not involve specialized encoding and decoding algorithms, and it has better fault-tolerance performance. But full replication has lower storage efficiency. Storage efficiency is the sum of effective capacity and free capacity divided by raw capacity. When storing N copies of replica, the disk utilization is only $1/N$. For relatively large storage systems, full replication brings extra storage overhead, resulting in high storage cost.

Along with the increase of the data that a storage system holds, a full replication method has been difficult to adapt to mass storage system for redundant mechanism in disk utilization and fault tolerance requirements. Therefore, erasure code is becoming a better solution for mass storage.

4.2 Erasure Code Types

Erasure code is derived from communication field. At first, it is mainly used to solve error detection and correction problems in data transmission. Afterwards, erasure code gradually applied to improve the reliability of storage systems. Thus, erasure code has been improved and promoted according to the characters of storage system. The main idea of erasure code is that the original data can be divided into k data fragments, and according to the k data fragments, m redundant fragments can be computed according some coding theory. The original data can be reconstructed by any of the $m + k$ fragments. There are many advantages of erasure code, the foremost of these is the high storage efficiency compared with the mirroring method.

There are many types of erasure code.

Reed-Solomon code [52] is an MDS code that can meets any number of data disks and redundant disk number. MDS code (maximum distance separable code) is a kind of code that can achieve the theoretically optimal storage utilization. The main idea of Reed Solomon code is to visualize the data encoded as a polynomial. Symbols in data are viewed as coefficients of a polynomial over a finite field. Reed Solomon code is a type of horizontal codes. Horizontal code has the property that data fragments and redundant fragments are stored separately. That is to say, each stripe is neither data stripe nor redundant stripe. Reed Solomon codes are usually divided into two categories: one is Vandermonde Reed Solomon code, and the other is Cauchy Reed

Solomon code [53]. The difference between these two categories of Reed Solomon codes is that they are using different generation matrix. For Vandermonde Reed Solomon code, the generation matrix is Vandermonde matrix, and multiplication on Galois filed is needed which is very complex. For Cauchy Reed Solomon code, it is Cauchy matrix, and every operation is XOR operation, which is coding efficient. Figure 7 shows the Encoding principle for Reed-Solomon codes.

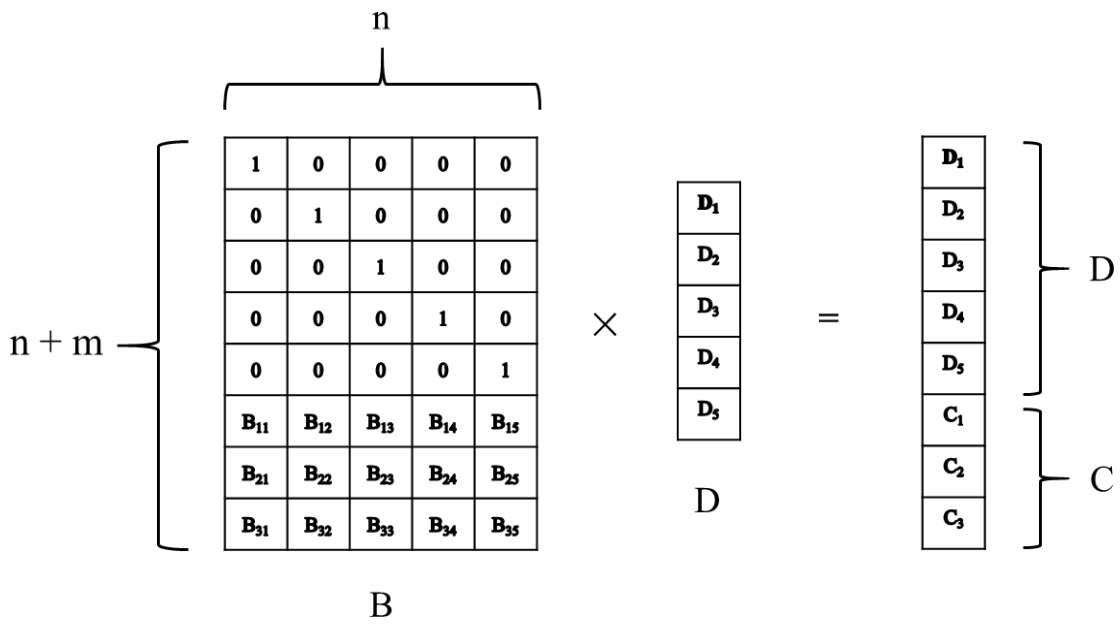

Figure 7 Reed-Solomon codes.

Compared with Reed Solomon Codes, Array Code [54] is totally based on XOR operation. Due to the efficient of encoding, decoding, updating and reconstruction, Array code is widely used. Array code can be categorized as two types due to the placement of data fragment and redundant fragment.

Horizontal parity array codes make data fragments and redundant fragments stored on different disks. By doing this, Horizontal parity array codes have better scalability. But most of it can just hold 2 disk failures. It has a drawback on updating data. Every time updating one data block will result in at least one read and one write operation on redundant disk. EVENODD code [55] is one kind of Horizontal parity array codes that used widely.

Vertical parity array codes make data fragment and redundant fragment stored in the same stripe. Because of this design, the efficiency of data update operation will be improved. However, the balance of vertical parity array code leads to a strong interdependency between the disks, which also led to its poor scalability. XCODE [56] is a kind of vertical parity array code, which has theoretically optimum efficiency on data update and reconstruction operation.

4.3 Erasure Code in Data Centers

In traditional storage systems such as early GFS and Windows Azure Storage, to ensure the reliability, triplication has been favored because of its ease of implementation. But triplication makes the stored data triple, and storage overhead is a major concern. So many system designers are considering erasure coding as an alternative. Most distributed file systems (GFS, HDFS, Windows Azure) create an append-only write workload for large block size. So data update performance is not a concern.

Using erasure code in distributed file systems, data reconstruction is a major concern. For one data of *k* data fragment and *m* redundant fragment, when any one of that fragment is broken or lost, to repair that broken fragment, *k* fragments size of network bandwidth will be needed. So some researchers found that the traditional erasure code does not fit distributed file system very well. In order to improve the performance of data repair, there are two ways.

One is reading from fewer fragments. In Windows Azure Storage System, a new set of code called Local Reconstruction Codes (LRC) [57] is adopted. The main idea of LRC is to reduce the number of fragments required to reconstruct the unavailable data fragment. To reduce the number of fragments needed, LRC introduced local parity and global parity. As Figure 8 shows below, x_0, x_1, x_2, y_0, y_1, y_2 are data fragments, p_x is the parity fragment of x_0, x_1, x_2. p_y is the parity fragment of y_0,y_1,y_2. p_0 and p_1 are global parity fragments. p_x and p_y are called local parity. p_0 and p_1 are global parity. When reconstructing x_0, instead of reading p_0 or p_1 and other 5 data fragment, it is more efficient to read x_1, x_2 and p_x to compute x_0. As we can see LRC is not a MDS, but it can greatly reduce the cost of data reconstruction.

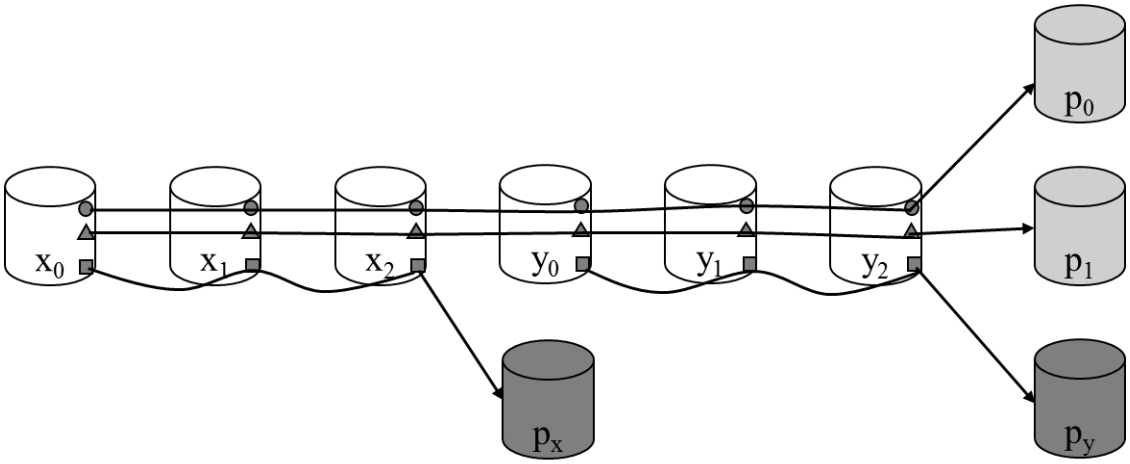

Figure 8 LRC codes.

Another way to improve reconstruction performance is to read more fragments but less data size from each. Regenerating codes [58] provide optimal recovery bandwidth among storage nodes. When reconstructing fragments, it does not just transmit the existing fragments, but sends a liner combination of fragments. By doing this, the

recovery data size to send will be reduced. Rotated Reed-Solomon codes [59] and RDOR[60] improve reconstruction performance in a similar way.

5 Consistency Models

5.1 Introduction

Constructing a globally distributed system requires many trade-offs among availability, consistency and scalability. Cloud storages are designed to serve for a large amount of internet-scale applications and platforms simultaneously, which is often named as infrastructure service. To meet requirements, a cloud storage must be designed and implemented as highly available and scalable, in order to serve consumers requests from all over the world.

One of the key challenges in build cloud storage is to provide a consistency guarantee to all client requests[63]. Cloud storage is a large distributed system deployed world-widely. It has to process millions of requests every hour. All the low-probability accidents in normal systems are often to happen in the datacenters of cloud storage. So all these problems must be taken care of in the design of the system. To guarantee consistent performance and high availability, replication techniques are often used in cloud storage. Although replication solves many problems, it has its costs. Different client requests may see inconsistent states of many replicas. To solve this problem, cloud storage must define a consistency model that all requests to replicas of the same data must follow.

Like many widespread distributed systems, cloud storage such as Amazon S3 often provide a weak consistency model called eventual consistency. Different clients may see different orders of updates to the same data object. Some cloud storage like Windows Azure also provides strong consistency that guarantees linearizability of every update from different clients. Details will be discussed in the following sub-sections.

5.2 Strong Consistency

Strong consistency is the most programmer-friendly consistency model. When a client commits an update, every other client would see the update in subsequent operations. Strong consistency can help achieve transparency of a distributed system. When developer uses a storage system with strong consistency, it appears like the system is a single component instead of many collaborating sub-components mixed together.

However, this approach has been proved as difficult to achieve since the middle of last century, in the database area for the first time. Databases are also systems with heavy use of data replications. Many of such database systems were design to shut down completely when it cannot satisfy this consistency because of node failures. But

this is not acceptable for cloud systems, which is so large that small failures are happening every minute.

Strong consistency has its weak points, one of which is that it lowers system availability. In the end of last century, with large-scale Internet systems growing up, designs of consistency model are rethought. Engineers and researchers began to reconsider the tradeoff between system availability and data consistency. In the year of 2000, CAP theorem was raised[61]. The theorem states that for three properties of shared-data systems—data consistency, system availability, and tolerance to network partition—only two can be achieved at any given time.

It's worth mentioning that the concept of consistency in cloud storage is different from the consistency in a transactional storage systems such as databases. The common ACID property (atomicity, consistency, isolation, durability) defined in databases is a different kind of consistency guarantee. In ACID, consistency means that the database is in a consistent state when a transaction is finished. No go-between situation is allowed.

5.3 Weak Consistency

According the CAP theory, a system can achieve both consistency and availability, if it does not tolerant network partitions. There many techniques which make this work, one of which is to use transaction protocols like two phase commit. The condition for this is that both client and server of the storage systems must be in the same administrative environment. If partition happens and client cannot observe this, the transaction protocol would fail. However, network partitions are very common in a large distributed systems, and as the scale goes up, the partition would be more often. This is one reason why one cannot achieve consistency and availability at the same time. The CAP theory leaves two choices for developers: 1) sticking to strong consistency and allowing system goes unavailable under partitions 2) using relaxed consistency [65] so that system is still available under network partitions.

No matter what kind of consistency model the system uses, it requires that the application developers are fully aware of the consistency model. Strong consistency is usually the easiest option for client developer. The only problem the developers have to deal with is to tolerate the unavailable situation that might happen to the system. If the system takes relaxed consistency and offers high availability, it may always accept client requests, but client developers have to remember that a write may get its delays and a read may not return the newest write. Then developers have to write the application in a way so that it can tolerant the delay update and stale read. There are many applications that can be design compatible for such relaxed consistency model and work fine.

There are two ways of looking at consistency. One is from the developer/client point of view: how they observe data updates. The other is from the server side: how

updates flow through the system and what guarantees systems can give with respect to updates.

Let's show consistency models using examples. Suppose we have a storage system which we treat it as a black box. To judge its consistency model we have several clients issuing requests to the system. Assume they are client A, client B, client C. All three clients issue both read and write requests to the system. The three clients are independent and irrelevant. They could run on different machines, processes or threads. The consistency model of the system can be defined by how and when observers (in this case the clients A, B, or C) see updates made to a data object in the storage systems. Assume client A has made an update to a data object:

- Strong consistency. After the update completes, any subsequent access (from any of A, B, or C) will return the updated value.
- Weak consistency. The system does not guarantee that subsequent accesses will return the updated value. A number of conditions need to be met before the value will be returned. The period between the update and the moment when it is guaranteed that any observer will always see the updated value is dubbed the inconsistency window.

There are many kinds of weak consistency, we list some of the most common ones as below.

- Causal consistency [66]. If client A has communicated to client B that it has updated a data item, a subsequent access by client B will return the updated value, and a write is guaranteed to supersede the earlier write. Access by client C that has no causal relationship to client A is subject to the normal eventual consistency rules.
- Eventual consistency [62]. This is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme. The most popular system that implements eventual consistency is the domain name system. Updates to a name are distributed according to a configured pattern and in combination with time-controlled caches; eventually, all clients will see the update.[64]
- Read-your-writes consistency. This is an important model where client A, after having updated a data item, always accesses the updated value and never sees an older value. This is a special case of the causal consistency model.
- Session consistency. This is a practical version of the previous model, where a client accesses the storage system in the context of a session. As long as the

session exists, the system guarantees read-your-writes consistency. If the session terminates because of a certain failure scenario, a new session must be created and the guarantees do not overlap the sessions.

- Monotonic read consistency. If a client has seen a particular value for the object, any subsequent accesses will never return any previous values.
- Monotonic write consistency. In this case, the system guarantees to serialize the writes by the same client. Systems that do not guarantee this level of consistency are notoriously difficult to program.

These consistency models are not exclusive and independent. Some of the above can be combined together. For example, the monotonic read consistency can be combined with session-level consistency. The combination of the two consistency is very practical for developers in a cloud storage system with eventual consistency. These two properties make it much easier for application developers to build up their apps. They also allow the storage system to keep a relax consistency and provide high availability. As you can see from these consistency models, quite a few different circumstances are possible. Applications need to choose whether or not one can deal with the consequences of particular consistency.

6 Cloud of Multiple Clouds

6.1 Introduction

Although cloud storage providers claim that their products are cost saving, trouble-free, worldwide 24/7 available and reliable, reality shows that (1) such services are sometimes not available to all customers; and (2) customers may experience vastly different accessibility patterns from different geographical locations. Furthermore, there is also a small chance that clients may not even be able to retrieve their data from a cloud provider at all, which usually occurrs due to network partitioning and/or temporary failure of cloud provider. For example, authors of [67] reported that this may also cause major cloud service providers to fail providing services for hours or days sometimes. Although cloud providers sign Service Level Agreements (SLA) with their clients to ensure availability of their services, users have complained that these SLAs are sometimes too tricky to break. Moreover, even when a SLA is violated, the compensation is only a minor discount for the payment and not to cover a customer's loss resulted by the violated SLA.

Global access experience can be considered as one specifically important issue of availability. In current major cloud storages, users are asked to create region-specific accounts/containers before putting their data blobs/objects into them. The storage provider then stores data blobs/objects into a datacenter in the selected locations; some providers may also create cross-region replicas solely for backup and disaster recovery. A typical result of such topology is an observation where users may

experience vastly different services based on the network condition between clients and the datacenter holding their required data. Data loss and/or corruption are other important potential threats to users' data should it be stored on a single cloud provider only. Although users of major cloud storage providers have rarely reported data loss and/or corruption, prevention of such problems are not 100% guaranteed either. Medium to small sized cloud providers may provide a more volatile situation to their customers as they are also in danger of bankruptcy as well.

In this section, we present a system named μLibCloud to address the two aforementioned problems of cloud customers; i.e., (1) availability of data as a whole and (2) different quality of services for different customers accessing data from different locations on the globe. μLibCloud is designed and implemented to automatically and transparently stripe data into multiple clouds –similar to RAID's principle in storing local data. μLibCloud is developed based on Apache libCloud project [3], and evaluated through global-wide experiments.

Our main contributions include: (1) to conduct global-wide experiments to show how several possible factors may affect availability and/or global accessibility of cloud storage services; (2) to use erasure codes based on observations. We then design and implement μLibCloud using erasure code to run benchmarks accessing several commercial clouds from different places in the world. The system proved the effectiveness of our method.

6.2 Architecture

Using a "cloud-of-cloud" rationale [68], μLibCloud is to improve availability and global access experience of data. Here the first challenge is how to efficiently and simultaneously use multiple cloud services. They follow different concepts and offer different ways to access their services. As shown in Figure 9, cloud storage providers usually provide REST/SOAP web service interface to developers along with their libraries for different programming languages for developers to further facilitate building cloud applications. To concurrently use multiple cloud storages, two options are available. The first option is to set up proxy among cloud storages and applications. In this case, all data requests need to go through this proxy. To store data in cloud storages, this proxy receives original data from client, divides the data into several shares, and sends each share to different clouds using different libraries. To retrieve data, it fetches data shares from each cloud, rebuilds the data, and sends it back to clients. The second option –more complicated– is to integrate the support for multiple cloud storages directly into a new client library –replacing original ones. In this case, client applications only use this newly provided library to connect to all clouds. The main difference between these two options is the transparency in the second option to spread/collect data to/from multiple clouds.

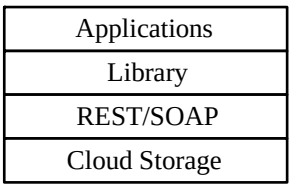

Figure 9 Layer abstraction of cloud storage

The first choice, is more straightforward in design; it uses a single layer for extra work, keeps the client neat and clean, includes many original libraries when implemented, and is usually run on independent servers. It also brings more complexity to system developers to maintain extra servers and their proper functioning. The second choice, on the other hand, benefits developers by providing them a unique tool; this approach also reduces security risk because developers do not need to put their secret keys on the proxy. It however also leads to other challenges on how to design and implement such systems; e.g., how multiple clients can coordinate with each other without extra servers. Furthermore, the client library must be efficient and resource saving because it needs to be run along with application codes.

In the design of µLibCloud, we chose to practice the second option so that it has less burden on application developers. We also assume that consumers who choose to use cloud storage rather than to build their own infrastructure would not want to set up another server to make everything work. Figure 10 shows the basic architecture of µLibCloud with a single client; this figure also shows how µLibCloud serves upper-level users, while hiding most of development complexities of such systems.

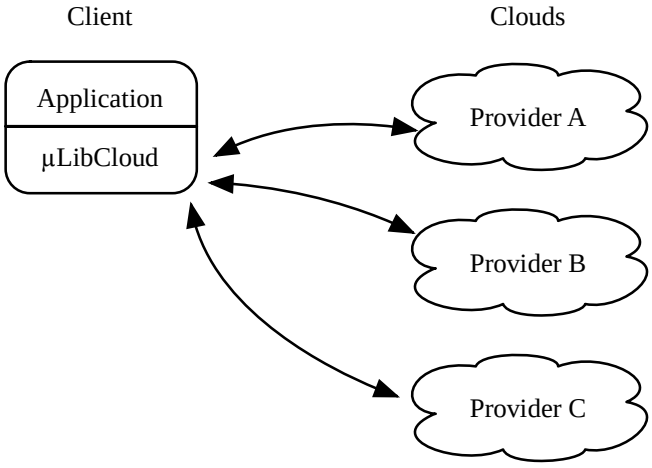

Figure 10 Architecture with single client

6.3 Data Striping

As described before, data is first encoded into several original and redundant shares, and then stored on different providers. Through this redundancy, data not only is protected against possible failures of particular providers –high availability, but also tolerates the instability of individual clouds and provides consistent performance.

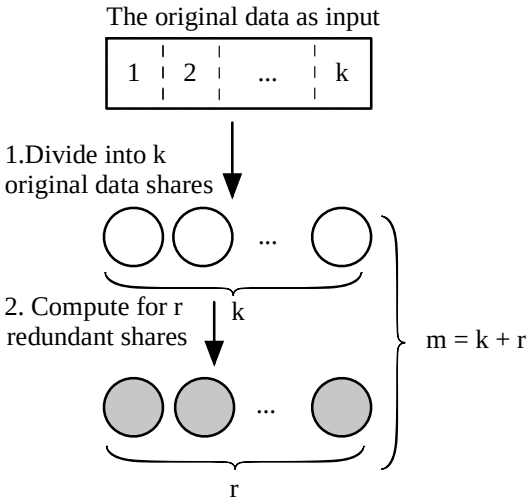

Figure 11 Principle of erasure coding

Among many possible choices for data encoding [69], we choose the most widely used erasure code [70] that is widely used in both storage hardware [71] and distributed systems [72]. Here, coding efficiency is a major concern because all the data striping algorithm work is performed at clients' side; i.e., large overheads that could decrease performance of applications is strongly unacceptable.

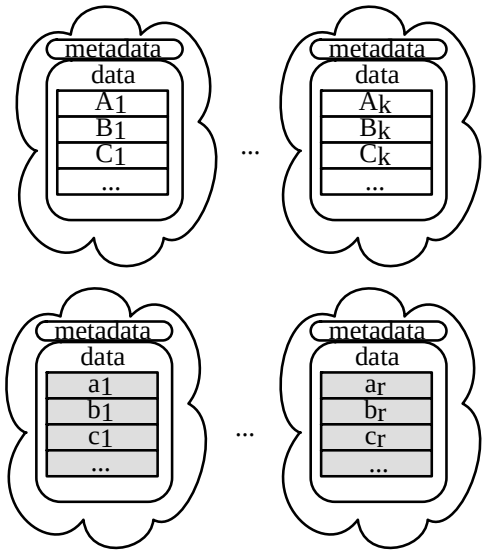

Figure 12 Data stripes stored in each cloud

Figure 11 shows principles of erasure coding. As can be seen, data is first divided into *k* equal-sized fragments called original data shares. Then, *r* parity fragments with the same size as original data shares are computed and called redundant data shares. This will generate a total of $m = k + r$ equal-sized shares. The erasure code algorithm guarantees any arbitrary *k* shares –out of total *m* shares– is sufficient enough to reconstruct the original data. Both *k* and *r* are positive values and are predefined by each user.

Here, we also define redundancy rate as $R = m / k$ to reflect the amount of storage overhead for storing data. For example, if $k = 1, m = 2$; then, $R = m / k = 200\%$. It means that each data takes twice of its original size when stored: one original and one replica. In this case, each piece is enough to reconstruct the original data – like RAID 1(mirroring). If $k = 4, m = 5$ (like RAID 5); then, $R = m / k = 120\%$. It means that we need extra 20% of storage to store any data. In this case, every four pieces –out of all available five pieces– are enough to reconstruct the original data.

In practice, we do not simply just divide an object into several parts and encode them, but the original data is first divided into several chunks, and then erasure coding is performed on each chunk; default chunk's size is usually 64KB (Figure 12). There are two benefits in splitting data into several chunks: (1) computation of erasure coding can be parallelized, and (2) reading and writing of file data –such as video and audio– can also be easily supported.

6.4 Retrieving Strategy

If a developer divides data into *(k, m)* shares, among all *m* parts of data the client library only needs *k* parts to reconstruct the original data. Although retrieving all parts of data could avoid the potential risks of failures, it is unnecessary in most cases. It also wastes more bandwidth and costs more money. Here, although retrieving *k* data shares to recover the data is enough, selecting the best possible *k* shares can be tricky. In μLibCloud we offer the following three data fetching strategies.

(1) Efficient: Users want to use the *k* most available clouds to retrieve data pieces. Here, to determine which ones are faster, μLibCloud dynamically measures their download speed. When retrieving an object, all metadata files are downloaded first and their link speed is recorded. Upon that, *k* fastest clouds to fetch data are selected. During downloading the main data, μLibCloud keeps recording the download speed to compute its average. The larger data is, the more accurate network estimation would be.

(2) Economical: If application is mainly run in the background –like a backup program storing data into clouds [73]–, users can tolerate spending more time. In such cases, economical cost is more important than speed. μLibCloud also offers a cost-saving mode, in which it will select k providers with lowest prices.

(3) Custom: We also offer an option, allowing developers to set priorities on their own. This may be preferable in case that they are using computing and storage resources provided by the same provider. For example, if a developer is deploying applications into EC2 and use storage of S3, it would be reasonable that s/he wants to use S3 as the first choice.

6.5 Mutual Exclusion

When there are more than one client in the system, they must be able to coordinate with each in certain ways to avoid conflicts. Such conflicts can result in not only client read failures, but also inconsistent states and/or even data loss. For example, if two clients concurrently write to the same data file without any locking, they may write to each other's share and produce problems. In the worst case scenario, if the provider takes an eventual consistency model (like Amazon S3), all unordered writes would succeed although only the later ones become effective. As a result, it would be very probable that a client succeeds modifying several data shares, while the other client succeeds in the rest of data shares; both clients would return successful, while data inconsistency has already occurred! The following options are among the most suitable one for our needs.

(1) Setting up a central lock server such as ZooKeeper [74] to coordinate all writes. This approach is easy and correct for a system like μLibCloud, yet with certain flaws. Firstly, with this approach clients need to maintain another system, which violates goals and principles of using clouds for simplicity in the first place. Secondly, coordinators like ZooKeeper usually has throughput issues because of their leader-follower architecture, especially in internet-scale situation. Although this can be reduced by manually partitioning data onto multiple groups of ZooKeeper systems, this would still make the system extremely complex.

(2) Running a client-client agreement protocol. Here, instead of deploying an additional central lock service, agreement protocols such as Paxos [75] handles the situation. This approach eliminates the trouble of bringing a lock service, but requires clients to be able to communicate with each other. In this case, frequent membership changes can seriously damage system performance. In fact, this approach is almost the same –in logic– as the first option if each client runs with a ZooKeeper member deployed to the same machine.

(3) Manipulating lock files on each cloud storage. Instead of setting up an additional lock server or running an agreement protocol among clients, there is another approach more suitable to this situation. Each client creates empty files on each cloud as lock-files; this is called mutual exclusion in the area of distributed algorithms [76]. This option is more difficult to achieve because each cloud is purely an object storage that offers neither computing ability, nor a common compare and swap (cas) semantics usually used in fulfilling lock services.

In order to achieve mutual exclusion without introducing new bottlenecks, we introduced Algorithm 1 based on the third option. This algorithm is an improved version of another algorithm formerly designed by Bessani a.l[77].

Algorithm 1 Mutual exclusion at client side

```
input: id, and providers[1, 2, .., m]
output: success or failure
for i from 1 to m do
  create a file named lock_id on provider[i]
end for
count = 0
for i from 1 to m do
  list all lock files on provider[i]
  if found any lock files created by other client then
    count = count + 1
  end if
end for
if count ≥ m/2 then
  for i from 1 to m do
    delete lock_id from provider[k]
  end for
else
  {critical section}          // lock succeeds
end if
```

Following comments is worth noting about this algorithms.

(1) The algorithm is fault tolerate to possible failures of less than $m/2$ providers. In case a client fails and stops during any step, we add a timestamp t_{create} to the name of each lock file. Thus, when a client lists a file name with the $t_{create}+t_{delta} < t_{now}$, s/he can confidently deletes the expiring lock. To maintain correctness, we must choose a t_{delta} large enough to cover the entire operation time when created; it must also be able to tolerate possible time differences among clients.

(2) To be correct, the algorithm requires each cloud to have an appropriate consistency model. To be specific, after each create command, all lists must see the creation. However, several major cloud providers, such as Amazon S3, employs an eventual consistency model [78]. It means the writes are not visible to reads immediately, and if one client detects a change, it does not imply other clients can also detect it. To tolerate eventual consistency, the client may need to wait for another time period, after each write to make sure it can be seen by all clients too. The time period is set by observation to model time delays among client [79].

Amazon S3 recently releases an enhanced consistency model to most of its cloud storages, namely ”read-after-write” consistency to ensure that for newly created objects, the write (not overwrite) can be seen immediately. Our algorithm (Algorithm 1) employs this feature in its locking system; this is why Algorithm 1 creates new lock files instead of writing to the old ones.

(3) The algorithm is obstruction-free [80]; i.e., it is still possible –although very rare– that no client can progress. This flaw could be tolerated because most applications tend to have many more reads than writes –where only very few writes require mutual exclusion.

7 Privacy and Security of Storage System

7.1 Introduction

In the last few years, cloud computing has enabled more and more customers (such as companies or developers) to run their applications on the remote servers with elastic storage capacity and computing resources required on demand. The proliferation of cloud computing encourages customers to store and keep their data in the cloud instead of maintaining local data storage [32][33][34][38][39]. However, a key factor that may hinder the process of data migration from local storage to the cloud is the potential privacy and security concerns inside clouds [33][34]. As customers don't own and manage remote servers directly by themselves, any malicious applications or administrators in the cloud can get access to, abuse or even damage the data of normal customers' applications. This phenomenon is especially adverse to the confidentiality of sensitive data objects of customers such as banks or financial companies. Under this circumstance, datacenters in the cloud must maintain strong protections on the privacy and security of data objects against untrusted applications, servers and administrators during the process of data storing and accessing [34][36][39][46].

To guarantee data privacy and security in storage system of datacenters in the cloud, several basic solutions such as data access control [38][39][40][41], data isolation [36][37][42][46][47] and cryptographic techniques [35][40][43][44][45] have been proposed by researchers. All these solutions are intended to meet different requirements of data privacy and security and to make even the most privacy and security demanding applications to migrate their sensitive data into cloud with no concerns. In this section, combined with our experience of building privacy and security policies in datacenters in the cloud, we will present how these mechanisms can be used in a real world.

7.2 Fine-grained Data Access Control

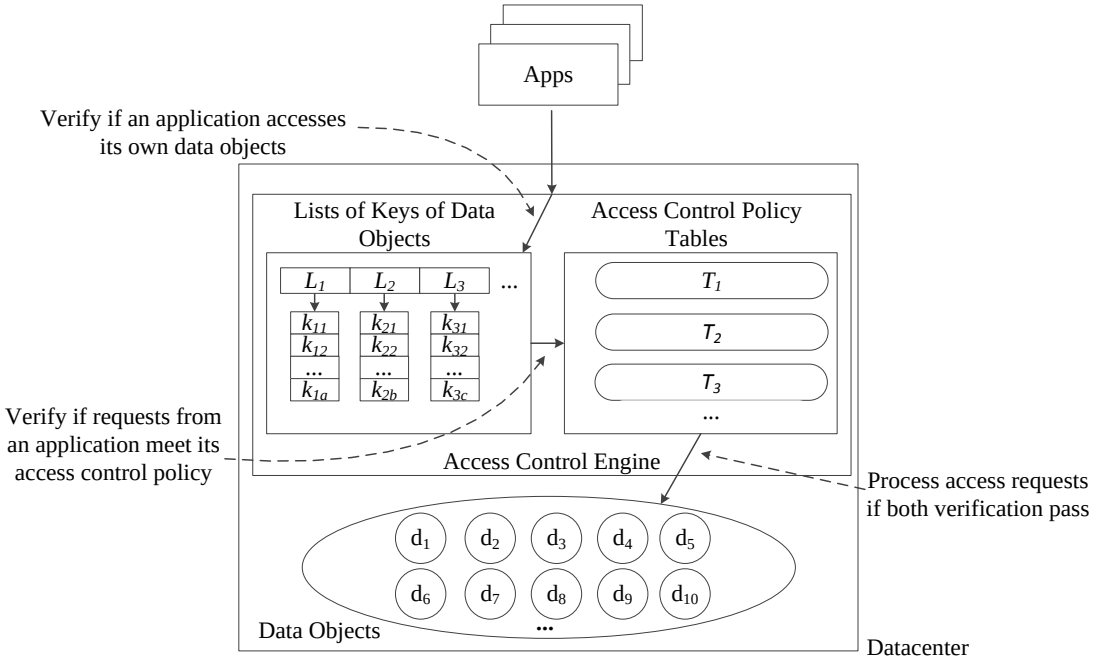

Figure 13 The overview of the fine-grained access control mechanism on the data object level.

Data access control is highly related to the privacy and security provided to applications when they are accessing the data [33][38][39][41]. Applications, if not allowed, don't have the authority to access the data of others. Besides, each application may have its own access control policies to maintain the data privacy and security among its users. For instance, one application may require that only its administrators can have the authority to modify and delete its data and other common users can only read these data. Therefore, storage systems in datacenters must ensure strict and flexible data access control mechanisms for upper applications to secure the data object sets of every application.

Algorithm 2 The procedure of storing a data object.

Input: P_n, d_m

1. *Get* k_m *of* d_m
2. *Get* L_n *of* P_n
3. *Insert* k_m *into* L_n
4. *Keep* d_m *in physical storage*

Figure 13 illustrates the overview of a fine-grained access control mechanism on data object level in a datacenter. As presented in Figure 13, there are two main data structures for the correct process of fine-grained data-object-level access control: a set of lists keeping the keys of data objects that belong to each application and a set of tables recording each application's access control policy. Every application owns its list of keys and access control policy table. When one application stores a data object into the datacenter, the storage system will allocate a globally unique key to this data

object and add this key into the list of this application, which means this data object does belong to such application. Denote an application as P_n, the list of P_n as L_n, a data object as d_m and the key of data object d_m as k_m, then the process of storing data in this mechanism could be summarized as Algorithm 2.

When an access request for a data object issued from an application arrives at the datacenter, the storage system will first get the key of the data object and verifies if this key is in this application's list. Storage system will forbid the application to access this data object if the verification fails. This procedure ensures that data objects of one application are isolated from the other applications against illegal intrusion. Moreover, if the verification passes, the system will further check if this access request meets the requirements listed in the access control policies table of the application. This will prevent unauthorized application users from abusing operations on data of this application that may potentially damage these data. Applications can set and modify their access control policies according to their own demands and the policy information are recorded in their access control policy tables respectively. The access request is accepted and processed only after the check in the access control policy table successes. Denote an access request as R_p and access control policies table of application P_n as T_n, then the procedure to process an access request can be illustrate as Algorithm 3.

With Algorithm 2 and Algorithm 3, the data privacy and security could be achieved across applications through fine-grained data-object-level access control mechanism without impacting the normal usage of data by authorized users of each application. Furthermore, as these two data structures (lists and tables) that are used by the access control mechanism could keep a consistent view across multiple datacenters using replication strategy presented in Section 3, the privacy and security of data could be easily guaranteed through this fine-grained data-object-level data access control mechanism among multiple datacenters in the cloud.

Algorithm 3 The procedure of processing an access request on a data object.

Input: P_n, R_p, d_m

1. *Get* k_m *of* d_m
2. *Get* L_n *of* P_n
3. **if** $k_m \in L_n$ **then**
4. *Get* T_n *of* P_n
5. **if** R_p *meets requirements in* T_n **then**
6. *Process* R_p
7. **else**
8. *Refuse to process* R_p
9. **end if**
10. **else**
11. *Refuse to process* R_p
12. **end if**

7.3 Security on Storage Server

Under fine-grained data-object-level access control mechanism, the privacy and security of applications' data could be protected against external untrusted users and applications. However, data stored in the storages servers of datacenters are still prone to abuse or compromise by untrusted processes running in these servers or malicious administrators of datacenters that can get the whole authority of the OS [36][37][51]. To address this issue, most studies [36][37][42][46][47][51] use virtual-machine-based protection mechanisms to isolate applications' data kept in hardware (memories and disks) of storage servers from operating systems and other processes, and to authenticate the integrity of these data. This protection ensures that even operating systems carry out the overall task of managing these data, they cannot read or modify them. With this guarantee, even though malicious administrators or untrusted processes get the authority of OS, they have no access to abusing or damaging the data stored in the hardware. When trusted applications request to get their data, this mechanism would make sure that these applications will be presented with a normal view of their original data, hiding the complex underlying details of protection. Hence, the privacy and security of applications' data can be maintained in storage servers of datacenters in the cloud.

Figure 14 characterizes the architecture of the privacy and security protection mechanism in storage servers. The key component, as shown in Figure 14, to protect the privacy and security of applications' data in hardware is the virtual machine monitor (VMM). The VMM could monitor the process/OS interactions such as system calls [36][42] and directly manage the hardware to isolate memories and disks from operating systems [37][42][46], which makes it possible to prevent the data privacy and security against malicious processes or administrators that can get the authority to control operating systems.

Generally, each process owns its independent virtual memory address space and is associated with a page table that maps the virtual memory address into the physical memory address [48] to use memory. The page tables of processes and the operations of address mappings are managed by the OS and thus, it has the authority to access the memory address space of all processes running on it. As applications' requests are served by specific processes in storage servers of datacenters, once malicious processes or administrators steal the operating system's authority, they can easily access the data of other normal processes through their page tables and threaten the privacy and security of applications' data. To address this challenge, VMM could protect the page tables of each process and complete the operations of memory address mappings instead of operating system [48][51]. The OS can only access its kernel memory space through its own page table, without interleaving with other processes. However, even though the OS doesn't know the distribution of processes' virtual memory in the physical memory, malicious processes or administrators could also access the physical memory through OS [48][49] and analyze or tamper the data in the memory [42][50]. As a result, VMM is responsible for keeping the data in the

physical memory in an encrypted and integrated view [49]. When a process requests to put data into memory, VMM will detect this request, encrypt the data and then put the encrypted data into the memory. If one trusted process requests to get its data in the memory, VMM will first authenticate the integrity of the encrypted data and then decrypt them before returning the original clear data to this process, which doesn't have to cover this middle process and just utilizes memory as normal. To complete the encrypting and decrypting procedures mentioned above, VMM holds a specific zone of memory that is secure enough against the attacks from operating systems and processes. Consequently, when processes are serving applications' requests in memory of storage servers, the privacy and security of their data in memory can be strongly protected.

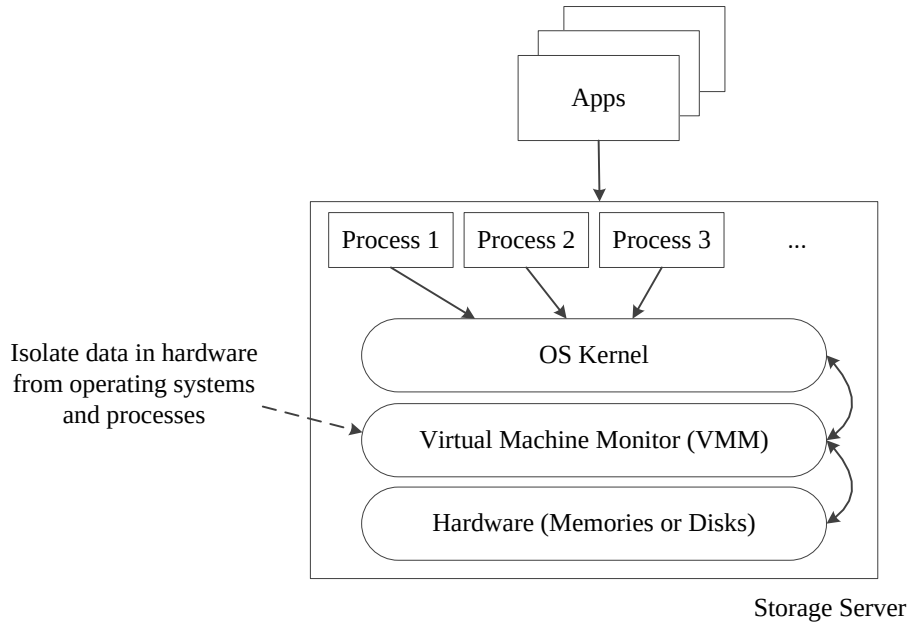

Figure 14 The architecture of the privacy and security protection mechanism in a storage server.

As most of applications' data will be stored into disks of storage servers in datacenters, it is also critical to guarantee the privacy and security of data in disks [36][42][51] not only because untrusted processes and administrators that get the authority of OS can directly access data in disks through I/O operations, but administrators could fetch disks manually. As a result, data in disks must also be stored in an encrypted view so that even some processes or administrators get control on the disks of storage servers, they have no way to abuse or compromise the data stored in them. VMM also has the responsibility for data encryption/decryption when processes interact with disks through the OS. When a process wants to write its data into disks, it will use a system call sys_write [48] and passes the data to the operating system, which will execute the operations to really write data to disks. VMM will detect this system call from the process and obtain the data before passing to the operating system. Then VMM encrypts these data and calculate the checksum of the encrypted data for future

integrity verification. After this procedure, VMM will transfer the encrypted data to the operating system that will normally write these data into disks. Similarly, when one process requests to get its data from disks, it will issue a system call sys_read to the operating system to fetch these data. VMM will also detect this system call and wait for the operating system to complete the read operations of the encrypted data. Then VMM authenticates the integrity of the encrypted data, decrypts them and return plain data to the process. All the underlying details of encryption/decryption are still hidded for the processes and to the operating system, although it manages the data during the operations of read and write, it only views data after encryption and can't threaten the privacy and security of the original data objects.

With these virtual-machine-based mechanisms, the data of applications can be kept in storage servers of datacenters without concerns of being abused or compromised by malicious processes or administrators in the datacenters. As data privacy and security can be achieved in hardware of each storage server, datacenters in the cloud have the ability to provide high privacy and security for applications to move their large sets of data into cloud and freely access their data on demands.

8 Conclusion and Future Directions

In this chapter we mainly discussed the architecture of modern cloud storage and several key techniques used in building such systems. Cloud storage systems are typically large distributed systems composed of thousands of machines and network devices over many datacenters across multiple continents. Cloud storage and cloud computing are the very mixture of modern storage and network technology. To build and maintain such systems calls for large amount of efforts from numerous developers and maintainers. Although we have discussed about replication, data striping, data consistency, security and some other issues, there are still much more of the iceberg we have not touched. Many conventional techniques in traditional storage techniques applied in cloud storage have the potentiality to evolve, such as the example we give about cloud-of-clouds, which arise from the traditional RAID system. To summarize, cloud storage is a valued area in both practice and research, and the goal of this chapter is to give a sight at it when it grows into the global scale.

[References]

[1] Varia, Jinesh. "Cloud architectures." White Paper of Amazon, jineshvaria. s3. amazonaws. com/public/cloudarchitectures-varia. pdf (2008).

[2] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman

Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11), pages 143-157, 2011.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. Dynamo: amazon's highly available key-value store. Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP' 07), pages 205-220, 2007.

[4] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google file system. Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP' 03), pages 29-43, 2003.

[5] Chang, Fay, et al. "Bigtable: A distributed storage system for structured data." ACM Transactions on Computer Systems (TOCS) 26.2 (2008): 4.

[6] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura , David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford, D. Woodford. Spanner: Google's globally-distributed database. Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI' 12), pages 251-264, 2012.

[7] Yair Sovran , Russell Power, Marcos K. Aguilera, Jinyang Li. Transactional storage for geo-replicated systems. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11), pages 385-400, 2011.

[8] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11), pages 401-416, 2011.

[9] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguica, Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12), pages 265-278, 2012.

[10] Luiz André Barroso, Urs Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Morgan & Claypool Publishers, DOI: 10.2200/S00193ED1V01Y200905CAC006, 2009.

[11] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Floyd, Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In 5th Conference on Innovative Data Systems Research, pages 223--234, 2011.

[12] Avinash Lakshman , Prashant Malik. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2), pages 35-40, 2010.

[13] D. B. Terry , M. M. Theimer , Karin Petersen , A. J. Demers , M. J. Spreitzer , C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. Proceedings of the fifteenth ACM Symposium on Operating Systems Principles (SOSP'95), pages 172-182, 1995.

[14] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno, Nick Puz, Daniel Weaver, Ramana Yerneni.

PNUTS: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment, 1(2), pages 1277-1288, 2008.
[15] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys (CSUR), 22(4), pages 299-319, 1990.
[16] Leslie Lamport. Paxos made simple. ACM SIGACT News Distributed Computing Column, 32(4), pages 18-25, 2001.
[17] Tushar D. Chandra, Robert Griesemer, Joshua Redstone. Paxos made live: an engineering perspective. Proceedings of the twenty-sixth annual ACM Symposium on Principles of Distributed Computing, pages 398-407, 2007.
[18] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI'06), pages 335-350, 2006.
[19] Jeff Dean. Designs, Lessons, and Advice from Building Large Distributed Systems. Keynote from LADIS, 2009.
[20] Stacy Patterson, Aaron J. Elmore, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi. Serializability, not serial: concurrency control and availability in multi-datacenter datastores. Proceedings of the VLDB Endowment, 5(11), PAGES 1459-1470, 2012.
[21] Werner Vogels. Eventually consistent. Communications of the ACM - Rural engineering development, 52(1), pages 40-44, 2009.
[22] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, Thomas Anderson. Scalable consistency in Scatter. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11), pages 15-28, 2011.
[23] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, Sean Quinlan. Availability in globally distributed storage systems. Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI'10), No. 1-7, 2010.
[24] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Carlos Maltzahn. CRUSH: controlled, scalable, decentralized placement of replicated data. Proceedings of the 2006 ACM/IEEE conference on Supercomputing, 2006.
[25] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, David G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13), 2013.
[26] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, Harbinder Bhogan. Volley: automated data placement for geo-distributed cloud services. Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation (NSDI'10), 2010.
[27] Anton Beloglazov, Rajkumar Buyya. Energy Efficient Resource Management in Virtualized Cloud Data Centers. Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pages 826-831, 2010.
[28] Zhichao Li, Kevin M. Greenan, Andrew W. Leung, Erez Zadok. Power Consumption in Enterprise-Scale Backup Storage Systems. Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12), pages 65-71, 2012.
[29] J. G. Koomey. Growth in data center electricity use 2005 to 2010. Technical report, Standord University, 2011.
[30] Yi Lin, Bettina Kemm, Marta Patiño-Martínez, Ricardo Jiménez-Peris.

Middleware based data replication providing snapshot isolation. Proceedings of the 2005 ACM SIGMOD international conference on Management of data, pages 419-430, 2005.
[31] Daniel Peng, Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation, 2010.
[32] Cong Wang, Qian Wang, and Kui Ren, Wenjing Lou. Privacy-Preserving Public Auditing for Data Storage Security in Cloud Computing. Proceedings of IEEE INFOCOM, 2010.
[33] S. Subashini, V. Kavitha. A survey on security issues in service delivery models of cloud computing. Journal of Network and Computer Applications, 34(1), pages 1-11, 2011.
[34] H. Takabi, J.B.D. Joshi, G. Ahn. Security and Privacy Challenges in Cloud Computing Environments. IEEE Security and Privacy, 8(6), pages 24–31, 2010.
[35] Kevin D. Bowers, Ari Juels, Alina Oprea. HAIL: a high-availability and integrity layer for cloud storage. Proceedings of the 16th ACM conference on Computer and Communications Security (CCS'09), pages 187-198, 2009.
[36] Fengzhe Zhang, Jin Chen, Haibo Chen, Binyu Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11), pages 203-216, 2011.
[37] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. Proceedings of the 13th international conference on Architectural Support for Programming Languages and Operating Systems, pages 2-13, 2008.
[38] Wassim Itani, Ayman Kayssi, Ali Chehab. Privacy as a Service: Privacy-Aware Data Storage and Processing in Cloud Computing Architectures. Proceedings of Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing, pages 711-716, 2009.
[39] Shucheng Yu, Cong Wang, Kui Ren, Wenjing Lou. Achieving Secure, Scalable, and Fine-grained Data Access Control in Cloud Computing. Proceedings of IEEE INFOCOM, 2010.
[40] Vipul Goyal, Omkant Pandey, Amit Sahai, Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. Proceedings of the 13th ACM conference on Computer and Communications Security (CCS'06), pages 89-98, 2006.
[41] Myong H. Kang, Joon S. Park, Judith N. Froscher. Access control mechanisms for inter-organizational workflow. Proceedings of the sixth ACM symposium on Access Control Models and Technologies, pages 66-74, 2001.
[42] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P. Yew, and W. Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. Parallel Processing Institute Technical Report, Number: FDUPPITR-2007-0801, Fudan University, 2007.
[43] Lein Harn, Hung-Yu Lin. A cryptographic key generation scheme for multilevel data security. Computer & Security, 9(6), pages 539-546, 1990.
[44] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. Proceedings of the 9th ACM conference on Computer and Communications Security (CCS'02), pages 88-97, 2002.

[45] Phillip Rogaway. Bucket hashing and its application to fast message authentication. CRYPTO, volume 963 of LNCS, pages 29–42, 1995.
[46] David Lie, Chandramohan A. Thekkath, Mark Horowitz. Implementing an untrusted operating system on trusted hardware. Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP'03), pages 178-192, 2003.
[47] Stephen T. Jones, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems, pages 14-24, 2006.
[48] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. Operating System Concepts. John Wiley & Sons, 2009.
[49] Guillaume Duc, Ronan Keryell. CryptoPage: an Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection. Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06), pages 483-492, 2006.
[50] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, Mark Horowitz. Architectural support for copy and tamper resistant software. ACM SIGPLAN Notices, 35(11), pages 168-177, 2000.
[51] Hou Qinghua, Wu Yongwei, Zheng Weimin, Yang Guangwen. A Method on Protection of User Data Privacy in Cloud Storage Platform. Journal of Computer Research and Development, 48(7), pages 1146-1154, 2011.
[52] Reed I S, Solomon G. Polynomial codes over certain finite fields [J]. Journal of the Society for Industrial & Applied Mathematics, 1960, 8(2): 300-304.
[53] Roth R M, Lempel A. On MDS codes via Cauchy matrices [J]. Information Theory, IEEE Transactions on, 1989, 35(6): 1314-1319.
[54] Blaum M, Farrell P, Tilborg H. Array Codes [M]. Amsterdam, Netherlands: Elsevier Science B V,1998
[55] Blaum M, Brady J, Bruck J, et al. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures[J]. Computers, IEEE Transactions on, 1995, 44(2): 192-202.
[56] Xu L, Bruck J. X-code: MDS array codes with optimal encoding[J]. Information Theory, IEEE Transactions on, 1999, 45(1): 272-276.
[57] Huang, Cheng, et al. "Erasure coding in windows azure storage." USENIX ATC. 2012.
[58] Dimakis A G, Godfrey P B, Wu Y, et al. Network coding for distributed storage systems[J]. Information Theory, IEEE Transactions on, 2010, 56(9): 4539-4551.
[59] Khan, Osama, et al. "Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads." Proc. of USENIX FAST. 2012.
[60] Xiang, Liping, et al. "Optimal recovery of single disk failure in RDP code storage systems." ACM SIGMETRICS Performance Evaluation Review. Vol. 38. No. 1. ACM, 2010.
[61] Brewer, Eric A. "Towards robust distributed systems." PODC. 2000.
[62] Vogels, Werner. "Eventually consistent." Communications of the ACM 52.1 (2009): 40-44.
[63] Birman, Kenneth P. "Consistency in Distributed Systems." Guide to Reliable Distributed Systems. Springer London, 2012. 457-470.
[64] Bermbach, David, and Stefan Tai. "Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior." Proceedings of the 6th Workshop on Middleware for Service Oriented Computing. ACM, 2011.

[65] Zhou, Yuanyuan, et al. "Relaxed consistency and coherence granularity in DSM systems: A performance evaluation." ACM SIGPLAN Notices. Vol. 32. No. 7. ACM, 1997.
[66] Adve, Sarita V., and Kourosh Gharachorloo. "Shared memory consistency models: A tutorial." computer 29.12 (1996): 66-76.
[67] Serious cloud failures and disasters of 2011.http://www.cloudways.com/blog/cloud-failures-disastersof-2011/.
[68] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, "Blueprint for the intercloud - protocols and formats for cloud computing interoperability," Internet and Web Applications and Services, International Conference on, vol. 0, pp. 328–336, 2009.
[69] R. G. Dimakis, P. B. Godfrey, Y. Wu, M. O. Wainwright, and K. Ramch, "Network coding for distributed storage systems," in In Proc. of IEEE INFOCOM, 2007.
[70] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," SIGCOMM Comput. Commun. Rev.,vol. 27, no. 2, pp. 24–36, Apr. 1997. [Online]. Available:http://doi.acm.org/10.1145/263876.263881
[71] H. P. Anvin. The mathematics of raid-6.http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf.
[72] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," Peer-to-Peer Systems, pp. 328–337, 2002.
[73] M. Vrable, S. Savage, and G. M. Voelker, "Cumulus: Filesystem backup to the cloud," Trans. Storage, vol. 5, no. 4, pp. 14:1–14:28, Dec. 2009. [Online]. Available:http://doi.acm.org/10.1145/1629080.1629084
[74] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: waitfree coordination for internet-scale systems," in Proceedings of the 2010 USENIX conference on USENIX annual technical conference, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.
[75] L. Lamport, "Paxos made simple," ACM SIGACT News, vol. 32, no. 4, pp. 18–25, 2001.
[76] N. A. Lynch, Distributed algorithms. Morgan Kaufmann, 1996.
[77] A. Bessani, M. Correia, B. Quaresma, F. Andr´e, and P. Sousa, "Depsky: dependable and secure storage in a cloud-of-clouds," in Proceedings of the sixth conference on Computer systems, ser. EuroSys' 11. New York, NY, USA: ACM, 2011, pp. 31–46.
[78] W. Vogels, "Eventually consistent," Communications of the ACM, vol. 52, no. 1, pp. 40–44, 2009.
[79] D. Bermbach and S. Tai, "Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior," in Proceedings of the 6th Workshop on Middleware for Service Oriented Computing. ACM, 2011, p. 1.
[80] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on. IEEE, 2003, pp. 522–529.