DSFS: A DATA STORAGE FACILITATING SERVICE FOR MAXIMIZING SECURITY, AVAILABILITY, PERFORMANCE, AND CUSTOMIZABILITY

A Thesis
Presented to
The Academic Faculty

By

Kyle Bilbray

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
December 2014

DSFS: A DATA STORAGE FACILITATING SERVICE FOR MAXIMIZING SECURITY, AVAILABILITY, PERFORMANCE, AND CUSTOMIZABILITY

Approved by:

Dr. Douglas Blough, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. George Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: August 2014

To Meema and Grandpa,

ACKNOWLEDGMENTS

I would like to thank Dr. Blough for the past year and hopefully next few years of guidance, as well as sparking my interest in the possibilities of this field. And I am extremely grateful to Dr. Riley and Dr. Yalamanchili for being part of my thesis committee.

I would also like to thank my family and friends for all of their support. Brian gave me so much help on getting the whole system set up and running. My mom and sisters, Haley and Alyssa, were always there for me when I needed it. My Uncle John never failed to make me laugh when I needed a pick-me-up, and Meema learned how to text just so I could talk to her all the time (that's how I see it).

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The objective of this thesis is to study methods for the flexible and secure storage of sensitive data in an unaltered cloud. While current cloud storage providers make guarantees on the availability and security of data once it enters their domain, clients are not given any options for customization [1, 2]. All availability and security measures, along with any resulting performance hits, are applied to all requests, regardless of the data's sensitivity or client's wishes. In addition, once a client's data enters the cloud, it becomes vulnerable to different types of attacks. Other cloud users may access or disrupt the availability of their peers' data, and cloud providers cannot protect from themselves in the event of a malicious administrator or government directive.

Current solutions use combinations of known encoding schemes and encryption techniques to provide confidentiality from peers and sometimes the cloud service provider, but its an all-or-nothing model [3, 4, 5]. A client either uses the security methods of their system, or does not, regardless of whether the client's data needs more or less protection and availability.

Our approach, referred to as the Data Storage Facilitating Service (DSFS), involves providing a basic set of proven protection schemes with configurable parameters that encode input data into a number of fragments and intelligently scatters them across the target cloud. A client may choose the encoding scheme most appropriate for the sensitivity of their data. If none of the supported schemes are sufficient for the client's needs or the client has their own custom encoding, DSFS can accept already encoded fragments and perform secure placement.

Evaluation of our prototype service demonstrates clear trade-offs in performance between the different levels of security encoding provides, allowing clients to choose how much the importance of their data is worth. This amount of flexibility is unique to DSFS

and turns it into more of a secure storage facilitator that can help clients as much or as little as required. We also see a significant effect on overhead from the service's location relative to its cloud when we compare performances of our own setup with a commercial cloud service.

The remainder of this thesis is structured as follows. Chapter 2 reviews past solutions to this problem as well as any related work. Chapter 3 discusses our design goals and motivations for this research. Chapter 4 provides an overview of the overall system our service will be a part of, while Chapter 5 goes into more detail on the architecture of DSFS itself. Chapter 6 details its implementation. Chapter 7 evaluates the performance of our implementation over various inputs and encoding methods. Finally, Chapter 8 recaps the results of our research and concludes the paper.

CHAPTER 2

ORIGIN AND HISTORY OF THE PROBLEM

Current cloud storage systems allow their clients to upload their data to off-site servers. Clients pay for however much data they choose to store, and need not worry about the unexpected operating expenses and issues that come with maintaining storage centers. A typical cloud provider will provide their customers certain guarantees as to the condition of any stored data as well as some limited number of options on how their data is stored.

Data availability is one of the most important characteristics of cloud storage. For example, Amazon S3 is a popular cloud storage service that guarantees 99.99% availability, as well 99.999999999% "durability" of any stored object over a given year [6], meaning 1 out of 10,000 objects is expected to be lost every 10,000,000 years. These two guarantees are achieved through the cloud-specific use of encoding primitives such as replication, which can also provide faster data access. However, simply using replication can present several drawbacks: storing multiple exact copies of the original data naturally requires more overall storage space, having the entire object stored in each location prevents any possibility of parallel data retrieval, and more seriously with sensitive data, when there are more copies of the data, there are more points of attacks and therefore an increased risk of compromise.

Furthermore, the cloud is more or less presented as a black box. Users give their data up to the cloud and rely upon SLAs and any internal policies to maintain their data privacy. That is not to say that cloud providers completely neglect security. Using S3 as an example of a typical cloud storage service, it offers several access control mechanisms for stored data, ranging from user level permissions on individual objects to query string authentication [6]. However, all this does is protect data from other users accessing the data through standard sanctioned channels. The cloud provider essentially has unfettered access to client data, leading to a reluctance of businesses to migrate their more private or sensitive data to

the cloud.

Data availability and security are achieved through the primitives described in section 2.1. Section 2.2 outlines the unique solutions that use these primitives and other innovative techniques.

2.1 Data Encoding Primitives

The three main techniques used as building blocks for developing security schemes are secret sharing, erasure coding, and replication. Secret sharing is designed primarily for data security, replication has no inherent security, but is the simplest way to improve data availability, and erasure coding is an imperfect form of secret sharing that can have more efficient availability than replication while still maintaining some basic privacy. Finally, protected memory is an extremely important low-level primitive, with its ability to allocate chunks of memory without fear of information leakage to even the cloud administrator.

2.1.1 Secret Sharing

Secret sharing is a type of encoding scheme in which an object, or secret, is transformed into a predefined number of shares, which are distributed amongst some number of participants. The object can only be fully reconstructed when a required number of the shares have been gathered. This approach is generally referred to as a (t,n)-threshold scheme, where any t of the n shares can reconstruct the secret. Secret sharing is a very useful mechanism in secure cloud storage, as it does not require any key maintenance or modification of the storage media. If the shares are placed correctly, it makes recovery of the secret by any attacker extraordinarily difficult.

There are, from a high level, two different types of secret sharing, sometimes called perfect and imperfect secret sharing:

Perfect: any number of shares obtained, less than the threshold, reveal no information about the secret. In other words, even with t-1 shares, the secret could still be anything in its entire value space. Shamirs and XOR secret sharing are two examples of

this type.

Imperfect: the threshold number of shares are still required to fully reconstruct the secret, but some information about the secret may be revealed with fewer shares. For example, if the secret was a string of length 10 and there were 5 participants using a (5,5) threshold scheme, each share could have 2 of the original and 8 null characters. Access to any of the shares would reveal at least these 2 correct characters, thus shrinking the value space of the secret. Erasure coding is a popular method that uses imperfect secret sharing, more for the reliability than the security it provides.

In our research, when we refer to secret sharing, we mean perfect secret sharing. Any use of imperfect secret sharing will instead be called by its more specific name, e.g. erasure coding.

Shamir's

Shamirs secret sharing is a type of secret sharing developed by Adi Shamir, based on the fact that, given a set of t points, there is only one unique polynomial of degree $t-1$ that will fit them all [7]. To encode a secret, a $t-1$ degree polynomial is generated where the first coefficient is the secret, and the rest are chosen at random. Each of the n total shares is made up of one unique point on the curve. When at least t of these points have been recovered, there is enough information to reconstruct the polynomial, and therefore the secret. With less than t points, an infinite number of polynomials could be found to fit.

The security of this scheme does depend on its implementation, however. With $t-1$ points, the possible values for the tth point can be reduced significantly such that the secret can be recovered with a manageable amount of testing. By using the appropriate finite-field arithmetic, this problem is corrected, within reason.

XOR

When $t = n$ in a (t, n)-threshold scheme, much less computationally expensive secret sharing methods like XOR secret sharing become possible.

- Given a secret, Z, of length m bytes, and the total number of shares to be created, n.
- Create $n-1$ random byte arrays, each also of length m, as the shares $s_1, s_2, ...s_{n-1}$
- The nth share is calculated as $s_n = Z \oplus s_1 \oplus s_2 \oplus ... \oplus s_{n-1}$, where $\oplus$ is the bitwise XOR operator.

When all n shares have been recovered, the secret can be reconstructed by XORing them all together again: $Z = s_1 \oplus s_2 \oplus ... \oplus s_n$.

XOR secret sharing is computationally much less expensive than Shamir's, as a bitwise XOR is its only operation. However, it requires that all n shares be recovered in order to reconstruct the secret. Manual replication or erasure coding of individual shares can compensate for this lack of inherent reliability.

2.1.2 Reed-Solomon Erasure Coding

Erasure coding is most commonly used for error correction, where it can take a message of length k symbols, and add on $n-k$ symbols, such that the message can be recovered from a subset of the symbols. Reed-Solomon erasure codes satisfy this property more specifically, correcting up to $n-k$ known missing symbols, as well as identifying up to $n-k$ errors or correcting up to $\frac{n-k}{2}$ unknown errors. By taking advantage of these properties, the Reed-Solomon codes can be used as more of an imperfect secret sharing scheme with threshold, k, and total shares, n [8].

Given input data of size m bytes, threshold, k, and total number of shares, n, the data is divided up into k blocks, each of size $\frac{m}{k}$. The remaining $n-k$ blocks of the same size are generated such that they meet the above properties. These blocks are the shares used in the secret sharing scheme, and as a result of the Reed-Solomon code properties, any k of them are sufficient to recreate the original data. Only $\frac{mn}{k}$ storage is required, while still allowing $n-k$ shares to be lost. When compared to replication with c copies, $n-k \geq c$, the same or better reliability as replication can be achieved with less overall required space, an important property when it comes to cloud storage.

2.1.3 Replication

The simplest way to provide reliability and availability of data is replication. It has been implemented on many cloud platforms as a way to guarantee some level of understandable reliability [9, 10]. The data is simply replicated a certain number of times, and each copy is stored in a different location. Locations are usually selected such that all copies would not likely be made inaccessible at once, and possibly even to decrease the time needed for a client to access the data. Replication requires an amount of storage space directly proportional to the number of replicas created, which could become a problem as the number of objects being stored in the cloud continues to increase, all with the same amount of replication.

2.1.4 Protected memory

This type of virtualized memory is protected from information leakage or attack, even from the cloud administrator. Data in the unprotected memory of a virtual machine running on the cloud is accessible to the cloud administrator, which leaves sensitive data vulnerable to theft as it passes through, even if the data is encrypted and the plaintext destroyed instantly. This topic has been the subject of recent research in virtualization and secure execution [11, 12], and will not be the focus of this thesis.

2.2 Secure, Distributed Storage

Each secure, distributed storage technique has a different approach that targets a particular unique goal. Tahoe and Farsite both attempt to create distributed file systems that are resistant to Byzantine faults, while still maintaining confidentiality. Subbiah explores security through a combination of encrypting data, secret sharing the encryption key, and replicating shares [13], then refines the concept in After examining the most common solutions, we were able to find a set of meaningful properties to help determine their general effectiveness.

2.2.1 Farsite

Farsite is a secure, scalable file system that logically functions as a centralized file server, but is physically distributed among a set of untrusted computers [14]. While Farsite was not designed to be run over the cloud, and would certainly not be viable without serious modifications overlaid over current clouds, it presents a comprehensive system with features that are still applicable.

Aside from its main target being academic and corporate workloads only, the only serious issue in Farsite is the lack of client security customization and efficient storage methods. Clients are not given any mechanisms to decide how their data is stored, and only replication is used for data reliability and availability. Security is maintained with a per-file access control list, metadata encryption, and a Merkle hash tree computed over the file data for integrity. The design of the Farsite system is optimal in the right situations, but without any flexibility, less-sensitive data storage is more expensive than necessary, and extremely confidential data may not be stored securely enough.

2.2.2 DepSky

A group of researchers designed a system called *DepSky* relatively recently at the University of Lisbon to try and improve the availability, integrity, and confidentiality of information stored in the cloud [3]. Their idea was to create a kind of "Cloud-of-Clouds" by encrypting, encoding, and replicating the data across multiple diverse clouds. The DepSky system could use any number of currently operating commercial clouds, with the assumption that no data stored on one cloud could unknowingly transfer to another cloud and that an employee of one cloud provider with complete access to data on one cloud would not normally have the same access on another cloud. DepSky acts like a standard cloud, but one in which each commercial cloud in the system is like a data center in which DepSky may choose to store data. Because of this, it also implements many of the usual features found in a cloud, such as access control, consistency, and versioning.

Unfortunately, while DepSky does achieve its goal of increased availability, integrity,

and confidentiality, several issues do emerge. Because the system uses diverse clouds with most likely little connection, the interfaces of each cloud will be unique. DepSky models each cloud as a "passive storage entity" that supports five simple operations, but custom code must be written to translate each DepSky operation to its cloud equivalent, or equivalents. The format of the data accepted by each cloud is especially important.

The DepSky team ran a number of tests on two of their implementations using 4 clouds, and compared the results to the same tests run on single clouds. The cost of using DepSky to store data across 4 clouds was about 2-4 times that of storing on one cloud, depending on which DepSky implementation was used, which is understandable considering that it stored 2-4 times the data size.

2.2.3 GridSharing

GridSharing is a data storage service designed to maintain security and fault tolerance in a collaborative work environment [15]. The motivation behind its creation is the issues with perfect secret sharing used in traditional security and reliability models. The excessive overhead produced by these types of schemes are a significant deterrent to their practical use.

The GridSharing approach is resistant to three types of server faults: crash (stops all computations and communications), leakage (reveals its data to an adversary), and byzantine (does not follow protoccol and may also reveal its data to an adversary). These three faults encompass the required security and availability properties. The system architecture involves a combination of XOR secret sharing and replication mechanisms that are comparable to standard encryption schemes in performance. Servers are arranged in a logical grid, where secret sharing is done across rows and shares are replicated along rows. In this model, a certain number of servers can suffer from each type of server fault. It has the added benefit that its dimensions can be varied to personalize the trade-offs between performance and security.

After performance testing, a number of conclusions were deduced. GridSharing significantly outperforms perfect sharing but is still 6-8 times slower than Rijndael encryptions and decryption. However, the framework provides much better fault tolerance to leakage-only and crash faults that make this overhead acceptable.

2.2.4 Tahoe

A filesystem called Tahoe was developed for AllMyData.com as a method of secure, distributed storage. It uses a combination of features to increase reliability and confidentiality of data, even in the worst case situations [16]. Data reliability is promoted using erasure coding, rather than direct data replication. Erasure coding allows the data to be distributed across any number of servers, such that only user-specified fraction of these servers are needed to recover the data. The amount of reliability is entirely dependent upon what the user requires and allows the user to pay for only the amount he/she wants.

Confidentiality is handled using a capability access control model. Files are encrypted, then erasure coded before being distributed throughout the system. Verify, read-only, and read-write capabilities are created from the various encryption keys and hashes throughout the process. In this way, only the holder of a read-enabled capability can decrypt the data, and only the holder of a write-enabled capability can change the data. At the same time, because the administrator of the Tahoe filesystem may need to verify the correctness of stored data, the verify capability can do just that. As long as the read-only and read-write capabilities are kept secret, the client need not trust the system. These capabilities are further enabled to diminish: a read-write capability may be diminished to a read-only capability, and a read-only capability may be diminished to a verify capability, but nothing else.

The security and reliability of the Tahoe filesystem appears to be legitimate enough, but there are no performance evaluations given or even mentioned in its paper. The lack of any experimental results or proof of correctness prevent the Tahoe filesystem from being an effective method of secure storage.

2.2.5 Belisarius

Belisarius is a unique Byzantine fault-tolerant storage service that does not sacrifice confidentiality [4]. Using the encoding primitive, Shamir's secret sharing, it is able to obfuscate stored data without the usual required key management systems [17]. This greatly simplifies the overall design and improves performance, relative to other BFT services [18, 19].

Because Belisarius offers no control over how data is encoded and stored, it is able to take advantage of Shamir's secret sharing inherent additive homomorphism. This feature could be particularly useful in situations that repeatedly increment a number, but need to keep the actual value hidden.

The main shortcomings of Belisarius are the lack of customization, and the required changes for implementation. All of the client's data may not have the same sensitivity, so not being able to change the security for each object and having to still suffer the additional overheads during storage and retrieval could be discouraging. A large part of the necessary request computations are also offloaded to client, which may be an issue when all the client wants to do is get its data. The rest of the request processing is done at each server, meaning the servers themselves must be modified to fit into the system.

CHAPTER 3

BACKGROUND

3.1 Design Goals

Our research has four main goals:

Security: In addition to all of the usual protections provided by a cloud provider, stored data should also be safe from the cloud provider itself. Even if a piece of data uploaded to a cloud provider is compromised through a remote exploit, an insider attack, or government requisition, the original data should remain protected.

Availability: Availability of data should remain at least as high as that of current cloud storage services. Data availability will be even more important when the objects being stored are private, sensitive, or in some way mission-critical for the client. One of the primary reasons for a client to store this information in the cloud is the data availability that can only be economically achieved in a large-scale system devoted to data storage.

Performance: The effect of any enacted security and availability improvements should not have a significant impact upon performance when compared to current cloud storage services. There will likely be a small static overhead independent of read or written data size due to necessary meta-data retrieval, and as input data size increases, some encoding schemes may become prohibitively expensive, but within a certain input data size range, overall latencies should be similar. Throughput will also be a central concern. Because simultaneous uploads are a common occurrence in cloud storage, our service should have high throughput to handle concurrent requests.

Customization: Clients should have control over how their data is stored. If they encode an object themselves, they must be able to specify fragment placement parameters

to meet the requirements of their custom encoding. Clients may also have region or availability zone preferences, unrelated to the encoding, that should be applied in addition to the encoding requirements.

3.2 Motivations

Cloud storage solves many of the problems inherent with maintaining and securing local storage. Unfortunately, with such a widely distributed architecture and widespread user base, several types of failures are introduced, along with the increased possibility of attack from a variety of sources. The specific kinds of failures and sources of attack that were the main motivation behind this research are described in more detail below.

3.2.1 Failures

Failures can be assigned to one of three main categories.

Localized : The most common of problems is the failure of an individual hard disk or server where data is being stored. Disks in cloud-type systems with large server populations have an Annualized Failure Rate of about 8% after their first year [20]. Data may be temporarily unavailable while the failed disk is replaced and its data restored from a backup, or in the case of insufficient preparation, it could be permanently lost. Any kind or size of storage system must worry about this problem and it requires manual intervention to correct. Having the cloud perform this maintenance for you is another incentive, as this type of small-scale failure is the simplest to prevent and is handled by most cloud providers.

Network : While the failure of a hard disk may cause the loss or reordering of data, sometimes a switch or router could fail, leading to a network outage. The data is still there where it should be, but its connection to the external network is down and the data is inaccessible, violating the cloud availability guarantee.

Widespread : In the worst kind of situation, an entire datacenter or group of servers could

all fail at once as the result of a natural disaster or other large-scale event. For companies maintaining their own storage hardware in a single facility, this can be something they cannot possibly prevent or recover from.

3.2.2 Attacks

Attacks originate from three different sources.

External : These type of attacks are unique in that they do not come through the sanctioned cloud infrastructure. For instance, a zero-day exploit may temporarily take down the entire system, an attacker could gain physical access to a server and attach an infected USB drive, or the client's authentication information or data could be compromised through snooping or a client-side attack. In addition, the architecture of the cloud itself may allow for data theft or denial of service attacks from peers that are storing and accessing data from the same physical resources.

Administrative : Specific employees of the cloud provider are given administrative access to the cloud architecture and subset of stored client data [1]. These administrators are often not permitted to access any private data, except in extenuating circumstances. However, there are those employees with malicious intentions who may access sensitive without the cloud provider's knowledge. There are also situations where an administrator could be required by legal ruling to allow government access. Attacks of this nature are difficult to prevent, or even notice, from the limited scope of a client.

Service-level : When cloud users store their data in the cloud, they introduce new software vulnerabilities into the system. Attacks may stem from security holes in the client's own applications, or even from malicious employees within the client's organization, and give others unauthorized access to stored data. These types of attacks can be minimized, but not completely eliminated, through basic security precautions like encrypted network connections, key protection, and employee validation.

CHAPTER 4

SYSTEM OVERVIEW

The overall system is made up of three parts: the clients, the DSFS, and the cloud. We will first describe the structure of the cloud before characterizing the system as a whole, as this knowledge is essential to understanding how the pieces fit together. Then we will illustrate several use cases for DSFS that would not be possible without our service.

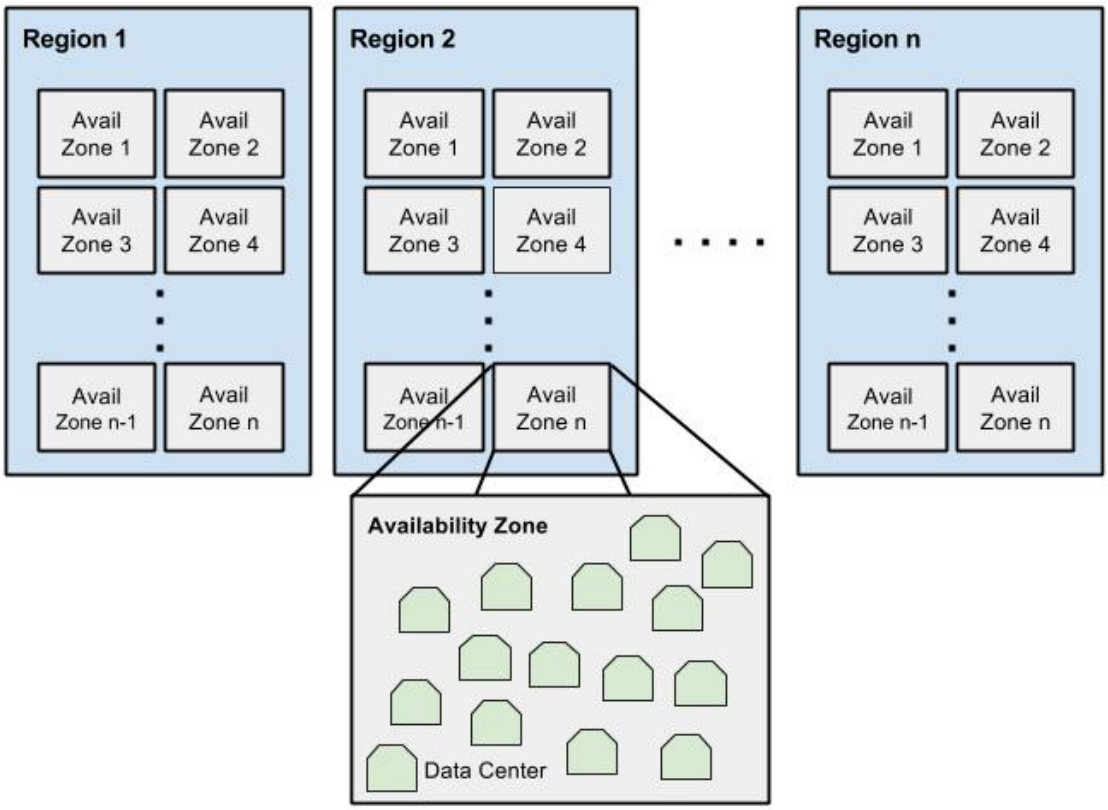


Figure 1: Three-tiered cloud infrastructure.

4.1 Cloud Infrastructure Model

The cloud is modelled after a standardized three-tier structure, illustrated in Figure 1, commonly found in commercial implementations like Amazon S3 [1].

Region : a zone designed specifically for data isolation. A region is made up of some number of availability zones that fall under one administrative domain; in other words, an administrator that can access one region, can access only that one region. Data that enters a region is further guaranteed not to move to another region without being specifically directed by the data owner.

Availability zone : a zone made up of multiple data centers. Availability zones are typically located geographically such that a large disaster destroying data in one zone will not affect data in other zones. Data that is stored in the parent region of some availability zones can then be replicated across these availability zones in order to meet availability requirements.

Data center : a facility housing multiple servers. Each data center typically has its own redundant power sources, network connections, environment controls, and security measures. A data center differs from an availability zone primarily in size, but all of its servers are also guaranteed to be in the same physical location.

4.2 System Model

DSFS is designed for clients trying to store sensitive data in the cloud. There is not one universally absolute method for secure cloud storage. Some clients may have limited resources and want to have someone else take care of the security for them, while others do not trust a service they did not implement themselves with any confidential information. With such a wide variety of secure approaches, DSFS implements a flexible service that can have as much or as little contribution to the process as the client desires. The service runs continuously and handles any incoming client request at any time. It can interface with any cloud that has the first two properties described in section 4.1.

DSFS supports multiple encoding schemes with customizable parameters. A unique data placement algorithm decides where in the cloud each of the encoded fragments should be stored such that security is maximized relative to the chosen encoding or any client

location constraints. If a client has their own encoding method, or they simply just don't trust the service with their un-secured data, DSFS can perform only the data placement based on what encoding parameters the client gives them. This way, clients can be sure that, while their encoding scheme may be secure, the placement of the output doesn't create any new vulnerabilities.

4.3 Use Cases

The following sections briefly cover several use cases for our service that would not be feasible with manual cloud storage alone.

4.3.1 Encryption Key Storage

Some of the most sensitive data that requires high availability for multiple users, but at the same time, maximum security, is cryptographic metadata, i.e. encryption keys. A simple solution to an untrustworthy cloud or third-party storage service is to encrypt all data before upload. There are plenty of unbroken cryptographic algorithms today that would prevent any attacker without the correct key from recovering the plaintext [21, 22]. A cloud administrator that had access to his region's data would see only random byte arrays, rather than anything of meaning.

However, though all the benefits of keeping important data securely in cloud are still kept, without any serious chance of information leakage, all related cryptographic metadata must be maintained by the user. Any encryption keys, block locations, initialization vectors, etc., necessary for decrypting a client's data can be compromised as the result of simple localized failures or attacks, jeopardizing the security of or even access to the original data. When sensitive data is shared among multiple users, metadata management becomes even more difficult. For example, if users share a common key for data encryption and decryption, this key must be safely distributed to each user when they are granted access and securely invalidated once their permission has been revoked.

DSFS will encode the important cryptographic metadata into fragments using secret

sharing, and distribute them across the cloud in such a way that, in addition to an attacker needing to discover and illicitly pull multiple of the original object's encoded fragments, a single cloud administrator would not be able to decipher any part of the object's cryptographic metadata with only the information found in their region. Storage and protection costs are offloaded to the cloud without relinquishing confidentiality.

4.3.2 Same Availability with Less Storage

Cloud providers do guarantee a certain high level of availability for any data stored in their system as an incentive for using their products [6]. This is typically done through replication within the data's home region. Sometimes, this replication is not sufficient, from the financial consequences of multiplied storage, or if, from the perpective of the client, there is the possibility of the region being compromised or failing all at once. The replication itself is inherently less safe, as it makes multiple complete copies of the data and stores them in multiple locations. DSFS provides the option to instead erasure code and securely store a client's data.

Erasure coding (see section 2.1.2) has the same availability benefits as replication, with better storage efficiency. Say, for example, a client storing data of size s desires a loss tolerance of $n_r - 1$, where n_r is some positive integer. When using replication, this requires n_r replicas, taking up a total of $n_r s$ storage. With (k, n_e) erasure coding, n_e fragments are required, where each fragment has a size of $\frac{s}{k}$. In order to maintain the same availability, k and n_e must be chosen such that $n_e - k = n_r - 1$. This produces a total storage requirement

of $n_e \frac{s}{k}$, resulting in erasure coding taking up less storage than replication as long as $k > 1$.

$$\textbf{Erasure coding} < \textbf{Replication}$$

$$n_e \frac{s}{k} < n_r s$$

$$(n_r + k - 1)\frac{s}{k} < n_r s$$

$$n_r + k - 1 < n_r k$$

$$n_r - 1 < k(n_r - 1)$$

$$1 < k$$

A single fragment's compromise will only reveal $\frac{1}{k}$ of the original data, and k fragments will be required to recover the data in its entirety. For data that is indistinguishable from random bytes, like encryption keys and hashes, bits and pieces of information will mean little except to shrink an attacker's testing space. The improved storage efficiency will also cut down on overall costs as fewer bytes will be stored.

4.3.3 Digital Information Laws

With the expansion of many companies to a global scale, the privacy of online data is becoming a prominent issue. Whether a company has its own servers in a different country than where it does business, or is using a cloud service with data centers worldwide and has the option to choose which it prefers, information security is a primary concern. Digital information protection laws vary across countries and continents, and will continue to change in the future.

For example, in the United States, most cloud data falls under the jurisdiction of the Stored Communications Act (SCA) [23]. Though the Fourth Amendment protects U.S. citizens from unreasonable search and seizures, the Third-Party Doctrine establishes that revealing information to a third party (i.e. a cloud provider) relinquishes this right and designates this information as a "stored communication" [24, 25]. Under the SCA, only a subpoena is needed to force an ISP to divulge a client's private information with prior

notice.

The European Union has more of a broad view on data protection with the Data Protection Directive (DPD) [26]. While the U.S. has an ad doc approach, creating laws in response to specific issues, the DPD has a more general definition of "personal data" and ensures that its seven directive principles are applied.

The rules put in place by the country housing the physical server may not even be the only concern, as recent legislation and court cases are attempting to apply one country's laws to data referencing or belonging to their citizens, but stored in a different country [27][28]. However, the effectiveness of this type of action will depend on existing mutual legal assistance treaties (MLATs) between the two countries. A client using DSFS may choose an encoding scheme such that a government request for the encoded fragments in their region would not give them a sufficient number for data recovery. A client could also specify preferences for where data should be stored, or how encoded data should be distributed. If privacy laws for the locations of certain regions' data centers are more conducive to data privacy, a client may prefer to have all data fragments stored in those regions. As the interpretation of new and existing privacy laws change, or new and existing regions move around, these preferences may change and with them, how a client needs their data stored.

CHAPTER 5

SERVICE ARCHITECTURE

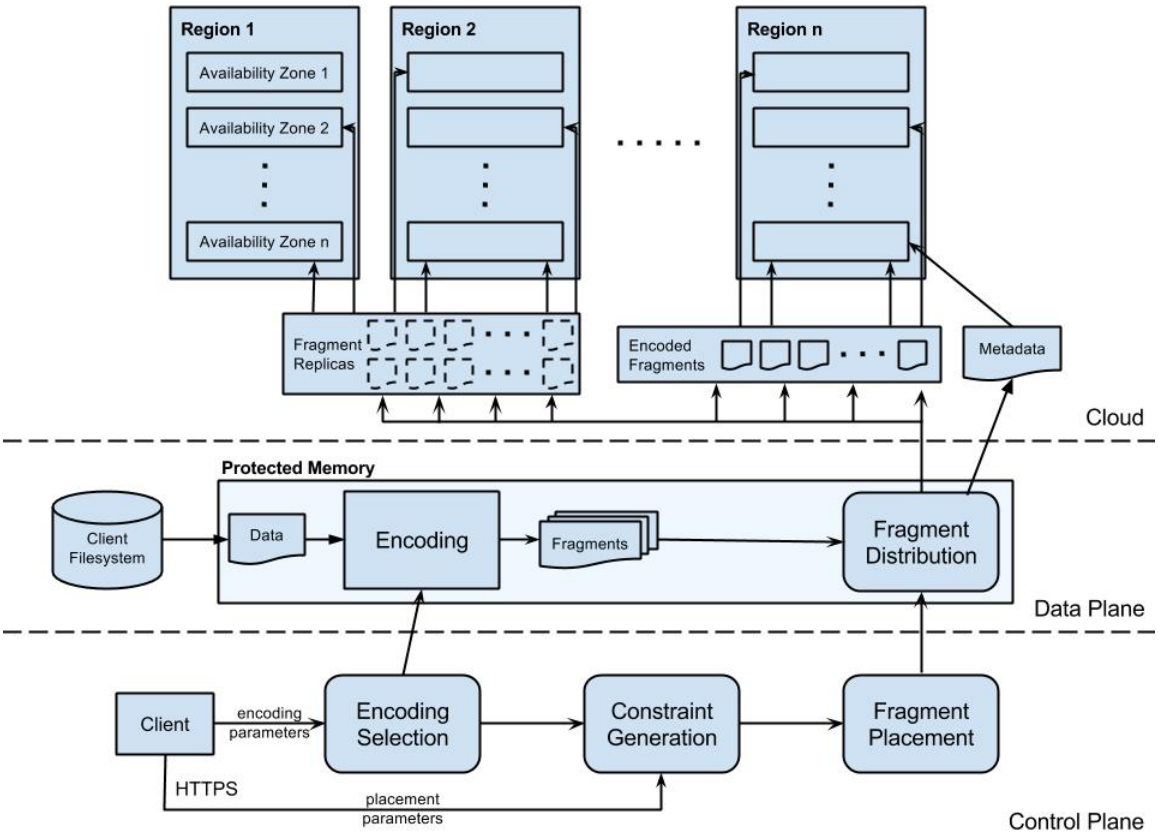


Figure 2: Service architecture.

5.1 Setup Options

DSFS is designed to be run as a service on one, or distributed across several, webservers, allowing maximum flexibility for the client. Depending on the basic security concerns of the user, our service can be run with three different approaches: *Local*, *Third-Party*, and *Single*, each with different trade-offs between resource requirements, performance, and transitional privacy.

5.1.1 Local

When security of the uploaded data is of the utmost importance, DSFS can be run on the client side. Network access, resource usage, metadata storage, and any other type of physical configuration is under the control of the user. Unfortunately, the inherent benefits of storing data and running scripts in the cloud are lost. Unless all, or the majority of all, service requests are guaranteed to come from nearby, the performance of external requests will degrade with the additional overhead of a network not designed for concurrent global accesses. In the worst case, essential files like encryption keys and data locations could be permanently lost after a localized failure, or compromised due to a malicious employee or successful attack.

5.1.2 Third-Party

The more convenient option is to run DSFS on a third-party, that is, a cloud or server not connected or related to the data's final destination cloud. The availability, security, and customization normally provided by the cloud will be unchanged, and most likely will be a significant improvement on anything the client could do alone. It does seem rather paradoxical if a client's primary objective is to protect their sensitive data from the cloud by encoding it before upload, and then for them to do the actual encoding in a different, but equally untrustworthy, cloud or server. However, there is a technique called protected memory (described in section 2.1.4) that prevents any information leakage to the administrator or peers. In the unfortunate event that this memory is somehow compromised, the attacker will only have whatever small fragments of memory it was able to recover, and would still have to illicitly obtain the encoded data from the cloud that houses it. This mode will suffer from having back-and-forth communication between multiple entities in order to entirely satisfy a request.

5.1.3 Single

For the highest cost efficiency and best performance, DSFS can be run on the same cloud service as the data it is securing. Some network communications may be within the cloud's network if the service is running in the same region as stored data fragments, so there will not necessarily be as many trips with variable latency through the Internet during data retrieval and upload. But clearly, the likelihood of a cloud administrator attack occurring is much higher, and the consequences of such a successful attack happening are much more severe when a single administrator could have access to both the encoded data and the software doing the encoding. Protected memory (expanded upon in section 2.1.4) should prevent unauthorized interference, but there are possible vulnerabilities and still the performance hit of having to somehow protect memory.

5.2 Control Plane

The client first communicates their preferences to DSFS through the control plane. These may include options like encoding method, preferred regions, security level, and reference key. The control plane will take all these constraints, along with any public knowledge like the cloud infrastructure or secondary region factors, into consideration as it decides how the user's data will be encoded and where each encoded value will be placed in the cloud. Whatever final parameters it generates are passed on to the data plane.

5.2.1 Client Input

Required parameters:

- *Request type*: whether the request is a put, get, delete, or list
- *Reference key*: the specific key used to reference this data, either now or in the future
- *Data*: the data itself if it is being stored in this request

Optional parameters (default):

- *Encoding method (None)*: how to encode the data before storage
- *Encoding parameters (N/A)*: the parameters for encoding the data
- *Replicas (3)*: how many replicas of the encoded data to make and store. This in itself is unique, as most storage services have a static amount of replication.

As DSFS evolves, we hope for the client to be able to specify region priorities, do their own custom encoding/placement and have our service perform verification, or possibly present their preferences in a simplified High, Medium, Low, manner.

5.2.2 Fragment Placement

The placement of data fragments is one of the foundations to DSFS's success. Security, availability, and performance all rely upon where fragments are placed in order to maintain their effectiveness.

5.2.2.1 Security

Many encoding methods, like secret sharing, encode input data into fragments. A threshold, k, of these fragments are required to reconstruct the secret. Because all fragments are being stored with the same cloud provider, they must be distributed across the cloud regions (4.1) to prevent certain types of attacks. The two inherent properties of a region are that a cloud administrator has access to only one region, and data that enters a region is guaranteed not to leave the region. By never storing $\geq k$ fragments in a region, one cloud administrator will never have access to enough fragments to reconstruct the secret. This type of placement can be seen in Fig. 3, which shows an object stored using a Secret-Sharing(3,5) scheme. No more than 2 fragments are ever in one region, which satisfies the encoding requirements.

Other clients may be concerned about their data privacy with respect to government. With probable cause and prior notice, or sometimes not even that, some governments can require a cloud provider to produce a client's data if it is physically located within their domain. The limitations on when this can happen, and what must be revealed vary by country,

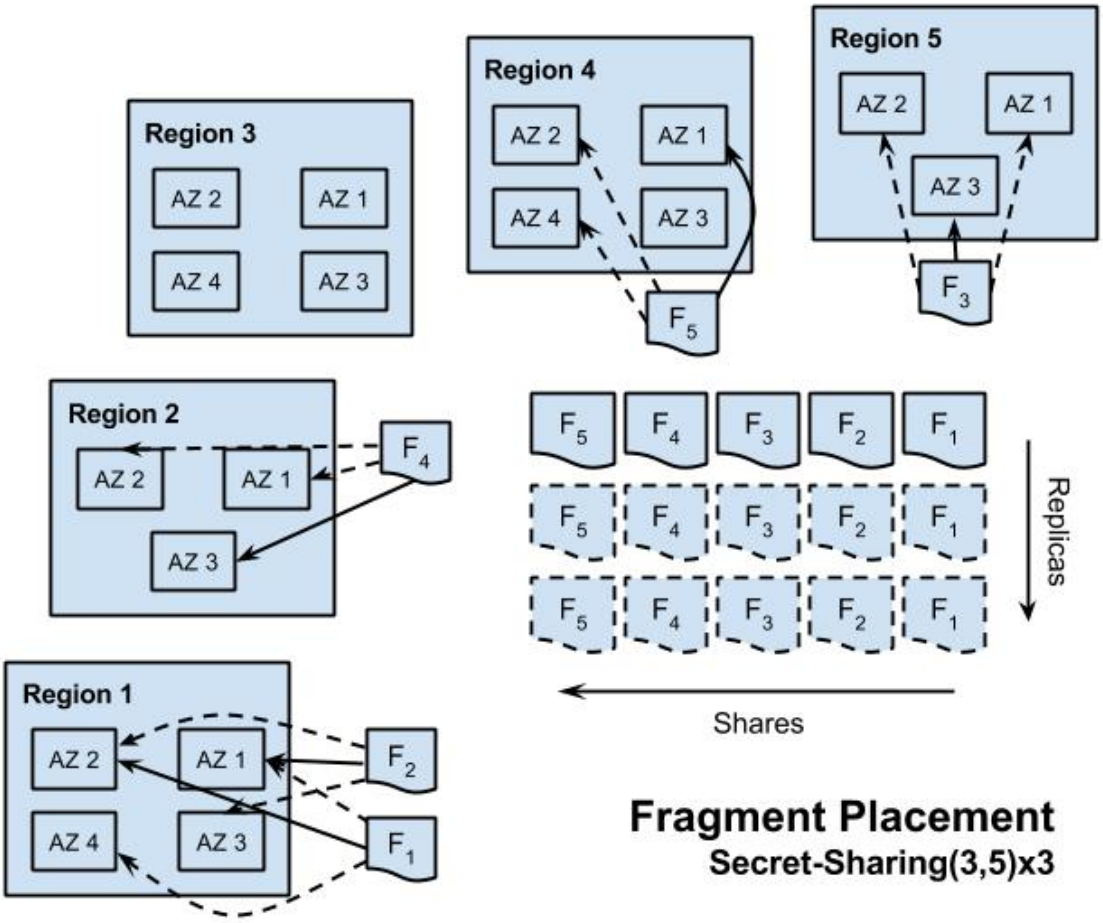


Figure 3: Fragment placement.

which may prompt clients to prefer storing their data in countries with more stringent protections. If the client is able to give region preferences, our service's fragment placement algorithm can take into account these preferences, and if some cannot be satisfied (i.e. prohibited regions), placement will fail.

5.2.2.2 Availability

Data availability is dependent upon the data's placement. Even when data is replicated, if all copies are placed in the same server or cluster of servers, they could all potentially be lost at once for the same reason. One of our primitives, the availability zone (4.1), is used to reduce the likelihood of stored data being permanently lost. Availability zones are located geographically such that multiple zones will not fail at the same time as the result

of the same problem.

Whether data is replicated or erasure coded, data fragments in one region are distributed across the availability zones within that region. When a threshold, k, out of n fragments are required to recover the original data, less than $n - k$ of them are stored in one availability zone. If a zone fails, there will still be enough fragments in other zones of the region to reconstruct the data. In the case of replication with n copies, no more than $n - 1$ copies will be in one zone, though less than one copy is placed in each zone when possible. Fig. 3 shows such an example where objects are replicated with $n = 3$. With this particular cloud setup and replication value, there are enough availability zones in each region to store each fragment's copy in its own zone, so it is clear that the loss of one availability zone in any given region would not be catastrophic for the fragments it housed.

5.2.2.3 Performance

Cloud performance during both storage and retrieval can be highly variable, depending on where data is coming from. Our service can take into account past latency, throughput, and dependability of the different regions and availability zones in choosing the data's destination. However, this will be the lowest priority factor, and all other encoding parameters will take precedence.

5.3 Data Plane

The data itself is submitted by the client to the DSFS instance, and transparently transferred to the data plane along with the parameters generated by the control plane. Any nontrivial or potentially sensitive data is allocated in protected memory (see section 2.1.4) from the moment of its creation to prevent eavesdropping by any peers or administrators. The data plane performs the encoding operations on the data it receives based on what the control plane has specified, and then stores the encoded fragments in the appropriate regions and availability zones of the cloud.

5.4 DSFS Security

DSFS itself takes some basic precautions to keep its operations generally secure. It uses RESTful principles with all client communications and services [29]. By maintaining minimal sensitive state regarding client requests, there are fewer points of attack. All client information is either provided by the client at the time of request, or is retrieved or recalculated by the server when it is needed. Some information may be kept as to the cloud infrastructure's current state or other non-request-specific observations, but anything directly related to the encoding and placement operation is not saved.

All client and cloud communications are through HTTP messages. For protection from eavesdroppers and man-in-the-middle attacks, whoever sets up DSFS can create their own SSL certificates and allow HTTPS connections.

While servicing client requests, DSFS handles the plaintext form of data and processes sensitive encoding and fragment placement metadata. At this time, if the service is being run in a cloud, these values are vulnerable to a memory dump or other attack by others within the cloud infrastructure. DSFS stores any possibly sensitive data in protected memory, which prevents the cloud hypervisor and any peers from reading this data as it passes through (2.1.4).

CHAPTER 6

IMPLEMENTATION

We implemented DSFS as a C/C++ application using FastCGI to interface with a web-server. The main objective was for our implementation to give performance results similar to the fully-fledged service, but still be written within our time constraints. We chose to implement four popular encoding schemes and give clients complete control over the encoding parameters. Our fragment placement algorithm satisfies all implemented encoding scheme constraints, like encoding threshold and replication. We do not allow any type of region preference by the client, nor do we consider past region or availability zone performance in our decisions. Our implementation will lose some small amount of performance by not recording region statistics, but the lack of client region selection would only significantly affect security from the clients' perspective.

6.1 Encoding

Our implementation supports four customizable types of encoding: *Secret-Sharing*, *XOR-Secret-Sharing*, *Erasure-Coding*, and *AES*, plus the option for no encoding. For more information on the details of how these encoding schemes work, see section 2.

- *Secret-Sharing* refers to Shamir's secret sharing with a (t, n)-threshold scheme. If the threshold and total fragments are unspecified by the client, they default to 3 and 5 respectively. Shamir's secret sharing is implemented using Crypto++, a C++ library with a broad range of cryptographic schemes which allows the encoding of arbitrary-length data.

- *XOR-Secret-Sharing* means the XOR secret sharing scheme that requires a (n, n)-threshold scheme. It defaults to $(3, 3)$ and is self-implemented, using the C++ rand() function to generate the random other shares.

- *Erasure-Coding* is more specifically Reed-Solomon erasure coding using a (t, n)-threshold scheme that defaults to $(3, 5)$. Our implementation uses Jerasure, a C/C++ library developed at the University of Tenessee specifically for storage applications [30].

- *AES*, or the Advanced Encryption Standard running in CBC mode, generates a random 256-bit key for encryption which is stored in the object metadata. AES is also implemented with Crypto++, in a similar fashion to Secret-Sharing. Crypto++ was chosen due to its large cryptographic scheme base, allowing us to easily add or change encryption methods we used and supported.

Because some encoding methods require input data to meet certain criteria, the length of the original data is stored in the object metadata (6.5), so any added padding can be removed after decoding.

6.2 Webserver

The Lighttpd web server is a secure, fast web-server with a very low memory footprint and efficient resource management that is perfect for running on small VMs. The fewer resources our implementation used, the more could be devoted to the cloud instances and test clients for more accurate evaluation. As the smallest, but still fully-functional, public web-server available, it was the best choice. It also supports FastCGI to use as the communication protocol, which became an important factor in our implementation's performance.

FastCGI is a protocol used to interface with a web server; a variation on the widely used Common Gateway Interface (CGI). FastCGI was designed for web servers that handle periods of rapid requests. At startup, FastCGI sets up a configurable number of persistent processes that handle all requests. Each individual process can handle many requests over its lifetime, avoiding the overhead of per-request process creation and termination that CGI normally suffers from. Multiple FastCGI servers can be configured simultaneously, which would simplify scalability and reliability in future derived implementations.

6.3 HTTP Communication

Clients communicate with the service through HTTP messages, setting custom HTTP header fields to specify any options and sending all data in the message body. A read is communicated through an HTTP GET request, and a write with an HTTP PUT request. Any returned information (e.g. the requested data or processing error) is sent in the message body of the HTTP response within the same HTTP session.

Valid HTTP header fields:

- HTTP_X_ACCOUNT_ID: account ID for the client storing the data. This level of access control is handled entirely using the cloud's infrastructure.
- HTTP_X_CONTAINER_NAME: name of the container the object will be stored in (like a folder). If a container does not already exist during a PUT request, the container is created.
- HTTP_X_OBJECT_NAME: reference value for the object in all future requests. If the object already exists, the previous object, including any fragments and metadata, will be purged before uploading the new value.
- HTTP_X_ENCODING_METHOD: encoding scheme to use before uploading the data. Valid values are Secret-Sharing, XOR-Secret-Sharing, Erasure-Coding, AES, and None.
- HTTP_X_ENCODING_THRESHOLD: the threshold number of fragments needed to recover the original data in an encoding scheme. Only used when required by the encoding method.
- HTTP_X_ENCODING_TOTAL_FRAGMENTS: the total number of fragments to be generated by an encoding method. Only used when required by the encoding method.
- HTTP_X_REPLICAS: the number of replicas to be stored in the cloud, whether they are copies of encoded fragments, copies of the data itself.

The options related to each of these fields may or may not have default values when left unspecified by the client (see section 5.2.1). All encoding and replication parameters are stored in the object metadata (6.5) for data recovery.

6.4 Data Placement

Our implementation is given a list of available cloud instances it may upload to. Each of these instances is interpreted as one availability zone, and the group of availability zones is subdivided into regions. As our implementation supports only the selection of encoding method and replication, the data placement algorithm only factors in the threshold, number of fragments, and number of replicas.

The placement begins first by distributing the fragments across the regions. It maintains pool of available regions, and a list of how many fragments have already been placed into each region. For each fragment, it randomly selects a region. If adding this fragment brings the region within one of the encoding threshold, the region is removed from the pool. Then the fragment replicas are randomly put in their region's availability zones, not placing more than one replica in the same zone. There is the possibility that a region may not have enough availability zones to hold all the fragment replicas, but this case is extremely unlikely with a reasonable replica count and is not handled. The names and final placements of all encoded fragments are stored in the object metadata (6.5) for data recovery.

6.5 Metadata

The object metadata is a key part of both the storage and recovery of data. Many components of the control and data planes store small pieces of information in the metadata during PUT operations which are essential to successfully processing a GET:

- *Fragment names*: encoded fragments are given unique names to prevent conflict during upload

- *Fragment locations*: by allowing random fragment locations, and placing the fragment locations in the metadata, an attacker cannot deduce anything about where fragments could be

- *Encoding method, threshold, total fragments, replicas*

- *AES key, IV*: these values are randomly created during data encoding, if AES encryption is selected. This was an implementation choice to not instead calculate them from other known values.

Protection of the metadata is of the utmost importance. Because the metadata contains the locations of all encoded fragments, as well as other information necessary for decoding, its compromise would reveal important information to an attacker. However, even with knowledge of each fragment's location, an attacker would still need to gain access to these locations across multiple regions, assuming the data was encoded and stored across multiple regions, in order to compromise the data. For availability and reliability, our implementation replicates the metadata across multiple availability zones using consistent hashing to decide placement. For security, we encrypt the metadata with AES using a 256-bit key derived from the client key and server key. Our current implementation concatenates the client key and server key, and generates the derived key using scrypt with random salt and initialization vector. The salt and initialization vector are both stored with the encrypted metadata so the key can be rederived after a GET. This method is not perfectly secure, but it is reasonable to assume a sufficient derived key generation algorithm could be implemented with little change in performance.

CHAPTER 7

EVALUATION

7.1 Cloud Setup

In our implementation, we wanted to have complete access and control to all aspects of the overall system for evaluation and understanding. This meant we could not simply use an existing commercial cloud, like the popular Amazon Simple Storage Service (S3), that hides most of its underlying architecture. We chose to run our own cloud using the open-source OpenStack Swift. Aside from its excellent documentation, the customization it allows its users was comparable to existing commercial clouds.

Swift is an open source cloud storage key-value store, with a design very similar to S3 [31]. A Swift instance is like a single region (4.1), isolated from any other regions. It is comprised of some number of storage nodes (i.e. availability zones) that store the actual data, along with one or more supervisor proxy servers that handle the storage requests. A client creates their own account and containers in that account; then they can store data as objects in any container.

However, Swift does not give the client control over which storage nodes will be responsible for their data; it uses consistent hashing rings to decide object placement. This client access limitation makes a Swift instance more like an availability zone in practice, though its intention is clearly like a region. The current S3 interface has a similar protocol and only provides clients the option to select a specific region [1], but we were able to adapt this to the system in our implementation. Swift has a relatively simple setup procedure that allows the entire cloud to be run transparently on a single server with a unique IP address, called Swift All-in-One (SAIO) [32]. Therefore, regions and availability zones can be added easily to the cloud: the system is provided a list of all Swift instance addresses. This list is only read at startup, so an update requires a restart of the whole system.

In order to compare the performance of Swift requests with an existing commercial

system, we also set up an Amazon S3 account. As explained above, S3 allows clients to place objects in chosen regions, similar to Swift. At the time of our evaluation, there were eight accessible regions, which we split into a set of regions and availability zones just as we did with the SAIO instances.

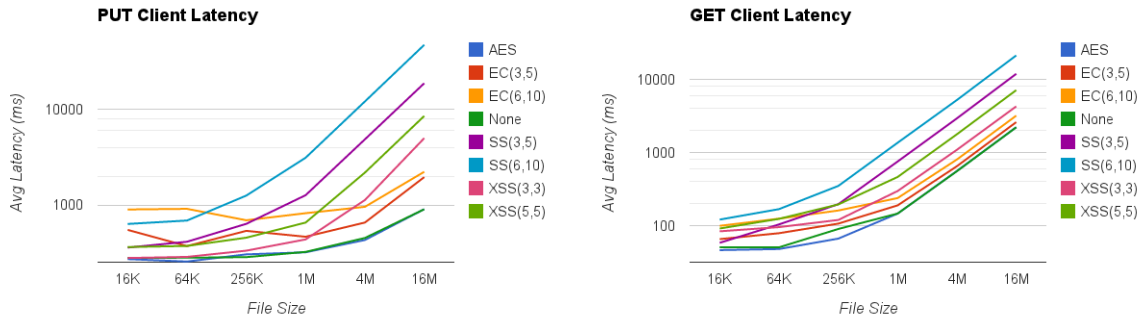
7.2 System Setup

All DSFS performance tests were run on virtual machines (VMs) through the Jedi cluster at Georgia Tech. This cluster is an 80-node, 760-core, Penguin Computing cluster with 30 Relion 1752 servers (2-socket, 6-core, 2.66GHz Intel X5650, 48GB RAM each) and 50 Relion 1702 servers (2-socket, 4-core, 2.4GHz Intel E5530, 24GB RAM each), configured with Openstack Nova, and linked by a dedicated high-performance backbone utilizing gigabit Ethernet. The client, webserver, and SAIO instances were all run on *ne.large* VMs, which have 8GB RAM, 10GB disk, and 4 cores, with installed Ubuntu 12.10. The webserver ran FastCGI with 8 forks, and the client VM that generated all requests had a maximum of 10 requests being executed simultaneously, using xargs to handle thread management. Curl was used from the client command line to initiate all service requests.

7.3 Performance

Our evaluation included 4 micro-benchmarks:

- Request latency: we measured the average latency of a client request relative to the input file size and encoding method.
- Request throughput: we evaluated the possible throughput of our implementation by varying the input file size for each encoding method with a static number of concurrent requests.
- Cloud i/o: we compared the average upload and download times of differently-sized objects to our SAIO instances with the time taken to upload and download the same objects to S3.



(a) PUT Latency

(b) GET latency

Figure 4: Average end-to-end latency.

- Encoding comparison: we show the differences in encoding and decoding overhead as the input file size increases

7.3.1 Request latency

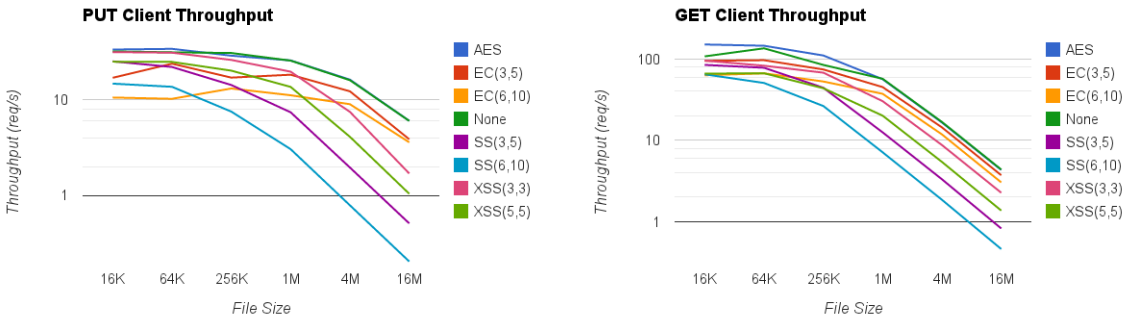
We evaluated the effect of input file size on the total runtime of a client request for each type of encoding method. Because our implementation of DSFS allows client file input, encoding method selection, and encoding parameter selection, we were able to perform all tests on the same running service with the same client without any inter-test modifications. For each measured input size (16KB to 16MB), we ran 100 tests using each valid encoding method (see section 6.1), both for GET and PUT requests. The latency was calculated using the Linux *time* command to measure the elapsed time of individual curl requests. As a result, the measured latency is more the end-to-end latency, as it encompasses any additional client overhead on top of the service fulfilling the client request.

Looking at Figures 4a and 4b, the general trend of request latency relative to object size for both PUT and GET is clear. All latencies increase as the object size increases, regardless of encoding method, however the encoding method used by the request affects how quickly latency increases. The starting latencies seem rather arbitrary as erasure coding appears to be worse than even secret sharing, but as the input size increases, everything levels out and goes back into an understandable order. Shamir's secret sharing has by far the most

overhead. Its encoding/decoding scheme is more computationally expensive than all other supported schemes, especially as the object size passes 1M, and the size of its encoded output is also much larger when compared to schemes with the same level of availability. XOR secret sharing has the same problem of multiplied output, but its encoding algorithm scales much better with larger data inputs. AES is nearly identical to having no encoding, as they both have 3 replicas, and the overhead of performing AES encryption and decryption on the tested inputs is trivial relative to upload time.

Erasure coding is only slightly more expensive than no encoding at all as file size gets larger. Even though its overall storage requirements are less than that of 3 replicas, it still must store a larger number of objects. For example, a $(3, 5)$ erasure coding scheme can lose 2 fragments, requires storage space of $\frac{5}{3}$, and produces 5 objects. Having a replication factor of 3 with no encoding can also support the loss of 2 fragments, requires 3 storage, and produces 3 objects. Therefore, a PUT of the same object would required the storage of 5 fragment objects for erasure coding, and 3 object copies for only replication. For a GET request, the difference is much greater. Reading erasure coded data requires retrieving at least 3 objects to decode, while replicated data only needs to retrieve more than 1 object if the first attempt fails. A future implementation should read at least a quorum of any existsing replicas to ensure the correctness of the data, which would likely make erasure coding the fastest approach.

Ideally a PUT request with specific parameters would take as long as its corresponding GET request for all encoding types, but this is not the case. A GET using Shamir's or XOR secret sharing is much faster than a similar PUT, because A PUT must upload all of the encoded fragments while a GET only has to download the minimum number before it can begin decode. However, an erasure coding GET is slower than a PUT, which seems counter intuitive. According to our evaluation results, the extra time came from a longer decode time, possibly a consequence of our erasure coding implementation, Jerasure. Nevertheless, both PUT and GET requests follow the same trends relative to input file size and



(a) PUT Throughput

(b) GET Throughput

Figure 5: Average throughput.

clearly show the different performances of each tested encoding scheme.

7.3.2 Request throughput

Request throughput was evaluated with a similar objective as latency, to evaluate the effect of input file size on the throughput of client requests. The same sizes and encoded methods were tested, each averaged over 100 rounds. Here we were limited by our available resources and were therefore not able to test our implementation as completely as we wished. The FastCGI application was spawned with 8 forks, a number we did not choose arbitrarily. Because our implementation was only designed to be run on one server at a time, we could have only one service running for the evaluation. Through some amount of experimentation, we deduced that the webserver VM could not handle any more than 8 forks without slowing down dramatically. This meant our service had 8 processes running continuously to handle up to 8 simultaneous client requests without queuing.

To get an accurate measure of throughput, we needed to stress the service by having more than 8 requests being sent at once. This was simple enough to attain with one client VM by using the Linux *xargs* command. We gave *xargs* a list of 100 curl requests, representing the 100 rounds of one specific file size and one encoding method, and we set the number of concurrent commands to 10. *xargs* placed these commands into a work queue, spawned 10 worker threads, and gave each thread a command from the queue until it was

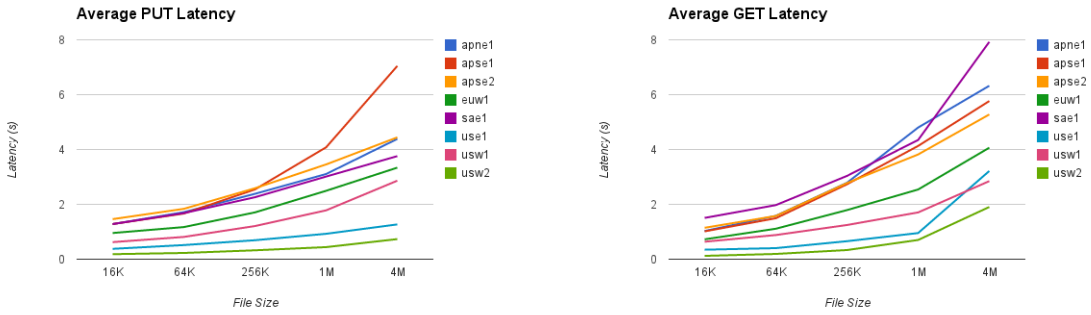
empty. We again used the Linux *time* command to measure the total execution time of all 100 rounds, thus including any additional overhead required by *xargs* to set up and maintain the thread pool. Throughput was calculated by dividing the total number of client requests, 100, over the elapsed time reported by *time*.

Figures 5a and 5b show the average throughput of PUT and GET requests over a range of input sizes and all valid encoding methods. Again, all types and values of requests follow the same general trend, where throughput decreases with increasing file size. When compared with the latency graphs from Figure 4, there is an almost identical pattern, demonstrating that having more concurrent client requests than the service is designed to handle does not significantly affect performance, at least for a 25% increase over 8 threads. Future work should, if the implementation is not made to scale itself up automatically with a large number of queued requests, stress the service to much larger extent. This will become a serious problem when many clients try to perform arbitrary operations on the same data at the same time as is likely in a real-world environment.

Again comparing the performance of similar PUT and GET requests, the throughput of GET requests is several times better than that of PUT requests. Though all testing was done with the same number of concurrent threads, the time taken by each thread was not the same. Having concurrent PUT requests requires more work by OpenStack having to maintain consistency throughout, for even though we knew all stored objects in our evaluation had different names, OpenStack did not and needed to ensure there was no possibility of conflict before allowing another PUT to happen in parallel. Simultaneous GETs do not affect each other with the same consistency issues and therefore in our implementation GETs have a higher throughput.

7.3.3 Cloud Communication

Because our system implementation was entirely within the same network (the client, service, and cloud were all running in Jedi cluster), we knew we should also compare part of our performance with a commercial cloud. Depending on DSFS's placement, the client



(a) Average S3 PUT latencies

(b) Average S3 GET latencies

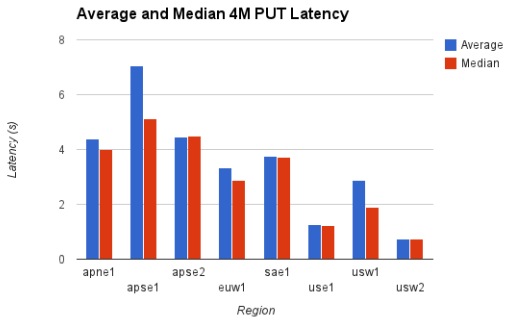
Figure 6: Comparison of communications to different cloud regions.

may be within the same network as DSFS, or DSFS may be in the same cloud as the data. This means that there will almost always be some part of the system communication that is required to send a non-trivial number of messages through the general Internet. We evaluated the effect of geographic distance on communication overhead, and how this overhead compares to the end-to-end latency of a complete request.

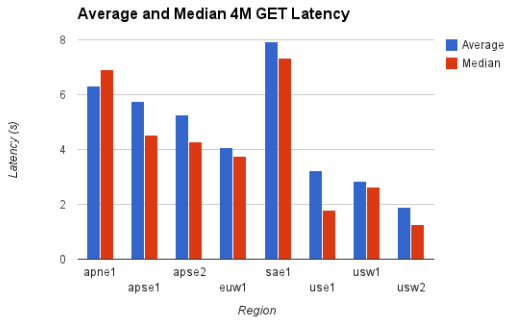
7.3.3.1 Latency by Region

We first compared the overhead of uploading and downloading data to and from different S3 regions. Looking at Figures 6a and 6b, the effect of distance on average communication overhead is clear. As the distance from the client to the cloud server increases, the elapsed time in any cloud request increases. Data stored in the closest region to our testing, US West 2, took much less time to store and retrieve than any other region.

The only oddities we noticed in our evaluation were the latency inconsistencies. For each region and size during our 100 rounds of testing, there were always about 20 requests that took 2 or more times longer than the median request latency. For example, the median request latency for 4M objects in region Asia Pacific Southeast 1 is 4.5s, but 27 of our 100 rounds had a latency ≥ 9. Because we cannot show every recorded data point, we chose to demonstrate this by comparing the average and median latencies for GET and PUT requests to each region for objects of size 4M. When the median is lower than the average, some



(a) Average vs Median S3 PUT 4M



(b) Average vs Median S3 GET 4M

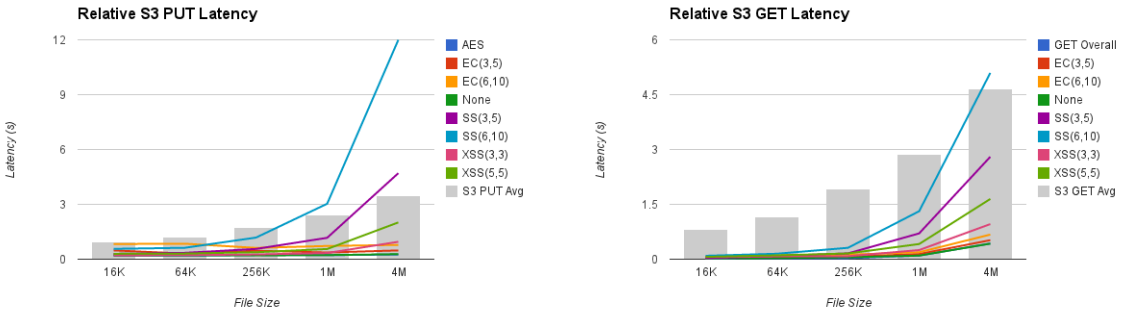
Figure 7: Demonstration of inconsistent S3 communication latencies.

points must have been higher than normal in order to bring up the average. Figures 7a and 7b show how the median latency is nearly always significantly different from the average. Looking at these graphs, we also noticed some of the differences between the GETs and the PUTs themselves. For certain regions, most significantly South America, but also Asia Pacific Northeast 1 and US East 1, GETs are much slower than PUTs.

We ran additional tests to try and discover the meaning behind these results. Times were recorded using standard C functions, and we modified `libs3`, the library used to communicate with Amazon S3, to measure time at as low a level as possible. The same variable latencies were recorded with all changes to all tests, and occur sometime during the `curl_easy_perform()` command that executes the actual `cURL` request. If 20% of all requests could have a substantial amount of additional latency, and the latency could be significantly different depending on whether the request is a GET or a PUT, this could be a serious deterrent to using Amazon S3 as the backing to DSFS.

7.3.3.2 Relative Latency

The overhead of communication with the cloud will exist no matter if DSFS is local to the client, in the same cloud as the data will be stored, or if it is even being used at all. It is therefore useful to compare this unavoidable overhead with the additional cost provided



(a) Relative S3 PUT Latency

(b) Relative S3 GET Latency

Figure 8: Comparison of cloud communication with remainder of request.

by DSFS. The two graphs in Figures 8a and 8b represent the communication cost as vertical bars. For a GET request, additional cost for any encoding method is less than half the communication cost up to 1M input size. PUT requests have the same results up to 256M. Note that this includes all tested encoding methods. Certain methods, like secret sharing 6,10 and erasure coding 6,10, are not commonly used but are included for a more comprehensive set of results. These graphs can be used a a reference for determining the cost of using DSFS for a client's particular situation, and deciding what security measures are appropriate.

CHAPTER 8

CONCLUSION

In this thesis we have detailed a flexible and secure data storage service, DSFS, designed to meet the varied needs of clients storing sensitive data in the cloud. Some of the systems mentioned previously provide protection from the cloud administrator and peer attacks, but offer the client little control or customization over the process. Others require modification of the client or an entire redesign of the cloud. DSFS instead runs as a service, including several proven protection schemes with client-configured parameters and a useful data placement algorithm that will place encoded fragments to maximize data security relative to the client's direction. DSFS requires only that the client can send HTTP messages, and that the cloud's infrastructure be composed of regions and availability zones. Each region must fall under one administrative domain, guarantee no unauthorized data movement to other regions, and be made up of multiple availability zones, where availability zones within the same region are unlikely to fail at the same time for the same reason.

Our initial prototype implemented only a limited part of the architecture to test possible features and accurately measure certain aspects of the performance. The effects of input data size on performance were as expected: latency and throughput decrease with input data size increase across all tested encoding methods. However, the evaluation of cloud requests for a popular cloud, Amazon S3, showed some surprising inconsistencies with 20% of all requests having a latency two or more times larger than the median. Though the cause of this additional overhead was narrowed down only to the cURL function call, so it may not be the fault of S3 itself, significant variations such as this may hinder the use of certain commercial clouds as the backing store to DSFS, especially when we want to make guarantees on DSFS's overall performance.

Looking at the average latency of a cloud request to any region, we show the exceptional performance of DSFS in relation to the unavoidable cloud communication. The most

commonly used encoding schemes, such as erasure coding and AES encryption, have an additional latency of less than 20% that of the cloud requests for for all tested input data sizes up to 4M, while the more expensive schemes, like Shamir's and XOR secret sharing, have better relative performance only for input less than 1M. Using our evaluation results, it will be much easier for a client to choose the optimal parameters to DSFS based on the trade-offs between security and input data size. If the client is able to understand and control how their data is stored, they will be more likely to feel safe storing sensitive information in the cloud.

REFERENCES

[1] "Buckets and Regions." http://docs.aws.amazon.com/AmazonS3/latest/dev/LocationSelection.html, Accessed: May 7, 2014.

[2] "Google Cloud Storage: Regional Buckets." http://docs.aws.amazon.com/AmazonS3/latest/dev/LocationSelection.html, Accessed: May 7, 2014.

[3] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: dependable and secure storage in a cloud-of-clouds," in *Proceedings of the sixth conference on Computer systems*, pp. 31–46, ACM, 2011.

[4] R. Padilha and F. Pedone, "Belisarius: Bft storage with confidentiality," in *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*, pp. 9–16, IEEE, 2011.

[5] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti, "Potshardsa secure, recoverable, long-term archival storage system," *ACM Transactions on Storage (TOS)*, vol. 5, no. 2, p. 5, 2009.

[6] "Amazon Web Services: Overview of Security Processes," whitepaper, Amazon Web Services, November 2013.

[7] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[8] S. B. Wicker and V. K. Bhargava, *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[9] "Best Practices for Using Amazon S3." http://aws.amazon.com/articles/1904, Accessed: May 7, 2014.

[10] "Database Replication." http://docs.oracle.com/cd/F49540_01/DOC/server.815/a67781/c31repli.htm, Accessed: January 14, 2014.

[11] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 203–216, ACM, 2011.

[12] J. Szefer and R. B. Lee, "A case for hardware protection of guest vms from compromised hypervisors in cloud computing," in *Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on*, pp. 248–252, IEEE, 2011.

[13] A. Subbiah, "A new approach for fault tolerant and secure distributed storage," in *DSN*, vol. 6, pp. 157–159, Citeseer.

[14] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 1–14, 2002.

[15] A. Subbiah and D. M. Blough, "An approach for fault tolerant and secure data storage in collaborative work environments," in *Proceedings of the 2005 ACM workshop on Storage security and survivability*, pp. 84–93, ACM, 2005.

[16] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: the least-authority filesystem," in *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pp. 21–26, ACM, 2008.

[17] M. K. Reiter, M. K. Franklin, J. B. Lacy, and R. N. Wright, "The ω key management service," in *Proceedings of the 3rd ACM conference on Computer and communications security*, pp. 38–47, ACM, 1996.

[18] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 253–267, ACM, 2003.

[19] L. Harn and C. Lin, "Detection and identification of cheaters in (t, n) secret sharing scheme," *Designs, Codes and Cryptography*, vol. 52, no. 1, pp. 15–24, 2009.

[20] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population.," in *FAST*, vol. 7, pp. 17–23, 2007.

[21] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.

[22] J. Jonsson and B. Kaliski, "Public-key cryptography standards (pkcs)# 1: Rsa cryptography specifications version 2.1," 2003.

[23] W. J. Robison, "Free at what cost: Cloud computing privacy under the stored communications act," *Geo. LJ*, vol. 98, p. 1195, 2009.

[24] A. Scolnik, "Protections for electronic communications: The stored communications act and the fourth amendment," *Fordham L. Rev.*, vol. 78, p. 349, 2009.

[25] L. R. Kattan, "Cloudy privacy protections: Why the stored communications act fails to protect the privacy of communications stored in the cloud.," *Vanderbilt Journal of Entertainment & Technology Law*, vol. 13, no. 3, 2011.

[26] F. H. Cate, "Eu data protection directive, information privacy, and the public interest, the," *Iowa L. Rev.*, vol. 80, p. 431, 1994.

[27] A. Kertesz and S. Varadi, "Legal aspects of data protection in cloud federations," in *Security, Privacy and Trust in Cloud Systems*, pp. 433–455, Springer, 2014.

[28] J. C. Francis, "In the matter of a warrant to search a certain e-mail account controlled and maintained by Microsoft Corporation." https://www.documentcloud.org/documents/1149373-in-re-matter-of-warrant.html, April 2014. District court case, New York magistrate.

[29] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[30] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2," *University of Tennessee, Tech. Rep. CS-08-627*, vol. 23, 2008.

[31] "swift 1.13.1.64.g08dba08 documentation." http://docs.openstack.org/developer/swift/, Accessed: Jan 12, 2014.

[32] "SAIO - Swift All In One." http://docs.openstack.org/developer/swift/development_saio.html, Accessed: Dec 17, 2014.