GeoShare: Experience with a Geographically Diverse Cloud Data Storage Service

Kyle Bilbray, Douglas Sigelbaum and *Douglas M. Blough*
School of Electrical and Computer Engineering
Georgia Institute of Technology

***Abstract*—In this report, we describe our experiences in developing GeoShare, a geographically diverse cloud data storage service. Users are becoming increasingly wary of storing their sensitive data within servers physically located in one country or within one political or legal boundary. GeoShare facilitates the fragmentation of information across administrative and geopolitical boundaries, thereby maintaining data confidentiality from powerful adversaries such as cloud administrators and nations. Our approach includes a method to calculate placements of information fragments to satisfy users' geographic constraints and simultaneously optimize system performance. We also present an approach to optimize cost/performance by considering the storage cost versus latency tradeoff that arises from replicating data across far-flung geographic regions. Our approaches have been fully implemented, tested, and evaluated in the GeoShare prototype cloud storage service, which runs on top of Amazon Web services and makes use of 8 world-wide regions accessible through the Amazon S3 cloud storage service. We demonstrate the system through experiments having Amazon EC2 clients all over the world simultaneously creating, uploading, and downloading objects using the GeoShare service.**

I. Introduction

Within the past few years, cloud storage has exploded. Users have flocked to cloud storage services due to their ability to provide access from anywhere and across multiple devices, along with assurances that data will always be available. Cloud storage is also an ideal medium for sharing data among multiple users, and services such as DropBox, GoogleDrive, and OneDrive are all routinely used for this purpose. However, users are becoming increasingly concerned about the security of their data stored in the cloud. High profile data breaches of both single individuals' data and data across many users have brought increased attention to security issues. Due to the multi-tenancy nature of cloud systems, the fact that cloud providers have access to the low-level cloud software and hardware, and potential abuses by authoritarian governments in certain locations, cloud data is extremely vulnerable to theft and abuse.

One simple solution to protecting data in the cloud is encryption. This solution works very well for a single user's data, which is always accessed from a specific device. In this situation, data is stored in the cloud only in encrypted form and the encryption/decryption key resides only on the user's device. This effectively eliminates the threat of the data being breached solely by accessing what is stored in the cloud. However, encryption does not work well when a single user wants to access the data from multiple devices nor when multiple users share data. In these situations, key distribution and management becomes complicated and changes to access permissions are costly. Storing encryption keys in the cloud directly does not solve the problem since it is, in general, possible that both an encrypted object *and* its encryption key are compromised, thereby allowing the data to be breached.

A common approach to this problem is to encrypt data, and use threshold secret sharing [1] to fragment the information about the encryption key into multiple shares (fragments), which can then be stored across multiple cloud servers. However, if an entity has access permission to a sufficient number of servers that are storing fragments of the encryption key, then the key, and consequently the encrypted object, are still at risk. If fragments are stored across multiple servers or datacenters within the same administrative domain, or if the servers or datacenters are all within a single political boundary where, for example, an authoritarian government could access them all, then this approach is not sufficient.

In this paper, we consider geographic dispersal of key fragments to provide stronger protections against these threats. Fragments are spread across geographically diverse regions that span political boundaries. We assume that each of these regions are administered separately and administrators do not collude across regions. This could naturally be satisfied by using a different cloud provider in each region. Alternatively, we envision that completely isolated administrative domains for different regions could become a selling point for a single cloud provider to attract security-conscious users. Users can specify constraints on what regions to use, or not to use, and which groups of regions might collude. Given these constraints, fragments are then spread in such a way that no single entity or colluding group can access enough fragments to reconstruct the key.

We have demonstrated this approach by building a prototype of a cloud storage service, called GeoShare, which spreads key fragments and replicas of encrypted data objects across geographically distinct regions provided by Amazon's S3 cloud storage service. This provides protections against entities that can compromise individual regions, or groups of regions as specifed by the user. We also exploit latency distributions to optimize performance in this setting. We have developed a key fragment placement algorithm that generates a non-deterministic set of fragment locations that satisfies the given geographic constraints and favors regions that tend to reduce average access latency. We also exploit geographic diversity to optimize the replication of encrypted objects across regions. For a fixed number of replicas, we show how to compute the replica locations that minimize average access latency. We also define a utility function that can be used to quantify

the tradeoff between storage cost and latency based on the number of replicas. We show how this function, along with the optimal replica placement solution, can be used to choose the best number of replicas and optimally assign them to different regions. All of these features are demonstrated and evaluated through a prototype cloud storage service built using various Amazon Web services with Amazon S3 providing the back-end cloud storage service with 8 geographically distinct regions.

II. Related Work

Most prior work on spreading data across a cloud storage service has focused on erasure coding data or using other types of coding schemes that provide similar properties to erasure codes, e.g. [2], [3], [4], [5], [6]. However, these codes do not provide the strict confidentiality required by our use case. While a threshold number of data pieces are required to fully reconstruct the data, partial information is leaked from individual pieces. In addition, these approaches typically just consider a cloud storage service as a collection of servers and do not consider its organization into datacenters and regions. In [7], cloud data is encrypted with keys that are secret shared across multiple servers. However, the work assumes that a dedicated set of servers outside of the main cloud storage service are provided for key storage. The work also does not consider geographic distribution nor the organization of the cloud storage service itself since the key service is separate.

Several works explicitly consider the use of multiple isolated administrative cloud storage domains, usually in the form of separate cloud storage service providers [8], [9], [10], [11]. In [10], the authors propose to directly secret share data objects across multiple cloud storage providers. The primary consideration in this work is cost, where it is assumed that different providers have different pricing. Thus, the paper determines how to distribute data fragments in a lowest cost fashion. One issue with this work is that the overhead of secret sharing is extremely high, which has prevented practical solutions where potentially large data items are directly secret shared. Also, the work focuses solely on an abstract model of multiple providers and pricing, but does not provide an implementation nor consider geographic constraints. Metasync [9] is a multi-provider system where encrypted data is replicated across different cloud storage providers. The paper places the burden of key management on users, and thus does not consider how to handle keys within the cloud storage services. The system is fully implemented and supports Box, Baidu, Dropbox, Google Drive, and OneDrive backend cloud storage services. DepSky [8] is similar to GeoShare in that it replicates encrypted data and secret shares keys. However, DepSky targets multi-provider settings but is not concerned with geographic constraints. So, if key fragments are stored across separate providers but the servers storing those fragments are all located in one country controlled by an unfriendly government, data is still at risk.

The only cloud storage work of which we are aware that specifically considers geographic constraints is SPANStore [11]. However, this work only considers replication across geographically-diverse locations but does not consider the confidentiality aspect. Also, the work primarily considers optimization of latency and cost but not geographic constraints imposed by users' concerns about collusion between sites.

III. Background on Amazon Web Services

While the architecture and approach that are described in Section IV are general, we have chosen to implement, test, and evaluate the approach in a real cloud environment. After examining several cloud providers, such as the Google Cloud Platform and Microsoft Azure, we chose to use Amazon Web Services (AWS) both as a base model for the GeoShare system and as our implementation platform. At the time of this project's inception, AWS supported more regions than both the Google Cloud Platform and Microsoft Azure, and at more reasonable prices. It also was much better established, having already existed for many years with many significant commercial clients, while the Google and Microsoft platforms were still in their formative stages.

AWS is a collection of remote computing services that make up a cloud platform. Amazon's two most popular services are Elastic Compute Cloud (EC2) and Simple Storage Service (S3). EC2 allows a user to instantiate a virtual machine in one of 8 geographically distinct regions around the globe. This is ideal for emulating GeoShare users employing our service from locations we cannot directly access ourselves, allowing for a comprehensive testbed. At the same time, we can run the GeoShare Web server in EC2, circumventing the issues of maintaining a local webserver. Amazon S3 provides storage of objects in the same 8 regions as EC2, with inherent intra-region replication for fault tolerance within a single region. While 8 regions is not quite as diverse an environment as we desired, it still has some amount of control that GeoShare can leverage.

While the list of Amazon regions is expanding, the 8 regions that were present when our experiments began and that were used in our implementation and evaluation are as follows:

- us-east-1 (use1), located in Northern Virginia
- us-west-1 (usw1), located in Northern California
- us-west-2 (usw2), located in Oregon
- sa-east-1 (sae1), located in Sao Paulo
- eu-west-1 (euw1), located in Ireland
- ap-northeast-1 (apne1), located in Tokyo
- ap-southeast-1 (apse1), located in Singapore
- ap-southeast-2 (apse2), located in Sydney

IV. System Architecture and Implementation

A. High-level Architecture

The GeoShare system is divided into three key components, as seen in Figure 1: the client, the metadata and access control service, and the backing cloud storage service(s). In our current implementation, the backing cloud storage service is Amazon S3 but, architecturally, this could be a different storage service with geographically diverse regions or it could be comprised of multiple storage services from different cloud providers. Similarly, the metadata and access control service currently runs in Amazon EC2 but could run on any cloud provider or dedicated server. By only allowing specific pieces of information to be passed from one system component to another, our

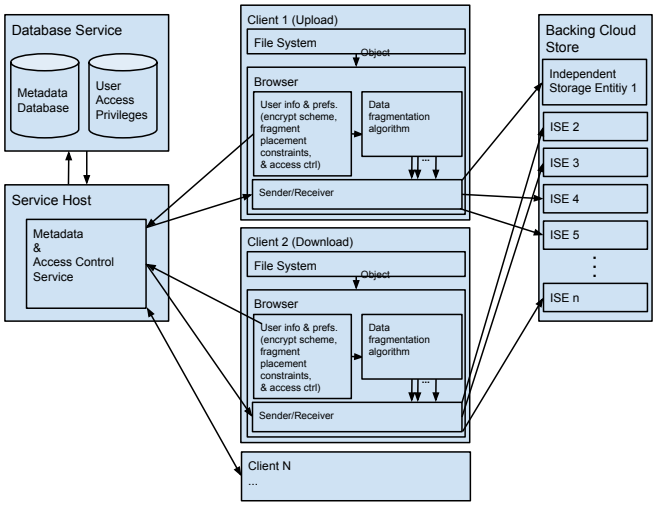

Fig. 1. High-level system architecture of GeoShare system

system can use data fragmentation and encryption to ensure that no single component has the ability to independently recover stored objects that are shared among multiple clients.

Our primary objective is to preserve data confidentiality while allowing data to be shared among multiple users. The standard mechanism for data confidentiality is encryption, which works quite well for individually-owned data, where the data owner can manage encryption keys independently of other users. However, for shared data, it is well known that the complexity of key management makes it untenable for users to store encryption keys locally. Thus, the storage service must somehow store and manage encryption keys, which means that the service must also guarantee confidentiality of the keys in addition to the data. One convenient solution to this problem is to encode encryption keys using secret sharing techniques [1] and spread the key shares across multiple storage servers. We adopt this approach in GeoShare. To be specific, users encrypt objects, replicate the encrypted objects, secret share the encryption keys, and store encrypted replicas and key shares across the independent storage entities in the GeoShare system.

For simplicity, we adopt S3's bucket storage model within GeoShare. Thus, users create buckets, which are required to have globally unique names across the GeoShare system, and each object is placed in one specific bucket. This makes the use of S3 for the backing storage service straightforward but this model can also be implemented on other cloud storage services that emply different storage models. For example, in hierarchical file system models, such as are used by Google Drive, Microsoft OneDrive, and DropBox, we can simply create a unique folder for each user-created bucket.

Client: A client accesses our GeoShare system through a RESTful web interface within a standard browser. This page handles all communication with the metadata and access control service and the backing cloud store, as well as any necessary encoding/decoding operations involving raw object data. The authentication information required for these communications is known only by the client and consists of the following:

- **GeoShare –** a username and password are used to authenticate the user to the metadata and access control service
- **Amazon S3 –** unique credentials generated by Amazon are used to send requests to S3

Metadata and access control service: Data placements are generated and maintained by the metadata and access control service based on a client request's specified parameters and performance data from the backing cloud storage service. All requests are validated against the client's GeoShare account information for object permissions and correct credentials before modifying or returning any requested metadata. Any generic Web server can run the metadata and access control service, which also makes use of an auxiliary database service to store critical data needed by the service. In our case, we chose to run the Web server in Amazon EC2, and we use Amazon RDS for metadata storage, in order to ensure continuous and consistent operation.

Backing cloud store: The backing cloud store(s) must have the ability to store, retrieve, and delete objects in multiple independent locations. It also must have some basic access control, e.g. the ability for users to share objects with specific other users of the same service(s). We chose the Amazon Simple Storage Service (S3) for its 8 independent world-wide regions, and its high degree of customizability. However, other cloud storage services, such as Google Drive and DropBox, also satisfy these requirements.

A typical request follows the flows shown in Figure 1. Generally speaking, a client making a request queries the metadata and access control service for some information and then completes the operation by accessing multiple regions in the backing cloud storage service. The two most commonly used requests are the upload and download of an object, which are described next.

Upload: The client fills out a form with their GeoShare account information, the name of their object file, and requirements on how their object should be stored, e.g. bucket name and (optional) data placement constraints. The account information and object preferences are consolidated into a single HTTP GET request, which is sent to the metadata and access control service. If the account is valid and has permission to modify the given object, the service uses the specified object parameters along with known inter-region performance data to generate placements for the object replicas and key fragments. Note that the metadata and access control service does not receive the object itself but only knows its name and where its parts will be stored. All received and generated object metadata is stored by GeoShare in its metadata database and associated with the appropriate account, while the new placements, and any previous placements of the same object, are returned to the client. The client then encrypts the object with a random key, secret shares the key into the appropriate number of fragments, replicates the encrypted object the specified number of times, and uploads the replicas and key shares to the backing cloud store using its storage credentials and the received placements.

Download: In addition to the GeoShare account information required for every request type, the client needs only to specify the name of the object they are trying to retrieve,

along with the bucket it is stored in. This information is sent to the metadata and access control service in an HTTP GET request. The account itself is validated against existing GeoShare accounts, and the requested object is checked for permissions. Assuming the access is permitted, the placements of all object replicas and key fragments are returned to the client, as well as any object metadata necessary for recovery. The client then downloads the object's key fragments and one object replica from the cloud storage service, uses the key fragments to recover the key, and uses the key to decrypt the object replica.

At no point in any request does any component but an authorized client have enough information to recover a raw object. The metadata and access control service knows the placements of data with the cloud storage service but does not have the account permissions necessary to actually access them. Any group of storage entities, as defined by the user (see Section IV-B2), sees less than the threshold number of key fragments and so cannot recover the key for any object replica.

B. Metadata and Access Control Service Design

1) Metadata: The metadata and access control service calculates and maintains object storage metadata for each user, as well as access control data for each bucket. Our implementation is a persistent Java HTTP server application running on an Amazon EC2 instance, connected to a MySQL database.

When uploading an object, the client needs to know where to send object replicas and key fragments after encrypting the object and generating secret shares of the encryption key. Because S3 buckets are globally unique and can only exist in a single pre-defined region, only the Amazon S3 bucket and object names are required. On each upload, the client sends an HTTP GET request to the metadata and access control service that includes (* = optional):

- GeoShare bucket and object names
- client location
- encoding parameters*
- S3 region groupings, exclusions, and/or demand*

The metadata and access control service receives the request, calculates the necessary replica and fragment placements and sends the results back to the client.

For a GeoShare object download, the client again sends an HTTP GET request to the metadata and access control service, containing the GeoShare bucket and object names. The metadata and access control service receives the request, queries the metadata database, and sends back any pertinent encoding information with the exact S3 bucket and object names to describe where all object replicas and key fragments are stored.

2) Key Fragment Placement: An important part of the GeoShare system, essential to data confidentiality, is the algorithm to calculate key fragment placements for newly generated AES-256 object keys. A deterministic fragment placement scheme could provide assistance to an adversary attempting to subvert the system. Thus, fragment placement selection is done using a biased random process. This process is parameterized by a table of latency data from every S3 region to every other S3 region. Additionally, it can be constrained by a maximum number of fragments allowed in a region or group of regions. The probability distribution is initialized from the empirical download times of each of the regions to the client's location. More specifically, the normalized set of probabilities of storing a fragment in each region is calculated by taking these download times and dividing each by the sum of all download times. This probability distribution favors regions that have low latencies relative to the client's region, but also provides an element of randomness so that the same regions are not selected for every one of a client's keys.

When calculating the fragment placements, the metadata and access control service will also know which regions the client chooses to group together so that it does not place more than the threshold number of fragments in a single group. By default, there is a group for each individual region because it is assumed that the user never wants more than the threshold number of fragments in a single region. However, if the user wants to restrict all regions in the U.S. from collectively receiving the threshold number of fragments, then a group will be created containing all S3 regions in the U.S. (USE1, USW1, and USW2). During the fragment placement selection, if a group gets selected (threshold - 1) times, then each region in that group is removed from the set of regions to choose from, the probability distribution is reinitialized, and the selection process continues.

3) Replica Placement: We assume that objects are shared world-wide and, hence, we replicate objects across geographical regions to try to avoid long inter-region latencies. A side benefit of replication is that an entire region outage can be tolerated. While most storage services, such as S3, automatically replicate data within a region to tolerate individual datacenter outages, certain regions might have all of their datacenters fairly close together. For example, many S3 regions are confined to a single city, such as Tokyo, Sydney, Singapore, Frankfurt, or Sao Paulo. Thus, it is possible that a natural disaster could potentially affect all datacenters within a single region.

Given that objects are replicated in GeoShare the system must decide where to place the replicas.[1] Because replicas are encrypted, and key fragments randomly distributed, object replicas can be placed deterministically, e.g. to provide the minimum average download latency for all users. Using the same inter-region transfer latency data used by our key fragment placement algorithm, the system can determine the optimal regions in which to place replicas. With only 8 S3 regions, it is sufficient to determine replica locations through an exhaustive brute-force calculation. If available, the user demand distribution can also be factored into the optimal placement (e.g. a region with more user demand might have a larger effect on the overall average latency than a region with less demand).

[1] In this subsection, we assume that the number of replicas is pre-determined, e.g. specified by the client as a parameter when submitting the upload request. In the "Utility Function" subsection, we consider the case where both the number of replicas and their locations must be chosen.

While a brute-force calculation is feasible with the limited storage location options of S3, this method scales exponentially with the number of regions. The optimal replica location problem is similar to the p-median problem [12], which is commonly applied to facility location to decide the best placement of facilities that minimizes transportation costs. By factoring in user demand, storage transfer and duration costs, inter-region latencies, and region preferences, a modified form of Lagrangian relaxation can be used to determine optimal replica placement for much larger numbers of regions than are available in S3. A test implementation of this technique, written in MATLAB, completed within a few seconds with thousands of regions.

4) Utility Function: In some cases, clients might not know the best number of replicas to create a priori. They may want their global user base to see fast downloads, but without putting a replica in every possible region, since they will incur increased storage costs as the number of replicas increases. Or conversely, if minimizing storage cost is a high priority, a client may want to use the fewest number of replicas they can while achieving relatively low average latency. Using these two attributes as the basis, we created a simple utility function approach to compute the optimal number of replicas.

The utility function defines the value that the user receives from a given latency, averaged over all client requests, and storage cost (for simplicity, just the number of replicas). Shorter latencies and smaller storage costs have higher value and longer latencies and higher storage costs have lower value. Since it is unrealistic to assume the user manually specifies the utility of every possible latency, storage cost pair, we developed a parameterized utility function that calculates utility using weights that the user chooses for latency and storage cost, w_l and w_s, where $w_l + w_s = 1$. Using these weights, we scale utility from a value of 1, when both latency and storage cost are their absolute minimums to a value of 0, when both latency and storage cost are their absolute maximum. Clearly, it is very unlikely that the system would be able to simultaneously minimize latency and storage cost, so achieving a utility of 1 is, in general, not possible. Finding the optimal utility is a matter of evaluating utilities for the feasible solution points in the system and choosing the highest value. By setting the weights properly, the user can specify whether they prioritize latency or storage cost in determining utility. The utility function we use to achieve this behavior is:

$$U(l,s) = 1 - \left[\frac{w_l (l - \min L)}{\max L - \min L} + \frac{w_s (s - \min S)}{\max S - \min S} \right]$$

w_l: weight of latency attribute

w_s: weight of storage cost attribute

$\min L, \max L$: the absolute minimum and maximum latencies possible in the system for a given user demand distribution

$\min S, \max S$: the absolute minimum and maximum storage costs in the system

To find the set of valid l, s pairs, we leverage the optimal replica placement algorithm from Section IV-B3 to determine the optimal replica locations for each possible number of replicas. Then, using a table of known inter-region latencies, we can determine the average latency, l, for the given user demand distribution when the optimal replica placement is used for s replicas.

5) Access Control: Storing the user's metadata locally would increase performance in retrieving metadata, but would also restrict users from accessing their objects through multiple distinct clients and make it difficult to share objects among multiple users. The metadata and access control service exists mainly to manage all this metadata, facilitating object access from arbitrary clients through a Web application. The security of these transactions is ensured for both client/metadata-service interactions and client/storage-service interactions. For the client/metadata-service interactions, there are 3 basic components of metadata:

user a distinct account name and password hash allowing a user to submit requests to the GeoShare system

object the exact backing cloud storage locations of the object's replicas and key fragments, the GeoShare bucket name, and encoding information

bucket which GeoShare users have permission to read, write, and/or modify the bucket

Each time a user creates a new bucket, our service generates a corresponding bucket metadata entry, giving this user read, write, and modify access permissions. Because S3 bucket names are global, we made GeoShare bucket names global, greatly simplifying the bucket lookup and validation process. When a user shares a GeoShare bucket, which is only possible if she has modify access permission, she simply names a user and lists what permissions he is being granted and the relevant bucket metadata entry is updated. Care should be used when giving modify access permissions, because a shared user could then revoke the original user's permissions. Upon a GeoShare user upload, download, or update request, the metadata and access control service always checks if the user has permission to perform the action.

Managing access control is simple enough within our metadata service, but access must also be carefully controlled within the backing cloud storage service itself in case the necessary metadata is already known. Most of these services have extremely flexible object-level access control, but to generalize our GeoShare system, we only assume that there is a directory-level, or related-level, control. The same permissions described earlier for the metadata and access control service are applied to every object uploaded to the backing cloud store. Therefore, even if an attacker knew the locations of an object replica and enough key fragments, they would also need the backing cloud account information of a user with permission to retrieve them.

C. Client

One of the most important goals for our client interface is ease of access. The client should be able to access or modify stored sensitive objects from any standard computer without installing additional software or plugins. It is a given that every computer has at least one Web browser installed, and the most common Web browsers (Google Chrome, Mozilla Firefox, Microsoft Internet Explorer, etc) support JavaScript. As a result, we chose to implement the client part of the

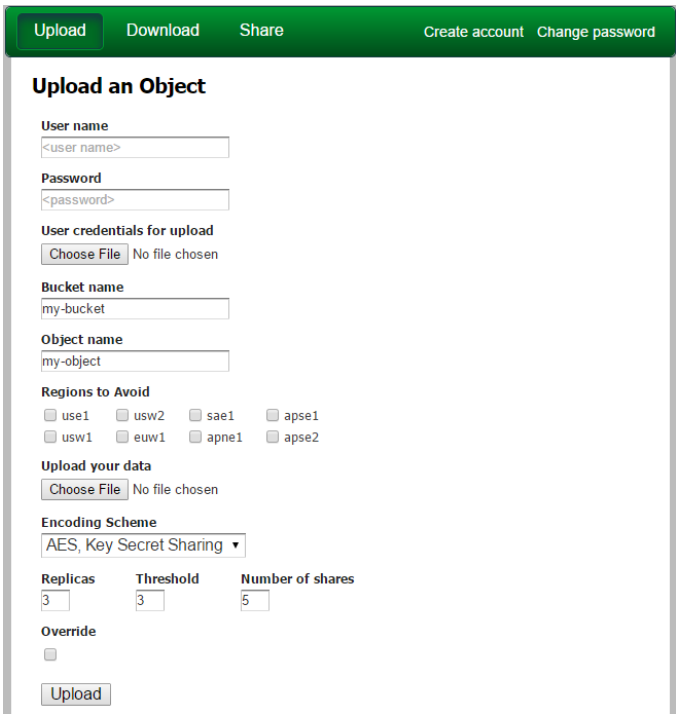

Fig. 2. Client web interface for uploading an object.

GeoShare service as a set of Web pages with all required functions written in JavaScript.

For any request, the client fills out and submits a simple Web form with parameters, such as their GeoShare account information, object name, and placement constraints. Our JavaScript code generates and sends the appropriate request to the metadata and access control service as a standard XMLHttpRequest. In the case of an upload, the object key is simultaneously encoded into fragments using the CryptoJS [13] and secrets.js [14] JavaScript libraries. For a typical upload or download, the metadata and access control service returns the placements of the new or existing key fragments and object replicas. An example of the Web form for an upload operation is shown in Figure 2.

All communication with the backing cloud storage service, Amazon S3, is done using the AWS JavaScript SDK [15]. If an object is being updated with new parameters (e.g. a new encoding scheme or placement preferences), the new key fragment or object replica placements may be different than the existing placements. In this case, the client code first deletes any obsolete fragments and replicas from the appropriate S3 regions before uploading the new fragments and replicas to the new locations. In the case of a download, data pieces are individually downloaded from the appropriate S3 regions and then used to reconstruct the original object at the client.

As mentioned in Section IV-B5, the S3 access control is a significant part of how we maintain the secure separation of the metadata and access control service and the cloud storage service. A client's provided S3 credentials are used to configure and authenticate these communications to use the client account, and are at no point available to the metadata and access control service. When a client creates a bucket, the client software creates a default bucket policy such that only that user may access or modify it, and any objects it contains[16]. If a client wishes to share their bucket with another user, the client modifies the S3 bucket's policy to allow the other user the appropriate access to the bucket.[2] By default, only the original creator of the bucket will have permission to share it with other users, but it is a simple policy change to allow others this control, though a complete transfer of ownership is more complex and we have not included that capability in our current implementation.

V. Full System Test and Evaluation

In this section, we report on various tests that we performed to evaluate the GeoShare system. For some of these, we did complete experiments using the AWS platform. For others, we extracted baseline data from a set of AWS experiments and used these data to extrapolate results. The baseline data we collected was a set of inter-region latencies determined as follows. We deployed clients all over the world using an EC2 client that we could place in any region world-wide. These clients created and uploaded 100 32 KB objects to each of the 8 S3 regions, and then downloaded all of the objects they had previously uploaded. From these tests, we computed the average download latency for each client region, server region pair. These 64 average latencies formed the baseline data set used in some experiments.

A. Evaluation of Replica Placement

While data is typically replicated within a region in a commercial cloud storage system, there is no attempt to improve global access to the data nor to tolerate scenarios where an entire region's datacenters go off-line simultaneously. The replica placement algorithm presented in Section IV-B3 is designed to provide the optimal replica locations given a pre-set number of replicas and request distribution. We compared the optimal placement to two simple algorithms: placing replicas in random regions (random), and placing a replica in the home region while distributing the rest randomly to the remaining regions (home+random).

We used the inter-region latency data described previously to extrapolate the overall average latencies for a variety of user demand distributions and placement strategies using 3 replicas. First, the placements of the 3 replicas were generated using our optimal placement algorithm, random placement, and home+random placement. Then, using these placements, the latency seen by a client in each region was determined based on the latency to the closest region containing a replica. Finally, the overall average latencies were calculated by weighting the selected inter-region latencies according to the user demand from each region. Figure 3 shows these average latencies, seen by users with 3 replicas and the following user demand distributions, from left to right:

- 75% from use1, 25% from other regions
- 50% use1, 25% apne1, 25% other
- 40% use1, 20% apne1 20% euw1, 20% other
- 30% use1, 70% other

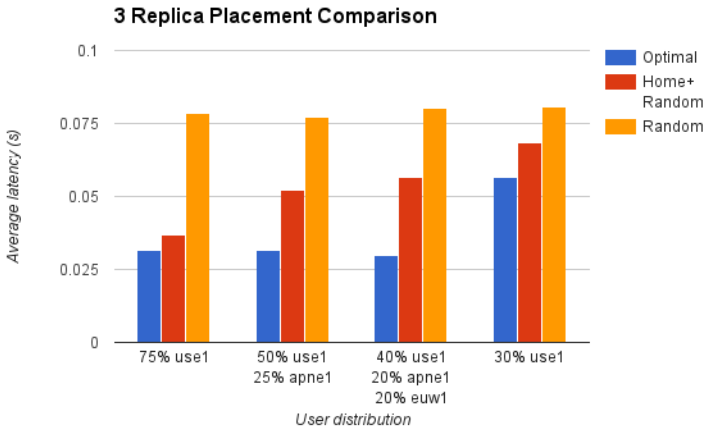

Fig. 3. Average downloading latencies with optimal, home+random, and random replica placement strategies.

It is clear that our optimal placement algorithm produces latencies that are significantly lower than random or home+random placements. The home+random strategy is an improvement over a purely random placement, but because neither takes into account inter-region latencies when placing the remaining replicas, the optimal placement will always be better in cases with more than one replica. The optimal placement algorithm also takes advantage of additional user demand distribution knowledge, but even in the cases of 75% use1 and 30% use1, where optimal and home+random are making use of the same distribution, optimal still outperforms home+random.

B. Evaluation of Utility Function

The utility function described in Section IV-B4 can help a GeoShare client decide how many replicas they should use based on their storage and latency preferences. The number of recommended replicas should ideally increase as the client's desire for faster overall downloads increases, and decrease as the desire for a smaller storage footprint increases. For these experiments, 32 KB objects were encoded using AES-SSS$(3, 5, r)$,[3] $r \in \{1, \ldots, 8\}$. For each value of r, 8 different user demand distributions were considered, resulting in 64 experiments. A single experiment, with a particular user demand distribution and value of r, consisted of the following:

1) one client was placed in each of the 8 regions where they created and uploaded 100 objects according to the random key fragment placement and optimal replica placement algorithms,
2) clients in each region downloaded the shares for each of the 800 total keys and one replica of each of the 800 objects, accessing the closest replica for each object, and
3) the average latency for each client download region was calculated.

[2]Should a user want to have per object policies, she can simply create a unique bucket for each object she owns.

[3]3-out-of-5 Shamir's secret sharing (SSS) for encryption keys and r replicas of the encrypted object (including the original copy).

TABLE I. UTILITY FOR VARIOUS PRIORITIES (25% APSE1)

	(Latency, Storage) Priority				
	(0.25,0.75)	**(0.4,0.6)**	**(0.5,0.5)**	**(0.6,0.4)**	**(0.75,0.25)**
1	0.85846918	0.77355069	0.71693837	0.66032604	0.57540755
2	0.81740433	0.79356121	0.77766580	0.76177038	0.73792727
3	0.74955165	0.77071121	0.78481759	0.79892396	0.82008352
4	0.65437212	0.70413825	0.73731567	0.77049309	0.82025922
5	0.54294727	0.61157278	0.65732311	0.70307345	0.77169896
6	0.44028480	0.53302711	0.59485532	0.65668352	0.74942583
7	0.33813854	0.45530738	0.53341993	0.61153249	0.72870133
8	0.23641422	0.37826275	0.47282843	0.56739412	0.70924265

For each value of r and a particular user demand distribution, the total average latency was computed by weighting the experimental latencies according to the demands from each region. These average latencies and numbers of replicas were then input to the utility function specified in Section IV-B4.

Table I illustrates the overall utility produced for a representative case, in which 25% of the demand comes from apse1 and the rest is evenly distributed across the remaining regions, with several different weightings of storage cost and latency. The shaded values represent the optimal utility for each different weighting. We can see from the table that as more weight is placed on latency, and thus less on storage cost, the number of replicas increases. When storage cost is the clear priority, e.g. with a storage weight of 0.75, one replica is optimal. When storage cost and latency have equal weights, 3 replicas become optimal. The optimal replica count is further increased to 4 as the latency weight is increased to 0.75. Note that, in this case, when weighting latency 3 times as high as storage cost, only half of the 8 regions will have replicas stored in them. This is an indication that excellent global performance can be achieved in S3 with relatively low replication cost.

C. Evaluation of Object Distribution

One thing that we were interested to see is what kind of distribution of object replicas across regions was produced by the combination of our proposed methods. The user first applies our utility function to find the best number of replicas for their latency and storage preferences, then places each replica as specified by our deterministic object replica placement algorithm described in Section IV-B3. Assuming the GeoShare service is used by a variety of clients that are uniformly spread across all 8 regions, and that 50% of demand for each object comes from its home region, we were able to calculate the overall proportion of replicas in each region. We used the previously described inter-region latency data and corresponding numbers of replicas as inputs into our utility function for 5 different combinations of latency and storage weights, and the 8 different user demand distributions. The number of replicas with the largest utility for each user demand distribution was selected, and the optimal placement of these replicas was calculated using our optimal placement algorithm.

When we consider all the replicas produced by these 5 different weights, and 8 user demand distributions having the "home" in each region, we get the replica distributions seen in Table II. When storage costs are minimized ($w_s = 0.75$), a single replica is used, and placed in the "home" region, resulting in a uniform object distribution. As storage cost becomes less of a priority, some user demand distributions indicate

TABLE II. OBJECT DISTRIBUTION ACROSS REGIONS WITH BEST UTILITY AND OPTIMAL PLACEMENT

Weights (lat,stor)	"Home" Location							
	apne1	**apse1**	**apse2**	**euw1**	**sae1**	**use1**	**usw1**	**usw2**
(0.25, 0.75)	0.13	0.13	0.13	0.13	0.13	0.13	0.13	0.13
(0.4, 0.6)	0.23	0.08	0.08	0.08	0.08	0.23	0.08	0.15
(0.5, 0.5)	0.27	0.07	0.13	0.07	0.07	0.20	0.07	0.13
(0.6, 0.4)	0.29	0.06	0.12	0.06	0.06	0.24	0.06	0.12
(0.75, 0.25)	0.19	0.12	0.12	0.12	0.12	0.15	0.04	0.15

TABLE III. KEY FRAGMENT DISTRIBUTION FOR EACH CLIENT

	Client Location							
	apne1	**apse1**	**apse2**	**euw1**	**sae1**	**use1**	**usw1**	**usw2**
apne1	119	112	85	55	49	86	71	76
	23.8%	22.4%	17.0%	11.0%	9.8%	17.2%	14.2%	15.2%
apse1	80	115	67	53	33	37	60	53
	16.0%	23.0%	13.4%	10.6%	6.6%	7.4%	12.0%	10.6%
apse2	73	65	127	58	52	59	76	62
	14.6%	13.0%	25.4%	11.6%	10.4%	11.8%	15.2%	12.4%
euw1	51	36	37	131	66	72	62	72
	10.2%	7.2%	7.4%	26.2%	13.2%	14.4%	12.4%	14.4%
sae1	30	38	39	44	157	58	47	47
	6.0%	7.6%	7.8%	8.8%	31.4%	11.6%	9.4%	9.4%
use1	50	35	63	66	54	102	40	44
	10.0%	7.0%	12.6%	13.2%	10.8%	20.4%	8.0%	8.8%
usw1	57	45	47	34	34	41	78	73
	11.4%	9.0%	9.4%	6.8%	6.8%	8.2%	15.6%	14.6%
usw2	40	54	35	59	55	45	66	73
	8.0%	10.8%	7.0%	11.8%	11.0%	9.0%	13.2%	14.6%

maximum utility at 2 or 3 replicas. In these cases, a replica is always stored in the "home" region, and the additional replicas are often stored in the regions apne1 and use1, which generally provide the best overall latencies to other regions. These two regions therefore accumulate a higher percentage of replicas than other regions. Once lower latency becomes a much stronger preference, more replicas are recommended, and are naturally spread much more uniformly throughout all regions. It should be noted that in nearly all cases, usw1 is the least-used region. This is because usw1's latencies to other regions are always slightly longer than those of usw2. With a limited number of replicas, it would not make sense to place a replica in usw1 in this case, since usw2 is always a better location.

D. Evaluation of Fragment Distribution

We also consider how key fragments are geographically distributed in our approach. While keys are only 40 bytes, and therefore have much less of a storage impact than full objects, the traffic generated by the simultaneous fragment requests is worth considering. Unlike our evaluation of the object replica distribution, key fragment placement is non-deterministic. We deployed one client in each of the 8 regions and had them simultaneously create and upload 100 objects using the AES-SSS$(3, 5, 3)$ encoding scheme, biasing the key fragment placement towards the regions with lowest latencies for the uploading client. Table III shows the resulting distribution of key fragments for a client in each of the 8 regions. To study the impact of grouping regions, we grouped the 3 U.S. regions when calculating key fragment placements. This imposed the constraint that no more than 2 key fragments may be placed across all U.S. regions combined.

TABLE IV. TOTAL FRAGMENT DISTRIBUTION ACROSS REGIONS

apne1	**apse1**	**apse2**	**euw1**	**sae1**	**use1**	**usw1**	**usw2**
653	498	572	527	460	454	409	427
16.3%	12.5%	14.3%	13.2%	11.5%	11.4%	10.2%	10.7%

We see from the table that the distribution of key fragments for a particular client can be fairly uneven across regions. For example, a client in EuropeWest stores about 26% of its key fragments in its own region and only about 7% in USWest1. Clients in other regions show similar variations. We also see the effect of grouping regions in this data. Even though the U.S. regions are the 3 regions with lowest latencies for clients in the U.S., the 3rd, 4th, and 5th fragments for a particular key must be stored outside the U.S. and so the other regions all have slightly higher likelihood of receiving fragments from U.S. clients than would be predicted based solely on latencies. For clients in non-U.S. regions, from 51 to 56 percent of key fragments are stored in the 3 fastest regions combined. For clients in the U.S., only 37–38% of key fragments are stored in the U.S. regions and the remaining 62–63% are stored outside of the U.S. Note that the percentage of key fragments stored inside the U.S. cannot exceed 40%, because that would imply that some keys have more than two shares in the U.S., which is not allowed by the grouping method and the $(3, 5)$ secret sharing scheme that is employed.

When we consider all of the key fragments produced by the 8 clients, the geographic distribution becomes much closer to a discrete uniform distribution. The overall geographic distribution of key fragments is shown in Table IV. In this table, we see that each region ends up with between 10.2% and 16.3% of all key fragments in the system. In a perfect discrete uniform distribution, each region would store exactly 12.5% of the fragments. While the actual distribution is not perfectly uniform, it is much closer to uniform than the distributions for individual clients. Other interesting items of note from Table IV include that AsiaPacificNortheast stores the largest percentage of fragments due to its relatively low latency to all other regions, and that the U.S. regions are the 3 most lightly loaded regions due to the fact that they are grouped and cannot store more than two fragments total for any one object.

Overall, we conclude that our key fragment placement algorithm, when applied on clients that are well spread across the world, does a good job of storing fragments in regions with low latency, while at the same time balancing load across all of the regions.

E. Evaluation of Region Outage Tolerance

In addition to reducing latencies for geographically-distributed clients, replicating objects across regions has the added benefit that outage of an entire region can be tolerated. In these experiments, we chose to simulate outage of the USWest2 region because it was the most used replica location over all tested user distributions and number of replicas. Similar to the Section V-A evaluation, inter-region latency data was used to calculate the average latencies both with and without the USWest2 region replicas. Figure 4 shows the average latencies with the same demand distributions as the earlier evaluation when 3 replicas are used before and after

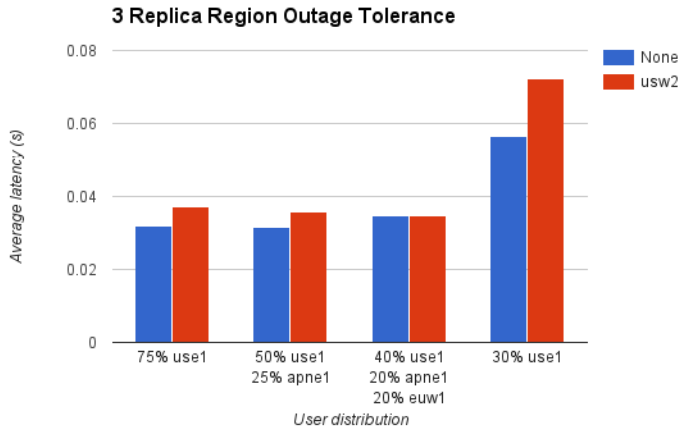

Fig. 4. Average Downloading Latencies with and without USWest2 Region for 3 Replicas

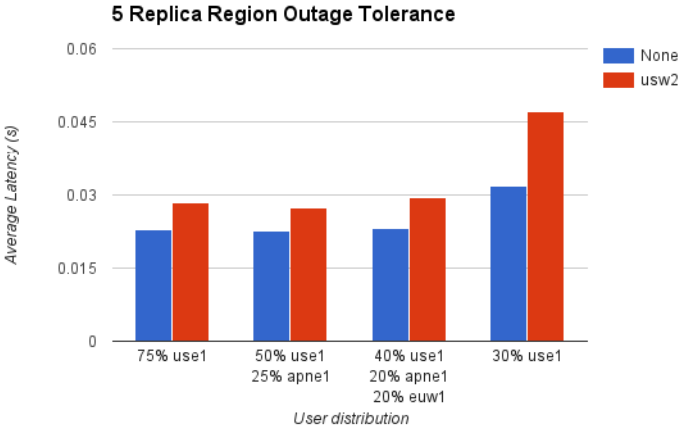

Fig. 5. Average Downloading Latencies with and without USWest2 Region for 5 Replicas

the outage of USWest2. Figure 5 shows the average latencies with the same distributions, but when 5 replicas are used.

We can see from the figures that, while there is a moderate latency impact, good performance is still achieved even with the complete outage of a region. In comparing the two figures, the latencies when using 5 replicas are always better than with 3 replicas, regardless of region functionality. The impact may become more severe when using fewer replicas, as each replica plays a more important role, but there is also the possibility of a replica not being placed in the downed region at all. This can be seen in the 40% use1, 20% apne1, 20% euw1 distribution with 3 replicas, where the average latency is unaffected by the loss of USWest2 because no replica was placed there.

VI. Conclusions

We have presented GeoShare, a cloud storage service that allows users to spread information across administrateive and geographic boundaries to prevent powerful organizations from deciphering their data stored in the cloud. A prototype service has been built on top of Amazon Web Services making use of 8 distinct geographic regions provided by Amazon S3 cloud storage service. Extensive evaluations demonstrated the usefulness of our approach. In particular, we demonstrated that users' geographic constraints can be fully satisfied while optimizing latency and storage cost.

Acknowledgements

This research was supported in part by the National Science Foundation under Grant IIP–1230740.

References

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

[2] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "RACS: A case for cloud storage diversity," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010, pp. 229–240.

[3] K. Bowers, A. Juels, and A. Oprea, "HAIL: A high-availability and integrity layer for cloud storage," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009, pp. 187–198.

[4] H. Chen and P. Lee, "Enabling data integrity protection in regenerating-coding-based cloud storage," in *Proceedings of the 31st IEEE Symposium on Reliable Distributed Systems*, 2012, pp. 51–60.

[5]

[6] C. Wang, Q. Wang, K. Ren, N. Cao, and W. Lou, "Toward secure and dependable storage services in cloud computing," *IEEE Transactions on Services Computing*, vol. 5, no. 2, pp. 220–232, 2012.

[7] Y. Tang, P. Lee, J. Lui, and R. Perlman, "Secure overlay cloud storage with access control and assured deletion," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 6, pp. 903–916, 2012.

[8] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and secure storage in a cloud-of-clouds," *ACM Transactions on Storage*, vol. 9, no. 4, pp. 12:1–12:33, 2013.

[9] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall, "MetaSync: File synchronization across multiple untrusted storage services," in *Proceedings of the USENIX Annual Technical Conference*, 2015, pp. 83–95.

[10] Y. Singh, F. Kandah, and W. Zhang, "A secured cost-effective multi-cloud storage in cloud computing," in *Proceedings of Infocom Workshops*, 2011, pp. 619–624.

[11] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 292–308.

[12] A. Santos, "Solving large p-median problems using a Lagrangean heuristic," ISIMA, Tech. Rep., 2009.

[13] "CryptoJS," https://code.google.com/p/crypto-js/, Accessed: September 15, 2014.

[14] "Secret sharing for JavaScript," https://github.com/amper5and/secrets.js/, Accessed: September 15, 2014'.

[15] "AWS SDK for JavaScript in the Browser," http://aws.amazon.com/sdk-for-browser/, Accessed: September 15, 2014.

[16] "Managing Access Permissions to Your Amazon S3 Resources," http://docs.aws.amazon.com/AmazonS3/latest/dev/intro-managing-access-s3-resources.html, Accessed: November 17, 2014.