Giza: Erasure Coding Objects across Global Data Centers

Yu Lin Chen, ***NYU & Microsoft Corporation;*** **Shuai Mu and Jinyang Li,** ***NYU;***
Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips, ***Microsoft Corporation***

https://www.usenix.org/conference/atc17/technical-sessions/presentation/chen-yu-lin

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the
2017 USENIX Annual Technical Conference
is sponsored by USENIX.

Giza: Erasure Coding Objects across Global Data Centers

Yu Lin Chen[⋆†], Shuai Mu[⋆], Jinyang Li[⋆], Cheng Huang[†], Jin Li[†], Aaron Ogus[†], Douglas Phillips[†]
[⋆]*New York University,* [†]*Microsoft Corporation*

Abstract

Microsoft Azure Storage is a global cloud storage system with a footprint in 38 geographic regions. To protect customer data against catastrophic data center failures, it optionally replicates data to secondary DCs hundreds of miles away. Using Microsoft OneDrive as an example, this paper illustrates the characteristics of typical cloud storage workloads and the opportunity to lower storage cost for geo-redundancy with erasure coding.

The paper presents the design, implementation and evaluation of Giza – a strongly consistent, versioned object store that applies erasure coding across global data centers. The key technical challenge Giza addresses is to achieve single cross-DC round trip latency for the common contention-free workload, while also maintaining strong consistency when there are conflicting access. Giza addresses the challenge with a novel implementation of well-known distributed consensus algorithms tailored for restricted cloud storage APIs. Giza is deployed to 11 DCs across 3 continents and experimental results demonstrate that it achieves our design goals.

1 Introduction

Microsoft Azure Storage is a global cloud storage system with a footprint in 38 geographic regions [27]. Since 2010, Azure Storage has grown from tens of petabytes to many exabytes, with tens of trillions of objects stored [15].

To protect customer data against disk, node, and rack failure within a data center (DC), Azure Storage applies Local Reconstruction Coding (LRC) [20] to ensure high availability and durability. LRC significantly reduces the storage cost over the conventional scheme of three-way replication.

To further protect customer data against catastrophic data center failures (say due to earthquake, tsunami, etc.), Azure Storage optionally replicate customer data to secondary DCs hundreds of miles away. It is essential to the customers that even in the unlikely, albeit inevitable, event of catastrophic data center failure, their data remain durable.

Geo-replication, however, doubles the storage cost. With many exabytes at present and exponential growth projected, it is highly desirable to lower the storage cost required for maintaining geo-redundancy.

1.1 Cross-DC Erasure Coding: Why Now?

Erasure coding across geographically distributed DCs is an appealing option. It has the potential to ensure durability in the face of data center failure while significantly reducing storage cost compared to geo-replication. The same economic argument that has driven cloud providers to erasure code data within individual data centers naturally extends to the cross-DC scenario.

However, when customer data is erasure coded and striped across multiple DCs, serving read requests would require data retrieval from remote DCs, resulting in cross-DC network traffic and latency. Furthermore, the recovery after catastrophic DC failure would trigger wide-area erasure coding reconstruction. While such reconstruction can be paced and prioritized based on demand, it nevertheless requires sufficient cross-DC network bandwidth to ensure timely recovery.

Therefore, cross-DC erasure coding only becomes economically attractive if 1) there are workloads that consume very large storage capacity while incurring very little cross-DC traffic; 2) there are enough cross-DC network bandwidth at very low cost.

For the former, Azure Storage indeed serves many customers with such workloads. Using Microsoft OneDrive as an example, Section 2 illustrates the characteristics of typical cloud storage workloads and why they are ideal for cross-DC erasure coding. For the latter, recent technological breakthroughs [26, 42] have dramatically increased bandwidth and reduced cost in cross-DC networking. For example, Facebook and Microsoft have teamed up to build *MAREA*, a new fiber optic cable under the Atlantic Ocean that will come online in 2017 with 160 Tbps capacity [12, 28]. The significant advancement in cross-DC networking is now making cross-DC erasure coding economically viable.

1.2 Challenges and Contributions

This paper presents Giza, a cloud object store that erasure codes and stripes customer data across globally distributed DCs. We aim to achieve two design goals. One, Giza should guarantee strong consistency while also minimizing operation latency. The other, Giza should make full use of existing cloud infrastructure to simplify its implementation and deployment.

Since reads and writes requires cross-DC communica-

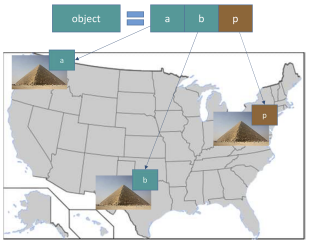
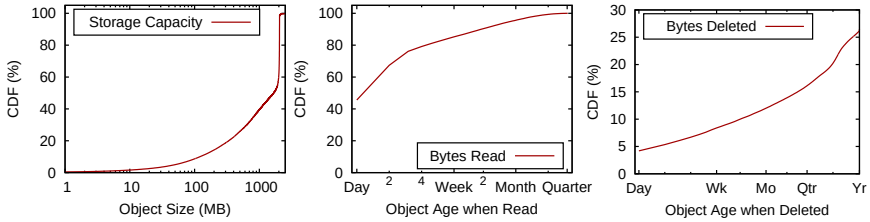


Figure 1: Storing Object in Giza



(a) **(b)** **(c)**

Figure 2: Microsoft OneDrive Characteristics

tion, latency is minimized when operations can complete within a single cross-DC roundtrip. This is possible to achieve for our target workloads (e.g. OneDrive), where objects are updated infrequently. Nevertheless, concurrent updates to the same object do exist. Furthermore, such conflicting access might originate from different DCs. Consider two concurrent requests updating the same object (with different data) from two separate DCs. Depending on network latency, the two requests may arrive at different data centers in different order. If not handled properly, this would result in data inconsistency.

To ensure strong consistency, one possible approach is to dedicate a primary DC that handles all updates and enforces execution order. However, requests from non-primary data centers have to be relayed to the primary first, incurring extra cross-DC latency, even when there are no concurrent updates. To guarantee strong consistency while minimizing latency in the common case, Giza employs FastPaxos [23], which incurs a single cross-DC roundtrip when there are no concurrent updates. When conflicting access do sometimes occur, Giza uses classic Paxos [24] and may take multiple cross-DC round trips to resolve the conflicts. We deem this to be an acceptable tradeoff.

We implement Giza on top of the existing Azure storage infrastructure. For each object, Giza stores its coded fragments in the Azure Blob storage of different DCs, and replicates its versioned meta-data containing the ids of coded fragments in the Azure Table storage of multiple DCs. Giza guarantees the consistency of versioned meta-data using Paxos/FastPaxos and it adapts both protocols to use the existing APIs of Azure Table storage.

To summarize, this paper makes the following contributions:

- We have designed and implemented Giza, a strongly consistent, versioned object store that erasure codes objects across globally distributed data centers.
- Giza achieves minimum latency in the common case when there are no concurrent conflicting access, and ensures strong consistency in the rare case under contention.
- Giza applies well-known distributed protocols–Paxos [24] and Fast Paxos [23] – in a novel way on top of restricted cloud storage APIs.
- Giza is deployed in 11 DCs across 3 continents and experimental results demonstrate that it achieves our design goals.

2 The Case for Giza

This section presents an overview of Giza, the characteristics of typical cloud storage workloads from Microsoft OneDrive, as well as the storage and networking tradeoffs exploited by Giza.

2.1 Giza Overview

Giza exploits the reduction in cross-DC bandwidth cost and leverages erasure coding to optimize the total cost of storing customer data in the cloud. It offers an externally strong consistent (linearizable [18]), versioned object store that erasure codes objects across global data centers.

Customers access Giza by creating Giza storage accounts. For each storage account, the customers have the flexibility to choose the set of data centers where their data are striped across. In addition, they can specify the erasure coding scheme. Giza employs classic $n = k + m$ Reed-Solomon coding, which generates m parity fragments from k data fragments. All n coded fragments are stored in separate DCs, which tolerates up to m arbitrary DC failures.

Figure 1 illustrates an exemplary flow of storing an object in Giza with 2 + 1 erasure coding. Giza divides the object into two data fragments (a and b) and encodes a parity fragment p. It then stores the coded fragments in 3 separate data centers.

Giza is accessible via *put*, *get*, and *delete* interface. In addition, Giza supports versioning. Each new *put* does not overwrite existing data, but rather creates a new version of the data. The old versions remain available until explicitly deleted.

2.2 Microsoft OneDrive Characteristics

Methodology: The data presented in this section is de-

rived from a three-month trace of the OneDrive service. OneDrive serves hundreds of millions of users and stores their objects which include documents, photos, music, videos, configuration files, and more. The trace includes *all* reads, writes, and updates to *all* objects between January 1 and March 31, 2016.

Large Objects Dominate: The size of the objects varies significantly, ranging from kilobytes to tens of gigabytes. While the number of small objects vastly exceeds that of large objects, the overall storage consumption is mostly due to large objects. Figure 2a presents the cumulative distribution of storage capacity consumption in terms of object size. We observe that less than 0.9% of the total storage capacity is occupied by objects smaller than 4MB. This suggests that, to optimize storage cost, it is sufficient for Giza to focus on objects of 4MB and larger∗. Objects smaller than 4MB can simply use the existing geo-replication option. This design choice reduces the overhead associated with erasure coding of small objects (including meta-data for the smaller object). As a result, all following analysis filter out objects smaller than 4MB.

Object Temperature Drops Fast: A common usage scenario of OneDrive is file sharing. Objects stored in the cloud are often shared across multiple devices, as well as among multiple users. Therefore, it is typical to observe reads soon after the objects are created. To this end, Figure 2b presents the cumulative distribution of bytes read in terms of object age when the reads occur†. It is worth pointing out that 47% of the bytes read occurred in the same day of object creation, 87% occurred within the same week, and merely less 2% occurred beyond one month. Since the temperature of the objects drops quickly, caching objects can be very effective (more below).

total reads (B) / writes (B)	2.3×	
cross-DC reads / writes with Giza	no caching	1.15×
	caching (day)	0.61×
	caching (week)	0.18×
	caching (month)	0.05×

Writes Dominate with Caching: The above table presents the effectiveness of caching. The ratio between the total amount of bytes reads to writes is 2.3×. As illustrated in Section 2.3, Giza incurs 1× and 0.5× cross-DC network traffic on writes and reads, respectively. Hence, the ratio between cross-DC traffic due to reads and writes is 1.15×. Given the temperature analysis, it is most effective for Giza to cache objects for a short period of time within one single DC. Serving reads from the caching DC dramatically reduces the cross-DC traffic due to reads. Indeed, when objects are cached for one day, the cross-DC traffic attribute to reads vs writes reduces to 0.61×. When objects are cached for one month, the ratio reduces to negligible 0.05×, in which case the cross-DC traffic is completely dominated by writes. Admittedly, caching the entire object also raises the total storage overhead to 2× (same as geo-replication) for a short period of time.

# of Versions	1	2	≥ 3
Percentage	57.96%	40.88%	1.16%

Concurrency is Rare, but Versioning is Required: The above table presents how often objects are updated and whether versioning is required. We observe that 57.96% of the objects are written once and never updated during the three-month period. For the remaining, 40.88% of the objects are updated exactly once and merely 1.16% are updated more than twice. In addition, we observe that only 0.5% of the updates are concurrent (within 1 second interval). This suggests that concurrent updates of same objects are rare in Giza (albeit possible).

Deletion is Not Uncommon: It turns out that OneDrive customers not only create new objects, but also delete old objects from time to time. To characterize how often objects are deleted and how long they have been stored upon deletion, we follow all the objects that were created during the first 3 months in 2016 and match them with object deletion trace up to one year after creation. For all the objects whose matching deletion trace records exist, we calculate the age of the objects upon deletion. Figure 2c plots the cumulative distribution of storage capacity consumption against object age‡.

We observe that a non-trivial portion of the objects were deleted within one year after their creation. These objects account for 26.5% of the total consumed storage capacity. On one hand, the amount of bytes deleted is much smaller than the total amount of bytes created, which partly explains the exponential growth of OneDrive's storage consumption. On the other hand, the percentage and amount of bytes deleted is non-trivial. This suggests that removing the deleted objects from underlining cloud storage and reclaiming capacity is crucial in achieving storage efficiency.

2.3 Giza Trade-offs

Giza offers flexible trade-offs in terms of storage cost and cross-DC network traffic, as summarized in Table 1. Although we cannot discuss the details of how Giza's trade-offs translate to overall cost reduction, our internal calculation indicates that Giza leads to savings of many millions of dollars annually for OneDrive alone.

∗Objects of tens of Gigabytes are divided into 4MB chunks before storing in cloud storage back-end.

†The analysis focuses on all the objects created during the three-month period. Hence, the object age is capped at three months.

‡The distribution curve is cut off at the right end, where the age of objects exceeds one year.

	Geo-Rep.	Giza		
# of DCs	2	3	5	7
Erasure coding	-	2 + 1	4 + 1	6 + 1
Storage overhead	2.6	1.9	1.6	1.5
Cost savings	-	**27%**	**38%**	**42%**
cross-DC traffic (put)	1x	1x	1x	1x
cross-DC traffic (get)	0	0.5x	0.75x	0.83x
DC rebuild	1x	2x	4x	6x

Table 1: Giza Trade-offs

Storage Cost: To tolerate single DC failure, geo-replication incurs the storage overhead of $2 \times 1.3 = 2.6$ (with single DC storage overhead at 1.3 [20]). With $k+1$ erasure coding, where k ranges from 2 to 6, Giza reduces the storage overhead to between 1.9 and 1.5, increasing cost savings from 27% to 42%. The storage cost savings come with inflated cross-DC traffic, examined below.

Cross-DC Traffic: For writes, Giza consumes same cross-DC traffic as geo-replication. With $k+1$ erasure coding, an object is encoded into $k+1$ fragments, where one fragment is stored in a local DC and the rest k in remote DCs. Hence, the ratio between cross-DC traffic and object size is $k/k = 1$, same as geo-replication. For reads, however, Giza consumes more cross-DC traffic. k fragments are required, where one is from the local DC and the rest $k-1$ from remote DCs. Hence, the ratio between cross-DC traffic and object size is $(k-1)/k$, which increases with k. In comparison, geo-replication serves reads entirely from the local DC and incurs no cross-DC traffic. However, as discussed in Sec **??**, the cross-DC read traffic can be cut down significantly with caching. Upon data center failure, Giza needs to rebuild lost data through erasure coding reconstruction, which requires k bytes of cross-DC traffic to reconstruct one byte of data. Geo-replication simply replicates every object and thus incurs 1× of cross-DC traffic.

Alternative Approach: Giza stripes individual objects across multiple DCs. This design leads to cross-DC traffic when serving reads. An alternative design is to first aggregate objects into large logical volumes (say 100GB) and then erasure code different volumes across multiple DCs to generate parity volumes [30]. Since every object is stored in its entirety in one of the DCs, cross-DC traffic is avoided during reads.

This design works great when objects are never deleted [30]. However, Giza must support deletion. Deleting objects from logical volumes (and canceling them from corresponding parity volumes) would result in complex bookkeeping and garbage collection, greatly increasing system complexity. In comparison, Giza keeps its design simple and relies on caching to drastically reduce the cross-DC traffic of reads to much lower than that of writes.

3 Design

This section presents the design of Giza, including the overall architecture, the data model, and the protocols for the *put*, *get*, and *delete* operations.

3.1 Overview and Challenges

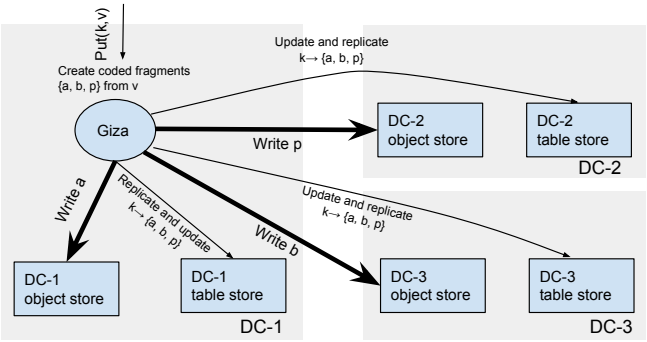


Figure 3: Giza architecture

Architecture Giza is a global-scale cloud storage system that spans across many data centers. It stores mutable, versioned objects. Figure 3 shows the architecture of Giza, which uses existing single-DC object and table stores. Giza stores an object through a *put* operation, consisting of a data operation and a metadata operation. These operations are executed in parallel to improve performance. On the data path, Giza splits and encodes the object into data and parity fragments. Each coded fragment is named by a unique identifier and stored in a different DC. Each update to the object creates a new version. The version numbers and the coded fragment IDs in each version constitutes the metadata of the object. On the metadata path, Giza replicates the metadata across the data centers.

Giza is implemented on top of the existing Azure Storage infrastructure. It stores the coded fragments in Azure Blob storage and the metadata in Azure Table storage. This layered approach provides two advantages. First, doing so allows the rapid development of Giza by reusing mature, deployed, and well-tested systems. Second, it simplifies the failure recovery and deployment: Giza runs on stateless nodes and can be readily integrated with the rest of the stateless cloud storage front-ends. Layering is commonly used in cloud infrastructure. For example, Percolator [32] supports transactions by layering over a fault-tolerant distributed table store.

Technical Challenges In Giza, each coded fragment is named by a unique identifier. As a result, fragments are immutable, which simplifies the data path.

The metadata path is more tricky, facing three main technical challenges:

1. *Building a strongly consistent, geo-replicated metadata store out of existing single-DC cloud tables.* Giza runs on stateless nodes and leverages existing well-tested cloud storage infrastructure to persist all data and metadata. The architecture simplifies development, deployment, and operation. This makes Giza quite different from other systems operating stateful servers (e.g., Cassandra, Megastore, Spanner, etc.). In addition, the cloud tables only guarantee consistency within single data center. Giza needs to orchestrate a collection of individual cloud tables across multiple data centers and achieve strong consistency globally.
2. *Jointly optimizing the data and metadata paths to achieve a single cross-DC round trip for read/write operations.* Most existing systems employ a primary-based approach, which incurs extra cross-DC round trip for secondary data centers. Giza, on the other hand, is leaderless and combines the data and metadata path in such a way that achieves single cross-DC round trips for both read and write from any data center.
3. *Performing garbage collection efficiently and promptly.* When a data object is deleted or its old versions are garbage collected, Giza must remove obsolete fragments and/or metadata from the underlying cloud blob and table storage. This turns out to be non-trivial because Giza's garbage collection mechanism must be able to handle data center failures while ensuring data consistency and durability.

3.2 Paxos using Cloud APIs

To address the above challenges, Giza adapts well-known distributed algorithms - Paxos and Fast Paxos - in a novel way on top of Azure Table.

3.2.1 Paxos and Fast Paxos in Giza: A Brief Primer

The Paxos algorithm [24] provides a mechanism to reach consensus among a set of *acceptors* and one or more *proposers*. A proposer initiates a Paxos voting process by first picking a distinguished *ballot*. All ballots are unique and can be compared to each other. The proposer sends requests and proposed values to the acceptors. Each acceptor decides whether to accept a request based on its own state. A proposed value is *committed* when it is accepted by a *quorum* of the acceptors. The acceptors update their states when a request or value is accepted.

Paxos is typically implemented via active acceptors, which are capable of comparing the ballot of incoming requests with their own states and deciding whether to accept the requests. Giza works differently and uses the cloud tables as the acceptors. It implements the acceptor logic leveraging Azure Table's *atomic conditional update* capability.

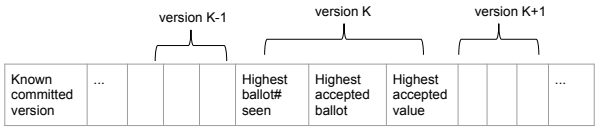


Figure 4: For each object, Giza stores the Paxos protocol state and the object metadata in a single row in the underlying cloud table.

Paxos takes 2 *phases* to reach consensus, where *phase* 1 prepares a ballot and *phase* 2 commits a value. Each phase takes 1 round trip, so applying Paxos in Giza results in 2 cross-DC round trips for the metadata path.

Fast Paxos [23] is a variation of Paxos that optimizes the performance over cross-DC acceptors. It employs two types of rounds: fast round and classic round. A fast round sends a PreAccept request and takes a single round trip to commit a value. A classic round resembles the two phases in Paxos and takes two round trips. The fast round in Fast Paxos requires a larger quorum. With 3 acceptors, a value is committed only when it is accepted by all the 3 acceptors (quorum size of 3). In comparison, Paxos is able to commit the value with 2 out of the 3 acceptors (quorum size of 2). The advantage of Fast Paxos is that when all the 3 acceptors respond, the value is committed in a single round trip. The requirement of larger quorum fits Giza perfectly, as Giza data path already requires storing fragments in 3 or more data centers.

Giza implements both Paxos and Fast Paxos. This paper discusses Fast Paxos only as its implementation requires more care (but achieves lower latency) than Paxos.

3.2.2 Metadata Storage Layout

Giza needs to persist the Paxos states together with the metadata for an object in the cloud table. We use one table row per object, with a dynamic number of columns, where each version of the object takes three columns. The layout of each table row is shown in Figure 4.

Each version is represented by a consecutive natural numbers, starting from 1. Every Giza write to the object creates a new version. For each version, Giza initiates a separate Paxos instance and uses Paxos to guard against races from concurrent writes and cloud table failures. The metadata of all versions and the states of all the Paxos instances are stored in the same table row. Specifically, the metadata contains a triplet of columns for each version (Figure 4). Two of the columns are Paxos states: `highest ballot seen` and `highest accepted ballot`. The other column, `highest accepted value`, stores the metadata, including the erasure coding scheme, the unique fragment IDs, and DCs that holds the fragments.

Giza additionally maintains a set of `known committed versions` for all those that have been

successfully committed. This is to facilitate both *put* and *get* operations, as discussed in the following sections.

3.2.3 Metadata Write - Common Case

The metadata path begins by choosing a proper new version number to initiate a Fast Paxos instance. Since version numbers need to be consecutive, the new version should succeed the most recently committed version. Giza identifies a proper version number in an optimistic fashion. Specifically, it reads `known committed versions` from the table in its local DC, then uses the next higher number as the new version number. In the uncommon case that the newly chosen version number has already been committed (but this DC missed the corresponding commit), the commit attempt would fail. Through the process, Giza learns the committed versions from the remote DCs, which allows it to choose a correct version number for retry.

Following Fast Paxos, Giza sends a PreAccept request to all the cloud tables, each located in a different DC. Each request is an *atomic conditional update* on the table row of the object. If there are no competing writes of the same object, the PreAccept request will succeed in updating the row. Otherwise, the PreAccept request will be rejected by the table and leave the row unchanged.

Whenever Giza receives a *fast quorum* of positive PreAccept responses, the corresponding version is considered to have been committed. Giza asynchronously sends a Commit confirmation to all the cloud tables to update the set of `known committed versions` to include the recently committed version. The Commit confirmation is again an atomic conditional update, which only succeeds if the version number is not yet included in the current set.

Since the Commit confirmation is completed asynchronously, the critical path only involves the PreAccept request and response. Hence, without conflict, the above described metadata write involves only one cross-DC round trip and is referred to as the *fast path*.

3.2.4 Metadata Write with Contention

The fast path may fail when Giza fails to collect a fast quorum of positive PreAccept responses. This may result from concurrent updates to the same object (contention), or because one or more cloud tables fail. In this case, Giza enters what is referred to as a *slow path* to perform classic Paxos in order to guarantee safety.

On the slow path, Giza first picks a distinguished ballot number and then replicates a Prepare request to write the ballot to all the metadata tables and wait for a majority of responses. The Prepare request is an atomic conditional update operation. The operation succeeds only if the `highest ballot seen` is no more than the ballot in the Prepare request. The operation also returns the entire row as a result.

Upon collecting a majority of successful replies, Giza needs to pick a value to commit. The rule for picking the value is categorized into three cases. In case 1, Giza looks for the highest accepted ballot in the replies. If there is one, the value from the reply is picked. In case 2, the replies contain no accepted value, but rather pre-accepted values. Giza picks the pre-accepted value returned by the maximum responses in the quorum. Both case 1 and 2 imply the possibility of an ongoing Paxos instance, so Giza picks the value so as to complete the Paxos instance first. It then starts with a new version and follows the fast path to commit its current metadata. In case 3, there is neither pre-accepted nor accepted value, which implies no real impact from contention. Giza picks its current metadata as the value and proceeds to the next steps.

Once Giza picks the value, it replicates an Accept request to all the metadata tables. The accept request is again an atomic conditional update; it succeeds in writing `highest accepted ballot` and `highest accepted value` if neither `highest ballot seen` nor `highest accepted ballot` is larger. As soon as a majority of Accept requests succeed, Giza considers the corresponding metadata write completed and sends acknowledgment to clients. Additionally, a Commit confirmation is replicated in the background, as described before.

3.2.5 Metadata Read

To get the metadata of the *latest* object version, it is *insufficient* for Giza to only read the corresponding metadata table row from its local DC. This is because the local DC might not be part of the majority quorum that has accepted the latest version. To ensure correctness, Giza needs to read the metadata rows from more than one DC.

In the common case, `known committed versions` is up-to-date and includes the latest committed version (say version k). Giza reads version k from the metadata table row in a local DC. It then confirms the lack of higher committed versions than k, from the metadata table row in a non-local DC. Hence, in the case that the metadata is replicated to 3 DCs, the metadata from 2 DCs (one local and one non-local) leads to a decisive conclusion that version k is the latest committed version. It is therefore safe for Giza to return version k to clients.

In general, Giza reads the metadata table rows from all the DCs. Whenever a majority rows have matching `known committed versions` and have not accepted any value for a higher version, Giza returns the metadata of the highest committed version.

If the replies contain an accepted value with a higher version number than the `known committed`

`versions`, Giza needs to follow a slow path similar to the one in the write operation. This is to confirm whether the higher version has indeed been committed.

3.3 Joint Optimization of Data and Metadata Operations

The naive version of Giza first writes out fragments (data and parity), and then writes out metadata, resulting in two or more cross-DC round trips. To reduce latency, we optimize Giza to execute the data and metadata paths in parallel. This is potentially problematic because either the data or metadata path could fail while the other one succeeds. Below, we describe how *put* and *get* cope with this challenge and ensure end-to-end correctness.

The *put* Operation: After generating the coded fragments, Giza launches the data and metadata paths in parallel. In the common case, Giza waits for both the data and the metadata paths to finish before acknowledging clients as well as replicating the commit confirmation. In other words, Giza ensures that `known committed versions` only include those whose data and metadata have both been successfully committed.

In one uncommon case, the data path succeeds, while the metadata path fails. Now, the fragments stored in the cloud blobs become orphans. Giza will eventually delete these fragments and reclaim storage through a cleaning process, which first executes Paxos to update the current version to *no-op*, discovers the orphan fragments as not being referenced in the metadata store, and then removes the fragments from the corresponding blob storage in all the DCs.

In another uncommon case, the data path fails, but the metadata path succeeds. This subtle case creates a challenge for the *get* operation, as addressed next.

The *get* Operation: A naive way to perform *get* is to first read the latest metadata and then retrieve the fragments. To reduce latency, Giza chooses an optimistic approach and parallelizes the metadata and the data paths.

For a *get* request, Giza first reads the metadata table row from a local DC. It obtains `known committed versions`, as well as the names and locations of the fragments of the latest version. Giza immediately starts reading the fragments from the multiple data centers. Separately, it launches a regular metadata read to validate that the version is indeed the latest. If the validation fails, Giza realizes there is a newer version. It in turn has to redo the data path by fetching a different set of fragments. This results in wasted efforts in its previous data fetch. Such potential waste, however, only happens when there is concurrent writes on the same object, which is rare.

Because the data and metadata paths are performed in parallel during *put*, it is possible (though rare) that the fragments for the latest committed version have not been written to the blob storage at the time of read. This happens if the metadata path in the *put* finishes before the data path, or the metadata path succeeds while the data path fails. In such case, Giza needs to fall back to read the previous version, as specified in `known committed versions`.

3.4 Deletion and Garbage Collection

The *delete* operation in Giza is treated as a special update of the object's metadata. When receiving a delete request (for either the entire object or specific versions), Giza executes the metadata path and writes a new version indicating the deletion. As soon as the metadata update succeeds, the deletion completes and is acknowledged.

The storage space occupied by deleted versions/objects is reclaimed through garbage collection. Giza garbage collection deletes the fragments from the blob storage and truncates the columns of the deleted versions from the metadata table row. It follows three steps: 1) fetching the metadata corresponding to the version to be garbage collected, 2) deleting the fragments in the blob storage, and 3) removing the columns of the deleted version from the metadata table row. The second step has to occur before the third one in case that the garbage collection process is interrupted and the fragments may become "orphans" without proper metadata pointing to them in the table storage.

Once all the versions of the object are deleted and garbage collected, Giza needs to remove the corresponding metadata table rows from all the DCs. This requires extra care, due to possible contention from a new *put* request. If the metadata table rows are removed brutally, the new *put* request may lead the system into an abnormal state. For instance, the *put* request could start at a data center where the metadata table row has already been removed. Giza would therefore assume that the object never existed and choose the smallest version number. Committing this version number is dangerous before the metadata table rows are removed from *all* the DCs, as this may result in inconsistency during future failure recovery.

Therefore, Giza resorts to a two-phase commit protocol to remove the metadata table rows. In the first phase, it marks the rows in all the DCs as `confined`. After this any other *get* or *put* operations are temporarily disabled for this object. In the second phase, all the rows are actually removed from the table storage. The disadvantage of this approach is obvious. It requires all the data centers to be online. Data center failure or network partition may pause the process and make the row unavailable (but can still continue after data center recovers or network partition heals).

4 Failure Recovery

Giza needs to cope with transient or permanent data center failures. Since Giza treats an entire data center as a fault domain, failures within a data center (server failures, network failures, etc...) are resolved by individual cloud object store and table store within each data center.

Transient DC failure: We broadly categorize transient DC failure to include temporary outages of the blob and table storage service in a DC. Transient DC failure may be caused by a temporary network partition or power failure. By design, Giza can still serve *get* and *put* requests, albeit at degraded performance. For example, when handling *put* requests, Giza may take more than one cross-DC round trip, because some of the DCs replicating the metadata are unavailable, resulting in fewer DCs than required for a fast path quorum.

When a data center recovers from transient failures, it needs to catch up and update the fragments in its blob storage and the metadata rows in its table storage. The process follows the Paxos learning algorithm [24]. For each object, Giza issues a read request of the metadata without fetching the fragments. If the local version matches the committed version, nothing needs to be done; if the local version is behind, the recovering process reads the fragments of all missing versions, reconstructs corresponding missing fragments and stores them in the blob storage, as well as updates the metadata row in the table storage.

Permanent DC Failure: Although extremely rare, a DC may fail catastrophically. The blob and table service within the DC may also experience long-term outages. We categorize these all as permanent DC failure.

Giza handles permanent DC failure by employing logical DC names in storage accounts. The mapping between a logical DC name to a physical DC location is stored in a separate service external to Giza. Upon a permanent DC failure, the same logical DC name is re-mapped from the failed DC to a healthy replacement. Giza metadata records logical DC names and therefore remains unchanged after the re-mapping. This is similar to DNS, where same domain name can be re-mapped to a different physical IP address. This way of handling failure is also reported in Chubby [8].

Upon the permanent DC failure, Giza launches recovery coordinators to reconstruct the lost fragments and re-insert the metadata rows in the replacement DC. The procedure is similar to how Giza handles transient failures yet may last longer. The reconstruction is paced and prioritized based on demand, with sufficient cross-DC network bandwidth in-place to ensure timely recovery.

	Coding	Data and Metadata DCs	Ping (max)
US-2-1	2 + 1	US(3/3)	46 ms
US-6-1	6 + 1	US(7/3)	71 ms
World-2-1	2 + 1	US(1/1), EU(1/1), JP(1/1)	240 ms
World-6-1	6 + 1	US(3/1), EU(2/1), JP(2/1)	241 ms

Figure 5: Giza Configuration (US(7/3) represents 7 DCs for data and 3 DCs for metadata, all in the US.)

5 Implementation

Giza is implemented in C++ and uses Azure Blob and Table storage to store fragments and metadata. The global footprint of Azure Storage allows for experimenting with a wide range of erasure coding parameters.

The Giza design relies on atomic conditional write. For Azure Table, we leverage its ETag mechanism. An unique ETag is generated by the table service for every write. To implement an atomic conditional write, a Giza node first reads the ETag of a table row. It then performs the condition check and issues the write request together with the ETag. Azure Table rejects the write request if the ETag in the request does not match the one in the table, which could only occur due to a concurrent write to the row.

To minimize latency, the Giza node *delegates* its conditional write requests to remote Giza nodes, which reside in the same DCs as the tables and act as proxies in reading the ETag and writing the local table row.

5.1 Experimental Setup

We run experiments using four configurations: US-2-1, World-2-1, US-6-1, and World-6-1. Figure 5 describes the data centers participating in each configuration, and the max ping latency between the DCs. Unless explicitly stated, all experiments erasure code objects of 4MB, the dominating size in our target workloads.

We also compare Giza with CockroachDB [9], an open source implementation of Google spanner. Our CockroachDB experiments use the US-2-1 configuration, as CockroachDB doesn't yet support world wide replication. In every data center, we run three CockroachDB instances for local DC replication. Each CockroachDB writes to a dedicated HDD with no memory caching. We have configured the CockroachDB instances following the recommended production setting by the CockroachDB developers. For example, we run NTP to synchronize clocks of the different CockroachDB instances.

6 Evaluation

For evaluation, we deploy Giza on top of the Microsoft Azure platform across 11 data centers (7 in North America, 2 in Europe and 2 in Asia). Giza nodes are Azure virtual machines with 16 cores, 56 GB of RAM, and Gigabit Ethernet. As describe in Section 3, all the Giza nodes are stateless. For each Giza storage account, a lo-

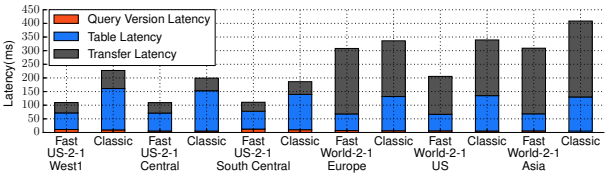
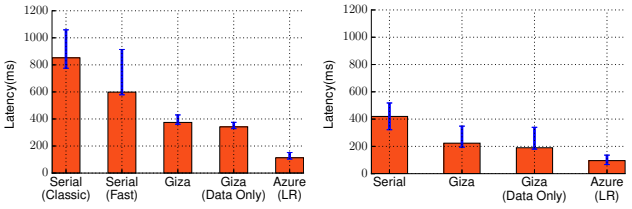


Figure 6: Fast Paxos and Classic Paxos Comparison



(a) Put **(b) Get**

Figure 7: Giza Overall Latency

cally redundant Azure Blob and Table storage account is created in every DC. Upon receiving *get* or *put* requests, the Giza nodes execute the data and the metadata paths to read or write objects.

6.1 Metadata Latency

We implement Giza's metadata path with both Classic and Fast Paxos. Here, we compare the performance of the two algorithms and examine their effects on Giza's metadata path latency. Figure 6 presents the metadata latencies and breakdowns for both US-2-1 and World-2-1 configurations. The results include running proposers in each of the DCs.

The metadata latency consists of three parts: query version latency, transfer latency, and table latency. The query version latency is determined by reading the possible highest version from the proposer's local table. This request is not part of the concensus protocol and is the same for both Fast and Classic Paxos. The transfer latency is the amount of time spent on network communication between the proposer and the furthest Giza proxy in a Paxos quorum. Here, the latency of Classic Paxos, which incurs two cross-DC round trips, is not strictly twice as much as the latency of Fast Paxos. This is because the Classic Paxos quorum is smaller than the Fast Paxos quorum. As a result, the distance between the proposer and the *furthest* proxy is smaller in a Classic Paxos quorum. The table latency is the latency for a Giza proxy to conditionally update its local DC table. Since Classic Paxos requires two rounds and hence two table updates, its table latency is twice that of Fast Paxos.

For the US-2-1 configuration, we observe that the metadata latency is dominated by table latency. In this case, Fast Paxos is much faster than Classic Paxos, regardless of the proposer's location.

For the World-2-1 configuration, transfer latency becomes a substantial part of the overall metadata latency. In this case, despite of taking two cross-DC round trips, the Classic Paxos implementation can have lower transfer latency. Nevertheless, the table latency of Classic Paxos is still twice that of Fast Paxos. As a result, the Fast Paxos implementation has lower latency, regardless of the proposer's location.

6.2 Giza Latency

The design of Giza went through multiple iterations and this section illustrates the performance gain for each iteration. For the interest of space, we focus on the World-2-1 configuration. All latency results include error bars representing the 10th and 95th percentile.

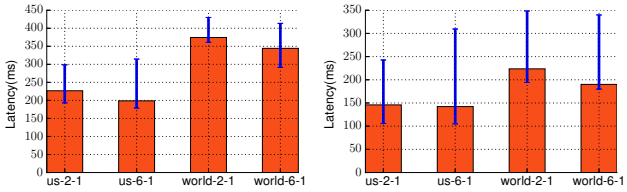
6.2.1 Giza *Put* Latency

Figure 7a shows the Giza overall *put* latency for 4MB data. We compare Giza with its two previous iterations where the metadata path is not parallelized with the data path. In the first iteration, Giza runs the data path first. After completing the data path, Giza runs the metadata path with the Classic Paxos implementation. In the second iteration, we replaced Classic Paxos with Fast Paxos, improving latency performance. Giza parallelizes metadata path with data path, which can results in extra metadata or data clean up if either path fails to complete. However, the performance gain is significant. We also included a baseline which is the time it takes a proposing data center to issue a blob store request to the farthest data center in the quorum. Finally, we include the latency for storing the 4MB data directly to Azure storage, which is locally replicated.

The results show that Giza's performance beats the other two alternatives in the common case and has closest latency to the baseline. The median latency of Giza's *put* is 374 ms, only 30 ms higher than the baseline. This is due to the latency of erasure coding 4MB data. On the other hand, the serial Paxos version takes 852 ms, and the serial Fast Paxos version takes 598 ms. In summary, the latency cost for tolerating data center failure with Giza is a little more than 3 time that of local replication.

6.2.2 Giza *Get* Latency

Figure 7b shows Giza's *get* performance comparison. The alternative design here is the non-optimistic *get* where the most current version for a blob is not assumed to be stored in the current data center. Hence, the metadata path and data path are executed sequentially, taking 419 ms. Giza's optimistic *get*, which runs the metadata path and data path in parallel, takes 223ms. Giza's *get* latency is higher than the baseline by 33 ms. The perfor-



(a) Put (b) Get

Figure 8: Performance for Giza in different setups

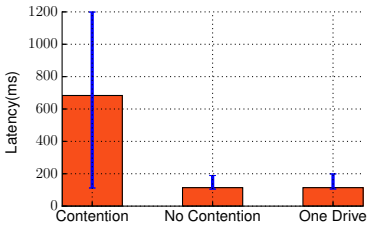


Figure 9: Contention vs No Contention

mance gap between Giza and baseline is higher because Giza needs to do a local table retrieval first before starting the datapath. In addition, it needs to decode the data fragments. Here the latency cost of erasure encoding on the read path with Giza is roughly twice that of reading from a locally redundant Azure storage.

6.3 Footprint Impact

Giza offers customers the flexibility to choose the set of data centers, as well as the erasure coding parameters (e.g., the number of data fragments k). It turns out that increasing k not only reduces storage overhead, but also overall latency. This is because the latency in Giza is often dominated by the data path. Erasure coding with a larger k results in smaller fragments and fewer bytes stored in each DC's blob storage. This reduces the data path latency and in turn the overall latency.

Figure 8a and Figure 8b present the latency impact given different Giza footprints and erasure coding parameters. All the requests are generated from US-Central. Comparing US-2-1 to US-6-1 (World-2-1 to World-6-1), it is clear that increasing k from 2 to 6 reduces the latency for both *put* and *get*.

6.4 Comparing Giza with CockroachDB

Ideally, we would like to compare Giza with an existing storage system with similar functionalities. However, there is no off-the-shelf erasure coded system. Hence, we implemented Giza on top of CockroachDB using its transaction support. To do this, we create four different tables in CockroachDB: one metadata table and three data tables (for storing coded fragments). The metadata table is replicated across all three DCs. Each of the data tables is replicated three times within its respective data center. This is to match the local replication of Azure Table within individual DCs.

We implement Giza's *put* as a transaction consisting of storing each coded fragment at the corresponding data table and storing the metadata information in the metadata table. Since CockroachDB is not optimized for storing large objects, we evaluate the performance of *put*s on 128KB objects. The median *put* latency of 128KB objects under CockroachDB is 333ms, much higher than that of Giza (<100ms).

We implement Giza's *get* as a transaction consisting of reading the metadata from the metadata table and two coded fragments from the data tables. The median *get* latency under CockroachDB is lower than that of Giza by 20%. This is because CockroachDB directly reads from local HDD, which is faster than Giza reading from Azure storage. To demonstrate this, we equalize the storage layer to substitute Azure latency with local HDD latency. Indeed, Giza's performance with equalized storage is slightly better than that of CockroachDB.

6.5 Giza Contention

Giza is optimized for low contention workloads. So, it employs a simple strategy for handling contention. In the event of contention, a Giza node that fails the fast round falls back to a classic round. In addition, Giza implements exponential back-off with the latency starting from the median cross-DC latency whenever prepare phase or accept phase further fails.

Figure 9 compares the performance of Giza driven by the OneDrive trace to that with no contention at all. In the OneDrive trace, only 0.5% of updates are concurrent (within 1 second interval). Hence, it is not surprising that the performance of Giza driven by the OneDrive trace is almost identical to that with no contention.

Figure 9 also presents the latency results of *adversary* contention. In this case, two Giza nodes within the same data center are issuing back-to-back concurrent puts to update the same object. This is definitely not the scenario that Giza targets. We include the results merely for the interest of our readers.

7 Related Work

Erasure Coding in Cluster Storage: Erasure coding has long been applied in many large-scale distributed storage systems [34, 41, 16, 1, 38, 35, 40], including productions systems at Facebook [6], Google [13, 14] and Microsoft Azure [20]. These solutions generalize the RAID approach [31, 39] to a distributed cluster setting. Giza is unique in synchronously replicating erasure coded data across WAN and minimizing cross-DC latency. In addition, Giza provides globally consistent

put and *get* with versioning support.

Erasure Coding in Wide Area Storage: HAIL [7], OceanStore [22, 33], RACS [2], DepSky [5] and NCCloud [19] all stripe and erasure code data at a global scale.

HAIL [7] is designed to withstand Byzantine adversaries. OceanStore [22, 33] assumes untrusted infrastructure and serializes updates via a primary tier of replicas. Giza operates in a trusted environment.

RACS [2] and DepSky [5] address conflicts caused by concurrent writers using Apache ZooKeeper [21], where readers-writer locks are implemented at per-key granularity for synchronization. Giza, on the other hand, implements consensus algorithms for individual keys and achieves strong consistency without centralized coordinators. In addition, Giza employs a leaderless consensus protocol. Updates may originate from arbitrary data centers and still complete with optimal latency without being relayed through a primary.

NCCloud [19] implements a class of functional regenerating codes [11] that optimize cross-WAN repair bandwidth. Giza employs standard Reed-Solomon coding and leaves such optimization to future.

Facebook f4 [30] is a production warm blob storage system. It applies erasure coding across data centers for storage efficiency. As discussed in Section 2.3, f4 avoids the deletion challenge by never truly deleting data objects. Whenever a data object is deleted, the unique key used to encrypt the object is destroyed while the encrypted data remains in the system. This simplification suits Facebook very well, because its deleted data only accounts for 6.8% of total storage and Facebook could afford not to reclaim the storage space [30]. This, unfortunately, is not an option for Giza, as our workloads show much higher deletion rate. Not reclaiming the physical storage space from deleted data objects would result in significant waste and completely void the gain from cross-DC erasure coding. Furthermore, not physically deleting customer data objects - even if encrypted - wouldn't meet the compliance requirements for many of our customers.

Separating Data and Metadata: It is common for a storage systems to separate data and metadata path, and design a separate metadata service to achieve better scalability, e.g., FARSITE [3] and Ceph [37]. Gnothi [36] replicates metadata to all replicas while data blocks only to a subset of the replicas. Cocytus [40] is a highly available in-memory KV-store that applies replication to metadata and erasure coding to data so as to achieve memory efficiency. Giza follows a similar design path, and store data in commodity cloud blob storage and metadata in commodity NoSQL table storage.

Consistency in Global Storage: Megastore [4] and Spanner [10] applies Multi-Paxos to maintain strong consistency in global databases. Both of them requires two round trips for a slave site to commit. Mencius [25] takes a round-robin approach for proposers in different sites, amortizing commit latency. EPaxos [29] uses fine-grained dependency tracking at acceptor-side to ensure low commit latency for both non-contended and contended requests. In comparison, Giza takes a refined approach based on FastPaxos [23], separating metadata and data path before committing. This design choice allows Giza to serve most requests still in single cross-DC round trip while keeping servers stateless, using the limited ability of table service. Metasync [17] implements Paxos using the append functionality provided by cloud file synchronization services such as DropBox, OneDrive. By contrast, Giza implements Paxos using conditional-write APIs of cloud tables. The latter leads to a more efficient implementation as clients do not need to download and process logs from the cloud storage in order to execute Paxos.

8 Conclusion

In this paper, we present the design and evaluation of Giza – a strongly consistent, versioned object store that encodes objects across global data centers. Giza implements the Paxos consensus algorithms on top of existing cloud APIs and have separate data and metadata paths. As a result, Giza is fast in normal operation for our target workloads. Our evaluation of Giza on a deployment over 11 DCs across 3 continents demonstrates that Giza achieves much lower latency than naively adopting a globally consistent storage system.

Acknowledgments

We thank Andy Glover, Jose Barreto, Jon Bruso, Ronakkumar Desai, Joshua Entz from the OneDrive team for their many contributions. Special thanks go to Jeff Irwin for his contributions that helped enable Giza. We also thank all of the members of the Azure Storage team for invaluable discussions and iterations, as well as Taesoo Kim and anonymous reviewers for their insightful feedback. This work was partially supported by ONR grant N00014-16-1-2154.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, et al. Ursa minor: versatile cluster-based storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. Dec. 2005.

[2] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon RACS: a case for cloud storage diversity. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*. June 2010.

[3] 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. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Dec. 2002.

[4] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh Megastore: providing scalable, highly available storage for interactive services. In *Proceedings of Biennial Conference on Innovative Data Systems Research (CIDR)*. Jan. 2011.

[5] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa DepSky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*. Apr. 2011.

[6] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling Hdfs raid. In *Hadoop user group meeting*. 2010.

[7] K. D. Bowers, A. Juels, and A. Oprea HAIL: a high-availability and integrity layer for cloud storage. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*. Nov. 2009.

[8] M. Burrows The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Nov. 2006.

[9] *CockroachDB*. http://www.cockroachlabs.com/.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Oct. 2012.

[11] R. G. Dimakis, P. B. Godfrey, Y. Wu, M. O. Wainwright, and K. Ramch Network coding for distributed storage systems. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*. 2007.

[12] *Facebook and Microsoft to Build Fiber Optic Cable Across Atlantic*. http://www.wsj.com/articles/facebook-and-microsoft-to-build-fiber-optic-cable-across-atlantic-1464298853. May 2016.

[13] A. Fikes Storage architecture and challenges. *Talk at the Google Faculty Summit* (2010).

[14] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan Availability in globally distributed storage systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Oct. 2010.

[15] A. Greenberg SDN for the cloud. In *Keynote in the 2015 ACM Conference on Special Interest Group on Data Communication*. 2015.

[16] A. Haeberlen, A. Mislove, and P. Druschel Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*. May 2005.

[17] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall MetaSync: file synchronization across multiple untrusted storage services. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*. July 2015.

[18] M. P. Herlihy, and J. M. Wing Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[19] Y. Hu, H. Chen, P. Lee, and Y. Tang NCCloud: applying network coding for the storage repair in a cloud-of-clouds. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. Feb. 2012.

[20] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in Windows Azure storage. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*. June 2012.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*. June 2010.

[22] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Nov. 2000.

[23] L. Lamport *Fast Paxos*. Tech. rep. MSR-TR-2005-112. Microsoft Research, 2005.

[24] L. Lamport Paxos made simple. *ACM SIGACT News* 32, 4 (2001), 18–25.

[25] Y. Mao, F. P. Junqueira, and K. Marzullo Mencius: building efficient replicated state machines for WANs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Dec. 2008.

[26] R. Mears, L. Reekie, S. Poole, and D. Payne Low-threshold tunable CW and Q-switched fibre laser operating at 1.55 μm. *Electronics Letters* 3, 22 (1986), 159–160.

[27] *Microsoft Azure Regions*. https://azure.microsoft.com/en-us/regions/.

[28] *Microsoft, Facebook to lay massive undersea cable*. http://www.usatoday.com/story/experience/2016/05/26/microsoft-facebook-undersea-cable-google-marea-amazon/84984882. May 2016.

[29] I. Moraru, D. G. Andersen, and M. Kaminsky There is more consensus in egalitarian parliaments. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*. Nov. 2013.

[30] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm BLOB storage system. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Oct. 2014.

[31] D. Patterson, G. Gibson, and R. Katz A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*. June 1988.

[32] D. Peng, and F. Dabek Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Oct. 2010.

[33] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz Pond: the OceanStore prototype. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. Mar. 2003.

[34] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence FAB: building distributed enterprise disk arrays from commodity components. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Oct. 2004.

[35] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur Xoring elephants: Novel erasure codes for big data. *The Proceedings of the VLDB Endowment (PVLDB)* 6, 5 (Mar. 2013).

[36] Y. Wang, L. Alvisi, and M. Dahlin Gnothi: separating data and metadata for efficient and available storage replication. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*. June 2012.

[37] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn Ceph: A scalable, high-performance distributed file system. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Nov. 2006.

[38] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou Scalable Performance of the Panasas Parallel File System. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. Feb. 2008.

[39] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)* 14, 1 (Feb. 1996).

[40] H. Zhang, M. Dong, and H. Chen Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. Feb. 2016.

[41] Z. Zhang, S. Lin, Q. Lian, and C. Jin RepStore: a self-managing and self-tuning storage backend with smart bricks. In *Proceedings of International Conference on Autonomic Computing*. 2004.

[42] B. Zhu, T. Taunay, M. Fishteyn, X. Liu, S. Chandrasekhar, M. Yan, J. Fini, E. Monberg, and F. Dimarcello 112-Tb/s space-division multiplexed DWDM transmission with 14-b/s/Hz aggregate spectral efficiency over a 76.8-km seven-core fiber. *Optics Express* 19, 17 (2011), 16665–16671.