Selective Data Replication for Geo-Distributed Blob Storage

Haoran Wang
University of Michigan

Arjun Khurana
University of Michigan

Rui Chen
University of Michigan

Abstract

User file storage is moving towards geo-distributed systems. Unfortunately, there are several challenges with regards to request latency. For our work, we focus on blob storage and show how a de-centralized approach improves access time over a centralized approach. Next, we come up with a metric that determines data popularity and implements a copy-popular policy, as opposed to the common copy-everything policy. We also propose a new consistency protocol, ReadClock, that helps manage geo-distributed state machine. Finally, we show how our policy improves the space consumption by $50\% - 66\%$ while achieves the same level performance with regard to end-to-end latency.

1 Introduction

Users are creating data exponentially with the increase in social network behaviors. Apps like Instagram, Twitter, and Facebook are among the few that allow billions of users to upload information instantly. The shift in real-time data has increased the need for cloud systems, but more importantly has increased the need for quick access of information. Some of these files include photos and videos, which are uploaded once, frequently accessed, never updated and rarely deleted. These files are called Binary Large OBjects (BLOBs) [20]. Since blobs are generated in an exponential rate and hundreds of millions of users view these massive media objects all around the world which puts a lot pressure on data availability.

In today's architecture, many tech corporations including Amazon, Google and Microsoft own their own DCs, which span the continental US to several cities across the globe. Both Industry and Academia have heavily considered the data availability problem into data center level. For better data availability, creating replicas is the popular and the main solution deployed in practice. It incurs two major problems. First, the replica allocation problem or determining how many replicas to allocate for each file. And second, the replica placement problem, placing data as close as possible to users. Traditionally, files are replicated in a static fashion, i.e. the data replication scheme is designed by the distributed manager and it remain fixed until manual reallocation is executed. For Ambry, any update to the cluster manager may need to restart the system. If the read-write patterns are fixed and are known a priority, this is a reasonable solution. However, if the read- write patterns change dynamically, in unpredictable ways, a static replication scheme may lead to several performance problems. We take a deep dive into a geo-distributed blob storage systems.

2 Motivation

The trending geo-distributed blob storage systems have obvious drawbacks that this project aims to tackle.

2.1 Copy-Everything

A popular strategy for newly written blobs is to blindly populate them to every data center, whatever popularity the blob has. Ambry [20], LinkedIn's geo-distributed system, enforced this policy in all three of their data centers. Haystack [20], Facebook's photo storage for hot data was doing so as well [7]. A more advanced approach it took was to build an auxiliary warm storage system, F4 [18], that stored data when it becomes cold, and implemented different set of services (e.g. Erasure Coding) that best fit the nature of cold data, very few read requests. However, when it comes to hot data, it still populated them globally. Sophisticated optimization such as batching tasks and caching data were used to reduce the cost of populating at such large scale, but there were still many wasted efforts as 95% of the files written are rarely accessed [21] – a lot of actually cold data will get populated as well. Such skew of popularity is a common access pattern for cloud storage system, where

the Zipf's distribution of access numbers is usually observed [21] [4] [18]. Our project intends to avoid copying cold data, but "copy-popular", thus saves networking and storage resources. One of the justifications for the copy-everything policy is that, from the angle of user experience, data availability is paramount especially when popularity cannot be predicted. Therefore we also intended to prove that these concerns can be mitigated by copying hot data only.

2.2 Popularity Measurement

Theoretically, the only accurate measurement for popularity of a certain piece of data is the number of read requests over a duration of time. Unfortunately, this fine-grained state machine is impossible to maintain in a large-scale storage system. F4 used a coarse-grained, indirect measurement instead – it gives each type of blob a time threshold (e.g. 3 month for photos) and blobs older than that threshold time are considered cold. This is to some degree effective and accurate in the context of social network data, where usually new is hot, old is cold. However, there are still giant number of false positives, as many newly written data are never accessed again. We need a solution that comes back to the measurement by counting read request, while avoiding bookkeeping too many states.

2.3 Geo-Distributed State Machine

There are many hardware-centric enhanced networking solutions such as Infiniband [6] and Myrinet [12] for intra-DC traffic. However, the end-to-end latency for transferring data in a geo-distributed system cannot be reduced dramatically by any hardware or software solutions, especially among cross-continent data centers. For instance, it takes 30-40ms for light to travel across the Pacific Ocean, let alone pyramids of necessary overheads from routing and networking/hardware abstractions. In this setting, an everything-goes-to-master-first model is definitely not an option as global requests cannot afford to always talk to a location-specific master first. Therefore, many solutions have static configurations for each data center with manual synchronization. Ambry preallocated partitions in each of its DCs, and when all of them are nearly full it needs to update cluster manager for each DC and scheduled secure restarts. However, populating data in a storage system is one of the background services that are not urgent/high-priority and that the latency of transferring and updating states is not a big concern, therefore implementing a geo-distributed state machine, that maintains the mapping of data to its logical or physical locations, is definitely necessary and practical.

3 System Overview

3.1 Partition-Level Read Counts

Many of the storage solutions used a log-like, append-only logical structure that organizes blobs into bigger units and increases write and sequential read throughput. F4 uses Volume while Ambry uses Partition, Azure [8] uses Partition and Stream (they separated intra- and inter- DC abstractions for replication), as their representation of this abstraction. A partition is read-write when not full, and read-only when full and they are the minimal unit for storage (cannot be spanned across physical disks). Size of each partition is typically 10G - 100GB, thus the total number of partitions is typically 10,000 - 100,000 in a PB-sized datacenter. It's obviously feasible to keep states at this magnitude. In light of this, our design is largely based on Ambry and all the operations are partition based, including creating replicas and popularity mechanism. Read requests for blobs will be counted towards the read count for the partitions that store these blobs. We consider a certain partition "hot" if and only if the read count for that partition exceeds a certain threshold.

3.2 Ambry-based design

For our project, we only focus on inter-DC problem. We assume that intra-DC problems, including hardware layouts, space compaction of partitions and load balancing are already solved.

Although load balancing is not one of the goals of this project, we still make effort to make sure not to present a design that fundamentally excludes load balancing. More discussion can be found in Appendix E.

3.3 Combination of Consistency and Popularity

The introduction of read count and the idea of a dynamic geo-distributed state machine bring about the design concerns for synchronize global data structures. A straightforward approach is to use a third-party open-source framework to maintain shared states. The Ambry paper mentioned a Zookeeper-based [13] cluster configuration services, though they haven't implemented that in production yet. However, it seems like an overkill due to the deferrable nature of populating services. Moreover, the huge number of states of numerous blobs in a cloud system will easily turn the central configuration services into a synchronization barrier. We propose ReadClock, a distributed consensus protocol that uses global read count as the clock value for a vector clock. The reuse of read counts is beneficial for that 1) It reduces one column (than if there are extra clock values to maintain) from

the shared global data structure, thus reduced (number of rows) states to be maintained, 2) It's light-weight 3) It gives global or partially global view of data popularity for each of the DCs.

3.4 ReadClock

The major theoretical part of this project is we introduce a new protocol named *ReadClock*, that is a variation of vector clock [15] but has relaxed requirements and more sophisticated functionalities.

3.4.1 Vector Clock

Modern distributed systems typically used Paxos-based [16] distributed consensus protocol [8–11]. For relaxed consistency requirements, e.g. eventual-consistency, the options are commonly SLA-based [5, 13, 22] solutions. However, they are too heavy-weighted and are not strictly-speaking distributed – usually a parliament is elected and the distributed protocol is only enforced among the leaders. As discussed in section 2.3, we cannot afford a synchronization barrier in a geo-distributed setting, therefore we go back to the origin of distributed consensus, vector clock[time space and the ordering of events] to look for answers. It is considered to be the foundation for Paxos, and furthermore all paxos-based protocols.

3.4.2 Choice of Clock Value

In Leslie Lamport's original design, the clock value for each processes can be any real number that is 1) strictly increasing on each process 2) the event of receiving a message will set the receiver process's clock value to be strictly greater than any of its previous clock value and the sender's clock value upon sending this message. In most of the implementations the value is just a sequence number that is not meaningful in real world. ReadClock introduces local and global read count for each partition to maintain the ReadStamp, instead of actual timestamp or sequence number.

3.4.3 Denotations

We define the following denotations for a fixed partition p.

Event e - an event happening on partition p. It's either the handling of a read request (for any of the blobs in p) or a message sending/receiving

GlobalReadStamp $G_i(e)$ - the total (global) number of read requests on partition p, from DC_i's point of view, at the moment of event e

LocalReadStamp $L_i(e)$ - the number of local read requests on partition p at DC_i, at the moment of event e. This read request is eventually handled by DC_i. (At first it might be received by another DC_j, but only counted towards DC_i's read count)

Request Message R_{ij} - The request message value sended from DC_i to DC_j regarding popularity on partition p

Acknowledgement Message A_{ji} - The acknowledge message value sending from DC j to DC i regarding popularity on partition p

Event - Sent an ACK Message Last Time M_{ij} - the event that the **last time** DC_i sends ACK message to DC_j

3.4.4 The Protocol

The ReadClock ticks in the following situations:

1. For each DC i, every time there is a local read request event r (a read request on partition p handled locally
   $L_i(r) \leftarrow L_i(r) + 1$
   $G_i(r) \leftarrow G_i(r) + 1$

2. If DC i sends a request message to DC j
   $R_{ij} \leftarrow G_i(sendR)$
   $G_j(receiveR) \leftarrow R_{ij} + L_j(receiveR) - L_j(M_{ji})$

3. After receiving message from DC i, DC j will reply with an acknowledgement message A
   $A_{ji} \leftarrow Lj(sendA)$
   $G_i(receiveA) \leftarrow A_{ji} + G_i(receiveA)$

Proposition 1 function L_i and G_i both are strictly increasing on DC_i

Proposition 2: $G_i(sendR) < G_j(receiveR)$

Proposition 3: $G_j(sendA) < G_i(receiveA)$ is not necessary true (this is relaxed from condition 2 of vector clock)

Proposition 4: G_i is the best knowledge of global read count for DC_i

Proof of propositions can be found in Appendix A

To sum up the major differences between ReadClock and Vector Clock:

- ReadClock has **"meaningful"** clock values – they are global read counts (to the best knowledge) for a certain partition on a certain DC, this has further usage in distributed locking and popularity measurement, which will be discussed in 4.2.5 and 4.2.6

- The resetting of clock values are different in receiving a request message than receiving an ack message

- **Relaxed** the conditions for message passing in vector clock – setting bigger clock values **only have to be at request message receiving**, not have to be ack message receiving

3.4.5 Distributed Locking

The distributed locking algorithm of read clock is exactly the same as [vector clock] (for a fixed partition p)

1. If DC i wants to acquire a lock, it needs to send request to all other DCs. It will put this request on its local request queue, and the receivers will do so as well.
2. If DC i wants to release a lock, it needs to send request to all other DCs. It will remove that from its local request queue, receivers will do so as well.
3. When deciding if DC i can enter a critical section (granted the lock), all it needs to do is to examine its local request queue (without talking to a central server) on whether:
    - It receives ack messages from all other DCs
    - Its request's GlobalReadStamp is the smallest of its queue

Even with proposition 3, the guarantees from vector clock still hold:

1. Only one DC will be granted the lock at a certain moment
2. If the lock is released, it will be eventually granted to another one (if there are more requests)

The proof of the guarantees can be found in Appendix B.

3.4.6 Popularity and Consistency

There could be a lot of false positives for hot data when a skew of popularity is at DC level. A common case is in the cross-country storage systems, where a video could be popular in the U.S. while very rarely accessed in China. It would still be a waste of effort to populate this video from a DC in the U.S. to a DC in China. By the advantage of meaningful clock values, the data will get populated only if it passes a certain threshold locally as well as it is not the vast majority of global counts (others also contribute to the global count, meaning data is popular in others' too). Besides this, distributed locking makes sure there are no two DCs trying to do the same thing – which further avoids waste of networking resources. The combination of popularity measurement (meaningful clock values) and consistency (distributed locking) makes sense because to gather global read counts, one DC needs to talk to others, and along the communication it might as well update clock values, hence the reusing of read count. Another good outcome is, according to proposition 4, Each DC has their best, though still partial view of global popularity, which further makes sure they are making the right decision (whether or not to populate).

Partition ID	Stored in which DCs	Status
Partition A	DC1, DC2, DC3	R
Partition B	DC2	RW
...	...	...

Table 1: Replica Map

Partition ID	Local Read Count	Global Read Count
Partition A	7	20
Partition B	3	5

Table 2: ReadMap

4 Implementation

We implemented a distributed storage system simulator from scratch in about 3,500 lines of Go code (with a few python and bash scripts). Each running instance of the server simulates a data center and there are multiple types of read/write clients that are interactive with the server. The interaction is done by manipulating in-memory data structures and transferring data across the network. More details on simulation will be discussed in section 6.

4.1 Architecture

The storage server consists of a front-end, a storage table and a logging layer. The front-end stores information regarding where a certain partition can be found, the global state machine guarded by *ReadClock*, and routing. The storage tables stores each partition and the blobs they contain, but it's an in-memory data structure – the logging layer will periodically log all in-memory data structures to disk in JSON format in case of fatal situations. Fortunately, there have been no server crashes; probably because of our limited experiment duration. We also implemented a centralized server for comparison. For brevity the implementation details are omitted.

4.1.1 Data Structures

Each running server will maintain three data structures, *ReplicaMap*, *StorageTable* and *ReadMap*. *ReplicaMap* stores the mapping of a partition to a list of DCs that store it, which is the global state machine shared across all DCs. *StorageTable* is a mock up for actual disk storage, which stores information for partition and each blob it contains. However it doesn't store actual files, only metadata that contains file size and create timestamp. *ReadMap* maintains local and global read count for each partition, as discussed in 3.2. An example of the three are in Tables 1, 2, and 3.

Partition ID	Blob List	Metadata
Partition A	{Blob:1, size:20MB}	Size:50Mb, ...
Partition B	{Blob:3, size:25MB}	Size:50mb, ...

Table 3: Storage Table

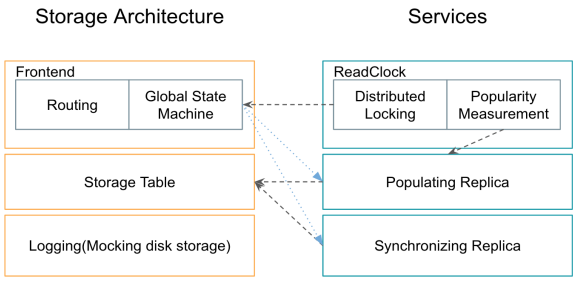

Figure 1: Storage Architecture and Services

4.1.2 Background Services

If one partition becomes hot (the read count exceeds a certain limit), the populating service will populate this particular partition to all other data centers. After populated, each partition/replica will share the same partition ID. Any additional write request will write to only one of the replica, hence others will be out of date. The sync-replica service will make sure all replicas (partition with same partition ID) will be in-sync by launching asynchronous compare-and-merge tasks. This is the same as Ambry's design. The interaction between data structures and background services is shown in (Figure 1), and will be further discussed in section 4.1.3.

4.1.3 System Interactions

Client-Server Interaction

A write client must specify a (content, size) pair to write to the cluster. Since the simulation server does not store actual files, the size is important for following operations. One drawback we have is that we do not have the strategy in Ambry that breaks big blobs down to smaller chunks to fit in one partition for simplicity, therefore the size must be set smaller than a partition maximum size. After a successful write, the server will return a (partitionID, blobID) pair to the writer, which serves as a unique identifier for reads. We use RFC-36 to generate UUIDs. There is an extremely small probability of collision but we assume that won't happen in our experiments (Ambry does have solution for collision, which we did not implement for simplicity).

Each write request will go to the nearest DC or a certain DC (according to different types of write client), scan the StorageTable, and find the first partition that has

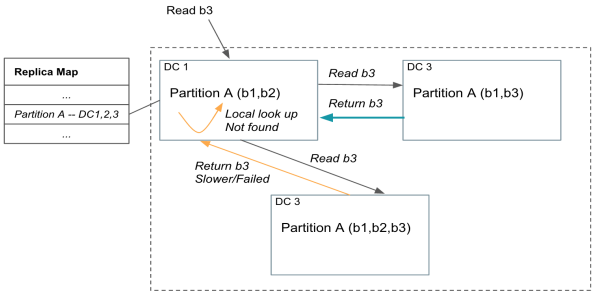

Figure 2: Making additional read requests, finishes as soon as one of them is done

enough space to store the blob. The UUID of that partition and the newly-generated blobID will be returned. A read client gives the cluster a (partitionID, blobID) to find a file. (We assume that a file name is already translated to this pair). As discussed above, a blobID is already unique globally, but to speed up the look-up, the read client still needs to feed a partitionID. After a successful read, a (content, size) pair will be returned.

The read client will go the the nearest DC or a certain DC (according to different types of read client), check *ReplicaMap* to find where the partition is stored. It will then send read request to all the DC(s) that stores the information and will return as soon as one of them is done, as shown in (Figure 2).The additional reads are needed because 1) A populated partition might not be up-to-date due to batched async writes, so the first DC the read request goes to might have the partition but do not have the blob 2) there might be stragglers/failures for read from other DCs. The idea is inspired by Cassandra [14], where the factor of how many other replicas to contact is specified by client to make sure of read-after-write consistency, and EC-Cache [21] where additional reads are critical for read latency. The read client will also update read count for the partition that contains the target blob, in the DC where the request is eventually handled.

Server-Server Interaction

StorageTable and *ReadMap* is local to each server. *StorageTable* is updated only when there are write requests served locally. *ReadMap* is updated only when there are read requests served locally or messaging acorss serves. The logic for maintaining *ReadMap* is specified by *ReadClock* in section 4.2.

Populating service will write partition as a whole to different DCs. Sync-partition service will compare partitions and merge them (in set semantics), where the writes happen at blob-level. The log-like, append-only structure makes sure the compare and merge is fairly straightforward. In Ambry [20], an additional journal is kept for each partition so that only the journal is compared during communicating, and later only the difference part (of

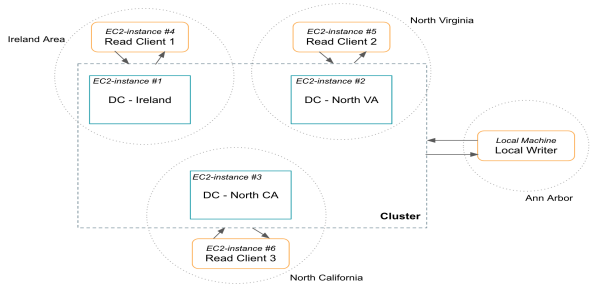

Figure 3: Test environment setup

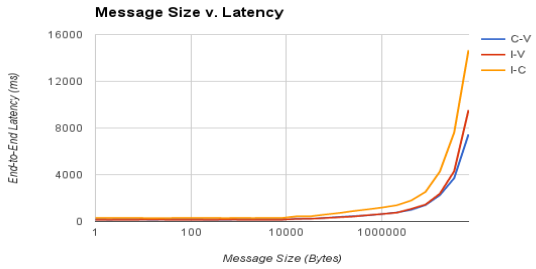

Figure 4: Experiment to correlate file size and transfer latency (on actual files). I stands for Ireland, V stands for north Virginia and C stands for north California.

blobs) will be transferred to sync partitions. In our implementation, each partition is just a fake file, therefore we transfer the whole partition structure, but only simulate transfer latency for the difference part.

4.2 Optimization for Concurrent Reads

Optimizing for highly concurrent accesses has been one of our design goals since the very beginning. The key performance factor for a storage system is read throughput, therefore we made effort to design data structures that best eliminates synchronization barrier for concurrent reads. More details can be found in Appendix C.

Unfortunately, during our experiments, we discovered that the bottleneck for concurrent access is actually the limit for number of open TCP connections / open files. Due to our limit budget we could not afford AWS instances with better networking performance. However we still propose above discussion in Appendix C to maintain our system design philosophy.

5 Evaluation

We evaluated both our centralized and decentralized designs using Amazon's EC2 clusters [3]. A zipf distributions was used to represent the file sizes and the popularity. With regards to the file size, we scaled the distribution to fit between 1 and 10 GB and files are read based on their popularity. Files with larger popularity value are more liked to be accessed. One important aspect deserves mention is that we don't store actual files in our system, but file size instead. More discussion about the system overhead will be covered below.

The experimental infrastructure was shown in (Figure 3). We had three servers located in three different regions. One in Ireland, one in N. Virginia, and one in N. California. A local client wrote the number of files randomly between the three servers. And, the final experiment used three different clients in each region respectively. The highlights of the evaluation results are:

- Amazon EC2 End-to-End Latency
- Copy-everything v. Copy-popular
- Centralized v. Decentralized

5.1 Amazon EC2 End-to-End Latency

(Figure 4) shows the end-to-end latencies between our three different regions. The x axis is log scale of the number of bytes we transferred, and the y-axis is the latency in milliseconds. The first name is the client and the second name is the server. For example (C-V) represents California as the client and N. Virginia as the server. The greatest latency was between Ireland and California with the smallest being between W.Virginia and California. We used these numbers to represent average time per data, so throughout the rest of the experiments we were able to simulate the transfer time as a latency function, based on the file size (in milliseconds).

5.2 Copy Everything v. Copy Popular

In this experiment, we are comparing two different policies: copy-everything and copy-popular. The former stores a single copy of data among the DCs. The latter remains at 3 times this number because they replicate all data in all the DCs. Therefore the start point follows the proportion with 3:1. As time progresses, more read requests poured in, but the number of storage space increases slowly for copy-popular because our threshold is set such that only 5% of files will be considered hot. The result is shown in (Figure 5)

We also compare end-to-end latency for both of the two policies. First we fixed a read client at a certain location, and read a fixed data set. We didn't turn on the traffic control [2] to mock up performance impact on copy-everything, and the result in (Figure 6) shows that the performance of two policies are similar. Note that our policy achieved the roughly the same with a lot lesser

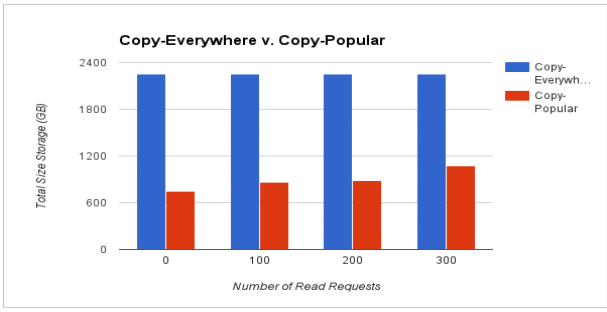

Figure 5: Comparison between copy-everything and copy-popular with regard to space efficiency

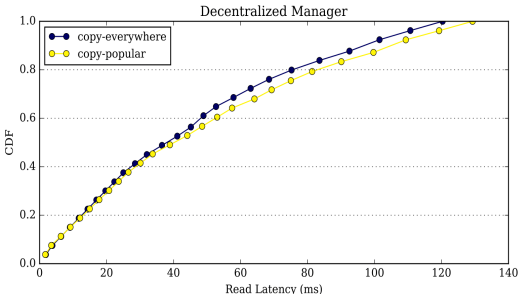

Figure 6: Comparison between copy-everything and copy-popular, without traffic control, fixed read client location

space consumption, when combined the result in (Figure 5)

We then chose different read clients in different locations (near Ireland, north California or north Virginia), to simulate the fact that a read user could appear anywhere in the world. Traffic control was turned on this time, but due to lack of theoretical support, we didn't know the measurement and setting for performance impact of copy-everything. We chose to simulate increasing of average latency, and added 40ms, 90ms, 140ms to copy-everything, the results are shown in (Figure 7, 8, 9). Due to the inaccuracy of traffic control, the actual latency impact is usually within $\pm(10-20)$ms around target latency.

Note that if TC is off or set to a small number, the above result is better for copy-everything. This is different from (Figure 6) because we are using read clients in different locations, therefore inter-DC traffic becomes huge and without the proper "punishment" of TC, copy-everything will make sure every piece of data is in every DC very soon, hence better performance.

5.3 Centralized v.s. Decentralized

The final experiment compared out centralized manager to our decentralized manager. In this experiment, we

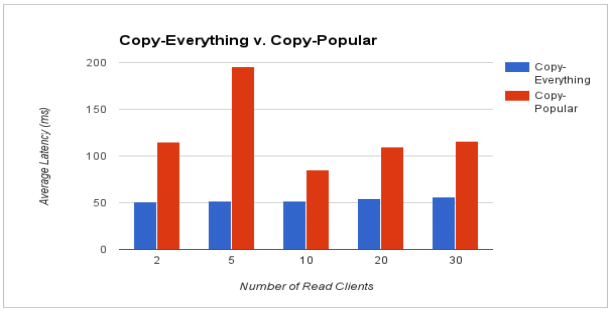

Figure 7: Comparison between copy-everything and copy-popular, with 10-60ms simulated latency increase for copy-everything

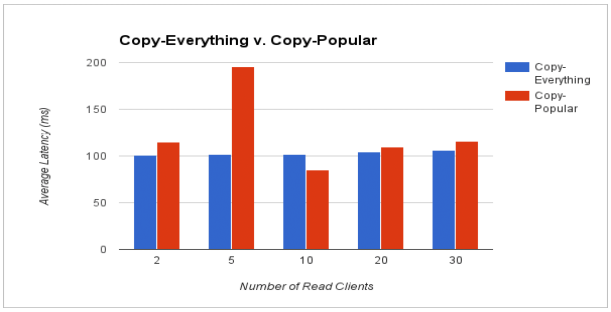

Figure 8: Comparison between copy-everything and copy-popular, with 60-110ms simulated latency increase for copy-everything

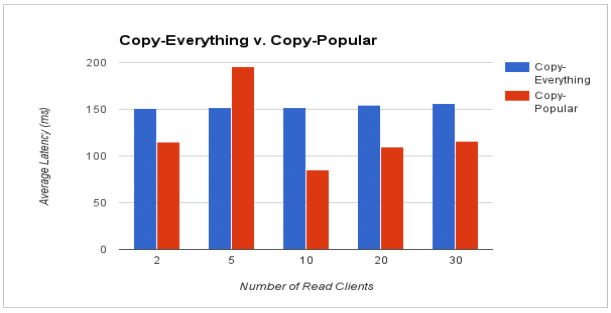

Figure 9: Comparison between copy-everything and copy-popular, with 110-160ms simulated latency increase for copy-everything

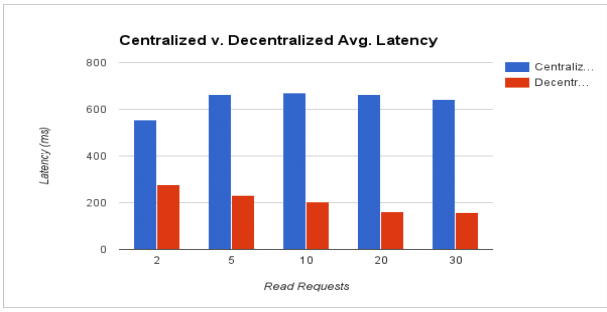

Figure 10: Comparison between centralized and decentralized manager, with regard to average read latency

looked at read access times particularly in Ireland for both cases. In a geographically distributed environment, it seems that the decentralized manager will exceed the centralized manager with regards to read access times as shown in (Figure 10). Based on our experiment, it appears that over the course of time, we must have been accessing popular data frequently. Therefore, the decentralized system could send the query information back immediately instead of having to access the central manager for information. We haven't been able to analyze and experiment in which setting the centralized manager will out-perform decentralized manager, because we haven't decoupled popularity management and routing server in our centralized manager implementation.

6 Related Work

Geo-Distributed Systems Several companies have published papers on their geo-distributed systems. Facebook originally developed haystack and currently F4. But F4 only deals with warm blobs based on a particular time threshold. LinkedIn developed Ambry [20], which is a decentralized system, but the implementation enforces the "copy-everything-everywhere" policy. Microsoft describes their Azure Cloud system. It maintains multiple stamps, an abstraction for datacenters. They migrate data based on stamp utilization and only copy for fault tolerance.

Selective Replica and Popularity Measurement Selective replica has primarily been discussed with regards to intra-DC (to inter-DC only in recent years). For CDRM [23], DARE [1], they provided a intra-DC solution for replica based on access patterns. For the algorithm presented by [19], it applied centralized manager for inter-DC design and defined popularity into several levels, but did not consider consistency.

Multi-DC Consistency MDCC [17] and Menfucius [17] discuss consistency protocols across datacenters, but they are both too strong (Read-after-Write). In our case, we can relax the protocol quite a bit because the global state machine doesn't have to be always up-to-date due to the fact that additional reads is implemented.

Auto Reconfiguration Several of the Geo-Distributed System statically set configurations, like Ambry. Others, for example, Tuba [5] and Pileus [22] have auto reconfig services but aim for primary/secondary replica settings.

7 Discussion

While we were able to test our approach very well, there were several limitations on the experimental setup. First, we were using a free tier of Amazon EC2 instances, we were limited on data transfer and number of instances running at a single time. Therefore, we were only able to run a 3 server experiment. If our budget was not limited, we could have expanded to even more DCs. Hopefully, this would have shown an even greater improvement of our decentralized approach. However, there is always a chance that with even more DCs, there would have been a larger overhead for coordination.

As we mentioned earlier, we used a simulated value for latency transfers. These numbers were calculated based on actual runs of our end-to-end latency experiment, and instead of transferring the message sizes, we transferred a small amount of data, but waited the calculated amount of time. Therefore, it might not have represented the actual system perfectly. For example, DCs have cyclic usage patterns, and there is a chance that some times of days would have much less bandwidth available.

There is also a limit for concurrently open TCP connections and open files for the instances (as mentioned in section 4.3). This is the bottleneck for read throughput. We weren't able to test our server in terms of 100+ IOPS and consequently the impact for end-to-end latency in more extreme settings is not considered. The solution would be utilizing higher-tier AWS instances and more advanced RPC framework. We are confident that, if getting more budget, we can test real-world latency for transferring real files. It's feasible to implement this on top of our code base since Go has open-source mock-up file system library.

On the other hand, we assume that intra-DC problems are already solved – including load balancing, fault-tolerance, caching, chunking etc.. We performed our experiments by excluding these variables, despite them being non-negligible in a real system. And, they would influence read latency dramatically. It would be thrilling to test our system in a real-world geo-distributed cluster.

8 Conclusion

Geo-distributed networks are becoming prevalent in this day and age. We looked at 2 major problems. Firstly, we compared the centralized and decentralized approach for distributed systems. Centralized one works well and is stable with small amount of requests, but de-centralized one is even better, with only 1/3 read latency of centralized one. Secondly, we looked into a copy policy based upon a particular threshold. We found that based on bandwidth and latency limitations, the decentralized approach was much more favorable. We were able to save up to 66.7% of space with popularity copy policy. As more partitions are shared, the space efficiency becomes lower but file availablilty increases.

References

[1] Cristina L Abad, Yi Lu, and Roy H Campbell. Dare: Adaptive data replication for efficient cluster scheduling. In *2011 IEEE International Conference on Cluster Computing*, pages 159–168. Ieee, 2011.

[2] Werner Almesberger. Linux network traffic control–implementation overview. In *5th Annual Linux Expo*, number LCA-CONF-1999-012, pages 153–164, 1999.

[3] Amazon. Amazon ec2. http://aws.amazon.com/ec2.

[4] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*, pages 287–300. ACM, 2011.

[5] Masoud Saeida Ardekani and Douglas B Terry. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 367–381, 2014.

[6] InfiniBand Trade Association et al. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[7] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, volume 10, pages 1–8, 2010.

[8] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[10] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[11] J Ellis. Lightweight transactions in cassandra 2.0. *Datastax Developer Blog [Online]*, 2013.

[12] Robert E Felderman, Alan E Kulawik, Charles L Seitz, J Seizovic, Wen-King Su, et al. Myrinet: A gigabit-per-second local area network. *IEEE micro*, (February):29–36, 1995.

[13] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[14] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[15] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[16] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[17] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *OSDI*, volume 8, pages 369–384, 2008.

[18] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru

Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebook's warm blob storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, 2014.

[19] Julia Myint and Axel Hunger. Comparative analysis of adaptive file replication algorithms for cloud data storage. In *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*, pages 115–123. IEEE, 2014.

[20] Shadi A Noghabi, Sriram Subramanian, Priyesh Narayanan, Sivabalan Narayanan, Gopalakrishna Holla, Mammad Zadeh, Tianwei Li, Indranil Gupta, and Roy H Campbell. Ambry: Linkedins scalable geo-distributed object store.

[21] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[22] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.

[23] Qingsong Wei, Bharadwaj Veeravalli, Bozhao Gong, Lingfang Zeng, and Dan Feng. Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster. In *2010 IEEE international conference on cluster computing*, pages 188–196. IEEE, 2010.

Appendix A. Proof of Propositions in Read-Clock

Proposition 1 function L_i and G_i both are strictly increasing on DC_i

Proof Every read request will increase L_i and G_i by 1, therefore L_i and G_i are strictly increasing. QED.

Proposition 2: $G_i(sendR) < G_j(receiveR)$

Proof According to the protocol, $G_j(receiveR) = G_i(sendR) + L_j(receiveR) - L_j(M_{ji})$ Use proposition 1, L_j is strictly increasing, and M_{ji} by definition happens before receiving R, therefore $L_j(receiveR) - L_j(M_{ji}) > 0$, QED

Proposition 3: $G_j(sendA) < G_i(receiveA)$ is not necessary true (this is relaxed from condition 2 of vector clock)

Proof Before sending A, DC_j could do a really fast collecting and update G_j , But this won't affect local read count L_j . Because of asynchronous message passing, and $\sum_{k\neq i,k\neq j} L_k$ with regard to DC_i's request could be bigger than the sum with regard to DC_j's request, therefore although $(L_i + L_j)$ will be non-decreasing, the total sum $\sum L_k$ might still be bigger, therefore $G_j(sendA) = \sum L_k + L_j(sendA) - L_j(receiveA')$ could be bigger than $G_i(receiveA)$ (where A' denotes the last ack message DC_j receives from others)

Proposition 4: G_i is the best knowledge of global read count for DC_i

Proof

1. If DC_i is the requester, it must talk to all other DCs to produce collective read counts globally. There might be other read requests happening after the moment DC_i receives all ACK messages, but there is no way of knowing those without communicating with other DCs again. Therefore G_i is the **best** knowledge for DC_i as of that moment.
2. if DC_i is one of the repliers. Denote DC_j as the requester. According to the protocol, upon receiving R_{ji} , $G_i(receiveR) = G_j(sendR) + L_i(receiveR) - L_i(M_{ij})$ According to 1), $G_i(sendR)$ is the best knowledge of DC_j, incorporating read counts from all DCs (including DC_i). Therefore the only part DC_i doesn't know is the number of read requests happening between the moment it receives DC_j's request and the last time it sends its local read count to the requester, by adding those up, it will be DC_i's best knowledge as of that moment. Still there might be new read requests happening at DCs other than DC_i and DC_j, but since DC_i is the replier, it doesn't talk to them, thus no way of knowing their local read counts.

Appendix B. Proof of Guarantees in Distributed Locking

Guarantees:

1. Only one DC will be granted the lock at a certain moment
2. If the lock is released, it will be eventually granted to another one (if there are more requests)

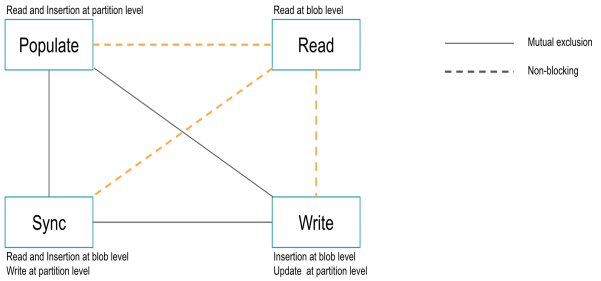

Figure 11: Synchronization barrier for system interactions

Proof of Guarantee 1 by contradiction If there are more than two DCs holding the lock, denote two of them as DC_i and DC_j.
Since DC_i "thinks" it's granted the lock, it must have sent its request to all others, including DC_j, and receives all others' acks. If the readstamp(timestamp) is G_i, then G_i must be in both DC_i and DC_j's request queue.
Similarly, the request readstamp(timestamp) for DC_j, G_j must be in both DC_i and DC_j's request queue.
Without loss of generality, assume $G_i < G_j$, **since G_i and G_j are both in DC_j's request queue before DC_j makes the decision** $(*)$, DC_j couldn't possibly think that G_j is the smallest. QED

Note that $(*)$ is guaranteed by setting clock value bigger than any of original value or the message value **only when receiving the request messages**, since the receiving of request message is **the only event that makes a clock in one DC "jump"**, while the ACK messages are merely for making sure that others did receive its request.

Guarantee 2 can be proved similarly.

Appendix C. Optimization for Concurrent Reads

The following access patterns are common on our server, as shown in (Figure 11)

Client read and client write: A client read will scan a partition's blob list, find the target and return; in the meantime a client write will only append a new blob at the the end of the blob list. There is no update to any of the blobs, and Go ensures that a reader and an "inserter" that only appends at end doesn't conflict in a list. Further more, a client write only updates partition metadata of a partition (update size), which a client read doesn't read from, therefore there is no client-read and client-write conflict for a certain partition.

Client read and populating partition: Populating a partition will copy that partition first and then transfer the copy, there is no conflict for copying and reading the same partition.

Client read and synchronizing partition: In our implementation ,as discussed in section 4.1.3, there is no journal kept for each partition, therefore each partition is copied as a whole to be transferred and compared. The copying and reads certainly do not conflict. Upon receiving the delta (difference of two partitions sharing the same ID), each of them is appended to the end of the blob list like a client write, which we already proved that there is no reader-"appender" conflict as well. It's trivial to show that populating service, client write and synchronization service has pairwise conflict for a certain partition, however client read doesn't conflict with any of them, therefore in design of our data structures, reads doesn't have to use lock to guard critical section.

Appendix D. Why Golang

We chose Go language to implement our server for the reason that it was invented for multi-threading infrastructure development. The three major data structures, *ReplicaMap*, *StorageTable* and *ReadMap* are concurrently manipulated by multiple threads(goroutines), therefore the light-weight creation/switch/destruction for goroutines make them easier to manage. Moreover, there are many built-in libraries in Go which further reduces the amount of work for set-up, preparation and tuning of our development.

Appendix E. Load Balancing

An important practice is that inter-DC replicating is still needed with the very existence of CDN, even without fault-tolerance concerns, due to limit of cache size for the CDN. Actually, our policy that only populates hot data is further justified by CDN – if there are very few read requests, CDN caching do the work without suffering user experience; if much more requests emerged, then and only then a batched cross-DC replicating will be scheduled. We don't have the chunk (breaking down larger blobs) abstraction neither just for simplicity.

In Ambry's original design, each blob will be appended to a randomly selected partition for load balancing; we 1) chose a smaller partition size (10X smaller) and 2) append to partitions sequentially. The primary reason for these is to avoid false positives – if one partition is a mix of very few hot blobs and large number of

cold blobs, all those cold ones will be replicated (inter-DC) as well. Since hot data will be highly likely new data, quickly filling up a partition and make decision whether or not to replicate to other DCs is crucial in our design. Also smaller partition size will reduce false positives. The concern of load imbalance is mitigated by smaller partition size as well, for the fact that mixing of cold and hot ones are at a smaller granularity will make a more likely balanced load. We maintain above reasoning, though most of them are just control variables/static settings in our implementation, in order to present a design that would make sense in real-world systems.