Managing Object Versioning in Geo-Distributed Object Storage Systems

João Neto[*]
KTH
joaon@kth.se

Vianney Rancurel
Scality
vianney.rancurel@scality.com

Vinh Tao
Scality and UPMC-LIP6
vinh.tao@lip6.fr

ABSTRACT

Object versioning is the keystone for implementing eventual consistency in modern geo-distributed object storage systems such as Amazon S3. Despite this, the study of implementing object versioning has not been given a lot of attention in either academic or industrial communities. The selection of an implementation method is not considered as an important factor impacting the overall system performance under different workloads.

In this paper, we present our study of two methods of implementing object versioning in geo-distributed object storage systems and on how these impact the performance of these systems under different workloads. We propose and analyze the advantages and disadvantages of (1) Write-Repair approach and (2) Read-Repair approach. From our experiments, we found that the choice of approach significantly impacts the performance of storage systems, which in turn impact the performance of applications built on top of these systems.

1. INTRODUCTION

NoSQL key-value stores such as Amazon S3 [1] and the likes of Basho Riak [3] and Apache Cassandra [2] have contributed to the movement of computing to its third-platform [15, 6], in which cloud computing and cloud storage play an important role for the availability of the services in this ecosystem. At the core of cloud storage systems, eventual consistency is the main mechanism that enables these systems to provide highly-available services to the end-users.

[*]Now at Universitat Politècnica de Catalunya. The work was done when the author was in an internship at Scality.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ScienceCloud'16, June 01 2016, Kyoto, Japan

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4353-4/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2913712.2913714

Although the use of eventual consistency has become mainstream, the details on how to implement its core mechanism—object versioning—have not yet been discussed thoroughly in the context of geo-distributed object storage systems. This leaves unanswered questions: (1) what are the different implementation approaches of object versioning and (2) how does each of these approaches impact the performance and the usage of the resulting storage systems in different scenarios and different workloads.

We have generalized and implemented two different mechanisms, Write-Repair and Read-Repair, for controlling object versioning in our geo-distributed object storage system. The former approach resolves all of the existing conflicts at the time an update is committed and provides readers the final value of the update, while the latter does not require conflict resolution at the commit time, enabling updates to commit quickly. Our experimental results have shown that both Write-Repair and Read-Repair provide their own advantages, which are fast read for the former and fast commit for the latter, while exhibiting their disadvantages. Beyond this result, we also provide the requirements for an efficient implementation of each of the aforementioned mechanisms that could impact their overall performance in different real-world use-cases.

Our contributions can be summarized as following:

- We explore the two different approaches of implementing object versioning for eventual consistency in geo-distributed storage systems and their impact on the performance of these systems under different workloads.
- We generalize these approaches into two different representative implementation methods, namely, Write-Repair and Read-Repair, each with its own advantages and disadvantages. We also analyze and provide some useful requirements and criteria in order to efficiently implement each approach.

The rest of the paper is organized as follows: In Section 2.1, we introduce some background information on the problems in geo-distributed storage systems and their resolution. In Section 2.2, we describe the design of our system with its analysis. We present in detail the implementation of our system and the two approaches on eventual consistency in Section 3. Finally, we report our experiments and their results in Section 4.

2. BACKGROUND

2.1 Geo-Replication Issues

A typical geo-distributed storage system spans multiple datacenters (or sites), each of which located at a different geographical location. To improve availability, these systems usually replicate their data at all sites. For maintaining the consistency across sites, these systems usually use locking (in the form of a geo-distributed lock), or have a single master site that decides the order of all updates from all sites. Because of the high latency of inter-datacenter networks—which is much higher than that of the intra-datacenter networks—committing updates in these systems using traditional locking approaches is usually prohibitively expensive, especially for those systems at the global scale.

In the simple eventual consistency approach, each site of a geo-distributed storage system can commit updates locally to avoid inter-datacenter coordination before propagating these updates to the other sites. When all sites have received and committed all local updates from the other sites, they will converge to the same state.

The apparent issue with this approach is that updates from different sites can concurrently target the same object in the storage systems. Then deciding which update to keep and which to ignore (thus to rollback) is problematic. Strong eventual consistency [12] instead opts to keep all updates in different versions of the target object—this mode is known as object versioning—and then tries to resolve any existing conflict between these updates. In real-world geo-distributed object storage systems with object versioning, such as Amazon S3, any update to any object creates a new unique version of that object which is identified by a unique id in the response to the update. Therefore, systems with object versioning never encounter the issue of rolling back an update when it conflicts with some other concurrent updates.

Because having multiple concurrent versions of an object may confuse the applications when choosing which version of the object to use, geo-distributed object storage systems with object versioning also have to provide a deterministic mechanism for each datacenter to determine the latest version of an object out of a list of its concurrent versions. A solution for this issue is (1) to have the partial order between different versions of an object to narrow down the scope of the latest versions, and then (2) to use the Last-Writer-Wins (LWW) approach to deterministically decide the latest version out of the list of concurrent versions. The problem of ordering updates in distributed systems is usually solved using version vectors [7, 9]. This mechanism tracks the partial order of these updates. For concurrent updates targeting the same object, their conflict can be resolved using the LWW approach. The LWW approach decides which update should be considered the latest value of the updated object, using factors such as real-timestamp or a preference list. Though the decision of the LWW approach is arbitrary, this decision must be deterministic across sites.

Implementing the Last-Writer-Wins approach also exposes some options that affect the performance of different geo-distributed object storage systems with different workloads. This is the main target of our study as will be described in the next sections.

2.2 Object Storage System Design and Assumptions

In this section, we provide our system design along with our discussion of the design decisions that we made. This design provides the basis for our replication and conflict resolution models in the later sections.

Assumption 1: We have separate storage systems for metadata and data in our object storage system. The separation of metadata and data has become a common pattern in many other distributed systems, including Google File System [5], Boxwood [10] and Hadoop Distributed File System [13]. It provides a good separation-of-concerns practice in designing and implementing distributed storage systems that frees us from having to deal with replication of data.

We share the same point of view regarding this pattern. In our view, the real bottleneck of a system is the metadata path. By making the data content (data, for short) immutable using the Copy-on-Write technique we can remove the data, which usually is much larger in size as compared to metadata, from the critical path. When data is stored in an immutable way, it can be replicated easily in geo-distributed systems. Because of the Copy-on-Write technique, each data item in the system is as unique as the others; modifying an object creates a new version of that object. Therefore, there is no conflict in writing to the same object in the data part of the system and we can have a single storage system spanning multiple sites for storing data. On the other hand, metadata is mutable, which means it can be modified by users. In a distributed setup where data is replicated on different sites, the metadata of the same object could be concurrently modified by different users from different sites. Therefore, we keep a full replica of the metadata at each site to enable local operations.

Our system design is show in Figure 1. We have a dedicated data storage, called stretch data ring that stores immutable data. This dedicated storage system provides a simple key-value store API (with PUT, GET and DELETE), and is based on Chord [14]. The data store is composed of all data nodes from all sites and provides the same namespace to all users on all sites. Each data object is represented by a unique key; the

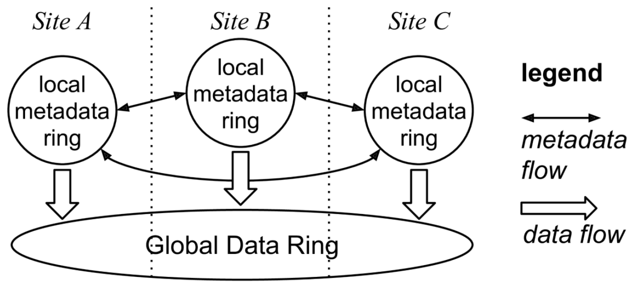

Figure 1: The design of our system with stretch rings.

process of generating a key is deterministic, without the need of coordination between sites. In a normal usage scenario, if a user on site A writes a data item which is identified by key k to the stretch data ring, then eventually users on all sites will be able to retrieve this data item by looking up k in the stretch data ring. There are different metadata rings, each of which stores the whole copy of the system's metadata. All metadata rings talk directly to each other to exchange their local updates while all of the data is redirected to the shared data ring.

Assumption 2: The metadata storage system provides the API of a key-value store with range query ability for the keys. This range query feature is an essential feature for implementing LWW in our object storage system, as it is described in the next sections. It enables us to efficiently look for the latest concurrent versions of an object without any knowledge about these versions.

3. IMPLEMENTING EVENTUAL CONSISTENCY

In this section, we describe the common behaviour of both approaches of our eventually consistent geo-distributed storage system, as well as the implementation requirements.

3.1 Object Versioning Behavior

A common solution to the problem of conflicting updates from different sites is to make most of the metadata immutable. This is achieved by making every update to an object a separate, independent version entry in the version history of an object. Amazon S3 is arguably the most well-known example implementing this approach. When versioning is enabled, any update to an object in Amazon S3 creates a new version of the the object; this version information is returned to the client in the response of the update request. Users may use this version information to select the version of the object to retrieve.

With this mechanism, updates to the same object from any of the other sites also create another version of the object and append this version to the object's version history. Users on the other sites can explore the

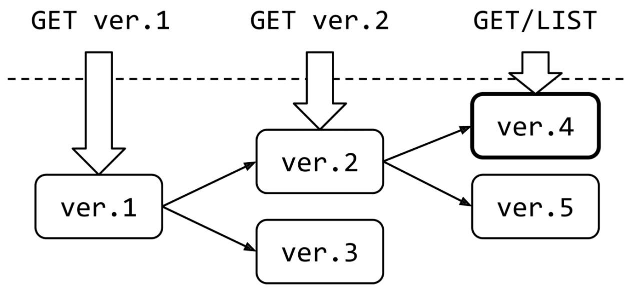

Figure 2: Example of the versioning API. Arrows describe the partial order ($A \rightarrow B$ means A happened-before B following Lamport's notation [9]) of the different versions.

newest versions of the object by querying the object without any prior knowledge about the latest version. At this point, we can see that the object itself is the single entry point to access the latest version and the history of the object. An example of this API is described in Figure 2, where an object has several versions, with those latest versions being `ver.3`, `ver.4`, and `ver.5`; in this example, `ver.4` is selected as the latest version, and all read requests such as `GET` and `LIST` will go directly to this version. Users are still able to retrieve the earlier version by specifying the version number in their `GET` requests.

The implementation on how to select this latest version and how to manage the version history of an object is the technical problem that we tackle and describe in the following section.

3.2 Implementation Approaches

From the above analysis, we arrive at two different approaches for implementing this entry point: Write-Repair and Read-Repair. These designs are based on the fact that the process of identifying which update should be the latest one (we name this process LWW) could be done when committing the update or when reading from the entry point.

3.2.1 Write-Repair

The Write-Repair approach performs the LWW process at the time it commits updates. In order to do so, the system first gets the version history of the target object of an update. Then, it determines the latest version of that object by comparing the version of the update with the latest versions in the version history. Finally, the system updates the object's version history and writes both this history and the latest version back to the metadata ring.

Thus, this approach needs two different locations to store the metadata of an object: one for version history and the other for metadata of the latest version. Also, the keys for the latest version and for the version history of an object should be known by any other sites without any knowledge about the version of an object.

Writing to an object in the Write-Repair approach requires two round-trip communications with the metadata ring—the first one is for retrieving the version history and the second one is for writing back the version history and the metadata for the version determined to be the latest. However, the advantage of this approach is that reading the latest version of an object is simply done by querying the known key for that object. With this implementation, there is no need for any change at the existing key-value store in connection with a read.

Another drawback of the Write-Repair approach is that the metadata for the latest version of an object (the one returned as the default on a read if no specific version is requested) is mutable. Because committing an update on an object needs to also update the default version of that object, there can be no concurrent updates in a single site on a single object, otherwise one write could overwrite the other. This property results in the requirement that all updates on a single object must be serialized (though real-world implementations can minimize the effect of locking by batching updates).

This approach is depicted in the left part of Figure 3. A client that wants to update an object `foo` has to do it in two round-trips to update the version history of `foo` by (1) retrieving version history of the key at `v:foo`, then updating this history in the LWW process, and finally (2) writing this version history back along with the latest version to the metadata ring. The figure also describes the process of reading the latest version of `foo`. This is done by simply retrieving `foo` which is the key for `foo`'s latest version.

3.2.2 Read-Repair

In order to enable single-roundtrip updates to the storage system, the Read-Repair approach requires the LWW process to be performed whenever an object is read. We achieve this by making the metadata immutable, meaning all updates to metadata also create new versions of metadata, just like they do for the data part. This could be done easily by generating a random key for each version of any object. However, all versions of the same object should be somewhat related to be retrieved efficiently in order to be able to determine the latest version of an object at read.

We use prefix search to find all versions of an object. When committing an update on an object, we create a key with a deterministic prefix, which is the identity of the object, and with a random suffix, which is the version identifier. When reading the version history of an object, we simply do a prefix query on the deterministic prefix. This approach ensures we have all of the required versions. From these versions, calculating the LWW can be done easily.

This approach enables us to have single-roundtrip writes. In fact, by making metadata immutable, writing to the whole system is done in a single roundtrip without being blocked by any other concurrent writes to the same object.

The disadvantage is that the underlying key-value store must have range query capabilities. This is not always an easy requirement to satisfy. Furthermore, performing a range query usually requires the scan of, or at least an iteration through, different pages in a distributed B tree (the common index implementation to support range queries). This type of search is not cheap, compared to the other exact search approaches. The implementation of the key-value store chosen for the Read-Repair approach should therefore be efficient at performing range queries.

Another issue with Read-Repair is that, in geo-distributed setups, it is very expensive in terms of read operations, when compared to the Write-Repair systems. Whenever a read request is made for the latest version of an object, the site receiving the request has to retrieve all of the latest versions of that object from its local storage for all sites that are known to the local site. This results in many more (local) round-trips for a simple read, when compared to that of Write-Repair. Listing all of the objects available in a Read-Repair system is the most expensive operation of all: in this case it has to scan for all versions of all objects available in the system to find the latest version of each object.

An example of the Read-Repair approach is shown on the right side of Figure 3. In this example, writing to an object `foo` is done by writing to a key `foo_i` with `_i` as the unique version identifier of that update. Reading the latest version of `foo` requires a scan for all keys with the prefix `foo`. The key of the latest version in the retrieved list is then used to retrieve `foo`'s metadata.

3.3 Implementation Details

The implementation of both Write-Repair and Read-Repair is quite straightforward.

Write-Repair The keyscheme for Write-Repair is implemented in the following way.

- The main key is the combination of the object's identifier *id* and a specific prefix *pre*; using a specific prefix for the main keys enables us to do the listing function by simply issuing a prefix search for *pre* as in normal key-value stores. The combination for the main key is $\{pre{:}id\}$. With the example of writing to `foo`, the main key would be `A:foo` where the capital letter `A` is the prefix in our implementation; listing all keys would be done by prefix searching for `A:`. The value associated with a main key of an object is always the latest version of that object. This enables the storage system to retrieve the latest version of an object by simply searching for its main key.
- The key for a version of an object is compiled easily, with the object's identifier as the prefix and the specific unique version identifier `vid` as the suffix. A specific version `v4` of an object `foo` could then be retrieved by searching for the key `V:foo:v4`. The value of a specific version of an object is a

structure which includes the version vector of that version, a timestamp associated with the version, the identifier of the site on which the version was created, and the metadata of the object itself.

Read-Repair The keyscheme for Read-Repair is implemented in a more complicated way. Because we cannot determine the latest version of an object using a specific key, we need to list all versions of the object to find the latest version.

- The keyscheme is represented by `pre:id:sid:ts`, where `pre` is a prefix, `id` is the identifier of the object, `sid` is the identifier of the site where the update was issued, and `ts` is the local real timestamp of the update. The inclusion of the real timestamp in the key enables us to specify the scope of range queries for `GET` and `LIST` operations.
- When retrieving the latest versions of an object while using Read-Repair, we can do it in different ways. The first approach is to simply issue a full range query to get all versions of an object and then searching through them to find the latest version. The second approach is to issue multiple very light range queries to get the versions, each of which represents the latest version of the object at a given site. The first approach is straightforward and can be used for the systems with a low probability of object rewrites, or when the number of sites is very large. The second approach is more efficient when an object is rewritten many times from many sites, or when the number of sites is very small. In the range queries on the second approach we restrict the upper-bound to the latest timestamp—which is the local time at the moment of issuing the query and it is the real upper-bound—and the number of results to 1; this ensures the query retrieves the latest version of the object for the given site. Since we have multiple sites, we must issue locally as many range queries as the number of sites, in order to get the latest versions of that object from all the known sites.
- The LWW version is decided in a similar manner as in Write-Repair. Keys are compared using their version vectors to see which keys are logically more up-to-date. Then, out of the latest versions, the version with the most recent timestamp is chosen. In the case that there are multiple latest versions with the same physical timestamp (meaning they must be from different sites), the LWW version is chosen based on a site preference list. Again, this does not necessarily result in the latest version in terms of real time but is a deterministic method for selecting the latest version.

4. EVALUATION

In this section we compare the performance of our geo-distributed storage system using both implementations with Write-Repair and Read-Repair.

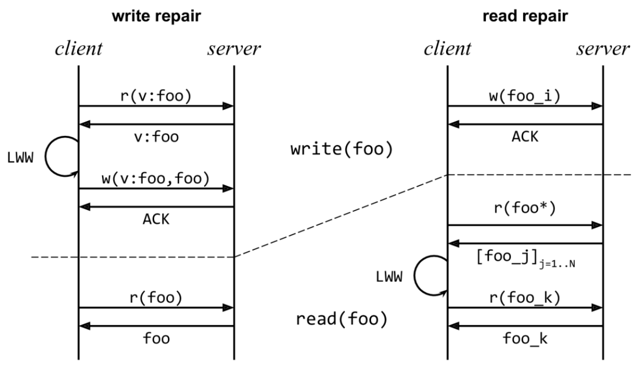

Figure 3: Write Repair and Read Repair mechanisms.

4.1 Experimental Setups

4.1.1 System Implementation

We implemented the Chord protocol as the underlying routing mechanism for looking up in our distributed system. We used the algorithm `SHA1` for the hash function.

In order to support range query for Read-Repair, we implemented a distributed B+Tree to use as the key-value store for the metadata rings. This B+Tree was also used for the stretch data ring. The Chord protocol and the B+Tree were implemented in C language.

We also implemented a connector, which is a component that forwards requests and performs all of the geo-replication logic, with both Write-Repair and Read-Repair approaches. A connector on a site received requests from clients at that site, forwarded data to the stretch data ring, indexed the data stripes[1] and stored this index in the metadata rings. The connector also handled the versioning information of all objects in the system by assigning version vector and timestamp to each update, and propagated this information along with the update to the other sites. Because data is replicated automatically inside the stretch data ring, a connector only needs to propagates the changed keys and their metadata. We implemented the connector in NodeJS.

4.1.2 Setups

Our experiments were conducted on a cluster of 6 data storage servers and 1 connector server. Each of the 6 storage servers hosted 6 logical data nodes of the stretch data ring and 6 logical metadata nodes of the metadata ring. Each of the storage servers were configured with an Intel Xeon E5-1620 CPU, 64GB RAM, 240GB SSD (x2), 2TB SATA (x2). The connector had the same configuration, except it did not have the SSDs. These servers were connected by 1Gbps links with an average measured latency of 0.065ms.

[1]Large data objects were split into smaller parts named stripe which were 1MB in size.

We decided not to have a full implementation of multiple sites because of a few factors. The replication between sites does not perform any specific task other than just applying remote updates as if they were local. Therefore, we believe that by just emulating the geo-replication and the remote updates by using local updates, we can achieve the same effect in the system. Moreover, our purpose is to test the trade-off of the different implementations of eventual consistency. Therefore, it is enough use any consistent workload on both of them to find the trade-off. In our implementation, we also used the first approach of Read-Repair to find the latest version, in which we list all versions of an object and search through the version list.

4.1.3 *Evaluation Strategy*

We performed different operations (PUT/GET) in the system with objects of varying sizes while varying number of updates to see how Write-Repair and Read-Repair would perform differently, in terms of latency. We only measured the latency for performing requests on metadata (rather than on both metadata and on data) because we believe that the latency for data only depends on the size of the data and the network bandwidth, and does not depend on different implementations of eventual consistency.

In the first experiment, we ran a sequential test of (1000) PUTs and GETs to see how each implementation performed a single operation. We changed the size of the objects (4KB, 1MB, and 16MB) to determine whether the performance difference between Write-Repair and Read-Repair should be significant as object size increases.

In the second experiment, we ran sequential batches of only PUTs in order to determine the behavior of these approaches under high concurrency. We changed the number of requests per batch from 1K, 10K, to 100K to apply different concurrency levels.

4.2 Experimental Results

4.2.1 *Objects of different sizes*

The result of the first experiment is shown in Figure 4. In the case of sequential requests of 4KB objects, the expected performance difference between the different approaches is shown clearly. The average latency for GETs in Read-Repair (20ms) is the double of that of Write-Repair (10ms) while the average latency for PUTs of Write-Repair (37ms) is 1.5x larger than that of Read-Repair (24ms).

As the size of the objects increased to 1MB, the latency for GETs in both implementations remain the same as for objects of 4KB, but the latency for PUTs increased to 38ms and 50ms for Read-Repair and Write-Repair, respectively. With an object size of 1MB, there is still only a single data strip so the metadata to be retrieved should be the same as with a 4KB object. Thus, the lack of increase in latency for GETs is expected. The increase in the latency for PUTs is likely due to the increase in the usage of the network.

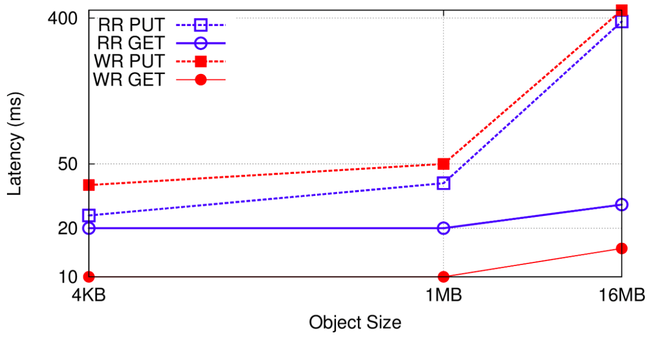

Figure 4: Evaluation of the systems with sequential requests (1000) of objects of different sizes.

As the size of the objects increased to 16MB, the latency for GETs increased by around 50% while the PUTs latency skyrocketed to 10 times as much as PUTs for objects of 1MB. The increase in the latency for GETs is due to the increase in the size of the metadata for each object (now metadata has 16 keys for 16 data stripes). For PUTs, we assume that the increased network usage in order to write a large amount of data increased the PUTs latency.

Throughout the experiments, the latency for PUTs with the Read-Repair approach was better than when using Write-Repair, while the situation was reversed for GETs. This result is consistent with the number of roundtrips that each approach has to take to perform the given tasks.

4.2.2 *Batches of different sizes*

In the second experiment with concurrent PUTs only, Read-Repair still performed better in terms of latency. The latency for the Write-Repair approach was from double to triple that of the Read-Repair approach at the beginning of our experiment, and the difference was maintained at a good level as we increased the number of concurrent batches in the system; this is shown in Figure 5 clearly.

4.2.3 *Implications*

There are some important implications from these results. First, the Read-Repair approach has better latency for `PUT`s while the Write-Repair approach is better at `GET`s and `LIST`s. Second, as the sizes of the objects increase, these different approaches will be more likely to converge at some point. And third, the efficient implementation of Read-Repair requires a good underlying key-value store with range query capabilities.

5. RELATED WORK

Amazon's Dynamo [4] paper was the first description of an object storage system that achieved eventual consistency by implementing object versioning; it has

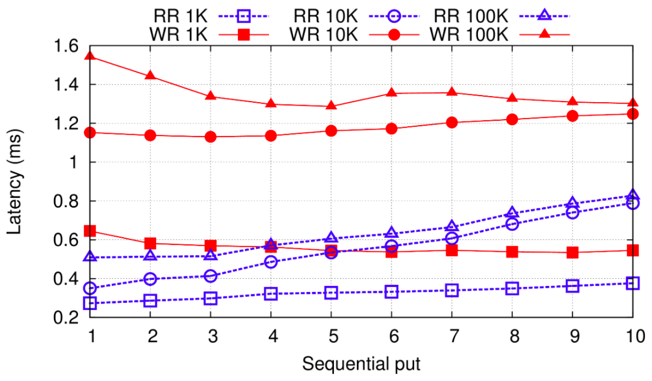

Figure 5: Latency of Write-Repair and Read-Repair in putting batches with different concurrency.

inspired a lot of the other systems to follow its model, including Cassandra [8], and it has made cloud storage and especially NoSQL a new trend for research and development. Even though Dynamo implements object versioning, Dynamo still requires that users specify the original version of an object in order to write a new version of the same object to the system. To keep the partial order, Dynamo relies on the manual intervention from users to semantically resolve conflicts between concurrent updates (which are detected by using version vectors). Dynamo returns the latest versions of an object on a read request and the user must resolve the conflict. It can thus be classified as a Read-Repair system (with manual intervention required for repair). Object storage systems from today have improved since the Dynamo paper, as users in most recent systems are not required to have prior knowledge about the previous versions of an object in order to write a new version; it's enough to use version vector to keep the partial order between versions. This progress is reflected in Amazon S3's API, which is a more common service of Amazon than its DB counterpart.

A representative example of using write-repair is in the case of distributed file systems, where the operations related to reading and listing such as `open` and `ls` are extremely common. In these systems, such as our previous work on the SFR [11] file system, a client committing an update is expected to: (1) read the default version of a file, which stores its latest version, then (2) update this default version to the version that the client is going to write and only then (3) write this default version back. This approach blocks concurrent writes from the same site. However, it is easy to implement, and more importantly, it works well in the environment of distributed file systems where the listing operation (`ls` in Unix-like file systems) is extremely common though expensive. For example, the system when receiving a list operation has to find the list of sub-directories and sub-files of a directory, then for each of them, the system has to get the attributes of the latest version of that sub-directory or sub-file. If the system relies on Read-Repair, to find each latest version there would have to be a large number of range query requests. If the system relies on Write-Repair, the system would just have to do a simple read of the latest version.

6. CONCLUSION

We presented our study on different approaches for implementing eventual consistency in geo-distributed storage systems. We have shown that eventual consistency can be implemented with different approaches and that the chosen approach will impact both the implementation of a system and the performance of the system. Through our experiments, we confirmed that the latency for PUTs is higher with a Write-Repair approach and the latency for GETs is higher with a Read-Repair approach. So, if a use case involves a lot of reads and listing of objects, a Write-Repair system should be considered. Meanwhile, the Read-Repair approach could improve write latency for small-write workloads which involve mostly writes (especially writes of small objects) but this improved write latency comes at the expense of the added complexity of required range queries on reads. Our work is an attempt towards optimizing eventual consistency geo-distributed storage systems.

Acknowledgement

We thank our anonymous reviewers for their constructive comments as well as to Sana Imtiaz and Lauren Spiegel for the valuable feedback on early versions of this article.

7. REFERENCES

[1] Amazon Simple Storage Service. http://aws.amazon.com/s3/. Accessed: 2014-12-31.

[2] Apache. Apache Cassandra. http://cassandra.apache.org/. Accessed: 2015-08-24.

[3] Basho. Conflict Resolution. http://docs.basho.com/riak/latest/dev/using/conflict-resolution/. Accessed: 2015-04-27.

[4] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[5] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[6] IDC. Third platform. http://www.idc.com/prodserv/3rd-platform/. Access: 2015-09-21.

[7] Jr, D. S. P., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B. J., Walton, E., Chow, J. M., Edwards, D., Kiser, S., and Kline, C. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, 3 (1983), 240–247.

[8] Lakshman, A., and Prashant, M. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* (2010), 1–6.

[9] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (1978), 558–565.

[10] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. Boxwood: Abstractions as the Foundation for Storage Infrastructure. OSDI'04, USENIX Association, pp. 8–8.

[11] Segura, M., Rancurel, V., Tao, V., and Shapiro, M. Scality's Experience with a Geo-distributed File System. In *Proceedings of the Posters & Demos Session* (New York, NY, USA, 2014), Middleware Posters and Demos '14, ACM, pp. 31–32.

[12] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. A comprehensive study of Convergent and Commutative Replicated Data Types. rr 7506, Inria, rocq, 2011.

[13] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (May 2010), pp. 1–10.

[14] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2001), SIGCOMM '01, ACM, pp. 149–160.

[15] Wikipedia. Third platform. https://en.wikipedia.org/wiki/Third_platform. Access: 2015-09-21.