Anticipating Requests to Improve Performance and Reduce Costs in Cloud Storage

Hylson Vescovi Netto
Universidade Federal de Santa Catarina, Instituto Federal Catarinense, Brazil

Lau Cheuk Lung
Universidade Federal de Santa Catarina
Florianópolis, Brazil

Tulio Alberton Ribeiro
Universidade Federal de Santa Catarina
Florianópolis, Brazil

Miguel Correia
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Aldelir Fernando Luiz
Instituto Federal Catarinense, Campus Blumenau, Brazil

ABSTRACT

Clouds are a suitable place to store data with scalability and financial flexibility. However, it is difficult to ensure the reliability of the data stored in a cloud. Byzantine fault tolerance can improve reliability, but at a high cost. This paper presents a technique that anticipates requests in order to reduce that cost. We show that this technique improves the performance in comparison with related works and maintains the desired data reliability.

Keywords

Cloud Storage, Quorums, Byzantine Fault Tolerance

1. INTRODUCTION

The Cloud Computing paradigm is making available many services in the Internet. Among those services, storage is one of the most used. Features like elasticity and pay-per-use have attracted an increasing number of consumers and the number of public clouds has followed the growth in demand. Passive storage services like Amazon S3, Azure Storage, and HP Object Storage are a cheap and flexible option.

Problem statement: when consumers want to store data in clouds, the following issues arise[1]: *a*) clouds can fail; *b*) using a single cloud can lead to vendor lock-in (difficult to change of cloud, e.g., when they fail or costs vary); *c*) bad choice of cloud data centers may lead to reduced performance or survivability; *d*) data shared using the cloud may lead to integrity issues.

Byzantine fault tolerance (BFT) [4] can contribute to solve these issues but existing works have some limitations: need of active clouds [8, 3]; use of timeouts [6]; high costs [2]. Active clouds (e.g., Amazon EC2 and Google Computer Engine) can run algorithm code, unlike passive storage services (S3), but they require additional management. Timeouts are hard to setup in large networks such as the Internet. High costs are obviously undesirable.

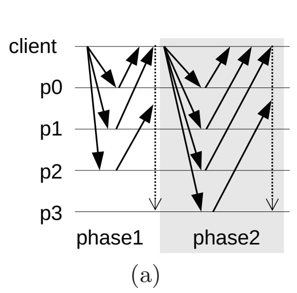
Our approach: we modify Byzantine quorum algorithms to allow the anticipation of certain communication phases, i.e., their execution before the conclusion of the previous phases. With the early activation of subsequent phases it is possible to obtain better performance and to use less resources. We validate this technique in a storage system named RafeStore, which deals with multi-version data whose new values can be saved with blind writes [1]. Applications as forums and RSS can be modeled in this way.

Contribution: this paper presents a novel technique named *anticipation of requests*, which aims to improve performance and reduce resource usage in systems that use Byzantine quorum algorithms.

[1]Examples of real cases: www.hylson.com/cnews.html

2. RELATED WORK

Research on reliable storage in clouds is fairly recent. Some works concern replication in multi-clouds. DepSky [2] stores data in multiple clouds using Byzantine quorum algorithms [7]; f faults are tolerated using $3f + 1$ clouds. Although DepSky uses erasure codes to reduce the amount of data stored in each cloud, its costs are around the double of systems that use a single cloud without replication. SPANStore [8] aims at reducing costs, allowing the clients to relax the consistency of data or to increase latencies of operation. SPANStore, however, requires active clouds to find the set of clouds that will comply with the restrictions of latency, consistency and cost. Active clouds have payment models based on how many hours the virtual machines are turned on, or how many hours the processing instances are working, while costs in passive clouds are oriented to data manipulation (read, write, store). Therefore, in passive clouds storage costs are only proportional to the amount of data used. MDStore [3] tolerates Byzantine faults and reduces the number of data clouds to $2f + 1$, but it requires active clouds and runs an atomic metadata service. Hybris [6] reduces the data clouds to $f + 1$ and uses a timeout to decide if f more clouds should be contacted to ensure that $f+1$ clouds will store the data. Timeouts can be safely used in synchronous networks, however the Internet can be characterized as a partial synchronous environment. Moreover, real contracts with clouds consider average demand [5], but peak values can generate higher latencies than the expected timeouts. It means that when a peak value happens the system will break operations due to time restrictions instead of treating the event only as a higher transmission time.



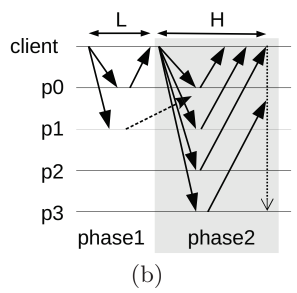


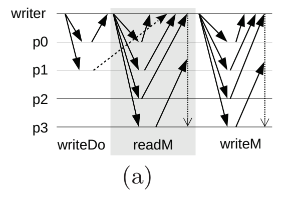
Figure 1: Contacting clouds with Byzantine quorum algorithms (a) without anticipation of requests and (b) with anticipation of requests.

3. PROPOSAL

Byzantine quorum algorithms are based on the idea that a minimum number of servers (clouds in our case) will always reply to a request [7]. Quorums algorithms aim to tolerate faults by masking them: the intersection among subsets of servers (quorums) has always enough correct servers to ensure the consistency of the replicated data. Figure 1(a) shows a protocol that uses quorums. The scenario has four clouds ($n = 4$) and tolerates one Byzantine fault ($f = 1$), following the usual relation of Byzantine quorum algorithms: $n = 3f + 1$. In the first phase, three clouds are contacted; when two replies return from clouds $p0$ and $p1$, the request to the third cloud $p2$ is cancelled, i.e., becomes useless. In phase 2, four clouds are contacted; when there are three replies, the request to cloud $p3$ is cancelled. These cancelled requests involve additional costs, bandwidth and latency.

In this paper we propose to reduce the waste caused by cancelled requests by *anticipating* communication phases in scenarios where continuous phases occur. *Anticipation* in this paper means to start requests earlier, before they would be started in Byzantine quorum algorithms. Figure 1(b) shows a quorum protocol in which two phases also require two and three replies in phases 1 and 2, respectively, as in Figure 1(a). However, we anticipate phases as follows: when the first reply of $phase1$ arrives, $phase2$ starts running as in a Byzantine quorum algorithm. When $phase2$ finishes, a *crucial* verification happens: *did the remaining reply from* $phase1$ *arrive*? If so, the protocol terminates; otherwise, extra actions will be required to ensure the amount of replies in the quorum. Note that we could appeal to a timeout mechanism for verifying if the required replies did arrive, but we do not specify time bounds as they could be problematic (see Section 2).

We can anticipate requests when storing data in multiple passive cloud providers in wide-area networks, e.g., to store data in clouds on the Internet. Passive clouds have standard storage interfaces, e.g., key-value storage with interfaces like $write(key, value)$ and $read(key)$. There is a specific type of storage application where data has multiple version and the content of new values is not related to previous ones. Many applications can make use of this approach. An example could be a panel with the name of the next person to be called in a waiting queue. The name that appears on the panel can be considered a variable with multi-version values, in which the name of the next person is not related to the name of the last person in the queue; the queue can be represented as a list of values for a variable $Name$ with different versions. Another example can be a forum, in which a topic can be defined as the initial value of a variable X and sequential posts to the topic can be considered the next values of X. To list the discussion, we can show all values of X. We define this kind of data as having *independent content*. This means that new values of X can be saved like blind writes [1] or write-only operations; only metadata, which controls the version of the data, needs to be updated.



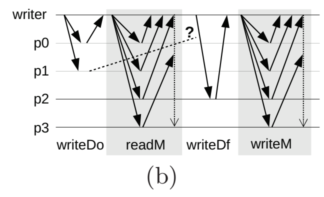


Figure 2: Write operation in (a) optimized operation and (b) with extra action required.

Figure 2(a) represents a quorum protocol with this semantic of multi-version values for data with independent content using the *anticipated requests* technique. This protocol is designed to tolerate Byzantine faults. Considering the number of tolerated faults equals one ($f = 1$), four clouds are required to store metadata (as in DepSky [2]) and $f + 1$ clouds effectively store the data (as in MDStore [3]). However, in MDStore the usage of Byzantine quorum algorithms requires contacting $2f + 1$ clouds to ensure that $f + 1$ clouds will store the data; using anticipation of requests, only $f + 1$ clouds are contacted to store data, and just in case of faults extra clouds are contacted. The phase $writeDo$ writes new values. When the first reply from $writeDo$ arrives, phase $readM$ starts in order to obtain the last data version. When phase $readM$ ends, it is verified if the remaining reply from phase $writeDo$ arrives – this is $true$ in Figure 2(a). As follows, phase $writeM$ updates metadata with the new data version. In Figure 2(b) the remaining reply from $writeDo$ phase does not arrive on time and an extra phase ($writeDf$) is required. The third cloud $p2$ is contacted to store the data value, and phase $writeM$ cannot start until the reply of $p1$ or $p2$ arrives. In Figure 2(b) the reply from $p2$ is represented as the one that concludes the phase of data writing.

The round-trip times (RTT) between the client and the clouds have to be considered in order to establish a rule about when the use of the anticipating requests technique is beneficial. The goals of the approach shown in Figure 2 are good performance and geo-replication. So, if $p0$ and $p1$ are the clouds with lowest RTT, this means that cloud $p0$ has to have low RTT and cloud $p1$ has to be far from $p0$, which means that $p1$ should have a greater RTT than that for $p0$ (without loss of generality). With this in mind, Figure 1(b) presents the latencies spent on each phase. It is possible to conclude that the latency of $p1$ should be lower than or equal to $L + H$, where L is the latency of the fastest cloud and H is the latency of the slowest cloud in the set of the $2f + 1$ fastest clouds. If HH is the latency of the slowest cloud (highest RTT), we can describe the favourable condition for the use of the anticipated request technique in this distributed storage problem as Condition 1:

Condition 1. Latency of clouds should obey the rule

$$HH \leq (L + H)$$

We named the presented system **RafeStore**: reliable, available, fast and economic storage system. Our goals are to improve performance with the anticipation of requests and reduce costs by contacting a smaller number of data clouds than in previous works. We also consider that geo-replication is desired for data survival and the RTT of clouds is previously known. Some details about how to link the written data with the metadata, drivers of clouds and others can be verified in the source code of the prototype available at www.hylson.com/rafestore15.

4. EVALUATION

The usage of *anticipation of requests* will be validated with the RafeStore storage system. Using this technique and satisfying Condition 1, a better performance is expected and only $f + 1$ clouds should be contacted. The first hypothesis to be checked can be stated as:

HYPOTHESIS 1. *If latencies of data clouds obey Condition 1, RafeStore will perform faster than Byzantine quorum algorithms, and no more than $f + 1$ data clouds will be contacted to store the data.*

The use of hybrid clouds is becoming common practice[2]. Moreover, it is fair to assume that the consumer can store some data in a local storage, increasing performance and maintaining complete control over his own provider. With this in mind, a consumer can use RafeStore with a local storage acting as a data cloud, if the consumer needs to store much data, or only to store metadata if consumer has a local storage with relatively limited capacity. We believe that RafeStore can be benefited by the use of a local storage, and the second hypothesis can be declared as:

HYPOTHESIS 2. *RafeStore can perform better than Byzantine quorum algorithms if there is a local storage to store data and metadata, or only metadata.*

Scenarios where Condition 1 is not satisfied could benefit from the fact that less clouds will be contacted in comparison to Byzantine quorum algorithms. Thus, it is possible that RafeStore can perform well even when Condition 1 is false. The third hypothesis is:

HYPOTHESIS 3. *RafeStore can perform as well as Byzantine quorum algorithms even in configurations where Condition 1 is violated.*

To test the hypotheses, experiments were conducted in a real distributed system. Table 1 shows details about the cloud providers: the region of the cloud data center and the estimated RTT[3]. The local provider was a computer with an Intel i7 3.5Ghz CPU with a 7200RPM HD in a local network Ethernet 10/100MBits, running the Redis database[4]. The cloud storage services were Amazon S3, Azure Storage, and HP Object Storage. The clients run in a computer with the same configuration as the local provider, on the same network. The experiments were executed in February 2015.

RafeStore is compared with the protocols of DepSky [2], which use Byzantine quorum algorithms to write and read

[2]hylson.com/cnews.html, items 13 to 15

[3]RTT estimates were obtained by writing a few bytes in the clouds 10 times.

[4]www.redis.io

Table 1: Information about Cloud Providers

id	provider	region	RTT
p0	Local	south of Brazil (SC state)	5ms
p1	Azure	west of Europe (Netherlands)	315ms
p2	HP	east of US (District of Columbia)	750ms
p3	AWS	west of USA (Oregon)	1200ms

Table 2: Arrangement of Cloud Providers

config.	data	extra data	only metadata
1	p0 and p1	p2	p3
2	p1 and p2	p3	p0
3	p0 and p3	p1	p2

data and metadata and to provide the same semantics offered by the clouds. The protocol DepSky-A replicates data completely, while the protocol DepSky-CA creates fragments using erasure codes and ciphers the data. DepSky's protocols contact $3f + 1$ clouds to store data and metadata in at least $2f + 1$ clouds. All protocols were implemented in Java; the erasure codes were taken from the JEC library, from the DepSky code[5]. The privacy (encryption) of DepSky-CA was not implemented since our goal was to evaluate the performance improvement using the anticipated requests technique in comparison to Byzantine quorum algorithms. The size of data was varied from 1KB (the size of a text paragraph) up to 16MB (the size of a high-res photo), with a multiplicative factor of 4 between the sizes. Clouds that store metadata are named $p0 \dots p3$; $p0$ and $p1$ also store data. This system tolerates one Byzantine fault. Table 2 shows the Configurations designed to test the hypotheses. Configuration 1 defines cloud providers in increasing order of latencies, which is expected to provide the best performance for RafeStore; Configuration 2 assumes that local storage acts as a cloud that store only metadata. Configurations 1 and 2 will test hypothesis 1 and 2. Configuration 3 breaks the Condition 1 in order to test hypothesis 3.

Results: Figure 3 presents latencies of write operations with RafeStore and the protocols of DepSky, using Configuration 1. RafeStore performs better than the other protocols. However, unexpectedly Condition 1 stayed true only for data of sizes up to 64KB. For the remaining data sizes, $2f + 1$ data clouds have to be contacted in order to ensure a quorum of $f + 1$ replies from data clouds. Table 3 shows how many times only $f + 1$ data clouds were contacted during the execution of the experiment with Configuration 1. Analyzing this behaviour, we can see that the time between the upload of data in local storage and the end of phase $readM$ was not enough to upload the data in $p1$, provided that the latency of $p1$ is much bigger than the latency of $p0$; this happened for data larger than 64KB. One possible solution to this issue is to upload data in a multi-part way – it is really possible in clouds like Amazon S3 – when data is bigger than some specific value. In our experiment, that value was 64KB. The number of parts into which the data should be divided for a multi-part upload is a point of future investigation. Considering hypothesis 1, it is possible to conclude that, although RafeStore had better latencies than the other protocols for all data sizes, there is not enough evidence that RafeStore contacts only $f + 1$ data clouds for all data sizes considered in the experiment.

[5]https://code.google.com/p/depsky

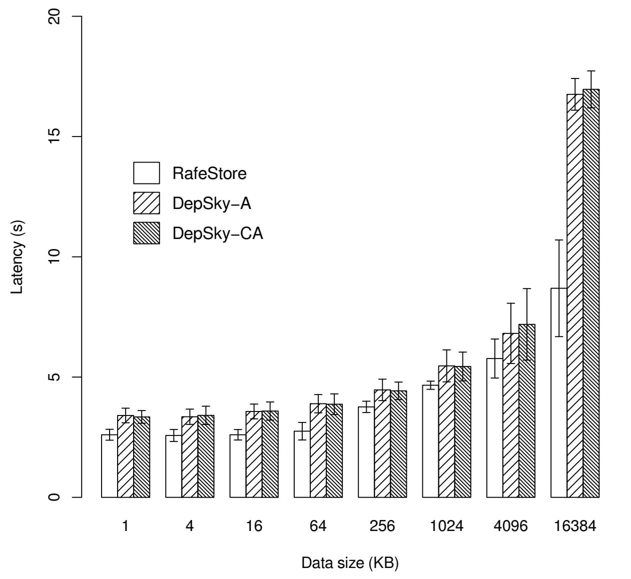


Figure 3: Latencies and data sizes, Configuration 1.

Table 3: Number of successful write operations (in RafeStore) contacting only $f+1$ data clouds.

data	KB					MB		
size	1	4	16	64	256	1	4	16
config. 1	60	59	60	52	0	0	0	0
config. 2	52	54	47	58	60	53	59	46

Configurations 2 and 3 presented latency graphics quite similar to Figure 3, so they are not presented. For Configuration 2, the overall latency increased in comparison with Configuration 1; this was expected since all data clouds have high RTT (in comparison with the local storage). Nevertheless, RafeStore performs as well as the DepSky protocols for all data sizes. Additionally, the majority of requests contacted only $f+1$ data clouds, as Table 3 shows. We can declare hypothesis 2 as *true*. Also, in Configuration 2 the dominating time concerns the write data phase, as shows Figure 4. In DepSky's protocols the phase of writing data also dominates. Details about latencies are available in www.hylson.com/rafestore15.

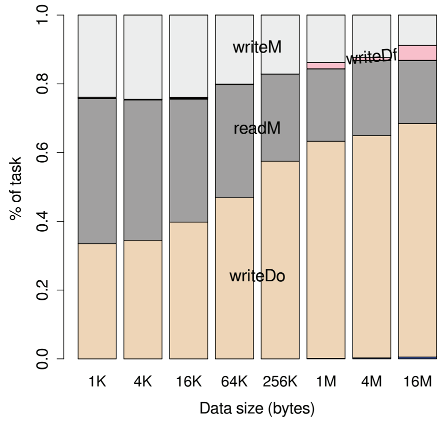


Figure 4: Task distribution in RafeStore, Config. 2.

In Configuration 3, RafeStore presented latencies higher than other protocols for the data sizes 256KB, 1MB and 4MB. However, data with 16MB performs better than the other protocols. Some assertions can clarify this behavior: DepSky protocols do not distinguish between clouds to store data and metadata, and the performance of these protocols maintained similar performances in all configurations. Although costs are directly related to the amount of data stored in clouds, maybe performance is more related to how many clouds have to store the data than just the size of the data. Considering these conjectures and the results of uploading 16MB, we can make some questions to be verified in future works: *i*) *what option could present better performance: send an integral data to one cloud or send fragments to multiple clouds? ii*) *is there a data size threshold from which the answer of the previous question changes? iii*) *to consider the latencies of clouds – instead of making them transparent – and also considering data size could modify these decisions?* Based on the results and considering hypothesis 3, it is not possible to conclude that RafeStore can perform as well as in Byzantine quorum algorithms when Condition 1 is broken. However, as expected, every execution had to contact extra clouds.

5. CONCLUSION

In this paper we presented a technique to anticipate requests in Byzantine quorum algorithms in order to improve performance and use less resources. The latencies of clouds must be considered in the process of choosing which clouds will store data and metadata, but a condition was defined to guide the consumer in this task. A storage system named RafeStore was implemented to validate the proposal, and we found that the technique is effective for cloud storage applications that use data with independent content.

Acknowledgment: Miguel Correia is bolsista CAPES/Brasil (project LEAD CLOUDS). This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 and CNPq 455303/2014-2.

6. REFERENCES

[1] D. Agrawal and V. Krishnaswamy. Using multiversion data for non-interfering execution of write-only transactions. *SIGMOD Record*, 20(2):98–107, 1991.

[2] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. *ACM TOS*, 9(4):12:1–12:33, 2013.

[3] C. Cachin, D. Dobre, and M. Vukoli. Separating data and control: Asynchronous BFT storage with 2t+1 data replicas. In *SSS*, pages 1–17, 2014.

[4] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[5] B. Cho and M. K. Aguilera. Surviving congestion in geo-distributed storage systems. In *USENIX ATC*, pages 439–451, 2012.

[6] D. Dobre, P. Viotti, and M. Vukolić. Hybris: Robust hybrid cloud storage. In *ACM SoCC*, pages 1–14, 2014.

[7] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[8] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *ACM SOSP*, pages 292–308, 2013.