Triton: a Peer-Assisted Cloud Storage System

Antonio Davoli
Computer Science Department
Sapienza University of Rome
davoli@di.uniroma1.it

Alessandro Mei
Computer Science Department
Sapienza University of Rome
mei@di.uniroma1.it

ABSTRACT

In these days, we are witnessing the evolution of cloud storage systems. One of the most intriguing research branches is focusing on merging these systems with peer-to-peer (P2P) networks, absorbing their benefits and thus creating new hybrid architectures. In this work, we present Triton[1], a peer-assisted cloud storage system designed to reach fast operations on users' data. By taking advantage of a direct communication channel among users, performed via the P2P network that interconnects all of them, Triton reaches two main goals. First, provides an acceleration on sharing operations and second, reduces the level of trust users must give to the cloud storage providers. Our solution also improves consistency constraints on users' data and reduces the latency requested to reach an agreement among peers. In the experimental results, we show how our prototype widely enhances the bandwidth usage for data sharing operations and also achieve important reduction on latency for the agreement operations.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed Systems*

Keywords

Cloud storage, Convergence Consistency, P2P Networks

1. INTRODUCTION

In the recent years, the massive growth of cloud users contributed to change the design process of storage systems. Cloud technologies indeed have turned mainstream. Companies like Apple, Amazon, and Google are increasing the number of their cloud based services. The popularity of tools such as iCloud and Dropbox is reaching a growing number of users. The benefits of cloud services, such as cost reduction, rapid and elastic deployment, and an easy management, are widely recognized and many enterprise users are planning to port their solutions on cloud infrastructures. These systems run now on an impressive number of machines, often replicated on multiple geographical locations [5]. Therefore, satisfying the high availability requirements for a huge number of users spread all over the world, as well as declared in *service level agreements* (SLAs), has become an important task.

In order to improve performance and quality of service of these distributed solutions, researchers and designers started to follow several opportunities. One of the most interesting approach, which is rising in the last times, tries to increment overall system performance by incorporating the benefits of peer-to-peer (P2P) networks [2]. Network clients can achieve faster downloads by using smarter applications which are able to cooperate and directly exchange data during the download phase from the primary site. The overall system performance is thus improved by offloading part of the distribution task directly to users bringing to a more efficient use of the network bandwith.

However, even though this approach is appealing from the distribution point of view, it remains mainly employed to deliver static files (e.g. peer-to-peer live streaming solutions [15]). But, what happens when users need to modify their shared data? How can we exploit this hybrid architecture in order to achieve faster data consistency and better sharing performance?

To answer these questions, we propose Triton, a peer-assisted cloud storage solution appositely designed for hybrid scenarios where a high number of players, who are interconnected by a P2P network, collaborate on data in the cloud resources. Users can share contents among themselves and use the cloud resource as coordination rendezvouz for the update operations and for sharing a common base of information (e.g. files' metadata). Triton exploits the direct communication among users to push update information in order to converge to a global consistency without waiting that the updates are propagated in the cloud's internal servers, thus minimizing the *inconsistency window*. This inconsistency window comes from the usage of eventual consistency models, that in order to provide high-availability in case of network partitions, relax the consistency require-

[1] *Triton* is the largest moon of the planet Neptune. It is the only large moon in the Solar System with a retrograde orbit, which is an orbit in the opposite direction to its planet's rotation. Similarly, we see the trust of clients toward the cloud.

PaPEC'14 April 13 - 16 2014, Amsterdam, Netherlands

http://dx.doi.org/10.1145/2596631.2596644.

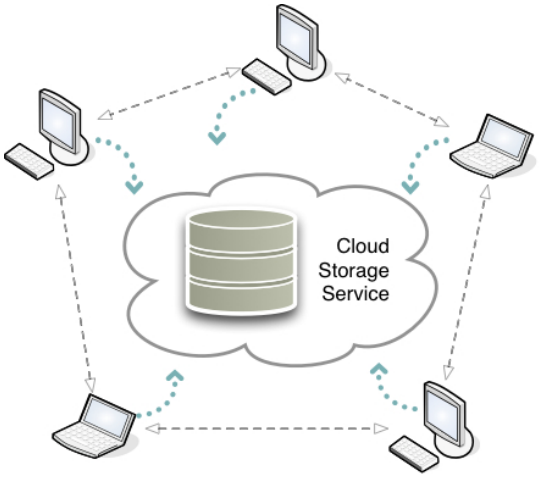

Figure 1: A Peer-Assisted Cloud Storage Deployment

ments. In Triton, instead of waiting that all the updates are propagated in the cloud's internal servers, and thus risking to access to stale data, peers start to push also the new information updates among themselves by exploiting a direct communication channel. Triton operations are based on a state machine replication protocol that poses minimum trust requirements in the cloud and also in the other peers. Our solution is designed to ensure interoperability with one or more cloud providers and does not require to run any special code on the cloud storage provider, as often happens when employed solutions such as Depot [16]. We build a prototype application that uses our protocol to provide a file sharing mechanism. The main contributions of this work are:

- proposing a new **architectural model** built upon a peer-assited cloud solution,
- designing a new **agreement protocol** for preventing concurrent accesses on shared data,
- creating a new **consistency model** for data and a file system implementation resistant to forking attacks [4].

In the next sections, first we describe the motivation behind this work, then we propose the design characteristics of Triton system, and finally we show the performance results for sharing operations and for latency reduction obtained by our Triton prototype.

2. MOTIVATION

Earlier work proposed many solutions for cloud storage with minimal trust assumptions [16]. However, these results do not consider a dynamic environment where users can show up and disappear continuously. In addition many of these solutions are designed to provide consistency models derived from *fork* consistency* [14]. *fork* consistency* is a weaker safety property than linearizability [9], and can be achieved when less than two thirds of the replica population are faulty. SUNDR [13] showed how to achieve *fork consistency* (slightly stronger than fork*, but still weaker than linearizability) in the presence of a faulty server and non-faulty clients [4]. However, a faulty server can mount a *forking attack* by concealing operations, which causes the system state to diverge into multiple possibilities for different clients. Even though these solutions permit to recovery from forking, performance are always influenced by latency of the recovery operations. Indeed, in the case of a fork event they require interaction, based on a gossip-based protocol, among users in order to overcome the fork status. In this work, we wanted to principally address this problem preventing the possibility of these faulty behaviors by explicitly enabling communication among nodes.

On the other hand, Byzantine fault tolerance solutions require to work with a fixed set of replicas, which are chosen at the beginning of the deployment phase and then rarely changed. In this way, it is impossible to change the total number of failures after the starting procedure. Alvisi et. al [1] proposed a solution with a faulty threshold variable in a fixed range and many other solutions rely on a monitoring node to manage the nodes participation in the replicas group. These solutions, however, do not address a dynamic and hybrid network where clients act also as replicas.

Moreover, to meet high-availability requirements cloud storage solutions often provide a weaker consistency model, called *eventual consistency* [18]. This model does not guarantee that updates are available as soon as the update operations have finished, leaving a temporal window of inconsistency. The Brewer's conjecture has aroused great interest among the scientific community. In fact, shortly after Gilbert and Lynch [8] provided a formal proof of the statement by proving that is indeed impossible to achieve all three properties in the asynchronous network model. The result's reformulation in according to the Gilbert and Lynch's work is:

THEOREM 1 (CAP THEOREM). *In a network subject to communication failures, it is impossible for any web service to implement an atomic read/write shared memory that guarantees a response to every request.*

The CAP Theorem illustrates a more general trade-off that often appears in the study of distributed computing: the impossibility of guaranteeing both *safety* and *liveness* in an unreliable distributed system [7]. With *Safety*, we intend that an algorithm is safe if nothing bad ever happens. The consistency properties of the CAP Theorem is a classic safety property. With *Liveness*, an algorithm is live if eventually something good happens [7]. The result simply expresses that it is not possible to achieve both safety and liveness in an unreliable distributed system [7]. In the last decade, researchers and designers have used the CAP results as motivation to push their efforts towards the exploration of new tradeoffs and solutions for distributed system. This new generation of systems, with few or no isolation guarantees, have been termed *NoSQL* [2], as opposed to the more traditional SQL systems. The NoSQL movement proposes alternative designs with less constrained consistency models and first focuses on producing high-availability solutions. The model adopted by NoSQL solutions is the BASE (as opposed to the ACID). The BASE properties are: *Basically Available*, *Soft state*, and *Eventual Consistency*. In [6], authors describe the design of Dynamo, a highly scalable distributed data store built appositely to statisfy high-availability constraints but that, however, sacrifice consistency in presence

[2] `http://en.wikipedia.org/wiki/NoSQL`

of network partitions by risking to propose to users stale version of data.

On the other hand, there are several peer-to-peer (P2P) systems that have looked at the problem of data storage and distribution. The first generation of P2P systems, such as Freenet and Gnutella[3], were predominantly used as file sharing systems. These were examples of unstructured P2P networks where the overlay links between peers were established arbitrarily. In these networks, a search query is usually flooded through the network to find as many peers as possible that share the data. P2P systems evolved to the next generation into what is widely known as structured P2P networks [4].

Oceanstore [11], for example, provides a global, transactional, persistent storage service that supports serialized updates on widely replicated data. To allow for concurrent updates while avoiding many of the problems inherent with wide-area locking. Oceanstore resolves conflicts by processing a series of updates, choosing a total order among them, and then applying them atomically in that order. It is built for an environment where the data is replicated on an untrusted infrastructure. However, Oceanstore does not consider consistency requirements in its design and works with a hierarchical overlay of nodes. Triton, instead, gives to users a mechanism to reduce this window by understanding if the data received by the cloud match effectively the last updated version. Furthermore, Triton design is directed to hybrid and peer-assisted cloud architectures where data are shared among the peers by also exploiting the cloud resources.

3. TRITON DESIGN

In this section, first we present the system model and the design goals behind Triton, together with an overview the Castro and Liskov's PBFT protocol [3], a mechanism for implementing a practical state machine replication solution. Then, we introduce Triton protocol details, its operations workflow and finally, we describe the consistency model employed by our solution.

3.1 System Model

The *state machine approach* is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. State machine replication is an abstraction that represents a deterministic service, in which a starting state (e.g., an empty database) and the sequence of read-compute-write operations at the service determine precisely the state of that service at the end of the operation sequence. Such state machines are relatively straightforward to implement on a single, single-threaded server at an individual computer, though any faults at that computer always cause a service failure.

State machine replication solutions work with a fixed number of replicas [3, 10]. This number is selected during the deployment phase and it represents the upper bound of contemporary faulty replicas the system is able to afford. Usually, protocols protect against f faulty nodes by working with at least $3f + 1$ replicas. Safety and liveness guarantees are thus no valid when more than a third of the replicas is faulty [12]. Estimation a priori of this number is a hard task and the result is an inflexible work environment.

[3] `http://freenetproject.org`, `http://www.gnutella.org`

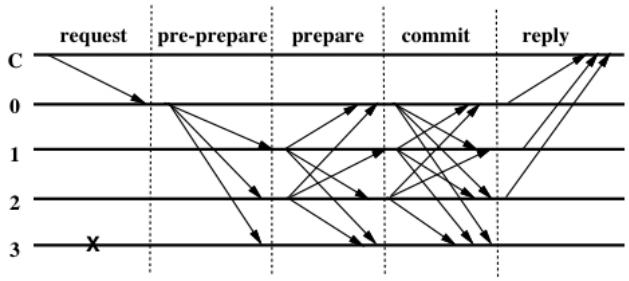

Figure 2: Three Phases Practical Byzantine Fault Tolerance

In this work, we investigated how a solution can be flexible and work in according to network changes. Moreover, we considered a system where replicas and clients are the same entities, and they sharing the same cloud resources. In our model indeed, we assume that nodes connection is based on a asynchronous network (e.g. Internet) and the network model may corrupt or fail to deliver messages and can introduce delay on communications. The adversary can take control of nodes but is not able to break the cryptographic schemes adopted. We also assume that nodes and also the cloud can act arbitrarily in according to a Byzantine failure model.

3.2 Background: PBFT

Castro and Liskov's PBFT protocol [3] is a replicated, fault tolerant mechanism for implementing a state machine [17]. For fault-tolerance reasons, it often makes sense to implement the state machine abstraction over a population of such potentially faulty computers interconnected via a potentially faulty network, hoping that even if some computers fail, the service as a whole can continue functioning correctly. Unfortunately, implementing the state machine abstraction over such a population and network is no simple task. In PBFT, each participating computer implements the entire state machine on its local replica of the service state, and replicas communicate with each other to ensure that they all execute the same sequence of operations, and mask individual computers faults [4].

In PBFT, a client c multicasts a *request* message:

$$< REQUEST, t, o, c >_{c,R,1}$$

to the N service replicas in replica set R, where o is the operation requested, and t is the timestamp. The client accepts a reply for its request (and only then can submit another) when it receives $\lfloor \frac{N-1}{3} \rfloor + 1$ valid matching REPLY messages, forming the *reply certificate*:

$$< REPLY, v, n, t, c, r >_{R,c,\lfloor \frac{N-1}{3} \rfloor + 1},$$

where v is the view number, n is the assigned sequence number, and r is the result of the request. A *view* is a particular assignment of roles to replicas: the single active *primary* vs. the passive *backups*; when the primary changes, so does the view number v.

Replicas linearize requests via a three-phase agreement protocol (Figure 2), starting when the primary (chosen to be the replica with identifier $p \equiv v \ mod N$ and indicated

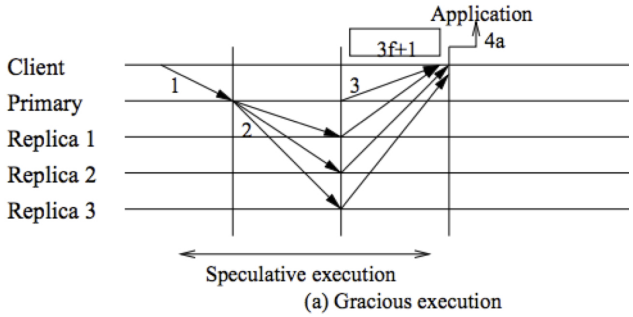

Figure 3: Zyzzyva: Speculative version of BFT

with 0 in Figure 2) multicasts to R a newly received request message *req*, encapsulated within a message

$$< PREPREPARE, v, n, req >_{p,R,1}.$$

When backup replica i receives this PREPREPARE, it multicasts to R a: $< PREPREPARE, v, n, req >_{i,R,1}$ message. Once replica j has collected $2\lfloor\frac{N-1}{3}\rfloor + 1$ PREPREPARE or PREPARE messages from distinct replicas for this request (which constitute the *prepared certificate* for this request of the form $< PREPARE, v, n, req >_{R,R,2\lfloor\frac{N-1}{3}\rfloor+1}$), the request becomes prepared. To complete the protocol, a replica with a prepared request then multicasts to R a:

$$< COMMIT, v, n, req >_{j,R,1}$$

message. When replica k collects $2\lfloor\frac{N-1}{3}\rfloor + 1$ such messages (which constitute the *committed certificate* of the form $< COMMIT, v, n, req >_{R,R,2\lfloor\frac{N-1}{3}\rfloor+1}$), the replica has established the linearized sequence for this request, committing to execute it as soon as it can; this concludes the *agreement* portion of the PBFT protocol for this request, whose purpose is to ensure that the replicas agree on a single operation sequence for the service, as more clients submit requests for further operations. A replica can execute the request in its local state as soon as it has finished executing the committed requests for all sequence numbers lower than n. It packages the result in a REPLY message, which it sends to the client directly. When the client has received a quorum of such matching replies – the *reply certificate* described above – the execution portion of the protocol concludes; the purpose of the execution portion is to represent to the client accurately the service state (and reply to the client request accordingly), as determined by executing the sequence of operations that the agreement protocol portion maintains. Furthermore, the PBFT suite also contains a view-change protocol that is used to change the system primary when the primary is suspected faulty.

3.3 Triton Protocol

Triton design is based on a state machine replication protocol designed for environments with an high variable number of players. In this hybrid environment, it is hard to deal with clients who, even though they act also as replicas, can not be imposed to stay always connected.

We designed Triton to work with a variable number of replicas, aiming at closing each round with the maximum, even if dynamic, quorum size available. Replicas number can increase or decrease during the time, and that brings two main problems. Keeping the leader always online, and cooperate in presence of byzantine behaviors. We developed this protocol with three main goals: (a) create a fast algorithm for replication, (b) manage an high variable number of members, (c) deal with environments where clients are simultaneously replicas. Solutions with fixed quorum are hard to tune when replicas are able to appear in late rounds or to close the connection early. On the other hand, in literature exist *speculative* approaches of state machine replication, such as Zyzzyva [10], that rely on clients to detect inconsistency. As showed in Figure 3, Zyzzyva uses speculation to reduce the cost of BFT replication. In Zyzzyva, replicas reply to a client request without first running an expensive three-phase commit protocol to agree on the order to process requests. Instead, they optimistically adopt the order proposed by a primary server, process the request, and reply immediately to the client. If the primary is faulty, replicas can become temporarily inconsistent with one another, but clients detect inconsistencies, help correct replicas converge on a single total ordering of requests, and only rely on responses that are consistent with this total order [10]. We designed Triton to adopt a speculative approach, but with a control mechanism that permits to other nodes to understand the status of operations. In Triton, indeed, active clients are part of the system and they try to work cooperatively in order to conclude the protocol steps. Forcing the peers to wait until the end of three phases increases delays, since the dynamic environment in Triton requires to finish the round as soon as possible.

3.4 Protocol Specifications

Triton acts as a funnel and helps to serialize accesses to shared cloud resources. The protocol has been designed to achieve a result as soon as possible, considering that often peers are online only for a limited amount of time. We developed a replication structure which permits to have different rounds happening at the same time. Thus, instead of using a global *view* for all the system, as proposed in classical BFT solutions, we propose a series of different *topics*, indipendent from each other and with different advancing statuses and different groups of clients involved. In any case, groups can have overlapping set of peers. We define as the *leader* for a particular topic the one who proposes a new round update. With this choice, we wanted to empower the leader so that he has the responsibility to care about the real commit of its updates.

Therefore, the only information peers need to know are the number of elements that are going to participate to each round and the time information (i.e. the last timestamp approved) about the last agreement. These fields are always updated and stored in the cloud resources. Triton protocol is composed by an *agreement phase* and a *check-topic* algorithms. The agreement part is a hybrid approach which merges a speculative commit with quorum requirements. The Figure 4 shows the protocol's steps.

An user who wants to update a topic creates a *request* message and broadcasts it to the other users sharing the same topic. The message contains:

$$< REQUEST, t, o, n, H, timestamp@c, c >_c,$$

where t is the topic, o is the operation, n is the sequence

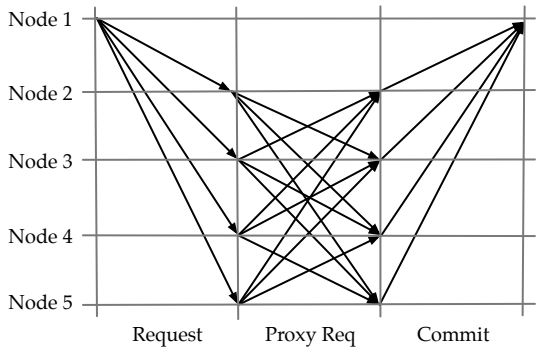

Figure 4: Triton Protocol Steps

number, H is the hash of history, $timestamp@c$ is the timestamp for the operation, and c symbolizes the user ID. The packet is protected with user's signature. Other members that received the message prepare a *proxy-request*, which is created by extending the initial requests with the id of receiver, by

$$< PROXYREQ, t, o, n, H, timestamp@c, c, p >_p.$$

When a peer received a quorum of at least $2f + 1$ message, it proceeds to the send a *commit* message. The commit message,

$$< COMMIT, t, o, n, H, R, timestamp@c, c, p >_p,$$

contains, other the other fields, an array R of all the *proxy requests* received. The leader node proceeds to commit the operations and uploads the new info on the cloud if the number of commit messages is bigger than the $2f + 1$ quorum.

In case of a number of messages under the described threshold, the requester node starts to contact peers that did not answered. In case of a new missing answer a new request with a less quorum threshold will be send. This permits to achieve an agreement with a smaller quorum but, as we discuss in the next section, permits to the system to guarantee liveness.

In the cloud are also stored meta information about the number and identities of peers for each topic. In order to update these information we use a *check-topic* algorithm that it is similar to the agreement phase but contains *heartbeats* information to check if elements are still connected to the network.

3.5 Triton Operations and Consistency

In this section, we describe how we use the Triton protocol to build a distributed storage system that achieves better consistency performance. Modern systems, such as Dynamo [6], are designed aiming at high availability performance. When a system is designed to provide high availability, consistency constraints get relaxed creating new forms of *eventual consistency*.

Therefore, by supposing that cloud storage provides eventual consistency guarantees, Triton increases performances and prevent users to access to stale data. Triton proposes a new consistency model that aim at reaching consistency of data faster than others model presented in literature. Indeed, one of the most important issue of eventual consistency model is related to the access to a file as a *whole resource*.

In our system, we propose instead to logically split the files, which are shared among the users, in smaller blocks of fixed size in order to decrease the amount of data exchanged by the peers after a file update. We use the cloud as coordination rendezvous where are stored only the meta information about files, such as the list of users who are sharing the files and a Merkle Tree computed by the file blocks to track the file updates[4].

There are two main reasons that motivate this result. First, files are subdivided in blocks, the little amount of size of these blocks permits the cloud storage to work with a less amount of data in its operation, propagating the updates faster in the data center. Second, users are an active part of the system, and through our pushing system, they will be aware of files modifications as soon as possible and directly from the actor who produced them. Furthermore, read operations are also improved because peers know which blocks are under a write operations. The selected tree node represents the *topic* of the protocol, as described in the previous section. If received a correct number of certificates, it proceeds to update the meta information in the cloud storage and push directly to the peers the *diff* data needed to update the file to the last version. The timestamp inserted into the messages permits to avoid the multiple clients win the round and modify the resources at the same time. After the winning of the round, peers are going to expect the message from the winner with the data information related to that operation. If this does not happen, they can raise an exception and block that update. The last data message actually acts as a further prevention against a malicious winner. If instead the cloud provides different information the network is able to understand the quality of cloud answer by communicating and checking the latest updates.

In the cloud are kept all the information to build the last version of the files. Thus, read operations are organized to be as fast as possible. A peers who wants to obtain a files retrieve the information about the hash tree that composes the blocks and a list of the peers who own the data blocks. In this way, it can start multiple downloads from multiple peers in order to receive the data as soon as possible and minimizing the cloud's access. When a peer receives a data block request it is aware if some round it proceeding related to that block (or some level higher), so it can wait to receive the new updates or give to the peers the last valid update.

Consistency *When cloud provides eventual consistency guarantees, Triton achieves a faster convergence consistency.* Suppose two users p_1 and p_2 share the same file f in the system. When p_1 starts a write operation, that means p_1 won the agreement round and proceeds with the update of the blocks set B of f. Now suppose that p_2 needs to read f. If the blocks requested by p_2 are not in the same subtree of B, the system guarantees that p_2 can ask to other peers the latest version of the blocks without any consistency violation. If p_2 accesses to some block in in the same subtree of B, we must consider two options. First, the update of p_1 has been delivered and the system returns the newest value, second, in case p_2 receives a stale data is always able to recognize it by checking the metadata received during the agreement steps. If a new client asks to the cloud information about a file and, once received the meta information, tries to recompose the file, peers can send to it a message to invalidate

[4]We are also working on a extended version where the cloud also keeps data incremental backups.

some blocks because in some update operation. In this way, clients can reach a faster consistency by reducing the cloud inconsistency window.

Triton architecture consistency model converges faster by exploiting communication and hierarchical data structures. Let suppose a user A uploads a file f of size N on the cloud storage provider. In a real world scenario this operation is going to take an amount of time t_N, which is related to the file size. Now consider that A alter a portion of file and must upload again on the cloud. This operation will require, again, around t_N. In case this file will be shared with another peer B, this needed times are going to increase. When Triton is employed, A will send to the cloud only a portion of the metadata required to the cloud to update the information on the file in less the t_N.

Forking Topics *Triton system permits a stronger defense against forking behaviors and also limit the users to read stale data.* Suppose that the cloud provides a modified version of the meta information to different users. This scenario will be detected as soon as users interact, because a request with an irregular value will raise a check by users on the contents. In this case, peers start a *check-topic* round to update all the users contents to the last committed version.

4. EXPERIMENTAL RESULTS

In this section, we describe the implementation details of our prototype system. We built Triton as a client library and created a file system based on the protocol's access policies. In this way, we guarantee a interoperability with several cloud storage providers. Many other systems instead require to run part of the code on a centralized server.

Triton Library We implemented a prototype version of the library by using the event-driven paradigm and it has been coded using the *libevent*[5] library. Thus, the main body can manage efficiently several connections, with the peers and the cloud, at the same time. In the experimental setup, we used our system interacting with the APIs provided by Amazon S3. Nonetheless, it is extremely easy to change the module and to work with other providers or private clouds.

TritonFS The prototype version of the Triton file system has been designed as a file system in user space (FUSE). It connects all the file systems calls to the Triton library functions in order to interact with the peers and the cloud resources and proposes a transparent interface to final users.

We present two kinds of experiments we conducted. The first was intented to compare protocol performance against a classical implementation of the previously described PBFT state replication protocol. The testbed used was a Intel Xeon Quad-Core, where we created up to eight virtual machines, each one running an instance of Triton and PBFT client. The Figure 5 presents the result of the run of Triton and BFT, which refers to an implementation of PBFT, with a 0/0 scenario. This signature means that message exchanges had no real payload, but the test aimed at understanding how latency increases just for the protocol usage with empty messages. As it is possible to see in Figure 5, by using a reduced numbers of messages and thus converging faster to the commit, Triton reaches the commit with 8 replicas in almost the half time required by BFT, 0,18 seconds against 0,24 seconds of BFT.

The Figure 6, instead, presents a similar test done in

[5]`http://libevent.org`

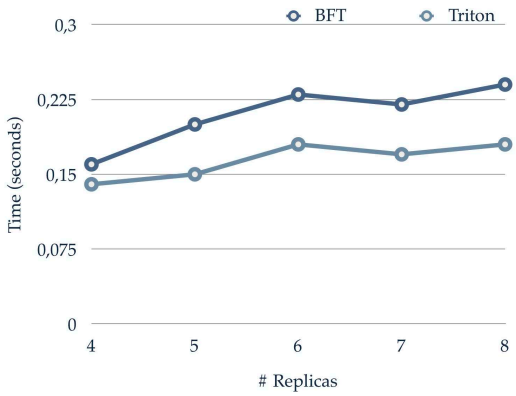

Figure 5: Triton vs BFT Latency Test without Payload

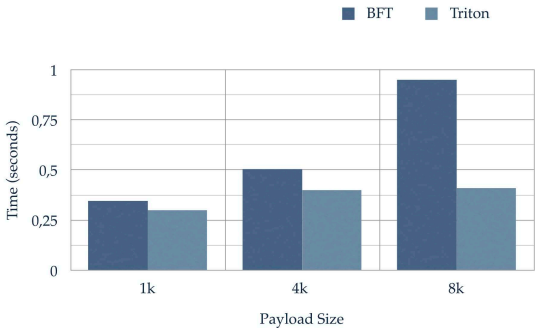

Figure 6: Triton vs BFT Latency Test with variable Payload

the same environments with eight but with an increasing amount of payload used in both the protocols. It is possible to see how Triton overcomes BFT in the scenario with a payload of $8k$. This result is a consequence of the cryptographic additional payload that both the protocols use, but by having a bigger number of messages to manage BFT tends to slow the performance down than Triton, that is able to complete a commit in less than the half amount of time required by BFT. Indeed, we obtained a total of 0,95 seconds for BFT against only 0,41 seconds of Triton.

Then, as second experiment, we wanted to investigate the performance of Triton compared to the usual cloud access. In the experiment, we employed six clients sharing the same cloud storage instance. Four of the clients were set up in virtual instances in the Amazon EC2 datacenter located in Oregon, and the other ones in our laboratory located in Rome. The Figure 7 shows the results of the read performance using Triton compared with the direct usage of the S3 cloud. The experiment presents the results the read time for different file sizes. We employed data size in the range from 16Mb to 512Mb. In our system, data blocks were asked to all the available peers. As it is possible to see in the Figure 7 in the case of file size equal to 512Mb, Triton, by exploiting the direct channel among the users and by using the network bandwidth more efficaciously, achieves a download time of 93 seconds in respect to 232 seconds required by the

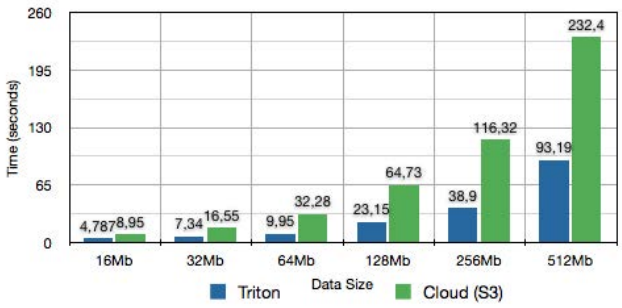

Figure 7: Triton Performance

S3 direct download. By using Triton we increased the read performance of a 2.5x factor.

5. CONCLUSION

In this work we presented Triton, a peer-assisted cloud storage systems that provides concurrent read and write operations. In addition, we showed how Triton consistency model decreases the inconsistency window of cloud data, by achieving faster downloads and also faster commit operations. Triton represents a first step of our investigation process of these new hybrid solutions, where peers and remote systems are closely interconnected each others forming more complex and challenging systems.

Further work. Considering the dynamism of the network, we are investigating two main directions: *protocol performance* under more complex failures models based on a game theoretic approach such as where peers behaviors are influenced by the previous rounds, and *protocol security* against membership and identity attacks. We are also planning to incorporate geographical locations of peers in the sharing process. In addition, we have planned to improve our prototype to study the performance results in large scale deployments. Another possible improvement we are investigating regards the integration of Triton with fully distributed social network, like Diaspora[6].

6. REFERENCES

[1] L. Alvisi, E. T. Pierce, D. Malkhi, M. K. Reiter, and R. N. Wright. Dynamic byzantine quorum systems. In *DSN 2000*.

[2] N. Carlsson, G. Dan, D. Eager, and A. Mahanti. Tradeoffs in cloud and peer-assisted content delivery systems. In *P2P 2012*.

[3] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI '99*.

[4] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 189–204, New York, NY, USA, 2007. ACM.

[5] J. C. Corbett and al. Spanner: Google's globally-distributed database. In *OSDI'12*.

[6] `https://joindiaspora.com/`

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07*.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. In *Journal of the ACM, 32(2):374âĂŞ382, 1985*.

[8] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *In ACM SIGACT News*, 2002.

[9] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[10] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *SOSP 2007*.

[11] J. Kubiatowicz, D. Bindel, C. Chen, Y., S., P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. In *SIGARCH Comput. Archit. News 28, 5 (Dec. 2000), 190-201.*

[12] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[13] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI'04*.

[14] J. Li and D. Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI'07*.

[15] N. Magharei, R. Rejaie, and Y. Guo. Mesh or multiple-tree: A comparative study of live p2p streaming approaches. In *INFOCOM 2007*.

[16] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: cloud storage with minimal trust. In *OSDI'10*.

[17] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. In *ACM Computing Surveys, 22(4):299âĂŞ319, 1990*.

[18] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.