Harmony: Towards Automated Self-Adaptive Consistency in Cloud Storage

Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu
INRIA Rennes-Bretagne Atlantique
Rennes, France
{*houssem-eddine.chihoub, shadi.ibrahim, gabriel.antoniu*}*@inria.fr*

María S. Pérez
Universidad Politécnica de Madrid
Madrid, Spain
mperez@fi.upm.es

***Abstract*—In just a few years cloud computing has become a very popular paradigm and a business success story, with storage being one of the key features. To achieve high data availability, cloud storage services rely on replication. In this context, one major challenge is data consistency. In contrast to traditional approaches that are mostly based on strong consistency, many cloud storage services opt for weaker consistency models in order to achieve better availability and performance. This comes at the cost of a high probability of stale data being read, as the replicas involved in the reads may not always have the most recent write. In this paper, we propose a novel approach, named Harmony, which adaptively tunes the consistency level at run-time according to the application requirements. The key idea behind Harmony is an intelligent estimation model of stale reads, allowing to elastically scale up or down the number of replicas involved in read operations to maintain a low (possibly zero) tolerable fraction of stale reads. As a result, Harmony can meet the desired consistency of the applications while achieving good performance. We have implemented Harmony and performed extensive evaluations with the Cassandra cloud storage on Grid'5000 testbed and on Amazon EC2. The results show that Harmony can achieve good performance without exceeding the tolerated number of stale reads. For instance, in contrast to the static eventual consistency used in Cassandra, Harmony reduces the stale data being read by almost 80% while adding only minimal latency. Meanwhile, it improves the throughput of the system by 45% while maintaining the desired consistency requirements of the applications when compared to the strong consistency model in Cassandra.**

***Keywords*-consistency; replications; data stale; Cassandra; cloud; self-adaptive**

I. Introduction

Cloud computing has become a very popular paradigm. Part of its success is due to its flexibility, elasticity, and scalability. Clients can lease just the resources they need on the cloud in a Pay-as-You-Go manner with very little knowledge of the physical resources. For cloud computing to be a real alternative to grid and cluster computing, it should perform well with everyday applications. Nowadays many of these applications are data-intensive: companies like Google, Amazon, and Facebook deal with peta- and terabytes of data everyday. In this context, storage management and performance within clouds is extremely important.

Storage systems often rely on replication to achieve availability, data durability, fault tolerance, disaster recovery. However, with the use of replication comes the issue of consistency. Insuring data consistency at all times by means of synchronous replication results in very high operation latencies and thus in bad performance. Moreover, cloud storage systems are deployed on a wide area scale and data are replicated over geographically distant areas. Consequently, latencies become even higher when ensuring strong consistency. These high latencies may generate significant financial losses for service providers that use such storage systems. For instance, the cost of a single hour of downtime for a system doing credit card sales authorizations has been estimated to be between $2.2M-$3.1M [1]. Consequently, cloud providers tend to rely on storage systems with eventual consistency. Eventual consistency allows the system to return some stale data at some points in time, but ensures that all data will *eventually* become consistent.

In this context, many cloud storage systems have been developed such as Amazon Dynamo [2], Cassandra [3], Google BigTable [4], Yahoo! PNUTS [5], and HBase [6]. These solutions are practical to use as cloud and web service storage backends. They allow many web services to scale up their systems in an extreme way, while maintaining performance with very high availability. For example, Facebook uses Cassandra to scale up to host data for more than 800 million active users [7]. However, the undoubted availability and performance of such solutions prove to be too costly in terms of inconsistency. As shown in [8], under heavy reads and writes some of these systems may return up to 66.61 % stale reads. This is an *alarming* rate, meaning that most probably two out of three reads are useless.

In this paper, we address these tradeoffs between consistency and performance on the one hand, and consistency and availability on the other. Accordingly, we propose an automated and self-adaptive approach, named Harmony, that tunes the consistency level at run-time to reduce the probability of stale reads caused by the cloud system dynamicity (i.e., the network latency which directly affects updates propagation to replicas) and the application's demands (i.e., the frequency of access patterns during reads, writes and updates), thus providing adequate tradeoffs between consistency and both performance and availability. Harmony embraces an intelligent estimation model to automatically identify the key parameter affecting the stale reads such as the system states (network latency) and application's requirements (current access pattern). Harmony, therefore, elastically scales up/down the number of replicas involved

in read operations to maintain a low (possibly zero) tolerable fraction of stale reads, hence, improving the performance of the applications while meeting the desired consistency level.

We have implemented Harmony with intensive evaluations on Cassandra cloud storage system on different platforms: Grid'5000 [9] – an academic experimental testbed based in France – and Amazon Elastic Compute Cloud (EC2) [10]. We use the Yahoo! Cloud Serving Benchmark (YCSB) [11] to mimic a real cloud serving environment with elastic access pattern workloads (e.g., heavy read load, heavy update load etc). We show that Harmony can achieve good performance without exceeding the tolerated number of stale reads on the applications. For instance, in contrast to the Cassandra's static eventual consistency, Harmony with 20% tolerated stale reads reduces the stale data being read by almost 80% while adding only minimal latency. Moreover, it improves the throughput of the system by 45% compared to the Cassandra's strong consistency model while maintaining the desired consistency requirements of the applications. The contributions of this work can be summarized as follows:

- We propose an automated self-adaptive approach that gradually and dynamically scales the consistency level to best suit the application requirements, while taking into account the system state.
- By means of probabilistic computations we provide an estimation of the stale reads rate in a storage system for a running workload.
- We evaluate the proposed approach with an extensive set of experiments both on a bare metal environment (Grid'5000) and on a cloud computing platform (Amazon EC2). The results show a significant improvement in performance compared to the traditional strong consistency approach, while providing better consistency than static eventual consistency approaches.

This paper is organized as follows. Section II briefly discusses consistency-performance and consistency-availability tradeoffs in cloud storage and zooms on the the eventual consistency model. Section III presents our adaptive model to handle consistency at run-time. Then section IV gives more details on how to estimate the amount of stale reads in the system. In section V we describe the Harmony implementation and present detailed results of experimental evaluations. Section VI discusses related work. Finally, section VII presents our conclusions and future work.

II. Background

A. Tradeoffs between consistency, performance and availability in cloud storage

The CAP theorem, which was introduced in [12] then proved in [13], tackled a real design challenge for today's storage systems. The theorem states that only two out of the three following properties can be guaranteed simultaneously: *Consistency, Availability, and Partition tolerance*[1]. Since partition tolerance is necessary for scalable distributed systems that rely on networking, a real tradeoff between consistency and availability needs to be defined. Twelve years after his CAP theorem, in [18] Brewer considers the tradeoff consistency-performance as even more important. He argues that partitions are rare and one obvious solution is to predict their occurrence times. In this case, the tradeoff between consistency and availability should be considered only during the partitions. In contrast, the consistency-performance tradeoff is a permanent one. Abadi [19] makes a connection between latency and availability. When latency is higher than some timeout the system becomes unavailable. Similarly, the system is available if latency is smaller than the timeout. However, the system can be available and exhibit high latencies nonetheless. This implies consistency-latency and consistency-availability tradeoffs are connected and exist outside CAP.

With cloud storage systems being deployed on a wide area scale with data replicated over geographically distant areas, latencies become even higher in this case for traditional storage systems that ensure strong consistency. Eventual consistency [20] therefore was introduced as an alternative to the traditional strong consistency. Such consistency allows the system to return some stale data at some points in time, but ensures that all data will become *eventually* consistent. Influenced by such tradeoffs, many storage system designers opt for BASE properties *(basically available, soft state, eventually consistent)* [21] instead of ACID *(Atomicity, Consistency, Isolation, and Durability)* in order to relax consistency rules, and hence favor performance and availability. Today, some of the main cloud storage providers rely on storage systems with eventual consistency. For example, Amazon Dynamo [2] is the storage system for most of Amazon services including Amazon Simple Storage Service (S3) [22]. Dynamo enables high availability and low latency for Amazon web services.

B. Zoom on eventual consistency in cloud storage

The way consistency is handled has a big impact on performance. Traditional synchronous replication (strong consistency) dictates that an update must be propagated to all the replicas before returning success. In cloud services where data updates occur often, it is difficult to keep the consistency among replicas in the entire cloud storage system. To solve this problem, eventual consistency with asynchronous quorum replication has been introduced. Here the consistency level is chosen on a per-operation basis and is represented by the number of replicas in the quorum

[1]Interestingly, Windows Azure Storage (WAS) [14] and Scatter [15] are distributed storage systems that strongly rely on Paxos [16] to simultaneously offer consistency and availability while tolerating faults in large-scale settings, but with an extra cost on performance in contrast to eventual consistency [17].

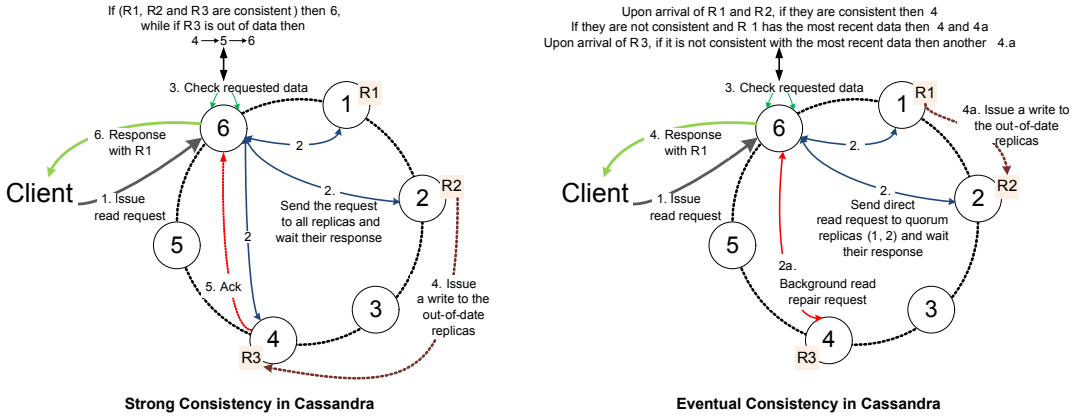


Figure 1. Synchronous replication *vs* Quorum replication in Cassandra: *When a client connects to a node in Cassandra cluster and issues a read request, this node will serve as the coordinator (node 6) for this particular operation. However, the read operation consistency is set to* ***ALL*** *"strong consistency", the coordinator will wait for all the replicas to respond with the requested data. If they are not consistent –stale data is detected– the coordinator will not respond to the client until the inconsistent nodes have been repaired with the newest data. While if the read consistency is set to* ***QUORUM****, the coordinator will answer the read request with the most recent data (based on the timestamp) when 2 out of 3 replicas are responding. Meanwhile an asynchronous process will repair the stale replicas at latter time, if any.*

(a subset of all the replicas). A quorum is computed as: $(replication\ factor/2)+1$. Data accesses and updates are performed to all replicas in the quorum. Thus, using this level for both read and write operations guarantees that the intersection of replicas involved in both operations contains at least one replica with the last update. A partial quorum has a smaller subset of replicas, hence returning the most recent data when read is issued, is not guaranteed.

Consistency levels in Cassandra. In the Cassandra storage system, several consistency levels [23] are proposed per-operation. A write of consistency level *one* implies that data have to be written to the commit log and memory table of at least one replica before returning a success. Moreover, as shown in Figure 1, a read operation with consistency level *ALL* (strong consistency) implies that the read operation must wait for all the replicas to reply with consistent data in order to return the data to the client. However, this will introduce higher latency if some replicas are inconsistent with the most current version. In contrast, a read consistency level of quorum, 2 of the 3 replicas are contacted to fulfill the read request and the replica with the most recent version would return the requested data. In the background, a read repair will be issued to the third replica and will check for consistency with the first two. If inconsistency occurs, an asynchronous process will be launched to repair the stale nodes at a latter time.

Many cloud storage systems such as Dynamo [2], Cassandra [3], Voledemort [24], and Riak [25] adopt asynchronous quorum replication [26][27]. This gives the application writer more flexibility when selecting the type of consistency that is appropriate for each operation. This is a useful feature, but until now no automatic adaptive model has been proposed for these systems. This means that the application writer has to choose the type of consistency for every operation, which is hard when no information is available regarding the read and write frequencies, network latency, and the system state in general, or when operating on a very large scale. We present Harmony, an approach which aims to make this task automatic for the operations that are not critical or do not need a strictly strong consistency. This is achieved using just a small hint about the application needs.

III. Elastic Adaptive Consistency Model

The goal of Harmony is to dynamically and elastically handle consistency at run time, in order to provide adequate tradeoffs between consistency and both performance and availability. Accordingly, Harmony considers not only the application requirements but also the storage system state. Moreover, rather than relying on a standard model based only on the access pattern to define the consistency requirement of an application – which is the case for most existing work – Harmony, in addition, uses the *stale read rate* of the application to precisely define such requirement.

Why use the stale reads rate to define the consistency requirements of an application? For example, we consider two applications that may have at some point the same access pattern. One is a web-shop application that can have heavy reads and writes during the busy holiday periods, and a social network application that can also have heavy access during important events or in the evening of a working day. These two applications may have the same behavior at some point and are the same from the point of view of the system when monitoring data accesses and network state as well, thus they may be given the same consistency level. However, the cost for stale reads is not the same for both applications. A social network application can tolerate a higher number of stale reads than a web-shop application: a stale read has no effects on the former, whereas it could result in anomalies for the latter. Consequently, defining the consistency level in accordance to the stale reads rate can precisely reflect the application requirement.

We propose our model for distributed storage systems. In this context, data may be replicated over geographically distant data centers. In order to predict the effect of weaker consistencies, we compute θ_{stale}, the estimation of the stale read rate in the system. The consistency requirement of an application should be determined by providing the rate of reads that should be fresh; in other words, the rate of stale reads that is tolerated by the application. Let this be app_stale_rate. For critical applications that require strong consistency, this rate should be 0%. Similarly, an application that does not need any consistency at all, such as an application that consists of only reads from archives, the rate should be 100% (which corresponds to static eventual consistency). A naïve way to map the consistency requirements of an application to the app_stale_rate is the following: for an application that needs an average consistency, the rate should be 50%. An application that needs less than average consistency should have a rate of 25%, and an application that requires higher consistency should use 75%. As part of future work, we plan to propose mechanisms to help an application administrator to determine such a rate in a more precise way. This rate is tunable and can be defined by studying the behavior of an application.

Additionally, in the case of distributed data replication, network latency may be high and thus, a performance-defining factor. Other than app_stale_rate, in our model, we consider the network latency and the application access pattern. We permanently, collect such information in order to estimate the stale read rate. From a higher level perspective, our solution uses the following decision scheme:

if $app\_stale\_rate \geq \theta_{stale}$ **then**
Choose eventual consistency (Consistency Level = One)
else
- ComputeX_n the number of always consistent replicas necessary to have $app\_stale\_rate \geq \theta_{stale}$
- Choose consistency level based on X_n

end if

The default consistency level is the basic eventual consistency that allows reading from only one replica. When such a level may not satisfy the consistency requirements of an application due to the growing number of stale reads, the number of replicas X_n that should be involved in the reading requests is computed. All the following read requests will be performed with Consistency level X_n. In the next section we explain in detail how we estimate the stale reads rate and how we compute the necessary number of replicas.

IV. Probabilistic stale read rate estimation

In this section, we propose an estimation of the stale read rate in the system by means of probabilistic computations. This estimation model requires basic knowledge of the application access pattern and of the storage system network latency. Network latency in this case is of high importance, since it is the determinant of the updates propagation time to other replicas. The access pattern, which includes read rates and write rates is a key factor to determine consistency requirements in the storage system. For instance, it is obvious that a heavy read-write access pattern would produce higher stale reads when adopting eventual consistency.

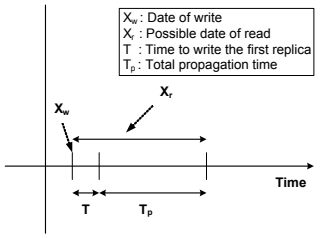


Figure 2. Situation that leads to a stale read

1) Stale read probability: We define the situation that leads to a stale read in Figure 2. The read may be stale if its starting time X_w is in the time interval between the starting time of the last write and the end of the propagation time of data to the other replicas. This situation is repeatable for any of the writes that may occur in the system. T_p in Figure 2 is the time necessary for the propagation of a write or an update to all the replicas. It is computed based on the network latency L_n and the average write size avg_w and should be represented as $T_p(L_n, avg_w)$, but in order to simplify the representation, it will be denoted as T_p in the rest of the paper.

Transactions arrivals are generally considered as a Poisson process as it is the common way to model them in literature [8][28]. We assume that the writes and the reads arrivals follow the Poisson distribution of parameter λ_w^{-1} (we chose λ_w^{-1} instead of λ_w in order to simplify subsequent formulas where the parameter will be inverted) and λ_r respectively. These parameters values change dynamically at run time following the read and write requests arrivals monitored in the storage system. Since the distribution of waiting time between two Poisson arrivals is an exponential process, the stochastic variables X_w and X_r of a write time and read time follow an exponential distribution of parameters λ_w^{-1} and λ_r respectively. The probability of the next read being stale corresponding to the aforementioned situation is given by formula (1) with N being the replication factor in the system and X being the number of replicas involved in the read operation. Here X_n =1 for the basic eventual consistency.

$$Pr(stale\_read) = \sum_{i=0}^{\infty}\Big(\frac{N-(X_n=1)}{N} Pr(X_w^i < X_r < X_w^i + T + T_p) + \frac{X_n=1}{N} Pr(X_w^i < X_r < X_w^i + T)\Big) \quad (1)$$

Having all the writes times (that may occur in the system) following the exponential distribution, the sum of X_w^i all the writes follows a $Gamma$ distribution of parameters i and

λ_w. Hence, the probability in formula (1) becomes:

$$Pr(stale\_read) = \sum_{i=0}^{\infty}(\frac{N-1}{N}\int_0^{\infty} f_w^i(t)(F_r(t+T+T_p) - F_r(t))dt + \frac{1}{N}\int_0^{\infty} f_w^i(t)(F_r(t+T) - F_r(t))dt) \quad (2)$$

The time T to write in the local memory is negligible in comparison to T_P and therefore, we can consider it as equal to 0. A simple replacement of the probability mass function of Poisson distribution and the cumulative distribution function of $Gamma$ distribution results in the following probability:

$$Pr(stale\_read) = \sum_{i=0}^{\infty}\frac{N-1}{N} \int_0^{\infty} t^{i-1}\frac{e^{-\frac{t}{\lambda_w}}}{\gamma(i)\lambda_w^i}(e^{-\lambda_r t} - e^{-\lambda_r(t+T_p)})dt \quad (3)$$

After simplifying formula (3), it becomes:

$$Pr(stale\_read) = \sum_{i=0}^{\infty}\frac{(N-1)(1-e^{-\lambda_r T_P})}{N(1+\lambda_r\lambda_w)^i} \int_0^{\infty} t^{i-1}\frac{e^{\frac{1+\lambda_r\lambda_w}{\lambda_w}t}}{\gamma(i)(\frac{\lambda_w}{1+\lambda_r\lambda_w})^i}dt \quad (4)$$

The right part of the function in (4) is the the *cumulative distribution function* of a $Gamma$ law of parameters $\frac{1+\lambda_r\lambda_w}{\lambda_w}$ and i, its value is equal to 1. Moreover, if we consider that:

$$\sum_{i=0}^{\infty}(\frac{1}{1+\lambda_r\lambda_w})^i = \frac{1}{\lambda_r\lambda_w} + 1 \quad (5)$$

The final value of the probability of next read to be stale, after simplification, is given by:

$$Pr(stale\_read) = \frac{(N-1)(1-e^{-\lambda_r T_p})(1+\lambda_r\lambda_w)}{N\lambda_r\lambda_w} \quad (6)$$

2) Computation of X_n, the number of replicas: To compute the number of replicas to be involved in a read operation necessary to maintain the desired consistency, we compute X_n in formula (1) to maintain the inequality (7) in order to provide a stale read rate smaller or equal to the app_stale_rate denoted as ASR for simplicity.

$$Pr(stale\_read) = \sum_{i=0}^{\infty}(\frac{N-X}{N}\int_0^{\infty} f_w^i(t)(F_r(t+T+T_p) - F_r(t))dt + \frac{X}{N}\int_0^{\infty} f_w^i(t)(F_r(t+T) - F_r(t))dt) \leq ASR \quad (7)$$

After simplification, and following similar steps for computing in formulas (3), (4), and (6), the number of the replicas X_n is given by the formula:

$$X_n \geq \frac{N((1-e^{-\lambda_r T_P})(1+\lambda_r\lambda_w) - ASR\lambda_r\lambda_w)}{(1-e^{-\lambda_r T_P})(1+\lambda_r\lambda_w)} \quad (8)$$

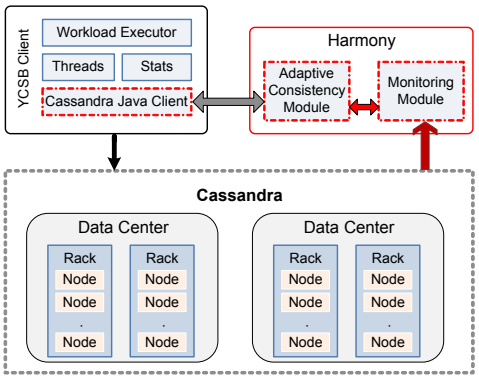


Figure 3. Harmony implementation and integration with Cassandra and Yahoo! Cloud Serving Benchmark: *While the monitoring module collects relevant information from the Cassandra storage cluster using Cassandra Nodetool and ping tool in multithreaded manner, the adaptive-consistency module estimates the read stale rate using the collected information as an input and then sends the most appropriate consistency level to the client "the modified Java client for Cassandra in YCSB"*

V. Evaluations

A. Harmony Implementation

Harmony can be applied to different cloud storage systems that featured with flexible consistency rules. Currently we have built Harmony in Apache Cassandra "Cassandra-1.0.2" [29]. As described in Section II, Cassandra gives the user flexible usage of consistency levels in a per-operation manner. In addition, Cassandra is proven to be very scalable, offering very good performances, and being widely used with large scale applications such as Facebook and Twitter. Figure 3 gives an overview of the Harmony implementation. Harmony is introduced as an extra layer on Cassandra that aims to provide the most appropriate level of consistency for reading data. The core of this layer consists of two modules. Both modules were implemented in Python 2.7.

The monitoring module collects relevant metrics needed for Harmony. The Cassandra Nodetool was used to collect the number of reads and writes in Cassandra storage, and the Ping tool was used to collect network latencies in the storage system network. The monitoring module was designed in a multithreaded manner in order to make it time-efficient and to reduce the monitoring time. Each thread collects data from a set of nodes and at the end an aggregation process is applied. The monitoring time is measured and taken into account when computing the read rates and write rates. This data is further communicated to the **adaptive consistency module**. This module is the heart of Harmony implementation. An estimation of the stale read rate is computed and then compared to the application stale read that can be tolerated (app_stale_rate) in order to provide an adequate consistency level for the running application at that point of time.

B. Evaluation Methodology

We adopt two complementary approaches to provide storage as a service for cloud clients. In our first approach,

cloud clients which can be applications running on cloud computing service such as Amazon EC2 [10] or Google App Engine [30], can connect to the storage service on an S3-like interface to lease their storage resources. This is an interface to a highly distributed scalable storage backend that will physically host and manage data. We set up a cloud storage testbed on the Grid'5000 experimental grid and cloud testbed [9] that federates 10 sites in France. In our second approach the storage service is provided within the virtual disks attached to the virtual machines (VMs) side by side with cloud clients. We set up the underlying storage system on Amazon EC2 clusters and serve applications running inside VMs.

Micro Benchmark. We aim at a micro benchmark representing typical workloads in current services hosted in clouds. Based on case studies [5][11], we have selected the Yahoo! Cloud Serving Benchmark (YCSB) framework [31]. YCSB is used to benchmark Yahoo!'s cloud storage system "PNUTS" [5]. It is extended to be used with a variety of *open-source* data stores such as mongoDB [32], Hadoop HBase [6] and Cassandra [3]. YCSB provides the characteristics of a real cloud serving environment such as scale-out, elasticity and high availability. For this purpose, several workloads have already been proposed in order to apply a heavy read load, heavy update load, and read latest load, among other workloads. Also, the benchmark is designed to make the integration of new workloads very easy. We use YCSB-0.1.3, as shown in Figure 3. We modify the provided Java client for Cassandra in order to allow read operations to be performed with different consistency levels at run time. The Java client uses Thrift [33] to communicate with the cluster and is provided with the set of hosts from which it should request a read or a write. The modified Java client reads data from Cassandra with the consistency level provided dynamically by the adaptive consistency module.

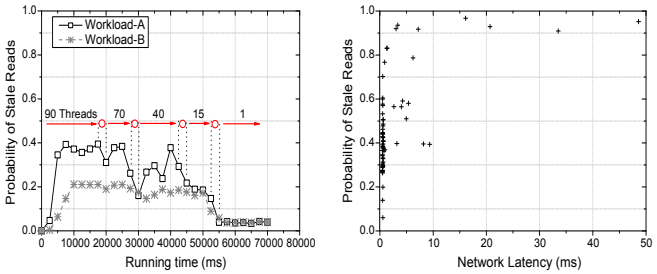
C. Experimental Setup

We evaluate Harmony with Cassandra deployed on both Grid'5000 and Amazon EC2.

Setup on Grid'5000. We use two clusters in the *Sophia* site with a total of 84 nodes and 496 cores. All nodes are equipped with x86_64 CPUs and 4GB of memory. The nodes are interconnected with Gigabit Ethernet. All nodes from the first cluster have two hard disks with combined capacity of 600GB per node. As for the second cluster, the nodes are all equipped with hard disks of 250GB.

Setup on Amazon EC2. We use 20 Virtual machines of type Large located in the us-east-1a availability zone in the east cost. Each virtual machine has 2 cores and 7.5GB of memory. The total size of disk storage available is 14.78TB.

In both experiments, Cassandra is configured in order to have a replication factor of 5. Moreover, "OldNetworkTopologyStrategy" was chosen as a replication strategy. This strategy ensures that data is replicated over all the clusters and racks. We deployed Cassandra on the two clusters on the Grid'5000 and on the 20 nodes of Amazon EC2. We use YCSB with workload-A which provides a heavy read-update data access. For the experiments that were conducted on Grid'5000, we initially inserted a load of 3 million rows and a total size of 14.3GB after replication. Each workload run had a total of 3 million operations consisting of reads and updates. While for the experiments that were conducted on Amazon EC2, an initial load of 5 million rows, with a total size of 23.85GB after replication, was inserted. Each workload that was run consisted of 10 million operations.



(a) Workloads and No. of client threads impacts

(b) Network Latency impacts

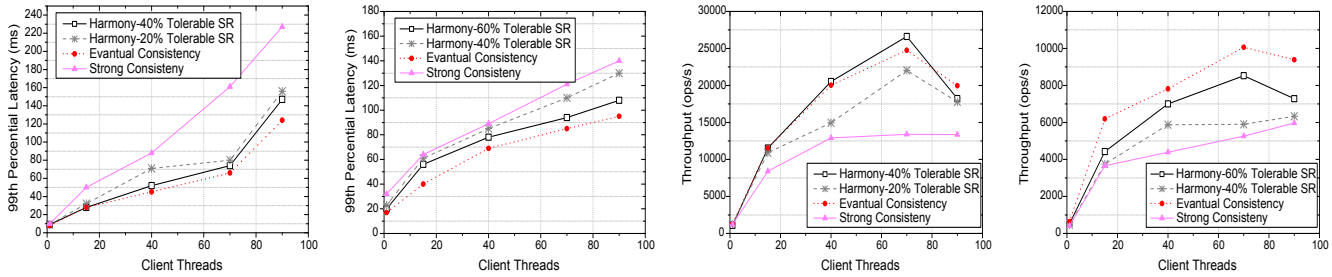
Figure 4. Stale read rate estimation in Harmony: (a) We show the impacts of both the workloads' access pattern and the number of client threads on the stale read estimation used in Harmony: *We used Grid'5000 as we can guarantee the network latency.* (b) We show the impact of network latency on the stale read estimation: *the variability on EC2 network latency.*

D. Stale Reads Estimation in Harmony

We first study the impacts of the workload access patterns, the number of clients, and network latency on the stale reads estimation. Accordingly, we used two workloads: workload-A, which has a heavy read-update access pattern, and workload-B, which has a heavy read access pattern with a small portion of writes representing approximately 5% of the total number of operations. We ran both workloads, varying the number of threads starting with 90 threads, then, 70, 40, 15 and finally, one thread.

The impacts of workloads' access patten and client number. As shown in Figure 4(a), the probability of reading stale data for workload-B is relatively smaller than the one for workload-A. This is because the number of updates is smaller. We observe that the number of updates plays very important role in causing stale reads even with a high number of reads. Moreover, we observe that the probability of reading stale data varies according to the number of threads. We can see that for workload-A, the probability of stale reads gradually decreases with the number of threads, because increasing the thread number increases the throughput and thus increases the reads and writes rate. Also, we notice the probability reduction gap is big during the transaction (changing the number of threads).

The impacts of network latency. In order to see the impact of network latency on the stale reads estimation we ran workload-A –varying the number of threads starting with



(a) Read operation latency on Grid'5000 (b) Read operation latency on Amazon EC2 (c) Throughput on Grid'5000 (d) Throughput on Amazon EC2

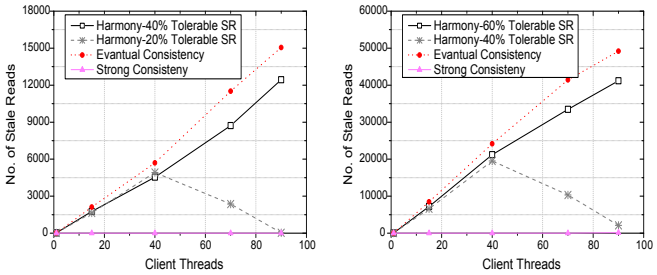
Figure 5. Latency and throughput in Harmony with S% tolerable stale reads (Harmony-S% Tolerable SR) against strong and eventual consistency on our both testbeds: *Grid'5000 and Amazon EC2*

90 threads, then, 70, 40, 15 and finally, one thread– on Amazon EC2 and measure the network latency during the run-time. Figure 4(b) presents the results. We can see that high network latency causes higher stale reads regardless of the number of the threads (higher latency dominates the probability of stale reads), while when the latency is small the probability will be varied according to the reads and writes rates (with smaller impacts of the network latency).

E. Latency and throughput of the applications in harmony

As mentioned earlier, in Harmony, the application requirements are defined as the stale reads rate that an application can tolerate during its running. Accordingly, we compare Harmony with two settings (two different tolerable stale read rates) with strong and eventual consistency on our both storage approaches (Grid'5000 and Amazon EC2). The first tolerable stale read rates are 40% for Grid'5000 and 60% for Amazon EC2 (these rates tolerate more staleness in the system implying lower consistency levels and thus less waiting time), and the second tolerable stale read rates are 20% for Grid'5000 and 40% for Amazon EC2 (these rates are more restrictive than the first ones, meaning that the number of read operations performed with a higher level of consistency is larger). Network latency is higher in Amazon EC2 than in Grid'5000 (5 times higher in the normal case), thus we choose higher stale read rate for the same workload with Amazon EC2. We run workload-A while varying the number of client threads.

Figures 5(a) and 5(b) presents the 99th percentile latency of read operations when the number of client threads increases on Grid'5000 and EC2 respectively. While the strong consistency approach provides the highest latency having all the reads to wait for the replies from all the replicas spread over different racks, the eventual consistency approach is the one that provides less latency because all the read operations are performed on one close replica but at the cost of consistency violation. We can clearly see that Harmony with both settings provides almost the same latency as the eventual consistency. Moreover, the latency increases by decreasing the tolerable stale reads rate of an application as the probability of stale read can easily get



(a) Staleness on Grid'5000 (b) Staleness on Amazon EC2

Figure 6. Staleness in Harmony with S% tolerable stale reads (Harmony-S% Tolerable SR) against strong and eventual consistency.

higher than these rates, which requires a higher consistency levels and, as a result, a higher latency.

In Figures 5(c) and 5(d), we show the overall throughput for read and write operations with different numbers of client threads. The throughput increases as the number of threads increases. However, the throughput decreases with more than 90 threads. This is because the number of client threads is higher than the number of storage hosts and threads are served concurrently. We can observe that the throughput is smaller with strong consistency. The fact that read operations with higher consistency levels have high latencies, makes the number of possible operations per second smaller. We can notice that our approach with a stale reads rate of 40% and 60% for Grid'5000 and Amazon EC2 respectively, provide very good throughput that can be compared to the one of static eventual consistency approach. But, while exhibiting very good throughputs, our adaptive policies provide a better consistency and fewer stale reads due to the fact that higher consistency levels are chosen only when it matters.

F. Actual Staleness in Harmony

In Figures 6(a) and 6(b) , we show that Harmony, with all the policies with different application tolerated stale reads rates, provides less stale reads than the eventual consistency approach. Moreover, we can see that, with a more restrictive tolerated stale reads rate, we get a smaller number of stale reads. We observe that with rates of 20% and 40% for Grid'5000 and Amazon EC2 respectively, the number of stale reads decreases when the number of threads grows over

40 threads. This is explained by the fact that with more than 40 threads the estimated rate of stale reads gets higher than 20% and 40% respectively, for most of the run time, and higher consistency levels are chosen, thus decreasing the number of stale reads. It needs to be pointed out that this number of stale reads is not the actual number of stale reads in the system in the normal run, but it is representative.

In fact, to measure the number of stale reads, we perform two read operations for every read operation in the workload. The first read is performed with the relevant consistency level chosen by our approach, and the second read is performed with the strongest consistency level. Then, we compare the returned timestamps from both reads, and if they do not match, it means that the read is stale. Although this helps to estimate the number of stale reads, it completely changes the latency of reads and the throughput in the system. Moreover, it directly affects the monitoring data about system state. Additionally, the second read with strong consistency level provides more time for the next write to be propagated to the other replicas and, thus more chances for the next read to be fresh.

VI. Related work

Eventual consistency was employed in cloud storage system as an alternative to traditional strong consistency to achieve scalable and high-performance services [20]. Many commercial cloud storage systems have already adopted the eventual consistency approach such as Dynamo [2] in Amazon S3 [22], Cassandra [3] in Facebook [7] and PNUTS [5] in Yahoo!. A fair number of studies have been dedicated to measuring the actual provided consistency in cloud storage platforms [8][34][35]. Wada *et al.* [8] investigate the consistency properties provided by commercial storage systems and report on how and under what circumstances consumer may encounter stale data. Also, they explore the performance gain of using weaker consistency constraints. Anderson *et al.* [34] propose an offline consistency verification algorithm and test three kind of consistency semantics on registers including safety, regularity, and atomicity in the Pahoehoe key-value store using a benchmark similar to YCSB [11]. They observed that consistency violations increase with the contention of accesses to the same key.

Moreover, in order to meet the consistency requirements of applications and reduce the consistency violation, some studies are done on adaptive consistency tuning in cloud storage systems [36] [37] [38]. Kraska *et al.* [36] propose a flexible consistency management that is able to adapt the resulting consistency level to the requirements stated by applications. The inconsistencies considered in their work are due to update conflicts. Accordingly, they build a theoretical model to compute the probability of update conflict, and then compare it to a threshold. As a result, they choose either serializability using strong consistency or session consistency, which is a weaker consistency. However, besides that their approach cannot be applied with eventual consistency as weaker consistency because: in eventual consistency, the staleness is due to the update propagation latency rather than just the conflict of two or more updates on different replicas. The threshold –used to determine the type of consistency– is computed based on the financial cost of pending update queues and not related to the storage backend itself. Wang *et al.* [38] propose an application-based adaptive mechanism of replica consistency. This mechanism was proposed with a specific replication architecture. The architecture relies on multi-primary replicas and secondary replicas where the latter are read-only replicas. Consistency is either strong or eventual and the choice between the two is made by comparing the read rate and the write rate to a threshold. The main limitation of this work is the arbitrary choice of a static threshold. In addition, this approach was proposed for their specific proposed replication architecture, which is not commonly used in current cloud storage solutions.

In contrast to the aforementioned work, Harmony is using the stale reads rate to define the consistency requirements of the application. Moreover, it dynamically alters the replicas number involved in an operation according to the estimated stale reads rate and the network latency, during run-time. Thus Harmony achieves adequate tradeoffs between consistency and both performance and availability.

VII. Conclusion

With the explosion of cloud storage businesses and the increasing number of web services migrating to the cloud, a strong consistency model becomes very costly when scalability and availability are required. Thus, weaker consistency models have been proposed, but these models may lead to far too much inconsistency in the system. In this paper, we present Harmony, a novel approach that handles data consistency in cloud storage adaptively by choosing the most appropriate consistency level dynamically at run time. In Harmony, we collect relevant information about the storage system in order to estimate the stale read rate when consistency is eventual, and make a decision accordingly. In order to be application-adaptive, Harmony takes into account the application's needs expressed by the stale read rate that can be tolerated. We show that our approach provides better performance than traditional approaches that are based on strong consistency. Moreover, it provides more adequate consistency than eventual consistency approaches. In addition, our solution is designed to be completely tunable to provide the system or the application administrator with the possibility of controlling the degree of compromise between performance and consistency.

For future work, we plan to provide a mechanism allowing the system to automatically divide data into different consistency categories without any human interaction by applying clustering techniques. Every category should be given the most appropriate consistency level in regard to

the data it encloses. Another enhancement is to propose a mechanism that models the application and computes the stale read rate that can be tolerated automatically.

ACKNOWLEDGMENTS

This work has been supported by the EU FP7 MCITN SCALUS project under grant agreement no. 238808 and by the HEMERA Large Wingspan Action led by INRIA. The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed. We would like to thank Louis-Claude Canon and Jésus Montes for helpful discussions and precious suggestions.

REFERENCES

[1] R. Peglar, "Eliminating planned downtime: the real impact and how to avoid it," May 2012. [Online]. Available: http://findarticles.com/p/articles/mi_m0BRZ/is_5_24/ai_n6095515/

[2] 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 *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220.

[3] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, April 2010.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th conference on usenix symposium on operating systems design and implementation*, 2006, pp. 205–218.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. arno Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!s hosted data serving platform," In Proc. 34th VLDB, Tech. Rep., 2008.

[6] "Apache HBase," February 2012. [Online]. Available: http://hadoop.apache.org/hbase/

[7] "Facebook Statistcs," January 2012. [Online]. Available: http://newsroom.fb.com/content/default.aspx?NewsAreaId=22

[8] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective," in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, 2011, pp. 134–143.

[9] Y. Jégou, S. Lantéri, J. Leduc *et al.*, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed." *Intl. Journal of High Performance Comp. Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[10] "Amazon Elastic Compute Cloud (Amazon EC2)," February 2012. [Online]. Available: http://aws.amazon.com/ec2/

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[12] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '00. New York, NY, USA: ACM, 2000, pp. 7–.

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

[14] B. Calder, J. Wang, A. Ogus, *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 143–157.

[15] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in scatter," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 15–28.

[16] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, p. 1825, 2001.

[17] J. Rao, E. J. Shekita, and S. Tata, "Using paxos to build a scalable, consistent, and highly available datastore," *Proc. VLDB Endow.*, vol. 4, no. 4, pp. 243–254, Jan. 2011.

[18] E. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23 –29, feb. 2012.

[19] D. J. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, pp. 37–42, 2012.

[20] W. Vogels, "Eventually consistent," *Commun. ACM*, pp. 40–44, 2009.

[21] D. Pritchett, "Base: An acid alternative," *ACM Queue*, vol. 6, pp. 48–55, May 2008.

[22] "Amazon Simple Storage Service (Amazon S3)," February 2012. [Online]. Available: http://aws.amazon.com/s3/

[23] "About Data Consistency in Cassandra," February 2012. [Online]. Available: http://www.datastax.com/docs/1.0/dml/data_consistency

[24] "Project Voldemort," May 2012. [Online]. Available: http://project-voldemort.com/

[25] "Riak: An Open Source Scalable Data Store," May 2012. [Online]. Available: http://wiki.basho.com/Riak.html

[26] M. Herlihy, "A quorum-consensus replication method for abstract data types," *ACM Trans. Comput. Syst.*, vol. 4, no. 1, pp. 32–53, Feb. 1986.

[27] D. K. Gifford, "Weighted voting for replicated data," in *Proceedings of the seventh ACM symposium on Operating systems principles*, ser. SOSP '79. New York, NY, USA: ACM, 1979, pp. 150–162.

[28] A. T. Tai and J. F. Meyer, "Performability management in distributed database systems: An adaptive concurrency control protocol," in *Proceedings of the 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, ser. MASCOTS '96. Washington, DC, USA: IEEE Computer Society, 1996.

[29] "Apache Cassandra," February 2012. [Online]. Available: http://cassandra.apache.org/

[30] "Google App Engine," February 2012. [Online]. Available: http://code.google.com/appengine/

[31] "Yahoo Cloud Serving Benchmark," February 2012. [Online]. Available: https://github.com/brianfrankcooper/YCSB/wiki

[32] "mongoDB," February 2012. [Online]. Available: http://www.mongodb.org/

[33] "Apache Thrift," February 2012. [Online]. Available: http://thrift.apache.org/

[34] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie, "What consistency does your key-value store actually provide?" in *Proceedings of the Sixth international conference on Hot topics in system dependability*, ser. HotDep'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16.

[35] D. Bermbach and S. Tai, "Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior," in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, ser. MW4SOC '11. New York, NY, USA: ACM, 2011, pp. 1:1–1:6.

[36] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: pay only when it matters," *Proc. VLDB Endow.*, vol. 2, pp. 253–264, August 2009.

[37] S. Sakr, L. Zhao, H. Wada, and A. Liu, "Clouddb autoadmin: Towards a truly elastic cloud-based data store," in *Proceedings of the 2011 IEEE International Conference on Web Services*, ser. ICWS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 732–733.

[38] X. Wang, S. Yang, S. Wang, X. Niu, and J. Xu, "An application-based adaptive replica consistency for cloud storage," in *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, nov. 2010, pp. 13 –17.