A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores

David Bermbach, Jörn Kuhlenkamp, Bugra Derre, Markus Klems, Stefan Tai
Institute of Applied Informatics (AIFB)
Karlsruhe Institute of Technology
Karlsruhe, Germany
firstname.lastname@kit.edu

***Abstract*—Applications often have consistency requirements beyond those guaranteed by the underlying eventually consistent storage system. In this work, we present an approach that guarantees monotonic read consistency and read your writes consistency by running a special middleware component on the same server as the application. We evaluate our approach using both simulation and real world experiments on Cloud storage systems.**

***Keywords*-service quality; consistency; middleware**

I. Introduction

Over the last few years, the success of the internet, web 2.0 and the resulting need for scalable software systems has triggered the development of an entirely new class of storage systems which are usually referred to as NoSQL systems [1]–[6] since their query interface is *N*ot *o*nly *SQL*. As mandated by Brewer's CAP theorem [7] and the PACELC model [8], they relax consistency guarantees in favor of latency and availability. Since cloud storage systems have the same quality of service (QoS) requirements, namely scalability, low latency, high availability, most of these systems typically also offer only weak consistency guarantees – a fact which can also be shown experimentally [9], [10].

Developing applications on top of only eventually consistent data stores is non-trivial as it is not possible to rely on correctness of data read from a data store: Results may be stale or replicas may have reordered requests before executing them, repeated reads may not yield the same results etc. While staleness is often not as critical, ordering is – so-called client-centric consistency guarantees (e.g., being able to read one's own writes) ease application development a lot.

These guarantees are hard to provide by a storage system as they can often only be realized by strictly consistent storage systems (usually not an option) or session stickiness. The latter can become an issue in the presence of failures or extensive load variability. Another alternative is shipping huge dependency trees with each update and executing only when all dependencies are fulfilled [11]. Of course, this is rather complex to implement (and, hence, prone to programming errors) and introduces an overhead in terms of data sent around. Also, as updates are potentially delayed often, staleness increases.

All in all, client-centric consistency guarantees are hard to provide by the storage system so that it is usually not feasible to do so. To our knowledge, no cloud storage service (i.e., a hosted service) exists which guarantees any kind of client-centric concistency.

We propose to implement these guarantees in a middleware layer running on the same server-side machine as an application. This application can then easily extend those guarantees to its individual end users via standard session handling mechanisms as state can then be handled entirely locally on the application's server. We argue that this approach introduces only a small overhead.

Another aspect is concurrent updates: Existing cloud data stores use a last write wins strategy, i.e., in the presence of concurrent updates writes are often lost, which is hard to handle in applications without the end user noticing. Imagine a web-based collaborative document editing application like Google Docs[1]: Two users edit a document at the same time and send their changes to the application which then saves them to the storage system. One of those edits will be overwritten by the other, even though both are valid. A better way would be to merge both versions upon a later read – but each user should be able to always see his or her own changes, no matter what the internal state of the storage system or even the application is.

So, the problem an application developer faces is, that distributed storage systems (and cloud storage services in particular) offer only weak or no consistency guarantees at all. At the same time, applications have certain consistency requirements and cannot get any guarantees from the cloud storage system which is typically accessed as a black box. Even if the storage system in question is not a cloud storage service but rather a self-hosted NoSQL system, application developers will rarely have time and insight to change the behavior of the storage system. NoSQL systems like Apache Cassandra [5] allow to specify different consistency levels per request but even a consistency level of ALL for all

[1] docs.google.com

requests will not necessarily always guarantee client-centric consistency, especially in the presence of failures.

In this work, we present an approach that guarantees monotonic read consistency and read your writes consistency by running a special middleware service on the same server as the application. It can be run with any kind of eventually consistent data store and shows to the client the same behavior as a causally consistent data store, i.e., it creates the client-side illusion of a causally consistent data store. Our approach is intended for a scenario, where a set of servers running both our middleware service and the application code uses an eventually consistent storage system accessed as a black box. End users communicate with the application using thin clients.

We start by describing several consistency models and their implications to applications in section II, before presenting our approach and implementation in section III. We also provide a formal proof regarding our claim of giving the illusion of causal consistency. Next, we evaluate our approach using both simulation and real world experiments with a sample application in section IV, before discussing related work and finally coming to a conclusion in section VI.

II. Background

In this section, we will discuss existing consistency models. These models always comprise two roles: a storage system and a set of clients interacting with the storage system. A client does not need to be an end user but rather refers to any machine that issues requests to the storage system. For our approach, a client, hence, is an application server. In this section, we will use the generic term *client* as the models presented do not depend on the kind of the client (application server, end user etc.). The only requirement for the presented models is that a client directly interacts with the storage system.

There are two main perspectives on consistency: data-centric and client-centric consistency [12]. While data-centric consistency guarantees focus on the internal state of a storage system, i.e., all replicas being identical, client-centric consistency guarantees focus on the consistency properties visible to a client having only black box access to a storage system. Therefor, data-centric consistency guarantees are important to system developers while to application developers only client-centric guarantees matter. Often, a system may appear consistent to a client at a point in time where data-centric consistency has not been reached, yet. An example would be, having one stale replica and accessing more than one replica for every read. Combined with means to identify the ordering of different versions (e.g., vector clocks) the client will never observe the data-centric inconsistency.

Please, note, that eventually consistent storage systems do not use locking mechanisms and, therefore, allow concurrent updates.

Apart from the two perspectives, there are also two dimensions of consistency: staleness and ordering. Staleness is a consistency property that can be observed both as a data-centric or client-centric guarantee. The client-centric staleness values are less than or equal to the data-centric staleness values [9]. But as long as staleness values are small they often do not matter to applications (there are exceptions, though). For example, if refreshing your email inbox shows you a new email you will never notice that your previous refresh simply read a stale replica.

An entirely different story is the ordering dimension of consistency [13]. To stick with the example of the email inbox: Imagine a situation where refreshing your inbox hides some unread emails that you were able to see before. This is a violation of monotonic read consistency – a guarantee which only addresses ordering of requests but not staleness. There are four main ordering guarantees which can be visible to a client (hence, client-centric consistency guarantees) [14]:

- *Monotonic read consistency* (MRC): After reading version n, the same client will never again read a version $< n$.
- *Read your writes consistency* (RYWC): After writing version n, the same client will never again read a version $< n$.
- *Monotonic writes consistency* (MWC): All writes by the same client will be serialized in their chronologic order, i.e., if there are two consecutive writes by the same client and a replica has already written the value of the second write, then this value will not be overwritten once the first write arrives at the replica.
- *Write follows read consistency* (WFRC): When a client has read version n and updates the value he has read (thus, essentially writing version $n + 1$), the update is guaranteed to execute only on replicas with a version number of at least n.

In general, eventually consistent storage systems (i.e., storage systems that do not guarantee strict consistency) can have inconsistencies (a) due to failures, (b) due to update propagation delays and message reordering and (c) due to concurrent updates.

It is usually not possible to resolve conflicts resulting from concurrent updates in the storage system as this requires application-specific knowledge on the semantics of the data items. For this purpose, we propose to relax the definitions from above slightly: A client still observes MRC or RYWC if the read result set of returned data items contains at least one version that fulfils the required guarantees. This means, for example, that for MRC (after a read of version n), it is acceptable to return a version $\geq n$ plus any conflicting version branches.

Brzezinski et al. [15] show that MRC, RYWC, MWC and WFRC combined can only be guaranteed by a storage system guaranteeing causal consistency (CC) which is one of

the strongest data-centric consistency guarantees apart from strict consistency and sequential consistency. CC requires that all requests that are causally related are executed on all replicas in the same order. Two requests o and p are causally related if (a) both are issued by the same client, (b) if o is a write and p returns the result of o or (c) if there exists an operation x so that p depends on x and x depends on o (transitivity). If either of those conditions hold, o will be executed before p on all replicas.

III. Approach

In the following, we will describe how consistency guarantees of an eventually consistent storage system can be increased using an external middleware service running on the same machine as the application.

Our approach includes several roles apart from the storage system (see figure 1 for a high level overview of the scenario we want to address):

1) *App server*: An application with consistency requirements beyond those guaranteed by the storage system is running on one or more machines. A single server running application code is called an app server. App servers do not need to interact directly but may of course do so.
2) *Library*: The middleware service which we propose is currently implemented as a library (hence, the name) which can be used directly within the application. Each app server uses its own library instance running on the same machine. Library instances do not need to communicate with other library instances. App server requests to the storage system are routed via the library.
3) *End user*: An end user usually uses a software client (e.g., a browser) to interact with one app server instance. This software client is running on a different machine than all app servers or the storage system. To avoid confusion, we will use the term *end user* for the software client and the underlying physical device the actual end user is using.

As an example setup: Amazon S3[2] could be used as the storage system. The app servers could be distributed over multiple compute clouds and on-premises servers running a standard web application (e.g., an internet forum). End users of said web application might access the web application via browsers on their desktop computers or mobile devices.

Vector clocks are a well known mechanism to capture causality. In the context of eventually consistent distributed storage systems, they can be used to describe a version history like, e.g., in Amazon Dynamo [3]. Whenever conflicting versions exist on some replica, it is then possible to resolve all conflicts automatically that were not caused by concurrent updates (e.g., conflicts caused by update propagation delays or fail-recover errors).

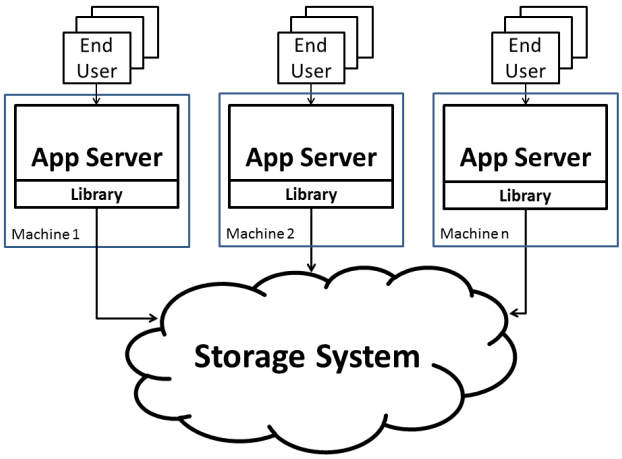

Figure 1. Intended Use Case

Our approach uses vector clocks to identify (a) which version was last *seen* by a particular app server and (b) which version was last *written* by a particular app server. This allows to create a total order for all writes by the same app server (a prerequisite for MWC and WFRC) and for writes that update another app server's write upon reading it (a prerequisite for CC). It also allows to check whether MRC and RYWC are violated or not. The data itself and the corresponding vector clock metadata are persisted under the same key.

Now, as some inconsistencies can be identified, the app server at least knows about them. This leaves the issue of dealing with them which we propose to do via client-side caching, i.e., on the app server.

- Whenever an app server requests a datum from the storage system, the library reads from the storage system and adds a copy of that datum to its local cache if the cache does not already contain that datum in that exact version (identified by its vector clock). Older versions superseded by the data read from the storage system are replaced in the cache and all conflicting versions are returned to the app server. The app server is then expected to merge conflicting versions (this could even be dropping all but one version) at application level considering the semantics of the data. This conflict resolution scheme is similar to the approach taken in Amazon Dynamo [3].
- Whenever an app server requests a write following a read, the library assigns a vector clock value that supersedes all cached values of that datum and writes it to the storage system. Afterwards, the library replaces all cached versions with the value and vector clock it just wrote to the storage system.

[2]aws.amazon.com/s3

- Whenever an app server spontaneously writes a value, the library assigns a vector clock that is in conflict with all cached values to guarantee that no versions are lost due to this concurrent update. It then adds the data to the local cache and writes it to the storage system.
- Storage systems guaranteeing CC, typically capture only internal causality but not external causality [12] where a client of the storage system gets to know a new version not via the storage system but rather using external means of communication with another client. To also capture external causality, the library offers a notification feature which allows the app server to update the cache directly. As long as the app server uses this feature whenever he gets knowledge of an update using external means of communication, the library guarantees to capture external causility as well.

Our approach guarantees MRC and RYWC and creates the client-side (i.e., at the app server) illusion of per key CC[3] as so-called session consistency, i.e., guarantees exist for the duration of a session but not beyond. A session is initiated by an app server and ends either if the app server terminates the session, if the app server fails or if the cache fails. Furthermore, while a library outside of the storage system cannot guarantee MWC[4] it certainly helps towards that goal by reissuing "all" writes in correct order with every update, thus, increasing the probability of correct serialization within the storage system. The same holds true for WFRC.

Commercial cloud storage offerings typically return only one version, using a last write wins strategy internally. This can result in potentially many lost updates based on the rate of concurrent updates. Our approach avoids this completely as long as there is for every update at least one session which holds that update within its local cache.

A. Overhead and Intended Use Case

Our approach adds some overheads: First, there is a storage overhead for persisting the vector clocks within the storage system (which will typically create additional cost for cloud storage systems). Second, there is a compute overhead involved for every read as at least two vector clocks need to be compared. Third, there is another (local) storage overhead for keeping a persistent local cache.

The first overhead for persisting the vector clocks is negligible for most scenarios. A vector clock containing 100 entries, for example, requires less than 500 Bytes in its current (not very efficient) implementation. The size could be further reduced using, e.g., compression.

The second overhead for comparing vector clocks directly depends on the number of entries in the respective vector clocks. Therefore, it is desirable to keep the number of app servers relatively small – since a typical app server should be able to handle hundreds or thousands of end users in parallel, the number of app servers is small compared to the number of end users served. With today's server's compute power, the second overhead should, hence, not become an issue. Furthermore, the size of the vector clock does not directly depend on the number of app servers involved but on the number of app servers issuing writes. Hence, read-heavy workloads should not create any problem at all while for write-heavy workloads there could be mechanisms like routing updates for a certain key always via the same app server to further reduce the number of vector clock entries.

In our intended use case, the third overhead (storage cost of the local cache) does not really matter: A set of independent app servers interacts with a storage system using our library. As app servers typically do not persist data locally (apart from log files), the local hard disk drive should be more or less unused and can, hence, be used for a local cache without or with little side effects. Furthermore, there is a relation between the update frequency and the size of a datum: Small files are updated frequently while very large files are mostly write-once data. Thus, the third overhead of keeping a persistent local cache should become negligible as well as only small files need to be cached.

Even though the decision might be different for different applications, we believe that typically the consistency benefits far outweigh the overheads incurred. Our approach is especially helpful, if there is a huge number of concurrent requests to only a few keys with small data items. The smaller the size and number of the data items, the more feasible is our approach. If there are only a few concurrent updates, the feasibility of using our approach depends on whether the data store guarantees MRC. To our knowledge, no cloud storage service or production-ready open source system exists that can give these guarantees. So, we believe it is safe to say that any scenario where data items are updated from time to time (instead of being written only once), can benefit from using our approach.

B. Handling Sessions

From a quality of service perspective, the consistency guarantees of the library should be extended to the end user as he is the one who will be irritated if unread emails vanish, he cannot read the blog comment he just posted and so on. At the same time, the local cache might grow over time as more and more data items from the storage system are accessed by the end users. Also, conflicting versions need additional space. To limit the size of the local cache, we propose the following approach to session handling:

1) Start a library session on the app server.
2) Accept end user requests and hold sessions with them using standard session management features of the app server.

[3]We cannot actually guarantee CC without having control over the storage system itself, but to the app server our library guarantees the same behavior as the storage system would do if it offered CC.

[4]It cannot affect the behavior of the storage system which may just decide to drop arbitrary updates.

3) When the local cache has grown too much, restart from step 1 and accept new end user sessions only within the scope of the new session.
4) Terminate the old library session as soon as all end user sessions have completed.

When the main goal is avoiding lost updates, then another strategy would be to drop items from the cache as soon as an update to the respective app server's write has been read by that same app server. This works because reading an update guarantees that another app server has included the own value in its cache. Of course, while this avoids lost updates and keeps the cache small, it does not address MRC or RYWC.

If there is only one app server (i.e., no concurrent updates), this app server can still benefit from the MRC and RYWC guarantees. In that particular case, the cache size can be limited if the maximum inconsistency window is known, e.g., via consistency benchmarking. Items can be dropped from the cache when the last update to them has been executed more than t units of time ago, where t is the duration of the inconsistency window plus a safety margin.

C. Consistency Guarantees

We show that the usage of the provided library guarantees the same client-centric consistency levels as a storage system that implements causal consistency. In the following, C_i stands for app server number i. This is without loss of generality as the same guarantees hold for any client of a storage system using our library.

Definition (Operations) 1: Let O_{Li} denote the set of operations of a library L_i. We refer to the operations themselves as:

- $w_i(x)v$ - request of C_i to perform a write operation under key x and value v.
- $r_i(x)v$ - request of C_i to perform a read operation under key x and value v.
- $o_i(x)v$ request of C_i to perform any operation under key x and value v.

Definition (Vector Clocks) 1: We refer to the vector clock of an operation $o_i(x)v$ as $vc(o_i(x)v) \in VC$ where VC denotes the set of vector clocks visible to a single library. The binary relation $<$ imposes a linear order on the set VC. We call $<$ the *data-centric view order*.

Definition (App server's View Order) 1: The binary relation $\overset{C_i}{\mapsto}$ orders the views on results of write operation for an app server C_i. We refer to $\mapsto$ as *app server's view order*.

THEOREM 1: The library gives the same consistency guarantees as a storage system that guarantees causal consistency.

Proof: Causal consistency requires that the view order of a client (in our case of an app server) on operations follows the causal order of operations [15]. An operation $o1$ causally precedes an operation $o2$ ($o1 \rightsquigarrow o2$) if one of the following conditions hold [**?**], [15]:

$$\exists_{C_j}(o_1 \overset{C_j}{\rightarrow} o_2) \tag{1}$$

$$o_1 = w(x)v \wedge o_2 = r(x)v \tag{2}$$

$$\exists_{o \in O}\,(o_1 \rightsquigarrow o \wedge o \rightsquigarrow o_2) \tag{3}$$

We use the notation $o_a \rightsquigarrow o_b|(1)$ to denote that operation o_a causally precedes operation o_b according to condition (1). At the client-side, a storage system guarantees causal consistency if the following condition is preserved [15]:

$$\forall_{C_i}\forall_{o_1,o_2 \in OW \cup O_{C_i}}\left(o_1 \rightsquigarrow o_2 \Rightarrow o_1 \overset{C_i}{\mapsto} o_2\right) \tag{4}$$

It is necessary to show that the library provides a causally consistent view on operations for all app servers according to (4). Consequently, we consider the different conditions for causal ordering (1)-(3).

1) We show that a causally consistent view is preserved if two operations are of causal order according to (1).

   Proof: We assume that an app server C_j requests two consecutive write operations $w(x)a$ and $w(x)b$ for key x: $w(x)a \overset{C_j}{\rightarrow} w(x)b$. By contradiction, we assume that an app server C_i exists that views the result of $w(x)a$ after the result of $w(x)b$: $w(x)b \overset{C_i}{\mapsto} w(x)a$. We consider the two cases that the library reads $w(x)a$ from the cache and remote storage after reading $w(x)b$.

   a) We assume the cache returns the result of $w(x)a$. In order for the local cache to return the result of $w(x)a$ after the result of $w(x)b$, the following condition must hold: $vc(w(x)b) < vc(w(x)a)$ which leads to contradiction because of $w(x)a \overset{C_j}{\rightarrow} w(x)b$ which implies $vc(w(x)a) < vc(w(x)b)$.
   b) We assume the remote storage returns the result of $w(x)a$. The assumption that the cache returns the result of $w(x)a$ leads to contradiction according to 1a). We assume that the cache returns the result of $w(x)b$ and the library returns the result of $w(x)a$. This implies $vc(w(x)b) < vc(w(x)a)$ which leads to contradiction according to 1a).

   ■

2) A causally consistent view is preserved for C_i if $o_1 \rightsquigarrow o_2$ according to (2).

   Proof: As the read does not change the state, the results of both o_1 and o_2 are identical. Hence, the app server's view order can never be violated.

   ■

3) A causally consistent view is preserved for C_i if $o_1 \rightsquigarrow o_2$ according to (3).

   Proof: Transitivity of the causally precedes relation $\rightsquigarrow$ is generally guaranteed by the linear ordering of the ordering relation $<$ over the set of vector clocks

VC. For the sake of completeness, we conduct the proof as follows:
We assume that an operation o_1 precedes an operation o according to condition (1) or (2). Furthermore, we assume operations o precedes operation o_2 according to condition (1) or (2). By contradiction we assume: $o_2 \overset{C_i}{\mapsto} o_1$. Therefore, four different cases should satisfy the condition (3) based on the assumptions made.

a) $o_1 \rightsquigarrow o|(1) \wedge o \rightsquigarrow o_2|(1)$, i.e., all operations are writes by the same app server: This case leads to contradiction according to the proof of (1) as the $<$ relation on the vector clocks is transitive itself.

b) $o_1 \rightsquigarrow o|(1) \wedge o \rightsquigarrow o_2|(2)$, i.e., o_1 and o are writes by the same app server, o_2 is a read by another app server: According to the proof of (2), o and o_2 have the same result which leads to contradiction according to the proof of (1).

c) $o_1 \rightsquigarrow o|(2) \wedge o \rightsquigarrow o_2|(1)$, i.e. o_1 is a write and o and o_2 are a read and a write by another app server:
We assume C_j orders a write operation $w(x)a$ and any app server C_k views the result of the write operation $w(x)a$ before updating the value with $w(x)b$: $\exists_{C_j} w(x)a \wedge \exists_{C_k} r(x)a \overset{C_k}{\rightarrow} w(x)b$.
We assume by contradiction that an app server C_n exists that views the result of $w(x)a$ after the result of $w(x)b$: $w(x)b \overset{C_n}{\mapsto} w(x)a$.
We consider the two cases that the library views the result of $w(x)a$ from the cache and remote storage after returning the result of $w(x)b$ to any client C_n.

   i) We assume the cache returns the result of $w(x)a$. Since C_n has already seen the result of $w(x)b$ (either because he issued $w(x)b$ or because he read the result of $w(x)b$), the cache contains the result of $w(x)b$ and must return $w(x)b$ as $vc(w(x)a) < vc(w(x)b)$. This leads to contradiction.

   ii) We assume the remote storage returns the result of $w(x)a$: According to 3ci), the cache returns $w(x)b$. Because $vc(w(x)a) < vc(w(x)b)$, the library always returns $w(x)b$ which contradicts the original assumption of $w(x)b \overset{C_n}{\mapsto} w(x)a$.

d) $o_1 \rightsquigarrow o|(2) \wedge o \rightsquigarrow o_2|(2)$: This case leads to contradiction according to the proof of (2) as o would have to be a write and a read at the same time which is not possible.

■

■

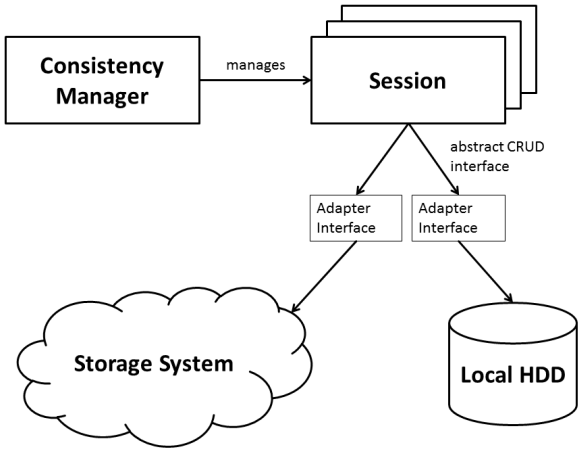

Figure 2. Basic Library Architecture

D. Implementation

Our approach has been prototypically implemented as a Java 6 library. Users can start one or more sessions with the same or different storage systems, each using its own local cache which is persistently written to disk. This is done via the ConsistencyManager singleton. Within the scope of a session, it is then possible to interact with cache and data stores using an adapter framework which offers operations to read, write and delete data items. Currently, adapters exist for the Amazon services S3[5], DynamoDB[6] and SimpleDB[7] – additional adapters just have to implement an interface and specify a mapping of multi-dimensional keys to the underlying data store. Figure 2 shows a high-level overview of the library's architecture.

Whenever a session is terminated (either directly by the user or indirectly via a crash of one or more components), the cache on the local disk is removed.

IV. Evaluation

To evaluate our approach as well as to test our library we first implemented and used a simulation environment to verify that our approach works under adverse conditions. Afterwards, we switched to experiments with a sample application running on top of several cloud storage offerings.

A. Simulation

A special storage adapter (see section III-D) just randomly drops updates and returns arbitrary values. We used it in a configuration where it created violations of MRC and RYWC for about 50% of all requests to really strain our library (commercial cloud storage offerings usually offer lower rates of violations, e.g., [9]). Next, we implemented a

[5]aws.amzon.com/s3
[6]aws.amzon.com/dynamodb
[7]aws.amzon.com/simpledb

test application which just reads and writes random values and checks each time for MRC and RYWC violations and used it with this adapter.

During several billion simulated requests each, we never encountered any consistency violations as long as our library was used. When running the same setup without our library, we could observe the expected number of inconsistencies.

B. Sample Application

As our sample application, we chose an internet forum as this allows to easily check for inconsistencies. We believe, though, that any other application would show the same or comparable results. Our internet forum implementation had ten conversation threads. During every test run clients would randomly choose one, read it completely, add a response and write it back as well as check for violations of MRC or RYWC. Each benchmark was repeated several times and used 30 clients which were deployed on 30 EC2[8] micro instances in the region eu-west, ten per availability zone. Each client executed 1,000 test runs, thus, totalling a number of 30,000 reads and writes each per benchmark. We ran benchmarks with and without our library for S3, DynamoDB (consistent and eventually consistent reads) and SimpleDB (also consistent and eventually consistent reads). Whenever we incurred an availability error, the benchmark just repeated the respective test run until it completed successfully. Each test configuration (e.g., DynamoDB with library and consistent reads) was repeated several times.

C. Results

As expected, all test runs using our library showed no violations of MRC and RYWC. When not using the library, i.e., accessing the data store directly, we could see huge numbers of consistency violations. Figures 3 and 4 show box plots of the number of consistency violations incurred by our 30 clients (CR stands for consistent read, ER stands for eventually consistent read). A value of 500 means that out of 1,000 reads half were consistency violations. Results for each test configuration were relatively stable when rerunning our benchmarks, i.e., the standard deviation values of repeated tests within one test configuration were close to zero.

Apart from counting inconsistencies, we also measured latencies for reads and writes to determine whether there is any relevant latency overhead caused by our library. During several S3 benchmarks, we did not see any deviations – neither between different benchmarks nor depending on using the library or not. Based on that, we believe it is feasible to say that the latency overhead caused by our library is negligible. In fact, values with library were often a little less than without library.

In our DynamoDB and SimpleDB benchmarks we saw extensive variability (between a few hundred and several thousand milliseconds) in the latency values as well as a large number of availability issues – both for test runs with and without library. This could either be due to general problems of the service at the time of our test run or due to creating too much load[9]. We could neither see arguments in favor of or against a latency overhead of our library during those tests as it was just not possible to identify a statistically relevant result.

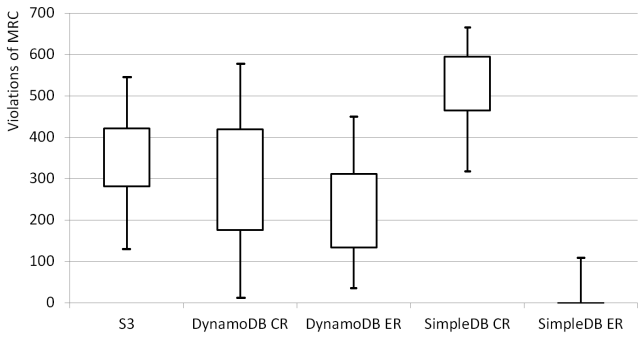

Figure 3. Violations of MRC without Library

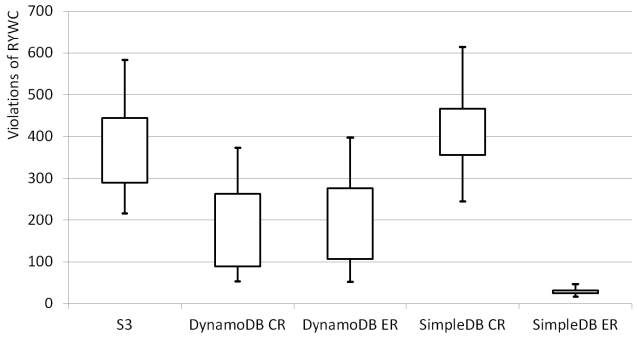

Figure 4. Violations of RYWC without Library

D. Discussion

When comparing our results to the benchmarking results of Bermbach and Tai [9], it seems that the number of violations of RYWC depends on the number of concurrent updates as well as the update propagation speed of the data store which typically implements a last write wins strategy: If the update propagation is too slow, the value will be overwritten by the next update. If there are only a few concurrent updates, there is more time to propagate the update. The number of MRC violations, in contrast, depends more on the rate of updates than its origin plus the degree of session stickiness offered by a system.

[8] aws.amazon.com/ec2

[9] In [16] Kossmann et al. discovered that several cloud storage systems, including SimpleDB, scale very poorly. For S3, in contrast, they could not find any scalability limitations.

Both the MRC and RYWC results for SimpleDB are interesting in that matter as they hint that a consistent read contains the newest value written which just might be a concurrent update whereas the eventually consistent read is likely to read the same replica which the client interacted with before, i.e., a high degree of session stickiness. This can also be seen in the MRC results for DynamoDB, though, definitely less pronounced.

Our evaluation clearly shows that especially applications with large numbers of concurrent updates benefit from using our approach. Previous results [9] show that even when this is not the case, consistency violations occur at least from time to time and applications must be able to cope with it. Often this will mean implementing some variant of our approach at application level, e.g., by reloading data whenever a violation is detected [17].

Based on our results, we believe it is a better idea to handle such violations at the middleware level, i.e., in our case by using our library, than at either application level or within the storage system.

A downside of our approach is an overhead both in terms of cost and time for persistence, network transfer and comparison of the vector clocks. For our test application, this overhead was negligible. It would be interesting to see, though, how this changes for different applications and workload – especially, when considering worst case applications with very large numbers of app servers issuing a very high write load and, thus, creating vector clocks with many entries as well as many conflicting versions. A detailed analysis of this is beyond the scope of this work.

V. Related Work

Mahajan et al. [18] show that an always available system (which is a necessary condition for cloud storage systems) can give no stronger consistency guarantees than CC. Brzezinski et al. [15], [19] investigate the relationship between data-centric and client-centric consistency models and show that data-centric CC can only be reached if all four client-centric models (MRC, RYWC, MWC and WFRC) are guaranteed. To our knowledge no previous approach exists that directly addresses the client-side guarantees as all previous approaches solely focus on data-centric guarantees and offer client-centric consistency only as a by-product.

COPS [11] and MDCC [20] both offer stricter consistency guarantees than eventual consistency. In COPS a client-side library adds metadata comparable to our vector clocks but in contrast to our approach this metadata is evaluated by the storage system below. So, their approach guarantees causal consistency but cannot run on top of arbitrary cloud storage systems, instead it is a storage system itself. MDCC also uses a client-side library but Generalized Paxos is used to offer strong consistency guarantees on top of arbitrary storage systems. This comes with a price, though: As the libraries need to communicate, this directly affects latency and under adverse conditions availability. Megastore [6] is implemented on top of Google BigTable [1] to increase its consistency guarantees via 2PC and Paxos. To our understanding, though, consistency is not addressed by the also included client-side library but rather by a separate service running either Paxos or 2PC.

Like Footloose [21], our library can be used to support offline operations with local progress on cached files and reconciliation upon reestablishment of connection, a concept termed physical eventual consistency. The difference, though, is that Footloose itself is a storage system and not intended to be used with existing eventually consistent data stores.

Krishnamurty et al. [22] propose an approach which allows clients to specify consistency requirements for operations but focuses on staleness. Yu and Vahdat [13] introduced the notion of a conit, a consistency unit, with the three dimensions staleness, order error and numerical error. Their middleware prototype allows clients to specify bounds on each of the dimensions. While this is certainly helpful, it is not clear how order error shall relate to client-centric consistency gaurantees like MRC or RYWC.

Brantner et al.'s implementation of a database on top of Amazon S3 [17] adds additional database features to S3 which is more a blob store than a database. They address consistency guarantees only as a sidenote and do not use client-side caching. If MRC and RYWC are desired additional metadata is used to identify violations and to refetch data in that case. For MWC and WFRC they incur the same problems as our approach.

Several approaches to consistency benchmarking [9], [10], [23], [24] allow to quantify staleness or to count violations of, e.g., MRC. This is certainly helpful in terms of information but has no influence on the actual guarantees of a datastore. The approach by Bailis et al. [25] allows to give predictions for consistency guarantees of dynamo-style [3] quorum systems for scenarios where no failures occur. This again has no effect on the actual consistency guarantees of a storage system but at least increases transparency regarding QoS levels.

VI. Conclusion

Applications running on top of distributed storage systems often have consistency requirements that cannot be fulfilled by the underlying storage system. This is especially the case for cloud storage and NoSQL systems.

In this work, we have presented an approach to increase consistency guarantees by using a middleware component running on the same server-side machines as the application code. This middleware service uses vector clocks and client-side caching to guarantee monotonic read consistency as well as read your writes consistency. It also helps towards reaching monotonic write consistency as well as write follows read consistency and reduces the chance of lost updates

due to last write wins strategies within the storage system.

In our evaluation, we have shown that (a) this is a real world problem when using cloud storage systems, (b) that our approach can give the respective guarantees and (c) that the overhead for the considered scenario is negligible in relation to the benefits of using it.

Future work should address efficient strategies for managing optimal session lengths to further reduce the overhead of our approach. Also, based on [26] different types of data might have different consistency requirements. Combinining our approach with consistency rationing could further reduce the overhead and should widen the range of feasible use cases.

Another interesting approach which we leave for future work would be to add some kind of agreement protocol to our library which asserts that different instances of our library agree on a total order for conflicting vector clocks. This would create the client-side illusion of Sequential Consistency as long as no failures occur. In the presence of failures, this extension could then decide to either block until the failure has been resolved (and thus continue to guarantee Sequential Consistency while sacrificing availability) or to use the library in its present state as a fallback solution still guaranteeing Causal Consistency while asserting continuous availability of the system.

ACKNOWLEDGMENT

The authors would like to thank Amazon Web Services who provided us with research grants to conduct our experiments.

REFERENCES

[1] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[2] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[3] 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 *Proc. SOSP*, 2007.

[4] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[5] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[6] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: providing scalable, highly available storage for interactive services," in *Proceedings of Conference on Innovative Data Systems Research*.

[7] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, p. 59, 2002.

[8] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37 –42, feb. 2012.

[9] 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*. ACM, 2011, p. 1.

[10] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade offs in commercial cloud storages: the consumers' perspective," in *5th biennial Conference on Innovative Data Systems Research, CIDR*, vol. 11, 2011.

[11] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 401–416.

[12] A. S. Tanenbaum and M. V. Steen, *Distributed Systems - Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Pearson Education, 2007.

[13] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 3, pp. 239–282, 2002.

[14] W. Vogels, "Eventually consistent," *Queue*, vol. 6, pp. 14–19, October 2008. [Online]. Available: http://doi.acm.org/10.1145/1466443.1466448

[15] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak, "From session causality to causal consistency," in *Proc. of 12th Euromicro Conf. on Parallel, Distributed and Network-Based Processing (PDP2004)*. Citeseer, 2004, pp. 152–158.

[16] D. Kossmann, T. Kraska, and S. Loesing, "An evaluation of alternative architectures for transaction processing in the cloud," in *Proceedings of the 2010 international conference on Management of data*. ACM, 2010, pp. 579–590.

[17] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a database on s3," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 251–264.

[18] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, availability, and convergence," *Technical Report TR-11-22, Computer Science Department, University of Texas at Austin*, 2011.

[19] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak, "Session guarantees to achieve pram consistency of replicated shared objects," *Parallel Processing and Applied Mathematics*, pp. 1–8, 2004.

[20] T. Kraska, G. Pang, M. Franklin, and S. Madden, "Mdcc: Multi-data center consistency," *Arxiv preprint arXiv:1203.6049*, 2012.

[21] J. Paluska, D. Saff, T. Yeh, and K. Chen, "Footloose: A case for physical eventual consistency and selective conflict resolution," in *Mobile Computing Systems and Applications, 2003. Proceedings. Fifth IEEE Workshop on*. IEEE, 2003, pp. 170–179.

[22] S. Krishnamurthy, W. Sanders, and M. Cukier, "An adaptive framework for tunable consistency and timeliness using replication," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 17–26.

[23] M. Klems, D. Bermbach, and R. Weinert, "A runtime quality measurement framework for cloud database service systems," in *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology*. Springer, 2012, Inproceedings, to appear.

[24] M. Klems, M. Menzel, and R. Fischer, "Consistency benchmarking: Evaluating the consistency behavior of middleware services in the cloud," *Service-Oriented Computing*, pp. 627–634, 2010.

[25] P. Bailis, S. Venkataraman, J. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," in *Technical report UC Berkeley, EECS-2012-4*.

[26] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: Pay only when it matters," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 253–264, 2009.