Optimistic Causal Consistency for Geo-Replicated Key-Value Stores

Kristina Spirovska, Diego Didona, Willy Zwaenepoel
École polytechnique fédérale de Lausanne
Email: first.last@epfl.ch

***Abstract*—Causal consistency is an attractive consistency model for geo-replicated data stores because it hits a sweet spot in the ease of programmability vs performance trade-off.**

In this paper we propose a new approach to causal consistency, which we call Optimistic Causal Consistency (OCC). The optimism of our approach lies in the fact that updates from a remote data center are immediately made visible to clients in the local data center. A client, hence, always reads the freshest version of an item, whose dependencies, however, might have not been installed in the local data center yet. When serving a read request, a server can detect whether it has not received such dependencies yet. This is achieved without inter-server synchronization thanks to cheap dependency meta-data supplied by the client. Upon detecting a missing dependency, the server waits to receive it.

This approach contrasts with the design of existing systems, which are prone to expose stale versions of a data items, to ensure that clients only see versions whose dependencies have already been replicated in the local data center.

OCC explores a novel trade-off in the landscape of consistency models. Because network partitions are practically rare events, OCC partially trades availability to improve other performance metrics. On the one side, OCC maximizes the freshness of data returned to clients and reduces the communication overhead. On the other side, a server might need to wait before serving a client's request, leading the system to be unavailable in case of a network partition. To overcome this limitation, we propose a recovery mechanism that allows an OCC system to fall back to a pessimistic protocol to recover availability.

We implement OCC in a new system, which we call POCC. We compare POCC against a recent (pessimistic) approach to causal consistency using heterogeneous workloads on an Amazon AWS deployment encompassing up to 96 nodes scattered over 3 data centers. We show that POCC is able to maximize the freshness of data returned to client while providing comparable or better performance than its pessimistic counterpart in a wide range of production-like workloads.

I. Introduction

Over the last years geo-replication has become the *de facto* standard for storage systems underlying large scale web applications. Data is sharded across several machines within a data center (DC) to achieve scalability and then replicated across data centers to deliver lower response times to geographically distributed clients and to achieve fault tolerance [1], [2], [3].

A critical choice when designing a geo-replicated data platform is the consistency guarantees that it achieves. At one end of the consistency spectrum, strong consistency [4] provides simple semantics, but incurs high latency and does not tolerate network partitions [5]. At the other end, eventual consistency provides excellent performance and tolerates partitions [6], but it is hard to program against.

Causal Consistency. Causal consistency [7] lies between these two endpoints and is an attractive model for building geo-replicated data stores. Causal consistency has garnered much attention as it hits a sweet spot in the ease of programming vs performance trade-off for this kind of systems [8], [9], [10], [11], [12], [13], [14]. On the one hand, it avoids the long latencies and inability to tolerate network partitions of strong consistency. On the other hand, it is easy to reason about and avoids some anomalies allowed under eventual consistency.

Existing systems employ different techniques to achieve causal consistency, but they all share the same key mechanism. When serving a read operation, a server within a DC returns the most recent version of an item whose causal dependencies are known to have been already replicated in that DC. Such version might be not the freshest one. This invariant allows such data stores to tolerate network partitions and DC failures. It demands, however, the implementation of dependency checking [8], [9] or stabilization protocols [13], [15] to track the delivery of dependencies. Not only do these protocols result in computational and communication overhead, but they also delay the visibility of new versions of data items, increasing the staleness of the data returned to clients.

Optimistic Causal Consistency. In this paper we argue that existing protocols are too *pessimistic* for modern data center deployments and we propose Optimistic Causal Consistency (OCC). According to OCC a server always returns the most recent available version of an item. Potentially unresolved dependencies are detected by the server upon serving a read operation, without the need for synchronization with other servers. This is accomplished by means of cheap dependency meta-data supplied by the client. When a potential unresolved dependency is detected, a server waits to receive it.

The effectiveness of our optimistic approach stems from two main insights. First, recent works have revealed that updates replication in data stores exhibits a *naturally consistent order*. Namely, a data item is typically replicated (and accessed) after its dependencies have already been propagated [16], [17], [18]. Hence, the most recent version of a data item could typically be returned without violating consistency [19]. OCC leverages this insight by having servers waiting to receive missing dependencies when serving a client's request, rather than relying on expensive dependency checking and stabilization protocols. Servers rarely incur such waiting overhead, because of the naturally consistent order of updates. Hence, OCC maximizes

the freshness of data returned to clients and improves resource efficiency with negligible impact on performance.

Second, network partitions can be considered relatively rare events (and complete DC failures even more rare) [17], [20], [21], [22]. OCC leverages this insight by avoiding (most of) the overhead associated with network partition tolerance during normal operative conditions, and by incurring it only when a network partition is actually occurring. To this end, OCC entails the infrequent –hence cheap– execution of a stabilization protocol to allow a system to fall-back to operate according to a pessimistic scheme in presence of a network partition. This design contrasts with existing ones, which incur dependency checking overhead and expose stale data items to clients even under normal operational conditions.

Contributions. We make the following contributions.
i) We introduce Optimistic Causal Consistency, discussing its benefits and inherent performance vs availability trade-offs.
ii) We present the design and implementation of POCC, a causally consistent protocol that implements OCC. POCC relies on physical clocks and vector dependency clocks to succinctly keep track of causal dependencies.
iii) We assess the benefits brought by OCC by comparing POCC with the (pessimistic) state-of-the-art design to causal consistency based on physical vector clocks. We run heterogeneous workloads on a large scale infrastructure on Amazon, comprising up to 96 nodes scattered across 3 geo-distributed sites. Our results indicate that optimistic causal consistency is effective in minimizing the staleness of data returned to client and delivers performance that is competitive or better than its pessimistic counterpart in a wide range of realistic workloads.

The remainder of the paper is structured as follows. Section II describes the target system model. Section III describes the approach underlying OCC. Section IV introduces POCC. Section V provides the evaluation of POCC. Section VI discusses related work. Section VII concludes the paper. The Appendix provides POCC correctness arguments.

II. Definitions and System model

A. Causal consistency

Causal consistency requires that servers of a system return values that are consistent with the order defined by the *causality* relationship. Causality is a happens-before relationship between two events [23], [7]. For two operations a, b, we say that a causally depends on b, and write $a \rightsquigarrow b$, if and only if at least one of the following conditions holds:

- Thread-of-execution: a and b are operations in a single thread of execution, and a happens before b.
- Reads-from: a is a write operation, b is a read operation, and b reads the value written by a.
- Transitivity: there is some other operation c such that $a \rightsquigarrow c$ and $c \rightsquigarrow b$.

Unless stated otherwise, we use lower case letters, e.g., x, to refer to a key and the corresponding capital letter, e.g., X to refer to a version of the key. A version X of a data item x is causally dependent on a version Y of data item y if the write of X causally depends on the write of Y.

Finally, we define an item X *stable* in a DC if all the dependencies of X have already been replicated in such DC.

B. Convergent conflict handling

Two operations a, b are concurrent if neither $a \rightsquigarrow b$ nor $b \rightsquigarrow a$. If both a and b are concurrent write operations on the same key, they *conflict*. Namely, their result can be propagated to replicas in different orders, potentially leading to replicas to diverge forever. To enforce that different replicas converge to the same value for any key, a protocol must implement a convergent conflict handling procedure. Such procedure must implement a commutative and associative function, such that replicas can manage update replication messages in the order they receive them and converge to the same state.

The most popular convergent conflict handling functions is the last-writer-wins rule (Thomas' write rule [24]): given two updates, one of them is deterministically decided to having occurred later than the other, determining the value of the value written. The implementation of OCC that we present applies, for simplicity, the last-writer-wins rule to achieve convergent conflict handling. OCC can, however, be implemented also in systems that manage conflicts differently [8], [13], [15].

C. System model

We assume a distributed key-value store that manages a large set of data items. The data-set is split into N partitions and each key is deterministically assigned to a single partition according to a hash function. Each partition is replicated at M different sites, each corresponding to a different datacenter. Hence, a full copy of the data is stored at each datacenter.

We assume a multiversion data store. An update operation creates a new version of an item. In addition to the actual value of the key, each version also stores some metadata, in order to track causality. The system periodically garbage-collects old versions of items. We further assume nodes in the system can communicate through point to point lossless FIFO channels.

The system provides the following operations to the clients:

- PUT(key, val): A PUT operation assigns value *val* to an item identified by *key*. If item *key* does not exist, the system creates a new item with initial value *val*. Else, a new version storing *val* is created.
- val $\leftarrow$ GET(key): A GET operation returns the value of the item identified by *key*. A GET operation is such that its return value does not break causal consistency as explained in the following. Assume X is a version of data item x, Y is a version of data item y and $X \rightsquigarrow Y$. Suppose that a client c issues a GET(y) operation, receiving Y as result. Then, any subsequent GET(x) operation issued by c must return either X or a version X' such that $X' \not\rightsquigarrow X$.
- $\langle$vals$\rangle$ $\leftarrow$ RO-TX$\langle$keys$\rangle$: This operation provides a causally consistent read-only transaction [8], [9]. Namely, assume X and Y are two versions of items x and y, respectively. If a read-only transaction returns X and Y,

and X$\rightsquigarrow$Y, then there does not exist another version of x, X', such that X$\rightsquigarrow$X'$\rightsquigarrow$Y.

At the beginning of a session, a client c connects to a node n_c in the closest datacenter according to some load balancing scheme. c does not issue the next operation until it receives the reply to the current one. Operations on data items not stored by n_c are transparently forwarded to the node(s) responsible for such data items, and the result is relayed back to c by n_c.

III. Optimistic Causal Consistency

In this Section, we describe the design of OCC. We start by presenting the key principles on which OCC is based. Then we discuss the data freshness vs availability trade-off that OCC explores. Finally, we describe the mechanism employed by OCC to preserve availability during network partitions.

A. The design of OCC

The overall goal of OCC is to maximize the freshness of data returned to clients without the need to rely on dependency checking messages or stabilization protocols to enforce causal consistency. To achieve this goal, OCC implements a client-assisted lazy dependency resolution protocol. In the following, we first outline how OCC maximizes the freshness of returned data items, and then we describe how its dependency resolution protocol contributes to achieve such goal.

Data freshness maximization. Maximizing the freshness of data returned to the client takes a different meaning depending on whether the client performs a GET operation or a RO-TX. Upon receiving a GET(x) request from a client c, a server p always returns the freshest version of x that is compatible with the history of c. If c does not depend on any version of x, then p returns the most recent version of x that it locally stores. If c, instead, has established a dependency towards a specific version X, then p returns the freshest locally available version of x, noted X', such that $X' \not\rightsquigarrow X$. The difference with existing, pessimistic designs lies in that OCC allows p to return X' even if X' is not locally stable yet.

In case c reads x from p in the context of a RO-TX operation, p cannot safely return the freshest available version of x. Assume that X is the freshest version of x and that there exist Z, Z' such that $Z \rightsquigarrow Z' \rightsquigarrow X$. If c reads Z within a transaction, returning X in the same transaction would violate the semantics of the RO-TX operation as defined in Section II. To explain how OCC maximizes the freshness of data returned within the scope of a transaction, we introduce the concept of snapshot visible to a transaction. We define it as the set of data items that can be returned to c as a result of a transaction without violating the semantics of the RO-TX operation. The optimism of OCC, then, lies in how the boundaries of a transaction's snapshot are determined. In OCC, the boundaries of a transactional snapshot are defined on the basis of the items currently received by nodes in the local data center when the transaction is issued. In existing systems, instead, such boundaries are determined by the set of items that are stable at the time the transaction is issued.

Client-assisted lazy dependency resolution. Because OCC exposes unstable items it can be that a client c establishes a dependency towards an item Z that has not been received yet by its corresponding local partition p_Z. If then c wants to read such item, OCC must prevent p_Z from returning a version of z that is not causally consistent with c's history. OCC achieves causal consistency despite exposing unstable data by means of a client-assisted lazy dependency resolution protocol, that we explain in the following.

Clients store information about the causal dependencies established when performing reads. If c reads Y and there exists X such that $X \rightsquigarrow Y$, c needs to record it has established a dependency towards X and Y. The client-side dependency meta-data is supplied by c when it performs a later operation.

For a read operation (GET or RO-TX), such information is needed to allow a server p to determine whether its own state is consistent with c's history. Referring to the previous example, if c wants to read X after it has read Y, the dependency meta-data provided by c, together with some state information that p locally stores, allows p to check whether it has already received X or not. If p has already received X, then the freshest local version of x that p stores is compatible with c's history, and it is returned to c. If p has not received X yet, p must receive it before serving c's request. In this case, p simply stalls c's request until it receives X.

For a PUT operation, the client-side dependency information is needed to know what the newly created item depends on. Moreover, such information might also be needed to enforce state convergence. Some convergent conflict handling schemes, in fact, require that upon writing Y on server p, p must have already delivered all the versions of y on which Y causally depends [8], [10], [13]. In this case, in OCC, p must wait, if necessary, until it has received the freshest version of y on which the issuing client depends.

On the one hand, this lazy dependency resolution scheme is cheap to implement, as it does not require any synchronization among servers. On the other hand, it potentially introduces delays to serve an operation, thus hurting response times. The effectiveness of OCC relies on the insight that, typically, updates are propagated in order of creation. Therefore, a server typically already stores a version of a key that is causally consistent with a client's history. Moreover, suppose that two items X and Y such that $X \rightsquigarrow Y$ are propagated in a remote DC in an order that does not respect causality (i.e., first Y, then X). A client c in that DC would be prone to be stalled only if it first accessed Y and then tried to read X. By the time this sequence of operations takes place, it is very likely that X has been installed at the DC, avoiding any stall for c.

These claims are backed by empirical evidence gathered on popular commercial cloud data stores [16], [19], [25], a massive scale deployment such as Facebook's [18] and are also confirmed by analytical models [17].

OCC can be implemented with any dependency tracking mechanism that has been proposed in literature, e.g., dependency lists [8], dependency matrices [10], physical scalar clocks [13] and physical vector clocks [15]. Because the client

has to supply dependency information upon each operation, however, it is paramount to encode dependencies in the most succinct fashion possible. As we shall show in Section IV, POCC, our implementation of OCC, meets this requirement by relying on physical vector clocks, which can keep track of dependencies with a overhead that is only linear with the number of data centers in the system.

B. Data Freshness vs Availability

OCC aims to maximize the freshness of data exposed to clients. Its potentially blocking behavior, however, makes it vulnerable to network partitions. Suppose, for example, there are two items X, Y such that $X \rightsquigarrow Y$. Assume Y gets replicated to DC', but X is prevented from doing so by a network partition. If a client c in DC' establishes a dependency on Y and then tries to read X from node p, p must block until the network partition heals to receive X and, thus, to preserve causal consistency. If the partition does not heal, however, the client's request is blocked indefinitely. Namely, OCC is not *always available*, because it cannot guarantee that any client's operation is always completed in a finite amount of time.

Existing pessimistic causally consistent protocols do seamlessly tolerate network partitions, and full DC failures (which can be modeled as un-healed network partitions). OCC trades this ability for a higher freshness in data returned to the client. This might seem an unduly high cost to pay, given that the CAP theorem states it is possible for a service to be both available and network partition tolerant [5].

Highly available OCC. To circumvent this limitation, we propose to augment OCC with a recovery procedure aimed to regain availability during network partitions. This procedure follows the structure outlined by Brewer to breach the CAP boundaries and it is based on three phases [21]. In the first phase, the network partition is detected. In the second one, the system enters a partition-tolerant mode, in which it may give up some functionalities or properties. The third phase is the recovery one, during which the system switches back to run its (non network partition tolerant) protocols.

We explain our recovery mechanism starting from the aforementioned blocking condition example. If p blocks while serving a request from c, as soon as p realizes there is a network partition occurring, it closes the session with c. A network partition can be identified by p if it blocks for more than a configurable amount of time. At this point, c re-initializes its session. This new session is managed according to a pessimistic protocol and, therefore, is ensured not to block even during the ongoing network partition. The cost for this session re-initialization is that the client might not be able to see the same version of some data items read or written in the optimistic session. If and when the network partition heals, the session can be promoted again to be managed according to the optimistic approach.

Equipped with this recovery mechanism, OCC can be implemented in a *highly available* fashion, representing a novel and unexplored trade-off between performance and availability. Specifically, OCC trades the ability to *seamlessly* tolerate network partitions –only minimally impacting the availability property– with a higher freshness in data returned to clients.

We believe that, for many applications, this is a reasonable, if not favorable, trade-off. As already stated, in fact, careful engineering and redundant links make modern geo-distributed data-centers reliable enough to regard network partitions as infrequent events [20], [21], [22], [26]. This is also confirmed by the successful implementation and deployment of strongly consistent geo-replicated storage systems, which, by the CAP theorem, do not tolerate network partitions by design [1].

Further, if a network partition does occur, the cost of re-initializing a client session, both in terms of time and data visibility, is affordable for many applications, e.g., social networks. In particular, it is important to note that our recovery mechanism does not expose the application to any behavior or anomaly that is not encompassed by a pessimistic implementation of causal consistency. In fact, an application that relies on the causal semantics is naturally coded to tolerate the unexpected re-initialization of a session. This stems by the stickiness property of causal consistency [27]. Suppose a client c establishes a session towards a server in DC'. c can lose the connection towards DC' because of a network partition or because of the failure of DC'. In that case, similarly to what happens in our recovery protocol, c opens a new session towards a new server in another data center, which might have not received item versions on which c depends on (and that c has potentially written) [1].

In the case of a full DC failure or, equivalently, of a network partition that does not heal, OCC can be subject to what we define the phenomenon of the *lost update*. Assume, for example, there exist X, Y such that $X \rightsquigarrow Y$ and both have been created in DC'. It can happen that Y gets replicated in DC'' and X is prevented to do so because of the failure of DC'. Then, Y can be read in DC'' and new items depending on Y can be written, establishing a dependency towards an item that will never be received by DC''. As a consequence, an OCC would fall-back to run a pessimistic protocol without the possibility to switch back to operating optimistically.

A possible mechanism to recover from this situation is to discard items that depend on a lost update and that have been created after the failure of DC'. This recovery mechanism causes the loss of some updates. As previously discussed, however, data loss is a consequence of a DC failure that is already encompassed also by pessimistic approaches to causal consistency. In pessimistic approaches, however, only updates originated in the failed DC can be lost. In OCC, instead, also updates from healthy DCs might get discarded.

We finally note that the implementation of a mechanism to recovery from full DC failure is a requirement also for other state-of-the-art protocols. In GentleRain [13], for example, this is because a full DC failure prevents nodes in other DCs to install remote updates. In Cure [15], instead, a recovery mechanism is needed because healthy DCs might have not

[1] This limitation is addressed for example by SwifCloud [14], which enables the transparent client fail-over to another DC [14]. SwiftCloud, however, uses caching at the client side and also delays the visibility of local updates.

Algorithm 1 POCC: operations at client c

```
1: function GET(key k)
2:     send ⟨GETReq k, RDV_c⟩ to server
3:     receive ⟨GETReply v, ut, DV, sr⟩
4:     RDV_c ← max{RDV_c, DV}                          ▷ Track dependencies
5:     DV_c ← max{RDV_c, DV_c}
6:     DV_c[sr] ← max{DV_c[sr], ut}                    ▷ Update client's dependencies
7:     return v
8: end function

9: function PUT(key k, value v)
10:    send ⟨PUTReq k, v, DV_c⟩ to server
11:    receive ⟨PUTReply ut⟩
12:    DV_c[m] ← ut                                    ▷ Update client's dependency at local replica
13: end function

14: function RO-TX(key-set χ)
15:    send ⟨RO-TX-Req χ, RDV_c⟩ to server p_m^n
16:    receive ⟨RO-TX-Resp D⟩
17:    for (d ∈ D) do
18:        read d as if it was the result of a GET
19:    end for
20: end function
```

Algorithm 2 POCC: operations at server p_n^m

```
1: upon receive ⟨GETReq k, RDV_c⟩ do
       ▷ Ensure p_n^m's state is consistent with c's history
2:     wait until VV_n^m[i] ≥ RDV_c, i = 0...M−1, i ≠ m
3:     d = argmax_{d.ut}{d : d.k == k}                 ▷ Version of k with highest timestamp
4:     send ⟨GETReply d.v, d.ut, d.DV, d.sr⟩ to client

5: upon receive ⟨PUTReq k, v, DV_c⟩ do
       ▷ Ensure p_n^m's state is consistent with c's history. (Optional: see Sec. IV-B)
6:     wait until VV_n^m[i] ≥ DV_c, i = 0...M−1, i ≠ m
7:     wait until max{DV_c} < Clock_n^m
8:     VV_n^m[m] ← Clock_n^m                           ▷ Update version vector
9:     create new item d
10:    d ← ⟨k, v, m, VV_n^m[m], DV_c⟩
11:    insert d to version chain of key k
12:    for each server p_n^j, j ∈ {0...M−1}, j ≠ m do  ▷ Replicate d
13:        send ⟨REPLICATE d⟩ to p_n^j
14:    end for
15:    send ⟨PUTReply d.ut⟩ to client

16: upon receive ⟨REPLICATE d⟩ from p_n^j do
17:    insert d to version chain of key d.k
18:    VV_n^m[j] ← d.ut

19: upon every Δ time do
20:    ct ← Clock_n^m
21:    if ct ≥ VV_n^m[m]+Δ then
22:        VV_n^m[m] ← ct
23:        for each server p_n^j, j ∈ {0...M−1}, k ≠ m do
24:            send ⟨HEARTBEAT ct⟩ to p_n^j
25:        end for
26:    end if

27: upon receive ⟨HEARTBEAT ct⟩ from p_n^j do
28:    VV_n^m[j] ← ct

29: upon receive ⟨RO-Xact χ, RDV_c⟩ from c do
30:    χ_i = {k ∈ χ : partition(k) == i}               ▷ Set of requested keys per node
31:    D = ∅                                           ▷ Items to return to client
32:    TV = max{VV_n^m, RDV_c}                         ▷ Entry wise
33:    for (i s.t. χ_i ≠ ∅) do
34:        send ⟨SliceREQ χ_i, TV to p_i^m⟩
35:        receive ⟨SliceRESP D_i⟩ from p_i^m
36:        D ← D ∪ D_i
37:    end for
38:    reply ⟨D⟩ to c

39: upon receive ⟨SliceREQ χ, TV⟩ from p_i^m do
40:    wait until VV_n^m ≥ TV                          ▷ Ensure p_n^m installed all updates in the snapshot
41:    D = ∅
42:    for k ∈ χ do
43:        D_k = {d : d.k == k ∧ d.DV ≤ TV}             ▷ Compute visible versions set
44:        d_k ← argmax_{D_k}{d.ut}                     ▷ Freshest visible version
45:        D ← D ∪ d_k
46:    end for
47:    reply ⟨SliceRESP D⟩ to p_i^m
```

received the same set of updates from the failed DC, leading their states to diverge.

IV. POCC: A SCALABLE IMPLEMENTATION OF OCC

This section presents the design of POCC, a system that implements OCC. In this paper, we do not present the implementation of the protocols needed to recover from network partitions or full DC failures, but we only focus on the protocol run by POCC during normal operational behavior.

In POCC each server is equipped with a physical clock, which provides monotonically increasing timestamps. We assume that such clocks are loosely synchronized by a time synchronization protocol, such as NTP [28]. The correctness of our protocol does not depend on the synchronization precision.

In POCC each update u is assigned a physical clock timestamp that represents the time at which the corresponding item has been created. u is also assigned a dependency vector with one entry per DC. Because dependencies are tracked at the granularity of the DC this vector tracks *potential* dependencies. Namely, if the i-th entry of the vector is t, the update is marked as *potentially* dependent on *every* item originated from the i-th DC with timestamp lower than t. Dependency vectors represent a trade-off between meta-data efficiency and dependency tracking granularity. On the one hand, they allow POCC to succinctly track dependencies. On the other hand they they might cause a client's request to be (uselessly) stalled because of a *potentially* unresolved dependency that does not correspond to any real dependency.

We now describe the protocol in detail. An informal proof of POCC correctness is provided in Appendix.

A. Meta-data

Item. An item version d is represented as a tuple ⟨k,v,sr,ut,dv⟩. k is the unique id that identifies the key of which d is a version. v is the value of the item. sr is the source replica of d, namely the id of the data center in which d has been created by means of a PUT operation. ut is the update time, i.e., the physical timestamp of the item, computed as the creation time of the item at its source replica. dv represents a dependency vector and consists of M entries. $dv[i]$ is the update time of the item d' with the highest timestamp such that i) d' has been originated at the $i-$th replica and ii) d potentially depends on d'.

Client. A client c maintains during its session a dependency vector DV_c and a read dependency vector RDV_c, both consisting of M entries. $DV_c[i]$ is the update time of the item d with the highest timestamp such that i) d has been originated in the $i-$th datacenter and ii) c has established a potential dependency towards d. This vector is stored together with any item c writes. $RDV_c[i]$ is the update time of the item d with the highest timestamp such that i) d has been originated at the

$i-$th DC and *ii*) c has read an item that potentially depends on d. Namely, RDV is computed as the entry-wise maximum of the dependency vectors associated with items read by c.

Server. p_n^m maintains a version vector VV_n^m [29], [7] consisting of M physical timestamps corresponding to updates seen by p_n^m. $VV_n^m[m]$ is the highest update timestamp of any update originated at p_n^m. Similarly, p_n^m has received all updates with timestamp up to $VV_n^m[i](i \neq m)$ from p_n^i.

B. Operations

We now describe how servers running POCC support GET and PUT operations issued by clients, replicate updates and exchange heartbeats to stay synchronized. Algorithms 1 and 2 show the pseudo-code of the protocol running at the client and server side, respectively. In the discussion we indicate a target client as c. We refer to the server which handles c's request as p_n^m. p_n^m can be the node with which c has established a session, or the node to which the request has been forwarded (as described in Section II). As said in Section II, concurrent updates to the same key are handled by means of the last-writer-wins rule. The "last" version of a data item is the one with the highest update timestamp. Ties are broken by looking at the source replica id (lowest wins).

GET operation. c sends a request $\langle$GET k, $RDV_c\rangle$, where k is the key of the item to be read. p_n^m checks whether its version vector is entry-wise greater than or equal to RDV_c. This check is not done for the m-th entry because dependencies towards local items are trivially always satisfied. If the check is positive, it means that p_n^m has received all the items of the n-th partition on which c potentially depends. In this case, p_n^m returns the version in k's item chain with the highest update timestamp. If at least one entry of VV_n^m is smaller than RDV_c, it means that c potentially depends on an item d that p_n^m has not replicated yet. Since d can represent an instance associated with key k, p_n^m has to wait for its receipt, or else it might return a version of k that is not causally consistent with c's history. Therefore, p_n^m blocks until $RDV_c[i] \leq VV[i], i \neq m$.

Once p_n^m has determined the correct version d to return, it replies to the client with d's value, update timestamp, dependency vector and source replica. The client then tracks the newly established dependencies, by updating RDV_c and DV_c with d's dependencies and DV_c with d's update time.

PUT operation. c sends a request $\langle$PUT $k, v, DV_c\rangle$, where k is the key of the item to be written and v is the desired value to associate with k. p_n^m, first waits until the local physical clock is higher than the maximum timestamp in DV_c. In this way, p_n^m ensures that the soon-to-be-created item d has a higher update timestamp than any of its potential dependencies.

p_n^m then updates the local entry of its version vector with its physical clock time and creates a new version d of k. Then, d is inserted in the version chain corresponding to k. Finally, p_n^m sends a reply with the update time of the newly created item version to c. Upon receiving d as a reply, c sets the $m-$th entry of DV_c to $d.ut$ to track the newly established dependency.

Updates replication. p_n^m replicates local updates asynchronously by sending them in update timestamp order to its replicas at the other DCs. When a server receives a replicated update d from replica p_n^m, it inserts d in the corresponding version chain and advances its m-th entry of its local version vector to the update time of d.

Heartbeats. If p_n^m does not receive update requests from clients, it does not send replication messages to its replicas either. Therefore, other replicas cannot increase the m-th entry in their version vector. Keeping the version vector up-to-date, as we shall see, is necessary to implement a garbage collection protocol to remove stale versions of data items. Therefore, a partition that does not receive requests for local updates for a time longer than Δ broadcasts its latest clock time to its replicas. It does so by piggybacking the clock time in the heartbeat messages used by failure detectors. Heartbeat messages and update replication messages are sent in order of increasing update timestamps and clock values.

RO-TX. A client c sends a request $\langle$RO-TX $\chi, RDV_c\rangle$ to a server p_n^m, where χ is the set of keys to be read. Upon receiving the request, p_n^m first determines the timestamp vector of the snapshot visible to the transaction TV. As said in Section III-A, when serving a transaction p_n^m cannot simply return the freshest version of a key. Thus, TV has to be as recent as possible, to avoid returning excessively stale versions of the requested data items, and must include all potential dependencies established by the client. Hence, TV is computed as the entry-wise maximum between VV_n^m and RDV_c. Then, p_n^m sends TV to every node p_i^m that holds at least one key in χ, together with the set of keys p_i^m has to read.

Garbage collection. Periodically, nodes within a DC exchange the vector corresponding to the aggregate maximum of the TV vectors corresponding to active transactions. If on p_n^m there is no running transaction, p_n^m sends VV_n^m. The servers, then, compute the garbage collection vector GV as the aggregate minimum of the received vectors. Then, each server p_m^m scans the version chain of keys it replicates in descending timestamp order. For each key k, p_n^m retains up to and including the first version d_k such that $d_k.DV \leq GV$. Namely, p_n^m keeps the oldest version of k that can be accessed within the scope of a transaction, and removes oldest versions.

C. Highly available POCC

To achieve high availability in presence of network partitions POCC must be able to fall-back to a pessimistic protocol. Because POCC is based on physical vector clocks, POCC can be easily augmented to fall-back to run the protocol implemented in Cure [15], a recent causally consistent system also based on physical vector clocks [2]. We call Highly Available POCC (HA-POCC) the resulting augmented version of POCC.

In Cure, nodes within a DC periodically run a stabilization protocol. It consists of exchanging their version vectors and

[2]Cure's programming model is transaction-centric. It is, however, straightforward to adapt Cure to support GET and PUT operations.

computing the aggregate minimum called Globally Stable Snapshot (GSS). $GSS[i] = t$ means that all nodes within a DC have installed all updates originated in the i-th DC. In Cure a remote item d is visible only if $d.DV \leq GSS$, meaning that all d's dependencies have been received. Local items are immediately visible, because they depend only on stable items.

HA-POCC runs this stabilization protocol much less frequently than Cure, because HA-POCC only needs to use the GSS in the uncommon case of suffering from a network partition. HA-POCC, thus, reduces the overhead associated with the stabilization protocol during normal operational behavior. Because the GSS is infrequently updated, however, this higher resource efficiency comes at the cost of an increased perceived data staleness during network partitions. We argue this is a favorable trade-off, since network partitions are rare events.

During network partitions, HA-POCC runs Cure's protocol except for one detail. In Cure, local data items are always visible, because they depend on stable items. In HA-POCC, however, a local item can depend on remote items that are not stable yet, or that have not even been replicated in the current DC yet. Therefore, HA-POCC needs to distinguish between clients that are running the optimistic protocol and clients that are running the pessimistic protocol. In this way, servers can recognize a local item d created by an optimistic client and make d visible to pessimistic clients only if d is stable according to the pessimistic protocol.

Finally, HA-POCC adopts the garbage collection protocol of Cure, because HA-POCC needs to keep the oldest version of an item that could be accessed by the pessimistic protocol.

V. Evaluation

In this section we evaluate the effectiveness of OCC. Because OCC departs from the traditional way causal consistency is achieved, in this paper we primarily want to assess whether OCC can be efficiently implemented and can deliver performance that is competitive with pessimistic designs while increasing data freshness. Specifically, the focus of this evaluation is on performance during normal operational behavior, i.e., in the absence of network partitions, which is the behavior expected during the vast majority of its running time. We leave the investigation of the behavior of the system during a network partition and its recovery for future work.

To evaluate the effectiveness of OCC, we compare the performance achieved by POCC with the one of Cure*, a reimplementation of Cure [15], a state-of-the-art causally consistent system based on vector clocks. Because Cure only supports transactional operations, we have augmented Cure* with support for simple GET and PUT operations. We can compare POCC and Cure* in a fair manner because the amount of meta-data exchanged by clients and servers to implement the operations is the same. The two mainly differ in that POCC does not run any stabilization protocol and does not need to search for a stable version of a key when serving a GET.

A. Experimental test-bed

We consider an Amazon AWS deployment consisting of 3 DCs and 32 partitions per DC. The data centers are located in Oregon, Virginia and Ireland. We use *c*4.*large* instances, corresponding to 2 virtual CPU and 3.75 GB of RAM.

All data is kept in memory and no fault tolerance mechanism is implemented. This allows us to isolate the effects of OCC from the dynamics of the specific fault tolerance mechanism implemented, e.g., primary copy [30], Paxos [31] or chain replication [32].

Each data partition is composed of one million key-value pairs. We consider small items, with both keys and values of 8 bytes, because they are predominant in many production workloads [2], [33], [34], [35]. Keys are chosen within each partition according to a zipf distribution with parameter 0.99.

Clients are collocated with servers, establish their sessions with the collocated server and perform operations in closed loop. We introduce a think time between client operations to simulate a more realistic workload, in which a client does not simply injects requests but also processes the result of a performed operation. Naturally, having a think time between operations also lowers the chances that a request blocks when using OCC, because it gives time to servers to receive potentially missing client dependencies. We set the think time to 25 milliseconds. This value is low enough to avoid masking the blocking dynamics in POCC and high enough to allow us to fully load the compared systems without requiring an excessive number of client threads. We note that 25 milliseconds is orders of magnitude lower than the think time used in standard session-based benchmarks like TPC-W [36], TPC-C [37], SpecWeb2005 [38] and Rubis [39], and therefore allows us to benchmark the effectiveness of OCC in a particularly challenging setting.

We run NTP [28] to keep physical clocks synchronized. As in previous work [15], clocks are synchronized before each experiment. We use the NTP server 0.amazon.pool.ntp.org.

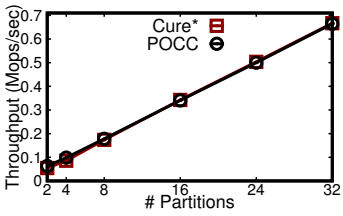
Both POCC and Cure* use the last-writer-wins rule to arbitrate conflicting updates to the same key. In POCC we enable the waiting condition in the server side handling of a PUT request (Algorithm 2 Line 6). Despite this not being needed to implement the last-writer-wins rule, this allows us to observe the behavior that POCC would have when implementing a different convergence conflict handling mechanism.

Heartbeats are sent by a node if it does not serve any put request for 1 millisecond. The stabilization protocol in Cure* is run every 5 milliseconds.

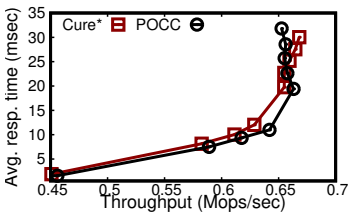
B. Get-Put workload

We start by experimenting with a workload in which clients issue GET and PUT requests. In such a workload, a GET:PUT ratio of $N:M$ means that each client issues N consecutive GETs followed by one PUT. Each GET operation targets a different partition. The PUT operation is issued against a key in a partition chosen uniformly at random.

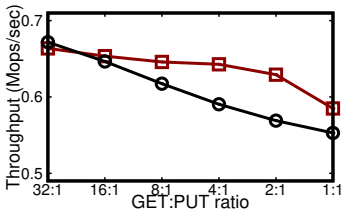
Scalability. In the first experiment, we asses the scalability of POCC. To this end, we deploy POCC and Cure* on a system composed of different partitions (from 2 to 32) and we measure the maximum achievable throughput. The GET:PUT ratio is $p:1$, where p denotes the number of partitions.



(a) Throughput while varying the number of partitions.



(b) Avg. resp. time on 32 partitions with a 32:1 GET:PUT workload.



(c) Throughput on 32 partitions with different GET:PUT workloads.

Fig. 1. Scalability and performance with workloads composed of GET and PUT operations in Cure* and POCC.

Figure 1a reports the result of this experiment. The plot shows that the two systems achieve basically the same throughput, showing that the optimistic approach can be implemented with no throughput loss with respect to the corresponding pessimistic implementation.

Response time. In the next experiment we fix the number of partitions to 32 and we investigate the average operation response time achieved while increasing the workload intensity. The result of the experiment is depicted by Figure 1b. It is possible to see that POCC achieves a slightly lower response time than Cure* before reaching the saturation point at about 0.65 Mops/sec. This is because, as we shall see more in detail later, POCC rarely blocks upon serving an operation and it is more resource efficient than Cure*. POCC, in fact, never traverses an item's version chain, nor needs to run any stabilization protocol (or can run it much more infrequently). Under very high load, POCC performs slightly worse than Cure*, because the better resource efficiency is outweighed by the incidence and extent of operations blocking.

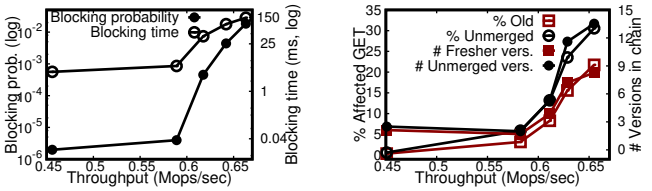
Sensitivity to write intensity. We now assess the sensitivity of the two systems to the workload write intensity. To this end, we run workloads with different GET:PUT ratios (from 32:1 to 1:1) on a 32 partitions deployment. As shown by Figure 1c, the throughput achieved by the two systems naturally decreases the write intensity increases. The performance degradation, however, is more evident for POCC because of its potentially blocking behavior. In fact, a higher update rate implies a higher likelihood that the delivery of a replicated updated is delayed yielding an operation to block in another DC. POCC, however, remains competitive with Cure*, achieving a maximum throughput loss of 10% (corresponding to the 2:1 ratio). This result shows that OCC is more suited for read intensive workloads. Luckily, typical production workloads are heavily read dominated, attaining GET:PUT ratios that are even much higher than the one targeted by our evaluation (up to 300:1) [3], [33], [40].

Blocking dynamics vs staleness. We now investigate the blocking behavior of POCC and the staleness of data returned to client by Cure*. Evaluating these two metrics allows us to understand the implications of the pessimistic and optimistic designs, and it is instrumental to better understand the performance results presented so far. To this end, Figure 2 reports statistics about the blocking incidence with POCC (Figure 2a) and about the data staleness perceived by the clients with Cure* (Figure 2b). The results refer to the aforementioned 32:1 GET:PUT workload run over 32 partitions.

Figure 2a, specifically, reports the probability that, in POCC, an operation blocks and the average blocking time for a blocked operation. The plot shows that the blocking probability is negligible under moderate to high load and that becomes noticeable only as the system approaches the maximum achievable throughput (0.65 Mops/sec). A similar dynamic is exhibited by the blocking time, which is in the order of a few microseconds as long as the system is not heavily loaded. The plot shows that, as long as the workload intensity is not too close to the maximum sustainable (i.e., up to 0.6 Mops/sec), the possible blocking behavior of POCC has a negligible effect. In particular, up to 0.6 Mops/sec the blocking probability is lower than 0.001, meaning that the 99.999-th percentile of the operation response times is not affected by the blocking behavior of POCC. Only when the workload reaches POCC's saturation point the system becomes very prone to stall client requests, reaching a block probability higher than 0.01.

To describe the results corresponding to data staleness in Cure* we first provide a definition of *old* and *unmerged* returned data item. We define a data item "old" if the version returned to the client is not the one with the highest timestamp. We define a data item "unmerged" if there is at least one version of the data item that is not stable yet, regardless of whether the version of the item returned to the client is the freshest. Naturally, an old item is also unmerged, because it means that the returned version has not been merged with other remote versions with higher timestamp that are not visible yet. The vice-versa is not true because it is possible that the version with the highest timestamp is visible in because it is local, yet there are older versions not yet stable.

We measure statistics for both old and unmerged data items to characterize the data staleness perceived by clients depending on the convergence conflict handling mechanism implemented. Under the last-writer-wins rule, a client sees stale data if the returned version of the requested item is old. With more complex convergent conflict handling mechanisms, instead, a client sees stale data if the returned item version has not been merged with all the concurrent and possibly



(a) Blocking behavior in POCC. (b) Data staleness on Cure*.

Fig. 2. Blocking incidence in POCC and perceived data staleness in Cure* (32 partitions, 32:1 GET:PUT workload).

conflicting updates to the corresponding key.

Figure 2b shows the percentage of old data items returned in Cure* and the number of fresher versions available in the version chain of an old returned data item. The figure also shows the percentage of unmerged data items returned in Cure* and the number of unmerged versions available in the version chain of an unmerged returned data item. The plot shows that the probability to return stale data increases with the load. This is not only because of a higher remote update rates. It is also because higher contention on physical resources slows down the execution of the stabilization protocol needed to identify stable versions. In fact at high load, but before the saturation point, the percentage of returned old, resp. unmerged, items almost reaches 15%, resp. 10%. When Cure* is highly loaded, the percentage of returned stale data items grows as high as 30%.

We note that in bigger deployments the execution of the stabilization protocol would naturally progress at a lower pace, with a commensurate increase in the perceived staleness. In addition, these results correspond to running the stabilization protocol every 5 milliseconds. Higher values would allow the system to reach a higher throughput (as shown in previous work [13]), but would come at the cost of an increased data staleness. By contrast, POCC is immune to this trade-off.

Summary of the results. Figure 1b and Figure 2 reveal that POCC is competitive with Cure* even under high load, when POCC is prone to block more frequently and for a longer amount of time. POCC achieves this result because blocking is a rare event, thus not hurting the the vast majority of operations, and because a blocked operation yields the CPU, thus increasing the amount of computational resources available to quickly serve non-blocked operations. In Cure*, instead, data staleness yields to a higher CPU demand to traverse the version chain upon serving a GET operation and all the requests concurrently compete for the CPU, mutually hurting each other's performance.

C. Transactional workload

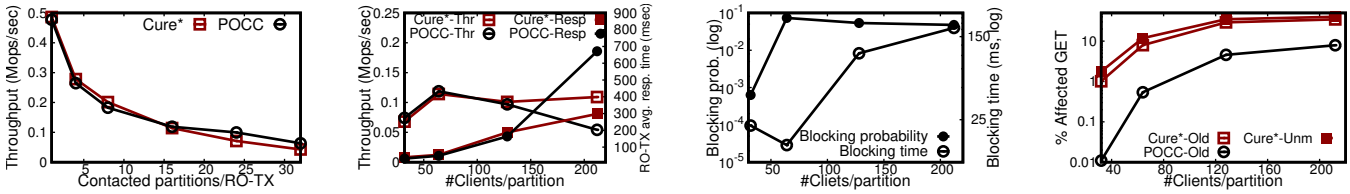
We now assess the performance of POCC when facing a workload composed of PUT operations and transactional reads. To this end, we run on 32 partitions a workload in which each client first issues a RO-TX to read p items corresponding to p distinct partitions, and then performs a random PUT. p is varied from 1 to 32.

Transactional scalability. Figure 3a shows the throughput achieved by POCC and Cure* while increasing the number of partitions involved in a read-only transaction. The plot shows that the performance of POCC and Cure* are comparable – with POCC being slightly better in general– when the number of involved partitions is small. The gain of POCC over Cure* becomes more noticeable (up to 15%) when a transactions involves the majority of the partitions in the system. This is because, as the number of involved partition increases, executing a transaction becomes more resource demanding. POCC, then, achieves a higher throughput because it is more resource efficient than Cure*.

Throughput and response time. We now investigate the throughput and response time dynamics while increasing the number of active clients. Figure 3b depicts the throughput and average RO-TX response time achieved by POCC and Cure* when transactions involve half of the partitions in the system. The plot reveals that, despite reaching the same maximum throughput, the two systems exhibit very different performance dynamics. The throughput of POCC, in fact, drops after having reached its maximum value. This is caused by a surge in the average response time of read-only transactions. By contrast, the throughput of Cure* plateaus after having reached its maximum value, and its average RO-TX response time steadily increases with the number of active clients. The difference in the behavior exhibited by the systems is due to the blocking dynamics of POCC, as we explain in the next paragraph.

Blocking behavior. Figure 3c shows the probability that a PUT or a transactional read stalls on a server and the average amount of time a stalled request is delayed. The reported results correspond to the same workload as in the previous paragraph. The plot shows highly non-linear dynamics. The operation blocking probability peaks in correspondence of 64 active threads per partition, which corresponds to the peak throughput. In correspondence of lower throughputs, the blocking probability decreases because of the lower update rate in the system. The blocking time of stalled operations instead, decreases from 32 to 64 clients and then quickly grows. We argue that the rationale behind this dynamic is as follows. At low throughput, there is also a low rate of updates. Hence, provided that an operation blocks on a server p, p is expected to wait a relatively long time before receiving the update or the heartbeat that unblocks the pending operation. In correspondence of 64 clients per partition, the throughput is high enough to reduce such waiting time. When the number of clients grows too high, performance and blocking dynamics are mainly determined by the high contention on physical resources. This causes the delayed processing of updates and heartbeats messages, yielding to very high blocking times (and blocking probability). This, on its turn, yield the throughput to plummet.

Data staleness. As already explained in Section III-A, with transactions, OCC does not in general return the freshest version of a key. Therefore, both POCC and Cure* are prone



(a) Throughput while varying number of contacted partitions per transaction.

(b) Throughput and avg. resp. time with different # clients per partition.

(c) Blocking behavior in POCC with different # clients per partition.

(d) Data staleness in POCC and Cure* with different # clients per partition.

Fig. 3. Comparison of POCC and Cure* with a workload composed of PUT and RO-TX operations. The systems are deployed on 32 partitions per DC.

to return stale data. In this paragraph we compare the staleness of data returned to client by POCC and Cure*. In particular, Figure 3d reports the percentage of old data items returned by the two systems and of unmerged items returned by Cure*. The plot does not report statistics for unmerged items in POCC. In POCC, in fact, all the item versions behind the returned one in the version chain have already been merged. Therefore, in the context of transactional reads in POCC, the definitions of old and unmerged item coincide.

The plot shows that the staleness exhibited by POCC is two orders of magnitude lower than Cure*'s. This is because POCC defines the boundaries of items visible within a transaction on the basis of the issuing client's history and the set of items *received* by the transaction coordinator node. Cure*, instead, defines such boundaries in terms of items that are *stable* on the transaction coordinator node, thus being much more prone to read old values.

Summary of the results. We have shown that OCC is able to achieve higher performance than a pessimistic design in the case of workloads composed of PUT and RO-TX operations. The cause for this performance increase is mainly the higher resource efficiency enabled by OCC. Transactions are, in fact, more demanding both in terms of computational resources and communication. Therefore, with a workload composed of transactions, POCC higher resource efficiency pays off more than the case of a workload consisting of simple GET operations. Our results also confirm the capability of OCC to deliver much fresher data to clients than what is made possible by a pessimistic design.

VI. Related work

Our proposal is primarily related to the vast literature on causally consistent systems, which include Eiger [9], Bolt-on causal consistency [12], ChainReaction [11], Orbe [10], GentleRain [13], SwiftCloud [14] and Cure [15]. These systems differ in the mechanism they employ to achieve causal consistency. Some of them use logical clocks and rely on the exchange of dependency messages to determine the visibility of remote updates. Dependencies are tracked at the item [8], operation [9] or partition [10] granularity. Other systems use physical clocks [13], [15] and run a periodic stabilization protocol to identify stable items. The common trait of these systems is that they are pessimistic, i.e., a remote data item becomes locally visible only when all its dependencies have been received in the local data center. OCC differs from this approach because it makes any update visible as soon as it is received by a server. The primary aim of OCC are maximizing data freshness and alleviating the overhead incurred by dependency checking and stabilization protocols.

OCC is also related to the literature on *speculative* systems. Speculation consists of optimistically processing an operation even if previous operations have not been fully completed yet and only their tentative result is available. If the tentative result differs from the later final one, the system rollbacks to a consistent state. Speculation has been investigated in a variety of contexts, including processor architectures [41], operating systems [42], simulators [43], concurrency control schemes for transactional systems [44], and recent systems that can concurrently support different consistency levels [19]. The primary aim of speculation is to reduce operation response times by avoiding to wait for the expensive computation of the final value of an operation. OCC, instead, embraces an optimistic approach to maximize the freshness of data returned to client and to reduce the overhead for dependency checking/stabilization. As we have shown, however, in many cases the better resource efficiency enabled by OCC can also yield performance gains in terms both of response times and achievable throughput.

VII. Conclusion and Future Work

We have presented OCC, an optimistic approach to achieve causal consistency in geo-replicated key-value stores. OCC regards network partitions and data center failures as rare events in modern deployments. Therefore, OCC trades some of the availability properties achievable by causal consistency for a higher freshness in the data returned to clients and a higher resource efficiency.

We have implemented OCC in a system named POCC. We have shown that POCC can achieve performance that is comparable or better than its "pessimistic" counterpart in a wide range of workloads, while being able to reduce (or eliminate) the staleness of the data returned to clients.

In this paper, we have focused on assessing the benefits of OCC during normal operational behavior, i.e., in the absence of network partitions. As future work, we plan to quantitatively assess the performance and behavior of POCC in presence of network partitions and full data center failures.

REFERENCES

[1] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[2] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proc. of NSDI*, 2013.

[3] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, and R. H. Campbell, "Ambry: Linkedin's scalable geo-distributed object store," in *Proc. of SIGMOD*, 2016.

[4] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[5] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proc. of PODC*, 2000.

[6] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009.

[7] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[8] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proc. of SOSP*, 2011.

[9] ——, "Stronger semantics for low-latency geo-replicated storage," in *Proc. of NSDI*, 2013.

[10] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proc. of SoCC*, 2013.

[11] S. Almeida, J. a. Leitão, and L. Rodrigues, "Chainreaction: A causal+ consistent datastore based on chain replication," in *Proc. of EuroSys*, 2013.

[12] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proc. of SIGMOD*, 2013.

[13] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *Proc. of SoCC*, 2014.

[14] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro, "Write fast, read in the past: Causal consistency for client-side applications," in *Proc. of Middleware*, 2015.

[15] D. D. Akkoorath, A. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *Proc. of ICDCS*, 2016.

[16] 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.

[17] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 776–787, Apr. 2012.

[18] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd, "Existential consistency: Measuring and understanding consistency at facebook," in *Proc. of SOSP*, 2015.

[19] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi, "Incremental consistency guarantees for replicated objects," in *Proc. of OSDI*, 2016.

[20] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proc. of SOSP*, 2011.

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

[22] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, "F10: A fault-tolerant engineered network," in *Proc. of NSDI*, 2013.

[23] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[24] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180–209, Jun. 1979.

[25] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie, "What consistency does your key-value store actually provide?" in *Proc. of HotDep*, 2010.

[26] P. Bailis and K. Kingsbury, "The network is reliable," *Queue*, vol. 12, no. 7, pp. 20:20–20:32, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2639988.2639988

[27] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 181–192, Nov. 2013. [Online]. Available: http://dx.doi.org/10.14778/2732232.2732237

[28] "NTP: The network time protocol," http://www.ntp.org.

[29] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of mutual inconsistency in distributed systems," *IEEE Trans. Softw. Eng.*, vol. 9, no. 3, pp. 240–247, May 1983.

[30] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proc. of PODC*, 1988.

[31] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[32] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proc. of OSDI*, 2004.

[33] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. of SIGMETRICS*, 2012.

[34] "How much text versus metadata is in a tweet?" http://goo.gl/EBFIFs.

[35] "Storing hundreds of millions of simple key-value pairs in redis," http://goo.gl/ieeU17.

[36] D. F. García and J. García, "Tpc-w e-commerce benchmark evaluation," *Computer*, vol. 36, no. 2, 2003.

[37] D. R. Llanos and B. Palop, "Tpcc-uva: an open-source tpc-c implementation for parallel and distributed systems," in *Proc. of IPDPS*, 2006.

[38] R. Hariharan and N. Sun, "Workload characterization of specweb2005," in *SPEC Benchmark Workshop*, 2006.

[39] S. Elnikety, S. Dropsho, E. Cecchet, and W. Zwaenepoel, "Predicting replicated database scalability from standalone database profiling," in *Proc. of EuroSys*, 2009.

[40] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebook's distributed data store for the social graph," in *Proc. of ATC*, 2013.

[41] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., 2011.

[42] E. B. Nightingale, P. M. Chen, and J. Flinn, "Speculative execution in a distributed file system," *ACM TOCS*, vol. 24, no. 4, Nov. 2006.

[43] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto, "Time warp operating system," in *Proc. of SOSP*, 1987.

[44] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM TODS*, vol. 6, no. 2, Jun. 1981.

APPENDIX: CORRECTNESS OF POCC

We now provide an informal proof sketch that POCC provides causal consistency. Namely, we show that an item returned to a GET operation never violates causal consistency, and that the set returned by a RO-TX is a causal snapshot according to the definition provided in Section II.

We start by proving two Propositions that we will use to build the correctness proof.

Proposition 1. *Let X, Y be two data items such that $X \rightsquigarrow Y$. Then, $Y.DV[X.sr] \geq X.ut$.*

Proof. This invariant stems from the fact that a client c tracks the dependencies established by any read/written item and stores such dependencies with the data items it writes. Specifically, upon reading an item d, c transitively tracks d's dependencies by computing the entry-wise maximum between DV_c and the $d.DV$, and storing the result as new DV_c.

Further, upon reading/writing an item d, noting i the source replica of d, c tracks the direct dependency on d by setting $DV_c[i] = max\{DV_c[i], d.ut\}$. Finally, when writing a data item d, c stores DV_c with d. ■

Proposition 2. *Let X, Y be two data items such that $X \rightsquigarrow Y$. Then, $X.ut < Y.ut$.*

Proof. A client c that has established a dependency towards X is such that $DV_c[X.sr] \geq X.ut$. When c tries to write Y on p_n^m, Algorithm 2 Line 7 enforces that $Clock_n^m$ is strictly higher than the maximum value in DV_c. Therefore, because $Clock_n^m$ is used as update timestamp for Y, it is $Y.ut > X.ut$. ■

We now show that POCC delivers causal consistency.

Proposition 3. Causally consistent GET operation. *Let client c read Y such that $X \rightsquigarrow Y$. Then, if c later issues a $GET(x)$ operation, c reads X' such that either $X' = X$ or $X' \not\rightsquigarrow X$.*

Proof. The proposition is verified because upon processing a *GET* operation issued by c for a key x, a node p_n^m waits until it is positive to return a version of x that complies with c's history. We prove this by contradiction, by assuming that there is a $X_0 \rightsquigarrow X \rightsquigarrow Y$ and showing that if p_n^m returns X_0, then it has not respected the protocol of POCC.

Because $X_0 \rightsquigarrow X$, it is, by virtue of Proposition 2, $X_0.ut < X.ut$. In addition, because $X \rightsquigarrow Y$, Proposition 1 enforces that $DV_c[X.sr] \geq X.ut$. If p_n^m returns X_0 it means that it has not received X yet. If p_n^m had received X, it would have returned it because X has a higher timestamp than X_0 and POCC returns the version of a key with the highest timestamp. To return X, however, p_n^m must pass the waiting condition in Algorithm 2 Line 2. In particular, because $DV_c[X.sr] \geq X.ut \geq X_0.ut$, it must be that $VV_n^m[X.sr] \geq X.ut$. This, however, is impossible because p_n^m has not received X and updates and heartbeats are sent by nodes in timestamp order.

■

Proposition 4. Causally consistent RO-TX operation. *The result of a RO-TX issued by a client c represents a causally consistent snapshot and is causally consistent with the history of operations issued by c.*

Proof. We first show that the data returned to c are causally consistent with c's history. This stems from the fact that the snapshot vector corresponding to a read-only transaction is computed as the entry-wise maximum between the version vector of the transaction coordinator and the read dependency vector of the client (Algorithm 2 Line 32). This means that the snapshot includes every item read or written by c, and any transitive dependency induced by such items. Before serving a transactional read, a server p_n^m waits until its vector clock is at least entry-wise equal to the snapshot vector (Algorithm 2 Line 40). Therefore, for the same reasons presented for the GET operation case, POCC enforces that p_n^m does not return any item that is not causally consistent with c's history.

We now show that a RO-TX operation returns a causal snapshot. Namely, assume that $X \rightsquigarrow X' \rightsquigarrow Y$ and that c reads x and y in the same transaction. Then, if the returned snapshot contains Y, it also contains X'.

By contradiction, assume that the returned snapshot includes Y and X. Because $X \rightsquigarrow X' \rightsquigarrow Y$, it stems from Proposition 1 that $Y.DV[X'.sr] > X'.ut$ and $Y.DV[X.sr] \geq X.ut$. In addition, since Y is in the returned snapshot, it follows that $TV \geq Y.DV$ (Algorithm 2 Line 43). Therefore, it is $SV[X'.sr] \geq Y[X'.sr] > X'.ut$ and $SV[X.sr] \geq Y[X.sr] > X.ut$. Then, by virtue of the blocking condition in Algorithm 2 Line 40, the server replicating x, noted p_x has received both X and X' by the time it serves the RO-TX request from the client. Then, p_x cannot return X instead of X' because *i*) $X \rightsquigarrow X'$ and Proposition 2 enforces that $X.ut < X'.ut$ and *ii*) p_x returns the item in the version chain with the highest timestamp (provided that its dependency vector is entry-wise smaller than or equal to TV). ■