Optimistic Causal Consistency for Geo-Replicated Key-Value Stores

Kristina Spirovska, Diego Didona, and Willy Zwaenepoel, *Fellow, IEEE*

Abstract—Causal consistency (CC) is an attractive consistency model for geo-replicated data stores because it hits a sweet spot in the ease-of-programming versus performance trade-off. We present a new approach for implementing CC in geo-replicated data stores, which we call Optimistic Causal Consistency (OCC). OCC's main design goal is to maximize data freshness. The optimism in our approach lies in the fact that the updates replicated to a remote data center are made visible immediately, without checking if their causal dependencies have been received. Servers perform the dependency check needed to enforce CC only upon serving a client operation, rather than on receipt of a replicated data item as in existing systems. OCC offers a significant gain in data freshness, which is of crucial importance for various types of applications, such as real-time systems. OCC's potentially blocking behavior makes it vulnerable to network partitions. We therefore propose a recovery mechanism that allows an OCC system to fall back on a pessimistic protocol to continue operating during network partitions. We implement POCC, the first causally consistent geo-replicated multi-master key-value data store designed to maximize data freshness. We show that POCC improves data freshness, while offering comparable or better performance than its pessimistic counterparts.

Index Terms—Optimistic causal consistency, causal consistency, geo-replication, key-value data stores, read-only transactions, data freshness

1 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 DCs to deliver lower response times to geographically distributed clients and to achieve fault tolerance [1], [2], [3].

A critical decision when designing a geo-replicated data platform is the choice of consistency model. At the 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 the anomalies it allows lead to complexities in application development.

Causal Consistency. Causal consistency [7] lies between these two endpoints and is an attractive model for building geo-replicated data stores because it hits a sweet spot in this tradeoff between ease of programming and performance [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]. On the one hand, it avoids the long latencies and the inability to tolerate network partitions of strong consistency. On the other hand, it is easy to reason about and avoids some of the anomalies allowed under eventual consistency.

Existing systems employ different techniques to achieve causal consistency, but, with one exception [19], developed at the same time as this work was originally published, 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.

This invariant allows such data stores to tolerate network partitions and DC failures. It requires, however, the implementation of dependency checking [8], [9] or stabilization protocols [13], [18] 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. Besides increased data visibility latency, these dependency checking and stabilization protocols can cause severe slowdown when one component of the system is lagging behind the others [19]. This problem has been pointed out as one of the barriers to the deployment of causal consistency in production environments [20].

Optimistic Causal Consistency. In this paper we argue the aforementioned protocols are too *pessimistic* for modern data center deployments. To tackle the issues that affect such pessimistic systems, we propose **O**ptimistic **C**ausal **C**onsistency (OCC). With OCC a server always returns the most recent available version of an item, even if some of its causal dependencies have not been replicated locally. The system keeps sufficient state to detect and block a later read on such a causal dependency, until it is locally replicated. This state is maintained by the CALDeR (**C**lient-**A**ssisted **L**azy **De**pendency **R**esolution) protocol, a novel dependency tracking protocol that relieves the data store from the

- *Kristina Spirovska is with École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland. E-mail: tina.spirovska@gmail.com.*
- *Diego Didona is with IBM Research Europe, 8803 Rüschlikon, Switzerland. E-mail: ddi@zurich.ibm.com.*
- *Willy Zwaenepoel is with the University of Sydney, Camperdown, NSW 2006, Australia. E-mail: willy.zwaenepoel@epfl.ch.*

Manuscript received 23 Oct. 2019; revised 4 Sept. 2020; accepted 15 Sept. 2020. Date of publication 25 Sept. 2020; date of current version 8 Oct. 2020. (Corresponding author: Kristina Spirovska.)
Recommended for acceptance by R. Tolosana.
Digital Object Identifier no. 10.1109/TPDS.2020.3026778

responsibility of tracking the delivery of updates by shifting it to the client. CALDeR eliminates the need for synchronization among partitions, which in turn diminishes the problems caused by slower partitions that are lagging behind.

The main goal of OCC is to maximize data freshness. Data freshness represents a desirable property for all types of systems. Until recently, it had been omitted from the primary design goals of causally consistent systems, since it makes enforcing causal consistency a challenge. However, with the proliferation of real-time systems, IoT devices, high-frequency trading systems, collaborative systems and massively multi-player online games, data freshness becomes a critical design goal [21], [22], [23].

Contributions. We make the following contributions.

i We introduce Optimistic Causal Consistency, discussing its benefits and inherent performance versus availability trade-offs.

ii We present the design and implementation of POCC (**O**ptimistic **C**ausal **C**onsistency with **P**hysical clocks), a causally consistent protocol that implements OCC. POCC relies on physical clocks and dependency vectors to keep track of causal dependencies.

iii We discuss the trade-off between data freshness, performance and availability of OCC. Furthermore, we discuss a highly available version of POCC that can fall back on a pessimistic approach to remain available during network partitions.

iv We assess the benefits brought by OCC by comparing POCC with a (pessimistic) state-of-the-art design for causal consistency based on physical vector clocks. We run heterogeneous workloads on a large-scale infrastructure on Amazon, comprising up to 96 servers scattered across 3 geo-distributed sites. Our results indicate that optimistic causal consistency is effective in reducing the staleness of data returned to clients and delivers performance that is competitive or better than its pessimistic counterpart for a wide range of realistic workloads.

Roadmap. The remainder of the paper is structured as follows. Section 2 describes the motivation behind OCC. Section 3 describes causal consistency and the target system model. Section 4 presents the approach underlying OCC. Section 5 describes the protocols of POCC. Section 6 discusses the correctness of POCC. Section 7 introduces the highly available version of OCC. Section 8 provides the evaluation of POCC. Section 9 discusses related work. Section 10 concludes the paper and identifies future work.

2 Motivation

Geo-replicated systems that enforce causal consistency using a pessimistic approach can suffer from *performance degradation* and *increased data visibility latency* under high load and when co-located with tenants with high resource demands. Under these conditions, the dependency tracking and stabilization protocols, although seemingly inexpensive, become the main bottleneck of the system.

For example, consider an event ticketing system designed to serve millions of events and event guests. The system is partitioned for scalability and geo-replicated in a multi-master fashion over two data centers for fault tolerance. A very popular artist, with millions of fans, uses this system to sell tickets for a concert and announces that the sale of the tickets will start on a particular date and time. As usually happens with popular events, the tickets are sold out within the first few minutes after the sale has started. Building such a system with eventual consistency is possible, but identifying and addressing all possible anomalies is hard. Hence, a stronger consistency model is desired. Strong consistency is deemed prohibitively expensive in terms of performance. As a result, causal consistency becomes a reasonable choice.

Assume that the data for this particular concert is stored on partition p. A large number of concurrent users, all requesting tickets at nearly the same time, results in high load on p, causing it to lag behind the other servers. In a pessimistic implementation of causal consistency an update can be made visible only after all of its causal dependencies have already been made visible, including those residing on other partitions. Therefore, partitions have to synchronize to track the delivery of updates. As a result, the slowest partition, in this case p, stalls the progress of the whole system. The number of updates that cannot be made visible increases, not just on p, but on all partitions, causing performance degradation on all partitions, and affecting all events hosted on the ticketing system, despite the fact that they are unrelated.

A related issue is the increase of the data visibility latency. Proceeding with the same example, assume that the event ticketing system is deployed on AWS, in the North Virginia (us-east-1) and Ohio (us-east-2) regions. The average one-way latency between these regions is $\sim$5 ms [24]. In existing systems stabilization protocols are typically run every 5-10 ms [13], [15], [16]. Allowing in addition for clock skew and the time to execute the stabilization protocol, it could take up to 15 ms to make updates visible. This causes data visibility latency of up to 20 ms (5 ms for geo-replication + 15 ms for stabilization). So, even in the absence of high load, a client could potentially experience a 4x increase in data visibility latency. The data visibility latency could drastically increase if *any* partition of the data store is facing a heavy load, or if it is co-located with a tenant with high resource demands.

Running a highly available multi-master data store implies having conflicting updates. Considering the high probability of write conflicts for frequently updated items, every additional millisecond of visibility latency increases the chance of conflicts. Relating to our example, having greater visibility latency when calculating the number of remaining concert tickets could cause a higher number of overbooked tickets. If, for simplicity, we assume that each data center initiates ticket reservations at the same constant speed, a 4x increase of visibility latency could result in 4x more overbooked tickets. If instead the system were able to provide fresher data, the number of overbookings could have been kept at the minimum that is achievable without losing availability or performance.

These problems serve as motivation for a different approach to implement causal consistency that is more resilient to load spikes and co-location with tenants with high resource demands, and that at the same time provides fresher data.

The effectiveness of our optimistic approach stems from two main insights. First, recent works have revealed that update replication in data stores exhibits a *naturally consistent order*. Specifically, a data item is typically replicated (and accessed) after its dependencies have already been propagated [25], [26], [27]. Hence, the most recent version of a data item can typically be returned without violating consistency [28]. OCC leverages this insight by immediately returning the freshest version of a data item to a client. If that client later reads a missing dependency, that read is made to wait, but this happens only rarely, because of the naturally consistent order of updates. Hence, OCC maximizes the freshness of data returned to clients.

Second, network partitions are relatively rare events (and complete DC failures even more rare) [26], [29], [30], [31]. OCC leverages this insight by avoiding (most of) the overhead associated with network partition tolerance during normal operating conditions, and by incurring it only when a network partition actually occurs. OCC requires the execution of a stabilization protocol only to allow the system to fall-back on a pessimistic scheme in the presence of a network partition. This design contrasts with existing pessimistic systems, which must frequently execute stabilization protocols or perform ongoing dependency checking.

3 Definitions and System Model

3.1 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 [7], [32]. For two operations $a,\ b$, we say that b causally depends on a, 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 as *stable* in a DC if all the dependencies of X have been replicated in such DC.

3.2 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 are said to *conflict*. Given that updates can be propagated to replicas in different orders, conflicting updates can lead to replicas diverging forever. To enforce that different replicas converge to the same value for any key, a protocol must implement a convergent conflict handling procedure, a commutative and associative function [9], 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 function is the last-writer-wins rule (Thomas' write rule [33]): given two updates, a server can deterministically determine which update is last, and that update determines the value of the item. 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], [18].

3.3 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.

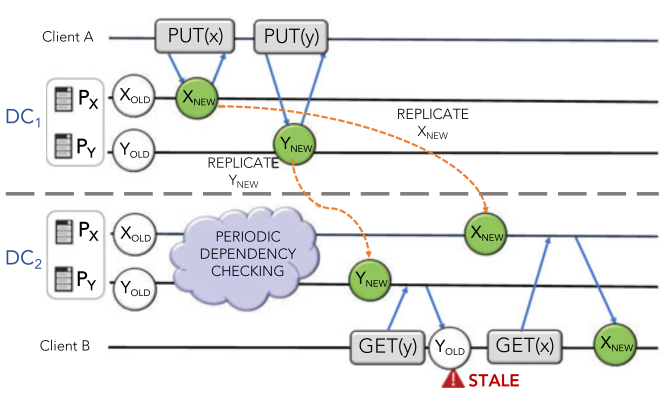
We assume a multiversion data store. An update operation creates a new version of an item. We say that one item version is fresher than another, if it comes after it with respect to the causally consistent order. In addition to the actual value of the key, each version also stores some metadata, in order to track causality. The system periodically garbage-collects old versions of items. We further assume servers in the system can communicate through point-to-point lossless FIFO channels.

At the beginning of a session, a client c connects to a server p in the closest data center 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 p are transparently forwarded to the server(s) responsible for such data items, and the result is relayed back to c by p.

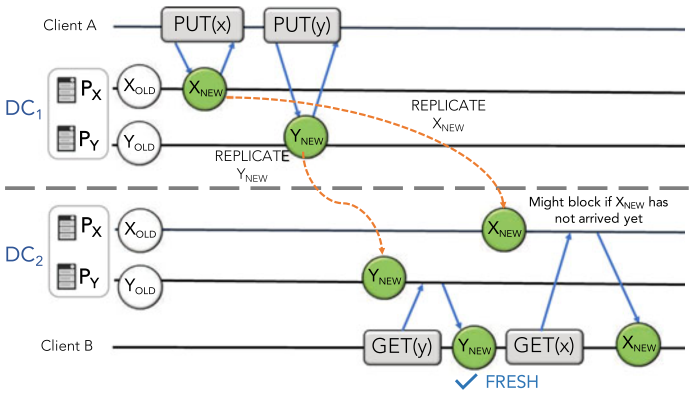
The system provides the following operations to clients:

- 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], that enables the client to read the set of items specified as argument with the following consistency guarantee. 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 $X \rightsquigarrow X' \rightsquigarrow Y$.

Availability. We use the term availability to indicate that a client operation never blocks as the result of a network partition between DCs [5].



(a) Pessimistic case



(b) Optimistic case

Fig. 1. In systems with pessimistic behaviour (a), a stale version of a y, Y_{OLD}, is returned because X_{NEW} on which Y_{NEW} depends has not been yet replicated in DC$_2$. OCC (b), on the other hand, maximizes data freshness by returning the freshest version of y, Y_{NEW}. The trade-off is that OCC would have to block if client B requested X_{NEW} before it had been replicated to DC$_2$.

4 Optimistic Causal Consistency

We first contrast the optimistic approach of OCC with the pessimistic one of existing systems. Then, we explore the trade-off between data freshness, performance and availability. Next, we describe how OCC enforces causal consistency efficiently by shifting the burden of dependency checking to the clients using the CALDeR protocol. We also describe how OCC improves data freshness. Then, we provide a discussion that highlights the rationale behind the design of OCC. Finally, we describe how OCC can be augmented to preserve availability during network partitions.

4.1 The Design of OCC

We describe OCC by means of an example, depicted in Fig. 1, in which we contrast it with existing pessimistic approaches. In our example, the initial versions of x and y are, respectively, X_{OLD} and Y_{OLD}, and are stored by partitions p_x and p_y. Client A in DC$_1$ creates X_{NEW} and Y_{NEW} such that $X_{NEW} \rightsquigarrow Y_{NEW}$. By network asynchrony, however, Y_{NEW} is received in DC$_2$ before X_{NEW}. After Y_{NEW} is received in DC$_2$, client B in DC$_2$ issues a GET operation on y and a subsequent GET on x.

By causal consistency, if client B sees Y_{NEW}, then the subsequent GET on x must return X_{NEW}. Enforcing this behavior is nontrivial because DC$_2$ has received Y_{NEW} but not its causal dependency X_{NEW}. Hence, client B is at risk of observing Y_{NEW} and X_{OLD}, which violates causal consistency.

Existing Pessimistic Approaches. State of the art systems use different methods to tackle this issue. The key common mechanism, however, is that a replicated version is made visible to clients within a DC only when all of the item's dependencies have been received in that DC. With reference to our example, pessimistic approaches enforce that Y_{NEW} can be read by client B in DC$_2$ only if X_{NEW} has also been received and can be read in DC$_2$. This behavior is depicted in Fig. 1a.

On the one hand, this approach allows a system to tolerate network partitions. A data store partition can always return a causally consistent version to a client without waiting for the receipt of a version from a remote DC. On the other hand, this approach is prone to return stale items to the clients, even though fresher versions have been received. In Fig. 1a, p$_y$ in DC$_2$ returns Y_{OLD} to client B, despite the fact that Y_{NEW} has been received already. In addition, pessimistic approaches incur two sources of overhead. First, they need to implement a distributed dependency tracking protocol, to communicate which items have been received in a DC, and hence to determine which versions can be made visible. Such protocol is implemented as a background process, hence it is not on the critical path of serving a client request, but it consumes CPU and network bandwidth. Second, when serving a client request, a partition needs to find the freshest visible version of the requested key, typically by traversing a number of versions of the key received so far by the partition. Performing such a traversal directly impacts the latency of the read operation.

Optimistic Approach. OCC departs from conventional approaches in that it allows a partition in a DC to return a version of an item to a client despite the fact that the version's dependencies have not been received in the DC. The optimism of the approach lies in the assumption that the dependencies of such item will be received in the DC by the time the client wants to access them in subsequent operations.

Fig. 1b shows how OCC behaves in our example. p$_y$ in DC$_2$ returns Y_{NEW} to client B despite the fact that X_{NEW} has not yet been received by DC$_2$. Then, by the time client B reads x, X_{NEW} has been received in DC$_2$ and can be returned, preserving causal consistency. If X_{NEW} had not yet been received, then the operation would have to wait for that to happen.

The clearest benefit of OCC is that it increases the freshness of the data returned to clients. Not only does OCC improve the user experience, but it also has a positive effect on performance. First, OCC does not impose the overhead of running protocols for tracking the delivery of dependencies. OCC relieves the data store of this responsibility and pushes it to the client, by storing inexpensive meta-data for detection of missing dependencies in the client. Second, OCC improves the response time by not having to traverse the item version chain.

4.2 Trade-Offs

OCC's potentially blocking behavior makes it vulnerable to network partitions, since a GET operation of a client can block. With reference to our example illustrated in Fig. 1b, if client B reads x before X_{NEW} is received by DC$_2$, then, in order to preserve causal consistency, p$_x$ in DC$_2$ needs to wait for the receipt of X_{NEW} before it can respond to the GET. Furthermore, if X_{NEW} cannot be replicated in DC$_2$ due to a network partition, since client B has already established a dependency on X_{NEW} by reading Y_{NEW}, p_x in DC$_2$ must block the GET on x until the network partition heals and it receives X_{NEW}.

In such a case, the client is given the flexibility to choose between three options. First, it can wait for the network partition to heal. Next, it can choose to reconnect to another data center. This may or may not solve the problem, because that data center could be missing X_{NEW} as well. Finally, it can continue execution with eventual consistency until the problem is resolved. In other words, OCC offers the client a choice between availability and consistency, depending on the use case. We further address this issue by presenting a highly available OCC design, described in Section 7.

Existing pessimistic causally consistent protocols seamlessly tolerate network partitions and full DC failures, which can be modeled as unhealed network partitions. OCC trades this ability for better freshness in the data returned to the client and greater resource efficiency.

4.3 Client-Assisted Lazy Dependency Resolution

The challenge in OCC is to enforce causal consistency while exposing unstable item versions, and doing so in a cost-effective way, that, simultaneously, reduces both the meta-data overhead and the read response time. OCC achieves this by means of the CALDeR protocol. The main idea behind the CALDeR protocol is to empower the client with the ability to track whether a read operation is safe, and, hence, relieve the data store from the burden of running distributed protocols to track the delivery of dependencies.

Clients store information about the causal dependencies established when performing reads in the form of *client-side dependency meta-data*. If c reads Y and there exists an X such that $X \rightsquigarrow Y$, c records that 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), client-side dependency meta-data 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 is compatible with c's history, and it is returned to c. If not, then p stalls c's request until it receives X.

For a PUT operation, the client-side dependency meta-data 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 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.

4.4 Discussion

OCC presents a different way to implement causally consistent systems by offering better data freshness and higher resource efficiency than the existing state-of-the art systems. The trade-off is that it is potentially blocking. The effectiveness of OCC relies on the insight that blocking is a rare event because 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. These claims are backed by empirical evidence gathered on popular commercial cloud data stores [25], [28], [34], including a massive deployment such as Facebook's [27], and are also confirmed by analytical models [26]. Furthermore, our experimental results in Section 8 show that OCC has low blocking probability.

5 Implementation

We first give a high level overview of OCC's protocols for the GET operation and the RO-TX transaction. Then, we present POCC, a scalable implementation of OCC with physical clocks.

5.1 Overview

The primary design goal of OCC is improving the freshness of data returned to the client. Doing so while preserving causal consistency requires the implementation of different protocols depending on whether the client performs a GET operation or a RO-TX. In the remainder of this subsection, we show how OCC improves data freshness in both cases, when serving a GET operation or when executing a RO-TX.

OCC can be implemented with any dependency tracking mechanism that has been proposed in literature, e.g., dependency lists [8], dependency matrices [10], Hybrid Logical Physical Clocks (HLC) [35], physical scalar clocks [13], physical vector clocks [18]. Because the client has to supply dependency information for each operation, however, it is of paramount importance to encode dependencies in a succinct fashion. As we show in Section 5.2, POCC, our implementation of OCC, meets this requirement by relying on physical vector clocks, which can keep track of dependencies with an overhead that is only linear with the number of data centers in the system.

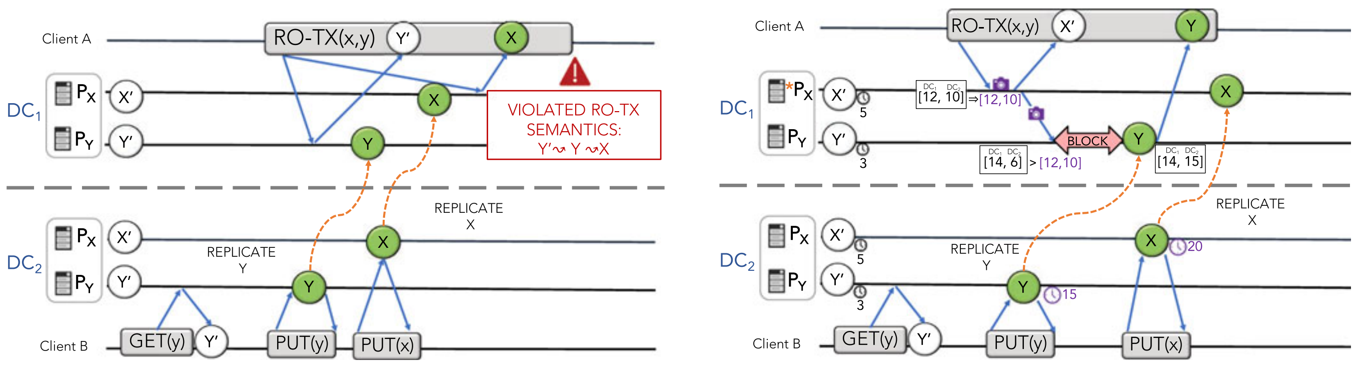
5.1.1 GET Operations

When a server p receives a GET(x) request from a client c, p returns the freshest version that it stores, provided that such version is fresh enough and does not violate causal consistency.

Assume that c has established a dependency towards X_{NEW}, e.g., by reading Y_{NEW}, with $X_{NEW} \rightsquigarrow Y_{NEW}$. If the freshest version of x stored by p is X^*, an even fresher version than X_{NEW}, then p can return X^* to c. If X_{NEW}, instead, is fresher than X^*, then p must block the client operation until p receives X_{NEW}.

5.1.2 Read-Only Transactions

Implementing RO-TX in an optimistic fashion is more challenging, because all the reads that are performed within a RO-TX must be causally consistent with the history of the



(a) Naive OCC RO-TX implementation

(b) Correct OCC RO-TX implementation

Fig. 2. Naive (incorrect) OCC implementation: naively maximizing data freshness in RO-TX by always returning the freshest present item versions could result in violation of the RO-TX semantics (a). Correct OCC implementation (b).

client and also among themselves. The complexity of RO-TX is further exacerbated by the fact that read operations on individual items part of the argument set take place in parallel on different partitions. Hence, a partition involved in an RO-TX does not know which keys and which versions are returned by other partitions involved in the same RO-TX. For these reasons, a partition that processes a RO-TX cannot simply return the freshest version of a requested key. Doing so may lead to a consistency violation, as we show with an example in Fig. 2.

In DC$_2$, client B does a series of operations by which it creates the following causal relation between different versions of items x and y: $Y' \rightsquigarrow Y \rightsquigarrow X$. In DC$_1$, client A, which has not established any dependencies so far, does a RO-TX involving keys x and y. When the read request for y gets to p_y, the freshest version of y present in p_y is Y'. When p_x gets the read request for x, the freshest present version of x is X.

If the partitions in DC$_1$ were to serve the reads within client A's RO-TX in the same fashion as two individual GET operations, then the RO-TX would return Y' and X. Such return values are not causally consistent, because there is Y such that $Y' \rightsquigarrow Y \rightsquigarrow X$. To avoid this erroneous behaviour, and at the same time to improve data freshness, OCC leverages the concept of a causal snapshot. A causal snapshot is a set of item versions such that all causal dependencies of those versions are also included in the snapshot. Upon starting, an RO-TX is assigned a causal snapshot, and, for each key in the RO-TX, each partition returns the freshest version of that key that belongs to the snapshot.

In OCC, the boundaries of a causal snapshot are defined on the basis of the items currently received by servers 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.

Fig. 2b shows how OCC implements RO-TX in the scenario represented in Fig. 2a. When c_A in DC_1 starts the RO-TX to read x and y, it first contacts the transaction coordinator, p_x, that orchestrates the RO-TX transaction and defines the boundaries of the RO-TX transaction snapshot. Since c_A has not established any dependencies yet, the snapshot is determined by the latest items received by p_x, or in their absence, the latest received heartbeat messages. In this example, the snapshot is encoded with the dependency vector [12, 10], which indicates that the snapshot includes all items that originated in DC_1 with timestamp lower or equal to 12 and all items that originated in DC_2 with timestamp lower or equal to 10. Next, p_x forwards the read request for y, together with the transaction snapshot, to p_y and returns X' to the c_A as X' is the latest received version of x that belongs to the snapshot. When p_y receives the read request, it has to wait until the latest item received from DC_2, which currently has the value 6, surpasses the snapshot value 10. After receiving Y, which is the latest value replicated from DC_2, p_y returns Y, as the freshest value of item y that belongs to the transaction snapshot.

5.2 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 [36]. 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 ith entry of the vector is t, the update is marked as *potentially* dependent on *every* item originated from the ith DC with timestamp lower than t. Dependency vectors represent a trade-off between metadata efficiency and dependency tracking granularity. On the one hand, they allow POCC to succinctly track dependencies. On the other hand, they might cause a client's request to be (needlessly) stalled because of a *potentially* unresolved dependency that does not correspond to any real dependency.

We now describe in more detail the meta-data stored and the protocols implemented by clients and servers in POCC.

5.3 Meta-Data

Item. An item version d is represented as a tuple $\langle k, v, sr, ut, dv \rangle$. k is the unique id that identifies the item 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 DC 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, assigned at 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 ith DC and $ii)$ d potentially depends on d'.

Algorithm 1. POCC: Operations at Client c

```
1: function GET(key $k$)
2:    send $\langle \textbf{GETReq } k, DV_c \rangle$ to $p_n^m$
3:    receive $\langle \textbf{GETReply } v, ut, DV, sr \rangle$ from $p_n^m$
4:    $DV_c \leftarrow max\{DV_c, DV\}$
5:    $DV_c[sr] \leftarrow max\{DV_c[sr], ut\}$ ▷ Update client's dependencies
6:    return v
7: end function
8: function PUT(key $k$, value $v$)
9:    send $\langle \textbf{PUTReq } k, v, DV_c \rangle$ to $p_n^m$
10:   receive $\langle \textbf{PUTReply } ut \rangle$ from $p_n^m$
11:   $DV_c[m] \leftarrow max\{DV_c[m], ut\}$ ▷ Update client's dependency at local replica
12: end function
13: function RO-TX(key-set $\chi$)
14:   send $\langle \textbf{RO-TX-Req } \chi, DV_c \rangle$ to $p_m^n$
15:   receive $\langle \textbf{RO-TX-Resp } D \rangle$
16:   for ($d \in D$) do
17:      read $d$ as if it was the result of a GET
18:   end for
19: end function
```

Client. A client c maintains during its session a dependency vector DV_c consisting of M entries, where M is the number of DCs. $DV_c[i]$ is the update time of the item d with the highest timestamp such that $i)$ d has been originated in the ith DC and $ii)$ c has established a potential dependency on d. This vector is stored together with any item c writes.

Server. A server p_n^m is identified by the partition id (n) and the DC id (m). In our description, m is the local DC and n is the local partition in the local DC of the server. p_n^m maintains a version vector VV_n^m [7], [37] 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 from 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.

Each server has access to monotonically increasing clock, $Clock_n^m$. Clocks are loosely synchronized by a time synchronization protocol such as NTP [36].

5.4 Operations

We now describe how servers running POCC support GET, PUT and RO-TX operations issued by clients, replicate updates and exchange heartbeats. Algorithms 1 and 2 show the pseudo-code of the protocol running at the client and server side, respectively. Algorithm 3 shows the pseudo-code for the creation and replication of updates and for sending heartbeats.

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 is the partition with which c has established a session or the partition to which the request has been forwarded, as described in Section 3. Concurrent updates to the same key are handled by means of the last-writer-wins rule (see also Section 3) . 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).

Algorithm 2. POCC: Operations at Server p_n^m

```
1: upon receive $\langle \textbf{GETReq } k, DV_c \rangle$ from c do
      ▷ Ensure $p_n^m$'s state is consistent with $c$'s history
2:    wait until $VV_n^m[i] \geq DV_c[i],\ i = 0 \ldots M-1, i \neq m$
3:    $d = argmax_{d.ut}\{d : d.k == k\}$ ▷ Version of $k$ with highest timestamp
4:    send $\langle \textbf{GETReply } d.v, d.ut, d.DV, d.sr \rangle$ to c
5: upon receive $\langle \textbf{PUTReq } k, v, DV_c \rangle$ from c do
6:    wait until $max\{DV_c\} < Clock_n^m$ ▷ Deal with clock skew
7:    $d \leftarrow \langle k, v, m, Clock_n^m, DV_c \rangle$
8:    update($d$) ▷ Create new item and store it in the chain
9:    $VV_n^m[m] \leftarrow$ d.ut ▷ Update version vector
10:   for each server $p_n^j$, $j \in \{0 \ldots M-1\}, j \neq m$ do
11:      send $\langle \textbf{REPLICATE d} \rangle$ to $p_n^j$ ▷ Replicate d
12:   end for
13:   send $\langle \textbf{PUTReply } d.ut \rangle$ to c
14: upon receive $\langle \textbf{RO-TX-Req } \chi, DV_c \rangle$ from c do
15:   $D = \emptyset$ ▷ D represents the set of key-value pairs for the requested keys
16:   $\chi_i = \{k \in \chi : partition(k) == i\}$ ▷ Set of requested keys per partition
17:   $TV = max\{VV_n^m, DV_c\}$ ▷ Timestamp vector of the transaction snapshot
18:   for ($i : \chi_i \neq \emptyset$) do
19:      send $\langle \textbf{SliceREQ } \chi_i, TV \rangle$ to $p_i^m$
20:      receive $\langle \textbf{SliceRESP } D_i \rangle$ from $p_i^m$
21:      $D \leftarrow D \cup D_i$
22:   end for
23:   send $\langle \textbf{RO-TX-Resp } D \rangle$ to c
24: upon receive $\langle \textbf{SliceREQ } \chi, TV \rangle$ from $p_i^m$ do
25:   wait until $VV_n^m \geq TV$ ▷ Ensure $p_n^m$ installed all updates in the snapshot
26:   $D = \emptyset$ ▷ D represents the set of key-value pairs for the requested keys
27:   for ($k \in \chi$) do
28:      $D_k = \{d : d.k == k \wedge d.DV \leq TV\}$ ▷ Compute visible versions set
29:      $D \leftarrow D \cup \{argmax_{d.ut}\{d \in D_k\}\}$ ▷ Freshest visible version of k
30:   end for
31:   send $\langle \textbf{SliceRESP } D \rangle$ to $p_i^m$
```

GET Operation. Client c sends a request $\langle$GET k, $DV_c\rangle$, 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 DV_c (Algorithm 2 Line 2). This check is not done for the mth entry because dependencies on local items are always trivially satisfied. If the check succeeds, it means that p_n^m has received all the items of the nth 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 (Algorithm 2 Line 3). If at least one entry of VV_n^m is smaller than DV_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 $DV_c[i] \leq VV_n^m[i], i \neq m$.

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

PUT Operation. c sends a request $\langle$PUT $k, v, DV_c\rangle$, where k is the key of the item to be written, v is the desired value to associate with k and DV_c is the client's dependency vector.

Upon receiving the PUT request, p_n^m, first waits until the local physical clock is higher than the maximum timestamp in DV_c (Algorithm 2 Line 6). 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.

Next, p_n^m creates a new version d of k, and inserts d in the version chain corresponding to k. Afterwards, p_n^m replicates the newly created item version d to the remote replicas. Then, p_n^m updates the local entry of its version vector with d's update time. 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 mth entry of DV_c to $d.ut$ to track the newly established dependency.

RO-TX. A client c sends a request $\langle$RO-TX $\chi, DV_c\rangle$ 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 (Algorithm 2 Line 17). As said in Section 4.1, 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 DV_c.

Reads for keys that cannot be served locally are sent in parallel to the corresponding partitions in the form of a request $\langle$SliceReq $\chi, TV\rangle$ (Algorithm 2 Line 19). Upon receiving a SliceReq, a server first waits until its version vector is entry-wise greater than or equal to the transaction snapshot vector TV (Algorithm 2 Line 25). Then, for each key k in the set of keys to be read χ, the server chooses and returns the freshest visible version, which is the version whose dependency vector DV is less or equal than TV (Algorithm 2 Line 28). When each server has returned to the coordinator, for each key, the version within the snapshot with the highest timestamp, p_n^m returns all the requested keys and values to c.

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 mth entry of its local version vector to the update time of d (Algorithm 3 Lines 6 – 7).

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 mth 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 (Algorithm 3 Line 13). 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.

Algorithm 3. POCC: Auxiliary Functions at Server p_n^m

```
 1: function update(d')
 2:   create new item $d : \langle d.k, d.v, d.sr, d.ut, d.dv\rangle \leftarrow \langle d'.k, d'.v, d'.sr, d'.ut, d'.dv\rangle$
 3:   insert item $d$ in the version chain of key $k$
 4: end function
 5: upon receive ⟨REPLICATE d⟩ from $p_n^j$ do
 6:   update($d$)                  ▷ Create new item and store it in the chain
 7:   $VV_n^m[j] \leftarrow d.ut$                  ▷ Update version vector
 8: upon every $\Delta$ time do
 9:   $ct \leftarrow Clock_n^m$
10:   if ($ct \geq VV_n^m[m] + \Delta$) then
11:     $VV_n^m[m] \leftarrow ct$   ▷ If there are no updates, advance the version vector
12:     for each server $p_n^j$, $j \in \{0 \ldots M-1\}, k \neq m$ do
13:       send ⟨HEARTBEAT ct⟩ to $p_n^j$     ▷ Send heartbeat to replicas
14:     end for
15:   end if
16: upon receive ⟨HEARTBEAT ct⟩ from $p_n^j$
17:   $VV_n^m[j] \leftarrow ct$
```

Garbage Collection. Periodically, partitions 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.

6 Correctness

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 operation is a causal snapshot according to the definition provided in Section 3.

We start by proving two Propositions that we 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 (Algorithm 1 Line 4). Further, upon reading/writing an item d, denoting by i the source replica of d, c tracks the direct dependency on d by setting $DV_c[i] = max\{DV_c[i], d.ut\}$ (Algorithm 1 Line 5 and Line 11, respectively). Finally, when writing a data item d, c stores DV_c with d (Algorithm 2 Line 7). □

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 writes Y on p_n^m, Algorithm 2 Line 6 enforces that $Clock_n^m$ is strictly higher than the maximum value in DV_c. Therefore, because $Clock_n^m$ is used as the update timestamp for Y, it follows that $Y.ut > X.ut$. □

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 server p_n^m waits until it can guarantee 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 $Y.DV[X.sr] \geq X.ut$. 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 $Y.DV[x.sr] = DV_c[X.sr] \geq X.ut \geq X_0.ut$, 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 servers in timestamp order. □

Proposition 4. Causally consistent RO-TX operation. *The result of a RO-TX operation 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 dependency vector of the client (Algorithm 2 Line 17). 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 entry-wise at least equal to the snapshot vector (Algorithm 2 Line 25). 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 belongs to the snapshot, it follows that $TV \geq Y.DV$ (Algorithm 2 Line 28). Therefore, $TV[X'.sr] \geq Y.DV[X'.sr] > X'.ut$ and $TV[X.sr] \geq Y.DV[X.sr] > X.ut$. Then, by virtue of the blocking condition in Algorithm 2 Line 25, the server replicating x, denoted by 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). □

7 Highly Available OCC

7.1 Design

In order to circumvent OCC's potential blocking behavior, 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 [30]. 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 functionality 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/consistency.

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 [29], [30], [31], [38]. 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 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 servers in other DCs to install remote updates. In Cure [18], instead, a recovery mechanism is needed because healthy DCs might have not received the same set of updates from the failed DC, leading their states to diverge.

7.2 Implementation

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 [18], a recent causally consistent system also based on physical vector clocks.[1] We call Highly Available POCC (HA-POCC) the resulting augmented version of POCC.

In Cure, partitions within a DC periodically run a stabilization protocol. It consists of exchanging their version vectors and computing the aggregate minimum called Globally Stable Snapshot (GSS). $GSS[i] = t$ means that all partitions within a DC have installed all updates originated in the ith DC with timestamp smaller or equal than t. 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.

8 Evaluation

Our main goal is to demonstrate that OCC improves data freshness while delivering performance that is competitive with pessimistic designs. The focus of this evaluation is on performance during normal operation, i.e., in the absence of network partitions. 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 that of Cure*, a re-implementation of Cure [18], a state-of-the-art causally consistent system, also based on vector clocks. Because Cure only supports transactional operations, we augment Cure* with support for GET and PUT operations. The comparison between POCC and Cure* is fair in the sense that 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.

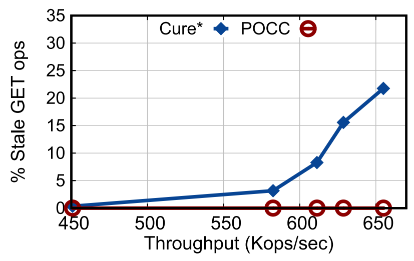
8.1 Experimental Environment

Platform. We consider an Amazon AWS deployment consisting of 3 DCs and 32 partitions per DC. The DCs are located in Oregon, North Virginia and Ireland. We use $c4.large$ instances, corresponding to 2 virtual CPUs and 3.75 GB of RAM.

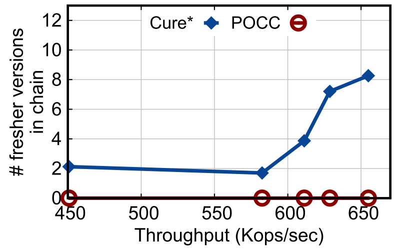
Each data partition is composed of one million key-value pairs. We consider small items, with both keys and values of 8 bytes, because they are predominant in many production workloads [2], [39], [40], [41]. Within each partition keys are chosen according to a zipf distribution with parameter 0.99.

Clients are collocated with servers, establish their sessions with the collocated server, and perform operations in a closed loop. We introduce a think time between client operations to simulate a more realistic workload, in which a client not only injects requests but also processes the result

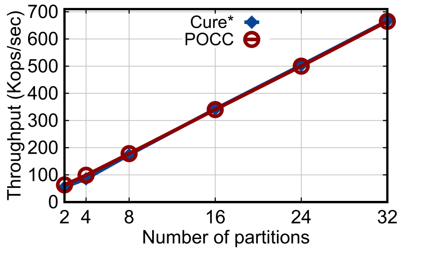
1. Cure's programming model is transaction-centric. It is, however, straightforward to adapt Cure to support GET and PUT operations.



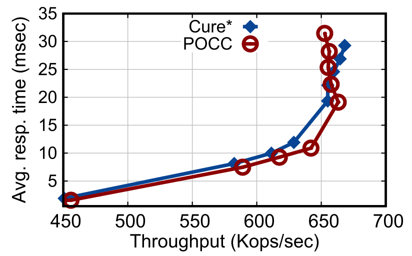
(a) Data staleness.



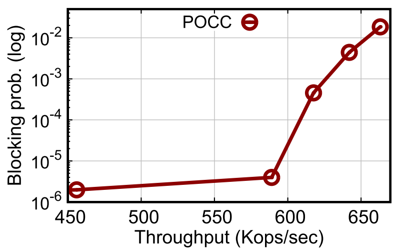
(b) Number of fresher versions in the item chain.



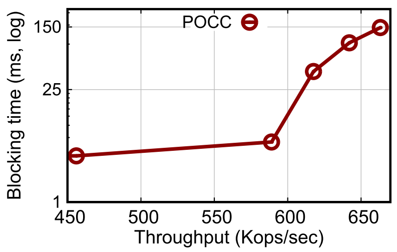
(c) Throughput while varying the number of partitions.



(d) Average response time on 32 partitions with a 32:1 GET:PUT workload.



(e) Blocking probability in POCC.



(f) Blocking time in POCC.

Fig. 3. Evaluation of data freshness, performance and blocking incidence of POCC for the single key operation workloads. The results for data staleness (a) and number of fresher versions in the item chain (b) are for the 32:1 GET:PUT workload on 32 partitions. In POCC, single key operations always return the freshest available version, so the percentage of stale reads and the number of fresher versions in the item chain for POCC is 0.

of an operation. Having a think time between operations lowers the chances that a request blocks when using OCC, because it provides extra time to receive potentially missing dependencies. However, we set the think time to 25 milliseconds, a value low enough to avoid masking the blocking dynamics in POCC, while high enough to fully load the compared systems without requiring an excessive number of client threads. We also note that 25 milliseconds is orders of magnitude lower than the think time used in standard session-based benchmarks like TPC-W [42], TPC-C [43], SpecWeb2005 [44] and Rubis [45], and therefore allows us to benchmark the effectiveness of OCC in a challenging setting.

Implementation. We implement POCC and Cure* in the same C++ code-base.[2] We run NTP [36] to keep physical clocks synchronized. As in previous work [18], 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 between conflicting updates to the same key. Heartbeats are sent by a server if it does not serve any PUT request for 1 millisecond. The stabilization protocol in Cure* is run every 5 milliseconds.

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 a specific fault tolerance mechanism, e.g., primary copy [46], Paxos [47] or chain replication [48].

8.2 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:1 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.

Data Staleness. As a representative of the protocols that enforce causal consistency in a pessimistic manner, Cure* exposes a certain degree of stale data to its clients. Our first goal is to quantify the degree of stale reads in Cure*. For that purpose, we run a workload with a 32:1 GET:PUT ratio on 32 partitions in 3 DCs.

To describe the results corresponding to data staleness in Cure* we first provide a definition of a *stale* data item. We define a data item as "stale" if there is at least one newer version in the item chain when the item is requested, i.e., if the version returned to the client is not the one with the highest timestamp. In order to quantify the data staleness incurred by Cure*, we utilize two metrics: (*i*) the percentage of stale data items returned to the client and (*ii*) the number of fresher item versions available in the version chain at the time a stale data item is returned to the client. We remind the reader that the percentage of stale data items returned to clients by POCC is 0, as is the number of fresher item versions available. When running workloads composed of single key GET and PUT operations, POCC always returns the freshest available item version, offering maximum data freshness to its clients.

Fig. 3a shows the percentage of stale data items returned in Cure*. The plot shows that the probability to return stale data increases with the load. At high load but before the saturation point, the percentage of stale items returned reaches 15 percent. When Cure* is even more highly loaded, the percentage of stale data items returned grows as high as 22 percent. The main reason is the higher remote update rates, but higher contention on physical resources also slows down the execution of the stabilization protocol needed to identify stable versions. In bigger deployments the execution of the

2. https://github.com/epfl-labos/POCC

stabilization protocol would naturally progress at a slower pace, with a commensurate increase in staleness. In addition, these results correspond to running the stabilization protocol every 5 milliseconds. Higher values allow the system to reach a higher throughput (as shown in previous work [13]), but at the cost of increased data staleness. By contrast, POCC is immune to this trade-off. A similar dynamic can be observed by Fig. 3b, which shows the number of fresher versions available in the version chain of a stale returned data item.

Scalability and Performance. Our next experiments are designed to show that the data freshness benefits of POCC do not come at the cost of scalability or other performance penalties.

First, we investigate the scalability of both systems in terms of throughput as a function of the number of partitions. To this end, we deploy POCC and Cure* on a system consisting of a number of partitions ranging from 2 to 32, and we measure the maximum achievable throughput. The GET:PUT ratio is N:1, where N denotes the number of partitions. Fig. 3c reports the result of this experiment. The plot shows that the two systems achieve basically the same throughput, showing that the optimistic approach does not incur any throughput loss in comparison to the corresponding pessimistic implementation.

Second, to assess the effect on response time, we fix the number of partitions at 32, and we investigate the average operation response time achieved while increasing the workload intensity. The result of the experiment is depicted by Fig. 3d. POCC achieves a slightly lower response time than Cure* before reaching the saturation point at about 650 Kops/sec. As we shall see in more detail later, POCC rarely blocks upon serving an operation, and it is more resource efficient than Cure*. In particular, POCC never traverses an item's version chain, and does not need to run a stabilization protocol. Under very high load, POCC performs slightly worse than Cure*, because POCC's better resource efficiency is outweighed by the cost of blocking.

Blocking Dynamics. To better understand the performance results presented so far, we investigate the blocking behavior of POCC. To this end, Fig. 3e reports the blocking probability of POCC, and Fig. 3f reports the blocking time. The results refer to the aforementioned 32:1 GET:PUT workload running on 32 partitions. In both figures, the y-axis is on a logarithmic scale.

Fig. 3e shows that the blocking probability is negligible under moderate to high load and becomes noticeable only as the system approaches the maximum achievable throughput (650 Kops/sec). Up to ~600 Kops/sec the blocking probability is lower than 0.001, meaning that the 99.999th percentile of the operation response time is not affected by the blocking behavior of POCC. The blocking time exhibits a similar dynamic, shown in Fig. 3f. It is on the order of a few microseconds as long as the system is not heavily loaded.

Resource Efficiency. Fig. 4 shows the breakdown of the amount of data exchanged in Cure* for the replication and the stabilization protocols, when varying the GET:PUT ratio from 32:1 to 1:1 on a 3 DCs and 32 partitions deployment. POCC does not run a stabilization protocol, so it only exchanges data for replication. Cure*, on the contrary, has to run a stabilization protocol in order to track the delivery of dependencies. The results are normalized with respect to the amounts of data exchanged in POCC at the same throughput. POCC exchanges up to 37 percent less metadata than Cure* because it does not need to run any stabilization protocol.

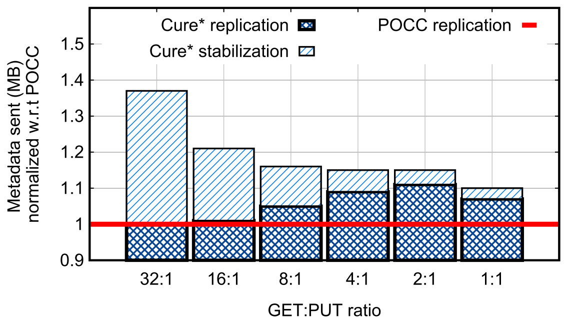


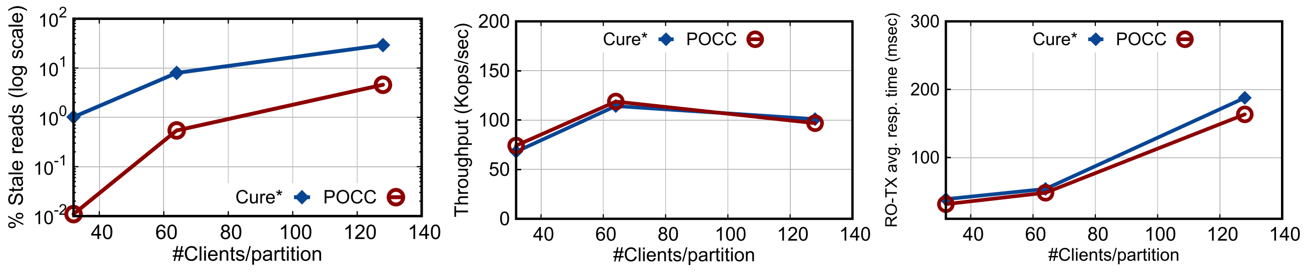
Fig. 4. POCC incurs lower overhead than Cure* because it does not need to run a stabilization protocol for tracking the delivery of updates.

Summary of the Results. Figs. 3a and 3b demonstrate the negative effect of the pessimistic implementation of causal consistency in Cure* on data freshness. POCC, on the contrary, always returns the freshest values for a requested item for the single key operations. Fig. 3c reveals that POCC is competitive with Cure*, and Fig. 3d shows that POCC even offers lower latencies until the saturation point is reached. Both figures show that achieving optimal data freshness comes at no significant performance penalty. Even under high load, when POCC blocks more frequently and for a longer amount of time, the performance of POCC remains competitive with that of Cure*. POCC achieves this result because blocking remains a rare event, without any effect on the vast majority of operations, and because a blocked operation yields the CPU, thus increasing the computational resources available to serve non-blocking operations. Fig. 3e attests that the blocking probability is negligible, and Fig. 3f shows that the blocking time is on the order of a few milliseconds. Fig. 4 reveals that POCC is more resource efficient than Cure*, because it does not need to run any stabilization protocol and because it never traverses the version chain of an item. In Cure*, instead, traversing the version chain upon serving a GET operation leads to higher CPU demand, and all requests concurrently compete for the CPU, mutually hurting each other's performance.

8.3 Transactional Workload

We now assess the data freshness benefits and 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 N items corresponding to N distinct partitions, and then performs a random PUT operation. N is varied from 1 to 32.

Data Staleness. As already explained in Section 5.1.2, with transactions, OCC does not in general return the freshest version of a key. Therefore, both POCC and Cure* are prone to return stale data. In this paragraph we compare the staleness of data returned to client by POCC and Cure*. In particular, Fig. 5a reports, on a logarithmic scale, the percentage of stale data items returned by the two systems.



(a) Data staleness in POCC and Cure* with different number of clients per partition. (b) Throughput with different number of clients per partition. (c) Average response time with different number of clients per partition.

Fig. 5. Evaluation of POCC and Cure* with a workload composed of PUT and RO-TX operations. The systems are deployed on 32 partitions per DC.

The plot shows that the staleness exhibited by POCC is two orders of magnitude lower than that of Cure*. POCC returns much fresher data because it 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 server. Cure*, instead, defines such boundaries in terms of items that are *stable* on the transaction coordinator, thus making it much more prone to read stale values.

Throughput and Response Time. We now investigate the throughput and response time dynamics while increasing the number of active clients. Figs. 5b and 5c depict the throughput and the average RO-TX response time, respectively, achieved by POCC and Cure* when transactions involve half of the partitions in the system. The plots reveal that the two systems exhibit very similar performance, with POCC achieving up to 12 percent lower average response time. This demonstrates that POCC achieves two orders of magnitude gain in data freshness, shown on Fig. 5a, without additional costs on performance. On the contrary, POCC achieves lower response time and slightly higher throughout.

Scalability. Fig. 6 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 partitions is small. The gain of POCC over Cure* becomes more noticeable (up to 15 percent) when transactions involve the majority of the partitions in the system, because, as the number of partitions increases, executing a transaction becomes more resource demanding. POCC, then, achieves a higher throughput because it is more resource efficient than Cure*.

Blocking Behavior. Fig. 7 shows two plots: (*i*) the probability that a PUT or a transactional read stalls on a server and (*ii*) the average amount of time a stalled request is delayed. The reported results correspond to the same workload as in the previous paragraph. The plots on Fig. 7 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 throughput, 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 increased blocking times (and blocking probability).

Summary of the Results. We have shown that OCC delivers much fresher data to clients (up to two orders of magnitude) compared to the pessimistic design of Cure*. Besides the improvement in data freshness, 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 of OCC. Transactions are more

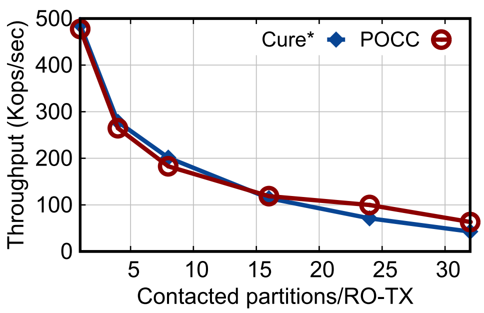


Fig. 6. Throughput of POCC and Cure* while varying the number of contacted partitions per transaction. The workload is composed of PUT and RO-TX operations, the systems are deployed on 32 partitions per DC. RO-TX touch 16 partitions.

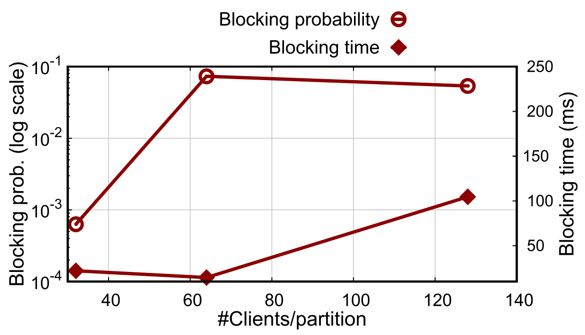


Fig. 7. Blocking behavior in POCC with different number of clients per partition. The left y-axe is represented in logarithmic scale.

demanding both in terms of computational resources and communication. Therefore, with a workload including transactions, POCC higher resource efficiency pays off more than in the case of a workload consisting of GET and PUT operations.

9 Related Work

Our work is primarily related to the vast literature on causally consistent systems, which include COPS [8], Eiger [9], Bolt-on causal consistency [12], ChainReaction [11], Orbe [10], GentleRain [13], Bolt-on CC [12], SwiftCloud [14], Saturn [49], Contrarian [15], Wren [16], CausalSpartan [50], COPS-SNOW [17], Cure [18] and PaRiS [51]. These systems differ in the mechanism they employ to achieve causal consistency. Some of them use logical clocks [8], [9], [10], and rely on the exchange of dependency messages to determine the visibility of remote updates. Dependencies are tracked at the item [8], [9], operation [9] or partition [10] granularity. Other systems use physical clocks [13], [18] or hybrid logical physical clocks [15], [16], [50], and run a periodic stabilization protocol to identify stable items. Saturn [49] relies on a metadata dissemination service with configurable data freshness. 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.

Occult [19] is an optimistic system, designed to avoid slowdown cascades [20]. Among existing systems, Occult is the most similar to OCC, and it was developed at about the same time as POCC (both Occult and the conference version of this paper appeared around the same time). Occult supports a stronger consistency level with respect to CC and, similarly to OCC, exposes unstable dependencies. There are three main differences between Occult and OCC. First, Occult implements master-slave replication model where only the master replica of a partition accepts writes, implying that some of the writes in Occult have to go to a remote master replica. In POCC, on the contrary, all the writes execute locally. Second, a read operation in Occult may have to be retried several times in case of unresolved dependencies, and may even have to contact the remote master replica, which might not be accessible due to a network partition. Retrying read operations has a negative impact on performance, comparable to blocking the read to receive the correct version to return. In POCC, a read operation whose dependencies are not resolved, waits for missing dependencies to arrive. However, all the read operations are executed in the local DC. Third, Occult may abort read-only transactions. By contrast, POCC never aborts read-only transactions. One benefit of aborting the read-only transactions in Occult is that by repeating the RO-TX, it could return fresher values than POCC because the transaction is executed at a later point of time. Occult is similar to POCC in the way that it tracks causality by using physical clock timestamps.

OCC is also related to the literature on *speculative* systems. Speculation consists of optimistically processing an operation even if previous operations have not yet fully completed and only their tentative result is available. If the tentative result differs from the final one, the system rollbacks to a consistent state. Speculation has been investigated in a variety of contexts, including processor architectures [52], operating systems [53], simulators [54], concurrency control schemes for transactional systems [55], software transactional memory [56] and recent systems that can concurrently support different consistency levels [28]. 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 and to reduce the overhead for dependency checking and stabilization. As we have shown, in many cases the better resource efficiency enabled by OCC can also yield performance gains in terms both of response times and achievable throughput.

Speculation has been widely investigated in the context of strongly consistent transactional systems [57], [58], [59] and, recently, also in the context of systems achieving lower consistency guarantees. With reference to a transactional systems, speculation entails executing a transaction even if the outcome of previous, potentially conflicting transaction has not been determined yet. If a transaction T waiting for validation aborts, it may cause a domino effect, aborting all or part of the transactions that have been started after T. Speculation optimistically assumes a low contention rate and aims to reduce the response time of a transaction by overlapping the execution and validation phases of subsequent transactions. Our proposal is, instead, framed in the context of a weaker consistency model and is primarily aimed to maximize the freshness of the data returned to client.

10 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 pessimistic implementations of causal consistency for better freshness of the data returned to clients and higher resource efficiency.

We have implemented OCC in POCC. We have shown that POCC achieves performance that is comparable or better than its pessimistic counterpart in a wide range of workloads, while being able to reduce or altogether eliminate the return of stale data 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 data center failures.

Acknowledgment

This research has been supported by The Swiss National Science Foundation through Grant No. 166306, and by an EcoCloud post-doctoral fellowship. The authors were also grateful to Amazon for providing a research grant through AWS Cloud Credits.

REFERENCES

[1] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[2] R. Nishtala *et al.*, "Scaling Memcache at Facebook," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.

[3] S. A. Noghabi *et al.*, "Ambry: Linkedin's scalable geo-distributed object store," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 253–265.

[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. 19th Annu. ACM Symp. Princ. Distrib. Comput.*, 2000, Art. no. 7.

[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," *Distrib. Comput.*, 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. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 401–416.

[9] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 313–328.

[10] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–14.

[11] S. Almeida, J. A. Leitão, and L. Rodrigues, "ChainReaction: A causal+ consistent datastore based on chain replication," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 85–98.

[12] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 761–772.

[13] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "GentleRain: Cheap and scalable causal consistency with physical clocks," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–13.

[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. 16th Annu. Middleware Conf.*, 2015, pp. 75–87.

[15] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel, "Causal consistency and latency optimality: Friend or Foe?" *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1618–1632, Jul. 2018.

[16] K. Spirovska, D. Didona, and W. Zwaenepoel, "Wren: Nonblocking reads in a partitioned transactional causally consistent data store," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2018, pp. 1–12.

[17] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, "The SNOW theorem and latency-optimal read-only transactions," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 135–150.

[18] D. D. Akkoorath *et al.*, "Cure: Strong semantics meets high availability and low latency," in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst.*, 2016, pp. 405–414.

[19] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I can't believe it's not causal! Scalable causal consistency with no slowdown cascades," in *Proc. 14th USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 453–468.

[20] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan, "Challenges to adopting stronger consistency at scale," in *Proc. 15th USENIX Conf. Hot Topics Operating Syst.*, 2015, Art. no. 13.

[21] J. Chu, "Data freshness, not speed, most important for IoT," 2018. [Online]. Available: http://news.mit.edu/2018/keeping-data-fresh-wireless-networks-0605

[22] S. Jones, "Why data freshness is more important than speed for IoT," 2018. [Online]. Available: https://it.toolbox.com/blogs/stevejones/why-data-freshness-is-more-important-than-speed-for-iot-080918

[23] S. Jones, "Data freshness, not speed, most important for IoT." 2018. [Online]. Available: https://www.networkworld.com/article/3281126/data-freshness-not-speed-most-important-for-iot.html

[24] M. Adorjan, "AWS inter-region latency monitoring." Accessed: Oct. 2, 2020. [Online]. Available: https://www.cloudping.co/grid

[25] 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 *Proc. 5th Biennial Conf. Innovative Data Syst. Res.*, 2011, pp. 134–143.

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

[27] H. Lu *et al.*, "Existential consistency: Measuring and understanding consistency at Facebook," in *Proc. 25th Symp. Operating Syst. Princ.*, 2015, pp. 295–310.

[28] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi, "Incremental consistency guarantees for replicated objects," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 169–184.

[29] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 385–400.

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

[31] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, "F10: A fault-tolerant engineered network," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 399–412.

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

[33] 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.

[34] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie, "What consistency does your key-value store actually provide?" in *Proc. 6th Int. Conf. Hot Top. Syst. Dependability*, 2010, pp. 1–16.

[35] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *Proc. Int. Conf. Princ. Distrib. Syst.*, 2014, pp. 17–32.

[36] NTP: The network time protocol, Accessed: Oct. 2, 2020. [Online]. Available: http://www.ntp.org

[37] D. S. Parker *et al.*, "Detection of mutual inconsistency in distributed systems," *IEEE Trans. Softw. Eng.*, vol. SE-9, no. 3, pp. 240–247, May 1983.

[38] 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

[39] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. 12th ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 53–64.

[40] How much text versus metadata is in a tweet? 2011. [Online]. Available: http://goo.gl/EBFIFs

[41] Storing hundreds of millions of simple key-value pairs in Redis, 2011. [Online]. Available: http://goo.gl/ieeU17

[42] D. F. García and J. García, "TPC-W E-commerce benchmark evaluation," *Computer*, vol. 36, no. 2, pp. 42–48, 2003.

[43] D. R. Llanos and B. Palop, "TPCC-UVa: An open-source TPC-C implementation for parallel and distributed systems," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp.*, 2006, p. 8.

[44] R. Hariharan and N. Sun, "Workload characterization of SPECweb2005," in *Proc. SPEC Benchmark Workshop*, 2006, pp. 2–7.

[45] S. Elnikety, S. Dropsho, E. Cecchet, and W. Zwaenepoel, "Predicting replicated database scalability from standalone database profiling," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, 2009, pp. 303–316.

[46] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proc. 7th Annu. ACM Symp. Princ. Distrib. Comput.*, 1988, pp. 8–17.

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

[48] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proc. 6th Conf. Symp. Operating Syst. Des. Implementation*, 2004, Art. no. 7.

[49] M. Bravo, L. Rodrigues, and P. Van Roy, "Saturn: A distributed metadata service for causal consistency," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 111–126.

[50] M. Roohitavaf, M. Demirbas, and S. Kulkarni, "CausalSpartan: Causal consistency for distributed data stores using hybrid logical clocks," in *Proc. IEEE 36th Symp. Reliable Distrib. Syst.*, 2017, pp. 184–193.

[51] K. Spirovska, D. Didona, and W. Zwaenepoel, "PaRiS: Causally consistent transactions with non-blocking reads and partial replication," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 304–316.

[52] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Mateo, CA, USA: Morgan Kaufmann, 2011.

[53] E. B. Nightingale, P. M. Chen, and J. Flinn, "Speculative execution in a distributed file system," *ACM Trans. Comput. Syst.*, vol. 24, no. 4, pp. 361–392, Nov. 2006.

[54] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto, "Time warp operating system," in *Proc. 11th ACM Symp. Operating Syst. Princ.*, 1987, pp. 77–93.

[55] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, Jun. 1981.

[56] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues, "SPECULA: Speculative replication of software transactional memory," in *Proc. IEEE 31st Symp. Reliable Distrib. Syst.*, 2012, pp. 91–100. [Online]. Available: https://doi.org/10.1109/SRDS.2012.67

[57] B. Kemme, F. Pedone, G. Alonso, and A. Schipher, "Processing transactions over optimistic atomic broadcast protocols," in *Proc. 19th IEEE Int. Conf. Distrib. Comput. Syst.*, 1999, pp. 424–431.

[58] U. Cetintemel, P. J. Keleher, and M. J. Franklin, "Support for speculative update propagation and mobility in Deno," in *Proc. 21st Int. Conf. Distrib. Comput. Syst.*, 2001, pp. 509–516.

[59] R. Palmieri, F. Quaglia, and P. Romano, "AGGRO: Boosting STM replication via aggressively optimistic transaction processing," in *Proc. 9th IEEE Int. Symp. Netw. Comput. Appl.*, 2010, pp. 20–27.

Kristina Spirovska received the BS and MS degrees in computer science from the Institute of Informatics, Faculty of Natural Sciences and Mathematics and the Faculty of Computer Science and Engineering, Skopje, Macedonia, in 2009 and 2013, respectively, and the PhD degree in computer and communication sciences from EPFL, Lausanne, Switzerland, in 2020. Her PhD research comprised designing, implementing and evaluating novel causally consistent geo-replicated data stores. Her research interests include distributed systems, key-value stores, consistency in geo-replicated data platforms, especially transactional semantics, and consistency guarantees in distributed data stores.

Diego Didona received the MS degree in computer engineering from the Sapienza Università di Roma, Rome, Italy, in 2010, and the PhD degree in computer engineering from the Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal, in 2015. He is a visiting scientist with the Cloud & AI Systems Department, IBM Research Zurich. His research interests include data platform designs for main memory and emerging storage devices, consistency in geo-replicated data platforms, and performance modeling applied to self-tuning systems.

Willy Zwaenepoel (Fellow, IEEE) received the BS degree from the University of Gent, Ghent, Belgium, in 1979, and the MS and PhD degrees from Stanford University, Stanford, California, in 1980 and 1984, respectively. He spent almost two decades with Rice University, where he was the Karl F. Hasselmann professor of computer science and electrical and computer engineering. In September 2002, he joined EPFL, where he was dean of the School of Computer and Communications Sciences for nine years. In July 2018, he joined the University of Sydney as dean of the Faculty of Engineering. He was elected fellow of the ACM, in 2000. In 2000, he received the Rice University Graduate Student Association Teaching and Mentoring Award. In 2007, he received the IEEE Tsutomu Kanai Award. He was elected to the European Academy in 2009. He was program chair of OSDI in 1996 and EuroSys in 2006, and general chair of MobiSys in 2004. He was also an associate editor of the *IEEE Transactions on Parallel and Distributed Systems* from 1998 to 2002. He has worked in a variety of aspects of operating and distributed systems, including microkernels, fault tolerance, parallel scientific computing on clusters of workstations, clusters for web services, mobile computing, database replication, and virtualization. He is well known for his work on the TreadMarks distributed shared memory system, which was licensed to Intel and became the basis for Intelś OpenMP cluster product. His current interests cover all aspects of software systems, especially distributed systems, and operating systems.