Causal Consistency and Latency Optimality: Friend or Foe? [Extended Version]

Diego Didona[1], Rachid Guerraoui[1], Jingjing Wang[1], Willy Zwaenepoel[1,2]
[1]EPFL, [2] University of Sydney
first.last@epfl.ch

ABSTRACT

Causal consistency is an attractive consistency model for geo-replicated data stores. It is provably the strongest model that tolerates network partitions. It avoids the long latencies associated with strong consistency, and, especially when using read-only transactions (ROTs), it prevents many of the anomalies of weaker consistency models. Recent work has shown that causal consistency allows "latency-optimal" ROTs, that are nonblocking, single-round and single-version in terms of communication. On the surface, this latency optimality is very appealing, as the vast majority of applications are assumed to have read-dominated workloads.

In this paper, we show that such "latency-optimal" ROTs induce an extra overhead on writes that is so high that it actually jeopardizes performance even in read-dominated workloads. We show this result from a practical as well as from a theoretical angle.

We present the Contrarian protocol that implements "almost latency-optimal" ROTs, but that does not impose on the writes any of the overheads present in latency-optimal protocols. In Contrarian, ROTs are nonblocking and single-version, but they require two rounds of client-server communication. We experimentally show that this protocol not only achieves higher throughput, but, surprisingly, also provides better latencies for all but the lowest loads and the most read-heavy workloads.

We furthermore prove that the extra overhead imposed on writes by latency-optimal ROTs is inherent, i.e., it is not an artifact of the design we consider, and cannot be avoided by *any* implementation of latency-optimal ROTs. We show in particular that this overhead grows linearly with the number of clients.

PVLDB Reference Format:
Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. Causal Consistency and Latency Optimality: Friend or Foe?. *PVLDB*, 11 (11): xxxx-yyyy, 2018.
DOI: https://doi.org/10.14778/3236187.3236210

1. INTRODUCTION

Geo-replication is gaining momentum in industry [9, 16, 20, 22, 25, 43, 50, 51, 65] and academia [24, 34, 47, 49, 59, 69, 70, 71] as a design choice for large-scale data platforms to meet the strict latency and availability requirements of on-line applications [5, 55, 62].

Causal consistency. To build geo-replicated data stores, causal consistency (CC) [2] is an attractive consistency model. On the one hand, CC has an intuitive semantics and avoids many anomalies that are allowed under weaker consistency models [25, 67]. On the other hand, CC avoids the long latencies incurred by strong consistency [22, 31] and tolerates network partitions [40]. CC is provably the strongest consistency level that can be achieved in an always-available system [7, 44]. CC has been the target consistency level of many systems [4, 19, 26, 27, 30, 40, 41]. It is used in platforms that support multiple levels of consistency [13, 39], and it is a building block for strong consistency systems [12] as well as for formal checkers of distributed protocols [29].

Read-only transactions. High-level operations such as producing a web page often translate to multiple reads from the underlying data store [50]. Ensuring that all these reads are served from the same consistent snapshot avoids undesirable anomalies, in particular the following well-known anomaly: Alice removes Bob from the access list of a photo album and adds a photo to it, but Bob reads the original permissions and the new version of the album [40]. Therefore, the vast majority of CC systems provide read-only transactions (ROTs) to read multiple items at once from a causally consistent snapshot [3, 4, 27, 40, 41]. Large-scale applications are often read-heavy [6, 43, 50, 51]. Hence, achieving low-latency ROTs is a first-class concern for CC systems.

Earlier CC ROT designs were blocking [3, 4, 26, 27] or required multiple rounds of communications to complete [4, 40, 41]. The recent COPS-SNOW system [42] shows that it is possible to perform CC ROTs in a nonblocking fashion, using a single round of communication, and sending only a single version of the keys involved. Because it exhibits these properties, the COPS-SNOW ROT protocol was termed *latency-optimal (LO)*. COPS-SNOW achieves LO by imposing additional processing costs on writes. One could argue that doing so is a correct tradeoff for the common case of read-heavy workloads, because the overhead affects the minority of operations and is to the advantage of the majority of them. This paper sheds a different light on this tradeoff.

Contributions. In this paper we show that the extra cost on writes is so high that so-called LO ROTs in practice exhibit performance inferior to alternative designs, even in read-heavy workloads. Not only does this extra cost reduce the available processing power, leading to lower throughput, but it also causes higher resource contention, and hence higher latencies. We demonstrate this counterintuitive result from two angles.

(1) From a practical standpoint, we propose Contrarian, a CC design that achieves all but one of the properties of a LO design, without incurring the overhead on writes that LO implies. In particular, it is nonblocking and single-version, but it requires two rounds of communication. Measurements in a variety of scenarios demonstrate that, for all but the lowest loads, Contrarian provides better latencies and throughput than an LO protocol.

(2) From a theoretical standpoint, we show that the extra cost imposed on writes to achieve LO ROTs is *inherent* to CC, i.e., it cannot be avoided by *any* CC system that implements LO ROTs. We also provide a lower bound on this extra cost in terms of communication overhead. Specifically, we show that the amount of extra information exchanged potentially grows linearly with the number of clients.

Roadmap. The remainder of this paper is organized as follows. Section 2 provides introductory concepts and definitions. Section 3 surveys the complexities involved in the implementation of ROTs. Section 4 presents our Contrarian protocol. Section 5 compares Contrarian and an LO design. Section 6 presents our theoretical results. Section 7 discusses related work. Section 8 concludes the paper. We provide the pseudo-code of Contrarian, and we sketch an informal proof of its correctness in an Appendix.

2. SYSTEM MODEL

We consider a multi-version key-value store, as in the vast majority of CC systems [3, 27, 40, 41, 42]. We denote keys by lower-case letters, e.g., x, and versions of keys by the corresponding upper-case letters, e.g., X.

2.1 API

The key-value store provides the following operations:

• $X \leftarrow GET(x)$ **:** returns a version of key x, or $\perp$, if there is no version identified by x.

• $PUT(x, X)$ **:** creates a new version X of key x.

• $(X, Y, ...) \leftarrow ROT(x, y, ...)$ **:** returns a vector $(X,\ Y,\ ...)$ of versions of keys $(x,\ y,\ ...\ )$. A ROT returns $\perp$ for a key x, if there is no version identified by x.

In the remainder of this paper we focus on PUT and ROT operations. DELETE can be treated as a special case of PUT.

2.2 Partitioning and Replication

We target a key-value store whose data set is split into $N > 1$ partitions. Each key is deterministically assigned to one partition by a hash function, and each partition is assigned to one server. A PUT(x, X) is sent to the partition that stores x. Read requests within a ROT are sent to the partitions that store the keys in the specified key set.

Each partition is replicated at $M \geq 1$ data centers (DC). Our results hold for both single and replicated DCs. In the case of replication, we consider a multi-master design, i.e., all replicas of a key accept PUT operations.

2.3 Properties of ROTs

2.3.1 LO ROTs.

We adopt the same terminology and definitions as in the original formulation of latency-optimality [42]. An implementation provides LO ROTs if it satisfies three properties: *one-version*, *one-round* and *nonblocking*. We now informally describe these properties. A more formal definition is deferred to § 6.

Nonblocking requires that a partition that receives a request to perform reads within a ROT can serve such reads without being blocked by any external event (e.g., the acquisition of a lock or the receipt of a message) [1]. *One-round* requires that a ROT is served in two communication steps: one step from the client to the servers to invoke the ROT, and another step from the servers to the client to return the results. *One-version* requires that servers return to clients only one version of each requested key.

2.3.2 One-shot ROTs.

As in Lu et al. [42], we consider *one-shot* ROTs [33]: the input arguments of a ROT specify all keys to be read, and the individual reads within a ROT are sent in parallel to the corresponding partitions. A read that depends on the outcome of an earlier read has to be issued in a subsequent ROT. We focus on one-shot ROTs for simplicity and because our results generalize: multi-shot ROTs incur at least the same overhead as one-shot ROTs.

2.4 Causal Consistency

The *causality order* is a happens-before relationship between any two operations in a given execution [2, 37]. For any two operations α and β, we say that β causally depends on α, and we write $\alpha \rightsquigarrow \beta$, if and only if at least one of the following conditions holds: *i)* α and β are operations in a single thread of execution, and α happens before β; *ii)* $\exists x, X$ such that α creates version X of key x, and β reads X; *iii)* $\exists \gamma$ such that $\alpha \rightsquigarrow \gamma$ and $\gamma \rightsquigarrow \beta$. If α is a PUT that creates version X of x, and β is a PUT that creates version Y of y, and $\alpha \rightsquigarrow \beta$, then (with a slight abuse of notation) we also say Y causally depends on X, and we write $X \rightsquigarrow Y$.

A *causally consistent* data store respects the causality order. Intuitively, if a client c reads Y and $X \rightsquigarrow Y$, then any subsequent read performed by c on x returns either X or a newer version. In other words, c cannot read $X' : X' \rightsquigarrow X$. A ROT operation returns item versions from a *causally consistent snapshot* [40, 45]: if a ROT returns X and Y such that $X \rightsquigarrow Y$, then there is no X' such that $X \rightsquigarrow X' \rightsquigarrow Y$.

To circumvent trivial implementations of causal consistency, we require that a version, once written, becomes *eventually visible*, meaning that it is available to be read by all clients after some finite time [11].

Causal consistency does not establish an order among concurrent (i.e., not causally related) updates on the same key. Hence, different replicas of the same key might diverge and expose different values [67]. We consider a system that eventually converges: if there are no further updates, then eventually all replicas of any key take on the same value, for instance using the last-writer-wins rule [64].

[1]The meaning of the term *nonblocking* in this paper follows the definition in Lu et al. [42], and is different from the definition used in the distributed transaction processing literature [17, 58].

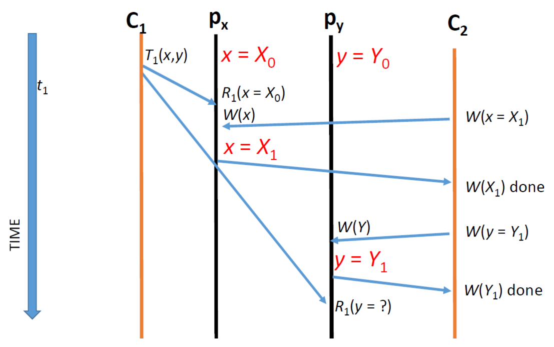


Figure 1: Challenges in implementing CC ROTs. C_1 issues $ROT(x, y)$. If T_1 returns X_0 to C_1, then T_1 cannot return Y_1, because there is a version of x, X_1, such that $X_0 \rightsquigarrow X_1 \rightsquigarrow Y_1$.

Hereafter, when we use the term causal consistency, eventual visibility and convergence are implied.

3. BACKGROUND

Challenges of CC ROTs. Even in a single DC, partitions involved in a ROT cannot simply return the most recent version of a requested key if one wants to ensure that a ROT observes a causally consistent snapshot. Consider the scenario of Figure 1, with two keys x and y, with initial versions X_0 and Y_0, and residing on partitions p_x and p_y, respectively. Client C_1 performs a ROT(x, y), and client C_2 performs a PUT(x, X_1) and later a PUT(y, Y_1). By asynchrony, the read on x by C_1 arrives at p_x before the PUT by C_2 on x, and the read by C_1 on y arrives at p_y after the PUT by C_2 on y. In this case, p_y cannot return Y_1 to C_1, because a snapshot consisting of X_0 and Y_1, with $X_0 \rightsquigarrow X_1 \rightsquigarrow Y_1$, violates the causal consistency property for snapshots (see Section 2.4).

Existing non-LO solutions. COPS [40] and Eiger [41] provide a first solution to the problem. In these protocols, a ROT(x, y) returns the latest versions of x and y, combined with meta-data that encodes their dependencies (a dependency graph in COPS and a timestamp in Eiger). The client uses this meta-data to determine whether the returned versions belong to a causally consistent snapshot. If not, then the client issues a second round of requests for those keys for which the versions it received do not belong to a causally consistent snapshot. In these requests it includes the necessary information for the server to identify which version has to be returned for each of those keys. This protocol is nonblocking, but requires (potentially) two rounds of communication and two versions of key(s) being communicated.

Later designs [3, 27] opt for a timestamp-based approach, in which each version has a timestamp *ts* that encodes causality (i.e., $X \rightsquigarrow Y$ implies $X.ts < Y.ts$), and each ROT also is assigned a *snapshot timestamp* (*st*). Upon receiving a ROT request, a partition first makes sure that its local clock has caught up to *st* [3], ensuring that all future versions have a timestamp higher than *st*. Then, the partition returns the most recent version with a timestamp $\leq st$. The snapshot timestamp is picked by a transaction *coordinator* [3, 27]. Any server can be the coordinator of a ROT. The client provides the coordinator with the highest timestamp it has

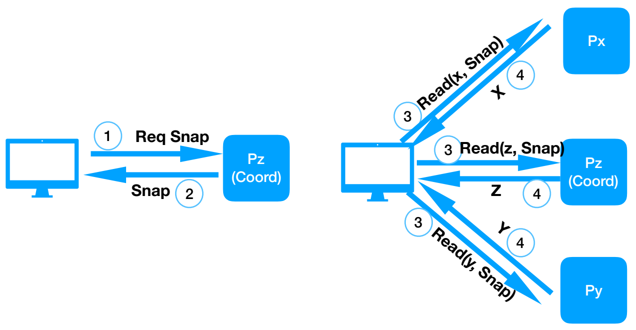


Figure 2: ROT implementation in the timestamp-based approach, requiring 2 rounds of client-server communication. Numbered circles depict the order of operations. The client always piggybacks on its requests the last snapshot it has seen (not shown), so as to observe monotonically increasing snapshots. Any server involved in a ROT can act as its coordinator.

observed, and the coordinator picks the transaction timestamp as the maximum of the client-provided timestamp and its own clock value. [2] This protocol returns only a single version of each key, but it always requires two rounds of communication: one to obtain the snapshot and one to read the key versions from said snapshot (as shown in Figure 2). In addition, if physical clocks are used to encode timestamps [3, 27], the protocol is also blocking, because a partition may need to wait for its physical clock to reach *st*.

LO CC ROTs. COPS-SNOW [42] is the first CC system to implement LO ROTs. We depict in Figure 3 how the COPS-SNOW protocol works using the same scenario as in Figure 1. Each ROT is given a unique identifier. When a ROT T_1 reads X_0, p_x records T_1 as a reader of x. It also records the (logical) time at which the read occurred. On a later PUT on x, T_1 is added to the "old readers of x", the set of transactions that have read a version of x that is no longer the most recent version, again together with the logical time at which the read occurred.

When C_2 sends its PUT on y to p_y, it includes in this request that this PUT is dependent on X_1. Partition p_y interrogates p_x as to whether there are old readers of x, and, if so, records the old readers of x into the old reader record of y, together with their logical time. When later the read of T_1 on y arrives, p_y finds T_1 in the old reader record of y. p_y therefore knows that it cannot return Y_1. Using the logical time in the old reader record, it returns the most recent version of y before that time, in this case Y_0. In the rest of the paper, we refer to this procedure as the *readers check*. By virtue of the readers check, COPS-SNOW is one-round, one-version and nonblocking.

COPS-SNOW, however, incurs a very high cost on PUTs. We demonstrate this cost by slightly modifying our example. Let us assume that hundreds of ROTs read X_0 before the PUT(x, X_1), as might well occur with a skewed workload in which x is a hot key. Then, all these transactions must be stored as readers and later as old readers of x, communicated to p_y, and examined by p_y on each incoming read from a ROT. Let us further modify the example by assuming that

[2]The client cannot pick *st* itself, because its timestamp may be arbitrarily far behind, compromising eventual visibility.

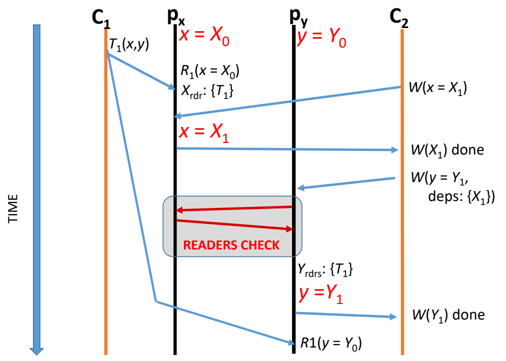


Figure 3: COPS-SNOW design. C_2 declares that Y_1 depends on X_0. Before completing the PUT of Y_1, p_y runs a "readers check" with p_x and is informed that T_1 has observed a snapshot that does not include Y_1.

C_2 reads keys from partitions p_i different from p_x and p_y before writing Y_1. Because C_2 has established a dependency on all the versions it has read, in order to compute the old readers for y, p_y needs to interrogate not only p_x, but all the other partitions p_i.

Challenges of geo-replication. Further complications arise in a geo-replicated setting with multiple DCs. We assume that new versions are replicated asynchronously, so a new version X may arrive at a DC before its causal dependencies. COPS, Eiger and COPS-SNOW deal with this situation through a technique called *dependency checking*. When X is replicated, the list of causal dependencies of X is sent along (without the corresponding values). Before X is installed, the system checks by means of dependency check messages to other partitions that X's causal dependencies are present. When X's dependencies have been installed in the DC, X can be installed as well. In COPS-SNOW, in addition, the readers check for X proceeds in a remote DC as it does in the DC where X has been created.

An alternative technique, commonly used with timestamp-based methods, is to use a *stabilization protocol* [3, 8, 27]. Variations exist, but in general each DC establishes a cutoff timestamp below which it has received all remote versions. Versions with a timestamp lower than this cutoff can be installed. Stabilization protocols are more lightweight than dependency checking [27], but they lead to a complication in making ROTs nonblocking, in that one needs to ensure that the snapshot timestamp assigned to a ROT is below the cutoff timestamp, so that there is no blocking upon reading.

4. CONTRARIAN

Contrarian implements all but one of the properties of latency-optimal ROTs, without incurring the overhead that stems from achieving all of them. In this section we describe the salient aspects of the design of Contrarian, and the properties it achieves. We present a detailed description of the protocols implemented in Contrarian in an Appendix.

4.1 Tracking causality

Contrarian uses logical timestamps and a stabilization protocol to implement CC, but unlike what was described in Section 3, it tracks causality using dependency *vectors*, with one entry per DC, instead of scalar timestamps, and the stabilization protocol determines, in each DC, a *vector* of cutoff timestamps, also with one entry per DC [3]. We refer to this vector as the *Global Stable Snapshot (GSS)*.

The GSS encodes the set of remote versions that are *stable* in the DC. A version is stable in the DC when all its dependencies have been received in the DC. A remote version can be read by clients in a DC only when it is stable. Determining when a remote version is stable is important to achieve nonblocking ROTs. Assume $Y \rightsquigarrow Z$ and Z is made accessible to clients in DC_i before Y is received in DC_i. Then, if a client in DC_i reads Z and subsequently wants to read y, the latter read might block waiting for Y to be received in DC_i. The dependencies of a version created in DC_i on other versions created in the same DC_i are trivially satisfied. Hence, versions created in DC_i are stable in DC_i immediately after being created [3].

Encoding dependencies. Each version X tracks its causal dependencies by means of a dependency vector DV, with one entry per DC. If $X.DV[i] = t$, then X (potentially) causally depends on all versions created in DC_i with a timestamp up to t. Similarly, each client c maintains a dependency vector to track the set of versions on which c depends. The semantics of the entries of the dependency vector maintained by clients is the same as in the dependency vectors of versions.

$X.DV$ encodes the causal dependencies established by the client c that creates X by means of a PUT. When performing the PUT, c piggybacks its dependency vector. The partition that serves the PUT sets the remote entries of $X.DV$ to the values in the corresponding entries of the dependency vector provided by the client. The local entry of $X.DV$ is the timestamp of X. This timestamp is enforced to be higher than any timestamps in the dependency vector provided by the client. This enforces causality: if $Y \rightsquigarrow X$, then the timestamp of X is higher than the timestamp of Y.

X is considered stable in a remote DC_r when all X's dependencies have already been received in DC_r. This condition is satisfied if the remote entries in $X.DV$ are smaller than or equal to the corresponding entries in the current GSS of the partition that handles x in DC_r.

GSS computation. The GSS is computed independently within each DC. Each entry tracks a lower bound on the set of remote versions that have been received in the DC. If $GSS[i] = t$ in a DC, it means that all partitions in the DC have received all versions created in the i-th DC with timestamp up to t.

The GSS is computed as follows. Every partition maintains a version vector VV with one entry per DC. $VV[m]$ is the timestamp of the latest version created by the partition, where m is the index of the DC. $VV[i], i \neq m$, is the timestamp of the latest update received from the replica in the $i-$th DC. Periodically, the partitions in a DC exchange their VVs and compute the GSS as the aggregate minimum vector. Hence, the GSS encodes a lower bound on the set of remote versions that have been received by *every* partition in the DC. The partitions also move their local clocks forward, if needed, to match the highest timestamp corresponding to the local entry in any of the exchanged VVs.

[3]This also implies that the local entry of the GSS is not used to track dependencies. However, the local entry is kept in our discussion for simplicity, so that the i-th entry in the GSS refers to the i-th DC.

To ensure that the GSS progresses even in absence of updates, a partition sends a heartbeat message with its current clock value to its replicas if it does not process a PUT for a given amount of time.

4.2 ROT implementation

Contrarian's ROT protocol runs in 2 rounds, is one-version, and nonblocking. In other words, it sacrifices one round in latency compared to the theoretically LO protocol, but retains the low cost of PUTs in non-LO designs.

Contrarian uses the coordinator-based approach described in Section 3 and shown in Figure 2. The client identifies the partitions to read from, and selects one of them as the coordinator for the ROT. The client sends its dependency vector to the coordinator, which picks the snapshot corresponding to the ROT and sends it back to the client. The client then contacts the partitions involved in the ROT, communicating the list of keys to be read and the snapshot of the ROT.

The ROT protocol uses a vector SV to encode a snapshot. The local entry of SV is the maximum between the clock at the coordinator and the highest local timestamp seen by the client. The remote entries of SV are given by the entry-wise maximum between the GSS at the coordinator and the dependency vector of the client. Upon receiving a ROT request with snapshot SV, a partition moves its own clock to match the local entry of SV, if needed. A version Y belongs to the snapshot encoded by SV if $Y.DV \leq SV$. For any requested key, a partition returns the version belonging with the highest timestamp that belongs to the specified snapshot.

Freshness of the snapshots. The GSS is computed by means of the minimum operator. Because logical clocks on different partitions may advance at different paces, a laggard partition in one DC can slow down the progress of the GSS, thus increasing the staleness of the ROT snapshots. A solution to this problem is to use loosely synchronized physical clocks [3, 26, 27]. However, physical clocks cannot be moved forward to match the timestamp of an incoming ROT, which can compromise the nonblocking property [3].

To achieve fresh snapshots and nonblocking ROTs, Contrarian uses Hybrid Logical Physical Clocks (HLC) [35]. In brief, an HLC is a logical clock that generates timestamps by taking the maximum between the local physical clock and the highest timestamp seen by the node plus one. On the one hand, HLCs behave like logical clocks, so a server can move its clock forward to match the timestamp of an incoming ROT request, thereby preserving the nonblocking behavior of ROTs. On the other hand, HLCs behave like physical clocks, because they advance even in absence of events and inherit the (loosely) synchronized nature of the underlying physical clocks. Hence, the stabilization protocol identifies fresh snapshots. The correctness of Contrarian does not depend on the synchronization of the clocks, and Contrarian preserves its properties even if it were to use plain logical clocks.

4.3 ROT Properties

Nonblocking. Contrarian implements nonblocking ROTs by using logical clocks and by including in the snapshot assigned to a ROT only remote versions that are stable in the DC. Then, Contrarian's ROT protocol is nonblocking, because *i*) partitions can move the value of their local clock forward to match the local entry of SV, and *ii*) the remote entries of SV correspond to a causally consistent snapshot of remote versions that are already present in the DC.

Despite embracing the widely-used coordinator-based approach to ROTs, nonblocking ROTs in Contrarian improve upon existing designs. These designs can block (or delay by retrying) ROTs due to clock skew [3], to wait for the receipt of remote version [26, 27, 46, 60], or to wait for the completion of a PUT operation in the DC where the ROT takes place [4].

One-version. Contrarian achieves the one-version property, because partitions read the version with the highest timestamp within the snapshot proposed by the coordinator.

Eventual visibility. Contrarian achieves eventual visibility, because every version is eventually included in every snapshot corresponding to a ROT. Let X be a version created on partition p_x in DC_i, and let ts be its timestamp. p_x piggybacks its clock value (that is at least ts) during the stabilization protocol. Therefore, each partition in DC_i sets its clock to be at least ts.

By doing so, Contrarian ensures that every coordinator in DC_i eventually proposes a ROT snapshot whose local entry is $\geq ts$. Furthermore, every partition in DC_i eventually sends a message with timestamp $\geq ts$ to its replicas (either by a replication or a heartbeat message). Hence, the i-th entry in each remote VV eventually reaches the value ts. Therefore, every i-th entry in the GSS computed in every DC eventually reaches the value ts. Because the remote entries of ROT snapshots are computed starting from the GSS, Contrarian ensures that X and its dependencies are eventually stable in remote DCs and included in all ROT snapshots.

5. EXPERIMENTAL STUDY

We show that the resource demands to perform PUT operations in the LO design are in practice so high that they not only affect the performance of PUTs, but also the performance of ROTs, even with read-heavy workloads. In particular, with the exception of scenarios corresponding to very modest loads, where the two designs are comparable, Contrarian achieves ROT latencies that are lower than the state-of-the-art LO design. In addition, Contrarian achieves higher throughput for all workloads we consider.

5.1 Experimental environment

Implementation and optimizations. We implement Contrarian and the COPS-SNOW design in the same C++ codebase. Clients and servers use Google Protocol Buffers [28] for communication. We call CC-LO the system that implements the design of COPS-SNOW. We improve its performance over the original design by more aggressive eviction of transactions from the old readers record. Specifically, we garbage-collect a ROT id after 500 msec from its insertion in the readers record of a key (vs. the 5 seconds of the original implementation), and we enforce that each readers check response message contains at most one ROT id per client, i.e., the one corresponding to the most recent ROT of that client. These two optimizations reduce by one order of magnitude the number of ROT ids exchanged, approaching the lower bound we derive in Section 6.

We use NTP [52] to synchronize clocks in Contrarian, the stabilization protocol is run every 5 msec, and a partition

Table 1: Workload parameters considered in the evaluation. The default values are given in bold.

Parameter	Definition	Value	Motivation
Write/read ratio (**w**)	#PUTS/(#PUTs+#individual reads)	0.01	Extremely read-heavy workload
		0.05	Default read-heavy parameter in YCSB [21]
		0.1	Default parameter in COPS-SNOW [42]
Size of a ROT (**p**)	# Partitions involved in a ROT	**4**,8,24	Application operations span multiple partitions [50]
Size of values (**b**)	Value size (in bytes). Keys take 8 bytes.	**8**	Representative of many production workloads [6, 50, 56]
		128	Default parameter in COPS-SNOW [42]
		2048	Representative of workloads with large items
Skew in key popularity (**z**)	Parameter of the zipfian distribution.	**0.99**	Strong skew typical of many production workloads [6, 14]
		0.8	Moderate skew and default in COPS-SNOW [42]
		0	No skew (uniform distribution) [14]

sends a heartbeat if it does not process a PUT for 1 msec (similarly to previous systems [27, 60]).

Platform. We use an AWS platform composed of up to 3 DCs (Virginia, Oregon and Ireland). Each DC hosts 45 server virtual machines (VM), corresponding to 45 partitions, and 45 client VMs. We use c5.xlarge instances (4 virtual CPUs and 8 GB of RAM) that run Ubuntu 16.04 and a 4.4.0-1022-aws Linux kernel.

Methodology. We generate different loads for the system by spawning different numbers of client threads, which issue operations in a closed loop. We spawn from 1 to 1,800 client threads per DC, uniformly distributed across the client VMs.

Each point in the performance plots we report corresponds to a different number of client threads (starting from 1 per DC). We spawn as many client threads as necessary to saturate the resources of the systems. Increasing the number of threads past that point leads the systems to deliver lower throughput despite serving a higher number of client threads. We do not report performance corresponding to severe overload. Therefore, the performance plots of the two systems may have a different number of points for the same workload, because the systems may saturate with different number of client threads.

Experiments run for 90 seconds. We have run each experiment up to 3 times, with minimal variations between runs, and we report the median result.

Workloads. Table 1 summarizes the workload parameters we consider. We use read-heavy workloads, in which clients issue ROTs and PUTs according to a given write/read ratio (w), defined as #PUT/(#PUT + #READ). A ROT reading k keys counts as k READs. ROTs span a target number of partitions (p), chosen uniformly at random, and read one key per partition. Keys in a partition are chosen according to a zipfian distribution with a given parameter (z). Every partition stores 1M keys. Keys are 8 bytes long, and items have a constant size (b).

We use a default workload with w = 0.05, i.e., the default value for the read-heavy workload in YCSB [21]; z = 0.99, which is representative of skewed workloads [6]; p = 4, which corresponds to small ROTs (which exacerbate the extra communication in Contrarian); and b = 8, as many production workloads are dominated by tiny items [6]. We generate additional workloads by changing the value of one parameter at a time, while keeping the other parameters at their default values.

Performance metrics. We focus our study on the latencies of ROTs, because, by design, CC-LO favors ROT latencies over PUTs. As an aside, in our experiments CC-LO incurs up to one order of magnitude higher PUT latencies than Contrarian. We study how the latency of ROTs varies as a function of system throughput and workload parameters. We measure the throughput as the number of PUTs and ROTs performed per second.

We focus on 95-th percentile latency, which is often used to study the performance of key-value stores [38, 50]. By reporting the 95-th percentile, we capture the behavior of the vast majority of ROTs, and factor out the dynamics that affect the very tail of the response time distribution. We report and discuss the average and the 99-th percentile of the ROT latencies for a subset of the experiments. As a final note, the worst-case latencies achieved by Contrarian and CC-LO are comparable, and on the order of a few hundreds of milliseconds.

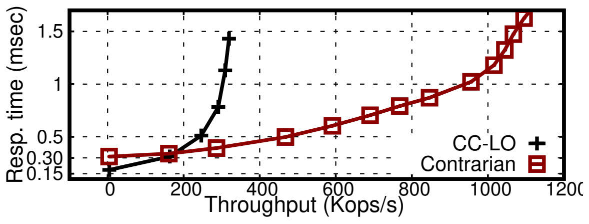
5.2 Default workload

Figures 4a and 4b show the performance of Contrarian and CC-LO with the default workload running on 1 DC and on 3 DCs, respectively. Figure 4c reports the readers check overhead in CC-LO in a single DC. Figure 4d depicts the average and the 99-th percentile of ROT latencies in a single DC.

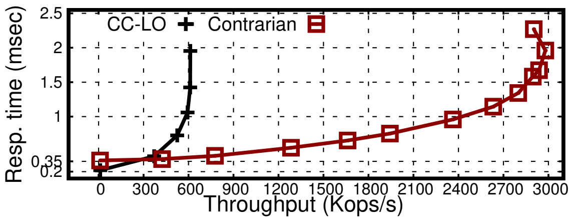
Latency. Contrarian achieves lower latencies than CC-LO for nontrivial throughput values. Contrarian achieves better latencies than CC-LO by avoiding the extra overhead incurred by performing the readers check. This overhead induces higher resource utilization, and hence higher contention on physical resources. Ultimately, this leads to higher latencies, even for ROTs.

ROTs in Contrarian become faster than in CC-LO starting from loads corresponding to ≈200 Kops/s in the single-DC case and to ≈350 Kops/s in the geo-replicated case, i.e., ≈ 17% and ≈ 12% of the maximum throughput achievable by Contrarian. Contrarian achieves better latencies than CC-LO in the geo-replicated case starting from a relatively lower load than in the single-DC case. This result is due to the higher replication costs in CC-LO, which has to communicate the dependency list of a replicated version, and perform the readers check in all DCs. CC-LO achieves faster ROTs than Contrarian only at very moderate loads, which correspond to under-utilization scenarios. At the lowest load (corresponding to a single thread running per DC), in the single-DC case ROTs in Contrarian take 0.31 msec vs. 0.18 in CC-LO; in the geo-replicated scenario, ROTs in Contrarian take 0.36 msec vs. 0.22 in CC-LO.

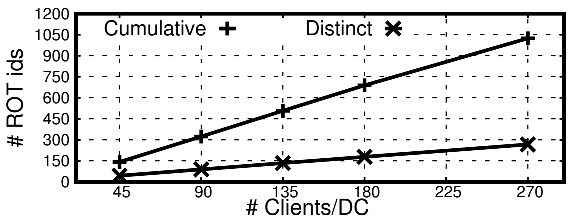
Throughput. Contrarian achieves a higher throughput than CC-LO. Contrarian's maximum throughput is 3.7x CC-LO's in the 1-DC case (1150 Kops/s vs. 310), and 5x in the 3-DC case (3000 Kops/s vs. 600). In addition, Contrarian achieves a 2.6x throughput improvement when scaling from



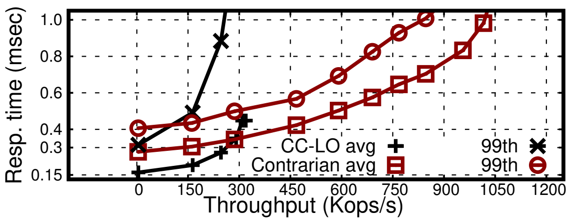
(a) Throughput vs. 95-th percentiles of ROT latencies (1 DC).



(b) Throughput vs. 95-th percentiles of ROT latencies (3 DCs).

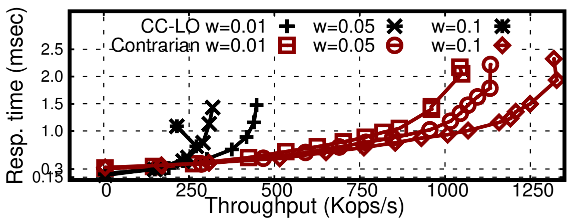


(c) Old readers check overhead in CC-LO (1 DC).

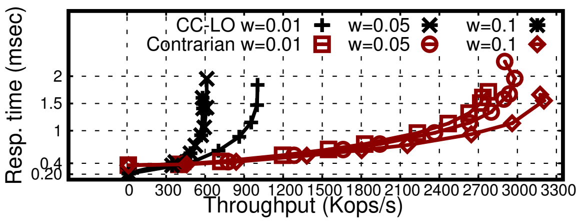


(d) Throughput vs. average/99-th percentile of ROT latencies (1 DC).

Figure 4: Performance with the default workload. Contrarian achieves better latencies (except at very modest load) and higher throughput (a,b) by avoiding the extra overhead posed by CC-LO on PUTs (c). The effects of the overhead incurred by CC-LO is more evident at the tail of the latency distribution (d).



(a) 1 DC.



(b) 3 DCs

Figure 5: Performance with different w/r ratios. Contrarian achieves lower ROT latencies than CC-LO, except at very moderate load and for the most read-heavy workload. Contrarian also consistently achieves higher throughput. Higher write intensities hinder the performance of CC-LO because the readers check is triggered more frequently.

1 to 3 DCs. By contrast, CC-LO improves its throughput only by ≈2x. Contrarian achieves higher throughput values and better scalability by avoiding the resource utilization to perform the readers check and by implementing a lightweight stabilization protocol.

Overhead analysis. Figure 4c reports, as a function of the number of client threads, the average number of ROT ids collected during a readers check. The same ROT id can appear in the readers set of multiple keys. Hence, we report both the total number of ROT ids collected, and the number of distinct ones. The overhead of a readers check grows linearly with the number of clients in the system. This result matches our theoretical analysis (Section 6) and highlights the inherent scalability limitations of LO. For example, at peak throughput, corresponding to 270 client threads, a readers check collects on average 1023 ROT ids, of which 267 are distinct. Using 8 bytes per ROT id, the readers check causes on average 9KB of data to be collected.

Tail vs. average latency. We now investigate the effect of Contrarian's and CC-LO's design on the distribution of ROT latencies. To this end, we report in Figure 4d the average ROT latency and the 99-th percentile (1 DC). In terms of the 99-th percentile, Contrarian wins over CC-LO starting at a load value of approximately 100 Kops/s, much lower than the load value at which Contrarian wins over CC-LO for the 95-th percentile. In terms of the average, CC-LO's wins up to 290 Kops/s, which is close to CC-LO's peak throughput. This experiment shows that the extra overhead imposed by LO does not affect all ROTs in the same way, and that, in particular, its effect is more evident at the tail of the distribution of ROT latencies. This result is explained as follows. At one end of the spectrum, some ROTs do not experience any reders check overhead, and benefit from the one-round nature of CC-LO. Since the average latency is computed over all ROTs, these "lucky" ROTS figure in the calculation, resulting in a low average latency for CC-LO. At the other end, some ROTs experience a great deal of readers check overhead, and this overhead dwarfs the benefit of the one-round natire of CC-LO. The 99-th percentile measures the latency of these "unlucky" ROTs. More precisely, it is the lower bound on the latency experienced by the slowest 1% of the ROTs. Since performance of key-value stores is often quoted in terms of tail latencies, we argue that Contrarian offers an important advantage in this regard.

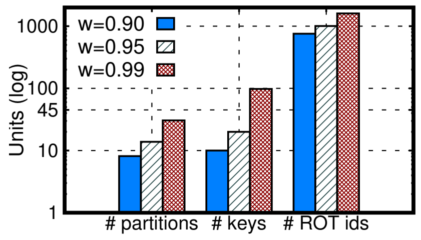


Figure 6: Average overhead per readers check in CC-LO as a function of w: # partitions involved, # keys checked and # ROT ids exchanged (1 DC, 270 client threads).

5.3 Effect of write intensity

Figure 5 shows how the write intensity (w) of the workload affects the performance of the systems in the 1-DC case (a) and in the 3-DC case (b). Figure 6 reports the effect of write intensity on the overhead to perform the readers check in CC-LO (1 DC, 270 client threads).

Latency. Similar to what was seen with the default workload, for nontrivial load conditions Contrarian achieves lower ROT latencies than CC-LO both with and without geo-replication, and with all write intensity values. The best case for CC-LO is with w = 0.01, when readers checks are more rare.

Throughput. Contrarian achieves a higher throughput than CC-LO in all scenarios, from a minimum of 2.33x in the 1-DC case for w=0.01 (1050 vs. 450 Kops/s) to a maximum of 3.2x in the 3-DC case for w=0.1 (3200 vs. 1000 Kops/s). The throughput of Contrarian grows with the write intensity, because PUTs only touch one partition and are thus faster than ROTs. Instead, higher write intensities hinder the performance of CC-LO, because they cause more frequent execution of the expensive readers check.

Overhead analysis. Surprisingly, the latency benefits of CC-LO are not very pronounced, even at the lowest write intensity. The explanation resides in the inherent tension between the frequency of writes and their costs, as shown Figure 6. On the one hand, a high write intensity leads to frequent readers check on relatively few keys (because few keys are read before performing a PUT). As a result, fewer partitions need to be contacted on a readers check and fewer ROT ids exchanged. On the other hand, a low write intensity leads to more infrequent readers checks, that, however, are more costly, because they lead to contacting more partitions and exchanging more ROT ids.

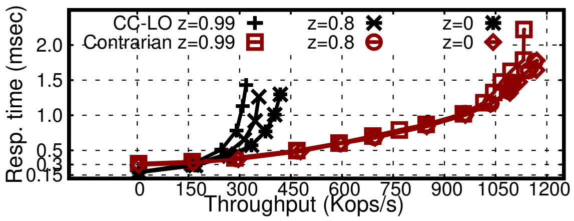
5.4 Effect of skew in data popularity

Figure 7 depicts how the performance (a) and the readers check overhead (b) vary with the skew in data popularity (z). We analyze the single-DC case to factor out replication dynamics (which are different in Contrarian and CC-LO) and to focus on the *inherent* costs of LO.

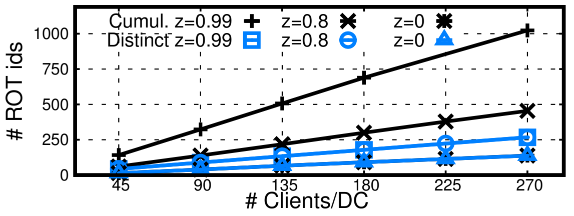
Latency. Similar to earlier results, Contrarian achieves ROT latencies that are lower than CC-LO's for nontrivial load conditions (> 150 Kops/s, i.e., less than 1/7 of Contrarian's maximum throughput).

Throughput. Increased data popularity skew has little effect on Contrarian, but it hampers the throughput of CC-LO. The performance of CC-LO degrades, because a higher skew causes longer causal dependency chains among operations [11, 27], leading to a higher overhead incurred by the readers checks.

Overhead analysis. With low skew, a key x is infrequently accessed, so it is likely that many entries in the reader list of x can be garbage-collected by the time x is involved in a readers check. With higher skew levels, a few hot keys are accessed most of the time, which causes the old reader record to contain many fresh entries. High skew also leads to more duplicates in the ROT ids retrieved from different partitions, because the same ROT id is likely to be present in many the old reader record. Figure 7b portrays these dynamics. The reported plots also show that, at any skew level, the number of ROT ids exchanged during a readers check grows linearly with the number of clients (which matches the results of our theoretical analysis in Section 6).



(a) Throughput vs. 95-th percentile of ROT latencies.



(b) Old readers check overhead in CC-LO.

Figure 7: Effect of the skew in data popularity (1 DC). Skew hampers the performance of CC-LO (a), as it leads to long causal dependency chains among operations and thus to much information exchanged during the readers check (b).

5.5 Effect of size of transactions

Figure 8 shows the performance of the systems while varying the number of partitions involved in a ROT (p). We again focus on the single-DC platform.

Latency. Contrarian achieves ROT latencies that are lower than or comparable to CC-LO's for any number of partitions involved in a ROT. The latency benefits of CC-LO over Contrarian at low load decrease as p grows, because contacting more partitions amortizes the impact of the extra communication round needed by Contrarian to execute a ROT. At the lowest load, with $p = 4$, the latency of ROTs in Contrarian is 1.72x the one in CC-LO (0.31 msec vs. 0.18). With $p = 32$, instead, the latency of ROTs in Contrarian is only 1.46x the one in CC-LO (0.6 msec vs. 0.41).

Throughput. Contrarian achieves a throughput increase with respect to CC-LO that ranges from 3.4x ($p = 4$) to 4.25 ($p = 32$). Higher values of p amortize the extra resource demands for contacting the coordinator in Contrarian, and hence allow Contrarian to achieve a comparatively higher throughput with respect to CC-LO.

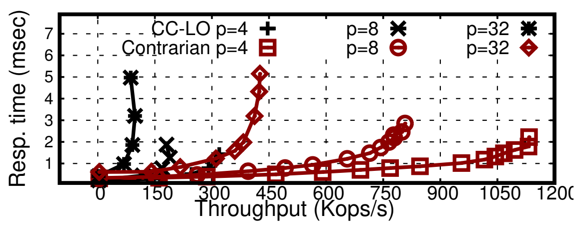


Figure 8: Throughput vs. 95-th percentile of ROT latencies while varying # partitions involved in a ROT (1 DC).

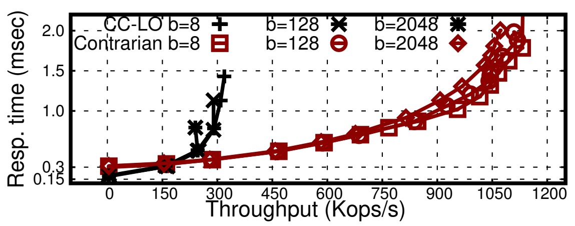


Figure 9: Throughput vs 95-th percentile of ROT latencies while varying the size of items (1 DC).

5.6 Effect of size of values

Figure 9 reports the performance of Contrarian and CC-LO when manipulating values of different sizes (b). Larger values naturally result in higher CPU and network costs for marshalling, unmarshalling and transmission operations. As a result, the maximum throughput of the systems decreases and the latency increases.

Contrarian maintains its performance lead over CC-LO for any value size we consider, except for throughput values lower than 150 Kops/s. We could only experiment with values of size up to 2 KB because of memory limitations on our machines. We argue that with even bigger values the performance differences between the two systems would decrease. With bigger values, in fact, the performance of the two systems would be primarily determined by the resource utilization to store and communicate values, rather than by differences in the designs.

6. THEORETICAL RESULTS

Our experimental study shows that the state-of-the-art CC design for LO ROTs delivers sub-optimal performance, caused by the overhead (imposed on PUTs) for dealing with old readers. One can, however, conceive of alternative CC implementations. For instance, rather than storing old readers at the partitions, one could contemplate an implementation which stores old readers at the client, when the client does a PUT. This client could then forward this information to other partitions on subsequent PUTs. Albeit in a different manner, this implementation still communicates the old readers between the partitions where causally related PUTs are performed. One may then wonder: is there an implementation that avoids this overhead, in order not to exhibit the performance issues we have seen with CC-LO in Section 5?

We now address this question. We show that the extra overhead on PUTs is *inherent* to LO. Furthermore, we show that the extra overhead grows with the number of clients, implying the growth with the number of ROTs and echoing the measurement results we have reported in Section 5. Our theorem applies to the system model described in Section 2. We refine some aspects of the model for the purpose of establishing our theoretical results. We provide a more precise system model in Section 6.1, and a more precise definition of LO in Section 6.2. Then we present our theorem in Section 6.3 and its proof in Section 6.4.

6.1 Assumptions

For the ease of definitions as well as proofs, we assume the existence of an accurate real-time clock to which no partition or client has access. When we mention time, we refer to this clock. Furthermore, when we say that two client operations are concurrent, we mean that the durations of the two operations overlap according to this clock.

Among other things, this clock allows us to give a precise definition of eventual visibility. If PUT(x, X) starts at time T (and eventually ends), then there exists a finite time $\tau_X \geq T$ such that any ROT that reads x and is issued at time $t \geq \tau_X$ returns either X or some X' such that PUT(x, X') starts no earlier than T; we say X is *visible* since τ_X.

We assume the same APIs as described in Section 2.1. Clients and partitions exchange messages whose delays are finite, but can be unbounded. The clock drift between the local clocks of clients and partitions can be arbitrarily large and infinite. We assume that reads do not rely on the clients' local clocks. By doing so, eventual visibility does not depend on the advancement of the clients' clock, and depends solely on the state of the key-value store and the actions undertaken by the partitions implementing it.

We assume that an idle client does not send messages. When performing an operation on some keys, a client sends messages only to the partitions which store values for these keys. Vice versa, a partition sends messages to client c only when responding to an operation issued by c. Clients do not communicate with each other, and issue a new operation only after its previous operation returns. We assume at least two partitions and a potentially growing number of clients.

6.2 Properties of LO ROTs

We adopt the definition of LO ROTs from [42], which refers to three properties: *one-round*, *one-version*, and *non-blocking*.

• *One-round*: For every ROT α of client c, c sends one message to each partition p involved in α and receives one message from p.

• *Nonblocking*: For any partition p to which c sends a message, p eventually sends one message (the one defined in the one-round property) to c, even if p receives no message from a partition during α. This definition essentially states that a partition cannot communicate with other partitions when serving a ROT to decide which version of a key to return to the ROT. This definition extends the more restrictive one given in Section 2, which also disallows blocking p, e.g., by the acquisition of a lock or for the expiration of a timer. To establish our theoretical results, it suffices to disallow blocking p by inter-partition communication during a ROT. Because our proof holds for a more general definition of non-blocking, it implies that the proof also holds for the more restrictive definition in Section 2.

• *One-version*: Let M be the maximum amount of information that, for each ROT α of client c, can be calculated by any implementation algorithm based on the messages which c receives during α. [4] Then, given any (non-empty) subset of partitions, Par, and given the messages which c receives from Par during α, M contains only one version per key for the keys which Par stores and α reads.

6.3 The cost of *LO*

Definitions. We introduce some additional terminology before we state the theorem

We say that a PUT operation α *completes* if *i*) α returns to the client that issued α; and *ii*) the value written by α becomes visible. We say that a PUT operation α is *dangerous* if α causally depends on some PUT that overwrites a non-$\perp$ value.

If client c issues a ROT operation that reads x, then we say c is a reader of x. We call client c an *old reader* of x, with respect to $\mathrm{PUT}(y, Y_1)$,[5] if c issues a ROT operation which (1) is concurrent with $\mathrm{PUT}(x, X_1)$ and $\mathrm{PUT}(y, Y_1)$ and (2) returns X_0, where $X_0 \rightsquigarrow X_1 \rightsquigarrow Y_1$.

Theorem 1 (Cost of LO ROTs)**.** *Achieving LO ROT requires communication, potentially growing linearly with the number of clients, before every dangerous PUT completes.*

Intuition of the result. After a dangerous PUT on y completes, partition p_y needs to choose between the newest version of y (i.e., the one written by the dangerous PUT) and a previous one to be returned to an incoming ROT. The knowledge of the old readers with respect to the dangerous PUT allows p_y to determine a version.

As the ROT must be nonblocking, p_y cannot wait for messages containing that information during the ROT protocol after the dangerous PUT completes. As the ROT must be one-round and one-version, the client which requests the ROT cannot choose between versions sent in different rounds or between multiple versions sent in the same round.

Thus p_y needs the knowledge of old readers before or at the latest by the time the dangerous PUT on y completes. Assuming that there are D clients and since in the worst case they can all be old readers, an LO ROT protocol needs, in the worst case, at least D bits of information to encode the old readers.

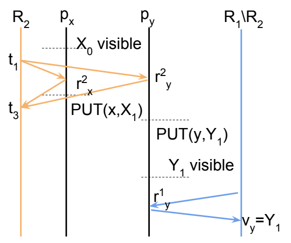
6.4 Proof

Proof overview. The proof assumes the scenario in Figure 10, which depicts executions in which $X_0 \rightsquigarrow X_1 \rightsquigarrow Y_1$. Without loss of generality we consider that such executions are the result of client c_w doing four PUT operations in the following order: $\mathrm{PUT}(x, X_0)$, $\mathrm{PUT}(y, Y_0)$, $\mathrm{PUT}(x, X_1)$ and $\mathrm{PUT}(y, Y_1)$; c_w issues each PUT (except the first one) after the previous PUT completes.

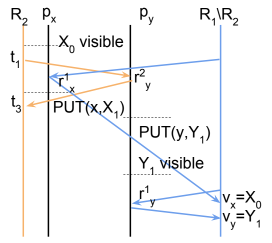
To prove Theorem 1, we consider the worst case: all clients except c_w can be readers. We identify similar executions where a different subset of clients are readers. Let $\mathcal{D}$ be the set of all clients except c_w. We construct the set $\mathcal{E}$ such that each execution has one subset of $\mathcal{D}$ as readers. Hence $\mathcal{E}$

[4]As values can be encoded in different ways in messages, we use the amount of information in the messages. For example, if a message contains X_1 and $X_1 \oplus X_2$, then in the plaintext, there is only one version, yet some implementation can calculate two versions from the plaintext. Our definition of the one-version property excludes such messages as well as such implementations.

[5]The definition of an old reader of x here specifies a certain PUT on y is to emphasize the causal relation $X_1 \rightsquigarrow Y_1$.



(a) Execution E_2



(b) Execution E^* (r_x^2 omitted)

Figure 10: Two (in)distinguishable executions in the proof of Theorem 1.

contains $2^{|\mathcal{D}|}$ executions in total. We later show that for at least one execution in $\mathcal{E}$ the communication carrying readers grows linearly with $|\mathcal{D}|$, and thereby prove Theorem 1.

$2^{|\mathcal{D}|}$ executions $\mathcal{E}$. Each execution $E \in \mathcal{E}$ is based on a subset R of $\mathcal{D}$ as readers. Every client c in R issues $\mathrm{ROT}(x, y)$ at the same time t_1. By the one-round property, c sends two messages $m_{x,req}$, $m_{y,req}$ to p_x and p_y respectively at t_1. We denote the event that p_x receives $m_{x,req}$ by r_x, the event that p_y receives $m_{y,req}$ by r_y. By the nonblocking property, p_x and p_y can be considered to receive messages from c and send messages to c at the same time t_2.[6] Finally, c receives messages from p_x and p_y at the same time t_3. We order events as follows: X_0 and Y_0 are visible, t_1, $r_x = r_y = t_2$, $\mathrm{PUT}(x, X_1)$ is issued, t_3, $\mathrm{PUT}(y, Y_1)$ is issued. Let τ_{Y_1} be the time when $\mathrm{PUT}(y, Y_1)$ completes. For every execution in $\mathcal{E}$, t_1, t_2, t_3 take the same values while τ_{Y_1} actually denotes the maximum value of all executions in $\mathcal{E}$.

To emphasize the burden on p_y, we consider communication that *precedes* a message that p_y receives: we say message a precedes message b if (1) some process p sends b after p receives a, or (2) $\exists$ message c such that a precedes c and c precedes b. To simplify the terminology, we also say a precedes b if a and b refer to the same message. The executions in $\mathcal{E}$ are the same until time t_1. Since t_1, these executions, especially, the communication between p_x and p_y may change. We construct all executions in $\mathcal{E}$ together: if at some time point, in one execution, some server sends a message, then we construct all other executions such that the same server sends the same message except that the server is p_x, p_y or contaminated by p_x or p_y. By contamination, we mean that at some point, p_x or p_y sends message m but we are unable to construct all other executions to do the same; then the message m and server s which receives m are contaminated and s can further contaminate other servers. In our construction, we focus on the non-contaminated messages which are received at the same time across all executions in $\mathcal{E}$. For other messages, if in at least two executions, the same contaminated message m can be sent, then we let m be received at the same time across these executions and be considered as a non-contaminated message; otherwise, we do not restrict the schedule.

We show that the worst-case execution exists, as promised by our proof overview, in our construction of $\mathcal{E}$. To do so, we first show a property of $\mathcal{E}$; i.e., for any two executions

[6]p_x and p_y may receive messages at different times, and spend unbounded time on local computation but eventually send messages to c; in both cases, the proof still holds. The same time t_2 is assumed for the simplicity of presentation.

E_1, E_2 in $\mathcal{E}$ (with different readers), the communication of p_x and p_y must be different, as formalized in Lemma 1.[7]

Lemma 1 (Different readers, different messages)**.** *Consider any two executions $E_1, E_2 \in \mathcal{E}$. In $E_i, i \in \{1,2\}$, denote by M_i the messages which p_x and p_y send to a process not in $\mathcal{D}$ and which precede some message that p_y receives during $[t_1, \tau_{Y_1}]$ in E_i, and denote by str_i the concatenation of ordered messages in M_i ordered by the time when every message is sent. Then $str_1 \neq str_2$.*

The main intuition behind Lemma 1 is that if communication were the same regardless of readers, p_Y would be unable to distinguish readers from *old* readers. Suppose now by contradiction that $str_1 = str_2$. Then our construction of $\mathcal{E}$ allows us to construct an special execution E^* based on E_2 (as well as E_1). Let the subset of $\mathcal{D}$ for E_i be R_i for $i \in \{1,2\}$. W.l.o.g., $R_1 \backslash R_2 \neq \emptyset$. We construct E^* such that clients in $R_1 \backslash R_2$ are old readers (and show that E^* breaks causal consistency due to old readers).

Execution E^* with old readers. In E^*, both R_1 and R_2 issue ROT(x,y) at t_1. To distinguish between events (and messages) resulting from R_1 and R_2, we use superscripts 1 and 2 to denote the events, respectively. For simplicity of notations, in E_2, we call the two events at the server-side (i.e., p_x and p_y receive messages from R_2 respectively) also r_x^2 and r_y^2, illustrated in Figure 10a. In E^*, we now have four events at the server-side: r_x^1, r_y^1, r_x^2, r_y^2. We construct E^* based on E_2 by scheduling r_x^1 and r_y^2 in E^* at t_2 (the same time as r_x^2 and r_y^2 in E_2), and postponing r_y^1 (as well as r_x^2), as illustrated in Figure 10b. The ordering of events in E^* is thus different from E_2. More specifically, the order is: X_0 and Y_0 are visible, t_1, $r_x^1 = r_y^2 = t_2$, PUT(x, X_1) is issued, PUT(y, Y_1) is issued, τ_{Y_1}, r_y^1 (for every client in $R_1 \backslash R_2$ as r_y^2 has occurred), r_x^2 (for every client in $R_2 \backslash R_1$, not shown in Figure 10b), $R_1 \backslash R_2$ returns ROT. By asynchrony, the order is legitimate, which results in old readers $R_1 \backslash R_2$.

Proof of Lemma 1. Our proof is by contradiction. As $str_1 = str_2$, according to our construction, p_y does not receive any message preceded by some different contaminated message in E_1 and E_2. Therefore even if we replace r_x^2 in E_2 for r_x^1 in E^* (as in E_1), then by τ_{Y_1}, p_Y is unable to distinguish between E_2 and E^*.

Previously, our construction of E_2 is until τ_{Y_1}. Let us now extend E_2 so that E_2 and E^* are the same after τ_{Y_1}. Namely, in E_2, after τ_{Y_1}, every client $c_1 \in R_1 \backslash R_2$ issues ROT(x,y); and as illustrated in Figure 10, r_y^1 is scheduled at the same time in E_2 and in E^*.

Let $\vec{v}$ be the return value of c_1's ROT in either execution. By eventual visibility, in E_2, $v_y = Y_1$. We now examine E^*. By eventual visibility, as t_1 is after X_0 and Y_0 are visible, $v_x, v_y \neq \perp$. As r_x^1 is before PUT(x, X_1) is issued, $v_x \neq X_1$. By p_y's indistinguishability between E_2 and E^*, and according to the one-version property, $v_y = Y_1$ as in E_2. Thus in E^*, $v_x = X_0$ and $v_y = Y_1$, a snapshot that is not causally consistent. A contradiction. ☐

[7]Lemma 1 abstracts the way of communication between p_x and p_y so that it is independent of certain implementations, and covers the following example implementations of communication for old readers as in CC-LO, as the example introduced at the beginning of this section, as well as the following: p_y keeps asking p_x whether a reader of y is a reader which returns X_0 to determine whether all readers that return X_0 have arrived at p_y (so that there is no old reader with respect to Y_1).

Lemma 1 demonstrates a property for any two executions in $\mathcal{E}$, which implies another property of $\mathcal{E}$: if for any two executions, communication has to be different, then for all executions, the number of possibilities of what is communicated grows with the number of elements in $\mathcal{E}$. Recall that $|\mathcal{E}|$ is a function of $|\mathcal{D}|$. Hence, we connect the communication and $|\mathcal{D}|$ in Lemma 2.

Lemma 2 (Lower bound on the cost)**.** *Before PUT(y, Y_1) completes, in at least one execution in $\mathcal{E}$, the communication of p_x and p_y takes at least $\mathcal{L}(|\mathcal{D}|)$ bits where $\mathcal{L}$ is a linear function.*

Proof of Lemma 2. We index each execution E by the set R of clients which issue ROT(x,y) at time t_1. We have therefore $2^{|\mathcal{D}|}$ executions: $\mathcal{E} = \{E(R) | R \subseteq \mathcal{D}\}$. Let $b(R)$ be the messages which p_x and p_y send in $E(R)$ as defined in Lemma 1, and let $B = \{b(R) | R \subseteq \mathcal{D}\}$. By Lemma 1, we can show that $\forall b_1, b_2 \in B, b_1 \neq b_2$. Then $|B| = |\mathcal{E}| = 2^{|\mathcal{D}|}$. Therefore, it is impossible that every element in B has fewer than $|\mathcal{D}|$ bits. In other words, in $\mathcal{E}$, we have at least one execution $E = E(R)$ where $b(R)$ takes at least $\log_2(2^{|\mathcal{D}|}) = |\mathcal{D}|$ bits, a linear function in $|\mathcal{D}|$. ☐

Recall that $|\mathcal{D}|$ is a variable that grows linearly with the number of clients. Thus following Lemma 2, we find $\mathcal{E}$ contains a worst-case execution that supports Theorem 1 and we thus complete the proof of Theorem 1.

Remark on implementations. The proof shows the necessary communication of readers when each client issues one operation. Here we want to make the link back to the implementation of LO ROTs in CC-LO. One may wonder in particular about the relationship between the transaction identifiers that are sent as old readers in CC-LO, and the worst-case communication linear in the number of clients derived in the theorem. In fact, the CC-LO implementation considers that clients may issue multiple transactions at the same time, and then different ROTs of a single client should be considered as different readers, hence the use of transaction identifiers to distinguish one from another.

A final comment is on a straw-man implementation where each operation is attached to the output of a Lamport Clock [37] (called logical time below) alone. Such implementation (without communication of potentially old readers) still fails. The problem is that the number of increments in logical time after ROTs is at most the number of all ROTs, i.e., $|\mathcal{D}|$. Then for some E_1 and E_2, Lemma 1 does not hold, i.e., the communication is the same. Although when issuing the ROT, client c in $R_1 \backslash R_2$ can send logical time to servers, the logical time sent in E_2 and E^* is the same and thus does not help p_y to distinguish between E_2 and E^*, resulting in the violation of causal consistency again. Hence communication of readers, as Theorem 1 indicates, is still required for this straw-man implementation.

7. RELATED WORK

CC systems. Table 2 classifies existing systems with ROT support according to the cost of performing ROT and PUT operations. COPS-SNOW is the only LO system. COPS-SNOW achieves LO at the expense of more costly writes, which carry detailed dependency information and incur extra communication overhead.

Table 2: Characterization of CC systems with ROTs support, in a geo-replicated setting. N, M and K represent, respectively, the number of partitions, DCs, and clients in a DC. † indicates a single-master system, and P represents the number of DCs that act as master for at least one partition. $c \leftrightarrow s$, resp., $s \leftrightarrow s$, indicates client-server, resp. inter-server, communication.

System	ROT latency optimality			Write cost				Clock
	Nonblocking	#Rounds	#Versions	Communication		Meta-data		
				$c \leftrightarrow s$	$s \leftrightarrow s$	$c \leftrightarrow s$	s↔s	
COPS [40]	✓	≤ 2	≤ 2	1	-	|deps|	-	Logical
Eiger [41]	✓	≤ 2	≤ 2	1	-	|deps|	-	Logical
ChainReaction [4]	✗	≥ 2	1	1	≥ 1	|deps|	M	Logical
Orbe [26]	✗	2	1	1	-	NxM	-	Logical
GentleRain [27]	✗	2	1	1	-	1	-	Physical
Cure [3]	✗	2	1	1	-	M	-	Physical
OCCULT† [46]	✓	≥ 1	$\geq$1	1	-	O(P)	-	Hybrid
POCC [60]	✗	2	1	1	-	M	-	Physical
COPS-SNOW [42]	✓	1	1	1	O(N)	|deps|	O(K)	Logical
Contrarian	✓	2	1	1	-	M	-	Hybrid

ROTs in COPS and Eiger might require two rounds of client-server communication and rely on fine-grained protocols to track and check the dependencies of replicated updates (see Section 3), which limit their scalability [3, 26, 27, 61]. ChainReaction uses a potentially-blocking and potentially multi-round protocol based on a per-DC *sequencer* node. Orbe, GentleRain, Cure and POCC use a coordinator-based approach similar to Contrarian but use physical clocks and hence may block ROTs because of clock skew. In addition, Orbe and Gentlerain may block ROTs to wait for the receipt of remote updates. Occult uses a primary-replica approach and uses HLCs to avoid blocking due to clock skew. Occult implements ROTs that run in potentially more than one round and that potentially span multiple DCs (i.e., it does not tolerate cross-DC network partitions).

By contrast, Contrarian uses HLCs to implement ROTs that are nonblocking, one-version, complete in 2 rounds of communication and tolerate cross-DC network partitions.

Other CC systems include SwiftCloud [68], Bolt-On [11], Saturn [19], Bayou [54, 63], PRACTI [15], ISIS [18], lazy replication [36], causal memory [2], EunomiaKV [30] and CausalSpartan [57]. These systems either do not support ROTs, or target a different model from the one considered in this paper, e.g., they do not implement sharding the data set in partitions. Our theoretical results require at least two partitions. Investigating the cost of LO in other system models is an avenue for future work.

CC is also implemented by systems that support different consistency levels [24], implement strong consistency on top of CC [12], and combine different consistency levels depending on the semantics of operations [13, 39] or on target SLAs [5, 62]. Our theorem provides a lower bound on the overhead of LO ROTs with CC. Hence, any system that implements CC or a strictly stronger consistency level cannot avoid such overhead. We are investigating how the lower bound on this overhead varies depending on the consistency level, and what is its effect on performance.

Theoretical results on CC. Lamport introduces the concept of causality [37], and Hutto and Ahamad [32] provide the first definition of CC, later revisited from different angles [1, 23, 48, 66]. Mahajan et al. prove that real-time CC is the strongest consistency level that can be obtained in an always-available and one-way convergent system [44]. Attiya et al. introduce the observable CC model and show that it is the strongest that can be achieved by an eventually consistent data store implementing multi-value registers [7].

The SNOW theorem [42] shows that LO can be achieved by any system that *i*) is not strictly serializable [53] or *ii*) does not support write transactions. Based on this result, the SNOW paper suggests that any protocol that matches one of these two conditions can be *improved* to be LO. In this paper, we *prove* that achieving LO in CC implies an extra cost on writes, which is inherent and significant.

Bailis et al. study the overhead of replication and dependency tracking in geo-replicated CC systems [10]. By contrast, we investigate the inherent cost of LO CC designs, i.e., even in absence of (geo-)replication.

8. CONCLUSION

Causally consistent read-only transactions (ROT) are an attractive primitive for large-scale systems, as they eliminate a number of anomalies and ease the task of developers. Because many applications are read-dominated, low latency of ROTs is key to overall system performance. It would therefore appear that *latency-optimal* (LO) ROTs, which provide a nonblocking, single-version and single-round implementation, are particularly appealing.

In this paper we show that, surprisingly, LO induces a resource utilization overhead that can actually jeopardize performance. We show this results from two angles. First, we present an almost LO protocol that, by avoiding the aforesaid overhead, achieves better performance than the state-of-the-art LO design. Then, we prove that the overhead posed by LO is inherent to causal consistency, i.e., cannot be avoided by any implementation. We provide a lower bound on such overhead, showing that it grows linearly with the number of clients.

Acknowledgements

We thank the anonymous reviewers, Sandhya Dwarkadas, Panagiota Fatourou and Alan Fekete for their valuable suggestions. We also thank Haonan Lu for his help in understanding the design and implementation of COPS-SNOW. This research has been supported by the European ERC Grant 339539 - AOC, an EcoCloud post-doctoral research fellowship, and Amazon through AWS Cloud Credits.

9. REFERENCES

[1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Cambridge, MA, USA, 1999. AAI0800775.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[3] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguica, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, volume 00, pages 405–414, June 2016.

[4] S. Almeida, J. a. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[5] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 367–381, Broomfield, CO, 2014. USENIX Association.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[7] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 385–394, New York, NY, USA, 2015. ACM.

[8] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems (2Nd Ed.)*, pages 55–96. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[9] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford. Spanner: Becoming a sql system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 331–343, New York, NY, USA, 2017. ACM.

[10] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 22:1–22:7, New York, NY, USA, 2012. ACM.

[11] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[12] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, New York, NY, USA, 2015. ACM.

[13] V. Balegas, C. Li, M. Najafzadeh, D. Porto, A. Clement, S. Duarte, C. Ferreira, J. Gehrke, J. Leitão, N. Preguiça, R. Rodrigues, M. Shapiro, and V. Vafeiadis. Geo-replication: Fast if possible, consistent if necessary. *Data Engineering Bulletin*, 39(1):81–92, Mar. 2016.

[14] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, Santa Clara, CA, 2017. USENIX Association.

[15] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association.

[16] P. A. Bernstein, S. Burckhardt, S. Bykov, N. Crooks, J. M. Faleiro, G. Kliot, A. Kumbhare, M. R. Rahman, V. Shah, A. Szekeres, and J. Thelin. Geo-distribution of actor-based services. *Proc. ACM Program. Lang.*, 1(OOPSLA):107:1–107:26, Oct. 2017.

[17] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[18] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, Jan. 1987.

[19] M. Bravo, L. Rodrigues, and P. Van Roy. Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 111–126, New York, NY, USA, 2017. ACM.

[20] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[22] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner:

Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013.

[23] N. Crooks, Y. Pu, L. Alvisi, and A. Clement. Seeing is believing: A client-centric specification of database isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 73–82, New York, NY, USA, 2017. ACM.

[24] N. Crooks, Y. Pu, N. Estrada, T. Gupta, L. Alvisi, and A. Clement. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1615–1628, New York, NY, USA, 2016. ACM.

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[26] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[27] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, New York, NY, USA, 2014. ACM.

[28] Google. Protocol buffers. https://developers.google.com/protocol-buffers/, 2017.

[29] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, New York, NY, USA, 2016. ACM.

[30] C. Gunawardhana, M. Bravo, and L. Rodrigues. Unobtrusive deferred update stabilization for efficient geo-replication. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 83–95, Santa Clara, CA, 2017. USENIX Association.

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

[32] P. W. Hutto and M. Ahamad. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings.,10th International Conference on Distributed Computing Systems*, pages 302–309, May 1990.

[33] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[34] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[35] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *Principles of Distributed Systems (OPODIS)*, pages 17–32. Springer International Publishing, 2014.

[36] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

[37] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[38] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[39] C. Li, J. a. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

[40] 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[41] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[42] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The snow theorem and latency-optimal read-only transactions. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 135–150, Berkeley, CA, USA, 2016. USENIX Association.

[43] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM.

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

[45] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[46] 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 *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 453–468, 2017.

[47] H. Moniz, J. a. Leitão, R. J. Dias, J. Gehrke, N. Preguiça, and R. Rodrigues. Blotter: Low latency transactions for geo-replicated storage. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 263–272, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[48] D. Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, Jan. 1993.

[49] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi. Minimizing commit latency of transactions in geo-replicated data stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1279–1294, New York, NY, USA, 2015. ACM.

[50] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[51] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, and R. H. Campbell. Ambry: Linkedin's scalable geo-distributed object store. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 253–265, New York, NY, USA, 2016. ACM.

[52] NTP. The network time protocol. http://www.ntp.org, 2017.

[53] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979.

[54] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 288–301, New York, NY, USA, 1997. ACM.

[55] M. R. Rahman, L. Tseng, S. Nguyen, I. Gupta, and N. Vaidya. Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores. *ACM Trans. Auton. Adapt. Syst.*, 11(4):20:1–20:36, Jan. 2017.

[56] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 95–110, New York, NY, USA, 2017. ACM.

[57] M. Roohitavaf, M. Demirbas, and S. Kulkarni. Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 184–193, Sept. 2017.

[58] D. Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 133–142, New York, NY, USA, 1981. ACM.

[59] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[60] K. Spirovska, D. Didona, and W. Zwaenepoel. Optimistic causal consistency for geo-replicated key-value stores. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2626–2629, June 2017.

[61] K. Spirovska, D. Didona, and W. Zwaenepoel. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In *48th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

[62] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[63] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[64] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.

[65] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1041–1052, New York, NY, USA, 2017. ACM.

[66] P. Viotti and M. Vukolic. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, 2016.

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

[68] 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 *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, New York, NY, USA, 2015. ACM.

[69] I. Zhang, N. Lebeck, P. Fonseca, B. Holt, R. Cheng, A. Norberg, A. Krishnamurthy, and H. M. Levy. Diamond: Automating data management and storage for wide-area, reactive applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 723–738, GA, 2016. USENIX Association.

[70] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 263–278, New York, NY, USA, 2015. ACM.

[71] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving

serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.

Table 3: Definition of symbols.

Symbol	Definition
N	Number of partitions
M	Number of replicas per partition
m	Index used to refer to the local DC
d	A version $\langle k, v, sr, DV\rangle$
c	A client (in the local DC)
DV_c	Dependency vector at client c
p_n^m	The $m-$th replica of the $n-$th partition
GSS_n^m	Global stable snapshot of p_n^m
$Clock_n^m$	Physical clock time of p_n^m
HLC_n^m	Hybrid clock time of p_n^m
VV_n^m	Hybrid version vector of p_n^m
SV	Snapshot vector of a ROT

APPENDIX

A. CONTRARIAN. PROTOCOLS

Table 3 introduces the notation we use in our discussion. Algorithm 1 and Algorithm 2 report the pseudo-code of the protocols implemented, respectively, by clients and servers. We focus on the protocols for PUT and ROT operations.

A.1 Meta-data

Version. A version d is a tuple $\langle k, v, sr, DV\rangle$. k, v is the key-value pair specified by the client when creating the version by means of a PUT. sr is the *source replica* of d, i.e., the id of the DC in which d has been created. DV is a dependency vector with M entries. $DV[m]$ is the *update timestamp* of d, which is assigned to d upon creation by means of a PUT operation. $DV[j] = t$ means that d potentially causally depends on versions created in the j-th DC with an update timestamp up to t.

Client. Each client c maintains a dependency vector DV_c, with M entries. DV_c summarizes the causal dependencies established by c similarly to what is done by the dependency vector of versions. c piggybacks DV_c on each PUT request to encode the dependencies of the newly created version, and on each ROT request to ensure the monotonicity of the snapshot observed by c (i.e., to ensure that the snapshot from which the ROT reads includes all versions on which c potentially depends).

Server. Each server p_n^m has access to a monotonically increasing (hardware) physical clock, $Clock_n^m$, and maintains a (software) hybrid clock HLC_n^m.

p_n^m also maintains two vector clocks of HLCs with one entry per DC: the version vector VV_n^m and the global stable snapshot GSS_n^m. $VV_n^m[m]$ is the version clock of p_n^m and corresponds to the update timestamp of the latest PUT completed on p_n^m, or the timestamp included in the latest heartbeat sent by p_n^m. $VV_n^m[i], i \neq m$, indicates the timestamp of the latest version received by p_n^m that has been received from the replica in the $i-$th DC.

$GSS_n^m[i] = t, i \neq m$, means that p_n^m is aware that all nodes in the $m-$th DC have received all versions created in the $i-$th DC with update timestamp up to t. Hence, p_n^m knows that it can read these versions without blocking. The local entry of GSS_n^m is not used because versions created in the m-th DC are trivially already present in the m-th DC. We keep such local entry in our description for the sake of simplicity, because it allows us to refer to the i-th entry of the GSS when evaluating the visibility predicate on versions created in the i-th DC.

A.2 Protocols

PUT. c sends a PUT-Req $\langle k, v, DV_c\rangle$ message to the partition p_n^m that is responsible for key k. The request specifies the key-value pair k, v to be written and the dependency vector of c. Upon receiving c's request, p_n^m first updates GSS_n^m to be entry-wise at least as high as the latest one seen by c, represented by DV_c (Algorithm 2 Line 2). This ensures that p_n^m exposes to later operations a snapshot of the data store that includes all dependencies of the newly created version.

Then, p_n^m determines the HLC update timestamp ut to assign to the new version of k to be created, which we denote d. ut is computed as the maximum of the current physical clock value on p_n^m, the highest timestamp seen by the client plus one, and the current value of the HLC on p_n^m plus one (Algorithm 2 Line 3). By this assignment, ut reflects causality, i.e., $X \rightsquigarrow Y$ implies that the update timestamp of X is lower than update the timestamp of Y. p_n^m sets the dependency meta-data of d as follows (Algorithm 2 Lines 4–7). The remote entries of $d.DV$ are set to the corresponding values in DV_c. The local entry in $d.DV$ is ut. Then, p_n^m inserts d in the version chain of key k and updates its version vector (Algorithm 2 Lines 8–9). Then, p_n^m replies to c with its current HLC value. This is used by c to update $DV_c[m]$ (Algorithm 1 Line 4). Finally, p_n^m replicates d by sending to its replicas a copy of d. Upon receiving such replication message, a replica p_n^i inserts a copy of d in the version chain corresponding to $d.k$ and sets $VV_n^i[m]$ to ut (Algorithm 2 Lines 12–14).

ROT. c initiates a ROT operation by identifying the set of partitions that store at least one key to be read (Algorithm 1 Lines 7–8). One of these partitions is chosen as coordinator, which we note p_{coord} (Algorithm 1 Line 9).

c sends to p_{coord} a Snap-Req $\langle DV_c\rangle$ message, which provides the dependency vector of c. Upon receiving the request from c, p_{coord} updates HLC_n^m and GSS_n^m using, respectively, the local and remote entries in DV_c (Algorithm 2 Lines 16–17). Then, p_{coord} builds the snapshot vector SV

Algorithm 1 Client c at data center m.

```
1:  function PUT(key $k$, value $v$)
2:      send ⟨PUT-Req $k, v, DV_c$⟩ to $p_k$ server
3:      receive ⟨PUTReply $ut$⟩ from $p_k$
4:      $DV_c[m] \leftarrow ut$                    ▷ Update client's local dependency
5:  end function

6:  function RO-TX(key-set $\chi$)
7:      $\chi_i \leftarrow \{k \in \chi : p_i^m\ stores\ key\ k\}$
8:      $\Pi \leftarrow p_i^m : \chi_i \neq \emptyset$                    ▷ Involved partitions
9:      $p_{coord} \leftarrow$ random in $\Pi$                    ▷ Coordinator partition
10:     send ⟨Snap-Req $DV_c$⟩ to $p_{coord}$
11:     receive ⟨Snap-Resp $SV$⟩ from $p_{coord}$
12:     $DV_c \leftarrow max\{DV_c, SV\}$          ▷ Update snapshot known to c
13:     send ⟨ROT-Req $SV, \chi_i$⟩ to $p_i^m\ \forall p_i^m \in \Pi$
14:     for $p_i^m \in \Pi$ do
15:         receive ⟨ROT-Resp $D_i, HLC_i^m$⟩ from $p_i^m$
16:         $DV_c[m] \leftarrow max\{DV_c[m], HLC_i^m\}$
17:     end for
18:     return $\bigcup D_i,\ \forall p_i^m \in \Pi$                ▷ Return received versions
19: end function
```

Algorithm 2 Server p_n^m serving clients requests.

```
1: upon receive ⟨PUT-Req $k, v, DV_c$⟩ from $c$ do
       ▷ Update HLC and generate timestamps that reflects causality
2:     $HLC_n^m \leftarrow max\{Clock_n^m, max\{DV_c\}+1, HLC_n^m+1\}$
       ▷ Update GSS to include c's dependencies.
3:     $GSS_n^m \leftarrow max\{GSS_n^m, DV_c\}$
4:     $ut \leftarrow HLC_n^m$
5:     create new item $d$
6:     $DV \leftarrow DV_c$; $DV[m] \leftarrow ut$
7:     $d \leftarrow < k; v; m; DV >$
8:     insert $d$ in the version chain of key $k$
9:     $VV_n^m[m] \leftarrow ut$
10:    send ⟨PUTReply $HLC_n^m$⟩ to client
11:    send ⟨Replicate d⟩ to $p_n^i, i = 0 \ldots M, i \neq m$

12: upon receive ⟨Replicate $d$⟩ from $p_n^i$ do
13:    insert $d$ in version chain of key $d.k$
14:    $VV_n^m[i] \leftarrow d.DV[i]$

15: upon receive ⟨Snap-Req $DV_c$⟩ from $c$ do
16:    $HLC_n^m \leftarrow max\{HLC_n^m, DV_c[m]\}$                ▷ Update HLC
       ▷ Update GSS to include c's dependencies.
17:    $GSS_n^m \leftarrow max\{GSS_n^m, DV_c\}$
18:    $SV \leftarrow GSS_n^m$; $SV[m] \leftarrow HLC_n^m$
19:    send ⟨Snap-Resp $SV$⟩ to $c$

20: upon receive ⟨ROT-req $SV, \chi$⟩ from $c$ do
21:    $HLC_n^m \leftarrow max\{HLC_n^m, SV[m]\}$                  ▷ Update HLC
       ▷ Install snapshot corresponding to SV
22:    $GSS_n^m \leftarrow max\{GSS_n^m, SV\}$
23:    $D \leftarrow \emptyset$                          ▷ Set of versions to return.
24:    for $k \in \chi$ do                  ▷ Find freshest visible versions of k
25:        $D_k \leftarrow d : d.k == k \wedge d.DV \leq SV$
26:        $d \leftarrow$ version with highest update timestamp $\in D_k$
27:    end for
28:    send ⟨ROT-Resp$D, HLC_n^m$⟩ to $c$

29: upon $\nexists$ ongoing PUT $\wedge$ time with no PUT $\geq \Delta$ do
30:    $HLC_n^m \leftarrow max\{Clock_n^m, HLC_n^m\}$
31:    $VV_n^m[m] \leftarrow HLC_n^m$
32:    send ⟨HEARTBEAT $VV_n^m[m]$⟩ to $p_n^i, \forall i = 0 \ldots M, i \neq m$

33: upon receiving ⟨HEARTBEAT ht⟩ from $p_n^j$ do
34:    $VV_n^m$[j]←ht

   ▷ Stabilization
35: upon every $\Delta_G$ time do
   ▷ Update GSS
36:    $GSS_n^m[j] \leftarrow min\{VV_i^m[j]\}, \forall j = 0 \ldots M-1, \forall i = 0 \ldots N-1$
       ▷ Advance local clock
37:    $t_m \leftarrow max\{VV_i^m[m]\}, \forall i = 0 \ldots N-1$
38:    $HLC_n^m \leftarrow max\{HLC_n^m, t_m\}$
```

for the ROT. The remote entries of the vector correspond to GSS_n^m. The local entry corresponds to HLC_n^m (Algorithm 2 Line 18). Updating GSS_n^m and HLC_n^m, and building SV starting from them ensures that the snapshot visible to the ROT is at least as fresh as the one accessed by c so far. p_{coord} replies to c with SV.

Upon receiving the snapshot vector SV for the ROT, c first updates its DV_c to be at least as high as SV entry-wise (Algorithm 1 Line 12). Then, c sends a message ROT-Req $\langle \chi_i, SV \rangle$ to each partition p_i^m that stores at least one key to be read. Upon receiving such a message, a partition first updates its own HLC and GSS (Algorithm 2 Lines 21–22). Then, for each key in χ_i, the partition returns to c the version with the highest update timestamp that belongs to the snapshot defined by SV (Algorithm 2 Lines 23–27). The partition also returns to c the current value of its hybrid clock. The client updates the local entry of its dependency vector with such hybrid clock value (Algorithm 1 Line 16), collects all returned versions, and returns them to the application (Algorithm 1 Line 15).

Stabilization. Periodically, nodes within a DC exchange their version vectors. GSS_n^m is computed as aggregate minimum of all these vectors (Algorithm 2 Line 36). Hence, $GSS_n^m[i] = t, i \neq m$ indicates that p_n^m is aware that all partitions within the m-th DC have received all updates with update timestamp up to time t originated from the i-th DC. p_n^m also advances HLC_n^m to match the highest local entry in the exchanged version vectors (Algorithm 2 Lines 37–38). This ensures that recently created local versions are included in future ROT snapshots.

To enhance the scalability of the stabilization protocol, in our prototype partitions within a DC are arranged in a tree [3, 27]. The aggregation of the GSS is performed from the leaves up to the root of the tree. The GSS is then propagated down the tree from the root partition.

If a partition does not receive PUT requests, its version clock does not advance and the GSS computed in other DCs cannot progress. To ensure the progress of the GSS, if p_n^m does not receive a PUT request for a given amount of time, it sends a heartbeat message with its updated version clock (Algorithm 2 Lines 29–32). Heartbeat messages and update replication messages are sent (and received) in order of increasing timestamps. Upon receiving a heartbeat message from p_n^i with timestamp ht, p_n^m sets the i-th entry of its version vector to ht (Algorithm 2 Lines 33-34).

B. CORRECTNESS

We provide an informal proof sketch showing that Contrarian achieves causal consistency.

Lemma 3. *If c has established a dependency towards X, then $DV_c \geq X.DV$.*

Proof. We consider three cases, corresponding to the three scenarios that lead to establishing a causal dependency (see Section 2.4). *i)* If c has written X on p_x, the lemma follows from the fact that $X.DV$ is built starting from DV_c, and that after X is written, c sets the local entry of DV_c to be equal to the value of the hybrid clock on p_x, which in turn is higher than or equal to the timestamp of X. *ii)* If c has read X, then c has issued a ROT with a snapshot vector $SV \geq X.DV$. Then, the lemma follows because upon returning from the ROT, $DV_c \geq SV$. *iii)* If c has read $Z : X \leadsto Z$, then the lemma follows by induction, assuming that $X.DV \geq Z.DV$ and using the previous two cases as base steps in the inductive process. □

Lemma 4. *If $X \leadsto Y$, then $X.DV < Y.DV$.*

Proof. Assume client c writes Y. When performing the PUT(y,Y) operation on p_y, c provides its dependency vector DV_c. By Lemma 3, if c has established a dependency on X, $DV_c \geq X.DV$. Then, the lemma holds because p_x enforces that the remote entries of $Y.DV$ are at least as high as the corresponding entries in DV_c, and that the local entry of $Y.DV$ is strictly higher than any entry in DV_c. □

Lemma 5. *Assume $X \leadsto Y$ and that c has established a dependency towards Y. Then, any later ROT by c that targets x returns a version of x, X', such that $X' \not\leadsto X$.*

Proof. If c depends on Y, then $DV_c \geq Y.DV$ by Lemma 3. If $X' \leadsto X \leadsto Y$, then $X'.DV \leq X.DV \leq Y.DV$ by Lemma 4, and the timestamp of X' is strictly lower than the timestamp of X by construction (Algorithm 2 Lines 1–7). Then, any ROTs issued by c has a snapshot vector $SV \geq DV_c$, which includes X. Hence, the lemma follows because a partition returns the version of a key with the highest timestamp within a snapshot (Algorithm 2 Lines 24–27). □

Lemma 6. *Assume $X \rightsquigarrow X' \rightsquigarrow Y$ and c issues a ROT(x,y) operation. If c reads Y within said ROT, then c reads X'.*

Proof. Let the ROT be assigned a snapshot vector SV, with local entry ts. If Y is read within the ROT, then $Y.DV \leq SV$. We distinguish between two cases: X' is remote and X' is local.

X' is remote. The remote entries of SV are built starting from the corresponding entries of the GSS, which only includes remote versions whose causal dependencies have already been received in the DC. Because $X' \rightsquigarrow Y$, $X'.DV < Y.DV \leq SV$. Hence, if Y is read by the ROT, X' has already been received by the corresponding partition p_x. Moreover, because $X \rightsquigarrow X'$, the timestamp of X is lower than the timestamp of X', and the ROT returns the version of x within SV with the highest timestamp.

X' is local. There are three sub-cases to consider. In the first case, X' has been already created on p_x by the time the ROT arrives on p_x. Then, because $X' \rightsquigarrow Y$, $X'.DV < Y.DV \leq SV$, so X' is visible to the ROT. Hence, p_x returns X' to the ROT because $X \rightsquigarrow X'$ and hence the timestamp of X is lower than the one of X'.

In the second case, the PUT(x, X') operation that creates X' has not arrived yet on p_x by the time the ROT arrives on p_x. Then, as soon as the ROT message arrives on p_x, p_x moves its HLC to be at least as high as ts. Hence, when the PUT operation creates X', X' is assigned a timestamp that is strictly higher than ts. Therefore, neither X' nor any $Y : X' \rightsquigarrow Y$ are visible within the snapshot defined by SV, whose local entry is ts.

In the third case, the PUT(x,X') operation and the ROT request are concurrent, and X' is assigned a timestamp lower than ts. Because PUT(x,X') has not completed by the time the ROT reads x on p_x, p_x might not return X' to the ROT, even if $X'.DV \leq SV$. However, this cannot lead to breaking CC because it is impossible for $Y : X' \rightsquigarrow Y$ to have a timestamp $\leq ts$. In fact, before serving the read operation within the ROT, p_x moves its HLC to be at least as high as ts. This means that any later reply messages sent by p_x to any clients carry a timestamp at least as high as ts (Algorithm 2 Line 10, 19 and 28). Hence, the local entry of the dependency value of any clients that establish a dependency towards X' is at least as high as ts (Algorithm 1 Line 4, 12 and 16). As a consequence, any $Y' : X' \rightsquigarrow Y'$ has a timestamp higher than ts and, hence, is not visible to the ROT according to SV.

□