Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store

Kristina Spirovska
EPFL
kristina.spirovska@epfl.ch

Diego Didona
EPFL
diego.didona@epfl.ch

Willy Zwaenepoel
EPFL
willy.zwaenepoel@epfl.ch

***Abstract*—Transactional Causal Consistency (TCC) extends causal consistency, the strongest consistency model compatible with availability, with interactive read-write transactions, and is therefore particularly appealing for geo-replicated platforms.**

This paper presents Wren, the first TCC system that at the same time i) implements nonblocking read operations, thereby achieving low latency, and ii) allows an application to efficiently scale out within a replication site by sharding.

Wren introduces new protocols for transaction execution, dependency tracking and stabilization. The transaction protocol supports nonblocking reads by providing a transaction with a snapshot that is the union of a fresh causal snapshot S installed by every partition in the local data center and a client-side cache for writes that are not yet included in S. The dependency tracking and stabilization protocols require only two scalar timestamps, resulting in efficient resource utilization and providing scalability in terms of replication sites. In return for these benefits, Wren slightly increases the visibility latency of updates.

We evaluate Wren on an AWS deployment using up to 5 replication sites and 16 partitions per site. We show that Wren delivers up to 1.4x higher throughput and up to 3.6x lower latency when compared to the state-of-the-art design. The choice of an older snapshot increases local update visibility latency by a few milliseconds. The use of only two timestamps to track causality increases remote update visibility latency by less than 15%.

I. Introduction

Many large-scale data platforms rely on geo-replication to meet strict performance and availability requirements [1], [2], [3], [4], [5]. Geo-replication reduces latencies by keeping a copy of the data close to the clients, and enables availability by replicating data at geographically distributed data centers (DCs). To accommodate the ever-growing volumes of data, today's large-scale on-line services also partition the data across multiple servers within a single DC [6], [7].

Transactional Causal Consistency (TCC). TCC [8] is an attractive consistency level for building geo-replicated data-stores. TCC enforces causal consistency (CC) [9], which is the strongest consistency model compatible with availability [10], [11]. Compared to strong consistency [12], CC does not suffer from high synchronization latencies, limited scalability and unavailability in the presence of network partitions between DCs [13], [14], [15]. Compared to eventual consistency [2], CC avoids a number of anomalies that plague programming with weaker models. In addtion, TCC extends CC with interactive read-write transactions, that allow applications to read from a causal snapshot and to perform atomic multi-item writes.

Enforcing CC while offering always-available interactive multi-partition transactions is a challenging problem [7]. The main culprit is that in a distributed environment, unavoidably, partitions do not progress at the same pace. Current TCC designs either avoid this issue altogether, by not supporting sharding [16], or block reads to ensure that the proper snapshot is installed [8]. The former approach sacrifices scalability, while the latter incurs additional latencies.

Wren. This paper presents Wren, the first TCC system that implements nonblocking reads, thereby achieving low latency, and allows an application to scale out by sharding. Wren implements CANToR (Client-Assisted Nonblocking Transactional Reads), a novel transaction protocol in which the snapshot of the data store visible to a transaction is defined as the union of two components: i) a fresh causal snapshot that has been installed by *every* partition within the DC; and ii) a per-client cache, which stores the updates performed by the client that are not yet reflected in said snapshot. This choice of snapshot departs from earlier approaches where a snapshot is chosen by simply looking at the local clock value of the partition acting as transaction coordinator.

Wren also introduces Binary Dependency Time (BDT), a new dependency tracking protocol, and Binary Stable Time (BiST), a new stabilization protocol. Regardless of the number of partitions and DCs, these two protocols assign only two scalar timestamps to updates and snapshots, corresponding to dependencies on local and remote items. These protocols provide high resource efficiency and scalability, and preserve availability.

Wren exposes to clients a snapshot that is slightly in the past with respect to the one exposed by existing approaches. We argue that this is a small price to pay for the performance improvements that Wren offers.

We compare Wren with Cure [8], the state-of-the-art TCC system, on an AWS deployment with up to 5 DCs with 16 partitions each. Wren achieves up to 1.4x higher throughput and up to 3.6x lower latencies. The choice of an older snapshot increases local update visibility latency by a few milliseconds. The use of only two timestamps to track causality increases remote update visibility latency by less than 15%.

We make the following contributions.

1) We present the design and implementation of Wren, the first TCC key-value store that achieves nonblocking reads, efficiently scales horizontally, and tolerates network partitions

between DCs.

2) We propose new dependency and stabilization protocols that achieve high resource efficiency and scalability.

3) We experimentally demonstrate the benefits of Wren over state-of-the-art solutions.

Roadmap. The paper is organized as follows. Section 2 describes TCC and the target system model. Section 3 presents the design of Wren. Section 4 describes the protocols in Wren. Section 5 presents the evaluation of Wren. Section 6 discusses related work. Section 7 concludes the paper.

II. System model and definitions

A. System model

We consider a distributed key-value store whose data-set is split into N partitions. Each key is deterministically assigned to one partition by a hash function. We denote by p_x the partition that contains key x.

The data-set is fully replicated: each partition is replicated at all M DCs. We assume a multi-master system, i.e., each replica can update the keys in its partition. Updates are replicated asynchronously to remote DCs.

The data store is multi-versioned. An update operation creates a new version of a key. Each version stores the value corresponding to the key and some meta-data to track causality. The system periodically garbage-collects old versions of keys.

At the beginning of a session, a client c connects to a DC, referred to as the local DC. All c's operations are performed within said DC to preserve availability [17] [1]. c does not issue another operation until it receives the reply to the current one. Partitions communicate through point-to-point lossless FIFO channels (e.g., a TCP socket).

B. Causal consistency

Causal consistency requires that the key-value store returns values that are consistent with *causality* [9], [18]. For two operations a, b, we say that b causally depends on a, and write $a \rightsquigarrow b$, if and only if at least one of the following conditions holds: $i)$ a and b are operations in a single thread of execution, and a happens before b; $ii)$ a is a write operation, b is a read operation, and b reads the version written by a; $iii)$ there is some other operation c such that $a \rightsquigarrow c$ and $c \rightsquigarrow b$. Intuitively, CC ensures that if a client has seen the effects of b and $a \rightsquigarrow b$, then the client also sees the effects of a.

We use lower-case letters, e.g., x, to refer to a key and the corresponding upper-case letter, e.g., X, to refer to a version of the key. We say that X causally depends on Y if the write of X causally depends on the write of Y.

We use the term availability to indicate that a client operation never blocks as the result of a network partition between DCs [19].

[1]Wren can be extended to allow a client c to move to a different DC by blocking c until the last snapshot seen by c has been installed in the new DC.

C. Transactional causal consistency

Semantics. TCC extends CC by means of interactive read-write transactions in which clients can issue several operations within a transaction, each reading or writing (potentially) multiple items [8]. TCC provides a more powerful semantics than one-shot read-only or write-only transactions provided by earlier CC systems [7], [15], [20], [21]. It enforces the following two properties.

1. Transactions read from a causal snapshot. A causal snapshot is a set of item versions such that all causal dependencies of those versions are also included in the snapshot. For any two items, x and y, if $X \rightsquigarrow Y$ and both X and Y belong to the same causal snapshot, then there is no X', such that $X \rightsquigarrow X' \rightsquigarrow Y$.

Transactional reads from a causal snapshot avoid undesirable anomalies that can arise by issuing multiple individual read operations. For example, they prevent the well-known anomaly in which person A removes person B from the access list of a photo album and adds a photo to it, only to have person B read the original permissions and the new version of the album [15].

2. Updates are atomic. Either all items written by a transaction are visible to other transactions, or none is. If a transaction writes X and Y, then any snapshot visible to other transactions either includes both X and Y or neither one of them.

Atomic updates increase the expressive power of applications, e.g., they make it easier to maintain symmetric relationships among entities within an application. For example, in a social network, if person A becomes friend with person B, then B simultaneously becomes friend with A. By putting both updates inside a transaction, both or neither of the friendship relations are visible to other transactions [21].

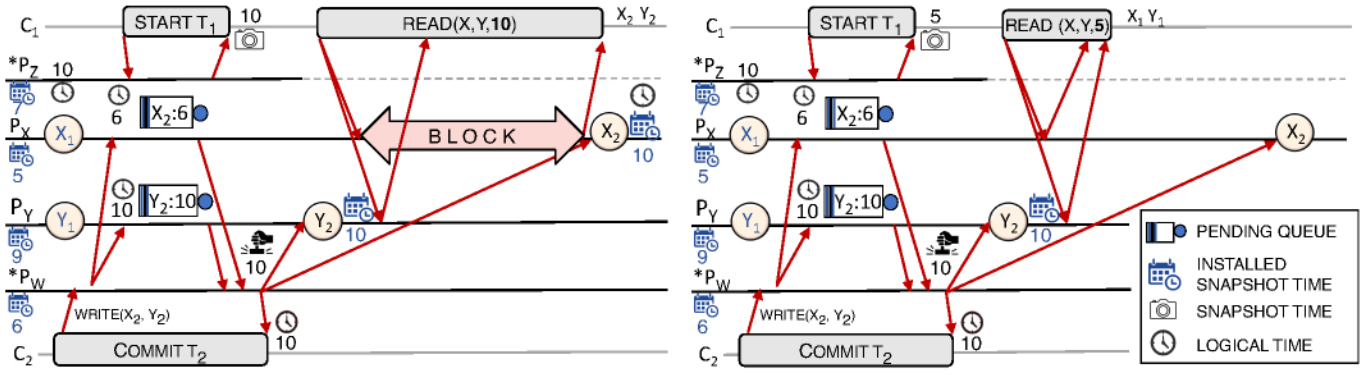
Conflict resolution. Two writes are conflicting if they are not related by causality and update the same key. Conflicting writes are resolved by means of a commutative and associative function, that decides the value corresponding to a key given its current value and the set of updates on the key [15].

For simplicity, Wren resolves write conflicts using the last-writer-wins rule based on the timestamp of the updates [22]. Possible ties are settled by looking at the id of the update's originating DC combined with the identifier of transaction that created the update. Wren can be extended to support other conflict resolution mechanisms [8], [15], [21], [23].

D. APIs

A client starts a transaction T, issues read and write (multi-key) operations and commits T. Wren's client API exposes the following operations:

- $< T_{ID}, S > \leftarrow START()$ **:** starts an interactive transaction T and returns T's transaction identifier T_{ID} and the causal snapshot S visible to T.
- $\langle vals \rangle \leftarrow READ(k_1, ..., k_n)$ **:** reads the set of items corresponding to the input set of keys within T.
- $WRITE(\langle k_1, v_1 \rangle, ..., \langle k_n, v_n \rangle)$ **:** updates a set of given input keys to the corresponding values within T.



(a) Blocking reads in existing systems.

(b) Nonblocking reads in Wren.

Fig. 1: In existing systems (a), a transaction can be assigned a snapshot that has not been installed by every partition in the local DC. c_1's transaction is assigned timestamp 10, but p_x has not installed snapshot 10 by the time c_1 reads. This leads p_x to block c_1's read. In Wren (b), c_1's transaction is assigned a timestamp corresponding to a snapshot installed by every partition in the local DC, thus avoiding blocking. The trade-off is that older versions of x and y are returned.

•***COMMIT*()** : finalizes the transaction T and atomically updates the items modified by means of a WRITE operation within T, if any.

In TCC, conflicting updates do not cause transactions to abort, because they are resolved by the conflict resolution mechanism. Transactions can abort by means of explicit APIs, or because of system-related issues, e.g., not enough space on a server to perform an update. For simplicity, we do not consider aborts in this paper.

III. The design of Wren

We first illustrate the challenge in providing nonblocking reads, by showing how reads can block in the state-of-the-art Cure system [8]. We then present CANToR (§ III-B), and BDT and BiST (§ III-C). We discuss fault tolerance and availability in Wren (§ III-D).

A. The challenge in providing nonblocking reads

For the sake of simplicity, we assume that a transaction snapshot S is defined by a logical timestamp, denoted st. We say that a server has *installed* a snapshot with timestamp t if the server has applied the modifications of all committed transactions with timestamp up to and including t. Once a server installs a snapshot with timestamp t, the server cannot commit any transaction with a timestamp $\leq t$.

Achieving nonblocking reads in TCC is challenging, because they have to preserve consistency and respect the atomicity of multi-item (and hence multi-partition) write transactions. Assume that a transaction writes X and Y. A transaction T that reads x and y, must either see both X and Y or neither of them. The complexity of the problem is increased by the fact that the reads on individual keys in a transactional READ may proceed in parallel. In other words, a $READ(T_{ID}, x, y)$ sends in parallel a $read(x)$ operation to p_x and a $read(y)$ operation to p_y, and the $read(x)$ taking place on p_x is unaware of the item returned by the $read(y)$ on p_y.

Cure [8] provides the state-of-the-art solution to this problem. When a client c starts a transaction T, T is assigned a causal snapshot S by a randomly chosen *coordinator* partition. S includes all previous snapshots seen by c. To this end, the coordinator sets st as the maximum between the highest snapshot timestamp seen by c and the current clock value at the coordinator. When T commits, it is assigned a commit timestamp by means of a two-phase commit (2PC) protocol. Every partition that stores an item modified by T proposes a timestamp (strictly higher than st), and the coordinator picks the maximum as the commit timestamp ct of T. All items written by T are assigned ct as timestamp. Because $ct > st$, all such writes carry the information that they depend on the items in S, whose timestamps are less than or equal to st.

Cure achieves causality and enforces atomicity. If a transaction is assigned a snapshot timestamp st, the individual read operations of a READ transaction can in parallel read the version of any requested key with the highest timestamp $\leq st$. This protocol, however, enforces causality and atomicity at the cost of potentially blocking read operations. We show this behavior by means of an example, depicted in Figure 1a.

To initiate T_1, client c_1 contacts a coordinator partition, in this case p_z. T_1 is the first transaction issued by c_1, so c_1 does not piggyback any snapshot timestamp to initiate a transaction. The local time on p_z is 10. To maximize the freshness of the snapshot visible to T_1, p_z assigns to T_1 a timestamp equal to 10. In the meantime, c_2 commits T_2, which writes X_2 and Y_2. During the 2PC, p_x proposes 6 as commit timestamp, i.e., the current clock's value on p_x. Similarly, p_y proposes 10. The coordinator of T_2, p_w, picks the maximum between these two values and assigns to T_2 a commit timestamp 10. p_y receives the commit message, writes Y_2 and installs a snapshot with timestamp 10. p_x, instead, does not immediately receive the commit message, and its snapshot still has the value 5.

At this point, c_1 issues its READ(T_1, x, y) operation by sending a request to p_x and p_y with the snapshot timestamp of T_1, which is 10. p_y has installed a snapshot that is fresh

enough, and returns Y_2. Instead, p_x has to block the read of T_1, because p_x cannot determine which version of x to return. p_x cannot safely return X_1, because it could violate CC and atomicity. p_x cannot return X_2 either, because p_x does not yet know the commit timestamp of X_2. If X_2 were eventually to be assigned a commit timestamp > 10, then returning X_2 to T_1 violates CC. p_x can install X_2 and the corresponding snapshot only when receiving the commit message from p_w. Then, p_x can serve c_1's pending read with the consistent value X_2.

Similar dynamics characterize also other CC systems with write transactions, e.g., Eiger [21].

B. Nonblocking reads in Wren

Wren implements CANToR, a novel transaction protocol that, similarly to Cure, is based on snapshots and 2PC, but avoids blocking reads by changing how snapshots visible to transactions are defined. In particular, a transaction snapshot is expressed as the union of two components:

1) a fresh causal snapshot installed by every partition in the local DC, which we call *local stable snapshot*, and
2) a client-side cache for writes done by the client and that have not yet been included in the local stable snapshot.

1) Causal snapshot. Existing approaches block reads, because the snapshot assigned to a transaction T may be "in the future" with respect to the snapshot installed by a server from which T reads an item. CANToR avoids blocking by providing to a transaction a snapshot that only includes writes of transactions that have been installed at all partitions. When using such a snapshot, then clearly all reads can proceed without blocking.

To ensure freshness, the snapshot timestamp st provided to a client is the largest timestamp such that all transactions with a commit timestamp smaller than or equal to st have been installed at all partitions. We call this timestamp the *local stable time* (LST), and the snapshot that it defines the *local stable snapshot*. The LST is determined by a stabilization protocol, by which partitions within a DC gossip the latest snapshots they have installed (§ III-C). In CANToR, when a transaction starts, it chooses a transaction coordinator, and it uses as its snapshot timestamp the LST value known to the coordinator.

Figure 1b depicts the nonblocking behavior of Wren. p_z proposes 5 as snapshot timestamp (because of p_x). Then c_1 can read without blocking on both p_x and p_y, despite the concurrent commit of T_2. The trade-off is that c_1 reads older versions of x and y, namely X_1 and Y_1, compared to the scenarion in Figure 1a, where it reads X_2 and Y_2.

Only assigning a snapshot slightly in the past, however, does not solve completely the issue of blocking reads. The local stable snapshot includes *all* the items that have been written by *all* clients up until the boundary defined by the snapshot and on which c (potentially) depends. The local stable snapshot, however, might not include the most recent writes performed by c in earlier transactions.

Consider, for example, the case in which c commits a transaction T, that includes a write on item x, and obtains a value ct as its commit timestamp. Subsequently, c starts another transaction T_i and obtains a snapshot timestamp st smaller than ct, because ct has not yet been installed at all partitions. If we were to let c read from this snapshot, and it were to read x, it would not see the value it had written previously in T.

A simple solution would be to block the commit of T until $ct \geq LST$. This would guarantee that c can issue its next transaction T' only after the modifications of T have been applied at *every partition in the DC*. This approach, however, introduces high commit latencies.

2) Client-side cache. Wren takes a different approach that leverages the fact that the only causal dependencies of c that may not be in the local stable snapshot are items that c has written itself in earlier transactions (e.g., x). Wren therefore provides clients with a private cache for such items: all items written by c are stored in its private cache, from which it reads when appropriate, as detailed below.

When starting a transaction, the client removes from the cache all the items that are included in the causal snapshot, in other words all items with commit timestamp lower than its causal snapshot time st. When reading x, a client first looks up x in its cache. If there is a version of x in the cache, it means that the client has written a version of x that is not included in the transaction snapshot. Hence, it *must* be read from the cache. Otherwise, the client reads x from p_x. In either case, the read is performed without blocking [2].

C. Dependency tracking and stabilization protocols

BDT. Wren implements BDT, a novel protocol to track the causal dependencies of items. The key feature of BDT is that every data item tracks dependencies by means of only two scalar timestamps, regardless of the scale of the system. One entry tracks the dependencies on local items and the other entry summarizes the dependencies on remote items.

The use of only two timestamps enables higher efficiency and scalability than other designs. State-of-the-art solutions employ dependency meta-data whose size grows with the number of DCs [8], [16], partitions [24] or causal dependencies [7], [15], [21], [25]. Meta-data efficiency is paramount for many applications dominated by very small items, e.g., Facebook [3], [26], in which meta-data can easily grow bigger than the item itself. Large meta-data increases processing, communication and storage overhead.

BiST. Wren relies on BDT to implement BiST, an efficient stabilization protocol to determine when updates can be included in the snapshots proposed to clients within a DC (i.e., when they are *visible* within a DC). BiST allows updates originating in a DC to become visible in that DC without waiting for the receipt of remote items. A remote update d, instead, is visible in a DC when it is *stable*, i.e., when all the causal dependencies of d have been received in the DC.

[2]The client can avoid contacting p_x, because Wren uses the last-writer-wins rule to resolve conflicting updates (see § II-A). With other conflict resolution methods, the client would always have to read the version of x from p_x, and apply the updates(s) in the cache to that version.

BiST computes two cut-off values that indicate, respectively, which local and remote items can become visible to transactions within a DC. The local component computed by BiST is the LST, which we described earlier. The remote component is the Remote Stable Time (RST), that, similarly to the LST, indicates a lower bound on remote snapshots that have been installed by every node within the local DC.

By decoupling local and remote items, BiST allows transactions to determine the visibility of local items without synchronizing with remote DCs [8], in contrast to systems that use a single scalar timestamp for dependency tracking [20], [24]. This decoupling enables availability and nonblocking reads also in the geo-replicated case, because a snapshot visible to a transaction includes only remote items that have already been received in the local DC.

With BiST, periodically partitions within a DC exchange the commit timestamp of the latest local and remote transactions they have applied. Then, each partition computes the LST, resp., RST, as minimum of the received timestamps corresponding to local, resp., remote transactions. Therefore, LST and RST reflect local and remote snapshots that have been already installed by all partitions in the DC, and from which transactions can read without blocking, as we explain in the following.

Snapshots and nonblocking reads. When a transaction T starts, the local, resp., remote, entry of the corresponding snapshot S is set to be the maximum between the LST, resp., RST on the coordinator and the highest LST, resp. RST, value seen by the client, ensuring that clients see monotonically increasing snapshots.

T uses the timestamps in S to determine the version of an item that it can read, namely the freshest version of an item that falls within the visible snapshot (let alone the items that are read from the client-side cache, as described in § III-B). S includes local, resp., remote, items whose timestamp is lower than the LST, resp., the RST. Because both LST and RST reflect snapshots installed by every partition in the DC, T can read from S in a nonblocking fashion.

Trade-off. BiST enables high scalability and performance at the expense of a slight increase in the time that it takes for an update to become visible in a DC (the so called *visibility latency*). By using BiST, Wren only tracks the lower bound on the commit time of local transactions (LST) and replicated transactions coming from all the remote DCs (RST). We describe this trade-off by sketching in Figure 2 how BiST and other existing stabilization protocols work at a high level.

In the example, the local DC (DC2) has committed transactions with timestamp up to 15. It has received commits from DC0 with timestamp up to 4, and from DC1 with timestamp up to 6. Wren exposes to transactions remote items with timestamp up to 4, the minimum of 4 and 6. Cure [8] uses one timestamp per DC, so transactions can see items from DC_0 with timestamp up to 4 and from DC_1 with timestamp up to 6. GentleRain [20] uses a single timestamp (the local one) to encode both local and remote snapshots, so transactions

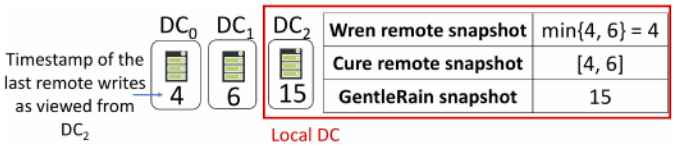


Fig. 2: Resource efficiency vs freshness in BiST (one partition per DC). DC_2 is the local DC.

can see all items up to timestamp 15. However, they have to block until the local DC has received *all* remote updates with timestamps lower than or equal to 15.

Timestamps. So far, we have assumed that Wren uses logical, Lamport clocks to generate timestamps. Wren, instead, uses Hybrid Logical Physical Clocks (HLC) [27]. In brief, an HLC is a logical clock whose value on a partition is the maximum between the local physical clock and the highest timestamp seen by the partition plus one. HLCs combine the advantages of logical and physical clocks. Like logical clocks, HLCs can be moved forward to match the timestamp of an incoming event. Like physical clocks, they advance in the absence of events and at approximately the same pace.

Wren's use of HLCs improves the freshness of the snapshot determined by BiST, which, as a by-product, also reduces the amount of data stored in the client-side caches. HLCs have previously been employed by CC systems to avoid waiting for physical clocks to catch up when generating timestamps for updates [28], [29]. HLCs alone, however, do not solve the problem of blocking reads with TCC. A snapshot timestamp can be "in the future" with respect to the installed snapshot of a partition, regardless of whether the clock is logical, physical or hybrid.

D. Fault tolerance and Availability

Fault tolerance (within a DC). Similarly to previous transactional systems based on 2PC, Wren can integrate fault-tolerance capabilities by means of standard replication techniques such as Paxos [30].

Wren preserves nonblocking reads, even if such fault tolerance mechanisms are enabled. In blocking systems, instead, fault tolerance increases the latency incurred by transactions upon blocking, because it increases the duration of a commit.

The failure of a server blocks the progress of BiST, but only during the short amount of time during which a backup partition has not yet replaced the failed one. The failure of a client does not affect the behavior of the system. The clients only keep local meta-data, and cache data that have already been committed to the data-store.

Availability (between DCs). BiST is always available. Transactions are never blocked or delayed as a result of the disconnection or failure of a DC. The disconnection of DC_i causes the RST to freeze in all DCs that get disconnected from DC_i. However, because BiST decouples local from remote dependencies, any RST assigned to a transaction refers to a snapshot that is already available in the DC, and the transaction can thus proceed. The LST, instead, always advances,

Algorithm 1 Wren client c (open session towards p_n^m).

1: **function** START
2: send $\langle$**StartTxReq** $\boldsymbol{lst_c, rst_c}\rangle$ to p_n^m
3: receive $\langle$**StartTxResp** $\boldsymbol{id, lst, rst}\rangle$ from p_n^m
4: $rst_c \leftarrow rst$; $lst_c \leftarrow lst$; $id_c \leftarrow id$
5: $RS_c \leftarrow \emptyset; WS_c \leftarrow \emptyset$
6: Remove from WC_c all items with commit timestamp up to lst_c
7: **end function**

8: **function** READ(χ)
9: $D \leftarrow \emptyset$; $\chi' \leftarrow \emptyset$
10: **for each** $k \in \chi$ **do**
11: $d \leftarrow$ check WS_c, RS_c, WC_c (in this order)
12: **if** $(d \neq NULL)$ **then** $D \leftarrow d$
13: **end for**
14: $\chi' \leftarrow \chi \setminus D.keySet()$
15: send $\langle$**TxReadReq** $\boldsymbol{id_c, \chi'}\rangle$ to p_n^m
16: receive $\langle$**TxReadResp** $\boldsymbol{D'}\rangle$ from p_n^m
17: $D \leftarrow D \cup D'$
18: $RS_c \leftarrow RS_c \cup D$
19: **return** D
20: **end function**

21: **function** WRITE(χ)
22: **for each** $\langle k, v\rangle \in \chi$ **do** ▷ *Update WS_c or write new entry*
23: **if** $(\exists d \in WS : d == k)$**then** $d.v \leftarrow v$ **else** $WS_c \leftarrow WS_c \cup \langle k, v\rangle$
24: **end for**
25: **end function**

26: **function** COMMIT ▷ *Only invoked if $WS \neq \emptyset$*
27: send $\langle$**CommitReq** $\boldsymbol{id_c, hwt_c, WS_c}\rangle$ to p_n^m
28: receive $\langle$**CommitResp** $\boldsymbol{ct}\rangle$ from p_n^m
29: $hwt_c \leftarrow ct$ ▷ *Update client's highest write time*
30: Tag WS_c entries with hwt_c
31: Move WS_c entries to WC_c ▷ *Overwrite (older) duplicate entries*
32: **end function**

Algorithm 2 Wren server p_n^m - transaction coordinator.

1: **upon** receive $\langle$**StartTxReq** $\boldsymbol{lst_c, rst_c}\rangle$ from c **do**
2: $rst_n^m \leftarrow max\{rst_n^m, rst_c\}$ ▷ *Update remote stable time*
3: $lst_n^m \leftarrow max\{lst_n^m, lst_c\}$ ▷ *Update local stable time*
4: $id_T \leftarrow generateUniqueId()$
5: $TX[id_T] \leftarrow \langle lst_n^m, \min\{rst_n^m, lst_n^m - 1\}\rangle$ ▷ *Save TX context*
6: send $\langle$**StartTxResp** $\boldsymbol{id_T, TX[id_T]}\rangle$ ▷ *Assign transaction snapshot*

7: **upon** receive $\langle$**TxReadReq** $\boldsymbol{id_T, \chi}\rangle$ from c **do**
8: $\langle lt, rt\rangle \leftarrow TX[id_T]$
9: $D \leftarrow \emptyset$
10: $\chi_i \leftarrow \{k \in \chi : partition(k) == i\}$ ▷ *Partitions with $\geq$ 1 key to read*
11: **for** $(i : \chi_i \neq \emptyset)$ **do**
12: send $\langle$**SliceReq** $\boldsymbol{\chi_i, lt, rt}\rangle$ to p_i^m
13: receive $\langle$**SliceResp** $\boldsymbol{D_i}\rangle$ from p_i^m
14: $D \leftarrow D \cup D_i$
15: **end for**
16: send $\langle$**TxReadResp** $\boldsymbol{D}\rangle$ to c

17: **upon** receive $\langle$**CommitReq** $\boldsymbol{id_T, hwt, WS}\rangle$ from c **do**
18: $\langle lt, rt\rangle \leftarrow TX[id_T]$
19: $ht \leftarrow max\{lt, rt, hwt\}$ ▷ *Max timestamp seen by the client*
20: $D_i \leftarrow \{\langle k, v\rangle \in WS : partition(k) == i\}$
21: **for** $(i : D_i \neq \emptyset)$ **do** ▷ *Done in parallel*
22: send $\langle$**PrepareReq** $\boldsymbol{id_T, lt, rt, ht, D_i}\rangle$ to p_i^m
23: receive $\langle$**PrepareResp** $\boldsymbol{id_T, pt_i}\rangle$ from p_i^m
24: **end for**
25: ct $\leftarrow max_{i:D_i \neq \emptyset}\{pt_i\}$ ▷ *Max proposed timestamp*
26: **for** $(i : D_i \neq \emptyset)$ **do** send $\langle$**Commit** $\boldsymbol{id_T, ct}\rangle$ to p_i^m **end for**
27: delete TX$[id_T]$ ▷ *Clear transactional context of c*
28: send $\langle$**CommitResp** $\boldsymbol{ct}\rangle$ to c

ensuring that clients can prune their local caches even if a DC disconnects.

IV. Protocols of Wren

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

A. Meta-data

Items. An item d is a tuple $\langle k, v, ut, rdt, id_T, sr\rangle$. k and v are the key and value of d, respectively. ut is the timestamp of d which is assigned upon commit of d and summarizes the dependencies on local items. rdt is the remote dependency time of d, i.e., it summarizes the dependencies towards remote items. id_T is the id of the transaction that created the item version. sr is the source replica of d.

Client. In a client session, a client c maintains id_c which identifies the current transaction, and lst_c and rst_c, that correspond to the local and remote timestamp of the transaction snapshot, respectively. c also stores the commit time of its last update transaction, represented with hwt_c. Finally, c stores WS_c, RS_c and WC_c corresponding to the client's write set, read set and client-side cache, respectively.

Servers. A server p_n^m is identified by the partition id (n) and the DC id (m). In our description, thus, m is the local DC of the server. Each server has access to a monotonically increasing physical clock, $Clock_n^m$. The local clock value on p_n^m is represented by the hybrid clock HLC_n^m.

p_n^m also maintains VV_n^m, a vector of HLCs with M entries. $VV_n^m[i], i \neq m$ indicates the timestamp of the latest update received by p_n^m that comes from the n-th partition at the i-th DC. $VV_n^m[m]$ is the version clock of the server and represents the local snapshot installed by p_n^m. The server also stores lst_n^m and rst_n^m. $lst_n^m = t$ indicates that p_n^m is aware that every partition in the local DC has installed a local snapshot with timestamp at least t. $rst_n^m = t'$ indicates that p_n^m is aware that every partition in the local DC has installed all the updates generated from all remote DCs with update time up to t'.

Finally, p_n^m keeps a list of prepared and a list of committed transactions. The former stores transactions for which p_n^m has proposed a commit timestamp and for which p_n^m is awaiting the commit message. The latter stores transactions that have been assigned a commit timestamp and whose modifications are going to be applied to p_n^m.

B. Operations

Start. Client c initiates a transaction T by picking at random a *coordinator* partition (denoted p_n^m) and sending it a start request with lst_c and rst_c. p_n^m uses these values to update its lst_n^m and rst_n^m, so that p_n^m can propose a snapshot that is at least as fresh as the one accessed by c in previous transactions. Then, p_n^m generates the snapshot visible to T. The local snapshot timestamp is lst_n^m. The remote one is set as the minimum between rst_n^m and $lst_n^m - 1$. Wren enforces the remote snapshot time to be lower than the local one, to efficiently deal with concurrent conflicting updates. Assume c wants to read x, that c has a version X_l in its private cache with commit timestamp $ct > lst_n^m$, and that there exist a visible remote X_r with commit timestamp $\geq ct$. Then, c must retrieve X_r, its commit timestamp and its source replica to determine whether X_l or X_r should be read according to the last writer wins rule. By forcing the remote stable time to be lower than

Algorithm 3 Wren server p_n^m - transaction cohort.

1: **upon** receive $\langle$**SliceReq** $\boldsymbol{\chi}, \boldsymbol{lt}, \boldsymbol{rt}\rangle$ from p_i^m **do**
2: $rst_n^m \leftarrow max\{rst_n^m, rst\}$ ▷ *Update remote stable time*
3: $lst_n^m \leftarrow max\{lst_n^m, lst\}$ ▷ *Update local stable time*
4: $D \leftarrow \emptyset$
5: **for** $(k \in \chi)$ **do**
6: $D_k \leftarrow \{d : d.k == k\}$ ▷ *All versions of k*
7: $D_{lv} \leftarrow \{d : d.sr == m \wedge d.ut \leq lt \wedge d.rst \leq rt\}$ ▷ *Local visible*
8: $D_{rv} \leftarrow \{d : d.sr \neq m \wedge d.ut \leq rt \wedge d.rst \leq lt\}$ ▷ *Remote visible*
9: $D_{kv} \leftarrow \{D_k \cap \{D_{lv} \cup D_{rv}\}\}$ ▷ *All visible versions of k*
10: $D \leftarrow D \cup \{argmax_{d.ut}\{d \in D_{kv}\}\}$ ▷ *Freshest visible vers. of k*
11: **end for**
12: reply $\langle$***SliceResp*** $\boldsymbol{D}\rangle$ to p_i^m

13: **upon** receive $\langle$**PrepareReq** $\boldsymbol{id_T, lt, rt, ht, D_i}\rangle$ from p_i^m **do**
14: $HLC_n^m \leftarrow max(Clock_n^m, ht + 1, HLC_n^m + 1)$ ▷ *Update HLC*
15: pt $\leftarrow HLC_n^m$ ▷ *Proposed commit time*
16: $lst_n^m \leftarrow max\{lst_n^m, lt\}$ ▷ *Update local stable time*
17: $rst_n^m \leftarrow max\{rst_n^m, rt\}$ ▷ *Update remote stable time*
18: $Prepared_n^m \leftarrow Prepared_n^m \cup \{id_T, rt, D_i\}$ ▷ *Append to pending list*
19: send $\langle$**PrepareResp** $\boldsymbol{id_T, pt}\rangle$ to p_i^m

20: **upon** receive $\langle$**CommitReq** $\boldsymbol{id_T, ct}\rangle$ from c **do**
21: $HLC_n^m \leftarrow max(HLC_n^m, ct, Clock_n^m)$ ▷ *Update HLC*
22: $\langle id_T, rst, D\rangle \leftarrow \{\langle i, r, \phi\rangle \in Prepared_n^m : i == id_T\}$
23: $Prepared_n^m \leftarrow Prepared_n^m \setminus \{\langle id_T, rst, D\rangle\}$ ▷ *Remove from pending*
24: $Committed_n^m \leftarrow Committed_n^m \cup \{\langle id_T, ct, rst, D\}$ ▷ *Mark to commit*

Algorithm 4 Wren server p_n^m - Auxiliary functions.

1: **function** UPDATE($\boldsymbol{k, v, ut, rdt, id_T}$)
2: create $d : \langle d.k, d.v, d.ut, d.rdt, d.id_T, d.sr\rangle \leftarrow \langle k, v, ut, rdt, id_T, m\rangle$
3: insert new item d in the version chain of key k
4: **end function**

5: **upon** Every Δ_R **do**
6: **if** $(Prepared_n^m \neq \emptyset)$ **then** $ub \leftarrow min_{\{p.pt\}}\{p \in Prepared_n^m\} - 1$
7: **else** $ub \leftarrow max\{Clock_n^m, HLC_n^m\}$ **end if**
8: **if** $(Committed_n^m \neq \emptyset)$ **then** ▷ *Commit tx in increasing order of ct*
9: $C \leftarrow \{\langle id, ct, rst, D\rangle\} \in Committed_n^m : ct \leq ub$
10: **for** $(T \leftarrow \{\langle id, rst, D\rangle\} \in (group\ C\ by\ ct))$ **do**
11: **for** $(\langle id, rst, D\rangle \in T)$ **do**
12: **for** $(\langle k, v\rangle \in D)$ **do** update $(\boldsymbol{k, v, ct, rst, id})$ **end for**
13: **end for**
14: **for** $(i \neq m)$ send $\langle$**Replicate** $\boldsymbol{T, ct}\rangle$ to p_n^i **end for**
15: $Committed_n^m \leftarrow Committed_n^m \setminus T$
16: **end for**
17: $VV_n^m[m] \leftarrow ub$ ▷ *Set version clock*
18: **else**
19: $VV_n^m[m] \leftarrow ub$ ▷ *Set version clock*
20: **for** $(i \neq m)$ **do** send $\langle$**Heartbeat** $\boldsymbol{VV_n^m[m]}\rangle$ to p_n^i **end for**
21: **end if**

22: **upon** receive $\langle$**Replicate** $\boldsymbol{T, ct}\rangle$ from p_n^i **do**
23: **for** $(\langle id, rst, D\rangle \in T)$ **do**
24: **for** $(\langle k, v\rangle \in D)$ **do** update $(\boldsymbol{k, v, ct, rst, id})$ **end for**
25: **end for**
26: $VV_n^m[i] \leftarrow ct$ ▷ *Update remote snapshot of i-th replica*

27: **upon** receive $\langle$**Heartbeat t**$\rangle$ from p_n^i **do**
28: $VV_n^m[i] \leftarrow t$ ▷ *Update remote snapshot of i-th replica*

29: **upon** Every Δ_G **do** ▷ *Compute remote and local stable snapshots*
30: $rst_n^m \leftarrow min_{\{i=0,\ldots,M-1, i\neq m; j=0,\ldots,N-1\}} VV_j^m[i]$ ▷ *Remote*
31: $lst_n^m \leftarrow min_{\{i=0,\ldots,N-1\}} VV_i^m[m]$ ▷ *Local*

lst – and hence of ct – the client knows that the freshest visible version of x is X_l, which can be read locally from the private cache [3].

After defining the snapshot visible to T, p_n^m also generates a unique identifier for T, denoted id_T, and inserts T in a private data structure. p_n^m replies to c with id_T and the snapshot timestamps.

Upon receiving the reply, c updates lst_c and rst_c, and evicts from the cache any version with timestamp lower than lst_c. c can prune the cache using lst_c because p_n^m has enforced that the highest remote timestamp visible to T is lower than lst_n^m. This ensures that if, after pruning, there is a version X in the private cache of c, then $X.ct > lst$ and hence the freshest version of x visible to c is X.

Read. The client c provides the set of keys to read. For each key k to read, c searches the write-set, the read-set and the client cache, in this order. If an item corresponding to k is found, it is added to the set of items to return, ensuring read-your-own-writes and repeatable-reads semantics. Reads for keys that cannot be served locally are sent in parallel to the corresponding partitions, together with the snapshot from which to serve them. Upon receiving a read request, a server first updates the server's LST and RST, if they are smaller than the client's (Alg. 2 Lines 2–3). Then, the server returns to the client, for each key, the version within the snapshot with the highest timestamp (Alg. 3 Lines 6–10). c inserts the returned items in the read-set.

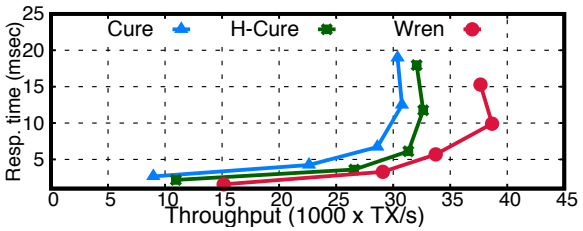
Write. Client c locally buffers the writes in its write-set WS_c. If a key being written is already present in WS_c, then it is updated; otherwise, it is inserted.

Commit. The client sends a commit request to the coordinator with the content of WS_c, the id of the transaction and the commit of its last update transaction hwt_c, if any. The coordinator contacts the partitions that store the keys that need to be updated (the cohorts) and sends them the corresponding updates and hwt_c. The partitions update their HLCs, propose a commit timestamp and append the transaction to the pending list. To reflect causality, the proposed timestamp is higher than the snapshot timestamps and hwt_c. The coordinator then picks the maximum among the proposed timestamps [32], sends it to the cohort partitions, clears the local context of the transaction and sends the commit timestamp to the client. The cohort partitions move the transaction from the pending list to the commit list, with the new commit timestamp.

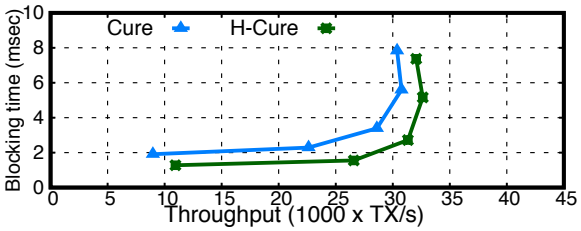
Applying and replicating transactions. Periodically, the servers apply the effects of committed transactions, in increasing commit timestamp order (Alg. 4 Lines 6-20). p_n^m applies the modifications of transactions that have a commit timestamp lower than the lowest timestamp present in the pending list. This timestamp represents the lower bound on the commit timestamps of future transactions on p_n^m. After applying the transactions, p_n^m updates its local version clock and replicates the transactions to remote DCs. When there are more transactions with the same commit time ct, p_n^m updates its local version clock only after applying the last transaction with the same ct and packs them together to be propagated in one replication message (Alg. 4 Lines 10–17).

If a server does not commit a transaction for a given amount of time, it sends a heartbeat with its current HLC to its peer

[3]The likelihood of rst_n^m being higher than lst_n^m is low given that i) geo-replication delays are typically higher than the skew among the physical clocks [31] and ii) rst_n^m is the minimum value across all timestamps of the latest updates received in the local DC.



(a) Throughput vs average TX latency.



(b) Mean blocking time in Cure and H-Cure. Wren never blocks.

Fig. 3: Performance of Wren, H-Cure and Cure on 3 DCs, 8 partitions/DC, 4 partitions involved per transaction, and 95:5 r:w ratio. Wren achieves better latencies because it never blocks reads (a). H-Cure achieves performance in-between Cure and Wren, showing that only using HLCs does not solve the problem of blocking reads in TCC. Cure and H-Cure incur a mean blocking time that grows with the load (b). Because of blocking, Cure and H-Cure need higher concurrency to fully utilize the resources on the servers. This leads to higher contention on physical resources and to a lower throughput (a).

replicas, ensuring the progress of the RST.

BiST. Periodically, partitions within a DC exchange their version vectors. The LST is computed as the minimum across the local entries in such vectors; the RST as minimum across the remote ones (Alg. 4 Lines 30–32). Partitions within a DC are organized as a tree to reduce communication costs [20].

Garbage collection. Periodically, the partitions within a DC exchange the oldest snapshot corresponding to an active transaction (p_n^m sends its current visible snapshot if it has is no running transaction). The aggregate minimum determines the oldest snapshot S_{old} that is visible to a running transaction. The partitions scan the version chain of each key backwards and keep the all the versions up to (and including) the oldest one within S_{old}. Earlier versions are removed.

C. Correctness

Because of space constraints, we provide only a high-level argument to show the correctness of Wren.

Snapshots are causal. To start a transaction, a client c piggybacks the freshest snapshot it has seen, ensuring the monotonicity of the snapshot seen by c (Alg. 2 Lines 1–6). Commit timestamps reflect causality (Alg. 2 Line 19), and BiST tracks a lower bound on the snapshot installed by every partition in a DC. If X is within the snapshot of a transaction, so are its dependencies, because $i)$ dependencies generated in the same DC where X is created have a timestamp lower than X and $ii)$ dependencies generated in a remote DC have a timestamp lower than $X.rdt$. On top of the snapshot provided by the coordinator, the client applies its writes that are not in the snapshot. These writes cannot depend on items created by other clients that are outside the snapshot visible to c.

Writes are atomic. Items written by a transaction have the same commit timestamp and RST. LST and RST are computed as the minimum values across all the partitions within a DC. If a transaction has written X and Y and a snapshot contains X, then it also contains Y (and vice-versa).

V. Evaluation

We evaluate the performance of Wren in terms of throughput, latency and update visibility. We compare Wren with Cure [8], the state-of-the-art approach to TCC, and with H-Cure, a variant of Cure that uses HLCs. By comparing with H-Cure, we show that using HLCs alone, as in existing systems [29], [33], is not sufficient to achieve the same performance as Wren, and that nonblocking reads in the presence of multi-item atomic writes are essential.

A. Experimental environment

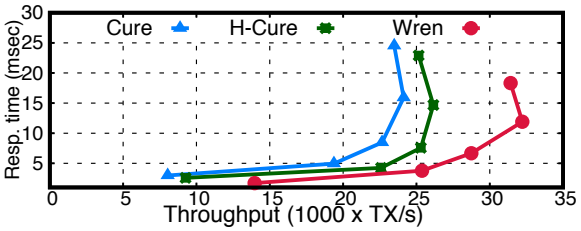
Platform. We consider a geo-replicated setting deployed across up to 5 replication sites on Amazon EC2 (Virginia, Oregon, Ireland, Mumbai and Sydney). When using 3 DCs, we use Virginia, Oregon and Ireland. In each DC we use up to 16 servers (m4.large instances with 2 VCPUs and 8 GB of RAM). We spawn one client process per partition in each DC. Clients issue requests in a closed loop, and are collocated with the server partition they use as coordinator. We spawn different number of client threads to generate different load conditions. In particular, we spawn 1, 2, 4, 8, 16 threads per client process. Each "dot" in the curve plots corresponds to a different number of threads per client.

Implementation. We implement Wren, H-Cure and Cure in the same C++ code-base [4]. All protocols implement the last-writer-wins rule for convergence. We use Google Protobufs for communication, and NTP to synchronize physical clocks. The stabilization protocols run every 5 milliseconds.

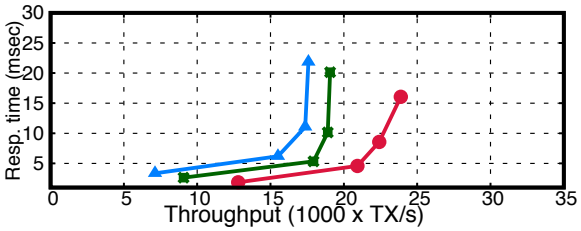
Workloads. We use workloads with 95:5, 90:10 and 50:50 r:w ratios. These are standard workloads also used to benchmark other TCC systems [8], [16], [34]. In particular, the 50:50 and 95:5 r:w ratio workloads correspond, respectively, to the update-heavy (A) and read-heavy (B) YCSB workloads [35]. Transactions generate the three workloads by executing 19 reads and 1 write (95:5), 18 reads and 2 writes (90:10), and 10 reads and 10 writes (50:50). A transaction first executes all reads in parallel, and then all writes in parallel.

Our default workload uses the 95:5 r:w ratio and runs transactions that involve 4 partitions on a platform deployed over 3 DCs and 8 partitions. We also consider variations of this workload in which we change the value of one parameter

[4]https://github.com/epfl-labos/wren

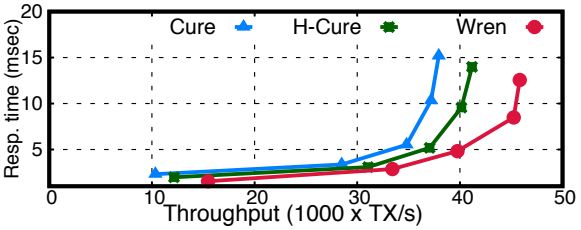


(a) Throughput vs average TX latency (90:10 r:w).

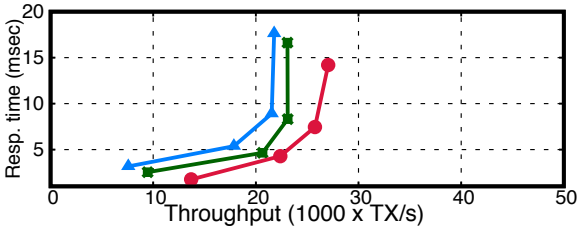


(b) Throughput vs average TX latency (50:50 r:w).

Fig. 4: Performance of Wren, Cure and H-Cure with different 90:10 (a) and 50:50 (b) r:w ratios, 4 partitions involved per transaction (3DCs, 8 partitions). Wren outperforms Cure and H-Cure for both read-heavy and write-heavy workloads.



(a) Throughput vs average TX latency (p=2).



(b) Throughput vs average TX latency (p=8).

Fig. 5: Performance of Wren, Cure and H-Cure with transactions that read from 2 (a) and 8 (b) partitions with 95:5 r:w ratio (3DCs, 8 partitions). Wren outperforms Cure and H-Cure with both small and large transactions.

and keep the others at their default values. Transactions access keys within a partition according to a zipfian distribution, with parameter 0.99, which is the default in YCSB and resembles the strong skew that characterizes many production systems [26], [36], [37]. We use small items (8 bytes), which are prevalent in many production workloads [26], [36]. With bigger items Wren would retain the benefits of its nonblocking reads. The effectiveness of BDT and BiST would naturally decrease as the size of the items increases, because meta-data overhead would become less critical.

B. Performance evaluation

Latency and throughput. Figure 3a reports the average transaction latency vs. throughput achieved by Wren, H-Cure and Cure with the default workload. Wren achieves up to 2.33x lower response times than Cure, because it never blocks a read due to clock skew or to wait for a snapshot to be installed. Wren also achieves up to 25% higher throughput than Cure. Cure needs a higher number of concurrent clients to fully utilize the processing power left idle by blocked reads. The presence of more threads creates more contention on the physical resources and implies more synchronization to block and unblock reads, which ultimately leads to lower throughput.

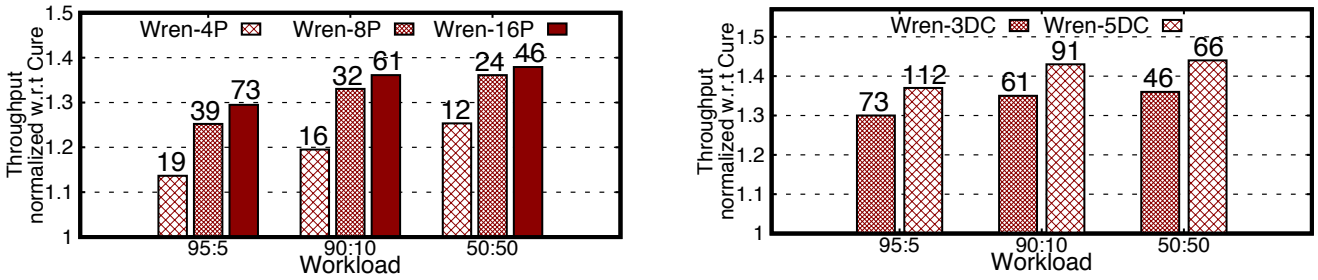
Wren also outperforms H-Cure, achieving up to 40% lower latency and up to 15% higher throughput. HLCs enable H-Cure to avoid blocking the read of a transaction T because of clock skew. This blocking happens on a partition if the local timestamp of T's snapshot is t, there are no pending or committed transactions on the partition with commit timestamp lower than t, but the physical clock on the partition is lower than t. HLCs, however, cannot avoid blocking T if there are pending transactions on the partition, and T is assigned a snapshot that has not been installed on the partition.

Statistics on blocking in Cure and H-Cure. Figure 3b provides insights on the blocking occurring in Cure and H-Cure, that leads to the aforementioned performance differences. The plots show the mean blocking time of transactions that block upon reading. A transaction T is considered as blocked if at least one of its individual reads blocks. The blocking time of T is computed as the maximum blocking time of a read belonging to T.

Blocking can take up a vast portion of a transaction execution time. In Cure, blocking reads introduce a delay of 2 milliseconds at low load, and almost 4 milliseconds at high load (without considering overload conditions). These values correspond to 35-48% of the total mean transaction execution time. Similar considerations hold for H-Cure. The blocking time increases with the load, because higher load leads to more transactions being inserted in the pending and commit queues, and to higher latency between the time a transaction is committed and the corresponding snapshot is installed.

C. Varying the workload

Figure 4a and Figure 4b report the average transaction latency as a function of the load for the 90:10 and 50:50 r:w ratios, respectively. Figure 5a and Figure 5b report the same metric with the default r:w ratio of 95:5, but with $p = 2$ and $p = 8$ partitions involved in a transaction, respectively. These figures show that Wren delivers better performance than Cure and H-Cure for a wide range of workloads. It achieves transaction latencies up to 3.6x lower then Cure, and up to 1.6x lower than H-Cure. It achieves maximum throughput up to 1.33x higher than Cure and 1.23x higher than H-Cure. The



(a) Throughput when varying the number of partitions/DC (3 DCs). (b) Throughput when varying the number of DCs (16 partitions/DCs).

Fig. 6: Throughput achieved by Wren when increasing the number of partition per DC (a) and DCs in the system (b). Each bar represents the throughput of Wren normalized w.r.t. to Cure (y axis starts from 1). The number on top of each bar reports the absolute value of the throughput achieved by Wren in 1000 x TX/s. Wren consistently achieves better throughput than Cure and achieves good scalability both when increasing the number of partitions and the number of DCs.

peak throughput of all three systems decreases with a lower r:w ratio, because writing more items increases the duration of the commit and the replication overhead. Similarly, a higher value of p decreases throughput, because more partitions are contacted during a transaction.

D. Varying the number of partitions

Figure 6a reports the throughput achieved by Wren with 4, 8 and 16 partitions per DC. The bars represent the throughput of Wren normalized with respect to the throughput achieved by Cure in the same setting. The number on top of each bar represents the absolute throughput achieved by Wren.

The plots show three main results. First, Wren consistently achieves higher throughput than Cure, with a maximum improvement of 38%. Second, the performance improvement of Wren is more evident with more partitions and lower r:w ratios. More partitions touched by transactions and more writes increase the chances that a read in Cure targets a laggard partition and blocks, leading to higher latencies, lower resource efficiency, and worse throughput. Third, Wren provides efficient support for application scale-out. When increasing the number of partitions from 4 to 16, throughput increases by 3.76x for the write-heavy and 3.88x for read-heavy workload, approximating the ideal improvement of 4x.

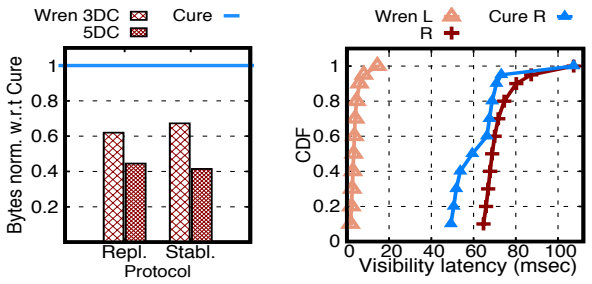


Fig. 7: (a) BiST incurs lower overhead than Cure to track the dependencies of replicated updates and to determine transactional snapshots (default workload). (b) Wren achieves a slightly higher **R**emote update visibility latency w.r.t. Cure, and makes **L**ocal updates visible when they are within the local stable snapshot (3 DCs).

E. Varying the number of DCs

Figure 6b shows the throughput achieved by Wren with 3 and 5 DCs (16 partitions per DC). The bars represent the throughput normalized with respect to Cure's throughput in the same scenario. The numbers on top of the bars indicate the absolute throughput achieved by Wren.

Wren obtains higher throughput than Cure for all workloads, achieving an improvement of up to 43%. Wren performance gains are higher with 5 DCs, because the meta-data overhead is constant in BiST, while in Cure it grows linearly with the number of DCs. The throughput achieved by Wren with 5 DCs is 1.53x, 1.49x, and 1.44x higher than the throughput achieved with 3 DCs, for the 95:5, 90:10 and 50:50 workloads, respectively, approximating the ideal improvement of 1.66x. A higher write intensity reduces the performance gain when scaling from 3 to 5 DCs, because it implies more updates being replicated.

F. Resource efficiency

Figure 7a shows the amount of data exchanged in Wren to run the stabilization protocol and to replicate updates, with the default workload. The results are normalized with respect to the amounts of data exchanged in Cure at the same throughput. With 5 DCs, Wren exchanges up to 37% fewer bytes for replication and up to 60% fewer bytes for running the stabilization protocol. With 5 DCs, updates, snapshots and stabilization messages carry 2 timestamps in Wren versus 5 in Cure.

G. Update visibility

Figure 7b shows the CDF of the update visibility latency with 3 DCs. The visibility latency of an update X in DC_i is the difference between the wall-clock time when X becomes visible in DC_i and the wall-clock time when X was committed in its original DC (which is DC_i itself in the case of local visibility latency). The CDFs are computed as follows: we first obtain the CDF on every partition and then we compute the mean for each percentile.

Cure achieves lower update visibility latencies than Wren. The remote update visibility time in Wren is slightly higher

than in Cure (68 vs. 59 milliseconds in the worst case, i.e., 15% higher), because Wren tracks dependencies at the granularity of the DC, while Wren only tracks local and remote dependencies (see § III-C Figure 2). Local updates become visible immediately in Cure. In Wren they become visible after a few milliseconds, because Wren chooses a slightly older snapshot. We argue that these slightly higher update visibility latencies are a small price to pay for the performance improvements offered by Wren.

VI. Related Work

TCC systems. In Cure [8] a transaction T can be assigned a snapshot that has not been installed by some partitions. If T reads from any of such laggard partitions, it blocks. Wren, on the contrary, achieves low-latency nonblocking reads by either reading from a snapshot that is already installed in all partitions, or from the client-side cache.

Occult [34] implements a master-slave design in which only the master replica of a partition accepts writes and replicates them asynchronously. The commit of a transaction, then, may span multiple DCs. A replicated item can be read before its causal dependencies are received, hence achieving the lowest data staleness. However, a read may have have to be retried several times in case of missing dependencies, and may even have to contact the remote master replica, which might not be accessible due to a network partition. The effect of retrying in Occult has a negative impact on performance, that is comparable to blocking the read to receive the right value to return. Wren, instead, implements always-available transactions that complete wholly within a DC, and never block nor retry read operations.

In SwiftCloud [16] clients declare the items in which they are interested, and the system sends them the corresponding updates, if any. SwiftCloud uses a sequencer-based approach, which totally orders updates, both those generated in a DC and those received from remote DCs. The sequencer-based approach ensures that the stream of updates pushed to clients is causally consistent. However, sequencing the updates also makes it cumbersome to achieve horizontal scalability. Wren, instead, implements decentralized protocols that efficiently enable horizontal scalability.

Cure and SwiftCloud use dependency vectors with one entry per DC. Occult uses one dependency timestamp per master replica. By contrast, Wren timestamps items and snapshots with constant dependency meta-data, which increases resource efficiency and scalability.

The trade-off that Wren makes to achieve low latency, availability and scalability is that it exposes snapshots slightly older than those exposed by other TCC systems.

CC systems. Many CC systems provide weaker semantics than TCC. COPS [15], Orbe [24], GentleRain [20], ChainReaction [25], POCC [38] and COPS-SNOW [7] implement read-only transactions. Eiger [21] additionally supports write-only transactions. These systems either block a read while waiting for the receipt of *remote* updates [20], [24], [38], require a large amount of meta-data [7], [15], [21], or rely on a sequencer process per DC [25].

Highly available transactional systems. Bailis et al. [39], [40] propose several flavors of transactional protocols that are available and support read-write transactions. These protocols rely on fine-grained dependency tracking and enforce a consistency level that is weaker than CC. TARDiS [41] supports merge functions over conflicting *states* of the application, rather than at key granularity. This flexibility requires a significant amount of meta-data and a resource-intensive garbage collection scheme to prune old states. Moreover, TARDiS does not implement sharding. GSP [42] is an operational model for replicated data that supports highly available transactions. GSP targets non-partitioned data stores and uses a system-wide broadcast primitive to totally order the updates. Wren, instead, is designed for applications that scale-out by sharding and achieves scalability and consistency by lightweight protocols.

Strongly consistent transactional systems. Many systems support geo-replication with consistency guarantees stronger than CC (e.g., Spanner [1], Walter [43], Gemini [44], Lynx [45], Jessy [46], Clock-SRM[47], SDUR [48] and Droopy [49]). These systems require cross-DC coordination to commit transactions, hence they are not always-available [11], [19]. Wren targets a class of applications that can tolerate a weaker form of consistency, and for these applications it provides low latency, high throughput, scalability and availability.

Client-side caching. Caching at the client side is a technique primarily used to support disconnected clients, especially in mobile and wide area network settings [50], [51], [52]. Wren, instead, uses client-side caching to guarantee consistency.

VII. Conclusion

We have presented Wren, the first TCC system that at the same time implements nonblocking reads thereby achieving low latency and allows applications to scale-out by sharding. Wren implements a novel transactional protocol, CANToR, that defines transaction snapshots as the union of a fresh causal snapshot and the contents of a client-side cache. Wren also introduces BDT, a new dependency tracking protocol, and BiST, a new stabilization protocol. BDT and BiST use only 2 timestamps per update and per snapshot, enabling scalability regardless of the size of the system. We have compared Wren with the state-of-the-art TCC system, and we have shown that Wren achieves lower latencies and higher throughput, while only slightly penalizing the freshness of data exposed to clients.

Acknowledgements

We thank the anonymous reviewers, Fernando Pedone, Sandhya Dwarkadas, Richard Sites and Baptiste Lepers for their valuable suggestions and helpful comments. This research has been supported by The Swiss National Science Foundation through Grant No. 166306, by an EcoCloud post-doctoral research fellowship, and by Amazon through AWS Cloud Credits.

REFERENCES

[1] J. C. Corbett, J. Dean, M. Epstein, and et al., "Spanner: Google's Globally-distributed Database," in *Proc. of OSDI*, 2012.

[2] G. DeCandia, D. Hastorun, M. Jampani, and et al., "Dynamo: Amazon's Highly Available Key-value Store," in *Proc. of SOSP*, 2007.

[3] R. Nishtala, H. Fugal, S. Grimm, and et al., "Scaling Memcache at Facebook," in *Proc. of NSDI*, 2013.

[4] S. A. Noghabi, S. Subramanian, P. Narayanan, and et al., "Ambry: LinkedIn's Scalable Geo-Distributed Object Store," in *Proc. of SIGMOD*, 2016.

[5] A. Verbitski, A. Gupta, D. Saha, and et al., "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases," in *Proc. of SIGMOD*, 2017.

[6] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. a. Paulo, J. Pereira, and R. Vilaça, "MeT: Workload Aware Elasticity for NoSQL," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, 2013.

[7] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, "The SNOW Theorem and Latency-Optimal Read-Only Transactions," in *OSDI*, 2016.

[8] D. D. Akkoorath, A. Tomsic, M. Bravo, and et al., "Cure: Strong semantics meets high availability and low latency," in *Proc. of ICDCS*, 2016.

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

[10] H. Attiya, F. Ellen, and A. Morrison, "Limitations of Highly-Available Eventually-Consistent Data Stores," in *Proc. of PODC*, 2015.

[11] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, Availability, Convergence," Computer Science Department, University of Texas at Austin, Tech. Rep. TR-11-22, May 2011.

[12] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[13] K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast," *ACM Trans. Comput. Syst.*, vol. 9, no. 3, pp. 272–314, Aug. 1991.

[14] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing High Availability Using Lazy Replication," *ACM Trans. Comput. Syst.*, vol. 10, no. 4, pp. 360–391, Nov. 1992.

[15] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS," in *Proc. of SOSP*, 2011.

[16] M. Zawirski, N. Preguiça, S. Duarte, and et al., "Write Fast, Read in the Past: Causal Consistency for Client-Side Applications," in *Proc. of Middleware*, 2015.

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

[18] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[19] E. A. Brewer, "Towards Robust Distributed Systems (Abstract)," in *Proc. of PODC*, 2000.

[20] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks," in *Proc. of SoCC*, 2014.

[21] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger Semantics for Low-latency Geo-replicated Storage," in *Proc. of NSDI*, 2013.

[22] R. H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180–209, Jun. 1979.

[23] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free Replicated Data Types," in *Proc. of SSS*, 2011.

[24] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks," in *Proc. of SoCC*, 2013.

[25] S. Almeida, J. a. Leitão, and L. Rodrigues, "ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication," in *Proc. of EuroSys*, 2013.

[26] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," in *Proc. of SIGMETRICS*, 2012.

[27] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical Physical Clocks," in *Proc. of OPODIS*, 2014.

[28] C. Gunawardhana, M. Bravo, and L. Rodrigues, "Unobtrusive Deferred Update Stabilization for Efficient Geo-Replication," in *Proc. of ATC*, 2017.

[29] M. Roohitavaf, M. Demirbas, and S. Kulkarni, "CausalSpartan: Causal Consistency for Distributed Data Stores using Hybrid Logical Clocks," in *SRDS*, 2017.

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

[31] H. Lu, K. Veeraraghavan, P. Ajoux, and et al., "Existential Consistency: Measuring and Understanding Consistency at Facebook," in *Proc. of SOSP*, 2015.

[32] J. Du, S. Elnikety, and W. Zwaenepoel, "Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks," in *Proc of SRDS*, 2013.

[33] D. Didona, K. Spirovska, and W. Zwaenepoel, "Okapi: Causally Consistent Geo-Replication Made Faster, Cheaper and More Available," *ArXiv e-prints, https://arxiv.org/abs/1702.04263*, Feb. 2017.

[34] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades," in *Proc. of NSDI*, 2017.

[35] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. of SoCC*, 2010.

[36] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, "Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores," in *Proc. of SIGMOD*, 2015.

[37] O. Balmau, D. Didona, R. Guerraoui, and et al., "TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores," in *Proc. of ATC*, 2017.

[38] K. Spirovska, D. Didona, and W. Zwaenepoel, "Optimistic Causal Consistency for Geo-Replicated Key-Value Stores," in *Proc. of ICDCS*, 2017.

[39] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly Available Transactions: Virtues and Limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 181–192, Nov. 2013.

[40] P. Bailis, A. Fekete, J. M. Hellerstein, and et al., "Scalable Atomic Visibility with RAMP Transactions," in *Proc. of SIGMOD*, 2014.

[41] N. Crooks, Y. Pu, N. Estrada, T. Gupta, L. Alvisi, and A. Clement, "TARDiS: A Branch-and-Merge Approach To Weak Consistency," in *Proc. of SIGMOD*, 2016.

[42] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fahndrich, "Global Sequence Protocol: A Robust Abstraction for Replicated Shared State," in *Proceedings of ECOOP*, 2015.

[43] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional Storage for Geo-replicated Systems," in *Proc. of SOSP*, 2011.

[44] V. Balegas, C. Li, M. Najafzadeh, and et al., "Geo-Replication: Fast If Possible, Consistent If Necessary," *Data Engineering Bulletin*, vol. 39, no. 1, pp. 81–92, Mar. 2016.

[45] Y. Zhang, R. Power, S. Zhou, and et al., "Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems," in *Proc. of SOSP*, 2013.

[46] M. S. Ardekani, P. Sutra, and M. Shapiro, "Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems," in *Proc. of SRDS*, 2013.

[47] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone, "Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks," in *Proc. of DSN*, 2014.

[48] D. Sciascia and F. Pedone, "Geo-replicated storage with scalable deferred update replication," in *Proc. of DSN*, 2013.

[49] S. Liu and M. Vukolić, "Leader Set Selection for Low-Latency Geo-Replicated State Machine," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1933–1946, July 2017.

[50] M. E. Bjornsson and L. Shrira, "BuddyCache: High-performance Object Storage for Collaborative Strong-consistency Applications in a WAN," in *Proc. of OOPSLA*, 2002.

[51] D. Perkins, N. Agrawal, A. Aranya, and et al., "Simba: Tunable End-to-end Data Consistency for Mobile Apps," in *Proc. of EuroSys*, 2015.

[52] I. Zhang, N. Lebeck, P. Fonseca, and et al., "Diamond: Automating Data Management and Storage for Wide-Area, Reactive Applications," in *Proc. of OSDI*, 2016.