2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication

Kristina Spirovska[1], Diego Didona[1], Willy Zwaenepoel[1,2]
[1] EPFL, [2] University of Sydney

***Abstract*—Geo-replicated data platforms are the backbone of several large-scale online services. Transactional Causal Consistency (TCC) is an attractive consistency level for building such platforms. TCC avoids many anomalies of eventual consistency, eschews the synchronization costs of strong consistency, and supports interactive read-write transactions. Partial replication is another attractive design choice for building geo-replicated platforms, as it reduces storage requirements and update propagation costs.**

This paper presents PaRiS, the first TCC system that supports partial replication and implements non-blocking parallel read operations. The latter reduce read latency which is of paramount importance for the performance of read-intensive applications. PaRiS relies on a novel protocol to track dependencies, called Universal Stable Time (UST). By means of a lightweight background gossip process, UST identifies a snapshot of the data that has been installed by every data center (DC) in the system. Hence, transactions can consistently read from such a snapshot on any server in any replication site without having to block. Moreover, PaRiS requires only one timestamp to track dependencies and define transactional snapshots, thereby achieving resource efficiency and scalability.

We evaluate PaRiS on an AWS deployment composed of up to 10 replication sites. We demonstrate a performance gain of non-blocking reads vs. a blocking alternative (up to 1.47x higher throughput with 5.91x lower latency for read-dominated workloads and up to 1.46x higher throughput with 20.56x lower latency for write-heavy workloads). We also show that the throughput penalty incurred to implement causal consistency, compared to variant without the causal consistency guarantees, is as low as 20% for read-heavy workloads and 37% for write-heavy workloads. We furthermore show that PaRiS scales well with the number of DCs and partitions, while being able to handle larger data-sets than existing solutions that assume full replication.

I. Introduction

Modern large-scale data platforms rely on geo-replication and sharding to store and manipulate large volumes of data efficiently [1]–[4]. Geo-replication allows keeping a copy of the data in a data center (DC) closer to the users, thus reducing access latencies. Sharding enables horizontal scalability, by slicing the dataset in disjoint partitions, each of which can be assigned to a different server.

In geo-replicated environments, partial replication is an effective technique to reduce storage requirements and replication costs. In partial replication, each DC stores only a subset of the partitions, as opposed to the full replication case, where each DC stores the whole dataset. Hence, the system can scale to a higher number of partitions with respect to a full replication approach, and updates performed in one DC are propagated to fewer replicas.

Causal Consistency (CC) has emerged as an attractive consistency model for geo-replicated data platforms [5]–[15]. CC provides an intuitive semantics and avoids many anomalies allowed by weaker models, such as eventual consistency [16]. Moreover, CC eschews the high synchronization costs of stronger consistency levels, such as linearizability [17]. Transactional CC (TCC) [7], [8] extends CC by providing transactions that observe a causally consistent view of the data, and can perform atomic multi-object writes.

PaRiS. This paper presents PaRiS, the first system that implements TCC on a partially replicated data platform, and that supports non-blocking parallel read operations (and hence, non-blocking read-only transactions). Parallel non-blocking reads are an important requirement to guarantee good performance [1], [18], [19], especially for the large class of important read-intensive applications [20]–[22].

Achieving non-blocking parallel transactional reads with partial replication is challenging, because different reads within the same transaction may be served in parallel by servers in different DCs. With full replication, a transaction executes entirely within a single DC, and therefore protocols for full replication are not prepared to deal with the situation of reads of the same transaction being served in different DCs. For instance, different sets of transactions may have been committed in different DCs, and naive application of full replication protocols may lead to reads returning inconsistent results.

PaRiS addresses this issue by means of a new causal dependency tracking protocol, that we call Universal Stable Time (UST). In short, UST identifies a snapshot of the dataset that has been installed in *all* DCs. Hence, a transaction in any DC can read from such a snapshot without blocking. In addition to the snapshot defined by UST, PaRiS equips clients with a private cache, in which clients store their own updates that are not yet reflected in the snapshot identified by the UST. The combination of these two techniques, UST and private caches, suffices to implement TCC with non-blocking reads in a partial replication setting. PaRiS computes UST efficiently by means of a periodic, lightweight intra- and inter-DC gossiping protocol. In addition, PaRiS uses only one timestamp to track dependencies and to define transactional snapshots, thus enabling scalability both in terms of number

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00038

of DCs and number of partitions per DC.

Overall, PaRiS achieves low latency, low storage requirements and rich transactional semantics. This combination represents a significant improvement over existing systems that either block read operations to preserve consistency, do not support partial replication, or do not support generic read-write transactions.

The trade-off made by PaRiS –which is provably unavoidable [19]– is to expose to transactions a view of the data that is slightly in the past. We argue that a moderate increase in data staleness is a reasonable price to pay for the performance benefits brought about by PaRiS.

We evaluate PaRiS on an AWS deployment comprising of up to 10 DCs, and with heterogeneous workloads with different degrees of data access locality. We show that PaRiS scales well with the number of DCs and partitions, while being able to handle larger datasets than existing solutions that assume full replication.

Roadmap. The remainder of the paper is organized as follows. Section II presents the system model. Section III describes the design of PaRiS. Section IV describes the protocols and the correctness of PaRiS. Section V reports the results of the evaluation of PaRiS. Section VI discusses related work. Section VII concludes the paper.

II. Definitions and System Model

A. Causal Consistency

A system is causally consistent if it returns values that are consistent with the order defined by the *causality* relationship. Causality is defined as a happens-before relationship between two events [23], [24]. 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 operation b and $a \rightsquigarrow b$, then the client also sees the effects of operation a.

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

B. Transactional Causal Consistency

Semantics. TCC extends CC by means of interactive read-write transactions in which clients can perform multiple reads and writes, each of which can operate on multiple items. TCC enforces two properties.

1. Read from a causally consistent snapshot. A causally consistent snapshot of a transaction T is a view of the datastore that includes the effects of all transactions that causally precede T [5], [8], [10]. For any two items, x and y, if $X \rightsquigarrow Y$ and both X and Y belong to the same causally consistent snapshot, then there is no other X', such that $X \rightsquigarrow X' \rightsquigarrow Y$. Transactional reads from a causally consistent snapshot prevent undesirable anomalies which can arise by issuing consecutive individual read operations [5].

2. Atomic updates. Either all the 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.

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 new value of a key given its current value and the set of updates performed on it [5].

For simplicity, PaRiS resolves write conflicts using the last-writer-wins rule [25] based on the timestamp of the updates. Possible ties are settled by looking at the id of the DC combined with the identifier of the transaction that created the update. PaRiS can be extended to support other conflict resolution mechanisms [5], [6], [8], [26].

C. System model

We consider a distributed key-value store whose dataset is split into N partitions. We assume that each server is assigned a single partition, and we denote by p_x the server responsible for key x. Each partition p_i is replicated at R different DCs, where R is the replication factor. There are M DCs in total where $M > R$, hence, only a subset of the full dataset is present in each DC. A client reads locally whenever possible, otherwise can contact any remote replica of a key.

We assume a multi-master system, i.e., all replicas can update keys for which they are responsible. Updates are replicated asynchronously to remote DCs.

We assume a multi-version data store. 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 data items. Partitions communicate through point-to-point lossless FIFO channels (e.g., TCP sockets).

At the beginning of a session, a client c connects to a partition p in one DC according to some load balancing scheme. This DC is referred to as the local DC. The partition p serves all c's operations. If p does not store a key k targeted by an operation, p transparently forwards the operation to a partition storing k, possibly in a different DC. c does not issue the next operation until it receives the reply to the current one.

Availability. We use the term availability to indicate that a client operation is never blocked in the presence of a network partition [8], [27].

D. APIs

PaRiS's programming interface offers the following operations for interactive read-write transactions:

• $< T_{ID}, S > \leftarrow START - TX()$ **:** starts an interactive transaction T and returns T's transaction identifier T_{ID} and the causally consistent snapshot S visible to T.

• $\langle vals \rangle \leftarrow READ(T_{ID}, k_1, ..., k_n)$: reads in parallel the set of items corresponding to the input set of keys for a transaction identified by T_{ID}.

• $WRITE(T_{ID}, \langle k_1, v_1 \rangle, ..., \langle k_n, v_n \rangle)$: updates a set of keys, given as input, to the corresponding values for a transaction with T_{ID}.

• $COMMIT - TX(T_{ID})$: finalizes the transaction T_{ID} and atomically updates items that have been modified by means of a WRITE operation in the scope of the transaction, if any.

Clients can issue multiple read and write operations that can operate on multiple keys, between the start and the commit of T.

Under the TCC programming model, conflicting updates are resolved rather than forbidden. Therefore, transactions never abort due to conflicts. Although transactions can abort by means 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. Design of PaRiS

PaRiS implements, in a scalable manner, TCC with parallel transactional reads in a partially replicated and sharded system. We first illustrate the challenges involved in doing so in Section III-A. Next, in Section III-B we present how PaRiS overcomes these challenges by a novel dependency tracking protocol and the use of a small client-side cache. Finally, in Section III-C we discuss fault tolerance and availability in PaRiS.

A. Challenges of partial replication

Since TCC must simultaneously guarantee the preservation of causal consistency and the atomicity of multi-item writes, achieving non-blocking reads while maintaining TCC is challenging. In a fully replicated environment, the task of enforcing this behavior is eased by two invariants: i) all remote updates are received by all DCs, and hence every DC receives the dependencies of each update, and ii) all updates of a transaction are performed within the same DC, and hence all the updates of the same transaction can be found in each DC. As example, assume that $X \rightsquigarrow Y$, and that X and Y are the latest versions of their keys, x and y, respectively. Causal consistency dictates that if a client reads x and y in a transaction, then if Y is returned, X has to be returned as well. Furthermore, assume that Z, the last version of key z, has been written by the same transaction as Y. TCC further implies that either both Y and Z are visible to the transaction, or none of them is. In full replication, some sort of communication among partitions in the *same* DC is enough to ensure that Y is visible in the DC only after X, and that Y and Z are atomically visible.

Partial replication, instead, violates the two invariants described above. This leads to a new set of challenges of enforcing consistency and atomicity. First, tracking consistency is harder. In the previous example with keys x and y, X may be replicated from DC_0 to DC_1, and Y from DC_0 to DC_2. Then, assume that a transaction from DC_3 reads x in DC_1 and y in DC_2. The transaction has to ensure that Y is read in DC_2 only if also X is read, concurrently, in DC_1. Similarly, enforcing atomicity is harder. Assume that Z is replicated from DC_0 to DC_1, and Y from DC_0 to DC_2, and that a transaction from DC_3 reads y and z. Then, the transaction in DC_3 has to ensure that either both Y and Z or none of them are read in an atomic fashion.

Addressing these two challenges is made more difficult by the fact that a read operation can target any replica of the target key. Hence, consistency and atomicity have to be preserved despite the fact that different transactions targeting the same keys can hit different replicas of those keys. The complexity of the problem is further exacerbated by the fact that different replicas of a version X may be in different DCs that store different sub-sets of the dependencies of X.

One possible solution to these challenges could be allowing more than one round of client-server communication to perform a single parallel read operation. Servers can return possibly inconsistent versions in the first round(s), and the client can detect and fix these violations by issuing additional read requests [5], [6], [9].

Another possible solution could be blocking a read on a partition until the partition knows that all other involved partitions are serving the read operations from the same causally consistent snapshot of the data store [8], [10], [12].

Clearly, these solutions increase the latency experienced by the transactions and reduce the achievable throughput, because they introduce waiting times or require extra communication.

B. Non-blocking reads in partial replication by PaRiS

PaRiS addresses these challenges by a combination of a novel dependency tracking protocol, called UST, and a client-side cache. UST identifies snapshots of the data store that can be read by transactions without blocking. These snapshots are such that they have been already installed by every DC, so they are slightly in the past. The client-side cache stores the versions written by the client that are not yet reflected in the snapshot determined by UST, allowing clients to observe monotonically increasing snapshots even if UST identifies slightly stale snapshots. We now explain how PaRiS leverages UST and the client-side cache in its transactional protocol.

Transactions in PaRiS. PaRiS identifies key versions and snapshots by means of a scalar timestamp. Upon starting, a transaction is assigned a snapshot timestamp st that, together with the content of the client-side cache, determines the snapshot visible to the transaction. Upon completing, each transaction that writes at least one key is assigned a commit timestamp that reflects causality, determined by means of a two-phase commit (2PC) protocol.

Non-blocking reads in PaRiS. The key idea in PaRiS is to identify a snapshot that has been installed by each DC. We define such snapshot as *stable*. A stable snapshot with timestamp ts contains all versions with a timestamp $\leq ts$, and

indicates that *every* transaction T with a commit timestamp $\leq ts$ has been applied in *every* data center that stores a replica of the keys written by T.

Hence, a transaction can read from a stable snapshot without blocking or running multiple client-server rounds, regardless of the DC in which the individual reads of the transaction are performed.

A coordinator partition is responsible for assigning a stable snapshot to a transaction that starts. Any node can act as the coordinator of any transaction. The coordinator enforces the monotonicity of the snapshots assigned to transactions issued by the same client. To this end, the client piggybacks its last observed snapshot timestamp on the transaction start message to the coordinator.

UST. UST is the new protocol implemented by PaRiS to identify, in a scalable fashion, stable snapshots. Each partition maintains a version vector that indicates the timestamps of the latest applied transactions, both the ones executed by the partition itself and the ones received from remote replicas. Periodically, partitions within the same DC and across DCs exchange, by means of a gossiping protocol, the minimum of the timestamps in their version vectors. The aggregate minimum of the exchanged values identifies a timestamp such that all transactions with lower timestamps have been applied by all partitions in every DC.

UST identifies stable causally consistent snapshots with a single timestamp, regardless of the scale of the system. This enables high scalability and efficiency, by reducing partition-to-partition and client-to-partition communication overhead.

Cache. UST alone cannot enforce causality. In fact, the commit timestamp assigned to a transaction T issued by client c is higher than the stable snapshot assigned to T, which allows the commit timestamps to reflect causality. The commit timestamp of T may be higher than the snapshot assigned by the next transaction issued by c. In that case, such snapshot would not include the modifications performed by c in T, which may lead to violation of the read-your-own-writes property required by causal consistency.

PaRiS overcomes this issue by storing on the client the versions written by the client. Upon receiving a snapshot timestamp st, a client c removes from the cache all versions with timestamp $\leq st$. These versions are already included in the snapshots visible to any future transaction issued by c. When reading key x, c first checks its cache. If a version exists in the cache, that version has to be read by c to enforce the read-your-own-writes property. Else, c issues a read request to a replica. In both cases, the read completes without blocking.

Generating timestamps. As in recent proposals [9], [15], [27], [28], PaRiS uses Hybrid Logical Physical Clocks (HLC) [29] to generate timestamps. An HLC is a logical clock whose value on a partition is the maximum of the local physical clock and the highest timestamp seen by the partition plus one. Like logical clocks, HLCs can be moved forward to match the timestamp of an incoming event, without blocking to wait for the local physical clock to catch up with the timestamp of the incoming event. Like physical clocks, HLCs advance in the absence of events and at approximately the same pace. Hence, HLCs improve the freshness of the snapshot determined by UST over a solution that uses logical clocks, which can advance at very different rates on different partitions.

C. Fault tolerance

Failures (within a DC). PaRiS can tolerate failures of a server by integrating existing solutions for 2PC-based systems, e.g., based on Paxos [30]. Reads remain non-blocking with such mechanisms enabled, because they access a snapshot corresponding to transactions that have *already* been committed.

As in previous systems based on dependency aggregation protocols, the failure of a server blocks the progress of UST, but only as long as a backup has not taken over.

Client failures are transparent to the system. The clients only keep local meta-data and cache data that have already been committed to the data-store. The transaction contexts corresponding to failed clients are cleaned up after a timeout.

Availability (among DCs). PaRiS achieves availability in a DC as long as one replica per partition is reachable by the DC, so that remote items can be read, and the 2PC can be executed. In fact, remote reads can be performed without blocking by any replica, because the snapshot visible to a transaction is already installed in all DCs. In addition, local operations never block.

If a DC partitions from the rest of the system, that DC cannot serve any transaction that involves reading or writing a remote item. The remaining DCs remain available and clients can still issue transactions, but the UST freezes at all DCs, because it is computed as a system-wide minimum. As a result, transactions see increasingly stale snapshots of the data, and the client cache cannot be pruned.

IV. Protocols of PaRiS

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

A. Meta-data

Items. An item d is represented as the tuple $\langle k, v, ut, id_T, sr \rangle$. k and v are the key and value of d, respectively. ut is the timestamp of d which is assigned upon the commit of the transaction that created d and determines the snapshot to which d belongs. id_T is the id of the transaction that created the item version. sr is the id of the DC where the item is created.

Clients. In a client session, a client c maintains the highest stable snapshot timestamp known to c, denoted by ust_c, and the commit time of its last update transaction, noted hwt_c. The client also maintains a private cache WC_c, which stores items written by c for which c does not yet know whether they belong to the stable snapshot. Finally, the client maintains the meta-data and data of the transaction that is currently running: id_c, which is the unique identifier of the transaction, and WS_c and RS_c, which correspond to the transaction's write set and read set, respectively.

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

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

8: **function** READ(χ)
9: $D \leftarrow \emptyset$ ▷ *D represents the set of key-value pairs for the requested keys*
10: **for each** $k \in \chi$ **do**
11: $v \leftarrow$ look up k in WS_c, RS_c, WC_c (in this order) ▷ *The value might already be available in the client*
12: **if** $(v \neq NULL)$ **then** $D \leftarrow D \cup \langle k, v\rangle$
13: **end for**
14: $\chi' \leftarrow \chi \setminus keys(D)$
15: send $\langle$**ReadReq** $\boldsymbol{id_c}, \boldsymbol{\chi'}\rangle$ to p_n^m
16: receive $\langle$**ReadResp** $\boldsymbol{D'}\rangle$ 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** $\boldsymbol{id_c}, \boldsymbol{hwt_c}, \boldsymbol{WS_c}\rangle$ 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 Server p_n^m - transaction coordinator.

1: **upon** receive $\langle$**StartTxReq** $\boldsymbol{ust_c}\rangle$ from c **do**
2: $ust_n^m \leftarrow max\{ust_n^m, ust_c\}$ ▷ *Update universal stable time*
3: $id_T \leftarrow generateUniqueId()$
4: $TX[id_T] \leftarrow ust_n^m$ ▷ *Save TX context*
5: send $\langle$**StartTxResp** $\boldsymbol{id_T}, \boldsymbol{TX[id_T]}\rangle$ ▷ *Assign transaction snapshot*

6: **upon** receive $\langle$**ReadReq** $\boldsymbol{id_T}, \boldsymbol{\chi}\rangle$ from c **do**
7: $ust \leftarrow TX[id_T]$
8: $D \leftarrow \emptyset$
9: $\chi_i \leftarrow \{k \in \chi : partition(k) == i\}$ ▷ *Partitions with $\geq$ 1 key to read*
10: **for** $(i : \chi_i \neq \emptyset)$ **do** ▷ *Done in parallel*
11: $j = getTargetDCForPartition(i)$ ▷ *Returns an id of a DC that replicates partition i*
12: send $\langle$**ReadSliceReq** $\boldsymbol{\chi_i}, \boldsymbol{ust}\rangle$ to p_i^j
13: receive $\langle$**ReadSliceResp** $\boldsymbol{D_i}\rangle$ from p_i^j
14: $D \leftarrow D \cup D_i$
15: **end for**
16: send $\langle$**ReadResp** $\boldsymbol{D}\rangle$ to c

17: **upon** receive $\langle$**CommitReq** $\boldsymbol{id_T}, \boldsymbol{hwt}, \boldsymbol{WS}\rangle$ from c **do**
18: $\langle ust\rangle \leftarrow TX[id_T]$
19: $ht \leftarrow max\{ust, hwt\}$ ▷ *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: $j = getTargetDCForPartition(i)$ ▷ *Returns an id of a DC that replicates partition i*
23: send $\langle$**PrepareReq** $\boldsymbol{id_T}, \boldsymbol{ust}, \boldsymbol{ht}, \boldsymbol{D_i}\rangle$ to p_i^j
24: receive $\langle$**PrepareResp** $\boldsymbol{id_T}, \boldsymbol{pt_i}\rangle$ from p_i^j
25: **end for**
26: ct $\leftarrow max_{i:D_i \neq \emptyset}\{pt_i\}$ ▷ *Max proposed timestamp*
27: **for** $(i : D_i \neq \emptyset)$ **do** ▷ *Done in parallel*
28: $j = getTargetDCForPartition(i)$ ▷ *Returns an id of a DC that replicates partition i*
29: send $\langle$**CommitReq** $\boldsymbol{id_T}, \boldsymbol{ct}\rangle$ to p_i^j
30: **end for**
31: delete TX[id_T] ▷ *Clear transactional context of c*
32: send $\langle$**CommitResp** $\boldsymbol{ct}\rangle$ to c

Server. A server p_n^m is identified by the partition id (n), and the DC id (m). p_n^m stores the replica id (r), where $r \leq R$, the replication factor of partition p_n^m.

Each server has access to a physical clock $Clock_n^m$, that generates monotonically increasing timestamps. Physical clocks are loosely synchronized by a time synchronization protocol such as NTP [31]. The correctness of the protocol does not depend on the synchronization precision. The local clock value on p_n^m is represented by the hybrid logical clock HLC_n^m. p_n^m also maintains two vector clocks VV_n^m and GSV_n^m, that represent vectors of HLCs. VV_n^m, has R entries, one for each replica of partition n. $VV_n^m[i], i \neq r$, indicates the timestamp of the latest update received by p_n^m that comes from the i-th replica of partition n. $VV_n^m[r]$ is the version clock of the server and represents the local snapshot installed by p_n^m. GSV_n^m, or Global Stabilization Vector, has M entries. $GSV_n^m[i] = t$ means that p_n^m is aware that all partitions in DC_m have installed all events generated in the DC_i with timestamp up to t. The server also stores the UST of p_n^m, denoted by ust_n^m. $ust_n^m = t$ indicates that p_n^m is aware that every partition in every DC has installed a snapshot with timestamp at least t.

Finally, following standard practice for systems that perform a 2PC protocol, p_n^m keeps two queues, one with prepared and one with committed transactions. The former, denoted by $Prepared_n^m$, stores transactions for which p_n^m has proposed a commit timestamp and for which p_n^m is waiting for the commit message. The latter, denoted by $Committed_n^m$ stores transactions that have been assigned a commit timestamp and whose modifications are going to be applied to p_n^m.

B. Operations

Algorithm 1 provides pseudo-code for the client protocol, Algorithm 2 and Algorithm 3 for the protocols executed by a server to run a transaction, for the cases in which the server is or is not the transaction coordinator, respectively, and Algorithm 4 for the replication and the UST protocols.

Start. Client c starts a transaction T by picking at random a coordinator partition (denoted p_n^m) in the local DC and sending it a start request with ust_c. p_n^m uses ust_c to update ust_n^m, so that p_n^m can assign to the new transaction a snapshot that is at least as fresh as the one accessed by c in previous transactions. p_n^m uses its updated value of ust_n^m as snapshot for T. p_n^m also generates a unique identifier for T, denoted id_T, and inserts T in a private index of transactions TX. p_n^m replies to c with id_T and the snapshot timestamp ust_n^m.

Upon receiving the reply, c updates ust_c and evicts from the cache any version with timestamp lower than or equal to ust_c. c can prune such versions because the UST protocol ensures that they are included in the snapshot installed by any partition in the system. This means that if, after pruning, there is a version X in the private cache of c, then $X.ct > ust$ and hence the freshest version of x visible to c is X.

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 to the transaction coordinator p_n^m together with the transaction

Algorithm 3 Server p_n^m - transaction cohort.

1: **upon** receive $\langle$**ReadSliceReq** $\boldsymbol{\chi}, \boldsymbol{ust}\rangle$ from p_i^j **do**
2: $ust_n^m \leftarrow max\{ust_n^m, ust\}$ ▷ *Update universal stable time*
3: $D \leftarrow \emptyset$
4: **for** $(k \in \chi)$ **do**
5: $D_{sv} \leftarrow \{d : d.k == k \wedge d.ut \leq ust\}$ ▷ *Universally visible*
6: $D \leftarrow D \cup \{argmax_{d.ut}\{d \in D_{kv}\}\}$ ▷ *Freshest visible vers. of k*
7: **end for**
8: send $\langle$**SliceResp** $\boldsymbol{D}\rangle$ to p_i^j

9: **upon** receive $\langle$**PrepareReq** $\boldsymbol{id_T, ust, ht, D_i}\rangle$ **do** from p_i^j
10: $HLC_n^m \leftarrow max(Clock_n^m, ht + 1, HLC_n^m + 1)$ ▷ *Update HLC*
11: $ust_n^m \leftarrow max\{ust_n^m, ust\}$ ▷ *Update universal stable time*
12: pt $\leftarrow max\{HLC_n^m, ust_n^m\}$ ▷ *Proposed commit time*
13: $Prepared_n^m \leftarrow Prepared_n^m \cup \{id_T, pt, D_i\}$ ▷ *Append to pending list*
14: send $\langle$**PrepareResp** $\boldsymbol{id_T, pt}\rangle$ to p_i^j

15: **upon** receive $\langle$**CommitReq** $\boldsymbol{id_T, ct}\rangle$ **do** from p_i^j
16: $HLC_n^m \leftarrow max(HLC_n^m, ct, Clock_n^m)$ ▷ *Update HLC*
17: $\langle id_T, pt, D\rangle \leftarrow \{\langle i, r, \phi\rangle \in Prepared_n^m : i == id_T\}$
18: $Prepared_n^m \leftarrow Prepared_n^m \setminus \{\langle id_T, pt, D\rangle\}$ ▷ *Remove from pending*
19: $Committed_n^m \leftarrow Committed_n^m \cup \{\langle id_T, ct, D\}$ ▷ *Mark to commit*

Algorithm 4 Server p_n^m - Auxiliary functions.

1: **function** UPDATE($\boldsymbol{k, v, ut, id_T}$)
2: create $d : \langle d.k, d.v, d.ut, id_T, d.sr\rangle \leftarrow \langle k, v, ut, id_T, m\rangle$
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: $\rho_n \leftarrow Replicas(n)$
9: **if** $(Committed_n^m \neq \emptyset)$ **then** ▷ *Commit tx by increasing order of ct*
10: $C \leftarrow \{\langle id, ct, D\rangle\} \in Committed_n^m : ct < ub$
11: **for** $(T \leftarrow \{\langle id, D\rangle\} \in (group\ C\ by\ ct))$ **do**
12: **for** $(\langle id, D\rangle \in T)$ **do**
13: **for** $(\langle k, v\rangle \in D)$ **do** update $(\boldsymbol{k, v, ct, id})$ **end for**
14: **end for**
15: **for** $(j \in \rho_n \wedge j \neq r)$ **do** send $\langle$**Replicate** $\boldsymbol{T, ct}\rangle$ to p_n^j **end for**
16: $Committed_n^m \leftarrow Committed_n^m \setminus T$
17: **end for**
18: $VV_n^m[m] \leftarrow ub$ ▷ *Set version clock*
19: **else**
20: $VV_n^m[m] \leftarrow ub$ ▷ *Set version clock*
21: **for** $(j \in \rho_n \wedge j \neq r)$ **do** send $\langle$**Heartbeat** $\boldsymbol{VV_n^m[m]}\rangle$ to p_n^j **end for** ▷ *Send heartbeat to replicas*
22: **end if**

23: **upon** receive $\langle$**Replicate** $\boldsymbol{T, ct}\rangle$ from p_n^j **do**
24: **for** $(\langle id, D\rangle \in T)$ **do**
25: **for** $(\langle k, v\rangle \in D)$ **do**
26: update $(\boldsymbol{k, v, ct, id})$
27: **end for**
28: **end for**
29: $i \leftarrow GetReplicaIdForDC(j)$
30: $VV_n^m[i] \leftarrow ct$ ▷ *Update version clock of i-th replica of n-th partition*

31: **upon** receive $\langle$**Heartbeat t**$\rangle$ from p_n^j **do**
32: $i \leftarrow GetReplicaIdForDC(j)$
33: $VV_n^m[i] \leftarrow t$ ▷ *Update version clock of i-th replica in n-th partition*

34: **upon** every Δ_G time **do** ▷ *Gather global stable times from other DCs*
35: $GSV_n^m[j] \leftarrow min\{VV_i^m[j]\}, \forall j = 0 \ldots M-1, \forall i = 0 \ldots N-1$

36: **upon** every Δ_U time **do** ▷ *Compute universal stable time*
37: $minGST \leftarrow min\{GSV_i^m[j]\}, \forall j = 0 \ldots M-1, \forall i = 0 \ldots N-1$
38: $ust_n^m \leftarrow max\{minGST, ust_n^m\}$ ▷ *Enforce monotonicity of* ust_n^m

id. p_n^m retrieves the snapshot corresponding to the transaction, and sends to each involved partition the set of keys to be read, in parallel. Because each DC only stores a subset of the full data set, some of the contacted partitions may belong to a remote DC that replicates the partitions where the keys belong. Remote DCs can be chosen depending on geographical proximity or on some load balancing scheme.

Upon receiving a read request, regardless of whether it originates from the local DC or from a remote one, p_n^m first updates its ust_n^m, if it is smaller than the transaction's snapshot (Alg. 3 Line 2). Next, the server returns, for each key, the version with the highest timestamp below the snapshot timestamp (Alg. 3 Lines 4–7). As we shall see shortly, the commit protocol of PaRiS allows concurrent updates on the same key, both within a DC and in different DCs. This is typically the case in TCC systems to avoid costly validation protocols for update transactions [8], [27]. In case multiple versions of a key are assigned the same timestamp, PaRiS totally orders versions by a concatenation of timestamp, transaction id and source data center id, in this order. Once p_n^m has received the reply from all the partitions contacted, p_n^m sends the items to the client, which inserts them in its 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 in WS_c.

Commit. To finalize the transaction, the client sends a commit request to the coordinator with the content of WS_c, the id of the transaction and the commit timestamp of its last update transaction hwt_c, if any. The commit protocol of PaRiS is based on the two-phase commit (2PC) protocol. The coordinator contacts the partitions that store the keys that need to be updated and sends them the corresponding updates and hwt_c. Some of the contacted partitions may belong to a remote DC. Each partition involved first updates its clock to be at least as high as the maximum of the transaction's snapshot timestamp and hwt_c. Then, each partition increases its clock and sends the updated clock value to the coordinator as a commit timestamp. The proposed timestamp reflects causality because it is higher than both the snapshot timestamp and hwt_c. Each partition also inserts the transaction id, the set of keys to be modified on the partition and the proposed timestamp in the queue of pending transactions.

The coordinator picks the maximum among the proposed timestamps, sends it to the partitions involved in the transaction, removes the transaction from the private index of transactions TX and sends the commit timestamp to the client. Upon receiving the commit message, a partition increases its clock to match the commit time, if needed, and moves the transaction from the pending queue to the commit queue, with the new commit timestamp.

Applying and replicating transactions. Periodically, the effects of transactions committed by p_n^m are applied on p_n^m, in increasing commit timestamp order (Alg. 4 Lines 6-21). p_n^m applies the modifications of transactions that have a commit timestamp strictly 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 update operations in the applied transactions to its remote replicas.

If p_n^m does not commit a transaction for a given amount of time, p_n^m updates its local clock, and sends it to its peer

replicas by means of a heartbeat message. This ensures the progress of the UST also in the absence of updates.

Stabilization protocol. Every Δ_G time units, partitions within a data center exchange the minimum of their version vectors to compute the global stable time (GST) of the local data center. Similarly to previous work [8], [10], PaRiS organizes nodes within a DC as a tree to reduce message exchanges. The GST is progressively aggregated from the leaves to the root, and then propagated from the root to all the nodes in the DC. Next, all the roots from each DC exchange their GST values.

Every Δ_U time units, the roots compute the ust_n^m as the aggregate minimum of the received GSTs and propagate it to all the other nodes in the DC.

Garbage collection. Periodically the partitions exchange the oldest snapshot corresponding to an active transaction (p_n^m sends its current stable snapshot timestamp if it has no running transactions). 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 remove all versions older than S_{old}. The same protocol that computes the UST also computes S_{old}.

C. Correctness

We now provide an informal proof sketch that PaRiS provides causal consistency by showing that i) reads observe a causally consistent snapshot and ii) writes are atomic.

Lemma 1. *The snapshot time sn_T of a transaction T is always lower than the commit time of T, $sn_T < T.ct$.*

Proof: Let T be a transaction with snapshot time sn_T and commit time ct. The snapshot time is determined during the start of the transaction (Alg. 2 Line 4). The commit time is calculated in the commit phase of the 2PC protocol, as maximum value of the proposed prepare times of all partitions participants in T (Alg. 2 Line 26). In order to reflect causality when proposing a prepare timestamp, each partition proposes higher timestamp than the snapshot timestamp (Alg. 3 Line 12). Thus, the commit time of a transaction, ct, is always greater than the snapshot time, sn_T. ∎

Proposition 1. *If an update u_2 causally depends on an update u_1, $u_1 \rightsquigarrow u_2$, then $u_1.ut < u_2.ut$.*

Proof: Let c be the client that wrote u_2. There are three cases upon which u_2 can depend on u_1, described in Section II-A: *1*) c committed u_1 in a previous transaction; *2*) c has read u_1, written in a previous transaction and *3*) c has read u_3, and there exists a chain of direct dependencies that lead from u_1 to u_3, i.e. $u_1 \rightsquigarrow ... \rightsquigarrow u_3$ and $u_3 \rightsquigarrow u_2$.

Case 1. When a client commits a transaction, it piggybacks the last update transaction commit time hwt_c, if any, to its commit request for the transaction coordinator (Alg. 1 Line 27) which is, furthermore, piggybacked as ht in its prepare requests to the involved partitions (Alg. 2 Line 23). To reflect causality when proposing a commit timestamp, each partition proposes higher timestamp than both ht and the snapshot timestamp (Alg. 3 Lines 10–14). The coordinator of the transaction chooses the maximum value from all proposed times from the participating partitions (Alg. 2 Line 26) to serve as commit time, ct, for all the updated items in the transaction. The new version of the data item is written in the key-value store with ct as its update time, ut (Alg. 4 Lines 2 and 13). When c commits the transaction that updates u_2, it piggybacks the commit time of the transaction that updated u_1. Hence, from the discussion above it follows that $u_1.ct < u_2.ct$. Because the commit time of a transaction is the update time of all the data item versions updated in the transaction (Alg. 4 Lines 2 and 13), we have $u_1.ut < u_2.ut$.

Case 2: c could have read u_1 either from c's client cache or from the transaction's causally consistent snapshot sn_T.

If c has read u_1 from c's client cache, then c has written u_1 either in a previous transaction in the same thread of execution or in the current one. If c wrote u_1 in the same transaction where u_2 is also written, then it is not possible to have $u_1 \rightsquigarrow u_2$ because all the updates from that transaction will be given the same commit, i.e. update timestamp, indicating that $u_1.ut = u_2.ut$. Thus, u_1 must be written in a previous transaction and from *Case 1* it follows that $u_1.ut < u_2.ut$.

Next, we will consider the case when c read u_1 from the causally consistent snapshot sn_T that contains u_1. When a transaction T is started, the snapshot sn_T is determined by Alg. 2 Line 2, $sn_T = max\{ust_c, ust_n^m\}$. From Alg. 3 Line 5 we have that $u_1.ut \leq ust = sn_T$. From Lemma 1 it follows that $u_1.ut \leq sn_t < u_2.ct = u_2.ut$. Therefore, $u_1.ut < u_2.ut$.

Case 3: If u_2 depends on u_1 because of a transitive dependency out of $c's$ thread-of-execution, it means that there exists a chain of direct dependencies that lead from u_1 to u_2, i.e., $u_1 \rightsquigarrow ... \rightsquigarrow u_3$ and $u_3 \rightsquigarrow u_2$. Each pair in the transitive-chain, belongs to either *Case 1* or *Case 2*. Hence, the proof of *Case 3* comes down to chained application of the correctness arguments from *Case 1* and *Case 2*, proving that each element has an update time lower than its successor's. ∎

Proposition 2. *A partition vector clock $VV_n^m[i] = t$ implies that p_n^m has received all updates from $i-th$ replica with commit time, $ct \leq t$.*

Proof: We need to show that this proposition is valid for both local and remote updates. To prove the former, we show that there are no pending local updates with commit timestamp $ct \leq t$. When p_n^m updates the local replica vector clock entry $VV_n^m[r]$, it finds the minimum prepare timestamp of all transactions that are currently in the prepare phase (Alg 4. Line 6). Because the commit time is calculated as the maximum of all prepare times (Alg 2. Line 26) and the HLC clock is monotonic (Alg. 3 Lines 10 and 16), it is guaranteed that all future transactions will receive a commit time which is greater than or equal to this minimum prepare timestamp. So, when the $VV_n^m[r]$ is set to the minimum of the prepare times of all transactions in the prepare queue minus 1 (Alg. 4 Line 6), p_n^m has already received all updates for the snapshot $VV_n^m[i] = t$.

To show the latter, we use proof by contradiction. Assume

there is a remote update u from $i-th$ replica such that $u.ct < t$, and p_n^m has not received u. By Alg. 4 Line 30, the partition would have received an update u' such that $u'.ct = t$. The updates are sent in the order of their commit timestamps (Alg. 4 Lines 9–16). Hence, if $u'.ct > u.ct$ the p_n^m could not have received another update u' before u. Therefore, $u.ct > t$, implying that $u.ct \not< t$, leading to the contradiction. ■

Proposition 3. *Snapshots in PaRiS are causal.*

Proof: 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 Line 2). Commit timestamps reflect causality (Alg. 2 Line 26), and UST tracks a lower bound on the snapshot installed by every partition in all DCs (Alg. 4 Lines 36-38). If X is within the snapshot of a transaction, so are its dependencies (Proposition 1). On top of the snapshot provided by the coordinator, c also can read the writes, that are not yet included in the snapshot, from the cache. These writes cannot depend on items created by other clients that are outside the snapshot visible to c. ■

Proposition 4. *Writes are atomic.*

Proof: Although updates are made visible independently on each partition p_n^m involved in the commit phase, either all updates are made visible or none of them are. All updates from a transaction belong to the same snapshot, because they all receive the same commit timestamps (Alg. 2 Line 29). The updates are being installed in the order of their commit timestamps (Alg. 4 Lines 9–16). The visibility of the item versions is determined by the transaction snapshot (Alg. 3 Line 5), which is based on the value of p_n^m's universal stable time ust_n^m. ust_n^m is computed by the UST protocol as the aggregate minimum of the version vectors entries of all partitions of all data centers (Alg. 4 Lines 34–38). ■

PaRiS implements TCC, as every transaction reads from a causally consistent snapshot (Proposition 3) that includes all effects (Proposition 4) of its causally dependent transactions.

V. Evaluation

Competitor systems. To assess the advantages of non-blocking reads, we compare PaRiS against a blocking protocol, which we call Blocking Partial Replication, or BPR, inspired by [11]. In BPR, the snapshot of a transaction T of client c is determined as the maximum of the highest causally consistent snapshot seen by c and the clock value of the transaction coordinator. BPR uses one timestamp to encode transactional snapshots, so we can compare fairly the resource efficiency of PaRiS versus the one of BPR. BPR also favors the freshness of the snapshots that are visible to transactions. BPR, however, implies having blocking transactional reads, because the server must ensure that the returned version belongs to a causally consistent snapshot. To this end, a partition blocks a read operation with snapshot timestamp t until it has applied all local and remote transactions with timestamp up to t. By being blocking and choosing the freshest possible snapshot, BPR does not need a stabilization protocol.

	OR	NV	IR	MU	SY
Oregon		85.72	144.52	238.11	157.13
N. Virginia	88.28		80.4	190.13	207.98
Ireland	139.32	76.47		137.53	279.75
Mumbai	253.07	190.42	132.74		268.66
Sydney	148.69	224	291.34	307.53	

Fig. 1: Round-trip latencies in ms between AWS DCs used for the default experimental configuration.

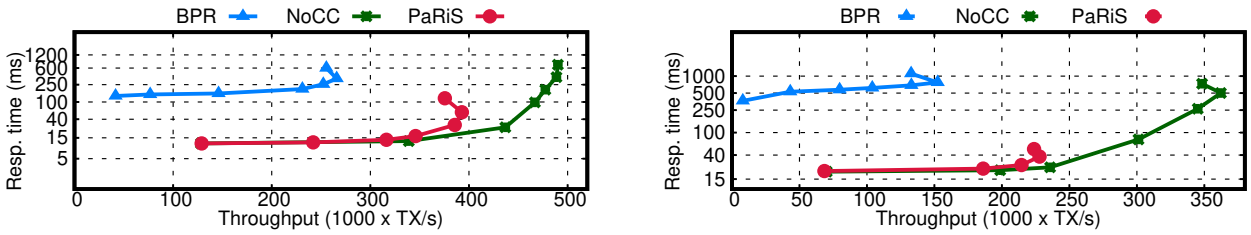
We do not compare to the second alternative possible design, discussed in Section III, because to our best knowledge, a two-round protocol designed for a multi-master TCC partially replicated system currently does not exist. The two existing two-round protocols, [6] [9], are both designed for a fully replicated system, and it is very difficult to adapt them to partial replication. The replication scheme of [6] is based on dependency checking, and [9] has a master-slave system model which, additionally, can abort even read-only transactions. None of the systems supports partial replication and general-purpose read-write transactions.

To measure the overhead of implemening causal consistency in PaRiS, we additionally compare PaRiS to an implementation that does not guarantee causal consistency, which we refer to as NoCC. NoCC is implemented in the same codebase as PaRiS, with several key differences: *i)* it does not read from a causally consistent snapshot; *ii)* it does not have the start phase of transactions (one round trip less than PaRiS); *iii)* it does not traverse an item version chain because it always returns the latest received item version; *iv)* it does not run a stabilization protocol and *v)* for a fair comparison with PaRiS, NoCC provides transaction atomicity by a simplified variation of the 2PC in the commit phase where there are no adjustments made to the HLC clock (each participant in the prepare phase just proposes its clock value + 1).

Comparison with fully replicated systems is beyond the scope of the paper, but the trade-offs are rather intuitive. To achieve non-blocking reads, PaRiS uses a snapshot that is installed in every DC, complemented with the client-side cache. A fully replicated system, besides the client-side cache, only needs a snapshot installed in all partitions in the local DC to avoid blocking. A fully replicated system has lower latency on reads and writes, because they are local, and lower visibility latency, because it needs to track only the lower bound on item versions installed in the local DC. However, a partial replication system reduces the replication costs and storage requirements.

A. Experimental environment

Platform. We consider a geo-replicated setting deployed across up to 10 replication sites on Amazon EC2 (North Virginia, Oregon, Ireland, Mumbai, Sydney, Canada, Seul, Frankfurt, Singapore and Ohio). When using 3 DCs, we use North Virginia, Oregon and Ireland. When using 5 DCs, we use the previously mentioned 3 DCs plus Mumbai and Sydney and the round-trip latencies among them are shown



(a) Throughput vs average TX latency (95:5 r:w ratio).

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

Fig. 2: Performance of PaRiS, BPR and NoCC (y-axis in logarithmic scale) with 95:5 (a) and 50:50 (b) r:w ratios, 4 partitions involved per transaction (5 DCs, 45 partitions, replication factor is 2, 18 machines per DC). PaRiS outperforms BPR for both read-heavy and write-heavy workloads. As long as the systems are not overloaded, the throughput of PaRiS is almost the same as NoCC, whereas NoCC achieves higher max throughput under high load. The last points in each line represent overload conditions, that lead throughput to decrease and latency to increase.

on Figure 1. In each DC we use up to 18 servers (c5.xlarge instances with 4 VCPUs and 8 GB of RAM). The replication factor is 2. We choose this value because it allows us to use 3 as minimum number of DCs in our experiment and use partial replication. We use hash-based partitioning scheme, where each key is deterministically assigned to one partition by a hash function.

We spawn one client process per partition in each DC. Clients are co-located with the server partition they use as a transaction coordinator. The clients issue requests in a closed loop. To generate different load conditions, we spawn different number of threads per client process. Depending on the type of the workload or the protocol, a different number of threads is needed to saturate the target system. Increasing the number of threads past that point leads the systems to deliver lower throughput despite serving a higher number of client threads. Each "dot" in the curve plots we report corresponds to a different number of active threads per client process.

Implementation. We implement PaRiS, BPR and NoCC in the same C++ codebase. All the protocols implement the last-writer-wins rule for convergence. We use Google Protobufs for communication, and NTP to synchronize physical clocks. For PaRiS, the stabilization protocols run every 5 milliseconds.

Workloads. We use workloads with 50:50 and 95:5 r:w ratios that correspond to the write-heavy (A) and read-heavy (B) profiles of YCSB workloads [32]. These are standard workloads also used to benchmark other TCC systems [7]–[9], [27]. To accommodate the transactional nature of PaRiS, we extend YCSB by wrapping the reads and writes into a transaction, with two additional operations for the start and for the commit of a transaction. Transactions generate both workloads by executing 19 reads and 1 write (95:5), and 10 reads and 10 writes (50:50). Hence, in each workload a transaction executes 20 operations per transaction, excluding the start and commit operation. A transaction first executes all the reads in parallel, and then all the writes in parallel.

A transaction can target only partitions in the local DC, or can touch random partitions in remote DCs. In the first case, we say that a transaction is "local-DC"; else, we say it is "multi-DC". When accessing a remote partition, a client can choose among two replicas. We assign to every client in a DC the same preferred remote replica for each partition. We vary the preferred replica in the DCs using a round-robin assignment, to balance the load. To evaluate the effect of the partial replication, we use workloads with 100:0, 95:5, 90:10 and 50:50 local-DC:multi-DC ratios.

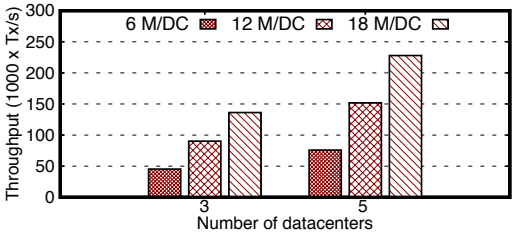
The default workload we consider uses a 95:5 r:w ratio, a 95:5 local-DC:multi-DC ratio and runs transactions that involve 4 partitions on a platform deployed over 90 machines spread over 5 DCs. The default deployment has 45 partitions that are replicated with replication factor 2. Hence, each DC has a total of 18 machines.

We also consider variations of this workload in which we change the value of one parameter 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 [33]–[35]. We use small items (8 bytes), which are prevalent in many production workloads [33], [34].

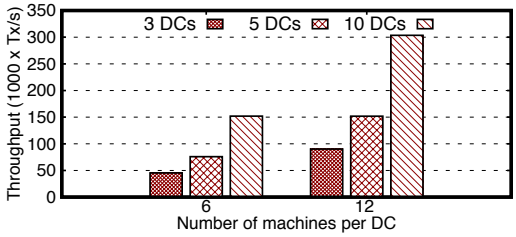
B. Latency and throughput

Blocking vs. non-blocking. Figures 2a and 2b report the average transaction latency vs. throughput achieved by PaRiS and BPR with the 95:5 (the default) and with the 50:50 r:w ratios. In the read-heavy case, PaRiS achieves up to 5.91x lower response times and up to 1.47x higher throughput than BPR. PaRiS also achieves up to 20.56x lower response times and up to 1.46x higher throughput than BPR for the write-heavy workload. PaRiS achieves lower latencies because it never has to wait for a snapshot to be installed. PaRiS achieves higher throughput because it does not incur any overhead to block/unblock read requests. Because BPR is a blocking protocol, it needs a higher number of concurrent client threads to fully utilize the processing power left idle by blocked reads, creating more contention on the physical resources and more synchronization overhead to block and unblock reads, which ultimately leads to lower throughput.

Blocking time. The average blocking time of the read phase of a transaction in BPR is 29 ms for the top throughput in the

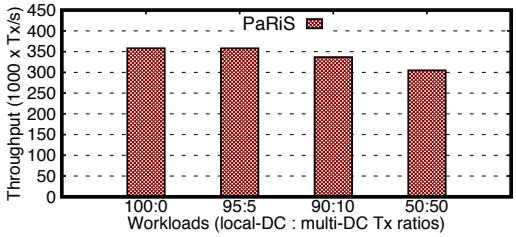


(a) Throughput when varying the number of machines per DC.

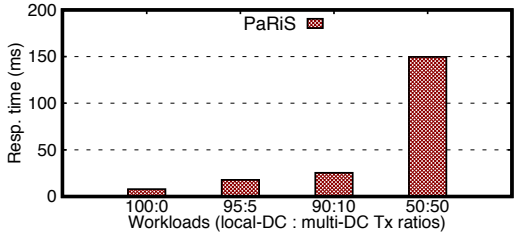


(b) Throughput when varying the number of DCs.

Fig. 3: Throughput achieved by PaRiS when increasing the number of machines per DC (a) and DCs (b). PaRiS achieves good scalability both when increasing the number of machines per DCs and DCs.



(a) Throughput when varying the locality of the transactions.



(b) Latency when varying the locality of the transactions.

Fig. 4: Throughput (a) and latency (b) achieved by PaRiS when varying the locality of the transactions with 100:0, 95:5, 90:10 and 50:50 local-DC:multi-DC ratio for the default workload.

read-dominated workload (Figure 2a) and 41 ms for the top throughput in the write-heavy workload (Figure 2b).

Causal vs. non-causal. Figures 2a and 2b show that PaRiS achieves the same latency as NoCC for both read-heavy and write-heavy workloads, as long the systems are not overloaded. NoCC achieves up to 1.24x higher maximum throughput for the read-heavy workload and up to 1.59x higher maximum throughput for the write-heavy workload. This performance difference is due to the overhead incurred by PaRiS to implement causal consistency.

C. Scalability

Varying the number of machines per DCs. Figure 3a reports the throughput achieved by PaRiS when using 6, 12 and 18 machines/DC. We consider two geo-replicated deployments that use 3 and 5 DCs. In both cases, PaRiS achieves the ideal improvement of 3x when scaling from 6 to 18 machines/DC. This result showcases the ability of PaRiS to scale horizontally regardless of the number of DCs on which it is deployed.

Varying the number DCs. Figure 3b reports the throughput achieved by PaRiS when deployed on 3, 5 and 10 DCs. We consider two cases corresponding to 6 and 12 machines/DC. In both cases PaRiS achieves the ideal improvement of 3.33x, when scaling from 3 to 10 DCs. This result shows that PaRiS scales well to higher numbers of DCs for different sizes of the platform within each DC.

D. Varying data access locality

Figure 4a reports the maximum throughput achieved by PaRiS when varying the locality ratio (local-DC:multi-DC) of transactions from 100:0 to 50:50, for the default workload. Figure 4b shows the average transaction latency corresponding to the throughput values reported in Figure 4a. Performance deteriorates as the percentage of local accesses decreases. The maximum achievable throughput drops slightly, from 350 to 300 KTx/sec. As expected, latency is more heavily penalized, increasing from 8 to 150 ms. The number of threads needed to saturate the system increases as the locality decreases (from 32 to 512 in this case), because requests spend more time traveling between DCs, explaining the small drop in maximum throughput of only 16% compared to the order-of-magnitude increase in latency.

As any partially replicated system, PaRiS targets workloads with high locality in the data access pattern. In case of limited locality, the performance penalty incurred by PaRiS, and partial replication in general, is the inevitable price to pay to enable higher storage capacity.

E. Data staleness

We measure the staleness of the data returned by PaRiS by measuring the *visibility latency* of updates. 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 received in DC_i. We factor out the latency of the update propagation from the DC where the item was created to DC_i, because this communication delay is always present in any replication protocol.

Figure 5 shows the CDF of the update visibility latency achieved by PaRiS and BPR with 5 DCs and the default workload. Each point in the CDF is computed as the average across the corresponding values on each node.

The update visibility time in PaRiS is higher than in BPR. That is to be expected because UST identifies a lower bound

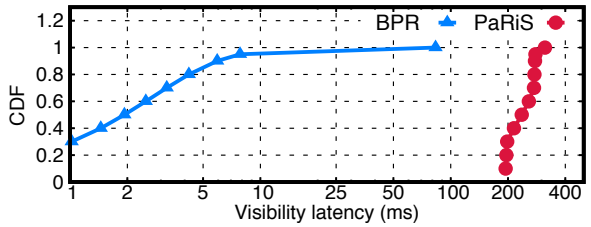


Fig. 5: PaRiS has higher update visibility latency than BPR (logarithmic scale), for the default workload.

of the update time of transactions applied in the whole cluster. The worst-case difference between the visibility latency in PaRiS and BPR is around 200 ms. This aligns with the communication latency between the two farthest DCs, Sydney and Mumbai, as shown on Figure 1. This latency is incurred by these two DCs to exchange their knowledge about versions that are known to have been installed in other DCs. BPR trades data freshness for performance, because it exposes more recent snapshots of the data at the cost of blocking reads, therefore achieving much lower performance than PaRiS.

F. Resource efficiency

Figure 6 shows the percentage of the total data exchanged in PaRiS to run UST and to replicate updates, when varying the locality ratio (local-DC:multi-DC) of transactions from 100:0 to 50:50 for the default workload. For the stabilization protocol, we measured the total amount of data exchanged within each DC (shown as INTRA-DC) and between DCs (shown as INTER-DC). As shown on Figure 6, the major part of the total amount of exchanged data (64% - 72%) is sent for the purpose of replication, 26% -34% for the intra-DC part of the UST protocol and, finally, only 2% for the inter-DC part of the UST protocol. UST's stabilization messages carry only a single timestamp, regardless of the number of partitions and DCs in the system, which contributes to the high resource efficiency in PaRiS.

VI. Related work

Table I classifies existing CC systems according to the transactional model they expose, the capability of achieving non-blocking parallel reads, support for partial replication, and meta-data requirements. The table focuses on systems that target the system model described in Section II-C.

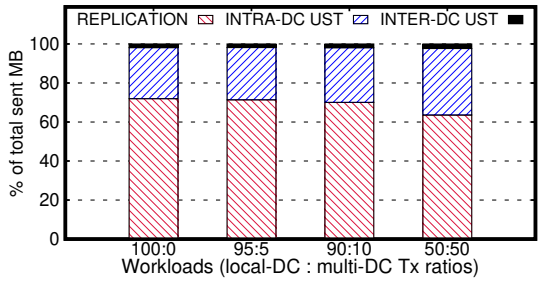


Fig. 6: The overhead of UST compared to replication.

System	Txs	Nonbl. reads	Partial rep.	Meta-data
COPS [5]	ROT	✓	✗	O|deps|
Eiger [6]	ROT/WOT	✓	✗	O|deps|
ChainReaction [12]	ROT	✗	✗	M
Orbe [11]	ROT	✗	✗	1 ts
Gentlerain [10]	ROT	✗	✗	1 ts
POCC [13]	ROT	✗	✗	M
COPS-SNOW [18]	ROT	✓	✗	O|deps|
OCCULT [9]	Generic	✗	✗	O(M)
Cure [8]	Generic	✗	✗	M
Wren [27]	Generic	✓	✗	2 ts
AV [19]	Generic	✓	✗	M
Xiang, Vaidya [36]	✗	✗	✓	1 ts
Contrarian [14]	ROT	✓	✗	M
C^3 [37]	✗	✓	✓	M
Saturn [38]	✗	✓	✓	1 ts
Karma [39]	ROT	✓	✓	O|deps|
CausalSpartan [15]	✗	✓	✗	M
Bolt-on CC [40]	✗	✓	✗	M
EunomiaKV [28]	✗	✓	✗	M
PaRiS (this work)	Generic	✓	✓	1 ts

TABLE I: Taxonomy of the main CC systems. M is the number of DCs. ts stands for timestamp. For systems that do not support transactions, the *non-blocking read* property refers to single-item reads. PaRiS is the only system that supports partial replication with generic transactions, non-blocking parallel reads, and constant meta-data to track dependencies.

The vast majority of the systems assume full replication and provide none or restricted transactional capabilities. This class of systems includes COPS [5], Eiger [6], ChainReaction [12], Orbe [11], GentleRain [10], Bolt-on CC [40], Contrarian [14], POCC [13], CausalSpartan [15], COPS-SNOW [18] and EunomiaKV [28]. All these systems implement one-shot read-only transactions, and only Eiger additionally supports one-shot write-only transactions.

A few systems support partial replication, i.e., Saturn [38], C^3 [37], Karma [39] and the one by Xiang and Vaidya [36]. These systems, however, implement only single-object read and write operations. Among them, only Karma discusses extensions to support read-only transactions by using an approach similar to Eiger's.

To the best of our knowledge, only four systems implement TCC. Among these, OCCULT[1] [9] and Cure [8] can block reads on a node waiting for a snapshot to be installed on such node. Wren [27] and AV [19] avoid blocking by identifying stable snapshots in a way that is similar to PaRiS. All these systems, however, target full replication.

Other relevant systems include TARDiS [41], GSP [42], SwiftCloud [7], Lazy Replication [43], ISIS [44] and Bayou [45]. These systems do not implement sharding, and hence neither partial replication. Many protocols have also been proposed to implement causally consistent distributed shared memories, e.g., [46]–[48]. These protocols do not support transactions and require more meta-data than PaRiS.

PaRiS is also related to systems that implement stronger consistency levels and support partial replication, such as Jessy [49], P-Store [50], STR [51]. On the one hand, these

[1]OCCULT may retry read operations multiple times, instead of blocking the read. Retrying has the same effect on latency of blocking the read until the correct version to read is available on the sever that processes the operation.

systems allow fewer anomalies than what is allowed by TCC [8] and provide fresher data to the clients. On the other hand, they incur higher synchronization costs to determine the outcome of transactions. PaRiS targets applications that can tolerate weaker consistency and some degree of data staleness, e.g., social networks, and offers them low latency, scalability and high storage capacity.

VII. Conclusion

We present PaRiS, the first system that implements TCC in a partially replicated system and achieves non-blocking read operations. PaRiS implements a novel dependency tracking protocol, called UST, which requires only one timestamp to track dependencies. UST identifies a snapshot of the data that is available at *every* DC, thereby enabling non-blocking reads regardless of the DC in which the read takes place.

We evaluate PaRiS on a data platform replicated on up to 10 DCs. PaRiS scales well and achieves lower latency than the blocking alternative, while being able to handle larger datasets than existing solutions that assume full replication.

Acknowledgments

We thank the anonymous reviewers for their valuable suggestions and helpful comments. This research has been supported by The Swiss National Science Foundation through Grant No. 166306 and by an EcoCloud postdoctoral research fellowship.

References

[1] J. C. Corbett, J. Dean, M. Epstein, and et al., "Spanner: Google's Globally Distributed Database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[2] 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*, ser. WWW '17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 263–272. [Online]. Available: https://doi.org/10.1145/3038912.3052603

[3] 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.

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

[5] 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," 2011.

[6] ——, "Stronger Semantics for Low-latency Geo-replicated Storage," in *Proc. of NSDI*, 2013.

[7] 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.

[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] 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.

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

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

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

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

[14] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel, "Causal Consistency and Latency Optimality: Friend or Foe?" *Proc. VLDB Endow.*, vol. 11, no. 11, Jul. 2018.

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

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

[17] 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.

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

[19] A. Z. Tomsic, M. Bravo, and M. Shapiro, "Distributed Transactional Reads: The Strong, the Quick, the Fresh & the Impossible," in *Proceedings of the 19th International Middleware Conference*, ser. Middleware '18, 2018.

[20] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia Workload Analysis for Decentralized Hosting," *Comput. Netw.*, vol. 53, no. 11, Jul. 2009.

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

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

[23] 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.

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

[25] 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.

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

[27] K. Spirovska, D. Didona, and W. Zwaenepoel, "Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store," in *Proceedings of DSN*, 2018.

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

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

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

[31] NTP, "The Network Time Protocol," http://www.ntp.org, 2017.

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

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

[34] 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.

[35] 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.

[36] Z. Xiang and N. H. Vaidya, "Global Stabilization for Causally Consistent Partial Replication," *CoRR*, vol. abs/1803.05575, 2018. [Online]. Available: http://arxiv.org/abs/1803.05575

[37] P. Fouto, J. Leito, and N. Preguia, "Practical and Fast Causal Consistent Partial Geo-Replication," in *Proceedings of NCA*, 2018.

[38] M. Bravo, L. Rodrigues, and P. Van Roy, "Saturn: A Distributed Metadata Service for Causal Consistency," in *Proc. of EuroSys*, 2017.

[39] T. Mahmood, S. P. Narayanan, S. Rao, T. N. Vijaykumar, and M. Thottethodi, "Karma: Cost-effective Geo-replicated Cloud Storage with Dynamic Enforcement of Causal Consistency," *IEEE Transactions on Cloud Computing*, 2018.

[40] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on Causal Consistency," in *Proc. of SIGMOD*, 2013.
[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] 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.
[44] 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. [Online]. Available: http://doi.acm.org/10.1145/128738.128742
[45] 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*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 172–182. [Online]. Available: http://doi.acm.org/10.1145/224056.224070
[46] R. Baldoni, A. Milani, and S. Tucci Piergiovanni, "Optimal propagation-based protocols implementing causal memories," *Distributed Computing*, vol. 18, no. 6, pp. 461–474, 2006.
[47] Z. Xiang and N. H. Vaidya, "Brief Announcement: Partially Replicated Causally Consistent Shared Memory," in *Proceedings of PODC*, 2018, pp. 273–275.
[48] T. Hsu and A. D. Kshemkalyani, "Value the Recent Past: Approximate Causal Consistency for Partially Replicated Systems," *IEEE TPDS*, vol. 29, no. 1, 2018.
[49] 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.
[50] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine Partial Replication in Wide Area Networks," in *Proceedings of SRDS*, 2014.
[51] Z. Li, P. V. Roy, and P. Romano, "Transparent Speculation in Geo-replicated Transactional Data Stores," in *Proceedings of HPDC*, 2018.