UNISTORE: A fault-tolerant marriage of causal and strong consistency

Manuel Bravo Alexey Gotsman Borja de Régil
IMDEA Software Institute

Hengfeng Wei *
Nanjing University

Abstract

Modern online services rely on data stores that replicate their data across geographically distributed data centers. Providing strong consistency in such data stores results in high latencies and makes the system vulnerable to network partitions. The alternative of relaxing consistency violates crucial correctness properties. A compromise is to allow multiple consistency levels to coexist in the data store. In this paper we present UNISTORE, the first fault-tolerant and scalable data store that combines causal and strong consistency. The key challenge we address in UNISTORE is to maintain liveness despite data center failures: this could be compromised if a strong transaction takes a dependency on a causal transaction that is later lost because of a failure. UNISTORE ensures that such situations do not arise while paying the cost of durability for causal transactions only when necessary. We evaluate UNISTORE on Amazon EC2 using both microbenchmarks and a sample application. Our results show that UNISTORE effectively and scalably combines causal and strong consistency.

1 Introduction

Many of today's Internet services rely on geo-distributed data stores, which replicate data in different geographical locations. This improves user experience by allowing accesses to the closest site and ensures disaster-tolerance. However, geo-distribution also makes it more challenging to keep the data consistent. The classical approach is to make replication transparent to clients by providing strong consistency models, such as linearizability [32] or serializability [69]. The downside is that this approach requires synchronization between data centers in the critical path. This significantly increases latency [1] and makes the system unavailable during network partitionings [27]. Thus, even though several commercial geo-distributed systems follow this approach [17, 19, 25, 62, 70], the associated cost has prevented it from being adopted more widely.

An alternative approach is to relax synchronization: the data store executes an operation at a single data center, without any communication with others, and propagates updates to other data centers in the background [20, 65]. This minimizes the latency and makes the system *highly available*, i.e., operational even during network partitionings. But on the downside, the systems following this approach provide weaker consistency models: e.g., eventual consistency [65,68] or *causal consistency* [2]. The latter is particularly appealing: it guarantees that clients see updates in an order that respects the potential causality between them. For example, assume that in a banking application Alice deposits $100 into Bob's account ($u_1$) and then posts a notification about it into Bob's inbox (u_2). Under causal consistency, if Bob sees the notification (u_3), and then checks his account balance (u_4), he will see the deposit. This is not guaranteed under eventual consistency, which does not respect causality relationships, such as those between u_1 and u_2. In some settings, causal consistency has been shown to be the strongest model that allows availability during network partitionings [7, 44]. It has been a subject of active research in recent years, with scalable implementations [3, 42, 47] and some industrial deployments [54, 66].

However, even causal consistency is often too weak to preserve critical application invariants. For example, consider a banking application that disallows overdrafts and thus maintains an invariant that an account balance is always non-negative. Assume that the balance of an account stored at two replicas is 100, and clients concurrently issue two `withdraw`(100) operations (u_5 and u_6) at different replicas. Since causal consistency executes operations without synchronization, both withdrawals will succeed, and once the replicas exchange the updates, the balance will go negative. To ensure integrity invariants in examples such as this, the programmer has to introduce synchronization between replicas, and, since synchronization is expensive, it pays off to do this sparingly. To this end, several research [9,40,41,64] and commercial [5, 6, 28, 48, 57] data stores allow the programmer to choose whether to execute a particular operation under weak or strong consistency. For example, to preserve the integrity

*Also with the State Key Laboratory for Novel Software Technology, Software Institute.

invariant in our banking application, only withdrawals need to use strong consistency, and hence, synchronize; deposits may use weaker consistency and proceed without synchronization.

Given the benefits of causal consistency, it is particularly appealing to marry it with strong consistency in a geo-distributed data store. But like real-life marriages, to be successful this one needs to hold together both in good times and in bad – when data centers fail due to catastrophic events or power outages. Unfortunately, none of the existing data stores meant for geo-replication combine causal and strong consistency while providing such fault tolerance [9, 40, 41]. In this paper we present UNISTORE – the first fault-tolerant and scalable data store that combines causal and strong consistency. More precisely, UNISTORE implements a transactional variant of *Partial Order-Restrictions consistency (PoR consistency)* [30,41]. This guarantees transactional causal consistency by default [3] and allows the programmer to additionally specify which pairs of transactions *conflict*, i.e., have to synchronize. For instance, to preserve the integrity invariant in our previous example, the programmer should declare that withdrawals from the same account conflict. Then one of the withdrawals u_5 and u_6 will observe the other and will fail.

The key challenge we have to address in UNISTORE is to maintain liveness despite data center failures. Just adding a Paxos-based commit protocol for strong transactions [18, 19, 34] to an existing causally consistent protocol does not yield a fault-tolerant data store. In such a data store, a committed strong transaction t_2 may never become visible to clients if a causal transaction t_1 on which it depends is lost due to a failure of its origin data center. This compromises the liveness of the system, because no transaction t_3 conflicting with t_2 can commit from now on: according to the PoR model, one of the transactions t_2 and t_3 has to observe the other, but t_2 will never be visible and t_2 did not observe t_3.

UNISTORE addresses this problem by ensuring that, before a strong transaction commits, all its causal dependencies are *uniform*, i.e., will eventually become visible at all correct data centers. This adapts the classical notion of uniformity in distributed computing to causal consistency [15]. UNISTORE does so without defeating the benefits of causal consistency. Causal transactions remain highly available at the cost of increasing the latency of strong transactions: a strong transaction may have to wait for some of its dependencies to become uniform before committing. To minimize this cost, UNISTORE executes causal transactions on a snapshot that is slightly in the past, such that a strong transaction will mostly depend on causal transactions that are already uniform before committing. Furthermore, UNISTORE reuses the mechanism for tracking uniformity to let clients make causal transaction durable on demand and to enable consistent client migration.

In addition to being fault tolerant, UNISTORE scales horizontally, i.e., with the number of machines in each data center; this also goes beyond previous proposals [9, 40, 41]. To this end, UNISTORE builds on Cure [3] – a scalable implementation of transactional causal consistency. Our protocol extends Cure with a novel mechanism that distributes the task of tracking the set of uniform transactions among the machines of a data center. We also add the ability for data centers to forward transactions they receive from others, so that a transaction can propagate through the system even if its origin data center fails. Finally, we carefully integrate an existing fault-tolerant atomic commit for strong transactions [18] into the protocol for causal consistency.

We have rigorously proved the correctness of the UNISTORE protocol (§7 and §D). We have also evaluated it on Amazon EC2 using both microbenchmarks and a more realistic RUBiS benchmark. Our evaluation demonstrates that UNISTORE scalably combines causal and strong transactions, with the former not affecting the performance of the latter. Under the RUBiS mix workload, causal transactions exhibit a low latency (1.2ms on average), and the overall average latency is 3.7× lower than that of a strongly consistent system.

2 System Model

We consider a geo-distributed system consisting of a set of data centers $\mathcal{D} = \{1, \dots, D\}$ that manage a large set of data items. A data item is uniquely identified by its *key*. For scalability, the key space is split into a set of logical partitions $\mathcal{P} = \{1, \dots, N\}$. Each data center stores replicas of all partitions, scattered among its servers. We let p_d^m be the replica of partition m at data center d, and we refer to replicas of the same partition as *sibling* replicas. As is standard, we assume that $D = 2f + 1$ and at most f data centers may fail. We call a data center that does not fail *correct*. If a data center fails, all partition replicas it stores become unavailable. For simplicity, we do not consider the failures of individual replicas within a data center: these can be masked using standard state-machine replication protocols executing within a data center [37, 55].

Replicas have physical clocks, which are loosely synchronized by a protocol such as NTP. The correctness of UNISTORE does not depend on the precision of clock synchronization, but large drifts may negatively impact its performance. Any two replicas are connected by a reliable FIFO channel, so that messages between correct data centers are guaranteed to be delivered. As is standard, to implement strong consistency we require the network to be *eventually synchronous*, so that message delays between sibling replicas in correct data centers are eventually bounded by some constant [23].

3 Consistency Model

A client interacts with UNISTORE by executing a stream of *transactions* at the data center it is connected to. A transaction consists of a sequence of operations, each on a single data item, and can be *interactive*: the data items it accesses are not known a priori. A transaction that modifies at least one data item is an *update* transaction; otherwise it is *read-only*.

A *consistency model* defines a contract between the data store and its clients that specifies which values the data store is allowed to return in response to client operations. UNISTORE implements a transactional variant of *Partial Order-Restrictions consistency (PoR consistency)* [30,41], which we now define informally; we give a formal definition in §B. The PoR model enables the programmer to classify transactions as either *causal* or *strong*. Causal transactions satisfy transactional *causal consistency*, which guarantees that clients see transactions in an order that respects the potential causality between them [2,3]. However, clients can observe causally independent transactions in arbitrary order. Strong transactions give the programmer more control over their visibility. To this end, the programmer provides a symmetric *conflict relation* $\bowtie$ on operations that is lifted to strong transactions as follows: two transactions conflict if they perform conflicting operations on the same data item. Then the PoR model guarantees that, out of two conflicting strong transactions, one has to observe the other.

More precisely, a transaction t_1 precedes a transaction t_2 in the *session order* if they are executed by the same client and t_1 is executed before t_2. A set of transactions T committed by the data store satisfies PoR consistency if there exists a *causal order* relation $\prec$ on T such that the following properties hold:

Causality Preservation. The relation $\prec$ is transitive, irreflexive, and includes the session order.

Return Value Consistency. Consider an operation u on a data item k in a transaction $t \in T$. The return value of u can be computed from the state of k obtained as follows: first execute all operations on k by transactions preceding t in $\prec$ in an order consistent with $\prec$; then execute all operations on k that precede u in t.

Conflict Ordering. For any distinct strong transactions $t_1, t_2 \in T$, if $t_1 \bowtie t_2$, then either $t_1 \prec t_2$ or $t_2 \prec t_1$.

Eventual Visibility. A transaction $t \in T$ that is either strong or originates at a correct data center eventually becomes visible at all correct data centers: from some point on, t precedes in $\prec$ all transactions issued at correct data centers.

If all transactions are causal, then the above definition specializes to transactional causal consistency [3, 16]. If all transactions are strong and all pairs of operations conflict, then we obtain (non-strict) serializability.

When $t_1 \prec t_2$, we say that t_1 is a *causal dependency* of t_2. Return Value Consistency ensures that all operations in a transaction t execute on a snapshot consisting of its causal dependencies (as well as prior operations by t). Transactions are atomic, so that either all of their operations are included into the snapshot or none at all. The transitivity of $\prec$, mandated by Causality Preservation, ensures that the snapshot a transaction executes on is *causally consistent*: if a transaction t_1 is included into the snapshot, then so is any other transaction t_2 on which t_1 depends (i.e., $t_2 \prec t_1$). The inclusion of the session order into $\prec$, also mandated by Causality Preservation, ensures session guarantees such as *read your writes* [63]. The consistency model disallows the causality violation anomaly from §1. Indeed, since $\prec$ includes the session order, we have $u_1 \prec u_2$ and $u_3 \prec u_4$. Moreover, Bob sees Alice's message, and by Return Value Consistency this can only happen if $u_2 \prec u_3$. Then since $\prec$ is transitive, $u_1 \prec u_4$, and by Return Value Consistency, Bob has to see Alice's deposit.

Causal consistency nevertheless allows the overdraft anomaly from §1: the withdrawals u_5 and u_6 may not be related by $\prec$, and thus may both execute on the balance 100 and succeed. The Conflict Ordering property can be used to disallow this anomaly by declaring that `withdraw` operations on the same account conflict and labeling transactions containing these as strong. Then one of the withdrawal transactions will be guaranteed to causally precede the other. The latter will be executed on the account balance 0 and will fail.

Finally, Eventual Visibility ensures that strong transactions and those causal ones that originate at correct data centers are durable, i.e., will eventually propagate through the system.

To facilitate the use of causal transactions, UNISTORE includes *replicated data types* (aka CRDTs), which implement policies for merging concurrent updates to the same data item [56]. Each data item in the store is associated with a type (e.g., counter, set), which is backed by a CRDT implementation managing updates to it. For example, the programmer can use a counter CRDT to represent an account balance. Then if two clients concurrently deposit 100 and 200 into an empty account using causal transactions, eventually the balance at all replicas will be 300. Using ordinary writes here would yield 100 or 200, depending on the order in which the writes are applied. More generally, CRDTs ensure that two replicas receiving the same set of updates are in the same state, regardless of the receipt order. Together with Eventual Visibility, this implies the expected guarantee of eventual consistency [65]. Due to space constraints, we omit details about the use of CRDTs from our protocol descriptions.

4 Key Design Decisions in UNISTORE

Baseline causal consistency. A causal transaction in UNISTORE first executes at a single data center on a causally consistent snapshot. After this it immediately commits, and its updates are replicated to all other data centers in the background. This minimizes the latency of causal transactions and makes them highly available, i.e., they can be executed even when the network connections between data centers fail.

As is common in causally consistent data stores [3, 22, 42], to ensure that causal transactions execute on consistent snapshots, a data center exposes a remote transaction to clients only after exposing all its dependencies. Then to satisfy the Eventual Visibility property under failures, a data center receiving a remote causal transaction may need to forward it to other data centers, as in reliable broadcast [11] and anti-entropy protocols for replica reconciliation [53].

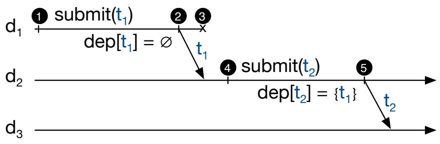

Figure 1: Why UNISTORE may need to forward remote causal transactions.

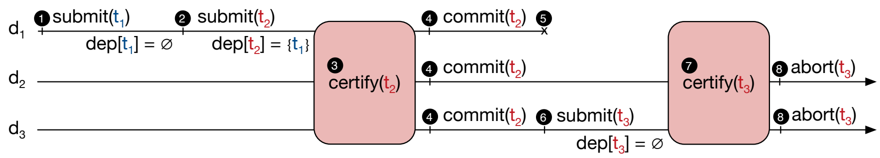

Figure 2: Why UNISTORE needs to ensure that the dependencies of a strong transaction are uniform before committing it.

Figure 1 depicts a scenario that demonstrates how Eventual Visibility could be violated in the absence of this mechanism. Let t_1 be a causal transaction submitted at a data center d_1 (event ❶). Assume that d_1 replicates t_1 to a correct data center d_2 (event ❷) and then fails (event ❸), so that t_1 does not get replicated anywhere else. Let t_2 be a transaction submitted at d_2 after t_1 becomes visible there, so that t_2 depends on t_1 (event ❹). Transaction t_2 will eventually be replicated to all correct data centers (event ❺). But it will never be exposed at any of them, because its dependency t_1 is missing. If data centers can forward remote causal transactions, then d_2 can eventually replicate t_1 to all correct data centers, preventing this problem.

On-demand strong consistency. UNISTORE uses optimistic concurrency control for strong transactions: they are first executed speculatively and the results are then *certified* to determine whether the transaction can commit, or must abort due to a conflict with a concurrent strong transaction [69]. Certifying a strong transaction requires synchronization between the replicas of partitions it accessed, located in different data centers. UNISTORE implements this using an existing fault-tolerant protocol that combines two-phase commit and Paxos [18] while minimizing commit latency. However, just using such a protocol is not enough to make the overall system fault tolerant: for this, before a strong transaction commits, all its causal dependencies must be uniform in the following sense.

DEFINITION 1. A transaction is *uniform* if both the transaction and its causal dependencies are guaranteed to be eventually replicated at all correct data centers.

This adapts the classical notion of uniformity in distributed computing to causal consistency [15]. UNISTORE considers a transaction to be uniform once it is visible at $f+1$ data centers, because at least one of these must be correct, and data centers can forward causal transactions to others.

The following scenario, depicted in Figure 2, demonstrates why committing a strong transaction before its dependencies become uniform can compromise the liveness of the system. Assume that a causal transaction t_1 and a strong transaction t_2 are submitted at a data center d_1 in such a way that t_1 becomes a dependency of t_2 (events ❶ and ❷). Assume also that t_2 is certified, committed and delivered to all relevant replicas (events ❸ and ❹) before t_1 is replicated to any data center, and thus before it is uniform. Now if d_1 fails before replicating t_1 (event ❺), no remote data center will be able to expose t_2, because its dependency t_1 is missing. This violates the Eventual Visibility property, and even worse, no strong transaction conflicting with t_2 can commit from now on. For instance, let t_3 be such a transaction, submitted at d_3 (event ❻). Because d_3 cannot expose t_2, transaction t_3 executes on a snapshot excluding t_2. Hence, t_3 will abort during certification (events ❼ and ❽): committing it would violate the Conflict Ordering property, since transactions t_2 and t_3 conflict, but neither of them is visible to the other. Ensuring that t_1 is uniform before committing t_2 prevents this problem, because it guarantees that t_1 will eventually be replicated at d_3. After this t_2 will be exposed to conflicting transactions at this data center, which will allow them to commit.

Minimizing the latency of strong transactions. Ensuring that all the causal dependencies of a strong transaction are uniform before committing it may significantly increase its latency, since this requires additional communication between data centers. UNISTORE mitigates this problem by executing causal transactions on a snapshot that is slightly in the past, which is allowed by causal consistency. Namely, UNISTORE makes a remote causal transaction visible to the clients only after it is uniform. This minimizes the latency of a strong transaction, since to commit it only needs to wait for causal transactions originating at the local data center to become uniform. We cannot delay the visibility of the latter transactions due to the need to guarantee *read your writes* to local clients.

On-demand durability of causal transactions. Client applications interacting with the external world require hard durability guarantees: e.g., a banking application has to ensure that a withdrawal is durably recorded before authorizing the operation. UNISTORE guarantees that, once a strong transaction commits, the transaction and its dependencies are durable. However, UNISTORE returns from a causal transaction before it is replicated, and thus the transaction may be lost if its origin data center fails. Ensuring the durability of every single causal transaction would require synchronization between data centers on its critical path, defeating the benefits of causal consistency. Instead, UNISTORE reuses the mechanism for tracking uniformity to let the clients pay the cost of durability only when necessary. Even though UNISTORE replicates causal transactions asynchronously, it allows clients to execute a *uniform barrier*, which ensures that the transactions they have observed so far are uniform, and thus durable.

Client migration. Clients may need to migrate between data centers, e.g., because of roaming or for load balancing. UNISTORE also uses the uniformity mechanism to preserve session guarantees during migration. A client wishing to migrate from its local data center d to another data center i first invokes a uniform barrier at d. This guarantees that the transactions the client has observed or issued at d are durable and will eventually become visible at i, even if d fails. The client then makes an *attach* call at the destination data center i that waits until i stores all the above transactions. After this, the client can operate at i knowing that the state of the data center is consistent with the client's previous actions.

Currently UNISTORE does not support consistent client migration in response to a data center failure: if the data center a client is connected to fails, the client will have to restart its session when connecting to a different data center. As shown in [71], this limitation can be lifted without defeating the benefits of causal consistency. We leave integrating the corresponding mechanisms into UNISTORE for future work.

5 Fault-Tolerant Causal Consistency Protocol

We first describe the UNISTORE protocol for the case when all transactions are causal. We give its pseudocode in Algorithms 1 and 2; for now the reader should ignore highlighted lines, which are needed for strong transactions. For simplicity, we assume that each handler in the algorithms executes atomically (although our implementation is parallelized). We reference pseudocode lines using the format algorithm#:line#.

5.1 Metadata

Most metadata in our protocol are represented by vectors with an entry per each data center, where each entry stores a scalar timestamp. However, different pieces of metadata use the vectors in different ways, which we now describe.

Tracking causality. The first use of the vectors is as vector clocks [26,46], to track causality between transactions. Given vectors V_1 and V_2, we write $V_1 < V_2$ if each entry of V_1 is no greater than the corresponding entry of V_2, and at least one is strictly smaller. Each update transaction is tagged with a *commit vector* *commitVec*. The order on these vectors is consistent with the causal order $\prec$ from §3: if $commitVec_1$ and $commitVec_2$ are the commit vectors of two update transactions t_1 and t_2 such that $t_1 \prec t_2$, then $commitVec_1 < commitVec_2$. For a transaction originating at a data center d with a commit vector *commitVec*, we call $commitVec[d]$ its *local timestamp*.

Each replica p_d^m maintains a log $\mathsf{opLog}[k]$ of update operations performed on each data item k stored at the replica. Each log entry stores, together with the operation, the commit vector of the transaction that performed it. This allows reconstructing different versions of a data item from its log.

Representing causally consistent snapshots. The protocol also uses a vector to represent a snapshot of the data store on which a transaction operates: a snapshot vector V represents all transactions with a commit vector $\leq V$. This snapshot is causally consistent. Indeed, consider a transaction t_1 included into it, i.e., $commitVec_1 \leq V$. Since any causal dependency t_0 of t_1 is such that $commitVec_0 < commitVec_1$, we have $commitVec_0 < V$, so that t_0 is also included into the snapshot. A client also maintains a vector $\mathsf{pastVec}$ that represents its *causal past*: a causally consistent snapshot including the update transactions the client has previously observed.

Tracking what is replicated where. Each replica p_d^m maintains three vectors that are used to compute which transactions are uniform. These respectively track the sets of transactions replicated at p_d^m, the local data center d, and $f+1$ data centers. Each of these vectors V represents the set of update transactions originating at a data center i with a local timestamp $\leq V[i]$. Note that this set may not form a causally consistent snapshot. The first vector maintained by p_d^m is $\mathsf{knownVec}$. For each data center i, it defines the prefix of update transactions originating at i (in the order of local timestamps) that p_d^m knows about.

PROPERTY 1. For each data center i, the replica p_d^m stores the updates to partition m by all transactions originating at i with local timestamps $\leq \mathsf{knownVec}[i]$.

Our protocol ensures that $\mathsf{knownVec}[d] \leq \mathsf{clock}$ at any replica in data center d. The vector $\mathsf{knownVec}$ at p_d^m records whether the updates to partition m by a given transaction are stored at this replica. In contrast, the next vector $\mathsf{stableVec}$ records whether the updates to *all* partitions by a transaction are stored at the local data center d.

PROPERTY 2. For each data center i, the data center d stores the updates by all transactions originating at i with local timestamps $\leq \mathsf{stableVec}[i]$. More precisely, we are guaranteed that $\mathsf{knownVec}[i]$ at any replica of $d \geq \mathsf{stableVec}[i]$ at any p_d^m.

Finally, the last vector $\mathsf{uniformVec}$ defines the set of update transactions that p_d^m knows to have been replicated at $f+1$ data centers, including d.

PROPERTY 3. Consider $\mathsf{uniformVec}[i]$ at p_d^m. All update transactions originating at i with local timestamps $\leq \mathsf{uniformVec}[i]$ are replicated at $f+1$ datacenters including d. More precisely: $\mathsf{knownVec}[i]$ at any replica of these data centers $\geq \mathsf{uniformVec}[i]$ at p_d^m.

When $\mathsf{uniformVec}$ is reinterpreted as a causally consistent snapshot, it defines transactions that p_d^m knows to be uniform according to Definition 1:

PROPERTY 4. Consider $\mathsf{uniformVec}$ at p_d^m. All update transactions with commit vectors $\leq \mathsf{uniformVec}$ are uniform.

Proof sketch. Consider a transaction t_1 that originates at a data center i with a commit vector $commitVec_1 \leq \mathsf{uniformVec}$ at p_d^m. In particular, $commitVec_1[i] \leq \mathsf{uniformVec}[i]$, and by Property 3, t_1 is replicated at $f+1$ data centers. We assume at most f failures. Then the transaction forwarding mechanism of our protocol (§4) guarantees that t_1 will eventually be replicated at all correct data centers. Consider now any causal dependency t_2 of t_1 with a commit vector $commitVec_2$. Since commit vectors are consistent with causality, $commitVec_2 < commitVec_1 \leq \mathsf{uniformVec}$. Then as above, we can again establish that t_2 will be replicated at all correct data centers, as required by Definition 1. □

5.2 Causal Transaction Execution

Starting a transaction. A client can submit a transaction to any replica in its local data center by calling START_TX(V), where V is the client's causal past pastVec (line 1:1, for brevity, we omit the pseudocode of the client). A replica p_d^m receiving such a request acts as the transaction *coordinator*. It generates a unique transaction identifier *tid*, computes a snapshot snapVec[*tid*] on which the transaction will execute, and returns *tid* to the client (we explain lines 1:2-3 and similar ones later). The snapshot is computed by combining uniform transactions from uniformVec (line 1:5) with the transactions from the client's causal past originating at d (line 1:6). The former is crucial to minimize the latency of strong transactions (§4), while the latter ensures *read your writes*.

Transaction execution. The client proceeds to execute the transaction *tid* by issuing a sequence of operations at its coordinator via DO_OP (line 1:9). When the coordinator receives an operation *op* on a data item k, it sends a GET_VERSION message with the transaction's snapshot snapVec[*tid*] to the local replica responsible for k (line 1:11). Upon receiving the message (line 1:18), the replica first ensures that it is as up-to-date as required by the snapshot (line 1:21). It then computes the latest version of k within the snapshot by applying the operations from opLog[k] by all transactions with commit vectors $\leq$ snapVec[*tid*]. The result is sent to the coordinator in a VERSION message. After receiving it (line 1:12), the coordinator further applies the operations on k previously executed by the transaction, which are stored in a buffer wbuff[*tid*]; this ensures *read your writes* within the transaction. If the operation is an update, the coordinator then appends it to wbuff[*tid*]. Finally, the coordinator executes the desired operation *op* and forwards its return value to the client.

Commit. A client commits a causal transaction by calling COMMIT_CAUSAL (line 1:26). This returns immediately if the transaction is read-only, since it already read a consistent snapshot (line 1:28). To commit an update transaction, UNISTORE uses a variant of two-phase commit protocol (recall that for simplicity we only consider whole-data center failures, not those of individual replicas, §2). The coordinator first sends a PREPARE message to the replicas in the local data center storing the data items updated by the transaction (line 1:29). The message to each replica contains the part of the write buffer relevant to that replica. When a replica receives the message (line 1:36), it computes the transaction's

Algorithm 1 Transaction execution at p_d^m.

1: **function** START_TX(V)
2:   **for** $i \in \mathcal{D} \setminus \{d\}$ **do**
3:     $\mathsf{uniformVec}[i] \leftarrow \max\{V[i], \mathsf{uniformVec}[i]\}$
4:   **var** $tid \leftarrow$ `generate_tid()`
5:   $\mathsf{snapVec}[tid] \leftarrow \mathsf{uniformVec}$
6:   $\mathsf{snapVec}[tid][d] \leftarrow \max\{V[d], \mathsf{uniformVec}[d]\}$
7:   $\mathsf{snapVec}[tid][strong] \leftarrow \max\{V[strong], \mathsf{stableVec}[strong]\}$
8:   **return** tid

9: **function** DO_OP(tid, k, op)
10:   **var** $l \leftarrow$ `partition(`k`)`
11:   **send** GET_VERSION($\mathsf{snapVec}[tid]$, k) **to** p_d^l
12:   **wait receive** VERSION($state$) **from** p_d^l
13:   **for all** $\langle k, op' \rangle \in \mathsf{wbuff}[tid][l]$ **do** $state \leftarrow \mathsf{apply}(op', state)$
14:   $\mathsf{rset}[tid] \leftarrow \mathsf{rset}[tid] \cup \{\langle k, op \rangle\}$
15:   **if** op is an update **then**
16:     $\mathsf{wbuff}[tid][l] \leftarrow \mathsf{wbuff}[tid][l] \cdot \langle k, op \rangle$
17:   **return** $\mathsf{retval}(op, state)$

18: **when received** GET_VERSION($snapVec$, k) **from** p
19:   **for** $i \in \mathcal{D} \setminus \{d\}$ **do**
20:     $\mathsf{uniformVec}[i] \leftarrow \max\{snapVec[i], \mathsf{uniformVec}[i]\}$
21:   **wait until** $\mathsf{knownVec}[d] \geq snapVec[d] \wedge$
        $\mathsf{knownVec}[strong] \geq snapVec[strong]$
22:   **var** $state \leftarrow \perp$
23:   **for all** $\langle op', commitVec \rangle \in \mathsf{opLog}[k].commitVec \leq snapVec$ **do**
24:     $state \leftarrow \mathsf{apply}(op', state)$
25:   **send** VERSION($state$) **to** p

26: **function** COMMIT_CAUSAL(tid)
27:   **var** $L \leftarrow \{l \mid \mathsf{wbuff}[tid][l] \neq \varnothing\}$
28:   **if** $L = \varnothing$ **then return** $\mathsf{snapVec}[tid]$
29:   **send** PREPARE(tid, $\mathsf{wbuff}[tid][l]$, $\mathsf{snapVec}[tid]$) **to** $p_d^l, l \in L$
30:   **var** $commitVec \leftarrow \mathsf{snapVec}[tid]$
31:   **for all** $l \in L$ **do**
32:     **wait receive** PREPARE_ACK(tid, ts) **from** p_d^l
33:     $commitVec[d] \leftarrow \max\{commitVec[d], ts\}$
34:   **send** COMMIT(tid, $commitVec$) **to** $p_d^l, l \in L$
35:   **return** $commitVec$

36: **when received** PREPARE(tid, $wbuff$, $snapVec$) **from** p
37:   **for** $i \in \mathcal{D} \setminus \{d\}$ **do**
38:     $\mathsf{uniformVec}[i] \leftarrow \max\{snapVec[i], \mathsf{uniformVec}[i]\}$
39:   **var** $ts \leftarrow \mathsf{clock}$
40:   $\mathsf{preparedCausal} \leftarrow \mathsf{preparedCausal} \cup \{\langle tid, wbuff, ts \rangle\}$
41:   **send** PREPARE_ACK(tid, ts) **to** p

42: **when received** COMMIT(tid, $commitVec$)
43:   **wait until** $\mathsf{clock} \geq commitVec[d]$
44:   $\langle tid, wbuff, \_ \rangle \leftarrow$ `find`$(tid, \mathsf{preparedCausal})$
45:   $\mathsf{preparedCausal} \leftarrow \mathsf{preparedCausal} \setminus \{\langle tid, \_, \_ \rangle\}$
46:   **for all** $\langle k, op \rangle \in wbuff$ **do**
47:     $\mathsf{opLog}[k] \leftarrow \mathsf{opLog}[k] \cdot \langle op, commitVec \rangle$
48:   $\mathsf{committedCausal}[d] \leftarrow \mathsf{committedCausal}[d] \cup$
        $\{\langle tid, wbuff, commitVec \rangle\}$

49: **function** UNIFORM_BARRIER(V)
50:   **wait until** $\mathsf{uniformVec}[d] \geq V[d]$

51: **function** ATTACH(V)
52:   **wait until** $\forall i \in \mathcal{D} \setminus \{d\}. \mathsf{uniformVec}[i] \geq V[i]$

prepare time ts from its local clock and adds the transaction to preparedCausal, which stores the set of causal transactions that are prepared to commit at the replica. The replica then returns *ts* to the coordinator in a PREPARE_ACK message.

When the coordinator receives replies from all replicas updated by the transaction, it computes the transaction's commit vector *commitVec*: it sets the local timestamp $commitVec[d]$ to the maximum among the prepare times proposed by the replicas (line 1:33), and it copies the other entries of *commitVec* from the snapshot vector $\mathsf{snapVec}[tid]$ (line 1:30). The latter reflects the fact that the transactions in the snapshot become causal dependencies of *tid*.

After computing *commitVec*, the coordinator sends it in a COMMIT message to the relevant replicas at the local data center (line 1:34) and returns it to the client (line 1:35). The client then sets its causal past pastVec to the commit vector. When a replica receives the COMMIT message (line 1:42), it removes the transaction from preparedCausal, adds the transaction's updates to opLog, and adds the transaction to a set $\mathsf{committedCausal}[d]$, which stores transactions waiting to be replicated to sibling replicas at other data centers.

5.3 Transaction Replication

Each replica p_d^m periodically replicates locally committed update transactions to sibling replicas in other data centers by executing PROPAGATE_LOCAL_TXS (line 2:1). Transactions are replicated in the order of their local timestamps. The prefix of transactions that is ready to be replicated is determined by $\mathsf{knownVec}[d]$: according to Property 1, p_d^m stores updates to m by all transactions originating at d with local timestamps $\leq \mathsf{knownVec}[d]$. Thus, the replica first updates $\mathsf{knownVec}[d]$ while preserving Property 1.

There are two cases of this update. If the replica does not have any prepared transactions ($\mathsf{preparedCausal} = \varnothing$), it sets $\mathsf{knownVec}[d]$ to the current value of the clock (line 2:2). This preserves Property 1 because in this case a new transaction originating at d and updating m will get a prepare time at m higher than the current clock (line 1:39), and thus also a higher local timestamp (line 1:33). If the replica has some prepared transactions, then they may end up getting local timestamps lower than the current clock. In this case, the replica sets $\mathsf{knownVec}[d]$ to just below the smallest prepared time (line 2:3). This preserves Property 1 because: *(i)* currently prepared transactions will get a local timestamp no lower than their prepare time; and *(ii)* as we argued above, new transactions will get a prepare time higher than the current clock and, hence, than the smallest prepare time.

After updating $\mathsf{knownVec}[d]$, the replica sends a REPLICATE message to its siblings with the transactions in $\mathsf{committedCausal}[d]$ such that $commitVec[d] \leq \mathsf{knownVec}[d]$, and then removes them from $\mathsf{committedCausal}[d]$. In other words, the replica sends all transactions from the prefix determined by $\mathsf{knownVec}[d]$ that it has not yet replicated.

When a replica p_d^m receives a REPLICATE message with a set of transactions *txs* originating at a sibling replica p_i^m (line 2:9), it iterates over *txs* in $commitVec[i]$ order. For each new transaction in *txs* with commit vector *commitVec*, the replica adds the transaction's operations to its log and sets $\mathsf{knownVec}[i] = commitVec[i]$. Since communication channels are FIFO, p_d^m processes all transactions from p_i^m in their local timestamp order. Hence, the above update to $\mathsf{knownVec}[i]$ preserves Property 1: p_d^m stores updates originating at p_i^m by all transactions with $commitVec[i] \leq \mathsf{knownVec}[i]$. Finally, the replica adds the transactions to $\mathsf{committedCausal}[i]$, which is used to implement transaction forwarding (§4). Due to

Algorithm 2 Transaction replication at p_d^m.

1: **function** PROPAGATE_LOCAL_TXS() ▷ Run periodically
2: **if** $\mathsf{preparedCausal} = \varnothing$ **then** $\mathsf{knownVec}[d] \leftarrow \mathsf{clock}$
3: **else** $\mathsf{knownVec}[d] \leftarrow \min\{ts \mid \langle\_,\_,ts\rangle \in \mathsf{preparedCausal}\} - 1$
4: **var** $txs \leftarrow \{\langle\_,\_,commitVec\rangle \in \mathsf{committedCausal}[d] \mid commitVec[d] \leq \mathsf{knownVec}[d]\}$
5: **if** $txs \neq \varnothing$ **then**
6: **send** REPLICATE(d, txs) **to** p_i^m, $i \in \mathcal{D} \setminus \{d\}$
7: $\mathsf{committedCausal}[d] \leftarrow \mathsf{committedCausal}[d] \setminus txs$
8: **else send** HEARTBEAT$(d, \mathsf{knownVec}[d])$ **to** p_i^m, $i \in \mathcal{D} \setminus \{d\}$

9: **when received** REPLICATE(i, txs)
10: **for all** $\langle tid, wbuff, commitVec\rangle \in txs$ in $commitVec[i]$ order **do**
11: **if** $commitVec[i] > \mathsf{knownVec}[i]$ **then**
12: **for all** $\langle k, op\rangle \in wbuff$ **do**
13: $\mathsf{opLog}[k] \leftarrow \mathsf{opLog}[k] \cdot \langle op, commitVec\rangle$
14: $\mathsf{committedCausal}[i] \leftarrow \mathsf{committedCausal}[i] \cup \{\langle tid, wbuff, commitVec\rangle\}$
15: $\mathsf{knownVec}[i] \leftarrow commitVec[i]$

16: **when received** HEARTBEAT(i, ts)
17: **pre:** $ts > \mathsf{knownVec}[i]$
18: $\mathsf{knownVec}[i] \leftarrow ts$

19: **function** FORWARD_REMOTE_TXS(i, j)
20: **var** $txs \leftarrow \{\langle\_,\_,commitVec\rangle \in \mathsf{committedCausal}[j] \mid commitVec[j] > \mathsf{globalMatrix}[i][j]\}$
21: **if** $txs \neq \varnothing$ **then send** REPLICATE(j, txs) **to** p_i^m
22: **else send** HEARTBEAT$(j, \mathsf{knownVec}[j])$ **to** p_i^m

23: **function** BROADCAST_VECS() ▷ Run periodically
24: **send** KNOWNVEC_LOCAL$(m, \mathsf{knownVec})$ **to** p_d^l, $l \in \mathcal{P}$
25: **send** STABLEVEC$(d, \mathsf{stableVec})$ **to** p_i^m, $i \in \mathcal{D}$
26: **send** KNOWNVEC_GLOBAL$(d, \mathsf{knownVec})$ **to** p_i^m, $i \in \mathcal{D}$

27: **when received** KNOWNVEC_LOCAL$(l, knownVec)$
28: $\mathsf{localMatrix}[l] \leftarrow knownVec$
29: **for** $i \in \mathcal{D}$ **do** $\mathsf{stableVec}[i] \leftarrow \min\{\mathsf{localMatrix}[n][i] \mid n \in \mathcal{P}\}$
30: $\mathsf{stableVec}[strong] \leftarrow \min\{\mathsf{localMatrix}[n][strong] \mid n \in \mathcal{P}\}$

31: **when received** STABLEVEC$(i, stableVec)$
32: $\mathsf{stableMatrix}[i] \leftarrow stableVec$
33: $G \leftarrow$ all groups with $f+1$ replicas that include p_d^m
34: **for** $j \in \mathcal{D}$ **do**
35: **var** $ts \leftarrow \max\{\min\{\mathsf{stableMatrix}[h][j] \mid h \in g\} \mid g \in G\}$
36: $\mathsf{uniformVec}[j] \leftarrow \max\{\mathsf{uniformVec}[j], ts\}$

37: **when received** KNOWNVEC_GLOBAL$(l, knownVec)$
38: $\mathsf{globalMatrix}[l] \leftarrow knownVec$

the forwarding, p_d^m may receive the same transaction from different data centers. Thus, when processing transactions in the REPLICATE message, it checks for duplicates (line 2:11).

5.4 Advancing the Uniform Snapshot

Replicas run a background protocol that refreshes the information about uniform transactions. This proceeds in two stages. First, a replica keeps track of which transactions have been replicated at the replicas of other partitions in the same data center. To this end, replicas in the same data center periodically exchange KNOWNVEC_LOCAL messages with their knownVec vectors, which they store in a matrix localMatrix (lines 2:24 and 2:27); in our implementation this is done via a dissemination tree. This matrix is then used to compute the vector stableVec, which represents the set of transactions that have been fully replicated at the local data center as per Property 2. To ensure this, a replica computes an entry stableVec$[i]$ as the minimum of knownVec$[i]$ it received from the replicas of other partitions in the same data center (line 2:29).

In the second stage of the background protocol, sibling replicas periodically exchange STABLEVEC messages with their stableVec vectors, which they store in a matrix stableMatrix (lines 2:25 and 2:31). This matrix is then used by a replica to compute uniformVec, which characterizes the update transactions that are replicated at $f+1$ data centers as per Property 3. To this end, a replica first enumerates all groups G of $f+1$ data centers that include its local data center (line 2:33). For each data center j the replica performs the following computation. First, for each group $g \in G$, it computes the minimum j-th entry in the stable vectors of all data centers $h \in g$: $\min\{\mathsf{stableMatrix}[h][j] \mid h \in g\}$. By Property 2 all update transactions originating at j with local timestamp $\leq$ the minimum have been replicated at all data centers in g. The replica then sets uniformVec$[j]$ to the maximum of the resulting values computed for all groups $g \in G$, to cover transactions that are replicated at any such group. According to Property 4, the transactions with commit vectors $\leq$ uniformVec are uniform, and now become visible to transactions coordinated by p_d^m (§5.2).

Replicas also update uniformVec in lines 1:2-3, 1:19-20 and 1:37-38 by incorporating snapVec$[i]$ for remote data centers i. This is safe because a transaction executes on a snapshot that only includes uniform remote transactions.

Finally, if a replica does not receive new transactions for a long time, it sends the value of its knownVec$[d]$ as a heartbeat (lines 2:8 and 2:16). This allows advancing stableVec and uniformVec even under skewed load distributions.

5.5 Transaction Forwarding

As we explained in §4, to guarantee that a transaction originating at a correct data center eventually becomes exposed at all correct data centers despite failures (Eventual Visibility), replicas may have to forward remote update transactions. To determine which transactions to forward, each replica keeps track of the update transactions that have been replicated at sibling replicas in other data centers. To this end, sibling replicas periodically exchange KNOWNVEC_GLOBAL messages with their knownVec vectors, which they store in a matrix globalMatrix (lines 2:26 and 2:37). Thus, p_i^m has received all update transactions from p_j^m with $commitVec[j] \leq \mathsf{globalMatrix}[i][j]$.

A replica p_d^m only forwards transactions when it suspects that a data center j may have failed before replicating all the update transactions originating at it to a data center i (this information is provided by a separate module). In this case, p_d^m executes FORWARD_REMOTE_TXS(i,j) (line 2:19). The function forwards the set of transactions txs received from p_j^m that have not been replicated at p_i^m according to globalMatrix$[i][j]$. For example, in Figure 1, UNISTORE will eventually invoke FORWARD_REMOTE_TXS(d_1,d_3) at replicas in d_2 to forward t_1. The replica p_d^m sends the transactions in txs to p_i^m in a REPLICATE message. If there are no update transactions to forward, p_d^m sends a heartbeat to p_i^m with knownVec$[j]$.

UNISTORE periodically deletes from committedCausal transactions that have been replicated at every data center (omitted from the pseudocode for brevity).

5.6 On-Demand Durability and Client Migration

A client may wish to ensure that the transactions it has observed so far are durable. To this end, the client can call UNIFORM_BARRIER(V) at any replica in its local data center d, where V is the client's causal past pastVec (line 1:49). The replica returns to the client only when all the transactions from pastVec that originate at d are uniform, and thus durable. Then the same holds for all transactions from pastVec, because the protocol only exposes remote transactions to clients when they are already uniform (§5.2).

A client wishing to migrate from its local data center d to another data center i first calls UNIFORM_BARRIER(V) at any replica in d with $V = \mathsf{pastVec}$, to ensure that the transactions the client has observed or issued at d will eventually become visible at i. The client then calls ATTACH(V) at any replica in i (line 1:51). The replica returns when its uniformVec includes all remote transactions from V (line 1:52). The client can then be sure that its transactions at i will operate on snapshots including all the transactions it has observed before.

6 Adding Strong Transactions

We now describe the full UNISTORE protocol with both causal and strong transactions. It is obtained by adding the highlighted lines to Algorithms 1-2 and a new Algorithm 3.

6.1 Metadata

The Conflict Ordering property of our consistency model requires any two conflicting strong transactions to be related one way or another by the causal order $\prec$ (§3). To ensure this, the protocol assigns to each strong transaction a scalar *strong timestamp*, analogous to those used in optimistic concurrency control for serializability [69]. Several vectors used as metadata in the causal consistency protocol (§5.1) are then extended with an extra *strong* entry.

First, we extend commit vectors and those representing causally consistent snapshots. Commit vectors are compared using the previous order $<$, but considering all entries; as before, this order is consistent with the causal order $\prec$. Furthermore, conflicting strong transactions are causally ordered according to their strong timestamps.

PROPERTY 5. For any conflicting strong transactions t_1 and t_2 with commit vectors $commitVec_1$ and $commitVec_2$, we have: $t_1 \prec t_2 \iff commitVec_1[strong] < commitVec_2[strong]$.

A consistent snapshot vector V now defines the set of transactions with a commit vector $\leq V$, according to the new $<$. The vectors $\mathsf{knownVec}$ and $\mathsf{stableVec}$ maintained by a replica p_d^m are also extended with a *strong* entry. The entries $\mathsf{knownVec}[strong]$ and $\mathsf{stableVec}[strong]$ define the prefix of strong transactions that have been replicated at p_d^m and the local data center d, respectively:

PROPERTY 6. Replica p_d^m stores the updates to m by all strong transactions with $commitVec[strong] \leq \mathsf{knownVec}[strong]$.

PROPERTY 7. Data center d stores the updates by all strong transactions with $commitVec[strong] \leq \mathsf{stableVec}[strong]$.

To ensure Property 7, the *strong* entry of stableVec is updated at line 2:30 similarly to its other entries. We do not extend uniformVec, because our commit protocol for strong transactions automatically guarantees their uniformity.

6.2 Transaction Execution

UNISTORE uses optimistic concurrency control for strong transactions, with the same protocol for executing causal and speculatively executing strong transactions. To this end, Algorithm 1 is modified as follows. First, the computation of the snapshot vector $\mathsf{snapVec}[tid]$ is extended to compute the *strong* entry (line 1:7), which is now taken into account when checking that a replica state is up to date (line 1:21). The *strong* entry of the snapshot vector is computed so as to include all strong transactions known to be fully replicated in the local data center, as defined by $\mathsf{stableVec}[strong]$. To ensure *read your writes*, the snapshot additionally includes strong transactions from the client's causal past, as defined by $V[strong]$. Finally, the coordinator of a transaction now maintains not only its write set, but also its read set rset that records all operations by the transaction, including read-only ones (line 1:14). The latter is used to certify strong transactions.

After speculatively executing a strong transaction, the client tries to commit it by calling COMMIT_STRONG at its coordinator (line 3:1). The coordinator first waits until the snapshot on which the transaction operated becomes uniform by calling UNIFORM_BARRIER (line 3:2): as we argued in §4, this is crucial for liveness. The coordinator next submits the transaction to a *certification service*, which determines whether the transaction commits or aborts (line 3:3, see §6.3). In the former case, the service also determines its commit vector, which the coordinator returns to the client. If the transaction commits, the client sets its causal past pastVec to the commit vector; otherwise, it re-executes the transaction.

Algorithm 3 Committing strong transactions at p_d^m.

1: **function** COMMIT_STRONG(*tid*)
2: UNIFORM_BARRIER($\mathsf{snapVec}[tid]$)
3: **return** CERTIFY(tid, $\mathsf{wbuff}[tid]$, $\mathsf{rset}[tid]$, $\mathsf{snapVec}[tid]$)

4: **upon** DELIVER_UPDATES(W)
5: **for all** $\langle wbuff, commitVec \rangle \in W$ in $commitVec[strong]$ order **do**
6: **for all** $\langle k, op \rangle \in wbuff$ **do**
7: $\mathsf{opLog}[k] \leftarrow \mathsf{opLog}[k] \cdot \langle op, commitVec \rangle$
8: $\mathsf{knownVec}[strong] \leftarrow commitVec[strong]$

9: **function** HEARTBEAT_STRONG() ▷ Run periodically
10: **return** CERTIFY($\bot$, $\varnothing$, $\varnothing$, $\vec{0}$)

The certification service also notifies replicas in all data centers about updates by strong transactions affecting them via DELIVER_UPDATES upcalls, invoked in an order consistent with strong timestamps of the transactions (line 3:4). A replica receiving an upcall adds the new operations to its log and refreshes $\mathsf{knownVec}[strong]$ to preserve Property 6.

Finally, a replica p_d^m that has not seen any strong transactions updating its partition m for a long time submits a dummy strong transaction that acts as a heartbeat (line 3:9). Similarly to heartbeats for causal transactions, this allows coping with skewed load distributions.

6.3 Certification Service

We implement the certification service using an existing fault-tolerant protocol from [18], with transaction commit vectors computed using the techniques from [29]. The protocol integrates two-phase commit across partitions accessed by the transaction and Paxos among the replicas of each partition. It furthermore uses white-box optimizations between the two protocols to minimize the commit latency. The use of Paxos ensures that a committed strong transaction is durable and its updates will eventually be delivered at all correct data centers (line 3:4). For each partition, a single replica functions as the Paxos leader. The protocol is described and formally specified elsewhere [18], and here we discuss it only briefly. Its pseudocode and formal specification are given in §A and §C, respectively.

The certification service accepts the read and write sets of a transaction and its snapshot vector (line 3:3). Even though the service is distributed, it guarantees that commit/abort decisions are computed like in a centralized database with optimistic concurrency control – in a total *certification order*. To ensure Conflict Ordering, the decisions are computed using a concurrency-control policy similar to that for serializability [69]: a transaction commits if its snapshot includes all conflicting transactions that precede it in the certification order. The certification service also computes a commit vector for each committed transaction by copying its per-data center entries from the transaction's snapshot vector and assigning a strong timestamp consistent with the certification order.

7 Proof of Correctness

We have rigorously proved that UNISTORE correctly implements the specification of PoR consistency for the case when the data store manages last-writer-wins registers. The proof uses the formal framework from [13, 14, 16] and establishes Properties 1-7 stated earlier. Due to space constraints, we defer the proof to §D.

8 Evaluation

We have implemented UNISTORE and several other protocols (listed in the following) in the same codebase, consisting of 10.3K SLOC of Erlang. We evaluate the protocols on Amazon EC2 using `m4.2xlarge` VMs from 5 different regions. Each VM has 8 virtual cores and 32GB of RAM. The RTT between regions ranges from 26ms to 202ms. Unless otherwise stated, our experiments deploy 3 data centers, thus tolerating a single data center failure: Virginia (US-East), California (US-West) and Frankfurt (EU-FRA). All Paxos leaders are located in Virginia. By default we use 4 replica machines per data center. Each machine stores replicas of 8 partitions, matching the number of cores. Clients are hosted on separate machines in each data center. We run each experiment for at least 5 minutes, with the first and the last minute ignored. Replicas propagate local update transactions (line 2:1) and broadcast vectors (line 2:23) every 5ms.

8.1 Does UNISTORE combine causal and strong consistency effectively?

We start by analyzing the performance of UNISTORE using RUBiS – a popular benchmark that emulates an online auction website such as eBay [40, 41]. It defines 11 read-only transactions and 5 update transactions, e.g., selling items, bidding on items, and consulting outstanding auctions. As in previous work [41], to make the benchmark more challenging, we add an extra update transaction `closeAuction` to declare the winner of an auction. We also borrow from [41] a conflict relation between RUBiS transactions that preserves key integrity invariants in the PoR consistency model. This marks four transactions as strong (`registerUser`, `storeBuyNow`, `storeBid` and `closeAuction`) and declares three conflicts between them. For example, `storeBid`, which places a bid on a item, conflicts with `closeAuction` if both act on the same item: this is needed to preserve the invariant that the winner of an auction is the highest bidder. Our RUBiS database is configured according to the benchmark specification: it is populated with 33,000 items for sale and 1 million users; client

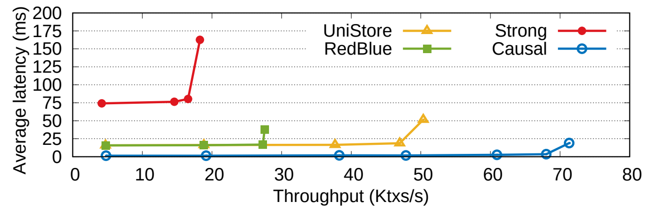

Figure 3: RUBiS benchmark: throughput vs. average latency.

think times are 500ms. We run the bidding mix workload of RUBiS with 15% of update transactions, which yields 10% of strong transactions.

We compare UNISTORE with STRONG, REDBLUE and CAUSAL. STRONG implements serializability [69] as a special case of UNISTORE where all transactions are strong and all pairs of operations conflict. REDBLUE implements redblue consistency [40], which like PoR, combines causal and strong consistency. However, it declares conflicts between all strong transactions. REDBLUE certifies strong transactions at a centralized replicated service, with a replica at each data center. CAUSAL implements causal consistency as a special case of UNISTORE where all transactions are causal. It cannot preserve the integrity invariants of RUBiS, but gives an upper bound on the expected performance.

Throughput and average latency. Figure 3 evaluates average transaction latency and throughput. As the figure shows, UNISTORE exhibits a high throughput: 72% and 183% higher than REDBLUE and STRONG respectively at their saturation point. This is expected, as UNISTORE implements the consistency model that enables the most concurrency. STRONG classifies all transactions as strong. This impacts performance because executing a strong transaction is significantly more expensive than executing a causal one. REDBLUE uses a centralized certification service that saturates before the UNISTORE's distributed service, creating a bottleneck. UNISTORE exhibits an average latency of 16.5ms, lower than 80.4ms of STRONG. The latency of REDBLUE is comparable to that of UNISTORE. This is because both systems mark the same set of transactions as strong. Still, REDBLUE declares conflicts between all strong transaction and thus aborts more transaction than UNISTORE: 0.12% vs 0.027%. The clients whose transactions abort have to retry them, thus increasing latency. Since the abort rate remains low in both cases, the difference in latency is negligible in our experiment. We expect a more significant difference in workloads with higher contention. Finally, in comparison to CAUSAL, UNISTORE penalizes throughput by 45%. This is the unavoidable price to pay to preserve application-specific invariants.

Latency of each transaction type. In UNISTORE, the latency of strong transactions is dominated by the RTT between Virginia (the leader's region) and California (Virginia's closest data center) – 61ms. Strong transactions exhibit a latency of 73.9ms on average. The latency varies depending on the

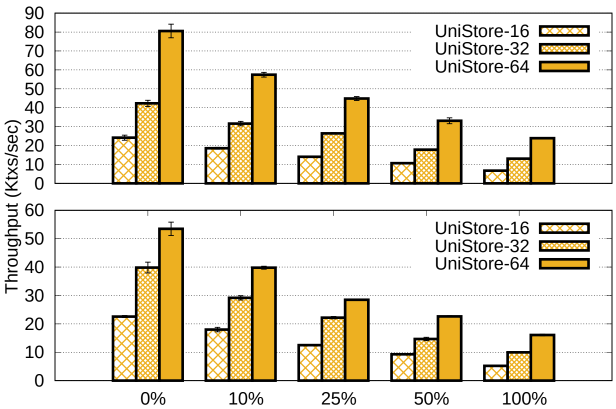

Figure 4: Scalability when varying the ratio of strong transactions with uniform data access (top) and under contention (bottom).

client's location: from 65.4ms on average at the leader's site to 93.2ms at the site furthest from the leader (Frankfurt). Since causal transactions do not require coordination between data centers, they exhibit a very low latency – 1.2ms on average, which is comparable to that of CAUSAL. This demonstrates that UNISTORE is able to mix causal and strong consistency effectively, as the latency of causal transactions remains low regardless of concurrently executing strong transactions.

8.2 How does UNISTORE scale with the number of machines?

We evaluate the peak throughput of UNISTORE as we increase the number of machines per each data center from 2 to 8, i.e., the number of partitions from 16 to 64. We use a microbenchmark with 100% of update transactions, where each transaction accesses three data items. We vary the ratio of strong transactions from 0% to 100% to understand their impact on scalability.

Scalability under low contention. For this set of experiments, the data items accessed by each transactions are picked uniformly at random. This yields a very low contention: e.g., with 16 partitions, the probability of two transactions accessing the same partition is 0.031. As shown by the top plot of Figure 4, UNISTORE is able to scale almost linearly even when the workload includes strong transactions: a 9.76% throughput drop compared to the optimal scalability. This is because, with uniform accesses, the task of committing transactions is balanced among partitions. Thus, when the number of partitions increases, so does the system's capacity. The scalability is not perfect due to the cost of the background protocol that computes stableVec, which grows logarithmically with the number of partitions. The plot also shows that strong transactions are expensive: 25.72% of throughput drop on average with 10% of strong transactions. The performance is dominated by the number of strong transactions that a partition can certify per second.

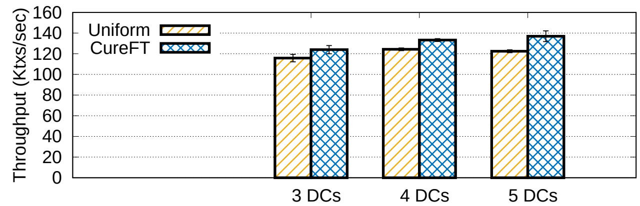

Figure 5: Throughput penalty of tracking uniformity.

Impact of contention. For this set of experiments, we set the ratio of strong transactions that access a designated partition to 20% to create contention. As shown by the bottom plot of Figure 4, UNISTORE is still able to scale fairly well under contention. But, as expected, contention has an impact on scalability: a 17.15% throughput drop from the optimal scalability compared to the 9.76% throughput drop in the experiments without contention.

8.3 What is the cost of uniformity?

We compare CUREFT to UNIFORM. CUREFT implements Cure [3], a causally consistent data store, and makes it fault tolerant by adding transaction forwarding (§4). UNIFORM is a simplified version of UNISTORE that removes all the mechanisms related to strong transactions. UNIFORM tracks uniformity and makes remote transactions visible only when these are uniform; CUREFT does not. We use a microbenchmark with only causal transactions and 15% of update transactions. Each transaction accesses three data items.

Throughput penalty. Figure 5 evaluates the cost of tracking uniformity. It shows the peak throughput when the number of data centers increases from 3 to 5. We first add Ireland and then Brazil. As we do this, the throughput remains almost constant. This is because each data center stores replicas of all partitions and the computational power gained when adding a data center is offset by the cost of replicating update transactions. As the figure shows, the cost of tracking uniformity is small: a 7.97% drop on average. The gap grows as we increase the number of data centers: a 10.61% drop on average with 5 data centers. This is because, to track uniform transactions, sibling replicas exchange messages: the more data centers, the more messages exchanged. The penalty can be reduced by decreasing the frequency at which sibling replicas exchange their stableVec (line 2:25), at the expense of an extra delay in the visibility of remote transactions.

Reading from a uniform snapshot. Figure 6 evaluates the delay on the visibility of remote transactions when reading from a uniform snapshot. We deploy four data centers: Virginia, California, Frankfurt and Brazil. We set $f = 2$ to tolerate 2 data center failures (when $f = 1$, UNIFORM shows no delay). Under such a configuration, a data center makes a transaction visible when it knows that 3 data centers store the transaction and its dependencies (§5.4). The figure shows the cumulative distribution of the delay before updates from

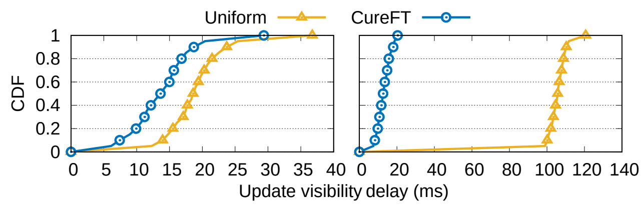

Figure 6: Left: California to Brazil (best case). Right: California to Virginia (worst case).

California are visible in Brazil and Virginia.

The extra delay at Brazil is only of 5ms at the 90th percentile. This is the best case scenario for UNIFORM because Brazil learns that Virginia stores a transaction originating at California only 2ms after receiving it. The worst case scenario for UNIFORM is when the origin and the destination datacenter are the closest ones. This is why the extra delay at Virginia is 92ms at the 90th percentile: Brazil learns that Frankfurt stores a transaction originating at California 88ms after receiving it. Note that when clients communicate only via the data store, the delay is unnoticeable. Even if clients communicate out of band, as the maximum extra delay is less than a 100ms, it is unlikely that a client will miss an update.

9 Related Work

Systems with multiple consistency levels. A number of data stores have combined weak and strong consistency, including several commercial and academic systems that combine eventual and strong consistency [5, 6, 28, 48, 57, 64, 72]. Several academic data stores combined causal and strong consistency [9, 36, 40, 41, 58, 64]. Pileus [64] funnels all updates through a single data center. In the fault-tolerant version of lazy replication [36], causal operations require synchronization between replicas on its critical path. In both cases, causal operations are not highly available, defeating the benefits of causal consistency. Walter [58] restricts causal operations to a specific type and lacks fault tolerance due to the use of two-phase commit across data centers. The remaining works [9, 40, 41] support highly available causal operations, but are not fault tolerant. First, they do not make causal operations uniform on demand to guarantee the liveness of strong operations. Thus, they suffer from the liveness issue we explained in §4 (Figure 2). In addition, these systems do not use fault-tolerant mechanisms even for strong transactions. They guard the use of strong transactions using mechanisms similar to locks; if the lock holder fails before releasing it, no other data center can execute a strong transaction requiring the same lock. This occurs even when the service handing locks is fault-tolerant, as in [41]. Finally, the above systems either do not include mechanisms for partitioning the key space among different machines in a data center or include per-data center centralized services, which limits their scalability (§8.2).

Some group communication systems mix causal and atomic broadcast [10, 67]. However, these systems do not provide mechanisms for maintaining transactional data consistency.

Several papers have proposed tools that use formal verification technology to ensure that consistency choices do not violate application invariants [9, 30, 33, 39, 50, 51]. Such tools can make it easier for programmers to use our system.

Causal consistency implementations. Our subprotocol for causal consistency belongs to a family of highly scalable protocols that avoid using any centralized components or dependency check messages [3, 22, 59–61]; other alternatives are less scalable [4, 8, 12, 21, 31, 42, 43, 47, 71]. While we base our causal consistency subprotocol on an existing one, Cure [3], we have extended it in nontrivial ways, by integrating mechanisms for tracking uniformity (§5.4) and for transaction forwarding (§5.5). Some of the above protocols [31, 60] use hybrid clocks instead of real time [35] to improve performance with large clock skews; this technique can also be integrated into UNISTORE.

SwiftCloud [71] implements *k-stability* [45], a notion similar to uniformity, to enable client migration. SwiftCloud relies on a single per-data center sequencer, which makes tracking k-stability easy, but the data store less scalable. Our protocol is more sophisticated, since we distribute the responsibility of tracking uniformity among the replicas in a data center.

Paxos variants. Several Paxos variants [24, 38, 49, 52] lower the latency by allowing commutative operations to execute at replicas in arbitrary orders. In contrast to them, UNISTORE implements PoR consistency, which allows causal transactions to execute without any synchronization at all.

10 Conclusion

This paper presented UNISTORE, the first fault-tolerant and scalable data store that combines causal and strong consistency. UNISTORE carefully integrates state-of-the-art scalable protocols and extends them in nontrivial ways. To maintain liveness despite data center failures, unlike previous work, UNISTORE commits a strong transaction only when all its causal dependencies are uniform. Our results show that UNISTORE combines causal and strong consistency effectively: 3.7× lower latency on average than a strongly consistent system with 1.2ms latency on average for causal transactions. We expect that the key ideas in UNISTORE will pave the way for practical systems that combine causal and strong consistency.

Acknowledgements. We thank our shepherd, Heming Cui, as well as Gregory Chockler, Vitor Enes, Luís Rodrigues and Marc Shapiro for comments and suggestions. This work was partially supported by an ERC Starting Grant RACCOON, the Juan de la Cierva Formación funding scheme (FJC2018-036528-I), the CCF-Tencent Open Fund (CCF-Tencent RAGR20200124) and the AWS Cloud Credit for Research program.

References

[1] D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2), 2012.

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

[3] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[4] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: A causal+ consistent datastore based on chain replication. In *European Conference on Computer Systems (EuroSys)*, 2013.

[5] Amazon. Read consistency. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html, 2020.

[6] Apache Cassandra. Read repair. https://cassandra.apache.org/doc/latest/operating/read_repair.html, 2020.

[7] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. *IEEE Trans. Parallel Distributed Syst.*, 28(1), 2017.

[8] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *International Conference on Management of Data (SIGMOD)*, 2013.

[9] V. Balegas, N. Preguiça, R. Rodrigues, S. Duarte, C. Ferreira, M. Najafzadeh, and M. Shapiro. Putting the consistency back into eventual consistency. In *European Conference on Computer Systems (EuroSys)*, 2015.

[10] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3), 1991.

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

[12] M. Bravo, L. Rodrigues, and P. Van Roy. Saturn: A distributed metadata service for causal consistency. In *European Conference on Computer Systems (EuroSys)*, 2017.

[13] S. Burckhardt. *Principles of Eventual Consistency*. Now Publishers, 2014.

[14] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *Symposium on Principles of Programming Languages (POPL)*, 2014.

[15] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2nd ed.)*. Springer, 2011.

[16] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *International Conference on Concurrency Theory (CONCUR)*, 2015.

[17] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure coding objects across global data centers. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[18] G. Chockler and A. Gotsman. Multi-shot distributed transaction commit. In *Symposium on Distributed Computing (DISC)*, 2018.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Symposium on Operating Systems Principles (SOSP)*, 2007.

[21] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symposium on Cloud Computing (SoCC)*, 2013.

[22] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Symposium on Cloud Computing (SoCC)*, 2014.

[23] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2), 1988.

[24] V. Enes, C. Baquero, T. F. Rezende, A. Gotsman, M. Perrin, and P. Sutra. State-machine replication for planet-scale systems. In *European Conference on Computer Systems (EuroSys)*, 2020.

[25] FaunaDB. What is FaunaDB? https://docs.fauna.com/fauna/current/introduction.html, 2020.

[26] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Australian Computer Science Conference (ASCS)*, 1988.

[27] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.

[28] Google. Balancing strong and eventual consistency with datastore. https://cloud.google.com/datastore/docs/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore, 2020.

[29] A. Gotsman, A. Lefort, and G. Chockler. White-box atomic multicast. In *International Conference on Dependable Systems and Networks (DSN)*, 2019.

[30] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'Cause I'm strong enough: reasoning about consistency choices in distributed systems. In *Symposium on Principles of Programming Languages (POPL)*, 2016.

[31] C. Gunawardhana, M. Bravo, and L. Rodrigues. Unobtrusive deferred update stabilization for efficient georeplication. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

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

[33] G. Kaki, K. Earanky, K. C. Sivaramakrishnan, and S. Jagannathan. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.

[34] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *European Conference on Computer Systems (EuroSys)*, 2013.

[35] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2014.

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

[37] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

[38] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[39] C. Li, J. Leitão, A. Clement, N. M. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[40] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguiça, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[41] C. Li, N. Preguiça, and R. Rodrigues. Fine-grained consistency for geo-replicated systems. In *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[42] 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 *Symposium on Operating Systems Principles (SOSP)*, 2011.

[43] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency georeplicated storage. In *Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

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

[45] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4), 2011.

[46] F. Mattern. Virtual time and global clocks in distributed systems. In *International Workshop on Parallel and Distributed Algorithms*, 1988.

[47] 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 *Conference on Networked Systems Design and Implementation (NSDI)*, 2017.

[48] Microsoft. Consistency levels in Azure Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels, 2020.

[49] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

[50] S. S. Nair, G. Petri, and M. Shapiro. Proving the safety of highly-available distributed objects. In *European Symposium on Programming (ESOP)*, 2020.

[51] M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro. The CISE tool: proving weakly-consistent applications correct. In *Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)*, 2016.

[52] F. Pedone and A. Schiper. Generic broadcast. In *International Symposium on Distributed Computing (DISC)*, 1999.

[53] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Symposium on Operating Systems Principles (SOSP)*, 1997.

[54] Redis Labs. Causal consistency in an active-active database. https://docs.redislabs.com/latest/rs/administering/database-operations/causal-consistency-crdb/, 2020.

[55] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[56] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2011.

[57] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundaram, M. G. Guajardo, A. Wawrzyniak, S. Boshra, R. Ferreira, M. Nassar, M. Koltachev, J. Huang, S. Sengupta, J. J. Levandoski, and D. B. Lomet. Schema-agnostic indexing with azure documentdb. *Proc. VLDB Endow.*, 8(12), 2015.

[58] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Symposium on Operating Systems Principles (SOSP)*, 2011.

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

[60] K. Spirovska, D. Didona, and W. Zwaenepoel. Paris: Causally consistent transactions with non-blocking reads and partial replication. In *International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[61] K. Spirovska, D. Didona, and W. Zwaenepoel. Optimistic causal consistency for geo-replicated key-value stores. *IEEE Transactions on Parallel and Distributed Systems*, 32(3), 2020.

[62] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *International Conference on Management of Data (SIGMOD)*, 2020.

[63] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *International Conference on Parallel and Distributed Information Systems (PDIS)*, 1994.

[64] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

[65] 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 *Symposium on Operating Systems Principles (SOSP)*, 1995.

[66] M. Tyulenev, A. Schwerin, A. Kamsky, R. Tan, A. Cabral, and J. Mulrow. Implementation of cluster-wide logical clock and causal consistency in MongoDB. In *International Conference on Management of Data (SIGMOD)*, 2019.

[67] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Commun. ACM*, 39(4), 1996.

[68] W. Vogels. Eventually consistent. *CACM*, 52(1), 2009.

[69] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001.

[70] YugabyteDB. Replication. https://docs.yugabyte.com/latest/architecture/docdb-replication/replication/, 2020.

[71] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *International Middleware Conference (Middleware)*, 2015.

[72] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Trans. Comput. Syst.*, 35(4), 2018.

A The Full UniStore Protocol for LWW Registers

Algorithms A1 – A10 given in this section define the full UniStore protocol, including parts omitted from the main text. This version of the protocol is specialized to the case when the data store manages last-writer-wins (LWW) registers.

Algorithm A1 shows the pseudocode of clients. We assume that each client is associated with a unique client identifier. Each client maintains the following variables:

- lc: The Lamport clock at this client.
- d: The data center to which this client is currently connected.
- p: The coordinator partition of the current ongoing transaction.
- tid: The identifier of the current ongoing transaction.
- pastVec: The client's causal past.

A client interacts with UniStore via the following procedures:

- $tid \leftarrow$ START(): Start a transaction and obtain an identifier *tid*.
- $v \leftarrow$ READ(k): Invoke a read operation on key k in the ongoing transaction and obtain a return value v.
- ok $\leftarrow$ UPDATE(k, v): Invoke an update operation on key k and value v in the ongoing transaction.
- ok $\leftarrow$ COMMIT_CAUSAL_TX(): Commit a causal transaction.
- $dec \leftarrow$ COMMIT_STRONG_TX(): Try to commit a strong transaction and obtain a decision $dec \in$ {COMMIT, ABORT}.
- ok $\leftarrow$ CL_UNIFORM_BARRIER(): Execute a uniform barrier.
- ok $\leftarrow$ CL_ATTACH(j): Attach to data center j.

Algorithms A2 – A6 show the pseudocode of replicas. The code needed for strong transactions in Algorithms A2, A3, and A5 is highlighted in red. Each replica p_d^m maintains a set of variables as follows.

- clock: The current time at p_d^m.
- rset: The read sets of transactions coordinated by p_d^m, indexed by transaction identifier *tid*.
- wbuff: The write buffers of transactions coordinated by p_d^m, indexed by transaction identifier *tid*, partition l, and key k.
- snapVec: The snapshot vectors of transactions coordinated by p_d^m, indexed by transaction identifier *tid*.
- opLog: The log of updates performed on keys managed by p_d^m, indexed by key k.
- knownVec, stableVec, uniformVec: The vectors used by p_d^m to track what is replicated where.
- preparedCausal: The set of causal transactions local to p_d^m that are prepared to commit.
- committedCausal: For each data center i, committedCausal[i] stores transactions waiting to be replicated by p_d^m to sibling replicas at other data centers than i.
- localMatrix: The set of knownVec received by p_d^m from other partitions in data center d. It is used to compute stableVec.
- stableMatrix: The set of stableVec received by p_d^m from sibling replicas. It is used to compute uniformVec.
- globalMatrix: The set of knownVec received by p_d^m from sibling replicas. It is used to track what has been replicated at sibling replicas.

We specialize the UniStore protocol to LWW registers in several ways. First, we add code for managing Lamport clocks, highlighted in blue. In particular, each (committed) transaction is associated with a Lamport timestamp, equal to the value of lc at its client when the transaction completes (lines A1:14 and A1:23). Lamport timestamps are totally ordered, with client identifiers used for tie-breaking (see also Definition 54).

Second, in Algorithm A2 we replace DO_OP by the following two procedures:

- $v \leftarrow$ DO_READ(tid, k): Execute a read operation on key k in a transaction with identifier *tid* and obtain a value v.
- DO_UPDATE(tid, k, v): Execute an update operation on key k and value v in a transaction with identifier *tid*.

Finally, in Algorithm A3 we modify the handler of message GET_VERSION:

- GET_VERSION($snapVec, k$): Read the latest value from key k based on the snapshot vector *snapVec*. Specifically, it returns the last update to key k of the transaction with the latest *commitVec* in terms of their Lamport clock order such that $commitVec \leq snapVec$ (line A3:5).

Algorithms A7 – A10 contain the implementation of the transaction certification service (see also §C). The certification service uses an instance of the leader election failure detector Ω_m for each partition m[1]. This primitive ensures that from some point on, all correct processes nominate the same correct process as the leader. For the case when the data store manages only registers, we assume that any two writes to the same object conflict. Other data types and conflict relations can be easily supported by modifying the certification check in Algorithm A8.

[1]Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. J. ACM, 1996.

Algorithm A1 Client operations at client *cl*

```
function START()
    p ← an arbitrary replica in data center d
    tid ← remote call START_TX(pastVec) at p          ▷ ts(START) ← snapVec_d^p[tid]
    return tid

function READ(k)
    ⟨v, lc⟩ ← remote call DO_READ(tid, k) at p
    if lc ≠ ⊥ then
        lc ← max{lc, lc}
    return v

function UPDATE(k, v)
    remote call DO_UPDATE(tid, k, v) at p
    return ok

function COMMIT_CAUSAL_TX()
    lc ← lc + 1                                       ▷ lclock(COMMIT_CAUSAL_TX) ← lc
    vc ← remote call COMMIT_CAUSAL(tid, lc) at p
    pastVec ← vc                                      ▷ ts(COMMIT_CAUSAL_TX) ← pastVec
    return ok

function COMMIT_STRONG_TX()
    lc ← lc + 1
    ⟨dec, vc, lc⟩ ← remote call COMMIT_STRONG(tid, lc) at p
    if dec = COMMIT then
        pastVec ← vc                                  ▷ ts(COMMIT_STRONG_TX) ← pastVec
        lc ← lc                                       ▷ lclock(COMMIT_STRONG_TX) ← lc
    return dec

function CL_UNIFORM_BARRIER()
    var p ← an arbitrary replica in data center d
    remote call UNIFORM_BARRIER(pastVec) at p         ▷ ts(CL_UNIFORM_BARRIER) ← pastVec
    lc ← lc + 1                                       ▷ lclock(CL_UNIFORM_BARRIER) ← lc
    return ok

function CL_ATTACH(j)
    var p ← an arbitrary replica in data center j
    remote call ATTACH(pastVec) at p                  ▷ ts(CL_ATTACH) ← pastVec
    lc ← lc + 1                                       ▷ lclock(CL_ATTACH) ← lc
    d ← j
    return ok
```

Algorithm A2 Transaction coordinator at p_d^m: causal commit

function START_TX(V)
  for $i \in \mathcal{D} \setminus \{d\}$ **do**
    $\mathsf{uniformVec}[i] \leftarrow \max\{V[i], \mathsf{uniformVec}[i]\}$
  var $tid \leftarrow$ `generate_tid()`
  $\mathsf{snapVec}[tid] \leftarrow \mathsf{uniformVec}$
  $\mathsf{snapVec}[tid][d] \leftarrow \max\{V[d], \mathsf{uniformVec}[d]\}$
  $\mathsf{snapVec}[tid][strong] \leftarrow \max\{V[strong], \mathsf{stableVec}[strong]\}$
  return tid

function DO_READ(tid, k)
  var $l \leftarrow$ `partition`(k)
  if $\mathsf{wbuff}[tid][l][k] \neq \bot$ **then**
    return $\langle \mathsf{wbuff}[tid][l][k], \bot \rangle$
  send GET_VERSION($\mathsf{snapVec}[tid], k$) **to** p_d^l
  wait receive VERSION(v, lc) **from** p_d^l
  $\mathsf{rset}[tid] \leftarrow \mathsf{rset}[tid] \cup \{k\}$
  return $\langle v, lc \rangle$

function DO_UPDATE(tid, k, v)
  var $l \leftarrow$ `partition`(k)
  $\mathsf{wbuff}[tid][l][k] \leftarrow v$
  $\mathsf{rset}[tid] \leftarrow \mathsf{rset}[tid] \cup \{k\}$

function COMMIT_CAUSAL(tid, lc)
  var $L \leftarrow \{l \mid \mathsf{wbuff}[tid][l] \neq \varnothing\}$
  if $L = \varnothing$ **then**
    return $\mathsf{snapVec}[tid]$
  send PREPARE($tid, \mathsf{wbuff}[tid][l], \mathsf{snapVec}[tid]$) **to** p_d^l, $l \in L$
  var $commitVec \leftarrow \mathsf{snapVec}[tid]$
  for all $l \in L$ **do**
    wait receive PREPARE_ACK(tid, ts) **from** p_d^l
    $commitVec[d] \leftarrow \max\{commitVec[d], ts\}$
  send COMMIT($tid, commitVec, lc$) **to** p_d^l, $l \in L$
  return $commitVec$

Algorithm A3 Transaction execution at p_d^m

when received GET_VERSION$(snapVec, k)$ **from** p
  for $i \in \mathcal{D} \setminus \{d\}$ **do**
    $\mathsf{uniformVec}[i] \leftarrow \max\{snapVec[i], \mathsf{uniformVec}[i]\}$
  wait until $\mathsf{knownVec}[d] \geq snapVec[d] \wedge \mathsf{knownVec}[strong] \geq snapVec[strong]$
  $\langle v, commitVec, lc\rangle \leftarrow \mathtt{snapshot}(\mathsf{opLog}[k], snapVec)$ ▷ returns the last update to key k by a transaction
  ▷ with the highest Lamport timestamp such that $commitVec \leq snapVec$
  send VERSION(v, lc) **to** p

when received PREPARE$(tid, wbuff, snapVec)$ **from** p
  for $i \in \mathcal{D} \setminus \{d\}$ **do**
    $\mathsf{uniformVec}[i] \leftarrow \max\{snapVec[i], \mathsf{uniformVec}[i]\}$
  var $ts \leftarrow \mathsf{clock}$
  $\mathsf{preparedCausal} \leftarrow \mathsf{preparedCausal} \cup \{\langle tid, wbuff, ts\rangle\}$
  send PREPARE_ACK(tid, ts) **to** p

when received COMMIT$(tid, commitVec, lc)$
  wait until $\mathsf{clock} \geq commitVec[d]$
  $\langle tid, wbuff, \_\rangle \leftarrow \mathtt{find}(tid, \mathsf{preparedCausal})$
  $\mathsf{preparedCausal} \leftarrow \mathsf{preparedCausal} \setminus \{\langle tid, \_, \_\rangle\}$
  for all $\langle k, v\rangle \in wbuff$ **do**
    $\mathsf{opLog}[k] \leftarrow \mathsf{opLog}[k] \cdot \langle v, commitVec, lc\rangle$
  $\mathsf{committedCausal}[d] \leftarrow \mathsf{committedCausal}[d] \cup \{\langle tid, wbuff, commitVec, lc\rangle\}$

function UNIFORM_BARRIER(V)
  wait until $\mathsf{uniformVec}[d] \geq V[d]$

function ATTACH(V)
  wait until $\forall i \in \mathcal{D} \setminus \{d\}.\ \mathsf{uniformVec}[i] \geq V[i]$

Algorithm A4 Transaction replication at p_d^m

function PROPAGATE_LOCAL_TXS() $\triangleright$ Run periodically
  if $\mathsf{preparedCausal} = \varnothing$ **then**
    $\mathsf{knownVec}[d] \leftarrow \mathsf{clock}$
  else
    $\mathsf{knownVec}[d] \leftarrow \min\{ts \mid \langle\_,\_,ts\rangle \in \mathsf{preparedCausal}\} - 1$
  var $txs \leftarrow \{\langle\_,\_,commitVec,\_\rangle \in \mathsf{committedCausal}[d] \mid commitVec[d] \leq \mathsf{knownVec}[d]\}$
  if $txs \neq \varnothing$ **then**
    send REPLICATE(d, txs) **to** p_i^m, $i \in \mathcal{D} \setminus \{d\}$
    $\mathsf{committedCausal}[d] \leftarrow \mathsf{committedCausal}[d] \setminus txs$
  else
    send HEARTBEAT$(d, \mathsf{knownVec}[d])$ **to** p_i^m, $i \in \mathcal{D} \setminus \{d\}$

when received REPLICATE(i, txs)
  for all $\langle tid, wbuff, commitVec, lc\rangle \in txs$ in $commitVec[i]$ order **do**
    if $commitVec[i] > \mathsf{knownVec}[i]$ **then**
      for all $\langle k, v\rangle \in wbuff$ **do**
        $\mathsf{opLog}[k] \leftarrow \mathsf{opLog}[k] \cdot \langle v, commitVec, lc\rangle$
      $\mathsf{committedCausal}[i] \leftarrow \mathsf{committedCausal}[i] \cup \{\langle tid, wbuff, commitVec, lc\rangle\}$
      $\mathsf{knownVec}[i] \leftarrow commitVec[i]$

when received HEARTBEAT(i, ts)
  pre: $ts > \mathsf{knownVec}[i]$
  $\mathsf{knownVec}[i] \leftarrow ts$

function FORWARD_REMOTE_TXS(i, j) $\triangleright$ forward transactions received from data center $j \neq d$ to data center $i \notin \{d, j\}$
  var $txs \leftarrow \{\langle tid,\_,commitVec,\_\rangle \in \mathsf{committedCausal}[j] \mid commitVec[j] > \mathsf{globalMatrix}[i][j]\}$
  if $txs \neq \varnothing$ **then**
    send REPLICATE(j, txs) **to** p_i^m
  else
    send HEARTBEAT$(j, \mathsf{knownVec}[j])$ **to** p_i^m

Algorithm A5 Updating metadata at p_d^m

function BROADCAST_VECS() ▷ Run periodically
  send KNOWNVEC_LOCAL$(m, \mathsf{knownVec})$ **to** p_d^l, $l \in \mathcal{P}$
  send STABLEVEC$(d, \mathsf{stableVec})$ **to** p_i^m, $i \in \mathcal{D}$
  send KNOWNVEC_GLOBAL$(d, \mathsf{knownVec})$ **to** p_i^m, $i \in \mathcal{D}$

when received KNOWNVEC_LOCAL$(l, knownVec)$
  $\mathsf{localMatrix}[l] \leftarrow knownVec$
  for $i \in \mathcal{D}$ **do**
    $\mathsf{stableVec}[i] \leftarrow \min\{\mathsf{localMatrix}[n][i] \mid n \in \mathcal{P}\}$
  $\mathsf{stableVec}[strong] \leftarrow \min\{\mathsf{localMatrix}[n][strong] \mid n \in \mathcal{P}\}$

when received STABLEVEC$(i, stableVec)$
  $\mathsf{stableMatrix}[i] \leftarrow stableVec$
  $G \leftarrow$ all groups with $f+1$ replicas that include p_d^m
  for $j \in \mathcal{D}$ **do**
    var $ts \leftarrow \max\{\min\{\mathsf{stableMatrix}[h][j] \mid h \in g\} \mid g \in G\}$
    $\mathsf{uniformVec}[j] \leftarrow \max\{\mathsf{uniformVec}[j], ts\}$

when received KNOWNVEC_GLOBAL$(l, knownVec)$
  $\mathsf{globalMatrix}[l] \leftarrow knownVec$

Algorithm A6 Committing strong transactions at p_d^m

function COMMIT_STRONG(tid, lc)
  UNIFORM_BARRIER$(\mathsf{snapVec}[tid])$
  $\langle dec, vc, lc \rangle \leftarrow$ CERTIFY$(tid, \mathsf{wbuff}[tid], \mathsf{rset}[tid], \mathsf{snapVec}[tid], lc)$
  return $\langle dec, vc, lc \rangle$

upon DELIVER_UPDATES(txs)
  for $\langle \_, wbuff, commitVec, lc \rangle \in txs$ in $commitVec[\mathrm{strong}]$ order **do**
    for $\langle k, v \rangle \in wbuff$ **do**
      $\mathsf{opLog}[k] \leftarrow \mathsf{opLog}[k] \cdot \langle v, commitVec, lc \rangle$
    $\mathsf{knownVec}[strong] \leftarrow commitVec[strong]$

function HEARTBEAT_STRONG() ▷ Run periodically
  return CERTIFY$(\bot, \varnothing, \varnothing, \vec{0}, \bot)$

Algorithm A7 Certification service at coordinator p_d^m

```
function CERTIFY(tid, wbuff, rset, snapVec, lc)
  var L ← {l | wbuff[l] ≠ ∅} ∪ {partition(k) | k ∈ rset}
  repeat
    send PREPARE_STRONG(tid, wbuff, rset, snapVec, lc) to Ω_l, l ∈ L
    wait receive ALREADY_DECIDED(tid, decision, commitVec, lc)
          ∨ receive ACCEPT_ACK(l, b_l, tid, vote_l, ts_l, lc_l) from a quorum for all l ∈ L
  until not timeout
  if received ALREADY_DECIDED(tid, decision, commitVec, lc) then
    return ⟨decision, commitVec, lc⟩
  else
    commitVec ← snapVec
    commitVec[strong] ← max{ts_l | l ∈ L}
    if ∃l ∈ L. vote_l = ABORT then decision ← ABORT
    else decision ← COMMIT
    lc ← max{lc_l | l ∈ L}
    send DECISION(b_l, tid, decision, commitVec, lc) to Ω_l, l ∈ L
    return ⟨decision, commitVec, lc⟩
```

Algorithm A8 Strong transaction certification at p_d^m

```
function CERTIFICATION_CHECK(W, rset, snapVec, lc)
  for all ⟨_, wbuff′, rset′, _, COMMIT, _, _⟩ ∈ preparedStrong do
    if (∃⟨k, _⟩ ∈ wbuff′[m]. k ∈ rset) ∨ (∃k ∈ rset′. ⟨k, _⟩ ∈ W) then
      return ⟨ABORT, ⊥⟩
  for all ⟨_, wbuff′, COMMIT, commitVec, lc′⟩ ∈ decidedStrong do
    if (∃⟨k, _⟩ ∈ wbuff′[m]. k ∈ rset) ∧ ¬(commitVec ≤ snapVec) then
      return ⟨ABORT, ⊥⟩
    if lc ≤ lc′ then
      lc ← lc′ + 1
  return ⟨COMMIT, lc⟩
```

Algorithm A9 Atomic transaction commit protocol at p_d^m

```
when received PREPARE_STRONG(tid, wbuff, rset, snapVec, lc) from p
    pre: status = LEADER
    if ∃⟨tid, _, decision, commitVec, lc⟩ ∈ decidedStrong then
        send ALREADY_DECIDED(tid, decision, commitVec, lc) to p
    else if ∃⟨tid, _, _, snapVec, vote, ts, lc⟩ ∈ preparedStrong then
        send ACCEPT(ballot, tid, wbuff, rset, snapVec, vote, ts, p, lc) to REPLICAS(m)
    else
        wait until clock > snapVec[strong]
        ts ← clock
        ⟨vote, lc⟩ ← CERTIFICATION_CHECK(wbuff[m], rset, snapVec, lc)
        send ACCEPT(ballot, tid, wbuff, rset, snapVec, vote, ts, p, lc) to REPLICAS(m)

when received ACCEPT(b, tid, wbuff, rset, snapVec, vote, p, ts, lc)
    pre: status ∈ {LEADER, FOLLOWER} ∧ ballot = b
    preparedStrong ← preparedStrong ∪ {⟨tid, wbuff, rset, snapVec, vote, ts, lc⟩}
    send ACCEPT_ACK(m, b, tid, vote, ts, lc) to p

when received DECISION(b, tid, decision, commitVec, lc)
    pre: status = LEADER ∧ ballot = b
    wait until clock ≥ commitVec[strong]
    send LEARN_DECISION(b, tid, decision, commitVec, lc) to REPLICAS(m)

when received LEARN_DECISION(b, tid, decision, commitVec, lc)
    pre: status ∈ {LEADER, FOLLOWER} ∧ ballot = b ∧ ∃⟨tid, wbuff, _, _, _, _, _⟩ ∈ preparedStrong
    preparedStrong ← preparedStrong \ {⟨tid, _, _, _, _, _, _⟩}
    decidedStrong ← decidedStrong ∪ {⟨tid, wbuff, decision, commitVec, lc⟩}

upon
     ∃⟨_, _, COMMIT, commitVec, _⟩ ∈ decidedStrong.
         commitVec[strong] > lastDelivered
       ∧ (¬∃⟨_, _, _, _, COMMIT, ts, _⟩ ∈ preparedStrong. lastDelivered < ts ≤ commitVec[strong])
       ∧ (¬∃⟨_, _, COMMIT, commitVec′, _⟩ ∈ decidedStrong. lastDelivered < commitVec′[strong] < commitVec[strong])
    pre: status = LEADER
    send DELIVER(ballot, commitVec[strong]) to REPLICAS(m)

when received DELIVER(b, ts)
    pre: status ∈ {LEADER, FOLLOWER} ∧ ballot = b ∧ lastDelivered < ts
    lastDelivered ← ts
    var W ← {⟨tid, wbuff[m], commitVec, lc⟩ | ∃⟨tid, wbuff, COMMIT, commitVec, lc⟩ ∈ decidedStrong.
                                               commitVec[strong] = ts}
    upcall DELIVER_UPDATES(W) to p_d^m
```

Algorithm A10 Atomic transaction commit protocol at p_d^m: recovery

upon $\Omega_m \neq \mathsf{trusted}$
  $\mathsf{trusted} \leftarrow \Omega_m$
  if $\mathsf{trusted} = p_d^m$ **then** $\textsc{recover}()$
  else send $\textsc{nack}(\mathsf{ballot})$ **to** $\mathsf{trusted}$

when received $\textsc{nack}(b)$
  pre: $b > \mathsf{ballot}$
  $\mathsf{ballot} \leftarrow b$
  $\textsc{recover}()$

function $\textsc{recover}()$
  send $\textsc{new\_leader}$(any ballot b such that $b > \mathsf{ballot} \land \mathsf{leader}(b) = p_d^m$) **to** $\textsc{replicas}(m)$

when received $\textsc{new\_leader}(b)$ **from** p
  if $\mathsf{trusted} = p \land \mathsf{ballot} < b$ **then**
    $\mathsf{status} \leftarrow \textsc{recovering}$
    $\mathsf{ballot} \leftarrow b$
    send $\textsc{new\_leader\_ack}(\mathsf{ballot}, \mathsf{cballot}, \mathsf{preparedStrong}, \mathsf{decidedStrong})$ **to** p
  else send $\textsc{nack}(\mathsf{ballot})$ **to** p

when received $\{\textsc{new\_leader\_ack}(b, cballot_j, preparedStrong_j, decidedStrong_j) \mid p_j \in Q\}$ **from a quorum** Q
  pre: $\mathsf{status} = \textsc{recovering} \land \mathsf{ballot} = b$
  var $J \leftarrow$ the set of j with maximal $cballot_j$
  $\mathsf{decidedStrong} \leftarrow \bigcup_{j \in J} decidedStrong_j$
  $\mathsf{preparedStrong} \leftarrow \{\langle tid, \_, \_, \_, \_, \_, \_ \rangle \in \bigcup_{j \in J} preparedStrong_j \mid \langle tid, \_, \_, \_, \_ \rangle \notin \mathsf{decidedStrong}\}$
  var $maxPrep \leftarrow \max\{ts \mid \langle \_, \_, \_, \_, \_, ts, \_ \rangle \in \mathsf{preparedStrong}\}$
  var $maxDec \leftarrow \max\{commitVec[\mathsf{strong}] \mid \langle \_, \_, \_, commitVec, \_ \rangle \in \mathsf{decidedStrong}\}$
  wait until $\mathsf{clock} \geq \max\{maxPrep, maxDec\}$
  $\mathsf{cballot} \leftarrow b$
  send $\textsc{new\_state}(\mathsf{ballot}, \mathsf{preparedStrong}, \mathsf{decidedStrong})$ **to** $\textsc{replicas}(m) \setminus \{p_d^m\}$

when received $\textsc{new\_state}(b, preparedStrong, decidedStrong)$ **from** p
  pre: $\mathsf{status} = \textsc{recovering} \land b \geq \mathsf{ballot}$
  $\mathsf{cballot} \leftarrow b$
  $\mathsf{preparedStrong} \leftarrow preparedStrong$
  $\mathsf{decidedStrong} \leftarrow decidedStrong$
  $\mathsf{status} \leftarrow \textsc{follower}$
  send $\textsc{new\_state\_ack}(b)$ **to** p

when received $\textsc{new\_state\_ack}(b)$ **from a set of processes that together with** p_d^m **form a quorum**
  pre: $\mathsf{status} = \textsc{recovering} \land \mathsf{ballot} = b$
  $\mathsf{status} \leftarrow \textsc{leader}$
  var $T \leftarrow \{commitVec[\mathsf{strong}] \mid \exists \langle \_, \_, \textsc{commit}, commitVec, \_ \rangle \in \mathsf{decidedStrong}.$
      $\forall \langle \_, \_, \_, \_, \textsc{commit}, ts, \_ \rangle \in \mathsf{preparedStrong}.\ ts > commitVec[\mathsf{strong}]\}$
  for all $ts \in T$ in increasing order **do**
    send $\textsc{deliver}(b, ts)$ **to** $\textsc{replicas}(m)$

function $\textsc{retry}(tid)$ ▷ Run periodically
  pre: $\exists \langle tid, wbuff, rs, snapVec, \_, \_, lc \rangle \in \mathsf{preparedStrong}$
  $\textsc{certify}(tid, wbuff, rs, snapVec, lc)$

B Consistency Model Specification

B.1 Relations

For a binary relation $\mathcal{R} \subseteq A \times A$ and an element $a \in A$, we define $\mathcal{R}^{-1}(a) = \{b \mid (b,a) \in \mathcal{R}\}$. For a non-empty set A and a total order $\mathcal{R} \subseteq A \times A$, we let $\max_{\mathcal{R}}(A)$ be the maximum element in A according to $\mathcal{R}$. Formally,

$$\max_{\mathcal{R}}(A) = a \Leftrightarrow A \neq \varnothing \wedge \forall b \in A.\ a = b \vee (b,a) \in \mathcal{R}.$$

If A is empty, then $\max_{\mathcal{R}}(A)$ is undefined. We implicitly assume that $\max_{\mathcal{R}}(A)$ is defined whenever it is used.

We call a binary relation a *(strict) partial order* if it is irreflexive and transitive. We call it a *total order* if it additionally relates every two distinct elements one way or another.

B.2 Operations and Events

Transactions in UNISTORE can start, read and write keys, and commit. We assume that each transaction is associated with a unique transaction identifier *tid* from a set TID (corresponding to line A2:8). Besides, clients can issue on-demand barriers and migrate between data centers.

Let *Key* and *Val* be the set of keys and values, respectively. We define O as the set of all possible operations

$$\begin{aligned} O = {} & \{\text{START}(tid) \mid tid \in \mathsf{TID}\} \cup \\ & \{\text{COMMIT\_CAUSAL\_TX}(tid) \mid tid \in \mathsf{TID}\} \cup \\ & \{\text{COMMIT\_STRONG\_TX}(tid, dec) \mid \\ & \quad tid \in \mathsf{TID}, dec \in \{\text{COMMIT}, \text{ABORT}\}\} \cup \\ & \{\text{CL\_UNIFORM\_BARRIER}\} \cup \\ & \{\text{CL\_ATTACH}(j) \mid j \in \mathcal{D}\} \cup \\ & \{\text{READ}(tid,k,v), \text{UPDATE}(tid,k,v) \mid \\ & \quad tid \in \mathsf{TID}, k \in Key, v \in Val\}. \end{aligned}$$

We denote each invocation of such an operation by an *event* from a set E, usually ranged over by e. A function $op : E \to O$ determines the operation a given event denotes. Formally, we use the following notation to denote different types of events.

- E: The set of all events.
- S: The set of START events. That is,

$$S = \{e \in E \mid \exists tid \in \mathsf{TID}.\ op(e) = \text{START}(tid)\}.$$

- R: The set of READ (read) events. That is,

$$R = \{e \in E \mid \exists tid \in \mathsf{TID}, k \in Key, v \in Val.\ op(e) = \text{READ}(tid,k,v)\}.$$

- U: The set of UPDATE (update) events. That is,

$$U = \{e \in E \mid \exists tid \in \mathsf{TID}, k \in Key, v \in Val.\ op(e) = \text{UPDATE}(tid,k,v)\}.$$

- C_{causal}: The set of COMMIT_CAUSAL_TX events. That is,

$$C_{causal} = \{e \in E \mid \exists tid \in \mathsf{TID}.\ op(e) = \text{COMMIT\_CAUSAL\_TX}(tid)\}.$$

- C_{strong}: The set of COMMIT_STRONG_TX events with decision $dec = \text{COMMIT}$. That is,

$$C_{strong} = \{e \in E \mid \exists tid \in \mathsf{TID}.\ op(e) = \text{COMMIT\_STRONG\_TX}(tid, \text{COMMIT})\}.$$

- $C \triangleq C_{causal} \uplus C_{strong}$: The set of all commit events.
- Q: The set of CL_UNIFORM_BARRIER events. That is,

$$Q = \{e \in E \mid op(e) = \text{CL\_UNIFORM\_BARRIER}\}.$$

- A: The set of CL_ATTACH events. That is,

$$A = \{e \in E \mid \exists j \in \mathcal{D}.\ op(e) = \text{CL\_ATTACH}(j)\}.$$

- R_k: The set of read events on key k. That is,

$$R_k = \{e \in E \mid \exists tid \in \mathsf{TID}, v \in Val.\ op(e) = \text{READ}(tid,k,v)\}.$$

- U_k: The set of update events on key k. That is,

$$U_k = \{e \in E \mid \exists tid \in \mathsf{TID}, v \in Val.\ op(e) = \text{UPDATE}(tid,k,v)\}.$$

For different types of events, we define

- $key(e)$: The key that the read or update event $e \in R \cup U$ accesses.
- $rval(e)$: The return value of the read event $e \in R$.
- $uval(e)$: The value written by the update event $e \in U$.

B.3 Transactions

Definition 1 (Transactions)**.** A transaction t is a triple (tid, Y, po), where

- $tid \in \mathsf{TID}$ is a unique transaction identifier;
- $Y \subseteq E \setminus (Q \cup A)$ is a finite, non-empty set of events;
- $po \subseteq Y \times Y$ is the program order, which is total.

We only consider well-formed transactions: according to the po order, t starts with a START event, then performs some number of READ/UPDATE events, and ends with a commit event (COMMIT_CAUSAL_TX or COMMIT_STRONG_TX).

In the following, we denote components of t as in $t.tid$. For simplicity, we assume a dedicated *initial* transaction t_0 which installs initial values to all possible keys before the system launches.

We use the following notations to denote different types of transactions.

- T: The set of all committed transactions.

- T_k: The set of committed transactions that update key k. We also use C_k to denote the set of commit events of transactions in T_k.
- T_{causal}: The set of transactions that end with the COMMIT_CAUSAL_TX events. We call them causal transactions. Causal transactions will always be committed.
- $T_{all\text{-}strong}$: The set of transactions that end with the COMMIT_STRONG_TX events. We call them strong transactions. Strong transactions can be committed or aborted.
- T_{strong}: The set of *committed* strong transactions.

We have $T = T_{causal} \uplus T_{strong}$. For each transaction $t \in T$, we define

- $tid(t) \in \mathsf{TID}$: The transaction identifier $t.tid$ of t.
- $events(t) \subseteq S \cup R \cup U \cup C$: The set $t.Y$ of events in t.
- $ws(t) \subseteq Key \times Val$: The write set of t. It is the set of keys with their values that t updates, which contains at most one value per key. Formally,

$$ws(t) \triangleq \{(key(e), uval(e)) \mid e \in t.Y \cap U\}.$$

- $rs(t) \subseteq Key$: The read set of t. It is the set of keys that t reads. Formally,

$$rs(t) \triangleq \{key(e) \mid e \in t.Y \cap R\}.$$

- $st(t) \in S$: The START event of t. Formally, it is the unique event in the set $t.Y \cap S$.
- $ct(t) \in C$: The commit event of t. Formally, it is the unique event in the set $t.Y \cap C$.
- $ud(t,k) \in U_k$: The *last* update event on key k, if any, in transaction t. Formally,

$$ud(t,k) \triangleq \max_{po}(events(t) \cap U_k).$$

Besides, we define

$$W(t) \triangleq \{k \in Key \mid \langle k, \_ \rangle \in ws(t)\}, \tag{1}$$
$$R(t) \triangleq rs(t) \cup W(t). \tag{2}$$

For a read event e on key k in transaction t, if there exist update events on k preceding e in t, then e is called an *internal* read event. Otherwise, e is called an *external* read event. We denote the sets of internal reads and external reads by R_{INT} and R_{EXT}, respectively. That is, $R = R_{\text{INT}} \uplus R_{\text{EXT}}$.

We also distinguish commit events for read-only transactions from those for update transactions, and denote their sets by C_{RO} and C_{RW}, respectively. That is, $C = C_{\text{RO}} \uplus C_{\text{RW}}$.

For notational convenience, for an event $e \in E \setminus (Q \cup A)$, we also define $tx(e)$ to be the transaction containing e and

$$st(e) \triangleq st(tx(e)),$$
$$ct(e) \triangleq ct(tx(e)).$$

B.4 Abstract Executions

Clients interacts with UNISTORE by issuing transactions and CL_UNIFORM_BARRIER and ATTACH events. We use histories to record such interactions in a single computation. Note that histories only record committed transactions.

Definition 2 (Histories)**.** A *history* is a tuple

$$H = (X, client, dc, so)$$

such that

- $X \subseteq T \cup Q \cup A$ is a set of committed transactions and CL_UNIFORM_BARRIER and ATTACH events;
- $client : X \to \mathbb{C}$ is a function that returns
  - the client $client(t)$ which issues the transaction $t \in (X \cap T)$,
  - the client $client(q)$ which issues the CL_UNIFORM_BARRIER event $q \in (X \cap Q)$, or
  - the client $client(a)$ which issues the ATTACH event $a \in (X \cap A)$;
- $dc : X \to \mathcal{D}$ is a function that returns the original data center $dc(t)$ of transaction $t \in (X \cap T)$, $dc(q)$ of CL_UNIFORM_BARRIER event $q \in (X \cap Q)$, or $dc(a)$ of ATTACH event $a \in (X \cap A)$;
- $so \subseteq X \times X$ is the *session order* on X. Consider $s_1, s_2 \in X$. We say that s_1 precedes s_2 in the session order, denoted $s_1 \xrightarrow{so} s_2$, if they are executed by the same client and s_1 is executed before s_2.

In the following, we denote components of H as in $H.X$ and often shorten $H.X$ by X when it is clear. Let $V_H \triangleq \bigcup(H.X \cap T).Y$ be the set of transactional events in history H.

A consistency model is specified by a set of histories. To define this set, we extend histories with two relations, declaratively describing how the system processes transactions and CL_UNIFORM_BARRIER events.

Definition 3 (Abstract Executions)**.** An *abstract execution* is a triple

$$A = ((X, client, dc, so), vis, ar)$$

such that

- $(X, client, dc, so)$ is a history;
- Visibility $vis \subseteq X \times X$ is a partial order;
- Arbitration $ar \subseteq X \times X$ is a total order.

For $H = (X, client, dc, so)$, we often shorten $((X, client, dc, so), vis, ar)$ by (H, vis, ar).

B.5 Partial Order-Restrictions Consistency

We aim to show that UNISTORE implements a transactional variant of *Partial Order-Restrictions consistency (*POR *consistency)* [40, 41] for LWW registers. A history H of UNISTORE satisfies POR, denoted $H \models$ POR, if it can be extended to an abstract execution that satisfies several axioms, defined in the following:

$$
\begin{aligned}
H \models \text{PoR} \Leftrightarrow \exists vis, ar.\ (H, vis, ar) \models\ & \\
& \text{RVAL} \wedge \\
& \text{CAUSALCONSISTENCY} \wedge \\
& \text{CONFLICTORDERING} \wedge \\
& \text{EVENTUALVISIBILITY}.
\end{aligned}
$$

UNISTORE satisfies PoR, denoted UNISTORE $\models$ PoR, if all its histories do.

Given an abstract execution $A = (H, vis, ar)$, the axioms are defined as follows.

Definition 4 (RVAL, [16]). The Return Value Consistency (RVAL) specifies the return value of each read event.

$$\text{RVAL} \triangleq \text{INTRVAL} \wedge \text{EXTRVAL}.$$

Here INTRVAL requires an internal read event e on key k to read from the last update event on k preceding e in the same transaction. Formally,

$$
\begin{aligned}
& \text{INTRVAL} \triangleq \forall e \in R_{\text{INT}} \cap R_k \cap V_H. \\
& \quad rval(e) = uval\big(\max_{po}(po^{-1}(e) \cap U_k)\big).
\end{aligned}
$$

EXTRVAL requires an external read event e on key k to read from the last update event on k in the last transaction preceding $tx(e)$ in ar, among the set of transactions visible to $tx(e)$. Formally,

$$
\begin{aligned}
& \text{EXTRVAL} \triangleq \forall e \in R_{\text{EXT}} \cap R_k \cap V_H. \\
& rval(e) = uval\Big(ud\big(\max_{ar}\big(vis^{-1}(tx(e)) \cap T_k\big), k\big)\Big).
\end{aligned}
$$

Definition 5 (CAUSALCONSISTENCY, [13]).

$$
\begin{aligned}
\text{CAUSALCONSISTENCY} \triangleq\ & \text{CAUSALVISIBILITY} \wedge \\
& \text{CAUSALARBITRATION},
\end{aligned}
$$

where

$$\text{CAUSALVISIBILITY} \triangleq (so \cup vis)^{+} \subseteq vis;$$

$$\text{CAUSALARBITRATION} \triangleq vis \subseteq ar.$$

The Conflict Ordering property requires that out of any two conflicting strong transactions, one must be visible to the other. Formally,

Definition 6 (Conflict Relation). The conflict relation, denoted by $\bowtie$, between strong transactions is a symmetric relation defined as follows:

$$
\begin{aligned}
& \forall t, t' \in T_{strong}.\ t \bowtie t' \Leftrightarrow \\
& \quad (R(t) \cap W(t') \neq \varnothing) \vee (W(t) \cap R(t') \neq \varnothing).
\end{aligned}
$$

Definition 7 (CONFLICTORDERING).

$$
\begin{aligned}
& \text{CONFLICTORDERING} \triangleq \forall t_1, t_2 \in X \cap T_{strong}. \\
& \quad t_1 \bowtie t_2 \Rightarrow t_1 \xrightarrow{vis} t_2 \vee t_2 \xrightarrow{vis} t_1.
\end{aligned}
$$

The Eventual Visibility property requires that a transaction that originates at a correct data center, that is visible to some CL_UNIFORM_BARRIER events, or that is a strong transaction eventually becomes visible at all correct data centers. Let $\mathcal{C} \subseteq \mathcal{D}$ be the set of correct data centers. Formally,

Definition 8 (EVENTUALVISIBILITY).

$$
\begin{aligned}
& \text{EVENTUALVISIBILITY} \triangleq \forall t \in X \cap T. \\
& dc(t) \in \mathcal{C} \vee (\exists q \in Q.\ t \xrightarrow{vis} q) \vee t \in T_{strong} \\
& \quad \Rightarrow \Big| \Big\{ t' \in T \mid \neg(t \xrightarrow{vis} t') \Big\} \Big| < \infty.
\end{aligned}
$$

C Transaction Certification Service Specification

C.1 Interface

The *Transaction Certification Service* (TCS) [18] is responsible for certifying strong transactions issued by transaction coordinators, computing commit vectors and Lamport clocks for committed transactions, and (asynchronously) delivering committed transactions to replicas.

Each strong transaction $t \in T_{all\text{-}strong}$ submitted to TCS may be associated with its read set $rs(t)$, write set $ws(t)$, snapshot vector $snapshotVec(t)$ (Definition 10), commit vector $commitVec(t)$ (Definition 11), and Lamport clock $lclock(t)$ (Definition 52). From $rs(t)$ and $ws(t)$ we can then define $W(t)$ and $R(t)$ according to (1) and (2), respectively. Note that we have $W(t) \subseteq R(t)$.

Transaction coordinators for strong transactions interact with TCS using two types of *actions*. Coordinators can make certification requests (corresponding to procedure CERTIFY of Algorithm A7) of the form

$$\mathsf{certify}(tid(t), ws(t), R(t), snapshotVec(t), C(t)),$$

where $t \in T_{all\text{-}strong}$ and $C(t) \in \mathbb{N}$ denotes the contribution of $client(t)$ to the Lamport clock of t. The TCS responses are of the form

$$\mathsf{decide}(tid(t), dec, vc, lc),$$

containing a decision dec from $\mathbb{D} = \{\text{COMMIT}, \text{ABORT}\}$ for t, a commit vector vc from $\mathbb{V}$ for t if $dec = \text{COMMIT}$, and a Lamport clock lc from $\mathbb{N}$ for t if $dec = \text{COMMIT}$. If $dec = \text{ABORT}$, then vc and lc are irrelevant.

Besides, TCS can deliver some payload W to a replica via **upcall** *actions* $\mathsf{deliver}(W)$ (corresponding to procedure DELIVER_UPDATES of Algorithm A6). We denote by $\mathsf{deliver}_d^m(W)$ the delivery of the payload W to a replica p_d^m, when the latter is relevant. The payload W in $\mathsf{deliver}_d^m(W)$

is a set of tuples of the form $\langle tid, wbuff, commitVec, lc\rangle$, each of which corresponds to the updates $wbuff \subseteq Key \times Val$ performed at a particular partition m by a particular committed strong transaction with transaction identifier tid, commit vector $commitVec$, and Lamport clock lc.

C.2 Certification Functions

TCS is specified using a certification function

$$F : 2^{T_{\text{strong}}} \times T_{\textit{all-strong}} \to \mathbb{D} \times \mathbb{V} \times \mathbb{N}. \tag{3}$$

For a strong transaction $t \in T_{\textit{all-strong}}$ and the set $T_c \subseteq T_{\textit{strong}}$ of previously committed strong transactions, $F(T_c, t)$ returns not only the decision $dec \in \mathbb{D}$, but also the commit vector $vc \in \mathbb{V}$ and Lamport clock $lc \in \mathbb{N}$ for t. We use $F_{\text{dec}}(T_c, t)$, $F_{\text{vec}}(T_c, t)$, and $F_{\text{lc}}(T_c, t)$ to select the first, second, and third component of $F(T_c, t)$, respectively.

The decision $F_{\text{dec}}(T_c, t)$ should satisfy

$$\begin{aligned} &F_{\text{dec}}(T_c, t) = \text{COMMIT} \Leftrightarrow \forall k \in R(t).\ \forall t' \in T_c. \\ &\quad \big(k \in W(t') \Rightarrow \textit{commitVec}(t') \le \textit{snapshotVec}(t)\big). \end{aligned} \tag{4}$$

The commit vector $F_{\text{vec}}(T_c, t)$ should satisfy

$$\begin{aligned} &(\forall i \in \mathcal{D}.\ F_{\text{vec}}(T_c, t)[i] = \textit{snapshotVec}(t)[i]) \\ &\wedge F_{\text{vec}}(T_c, t)[\textit{strong}] > \textit{snapshotVec}(t)[\textit{strong}] \\ &\wedge \forall t' \in T_c.\ t \bowtie t' \Rightarrow F_{\text{vec}}(T_c, t) \ge \textit{commitVec}(t'). \end{aligned} \tag{5}$$

The Lamport clock $F_{\text{lc}}(T_c, t)$ should satisfy

$$\begin{aligned} &F_{\text{lc}}(T_c, t) \ge C(t) \wedge \\ &\quad \big(\forall t' \in T_c.\ t \bowtie t' \Rightarrow F_{\text{lc}}(T_c, t) > \textit{lclock}(t')\big). \end{aligned} \tag{6}$$

C.3 Histories of TCS

TCS executions are represented by *histories*, which are (possibly infinite) sequences of certify, decide, and deliver actions. For a TCS history h, we use $\mathsf{act}(h)$ to denote the set of actions in h. For actions $act, act' \in \mathsf{act}(h)$, we write $act \prec_h act'$ when act occurs before act' in h. A strong transaction $t \in T_{\textit{all-strong}}$ *commits* in a history h if h contains $\mathsf{decide}(tid(t), \text{COMMIT}, \_, \_)$. We denote by $\mathsf{committed}(h)$ the projection of h to actions corresponding to the strong transactions that are committed in h.

Each history h needs to meet the following requirements.

(R1) For each strong transaction $t \in T_{\textit{all-strong}}$, there is at most one $\mathsf{certify}(tid(t), \_, \_, \_, \_)$ action in h.

(R2) For each action $\mathsf{decide}(tid, \_, \_, \_) \in \mathsf{act}(h)$, there is exactly one $\mathsf{certify}(tid, \_, \_, \_, \_)$ action in h such that

$$\mathsf{certify}(tid, \_, \_, \_, \_) \prec_h \mathsf{decide}(tid, \_, \_, \_).$$

(R3) For each action $\mathsf{deliver}(W) \in \mathsf{act}(h)$ and each $\langle tid, \_, \_, \_\rangle \in W$, there is *no* $\mathsf{decide}(tid, \text{ABORT}, \_, \_)$ action in h.

(R4) Every committed strong transaction is delivered at most once to each replica.

(R5) For each action $\mathsf{deliver}(W) \in \mathsf{act}(h)$ and each $\langle tid, \_, \_, \_\rangle \in W$, there is a $\mathsf{certify}(tid, \_, \_, \_, \_)$ action such that

$$\mathsf{certify}(tid, \_, \_, \_, \_) \prec_h \mathsf{deliver}(W).$$

(R6) At each replica p_d^m, committed strong transactions are delivered in the increasing order of their strong timestamps. Formally, for any two distinct actions $\mathsf{deliver}_d^m(W_1)$ and $\mathsf{deliver}_d^m(W_2)$ with payloads W_1 and W_2, respectively,

$$\begin{aligned} &\mathsf{deliver}_d^m(W_1) \prec_h \mathsf{deliver}_d^m(W_2) \Rightarrow \\ &\quad \forall \langle \_, \_, \textit{commitVec}_1, \_\rangle \in W_1. \\ &\quad \forall \langle \_, \_, \textit{commitVec}_2, \_\rangle \in W_2. \\ &\qquad \textit{commitVec}_1[\textit{strong}] < \textit{commitVec}_2[\textit{strong}]. \end{aligned}$$

A history is *complete* if every certify action in it has a matching decide action. A complete history h is *sequential* if it consists of consecutive pairs of certify and matching decide actions. For a complete history h, a *permutation* h' of h is a sequential history such that

- h and h' contain the same actions, i.e., $\mathsf{act}(h) = \mathsf{act}(h')$.
- Transactions are certified in h' according to their session order.

$$\begin{aligned} &\forall t, t' \in T_{\textit{all-strong}}.\ t \xrightarrow{so} t' \Rightarrow \\ &\quad \mathsf{decide}(tid(t), \_, \_, \_) \prec_{h'} \mathsf{certify}(tid(t'), \_, \_, \_, \_). \end{aligned}$$

C.4 TCS Correctness: Safety and Liveness

C.4.1 Safety of TCS

A complete sequential history h is *legal* with respect to a certification function F, if its results are computed so as to satisfy (4) – (6) according to F:

$$\begin{aligned} &\forall act = \mathsf{decide}(tid(t), dec, vc, lc) \in \mathsf{act}(h). \\ &\quad (dec, vc, lc) = F(\{t' \mid \\ &\qquad \mathsf{decide}(tid(t'), \text{COMMIT}, \_, \_) \prec_h act\}, t). \end{aligned}$$

A history h is *correct* with respect to F if $h \mid \mathsf{committed}(h)$ has a legal permutation. A TCS implementation is *correct* with respect to F if so are all its histories.

C.4.2 Liveness of TCS

TCS guarantees that every committed strong transaction will eventually be delivered by every correct data center. Formally,

$$\begin{aligned} &\forall act = \mathsf{decide}(tid, \text{COMMIT}, \_, \_) \in \mathsf{act}(h). \\ &\quad \forall m \in \textit{partitions}(tid).\ \forall c \in \mathcal{C}. \\ &\qquad \exists act' = \mathsf{deliver}_c^m(W) \in \mathsf{act}(h). \\ &\qquad\quad \langle tid, \_, \_, \_\rangle \in W \wedge act \prec_h act'. \end{aligned} \tag{7}$$

Here $\mathit{partitions}(\mathit{tid})$ denotes the set of partitions that a particular transaction with transaction identifier *tid* accesses.

A TCS implementation meets the liveness requirement if every history produced by its maximal execution satisfies (7).

C.5 TCS Correctness

The proof of TCS correctness is an adjustment of the ones in [18, 29].

Theorem 9. The TCS implementation in UNISTORE (Algorithms A7 – A10) is correct with respect to the certification function F in (3) and meets the liveness requirement in (7).

D The Proof of UNISTORE Correctness

Consider an execution of UNISTORE with a history $H = (X, \mathit{client}, \mathit{dc}, \mathit{so})$. We prove that H satisfies POR by constructing an abstract execution A (Theorem 84). We also establish the liveness guarantees of UNISTORE (Theorem 86).

D.1 Assumptions

We take the following assumptions about UNISTORE.

ASSUMPTION 1. For any replica p_d^m in data center d, clock at p_d^m is strictly increasing until d (may) crash.

ASSUMPTION 2. Replicas are connected by reliable FIFO channels: messages are delivered in FIFO order, and messages between correct data centers are guaranteed to be eventually delivered.

ASSUMPTION 3. We assume that in an execution of UNISTORE, any clients and up to f data centers may crash and that $D > 2f$.

ASSUMPTION 4. We assume fairness of procedures of UNISTORE: In an execution, if a procedure is enabled infinitely often, then it will be executed infinitely often.

ASSUMPTION 5. We consider only *well-formed* executions, in which for each client:

- transactions are issued in sequence; and
- both CL_UNIFORM_BARRIER and CL_ATTACH events can be issued only outside of transactions.

ASSUMPTION 6. We consider only executions where every causal commit event (i.e., COMMIT_CAUSAL_TX) completes and every strong commit event (i.e., COMMIT_STRONG_TX) that calls the TCS completes.

We make the last assumption to simplify the technical development. The other assumptions come from the system model.

D.2 Notations

We use cl to range over clients from a finite set $\mathbb{C}$. We also use the following notations to refer to different types of variables and their values (below are some typical examples).

- $\mathsf{snapVec}_d^m$: The variable snapVec at replica p_d^m.
- $(\mathsf{snapVec}_d^m)_e$: The value of variable $\mathsf{snapVec}_d^m$ after the event e is performed at replica p_d^m.
- $\mathsf{snapVec}_d^m(\tau)$: The value of $\mathsf{snapVec}_d^m$ at some specific time τ.
- $\mathsf{pastVec}_{cl}$: The variable pastVec at client cl.
- $(\mathsf{pastVec}_{cl})_e$: The value of variable $\mathsf{pastVec}_{cl}$ after the event e is performed at client cl.
- $\mathit{snapVec}_{(\text{GET\_VERSION},e)}$: The actual value of *parameter snapVec* of handler GET_VERSION for event e.
- $\mathit{commitVec}_{(\text{COMMIT\_CAUSAL},e)}$: The value of the *local variable commitVec* in procedure COMMIT_CAUSAL after event e is performed.

Besides, we use $\mathit{coord}(t)$ to denote the coordinator partition of transaction t.

Each transaction is associated with a snapshot vector and a commit vector.

Definition 10 (Snapshot Vector)**.** Let $t \in T$ be a transaction. Let $d \triangleq \mathit{dc}(t)$ and $m \triangleq \mathit{coord}(t)$. We define its snapshot vector $\mathit{snapshotVec}(t)$ as

$$\mathit{snapshotVec}(t) \triangleq (\mathsf{snapVec}_d^m)_{st(t)}[t].$$

Definition 11 (Commit Vector)**.** Let $t \in T$ be a transaction. Let $d \triangleq \mathit{dc}(t)$ and $m \triangleq \mathit{coord}(t)$. We define its commit vector $\mathit{commitVec}(t)$ as follows.

- If t is a read-only causal transaction, then
$$\mathit{commitVec}(t) \triangleq (\mathsf{snapVec}_d^m)_{ct(t)}[t].$$
- If t is an update causal transaction, then
$$\mathit{commitVec}(t) \triangleq \mathit{commitVec}_{(\text{COMMIT\_CAUSAL},ct(t))}.$$
- If t is a committed strong transaction, then
$$\mathit{commitVec}(t) \triangleq vc_{(\text{COMMIT\_STRONG},ct(t))}.$$

Lemma 12.

$$\forall t \in T.\ \mathit{commitVec}(t) \geq \mathit{snapshotVec}(t).$$

Proof. We perform a case analysis according to the type of t.

- CASE I: t is a read-only causal transaction. By Definition 10 of $\mathit{snapshotVec}(t)$, Definition 11 of $\mathit{commitVec}(t)$, and Assumption 5,
$$\mathit{commitVec}(t) = \mathit{snapshotVec}(t).$$
- CASE II: t is an update causal transaction. By lines A2:26 and A2:29,
$$\mathit{commitVec}(t) \geq \mathit{snapshotVec}(t).$$

- Case III: t is a strong transaction. By line A6:3 and (5),

$$commitVec(t) \geq snapshotVec(t).$$

$\square$

For client cl, we use $cur\_dc(cl)$ to denote the data center to which cl is currently attached. We also use $T|_{cl}$ to denote the set of transactions issued by cl. Formally,

$$T|_{cl} \triangleq \{t \in T \mid client(t) = cl\}.$$

For a transaction t and a partition m, we use $ws(t)[m]$ to denote the subset of $ws(t)$ restricted to partition m. Formally,

$$ws(t)[m] \triangleq \{\langle k, v\rangle \in ws(t) \mid \texttt{partition}(k) = m\}.$$

For notational convenience, we also define

$$log(t) \triangleq \{\langle k, v, commitVec(t), lclock(t)\rangle \mid \langle k, v\rangle \in ws(t)\},$$

and

$$log(t)[m] \triangleq \{\langle k, v, commitVec(t), lclock(t)\rangle \mid \langle k, v\rangle \in ws(t)[m]\}.$$

For a key $k \in Key$ and a transaction $t \in T_k$, let $log(t)[k]$ be the unique tuple

$$\langle k, v, commitVec(t), lclock(t)\rangle$$

in $log(t)$.

D.3 Metadata for Causal Transactions

A causal transaction is *committed* when COMMIT_CAUSAL for it returns. A causal transaction is *committed at replica* p_d^m when COMMIT for it at p_d^m returns.

D.3.1 Properties of $\mathsf{knownVec}$

Lemma 13. For any replica p_d^m in data center d, $\mathsf{knownVec}_d^m[d]$ is non-decreasing.

Proof. Consider two points of time τ_1 and τ_2 such that $\tau_1 < \tau_2$. We need to show that

$$\mathsf{knownVec}_d^m(\tau_1)[d] \leq \mathsf{knownVec}_d^m(\tau_2)[d].$$

Note that $\mathsf{knownVec}_d^m[d]$ is updated only at lines A4:3 or A4:5. We distinguish between the following four cases.

- Case I: Both $\mathsf{knownVec}_d^m(\tau_1)[d]$ and $\mathsf{knownVec}_d^m(\tau_2)[d]$ are set at line A4:3. By line A4:3 and Assumption 1,

$$\begin{aligned}\mathsf{knownVec}_d^m(\tau_1)[d] &= \mathsf{clock}_d^m(\tau_1)\\ &< \mathsf{clock}_d^m(\tau_2)\\ &= \mathsf{knownVec}_d^m(\tau_2)[d].\end{aligned}$$

- Case II: $\mathsf{knownVec}_d^m(\tau_1)[d]$ is set at line A4:3 and $\mathsf{knownVec}_d^m(\tau_2)[d]$ is set at line A4:5. By line A4:3,

$$\mathsf{knownVec}_d^m(\tau_1)[d] = \mathsf{clock}_d^m(\tau_1).$$

By the fact that $\mathsf{preparedCausal}_d^m(\tau_1) = \emptyset$, $\tau_2 > \tau_1$, and line A3:11,

$$\begin{aligned}&\forall \langle \_, \_, ts\rangle \in \mathsf{preparedCausal}_d^m(\tau_2).\\ &\qquad ts > \mathsf{clock}_d^m(\tau_1) = \mathsf{knownVec}_d^m(\tau_1)[d].\end{aligned}$$

Therefore, by line A4:5,

$$\begin{aligned}&\mathsf{knownVec}_d^m(\tau_1)[d]\\ &\quad \leq \min\{ts \mid \langle \_, \_, ts\rangle \in \mathsf{preparedCausal}_d^m(\tau_2)\} - 1\\ &\quad = \mathsf{knownVec}_d^m(\tau_2)[d].\end{aligned}$$

- Case III: $\mathsf{knownVec}_d^m(\tau_1)[d]$ is set at line A4:5 and $\mathsf{knownVec}_d^m(\tau_2)[d]$ is set at line A4:3. Let t_1 be the transaction in $\mathsf{preparedCausal}_d^m(\tau_1)$ that has the minimum ts. Formally,

$$t_1 \triangleq \operatorname*{argmin}_t \{ts \mid \langle tid(t), \_, ts\rangle \in \mathsf{preparedCausal}_d^m(\tau_1)\}.$$

By lines A4:5, A2:29, A3:15, and A4:3,

$$\begin{aligned}\mathsf{knownVec}_d^m(\tau_1)[d] &< commitVec(t_1)[d]\\ &\leq \mathsf{clock}_d^m(\tau_2)\\ &= \mathsf{knownVec}_d^m(\tau_2)[d].\end{aligned}$$

- Case IV: Both $\mathsf{knownVec}_d^m(\tau_1)[d]$ and $\mathsf{knownVec}_d^m(\tau_2)[d]$ are set at line A4:5. By lines A4:5 and A3:11,

$$\begin{aligned}&\mathsf{knownVec}_d^m(\tau_1)[d]\\ &\quad = \min\{ts \mid \langle \_, \_, ts\rangle \in \mathsf{preparedCausal}_d^m(\tau_1)\} - 1\\ &\quad \leq \min\{ts \mid \langle \_, \_, ts\rangle \in \mathsf{preparedCausal}_d^m(\tau_2)\} - 1\\ &\quad = \mathsf{knownVec}_d^m(\tau_2)[d].\end{aligned}$$

$\square$

Lemma 14. For $i \in \mathcal{D} \setminus \{d\}$, $\mathsf{knownVec}_d^m[i]$ at any replica p_d^m in data center d is non-decreasing.

Proof. Note that $\mathsf{knownVec}_d^m[i]$ ($i \in \mathcal{D} \setminus \{d\}$) can be updated only at lines A4:18 and A4:21. Therefore, this lemma holds due to lines A4:14 and A4:20. $\square$

Lemma 15. For $i \in \mathcal{D}$, $\mathsf{knownVec}_d^m[i]$ at any replica p_d^m in data center d is non-decreasing.

Proof. By Lemmas 13 and 14. $\square$

Lemma 16. For any replica p_d^m in data center d,

$$\mathsf{knownVec}_d^m[d] \leq \mathsf{clock}_d^m.$$

Proof. Note that $\mathsf{knownVec}_d^m[d]$ is updated only at lines A4:3 or A4:5.

- CASE I: $\mathsf{knownVec}_d^m[d]$ is updated at line A4:3. By Assumption 1,
$$\mathsf{knownVec}_d^m[d] \le \mathsf{clock}_d^m.$$
- CASE II: $\mathsf{knownVec}_d^m[d]$ is updated at line A4:5. By line A3:11, immediately after this update,
$$\mathsf{knownVec}_d^m[d] < \mathsf{clock}_d^m.$$

□

Lemma 17. Let p_d^m be a replica in data center d. Consider $\mathsf{knownVec}_d^m(\tau)[d]$ at time τ and transaction $t \in T_{causal}$ committed at p_d^m after time τ. Then
$$commitVec(t)[d] > \mathsf{knownVec}_d^m(\tau)[d].$$

Proof. Suppose that before time τ, $\mathsf{knownVec}_d^m[d]$ is last updated at time $\tau' < \tau$. Therefore,
$$\mathsf{knownVec}_d^m(\tau)[d] = \mathsf{knownVec}_d^m(\tau')[d].$$
We distinguish between two cases according to whether
$$\mathsf{preparedCausal}_d^m(\tau') = \varnothing$$
when $\mathsf{knownVec}_d^m[d]$ is updated at time τ'.

- CASE I: $\mathsf{preparedCausal}_d^m(\tau') = \varnothing$. By line A4:3,
$$\mathsf{knownVec}_d^m(\tau')[d] = \mathsf{clock}_d^m(\tau').$$
By line A3:11, line A2:29, and Assumption 1,
$$commitVec(t)[d] > \mathsf{clock}_d^m(\tau').$$
Therefore,
$$\begin{aligned} commitVec(t)[d] &> \mathsf{knownVec}_d^m(\tau')[d] \\ &= \mathsf{knownVec}_d^m(\tau)[d]. \end{aligned}$$
- CASE II: $\mathsf{preparedCausal}_d^m(\tau') \neq \varnothing$. We further distinguish between two cases according to whether
$$\langle tid(t), \_, \_ \rangle \in \mathsf{preparedCausal}_d^m(\tau').$$
  - CASE II-1: $\langle tid(t), \_, ts \rangle \in \mathsf{preparedCausal}_d^m(\tau')$. By lines A4:5 and A2:29,
$$\begin{aligned} commitVec(t)[d] &\ge ts \\ &> \mathsf{knownVec}_d^m(\tau')[d] \\ &= \mathsf{knownVec}_d^m(\tau)[d]. \end{aligned}$$
  - CASE II-2: $\langle tid(t), \_, \_ \rangle \notin \mathsf{preparedCausal}_d^m(\tau')$. By Lemma 16, Assumption 1, line A3:11, and line A2:29,
$$\begin{aligned} commitVec(t)[d] &> \mathsf{knownVec}_d^m(\tau')[d] \\ &= \mathsf{knownVec}_d^m(\tau)[d]. \end{aligned}$$

□

Lemma 18. Let $t \in T_{causal}$ be a causal transaction that originates at data center d and accesses partition m. If
$$commitVec(t)[d] \le \mathsf{knownVec}_d^m[d],$$
then
$$log(t)[m] \subseteq \mathsf{opLog}_d^m.$$

Proof. Suppose that the value $\mathsf{knownVec}_d^m[d]$ is set at time τ. By Lemma 17, t is committed at p_d^m before time τ. Therefore, by line A3:19,
$$log(t)[m] \subseteq \mathsf{opLog}_d^m.$$

□

The following lemmas consider the replication and forwarding of causal transactions.

Lemma 19. Let p_d^m be a replica in data center d. Let t_1 and t_2 be two transactions replicated by p_d^m to sibling replicas at time τ_1 and τ_2 (line A4:8), respectively. Then
$$\tau_1 < \tau_2 \Rightarrow commitVec(t_1)[d] < commitVec(t_2)[d].$$

Proof. Since t_1 is replicated at time τ_1, by line A4:6,
$$commitVec(t_1)[d] \le \mathsf{knownVec}_d^m(\tau_1)[d].$$
Assume that $\tau_1 < \tau_2$. We distinguish between two cases according to whether
$$\langle tid(t_2), \_, \_, \_ \rangle \in \mathsf{committedCausal}_d^m(\tau_1)[d].$$

- CASE I: $\langle tid(t_2), \_, \_, \_ \rangle \in \mathsf{committedCausal}_d^m(\tau_1)[d]$. Since t_2 is not replicated at time τ_1, by line A4:6,
$$commitVec(t_2)[d] > \mathsf{knownVec}_d^m(\tau_1)[d].$$
- CASE II: $\langle tid(t_2), \_, \_, \_, \rangle \notin \mathsf{committedCausal}_d^m(\tau_1)[d]$. Thus, t_2 is committed at p_d^m after time τ_1. By Lemma 17,
$$commitVec(t_2)[d] > \mathsf{knownVec}_d^m(\tau_1)[d].$$

Therefore, in either case,
$$commitVec(t_1)[d] < commitVec(t_2)[d].$$

□

Lemma 20. Let p_d^m be a replica in data center d. Consider a heartbeat $\mathsf{knownVec}_d^m(\tau_1)[d]$ sent by p_d^m at time τ_1 (line A4:11). Let t be a transaction replicated by p_d^m at time τ_2 (line A4:8). Then
$$\tau_1 < \tau_2 \Leftrightarrow \mathsf{knownVec}_d^m(\tau_1)[d] < commitVec(t)[d].$$

Proof. We first show that

$$\tau_1 < \tau_2 \Rightarrow \mathsf{knownVec}_d^m(\tau_1)[d] < \mathit{commitVec}(t)[d].$$

Assume that $\tau_1 < \tau_2$. We distinguish between two cases according to whether

$$\langle \mathit{tid}(t), \_, \_, \_ \rangle \in \mathsf{committedCausal}_d^m(\tau_1)[d].$$

- CASE I: $\langle \mathit{tid}(t), \_, \_, \_ \rangle \in \mathsf{committedCausal}_d^m(\tau_1)[d]$. By line A4:6,

$$\mathit{commitVec}(t)[d] > \mathsf{knownVec}_d^m(\tau_1)[d].$$

- CASE II: $\langle \mathit{tid}(t), \_, \_, \_ \rangle \notin \mathsf{committedCausal}_d^m(\tau_1)[d]$. Thus, t is committed at p_d^m after time τ_1. By Lemma 17,

$$\mathit{commitVec}(t)[d] > \mathsf{knownVec}_d^m(\tau_1)[d].$$

Next we show that (note that $\tau_1 \neq \tau_2$)

$$\tau_2 < \tau_1 \Rightarrow \mathit{commitVec}(t)[d] \le \mathsf{knownVec}_d^m(\tau_1)[d].$$

Since t is replicated by p_d^m at time τ_2, by line A4:6,

$$\mathit{commitVec}(t)[d] \le \mathsf{knownVec}_d^m(\tau_2)[d].$$

Assume that $\tau_2 < \tau_1$. By Lemma 13,

$$\mathsf{knownVec}_d^m(\tau_2)[d] \le \mathsf{knownVec}_d^m(\tau_1)[d].$$

Putting it together yields

$$\mathit{commitVec}(t)[d] \le \mathsf{knownVec}_d^m(\tau_1)[d].$$

□

Lemma 21. Let p_d^m be a replica in data center d. Then

$$\forall i \neq d.\, \forall \langle \mathit{tid}(t), \_, \_, \_ \rangle \in \mathsf{committedCausal}_d^m[i].$$
$$\mathit{commitVec}(t)[i] \le \mathsf{knownVec}_d^m[i].$$

Proof. By lines A4:17 and A4:18 and Lemma 14. □

Lemma 22. For $j \neq d$ and $i \notin \{d, j\}$, $\mathsf{globalMatrix}_d^m[i][j]$ at any replica p_d^m in data center d is non-decreasing.

Proof. Note that $\mathsf{globalMatrix}_d^m[i][j]$ can be updated only at line A5:17. Therefore, by Lemma 14, it is non-decreasing. □

Lemma 23. Let p_d^m be a replica in data center d. Let t_1 and t_2 be two transactions that originate at data center $j \neq d$ and are forwarded by p_d^m to sibling replica p_i^m in data center $i \notin \{d, j\}$ at time τ_1 and τ_2 (line A4:25), respectively. Then

$$\tau_1 < \tau_2 \Rightarrow \mathit{commitVec}(t_1)[j] < \mathit{commitVec}(t_2)[j].$$

Proof. Since t_1 is forwarded by p_d^m at time τ_1, by line A4:23,

$$\langle \mathit{tid}(t_1), \_, \_, \_ \rangle \in \mathsf{committedCausal}_d^m(\tau_1)[j].$$

By Lemmas 21 and 14,

$$\mathit{commitVec}(t_1)[j] \le \mathsf{knownVec}_d^m(\tau_1)[j]. \tag{8}$$

Assume that $\tau_1 < \tau_2$. We first argue that

$$\langle \mathit{tid}(t_2), \_, \_, \_ \rangle \notin \mathsf{committedCausal}_d^m(\tau_1)[j]. \tag{9}$$

Otherwise, by line A4:23,

$$\mathit{commitVec}(t_2)[j] \le \mathsf{globalMatrix}_d^m(\tau_1)[i][j].$$

By Lemma 22,

$$\mathit{commitVec}(t_2)[j] \le \mathsf{globalMatrix}_d^m(\tau_2)[i][j].$$

Therefore, by line A4:23, t_2 would not be forwarded by p_d^m to p_i^m at time τ_2. Thus, (9) holds. Since t_2 is forwarded by p_d^m to p_i^m at time τ_2,

$$\langle \mathit{tid}(t_2), \_, \_ \rangle \in \mathsf{committedCausal}_d^m(\tau_2)[j].$$

By Lemma 14 and line A4:14,

$$\mathit{commitVec}(t_2)[j] > \mathsf{knownVec}_d^m(\tau_1)[j]. \tag{10}$$

Putting (8) and (10) together yields

$$\mathit{commitVec}(t_1)[j] < \mathit{commitVec}(t_2)[j].$$

□

Lemma 24. Let p_d^m be a replica in data center d. Consider a heartbeat $\mathsf{knownVec}_d^m(\tau_1)[j]$ ($j \neq d$) sent by p_d^m to sibling replica p_i^m in data center $i \notin \{d, j\}$ at time τ_1 (line A4:27). Let t be a transaction that originates at data center j and is forwarded by p_d^m to p_i^m at time τ_2 (line A4:25). Then

$$\tau_1 < \tau_2 \Leftrightarrow \mathsf{knownVec}_d^m(\tau_1)[j] < \mathit{commitVec}(t)[j].$$

Proof. We first show that

$$\tau_1 < \tau_2 \Rightarrow \mathsf{knownVec}_d^m(\tau_1)[j] < \mathit{commitVec}(t)[j].$$

Assume that $\tau_1 < \tau_2$. We first argue that

$$\langle \mathit{tid}(t), \_, \_, \_ \rangle \notin \mathsf{committedCausal}_d^m(\tau_1)[j]. \tag{11}$$

Otherwise, since t is not forwarded at time τ_1, by line A4:23,

$$\mathit{commitVec}(t)[j] \le \mathsf{globalMatrix}_d^m(\tau_1)[i][j].$$

By Lemma 22,

$$\mathit{commitVec}(t)[j] \le \mathsf{globalMatrix}_d^m(\tau_2)[i][j].$$

Therefore, by line A4:23, t would not be forwarded by p_d^m to p_i^m at time τ_2. Thus, (11) holds. Since t is forwarded by p_d^m to p_i^m at time τ_2,

$$\langle tid(t), \_, \_, \_\rangle \in \mathsf{committedCausal}_d^m(\tau_2)[j].$$

By Lemma 14 and line A4:14,

$$\mathsf{knownVec}_d^m(\tau_1)[j] < commitVec(t)[j].$$

Next we show that (note that $\tau_1 \neq \tau_2$)

$$\tau_2 < \tau_1 \Rightarrow commitVec(t)[j] \leq \mathsf{knownVec}_d^m(\tau_1)[j].$$

Since t is forwarded by p_d^m to p_i^m at time τ_2, by line A4:23,

$$\langle tid(t), \_, \_, \_\rangle \in \mathsf{committedCausal}_d^m(\tau_2)[j].$$

By Lemmas 21 and 14,

$$commitVec(t)[j] \leq \mathsf{knownVec}_d^m(\tau_2)[j].$$

Assume that $\tau_2 < \tau_1$. By Lemma 14,

$$\mathsf{knownVec}_d^m(\tau_2)[j] \leq \mathsf{knownVec}_d^m(\tau_1)[j].$$

Putting it together yields

$$commitVec(t)[j] \leq \mathsf{knownVec}_d^m(\tau_1)[j].$$

□

Lemma 25. Let $t \in T_{causal}$ be a causal transaction that originates at data center i and accesses partition m. If

$$commitVec(t)[i] \leq \mathsf{knownVec}_d^m[i]$$

for replica p_d^m in data center $d \neq i$, then

$$log(t)[m] \subseteq \mathsf{opLog}_d^m.$$

Proof. Note that for $i \in \mathcal{D} \setminus \{d\}$, $\mathsf{knownVec}_d^m[i]$ can be updated only at lines A4:18 or A4:21 due to replication of transactions or heartbeats respectively, either directly from data center i (line A4:1) or indirectly from a third data center $j \neq i$ (line A4:22).

We proceed by induction on the length of the execution. In the following, for replica p_d^m in data center $d \in \mathcal{D}$, we denote the value of $\mathsf{knownVec}_d^m$ (resp. opLog_d^m) after k steps in an execution by $\mathsf{knownVec}_d^m(k)$ (resp. $\mathsf{opLog}_d^m(k)$).

- *Base Case.* $k = 0$. It holds trivially, since for replica p_d^m in any data center $d \neq i$,

$$\mathsf{knownVec}_d^m(0)[i] = 0.$$

- *Induction Hypothesis.* Suppose that for any execution of length k, we have

$$\forall d \in \mathcal{D} \setminus \{i\}.\ \forall t \in T_{causal}.\ \big(commitVec(t)[i] \leq \mathsf{knownVec}_d^m(k)[i] \Rightarrow log(t)[m] \subseteq \mathsf{opLog}_d^m(k)\big).$$

- *Induction Step.* Consider an execution of length $k+1$. If the $(k+1)$-st step of this execution does not update $\mathsf{knownVec}_d^m[i]$ for replica p_d^m in any data center $d \neq i$, then by the induction hypothesis,

$$\forall d \in \mathcal{D} \setminus \{i\}.\ \forall t \in T_{causal}.\ \big(commitVec(t)[i] \leq \mathsf{knownVec}_d^m(k+1)[i] \Rightarrow log(t)[m] \subseteq \mathsf{opLog}_d^m(k+1)\big).$$

  Otherwise, we perform a case analysis according to how $\mathsf{knownVec}_d^m[i]$ of replica p_d^m in data center $d \neq i$ is updated in the $(k+1)$-st step.
  - CASE I: $\mathsf{knownVec}_d^m[i]$ is updated due to delivery of a message from data center i. By Lemmas 13, 19, and 20, local transactions and heartbeats are propagated by p_i^m to sibling replicas in increasing order of their local timestamps $commitVec[i]$ and $\mathsf{knownVec}_i^m[i]$ values. Therefore, by Assumption 2 and the induction hypothesis,

$$\forall t \in T_{causal}.\ \big(commitVec(t)[i] \leq \mathsf{knownVec}_d^m(k+1)[i] \Rightarrow log(t)[m] \subseteq \mathsf{opLog}_d^m(k+1)\big).$$

  - CASE II: $\mathsf{knownVec}_d^m[i]$ is updated due to delivery of a message from a third data center $j \neq i$. By Lemmas 14, 23, and 24, transactions originating at data center i and heartbeats are forwarded by some replica, say $p_j^m (j \neq i)$, to sibling replicas in increasing order of their local timestamps $commitVec[i]$ and $\mathsf{knownVec}_j^m[i]$ values. Therefore, by Assumption 2 and the induction hypothesis,

$$\forall t \in T_{causal}.\ \big(commitVec(t)[i] \leq \mathsf{knownVec}_d^m(k+1)[i] \Rightarrow log(t)[m] \subseteq \mathsf{opLog}_d^m(k+1)\big).$$

□

Lemma 26 (PROPERTY 1). Let $t \in T_{causal}$ be a causal transaction that originates at data center i and accesses partition m. If

$$commitVec(t)[i] \leq \mathsf{knownVec}_d^m[i]$$

for replica p_d^m in data center d, then

$$log(t)[m] \subseteq \mathsf{opLog}_d^m.$$

Proof. By Lemmas 18 and 25. □

D.3.2 Properties of stableVec

Lemma 27. For $i \in \mathcal{D}$, $\mathsf{stableVec}_d^m[i]$ at any replica p_d^m in data center d is non-decreasing.

Proof. Note that $\mathsf{stableVec}_d^m[i]$ ($i \in \mathcal{D}$) can be updated only at line A5:8. By Lemma 15 and Assumption 2, $\mathsf{stableVec}_d^m[i]$ is non-decreasing. $\square$

Lemma 28 (**PROPERTY 2**)**.** For any replica p_d^m in data center d,

$$\forall i \in \mathcal{D}.\ \forall n \in \mathcal{P}.\ \mathsf{stableVec}_d^m[i] \leq \mathsf{knownVec}_d^n[i].$$

Proof. Note that $\mathsf{stableVec}_d^m[i]$ ($i \in \mathcal{D}$) can be updated only at line A5:8. By the way $\mathsf{stableVec}_d^m[i]$ is updated and Lemmas 13 and 14,

$$\forall n \in \mathcal{P}.\ \mathsf{stableVec}_d^m[i] \leq \mathsf{knownVec}_d^n[i].$$

$\square$

Lemma 29. Let $t \in T_{causal}$ be a causal transaction that originates at data center i and accesses partition n. If

$$commitVec(t)[i] \leq \mathsf{stableVec}_d^m[i]$$

for some replica p_d^m in data center d, then

$$log(t)[n] \subseteq \mathsf{opLog}_d^n.$$

Proof. By Lemma 28,

$$\mathsf{stableVec}_d^m[i] \leq \mathsf{knownVec}_d^n[i].$$

Therefore,

$$commitVec(t)[i] \leq \mathsf{knownVec}_d^n[i].$$

By Lemma 25,

$$log(t)[n] \subseteq \mathsf{opLog}_d^n.$$

$\square$

D.3.3 Properties of $\mathsf{uniformVec}$

Lemma 30. For $i \in \mathcal{D}$, $\mathsf{uniformVec}_d^m[i]$ at any replica p_d^m in data center d is non-decreasing.

Proof. Note that whenever $\mathsf{uniformVec}_d^m[i]$ is updated at lines A2:3, A3:3, A3:10, or A5:15, we take the maximum of it and some scalar value. $\square$

Lemma 31. Let $e \in E$ be an event issued by client cl to replica p_d^m in data center d. Then

$$e \in E \setminus Q \Rightarrow \\ \forall i \in \mathcal{D} \setminus \{d\}.\ (\mathsf{pastVec}_{cl})_e[i] \leq (\mathsf{uniformVec}_d^m)_e[i],$$

and

$$e \in Q \Rightarrow (\mathsf{pastVec}_{cl})_e[d] \leq (\mathsf{uniformVec}_d^m)_e[d].$$

Proof. We perform a case analysis according to the type of event e.

- CASE I: $e \in S$. By line A2:3,

$$\forall i \in \mathcal{D} \setminus \{d\}.\ (\mathsf{pastVec}_{cl})_e[i] \leq (\mathsf{uniformVec}_d^m)_e[i].$$

- CASE II: $e \in R \cup U$. In this case,

$$(\mathsf{pastVec}_{cl})_e = (\mathsf{pastVec}_{cl})_{st(e)}.$$

By CASE I,

$$\forall i \in \mathcal{D} \setminus \{d\}.\ (\mathsf{pastVec}_{cl})_{st(e)}[i] \leq (\mathsf{uniformVec}_d^m)_{st(e)}[i].$$

By Lemma 30,

$$\forall i \in \mathcal{D} \setminus \{d\}. (\mathsf{uniformVec}_d^m)_{st(e)} \leq (\mathsf{uniformVec}_d^m)_e.$$

Putting it together yields

$$\forall i \in \mathcal{D} \setminus \{d\}.\ (\mathsf{pastVec}_{cl})_e[i] \leq (\mathsf{uniformVec}_d^m)_e[i].$$

- CASE III: $e \in C_{causal}$. By line A1:16,

$$(\mathsf{pastVec}_{cl})_e = vc_{(\text{COMMIT\_CAUSAL\_TX},e)}.$$

By lines A2:24, A2:26, and A2:31,

$$\forall i \in \mathcal{D} \setminus \{d\}. \\ vc_{(\text{COMMIT\_CAUSAL\_TX},e)}[i] = (\mathsf{snapVec}_d^m)_e[tx(e)][i].$$

By line A2:5,

$$\forall i \in \mathcal{D} \setminus \{d\}. \\ (\mathsf{snapVec}_d^m)_e[tx(e)][i] = (\mathsf{uniformVec}_d^m)_{st(e)}[i].$$

By Lemma 30,

$$\forall i \in \mathcal{D} \setminus \{d\}. (\mathsf{uniformVec}_d^m)_{st(e)} \leq (\mathsf{uniformVec}_d^m)_e.$$

Putting it together yields

$$\forall i \in \mathcal{D} \setminus \{d\}.\ (\mathsf{pastVec}_{cl})_e[i] \leq (\mathsf{uniformVec}_d^m)_e[i].$$

- CASE IV: $e \in C_{strong}$. By line A1:22,

$$(\mathsf{pastVec}_{cl})_e = vc_{(\text{COMMIT\_STRONG\_TX},e)}.$$

By (5),

$$\forall i \in \mathcal{D} \setminus \{d\}. \\ vc_{(\text{COMMIT\_STRONG\_TX},e)}[i] = (\mathsf{snapVec}_d^m)_e[tx(e)][i].$$

Therefore, similar to CASE III, we have

$$\forall i \in \mathcal{D} \setminus \{d\}.\ (\mathsf{pastVec}_{cl})_e[i] \leq (\mathsf{uniformVec}_d^m)_e[i].$$

- CASE V: $e \in Q$. By line A3:22,

$$(\mathsf{pastVec}_{cl})_e[d] \leq (\mathsf{uniformVec}_d^m)_e[d].$$

- CASE VI: $e \in A$. By line A3:24,

$$\forall i \in \mathcal{D} \setminus \{d\}.\ (\mathsf{pastVec}_{cl})_e[i] \leq (\mathsf{uniformVec}_d^m)_e[i].$$

$\square$

Lemma 32. Let cl be a client and $d \triangleq cur\_dc(cl)$. At any time,

$$\forall i \in \mathcal{D} \setminus \{d\}.\ \mathsf{pastVec}_{cl}[i] \leq \mathsf{uniformVec}_d^m[i]$$

for some replica p_d^m in data center d.

Proof. By a simple induction on the number of events that cl issues and Lemmas 31 and 30. $\square$

Lemma 33. Let t be a transaction that originates at data center d. At any time,

$$\forall i \in \mathcal{D} \setminus \{d\}.\ snapshotVec(t)[i] \leq \mathsf{uniformVec}_d^m[i]$$

for some replica p_d^m in data center d.

Proof. By line A2:5 and Lemma 30. $\square$

Lemma 34 (PROPERTY 3). For any replica p_d^m in data center d,

$$\forall i \in \mathcal{D}.\ \exists g \subseteq \mathcal{D}.\ \Big(|g| \geq f+1 \land d \in g \land \big(\forall j \in g.\ \forall n \in \mathcal{P}.\ \mathsf{uniformVec}_d^m[i] \leq \mathsf{knownVec}_j^n[i]\big)\Big).$$

Proof. Fix $i \in \mathcal{D}$. We proceed by induction on the length of the execution. In the following, we denote the value of $\mathsf{knownVec}_d^m$, $\mathsf{stableVec}_d^m$, $\mathsf{uniformVec}_d^m$, $\mathsf{stableMatrix}_d^m$, and $\mathsf{pastVec}_{cl}$ (for some client cl) after k steps of an execution by $\mathsf{knownVec}_d^m(k)$, $\mathsf{stableVec}_d^m(k)$, $\mathsf{uniformVec}_d^m(k)$, $\mathsf{stableMatrix}_d^m(k)$, and $\mathsf{pastVec}_{cl}(k)$, respectively.

- *Base Case.* $k = 0$. It holds trivially since

$$\mathsf{uniformVec}_d^m(0)[i] = 0.$$

- *Induction Hypothesis.* Suppose that for any execution of length k, for any replica p_d^m in data center d,

$$\exists g \subseteq \mathcal{D}.\ |g| \geq f+1 \land d \in g \land \big(\forall j \in g.\ \forall 1 \leq n \leq N.\ \mathsf{uniformVec}_d^m(k)[i] \leq \mathsf{knownVec}_j^n(k)[i]\big).$$

- *Induction Step.* Consider an execution of length $k+1$. If the $(k+1)$-st step of this execution does not update $\mathsf{uniformVec}_d^m[i]$ for any replica p_d^m in data center d, then by the induction hypothesis and Lemma 15,

$$\exists g \subseteq \mathcal{D}.\ |g| \geq f+1 \land d \in g \land \big(\forall j \in g.\ \forall 1 \leq n \leq N.\ \mathsf{uniformVec}_d^m(k+1)[i] = \mathsf{uniformVec}_d^m(k)[i] \leq \mathsf{knownVec}_j^n(k)[i] \leq \mathsf{knownVec}_j^n(k+1)[i]\big).$$

Otherwise, we perform a case analysis according to how $\mathsf{uniformVec}_d^m[i]$ is updated.

  - CASE I: $\mathsf{uniformVec}_d^m[i]$ is updated at line A5:15. By line A5:12,

$$\exists g' \subseteq \mathcal{D}.\ |g'| \geq f+1 \land d \in g' \land \mathsf{uniformVec}_d^m(k+1)[i] = \max\big\{\mathsf{uniformVec}_d^m(k)[i], \min_{j \in g'} \mathsf{stableMatrix}_d^m(k+1)[j][i]\big\}. \tag{12}$$

By the induction hypothesis and Lemma 14,

$$\exists g'' \subseteq \mathcal{D}.\ |g''| \geq f+1 \land d \in g'' \land \big(\forall j \in g''.\ \forall n \in \mathcal{P}.\ \mathsf{uniformVec}_d^m(k)[i] \leq \mathsf{knownVec}_j^n(k)[i] \leq \mathsf{knownVec}_j^n(k+1)[i]\big). \tag{13}$$

By Lemma 27, for the particular $g' \subseteq \mathcal{D}$ in (12),

$$\forall j \in g'.\ \mathsf{stableMatrix}_d^m(k+1)[j][i] \leq \mathsf{stableVec}_j^m(k+1)[i]. \tag{14}$$

By Lemmas 28 and 14, for any replica p_j^m in data center j,

$$\forall n \in \mathcal{P}.\ \mathsf{stableVec}_j^m(k+1)[i] \leq \mathsf{knownVec}_j^n(k+1)[i]. \tag{15}$$

Therefore, for the particular $g' \subseteq \mathcal{D}$ in (12),

$$\forall j' \in g'.\ \forall n \in \mathcal{P}.\ \min_{j \in g'} \mathsf{stableMatrix}_d^m(k+1)[j][i] \leq \mathsf{knownVec}_{j'}^n(k+1)[i]. \tag{16}$$

By (12), (13), and (16), we can either take $g = g'$ in (12) or $g = g''$ in (13) such that

$$\forall j \in g.\ \forall n \in \mathcal{P}.\ \mathsf{uniformVec}_d^m(k+1)[i] \leq \mathsf{knownVec}_j^n(k+1)[i].$$

Therefore,

$$\exists g \subseteq \mathcal{D}.\ |g| \geq f+1 \land d \in g \land \big(\forall j \in g.\ \forall n \in \mathcal{P}.\ \mathsf{uniformVec}_d^m(k+1)[i] \leq \mathsf{knownVec}_j^n(k+1)[i]\big).$$

  - CASE II: $\mathsf{uniformVec}_d^m[i]$ ($i \in \mathcal{D} \setminus \{d\}$) is updated at line A2:3. Then there exists some client cl with $d = cur\_dc(cl)$ such that

$$\mathsf{uniformVec}_d^m(k+1)[i] = \max\big\{\mathsf{pastVec}_{cl}(k)[i], \mathsf{uniformVec}_d^m(k)[i]\big\}. \tag{17}$$

By the induction hypothesis and Lemma 14,

$$\exists g' \subseteq \mathcal{D}.\ |g'| \geq f+1 \land d \in g' \land \big(\forall j \in g'.\ \forall n \in \mathcal{P}.\ \mathsf{uniformVec}_d^m(k)[i] \leq \mathsf{knownVec}_j^n(k)[i] \leq \mathsf{knownVec}_j^n(k+1)[i]\big). \tag{18}$$

By Lemma 32, the induction hypothesis, and Lemma 15,

$$\exists g'' \subseteq \mathcal{D}.\ |g''| \geq f+1 \land d \in g'' \land \quad (19)$$
$$\big(\forall j \in g''.\ \forall n \in \mathcal{P}.$$
$$\mathsf{pastVec}_{cl}(k)[i] \leq \mathsf{knownVec}_j^n(k+1)[i].$$

By (17), (18), and (19), we can take $g = g'$ in (18) or $g = g''$ in (19) such that

$$\forall j \in g.\ \forall n \in \mathcal{P}.$$
$$\mathsf{uniformVec}_d^m(k+1)[i] \leq \mathsf{knownVec}_j^n(k+1)[i].$$

Therefore,

$$\exists g \subseteq \mathcal{D}.\ |g| \geq f+1 \land d \in g \land$$
$$\big(\forall j \in g.\ \forall n \in \mathcal{P}.$$
$$\mathsf{uniformVec}_d^m(k+1)[i] \leq \mathsf{knownVec}_j^n(k+1)[i]\big).$$

- CASE III: $\mathsf{uniformVec}_d^m[i]$ $(i \in \mathcal{D} \setminus \{d\})$ is updated at lines A3:3 or A3:10. Therefore, there exists some transaction t originating at data center d such that

$$\mathsf{uniformVec}_d^m(k+1)[i] = \quad (20)$$
$$\max\left\{\mathit{snapshotVec}(t)[i], \mathsf{uniformVec}_d^m(k)[i]\right\}.$$

By the induction hypothesis and Lemma 15,

$$\exists g' \subseteq \mathcal{D}.\ |g'| \geq f+1 \land d \in g' \land \quad (21)$$
$$\big(\forall j \in g'.\ \forall n \in \mathcal{P}.$$
$$\mathsf{uniformVec}_d^m(k)[i] \leq \mathsf{knownVec}_j^n(k)[i]$$
$$\leq \mathsf{knownVec}_j^n(k+1)[i]\big).$$

By Lemma 33, the induction hypothesis, and Lemma 15,

$$\exists g'' \subseteq \mathcal{D}.\ |g''| \geq f+1 \land d \in g'' \land \quad (22)$$
$$\big(\forall j \in g''.\ \forall n \in \mathcal{P}.$$
$$\mathit{snapshotVec}(t)[i] \leq \mathsf{knownVec}_j^n(k+1)[i].$$

By (20), (21), and (22), we can take $g = g'$ in (21) or $g = g''$ in (22) such that

$$\forall j \in g.\ \forall n \in \mathcal{P}.$$
$$\mathsf{uniformVec}_d^m(k+1)[i] \leq \mathsf{knownVec}_j^n(k+1)[i].$$

Therefore,

$$\exists g \subseteq \mathcal{D}.\ |g| \geq f+1 \land d \in g \land$$
$$\big(\forall j \in g.\ \forall n \in \mathcal{P}.$$
$$\mathsf{uniformVec}_d^m(k+1)[i] \leq \mathsf{knownVec}_j^n(k+1)[i]\big).$$

□

Lemma 35. For any replica p_d^m in data center d,

$$\forall i \in \mathcal{D}.\ \forall n \in \mathcal{P}.\ \mathsf{uniformVec}_d^m[i] \leq \mathsf{knownVec}_d^n[i].$$

Proof. By Lemma 34. □

Lemma 36. Let $t \in T_{causal}$ be a causal transaction that originates at data center i and accesses partition n. If

$$\mathit{commitVec}(t)[i] \leq \mathsf{uniformVec}_d^m[i]$$

for some replica p_d^m in data center d, then

$$\mathit{log}(t)[n] \subseteq \mathsf{opLog}_d^n.$$

Proof. By Lemma 35,

$$\mathsf{uniformVec}_d^m[i] \leq \mathsf{knownVec}_d^n[i].$$

Therefore,

$$\mathit{commitVec}(t)[i] \leq \mathsf{knownVec}_d^n[i].$$

By Lemma 25,

$$\mathit{log}(t)[n] \subseteq \mathsf{opLog}_d^n.$$

□

Lemma 37. For any replica p_d^m in data center d,

$$\mathsf{uniformVec}_d^m[d] \leq \mathsf{clock}_d^m.$$

Proof. By Lemma 35,

$$\mathsf{uniformVec}_d^m[d] \leq \mathsf{knownVec}_d^m[d].$$

By Lemma 16,

$$\mathsf{knownVec}_d^m[d] \leq \mathsf{clock}_d^m.$$

Putting it together yields

$$\mathsf{uniformVec}_d^m[d] \leq \mathsf{clock}_d^m.$$

□

D.3.4 Properties of pastVec

Lemma 38. Let $e \in S$ be a START event of transaction t issued by client cl. Then

$$(\mathsf{pastVec}_{cl})_e \leq \mathit{snapshotVec}(t).$$

Proof. By Definition 10 of $\mathit{snapshotVec}(t)$ and lines A2:2–A2:7. □

Lemma 39. For $i \in \mathcal{D}$, $\mathsf{pastVec}_{cl}[i]$ at any client cl is non-decreasing.

Proof. Note that $\mathsf{pastVec}_{cl}[i]$ $(i \in \mathcal{D})$ is updated only at lines A1:16 or A1:22 when some transaction is committed. Therefore, the lemma holds due to Lemmas 12 and 38. □

D.4 Metadata for Strong Transactions

Lemma 40. For any replica p_d^m in any data center d, $\mathsf{knownVec}_d^m[strong]$ is non-decreasing.

Proof. By (R6) and line A6:6. □

Lemma 41. For any replica p_d^m in any data center d, $\mathsf{stableVec}_d^m[strong]$ is non-decreasing.

Proof. By Lemma 40, Assumption 2, and line A5:9. □

Lemma 42 (PROPERTY 6). Let $t \in T_{strong}$ be a strong transaction that originates at data center i and accesses partition m. If

$$commitVec(t)[strong] \le \mathsf{knownVec}_d^m[strong]$$

for some replica p_d^m in data center d, then

$$log(t)[m] \subseteq \mathsf{opLog}_d^m.$$

Proof. Note that $\mathsf{knownVec}_d^m[strong]$ can be updated only at line A6:9. By (R6), all committed strong transactions with strong timestamps less than or equal to $\mathsf{knownVec}_d^m[strong]$ have been delivered to p_d^m. By line A6:8,

$$log(t)[m] \subseteq \mathsf{opLog}_d^m.$$

□

D.5 Timestamps

Definition 43 (Timestamps of Events)**.** Let $e \in C_{causal} \cup C_{strong} \cup Q \cup A$ be an event issued by client cl. We define its timestamp $ts(e)$ as

$$ts(e) \triangleq (\mathsf{pastVec}_{cl})_e.$$

Let $e \in S$ be a START event of transaction t. Let $d \triangleq dc(t)$ and $m \triangleq coord(t)$. We define its timestamp $ts(e)$ as

$$ts(e) \triangleq (\mathsf{snapVec}_d^m)_e[t].$$

See lines A1:3, A1:16, A1:22, A1:27, and A1:32 for START, COMMIT_CAUSAL_TX, COMMIT_STRONG_TX, CL_-UNIFORM_BARRIER, and CL_ATTACH events, respectively.

Definition 44 (Timestamps of Transactions)**.** The timestamp $ts(t)$ of a transaction t is that of its commit event, i.e.,

$$\forall t \in T.\ ts(t) \triangleq ts(ct(t)).$$

Lemma 45. Let $e \in S$ be a START event. Let $d \triangleq dc(tx(e))$ and $m \triangleq coord(tx(e))$. Then

$$(\forall i \in \mathcal{D}.\ ts(e)[i] \ge (\mathsf{uniformVec}_d^m)_e[i]) \wedge$$
$$ts(e)[strong] \ge (\mathsf{stableVec}_d^m)_e[strong].$$

Proof. By lines A2:5, A2:6, and A2:7. □

Lemma 46. Let $e \in R_{\text{EXT}}$ be an external read event. Let $d \triangleq dc(tx(e))$ and $m \triangleq coord(tx(e))$. Then

$$\begin{aligned} ts(st(e)) &= snapshotVec(tx(e)) \\ &= snapVec_{(\text{GET\_VERSION},e)} \\ &= (\mathsf{snapVec}_d^m)_{st(e)}[tx(e)]. \end{aligned}$$

Proof. By Definition 43 of timestamps, line A2:13, and line A2:6. □

Lemma 47. Let $e \in R_{\text{EXT}}$ be an external read event which reads from transaction t. Then

$$ts(t) \le ts(st(e)).$$

Proof. By line A3:5,

$$commitVec_{(\text{GET\_VERSION},e)} \le snapVec_{(\text{GET\_VERSION},e)}.$$

Since e reads from t,

$$ts(t) = commitVec_{(\text{GET\_VERSION},e)}.$$

By Lemma 46,

$$ts(st(e)) = snapVec_{(\text{GET\_VERSION},e)}.$$

Therefore,

$$ts(t) \le ts(st(e)).$$

□

Lemma 48.

$$\begin{aligned} &\forall e \in (C_{\text{RW}} \cap C_{causal}) \cup C_{strong}. \\ &\quad ts(e) = ts(tx(e)) = commitVec(tx(e)). \end{aligned}$$

Proof. By Definition 43 of timestamps and Definition 11 of $commitVec(tx(e))$. □

Lemma 49.

$$\forall t \in T.\ ts(t) \ge ts(st(t)).$$

Proof. By Lemmas 12, 46, and 48. □

D.6 Session Order

Lemma 50.

$$\begin{aligned} &\forall s_1, s_2 \in X.\ s_1 \xrightarrow{so} s_2 \Rightarrow \big(ts(s_1) \le ts(s_2) \\ &\quad \wedge \big(s_2 \in T \Rightarrow ts(s_1) \le ts(st(s_2)) \le ts(s_2)\big)\big). \end{aligned}$$

Proof. By Definitions 43 and 44 of timestamps and Lemma 39,

$$ts(s_1) \le ts(s_2),$$

and

$$s_2 \in T \Rightarrow ts(s_1) \le ts(st(s_2)).$$

Besides, by Lemma 12,

$$s_2 \in T \Rightarrow ts(st(s_2)) \le ts(s_2).$$

Therefore,

$$s_2 \in T \Rightarrow ts(s_1) \le ts(st(s_2)) \le ts(s_2).$$

□

D.7 Lamport Clocks

Definition 51 (Lamport Clocks of Events)**.** Let $e \in C_{causal} \cup C_{strong} \cup Q \cup A$ be an event issued by client cl. We define its Lamport clock $lclock(e)$ as

$$lclock(e) \triangleq (\mathsf{lc}_{cl})_e.$$

See lines A1:14, A1:23, A1:28, and A1:33 for COMMIT_CAUSAL_TX, COMMIT_STRONG_TX, CL_UNIFORM_BARRIER, and CL_ATTACH events, respectively.

Definition 52 (Lamport Clocks of Transactions)**.** The Lamport clock $lclock(t)$ of a transaction t is that of its commit event, i.e.,

$$\forall t \in T.\ lclock(t) \triangleq lclock(ct(t)).$$

Lemma 53. Let $e \in R_{\text{EXT}}$ be an external read event issued by client cl. Then

$$(\mathsf{lc}_{cl})_e < lclock(tx(e)).$$

Proof. If $tx(e)$ is a causal transaction, by line A1:14,

$$lclock(e) < lclock(ct(e)) = lclock(tx(e)).$$

If $tx(e)$ is a strong transaction, by line A1:19 and (6),

$$lclock(e) < lclock(ct(e)) = lclock(tx(e)).$$

□

Definition 54 (Lamport Clock Order)**.** The Lamport clock order lc on X is the total order defined by their Lamport clocks, with their client identifiers for tie-breaking.

Lemma 55.

$$so \subseteq lc.$$

Proof. By lines A1:14, A1:19, (6), A1:23, A1:28, and A1:33. □

Lemma 56. Let $e \in R_{\text{EXT}}$ be an external read event which reads from transaction t. Then

$$t \xrightarrow{lc} tx(e).$$

Proof. Suppose that e is issued by client cl. By line A1:8,

$$lclock(t) \le (\mathsf{lc}_{cl})_e.$$

By Lemma 53,

$$(\mathsf{lc}_{cl})_e < lclock(tx(e)).$$

Therefore,

$$lclock(t) < lclock(tx(e)).$$

By Definition 54 of lc,

$$t \xrightarrow{lc} tx(e).$$

□

D.8 Visibility Relation

Definition 57 (Visibility Relation)**.**

$$\forall s_1, s_2 \in X.\ s_1 \xrightarrow{vis} s_2 \Leftrightarrow \Big(\big(s_2 \in T \Rightarrow ts(s_1) \le ts(st(s_2))\big) \land \big(s_2 \in Q \cup A \Rightarrow ts(s_1) \le ts(s_2)\big)\Big) \land s_1 \xrightarrow{lc} s_2.$$

Theorem 58.

$$A \models \textsc{ConflictOrdering}.$$

Proof. We need to show that

$$\forall t_1, t_2 \in T_{strong}.\ t_1 \bowtie t_2 \Rightarrow t_1 \xrightarrow{vis} t_2 \lor t_2 \xrightarrow{vis} t_1.$$

Consider the history h of TCS. By Theorem 9, $h \mid \mathsf{committed}(h)$ has a legal permutation π. Suppose that

$$\mathsf{certify}(tid(t_1), \_, \_, \_, \_) \prec_\pi \mathsf{certify}(tid(t_2), \_, \_, \_, \_).$$

Since $t_1 \bowtie t_2$ and t_2 is committed, by (4),

$$commitVec(t_1) \le snapshotVec(t_2).$$

By Lemmas 46 and 48,

$$ts(t_1) \le ts(st(t_2)).$$

On the other hand, by (6),

$$lclock(t_1) < lclock(t_2).$$

By Definition 54 of lc,

$$t_1 \xrightarrow{lc} t_2.$$

Therefore, by Definition 57 of vis,

$$t_1 \xrightarrow{vis} t_2.$$

□

Lemma 59.

$$so \subseteq vis.$$

Proof. By Lemmas 50 and 55. □

Lemma 60. The visibility relation vis is a partial order.

Proof. We need to show that

- vis is irreflexive. This holds because lc is irreflexive.
- vis is transitive. Suppose that $s_1 \xrightarrow{vis} s_2 \xrightarrow{vis} s_3$. By Definition 57 of vis,

  $$s_1 \xrightarrow{lc} s_2 \xrightarrow{lc} s_3.$$

  By Definition 54 of lc,

  $$s_1 \xrightarrow{lc} s_3.$$

  Regarding timestamps, we distinguish between the following four cases and use Lemma 49.

– $s_2 \in Q \cup A \wedge s_3 \in Q \cup A$.

$$ts(s_1) \le ts(s_2) \le ts(s_3).$$

– $s_2 \in Q \cup A \wedge s_3 \in T$.

$$ts(s_1) \le ts(s_2) \le ts(st(s_3)).$$

– $s_2 \in T \wedge s_3 \in Q \cup A$.

$$ts(s_1) \le ts(st(s_2)) \le ts(s_2) \le ts(s_3).$$

– $s_2 \in T \wedge s_3 \in T$.

$$ts(s_1) \le ts(st(s_2)) \le ts(s_2) \le ts(st(s_3)).$$

By Definition 57 of *vis*,

$$s_1 \xrightarrow{vis} s_3.$$

□

Theorem 61.

$$A \models \textsc{CausalVisibility}.$$

Proof. By Lemmas 59 and 60,

$$(so \cup vis)^+ = vis^+ = vis.$$

□

Lemma 62 (**Property 5**)**.** For any two conflicting transactions t_1 and t_2,

$$t_1 \xrightarrow{vis} t_2 \Leftrightarrow commitVec(t_1)[strong] < commitVec(t_2)[strong].$$

Proof. We first show that

$$t_1 \xrightarrow{vis} t_2 \Rightarrow commitVec(t_1)[strong] < commitVec(t_2)[strong]. \tag{23}$$

Assume that $t_1 \xrightarrow{vis} t_2$. By Definition 57 of *vis*,

$$ts(t_1) \le ts(st(t_2)).$$

By Lemmas 46 and 48,

$$commitVec(t_1) \le snapshotVec(t_2).$$

Therefore,

$$commitVec(t_1)[strong] \le snapshotVec(t_2)[strong].$$

By (5),

$$commitVec(t_2)[strong] > snapshotVec(t_2)[strong].$$

Putting it together yields

$$commitVec(t_1)[strong] < commitVec(t_2)[strong].$$

Next we show that

$$t_1 \xrightarrow{vis} t_2 \Leftarrow commitVec(t_1)[strong] < commitVec(t_2)[strong].$$

Assume that

$$commitVec(t_1)[strong] < commitVec(t_2)[strong]. \tag{24}$$

Since $t_1 \bowtie t_2$, by Theorem 58,

$$t_1 \xrightarrow{vis} t_2 \vee t_2 \xrightarrow{vis} t_1.$$

By (23) and (24),

$$\neg(t_2 \xrightarrow{vis} t_1).$$

Therefore,

$$t_1 \xrightarrow{vis} t_2.$$

□

D.9 Execution Order

Definition 63 (Execution Points)**.** Let k be a key. The "execution point" $\mathtt{ep}(e,k)$ of event $e \in (R_{\text{EXT}} \cap R_k) \cup C_k$ is defined as follows:

- If $e \in R_{\text{EXT}} \cap R_k$, then $\mathtt{ep}(e,k)$ is at line A3:5;
- If $e \in C_k \cap C_{causal}$, then $\mathtt{ep}(e,k)$ is at line A3:19 for this particular key k;
- If $e \in C_k \cap C_{strong}$ then $\mathtt{ep}(e,k)$ is at line A6:8 for delivery of the update of $tx(e)$ on this particular key k. Note that DELIVER is asynchronous with the commit event e.

Definition 64 (Per-key Execution Order)**.** Let k be a key. Suppose that $\{e_1, e_2\} \subseteq (R_{\text{EXT}} \cap R_k) \cup C_k$. Event e_1 is executed before event e_2, denoted $e_1 \xrightarrow{eo_k} e_2$, if $\mathtt{ep}(e_1,k)$ is executed before $\mathtt{ep}(e_2,k)$ in real time.

Lemma 65. Let $k \in Key$ be a key, $t \in T_k$ be a transaction, and $e \in R_{\text{EXT}} \cap R_k$ be an external read event. Suppose that $d \triangleq dc(t) = dc(tx(e))$. Then

$$t \xrightarrow{vis} tx(e) \Rightarrow ct(t) \xrightarrow{eo_k} e.$$

Proof. By Definition 57 of *vis*,

$$ts(t) \le ts(st(e)).$$

Since $e \in R_{\text{EXT}}$, by Lemma 46,

$$ts(t) \le snapVec_{(\text{GET\_VERSION},e)}.$$

In the following, we distinguish between two cases according to whether $t \in T_{causal}$ or $t \in T_{strong}$. Let $m \triangleq \mathtt{partition}(k)$.

- CASE I: $t \in T_{causal}$. By Lemma 48,

$$ts(t) = commitVec(t) \le snapVec_{(\text{GET\_VERSION},e)}.$$

Therefore, after line A3:4 for e,

$$(\mathsf{knownVec}_d^m)_e[d] \ge snapVec_{(\text{GET\_VERSION},e)}[d] \ge commitVec(t)[d]. \quad (25)$$

By Lemma 18, COMMIT of Algorithm A3 for $ws(t)[m] \ni \langle k, \_ \rangle$ finishes before e starts at replica p_d^m. By Definition 64 of eo_k,

$$ct(t) \xrightarrow{eo_k} e.$$

- CASE II: $t \in T_{strong}$. By Lemma 48,

$$ts(t) = commitVec(t) \le snapVec_{(\text{GET\_VERSION},e)}.$$

Therefore, after line A3:4 for e,

$$(\mathsf{knownVec}_d^m)_e[strong] \ge snapVec_{(\text{GET\_VERSION},e)}[strong] \ge commitVec(t)[strong]. \quad (26)$$

By Lemma 42, DELIVER of Algorithm A6 for $ws(t)[m] \ni \langle k, \_ \rangle$ finishes before e starts at replica p_d^m. By Definition 64 of eo_k,

$$ct(t) \xrightarrow{eo_k} e.$$

□

D.10 Arbitration Relation

Definition 66 (Arbitration Relation)**.** We define the arbitration relation ar on X as the Lamport clock order between them, i.e.,

$$ar = lc.$$

Theorem 67.

$$A \models \text{CAUSALARBITRATION}.$$

Proof. By Definition 57 of vis and Definition 66 of ar,

$$vis \subseteq lc = ar.$$

□

D.11 Return Values

It is straightforward to show that INTRVAL holds for *internal* read events.

Theorem 68.

$$A \models \text{INTRVAL}.$$

Proof. Let $e \in R_{\text{INT}} \cap R_k$ be an internal read event. The transaction $tx(e)$ contains update events on k. By line A2:12, e reads from the last update event on k preceding e in $tx(e)$. □

Now let e be an *external* read event. For notational convenience, we define V_e to be the set of update transactions on k that are visible to $tx(e)$, and S_e the set of update transactions on k that are safe to read at line A3:5. By Assumption 6, (R5), and (R3), all transactions in S_e are committed. Formally,

Definition 69 (Visibility Set)**.** Let $e \in R_{\text{EXT}} \cap R_k$ be an external read event on key k.

$$V_e \triangleq vis^{-1}(tx(e)) \cap T_k.$$

Definition 70 (Safe Set)**.** Let $e \in R_{\text{EXT}} \cap R_k$ be an external read event on key k. Suppose that e is issued to replica p_d^m in data center d.

$$S_e \triangleq \{t \in T_k : ts(t) \le snapVec_{(\text{GET\_VERSION},e)} \wedge log[t][k] \in (\mathsf{opLog}_d^m)_e[k]\}.$$

Lemma 71. Let $e \in R_{\text{EXT}} \cap R_k$ be an external read event on key k. Suppose that e is issued to replica p_d^m in data center d. When e returns at p_d^m (line A3:5), we have

$$V_e \subseteq S_e.$$

Proof. For each $t \in V_e$, we need to show that $t \in S_e$. That is,

$$ts(t) \le snapVec_{(\text{GET\_VERSION},e)} \quad (27)$$

and

$$log[t][k] \in (\mathsf{opLog}_d^m)_e[k]. \quad (28)$$

We first show that (27) holds. Since $t \in V_e$,

$$t \xrightarrow{vis} tx(e).$$

By Definition 57 of vis,

$$ts(t) \le ts(st(e)).$$

By Lemma 46,

$$ts(t) \le snapVec_{(\text{GET\_VERSION},e)}.$$

To show that (28) holds, we perform a case analysis according to whether t is a local transaction in data center d or a remote one in data center $i \ne d$.

- CASE I: t is a local transaction in data center d. Since $t \xrightarrow{vis} tx(e)$, by Lemma 65,

$$ct(t) \xrightarrow{eo_k} e.$$

Therefore,

$$log[t][k] \in (\mathsf{opLog}_d^m)_e[k].$$

- CASE II: t is a remote transaction in data center $i \ne d$. We distinguish between two cases according to whether $t \in T_{causal}$ or $t \in T_{strong}$.

– CASE I: $t \in T_{causal}$. Since $i \neq d$, by line A3:3,

$$snapVec_{(\text{GET\_VERSION},e)}[i] \leq (\mathsf{uniformVec}_d^m)_e[i].$$

By (27),

$$ts(t)[i] \leq (\mathsf{uniformVec}_d^m)_e[i].$$

By Lemma 36,

$$log[t][k] \in (\mathsf{opLog}_d^m)_e[k].$$

– CASE II: $t \in T_{strong}$. By line A3:4,

$$snapVec_{(\text{GET\_VERSION},e)}[strong] \leq (\mathsf{knownVec}_d^m)_e[strong].$$

By (27),

$$ts(t)[strong] \leq (\mathsf{knownVec}_d^m)_e[strong].$$

By Lemma 42,

$$log[t][k] \in (\mathsf{opLog}_d^m)_e[k].$$

□

Theorem 72.

$$A \models \text{EXTRVAL}.$$

Proof. Let $e \in R_{\text{EXT}} \cap R_k$ be an external read event on key k. Suppose that e reads from transaction t in S_e. Since all transactions in S_e are committed, t is committed. By Lemma 47,

$$ts(t) \leq ts(st(e)).$$

By Lemma 56,

$$t \xrightarrow{lc} tx(e).$$

By Definition 57 of *vis*,

$$t \xrightarrow{vis} tx(e).$$

By Definition 69 of V_e,

$$t \in V_e.$$

Both V_e and S_e are totally ordered by *lc*. Since t is the latest one in S_e and $V_e \subseteq S_e$ (Theorem 71), t is also the latest one in V_e. Thus, e reads from t in V_e. That is, e reads from the update event $ud(t,k)$ of V_e. □

Theorem 73.

$$A \models \text{RVAL}.$$

Proof. By Theorems 68 and 72. □

D.12 Uniformity

D.12.1 Uniformity of Causal Transactions Originating at Correct Data Centers

Lemma 74. For any replica p_d^m in any correct data center $d \in \mathcal{C}$, $\mathsf{knownVec}_d^m[d]$ grows without bound.

Proof. Since data center d is correct, by Assumption 4, PROPAGATE_LOCAL_TXS of Algorithm A4 will be executed infinitely often.

- CASE I: Line A4:3 is executed infinitely often. By Assumption 1, $\mathsf{knownVec}_d^m[d]$ grows without bound.
- CASE II: Line A4:5 is executed infinitely often. That is, it is infinitely often that

$$\mathsf{preparedCausal}_d^m \neq \varnothing.$$

By Assumption 4, causal transactions in $\mathsf{preparedCausal}_d^m$ will eventually be committed and removed from $\mathsf{preparedCausal}_d^m$ (line A3:17). Thus, it is infinitely often that new causal transactions are prepared and added into $\mathsf{preparedCausal}_d^m$ (line A3:12) with larger and larger prepare timestamps (line A3:11). Therefore,

$$\min\{ts \mid \langle \_, \_, ts \rangle \in \mathsf{preparedCausal}_d^m\}$$

and

$$\mathsf{knownVec}_d^m[d] = \min\{ts \mid \langle \_, \_, ts \rangle \in \mathsf{preparedCausal}_d^m\} - 1$$

grow without bound.

□

Lemma 75. Let p_d^m be a replica in a correct data center $d \in \mathcal{C}$. If for some $j \in \mathcal{D}$ and some value $x \in \mathbb{N}$

$$\mathsf{knownVec}_d^m[j] \geq x,$$

then eventually

$$\forall c \in \mathcal{C}.\ \mathsf{knownVec}_c^m[j] \geq x.$$

Proof. Since data center d is correct, by Assumption 4, for each other data center $i \neq d$, replica p_d^m will keep

- *replicating* to data center i the write sets

$$\langle \_, wbuff, commitVec, \_ \rangle \in \mathsf{committedCausal}_d^m[j]$$

that have not been received by i from the perspective of d ($commitVec[d] \leq \mathsf{knownVec}_d^m[d]$ at line A4:6 and $commitVec[j] > \mathsf{globalMatrix}_d^m[i][j]$ at line A4:23);
- or sending *heartbeats* with up-to-date $\mathsf{knownVec}_d^m[j]$ to data center i (lines A4:11 and A4:27).

By Assumption 2, $\mathsf{knownVec}_c^m[j]$ at replica p_c^m of each correct data center $c \in \mathcal{C}$ will eventually be updated (lines A4:18 and A4:21) such that

$$\mathsf{knownVec}_c^m[j] \geq \mathsf{knownVec}_d^m[j] \geq x.$$

$\square$

Lemma 76. Let $d \in \mathcal{C}$ be a correct data center. For any replica p_c^m in any correct data center $c \in \mathcal{C}$, $\mathsf{uniformVec}_c^m[d]$ grows without bound.

Proof. By Lemmas 74 and 75, for any replica p_c^m in any correct data center $c \in \mathcal{C}$, $\mathsf{knownVec}_c^m[d]$ grows without bound. By lines A5:2 and A5:8, for any replica p_c^m in any correct data center $c \in \mathcal{C}$, $\mathsf{stableVec}_c^m[d]$ grows without bound. By line A5:3, Assumptions 2 and 3, and lines A5:12–A5:15, for any replica p_c^m in any correct data center $c \in \mathcal{C}$, $\mathsf{uniformVec}_c^m[d]$ grows without bound. $\square$

Lemma 77 (**Property 4**)**.** Let p_d^m be any replica in any data center d. For any time τ, there exists some time τ' such that

$$\forall i \in \mathcal{D}.\, \forall c \in \mathcal{C}.\, \forall n \in \mathcal{P}.$$
$$\mathsf{uniformVec}_c^n(\tau')[i] \geq \mathsf{uniformVec}_d^m(\tau)[i].$$

Proof. By Lemma 34, Assumption 3, and the fact that at most f data centers may fail,

$$\forall i \in \mathcal{D}.\, \exists c \in \mathcal{C}.\, \forall n \in \mathcal{P}.$$
$$\mathsf{uniformVec}_d^m(\tau)[i] \leq \mathsf{knownVec}_c^n(\tau)[i].$$

By Lemma 75, there exists some time τ'' such that

$$\forall i \in \mathcal{D}.\, \forall c \in \mathcal{C}.\, \forall n \in \mathcal{P}.$$
$$\mathsf{uniformVec}_d^m(\tau)[i] \leq \mathsf{knownVec}_c^n(\tau'')[i].$$

By Algorithm A5 and Assumption 3, there exists some time τ' such that

$$\forall i \in \mathcal{D}.\, \forall c \in \mathcal{C}.\, \forall n \leq \mathcal{P}.$$
$$\mathsf{uniformVec}_c^n(\tau')[i] \geq \mathsf{uniformVec}_d^m(\tau)[i].$$

$\square$

Lemma 78. Let $d \in \mathcal{C}$ be a correct data center and $t \in T_{causal}$ be a causal transaction that originates at d. Then for any replica p_c^m in any correct data center $c \in \mathcal{C}$, eventually

$$\forall i \in \mathcal{D}.\, ts(t)[i] \leq \mathsf{uniformVec}_c^m[i].$$

Proof. Since d is correct, by Lemma 76, there exists some time τ' such that

$$ts(t)[d] \leq \mathsf{uniformVec}_c^m(\tau')[d].$$

On the other hand, by Definition 43 of timestamps and Lemma 31 (let $n \triangleq coord(t)$ and $cl \triangleq client(t)$),

$$\forall i \in \mathcal{D} \setminus \{d\}.\, ts(t)[i] = (\mathsf{pastVec}_{cl})_{ct(t)}[i] \leq (\mathsf{uniformVec}_d^n)_{ct(t)}[i].$$

By Lemma 77, there exists some time τ'' such that

$$\forall i \in \mathcal{D} \setminus \{d\}.\, ts(t)[i] \leq (\mathsf{uniformVec}_c^m)(\tau'')[i].$$

Let

$$\tau \triangleq \max\{\tau', \tau''\}.$$

By Lemma 30,

$$\forall i \in \mathcal{D}.\, ts(t)[i] \leq \mathsf{uniformVec}_c^m(\tau)[i].$$

$\square$

D.12.2 Uniformity of Causal Transactions Visible to CL_UNIFORM_BARRIER events

Lemma 79. Let $t \in T_{causal}$ be a causal transaction and $q \in Q$ be a CL_UNIFORM_BARRIER event. If $t \xrightarrow{vis} q$, then for any replica p_c^m in any correct data center $c \in \mathcal{C}$, eventually

$$\forall i \in \mathcal{D}.\, ts(t)[i] \leq \mathsf{uniformVec}_c^m[i].$$

Proof. Since $t \xrightarrow{vis} q$, by Definition 57 of *vis*,

$$ts(t) \leq ts(q).$$

Suppose that q is issued by client cl to replica p_d^n in data center d and is returned at time τ_q. By Definition 43 of timestamps and Lemma 31,

$$ts(t)[d] \leq ts(q)[d] \leq (\mathsf{uniformVec}_d^n)_q[d].$$

By Lemma 77, there exists some time τ' such that

$$ts(t)[d] \leq (\mathsf{uniformVec}_c^m)(\tau')[d].$$

On the other hand, by Definition 43 of timestamps and Lemma 32,

$$\forall i \in \mathcal{D} \setminus \{d\}.\, ts(t)[i] \leq ts(q)[i] = (\mathsf{pastVec}_{cl})_q[i] \leq \mathsf{uniformVec}_d^l(\tau_q)[i]$$

for some replica p_d^l in data center d. By Lemma 77, there exists some time τ'' such that

$$\forall i \in \mathcal{D} \setminus \{d\}.\, ts(t)[i] \leq (\mathsf{uniformVec}_c^m)(\tau'')[i].$$

Let

$$\tau \triangleq \max\{\tau', \tau''\}.$$

By Lemma 30,

$$\forall i \in \mathcal{D}.\, ts(t)[i] \leq \mathsf{uniformVec}_c^m(\tau)[i].$$

$\square$

D.12.3 Uniformity of Strong Transactions

Lemma 80. For any replica p_c^m in any correct data center $c \in \mathcal{C}$, $\mathsf{knownVec}_c^m[strong]$ grows without bound.

Proof. By Assumption 4 and Theorem 9, replica p_c^m will either deliver committed strong transactions infinitely often (DELIVER of Algorithm A6) or submit dummy strong transactions infinitely often (HEARTBEAT_STRONG of Algorithm A6). Thus, $\mathsf{knownVec}_c^m[strong]$ grows without bound. □

Lemma 81. Let $t \in T$ be a transaction. Then for any replica p_c^m in any correct data center $c \in \mathcal{C}$, eventually

$$ts(t)[strong] \le \mathsf{stableVec}_c^m[strong].$$

Proof. By Lemma 80 and lines A5:2 and A5:9, $\mathsf{stableVec}_c^m[strong]$ grows without bound. Therefore, there exists some time τ such that

$$ts(t)[strong] \le \mathsf{stableVec}_c^m(\tau)[strong].$$

□

Lemma 82. Let $t \in T_{strong}$ be a strong transaction. Then for any replica p_c^m in any correct data center $c \in \mathcal{C}$, eventually

$$\forall i \in \mathcal{D}.\ ts(t)[i] \le \mathsf{uniformVec}_c^m[i].$$

Proof. Let $d \triangleq dc(t)$, $n \triangleq coord(t)$, and $cl \triangleq client(t)$. By Definition 43 of timestamps,

$$ts(t) = (\mathsf{pastVec}_{cl})_{\mathsf{ct}(t)}.$$

On the one hand, by Lemma 31,

$$\begin{aligned}\forall i \in \mathcal{D} \setminus \{d\}.\ ts(t)[i] &= (\mathsf{pastVec}_{cl})_{\mathsf{ct}(t)}[i] \\ &\le (\mathsf{uniformVec}_d^n)_{\mathsf{ct}(t)}[i].\end{aligned}$$

By Lemma 77, there exists some time τ' such that

$$\forall i \in \mathcal{D} \setminus \{d\}.\ ts(t)[i] \le (\mathsf{uniformVec}_c^m)(\tau')[i].$$

On the other hand, by Lemma 48,

$$ts(t)[d] = commitVec(t)[d].$$

By (5),

$$commitVec(t)[d] = snapshotVec(t)[d].$$

By lines A6:2 and A3:22,

$$snapshotVec(t)[d] \le (\mathsf{uniformVec}_d^n)_{\mathsf{ct}(t)}[d].$$

Putting it together yields,

$$ts(t)[d] \le (\mathsf{uniformVec}_d^n)_{\mathsf{ct}(t)}[d].$$

By Lemma 77, there exists some time τ'' such that

$$ts(t)[d] \le (\mathsf{uniformVec}_c^m)(\tau'')[d].$$

Let

$$\tau \triangleq \max\{\tau', \tau''\}.$$

By Lemma 30,

$$\forall i \in \mathcal{D}.\ ts(t)[i] \le \mathsf{uniformVec}_c^m(\tau)[i].$$

□

D.13 Eventual Visibility

Theorem 83.

$$A \models \textsc{EventualVisibility}.$$

Proof. Consider a transaction $t \in T$ such that

$$dc(t) \in \mathcal{C} \vee (\exists q \in Q.\ t \xrightarrow{vis} q) \vee t \in T_{strong}.$$

By Lemma 59, it suffices to show that for any client cl,

$$\left| T|_{cl} \right| = \infty \Rightarrow \exists t' \in T|_{cl}.\ t \xrightarrow{vis} t'.$$

By Lemmas 78, 79, 81, and 82, there exists some time τ such that

$$\begin{aligned}&\forall c \in \mathcal{C}.\ \forall n \in \mathcal{P}.\\ &\quad (\forall i \in \mathcal{D}.\ ts(t)[i] \le \mathsf{uniformVec}_c^n(\tau)[i]) \wedge \\ &\quad\ ts(t)[strong] \le \mathsf{stableVec}_c^n(\tau)[strong].\end{aligned} \tag{29}$$

Since $\left| T|_{cl} \right| = \infty$, there exists some correct data center $d \in \mathcal{C}$ to which cl issues an infinite number of transactions. Let $t' \in T$ be the first transaction issued by client cl to data center d which starts after time τ such that

$$lclock(t) < lclock(t').$$

Thus, by Definition 54 of lc,

$$t \xrightarrow{lc} t'.$$

Let $m \triangleq coord(t')$. Since d is correct, by (29),

$$\begin{aligned}&(\forall i \in \mathcal{D}.\ ts(t)[i] \le \mathsf{uniformVec}_d^m(\tau)[i]) \wedge \\ &ts(t)[strong] \le \mathsf{stableVec}_d^m(\tau)[strong].\end{aligned}$$

By Lemma 30,

$$\forall i \in \mathcal{D}.\ \mathsf{uniformVec}_d^m(\tau)[i] \le (\mathsf{uniformVec}_d^m)_{\mathsf{st}(t')}[i].$$

By Lemma 41,

$$\mathsf{stableVec}_d^m(\tau)[strong] \le (\mathsf{stableVec}_d^m)_{\mathsf{st}(t')}[strong].$$

By Lemma 45,

$$(\forall i \in \mathcal{D}.\ (\mathsf{uniformVec}_d^m)_{st(t')}[i] \le ts(st(t'))[i]) \wedge$$
$$(\mathsf{stableVec}_d^m)_{st(t')}[strong] \le ts(st(t'))[strong].$$

Putting it together yields

$$ts(t) \le ts(st(t')).$$

By Definition 57 of *vis*,

$$t \xrightarrow{vis} t'.$$

□

D.14 UniStore Correctness

Theorem 84.

$$\text{UNISTORE} \models \text{POR}.$$

Proof. By Theorems 58, 61, 67, 73, and 83. □

D.15 UniStore Liveness

Lemma 85. Each client migration CL_ATTACH to a correct data center will eventually terminate, provided that the client managed to complete its CL_UNIFORM_BARRIER call at its original data center just before CL_ATTACH.

Proof. Suppose that client cl in its current data center $d \triangleq cur\_dc(cl)$ issues a CL_ATTACH event a to replica p_c^m in correct data center $c \in \mathcal{C}$.

Let $e \in C$ be the last commit event issued by client cl before a. If e does not exist, then

$$\forall i \in \mathcal{D}.\ \mathsf{pastVec}_{cl}[i] = 0.$$

Therefore, the wait condition

$$\forall i \in \mathcal{D} \setminus \{c\}.\ \mathsf{uniformVec}_c^m[i] \ge 0$$

at line A3:24 at replica p_c^m will eventually hold. Thus, the CL_ATTACH event a eventually terminates.

Otherwise, suppose that e is issued to replica $p_{d'}^{n'}$ in data center d'. By Lemma 31,

$$\forall i \in \mathcal{D} \setminus \{d'\}.\ (\mathsf{uniformVec}_{d'}^{n'})_e[i] \ge (\mathsf{pastVec}_{cl})_e[i]. \quad (30)$$

Note that it is possible that $d' \ne d$, since there may exist other CL_ATTACH events between e and a. Therefore, we distinguish between the following two cases:

- CASE I: $d' = d$. By (30),

$$\forall i \in \mathcal{D} \setminus \{d\}.\ (\mathsf{uniformVec}_d^{n'})_e[i] \ge (\mathsf{pastVec}_{cl})_e[i]. \quad (31)$$

- CASE II: $d' \ne d$. Consider the last CL_ATTACH event, denoted a', before a. Suppose that a' is issued to replica p_d^n in data center d. By Lemma 31, when a' terminates,

$$\forall i \in \mathcal{D} \setminus \{d\}.\ (\mathsf{uniformVec}_d^n)_{a'}[i] \ge (\mathsf{pastVec}_{cl})_{a'}[i] = (\mathsf{pastVec}_{cl})_e[i]. \quad (32)$$

Let $q \in Q$ be the CL_UNIFORM_BARRIER event issued by client cl just before a. Suppose that q is issued to replica p_d^l in data center d. By Lemma 31,

$$(\mathsf{uniformVec}_d^l)_q[d] \ge (\mathsf{pastVec}_{cl})_q[d] = (\mathsf{pastVec}_{cl})_e[d]. \quad (33)$$

By (31), (32), (33), and Lemma 77, eventually for the correct data center c,

$$\forall i \in \mathcal{D}.\ \mathsf{uniformVec}_c^m[i] \ge (\mathsf{pastVec}_{cl})_e[i].$$

Therefore, the wait condition

$$\forall i \in \mathcal{D} \setminus \{c\}.\ \mathsf{uniformVec}_c^m[i] \ge (\mathsf{pastVec}_{cl})_e[i]$$

at line A3:24 at replica p_c^m will eventually hold. Thus, the CL_ATTACH event a eventually terminates. □

Theorem 86. Any client event issued at a correct data center will eventually terminate.

Proof. Consider any event e issued by client cl at a correct data center $c \in \mathcal{C}$. It suffices to show that each wait condition in the execution of e, if any, will eventually hold. In the following, we perform a case analysis according to the type of event e.

- CASE I: $e \in S$. The theorem holds trivially.
- CASE II: $e \in R$. If $e \in R_{\text{INT}}$, the theorem holds trivially. Otherwise, $e \in R_{\text{EXT}}$. By Lemmas 74 and 80, the wait condition at line A3:4 for e will eventually hold.
- CASE III: $e \in U$. The theorem holds trivially.
- CASE IV: $e \in C_{causal}$. Since data center c is correct, the wait condition at line A2:28 will eventually hold.
- CASE V: $e \in C_{strong}$. By Lemma 76, the wait condition at line A3:22 will eventually hold. Thus, line A6:2 will eventually terminate. Then, by Assumption 6, the procedure CERTIFY and thus the event e will eventually terminate.
- CASE VI: $e \in$ CL_UNIFORM_BARRIER. By Lemma 76, the wait condition at line A3:22 will eventually hold.
- CASE VII: $e \in$ CL_ATTACH. The theorem holds due to CASE VI and Lemma 85.

□