Cure: Strong semantics meets high availability and low latency

Deepthi Devaki Akkoorath*, Alejandro Z. Tomsic†, Manuel Bravo‡, Zhongmiao Li‡,
Tyler Crain†, Annette Bieniusa*, Nuno Preguiça§, Marc Shapiro†

*University of Kaiserslautern, †Inria & LIP6-UPMC-Sorbonne Universités
‡Université Catholique de Louvain, §NOVA LINCS

***Abstract*—Developers of cloud-scale applications face a difficult decision of which kind of storage to use, summarised by the CAP theorem. Currently the choice is between classical CP databases, which provide strong guarantees but are slow, expensive, and unavailable under partition; and NoSQL-style AP databases, which are fast and available, but too hard to program against. We present an alternative: Cure provides the highest level of guarantees that remains compatible with availability. These guarantees include: causal consistency (no ordering anomalies), atomicity (consistent multi-key updates), and support for high-level data types (developer friendly API) with safe resolution of concurrent updates (guaranteeing convergence). These guarantees minimise the anomalies caused by parallelism and distribution, thus facilitating the development of applications. This paper presents the protocols for highly available transactions, and an experimental evaluation showing that Cure is able to achieve scalability similar to eventually-consistent NoSQL databases, while providing stronger guarantees.**

I. Introduction

Internet-scale applications are typically layered above a high-performance distributed database engine running in a data centre (DC). A recent trend is to use geo-replication across several DCs to avoid wide-area network latency and to tolerate downtime. This scenario poses big challenges to the distributed database. Since network failures (called partitions) are unavoidable, according to the CAP theorem [20] the database design must sacrifice either strong consistency or availability. Traditional databases are "CP"; they provide consistency and a high-level SQL interface, but lose availability. NoSQL-style databases are "AP", highly available, which brings significant performance benefits. However, AP-databases expose application developers to inconsistency anomalies, and most provide only low-level key-value interface.

To alleviate this problem, recent work has focused on enhancing AP designs with stronger semantics [23, 24, 28]. This paper presents Cure, our contribution in this direction. While providing availability and performance, Cure supports: *(i)* causal+ consistency, ensuring that if one update happens before another, they will be observed in the same order, and that replicas converge to the same state under concurrent conflicting updates, *(ii)* support for high-level replicated data types (CRDTs) such as counters, sets, tables and sequences, with intuitive semantics and guaranteed convergence even in the presence of concurrent conflicting updates and partial failures, and *(iii)* transactions, ensuring that multiple keys (objects) are both read and written consistently, in an interactive manner.

Causal+ consistency (CC+) [6, 23] represents a sweet spot in the availability-consistency tradeoff. It is the strongest model compatible with availability [8] for individual operations. Since it ensures that the causal ordering of operations is respected, it is easier to reason about for programmers and users. Consider, for instance, a user who posts a new photo to her social network profile, then comments on the photo on her wall. Without causal consistency, a user might observe the comment but not be able to see the photo, which requires extra programming effort to avoid the anomaly at the application level.

CC+ requires that replicas converge to the same state under concurrent conflicting updates. For guaranteeing convergence, many existing causal+ consistent systems adopt the last-writer-wins rule [7, 17, 19, 23, 24], where the update that occurs "last" overwrites the previous ones. We rely on CRDTs, developer-friendly high-level data types that guarantee convergence and have rich semantics [27]. Operations on CRDTs are not only register-like assignments, but methods corresponding to a CRDT object's type. For example, a set supports *add(item)* and *remove(item)* operations. The implementation of a CRDT set guarantees that no matter the order in which a replica receives two conflicting add and remove operations, the state of the set will converge at different replicas without the need for synchronization or application conflict handling. For instance, the Bet365 developers report that using Set CRDTs changed their life, freeing them from low-level detail and from having to compensate for concurrency anomalies [25].

Performing multiple operations in a transaction enables the application to maintain relations between multiple objects or keys. *Highly Available Transactions* (HATs) eschew traditional strong isolation properties, which require synchronisation, in favour of availability and low latency [9, 14]. Existing CC+ HAT implementations provide either reading from a snapshot [7, 17, 19, 23, 24] or atomicity of updates [11, 24]; we introduce Transactional Causal Consistency (TCC), where all transactions provide both.

Taken together, the above features provide clear and strong semantics to developers. In fact, the combination of the three equip Cure with the strongest semantics ever provided by an always-available data store. The absence of any of these features can be compensated by ad-hoc mechanisms at the application level, but this is tricky and error prone [10].

The contributions of this paper are the following:

- a novel programming model providing causally consistent interactive transactions with high-level, confluent data types (§II-A);
- a high-performance protocol, supporting this programming model for geo-replicated datastores (§III);
- a comprehensive evaluation, comparing our approach to state-of-the-art data stores (§V).

II. Cure overview

A. Transactional Programming Model

A body of research has extended the causal+ consistency (CC+) model [23] by adding multi-key operations. There are two major efforts in this direction: *read-only transactions* [7, 17, 19, 23, 24] that provide clients with a consistent view of multiple keys, and *write-only transactions* [24] that permit clients to perform atomic multi-key write operations.

Unfortunately, strictly separating these two multi-key operations limits the flexibility of the programming model. For instance, it is a common practice to read an object before updating it. By separating these operations, concurrent updates over the same object may interleave. Consequently, programmers would have to deal with these anomalies at the application level, adding complexity.

We introduce a novel Transactional Causal Consistency (TCC) model, that explores a new spot along the consistency spectrum; the strongest semantics a system can achieve under high-availability and low-latency. Cure's TCC extends CC+ functionality by adding *interactive transactions* and *CRDT support* to ensure replica convergence.

Interactive transactions allow programmers to combine read and write operations flexibly within the same transaction, ensuring the following properties:

- Transactions read from a *causally consistent snapshot*, which represents a view of the data store that includes the effects of all transactions that causally precede it [19, 23]. For any pair of objects a and b, with a_i, the version i of a, and b_j, the version j of b, belong to the same causally consistent snapshot if there does not exist a_k, another version of a such that (i) a_k is created after a_i, and (ii) a_k is causally before b_j.
- A transaction updating multiple objects respects *atomicity*, i.e., all updates occur and are made visible simultaneously, or none does. Applying updates in this fashion creates a new causally consistent snapshot of the store.

CRDTs are high-level data types that can be replicated and modified concurrently while guaranteeing that replicas converge. CRDT abstractions such as counters, sets, maps/tables and sequences provide a richer interface for programmers than the usual key-value interface with last-writer-wins (LWW) conflict resolution, which is the default policy used by previous works [7, 17, 19, 23, 24].

Comparison to other models. Table I compares TCC with the consistency models implemented by other storage systems.

Serializability (SER) is the strongest isolation level defined by the SQL standard, providing a model equivalent to a sequential database.

Snapshot Isolation (SI) [13] is the isolation level provided by popular commercial databases such as SQL Server and Oracle. Under SI, a transaction reads from a snapshot and updates are applied respecting total order. Concurrent transactions conflict only if they update the same data items. Two concurrent transactions, t_1 and t_2, are not serializable when t_1 reads an object written by t_2 and t_2 reads an object written by t_1. This anomaly is called write skew or short fork (as the state of the database inside a transaction is like a fork from the database state).

Parallel Snapshot Isolation (PSI) [28] and Non-Monotonic Snapshot Isolation (NMSI) [26] are weaker forms of SI that allow the long fork anomaly, where it is possible that two concurrent transactions t_1 and t_2 commit, writing to different data items, and then two other transactions that start subsequently, where one sees the effects of t_1 but not t_2, and the other sees the effects of t_2 but not t_1. Still, PSI and NMSI avoid write/write conflicts by aborting transactions that attempt to update the same items concurrently.

Our consistency model, TCC, also allows short and long fork. However it adds a powerful new feature: convergent forks, where concurrent transactions can modify the same data items, with concurrent updates being merged using CRDTs[1].

TCC-S is the model provided by Eiger [24], which combines CC+ with *read-only* and *write-only transactions* (COPS [23], GentleRain [19], Orbe [17] and ChainReaction [7] provide a similar model but only support read-only transactions). Unlike TCC, TCC-S does not support interactive transactions that include reads and writes, but only read-only or write-only transactions. Additionally, convergence is achieved using last-writer-wins rules.

Finally, eventual consistency (EC), used in systems such as Dynamo [16] and the commercial Riak database [4], includes no support for transactions.

B. Design

We assume a geo-replicated key-value store that handles a large number of objects. The full set of objects is replicated

[1]Walter [28], an implementation of PSI, also supports a similar consistency model restricted to counters and sets.

	Stronger ⟵					⟶ Weaker
Properties	SER	SI	PSI NMSI	Cure	TCC-S	EC
Transactions	yes	yes	yes	yes	read-only write-only	no
Short fork	x	✓	✓	✓	✓	✓
Long fork	x	x	✓	✓	✓	✓
Convergent fork	x	x	x	✓	✓	✓
Convergence	–	–	–	CRDT	LWW	LWW

Table I: Properties of consistency models (x: disallowed).

across different data centres (DC) to provide availability and low latency. Each of the data centres has N partitions, where each partition stores a non overlapping subset of the key-space. We assume that every data centre employs the same partitioning scheme.

The challenge in the design of Cure consists of providing TCC under such a setting without compromising availability, while still achieving high scalability. To meet this goal, our protocol adopts three key design decisions. First, our protocol relies on the time-stamps of events to encode the causal dependencies which allow partitions to take decisions locally. This avoids using explicit dependency check messages, which substantially penalize performance [19]. Second, the protocol does not rely on centrally assigned time-stamps [18], as used in other systems. Third, the protocol decouples the problem of propagating updates among replicas from the problem of making these updates visible. This allows partitions to propagate updates pairwise without requiring coordination with other partitions. A lightweight protocol involving all partitions runs asynchronously to establish the set of transactions that are up to date, i.e. all causal dependencies are known, in a given data centre.

C. Programming interface

Cure's interface offers the following operations:

- TxId ← START_TRANSACTION(CausalClock)
  initiates a transaction that causally depends on all updates issued before CausalClock. It returns a transaction handle that is used when issuing reads and updates. When an application does not specify the CausalClock, the most recent version available in the data centre is used.
- Values ← READ_OBJECTS(Keys, TxId)
  returns the list of values that correspond to the state of the objects stored under the Keys in the version given in the transaction's snapshot.
- ok ← UPDATE_OBJECTS(Updates, TxId)
  declares a list of Updates for a transaction.
- CommitTime ← COMMIT(TxId)
  commits the transaction under transaction handler TxId and makes the updates visible.
- ok ← ABORT(TxId)
  discards the updates and aborts the transaction.

Under this programming model, a user issues transactions where each operation is defined by the CRDT specification.

cvc	Client clock
p_d^m	Partition m at DC d
$Clock_d^m$	Current physical time at p_d^m
pvc_d^m	Vector clock at p_d^m
GSS_d^m	Globally stable snapshot at p_d^m
$prepTx_d^m$	Prepared transactions at p_d^m
$committedTx_d^m$	Committed transactions at p_d^m
Log_d^m	Log of updates at p_d^m
PMC_d^m	Matrix of received pvc_d^i at p_d^m
T	Transaction
TC_T	Transaction Coordinator of T
svc_T	Snapshot vector clock of T
ct_T	Commit vector clock of T
$ws_T[m]$	Write set of T for partition m

Table II: Notation used in the protocol description.

III. Protocol description

Cure keeps multiple versions of each object in order to serve requests from causally consistent snapshots. Each version stores its value along with the metadata that encodes its causal dependencies. Old versions are periodically garbage collected by the system.

Our protocol assumes that each partition is equipped with a physical clock. Clocks are loosely synchronized by a time synchronization protocol such as NTP [3]. Each clock generates monotonically increasing timestamps. The correctness of the protocol does not depend on the synchronization precision, though clock skew between servers can impact performance.

Cure annotates updates with the transaction commit time; a vector clock with an entry per DC. Commit times produce a partial order that respects causal consistency. The protocol uses these commit times to make transactions visible in accordance with causality. Transactions originating at the local DC are immediately visible to clients when they commit, as their causal dependencies are automatically satisfied. In contrast, transactions arriving from remote DCs depend on the *globally stable snapshot* (GSS), which represents a view of the data store that is known to be available at every partition of the local DC; they are only made visible when the GSS advances past their commit timestamp. This ensures that all causally related transactions, which have a smaller commit timestamp, are already visible locally.

A. Notation and definitions

Table II introduces the notation followed in this section. We assume a total number of D DCs. A partition m at DC d, denoted by p_d^m, keeps the following state:

- pvc_d^m, a vector clock of size D, where position k indicates that p_d^m has received updates up to $pvc_d^m[k]$ from p_k^m;
- GSS_d^m, a vector clock of size D that denotes the latest globally stable consistent snapshot known by p_d^m, i.e., the snapshot that p_d^m knows to be available at all partitions of its DC. In order to advance GSS_d^m, partitions of the same DC periodically exchange their pvc. Each p_d^m computes its GSS_d^m as the aggregate minimum of known pvc_d;

Clients connect to Cure servers to issue transactions. A server receiving a client request to start transaction T acts as its *transaction coordinator (TC_T)*. TC_T keeps the necessary state for executing T (See Table II).

B. Transaction Execution

Algorithms 1 and 2 show the pseudocode of the protocol for executing transaction T at DC d followed by the transaction coordinator (TC_T) and the involved partitions, respectively.

Start transaction. The transaction coordinator TC_T starts by ensuring that it assigns T a snapshot no older than the last one seen by the client, represented by cvc (Alg. 1, line 3). This is necessary to ensure that clients observe monotonically increasing causally consistent views of the data store.

To define the causally consistent snapshot T will access, it sets the vector clock svc_T to include all remote transactions that are stable in the local DC plus all locally committed transactions. The former is achieved by setting the vector to the value of GSS_d^m (Alg. 1, line 4), while the latter is achieved by setting the entry for the local DC in svc_T to the maximum of either the physical clock of the server or the client's previously observed timestamp (Alg. 1, line 5). The snapshot must include the updates of all transactions that have a commit vector clock smaller than or equal to svc_T. This guarantees that the snapshot is causally consistent, since it includes the dependencies of all transactions.

Update objects. The client provides a list of key-update pairs, which TC_T buffers in a per-partition write set ($ws_T[m]$) to be sent, when committing T, to each updated partition at DC d.

Read objects. The client provides a list of keys that she wants to read. TC_T forwards, for each of the requested keys, a read request to the local replica that stores it (retrieved by the call to the PARTITION function). Upon receiving such request, and in order to respect the snapshot defined by svc_T, i.e., that the snapshot includes all updates with commit time smaller than svc_T, p_d^m might need to wait for its $pvc_d^m[d]$ to catch up (Alg. 2, line 2). Once this is satisfied, p_d^m returns the latest version of the object with commit time no newer than svc_T, which is retrieved by calling the SNAPSHOT function (Alg. 2, line 3). Once TC_T receives this reply, it applies the update operations on the same object (if any) issued by T during previous UPDATE_OBJECTS operations (Alg. 1, line 21), generating a new version of the object. Note that this is a consequence of providing more developer-friendly data types than just basic registers. TC_T caches this result until all objects in the operation are read. This process is repeated for every requested key. Once it finishes, TC_T returns all read values to the client.

Commit. When receiving a commit request from a client, TC_T starts a 2PC protocol to atomically commit the updates of transaction T at local DC d. In the first phase, TC_T sends a prepare message including $ws_T[m]$ to each of the updated partitions (Alg. 1, lines 28-30). Upon receiving such message, each partition takes the current value of its physical clock (Alg. 2, line 8) and proposes it as the transaction's

Algorithm 1 Transaction coordinator at server m of DC d

```
1:  function START_TRANSACTION(cvc)
2:      for k = 1...D, k ≠ d do
3:          wait until cvc[k] ≤ GSS_d^m[k]
4:      svc_T ← GSS_d^m
5:      svc_T[d] ← MAX(cvc[d], Clock_d^m)
6:      return T
7:
8:  function UPDATE_OBJECTS(T, Updates)
9:      for all ⟨Key, Operation⟩ ∈ Updates do
10:         p_d^i ← PARTITION(Key)
11:         if p_d^i ∉ UpdatedPartitions_T then
12:             UpdatedPartitions_T ← UpdatedPartitions_T ∪ {p_d^i}
13:         ws_T[i] ← ws_T[i] ∪ {⟨Key, Operation⟩}
14:     return ok
15:
16: function READ_OBJECTS(T, Keys)
17:     for all Key ∈ Keys do
18:         p_d^i ← partition(Key)
19:         Val ← send READ_KEY(svc_T, Key) to p_d^i
20:         for all ⟨Key, Operation⟩ ∈ ws_T[i] do
21:             Val ← APPLY_OPERATION(Val, Operation)
22:         Values ← Values ∪ {Val}
23:     return Values
24:
25: function COMMIT(T)
26:     if UpdatedPartitions_T = ∅ then
27:         return svc_T
28:     for all p_d^i ∈ UpdatedPartitions_T do
29:         send PREPARE(T, ws_T[i], svc_T) to p_d^i
30:         wait until receiving (T, PrepTime) from p_d^i
31:     CommitTime ←MAX(all prepare times)
32:     ct_T ← svc_T
33:     ct_T[d] ← CommitTime
34:     for all p_d^i ∈ UpdatedPartitions_T do
35:         send COMMIT(T, ct_T) to p_d^i
36:     return ct_T
```

Algorithm 2 Protocol executed by partition p_d^m

1: **function** READ_KEY(svc_T, Key)
2: wait until $svc_T[d] \leq pvc_d^m[d]$
3: $Val \leftarrow$ SNAPSHOT(Key, svc_T, Log_d^m)
4: send Val to TC_T
5:
6: **function** PREPARE(T, $ws_T[m]$, svc_T)
7: wait until $svc_T[d] \leq Clock_d^m$
8: $PrepTime \leftarrow Clock_d^m$
9: $Log_d^m \leftarrow Log_d^m \cup \{\langle ws_T[m], PrepTime, svc_T\rangle\}$
10: $prepTx_d^m \leftarrow prepTx_d^m \cup \{\langle T, PrepTime\rangle\}$
11: send $\langle T, PrepTime\rangle$ to TC_T
12:
13: **function** COMMIT(T, ct_T)
14: $Log_d^m \leftarrow Log_d^m \cup \{\langle ct_T\rangle\}$
15: $prepTx_d^m \leftarrow prepTx_d^m \setminus \{\langle T, PrepTime\rangle\}$
16: $committedTx_d^m \leftarrow committedTx_d^m \cup \{\langle T, ct_T\rangle\}$
17:
18: **function** PROPAGATE_TXS( ) ▷ Run periodically
19: **if** $prepTx_d^m \neq \varnothing$ **then**
20: $pvc_d^m[d] \leftarrow$MIN($prepTx_d^m$) -1
21: **else**
22: $pvc_d^m[d] \leftarrow Clock_d^m$
23: **if** $committedTx_d^m = \varnothing$ **then**
24: **for** $k = 1 \ldots D, k \neq d$ **do**
25: send HEARTBEAT($pvc_d^m[d], d$) to p_k^m
26: **return**
27: **for all** $\langle T, ct_T\rangle \in committedTx_d^m \mid ct_T < pvc_d^m[d]$ **do**
28: **for** $k = 1 \ldots D, k \neq d$ **do**
29: send REPLICATE_TX($ws_T[p], ct_T, svc_T, d$) to p_k^m
30: $committedTx_d^m \leftarrow committedTx_d^m \setminus \{\langle T, ct_T\rangle\}$
31:
32: **function** REPLICATE_TX($ws_T[p], ct_T, svc_T, k$)
33: $Log_d^m \leftarrow Log_d^m \cup \{\langle ws_T[p], ct_T, svc_T\rangle\}$
34: $pvc_d^m[k] \leftarrow ct_T[k]$
35:
36: **function** HEARTBEAT($TimeStamp, k$)
37: $pvc_d^m[k] \leftarrow TimeStamp$
38:
39: **function** BCAST_PVC() ▷ Run periodically
40: **for** $i = 1 \ldots N, i \neq m$ **do**
41: send UPDATE_GSS(m, pvc_d^m) to p_d^i
42:
43: **function** UPDATE_GSS(i, pvc)
44: $PMC_d^m[i] \leftarrow pvc$
45: **for** $k = 1 \ldots D, k \neq d$ **do**
46: $GSS_d^m[k] \leftarrow \min_{i=1\ldots N} PMC_d^m[i][k]$

commit timestamp. Next, it stores its write set in its log. TC_T computes the transaction's commit timestamp as the maximum of all proposed prepare timestamps (Alg. 1, line 31), and generates ct_T, the commit vector clock of T, by applying this commit time, at position d, to svc_T. The coordinator then sends a commit message to all involved partitions that includes the transaction's commit vector clock. When a partition receives the commit message, it removes T from $prepTx_d^m$, stores the ct_T in its log, and adds T and its commit timestamp to $committedTx_d^m$ for propagating its updates to the other DCs.

Choosing the maximum proposed timestamp as the commit timestamp of a transaction is important for correctness [18]. The read protocol waits for prepared transactions expected to be included in a snapshot (Alg. 2, line 2). If TC_T were to choose a ct smaller than the prepare timestamp of a participant partition, a transaction reading from the partition with svc smaller than the prepare timestamp but greater than this ct would not be delayed to include the committing transaction. Therefore, it would read an incorrect version.

C. Replication to remote DCs

Each partition periodically synchronizes with sibling partitions in other DCs. When there are no new updates to send, a heartbeat is sent. On receiving a heartbeat (Alg. 2, line 36), a replica advances $pvc_d^m[k]$ stating that it has received all updates from DC k until the received timestamp. When there are updates to send, a replica sends all committed updates with timestamp smaller than any prepared but not yet committed transaction[2] (Alg. 2, lines 27-29). On receiving an update replication message from DC k, a replica inserts the received updates in its log and advances $pvc_d^m[k]$, setting it to the update's commit timestamp $ct_T[k]$.

Our algorithm decouples propagating updates among replicas from making these updates visible. An update received from a remote replica is only made visible after it is known that all updates from the same transaction (and their dependencies) have already been received. To this end, partitions in each DC exchange their pvc_d vectors in the background (Alg. 2, line 39), and each partition m computes its GSS_d^m as the aggregate minimum of known pvc_d (Alg. 2, line 43).

IV. Discussion

A. Why vector clocks?

Cure uses a stabilization protocol for making updates visible while respecting causal consistency. In the work that introduced the GentleRain protocol, Du et al. [19] showed that this approach improves throughput over using explicit dependency check messages, which also incurs large dependency metadata. The protocol underlying GentleRain tracks causal dependencies using a single scalar timestamp. Although this provides a compact representation of dependencies, it penalises update visibility latency at remote sites, and limits progress in the presence of network partitions between DCs. Cure's global stabilization mechanism relies on a vector clock sized by the number of DCs in the system. In the following paragraphs, we discuss these problems and how the use of vector clocks reduces their unwanted effects.

Remote update visibility latency. We define the visibility latency of an update operation as the amount of time that

[2]A transaction being prepared with a given prepare timestamp can commit before a concurrent transaction with a lower one when they update different partitions.

passes between the moment it is committed at its local DC, and the time at the receiving partition at a remote DC when the partition makes that update visible to be safely read without violating causality. The use of a single scalar penalizes GentleRain because in order to make a remote update visible locally, the partition must wait until it receives a heartbeat with a time greater than that of the update from all other servers at all other DCs. In other words, in GentleRain, update visibility latency at a DC is dependent on the latency to the furthest DC.

By using a vector clock to timestamp events, Cure is able to make a remote update from some DC d with a commit vector clock ct_T visible when the servers in DC k have received all updates up to $ct_T[d]$ from replicas at DC d, i.e., $GSS_k^p[d] \geq ct_T[d]$. Therefore, similarly to implementations relying on dependency check messages, Cure incurs update visibility latency dependent on the latency to the DC where the update originated. The cost of this improvement is an increased meta-data size over the use of a single scalar, which we find reasonable for geo-replicated data stores, normally deployed at a small number of DCs.

Progress in the presence of failures. In the presence of a DC failure or network partition between DCs, relying on a single scalar timestamp means remote updates can no longer become visible. using a single scalar timestamp translates into stopping to make updates from *any* remote DC visible. In this situation, the state that a DC is able to observe from remote DCs remains frozen until the system recovers from the failure, while local updates continue to be made visible. In contrast, by using a vector clock, updates in Cure from healthy DCs are made visible continously even under network partitions, and only updates from the failed or disconnected DC remain frozen until the system recovers.

B. Session Guarantees

Cure ensures that transactions in a client session see (i) the effects of previously committed transactions by the same client, and (ii) monotonically-increasing snapshots of the data store. When a client finishes a read-only transaction, its snapshot vector clock is returned. Similarly, when a client successfully commits an update transaction, its commit vector clock is returned. The client vector clock cvc must accordingly be updated to the value returned by the last transaction completed. When a client starts a new transaction, it sends cvc with its request. If cvc is greater than the GSS at the server receiving the request, it is blocked until GSS proceeds past cvc. Otherwise, it starts immediately.

C. Efficient GSS computation

Under Cure, partitions within the same DC periodically exchange their pvc to compute their GSS. To do this efficiently, Cure builds a tree over all servers in a DC and computes an aggregate minimum using the tree [19]. When compared to a simple broadcast approach, this reduces the number of messages exchanged in the network, while computing and distributing GSS in a reasonable amount of time. This is important for remote update visibility latency as updates from remote transactions are only made visible after the updated GSS exceeds commit vector clock.

D. Garbage Collection

In Cure, partitions periodically exchange the oldest snapshot vector clock of its active transactions and compute the aggregate minimum. Once a partition computes this minimum, it can safely remove from its log the versions of its stored objects that are no longer required, i.e., that no transaction will request in the future.

E. CRDT support

Cure relies on operation-based CRDTs [27]. Their implementation requires adequate support from the system, as an object's value is defined not just by the last update, but also by the state it is applied on. This requires causal consistency to ensure that updates are neither lost nor delivered out-of-order. Hence, Cure encodes with each update a vector clock that represents the state to which it applies at a remote DC. As high-level updates are often not idempotent, safety also demands that updates are applied *exactly once*. Consider, for instance, an $increment$ operation over a counter being sent from its originating DC to a remote one. If the network replicated this message, applying the increment twice may leave the system in an inconsistent state. To avoid such scenarios, the updates are assigned unique identifiers and the log filters the duplicates when they are inserted.

F. Correctness

We provide an informal proof that Cure implements TCC by showing that the snapshot read by a transaction is causally consistent and respects the atomicity of committed transactions.

Proposition 1. If an update u_1 depends on an update u_2, then $u_2.ct < u_1.ct$.

An update u_1 depends on u_2 if the transaction of u_2 reads from a snapshot that contains u_1. From Alg. 2 line 7, the proposed timestamps are always greater than its snapshot time (in DC d, the entry d of its snapshot vector clock). Since the commit timestamp is generated as the maximum of proposed timestamps, the commit time of a transaction is always greater than its snapshot time. Then, by Alg 1 lines 32-33, the commit vector clock of an update is always greater than its snapshot vector clock.

Proposition 2. A partition vector clock $pvc_d^m = t$ implies that p_d^m has received all updates with commit vector clock $\leq t$.

First, we show that the proposition is valid for remote updates. We prove this by contradiction. Assume there is a

remote update u from DC j such that $u.ct < t$, and p_d^m has not received u. By Alg. 2 lines 33-34, the partition would have received an update u_1 such that $u_1.ct[j] = t[j]$. Because the updates are sent in the order of their timestamps, the partition cannot receive another update u_1 before u if $u_1.ct[j] > u.ct[j]$. Hence $u.ct[j] > t[j]$, implying $u.ct \not< t$, leading to the contradiction.

Now we show that there are no pending local updates with commit vector clock $\leq t$. When updating $pvc[d]$, the partition finds the minimum prepared time stamps of the transactions in the prepared phase. Since the physical clock is monotonic and the commit time is calculated as the maximum of all prepared times, it is guaranteed that all future transactions will receive a commit time which is greater than or equal to this minimum prepared time stamp. So, when the $pvc[d]$ is set to the minimum prepared time minus 1 (Alg. 2, line 20), the partition has already received all updates for the snapshot pvc.

Proposition 3. Reads return values from a causally consistent snapshot.

Proposition 1 and 2 and the delaying of reads until pvc includes the snapshot time (Alg 2 line 2), together guarantees that by including all updates in a partition which have a commit vector clock $\leq$ snapshot vector clock, the read returns values from a causally consistent snapshot. This is true even when reading from multiple partitions because it reads from the same snapshot.

Proposition 4. Reading from a snapshot respects atomicity.

Atomicity is not violated even though updates (local and remote) are made visible independently by each partition. All updates from a transaction belong to the same snapshot because they receive the same commit vector clock. A read is delayed until the same snapshot is available in all (accessed) partitions, thus reading all or no updates from a transaction.

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

V. Evaluation

A. Setup

We build Cure on top of Antidote [1], an open-source reference platform that we have created for fairly evaluating distributed consistency protocols. The platform is built using the Erlang/OTP programming language, a functional language designed for concurrency and distribution. To partition the set of keys across distributed physical servers we use riak-core [4], an open source distribution platform using a ring-like distributed hash table (DHT), partitioning keys using consistent hashing. Key-value pairs are stored in an in-memory hash table with updates being persisted to an on disk operation log using Erlang's disk-log module.

In addition to Cure, we have implemented eventual consistency, Eiger and GentleRain protocols for comparison. Our implementation of Eventual consistency is a single-versioned key-value store, supporting LWW registers, where the ordering of concurrent updates is determined by local physical clocks, and state-based CRDTs [27]. Eiger supports causal consistency using LWW registers and tracks one-hop nearest dependencies, requiring explicit dependency checks. GentleRain supports causal consistency using a global stable time mechanism, requiring all-to-all communication for updates from external DCs to become visible (local updates are visible immediately). In addition to LWW registers, Cure supports operation-based CRDTs. All three causally consistent protocols provide (static) read-only and atomic update transactions. Cure additionally supports interactive read and update transactions.

Objects in Cure, Eiger, and GentleRain are multi-versioned. For each key, a linked-list of recent updates and snapshots is stored in memory. Old versions are garbage collected when necessary. An update operation appends a new version of the object to the in-memory list and asynchronously writes a record to the operation log. If a client requests a version of an object that is no longer available in memory then it is retrieved from the operation log on disk.

Hardware. All experiments are run on the Grid5000 [21] experimental platform using dedicated servers. Each server consists of 2 CPUs Intel Xeon E5-2630 v3, with 8 cores/CPU, 126GB RAM, and 2 558GB hard drives. Nodes are connected through shared 10Gbps switches. The measured latency within the cluster was aproximatelly 0.15 ms. Before running each experiment, clocks were synchronized through an NTP [3] server running within the cluster.

Experiments are run using a variable number of DCs, each comprised by a variable number of servers, located within a Grid 5000 cluster at Rennes. Nodes within the same DC communicate using the distributed message passing framework of Erlang/OTP running over TCP. Connections across separate DCs use ZeroMQ [5] sockets running TCP, with each node connecting to all other nodes to avoid any centralization bottlenecks. To simulate the DCs being geo-distributed, we added a 50ms delay to all messages sent over ZeroMQ. Lost messages are detected at the application level and resent.

Workload generation. The data set used in the experiments includes 100k key-value pairs per partition with each pair being replicated at all 3 DCs. Tests are performed with LWW registers and CRDT sets.

A custom version of Basho Bench [2] is used to generate workloads with clients repeatably running single operation transactions of either a read or an update using a power law distribution over all keys. The ratio of reads and updates is varied depending on the benchmark. For Cure, Eiger, and

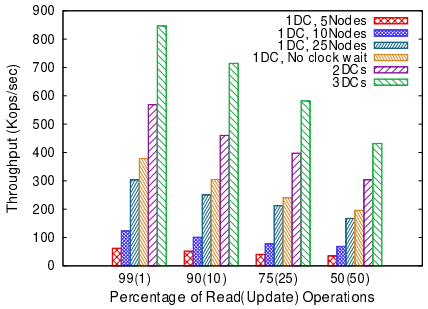
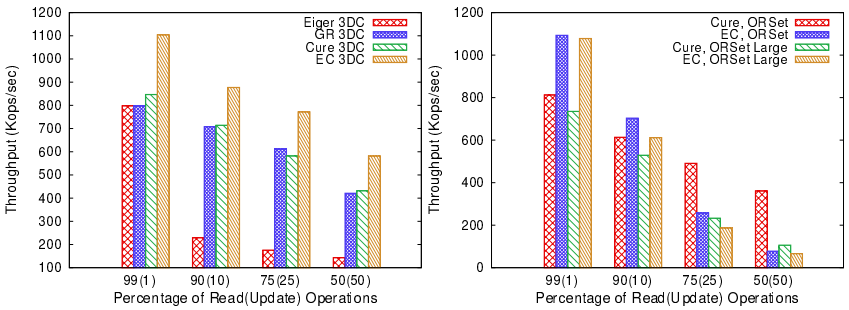


Figure 1: Scalability of Cure



(a) LWW registers (b) CRDT sets

Figure 2: Comparison of Cure to other systems

GentleRain, dependencies for ensuring causality are stored at each client, and are sent with each request and updated on commit. Clients are run on their own physical machines with a ratio of one client server per five Cure servers. We found this ratio to sufficiently load the system without overstressing it for any workload. Each client server uses 40 threads to send requests at full load. Each instance of the benchmark is run for two minutes with the first minute being used as a warm up period. Google's Protocol Buffer interface is used to serialize messages between Basho Bench clients and Antidote servers.

B. Cure's scalability

To evaluate the scalability of Cure we run a DC configuration and vary the number of servers from 5 to 25. Then, we run 2 and 3 DCs configuration with 25 servers each (50 and 75 Cure servers in total). In both cases the read/update ratio is varied from a 99 percent read workload to a 50 percent update workload. Objects are LWW registers of 1 kbyte size values.

As Figure 1 shows, throughput increases 4.8 times when going from 5 to 25 nodes within a single DC under all workloads. Furthermore, on the configurations of 2 and 3 DCs consisting of 25 servers each, we observe a 1.8x and a 2.8x respective increase for 99 percent reads, and 1.8x and 2.6x for 50 percent writes when compared to a single DC with the same number of servers.

The observed scalability is expected due to the decentralized design of Cure. Still, numbers do not show a perfect linear progression due to the cost of replicating updates across DCs and because the background stable time calculation becomes more expensive as the number of servers increases per DC.

For this experiment, the median latency for reads was 0.7 ms for all workloads, and grew between 1 and 2ms for writes when increasing the update rate. Writes are more expensive than reads as they require updating in memory data structures and writing to disk. Additionally, given that updates are replicated 3 times, they create a larger load on the system than reads.

Impact of waiting. In order to evaluate the impact of clock skew in performance, we implemented an unsafe version of Cure that avoids waiting for a snapshot to be ready at a partition receiving a read request (Alg. 2, line 2). The No clock wait bar shows the throughput obtained by this protocol when run at a single DC consisting of 25 servers, which displays up to 1.25x increase when compared to the correct implementation, corresponding to the read dominant workload.

C. Comparison to other systems

To evaluate the performance of Cure when compared to other protocols we run a 3 DC benchmark with 25 servers per DC, varying the update to read ratio. We compare all systems using LWW registers, and Cure to eventual consistency using CRDT sets. Figure 2 shows our results.

LWW registers. We compare all systems using LWW registers of 1 kbytes values each (Figure 2a). Unsurprisingly, eventual consistency performs better than all other protocols, outperforming Cure by approximately 30 percent across all workloads. Under eventual consistency, reads and updates are cheaper as they are single versioned and do not require causal dependency processing.

In the 99 percent read workload, causally consistent systems perform similarly, with Cure slightly outperforming the rest. This happens because at this ratio, the amount of dependency checks performed by Eiger is small. As soon as the update rate is increased to 10 percent, the cost of explicitly checking dependencies increases dramatically and the throughput of Eiger is negatively affected. This trend continues and remains throughout the higher update rate workloads. Cure and GentleRain perform similarly under all workloads. We expect a small additional overhead for GentleRain, which normally needs to retrieve slightly older versions of objects than Cure because of its larger remote update visibility latency, thus incurring extra processing of lists of object versions.

CRDT sets. We compare Cure to eventual consistency systems using CRDT sets (Figure 2b). Being causally consistent, Cure supports operation based CRDTs, where objects

transfer updates among replicas. On the contrary, eventual consistency can only support state-based CRDTs, where replicas synchronise by transferring object state. For this experiment, we use "small" and "big" sets that grow up to 10 and 100 elements of 100 bytes each (1 and 10 kbytes in total), respectively. Once sets reach this size, the workload balances the amount of add and remove operations to keep their average size constant.

For both sizes of sets, we observe a similar behavior. For 99 percent reads, eventual consistency outperforms Cure by a similar amount as with LWW registers. This is expected given that under this read-dominant workload, as updates are rare, systems behave similarly to the previous experiment, and therefore the same conclusions hold. For 90 percent reads, this difference becomes smaller. Finally, with higher update rates, Cure overtakes eventual consistency's performance. This is due to eventual consistency having to process large CRDT state transferred by each update operation at remote DCs (1 and 10 kbytes for small and big sets respectively). Under Cure , replicas transfer light operations (100 bytes when performing an add operation).

Update visibility latency. To calculate the stable time, each node within a DC broadcasts its vector clock to other nodes within the DC at a frequency of 10ms. Additionally, heartbeats between DCs are sent at a rate of 10ms in the absence of updates.

For all experiments, we measured the remote visibility latency observed by DC 1 for updates coming from DCs 2 and 3. Under Cure we observed an average remote update visibility latency of between 80 and 90 ms for updates originating at DCs 2 and 3. Under GentleRain, we observed a visibility latency of 90 ms for both DCs 2 and 3. Moreover, under the update-intensive workloads, we observed frequent short lived peaks of around 150 ms visibility latency for one or both of the DCs due to the cost of processing external updates. Under such conditions, Cure only observed that delay for updates from the affected DC while under GentleRain, the visibility latency of both DCs was penalized under load. The use of a single scalar penalizes GentleRain, which is able to make updates visible at the rate of the slowest DC (§IV-A). By using a vector clock, Cure is able to make updates coming from different DCs visible independently.

VI. Related Work

A large amount of research has explored the consistency vs. availability tradeoff involved in building distributed data stores. As a result, there are a number of systems providing different semantics to application developers. On one extreme of the spectrum, strongly-consistent systems [12, 15, 28] offer well-defined semantics through a simple-to-reason-about transactional interface. Unfortunately, due to the required intensive communication among parties, these solutions penalise latency in the general case and availability in the presence of failures and network partitions. On the other side of the spectrum lies eventual consistency, which includes Dynamo [16], Voldemort [29], Riak [4] and some configurations of Cassandra [22]. These systems offer excellent scalability, availability and low-latency at the cost of providing a model to programmers that is hard to reason about. They lack clear semantics and programming mechanisms (as transactions) that simplify application development. Cure takes an intermediate position in this tradeoff by embracing transactional causal+ consistency semantics.

Recently, a number of causally-consistent, partitioned and geo-replicated data stores were proposed [7, 17, 19, 23, 24]. These solutions offer a variety of limited but interesting transactional interfaces that aim at easing the development of applications. COPS [23] introduced the concept of causally-consistent read-only transactions, which other solutions, such as ChainReaction [7], Orbe [17] and GentleRain [19], adopted. Eiger [24], extended this transactional interface with causally-consistent write-only transactions. Cure provides programmers with stronger semantics, i.e., general transactions and support for confluent data types (CRDTs).

In order to decide when remote updates can be made visible, COPS, Eiger, ChainReaction and Orbe use mechanisms that rely on piggybacking causal dependency information with updates and exchanging explicit dependency check messages among partitions at remote data centres. Even when they employ various optimisations to reduce the size of dependencies and the number of messages, in the worst case their metadata grows linearly with the number of partitions [17]. GentleRain avoids such expensive checks. Instead, it uses a global stabilisation algorithm for making updates visible at remote data centres. This algorithm achieves throughput close to eventually consistent systems, at the cost of increased remote update visibility latency. Cure follows this design choice and achieves similar throughput while providing stronger semantics. Furthermore, by using a vector clock sized with the number of data centers, our protocol is able to reduce remote update visibility latency and is more resilient to network partitions and data centre failures.

Finally, SwiftCloud [30] addresses the challenge of providing causally consistent guarantess and fault-tolerance for client-side applications. Although the semantics provided by SwiftCloud are similar to ours, this work is orthogonal to Cure, since our focus is on making server-side causally consistent systems with rich semantics highly scalable, a problem that is not tackled by SwiftCloud.

VII. Conclusion

We have introduced Cure, a distributed storage system presenting the strongest semantics achievable while remaining highly available. Cure provides a novel programming model: causal+ consistency and CRDT support through an interactive transactional interface.

We have presented a highly-scalable protocol implementation over a partitioned geo-replicated setting. We have evaluated Cure showing that it presents scalability compatible with eventual consistency with both the number of servers per DC and the total number of DCs in the system, while offering stronger semantics. Our results also show that, when comparing Cure to existing causally-consistent systems that provide similar but weaker semantics under different workloads, it presents higher performance while achieving better update visibility latency and tolerance to full DC and network failures.

ACKNOWLEDGMENTS

We would like to thank our shepherd, David Shue, the anonymous reviewers, Christopher Meiklejohn, Michał Jabczyński, Marek Zawirski and Sérgio Duarte for their helpful comments on a previous version of this work. This research is supported in part by European FP7 project 609 551 SyncFree (2013–2016). Nuno Preguiça is partially supported by FCT-MCTES NOVA LINCS project (UID/CEC/04516/2013). Zhongmiao Li and Manuel Bravo are partially supported by the Erasmus Mundus Doctorate Programme under Grant Agreement No. 2012-0030.

REFERENCES

[1] Antidote reference platform. http://github.com/SyncFree/antidote, 2015.

[2] Basho bench. http://github.com/SyncFree/basho_bench, 2015.

[3] The network time protocol. http://www.ntp.org, 2015.

[4] Riak distributed database. http://basho.com/riak/, 2015.

[5] Zeromq. http://http://zeromq.org/, 2015.

[6] AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. W. Causal memory: definitions, implementation, and programming. *Distributed Computing 9*, 1 (Mar. 1995), 37–49.

[7] ALMEIDA, S., LEITÃO, J., AND RODRIGUES, L. ChainReaction: a causal+ consistent datastore based on Chain Replication. In *Euro. Conf. on Comp. Sys. (EuroSys)* (Apr. 2013).

[8] ATTIYA, H., ELLEN, F., AND MORRISON, A. Limitations of highly-available eventually-consistent data stores. In *Symp. on Principles of Dist. Comp. (PODC)* (Donostia-San Sebastián, Spain, July 2015), ACM, pp. 385–394.

[9] BAILIS, P., DAVIDSON, A., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow. 7*, 3 (Nov. 2013), 181–192.

[10] BAILIS, P., FEKETE, A., FRANKLIN, M. J., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Feral concurrency control: An empirical investigation of modern application integrity. In *Int. Conf. on the Mgt. of Data (SIGMOD)* (Melbourne, Victoria, Australia, 2015), Assoc. for Computing Machinery, pp. 1327–1342.

[11] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Scalable atomic visibility with RAMP transactions. In *Int. Conf. on the Mgt. of Data (SIGMOD)* (2014).

[12] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[13] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (1995), SIGMOD '95, ACM, pp. 1–10.

[14] BURCKHARDT, S., FÄHNDRICH, M., LEIJEN, D., AND SAGIV, M. Eventually consistent transactions. In *Euro. Symp. on Programming (ESOP)* (Tallinn, Estonia, Mar. 2012).

[15] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[16] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[17] DU, J., ELNIKETY, S., ROY, A., AND ZWAENEPOEL, W. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symp. on Cloud Computing* (Santa Clara, CA, USA, Oct. 2013), Assoc. for Computing Machinery, pp. 11:1–11:14.

[18] DU, J., ELNIKETY, S., AND ZWAENEPOEL, W. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Symp. on Reliable Dist. Sys. (SRDS)* (Braga, Portugal, Oct. 2013), IEEE Comp. Society, pp. 173–184.

[19] DU, J., IORGULESCU, C., ROY, A., AND ZWAENEPOEL, W. GentleRain: Cheap and scalable causal consistency with physical clocks. In *Symp. on Cloud Computing* (Seattle, WA, USA, Nov. 2014), Assoc. for Computing Machinery, pp. 4:1–4:13.

[20] GILBERT, S., AND LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 2 (2002), 51–59.

[21] GRID'5000. Grid'5000, a scientific instrument [...]. https://www.grid5000.fr/, retrieved April 2013.

[22] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[23] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symp. on Op. Sys. Principles (SOSP)* (Cascais, Portugal, Oct. 2011), Assoc. for Computing Machinery, pp. 401–416.

[24] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)* (Lombard, IL, USA, Apr. 2013), pp. 313–328.

[25] MACKLIN, D. Can't afford to gamble on your database infrastructure? why bet365 chose riak. http://basho.com/bet365/, Nov. 2015.

[26] SAEIDA ARDEKANI, M., SUTRA, P., PREGUIÇA, N., AND SHAPIRO, M. Non-Monotonic Snapshot Isolation. Rapp. de Recherche 7805, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, Nov. 2011.

[27] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. de Recherche 7506, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, Jan. 2011.

[28] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)* (Cascais, Portugal, Oct. 2011), Assoc. for Computing Machinery, pp. 385–400.

[29] SUMBALY, R., KREPS, J., GAO, L., FEINBERG, A., SOMAN, C., AND SHAH, S. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 18–18.

[30] ZAWIRSKI, M., PREGUIÇA, N., DUARTE, S., BIENIUSA, A., BALEGAS, V., AND SHAPIRO, M. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference* (2015), Middleware '15, pp. 75–87.