Cure: Strong semantics meets high availability and low latency

Deepthi Devaki Akkoorath*, Alejandro 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 high-level data types (developer friendly) with safe resolution of concurrent updates (guaranteeing convergence). These guarantees minimise the anomalies caused by parallelism and distribution, and facilitate the development of applications. This paper presents the protocols for highly available transactions, and an experimental evaluation showing that Cure is able to achieve performance 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 geo-replication across several DCs, in order 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 [23] 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: for instance, one of the critical services of the Bet365 application saw its latency decrease by 45×, from 90 minutes to 2 minutes on the same hardware, when moving from MySQL to Riak [30]. However, AP-databases provide only a low-level key-value interface and expose application developers to inconsistency anomalies.

To alleviate this problem, recent work has focused on enhancing AP designs with stronger semantics [27, 28, 33]. 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, *(ii)* transactions, ensuring that multiple keys (objects) are both read and written consistently, in an interactive manner, and *(iii)* high-level replicated data types (CRDTs) such as counters, sets, tables and sequences, with an inuitive semantics and guaranteed convergence even in the presence of concurrency and partial failures.

Causal consistency [6, 27] represents a sweet spot in the availability-consistency tradeoff. It is the strongest model compatible with availability [8] for individual operations. As it ensures that the causal order of the 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 at the application level; with causal consistency, this cannot happen.

Performing multiple operations in a transaction enables to maintain relations between multiple objects or keys. The operations in a transaction should read from the same *snapshot*, and any database state contains either all updates of a given transaction, or none *(atomicity)*. *Highly Available Transactions* (HATs) eschew the traditional isolation property, which requires synchronisation, in favour of availability [9, 15]. Existing HAT implementations provide either snapshots [7, 20, 22, 27, 28] or atomicity [11, 28]; Cure transactions have both. A Cure transaction is *interactive*, i.e., the objects it accesses need not be known in advance, unlike the above mentioned recent systems [33]. This is desirable for scenarios where a user first looks up some information, then acts upon it: with non-interactive transactions, this requires two separate transactions. For instance, Alice who is enrolling in university, selects a course from list of available courses, then checks if Sally is enrolled in it. A non-interactive system requires two transactions, and second one could fail if the course is not available any more, whereas a single interactive transaction would guarantee both actions occur or neither.

Cure supports conflict-free replicated data types (CRDTs) [32]. CRDTs are guaranteed to converge despite concurrent updates and failures. CRDT abstractions such as counters, sets, maps/tables and sequences are more developer-friendly, but require more underlying mechanism, than the usual key-value interface with last-writer-wins conflict resolution. For

$x, y \in \mathcal{O}$	Objects
T_i, for $i \in \mathbb{N}$	Transaction with unique id i
$u_i(x) \in \mathcal{U}$	Transaction T_i updates x
$r_i(x)$	Transaction T_i reads x
$\mathcal{S} \subseteq \mathcal{U}$	Snapshot of the replica
$\mathcal{S}_r(i)$	Snapshot read by T_i
$ws(i)$	Updates of T_i

Table I: Notation.

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 [29].

Combined, the above features make life much easier for developers. The developer may attempt to compensate for their absence with ad-hoc mechanisms at the application level, but this is tricky and error prone [10].

Finally, Cure implements a new approach to support parallelism between servers within the data centre, and to minimising the overhead for causal consistency in inter-DC traffic. Instead of the usual approach of checking whether a received update satisfies the causality conditions, which requires to wait for a response from a remote server, Cure makes updates visible in batches that are known to be safe [22]. Cure improves on previous work by represent causal dependencies as a single scalar per DC, thus reducing overhead, improving freshness and improving resilience to network partitions.

The contributions of this paper are the following:

- a novel programming model, based on highly-available transactions, with interactive reads and writes and high-level, confluent data types (§III);
- a high-performance protocol, supporting this programming model for geo-replicated datastores (§IV);
- a comprehensive evaluation, comparing our approach to state-of-the-art data stores (§V).

II. Background and Definitions

We introduce theoretical background and definitions supporting our system.

We first define causal consistency, and later extend the definition to transactions, that combines read and update operations. Finally, we discuss convergent conflict handling and the CRDT approach to address it. TableI shows the notations used.

A. Causal Consistency

For two operations a and b, we say b causally depends on a (or a is a causal dependency of b), expressed $a \rightsquigarrow b$, if any of the following conditions hold:

- *Thread-of-execution*: a and b are operations in a single thread of execution, and a happens before b.
- *Reads-from*: a is a write operation, b is a read operation, and b reads the value written by a.
- *Transitivity*: If $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

Accordingly, we say that a data store is causally consistent if, when an update is visible to a client, all causal dependencies of this update operation are also visible. For example, if at a given replica, Bob, a user of a social network, removes Alice from his friend's list and later makes a post on his wall (Alice's removal $\rightsquigarrow$ post), then Alice cannot see this post as the removal from the friend list causally precedes the posting on the wall.

B. Transactional Causal Consistency

For Cure, we consider a transactional variant of causal consistency to support multi-object operations: all the reads of a causal transaction are issued on the same causally consistent snapshot, and either all of a transaction's updates are visible, or none is.

A transaction T_i executes on a snapshot $\mathcal{S}_r(i)$, namely $\forall x, r_i(x)$ returns a set $\{u_j(x) | u_j(x) \in \mathcal{S}_r(i)\}$.

The set of all consistent snapshots is denoted as Σ_{CONS}. A snapshot $\mathcal{S} \in \Sigma_{\text{CONS}}$ if and only if it satisfies the following properties:

- *Atomicity*: $u_j(x) \in \mathcal{S} \Rightarrow \forall u_j(y) \in ws(j), u_j(y) \in \mathcal{S}$
- *Consistency*: If an update is included in a snapshot, all it causal dependencies are also in the snapshot.

$$\forall u_i(x) \in \mathcal{S}, u_j(y) \rightarrowtail_u u_i(x) \Rightarrow u_j(y) \in \mathcal{S}$$

A system of replicas is transactionally causal consistent if all transactions execute on a consistent snapshot: $\forall T_i, \mathcal{S}_r(i) \in \Sigma_{\text{CONS}}$.

C. Convergent conflict handling

Concurrent operations are not ordered under causal consistency; two causally unrelated operations can be replicated in any order in a weakly consistent system such as Cure, avoiding the need for synchronisation. If two concurrent operations, however, update the same key, then they are in conflict and can lead to divergence of the system. Such a condition usually requires ad-hoc handling in the application logic [19, 31, 35].

Alternatively, an automatic conflict handling mechanism can be employed to handle conflicts deterministically, in the same fashion, at every replica. Many existing causal+ consistent systems adopt the last-writer-wins rule [22, 27, 28], where the updates occurred "last" overwrites the previous ones.

We rely on CRDTs, high-level data types that guarantee confluence and have rich semantics [16, 32]. Operations on CRDTs are not register-like assignments, but methods corresponding to a CRDT object's type. CRDTs include sets, counters, maps, LWW registers, lists, graphs, among others. As an example, a set supports *add(item)* and *remove(item)* operations. The implementation of a CRDT set will guarantee that no matter in which order add and remove occur, the state of the set will converge at different replicas.

III. Cure Overview

We assume a distributed key-value data store that handles a very large number of data items, namely objects. The full set of objects, namely key-space, is replicated across different geo-locations to provide availability, low latency, and scalability. We refer to each of this geo-locations as data centre. There is a total of D data centres. Each of the data centres is partitioned in N parts. p_j^m denotes partition m of data centre j. Thus, each partition within a data centre is responsible for a non overlapping subset of the key-space. We assume that every data centre employs the same partitioning scheme. Clients interact through the key-value data store.

The rest of the section firstly described the goals of Cure and briefly explains how we tackle them in its design. We then describe the system architecture and, finally, we present Cure programming interface.

A. Cure Goals and System Overview

Cure's design aims at enhancing the scalability of the system in two dimensions: (i) as the number of machines increase within a data centre, and (ii) as the number of data centres increases.

To meet this scalability goal, our protocol adopts two key design decisions. First, the protocol decouples the inter-data-centre synchronisation from the intra-data-centre visibility mechanism. The main idea is to avoid using a centralised timestamp authority which would concentrate a large amount of load and become a potential bottleneck. Note that for a transactional causally consistent system, this centralized timestamp authority would be responsible for assigning commit timestamps and making both remote and local updates visible, in a causally consistent manner while enforcing isolation and atomicity of transactions. Our protocol completely eliminates the centralized timestamp authority and distributes the responsibility of the previously listed tasks across the partitions. Second, it has been empirically demonstrated that basing the implementation of causal consistency on explicit dependency check messages substantially penalizes system performance [22]. Thus, our protocol relies on a global stabilization mechanism that supplies partitions with enough information to take decision locally without violating causality.

The Cure protocol is inspired by Clock-SI [21], which we extended and modified to achieve Transactional Causal+ Consistency in a partitioned and geo-replicated system with support for high-level data types with rich confluent semantics (CRDTs). Furthermore, the global stabilization mechanism used by the protocol is inspired by the one proposed in GentleRain [22]. In addition, we extend the metadata used to enforce causal consistency to improve data freshness under network partitions and data centre failures.

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], and each clock generates monotonically increasing timestamps. The correctness of the protocol does not depend on the synchronization precision.

B. Architecture

Each partition in Cure mainly consists of the following four components (Figure 1):

- **Log:** This module implements a log-based persistent layer. Updates are stored in a log, which is persisted to disk for durability. The module also internally maintains a cache layer in order to make accesses to the log faster.
- **Materializer:** This module is responsible for generating the versions of the objects out of the updates issued by clients. The module is placed between the *Log* and the *Transaction Manager* modules. The goal of the *Materializer* is to have the versions required by the *Transaction Manager* prepared before they are requested. The implementation of this module is cumbersome due to the extra difficulties posed by the integration of CRDTs into a multi-versioning data store. The module also incorporates some pruning mechanisms in order to avoid penalizing system's performance over time.
- **Transaction Manager:** This module implements the transactional protocol of Cure (precisely described in §IV). It receives client's requests, executes and coordinates transactions, and replies back to clients with the outcome. It communicates to the materializer in order to read objects and write new updates.
- **InterDC Replication:** This module is in charge of fetching updates from the *Log*, and propagating them to other data centres. Communication is done partition-wise.

C. Programming Interface

Cure offers a transactional interface to clients with 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.
- `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.
- `update_objects(Updates, TxId)` declares a list of `Updates` for a transaction.
- `commit_transaction(TxId)` commits the transaction under transaction handler `TxId` and makes the updates visible.
- `abort_transaction(TxId)` discards the updates and aborts the transaction.

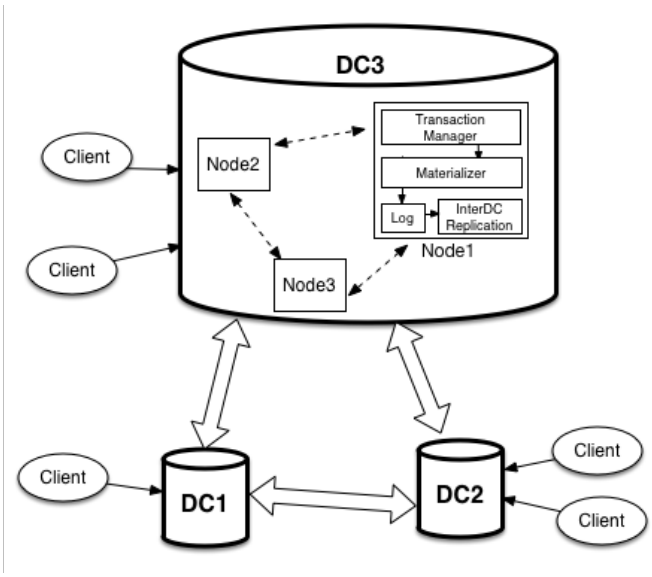

Figure 1: Cure's Architecture

pc_j^m	current partition's physical clock
pvc_j^m	time denoting the latest snapshot available in p_j^m
ss_j	$\phi(s)$, where s is the latest snapshot available in all partitions of DC j

Table II: Partition p_j^m state

Reads and updates can be composed in an arbitrary way, thus supporting flexible and dynamic interaction with the system.

IV. Protocol Description

In this section we present the transactional protocol used by Cure. The protocol assigns to all consistent snapshots an id $\phi(\mathcal{S}) \in \mathcal{T}$, $\phi : \Sigma_{\text{CONS}} \rightarrow \mathcal{T}$. $\mathcal{T} = \{v | v$ is a vector of size $D\}$.

Each partition p_j^m maintains the following state (Table II): a vector clock pvc_j^m that denotes the latest snapshot available at partition m in DC j; a $pvc_j^m[j]$ which corresponds to the partition's local data centre j and its value is assigned from partition's physical clock. Furthermore, each partition maintains another vector clock ss_j that denotes the latest consistent snapshot that p_j^m knows to be available in all partitions of its data centre j. The state maintained by each transaction is summarized in Table III.

Transactions are executed in two steps: 1) A consistent read algorithm guaranteeing that reads are from consistent snapshot; 2) a transaction execution and replication protocol ensuring that the vector clock of each partition pvc_j^m is correct.

The protocol implements a multiversioning system: If snapshot $\mathcal{S}$ is available in a replica then $\forall \mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{S}'$ is also available.

$T.vs \in \mathcal{T}$	vector clock representing causally preceding snapshot of T
$T.dc \in \{1..D\}$	DC in which T was executed and committed
$T.c \in \mathcal{T}$	vector clock which denotes the commit-time of T.

Table III: Transaction T state

A. Read protocol

The read protocol provides causally consistent snapshots to transactions across multiple partitions. There are two main aspects to discuss: (i) the assignment of transaction's consistent snapshot, and (ii) safely making snapshots available at partitions.

Assignment of consistent snapshot. A transaction is first assigned a snapshot time denoted by $T.vs$ (Alg. 1 line 13). This snapshot is carefully chosen not to violate causal consistency across transactions of the same client. Thus, client StartTransaction request piggybacks the largest snapshot seen by that client (Alg. 1 line 9). The protocol makes sure that the snapshot being assigned to the new transaction includes all previous snapshots seen by the client (Alg. 1 line 5).

Safe reads. When a partition m receives a read request, it has to make sure that the requested snapshot is available locally. A snapshot being available means that once the key's snapshot version is returned no further transactions can commit in that or in a previous snapshot. Our protocol masks this into the pvc_j^m. Thus, it waits until its $pvc_j^m \geq T.vs$ (Alg 2, line 6). What remains to discuss is how the protocol ensures that pvc_j^m only denotes available snapshots. The next subsection addresses this question.

Note that if a transaction reads from multiple partitions, it reads from the partial snapshots of the same snapshot, which is identified by $T.vs$. Thus the snapshot observed by the transactions $\in \Sigma_{\text{CONS}}$.

B. Maintaining Correctness of pvc_j^m

In order to guarantee that the read protocol always serves consistent snapshots, pvc_j^m has to be updated correctly, when a transaction is successfully committed and updates from a remote DC are received.

Definition 1. pvc_j^m is said to be *correct* if, $\forall \mathcal{S} \in \Sigma_{\text{CONS}}$: $\phi(\mathcal{S}) \preceq pvc_j^m \Rightarrow p^m(\mathcal{S}) \subseteq \mathcal{S}(p_j^m)$. $\mathcal{S}(p_j^m)$ denotes the partial snapshot available currently in p_j^m.

Local Commit Protocol. Cure uses a two-phase-commit protocol. First, the coordinator requests prepare timestamps from all partitions involved in the transaction (Alg. 1 lines 29-32). Each partition m replies with the current value of pc_j^m as its proposed commit-timestamp for T_i denoted by $pt_j^m(i)$ (Alg. 2 line 11). The local commit-time lc_i is calculated as $\max(pt_j^m(i)) \forall p^m \in T_i$.UpdatedPartitions (Alg. 1

Algorithm 1 Transaction coordinator TC in partition k, DC j

```
1:  function GETSNAPSHOTTIME(Clock cc)
2:      for all i = 0..D − 1, i ≠ j do
3:          vs[i] = ss_j[i]
4:      end for
5:      vs[j] = max(pc_j^k, cc[j])
6:      return vs
7:  end function
8:
9:  function STARTTRANSACTION(Transaction T, Clock cc)
10:     for all i = 0..D − 1, i ≠ j do
11:         wait until cc[i] ≤ ss_j[i]
12:     end for
13:     T.vs = GETSNAPSHOTTIME(cc)
14:     return T
15: end function
16:
17: function UPDATE(Transaction T, Key k, Operation u)
18:     p = partition(k)
19:     T.UpdatedPartitions = T.UpdatedPartitions ∪ {p}
20:     send EXECUTEUPDATE(T, k, u) to p
21: end function
22:
23: function READ(Transaction T, Key k)
24:     p = partition(k)
25:     send READKEY(T, k) to p
26: end function
27:
28: function DISTRIBUTEDCOMMIT(T)
29:     for all p ∈ T.UpdatedPartitions do
30:         send PREPARE(T) to p
31:         wait until receiving (T, prepared, timestamp) from p
32:     end for
33:     CommitTime = max(received timestamps)
34:     T.c = T.vs
35:     T.c[j] = CommitTime
36:     T.dc = j
37:     for all p ∈ T.UpdatedPartitions do
38:         send COMMIT(T) to p
39:     end for
40: end function
```

line 33). The commit-time of the transaction is generated from $T.vs$ and setting its j^{th} entry to lc_i. The coordinator then sends commit messages to T_i.UpdatedPartitions. $ws(i)$ is then included in all snapshots with id $\geq T_i.c$.

Note that when a partition p_j^m receives a read/update request from the transaction coordinator of T_i, it waits until $pvc_j^m \geq T_i.vs$ before executing the request (Alg. 2 line 2). This ensures that no transaction which has not started its commit phase and that involves p_j^m will ever commit in a snapshot which is suppose to be included in T_i read snapshot.

What remains to prove, regarding the correctness of pvc_j^m, is whether transactions which are already in their commit phase and involve p_j^m may commit before T_i read snapshot. The new pvc_j^m is calculated by the function UpdateClock (Alg. 2, line 21). The minimum of prepared timestamps of the transactions in prepared phase is identified. Since the physical clock is monotonically increasing, it is guaranteed that new prepare requests would receive newer clock times. Hence no new transaction will commit in partition p_j^m with a time less than the current minimum prepared timestamp. Thus, setting the $pvc_j^m[j]$ to the minimum prepared timestamp guarantees that $\forall T_i, T_i.c \leq pvc_j^m \Rightarrow ws(i) \subset \mathcal{S}(p_j^m)$. Thus pvc_j^m is correct.

Replication Algorithm. Once a transaction is committed locally, its update are asynchronously replicated to other data centres. A partition p_j^m sends updates to p_k^m independently of other partitions. A partition is responsible to send only the updates to the keys that are responsible to that partition. T_i/p^m denotes the set of updates of T_i that belongs to p^m, i.e. $\{u_i(x)|x \in p^m\}$.

The updates are send in $T.c[j]$ order. When p_k^m receives updates T_i/p^m from p_j^m, T_i/p^m is put in to a queue $q_k^m[j]$. A transaction $T = head(q_k^m[j])$ is applied if $T.vs \leq pvc_k^m$. This guarantees that the causal dependency is satisfied and the new partial snapshot is consistent. Since the updates are send in the order: $\nexists T : pvc_k^m[j] < T.c[j] < T_i.c[j] \wedge (T \notin q_k^m[j] \vee T$ is not already applied. So we can safely set $pvc_k^m[j]$ to $T.c[j]$.

The local transaction execution at p_j^m updates the entry $pvc_j^m[j]$ correctly, and the replication algorithm updates the entry for remote data centres correctly whenever it receives updates from other data centres. Thus it is guaranteed that whenever $pvc_j^m = s$, a transaction can safely read a consistent snapshot with id $\leq s$, thus guaranteeing Transactional Causal Consistency.

C. Extensions

We have extended the basic protocol as follows in order to guarantee progress, wait-free snapshot reads and guarantee monotonic reads even when clients connects to different data centres.

First we introduce stable snapshot ss_j which denotes the latest snapshot available at all partition in data centre j. ss_j is calculated in Alg 4, lines 22-26. As we have seen in the replication protocol, each partition is replicated independently of other partitions. This may result in some partitions having more recent snapshots and other partitions having older snapshots. This might require a transaction which reads a later snapshot from a partition to wait until it can read from another partition which has an older snapshot. In order to avoid this waiting, the transaction's snapshot time is assigned from ss_j. Since the clocks of partitions within the same data centre are less likely to be out of sync for a long time, we can assign $T.vs[j]$ to be the physical clock of the partition running the transaction coordinator, so that T can see the latest committed transactions in data centre j.

Secondly, in case of data centre failures, clients can optionally move to other data centre and provide its cc, which is the last observed snapshot by the client. If a

Algorithm 2 Transaction execution at partition m, DC j

```
1:  function ExecuteUpdate(Transaction T, Update u)
2:      wait until T.vs[j] ≤ pc_j^m
3:      log u
4:  end function
5:  function ReadKey(Transaction T, Key K)
6:      wait until T.vs[j] ≤ pvc_j^m[j]
7:      return snapshot(K, T.vs)
8:  end function
9:
10: function Prepare(Transaction T)
11:     prepareTime = pc_j^m
12:     preparedTransactions_j^m.add(T, prepareTime)
13:     send (T, prepared, prepareTime) to T's coordinator
14: end function
15:
16: function Commit(transaction T)
17:     log (T, commit, T.c, T.vs)
18:     preparedTransactions_j^m.remove(T)
19: end function
20:
21: function UpdateClock
22:     if preparedTransactions_j^m ≠ ∅ then
23:         timestamps = Get prepare timestamps in
    preparedTransactions_j^m
24:         pvc_j^m[j] = min(timestamps) − 1
25:     else
26:         pvc_j^m[j] = pc_j^m
27:     end if
28: end function
```

Algorithm 3 Replication Algorithm at the sender, running in partition p_i^m

```
1:  function ReplicateToDC(j)
2:      loop
3:          t = pvc_i^m[i]
4:          Transactions = {T | T.c[i] ≤ t, T.dc = i, not
    propagated to DC j yet }
5:          if Transactions = ∅ then
6:              heartbeat = new Transaction()
7:              heartbeat.c[i] = t
8:              heartbeat.dc = i
9:              heartbeat.vs = pvc_i^m
10:             send heartbeat to p_j^m
11:         else
12:             sort Transactions in ascending order of T.c[i]
13:             send Transactions to DC j
14:         end if
15:     end loop
16: end function
```

client clock cc is provided, the transaction coordinator would wait until the stable snapshot of data centre j has reached cc, ensuring that clients will always observe monotonically increasing snapshots.

Third, if a partition p_j^m does not execute any new transaction for a long time, the remote partition p_k^m's entry $pvc_k^m[j]$ will not be increased, resulting in a large gap compared to vector clocks of other partitions. Then the value of $ss_k[j]$ will not be updated, resulting in transactions reading old snapshots. In order to avoid this, partitions send a periodic heartbeat message with their latest pvc_j^m to other remote partitions.

Algorithm 4 Replication Algorithm at the receiver, running in partition p_j^m

```
1:  queue[i]: A queue for transactions received from DC i
2:
3:  function ReceiveTransaction(ListofTransactions Trans-
    actions, DC i)
4:      for all T in Transactions do
5:          enqueue(queue[i], T)
6:      end for
7:  end function
8:
9:  function ProcessQueue(i)
10:     ▷ This function is repeatedly called to process transactions
    from DC i
11:     T = getFirst(queue[i])
12:     T.vs[i] = 0
13:     if T.vs ≤ pvc_j^m then
14:         if T is not a heartbeat then
15:             log T
16:         end if
17:         pvc_j^m[i] = T.c[i])
18:         remove T from queue[i]
19:     end if
20: end function
21:
22: function CalculateStableSnapshot
23:     for all j = 0..D − 1 do
24:         ss_i[j] = min_{k=0...P−1} pvc_i^k[j]
25:     end for
26: end function
```

V. Evaluation

In this section, we evaluate Cure's implementation in terms of latency, throughput, and remote update visibility latency under different workloads and number of partitions. We compare Cure to Eventual Consistency (EC) and an implementation of state-of-art causal consistency, i.e., Eiger [28] and GentleRain [22].

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 for comparison. Our implementation of Eventual consistency is a single-versioned

key-value store, supporting last-write wins registers, where the ordering of concurrent updates is determined by local physical clocks. Eiger supports causal consistency using last-write wins 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 last-write wins registers, Cure and GentleRain support CRDT objects. All three causally consistent protocols provide (static) read-only and atomic update transactions, with Cure additionally supporting interactive read and update transactions as described in section III-C.

Objects in Cure, Eiger, and GentleRain are multi-versioned. For each key, a linked-list of recent updates and snapshots is stored in memory with old versions being 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 [24] experimental platform using dedicated servers. Each server runs 2 Intel Xeon E5520 CPUs with 4 cores/CPU, 24GB RAM, and 119GB SSDs for storage. Nodes are connected through shared 10Gbps switches with average round trip latencies of around 0.5ms. NTP is run between all machines to keep physical clocks synchronized [3].

All experiments are run using 3 DCs with a variable number of servers per DC. 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-located we add 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 ten thousand key-value pairs per partition (where the number of partitions is equal to the number of servers per DC) with each pair being replicated at all 3 DCs. All objects are last-write wins registers with eight byte keys and ten bytes values.

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 uniform random distribution over all keys. The ratio of reads and updates is varied depending on the benchmark. For Cure, Eiger, and 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 three Antidote servers with each client server using 120 processes 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.

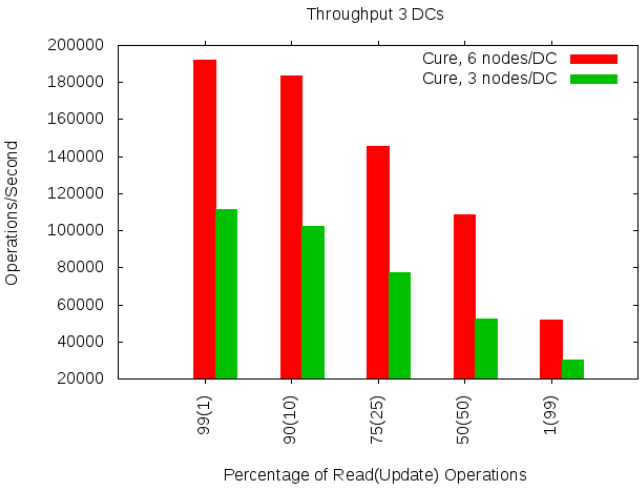

Figure 2: Scalability of Cure using 3 DCs with 3 and 6 nodes per DC

B. Cure's scalability

To evaluate the scalability of Cure (figure 2) we run a 3 DC configuration with 3 and 6 servers per DCs (9 and 18 Cure servers in total). In both cases the read/update ratio is varied from a 99 percent read workload to a 99 percent update workload. For all workloads Cure scales approximately linearly when going from 3 to 6 nodes. Specifically in the 99 percent read workload the throughput increases from 111196 ops/second to 192208 ops/sec (a $1.73x$ increase) and in the 99 percent update workload the throughput increases from 30326 ops/sec to 51971 ops/sec (a $1.71x$ increase). This increase in scalability is expected as transactions are completely distributed and only the stable time calculation becomes more expensive as the number of servers increases. To calculate the stable time each node within a DC broadcasts it vector to the nodes within the DC at a frequency of 100ms. Additionally, heartbeats between DCs are sent at a rate of 100ms in the absence of updates. Using these intervals, most updates become visible in external DCs after approximately 200ms.

The median latency for reads is about 1ms for all workloads with the latency for writes increasing from 3ms to 5ms as the update ratio increases. The writes are more expensive than the reads as they require both updating in memory data structures and writing to disk. Additionally, given that all

updates are replicated 3 times they create a much larger load on the system than reads.

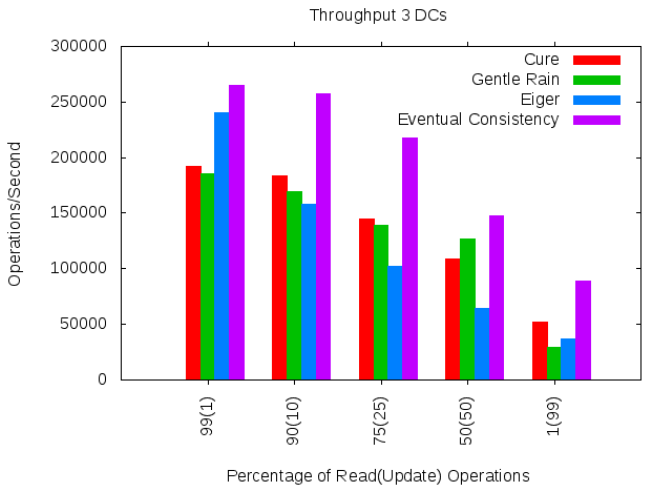

Figure 3: Comparison of Cure with other systems using 3 DCs

C. Comparison to other systems

To evaluate the performance of Cure (figure 3) when compared to other protocols we run a 3 DC benchmark with 6 servers per DC, varying the update and read ratio. Unsurprisingly eventual consistency performs better than all other protocols in all workloads, outperforming Cure by approximately 30 percent across all workloads. Both reads and updates are cheaper in eventual consistency as they are single versioned and do not require causal dependency calculations.

In the 99 percent read workload Eiger outperforms both Cure and GentleRain, achieving performance closer to eventual consistency. Both Cure and GentleRain have the overhead of periodically calculating the stable time and well as calculating calculating slightly stale versions of objects, which is not necessary in Eiger.

As soon as the update rate is increased to 10 percent, the cost of explicitly checking dependencies overtakes the cost of calculating the stable time and the throughput of Eiger drops below that of the other protocols. This trend continues and remains throughout the high update rate workloads. At the 50 percent update workload Eiger performs approximately 40 percent slower than Cure but it catches up in the 99 percent update workload to be only 30 percent slower. The reason for this is that Eiger tracks dependencies only up to the previous update, thus mostly only needing a single dependency check per update here, while requiring at least two in the 50 percent update workload.

When comparing the performance of Cure and GentleRain, they are mostly comparable, with Cure performing better in four of the five workloads. One interesting point though is that GentleRain performs worse than Eiger in the 99 percent update workload, we expect this is due to GentleRain needing to compute slightly older snapshot versions of objects than Cure because of its larger remote update visibility latency (to support CRDTs we generate the correct version of the object before updating it), increasing the cost of multi-versioning by a significant factor when the list of versions is being modified frequently. This update visibility latency is described in more detail in the following section.

D. Remote update visibility latency

As described previously, Cure and GentleRain use a stabilization mechanism to make updates visible at remote data centres while respecting causality. GentleRain uses a scalar global stable time and Cure a version vector.

We define the *visibility latency* of an update operation as the amount of time elapsed from the moment it is committed at its local DC (its commit time), and the time at a remote's replica server when the computation of the stable time at that DC allows that update to be safely read without violating causality. The use of a single scalar penalizes GentleRain in that, in order to make an update that originated at remote DC visible locally it must wait until it hears from all other servers at all other DCs with a heartbeat with a time greater than that of the update. By using a vector clock to time-stamp events, Cure is able to make a remote update from a DC i with a commit vector clock vcc visible when servers in j have received all updates up to $vcc[i]$ from replicas at DC i at the cost of slightly increase meta-data. In other words, in GentleRain, update visibility latency at a DC is dependent on the latency *to the furthest DC*. In contrast, in Cure it is *only dependent on* the latency to the DC where the update originated. In our evaluation, all DCs have a 50ms delay between them, thus visibility is only minimally increased in GentleRain when compared to CureṪhe heartbeats within DCs are broadcast at an interval of 100ms and heartbeats between DCs are sent at a rate of 100ms in the absence of updates. Using these intervals, most updates become visible in external DCs after approximately 200ms in Cure and between 200ms and 300ms in GentleRain.

1) Progress in the presence of network failures: As explained before, the use of a single scalar limits GentleRain to make updates visible that, in the presence of a DC failure or network partition between DCs, will stop making remote updates from *every* remote DC visible. Hence, under this situation, the state a DC is able to see from remote DCs will freeze until the system recovers from the failure, while local updates will continue to be made visible. In the case of Cure updates from healthy DCs can continue to be made visible under network partitions, and *only updates from the failed or disconnected DC* remain frozen until the system recovers.

VI. Related Work

A large amount of research has been destined to understanding consistency vs. availability tradeoff involved in building distributed data stores. As a result, there is a number of systems providing different semantics to application developers. On one extreme of the spectrum, strongly-consistent systems [12, 17, 18, 33, 36] offer well-defined semantics through a simple-to-reason-about transactional interface. Unfortunately, due to the intensive communication among parties required, these solutions penalise latency in the general case and availability in presence of failures and network partitions. Systems such as Spanner [17], Walter [33], Calvin [36] and Granola [18], aim at reducing the inter-datacentre synchronisation. However, none of them is yet capable of achieving low-latency operations. On the other side of the spectrum, there is eventual consistency, of which examples are Dynamo [19], Voldemort [34], Riak [4] and some configurations of Cassandra [26]. 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. Many previous system designers have acknowledged the usefulness and applicability of causal consistency. Early examples that had a profound impact in research are the ISIS toolkit [14], Bayou [31], lazy replication [25], causal memory [6] and PRACTI [13]. Unfortunately, these solutions are limited to single-machine replicas which make them not scalable to large geo-replicated data centres.

More recently, a number of causally-consistent, partitioned and geo-replicated data stores have been proposed [7, 20, 22, 27, 28]. These solutions offer a variety of limited, but interesting, transactional interfaces that aim at easing the development of applications. COPS [27] introduced the concept of causally-consistent read-only transactions, which other solutions, such as ChainReaction [7], Orbe [20] and GentleRain [22], adopted. In Eiger [28], this read-only transactional interface was extended with the introduction of the causally-consistent write-only transaction concept. Cure provides programmers with stronger semantics, i.e., general transactions and support for confluent data types (CRDTs).

Implementing causal consistency comes with a cost. Causally-consistent systems need to perform different verifications at each site in order to decide when updates coming from remote datacentres can be made visible without breaking the rules of causality. COPS, Eiger, ChainReaction and Orbe use different mechanisms that rely on piggybacking dependency information with propagated updates and exchanging explicit dependency check messages at remote datacentres. Even when they employ various optimisations to reduce the number of the size of dependencies and the number of messages, their worst-case behaviour remains linear in the number of partitions [22]. In recent work, Du et al. identified that most of the overhead introduced by implementing causal consistency in previous solutions comes from explicit dependency check messages [22]. Their work presented GentleRain, a system that avoids such expensive checks. Instead, it uses a global stabilisation algorithm for making updates visible at remote datacentres. This algorithm increases throughput, reaching numbers close to eventually consistent systems, at the cost of penalising remote update visibility. Cure follows this design choice and achieves throughput close to eventual consistency, while providing stronger semantics. Furthermore, as we have shown in §V, by versioning objects using a vector clock sized with the number of datacentres, our protocol is able to reduce remote update visibility latency and is more resilient to network partitions and datacentre failures when compared to GentleRain.

Finally, SwiftCloud [37] 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.

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] BELARAMANI, N., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. Practi replication. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3* (Berkeley, CA, USA, 2006), NSDI'06, USENIX Association, pp. 5–5.

[14] BIRMAN, K. P., AND RENESSE, R. V. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

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

[16] BURCKHARDT, S., GOTSMAN, A., YANG, H., AND ZAWIRSKI, M. Replicated data types: specification, verification, optimality. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 271–284.

[17] 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.

[18] COWLING, J., AND LISKOV, B. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 21–21.

[19] 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.

[20] 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.

[21] DU, J., ELNIKETY, S., AND ZWAENEPOEL, W. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems* (2013), SRDS '13, pp. 173–184.

[22] DU, J., IORGULESCU, C., ROY, A., AND ZWAENEPOEL, W. GentleRain: Cheap and scalable causal consistency with physical clocks. In *Symp. on Cloud Computing* (New York, NY, USA, Nov. 2014), ACM, pp. 4:1–4:13.

[23] 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.

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

[25] LADIN, R., LISKOV, B., SHRIRA, L., AND GHEMAWAT, S. Providing high availability using lazy replication. *Trans. on Computer Systems 10*, 4 (Nov. 1992), 360–391.

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

[27] 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.

[28] 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.

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

[30] MACKLIN, D. Private communication. Nov. 2015.

[31] PETERSEN, K., SPREITZER, M., TERRY, D., AND THEIMER, M. Bayou: Replicated database services for world-wide applications. In *Proceedings of the 7th Workshop on ACM SIGOPS European Workshop: Systems Support for Worldwide Applications* (New York, NY, USA, 1996), EW 7, ACM, pp. 275–280.

[32] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)* (Grenoble, France, Oct. 2011), X. Défago, F. Petit, and V. Villain, Eds., vol. 6976 of *Lecture Notes in Comp. Sc.*, Springer-Verlag, pp. 386–400.

[33] 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.

[34] 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.

[35] TERRY, D. B., DEMERS, A. J., PETERSEN, K., SPREITZER, M. J., THEIMER, M. M., AND WELCH, B. B. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on* (1994), IEEE, pp. 140–149.

[36] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 1–12.

[37] 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.