Towards Adaptive Transactional Consistency for Georeplicated Datastores

RUI BRAGA, U. Minnho and INESCTEC, Portugal
JOSÉ PEREIRA, U. Minnho and INESCTEC, Portugal
FÁBIO COELHO, U. Minnho and INESCTEC, Portugal

Developers of data-intensive georeplicated applications face a difficult decision when selecting a database system. As captured by the CAP theorem, CP systems such as Spanner provide strong consistency that greatly simplifies application development. AP systems such as AntidoteDB providing Transactional Causal Consistency (TCC), ensure availability in face of network partitions and isolate performance from wide-area round-trip times, but avoid lost-update anomalies only when values can be merged. Ideally, an application should be able to adapt to current data and network conditions by selecting which transactional consistency to use for each transaction. In this paper, we test the hypothesis that a georeplicated database system can be built at its core providing only TCC, hence, being AP, but allow an application to execute some transactions under Snapshot Isolation (SI), hence CP. Our main result is showing that this can be achieved even when all the interaction happens through the TCC database system, without additional communication channels between the participants. A preliminary experimental evaluation with a proof-of-concept implementation using AntidoteDB shows that this approach is feasible.

CCS Concepts: • **Information systems → Parallel and distributed DBMSs**; **Database transaction processing**; • **Computer systems organization → Cloud computing**; **Availability**.

Additional Key Words and Phrases: Transactions, georeplication, CAP theorem

1 Introduction

Large-scale data-intensive applications as pioneered by global cloud platforms are increasingly sought for a diversity of purposes, ranging from e-Commerce to scientific repositories. A key technological enabler for these applications is georeplication, that is, the ability to maintain and use updated copies of data in various geographically distributed data centers, closer to data producers and consumers, for higher availability and performance.

Georeplication implies challenges in system design, considering that network partitions are inevitable and, according to the CAP (Consistent Available Partitionable) theorem [14], either consistency or availability must be sacrificed, essentially choosing the lesser of two evils. Giving priority to consistency (CP – Consistent Partitionable), as in Spanner [15] that provides serializability, will result in downtime in the presence of network partitions. Giving priority to availability (AP – Available Partitionable), as in Dynamo [16] has performance benefits, but results in data anomalies due to neglecting data consistency.

Transactional Causal Consistency, also known as *causal+ transactions* [9, 27] provides a better balance, but can still result in anomalies such as *lost updates*. Briefly, this anomaly occurs when two concurrent transactions read the same value (e.g., the availability of a resource) and write it back after being updated (e.g., decremented by one unit after the resource is used). In this case, one of the operations would be lost (e.g., only one usage would be recorded). Systems such as AntidoteDB [8] work around this with CRDTs (Conflict-free Replicated Data Types) [25] to reconcile concurrent updates, but this cannot cope with all cases (e.g., allocating the last available resource). Using a serializable transactional CP system such as Spanner for this purpose is also not interesting, as most of the workload is usually free from these issues.

Authors' Contact Information: Rui Braga, U. Minnho and INESCTEC, Braga, Portugal; José Pereira, U. Minnho and INESCTEC, Braga, Portugal; Fábio Coelho, U. Minnho and INESCTEC, Braga, Portugal.

Ideally, a more versatile solution would allow the data store to adapt to the specific requirements of an application by choosing whether availability or consistency would be prioritized in each operation. In this paper, we explore the possibility that this is achieved at the application level by implementing Snapshot Isolation [12] for a subset of the data items using a client-side layer when accessing a TCC data store, without coordination mechanisms other than the TCC data store itself (i.e., no direct message passing between clients). Our contribution is thus threefold:

- First, we identify the key challenges in implementing SI over TCC in a minimal way, that is, making the most of existing mechanisms (e.g., multi-version) and guarantees.
- Second, we propose a detailed algorithm that solves these challenges.
- Third, we experimentally evaluate a proof-of-concept implementation and draw lessons on the feasibility of the approach and the design of TCC data stores.

The rest of this paper is structured as follows. Section 2 describes transactional isolation and the AntidoteDB system. Section 3 introduces our proposal and how it solves key challenges. Then, Section 4 describes the proposed algorithm in detail. Section 5 proposes a proof-of-concept implementation and evaluates it experimentally. Finally, Section 6 compares our proposal to related work and Section 7 concludes the paper:

2 Background

2.1 Transactional isolation

Snapshot Isolation (SI) [12] is a transactional isolation criterion that has become popular in database systems as it allows high-performance read-only transactions while avoiding most of the concurrency anomalies and can even be transformed into Serializability [18]. It provides the illusion that all read operations occur at the start of the transaction, that is, read the values that existed at the time the transaction started regardless of concurrent updates, and atomically persist all updated items at the end ensuring that no conflicting updates have committed.

As an example, consider a database that holds items that describe the availability of a resource x and two concurrent transactions that reserve a resource unit by reading the current availability ($r(x)$), subtracting from x, and then writing it back ($w(x)$) in the context of a transaction. If two concurrent transactions attempt to update the same resource, it might result in the following sequence of operations: $r(x)_1, r(x)_2, w(x)_1, w(x)_2$. If both transactions are committed, the updated by transaction 1 is lost, overwritten by transaction 2. Snapshot Isolation prevents this by aborting the second transaction that attempts to commit.

In contrast to Snapshot Isolation, Transactional Causal Consistency (TCC) [9] does not restrict updates by concurrent transactions and ensures only a causal order in snapshots that are visible to transactions. Therefore, in the previous example, both transactions would commit, which is not acceptable in some applications. However, this is precisely what allows transactions to be committed in data centers while the network is partitioned.

2.2 AntidoteDB

AntidoteDB is a georeplicated key-value database, which provides high availability, performance, and horizontal scalability like AP (Available Partitionable) databases. However, it also provides features that help programmers write correct applications with Transactional Causal Consistency, thus still performing well in a georeplicated environment. It also provides access to high-level replicated data types that work well in the presence of partial failures and can handle concurrent updates with Conflict-free Replicated Data Types (CRDTs) [7].

In order to provide fast parallel access to different data, AntidoteDB shards the data among the servers within a cluster using consistent hashing. A request is served by the servers that hosts a

copy of the data while a transaction that issues multiple operations contacts only the servers that have the objects it wants to access. This masterless design allows the database to be tolerant of node failures.

To be able to do all this, AntidoteDB is based on Cure [9], a highly scalable protocol that asynchronously replicates updates between clusters, so it remains highly available under network partition [5]. So, to achieve this, Cure adopts three major design decisions:

- The protocol does not rely on centrally assigned timestamps, as some other systems do.
- The ordering of events and its causal dependencies depend on the timestamps of events represented in vector clocks [1], which allows partitions to make decisions locally, explicitly avoiding checking dependencies through messages, and saving the protocol from performance penalties.
- The problem of propagating updates between replicas is decoupled from the problem of making the updates visible, which allows partitions to propagate updates without requiring any coordination with other partitions.

Cure keeps multiple versions of each object to be able to handle requests from causally consistent snapshots, and each version stores its value with the metadata necessary to encode its causal dependencies. Old and unused versions are periodically garbage collected through the exchange of the oldest snapshot vector clock of its active transactions between partitions and computing the aggregate minimum. Once this minimum is found, the stored versions of the objects that are no longer required can be safely removed as they will not be requested anymore.

The objects used to store values in Cure are operation-based CRDTs which require causal consistency and updates to be delivered in the correct order, because an object's value is not defined just by the last update, but also by the state it is applied on.

3 Approach

The cornerstone of our approach is that an SI transaction is implemented in a middleware layer and mapped to native operations in a TCC database system. However, read/write operations and transaction demarcation are intercepted and transformed by the middleware. Namely, writes can be augmented (e.g., with meta-information), reads filtered, and additional auxiliary transactions issued when appropriate.

For simplicity and performance, we want to map each operation of an SI transaction, as close as possible, to the corresponding operation in a TCC transaction. However, Snapshot Isolation poses two additional challenges. First, it has to avoid the lost-update anomaly where two concurrent transactions update the same item, hence, one of them getting overwritten. Second, it has to avoid the long-fork anomaly and ensure that snapshots observed by committed transactions are totally ordered.

3.1 Validation

To overcome the first challenge, we have to ensure (pessimistically) that no two concurrent transactions update the same item or (optimistically) that when two transactions have updated the same item concurrently, only one of them can actually commit successfully. In both cases, transactions need to be aware of each other, either to avoid writing or to avoid committing. If all communication is done through the TCC database system, this would be impossible with a single TCC transaction for each SI transaction, as no information flows between concurrent transactions. It could be done by mapping each read or write operation to its own, separate, TCC transaction, but this would severely impact performance.

Therefore, we propose an optimistic approach in which SI transactions are validated, in the context of a second TCC transaction, after requesting to commit. Briefly, each SI transaction tentatively commits updates in the context of the original TCC transaction. Later, a second transaction verifies that no conflicts have ocurred and makes them final. Otherwise, it rolls them back.

The first step for validation is establishing a total order for concurrent transactions such that, when a transaction is validated, the outcome of all preceding transactions is already known. We do this by having each site maintain its own scalar logical clock as an item in the data store that is updated only by its owner but read by all, and with Lamport's total order algorithm [20]. Lamport's algorithm is a fully decentralized solution for total order in a distributed system: Each site can, by itself, observe the clocks of all others, identify which operations are already *stable* (i.e., their past is fully known) and then use some agreed static ordering (e.g., using site ids) to break ties. However, this algorithm will block when partitioning occurs. The second step, which is deterministic, is to compare the write set of each transaction with the write set of its committed predecessors. Interestingly, this means that it can be performed by multiple uncoordinated concurrent processes, as all of them will decide the same outcome.

3.2 Versioning

To provide a snapshot for each transaction, the systems that provide SI keep multiple versions of each data item, tagged with a commit timestamp, and for each access, select the most recent that predates the start of the current transaction. Overwritten versions older than the oldest currently running transaction can be discarded. The system thus has to keep track of the transactions that are currently running to perform garbage collection.

We use a similar approach, as for each key written by an SI transaction, we keep multiple values indexed by the transaction that wrote them. However, garbage collection can be fully delegated to the TCC data store as follows. Recall that updating an item in a TCC transaction impacts only future transactions. Therefore, garbage collection is as easy as deleting values that have been overwritten as soon as an SI transaction is validated. Currently executing SI transactions have already initiated a TCC transaction and obtained a snapshot that will hold valid for the duration of the transaction.

4 Algorithm

In this section we describe the operations that are available for applications to execute SI transactions and, in detail, how each of them is implemented using a simplified TCC interface. The implementation of these operations relies on the periodic maintenance of the data structures performed by background *daemons*.

4.1 Notation

We build on AntidoteDB, a key-value CRDT store with transactional causal consistency where values can be registers, counters, sets, or maps. We consider the following simplified interface:

- $T \leftarrow$ TCC-BEGIN() – initiates a transaction, returning a handle that is used when issuing other operations.
- TCC-COMMIT(T) – commits the transaction, making its updates visible.
- TCC-ABORT(T) – discards the updates and aborts the transaction.
- $V \leftarrow$ TCC-READ(T, K) – returns the value that corresponds to the state of the object stored under K in the version given in the transaction's snapshot.
- TCC-UPDATE(T, K, O) – issues an update operation O to the current object V stored under K. If V is a register, we directly use the value to be set or $O = \perp$ to remove it. If V is a map, we use $[x \rightarrow y]$ to denote an update to the map, with $y = \perp$ to remove it.

We implement a key-value store with snapshot isolation that offers the following operations:

- $T \leftarrow$ SI-BEGIN() – initiates a transaction, returning a handle that is used when issuing other operations.
- SI-COMMIT(T) – commits the transaction, initiating the validation process.
- SI-ABORT(T) – discards the updates and aborts the transaction.
- $V \leftarrow$ SI-GET(T, K) – returns the opaque value that corresponds to the state of the object stored under K in the version given in the transaction snapshot.
- SI-PUT(T, K, V) – declares that an update will be made on the object under K with the opaque value V, for a given transaction handle T.

4.2 Data objects

This implementation is supported by the following persistently stored data objects:

- Log: This is a single global data object that keeps track of the state in which every transaction is in. It is a *RRMAP* (Remove-Resets Map) [5] and maps each transaction id to a triple $(status, sts, cts)$. The first value $state$ is the state in which it is in, which can be one validating V, committed C or aborted A. The second sts is the starting time stamp that determines the logical time at which the transaction started and thus the snapshot it used. The third, cts, is the logical time at which the transaction has committed and starts at $+\infty$.
- WS_{id}: For each transaction id, it contains a list of keys it updated. This data object is used to check if there are any conflicts between concurrent transactions.
- UClock_d: It stores the scalar logical clock of site d and is present in each DC. It is updated by a single site, but read by all, and is used for totally ordering committed transactions using Lamport's RSM (Replicated State Machine) algorithm.
- LClock_d: It stores a scalar logical clock lower than the oldest snapshot that might still be used by active transactions at site d. Therefore, it is used only for garbage collection purposes. It is updated by a single site, but read by all.
- K: For each key application key, the key K stores a *RRMAP* [5], that in turn maps the transaction identifiers to the last value the transaction has written in K.

Each client session stores the following transient variables:

- id_T: A globally unique identifier for the current SI transaction T.
- t_T^1: The handle for the main TCC transaction that supports the current SI transaction T.
- ws_T: The set of keys that have changed over the course of the current SI transaction T.

4.3 Operations

The client application starts by connecting to a AntidoteDB node that is going to act as a transaction coordinator, making the client eligible for issuing the operations.

Algorithm 1 Begin transaction at site d

```
1: function SI-BEGIN()
2:     id_T ← unique identifier
3:     ws_T ← {}
4:     t_T^1 ← TCC-BEGIN()
5:     return T
```

Algorithm 1 is used by the client to start a transaction. It starts by initializing local variables, generating a globally unique id id_T and setting an empty write-set ws_T. The key step (line 4) is

starting a transaction t_T^1 in the underlying store, which is to be used for all data read and write operations.

Note that there is no need to explicitly set a snapshot: Reading from t_T^1 implicitly provides a consistent snapshot. Moreover, no meta-data need to be written yet, as it can be done later and only if the transaction is set to actually be committed.

Algorithm 2 Put operation

```
1: function SI-PUT(T, K, V)
2:     TCC-UPDATE(t_T^1, K, [id_T → V])
3:     ws_T ← ws_T ∪ {K}
```

Algorithm 2 shows how a data item is updated. The value is stored as an additional pair $id_T \rightarrow V$ in the RRMAP object corresponding to the key K, where id_T is the identifier of the current transaction. The updated key is also added to the write set ws_T, that the application is tracking, containing all the keys updated by the transaction, which will be used later on for validation.

Algorithm 3 Get operation

```
1: function SI-GET(T, K)
2:     v ← TCC-READ(t_T^1, K)
3:     if v = ⊥ then
4:         return ⊥
5:     else if K ∈ ws_T then
6:         return v[id_T]
7:     else
8:         log ← TCC-READ(t_T^1, Log)
9:         if ∃id ∈ dom(v) : log[id] = (C, _, _) then
10:            return v[id]
11:        else
12:            return ⊥
```

Algorithm 3 shows how a client reads the value associated with an application key K. It starts by reading into v the map from K, that contains all pairs $id \rightarrow V$ of visible past transactions. These pairs, if any, have been written in line 2 of Algortihm 2. Therefore, if this value has not been written before, a null value is returned. If it has been written by the current transaction, the value that exists in the view of that transaction is returned. Otherwise, it searches for a value written in the past by other transaction that have already been committed, and, if it exists, returns the value for that view, that is, $v[id]$, otherwise returning null.

Note that, in contrast to traditional implementations of Snapshot Isolation, one does not have to explicitly search for the version that corresponds to the correct snapshot. This happens because:

- Values written after the transaction has started, that is, more recent than the snapshot of the current transaction, cannot be observed in the snapshot of that transaction because reads are made with reference to t_T^1 TCC transaction.
- Previous versions that have been overwritten by the transaction that is visible in the snapshot were removed when that transaction has been committed in Algorithm 7 (line 12).

Note that if uncommitted versions exist, this might indicate that if transaction T also updates the same item, it is unlikely that the transaction will be successfully validated and might decide to spontaneously abort.

Algorithm 4 Abort transaction

```
1: function SI-ABORT(T)
2:     TCC-ABORT(t_T^1)
```

Algorithm 4 shows that the option to abort a transaction t_T^1 will cause all the changes tracked within the view of that transaction to be discarded.

Algorithm 5 Commit transaction

```
1: function SI-COMMIT(T)
2:     if ws_T = ∅ then
3:         TCC-COMMIT(t_T^1)
4:         return True
5:     else
6:         sts ← max(c|(C, _, c) ∈ TCC-READ(t_T^1, Log))
7:         TCC-UPDATE(t_T^1, Log, [id_T → (V, sts, +∞)])
8:         TCC-UPDATE(t_T^1, WS_{id_T}, ws_T)
9:         TCC-COMMIT(t_T^1)
10:        repeat
11:            t^2 ← TCC-BEGIN()
12:            log ← TCC-READ(t^2, Log)
13:            TCC-COMMIT(t^2)
14:        until log[id_T] ≠ (V, _, _)
15:        return log[id_T] = (C, _, _)
```

Algorithm 5 shows how the client requests a transaction to be committed. If the write set ws_T is empty, then the transaction is considered read-only, and no further actions are needed. This is because there are no touched keys that would require an update to be triggered. Otherwise, the routine proceeds in two steps:

- First, it persists the metadata needed for validation (lines 6 to 9). This means adding an entry in the transaction log for transaction T, linking the snapshot used to the latest installed view of the available already committed transactions (i.e., the most recent committed transaction that can be observed) and the write-set held within transaction T. At this point, the commit timestamp that will be linked to transaction T is still unknown and is set to $+\infty$. The write-set of transaction T is then validated for conflicts with the latest install view that stems from the remaining committed write sets.
- Second, it repeatedly re-reads the log in a new transaction t^2 to poll for the outcome of validation (line 12), which will start the process to assign a commit timestamp, which will be done locally by the daemon at the local site.

Waiting for a transaction to be decided is the default behavior in transactional systems. However, it is usually the limiting factor for performance. This is also the case in our proposal, as it depends on the propagation of clock values between different data centers. It would also be possible to offer the application the option of not waiting. This is also often available in transactional systems as an asynchronous commit option. In fact, after line 9, the transaction is persistent and will not be rolled by faults, only by conflicts with other transactions.

4.4 Daemons

Concurrently with client operations, our proposal assumes background tasks that are executed by *deamons* at each site.

Algorithm 6 Clock daemon at site d

```
1:  loop
2:      t³ ← TCC-BEGIN()
3:      cts ← max(∀i : TCC-READ(t³, UClock_i))
4:      l ← TCC-READ(t³, LClock_d)
5:      log ← TCC-READ(t³, Log)
6:      for id ∈ log : id is from d ∧ log[id] = (V, s, +∞) do
7:          if s < l then
8:              TCC-UPDATE(t³, Log, [id → (A, s, +∞)])
9:          else
10:             cts ← cts + 1
11:             TCC-UPDATE(t³, Log, [id → (V, s, cts)])
12:     TCC-UPDATE(t³, UClock_d, cts)
13:     choose l|∀[id → (V, s, _)] ∈ log : l < s ∨ id is not from d
14:     TCC-UPDATE(t³, LClock_d, l)
15:     TCC-COMMIT(t³)
```

Algorithm 6 shows a core part of the proposed technique, the daemon responsible for totally ordering local transactions for later validation. Exactly one instance of this daemon needs to be running at each site. We do not address how this is done, but it has the same requirements as the maintenance of the servers in the local site.

In each iteration, this daemon first computes cts the maximum of clocks at all sites (lines 6 to 11). This value is used to order locally pending transactions (line 11) but also to update the local clock UClock_d (line 12).

It is important to note that if the daemon is able to read a certain value for the clock of a remote site, it means it is also able to see all the changes made by the remote site up to that clock's value. Therefore, updating the local clock informs remote sites that no further transactions will be ordered in the past, and thus allows them to be validated.

This daemon also arbitrarily advances LClock_d for garbage collection. Transactions with lower starting timestamps, still running or already pending, will have to be unilaterally aborted (line 8) as there will not exist sufficient information for validating them, as the write-sets of concurrent transactions that have been committed might not already been removed.

Algorithm 7 shows the second core part of the proposed technique, responsible for finding the outcome for each transaction. We present it as a daemon that runs periodically, but multiple instances can be running concurrently. In fact, it is possible to embed the same functionality within the clock daemon (Algorithm 6) or the commit procedure (Algorithm 5) to reduce the response time. The reason for this is that this algorithm is deterministic and idempotent.

The validation procedure first obtains an ordered list of pending transactions according to Lamport's algorithm as follows. It starts by reading *clock* the minimum of clocks from all sites (line 4). Then, it finds the set P of transactions waiting for validation (state V) whose commit timestamp is lower than *clock* (line 5). Finally, it traverses P in order of growing commit timestamp c and breaks ties with id. Given that we are sure that we already know all transactions before *clock* and the order is statically determined, this traversal is deterministic.

Algorithm 7 Validation daemon(s)

```
1:  loop
2:      t^4 ← TCC-BEGIN()
3:      log ← TCC-READ(t^4, Log)
4:      clock ← min(∀i : TCC-READ(t^4, UClock_i))
5:      P ← {(c, id, s) : log[id] = (V, s, c) ∧ c ≤ clock}
6:      for (c, id, s) ∈ P in lexicographical order do
7:          if VALIDATESI(t^4_t, id, s, log) then
8:              TCC-UPDATE(t^4, Log, [id → (C, s, c)])
9:              for K ∈ ws do
10:                 v ← TCC-READ(t^1_T, K)
11:                 if ∃id′ ∈ dom(v) : log[id′] = (C, _, _) then
12:                     TCC-UPDATE(t, K, [id′ → ⊥])
13:         else
14:             TCC-UPDATE(t^4, Log, [id → (A, s, c)])
15:             for K ∈ ws do
16:                 TCC-UPDATE(t, K, [id → ⊥])
17:         log ← TCC-READ(t^4, Log)
18:     TCC-COMMIT(t^4)
19:
20: function VALIDATESI(t, id, s, log)
21:     ws ← TCC-READ(t, WS_id)
22:     C ← {id′ : log[id′] = (C, _, c′) ∧ s < c′}
23:     return ∀id′ ∈ C : (TCC-READ(t, WS_id′) ∩ ws) = ∅
```

The criterion for Snapshot Isolation is encapsulated in the auxiliary VALIDATESI() function. A transaction can be committed if its write set does not intersect with the write set of any concurrent transaction, i.e., those whose commit timestamp c' is greater than the snapshot s.

If the outcome is true, the log entry is updated (line 8). Next, previous versions of items updated by this transaction are removed, thus ensuring the invariant needed for Algorithm 3 that at most one committed value is visible for each key. If not, the log entry is also updated as aborted (line 14) and, in this case, the updates produced by this transaction are removed.

Before proceeding to the next transaction in the list, the log is re-read as it might have changed in a way that influences future outcomes (i.e., has additional committed transactions).

Algorithm 8 Garbage collection daemon(s)

```
1: loop
2:     t^5 ← TCC-BEGIN()
3:     l ← min(∀i : TCC-READ(t^5, LClock_i))
4:     log ← TCC-READ(t^5, Log)
5:     ids ← {id | log[id] = (_, _, c) ∧ c < l}
6:     for id ∈ ids do
7:         TCC-UPDATE(t^5, Log, [id → ⊥])
8:         TCC-UPDATE(t^5, WS_id, ⊥)
9:     TCC-COMMIT()
```

Finally, Algorithm 8 shows garbage collection of metadata. It also is deterministic and idempotent, hence, multiple concurrent instances can be running. The key feature of this algorithm is to find

the minimum value of the clock for which log entries might still be needed (line 3). It then removes all entries in the log and corresponding write-sets that precede *l*.

5 Evaluation

To evaluate the feasibility of this approach with current TCC data stores, we implement a proof-of-concept on top of AntidoteDB and use a simple benchmark to perform experiments.

5.1 Implementation

The Snapshot Isolation layer is implemented using a single Python class called *SITransaction* that handles the client's connection to an AntidoteDB server, handles all the metadata related to the client session within the SI context, and has the methods that are used to implement the aforementioned SI transaction programming interface, using AntidoteDB's python client [4]. Furthermore, AntidoteDB is configured to ensure snapshot isolation within a datacenter [6].

The examples in Figure 1 show what a transaction writing the value "val1" in a key designated "key1", would look like with the AntidoteDB client and using the SI layer.

```
clt = AntidoteClient("127.0.0.1", 8100)
tx = clt.start_transaction()
key = Key("bucket", "key1", "MVREG")
val = bytes("val1", 'utf-8')
res = tx.update_objects(Register.AssignOp(key, val))
ok = tx.commit()
```

```
clt = AntidoteClient("127.0.0.1", 8100)
tx = SITransaction(clt, 1)
tx.start()
val = bytes("val1", 'utf-8')
tx.putValues([("key1", val)])
tx.commit()
```

Fig. 1. Invoking TCC and SI transactions.

5.2 Environment

All experiments are executed with AntidoteDB servers organized as 2 data centers with 2 nodes each, deployed in Docker containers on a server with 48 cores and 100GB RAM. The workload generator runs on the same server. The client threads are distributed evenly by all 4 nodes.

We consider two different workloads: First, we use the standard YCSB benchmark [2], which includes read-only and write-only operations, to measure the overhead of the *SITransaction* layer. By running it with an increasing number of client threads, i.e., the number of simultaneous requests to the database systems, we also measure how *SITransaction* scales with concurrency. For this workload, AntidoteDB with configured with `CERT=false`, meaning that it enforces Transactional Causal Consistency and thus achieves its best performance.

The second workload aims to assess the usefulness of an adaptive transactional system. It includes two different sets of operations: 95% of the operations are increments of a random counter, which can be done correctly with TCC. The remanining 5% are test-and-set operations for a random register, which is susceptible to the lost update anomaly and requires Snapshot Isolation. For this workload, we use two configurations:

- Adaptive transactional isolation, with AntidoteDB configured with `CERT=false` used directly for counter operations and *SITransaction* used for test-and-set operations. This is the configuration that produces the correct results and becomes possible with our proposal.
- The native AntidoteDB configured with `CERT=true` used directly for all operations. This configuration results in conflicts between transactions originating from the same data center being detected and prevented by AntidoteDB. Note that this is not enough to correctly execute the workload, as conflicts between transactions originating from different data centers will lead to anomalous behavior.

By increasing the number of items in the database, we reduce the probability of conflict. These experiments are executed with 96 client threads.

5.3 Results

Figure 2 shows the results obtained with the first workload. In detail, Figure 2a shows the latency of read-only transactions. It shows that even though a read-only SI transaction maps directly to a TCC transaction, there is overhead resulting from the need to read a more complex data structure that has potential future versions of the item.

Figure 2b shows the latency for update transactions. In addition to the meta-data that is added to each item written, in this case transactions have to wait for ordering and validation, this incurring in additional latency. In fact, ordering is affected by propagation latency between data centers, which in AntidoteDB occurs periodically and cannot be easily tuned. Moreover, as it does not have change notification, daemons and client layers have to poll for changes, thus further consuming resources.

Figure 2c shows the impact on transactional throughput. This is expected as we are using a closed-loop benchmark and the additional latency has a direct impact on throughput.

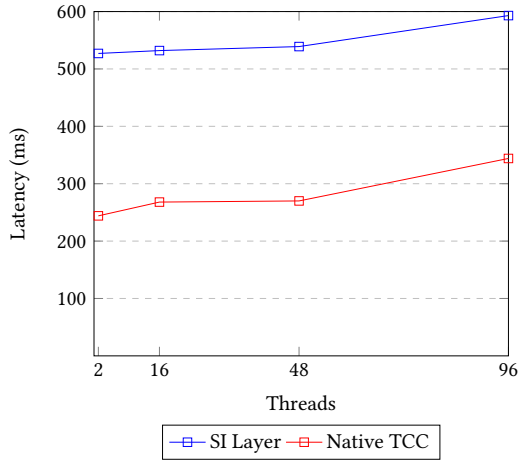
However, observing the second workload in Figure 3, the advantage of adaptivity is shown in the mixed scenario, since only SI transactions pay the additional performance cost, resulting in an overall performance close to the original TCC, while being able to perform operations with varying levels of consistency and, avoiding a significant amount of aborted transactions due to conflicts.

6 Related work

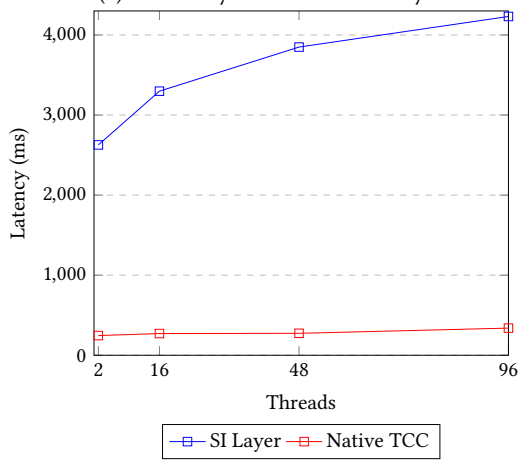
Causal transactions as a consistency criterion for AP scalable data stores can be traced to COPS-GT [22] which adds Get-Transactions, i.e., the ability to enforce mutual causal dependencies among multiple items read in a single operation, to the basic COPS protocol for causally consistent read operations in a georeplicated system. This work was later generalized to support also atomic updates of multiple items with Put-Transactions in Eiger [23] and later to consider partial replication [24] and optimal performance [21]. However, these proposals do not provide the interactive transactions that our proposal relies upon. It would be interesting to provide SI transactions on top of this model, even if a restricted to a non-interactive model.

In this paper, we use Cure [9] which provides both interactive causal transactions and CRDTs [10] for convergence. Other proposals offer interactive transactions with improved performance [26] and with partial replication [27]. However, these systems do not support the CRDTs on which our proposal is based. However, it would be interesting to explore the possibility of using only simple last writer wins registers.

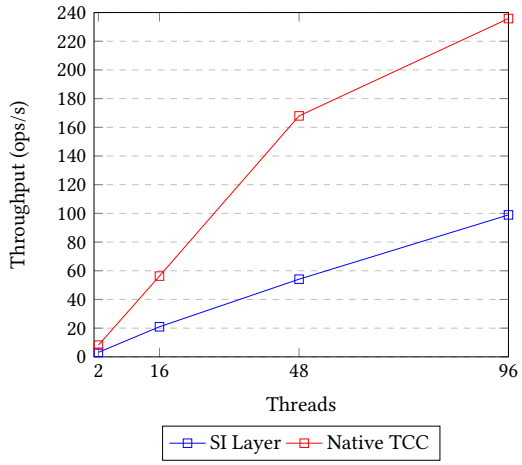
Proposals such as Dynamo [16], with eventual consistency, and RAMP [11], with read atomicity, propose distributed data stores that are weaker than causal consistency. It would also be interesting to explore our approach in such settings, in particular, with read atomicity. In fact, the client-side implementation of transactional atomicity in non-transactional key-value stores using a two-phase



(a) Read-only transaction latency.



(b) Update transaction latency.



(c) Transactional throughput.

Fig. 2. Performance comparison with YCSB.

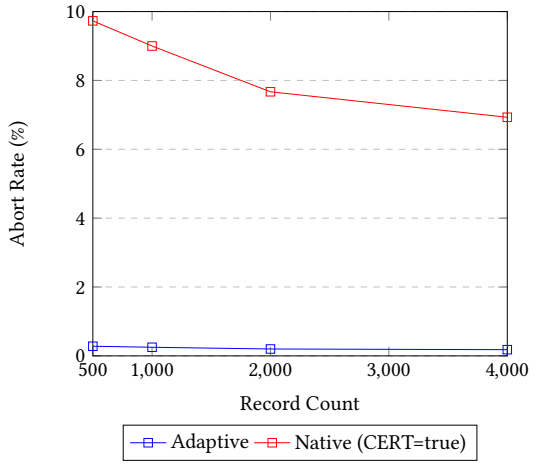


Fig. 3. Abort rate with mixed workload.

commit on individual data items has been proposed with the Cherry Garcia protocol [17]. This has been applied in practice [3, 28], but has substantial overhead on each operation.

Finally, there are multiple proposals for implementing various transactional systems in distributed non-transactional data stores that make use of external coordination mechanisms. That is, Saturn [13] provides transactional causal consistency for georeplicated data stores by controlling the visibility of updates. Omid [19] implements Snapshot Isolation using a centralized transaction validation oracle and a middleware layer that controls visibility. Spanner [15] implements serializability using a global clock and Paxos consensus for coordination.

7 Conclusion and future work

This paper aimed at a flexible middle ground between availability and consistency by implementing Snapshot Isolation over a database with Transactional Causal Consistency. The proposed solution enables developers to adapt isolation levels to the requirements of specific operations. This is particularly well-suited for modern distributed applications, where geographic distribution and responsiveness are critical, without compromising data integrity.

The main lesson learned from this work is that even though the proposed approach is feasible, performance is severely affected by the lack of control of dissemination between data centers and the lack of a notification mechanism that forces clients and daemons to poll for changes. It is an open research question to what extent these would help and designers of future highly available transactional systems should consider these features as a path to adaptivity.

To further bridge the gap between strong data consistency and the quickness that availability provides, the next step would be to enable TCC and SI transactions to modify the same items, integrating them further and making the transition between the two types of transaction seem seamless.

Acknowledgments

This work is co-financed by Component 5 – Capitalization and Business Innovation, integrated in the Resilience Dimension of the Recovery and Resilience Plan within the scope of the Recovery and Resilience Mechanism (MRR) of the European Union (EU), framed in the Next Generation EU, for the period 2021-2026, within project NewSpacePortugal, with reference 11.

References

[1] 1988. Virtual time and global states of distributed systems. https://homes.cs.washington.edu/~arvind/cs425/doc/mattern89virtual.pdf.

[2] 2019. Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB.

[3] 2021. Amazon DynamoDB Transactions. https://github.com/awslabs/dynamodb-transactions.

[4] 2022. Antidote Python Client. https://github.com/AntidoteDB/antidote-python-client.

[5] 2024. Antidote Project Documentation. https://antidotedb.gitbook.io/documentation/.

[6] 2024. Antidote Project Documentation - Features Configuration. https://antidotedb.gitbook.io/documentation/architecture/configuration.

[7] 2024. AntidoteDB. https://www.antidotedb.eu/.

[8] 2024. AntidoteDB Github. https://github.com/AntidoteDB/antidote.

[9] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 405–414. https://doi.org/10.1109/ICDCS.2016.98

[10] Paulo Sérgio Almeida. 2023. Approaches to Conflict-free Replicated Data Types. https://arxiv.org/abs/2310.18220. arXiv:2310.18220 [cs.DC]

[11] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3 (July 2016), 1–45. https://doi.org/10.1145/2909870

[12] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf. arXiv:cs/0701157 [cs.DB]

[13] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. 2017. Saturn: a Distributed Metadata Service for Causal Consistency. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 111–126. https://doi.org/10.1145/3064176.3064210

[14] Eric A. Brewer. 2000. Towards Robust Distributed Systems. https://dl.acm.org/doi/10.1145/343477.343502. , 7 pages. https://doi.org/10.1145/343477.343502

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J J Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (Aug. 2013), 1–22. https://doi.org/10.1145/2491245

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. https://dl.acm.org/doi/10.1145/1323293.1294281.

[17] Akon Dey, Alan Fekete, and Uwe Rohm. 2015. Scalable distributed transactions across heterogeneous stores. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 125–136. https://doi.org/10.1109/icde.2015.7113278

[18] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM trans. database syst.* 30, 2 (June 2005), 492–528. https://doi.org/10.1145/1071610.1071615

[19] Daniel Gómez Ferro, Flavio Junqueira, Ivan Kelly, Benjamin Reed, and Maysam Yabandeh. 2014. Omid: Lock-free transactional support for distributed data stores. In *2014 IEEE 30th International Conference on Data Engineering*. 676–687. https://doi.org/10.1109/ICDE.2014.6816691

[20] Leslie Lamport. 1979. Time, clocks, and the ordering of events in a distributed system. https://lamport.azurewebsites.net/pubs/time-clocks.pdf.

[21] Si Liu, Luca Multazzu, Hengfeng Wei, and David A Basin. 2024. NOC-NOC: Towards Performance-optimal Distributed Transactions. *Proc. ACM Manag. Data* 2, 1 (March 2024), 1–25. https://doi.org/10.1145/3639264

[22] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. 23rd ACM Symposium on Operating Systems Principles*. https://www.cs.cmu.edu/~dga/papers/cops-sosp2011.pdf

[23] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 313–328. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd

[24] Khiem Ngo, Haonan Lu, and Wyatt Lloyd. 2021. K2: Reading Quickly from Storage Across Many Datacenters. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 199–211. https://doi.org/10.1109/DSN48987.2021.00034

[25] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. 2016. Conflict-free Replicated Data Types. https://arxiv.org/pdf/1805.06358.pdf.

[26] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2018. Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 1–12. https://doi.org/10.1109/DSN.2018.00014

[27] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2019. PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 304–316. https://doi.org/10.1109/ICDCS.2019.00038

[28] Hiroyuki Yamada, Toshihiro Suzuki, Yuji Ito, and Jun Nemoto. 2023. ScalarDB: Universal Transaction Manager for Polystores. *Proceedings VLDB Endowment* (2023). https://doi.org/10.14778/3611540.3611563