PrompTCC: Transactional Causally Consistent Reads Can Be Fast and Fresh

Taras Lykhenko
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal
taras.lykhenko@tecnico.ulisboa.pt

Rafael Soares
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal
joao.rafael.pinto.soares@tecnico.ulisboa.pt

Luís Rodrigues
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal
ler@tecnico.ulisboa.pt

***Abstract*—Transactional Causal Consistency (TCC) is the strongest consistency model compatible with availability and, therefore, it avoids the pitfalls of the CAP theorem while simplifying the programming of cloud applications. Unfortunately, with previous implementations, TCC came at the cost of expensive reads. TCC has been implemented either using conservative approaches, that always require two communication rounds, or using optimistic approaches that, in good cases, require just one round, but in face of skewed workloads, that are common in real applications, can require three communication rounds. In this paper we propose a novel algorithm, named PrompTCC, that in most cases offers reads in just one round and that, even in face of skewed workloads, never takes more than two rounds. As a result, PrompTCC is able to closely approximate the performance of an eventually consistent system while providing stronger guarantees, achieving only 12% throughput and 20% latency penalty in realistic scenarios, outperforming state-of-the-art systems which present up to 37% and 60% throughput and latency degradation respectively.**

***Keywords*-Distributed Storage, Transactions, Causal Consistency.**

I. Introduction

Weakly-consistent key-value stores have emerged as a key technology to support large-scale cloud applications, being able to offer high throughput, availability, and scalability [1]. By providing only eventual consistency [2], these system have been able to circumvent the availability and scalability limitations associated with ACID transactional consistency criteria (such as serializability or snapshot isolation). However, experience has shown that eventual consistency makes application development difficult [3], so there is a keen interest in supporting alternative consistency models, that may simplify programming without compromising scalability [4].

Among the many consistency guarantees that have been proposed [5]–[7], causal consistency (CC) has been identified as the strongest consistency model that an always-available system can implement with one-way convergence [8]. Compared to eventual consistency, CC avoids a number of anomalies that plague programming with weaker models [9]. However, causal consistency only applies for individual operations, and can still lead to unexpected behavior when multiple operation are chained together [10]. Transactional Causal Consistency (TCC) [3] is a consistency model that extends CC by supporting read and write transactions, allowing applications to read from a causal snapshot and to perform atomic multi-key writes, while preserving the high availability properties of CC.

When implementing TCC, the performance of read-only transactions is of paramount importance [11], given that these transactions dominate many real-world workloads [12]. For instance, 99.8% of the operations for Facebook's distributed data store TAO are reads [13]. The latency of these reads is especially important because, as Facebook has reported that "*a single user request can result in thousands of requests, with a critical path that is dozens of subqueries long*" [4]. This means that, to serve a request, it is necessary to execute multiple transactions that, together, perform large number of reads (even if each individual transaction access fewer keys), making the client response time dominated by the read performance. It is therefore no surprise that several systems have attempted to implement TCC with support for efficient read-only transactions [10], [14]–[17].

There is plethora of results in the literature exploring the theoretical and practical performance limits of read-only transactions in this context. Ideally, one would like to implement read transactions while offering one-round communication, non-blocking operations, and using constant metadata [18]. Unfortunately, read transactions may interleave with concurrent write transaction, observing partial and/or inconsistent system states, which prevents optimal performance to be achieved unless important liveness guarantees, such as minimal progress, are discarded [18], [19]. Since most practical applications have freshness requirements, dropping minimal progress is not an option. It is therefore important to search for system that offer good trade-offs between performance and freshness.

Current state-of-the-art TCC systems are based on two main techniques: stabilization techniques (where read-transactions always observe consistent snapshots) and validation techniques (where transactions optimistically read the latest available values and subsequently check their mutual consistency, requiring additional communication rounds if inconsistencies arise). Wren [20] and Eiger [14] illustrate the tradeoffs involved in these approaches. Wren implements read-only transactions using always *two* communication rounds: one to obtain a stable snapshot (i.e. a snapshot containing all causal dependencies of its updates), and a second to read from the obtained snapshot.

2473-3105/24/$31.00 ©2024 IEEE
DOI 10.1109/PRDC63035.2024.00022

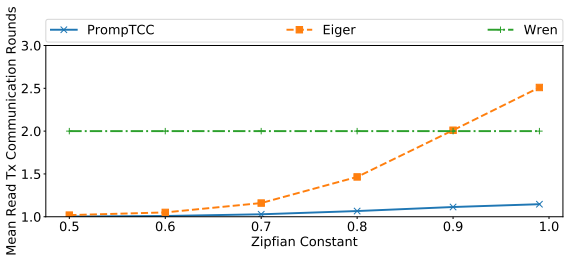

Fig. 1: Communication rounds vs workload skew

System	Strategy	W	N	R	Metadata Size
ChainReaction [17]	Sequencer	✗	✗	≥ 2	$O(M)$
Orbe [15]	Stabilization	✗	✗	2	$O(N \cdot M)$
GentleRain [16]	Stabilization	✗	✗	2	$O(1)$
COPS [10]	Checking	✗	✓	≤ 2	$O(\|deps\|)$
Cure [3]	Stabilization	✓	✗	2	$O(M)$
Eiger [14]	Checking	✓	✓	≤ 3	$O(\|deps\|)$
Wren [20]	Stabilization	✓	✓	2	$O(1)$
PromptTCC (this paper)	Stabilization	✓	✓	≤ 2	$O(N)$

TABLE I: Systems that offer causal consistency. W (write transactions). N (non-blocking reads). R (number of read rounds). N (number of partitions), M (number of data centers), *deps* (the number of client's dependencies).

Eiger may use only *one* round, if the values read are mutually consistent (i.e. they belong to the same causally consistent snapshot), but may also require up to *three* communication rounds in high contention workloads. This is shown in Figure 1, that illustrates the average number of communication rounds required to perform read-only transactions, in a workload with a read/write ratio of 90% for different workload skews (the skew, where a few keys are accessed very often while the remaining keys are accessed rarely, is modeled by Zipfian distribution).

Using a higher number of rounds in read operations not only increases latency but also decreases throughput. This is because multiple rounds of reads cause more data to be transferred over the network, which impairs the system performance. However, reducing the number of read rounds while offering freshness is not trivial. In optimistic approaches, when the read values are inconsistent, re-transmissions are inevitable. Re-transmissions may also be required when the read transaction does not have enough information to validate the read values, a common artifact from the small metadata used by systems such as Eiger.

In this paper we present PrompTCC, a new TCC system design that in most cases reads in 1 communication round, even when faced with skewed workloads, with only a fraction of requests requiring at most an additional second round. As a result, PrompTCC avoids most of the overhead associated with performing multiple rounds of data reads. PrompTCC leverages two complementary techniques to reduce the number of communication rounds. First, it leverages compressed vector clocks [21] to accurately validate the obtained read values, avoiding a significant number of false inconsistencies. Secondly, PrompTCC leverages a stabilization protocol to ensure that only stable values are made visible to transactions, ensuring that read transactions complete at most in two communication rounds.

We have experimentally compared PrompTCC against Eiger [14] and Wren [20] on an AWS deployment with up to 64 servers and 12800 partitions (shards). The results show that PrompTCC achieves up to 50% higher throughput and up to 50% lower latency when compared to previous work.

II. Related Work

The implementation of transactions in weakly consistent systems has deserved a significant amount of attention in recent years. It is therefore possible to find many different approaches to the problem in the literature; these can be classified according to the techniques used to ensure that a transaction observes a consistent snapshot, namely: stabilization, serialization, and explicit dependency checking. Next, we briefly discuss each of these techniques (this comparison is summarized in Table I). We only consider systems that provide minimal progress [19].

Stabilization-based techniques pre-assign a timestamp t to a transaction and force all partitions to postpone serving requests for that transaction until the effect of all write transactions that were executed with timestamp lower than t have been applied locally. Systems such as Orbe [15], GentleRain [16], and Cure [3], use physical clocks as the timestamp to order the transactions. Wren [20] uses hybrid clocks. In these systems, it is possible that, due to clock skew, a client reads a value with a timestamp in the "future" for some partition, forcing the partition to stall the operation until its clock catches up.

Serialization-based systems rely on a centralized component that is able to totally order read transactions with regard to write transactions. A system that implements this strategy is ChainReaction [17]. This centralized component becomes a bottleneck that limits the system scalability.

Systems that use explicit dependency checking require write transactions to be tagged with metadata that captures the client's causal past. Systems like COPS [10] and Eiger [14] use this strategy. In these systems, the client must piggyback causal dependency information and execute expensive dependency checks across partitions, incurring a large overhead.

In addition to the techniques used to ensure that transactions read from a consistent snapshot, there are other features that distinguish these systems.

First, some systems only implement read transactions and do not implement write transactions. Preventing users from using write transactions makes programming harder.

Second, only some of these system are *non-blocking*, in the sense defined by the SNOW and NOCS theorems [11], [18], which states that a system is non-blocking if each server can handle the operations within a read-only transaction without blocking for any external event. Blocking behaviors include waiting for a lock to be available, waiting for messages from other servers, waiting for messages from other clients, or waiting for a timeout to expire. ChainReaction, Orbe,

GenteleRain, and Cure block reads if the client last seen timestamp is greater than the current time of the partition.

Third, different system may require a different number of rounds to terminate a read transaction. Most systems require two or more rounds to execute a read transaction. This not only increases latency, but also contributes to a loss of throughput due to the additional resources consumed in exchanging data values more than once. For example, Eiger [14], requires a second round if the transaction read from a possibly inconsistent snapshot and even a third round if partially committed versions were read on the second round. PrompTCC reduces the likelihood of running a second round and avoids the third round completely, by always returning versions that have been committed at all participating partitions.

Finally, different systems require different amounts of metadata to compute a consistent snapshot. Because the size of metadata affects throughput (specially when handling small data size values), some systems opt to use metadata whose size is constant. Unfortunately, none of these systems execute *non-blocking* transactions in a single round.

III. System Model and Definitions

Cloud applications are typically built using two distinct tiers of machines: a front-end tier and a storage tier. The front-end tier is stateless and handles requests from users by executing application code that reads and writes data from the stateful storage tier. In the context of this work, when we say, "*a client writes a value´´*, we mean that an application running on a web or application server writes into the storage tier.

In order to scale, the data is partitioned across multiple shards, with each shard stored in a different set of machines. In turn, each shard may be replicated. As previous works, we assume that each shard is linearizable. However, we do not require the entire datacenter to be linearizable: write operations on different shards may be executed concurrently and observed in different orders by different clients.

A. Causal Consistency

The causal dependencies of an operation are determined by the *happened-before* relation ($\rightsquigarrow$) originally defined by Lamport [22] and subsequently adapted to shared memory systems [7] as follows:

Thread of Execution: If a and b are two operations executed by the same thread (i.e., by the same client), then $a \rightsquigarrow b$ if a is executed before b.

Reads From: If a is an update operation and b is a read operation that reads the value written by a, then $a \rightsquigarrow b$.

Transitivity: If $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

Let $w(A)$ and $w(B)$ be two write operations on keys A and B. Let $r(A)$ and $r(B)$ be two read operations by the same client, where $r(A)$ is executed before $r(B)$ and where $r(A)$ returns the value written by $w(A)$ and $r(B)$ returns the value written by $w(B)$. A storage system is said to be *Causally Consistent* (CC) if in the case that $w(B) \rightsquigarrow w(A)$ there is no write $w'(B)$ such that $w(B) \rightsquigarrow w'(B) \rightsquigarrow w(A)$.

B. Transactional Causal Consistency

Many applications include sequences of write operations, named transactions, whose effects are expected by the programmer to become visible atomically. However, causal consistency is defined for individual operations, regardless of how these operations are tied in the application logic. Thus, non-transactional causal consistency may yield unexpected results.

Consider for example two write transactions $T^1 = \langle w^1(A), w^1(B)\rangle$ and $T^2 = \langle w^2(A), w^2(B)\rangle$ where $T^1 \rightsquigarrow T^2$ and two read transactions $T^3 = \langle r^3(B), r^3(A)\rangle$ and $T^4 = \langle r^4(A), r^4(B)\rangle$. CC guarantees that in case that T^3 reads the value of B written by T^2 then, later it should read A written by T^2 (and not the previous value is written by T^1). This guarantee results from the fact that $w^1(A) \rightsquigarrow w^1(B) \rightsquigarrow w^2(A) \rightsquigarrow w^2(B)$ being independent from the way that operations are ordered in a transaction. However, CC allows that T^3 reads a value written by T^1 in B and afterward reads a value written in A by T^2. Also, CC allows T^4 to read a value written by T^1 in A and afterward read a value B written by T^2. Neither of these sequences violates causal consistency, but they introduce unexpected behaviors. This problem is better illustrated using a concrete example:

Example: Consider a social network application, where the friendship relations are symmetric, and one needs to ensure the following invariant. If user u_1 is visible in the friends list of u_2, then u_2 needs to be also visible in the friends list of u_1. Consider that k_i stores the friends list of u_i and that the transaction T^1 establishes a new relation of friendship between u_a and u_b, and that T^2 erases that relation. In this case, both T^3 and T^4 would read a state that would violate the application's invariant.

Transactional Causal Consistency (TCC) extends CC with additional guarantees, that are enforced on operations that are part of the same transaction. TCC avoids the described above anomalies by ensuring that transactions read from a causal snapshot, and that updates are atomically visible.

Atomic visibility guarantees that either all keys written by a transaction are visible to other transactions, or none are. Updates are atomically visible if a transaction writes $w(A)$ and $w(B)$, then any snapshot visible to other transactions either includes both $w(A)$ and $w(B)$ or neither one of them. Atomic visibility avoids the anomaly illustrated by T^3 and allows to maintain symmetric relations between entities.

IV. PrompTCC System Design

PrompTCC is an algorithm to implement Transactional Causal Consistency in a system where data is partitioned (sharded) among multiple servers. Therefore, to execute a transaction, a client may need to contact multiple servers. When a write transaction commits, the effects of the transaction will eventually become visible in all servers involved in the transaction. However, due to network delays, at a given point in time, the effects of a transaction may already be visible in one server and not yet visible in another server. Despite this asynchrony, PrompTCC ensures that clients always read from

a causally consistent snapshot. PrompTCC aims at offering TCC guarantees while preserving the following qualities, that are fundamental for achieving high performance:

- **Scalability**. The entire datacenter is not required to be linearizable, and transactions that update different shards can execute concurrently without mutual interference.
- **Response time**. Most read-only transactions execute in a single non-blocking communication round, even in face of skewed workloads.
- **Data freshness**. Reads are served with the most recent stable snapshot.

The algorithm is asynchronous and makes no timing assumptions regarding the network latency among data servers. Furthermore, its operations do not require synchronized clocks. Therefore, the algorithm can enforce TCC, even if different servers are located in different geographic locations. However, in the current paper, we focus on the operation of PrompTCC in a single data center. Also, the algorithm used by PrompTCC is orthogonal to the strategy used to shard data. PrompTCC can, therefore, operate with data stores that use different strategies to assign keys to partitions. We have built a prototype of PrompTCC on top of Cassandra [23], which uses consistent hashing to assign keys to partitions.

Before presenting the design of PrompTCC, we discuss the challenges of providing low latency and non-blocking read transactions on a sharded system.

A. Implementing Efficient Reads

As previously depicted, supporting TCC in a system without compromising latency is not trivial. A transaction consists of multiple read or write operations chained together. Operations on individual keys in a transaction may proceed in parallel. For example, when implementing a transaction $T = \langle r_a(A), r_b(B) \rangle$, it is possible to send a read request $r_a(A)$ to partition p_a and to send in parallel the read request $r_b(B)$ to partition p_b. In this case, the node that serves the read request on p_a is unaware of which version will be returned by the node that serves the read request on p_b. Therefore, the two nodes may return versions that are mutually inconsistent.

PrompTCC follows a validation approach to address the problem above, but includes mechanisms aimed at: i) more accurately detecting inconsistencies, reducing the number of cases where more than one round is necessary, and ii) avoiding the need for a third round, even in the worst case.

The amount of information that needs to be maintained to accurately capture concurrency may be prohibitively large and affect the system's performance [24], [25]. Thus, many systems reduce the amount of metadata at the cost of losing precision [16], leading to scenarios where some operations appear to be causally related, but are not. Not surprisingly, the right amount of metadata that needs to be kept for a given scenario is one of the most studied aspects of distributed systems [26]–[28].

As we discuss next, PrompTCC uses vector clocks to keep track of the most current causal snapshot. This provides an interesting trade-off between the size of the metadata and the number of false dependencies that trigger spurious executions of a second round of communication to serve read-only transactions. Furthermore, PrompTCC implements a number of optimizations that avoid the exchange of the entire vector clock in every message, reducing the overhead that would be incurred by a naive implementation. As a result, the scalability of PrompTCC is almost on par to that of alternative approaches that use less metadata and, thus, are prone to execute spurious extra communication rounds that delay read transactions.

To avoid the need for a third round, PrompTCC leverages a stabilization protocol similar to previous work [3], [20], only making a committed value visible once all participating partitions are ensured to have committed the transaction. By doing so, if a read transaction observes inconsistent results due to a partially visible transaction, clients may help the partition to advance its stable snapshot, ensuring that consistent results may be obtained in a second round.

B. Overview

We will now describe the metadata used by PrompTCC. For clarity of exposition, we first describe the non-optimized version of the system, which assumes the entire metadata is exchanged in every message. Later we describe a number of optimizations that reduce the amount of data exchanged by the implementation.

PrompTCC attaches to each read or write transaction a timestamp, which is materialized in the form of a vector clock of size P, where P is the number of data shards. Every time a write transaction changes the value of a key, a new version of that key is created, which is stored with the commit timestamp of that transaction. Each client c maintains a *dependency vector clock* (D_c), of size P, that keeps the timestamp of the last system state observed by the client, obtained either during a read transaction or after a client update transaction.

When a new key version is first created by an update transaction, it is stored in a *prepared* state; prepared versions are associated with a *tentative* timestamp. On commit time, the algorithm uses the set of individual tentative timestamps, associated with each key, to create a unique *final* timestamp, which is communicated to all partitions involved in the transaction. When a partition receives the final timestamp for a given version of a key, it puts that version in the *committed* state. Finally, once a partition knows all versions created by the same transaction are in the committed state, it puts its key versions in the *visible* state. Because the system is asynchronous, versions of different keys updated by the same transaction will not become visible simultaneously. There is a short period during which one version has been made visible in a partition, but the corresponding versions of the other keys are still in the committed state in other partitions. Clients that execute reads during this window may fail to obtain a consistent snapshot in the first round of a read transaction.

If concurrent transactions access disjoint sets of keys, reading from an inconsistent snapshot is rare. Unfortunately, practical workloads are skewed [12], [29], and concurrent

transactions may access the same keys. We aim to utilize finer-grained metadata to accurately detect inconsistent reads, and perform an additional communication round only when it is strictly needed.

Every partition i keeps a sequence number s_i that is incremented every time a transaction that writes on that partition commits. Moreover, every partition periodically sends the value of s_i to every other partition. This allows each partition to keep a *snapshot vector clock* (S_i) of size P, where each entry $S_i[j]$ holds the last value of s_j received at node i. The snapshot vector clock S_i is used to make updates visible. Any transaction with a commit timestamp lower or equal to S_i is guaranteed to have all versions committed in all partitions. Therefore, as soon as this predicate holds, the versions created by that transaction are moved from the committed to the visible state.

C. Client Library

Given that a transaction can span multiple shards, the client library is responsible for splitting the request and for forwarding the individual operations to the correct partitions. Moreover, the client library is responsible for re-assembling the replies from different partitions in a single response message. Before returning to the client, the library checks if the first round of parallel reads has returned a consistent snapshot and automatically triggers a second round if needed. The library also keeps track of causal dependencies.

Algorithm 1 Client code

```
1:  D_c                                                          ▷ State kept by the client
2:  function WRITE_ONLY_TRANS(T, keys-value-pairs)
3:      P(T) ← PARTITIONS(keys-value-pairs)
4:      coordinator^T ← PICK-COORDINATOR(P(T))
5:      for all i ∈ P(T) do                                      ▷ Send requests in parallel
6:          key-values-for-i ← SPLIT(keys-value-pairs, i)
7:          SEND ⟨write_request, T, key-values-for-i, P(T), coordinator^T, D_c⟩ to i
8:      end for
9:      RECEIVE ⟨write_response, final-ts^T⟩ from coordinator^T
10:     D_c ← final-ts^T
11: end function
12: function READ_ONLY_TRANS(T, keys)
13:     P(T) ← PARTITIONS(keys)
14:     for all i ∈ P(T) do                                      ▷ Perform reads from Round 1 in parallel
15:         keys-for-i ← SPLIT(keys, i)
16:         SEND ⟨read_request_1, keys-for-i, D_c⟩ to i
17:         RECEIVE ⟨read_response_1, S_i, values_round_1_i⟩ from i
18:         for all ⟨k, value_k, ts_k⟩ ∈ values_round_1_i do
19:             return_values[k] ← value_k
20:             return_timestamps[k] ← ts_k
21:         end for
22:     end for
23:     for all i ∈ P(T) do
24:         max-read-ts^T[i] ← MAX(return_timestamps[k][i])∀k
25:     end for
26:     D_c ← max-read-ts^T
27:     potentially-inconsistent-partitions ← {i : S_i < max-read-ts^T}
28:     if potentially-inconsistent-partitions ≠ ∅ then
29:         perform Round 2 on all i ∈ potentially-inconsistent-partitions
30:         // steps are similar to Round 1 above
31:     end if
32:     return return_values
33: end function
```

Algorithm 2 Partition i code

```
1:  s_i, S_i, store                                              ▷ State kept by the partition
                                                                 ▷ Handles clients write transaction request
2:  upon RECEIVE ⟨write_request, T, key-values, P(T), coordinator^T, D_c⟩ from c
    do
3:      s_i ← s_i + 1
4:      tentative-ts_i^T ← D_c; tentative-ts_i^T[i] ← s_i
5:      for all ⟨k, value_k⟩ ∈ key-values do
6:          store ← store ∪ ⟨k, value_k, T, tentative, tentative-ts_i^T⟩
7:      end for
8:      SEND ⟨prepared, T, s_i ⟩ to coordinator^T
9:  end Upon
10: upon RECEIVE ⟨prepared, T, s_j ⟩ from j do
11:     replied[T] ← replied[T] ∪ j
12:     tentative-ts-set[T][j] ← s_j
13:     if replied[T] = P(T) then                                ▷ all partitions have replied
14:         final-ts^T[p] ← tentative-ts-set[T][p], ∀p ∈ P(T)
15:         for all p ∈ P(T) do
16:             SEND ⟨commit, T, final-ts^T⟩ to p
17:         end for
18:     end if
19: end Upon
20: upon RECEIVE ⟨commit, T, final-ts^T⟩ from coordinator^T do
21:     final-ts-set[T] ← final-ts^T
22: end Upon                                                     ▷ Commit values
23: upon ∃T : S_i[i] + 1 ≥ final-ts-set[T][i] do
24:     commit-ts_i^T[p] ← MAX(tentative-ts_i^T[p], final-ts-set[T][p]), ∀p ∈ P
25:     store ← store \ ⟨k, value_k, T, tentative, tentative-ts_i^T⟩
26:     store ← store ∪ ⟨k, value_k, T, committed, commit-ts_i^T⟩
27:     S_i[i] ← MAX(S_i[i], final-ts-set[T][i])
28:     if i = coordinator^T then
29:         SEND ⟨write_response, T, final-ts-set[T]⟩ to c
30:     end if
31: end UponSingle
                                                                 ▷ Make values visible
32: upon ∃T : S_i[p] ≥ final-ts-set[T][p], ∀p ∈ P(T) do
33:     store ← store \ ⟨k, value_k, T, committed, commit-ts_i^T⟩
34:     store ← store ∪ ⟨k, value_k, T, visible, commit-ts_i^T⟩
35: end UponSingle
                                                                 ▷ Handles the first round of a read transaction request
36: upon RECEIVE ⟨read_request_1, keys, D_c⟩ from c do
37:     S_i[p] ← MAX(S_i[p], D_c[p]), ∀p ∈ P      ▷ trigger line 32 (makes updates
    visible)
38:     values ← ∅
39:     for all k ∈ keys do
40:         ⟨k, val_k, ts_k⟩ ← the most recent visible value for k in store
41:         values ← values ∪ ⟨k, val_k, ts_k⟩
42:     end for
43:     SEND ⟨read_response_1, S_i, values⟩ to c
44: end Upon
                                                                 ▷ Handles the second round of a read transaction request
45: upon RECEIVE ⟨read_request_2, keys, D_c⟩ from c do
46:     S_i[p] ← MAX(S_i[p], D_c[p]), ∀p ∈ P      ▷ trigger line 32 (makes updates
    visible)
47:     values ← ∅
48:     for all k ∈ keys do
49:         ⟨k, val_k, ts_k⟩ ← the most recent visible value for k in store: ts_k ≤ D_c
50:         values ← values ∪ ⟨k, val_k, ts_k⟩
51:     end for
52:     SEND ⟨read_response_2, S_i, values⟩ to c
53: end Upon
```

D. Write-Only Transactions

A write-only transaction allows a client to write multiple keys atomically, which may be stored in multiple partitions. PrompTCC implements write-only transactions in at most two rounds of communication (a transaction that updates a single partition finishes in one round). The algorithm does not require operations to acquire any locks. A write-only transaction T is executed as follows:

Let $\mathcal{P}(T)$ be the set of partitions that store keys modified by T. The client starts by choosing one of the partitions $coordinator^T \in \mathcal{P}(T)$ as the coordinator for the transaction.

In the first round, the client sends to each partition $\forall i \in$

$\mathcal{P}(T)$ the new value of the modified keys and the following metadata: the set of partitions $\mathcal{P}(T)$ involved in the transaction, the identifier of the *coordinator*^{T}, and the dependency vector clock of the client D_c. The client library then waits for the responses from each partition. Each partition, upon receiving this message, increments its sequence number s_i and creates a new version of the written keys. These versions are stored in the *prepared* state and are assigned a tentative commit timestamp *tentative-ts*$_i^T$ where *tentative-ts*$_i^T[j] = D_c[j], i \neq j$ and *tentative-ts*$_i^T[i] = s_i$ (Alg. 2 Line 4). The value of s_i is returned to the coordinator.

The second round begins after the coordinator receives a response from every partition involved in the transaction. The coordinator creates a final (commit) timestamp *final-ts*^{T} for the transaction, where *final-ts*$^T[i] = s_i, \forall i \in \mathcal{P}(T)$ (Alg. 2 Line 14). The coordinator sends this value to all partitions in $\mathcal{P}(T)$. After receiving the *final-ts*^{T}, each partition i waits until $S_i[i] + 1 \geq$ *final-ts*$^T[i]$ (Alg. 2 Line 23), at which point it is sure that all transactions with a commit timestamp lower than *final-ts*$_i^T$ are already committed in that partition. When this condition is met, the versions modified by transaction T are moved to the *committed* state, the associated tentative timestamp is replaced with the final commit timestamp, and a response is sent to the client. The write transaction terminates when the client receives a response from all the participating partitions, ensuring that all written values have already been committed by all participants. Finally, the coordinator sends the transaction *final-ts*^{T} to the client, that adopts *final-ts*^{T} as its new dependency vector clock D_c (Alg. 1 Line 10).

E. Stabilization

PrompTCC uses stabilization to ensure that only write transactions who have been committed in all partitions are made visible (Alg. 2 Line 32). Partitions periodically exchange s_i entry to compute the S_i. If the number of partitions is large, because the message complexity is $O(P^2)$, exchanging s_i entry by broadcast becomes too expensive and not scalable [16]. PrompTCC can efficiently derive the S_i by implementing the same strategy as in GentleRain [16], where the partitions are organized in a tree. The leaf nodes of the tree periodically push their s_i to their parent nodes, the parent gathers the s_i received from the children, adds its own, and send the set up in the tree. This process is repeated up to the root node. The root node fills a full vector S pushes it down the tree. The message complexity is $O(P)$. Each round of stabilization computation takes $2log_F(P)$ round trips, where F is the fanout of the aggregation tree. It is important to note that the stabilization is not strictly required for correctness. However, without the stabilization, the write transactions would have higher visibility latency and clients would read staler versions.

F. Read-Only Transactions

A read-only transaction allows a client to read multiple keys that may be stored in multiple partitions. PrompTCC implement read-only transactions in one round for most scenarios, and in at most two rounds in unfavorable conditions. Read-transactions are never blocked by other (read or write) transactions. A read-only transaction T is executed as follows:

PrompTCC starts by collecting the most recent visible value of each key in the first round of communication. The second round is executed if the first round fails to deliver versions from a consistent snapshot; this may happen if there is an update transaction that executes concurrently with the read transaction, and writes into two or more keys accessed by the read transaction.

Let $\mathcal{P}(T)$ be the set of partitions that store keys read by T. In the first round, the client sends, in parallel, the list of keys it wants to read along with its D_c to each partition $i \in \mathcal{P}(T)$. Each partition, upon receiving this message, uses D_c to update S_i by taking the max of each entry (i.e., $S_i[p] =$ *max*$(S_i[p], D_c[p]), \forall p \in P$) (Alg. 2 Line 37). This is possible since D_c represents the maximum timestamp the client has ever seen, and if the effects of a transaction are already visible it means that the transaction has already been committed on all partitions and thus making possible to advance S_i to D_c safely. This ensures that all writes that the client has observed in the past will be visible, thus ensuring atomicity and causality.

The involved partitions then return to the client the value of their snapshot vector clock S_i and, for each key k, the most recent version $\langle$*value*$_k^v$, *ts*$_k^v\rangle$ that is *visible* (Alg. 2 Line 40).

When the client receives a response from every participant, it sets *max-read-ts*^{T}, the maximum read timestamp for transaction T, a vector that captures the max values from all timestamps that have been read in the first round, i.e., *max-read-ts*$^T[j] = \text{MAX}(ts_k^v[j])\forall k, j$ (Alg. 1 Line 24). It also sets its dependency vector $D_c =$ *max-read-ts*^{T}.

The set of values read during the first round are guaranteed to be from a consistent snapshot if all partitions $i \in \mathcal{P}(T)$ are in future of the maximum read timestamp, i.e., they have snapshot vector clock S_i that is greater or equal than *max-read-ts*^{T}. If the condition is met, the client ends the transaction and returns the values to the client. If the condition is not met, the client has read from a snapshot that may be inconsistent and must execute a second round.

In the second round, the client sends again its D_c value (already updated with the value of *max-read-ts*^{T}) to the participants that may have returned inconsistent values in the first round. Partitions receiving this request, update their snapshot vector clock again (as in the the first round) and return the latest versions with $\langle$*value*$_k^v$, *ts*$_k^v\rangle$ that have *ts*$_k^v \leq$ *max-read-ts*^{T} (Alg. 2 Line 49). The second round ensures that a coherent causal snapshot will be returned. Thanks to the stabilization protocol, all transactions from any visible value are guaranteed to have been committed at all participating partitions. By advancing the snapshot vector clock, partitions make all committed concurrent transactions visible, thus ensuring that no third round is needed.

G. Read-Write Transactions

In the TCC model a Read-Write transaction can be implemented by buffering all writes and by applying them when the transaction commits. Therefore, Read-Write transactions

can be modeled by the execution of a Read-Only transaction immediately followed by a Write-Only transaction that applies the write set.

V. Using Metadata Parsimoniously

It is known that the use of large amounts of metadata may be a source of overhead and impair the system performance [20], [24]. This is a concern because PromptTCC uses vector clocks whose size grows linearly with the number of partitions in the system. In fact, as we have shown in the evaluation section, the overhead incurred by the size of the vector clock may be non-negligible. Fortunately, it is possible to optimize the algorithm described in the previous section to avoid sending the complete vector clock in most messages. These optimizations mitigate the costs of using vector clocks.

We now describe a refined version of PromptTCC that offers the same guarantees while exchanging a fraction of the metadata used by the version described in Section IV. We recall that vector clocks are exchanged between the client and the participants during the execution of a read-only transaction. Namely, in both rounds, the client sends to each partition its dependency vector D_c. Each partition sends back its snapshot vector clock S_i and, for each value read u, the timestamp of that value $u.ts$. In the following, we show that it is possible to avoid exchanging the full set of entries of these vector clocks.

First, the metadata that a client sends while executing a read transaction can be reduced. $D_c[i]$ captures the greatest version observed from partition i. This means that if the client observed this version, the transaction is committed in every partition and can become visible safely. So the client only needs to send $D_c[i]$ instead of sending the whole D_c. In fact, in our optimized implementation, the client also sends to each partition the lowest entry in $D_c[]$, i.e., $\text{MIN}(D_c[])$. The reason for this will become clear next. Thus, the client only sends two scalars to each partition.

Also, a participant can avoid sending the entire S_i back to the client when serving a read request. Instead of sending the S_i, a participant may send only the scalar that corresponds to the highest version visible in the corresponding partition. This optimization replaces the S_i vector clock by a single scalar.

Finally, timestamps of the values read can be compressed. First, instead of sending the individual timestamps of each value read, the partition can send a single timestamp *local-max-read-ts*$_i^T$, with the max value of all read timestamps. Furthermore, the partition only needs to send entries from *local-max-read-ts*$_i^T$ that are greater than the corresponding entries in D_c. Because we avoid sending the entire vector D_c from the client to the partition, the partition conservatively sends back any entry from *local-max-read-ts*$_i^T$ that is greater than the value $\text{MIN}(D_c[])$ provided by the client.

As a result of the last two optimizations above, a partition send back to the client a number of scalars that is always equal or smaller than P and, in many cases, just a two scalars. Together, these optimizations significantly reduce the amount of metadata exchanged by PromptTCC.

VI. Correctness Argument

We now show that PromptTCC implements TCC by showing that the snapshot read by a transaction is causally consistent and respects the atomicity of committed transactions. This proof is inspired by [3]. In the following we use $u.ts$ to denote the final commit timestamp assigned to the transaction that has produced update u.

Proposition 1. *If an update u_2 depends on an update u_1, then $u_2.ts > u_1.ts$.* An update u_2 depends on u_1 if the client's previous read transaction u_2 read from a snapshot that contains u_1. From Alg. 1 Line 26, the following inequality holds: $D_c \geq u_1.ts$. Since the *final-ts*^{T} entry for each participating partition is generated by a sequence number s_i attributed by the partition i to that transaction and s_i is monotonically increasing. Moreover, from Alg. 2 Line 24 all entries in the $u.ts$ corresponding to the non modified partitions will be equal to the entries of the D_c corresponding to that partitions. It is guaranteed that if the u_2 updates keys in the same partition as u_1, from Alg. 2 Line 3 the sequence number will be greater than the s_i attributed to u_1 by partition i, and for the keys updated in different partition from u_1 will be equal or greater than the D_c, as the commit timestamp is greater than D_c and, D_c is greater than $u_1.ts$, so $u_2.ts > u_1.ts$.

Proposition 2. *S_i implies that the partition p_i has received all updates with ts $\leq S_i$.* Now we show that there are no prepared updates with *ts* $\leq S_i$. When updating $S_i[i]$, the partition i finds the minimum prepared timestamps from the transactions in the prepared phase. As s_i is monotonically increasing and the $S_i[i]$ represents the minimum prepared timestamp minus 1 (Alg. 2 Line 27), it is guaranteed that future transactions will receive a *ts* which is greater than or equal to this minimum prepared timestamp, guaranteeing that the partition has already committed all updates for the snapshot S_i.

Proposition 3. *Reads return values from a causally consistent snapshot.* Propositions 1 and 2, together, guarantee that by including all updates in a partition with *ts* $\leq S_i$, the read returns values from a causally consistent snapshot. This is true even when reading from multiple partitions because it reads from the same snapshot.

Proposition 4. *Reading from a snapshot respects atomicity.* Atomicity is not violated even though updates are made visible independently by each partition. All updates from a transaction belong to the same snapshot because they receive the same *final-ts*^{T} . The visibility of an update is delayed until the same snapshot is available in all (accessed) partitions, thus reading all or no updates from a transaction.

The fact that PromptTCC implements TCC derives directly from Propositions 3 and 4, as every transaction reads from a causally consistent snapshot (Prop. 3) that includes all effects (Prop. 4) of its causally dependent transactions.

VII. Garbage Collection of Obsolete Versions

To limit the number of versions maintained in the system, partitions need to implement some sort of garbage collection

mechanism to delete obsolete versions. PrompTCC does not require the availability of synchronized clocks to operate. However, it can leverage loosely synchronized clocks when available, to implement garbage collection more efficiently. In particular, PrompTCC also implements the garbage collection mechanism proposed in Eiger [14], which discards old versions based on the passage of time as follows: First, transactions are aborted and re-started by the client, if they take more than a predefined timeout to complete. Second, each partition only retains old versions of keys for the duration of this timeout, such that they can still be served if needed during a second round.

VIII. Fault Tolerance

In a cloud environment, faults of components such as servers, network equipment, or even power outage are common [5]. So the design of the system needs to take in to account these types of scenarios. PrompTCC tolerates server failures by having each shard be served by a logical server that is implemented by a replica group of multiple servers. These replicas coordinate by running Paxos [30].

Client failure during transaction execution is handled considering its transaction type upon failure. Failure during a read transaction requires no recovery, as neither the client holds meaningful state nor the system state is affected. If failure occurs during a write transaction, it is handled depending on the left-over state after failure. A partition begins a recovery protocol once a timeout is reached while waiting for a committing transaction to progress. A partition acts as transaction coordinator and verifies the state of all participating partitions. If all have received the transaction write operations, the coordinator can complete the transaction following the clients commit protocol. Otherwise, the transaction is aborted.

If a network partitioning occurs between the servers running the partitions, the client reads will continue to return results, however returning stale results. Clients executing write transactions will stall until the partition is resolved.

IX. Evaluation

This section presents an extensive experimental evaluation of PrompTCC. To perform this evaluation we have implemented PrompTCC on top of Cassandra and deployed it in *AWS*. In the evaluation, we compare the performance of PrompTCC, namely in terms of throughput and latency, with a set of competing alternatives, namely: i) an eventual consistent system EC with no support for transactions; this allows to assess to overhead imposed by the different implementations of TCC. The eventual consistent system offers a baseline for throughput and latency, as it makes updates visible as soon as they are received, only needs one round to execute read transactions, and does not require any additional metadata; ii) a conservative protocol, namely Wren [20], that implements read-only transaction with a fixed cost of 2 communication rounds; iii) an optimistic protocol, namely Eiger [14], that can take up to 3 communication rounds to execute a read-only transaction, and; iv) finally, to assess the relevance of the metadata optimizations described in Section V, we also provide results for a basic version of PrompTCC that always exchanges full vector clocks, denoted PrompTCCnaive.

TABLE II: Parameters of the dynamic workload generator.

Parameter	Default	Range
Keys/Read	5	2-64
Keys/Write	5	2-64
Servers	8	1-64
Partitions	1600	200-12800
Value Size (B)	128	1 - 2048
Write Fraction	0.1	0.01-0.5
Zipfian Constant	0.99	0.5-0.99

A. Experimental Setup

Implementation. All systems used in the evaluation have been implemented using the same code base, as variants of Cassandra. We use Cassandra as our EC baseline, as done in [14]. Reads return immediately and writes are made visible upon execution, without relying on quorums. For Eiger, we used the original code from the authors, that is based on a fork of Cassandra version 1.0.7. We have implemented Wren and both the full and the naive versions of PrompTCC using Eiger's codebase.

Deployment. For running the experiments, the prototype was deployed in *AWS*. In these deployment, each server run within *m5.xlarge* instance with 4 vCPUs and 16 GB of memory. Each server machine runs one instance of the server process, with each server process being responsible for 200 logical shards. In order to benchmark the proposed architecture we have used a modified version of the stress test of Cassandra. The modifications we have introduced were the minimum necessary to make it work with the new client library.

Workload Characterization. Clients populate the key-value store before running the experiments. The experiments run in five trials of 2 minutes (this is twice the time used in [14], and more than enough for the system to reach a steady throughput). Furthermore, we have ignored the first and last quarters of each trial to avoid experimental artifacts due to warm up and cool down. We deploy one client server per server machine, each with its own client library and enough client processes to saturate the system. Clients are configured to execute requests to the system in closed-loop, waiting for the response to the previous request before placing a new request. The experiments used the workload generator from Eiger with Zipfian traffic generator, following different read and write ratios following the workloads present in Facebook's TAO [13]. As discussed before, read-write transactions are captured by the sequential execution of a read-only transaction followed by a write-only transaction. The keys accessed by each request are selected using a Zipfian distribution, with parameter 0.99, which is the default in YCSB [31] and resembles the strong skew that characterizes many production systems [12]. Our default workload uses the 90:10 read/write

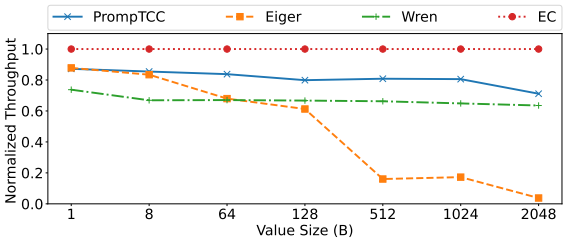

Fig. 2: Normalized throughput as a function of the value size.

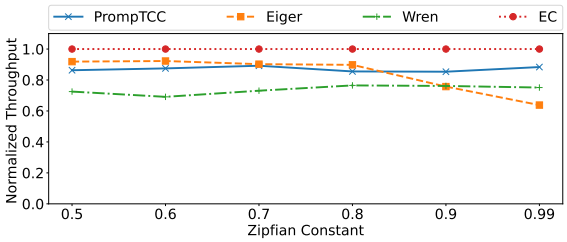

Fig. 3: Normalized throughput as a function of the workload skew.

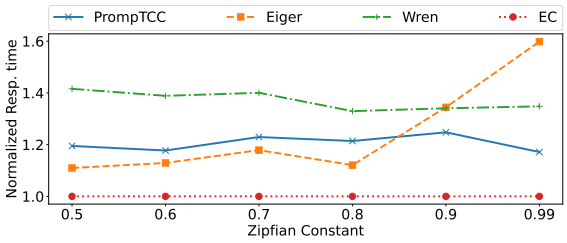

Fig. 4: Normalized response time as a function of the workload skew.

ratio and run transactions that access 5 keys. We also consider variations of this workload in which we change the value of one parameter and keep the others at their default values. The workload parameters are summarized in the Table II. We use the number of transactions executed by the clients per second as the throughput metric.

B. Data Size

We begin by showing the impact of multiple read rounds on the performance for different sizes of data values. Figure 2 presents the normalized throughput of the different systems w.r.t the eventual consistent baseline, when executing transactions with a 90:10 read/write ratio, as a function of the size of values accessed in each transaction (all transactions access 5 keys). It can be observed that PrompTCC is the system that better approximates the performance of an eventually consistent system. As expected, for very large values, the costs involved in the coordination becomes less relevant when compared with data transfer and disk access times. As a result, for value sizes greater than 512 bytes the performance of PrompTCC and Wren start approximating. Conversely, the throughput of Eiger falls considerably for large data value sizes, presenting a 96% penalty compared to the eventual consistent baseline. This happens because with Eiger 40% of the requests require a second round of reads, where data values need to be transferred again; naturally, the overhead of the second transfer is more noticeable for large values. PrompTCC may also need a second round of reads but avoids this penalty because the fraction of request where the second round is needed is negligible ($\approx 2\%$).

C. Data Access Skew

We now study the effect of the workload skew in the performance. For this experiment, we varied the skew in the keys accessed by transactions (i.e., some keys are more accessed than others) and measured its effect of the access skew on the throughput. Thus, we modeled the data access pattern using a Zipfian distribution: a workload with moderate skew is captured using a Zipfian constant value of 0.9 and a value of 0.99 captures a workload with high skew.

Figure 3 presents the normalized throughput of the different systems as a function of the contention in the key access. We can observe that, for workloads with small skew, Eiger and PrompTCC present almost identical performance, positioned between EC and Wren. However, as the workload becomes more skewed, we can observe a performance degradation in Eiger, given that a second round for reads become more likely. In highly skewed workloads, the throughput of Eiger falls below Wren. In face of highly skewed workloads, which are common in practical deployments, PrompTCC exhibits only 12% performance penalty compared to eventual consistency (against 25% by Wren and 37% by Eiger).

Figure 4 presents the response time of the different systems, when executing transactions with a 90:10 read/write ratio with the default workload, as a function of access skew. We can observe that PrompTCC is able to maintain a steady response time overhead of just 20% with the increase in skew. Eiger increases drastically, as more concurrent operations lead to a higher probability of a second, or even third, round for reads, resulting in a 60% response time degradation.

D. Read-Set Size

Figure 5 presents the normalized throughput of the different systems compared to eventual consistency, when executing transactions with a 90:10 read/write ratio, as a function of the read set size of each transaction (i.e., the number of keys read by a transaction). As it can be observed, PrompTCC is able to closely follow the performance of a system that only offers eventual consistency, with a penalty in the order of 15%. For smaller read sets, both Eiger and Wren exhibit a penalty in the order of 25% and 30% respectfully.

As the number of keys read by transactions increases each transaction takes longer to conclude and, naturally, the throughput decreases. For large number of reads, the costs associated with CPU utilization and with disk access time start to become dominant, and the overhead of the additional round introduced by Wren becomes less relevant. Still PrompTCC

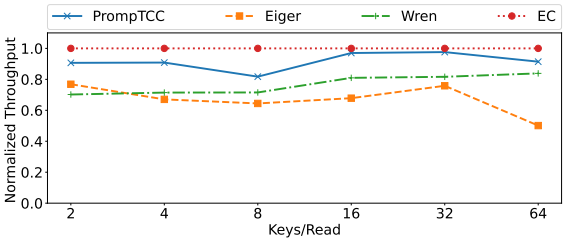

Fig. 5: Normalized throughput as a function of read Tx length

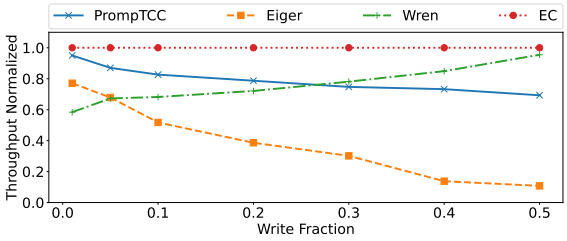

Fig. 6: Normalized throughput vs write/read ratio.

is the one that keeps performing closely to EC, with a slowdown of 10%. Wren has a 20% slowdown compared to EC. Eiger exhibits the lowest throughput with a slowdown of 50% compared to EC, due to the fact that reading more keys increases the chances of the first round returning an inconsistent view and partially committed transaction. In fact for large number of keys Eiger's transactions tend to require an average 2.9 rounds compared to 1.3 of PrompTCC.

E. Write/Read Ratio

Figure 6 presents the normalized throughput of the different systems when executing transactions with different write/read ratios (all transactions access 5 keys of 128 bytes). The figure unveils an interesting limitation of Eiger. For residual write ratios (for instance, 0.01%), Eiger and PrompTCC have almost the same number of second-round reads. Therefore, one could expect that both systems would exhibit the same performance. In reality, the throughput of Eiger is 20% lower that that of PrompTCC. The cause for this difference lies in the way Eiger keeps dependencies in the client. Eiger maintains a tree of dependencies that can only be purged when a write transaction is executed. For small ratio of write transactions the dependency tree keeps increasing and the CPU utilization at the client becomes a bottleneck.

The figure also shows that the throughput of PrompTCC decreases, as the percentage of write transactions increases. This is due to the higher write transaction cost and a higher probability of a second round. In fact, it is possible to observe that for large write/read ratios, Wren eventually outperforms the PrompTCC. Still, for most realistic write/read ratios, PrompTCC outperforms both Eiger and Wren.

F. Scalability

The ability to distribute the data across different servers is essential for scalability. Thus, all cloud storage systems split

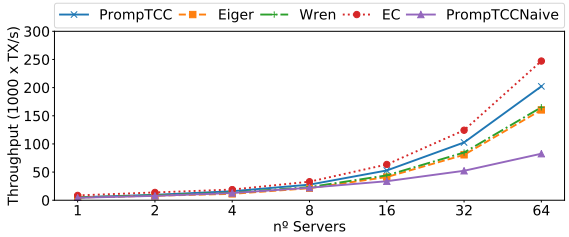

Fig. 7: Throughput as a function of the number of servers and proportionally changing the number of clients and keys.

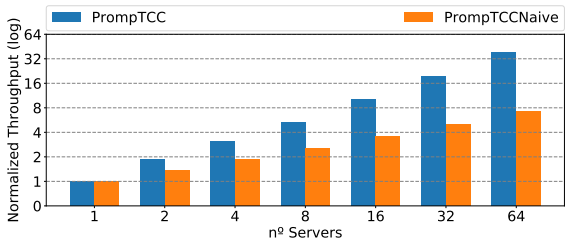

Fig. 8: Normalized throughput of PrompTCC changing the total number of servers and the number of clients and keys proportionally. Bars are normalized against 1 server.

the data into logical partitions and then let a different set of servers handle each partition. If a transaction accesses data that are in different servers, coordination among different servers is required. This experiment focuses on understanding how the system can scale horizontally. We do so by studying the effect of the number of partitions on the system throughput. We have been able to deploy the system on AWS using up to 64 servers and 12800 partitions.

We start by showing that PrompTCC is able to horizontally scale and sustain additional clients as more servers are added to the system. For this experiment, we augmented the number of servers and augmented proportionally the number of clients, such that the operations submitted by N client machines are fully loading N servers. Moreover, we proportionally increase the number of keys to keep the level of concurrency among clients constant for all points in the experiment.

Figure 7 shows the throughput for each system as we scale the number of servers from 1 to 64. We can observe that all systems scale almost linearly with the exception of the naive version of PrompTCC due to the linear increase of the metadata size. We are also able to observe that PrompTCC does not suffer from the same effect, showing the benefits of our vector clock optimizations.

We now we analyze in detail the difference in performance between the two implementations of PrompTCC. Figure 8 shows the throughput of both versions as we scale the number of servers from 1 to 64 (note that both axes are in log scale). The bars show the throughput normalized against the throughput of 1 server. PrompTCC scales out as the number of servers increases. However, this increase is not linear from 1 to 8 servers. The configuration with 1 server has the benefits of

batching: all operations that involve multiple keys are executed on a single machine. As the number of servers increases, the transactions span across multiple servers, and thus the system is no longer able to exploit batching effectively. In fact, in a system with many servers, most transactions tend always to access 5 different servers. This effect was also present in the original evaluation of Eiger [14].

Nevertheless, the ability of the system to scale perfectly is limited due to a number of overheads that are associated with the maintenance of multiple servers, such as background stabilization procedures or increased size of metadata. In particular, the potential negative effect of the increase in the metadata size, as a function of the number of servers, is illustrated by the performance of the naive implementation of PrompTCC. The figure shows that the optimization mechanisms embedded in PrompTCC, that reduce the amount of metadata exchanged, are effective at mitigating this negative impact.

X. Conclusions

In this paper we propose a novel system, named PrompTCC, that implements Transactional Causal Consistency (TCC) efficiently. PrompTCC supports write-transactions, offers non blocking reads, and is able to process most read-transactions in one round. To achieve this, PrompTCC makes a clever use of vector clocks, where only a small fraction of the entries are exchanged at each step of the protocol. We have implemented PrompTCC on top of Cassandra and have experimentally evaluated it on *AWS* against eventual consistency, Eiger [14] and Wren [20]. By avoiding multiple duplicated reads, PrompTCC presents competitive performance to eventual consistency baseline, presenting just 10% throughput and 20% latency penalties while offering stronger isolation and consistency to transactions, outperforming the up to 37% and 60% throughput and latency penalties of previous state-of-the-art systems.

Acknowledgments: This research partially supported by the Fundação para a Ciência e a Tecnologia (FCT) via scholarship UI/BD/153590/2022, the INESC-ID grant UIDB/50021/2020 (DOI:10.54499/UIDB/50021/2020) and the DACOMICO project (financed by the OE with ref. PTDC/CCI-COM-/2156/2021).

References

[1] M. Aguilera, J. B. Leners, and M. Walfish, "Yesquel: Scalable SQL storage for web applications," in *SOSP*, Monterey, CA, Oct. 2015.

[2] W. Vogels, "Eventually consistent," *Comm. of the ACM*, vol. 52, no. 1, 2009.

[3] D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *ICDCS*, Nara, Japan, Jun. 2016.

[4] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan, "Challenges to adopting stronger consistency at scale," in *HOTOS*, Kartause Ittingen, Switzerland, 2015.

[5] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," in *VLDB*, Trento, Italy, Aug. 2013.

[6] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Scalable atomic visibility with RAMP transactions," *ACM TODS*, vol. 41, no. 3, 2016.

[7] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. W. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, 1995.

[8] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," in *PODC*, San Sebastián, Spain, Jul. 2015.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *SOSP*, Santa Clara (CA), USA, 2007.

[10] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS," in *SOSP*, Cascais, Portugal, Oct. 2011.

[11] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, "The SNOW theorem and latency-optimal read-only transactions," in *OSDI*, Savannah (GA), USA, Nov. 2016.

[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, 2012.

[13] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li *et al.*, "TAO: Facebook's distributed data store for the social graph," in *ATC*, San Jose (CA), USA, Jun. 2013.

[14] W. Lloyd, M. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *OSDI*, Lombard (IL), USA, Apr. 2013.

[15] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *SOCC*, San Jose, CA, Oct. 2013.

[16] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "GentleRain: Cheap and scalable causal consistency with physical clocks," in *SOCC*, Seattle, WA, Nov. 2014.

[17] S. Almeida, J. a. Leitão, and L. Rodrigues, "ChainReaction: A causal+ consistent datastore based on chain replication," in *Eurosys*, Prague, Czech Republic, Apr. 2013.

[18] H. Lu, S. Sen, and W. Lloyd, "Performance-optimal read-only transactions," in *OSDI'20*, Virtual Event, Nov. 2020.

[19] D. Didona, P. Fatourou, R. Guerraoui, J. Wang, and W. Zwaenepoel, "Distributed transactional systems cannot be fast," in *SPAA*, Phoenix, AZ, 2019.

[20] K. Spirovska, D. Didona, and W. Zwaenepoel, "Wren: Nonblocking reads in a partitioned transactional causally consistent data store," in *DSN*, Luxembourg City, Luxembourg, Jun. 2018.

[21] R. Ladin, B. Liskov, and L. Shrira, "Lazy replication: exploiting the semantics of distributed services," in *PODC*, Quebec City, Quebec, Canada, 1990, p. 43–57.

[22] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, 1978.

[23] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," in *LADIS*, Big Sky, MT, Oct. 2009.

[24] M. Bravo, L. Rodrigues, and P. Van Roy, "Saturn: A distributed metadata service for causal consistency," in *Eurosys*, Belgrade, Serbia, 2017.

[25] S. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I can't believe it's not causal! scalable causal consistency with no slowdown cascades," in *OSDI*, Boston (MA), USA, Mar. 2017.

[26] B. Charron-Bost, "Concerning the size of logical clocks in distributed systems," *Information Processing Letters*, vol. 39, no. 1, 1991.

[27] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," *Information Processing Letters*, vol. 43, no. 1, 1992.

[28] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distributed computing*, vol. 7, no. 3, 1994.

[29] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating synergies between memory, disk and log in log structured key-value stores," in *ATC*, Santa Clara, CA, 2017.

[30] L. Lamport, "The part-time parliament," *ACM TOCS*, vol. 16, no. 2, 1998.

[31] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SOCC*, Las Vegas (NV), USA, 2010.