Tunable Causal Consistency: Specification and Implementation

Xue Jiang, Hengfeng Wei*, Yu Huang
State Key Laboratory for Novel Software Technology, Nanjing University, China
xuejiang1225@gmail.com, {hfwei, yuhuang}@nju.edu.cn

Abstract—To achieve high availability and low latency, distributed data stores often geographically replicate data at multiple sites called replicas. However, this introduces the data consistency problem. Due to the fundamental tradeoffs among consistency, availability, and latency in the presence of network partition, no a one-size-fits-all consistency model exists.

To meet the needs of different applications, many popular data stores provide tunable consistency, allowing clients to specify the consistency level per individual operation. In this paper, we propose tunable causal consistency (TCC). It allows clients to choose the desired session guarantee for each operation, from the well-known four session guarantees, i.e., read your writes, monotonic reads, monotonic writes, and writes follow reads. Specifically, we first propose a formal specification of TCC in an extended \((vis, ar)\) framework originally proposed by Burekhardt et al. Then we design a TCC protocol and develop a prototype distributed key-value store called TCCSTORE. We evaluate TCCSTORE on Aliyun. The latency is less than 38ms for all workloads and the throughput is up to about 1900 operations per second. We also show that TCC achieves better performance than causal consistency and requires a negligible overhead when compared with eventual consistency.

Keywords—Causal Consistency; Tunable Consistency; Session Guarantees

I. INTRODUCTION

Data Consistency Models. To achieve high availability and low latency, distributed data stores often geographically replicate data at multiple sites called replicas [1], [2]. However, this introduces the data consistency problem among replicas. Over the past forty years, more than 50 consistency models have been proposed [3], [4], ranging from linearizability [5] to eventual consistency [2], [6]. According to the CAP theorem [7], [8] and the PACELC tradeoff [9], there are fundamental tradeoffs among consistency, availability, and latency in the presence of network partition. Therefore, no a one-size-fits-all consistency model exists [3].

Tunable consistency. To meet the needs of different applications, many popular distributed data stores begin to provide tunable consistency [10]–[13], allowing clients to specify the consistency level per individual operation. Amazon DynamoDB provides eventual consistency as default and strong consistency with ConsistentRead [10]. Apache Cassandra offers a number of fine-grained read and write consistency levels, such as ANY, ONE, Quorum, and ALL [11]. For example, a read of level Quorum returns the value after a quorum of replicas have responded. MongoDB provides tunable consistency by exposing the writeConcern and readConcern parameters that can be set per operation [12]. For example, a read with readConcern = majority guarantees that the returned value has been written to a majority of replicas. Azure Cosmos DB offers five well-defined consistency levels for read operations, namely, from strongest to weakest, Strong, Bounded staleness, Session, Consistent prefix, and Eventual [13]. For example, Strong consistency offers linearizability [5], while Eventual consistency provides no ordering guarantees for reads.

Causal Consistency and Session Guarantees. Causal consistency guarantees that an update does not become visible to clients until all its causal dependencies are visible [2], [12], [14]–[19]. Consider the classic “Lost-Ring” example [16]. Alice first posts “I lost my ring”, and then posts “I have found it” after a while. Bob sees both posts of Alice, and comments “Glad to hear it”. Causal consistency can avoid the undesired situation that Charlie sees the comment by Bob but do not see the second post of Alice. Causal consistency has been shown equivalent to the conjunction of the four well-known session guarantees [20], i.e., read your writes (ryw), monotonic reads (mr), monotonic writes (mw), and writes follow reads (wfr) [21]. ryw ensures that any write becomes visible to the subsequent reads in the same session. mr requires successive reads on the same session observe monotonically increasing sets of writes. mw requires writes on the same session take effect in the session order. Finally, wfr establishes causality between two writes \(w_1\) and \(w_2\) via a read \(r_1\), if \(r_1\) reads from \(w_1\) and \(w_2\) follows \(r_1\) on the same session.

Tunable Causal Consistency: Motivation. Terry has demonstrated that it is desirable for different participants of a baseball game to maintain the score with different session guarantees [3]. For example, ryw is sufficient for the official scorekeeper to retrieve the latest score before producing a new one, while a radio reporter may need mr to ensure that the observed baseball scores are monotonically increasing. Therefore, it would be beneficial for a distributed data store to provide tunable causal consistency, i.e., allowing clients to choose the session guarantee per individual operation.

Tunable Causal Consistency: Related Work. As far as we

---

1 How is the consistency level configured? https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmConfigConsistency.html

2 Consistency levels in Azure Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels
know, NuKV is the only distributed key-value store that provides tunable causal consistency [22]. However, it has two major drawbacks in terms of specification and implementation.

- **Drawback in Specification.** NuKV lacks a formal specification of tunable causal consistency. Instead, NuKV uses implementation itself as specification. Specifically, it first defines three write sets to track the writes committed at each server, the writes issued by each client, and the writes observed by each client, respectively. Then the four session guarantees are defined as different constraints on these three write sets.

- **Drawback in Implementation.** NuKV implements per-key session guarantees. It keeps track of session guarantees for each partition, being able to avoid the problem of slowdown cascades cross keys maintained by different partitions. However, per-key session guarantees are insufficient for some applications. For example, the “Lost-Ring” example mentioned above involves two keys representing Alice’s and Bob’s posts, respectively. To avoid the undesired situation, causality should be established across these two keys.

In this paper, we propose a formal specification and a general implementation of tunable causal consistency (TCC).

- **Our First Contribution: Specification (Section II).** We formally specify TCC in an extended \((\mathit{vis, ar})\) framework originally proposed by Burckhardt et al. [1], [2]. The visibility relation \(\mathit{vis}\) specifies, for each operation, the set of operations that are visible to it. The arbitration relation \(\mathit{ar}\) indicates how the system resolves the conflicts due to concurrent operations that are not visible to each other. To specify TCC, we extend the \((\mathit{vis, ar})\) framework by adding a consistency level for each operation. Then we formalize TCC in two steps. First, we define the visibility relation for each individual session guarantee. Then we consider the multi-level constraints, which specify how one session guarantee influences another in terms of the visibility and arbitration relations. We find that in TCC any two session guarantees do not impose any constraints on each other.

- **Our Second Contribution: Implementation (Sections III and IV).** We design a TCC protocol which is general in the sense that it supports multiple keys. The protocol uses vector clocks to track dependencies for each kind of session guarantee. Then, we implement a prototype distributed key-value store called TCCSTORE, which provides TCC in the common sharded cluster deployment. We evaluate the performance of TCCSTORE on Aliyun.\(^{3}\) The latency is less than 38ms for all workloads and the throughput is up to about 1900 operations per second. We also show that TCC achieves better performance than causal consistency and requires a negligible overhead when compared with eventual consistency.

### II. Tunable Causal Consistency

In this section we formally specify TCC in the \((\mathit{vis, ar})\) framework [1], [2]. We follow the description of the \((\mathit{vis, ar})\) framework in [23] and extend it to support tunable consistency as needed. As discussed in Section I, we consider the set \(L \triangleq \{\mathit{ryw, mr, mw, wfr}\}\) of four session guarantees. We distinguish the set \(L_e \triangleq \{\mathit{ryw, mr}\}\) of two session guarantees constraining reads from that \(L_w \triangleq \{\mathit{mw, wfr}\}\) of the other two session guarantees constraining writes.

#### A. Relations and Orderings

Given a set \(A\), a binary relation \(R\) over \(A\) is a subset of \(A \times A\), i.e., \(R \subseteq A \times A\). For \(a, b \in A\), we use \(a \rightarrow b\) to denote \((a, b) \in R\). We use \(R^{-1}\) to denote the inverse relation of \(R\), i.e., \((a, b) \in R \iff (b, a) \in R^{-1}\). We define \(R^{-1}(b) \triangleq \{a \in A \mid (a, b) \in R\}\).

For two relations \(R\) and \(S\) over \(A\), their composition is \(R; S \triangleq \{(a, c) \mid \exists b \in A : a \xrightarrow{R} b \land b \xrightarrow{S} c\}\). For some subset \(A' \subseteq A\), the restriction of \(R\) to \(A'\) is \(R|_{A'} \triangleq R \cap (A' \times A)\). Let \(f : A \rightarrow B\) be a function from \(A\) to \(B\) and \(A' \subseteq A\). The restriction of \(f\) to \(A'\) is \(f|_{A'} \triangleq f \cap (A' \times B) = \{(a, f(a)) \mid a \in A'\}\).

A relation is called a (strict) partial order when it is irreflexive and transitive. A relation which is a partial order and total is called a total order.

#### B. Read/Write Registers

We focus on the key-value store which maintains a collection of integer read/write (or named get/put) registers. An integer (read/write) register supports two operations: \(\mathit{wr}(v)\) writes value \(v \in \mathbb{Z}\) to the register, and \(\mathit{rd}\) reads value from the register. We use \(\bot\) to indicate that writes return no values. Let \(\mathit{Op} = \{\mathit{wr}, \mathit{rd}\}\) and \(\mathit{Val} = \mathbb{Z} \cup \{\bot\}\).

The sequential semantics of registers is defined by a function \(\mathit{eval}_{\mathit{reg}} : \mathit{Op}^* \times \mathit{Op} \rightarrow \mathit{Val}\) that, given a sequence of operations \(S\) and an operation \(o\), determines the return value \(\mathit{eval}_{\mathit{reg}}(S, o) \in \mathit{Val}\) for \(o\) when \(o\) is performed after \(S\) [2]. If \(o\) is a \(\mathit{rd}\) operation, it returns the value of the last preceding \(\mathit{wr}\), or the initial value 0 if there are no prior writes [23]. Formally, for any operation sequence \(S\),

\[
\mathit{eval}_{\mathit{reg}}(S, \mathit{wr}(v)) = \bot,
\]

\[
\mathit{eval}_{\mathit{reg}}(S, \mathit{rd}) = v, \text{ if } \mathit{wr}(0) S = S_1 \mathit{wr}(v) S_2 \text{ and } S_2 \text{ contains no } \mathit{wr} \text{ operations.}
\]

#### C. Histories

Clients interact with the key-value store by performing operations on keys. The interactions visible to clients are recorded in a history. To support tunable consistency, we tag each operation with a consistency level.

**Definition 1** (Histories). A history is a tuple \(H = (E, \mathit{op}, \mathit{lvl}, \mathit{rval}, \mathit{so})\) such that

- \(E\) is the set of all events of operations invoked by clients in a single computation;
- \(\mathit{op} : E \rightarrow \mathit{Op}\) describes the operation of an event;
Fig. 1. A history consisting of two sessions p_1 and p_2 and two registers x and y. Here r.w(r, l) ◁ l denotes the operation of writing value r to register r with consistency level l (the return value ◁ l is omitted). Similarly, r.r(r, l) ◁ l denotes the operation of reading value r from register r with consistency level l. We use labels, such as a and b, to make events unique.

\[ p_1 : \begin{align*} & x.w(1, mw) \rightarrow y.r(ryw) \rightarrow y.w(3, wfr) x.r \rightarrow y.w(1, mw), \\ & y.w(2, mw) \rightarrow y.r(ryw) \rightarrow y.w(3, wfr) y.r \rightarrow y.w(1, mw). \end{align*} \]

\[ p_2 : \begin{align*} & y.w(1, mw) \rightarrow y.r(ryw) \rightarrow y.w(3, wfr) y.r \rightarrow y.w(1, mw), \\ & x.r(1, mw) \rightarrow x.w(2, mw) \rightarrow x.r(1, mw), \\ & y.w(2, mw) \rightarrow y.r(ryw) \rightarrow y.w(3, wfr) y.r \rightarrow y.w(1, mw). \end{align*} \]

Fig. 2. An abstract execution of the history in Fig. 1. Assume that x.w(1, mw) has been applied on replica R_1 when y.r(ryw) is executed on R_1. y.w(2, mw) has been applied on R_1 when x.r(rd) is executed on R_1. y.w(3, wfr) has been applied on replica R_1.

• lvl : E → L specifies the consistency level requested by the operation op(e) of an event e;
• rval : E → Val describes the value returned by the operation op(e) of an event e;
• so ⊆ E × E is a partial order over E, called the session order. It relates operations within a session in the order they were invoked by clients.

For a history H = (E, op, lvl, rval, so), we define:
• E_o is the set of events of all read operations in H.
• E_w is the set of events of all write operations in H.
• For a level l ∈ L, E_{l} \triangleq E_o \cup \{e ∈ E_r | lvl(e) = l\} is the set of events of all write operations and the read operations with consistency level l ∈ L.
• For a level l ∈ L, we use H_l to denote the restriction of H to the events E_l, i.e., H_l \triangleq (E_l, op|E_l, lvl|E_l, rval|E_l, so|E_l).

Example 1. Consider the history in Fig. 1 consisting of two sessions p_1 and p_2 and two registers x and y. We have E_{ryw} = \{x.w(1, mw), y.w(1, mw), y.w(3, wfr), a : y.r(ryw) ◁ b : y.r(ryw) ◁ \}. Moreover, so|E_{ryw} is the session order over E_{ryw}, including x.w(1, mw) → a : y.r(ryw) ◁ b : y.r(ryw) ◁ → y.w(3, wfr).

D. Abstract Executions
To justify the return value of an event in a history, we need to know the set of events that are visible to it and how these events are ordered. These are captured declaratively by the visibility and arbitration relations, respectively [1], [2].

Definition 2 (Abstract Executions). An abstract execution is a triple A = ((E, op, lvl, rval, so), vis, ar) such that

• (E, op, lvl, rval, so) is a history;
• Visibility vis \triangleq \bigcup_{l \in L} vis_l is an acyclic relation, where vis_l ⊆ E_l × E_l is the visibility relation for consistency level l ∈ L;
• Arbitration ar \subseteq E × E is a total order such that vis ⊆ ar.

Example 2. Fig. 2 shows an abstract execution of the history in Fig. 1. Assume that x.w(1, mw) \xrightarrow{vis_{ryw}} b : y.r(ryw) ◁ 1, y.w(2, mw) \xrightarrow{vis_{ryw}} x.r(rd) ◁ 1, and y.w(2, mw) \xrightarrow{ar} y.w(1, mw). In the following, we informally explain the other visibility and arbitration relations required by TCC.

For event b : y.r(ryw) ◁ 1, ryw requires that all writes placed before it in p_2 are visible to it. To satisfy ryw, we add the visibility relation y.w(1, mw) \xrightarrow{vis_{ryw}} b : y.r(ryw) ◁ 1. Similarly, we add the other vis_{ryw} relations as shown in Fig. 2.

For event x.r(rd) ◁ 1, mr requires that it sees all writes visible to b : y.r(ryw) ◁ 1. To satisfy mr, we add the visibility relation y.w(1, mw) \xrightarrow{vis_{ryw}} x.r(rd) ◁ 1 and x.w(1, mw) \xrightarrow{vis_{ryw}} x.r(rd) ◁ 1. To satisfy mr, we add the arbitration relation x.w(1, mw) \xrightarrow{ar} y.w(2, mw).

For event y.w(3, wfr), wfr requires that it should be performed after all the events visible to x.r(rd) ◁ 1. To satisfy wfr, we add the arbitration relations x.w(1, mw) \xrightarrow{ar} y.w(3, wfr), y.w(2, mw) \xrightarrow{ar} y.w(3, wfr), and y.w(1, mw) \xrightarrow{ar} y.w(3, wfr).

E. Consistency Models

Definition 3 (Consistency Models). A consistency model is a set of consistency predicates on abstract executions.

We write A \models P if the consistency predicate P is true on the abstract execution A.

Definition 4 (Satisfaction (Abstract Execution)). An abstract execution A satisfies consistency model M = \{P_1, \ldots, P_n\}, denoted A \models M, if A satisfies each consistency predicate in M. That is, A \models M \iff A \models P_1 \land \cdots \land A \models P_n.

We focus on histories that satisfy some consistency model.

Definition 5 (Satisfaction (History)). A history H satisfies consistency model M = \{P_1, \ldots, P_n\}, denoted H \models M, if it can be extended to an abstract execution that satisfies M. That is, H \models M \iff \exists \text{vis,ar.}(H,\text{vis,ar}) \models C.

F. Return Value Consistency

A common consistency predicate is the consistency of return values. In an abstract execution A, the return value of an event e is determined by its operation context, denoted cxtA(e), which is the restriction of A to the set \text{vis}_i|\text{op}(e) of events visible to e with respect to consistency level lvl(e). Since \text{ar} is a total order, the events in cxtA(e) can be ordered into a sequence to justify rval(e).
Definition 6 (Operation Context). Let $A = ((E, op, lv, rval, so), vis, ar)$ be an abstract execution. The operation context of $e \in E$ in $A$ is defined as

$$ctxt_A(e) \triangleq A|_{vis,e(op, vis, ar)}.$$ 

Accordingly, the return value consistency (RVAL) predicate is defined as follows.

**Definition 7** (Return Value Consistency). For read/write registers $reg$, the return value consistency predicate on an abstract execution $A$ is

$$RVAL(reg) \triangleq \forall e \in E. rval(e) = eval_{reg}(ctxt_A, op(e)).$$

**Example 3.** Consider the abstract execution in Fig. 2. We show how to justify the return value of the event $a : y. rd(yrw) \triangleright 1$. First, $vis_{yrw}(a) = \{x. wr(1, mw), y. wr(1, mw), y. wr(1, mw)\}$. Second, we have $x. wr(1, mw) \xrightarrow{ar} y. wr(2, mw) \xrightarrow{ar} y. wr(1, mw)$. Therefore, $a$ can be justified by the operation sequence $(x. wr(1, mw), y. wr(1, mw))$ and $(x. wr(1, mw), y. wr(2, mw), y. wr(1, mw))$, respectively.

G. Individual Level Constraints

We define TCC in the above extended $(vis, ar)$ framework in two steps: In this section we define the constraints specified by the four session guarantees individually. In the next section we define the multi-level constraints, specifying how one session guarantee influence another in terms of the visibility and arbitration relations.

1) Read Your Writes: $rw$ ensures that any write becomes visible to the subsequent reads in the same session.

**Definition 8** (Read Your Writes ($rw$)). Let $H$ be a history. The constraint $C_{rw}$ for $rw$ is

$$C_{rw} \triangleq \forall e \in E_r, e' \in E_w.$$

$$(1vl(e) = rw \land e' \xrightarrow{so} e) \implies e' \xrightarrow{vis} e.$$ 

Note that write events on other sessions ($x. wr(1, mw)$) can also be visible ($vis_{rwy}$) to $e$. For example, as shown in Fig. 2, $x. wr(1, mw)$ is visible ($vis_{rwy}$) to $b : y. rd(yrw) \triangleright 1$.

**Example 4.** Consider the history in Fig. 2. For event $b : y. rd(yrw) \triangleright 1$ with consistency level $rw$, since $y. wr(1, mw) \xrightarrow{so} b : y. rd(yrw) \triangleright 1$, to satisfy $C_{rw}$, we have $y. wr(1, mw) \xrightarrow{vis} b : y. rd(yrw) \triangleright 1$. Similarly, we have $y. wr(2, mw) \xrightarrow{vis} a : y. rd(yrw) \triangleright 1$ and $x. wr(1, mw) \xrightarrow{vis} a : y. rd(yrw) \triangleright 1$ due to $y. wr(2, mw) \xrightarrow{so} a : y. rd(yrw) \triangleright 1$ and $x. wr(1, mw) \xrightarrow{so} a : y. rd(yrw) \triangleright 1$, respectively.

2) Monotonic Reads: $mr$ requires successive reads on the same session observe monotonically increasing sets of writes.

**Definition 9** (Monotonic Reads ($mr$)). Let $H$ be a history. The constraint $C_{mr}$ for $mr$ is

$$C_{mr} \triangleq \forall e \in E_w, e_1 \in E_r, e \in E_r.$$

$$(1vl(e) = mr \land e_1 \xrightarrow{vis} e_2 \xrightarrow{so} e) \implies e_1 \xrightarrow{vis} e.$$ 

Note that $vis$ is either $vis_{rwy}$ or $vis_{mr}$, and $e_1$ and $e_2$ can be on different sessions.

**Example 5.** Consider the history in Fig. 2. Both $x. wr(1, mw)$ and $y. wr(1, mw)$ are visible ($vis_{mr}$) to $b : y. rd(yrw) \triangleright 1$. Since $b : y. rd(yrw) \triangleright 1 \xrightarrow{so} x. rd(mr) \triangleright 1$, to satisfy $C_{mr}$, both $x. wr(1, mw)$ and $y. wr(1, mw)$ should be visible ($vis_{mr}$) to $x. rd(mr) \triangleright 1$.

3) Monotonic Writes: $mw$ requires writes on the same session take effect in the session order.

**Definition 10** (Monotonic Writes ($mw$)). Let $H$ be a history. The constraint $C_{mw}$ for $mw$ is

$$C_{mw} \triangleq \forall e', e \in E_w.$$

$$(1vl(e) = mw \land e' \xrightarrow{so} e) \implies e' \xrightarrow{ar} e.$$ 

**Example 6.** Consider the history in Fig. 2. Since $x. wr(1, mw) \xrightarrow{so} y. wr(2, mw)$, $C_{mw}$ requires $x. wr(1, mw) \xrightarrow{ar} y. wr(2, mw)$.

4) Writes Follow Reads: $wfr$ establishes causality between two writes $w_1$ and $w_2$ via a read $r_1$, if $r_1$ reads from $w_1$ and $w_2$ follows $r_1$ on the same session.

**Definition 11** (Writes Follow Reads ($wfr$)). Let $H$ be a history. The constraint $C_{wfr}$ for $wfr$ is

$$C_{wfr} \triangleq \forall e_1 \in E_w, e_2 \in E_r, e \in E_w.$$

$$(1vl(e) = wfr \land e_1 \xrightarrow{vis} e_2 \xrightarrow{so} e) \implies e_1 \xrightarrow{ar} e.$$ 

Similar to Definition 9, $vis$ is either $vis_{rwy}$ or $vis_{mr}$, and $e_1$ and $e_2$ can be on different sessions.

**Example 7.** Consider the history in Fig. 2. Since $x. wr(1, mw) \xrightarrow{vis} b : y. rd(yrw) \triangleright 1$ and $b : y. rd(yrw) \triangleright 1 \xrightarrow{so} y. wr(3, wfr)$, $wfr$ requires $x. wr(1, mw) \xrightarrow{ar} y. wr(3, wfr)$.

H. Multi-level Constraints

Intuitively, it is not sufficient to separately satisfy the constraints corresponding to individual levels. It is also necessary to specify how one level influences another. For example, Boughnini et al. provide multi-level constraints for strong and weak consistency levels in [24].

In this section we define the multi-level constraints, which specify how one session guarantee influence another in terms of the visibility and arbitration relations. Consider two events $e'$ of level $l'$ and $e$ of level $l$ such that $e$ immediately follows $e'$ on the same session. We use $\psi_l'$ to denote the constraint that $l'$ imposes on $l$. 

172
To find the constraints imposed by one session guarantee on another, we enumerate all pairs of session guarantees. However, we find that any multi-level constraint has been enforced by a certain constraint for individual session guarantee.

**Definition 12** (Multi-level Constraints). The multi-level constraint for any pair of session guarantees $l', l \in L$ is $\psi_{l'} \subseteq \psi_l$. That is, $l'$ does not impose any constraints on $l$. We explain it by example.

**Example 8.** For example, if $l = rvw$, $e$ needs to observe all the writes preceding it on the same session, regardless of the level of $e'$. Suppose $e'$ is a read and $l = wrf$. The event $e$ should be ordered after all the writes visible to $e'$ in $ar$. This requirement is enforced by $C^{ar\theta}$.

**Definition 13** (Tunable Causal Consistency (TCC)),

$$TCC \triangleq RVAL(reg) \land \bigwedge_{l \in L} C^l \land \bigwedge_{l',l \in L} \psi_{l'}{l} \triangleq \top.$$

**Example 9.** According to Examples 3~8, the history in Fig. 2 satisfies TCC.

## III. Protocol

We assume a distributed key-value store that maintains multiple data items. The data is fully replicated across different datacenters. Inside each datacenter, data is partitioned into several partitions. All datacenters adopt the same partitioning strategy. As shown in Fig. 3, we consider a configuration consisting of $D$ datacenters, each of which consists of $N$ partitions. We assume that each partition is equipped with a physical clock, which generates increasing timestamps. Clocks are loosely synchronized by the classic time synchronization protocol NTP. Client operations can be executed on their local datacenters or remote ones.

We assume the data store keeps multiple versions of each data item. An update on a data item creates a new version of it. Our protocol offers two operations to the clients:

1) **PUT($k$, $v$, $l$):** A PUT operation assigns value $v$ to an item identified by key $k$ while ensuring consistency level $l$. If the item does not exist, a new item with value $v$ is created. If key $k$ exists, a new version with value $v$ is created.

2) **GET($k$, $l$):** A GET operation returns the value of the item identified by key $k$ while ensuring consistency level $l$.

---

**TABLE I**

| Notations | Description |
|-----------|-------------|
| hrvc      | highest read vector clock |
| hwvc      | highest write vector clock |
| cvc_r    | vector clock for reads at client c |
| cvc_w    | vector clock for writes at client c |
| $p^m_d$  | partition m in data center d |
| gsvc_d   | global stable vector clock at $p^m_d$ |
| gsvc_m   | vector clock at $p^m_d$ |
| clock_m  | physical clock at $p^m_d$ |
| PMC_m    | matrix of received cvc_m at $p^m_d$ |
| $\psi(L)$ | the partition that holds key k |
| k         | key |
| v         | value |
| l         | consistency level |
| vc        | vector clock |
| dvc       | dependency vector clock |

---

**A. States**

Table I provides a summary of notations used in the protocol.

1) **Client States:** Each client $c$ maintains two vector clocks: the highest read vector clock hrvc records the maximum timestamp of stable versions read by client $c$, and the highest write vector clock hwvc is the maximum timestamp of versions written by client $c$. The client also maintains two vector clocks $cvc_r$ and $cvc_w$ that record the maximum timestamps of versions read and written by the client, respectively.

2) **Server States:** For each partition $p^m_d$, clock_m records the value its physical clock. The server maintains a vector clock $pvc_m$ of size $D$, where $pvc_m[k]$ indicates that $p^m_d$ has received updates up to $pvc_m[k]$ from partition $p^m_d$. The server also maintains a stable vector $gsvc_m$ of size $D$ to denote the latest globally stable consistent view of the local datacenter known by $p^m_d$. i.e., the view that $p^m_d$ knows to be available at all partitions in the local datacenter. To advance $gsvc_m$, partitions in the same datacenter periodically exchange their $pvc$.

**B. Protocol**

We now informally describe how the PUT and GET operations are executed at clients and servers. We also describe the clock management and replication mechanism. The pseudocode and correctness proof can be found in Appendix of A and B in [25].

1) **GET($k$, $l$):** A client $c$ sends a GET request, containing the key $k$, and two vectors $vc_r$ and $vc_w$ to a server which stores key $k$. Let $O$ be a $D$ dimensional vector with all entries equal to zero. When the client sends a GET request, it can use $O$ or hrvc as $vc_r$ to guarantee ec (eventual consistency) or mr respectively. Similarly, it can use $O$ or hwvc as $vc_w$ to guarantee ec or rvw respectively.

When the server $p^m_d$ receives the GET request, it first checks whether the dependencies specified by $vc_r$ and $vc_w$ have been applied locally, by comparing its $gsvc$ with $vc_r$ and $vc_w$.  

---

\(^4\)NTP: http://www.ntp.org
The server blocks if for some \( i \in D \), \( v_{c[i]} \) or \( v_{w[i]} \) is greater than \( gsv_{c}^{m}[i] \). When both \( v_{c} \) and \( v_{w} \) are no greater than \( gsv_{c}^{m} \), the server retrieves the latest stable version in the version chain of requested key \( k \), which has an update timestamp no greater than the server’s \( gsv_{c} \). Finally, the value \( v \) of the version, its update timestamp, and \( gsv_{c}^{m} \) are returned to the client. Upon receiving the reply, the client updates its \( hrvc \) and \( cvc \) accordingly.

2) PUT\((k,v,l):\) A client \( c \) sends a PUT request, containing the key \( k \), the value \( v \), and its dependency time denoted by \( dvc \) to the server which stores key \( k \). The client chooses different \( dvc \) to provide different session guarantees. Choosing \( O, cvc \) or \( cvc \) guarantees eventual consistency, \( mw \) or \( wfr \), respectively.

When the server \( p_{m} \) receives the PUT request, it first checks whether the \( d \)-th entry of dependency time \( dvc \) sent by client is smaller than local physical clock \( clock_{m} \). If not, the server will wait until the condition becomes true. Then, the server updates the \( d \)-th entry of \( dvc \) and \( pvc_{d} \) with \( clock_{m} \). The server creates a new version of the item identified by key \( k \) by assigning it a tuple consisting of the key \( k \), the value \( v \), and the update timestamp \( dvc \), and inserts it into the version chain of the item and the set of updates which are waited to be replicated. Finally, the server sends a reply with the newly created \( dvc \) to the client. Upon receiving the reply, the client updates its \( hhvc \) and \( cvc \) accordingly.

3) Replication: Inside a datacenter, each partition periodically replicates the updates in the set \( updates \) in \( vc[d] \) order to its replicas at the other datacenters. When there is no update to replicate, a heartbeat is sent, which contains its latest clock time. If a server receives a heartbeat from datacenter \( i \), it updates the \( i \)-th entry of its \( pvc \). If it receives a replication request, it additionally inserts the received new version into the local corresponding version chain. However, this update is not visible to clients until the server’s \( gsvc \) becomes larger than its update timestamp. The servers inside each datacenter exchange their \( pvc \) vectors in the background, and each server \( p_{m} \) computes its \( gsvc_{m} \) as the aggregate minimum of all known \( pvc \).

### IV. Experimental Results

We develop a prototype distributed key-value store called TCCSTORE providing TCC. In this section we evaluate its performance in terms of throughput and latency, and investigate the effect of the locality of traffic and workload characteristics. For comparison, we also implement causal consistency and eventual consistency in TCCSTORE.

#### A. Experimental Setup

TCCSTORE is implemented in C++. We use Google Protocol Buffers\(^5\) for communication. We conduct our experiments in two datacenters on Aliyun, located at Shenzhen in South China and HangZhou in East China, respectively. The average RTT between them is about 27ms. We run all replicas on machines running Ubuntu 16.04 with 2 vCPUs, 2.5 GHz Intel Xeon (Cascade Lake), 4 GB memory, and 40 GB storage.

We run different number of client threads to generate different load conditions. We use one machine per datacenter with the same specification as all replicas to run client threads. Each client thread randomly decides to read or write and randomly picks one partition to perform operations. Our default workload uses the 50:50 read:write ratio and runs operations on a platform deployed over 2 datacenters (3 partitions per datacenter). Operations access keys within a partition according to a uniform distribution.

#### B. Effect of Locality of Traffic

In the first set of experiments, we investigate the effect of the locality of traffic on the performance of TCCSTORE.

We consider six cases, \( ec/ec, cc/cc, ryw/mw, ryw/wfr, mr/mw \) and \( mr/wfr \). Each case represents the consistency levels that can be chosen by clients. For example, in the \( ec/ec \) (resp. \( cc/cc \)) case, both levels for read and write operations can only be eventual consistency (resp. causal consistency). First, we consider these cases where clients only access their local datacenters.

Fig. 4(a) shows the effect of write proportion on the operation latency. The results are for 36 client threads per data-center. As we expect, latency increases as the write proportion increases, since write operations are more expensive than read operations. Fig. 4(b) shows how throughput changes as we change the write proportion. In all diagrams with throughput, we report the number of total operations done by the client threads inside one datacenter. As shown in Fig. 4(b), the throughput drops as we increase the write proportion.

We observe eventual consistency gains better performance compared with other cases. However, the increased latency of providing any session guarantee compared with eventual consistency is negligible (less than 1.5ms). Causal consistency has the highest latency and the lowest throughput in all cases. It requires up to about 8% higher latency and has up to about 6% lower throughput than other session guarantees.

Next, we take into account the cases with 25% remote traffic. Fig. 5(a) and Fig. 5(b) show the corresponding results. In all cases, eventual consistency still has the best performance. Unlike the cases where clients only access their local datacenters, the difference between providing causal consistency and session guarantees is obvious here. Causal consistency requires up to about 10% higher latency and has up to about 13% lower throughput than other session guarantees.

#### C. Effect of Workload

In this section, we want to see how the performance changes for workloads with various number of client threads. Fig. 6(a) and Fig. 6(b) show the effect of the number of client threads on the operation latency and throughput, respectively. To generate different load conditions, we run 12, 24, 36, 48, and 60 client threads per datacenter. As we expect, both latency and throughput increase in all cases as we run more client threads.

\(^5\)Google Protocol Buffers. https://developers.google.com/protocol-buffers/
Again, we observe that eventual consistency has better performance, and causal consistency still has the highest latency and lowest throughput in all cases. It requires up to about 34% higher latency and has up to about 11% lower throughput than other session guarantees.

Next, we take into account the cases with 25% remote traffic. Fig. 7(a) and Fig. 7(b) show the corresponding results. Again, we observe that the difference between causal consistency and session guarantees is more obvious. Causal consistency requires up to about 40% higher latency and has up to about 23% lower throughput than other session guarantees.

V. RELATED WORK

Specification Framework. Burckhardt [2] provides the \((\text{vis,ar})\) specification framework for eventually consistent distributed data stores based on the visibility and arbitration relations. Bouajjani et al. [24] provides a framework for specifying multilevel consistency. However, this framework is restricted to only two consistency levels, and is specific to a concrete implementation of data stores. In this work, we propose an implementation-independent formalization of TCC based on the four classic session guarantees.

Protocols of Tunable Consistency. Terry et al. present a protocol for ensuring session guarantees in [21], which is adopted in Bayou [26]. Bermbach et al. [27] provide a middleware to support \(\text{ryw}\) and \(\text{mr}\) on top of eventually consistent systems. Roohipaf et al. [22] present a protocol for providing various session guarantees in distributed key-value stores. To avoid slowdown cascades, they consider per-key session guarantees. Instead, we consider cross-key session guarantees [21]. Hence, we do not compare it in performance with our work in Section IV.

Systems Providing Tunable Consistency. Several popular data stores provide tunable consistency [10]–[13]. Amazon DynamoDB [10] provides eventual consistency as the default and stronger consistency with \text{ConsistentRead}. Cassandra [11] offers a number of fine-grained read/write consistency levels such as \text{ANY}, \text{ONE}, \text{QUORUM}, \text{ALL}. MongoDB [12] exposes the \text{readConcern} and \text{writeConcern} parameters to clients, which can be set per operation. Azure Cosmos DB [13] offers five well-defined levels, namely strong, bounded staleness, session, consistent prefix, and eventual.

Causally Consistent Systems. There are plenty of research prototypes and industrial deployments of causally consistent distributed data stores [16], [17], [28]–[30]. COPS [16] explicitly tracks causal dependencies for updates. GentleRain [17] uses loosely synchronized physical clocks to reduce the overhead of metadata and communication. To reduce the delay of visibility of updates, Occult [28] and Contrarian [29] use HLCs [31], and POCC [30] relies on both physical clocks and dependency vectors.

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose tunable causal consistency (TCC). TCC allows clients to choose the desired session guarantee for each operation. We first propose a formal specification of TCC. Then we design a TCC protocol supporting multiple keys and develop a prototype distributed key-value store. To make TCC more practically useful, it would be beneficial to explore how
to automatically choose the appropriate session guarantee for each operation in applications.

ACKNOWLEDGMENT

This work was partially supported by the CCF-Tencent Open Fund (Tencent RAGR20200201).

REFERENCES

[1] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, “Replicated data types: Specification, verification, optimality,” in Proceedings of the 41st ACM Symposium on Principles of Programming Languages (POPL’2014), 2014, pp. 271–283.

[2] S. Burckhardt, “Principles of eventual consistency,” Found. Trends Program. Lang., vol. 1, no. 1-2, pp. 1–150, 2014.

[3] D. Terry, “Replicated data consistency explained through baseball,” Commun. ACM, vol. 56, no. 12, pp. 82–89, 2013.

[4] P. Niotti and M. Vukolić, “Consistency in non-transactional distributed storage systems,” ACM Comput. Surv., vol. 49, no. 1, pp. 19:1–19:34, 2016.

[5] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” ACM Trans. Program. Lang. Syst., vol. 12, no. 3, pp. 463–492, 1990.

[6] W. Vogels, “Eventually consistent,” Communications of the ACM, vol. 52, no. 1, pp. 40–44, 2009.

[7] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” SIGACT News, vol. 33, no. 2, pp. 51–59, 2002.

[8] E. A. Brewer, “Towards robust distributed systems (abstract),” in Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC’2000), 2000, pp. 7–7.

[9] D. Abadi, “Consistency trade-offs in modern distributed database system design: Cap is only part of the story,” IEEE Computer, vol. 45, no. 2, pp. 37–42, 2012.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Siruvasramaniam, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s highly available key-value store,” in Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP’2007), 2007, pp. 205–220.

[11] A. Lakshman and P. Malik, “Cassandra: A decentralized structured storage system,” SIGOPS Oper. Syst. Rev., vol. 44, no. 2, pp. 35–40, 2010.

[12] M. Tylenen, A. Swerlin, A. Kamksy, R. Tan, A. Cabral, and J. Murov, “Implementation of cluster-wide logical clock and causal consistency in mongodb,” in Proceedings of the 2019 International Conference on Management of Data (SIGMOD’2019), 2019, pp. 636–650.

[13] J. R. Guay Paz, “Introduction to Azure Cosmos DB,” in Microsoft Azure Cosmos DB Revealled. Springer, 2018, pp. 1–23.

[14] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, “Causal memory: Definitions, implementation, and programming,” Distributed Computing, vol. 9, no. 1, pp. 37–49, 1995.

[15] M. Perrin, A. Mostefaoui, and C. Jard, “Causal consistency: Beyond memory,” in Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’2016), 2016.

[16] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “Don’t settle for eventual: Scalable causal consistency for wide-area storage with cope,” in Proceedings of the 23rd ACM SIGPLAN Symposium on Operating Systems Principles (SOSP’2011), 2011, pp. 401–416.

[17] J. Yu, C. Jorgulescu, A. Roy, and W. Zwaenepoel, “Gentlerain: Cheap and scalable causal consistency with physical clocks,” in Proceedings of the 2014 ACM Symposium on Cloud Computing (SOCC’2014), 2014, pp. 1–13.

[18] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “Stronger semantics for low-latency geo-replicated storage,” in Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI’2013), 2013, pp. 313–328.

[19] D. D. Akkoorath, A. Z. Tomanc, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, “Cure: Strong semantics meets high availability and low latency,” in Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS’2016), 2016, pp. 405–414.