Parameterized and Runtime-tunable Snapshot Isolation in Distributed Transactional Key-value Stores

Hengfeng Wei, Yu Huang, Jian Lu
State Key Laboratory for Novel Software Technology, Nanjing University, China
{hfwei, yuhuang, lj}@nju.edu.cn

Abstract—Several relaxed variants of Snapshot Isolation (SI) have been proposed for improved performance in distributed transactional key-value stores. These relaxed variants, however, provide no specification or control of the severity of the anomalies with respect to SI. They have also been designed to be used statically throughout the whole system life cycle. To overcome these drawbacks, we propose the idea of parameterized and runtime-tunable snapshot isolation. We first define a new transactional consistency model called Relaxed Version Snapshot Isolation (RvSI), which can formally and quantitatively specify the anomalies it may produce with respect to SI. To this end, we decompose SI into three “view properties”, for each of which we introduce a parameter to quantify one of three kinds of possible anomalies: $k_1$-BV ($k_1$-version bounded backward view), $k_2$-FV ($k_2$-version bounded forward view), and $k_3$-SV ($k_3$-version bounded snapshot view). We then implement a prototype partitioned replicated distributed transactional key-value store called CHAMELEON across multiple data centers. While achieving RVSI, CHAMELEON allows each transaction to dynamically tune its consistency level at runtime. The experiments show that RVSI helps to reduce the transaction abort rates when applications are willing to tolerate certain anomalies. We also evaluate the individual impacts of $k_1$-BV, $k_2$-FV, and $k_3$-SV on reducing the transaction abort rates in various scenarios. We find that it depends on the issue delays between clients and replicas which of $k_1$ and $k_2$ plays a major role in reducing transaction abort rates.

Keywords—Transactionally key-value stores, relaxed version snapshot isolation, runtime-tunable consistency.

I. INTRODUCTION

Distributed key-value stores have been widely deployed underlying modern large-scale Internet services such as electronic commerce and social networking. Well-known distributed key-value stores include both open source projects like Apache Cassandra\(^1\) [1] and MongoDB\(^2\) and commercial products like Google’s BigTable [2], Yahoo!’s PNUTS [3], and Amazon’s Dynamo [4]. To achieve high performance, high availability, and high scalability, large data sets are typically partitioned into multiple data shards, and each data shard is then independently replicated over its own master storage node and possibly several slave nodes [5].

Though application developers are satisfied with the easy-to-use interfaces such as \texttt{put(K key, V val)} and \texttt{get(K key)} provided by distributed key-value stores, there are increasing interests in transactional semantics for operations over an arbitrary group of data items. Without transactions, an application must explicitly coordinate accesses to shared data to avoid anomalies such as race conditions, partial writes, and overwrites, which are known hard to deal with [6]. With transactions, however, these anomalies are treated as low-level implementations, hidden from the developers by means of formally defined transactional consistency models.

Snapshot Isolation (SI) [7], among all the previously proposed transactional consistency models, avoids many of the undesirable anomalies for applications. The key idea of snapshot isolation is to provide each transaction with the “latest” consistent snapshot of all data items while avoiding write conflicts among concurrent transactions. This decoupling of reads and writes is quite intuitive to developers and eases the burden on programming and reasoning.

A major obstacle to providing snapshot isolation in distributed key-value stores is that distributed transactions satisfying strong transactional semantics often require intensive coordinations among multiple storage nodes, hamper transaction throughput, and result in poor system performance. For improved performance, several relaxed transactional semantics based on snapshot isolation have been proposed [6], [8]–[10], as reviewed in Section VII.

We argue, however, that these relaxed variants of snapshot isolation suffer from two main drawbacks. First, they provide no specification or control of the severity of the anomalies that are originally forbidden by snapshot isolation but introduced due to relaxation. At the worst, a transaction would have been executed on extremely stale data, rendering itself effectively worthless. In addition, without any guarantees, it would be much harder for the developers to write programs and reason about them. Second, like most of other transactional consistency models, these relaxed variants of snapshot isolation have been designed to be used statically throughout the whole system life cycle. In other words, the consistency model needs to be determined at the system design phase and remains unchanged once the system is deployed. However, no single individual consistency model satisfies all users in all situations [11]. Therefore, it would be desirable to allow each individual transaction to dynamically choose or tune its consistency level at runtime; see related work in Section VII.

---

\(^1\)Apache Cassandra: http://cassandra.apache.org/.
\(^2\)MongoDB: https://www.mongodb.com/.
To further motivate the requirements of explicitly specifying and quantitatively bounding the inconsistency and dynamically tuning consistency at runtime, consider an online bookstore application. Suppose we have a “table” Books and each book has several attributes: title, authors, publisher, sales, inventory, ratings, reviews, and so on. Different users might execute different transactions on the same Books table with different consistency models for different purposes. A customer (user $U_1$) who wants to obtain the basic information about a particular book of interest, would be satisfied with a transaction $T_1$ that responds quickly, even at the risk of retrieving out-of-date reviews of the book. Imagine a bookstore clerk (user $U_2$) is running transaction $T_2$ to check the inventory of a book (i.e., the number of copies in stock; besides other attributes such as title and publisher) in order to decide whether the stock needs to be replenished. In this scenario, transaction $T_2$ may want to read the inventory value updated by concurrent transactions that commit after $T_2$ starts. A third user $U_3$, as a sales analyst, is studying the relationship between sales and ratings of a book by periodically running a read-only transaction $T_3$. In this scenario, it is acceptable for $T_3$ to obtain stale or concurrently updated versions of both sales and ratings as long as they were from a consistent snapshot, or from two separate snapshots that differ by a bounded version distance.

To overcome these two drawbacks discussed above, we propose the idea of parameterized and runtime-tunable snapshot isolation. The two key challenges we address in this paper are: 1) how to specify bounded inconsistency with respect to snapshot isolation; and 2) how to implement it so as to support dynamic consistency choices. Accordingly, we make two main contributions as follows.

First, we define a novel transactional consistency model called Relaxed Version Snapshot Isolation (RVSI), which provides a formal specification for quantifying the anomalies it allows with respect to snapshot isolation. Generally speaking, RVSI is weaker than snapshot isolation but stronger than Read Committed isolation (RC) \[7\]. As with RC, RVSI guarantees that each transaction is committed atomically and only committed transactions can be observed by others. As with SI, RVSI also avoids write conflicts: if multiple concurrent transactions write to the same data items, at most one of them will commit. This write-conflict freedom property prevents the lost updates anomaly \[7\]. On the other hand, RVSI, being weaker than SI, does not require each transaction to obtain the latest preceding snapshot before it starts. Therefore, RVSI will introduce anomalies which are originally forbidden by SI. To precisely capture the anomalies RVSI allows, we first decomposes SI into three “view properties” (besides the write-conflict freedom property mentioned above), for each of which we then introduce a parameter to quantify one of three kinds of possible anomalies: 1) $k_1$-BV ($k_1$-version bounded backward view): RVSI allows transactions to observe stale data, as long as the staleness is deterministically bounded by $k_1$; 2) $k_2$-FV ($k_2$-version bounded forward view): RVSI allows transactions to observe data that are updated by concurrent transactions, as long as the “forward” level is deterministically bounded by $k_2$; and 3) $k_3$-SV ($k_3$-version bounded snapshot view): RVSI allows a transaction to observe two data items coming from different snapshots, as long as the “distance” between these two snapshots is deterministically bounded by $k_3$.

Second, we implement a prototype distributed transactional key-value store called CHAMELEON which achieves RVSI and allows each transaction to dynamically tune its consistency level at runtime. CHAMELEON supports data partitioning and data replication across multiple data centers. In CHAMELEON, all replicas of a partition consist of a master node and several slave nodes. To guarantee the atomic commitment of distributed transactions across multiple masters (i.e., partitions), CHAMELEON uses a two-phase commit (2PC) protocol \[12\], into which the checking of RVSI specification is integrated. After a transaction commits on a master, it is propagated asynchronously to its slaves.

We deploy CHAMELEON on Alibaba Cloud (Aliyun)\(^3\) which consists of 9 servers spanning 3 data centers in North China, South China, and East China. The experiments show that RVSI helps to reduce the transaction abort rates when applications are willing to tolerate certain anomalies. We also conduct extensive controlled experiments to evaluate the individual impacts of $k_1$-BV, $k_2$-FV, and $k_3$-SV on reducing the transaction abort rates in various scenarios. We find that it depends on the issue delays between clients and replicas which of $k_1$ and $k_2$ plays a major role in reducing transaction abort rates. Long-lived transactions due to larger issue delays are more likely to obtain data versions that are updated by concurrent transactions and whether they will be aborted are thus more sensitive to the parameter $k_2$. As the issue delays get smaller, the impacts of $k_1$ for $k_1$-BV on reducing transaction abort rates emerge and become more significant. In addition, given that $k_1$-BV or $k_2$-FV is (slightly) relaxed, increasing the value of $k_3$ helps to further reduce the transaction abort rates.

This paper is organized as follows. Section II reviews snapshot isolation. Section III defines RVSI. Section IV describes the CHAMELEON prototype system. Section V presents the RVSI protocols for data replication and partitioning. Section VI evaluates the impacts of RVSI specification on transaction abort rates in various scenarios. Section VII discusses the related work. Section VIII concludes.

### II. Transactions and Snapshot Isolation

In this section we review the terminology about transactions and the definition of snapshot isolation following \[13\].

A. Transactions and Histories

A distributed transactional key-value store consists of a set of data items, each of which may have multiple versions. Clients interact with the store by issuing read or write operations on the data items, in the form of transactions. A transaction $T_i$ begins with a start operation $s_i$, then contains a sequence of read or write operations $o_j$, and ends with a terminating operation, either a commit $c_i$ or an abort $a_i$.

To keep the notation simple, we assume that each transaction performs at most one read and at most one write on any data item. If $c_i \in T_i$, then we say that $T_i$ is committed, and if $T_i$ writes $x$, the version $x_i$ is committed and becomes a committed version at the time $T_i$ commits. If $a_i \in T_i$, then $T_i$ is aborted, and $x_i$ does not become part of the committed state. We use $w_i(x_i)$ to denote transaction $T_i$ writing version $i$ of data item $x$, and $r_i(x_j)$ transaction $T_i$ reading version $j$ of data item $x$ written by transaction $T_j$.

Executions of a distributed transactional key-value store are described through histories. A history $h$ over a set of transactions is defined as an (irreflexive) partial order, called time-precedes order $\prec_h$, over operations (i.e., start, read, write, commit, or abort) of those transactions such that

1) For every operation $o_i$ in $h$, transaction $T_i$ terminates in $h$;
2) The partial order $\prec_h$ is consistent with the order in which operations within a transaction are executed. That is, for any two operations $o_i$ and $o_j$ of the same transaction $T_i$, if $o_i$ precedes $o_j$ in $T_i$, then $o_i \prec_h o_j$ in $h$. Especially, $s_i \prec_h c_i$;
3) For every read $r_i(x_j) \in h$, there exists a write $w_j(x_j) \in h$ such that $w_j(x_j) \prec_h r_i(x_j)$;
4) For any two committed transactions $T_i$ and $T_j$, either $c_i \prec_h s_j$ or $s_j \prec_h c_i$. Two transactions $T_i$ and $T_j$ are concurrent if $s_i \prec_h c_j$ and $s_j \prec_h c_i$; and
5) Any two write operations on the same data item are totally ordered by $\prec_h$. This is called the version order of the data item.

B. Snapshot Isolation

Intuitively, snapshot isolation requires that each transaction read data from the system snapshot as of the time the transaction started [7]. It can be formally defined in terms of two properties, Snapshot Read and Snapshot Write [13].

**Definition 1**: A history $h$ is in Snapshot Isolation (SI) if and only if it satisfies

- (Snapshot Read). All reads of transaction $T_i$ occur at $T_i$’s start time.

$$\forall r_i(x_j \neq i), w_j(x_k), c_k \in h : (c_j \in h \land c_j \prec_h s_i \land (s_i \prec_h c_k \lor c_k \prec_h c_j)).$$

- (Snapshot Write). No concurrent committed transactions may write the same data item.

$$\forall w_i(x_i), w_j(x_i) \in h \implies (c_i \prec_h s_j \lor c_j \prec_h s_i).$$

Due to the Snapshot Write property, the clause $c_k \prec_h c_j$ in the Snapshot Read property implies $c_k \prec_h s_j$ and $x_k$ precedes $x_j$ in version order. Thus, the Snapshot Read property requires that only the latest committed version of each data item can be read. Appendix A characterizes SI in terms of anomalies [13].

III. RELAXED VERSION SNAPSHOT ISOLATION

Relaxed Version Snapshot Isolation (RVSI) is a natural generalization of SI by relaxing the Snapshot Read property in three ways. RVSI allows a transaction to observe “stale” committed data versions as long as their staleness is bounded or “concurrent” committed data versions as long as the concurrency level is bounded. When a transaction reads more than one data item, say $x$ and $y$, RVSI also allows it to obtain versions of $x$ and $y$ from two different snapshots, as long as the “distance” between these two snapshots is bounded. We next formally define RVSI. For convenience, the definitions below specify the bounds $k_1$, $k_2$, and $k_3$ globally with respect to a history. However, they can be easily generalized to support dynamic bounds per transaction or even with respect to each individual read operation or every pair of them; see the code snippet in Fig. 4.

Suppose that $r_i(x_j)$ is in transaction $T_i$. We consider two categories of bounded version constraints according to whether the transaction $T_j$ precedes or runs concurrently with $T_i$. First, for each individual read operation $r_i(x_j)$ of transaction $T_i$, $x_j$ can be any one of the latest $k$ committed versions of $x$.

**Definition 2**: A history $h$ satisfies $k_1$-version bounded backward view ($k_1 \in \mathbb{N}^+$), denoted $h \in k_1$-BV, if each read of transaction $T_i$ obtains the data item of within the latest $k_1$ committed versions before $T_i$ starts. Formally,

$$\forall r_i(x_j), w_k(x_k), c_k \in h (k = 1, 2, \ldots, m; k \neq j) :$$

$$c_j \in h \land \bigwedge_{k=1}^{m} (c_j \prec_h c_k \prec_h s_i) \implies m < k_1.$$

Second, a transaction $T_i$ is allowed to read from concurrent update transactions.

**Definition 3**: A history $h$ satisfies $k_2$-version bounded forward view ($k_2 \in \mathbb{N}$), denoted $h \in k_2$-FV, if each read of transaction $T_i$ obtains the data item of within the earliest $k_2$ committed versions updated by transactions concurrent with $T_i$. Formally,

$$\forall r_i(x_j), w_k(x_k), c_k \in h (k = 1, 2, \ldots, m; k \neq j) :$$

$$c_j \in h \land \bigwedge_{k=1}^{m} (s_i \prec_h c_k \preceq_h c_j) \implies m \leq k_2.$$

On the other hand, suppose that $r_i(x_j)$ and $r_i(y_j)$ are two read operations in transaction $T_i$. We constrain the “distance” between the two snapshots created by transaction $T_j$ and transaction $T_i$, respectively.

**Definition 4**: A history $h$ satisfies $k_3$-version bounded snapshot view ($k_3 \in \mathbb{N}$), denoted $h \in k_3$-SV, if a transaction...
$T_i$ reads committed versions $x_j$ and $y_l$ ($x \neq y, j \neq l$), and assume, without loss of generality, that $c_j \prec_h c_l$, then in the time interval $(c_j, c_l)$ there are at most $k_3$ other committed versions of $x$ which are fresher than $x_j$. Formally,

$$\forall r_i(x_j), r_i(y_l), w_k(x_k), c_k \in h$$

$$(k = 1, 2, \ldots, m; k \neq j; j \neq l; x \neq y) :$

$$\left( \bigwedge_{k=1}^{m} (c_j \prec_h c_k \prec_h c_l) \right) \implies m \leq k_3.$$

As required by $k_1$-BV, $k_2$-FV, and $k_3$-SV, RVSI prevents transactions from reading non-committed data. Finally, RVSI satisfies the Snapshot Write property of SI, which is also called the Write-Conflict Freedom (WCF) property [10].

**Definition 5:** A history $h$ is in relaxed version snapshot isolation if and only if (where, $k_1 \in \mathbb{N}^+, k_2, k_3 \in \mathbb{N}$)

$h \in k_1$-BV $\cap k_2$-FV $\cap k_3$-SV $\cap$ WCF.

Fig. 1 illustrates the definition of RVSI. The properties of RVSI in terms of anomalies are discussed in Appendix B. In particular, RVSI(1, 0, +) is equivalent to SI.

**IV. System Design of Chameleon**

We have implemented a prototype partitioned replicated distributed transactional key-value store called Chameleon and deployed it in a multiple data center cloud infrastructure. While achieving RVSI, Chameleon allows each individual transaction to dynamically tune its consistency level at runtime. Fig. 2 illustrates the three main components of Chameleon: the transactional key-value store, the client library, and the RVSI protocol.

**A. The Transactional Key-value Store**

Chameleon uses the classic key-value data model. Each key is composed of a row key and a column key. Sites in Chameleon are divided into masters and slaves, and each master has several slaves. For scalability, the whole keyspace is first partitioned among the masters using, for example, consistent hashing [4]. For fault tolerance and availability, each master then replicates the keys assigned to it over its slave sites and several secondary copies on corresponding slave sites. Transactions are first executed and committed on the masters and modify the primary copies. Committed transactions are then asynchronously propagated to slave sites where the secondary copies are updated [9]. For low latency, clients can read from nearby slave sites.

**B. Client Library**

The client library provides not only the traditional transaction APIs for beginning/ending a transaction and read-
// Initialize keys (ck, ck1, and ck2) here
ITx tx = new RVSIITx(/** context **/);

// Read and write
ITsCell tsCell = tx.read(ck);
ITsCell tsCell1 = tx.read(ck1);
ITsCell tsCell2 = tx.read(ck2);

// Specify RVSI specs. (e.g., SVSpec)
RVSI Spec sv = new SVSpec();
sv.addSpec({ck, ck1, ck2}, 2);
tx.collectRVSISpec(sv);

boolean committed = tx.end();

Figure 4. Code snippet for writing RVSI transactions.

ing/writing data items, but also APIs for specifying RVSI constraints. Fig. 4 illustrates how to write transactions with RVSI specifications using these APIs. BVSpec (resp. FVSpec) for $k_1$-BV (resp. $k_2$-FV) allows to specify different $k_1$’s (resp. $k_2$’s) for different read operations in a transaction, and SVSpec for $k_3$-SV allows to specify $k_3$’s for pairs of read operations. For convenience, BVSpec (resp. FVSpec) collects a group of read operations with the same value of $k_1$ (resp. $k_2$). Similarly, the value of $k_3$ in SVSpec for a set of read operations applies to every pair of them.

In the context APIs, the client library provides several data partitioning mechanisms, e.g., range partitioning or consistent hashing [4].

C. RVSI Protocol

The RVSI protocol has two parts: RVSI-MS for data replication in the master-slave architecture (Section V-A) and RVSI-MP for data partitioning across multiple partitions (Section V-B). RVSI-MS enforces RVSI in the simplified single-master architecture. The key issue to address is how to calculate RVSI constraints based on the RVSI specification. RVSI-MP extends RVSI-MS to support atomic commitment of distributed transactions over multiple partitions.

V. THE RVSI PROTOCOL

We first propose the RVSI-MS protocol for enforcing RVSI in a Master-Slave data replication system, and then the RVSI-MP protocol for supporting Multiple Partitions.

A. The RVSI-MS Protocol for Replication

We describe the RVSI-MS protocol in terms of how its participants (i.e., clients, the master site, and slave sites) handle the events they are responsible for and how they interact with each other. The events consist of clients issuing a transaction (including BEGIN, READ, WRITE, and END), the master responding to BEGIN or END from clients by starting (START) or committing (COMMIT) a transaction respectively, and the master and slaves responding to READ from clients. In addition, the master will send messages to slaves, triggering their RECEIVED events.

The RVSI-MS protocol is shown in Algorithm 1 of Appendix C-A, where rpc-call is synchronous and blocking while broadcast is asynchronous and non-blocking. We now assume that in a transaction a write is not followed by any read on the same data item. The general case is discussed in Section V-A2.

1) The RVSI-MS Protocol: We first present the concurrency control scheme used in the RVSI-MS protocol, focusing on how a transaction executes at the master and the slaves. It leaves the stubs for enforcing the RVSI version constraints we discuss later in Section V-A2.

All transactions are started and committed/aborted on the master (Lines 2 and 9). The master employs an Multi-Version Concurrency Control (MVCC) protocol to locally implements SI and the first-committee-wins rule [13]. Each transaction $T$ is assigned a globally unique start-timestamp (denoted $T.sts$) and a commit-timestamp (denoted $T.cts$).

Each read operation can be issued to any site which holds the data requested. The read of a transaction $T$ issued to the master will obtain the latest data version before the timestamp $T.sts$, under the control of the local MVCC protocol on the master (Line 13). All write operations of a transaction $T$ are buffered in $Twrites$ at the client side until $T$ is about to commit (Line 6). At that moment, the client first calculates the version constraints, denoted $T.vc$, in terms of $k_1$, $k_2$, and $k_3$, based on the results of its read operations and the RVSI specification (Line 8). Then $T.vc$, along with $Twrites$, is issued to the master, which decides whether $T$ can be committed by checking the version constraints (Line 15) and the write-conflict freedom property.

To commit a transaction $T$, its updates $Twrites$ are installed on the master and will be lazily propagated to the slaves (Lines 18–22). The updates are then performed at each slave site (Line 27). Since the version constraints are checked at commit, the updates of different transactions can be performed neither necessarily atomically nor in their commit order.

2) Calculating Version Constraints: Before showing how to calculate the RVSI version constraints on the read operations of a transaction, namely, to implement the ADD-VC procedure at the client side (Line 8), we first introduce some notations involving data versions.

On the master site, each version of a data item $x$ is associated with a globally unique timestamp, denoted $x.ts$, which equals the commit-timestamp of the transaction that installs this data version. Thus, timestamps induce a total order on all the versions of each data item on the master. We denote the position of some version of a data item $x$ in the total order by $x.ord$ (ord stands for “ordinal”). A version of data item $x$, $x.ver$, is denoted by a triple $(x.ts, x.ord, x.val)$, where $x.val$ denotes its value. For notational convenience,
we define, for each data item \(x\), a mapping \(O_x\) which takes as input a timestamp \(t\) and returns the ordinal number of the latest version of \(x\) committed before or at \(t\). That is,
\[
O_x(t) = \max\{x.\text{ord} \mid x.\text{ts} \leq t\}.
\]

With these mappings maintained on the master site, we calculate the version constraints for \(k_1\)-BV, \(k_2\)-FV, and \(k_3\)-SV separately. We assume that transaction \(T_i\) is about to commit and calls the procedure ADD-VC (Line 8).

1) Version constraint for \(k_1\)-BV. Suppose that \(r_1(x_j) \in T_i\) (namely, the read operation \(r_1(x)\) of transaction \(T_i\) returns the version \(x_j\) installed by transaction \(T_j\)). Add the constraint \(O_x(T_i.\text{sts}) - O_x(T_j.\text{cts}) < k_1\) to transaction \(T_i\)’s version constraint \(T_i.\text{vc}\). Since \(x_j\) is associated with the commit timestamp of \(T_j\), we have \(O_x(T_j.\text{cts}) \equiv \text{ord}(x_j)\).

2) Version constraint for \(k_2\)-FV. Suppose that \(r_1(x_j) \in T_i\). Add the constraint \(O_x(T_i.\text{cts}) - O_x(T_j.\text{cts}) \leq k_2\) to transaction \(T_i\)’s version constraint \(T_i.\text{vc}\).

3) Version constraint for \(k_3\)-SV. Suppose that \(r_1(x_j), r_1(y_l) \in T_i\). Without loss of generality, we assume that \(T_j.\text{cts} < T_i.\text{cts}\). Add the constraint \(O_x(T_i.\text{cts}) - O_x(T_j.\text{cts}) \leq k_3\) to transaction \(T_i\)’s version constraint \(T_i.\text{vc}\).

As mentioned before, we have assumed in RVSI-MS, that in a transaction a write is not followed by any read on the same data item. If it is not the case, the “read-your-writes” rule is used: the read operation after some write operation in the same transaction on the same data item reads from the write buffer. In this way, both the \(k_1\)-BV and \(k_2\)-FV version constraints are trivially satisfied. Suppose that \(r_1(y_l)\) reads from \(w_1(y_l)\) of the same transaction \(T_i\) and that \(r_1(x_j) \in T_i\). Given that \(T_j.\text{cts} < T_i.\text{sts}\), the version constraint for the \(k_3\)-SV specification involving \(x_j\) and \(y_l\) is
\[
O_x(T_i.\text{cts}) - O_x(T_j.\text{cts}) \leq k_3.
\]

In other words, \(k_3\)-SV reduces to \(k_1\)-BV. Similarly, it reduces to \(k_2\)-FV if \(T_j.\text{cts} > T_i.\text{sts}\).

b. The RVSI-MP Protocol for Partition

Distributed transactions spanning multiple masters/partitions need to be committed atomically, namely, all participating masters should agree on whether to commit or abort a transaction. To this end, we adopt the classic two-phase commit (2PC) protocol [12]. The main task of the RVSI-MP protocol (Algorithm 2 of Appendix C-B) for committing distributed transactions with RVSI specification is to integrate the calculation and checking of RVSI version constraints into the 2PC protocol.

Since distributed transactions span multiple master sites, the way of obtaining unique timestamps from the single master in RVSI-MS does not work any more. Instead, RVSI-MP assumes a timestamp oracle which generates globally unique timestamps [15] (Line 6). Clients make a request of the timestamp oracle for the start-timestamp of a transaction when the transaction begins (Line 2).

The participants involved in committing a transaction in RVSI-MP consist of the client, the timestamp oracle \(T\), a coordinator \(C\) for the 2PC protocol, and the master sites the transaction spans. When issuing a transaction \(T\) via END (Line 3), the client first calculates the RVSI version constraints \(T.\text{vc}\) as described in Section V-A2 (Line 4), and then asks a coordinator to commit it (Line 5).

The coordinator responsible for performing the 2PC protocol among multiple masters first splits the RVSI version constraints \(T.\text{vc}\) and the updates \(T.\text{writes}\) into groups, one per master (Line 9) according to the data partitioning strategy. Although the \(k_3\)-SV specification involves two data items (e.g., \(x_j\) and \(y_l\) in Definition 4), the version constraint for it only refers to one of them (depending on whether \(T_j.\text{cts}\) is larger than \(T_i.\text{cts}\); see Section V-A2). Thus, the split step for RVSI constraints applies not only to \(k_1\)-BV and \(k_2\)-FV, but also to \(k_3\)-SV that involves two data items. Then, the coordinator brings the masters into the prepare phase of the 2PC protocol by invoking \textsc{Prepare} (Line 11). In the prepare phase, each master independently checks all the version constraints assigned to it and the write-conflict freedom property (Line 24). If no violations occur on either master, the distributed transaction is ready to commit (Line 13). At the beginning of the commit phase, the coordinator first obtains a globally unique timestamp \(T.\text{cts}\) from the timestamp oracle as the commit-timestamp of this transaction (Line 14) and then asks each master to commit. As in the RVSI-MS protocol, the masters apply updates locally and propagate them to slaves.

We discuss two additional issues (i.e., atomicity of the commit-timestamps and entity group) about the RVSI-MP protocol in Appendix C-C.

VI. Experimental Evaluation

In this section we evaluate the impacts of RVSI specification on the transaction abort rates in various scenarios. Generally, the experimental results demonstrate that RVSI helps to reduce the transaction abort rates when applications are willing to tolerate certain anomalies. More specifically, we first deploy the Chameleon prototype system on Aliyun and observe that most transactions have been aborted because of violating \(k_2\)-FV. That is, on Aliyun the parameter \(k_2\) plays a significant role in reducing transaction abort rates.

To fully understand when the parameter \(k_1\) for \(k_1\)-BV take effect, we explore more scenarios in controlled experiments by identifying and adjusting three kinds of delays among clients, masters, and slaves. We find that it depends on the issue delays between clients and replicas which of \(k_1\) and \(k_2\) plays a major role in reducing transaction abort rates. In addition, given that \(k_1\)-BV and \(k_2\)-FV is (slightly) relaxed,
increasing the value of $k_3$ helps to further reduce the transaction abort rates. Section VI-A presents the experiments on Aliyun and Section VI-B presents the controlled ones on local hosts.

A. Experiments on Aliyun

We deploy the CHAMELEON prototype system on Aliyun with 3 data centers located in East China (ec), North China (nc), and South China (sc), respectively. Each data center comprises 3 nodes labeled, for example, ec1, ec2, and ec3. All nodes are with the same configuration (which is sufficient for experiments): a single CPU, 2048MB main memory, and 2Mbps network. Nodes ec1, nc2, and sc3 are designated as masters, with nc1 and sc1 being slaves of ec1, ec2 and sc2 of nc2, and ec3 and nc3 of sc3. In this way, replicas are stored across data centers. Masters can serve as coordinators of the 2PC protocols. The timestamp oracle service is on ec1. The clients which issue transactions are on the hosts in our lab (located in East China).

According to the ping logs for a week among these 9 nodes\textsuperscript{4}, the 95th percentile of (one-way) communication delays among nodes within the same data center is about 1 ~ 2ms, while that among nodes across data centers is about 15 ~ 25ms. Additional ping experiments show that the communication delays among clients and replicas range from 15ms to 20ms.

1) Workload Parameters and Metrics: In the experiments on Aliyun, we explore the following three categories of workload parameters listed in Table I \textsuperscript{5}.

Transaction-related parameters. The parameter #keys denotes the number of keys stored in CHAMELEON. In these experiments, we intentionally assume a small number of keys such that concurrent transactions are more likely to access the same data item. The parameter #clients denotes the number of clients, each of which issues #txs/client transactions. The number of operations in each transaction is characterized by a Binomial random variable #ops/tx. The read/write ratio of a transaction is denote by rwRatio. We explore three cases of rwRatio $= 1 : 2$, rwRatio $= 1 : 1$, and rwRatio $= 4 : 1$, representing write-frequent, read-write balanced, and read-frequent workloads, respectively. The data item accessed by each operation is determined by a Zipfian distribution with parameters of #keys and ziptExponent, which is often used to generate workloads with “hotspot” data items \textsuperscript{16}.

Execution-related parameters. We use three parameters — minInterval, maxInterval, and meanInterval — to control the time interval between the finishing of a transaction and the starting of the next one. The actual time interval is computed as follows: Take a sample of an exponential random variable

with parameter meanInterval, add it to minInterval, and take the minimum of the result and maxInterval.

RVSI-related parameters. We explore 6 RVSIs specifications as listed in Table I. Recall that RVSIs $(1, 0, 0)$ is equivalent to SI. To evaluate the impacts of RVSIs on the transaction abort rates, we adjust parameters #clients, rwRatio, and $(k_1, k_2, k_3)$. We report small values of $(k_1, k_2, k_3)$, because higher values show little effect on the transaction abort rates.

2) Experimental Results: Transactions abort for two reasons: One is “vc-aborted”, namely, a transaction has obtained data items that do not satisfy the RVSIs version constraints. The other is “wcf-aborted”, namely, the WCF property has been violated. The experimental results show that transaction abort rates due to “vc-aborted” are sensitive to different values of $k_1$, $k_2$, or $k_3$, but those due to “wcf-aborted” are not. This is because RVSIs relaxes the Snapshot Read property of SI but not the Snapshot Write property. Generally, as rwRatio increases, the transaction abort rates due to “vc-aborted” grow accordingly. Fig. 5 shows the transaction abort rates due to “vc-aborted”, denoted $vc(*, *, *)$, under common read-frequent workloads\textsuperscript{6}. It shows that the transaction abort rates due to “vc-aborted” can be greatly reduced by slightly increasing the values of $k_1$, $k_2$, or $k_3$. For example, #clients $= 30$, $vc(1, 0, 0)$ is 0.1994, while $vc(2, 1, 1)$ decreases it to 0.0091. Fig. 6 explores the details of the transaction abort rates due to “vc-aborted” in Fig. 5 by decomposing each $vc(*, *, *)$ into $bv(*, *, *)$ for “$k_3$-BV-aborted”, $fv(*, *, *)$ for “$k_2$-FV-aborted”, and $sv(*, *, *)$ for “$k_3$-SV-aborted” \textsuperscript{7}. It turns out that most “vc-aborted” transactions in the Aliyun scenario have been aborted because of violating $k_3$-FV (compared to those violating $k_3$-BV), and correspondingly that such abort rates can be greatly reduced by slightly increasing the value of $k_2$. For example, with #clients $= 30$, $fv(1, 0, 0)$ is

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5}
\caption{The transaction abort rates due to “vc-aborted” under read-frequent workloads.}
\end{figure}

\textsuperscript{4}See https://github.com/hengxin/aliyun-ping-traces for raw data.
\textsuperscript{5}Since the TPC-C benchmark is commonly used to benchmark relational databases and the YCSB benchmark \textsuperscript{16} for distributed key-value stores does not support transactions, we design our own workloads.
\textsuperscript{6}See https://github.com/hengxin/chameleon-transactional-kvstore for experimental results under read-write-balanced and write-frequent workloads.
\textsuperscript{7}To reduce clutter, Fig. 6 shows only the data for 4 RVSIs specifications.
0.1889 and \( f_v(2, 0, 0) \) is 0.1866, while \( f_v(1, 1, 0) \) is 0.0064. Moreover, given \( k_2 \geq 1 \), increasing the value of \( k_3 \) for \( k_3\)-SV helps to further reduce the transaction abort rates. For example, \( sv(1, 1, 0) \) is 0.0480 while \( sv(1, 1, 1) \) is 0.0018.

### B. Controlled Experiments on Local Hosts

To fully understand when the parameter \( k_1 \) for \( k_1\)-BV takes effect, we explore more scenarios in controlled experiments on local hosts. Table I lists the controlled parameters denoting three kinds of (one-way) delays: the issue delays (issueDelay) between clients and replicas (i.e., masters and slaves), the replication delays (replDelay) between masters and its slaves, and the 2PC coordination delays (2pcDelay) among masters. Fig. 7 (with \#clients = 30, \#txs/client = 800) shows that when the issueDelay gets shorter, the impacts of \( k_2\)-FV on transactions abort rates go weaker, and on the contrary the impacts of \( k_1\)-BV have begun to emerge. For example, with issueDelay = 20ms, \( bv(1, 0, 0) \) is 0.0057 while \( f_v(1, 0, 0) \) is 0.0251. With issueDelay = 15ms, \( bv(1, 0, 0) \) is 0.08225, larger than \( f_v(1, 0, 0) \) = 0.0393. When issueDelay decreases to 5ms, \( bv(1, 0, 0) \) increases to 0.01716, significantly larger than \( f_v(1, 0, 0) \) = 0.0045. As illustrated in Fig. 7, the turning point of the issue delays is about 15 \( \sim \) 20ms. Moreover, given \( k_1 \geq 2 \), increasing the value of \( k_3 \) for \( k_3\)-SV helps to further reduce the transaction abort rates. For example, with issueDelay = 5ms, \( sv(2, 0, 0) \) is 0.0950 while \( sv(2, 0, 1) \) is 0.0164.

Combining these experimental results on local hosts and

---

**Table I**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>#keys</td>
<td>Fixed</td>
<td>25 = 5 (rows) ( \times ) 5 (columns)</td>
<td>size of keyspace</td>
</tr>
<tr>
<td>#clients</td>
<td>Variable</td>
<td>5, 10, 15, 20, 25, 30</td>
<td>number of clients</td>
</tr>
<tr>
<td>#txs/client</td>
<td>Fixed</td>
<td>1000</td>
<td>number of transactions per client</td>
</tr>
<tr>
<td>#ops/tx</td>
<td>Random</td>
<td>( \sim ) Binomial(20, 0.5)</td>
<td>number of operations per transaction</td>
</tr>
<tr>
<td>rwRatio</td>
<td>Variable</td>
<td>1:2, 1:1, 4:1</td>
<td>#reads/#writes</td>
</tr>
<tr>
<td>zipExponent</td>
<td>Fixed</td>
<td>1</td>
<td>parameter for Zipfian distribution</td>
</tr>
<tr>
<td>minInterval</td>
<td>Fixed</td>
<td>0ms</td>
<td>minimum inter-transaction time</td>
</tr>
<tr>
<td>maxInterval</td>
<td>Fixed</td>
<td>10ms</td>
<td>maximum inter-transaction time</td>
</tr>
<tr>
<td>meanInterval</td>
<td>Fixed</td>
<td>5ms</td>
<td>mean inter-transaction time</td>
</tr>
<tr>
<td>(( k_1, k_2, k_3 ))</td>
<td>Variable</td>
<td>(1,0,0) (1,1,0) (1,1) (2,0,0) (2,0,1) (2,1,1)</td>
<td>for ( k_1)-BV, ( k_2)-FV, and ( k_3)-SV</td>
</tr>
</tbody>
</table>

---

Figure 6. The transaction abort rates because of violating \( k_1\)-BV, \( k_2\)-FV, or \( k_3\)-SV under read-frequent workloads.

Figure 7. The transaction abort rates because of violating \( k_1\)-BV, \( k_2\)-FV, or \( k_3\)-SV under write-frequent workloads.
those on Aliyun, we can conclude that it depends on the issue delays between clients and replicas which of \( k_1 \) and \( k_2 \) plays a major role in reducing transaction abort rates. The rationale behind is as follows: In the Aliyun scenario, the issue delays between clients out of Aliyun and replicas in Aliyun are relatively longer than those manually set in our controlled experiments on local hosts. With larger issue delays, a transaction lasts longer and more transactions happen to be concurrent with it. These long-lived transactions are more likely to obtain data versions that are updated by concurrent transactions and whether they will be aborted are thus more sensitive to the parameter \( k_2 \). Moreover, the larger the \( rwRatio \) is, the more significant the impact of \( k_2 \)-FV on reducing transaction abort rates is. Note that since neither 2pcDelay (i.e., the delays among masters) nor replDelay (i.e., the delays between masters and slaves) is counted in the read procedure, they have shown no direct relationship with the effects of \( k_1 \) or \( k_2 \) on transaction abort rates. On the other hand, as the issue delays get shorter, the impacts of \( k_1 \) for \( k_1 \)-BV on reducing transaction abort rates emerge and become more significant. In addition, given that \( k_1 \)-BV or \( k_2 \)-FV is (slightly) relaxed, increasing the value of \( k_3 \) helps to further reduce the transaction abort rates.

VII. RELATED WORK

We divide the related work into three categories.

**Relaxed variants of snapshot isolation.** For improved performance, several relaxed transactional semantics based on snapshot isolation have been proposed. Forward Consistent View (PL-FCV) [13] allows a transaction \( T_i \) to observe the updates of transactions that commit after it started, i.e., read “forward” beyond the start point as long as \( T_i \) observes a consistent snapshot. Generalized Snapshot Isolation (GSI) [8] allows the use of “older” snapshots rather than the “latest” snapshot only. Strong Session Snapshot Isolation [9] enforces the real-time ordering constraints on transactions that belong to the same session but not across sessions. Parallel Snapshot Isolation (PSI) [6], as well as Causal Snapshot Isolation (CSI) [17], requires that hosts within a site observe transactions according to a consistent snapshot and a common ordering of transactions, but enforces only causal ordering of transactions across sites. Non-Monotonic Snapshot Isolation (NMSI) [10], [18] allows transactions to observe snapshots that are not monotonically ordered.

Unlike RVSI, the relaxed variants mentioned above provide no specification or control of the severity of the anomalies with respect to snapshot isolation. The basic idea of decomposing SI into three “view properties” is inspired by the work on NMSI. For example, RVSI supports Forward Freshness as NMSI does without, however, enforcing causality. The \( k_2 \)-FV property of RVSI resembles PL-FCV, except that \( k_2 \)-FV by itself does not require a consistent snapshot.

**Bounded transactional inconsistency.** Being an extension of serializability (SR), Epsilon-Serializability (ESR) [19], [20] allows read-only epsilon-transactions to observe database states with bounded inconsistency introduced by concurrent update transactions. In an \( N \)-ignorant system [21], a transaction may be ignorant of the results of at most \( N \) prior transactions that it would have seen if the execution had been serial. Relaxed Currency and Consistency (C&C) semantics [22] allows a query to specify a currency bound referring to how up-to-date the query result should be and a consistency class grouping the data items that should be from the same snapshot. Relaxed Currency (RC-) Serializability [14] allows update transactions to read stale data satisfying their freshness constraints.

ESR allows read-only epsilon-transactions to observe uncommitted data updated by concurrent transactions. In contrast, RVSI enforces Read Committed (RC), meaning that only committed data is observable. The \( N \)-ignorant and RC-serializability semantics extend serializability while RVSI extends snapshot isolation. The RC-serializability and C&C semantics measure the bounded inconsistency by currency in real-time while RVSI does by data versions.

**Dynamic consistency choices.** Due to the parameters introduced for bounding transactional inconsistency, the systems implementing the SR, \( N \)-ignorant, RC-serializability, or C&C semantics naturally support dynamic consistency choices at runtime. Pileus [23] is a transactional key-value store with varying degrees of consistency dynamically chosen by applications. Specifically, each transaction reads from a snapshot determined by its requested consistency, covering strong, intermediate, and eventual consistency. Li et al. [24] presents SIEVE, a tool that allows applications to automatically choose consistency levels (based on the theory of RedBlue consistency [25]) in replicated systems. Xie et al. [26] presents Salt, allowing developers to combine strong (ACID) and weak (BASE) transactions within a single application. Tripathi et al. [27] propose a transaction model which supports four consistency levels including, from the strongest to the weakest, SR, CSI, CSI with commutative updates, and CSI with asynchronous concurrent updates.

Our prototype transactional key-value store allows each individual transaction to dynamically tune its consistency level at runtime by choosing proper parameter values for \( k_1 \)-BV, \( k_2 \)-FV, and \( k_3 \)-SV specification.

VIII. CONCLUSION

In this paper we propose the idea of parameterized and runtime-tunable snapshot isolation. We first define a new transactional consistency model called Relaxed Version Snapshot Isolation (RVSI), which can formally and quantitatively specify the anomalies it may produce with respect to SI. We then implement a prototype partitioned replicated distributed transactional key-value store called CHAMELEON across multiple data centers. While achieving RVSI, CHAMELEON allows each individual transaction to dynamically tune its consistency level at runtime.
Since $k_S$-SV of RVSI involves multiple data items, evaluating its impacts on transaction abort rates is more challenging. We plan to study it in future work, probably with data mining technologies. We also plan to evaluate the performance of CHAMELEON on realistic workloads and compare it to other systems. Furthermore, it is crucial to study the impacts or anomalies of RVSI from the perspective of developers.

ACKNOWLEDGMENTS

This work is supported by the National 973 Program of China (2015CB352202) and the Fundamental Research Funds for the Central Universities (020214380031).

REFERENCES


APPENDIX A.
CHARACTERIZATION OF SI IN TERMS OF ANOMALIES

Following the terminology and notations in [13], SI identifies exactly the set of histories which avoid the anomalies of G-I and G-SI (consisting of G-SIa and G-SIb).

G-I is defined through three properties of histories that are simple to check (and we omit the details here): 1) GIa: Aborted Reads; 2) GIb: Intermediate Reads; and 3) GIc: Circular Information Flow.

G-SI is defined through SSG, the “Start-ordered Serialization Graph” [13]. An SSG of a history h, denoted SSG(h), is a directed graph with committed transactions in h as nodes and several types of dependencies as directed edges. Given two committed transactions T_i and T_j in h, we consider four possible types of dependencies between them:
1) T_j start-depends (s-depends) on T_i, denoted T_i \rightarrow T_j, if T_j starts after T_i commits, i.e., c_i \prec_h c_j;
2) T_j directly read-depends (wr-depends) on T_i, denoted T_i \rightarrowwr T_j, if T_j reads the data version of, say x, written by T_i, i.e., r_j(x) \in h;
3) T_j directly write-depends (ww-depends) on T_i, denoted T_i \rightarrowww T_j, if they both write the same data item x, and x_i and x_j are consecutive versions of x in h’s version order.
4) T_j directly anti-depends (rw-depends) on T_i, denoted T_i \rightarrowrw T_j, if T_i reads version x_k written by a third committed transaction T_k, and T_j creates x’s next version x_j (after x_k) in h’s version order.

The G-SIa and G-SIb anomalies are defined as follows:
• G-SIa: Interference. A history h exhibits the G-SIa anomaly if SSG(h) contains a wr/ww-dependency edge from T_i to T_j, but without an s-dependency edge from T_i to T_j.
• G-SIb: Missed Effects. A history h exhibits the G-SIb anomaly if SSG(h) contains a directed cycle with exactly one rw-dependency edge.

APPENDIX B.
ANOMALIES ALLOWED BY RVSI

RVSI is parameterized with three variables k_1, k_2, and k_3, controlling what types of anomalies RVSI allows and how severely it deviates from SI. In this section we examine some typical cases of RVSI, denoted RVSI(k_1, k_2, k_3). (Irrelevant variables in some cases are denoted by *’s.) First of all, no additional anomalies with respect to SI are introduced in the case of RVSI(1, 0, *).

Theorem 1: RVSI(1, 0, *) is equivalent to SI.
Proof: The proof of SI \subseteq RVSI(1, 0, *) is easy and thus omitted. In the following, we prove that RVSI(1, 0, *) \subseteq SI. Consider some history h \in RVSI(1, 0, *). We shall prove that h \notin SI by showing that h does not exhibit the anomalies of GI or G-SI. Informally, GI requires that a transaction T_i can commit only if T_i has only read the updates of transactions that have committed by the time T_i commits [13], which is the case in RVSI. The formal proof is omitted here.

h does not exhibit G-SIa. Suppose that G-SIa is allowed and there is a wr/ww-dependency from T_i to T_j in SSG(h) without a corresponding s-dependency (i.e., (T_i \rightarrow T_j) \notin SSG(h)). If (T_i \rightarrow T_j) \notin SSG(h), then h \notin WCF. If (T_i \rightarrow T_j) \in SSG(h), then s_i \prec_h c_j. Therefore, we have k_2 \geq 1, contradicting k_2 = 0.

h does not exhibit G-SIb. Suppose that G-SIb is allowed and SSG(h) contains a cycle of the form \langle T_1, T_2, \ldots, T_n, T_i \rangle with exactly one rw-dependency edge. Without loss of generality, suppose that the only rw-dependency edge is from T_1 to T_2, that T_1 \rightarrow T_2 is due to the data item x, and that r_1(x) \in T_1, w_2(x) \in T_2. There exists another transaction T_k which writes x and x_k is the immediately previous version of x before x_2. Therefore, we have c_k \prec_h s_k \prec_h c_2 due to T_k \rightarrow T_2. We also have c_2 \prec_h s_3 \prec_h c_3 \prec_h \cdots \prec_h s_n \prec_h c_n \prec_h s_1 due to the other dependency edges. Putting it together, we obtain k \times c_k \prec_h c_2 \prec_h s_1 in h and conclude that k_1 \geq 2, contradicting k_1 = 1.

Figure 8. A history in RVSI(2, 0, \infty) which exhibits the “write skew” anomaly. To satisfy SI, r_4(x) = 0 in transaction T_4 should be r_4(x) = 1.

As with SI, RVSI(1, 1, *) allows the write skew anomaly where concurrent transactions make updates to different data items causing the state to fork [6], [7]. For example, in Fig. 8, concurrent transactions T_1 and T_3 both read from the same data items x and y, but then update different ones. With SI, the state merges after both transactions commit. Therefore, the transaction T_4 in Fig. 8 should observe the updated x and y. RVSI(2, 0, \infty), however, allows T_4 to observe the stale x by looking “backward”. Similarly, RVSI(1, 0, \infty) allows a transaction to observe the updates of transactions that commit after it started by looking “forward” beyond its start time. Lastly, RVSI(1, 0, 0, >) allows a transaction to obtain versions of different data items from different snapshots.

APPENDIX C.
THE RVSI PROTOCOL

A. The RVSI-MS Protocol for Replication
Algorithm 1 shows the RVSI-MS protocol for replication.

B. The RVSI-MP Protocol for Partition
Algorithm 2 shows the RVSI-MP protocol for partition.
C. Additional Issues about the RVSI-MP Protocol

Atomicity of the commit-timestamps. The definition of SI explicitly refers to global timestamps [7]. Particularly, for any two transactions, SI requires that \( c_i \preceq_h s_j \) or \( s_j \preceq_h c_i \). In implementation, we rely on additional concurrency control mechanism to ensure the “atomicity of the commit-timestamps” of transactions: No transactions can obtain the start-timestamps from the time some transaction has just obtained its commit-timestamp to the time its updates have been actually applied; as illustrated in Fig. 9. This can be achieved by synchronizing the timestamp oracle and the coordinator for the 2PC protocol (omitted in Algorithm 2).

Entity group. Recall that a typical \( k_3 \)-SV specification involves two transactions (e.g., \( T_j \) and \( T_l \)) and two data items (e.g., \( w_j(x_j) \in T_j \) and \( w_l(y_l) \in T_l \)), and its corresponding version constraint refers to \( \mathcal{O}_x(T\_cts) \). This implies that the timestamps associated with \( T_j \) and \( T_l \) should be comparable, although they update different data items. Nevertheless, if we know a priori that some data items will not appear in the same \( k_3 \)-SV specification, then we can split all the data items into several independent entity groups. Each entity group performs a separate instance of the RVSI-MP protocol, using its own timestamp oracle.
Algorithm 1 RVSI-MS: RVSI Protocol for Replication (for Executing Transaction $T$).

Client-side methods:
1: procedure BEGIN()
2: $T\text{.sts} \leftarrow \text{rpc-call} \text{START()}$ at master $M$
3: procedure READ($x$)
4: $x\text{.ver} \leftarrow \text{rpc-call} \text{READ}($$x$$)$ at any site
5: procedure WRITE($x$, $v$)
6: add $(x, v)$ to $T\text{.writes}$
7: procedure END($T$)
8: $T\text{.vc} \leftarrow \text{ADD-VC}()$
9: $c/a \leftarrow \text{rpc-call} \text{COMMIT}(T\text{.writes}, T\text{.vc})$ at $M$

Master-side data structures and methods:
$M\.ts$: for start-timestamps and commit-timestamps
\{ $x\text{.ver} = (x\.ts, x\.ord, x\.val)$ \}: set of versions of $x$
10: procedure START()
11: return $++M\.ts$
12: procedure READ($x$)
13: return the latest $x\.ver$ installed
14: procedure COMMIT($T\text{.writes}, T\text{.vc}$)
15: if CHECK-VC($T\text{.vc}$) \&\& write-conflict freedom then
16: $T\text{.cts} \leftarrow ++M\.ts$
17: $\triangleright$ apply $T\text{.writes}$ locally and propagate it
18: $T\text{.upvers} = \emptyset$ \quad \triangleright$collect updated versions
19: for $(x, v) \in T\text{.writes}$ do
20: $x\text{.new-ver} \leftarrow (T\.cts, ++x\.ord, v)$
21: add $x\text{.new-ver}$ to \{ $x\text{.ver}$ \} and $T\text{.upvers}$
22: broadcast \langle prop, $T\text{.upvers}$ \rangle to slaves
23: return $c$ denoting “committed”
24: return $a$ denoting “aborted”

Slave-side data structures and methods:
$x\text{.ver} = (x\.ts, x\.ord, x\.val)$: the latest version of $x$
25: procedure READ($x$)
26: return $x\text{.ver}$
27: upon RECEIVED(\langle prop, $T\text{.upvers}$ \rangle)
28: for $(x\text{.ver}' = (x\.ts', x\.ord', x\.val')) \in T\text{.upvers}$ do
29: if $x\.ord' > x\.ord$ then
30: $x\text{.ver} \leftarrow x\text{.ver}'$


Client-side methods:
1: procedure BEGIN()
2: return rpc-call \text{GETTS()} at $T$
3: procedure END()
4: $T\text{.vc} \leftarrow \text{ADD-VC}()$
5: $c/a \leftarrow \text{rpc-call} \text{C-COMMIT}(T\text{.writes}, T\text{.vc})$ at $C$

Timestamp oracle methods:
$T\.ts$: for start-timestamps and commit-timestamps
6: procedure GETTS()
7: return $++T\.ts$

Coordinator-side data structures and methods:
The coordinator $C$ executes the 2PC protocol with masters $M$ involved in $T$.
8: procedure $C\text{-COMMIT}(T\text{.writes}, T\text{.vc})$
9: split $T\text{.writes}$ and $T\text{.vc}$ with the data partitioning strategy
10: \triangleright the prepare phase:
11: \quad \text{rpc-call} \text{PREPARE}(T\text{.writes}, T\text{.vc})$ at each $M$
12: \triangleright the commit phase:
13: if all PREPARE($T\text{.writes}, T\text{.vc}$) return true then
14: $T\text{.cts} \leftarrow \text{rpc-call} \text{GETTS()}$ at $T$
15: \quad \text{rpc-call} \text{COMMIT}($T\text{.cts}, T\text{.writes}$) at each $M$
16: \quad else
17: \quad \text{rpc-call} \text{ABORT()}$ at each $M$
18: \quad return $a$ denoting “aborted”
19: if all COMMIT($T\text{.cts}, T\text{.writes}$) return true then
20: \quad return $c$ denoting “committed”
21: \quad else
22: \quad return $a$ denoting “aborted”

Master-side methods:
23: procedure \text{PREPARE}($T\text{.writes}, T\text{.vc}$)
24: \quad return \text{CHECK-VC}(T\text{.vc}) \&\& write-conflict freedom

25: procedure \text{COMMIT}(T\text{.vc}) \&\& write-conflict freedom
26: \quad apply $T\text{.writes}$ locally and propagate it
27: procedure \text{ABORT()}
28: \quad abort