# Tiga: Accelerating Geo-Distributed Transactions with Synchronized Clocks [Technical Report]

Jinkun Geng[⋆*], Shuai Mu[⋆], Anirudh Sivaraman[†], Balaji Prabhakar[‡]

[⋆]*Stony Brook University*, [†]*New York University*, [‡]*Stanford University*

## Abstract

This paper presents Tiga, a new design for geo-replicated and scalable transactional databases such as Google Spanner. Tiga aims to commit transactions within 1 wide-area roundtrip time, or 1 WRTT, for a wide range of scenarios, while maintaining high throughput with minimal computational overhead. Tiga consolidates concurrency control and consensus, completing both strictly serializable execution and consistent replication in a single round. It uses synchronized clocks to proactively order transactions by assigning each a *future* timestamp at submission. In most cases, transactions arrive at servers before their future timestamps and are serialized according to the designated timestamp, requiring 1 WRTT to commit. In rare cases, transactions are delayed and proactive ordering fails, in which case Tiga falls back to a slow path, committing in 1.5–2 WRTTs. Compared to state-of-the-art solutions, Tiga can commit more transactions at 1-WRTT latency, and incurs much less throughput overhead. Evaluation results show that Tiga outperforms all baselines, achieving 1.3–7.2× higher throughput and 1.4–4.6× lower latency. Tiga is open-sourced at https://github.com/New-Consensus-Concurrency-Control/Tiga.

## 1 Introduction

Distributed online transaction processing (OLTP) systems [30, 36, 38, 48, 49, 51, 64, 66, 73, 74, 85, 88, 92, 93, 96, 97] are fundamental to cloud infrastructures and online services. These systems partition data to scale and replicate data across different datacenters to tolerate server and datacenter outages. Data accesses are guaranteed to be strongly consistent for easy usage. Replication is *linearizable*, and operations are wrapped in transactions with *strict serializability*—together providing users with the illusion of having a single-copy, single-threaded storage with unlimited capacity. To provide this fault-tolerant transaction guarantee, the system uses a concurrency control protocol (e.g., two-phase locking/-commit [34]) to isolate transactions from each other, and a consensus protocol (e.g., Multi-Paxos [55]) to replicate data.

Both concurrency control and consensus protocols are inherently complex and impose significant performance overhead. The overhead includes extra computation (e.g., locking) and additional message roundtrips on the critical path to commit transactions. These additional message roundtrips are especially costly in geo-replicated settings. It may require multiple wide-area roundtrip times (WRTT) to commit a transaction, which significantly impacts latency. Prior work has attempted to reduce the latency overhead, but they often require techniques with substantial computational overhead (e.g., dependency tracking and cycle detection [70, 74]), thus reducing throughput. Additionally, the optimal latency is typically achievable only for a subset of cases, such as when transactions are commutative.

We ask this question in the paper: *Can we design a fault-tolerant transaction protocol that commits more transactions in 1 WRTT with less overhead?* Our answer is Tiga, a lightweight and low-latency protocol based on synchronized clocks. For a wide range of workloads and deployment settings (e.g., server co-location), Tiga can commit transactions in 1 WRTT. Tiga uses an efficient timestamp ordering approach to achieve strong consistency, which yields 2.0–4.5× higher throughput than the other protocols (e.g., Janus/Detock) that rely on intensive graph algorithms for the same consistency guarantee. Tiga achieves this goal through three key design decisions:

***Consolidating consensus and concurrency control.*** Both concurrency control and consensus protocols aim to establish a consistent order across servers, with one handling replicas and the other managing shards. When a system stacks two protocols together, it essentially overpays for achieving the same goal twice. To achieve 1-WRTT latency, as pointed out by previous works [70, 95], it is necessary to consolidate these two layers of protocols. Given this insight, Tiga is designed as a consolidated protocol that unifies concurrency control and consensus.

***Proactive ordering with synchronized clocks.*** Tiga uses timestamps to order transactions, which is a classic lightweight approach in concurrency control [86]. Using synchronized clocks, Tiga measures the one-way delay (OWD) from the transaction's sender (i.e., coordinator) to every participating server and assigns the transaction a future timestamp at submission. The transaction is expected to arrive at all participating servers before the future timestamp. This timestamp includes a *headroom*—an estimate of transmission delay to reach every participating server, which is calculated based on the measured OWDs (§3.1). The headroom effectively masks the heterogeneous latency from the sender to each server: Even if the transaction arrives early at some servers before its timestamp, the servers will hold the transactions until their local clocks pass the timestamp, and then process the transactions based on their timestamp order. This ordering approach reduces the occurrence of inconsistent arrival

[*] Part of the work was done at Stanford University.

orders at different servers, making Tiga's fast path more stable and allowing more transactions to be committed with low latency. Tiga leverages the significant improvements in clock synchronization accuracy over recent decades. Today's clock synchronization algorithms (e.g., Huygens [42]) can achieve microsecond accuracy across data centers and scale robustly [26, 28]. This enables Tiga to extract more performance, as the clock inaccuracy (a few microseconds) is often negligible compared to the headroom (10s of milliseconds).

***Minimizing server coordination overhead.*** Tiga's proactive ordering is best-effort: Transactions can still arrive later than the predicted timestamps—e.g., due to packet drop and retransmission—thereby violating the consistency requirements. Such violations can be subtle and may go undetected by clients or coordinators (Figure 15), unless they communicate with *all* shards during every transaction commit, which is undesirable in practice. To guarantee correctness, Tiga carefully designates one leader per shard, and coordinates these leaders to agree on a timestamp for every transaction. (1) In common deployments with full replication [11, 21, 29, 48, 62, 67, 70, 73, 79], the leaders can be co-located in one geographic region and incur negligible LAN overhead. This setup enables a 1-WRTT fast path to commit most transactions. (2) In more general deployments with partial replication, leaders cannot be co-located in a single geographic region and their coordination introduces WAN overhead. Tiga still allows commutative transactions to be committed via the 1-WRTT fast path. The extra WAN overhead arises only when multiple conflicting transactions are submitted concurrently: the later transactions will be blocked until the leaders reach timestamp agreement for the earlier ones. If the blocking occurs, it typically costs 0.5 WRTT, leading to a graceful performance degradation compared with the full-replication setup.

***Correctness challenge.*** Building a consolidated protocol like Tiga is non-trivial, as both consensus and concurrency control protocols are inherently complex. A major challenge is that it is error-prone. While timestamp-based consensus protocols can achieve linearizability, timestamp-based concurrency control protocols usually only achieve serializability, not *strict* serializability—i.e., they drop the external consistency guarantee provided by linearizability. This is summarized as "timestamp inversion" in recent work [63]. We carefully designed Tiga to avoid timestamp inversion. We have formally proved the correctness of Tiga in Appendix C and included the TLA+ specification in [39].

***Evaluation.*** We implement Tiga as a complete protocol that includes both normal processing and failure recovery. We compare Tiga to layered protocols (2PL/OCC+Paxos, NCC, Calvin+ and Detock) and consolidated protocols (Tapir and Janus) in Google Cloud, using a micro benchmark (MicroBench) and an industry-standard benchmark (TPC-C). We find:

(1) Under MicroBench, when contention is low (skew factor=0.5), Tiga outperforms all baselines by 1.3–7.2× in throughput and by 1.4–4.6× in median latency at close to their respective saturation throughputs. As we fix the load and increase the skew factor, all baselines degrade except Calvin+, but Calvin+ incurs 33% higher latency than Tiga.

(2) Under TPC-C, which contains both one-shot and multi-shot (interactive) transactions, 2PL/OCC+Paxos, Tapir and NCC yield very low throughput (1K–2K txns/s). Tiga outperforms the remaining baselines by 1.6-3.5× in throughput and 1.5-3.7× in median latency.

(3) Tiga's performance varies when using different clocks. Tiga can achieve high performance by using physical clocks if their synchronization error is negligible compared to the cross-region message delay (tens to hundreds of ms). Based on our evaluation, off-the-shelf clock synchronization services (e.g., `chrony` in Google Cloud) can already satisfy this requirement (with $< 5$ ms synchronization error).

## 2 Background and Intuition

***Common setup.*** Distributed OLTP systems are typically modeled as a sharded key-value store. Each shard is replicated across multiple geographic regions. We assume a partial replication setup: different shards can be replicated to different sets of regions. For simplicity, we often use examples with a full replication setup: each region contains a replica from every shard, but this is not necessary.

The system has three roles in transaction processing: *client*, *coordinator*, and *server*. A client sends the transaction request to a coordinator. The coordinator communicates with servers to commit the transaction and returns the execution result to the client. We primarily focus on one-shot transactions, which are written as a stored procedure to be executed on servers. In addition, we incorporate the decomposition technique [87] to support interactive transactions (details explained in Appendix F).

***Necessity of strict serializability***. Strict serializability requires that transactions' executions are equivalent to a serial execution on a single-copy system, and the transactions' executions also reflect the real-time ordering. While some systems [32, 94] choose to sacrifice real-time ordering for performance and only provide serializability, we find that non-strict serializability is insufficient for many practical cases. For example, (1) Banking systems: When transaction processing does not obey real-time ordering, account balances may appear inconsistent to clients; a withdrawal might not be reflected immediately, potentially leading to overdrafts or business errors. (2) Booking/ticket systems: Late booking orders may succeed over earlier ones, thereby causing unfairness among users. (3) Locking service: Clients may observe outdated lock states and perform unsafe operations under the false assumption of ownership.

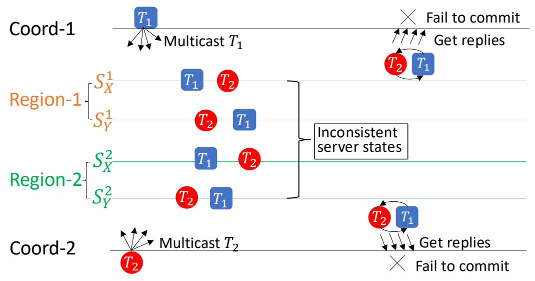


**Figure 1.** Tapir fails to commit transactions in the fast path because $T_1$ and $T_2$ arrive at servers in different orders, causing inconsistent server states.

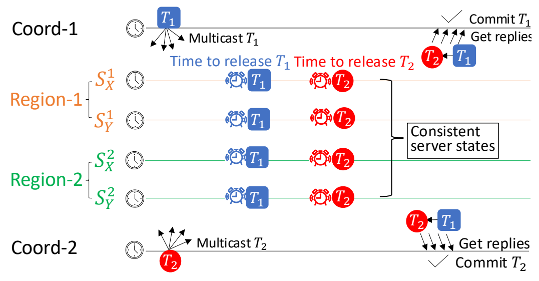


**Figure 2.** Tiga rectifies inconsistent arrival order based on synchronized clocks so that servers process $T_1$ and $T_2$ in the same order, and commit both in the fast path.

Therefore, we target strict serializability. Meanwhile, we aim to achieve fault tolerance, which guarantees strict serializability for all committed transactions in the presence of a minority of server/datacenter failures of any shard.

***Consolidated concurrency control and consensus and 1-WRTT fast path.*** To achieve strict serializability and fault tolerance, we need concurrency control and consensus protocols to coordinate the servers. Although each category of protocols is usually studied separately, it is recognized that they share the same goal—to achieve consistent ordering among all servers [45]. Prior work has proposed consolidating the two types of protocols into a single layer [70, 95], to reduce redundant coordination overhead. In particular, the commit latency in geo-replicated systems can be reduced from several WRTTs to 1–2 WRTTs by the consolidation.

Existing work has proposed having a 1-WRTT fast path for commutative transactions, i.e., transactions that do not conflict with each other. If transactions conflict, the fast path will fail, and more WRTTs are required to commit the transaction. Consider Figure 1, which shows the timestamp-based protocol Tapir [95] and illustrates the problem. The example has 2 coordinators and 2 shards, $X$ and $Y$. Both shards are replicated in 2 regions, Region-1 and Region-2 (technically a 3rd region is needed to tolerate failures; this is omitted to simplify discussions). Each coordinator multicasts a transaction ($T_1$/$T_2$) to all servers. $T_1$ and $T_2$ arrive at the 4 servers in different orders. This inconsistent ordering will form a cyclic dependency that is propagated across servers. Neither $T_1$ nor $T_2$ can commit in 1 WRTT (fast path). Thus, Tapir needs extra RTTs, which are also WRTTs if coordinators and servers are in different regions, to resolve the cycle.

We realize that the fast paths of Tapir and the others (e.g., Janus, Detock, and so on) fail under conflicts because these protocols *optimistically* process transactions based on their arrival order, but transactions' arrival order on different servers can often be different in geo-replicated settings [41].

***Intuition: proactive ordering with timestamps.*** Instead of being optimistic about arrival orders, Tiga chooses the *proactive ordering* approach—Tiga lets the coordinators predict a timestamp for transactions to arrive before multicast. Servers serialize transactions in their timestamp order. The timestamps are generated with synchronized clocks and indicate an approximate serialization point in the global ordering.

When clocks are used to serialize transactions, their accuracy directly impacts system performance. Classic clock synchronization techniques (e.g., NTP [68]) often suffer high synchronization errors (10s of milliseconds) [2], which limited the effectiveness of protocols relying on synchronized clocks.

In recent years, however, clock synchronization accuracy has been improved substantially in practice [12, 42, 59, 71]. For example, Huygens [42] has become generally deployable in the public cloud and can yield microsecond- or even nanosecond-level synchronization errors [26, 28]. In our evaluation, even the default NTP service of Google Cloud (i.e., `chrony`) [44] can steadily converge to 4.54 ms error thanks to well-built cloud infrastructure. This has opened up new opportunities for using physical clocks to serialize transactions: Multiple servers can share a common timeline given the high accuracy of the synchronized clocks. When receiving the transaction, these servers can take actions simultaneously according to its timestamp. However, it is worth noting that even though the clock synchronization accuracy has been greatly improved, there is no guarantee of a *deterministic* error bound. Even for advanced schemes like Huygens, the worst-case error can still go arbitrarily large in theory. Therefore, a desirable protocol should still assume the clock is loosely synchronized, following Liskov's design principle to "depend on clock synchronization for performance but not for correctness" [60].

Figure 2 illustrates Tiga's idea with the same example. When the coordinator multicasts $T_1$ (or $T_2$), it proactively assigns a future timestamp to its transaction, and the timestamp is decided by summing up the sending time and the expected delay from the coordinator to enough (i.e., a super quorum, §3.1) participating servers to commit the transaction. Servers receiving the transactions will hold them until the local clocks pass the transactions' timestamps. Thus, all participating servers can consistently process $T_1$ and $T_2$, and commit both transactions.

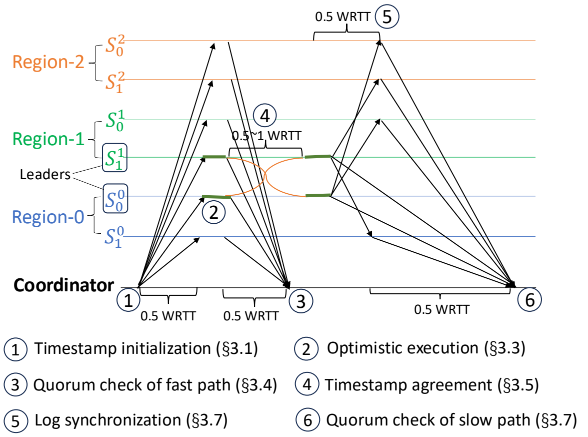

0.5 WRTT
5
Region-2
$S_0^2$
$S_1^2$
4
Region-1
$S_0^1$
0.5~1 WRTT
$S_1^1$
Leaders
$S_0^0$
2
Region-0
$S_1^0$
Coordinator
1
0.5 WRTT
0.5 WRTT
3
0.5 WRTT
6
1 Timestamp initialization (§3.1)
2 Optimistic execution (§3.3)
3 Quorum check of fast path (§3.4)
4 Timestamp agreement (§3.5)
5 Log synchronization (§3.7)
6 Quorum check of slow path (§3.7)

**Figure 3.** Workflow of Tiga. $S_s^r$ indicates the server's *replica-id* is $r$ and *shard-id* is $s$. The green solid bars ② indicate that servers optimistically execute the transaction. However, servers can only know whether the execution is valid after timestamp agreement ④. If the execution turns out to be invalid, servers will revoke the previous execution and re-execute the transaction (see Case-3 in §3.5).

- *shard-id*— shard identifier $(0, 1, \cdots, m-1)$.
- *replica-id*— replica identifier $(0, 1, \cdots, 2f)$.
- *g-view*— the global view, indexed by an integer which is incremented after every view change.
- *l-view*— the local view, indexed by a integer which is incremented or remain unchanged after every view change.
- *status*— one of NORMAL, VIEWCHANGE, or RECOVERING.
- *pq*— a priority queue used to hold incoming transactions, and release transactions according to their timestamp order.
- *log*— a list of transactions, which are appended in the order of their timestamps agreed upon by the participating shards.
- *sync-point*— the log position up to which this server's *log* is consistent with its leader (i.e., the leader that has the same *shard-id* as this server).
- *commit-point*— the log position up to which the server has checkpointed the state.

**Figure 4.** Local state of Tiga servers.

***Challenges.*** While having synchronized clocks can create favorable conditions for Tiga's fast path performance, using the simple rationale we just demonstrated itself is not sufficient to guarantee system correctness. The reason is three-fold. First, timestamps are not assumed to be always accurate. Clocks can be out of sync, or messages can take longer than predicted to arrive. Servers need extra mechanisms to deal with receiving transactions that are supposed to be processed in the past. Second, failures such as crashes and network partitions make it challenging to design a safe protocol that is resilient to corner cases such as recovering a dangling timestamp while the original coordinator has an unknown status (could be either crashed or just slowed). In such cases, the leader is holding the transaction with its timestamp, but the leader becomes isolated due to the network partition. Meanwhile, a new leader is elected and decides a new timestamp for the transaction. As a result, when the old leader becomes reconnected to the system, it must abandon the old timestamp for the transaction. Third, linearizability is a *local* property that is defined for a single shard but strict serializability is not [1, 47], which leads to the fact that a simple timestamp ordering solution is only serializable, losing the "strict" prefix which covers the external consistency or simply the linearizability part. This is also known as the timestamp inversion pitfall by recent work [63]. Extra cross-shard coordination steps need to be carefully designed to achieve correctness.

## 3 Tiga Design

Figure 3 illustrates the workflow of Tiga. We next follow the workflow to explain the protocol details. Figure 4 summarizes the state variables and data structures maintained at each Tiga server, and Algorithm 1 sketches the server's action in response to different events. We refer to these state variables and actions during our explanation. The full algorithm description of server action and coordination action is included in Appendix A.

### 3.1 Timestamp Initialization

The protocol starts with the coordinator multicasting transactions to servers, i.e., ① in Figure 3. When multicasting the transaction $T$, the coordinator needs to predict a future timestamp for $T$, so that $T$ can arrive at all involved servers just before the timestamp. The future timestamp $t$ is calculated by adding the headroom to the transaction's sending time $t_{send}$. We next describe how we estimate the headroom.

The headroom estimation is based on the measurement of the one-way delays (OWDs) between the coordinator and the servers. We use $C$ to represent the coordinator, and use $S_s^r$ to represent a server whose *replica-id* is $r$ and *shard-id* is $s$. We use $O(C, S_s^r)$ to represent the OWD from $C$ to $S_s^r$. These OWDs can be easily measured since the clocks have been synchronized among coordinators and servers. Huygens achieves clock synchronization errors of only a few microseconds (§5.7), which are negligible compared to WAN

**Algorithm 1** Server action

```
1: upon $receiving\ txn$, $T$ do
2:   if CONFLICT-DETECTION($T$)=OK then $pq$.insert($T$)
3:   else if AM-LEADER() then                    ▷ Only leaders can update $T.t$
4:     $T.t \leftarrow$ CLOCK-TIME()
5:     $pq$.insert($T$)
6: upon $clock\ time\ progressing$ do ▷ Periodically check the clock time
7:   $nowTime \leftarrow$ CLOCK-TIME()
8:   $releaseTxns \leftarrow [\,]$
     ▷ Enumerate txns based on timestamp order
9:   for $T \in pq$ do
10:    if $T.t > nowTime$ then break                ▷ $T$ has not expired
11:    if $\nexists T' \in pq : T'.t < T.t$ and $T'$ conflicts with $T$ then
12:      releaseTxns.append($T$)
13:  for $T \in$ releaseTxns do
14:    $\forall$ key $\in$ T.readSet, rMap[key]$\leftarrow T.t$
15:    $\forall$ key $\in$ T.writeSet, wMap[key]$\leftarrow T.t$  ▷ For conflict detection
16:    if AM-LEADER() then
17:      $ret \leftarrow$ EXECUTE($T$)                 ▷ Only leaders execute $T$
18:      $hash$ = CALCULATE-HASH($log$)
19:      SEND-FAST-REPLY($T$, $hash$, $ret$)
         ▷ $tSet$ contains $T$'s timestamps used by each leader
20:      $tSet \leftarrow$TIMESTAMP-AGREEMENT($T$)
21:      if $T.t = \max\{t : t \in tSet\}$ then
22:        if $tSet$.size()>1 then ▷ Some leaders used incorrect $T.t$
           ▷ After completing second round, leaders agree on $T.t$
23:          TIMESTAMP-AGREEMENT($T$)
24:        Append $T$ to $log$ and syncs $T.t$ with followers
25:        $pq$.erase(T)
26:      else                        ▷ This leader used smaller timestamp
27:        $T.t \leftarrow \max\{t : t \in tSet\}$
28:        $pq$.reposition($T$)
29:    else        ▷ Follower sends fast-reply without execution result
30:      SEND-FAST-REPLY($T$, $hash$, $null$)
31:      $pq$.erase(T)
32: upon $follower's\ receiving$ LOG-SYNC, $msg$ do
33:   Update $log$ to keep consistent with leader's $log$
34:   Advance follower's $sync$-$point$
35:   SEND-SLOW-REPLY($T$)
```

OWDs (tens to hundreds of milliseconds), thereby enabling accurate OWD measurement.

Assume that $T$ will be submitted to $m$ shards, and *shard-ids* are $0, \cdots, m-1$. $C$ will assign a future timestamp for $T$, by adding the headroom to its sending time $t_{send}$. The size of the added headroom will decide how likely $T$ can be committed in the fast or slow path.

To commit $T$ via the fast path, its future timestamp $t$ should be sufficiently large for $T$ to reach at least a super quorum $(1 + f + \lceil f/2 \rceil)$ of replicas in each shard.

$$t = t_{send} + \max_{0 \le s < m} \max_{r \in SQ_s} O(C, S_s^r) + \Delta$$

$SQ_s$ represents a super quorum of replicas from the shard (*shard-id* is $s$) that are closest to $C$, i.e., the replicas with smaller OWDs to $C$ than the remaining replicas in the shard. We choose $\Delta = 10ms$ in our implementation so that the headroom added to $t_{send}$ is slightly larger than the OWDs of the super quorum. The necessity of a super quorum (rather than a simple quorum of $f + 1$ servers) for fast path will be explained later in §3.4.

### 3.2 Conflict Detection and Timestamp Update

On each server, Tiga maintains a priority queue (denoted $pq$ in Algorithm 1) to buffer transactions and release them according to their timestamp order. Given a transaction $T$, the server performs *conflict detection* (line 2 in Algorithm 1) to decide whether $T$ can be accepted into $pq$.

***Conflict detection.*** The server checks $T$'s timestamp and will not accept it into $pq$ if $pq$ has already released another transaction $T'$, which has a larger timestamp and has read-write or write-write conflict with $T$ on the same keys. Since transactions are written as (or can be decomposed as) one-shot stored procedures, the server knows their read/write sets before execution. Thus, conflict detection can be implemented very efficiently: The server maintains two maps ($rMap$ and $wMap$). Both maps associate every data item (key) of the key-value store with a timestamp. When $T'$ is released from $pq$ (the release conditions will be explained in §3.3), the server uses $T'$'s timestamp to update the timestamp of every key that falls in $T'$'s read/write set (line 14–15 in Algorithm 1). When $T$ arrives, the server directly compares $T$'s timestamp with the recorded timestamps of keys that $T$ will read/write. $T$ will be accepted into $pq$ if its timestamp is larger than the timestamps of all conflicting transactions that have been released from $pq$ (line 1–2).

Not every transaction can be accepted into $pq$ after conflict detection. When $T$ arrives late at a leader server due to network delay or packet loss, its timestamp may be too small to be accepted into $pq$. In such cases, the leader updates $T$'s timestamp to the local clock time (line 4), after which $T$ can enter the leader's queue.

Followers, by contrast, do not perform timestamp updates. If $T.t$ is smaller than acceptable for the queue, the follower holds $T$ and waits for synchronization instructions from the leader in the slow path (§3.7, line 32–35 in Algorithm 1).

### 3.3 Optimistic Execution

For each transaction $T$ in the queue $pq$, followers refer to the local clock to determine when to release it. Once the local time surpasses $T.t$, the follower releases $T$ without executing it: $T$ is removed from the queue and then appended to the log list. After that, the follower sends a fast-reply to the coordinator to perform a quorum check (§3.4).

Leaders, on the other hand, must execute transactions before releasing them. To minimize latency (1 WRTT), leaders optimistically execute transactions without coordination. Periodically, the leader refers to its local clock to identify *expired* transactions (i.e., transactions whose timestamps have been passed by the current clock time) in its queue. It checks these transactions in timestamp order to decide whether each can be optimistically executed. When $T$ has reached the head of the queue without any conflicting transactions ahead, $T$ can be executed. However, $T$ will remain at the head of the queue after execution.

After executing $T$, the leader sends a fast-reply to the coordinator, including the execution results. $T$ stays at the head of the leader's queue to undergo timestamp agreement (see §3.5). After that, $T$ is either released (line 25) or repositioned in the queue with a larger timestamp (line 28).

Before $T$ can be released, the leader/follower records $T$'s timestamp with its read set and write set (line 14-15) for subsequent conflict detection (§3.2, line 2 in Algorithm 1). The follow-up transaction is no longer acceptable into the queue if it conflicts with $T$ but has a smaller timestamp.

### 3.4 Quorum Check of Fast Path

A server's fast-reply regarding transaction $T$ includes a hash value of the log list to represent its state before $T$. The hash of the log list is computed as the bitwise exclusive-or (XOR) of the hashes of all its entries. This allows the server to *incrementally* compute the hash. When adding/deleting a log entry $e$, the new hash is computed as $H_{new} = XOR(H_{old}, hash(e))$. We use a 160-bit SHA-1 hash and assume hashes do not collide in practice. Note that applying XOR on hashes does not make them vulnerable to collisions [16, 17]. It is a commonly applied technique in systems using incremental hash [4, 14, 15, 25, 37, 41]. Appendix D provides more details on how Tiga uses incremental hash. In addition, the fast-reply includes $T$'s timestamp.

The coordinator receives $T$'s fast-replies from all servers. $T$ is considered *fast-committed* on a shard if, from this shard, the coordinator receives a super quorum of fast-replies that have the same hash and timestamp for $T$. The super quorum must satisfy two conditions: (1) it contains the leader, and (2) its size is at least $1 + f + \lceil f/2 \rceil$. If these are met, the coordinator uses the optimistic results in the leaders' replies as $T$'s execution results on this shard.

The reason that Tiga's fast path requires a super quorum ($1+f+\lceil f/2 \rceil$) instead of a simple quorum ($f+1$) is similar to Fast Paxos [56]: Because the fast path omits leader–follower communication, a simple quorum lacks sufficient information for a new leader to distinguish committed from uncommitted transactions. Consider the leader and $f$ followers append $T_1$ and $T_3$ ($T_1 \rightarrow T_3$) in their log lists whereas the other $f$ followers append $T_2$ and $T_3$ ($T_2 \rightarrow T_3$) in the same positions of their log lists. Assuming the fast path only requires a simple quorum, then $T_1$ and $T_3$ will be considered committed. However, when the leader fails, both $T_1$ and $T_2$ exist among half of the remaining servers, thus the new leader cannot know whether $T_1 \rightarrow T_3$ or $T_2 \rightarrow T_3$ is previously committed. If the new leader mistakenly believes $T_2 \rightarrow T_3$ is previously committed, then $T_3$ will have a different execution result compared to that before the crash.

If $T$ is fast-committed on all its involved shards, the coordinator additionally checks whether leaders have consistent timestamps for $T$ in their fast-replies, because some leaders may have updated $T$'s timestamp whereas the others have not. If all participating leaders have used the same timestamp to execute $T$, $T$ is committed in the fast path.

### 3.5 Timestamp Agreement

After $T$'s execution, the leaders need to verify whether the execution is valid; i.e., all participating leaders should execute $T$ in the same timestamp order; otherwise, the execution may violate strict serializability and should be revoked. To support revoking, Tiga maintains multiple versions for each data item (key). $T$'s optimistic execution creates new versions of data. Once the server detects the execution is invalid, it erases the corresponding data versions. Note that the revoking operation is internal to Tiga, and does not cause application-related rollback.

To check the validity of $T$'s execution, leaders start a round of message exchange. Each leader notifies the other participating leaders of its local timestamp $T.t$. Then, each leader collects the full set of $T$'s timestamps used by different leaders, and computes the maximum as the agreed timestamp, $T.t_{agreed}$. Since all leaders operate on the same timestamp set, they deterministically compute the same $T.t_{agreed}$. The subsequent actions depend on three possible cases:

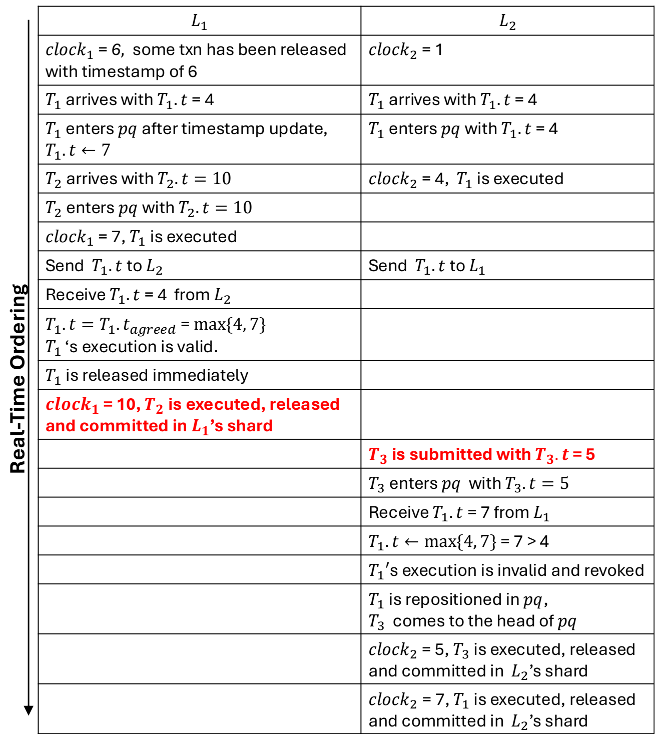
***Case-1:*** All timestamps match. This is the ideal case, which takes only 0.5 WRTT (④ in Figure 3) for leaders to notify each other. If every leader's local timestamp equals $T.t_{\text{agreed}}$, the timestamp agreement succeeds immediately. Each leader releases $T$ and then appends $T$ to its log.

***Case-2:*** This leader used $T.t_{agreed}$, but others did not. In this case, the leader's optimistic execution remains valid, but some other leaders used smaller timestamps. To avoid potential timestamp inversion (discussed in §3.6), the leader cannot release $T$ immediately. Instead, it initiates a second round of timestamp exchange (another 0.5 WRTT) to confirm that all leaders have updated $T$'s timestamp to $T.t_{\text{agreed}}$. Once confirmed, the leader proceeds as in Case-1. In this case, the timestamp agreement ④ takes 1 WRTT in total.

***Case-3:*** This leader used a timestamp smaller than $T.t_{agreed}$. This indicates that the leader's optimistic execution is invalid, so it revokes $T$'s execution. Then, it updates $T$'s timestamp: $T.t \leftarrow T.t_{agreed}$. After that, the leader initiates the second round of timestamp exchange (another 0.5 WRTT) to notify the other leaders. Since $T$'s timestamp changes to a larger value, $T$ will be repositioned in the leader's queue. Eventually, $T$ will come to the head again, and then the leader will re-execute $T$ with the agreed timestamp.

### 3.6 Avoiding Timestamp Inversion

Readers may wonder why the leader in Case-2, denoted $L_1$, cannot immediately release $T$ as this leader already used the agreed timestamp for $T$. The reason is the timestamp inversion pitfall [63], which hurts correctness, in particular strict


Real-Time Ordering

| $L_1$ | $L_2$ |
|---|---|
| $clock_1 = 6$, some txn has been released with timestamp of 6 | $clock_2 = 1$ |
| $T_1$ arrives with $T_1.t = 4$ | $T_1$ arrives with $T_1.t = 4$ |
| $T_1$ enters $pq$ after timestamp update, $T_1.t \leftarrow 7$ | $T_1$ enters $pq$ with $T_1.t = 4$ |
| $T_2$ arrives with $T_2.t = 10$ | $clock_2 = 4$, $T_1$ is executed |
| $T_2$ enters $pq$ with $T_2.t = 10$ | |
| $clock_1 = 7$, $T_1$ is executed | |
| Send $T_1.t$ to $L_2$ | Send $T_1.t$ to $L_1$ |
| Receive $T_1.t = 4$ from $L_2$ | |
| $T_1.t = T_1.t_{agreed} = \max\{4, 7\}$<br>$T_1$ 's execution is valid. | |
| $T_1$ is released immediately | |
| **$clock_1 = 10$, $T_2$ is executed, released and committed in $L_1$'s shard** | |
| | **$T_3$ is submitted with $T_3.t = 5$** |
| | $T_3$ enters $pq$ with $T_3.t = 5$ |
| | Receive $T_1.t = 7$ from $L_1$ |
| | $T_1.t \leftarrow \max\{4, 7\} = 7 > 4$ |
| | $T_1'$s execution is invalid and revoked |
| | $T_1$ is repositioned in $pq$,<br>$T_3$ comes to the head of $pq$ |
| | $clock_2 = 5$, $T_3$ is executed, released and committed in $L_2$'s shard |
| | $clock_2 = 7$, $T_1$ is executed, released and committed in $L_2$'s shard |

**Figure 5.** Illustration of timestamp inversion. With only one round of message exchange between $L_1$ and $L_2$, $T_3$ may be submitted after $T_2$ is committed, leading to real-time ordering $T_2 \longrightarrow T_3$ that contradicts serializable order $T_3 \longrightarrow T_1 \longrightarrow T_2$.

serializability. While $L_1$ has confirmed it used the proper timestamp $T.t_{agreed}$ for $T$, the other leader(s), denoted $L_2$, used a smaller timestamp $T.t < T.t_{agreed}$, and $L_1$ is uncertain whether $L_2$ has completed timestamp agreement and updated $T.t$ to $T.t_{agreed}$. As a result, if $L_1$ releases $T$ immediately, timestamp inversion may occur: After $L_1$ has committed some transactions with timestamps larger than $T.t_{agreed}$, $L_2$ could still commit other transactions with smaller timestamps than $T.t_{agreed}$.

Figure 5 shows a concrete sequence of events illustrating how timestamp inversion occurs. $L_1$ and $L_2$ are the leaders of two shards, and they process three transactions $T_1$, $T_2$ and $T_3$. $T_1$ is a multi-shard transaction that involves both $L_1$'s and $L_2$'s shards. $T_2$ is only processed by $L_1$'s shard; $T_3$ is only processed by $L_2$'s shard. The leaders' clocks (e.g., $clock_1$ and $clock_2$) are badly synchronized.

$L_1$'s event sequence indicates the dependency relation $T_1 \longrightarrow T_2$ and $L_2$'s event sequence indicates $T_3 \longrightarrow T_1$, so the only valid serializable schedule is $T_3 \longrightarrow T_1 \longrightarrow T_2$. However, in real time, $T_3$ starts after $T_2$ has been completed, indicating the real-time ordering relation $T_2 \longrightarrow T_3$, which contradicts the serializable order.

The fundamental reason behind timestamp inversion lies in the different guarantees of linearizability versus strict serializability. Linearizability is a **local** property within each shard—e.g., $L_1$ only needs to ensure its followers in the same shard use a consistent order between $T_1$ and $T_2$ with $L_1$ itself, and does not consider the ordering between $T_2$ and $T_3$, which are processed by different shards. The same holds for $L_2$. In contrast, strict serializability enforces a **global** order across shards. Although $T_2$ and $T_3$ do not directly conflict, they both access data involved in $T_1$, forming a dependency chain that induces a real-time order between them. This indirect dependency is not captured by linearizability, but is essential for preserving strict serializability.

To avoid timestamp inversion, when a leader notices it is holding a different timestamp from the other leaders for a transaction, it must ensure no other transactions with smaller timestamps (e.g., $T_3$) can be committed later. Specifically in Figure 5, $L_1$ should release $T_1$ after it confirms that $L_2$ has updated $T_1$ with the agreed timestamp, and $T_1$ has come to the head of the queue again (after repositioning). At this point, (1) $T_3$ has been executed on $L_2$ whereas $T_2$ remains in $L_1$'s queue, because $T_1$ is at the head of the queue and blocking $T_2$ from execution. (2) $L_2$ will no longer allow the other transactions (which conflict with $T_1$) to enter its queue with smaller timestamps than $T_1$'s agreed timestamp ($T_1.t = 7$). Thus, the second round of timestamp agreement rules out any real-time ordering violations that would otherwise conflict with the serializable schedule. We include the proof in Appendix C.

In contrast to the leaders, the followers do not engage in timestamp update and agreement, so they may have different timestamps from the leaders at this point. The potential leader-follower inconsistency will be detected in the fast path by comparing the hashes and $T$'s timestamp (§3.4) and resolved in the slow path (§3.7).

### 3.7 Log Synchronization and Slow Path

Tiga does not guarantee that all transactions are committed in the fast path. If a leader updates a timestamp, it causes inconsistency between itself and its followers. Therefore, after appending the transaction to its log, the leader advances its *sync-point* and also sends the followers a log synchronization message. In the synchronization messages, the leader includes the entry's position, unique identifier[1] and the timestamp agreed by leaders. When receiving the synchronization message, the followers update their logs to keep consistent with the leader's log: (1) If the follower's log contains some entry that does not exist in the leader's log, then the follower removes the entry; (2) If the leader's log contains some entry that does not exist in the follower's log, then the follower first tries to obtain the missing entry locally from its server. If the entry is not found, then the follower fetches it from the leader. (3) If some entry exists in both the leader's and

[1] The coordinator attaches a sequence number to the transaction at submission. The unique identifier for this transaction is to combine the coordinator-id and the sequence number.

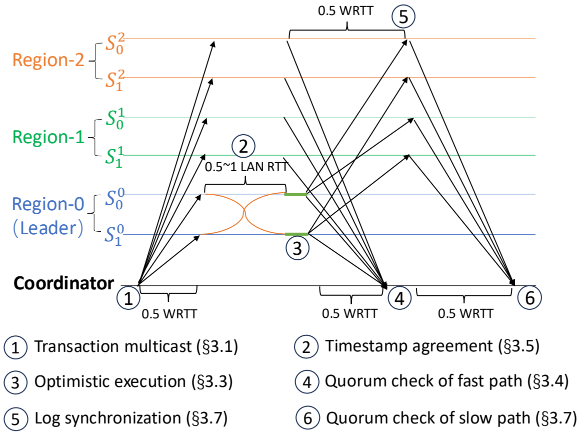

0.5 WRTT
5
Region-2
$S_0^2$
$S_1^2$
Region-1
$S_0^1$
$S_1^1$
2
0.5~1 LAN RTT
Region-0
(Leader)
$S_0^0$
$S_1^0$
3
Coordinator
1
0.5 WRTT
0.5 WRTT
4
0.5 WRTT
6
1 Transaction multicast (§3.1)
2 Timestamp agreement (§3.5)
3 Optimistic execution (§3.3)
4 Quorum check of fast path (§3.4)
5 Log synchronization (§3.7)
6 Quorum check of slow path (§3.7)

**Figure 6.** Workflow of Tiga (preventive approach).

the follower's logs but has different timestamps, then the follower updates the entry's timestamp to keep consistent with the leader.

After the log update, the follower advances its *sync-point* to indicate its log list has been synchronized with the leader's log list up to this point. Then, the follower sends slow-replies to the coordinators which have multicast those synchronized entries, notifying the coordinators that entries for these transactions have become consistent with the leader. Appendix E describes an optimization that does not require the followers to send the slow reply for every entry.

Followers also periodically report their *sync-point*s to the leader, so that the leader knows which log entries have been sufficiently replicated. After the leader confirms that the log entries have been surpassed by the *sync-points* from $f + 1$ servers of the same shard, the leader knows these entries are committed. The leader then advances its *commit-point* and notifies its followers of the updated *commit-point*. Followers can execute the log entries up to their *commit-point*s and generate checkpoints to accelerate failure recovery (§4).

A transaction is considered *slow-committed* on a shard if the coordinator (1) receives the fast-reply from the leader and (2) receives slow-replies from at least $f$ followers. If the transaction is either fast- or slow-committed on every involved shard, it is considered committed.

### 3.8 Optimization based on Leaders' Co-location

In §3.3-§3.5, we let the leaders start optimistic execution without waiting for timestamp agreement. The purpose is to minimize the latency of the fast path, because timestamp agreement costs additional WAN latency when leaders are separated across regions. However, skipping timestamp agreement in the fast path introduces the risk of invalid execution, i.e., different shards execute transactions based on inconsistent timestamp orders, incurring expensive rollback.

Alternatively, if timestamp agreement is *cheap*, i.e., it only costs LAN latency, then prioritizing timestamp agreement over execution is more desirable: It only adds negligible overhead to the commit latency, but avoids the rollback of invalid execution, because all the leaders execute the transactions according to their agreed timestamp order. Fortunately, we realize that this approach is commonly feasible in practical deployment. In typical geo-distributed OLTP systems [11, 21, 48, 62, 67, 70, 73, 79], each datacenter (region) usually contains a full copy of data, thus enabling co-location of all leaders within the same region. In addition, industry workloads also exhibit strong data locality. For example, the Yahoo! trace [29] reveals 85% regional locality for user data accesses; the typical edge workload [21] has 90% of intra-region transactions. By leveraging the co-location property, we can schedule timestamp agreement ahead of execution, as shown in Figure 6, in contrast to Figure 3.

***Choose the approach of timestamp agreement.*** Since there is no one-size-fits-all approach towards different deployments, Tiga incorporates both approaches into the protocol design, with the choice being configurable through its modified view change protocol (§4). Specifically, Tiga leverages Huygens' probe mesh to continuously monitor the OWDs between servers. Based on the measured OWDs, Tiga initializes a view change to designate the leader for every shard. Tiga tries to co-locate all leaders close to each other, so that it can schedule timestamp agreement before execution, which costs negligible LAN overhead but prevents invalid execution at its root. However, if co-location is infeasible, i.e., Tiga cannot find a group of leaders with OWDs below a predefined threshold (e.g., 10 ms), then the preventive approach becomes inefficient, prompting Tiga to adopt the detective approach (Figure 3). The view change message includes the planned approach (i.e., preventive or detective), so that all servers consistently adopt the planned approach after entering the new view.

## 4 Failure Recovery

Server failures in Tiga can be classified into two categories: leader failures and follower failures. Follower failures are relatively easier to deal with. A minority of follower failures in any shard do not interrupt service availability. Tiga can always use the slow path to commit transactions if the servers alive are insufficient for the super quorum in the fast path. When failed followers reboot, they catch up by synchronizing logs with the leader. Here, we mainly discuss leader failure handling. Further details and the correctness proof are included in Appendix B and Appendix C.

Tiga uses a *view-based* [76] protocol to facilitate leader failure recovery. A view records information on membership, including each member's role, i.e., as a leader or follower.

Tiga distinguishes between two views: a local view (*l-view*) which stores information about a shard, and a global view (*g-view*) about all shards. A global view includes all local views. Both the global views and the local views are indexed by unique and monotonically increasing integers. The views are managed by a *view manager*. The view manager is a simple service implemented on a replicated state machine that is resilient to failures, e.g., it could be built with ZooKeeper [9]. It is off the critical path of transaction processing in the common cases, so its performance has no significant impact.

Every server stores both the global and its local view. When the system is stable (no failures), all servers have the same global view, which also implies that servers of the same shard share the same local view. A server attaches the global and local view-ids to every message it sends out. When receiving a message from other shards, a server always checks and rejects the message if the message has a different global view. If the message is from within the shard, the server also checks whether it has the same local view.

The view manager detects server failure(s) using heartbeats, and launches a view change if a leader fails. The view change proceeds in the following steps.

① The view manager creates a new view that has new leader(s) to replace the failed one(s). When selecting new leaders, the view manager prioritizes choices that can make most leaders co-located in the same region, so that inter-leader timestamp agreement only costs LAN overhead after the system resumes normal processing in the new view. Based on the latency cost of timestamp agreement, the view manager decides whether to use the preventive or detective approach (§3.8) in the new view. The view manager also creates new view-ids by incrementing the old view-ids. This includes a new global view-id and new local view-ids for the shards whose leaders are changed.

② The view manager broadcasts the new view to all servers in the system. When a server receives a newer view (i.e., higher *<g-view,l-view>*), it will update its view, and switch its *status* from NORMAL to VIEWCHANGE.

At the start of the view change, the servers stop processing new transactions. Each server empties its queue and appends the transactions in the queue to its log list according to their timestamp order. The new leader is responsible for collecting the servers' log lists and rebuilding a new log list that contains all the previously committed transactions.

③ If a server is the new leader of a shard, it rebuilds a new log list based on the log lists from any $f + 1$ servers that remain alive in this shard. The reconstruction of the log list includes two parts: (a) The leader finds the server that is holding the largest *sync-point* among the $f + 1$ servers, and copies its log list up to the *sync-point*. (b) The leader continues to check the remaining entries. For any remaining entry with a larger timestamp than those recovered in (a), if it exists in the log lists of $\lceil f/2 \rceil + 1$ participating servers, then this entry will also be appended to the leader's log list according to its timestamp order.

④ Because the leaders' timestamp agreement *happens before* followers advance their *sync-points*, (a)'s log entries have the agreed timestamps across shards. But (b)'s log entries may have inconsistent timestamps across shards. Therefore, after rebuilding the log lists, the leaders conduct timestamp agreement for (b)'s log entries: (1) If a recovered transaction involves both $shard_1$ and $shard_2$, but it is only recovered in $shard_1$, then $shard_2$'s leader will pick the transaction from $shard_1$ to add to its own log list. (2) If a recovered transaction has inconsistent timestamps across shards, the leaders pick the maximum one as the agreed timestamp.

⑤ After timestamp agreement, each leader broadcasts its log list to its followers. Leaders execute the recovered logs and switch back to NORMAL. Followers use leaders' log lists to replace their old ones, then switch back to NORMAL.

To complete the overall design, the coordinator(s) in Tiga also cache the global view from the view manager. It only accepts replies that have the same global view-id. In case of a view change, the coordinator retries the transaction.

***Coordinator failure.*** If a coordinator fails, the servers will detect it after a timeout and launch a recovery coordinator to commit the transaction following the same coordinator procedure (§3.1, §3.4 and §3.7). The newly launched coordinator can always fetch the view information from the view manager, and itself is stateless. As a result, the coordinator failure does not trigger any view change.

***Checkpoints to accelerate recovery.*** Tiga incorporates a periodic checkpoint mechanism, a common practice for accelerating the recovery of transaction processing systems [23, 41, 58, 80]. Since each server maintains the *commit-point*, the server can safely execute the log entries prior to its *commit-point*, and checkpoint the system state. When servers fail, the new leader can restore the system state from the latest checkpoint rather than from scratch. The failed follower can first fetch the latest checkpoint from the leader and catch up, significantly speeding up recovery.

# 5 Evaluation

We build on the Janus codebase [69], which provides a high-performance implementation of several baseline protocols, including 2PL+Paxos, OCC+Paxos, Tapir, Janus and NCC. Using the same RPC library and runtime environment, we implement Tiga along with additional baselines, such as Detock [74] and an enhanced version of Calvin [88], namely Calvin+. Calvin+ replaces Calvin's Paxos-based consensus layer with Nezha [41], saving at least 1 WRTT in committing transactions. In total, we compare Tiga against 8 baselines across a range of workloads.

## 5.1 Evaluation Setup

***Workloads.*** We employ 2 benchmarks: a micro-benchmark (MicroBench) and the widely used TPC-C [31]. MicroBench pre-populates each shard with 1 million key-value pairs. Each transaction performs 3 read-write operations across different shards by incrementing 3 key-value pairs. The key-value pairs are selected using a Zipfian distribution [46]. We

**Table 1.** Maximum throughput ($10^3$ txns/s).

| Benchmark | 2PL+Paxos | OCC+Paxos | Tapir | Janus | Calvin+ | Detock | NCC | Tiga |
|---|---|---|---|---|---|---|---|---|
| MicroBench | 22.9 | 21.8 | 44.2 | 77.8 | 119.6 | 34.5 | 47.4 | 157.3 |
| TPC-C | 2.1 | 0.9 | 1.1 | 10.8 | 6.1 | 13.3 | 0.86 | 21.6 |

tune the skew factor of the Zipfian distribution to control the contention in MicroBench, where higher skew factors yield more contention. For TPC-C, we implement all 5 types of transactions according to the specification [31]. Additionally, we follow NCC's approach [63] and make 2 types (`Payment` and `Order-Status`) multi-shot (interactive) transactions.

***Baselines and testbed.*** We compare Tiga to the 8 baseline protocols. 2PL+Paxos utilizes the wound-wait mechanism [83] to prevent deadlocks. Detock performs only local replication at commit and performs geo-replication asynchronously [74]. To tolerate region failures, we make Detock perform synchronous geo-replication during transaction commit. In Detock, we evenly distribute the home directories of data items across regions. NCC's implementation does not tolerate server failure, and suggests using Paxos to achieve fault tolerance, so we implement NCC+ by placing NCC atop a Paxos replication layer. 2PL/OCC+Paxos and NCC inherently support interactive transactions by design. For the other protocols (Calvin+, Janus, Detock and Tiga), we integrate the decomposition technique [87] (explained in Appendix F) to support interactive transactions.

All experiments are conducted in Google Cloud. We use `n2-standard-16` VMs to run servers and coordinators (clients are co-located with coordinators on the same VMs). Data is replicated across 3 regions: South Carolina, Finland, and Brazil. In practical deployment, clients/coordinators can either be co-located or separated from the servers, so we consider both cases: (1) We deploy 2 coordinators in each of the 3 regions (local regions). (2) We also deploy 2 coordinators in the 4th region (remote region), Hong Kong, because some coordinators might not be allowed to co-locate with servers due to governmental regulations (e.g., GDPR [35], DSL [72]) or proprietary business reasons. The system is configured with 3 shards (9 servers in total) for MicroBench and 6 shards (18 servers in total) for TPC-C to be consistent with Janus' original setup.

***Evaluation method.*** We evaluate the performance of the protocols using an open-loop approach [89]: Each coordinator submits transactions at a given rate. The coordinator maintains a cap on the outstanding transactions and stops submitting new transactions once this cap is reached. Each test is repeated 5 times, and we report the median of the 5 trials. Since each region contains a full copy of the data, Tiga adopts the preventive approach in all evaluations except in §5.5 and §5.6. §5.5 compares the performance of Tiga's preventive and detective approaches. §5.6 evaluates the impact of headroom on Tiga's latency and rollback rate.

### 5.2 MicroBench

We first run MicroBench with a fixed skew factor of 0.5 and compare protocols' performance by increasing the submission rate of each coordinator (Table 1 and Figure 7-8). Then we fix per-coordinator rate at 8K txns/s, and compare the protocols' performance by varying skew factor from 0.5 to 0.99 (Figure 9). We measure the coordinators' throughput, commit rate, 50th and 90th percentile latency in each region.

Our evaluation highlights Tiga's efficiency in achieving strict serializability and fault tolerance, outperforming state-of-the-art protocols across various metrics. Specifically: (1) 2PL/OCC+Paxos reach their throughput bottlenecks very early due to the Paxos consensus layer. Besides, the added WRTTs by the consensus layer inflate commit latency and extend the locking window, leading to more aborts. (2) Tapir's commit rate decreases rapidly as the load increases, because more concurrent transactions arrive at the servers in different orders, making Tapir abort more transactions to resolve ordering inconsistencies. (3) Janus and Detock run expensive graph algorithms to resolve inconsistencies between servers. When the submission rate grows and/or the contention (skew factor) increases, the graph computation becomes a bottleneck. Detock incurs even more WRTTs due to its layered design. In addition, since the home directories of different data items are distributed across regions, Detock pays extra WAN RTTs for dependency collection, further impacting performance. (4) Calvin+ uses an epoch-based mechanism to predefine transactions' order, which is more robust to the various skew factors. However, it suffers from the straggler problem—when one shard is overloaded and slows down, the entire system is affected, reducing throughput and increasing latency. (5) NCC does not include fault tolerance for servers, and all servers are located in one region (South Carolina), so it costs only LAN latency in this region, and requires at least 1 WRTT in the other three regions. However, NCC uses Response Time Control (RTC) to guarantee strict serializability. RTC makes servers release a transaction only after the previous conflicting transaction sends back the commit notification. Thus, RTC artificially creates a 1-WRTT gap between these conflicting transactions. This leads to significant queueing delay. Under high load and contention, RTC limits NCC's throughput and causes latency to rise rapidly. Besides, after adding fault tolerance, NCC+ experiences further performance degradation.

Compared to the local region (South Carolina), Tiga's latency advantage becomes more pronounced in the remote region (Hong Kong). In the local region (Figure 7), Janus/-Tapir/Calvin+ can yield 1-WRTT latency at a low submission rate. However, in the remote region without co-located servers (Figure 8), they all require at least 2 WRTTs to commit. In contrast, Tiga consistently achieves 1-WRTT latency in both regions due to its efficient fast path design, delivering higher performance in more general deployment scenarios.

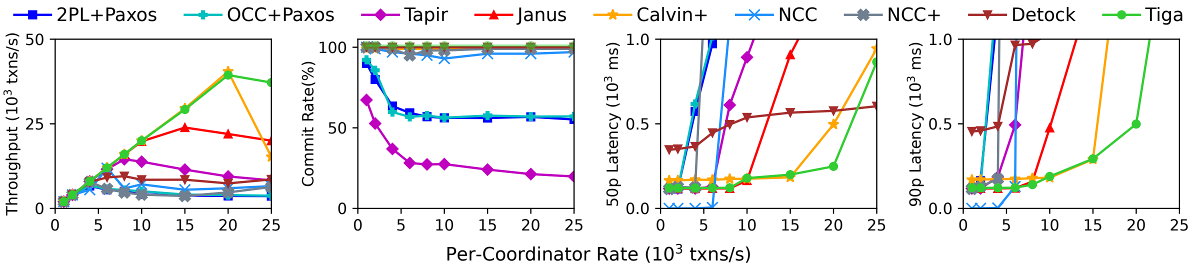


**Figure 7.** MicroBench (skew factor=0.5) with varying rates in local region (South Carolina).

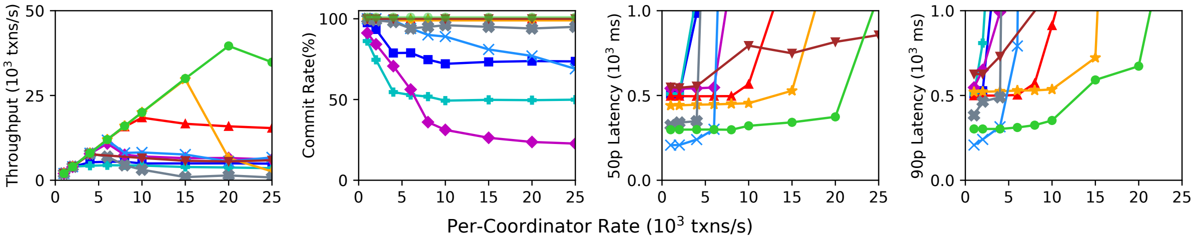


**Figure 8.** MicroBench (skew factor=0.5) with varying rates in remote region (Hong Kong).

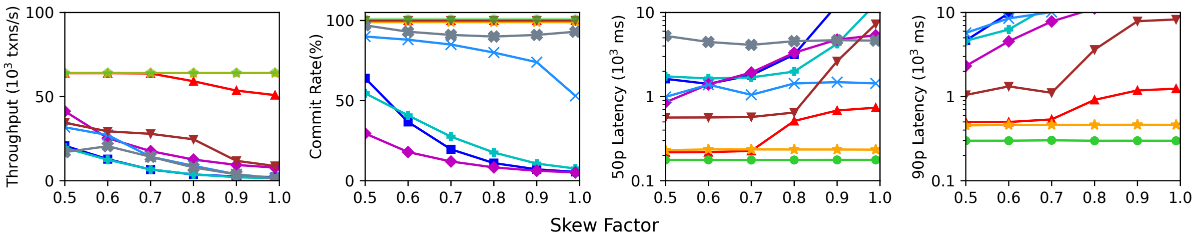


**Figure 9.** MicroBench (per-coordinator rate=8K txns/s) with varying skew factors (all regions).

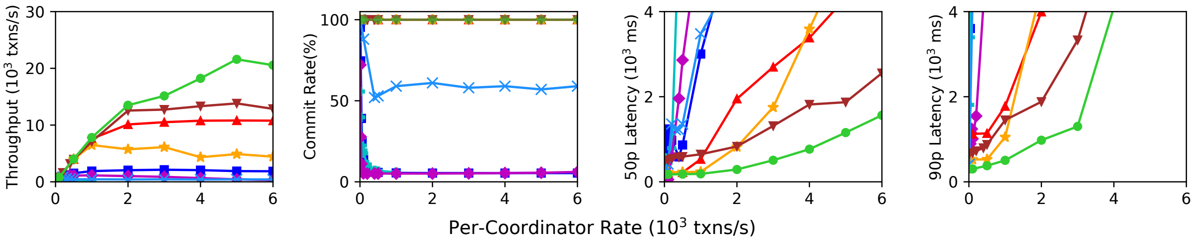
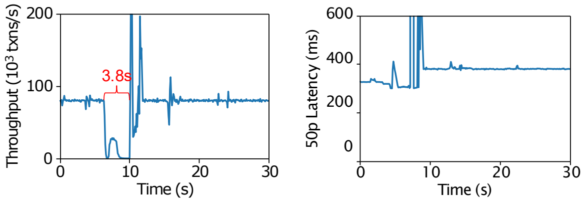


**Figure 10.** TPC-C with varying rates (all regions).

### 5.3 TPC-C

Compared with MicroBench, TPC-C exhibits more complexity and higher contention: (1) Over 92% of transactions are read-modify-write operations, with some requiring multiple shots to complete; (2) Since the data is stored in a column-based manner (as implemented by Janus), transactions can conflict frequently as long as they attempt to write the same column. (3) TPC-C transactions are more CPU-intensive than MicroBench, resulting in lower throughput for all protocols.

Table 1 and Figure 10 present the evaluation results, with three main takeaways. (1) Under such a high-contention workload, 2PL+Paxos, OCC+Paxos and Tapir all suffer from very low throughput due to frequent transaction aborts. Among them, 2PL+Paxos performs slightly better because its wound-wait mechanism reduces many transaction aborts. (2) NCC only achieves hundreds of txns/s of throughput, and NCC+'s throughput is even lower (not shown in Figure 10). NCC's poor performance stems not only from aborts but also from high queueing delays caused by its RTC mechanism. The queueing delay leads to a buildup of outstanding transactions, which can easily reach the cap during our open-loop



**(a)** Total throughput **(b)** Latency (Hong Kong)

**Figure 11.** Tiga performance before/after leader failure.

**Table 2.** Performance comparison after server rotation.

| | 2PL+Paxos | OCC+Paxos | Tapir | Janus | Calvin+ | NCC | Tiga |
|---|---|---|---|---|---|---|---|
| **Thpt** | 18.6 | 18.0 | 44.7 | 71.9 | 120.0 | 40.7 | 141.9 |
| +/−% | -18.8% | -17.4% | +1.1% | -7.5% | +0.3% | -16.5% | -9.7% |
| **Latency** | 1.09 | 1.11 | 0.44 | 0.46 | 0.67 | 0.73 | 0.30 |
| +/−% | +47.2% | +38.9% | +83.3% | +39.3% | +162% | +72.5% | +34.0% |

Since Detock already distributes the home directories of data items *across regions*, server rotation does not affect its performance.

tests, and prevent coordinators from issuing more transactions. (3) Janus, Calvin+ and Detock all benefit from being largely abort-free, as does Tiga. Under TPC-C, Calvin+ becomes less efficient than Janus and Detock, as more shards are involved and the straggler effect becomes more distinct. However, all baselines are less efficient than Tiga's approach based on synchronized clocks, enabling Tiga to achieve the highest throughput and lowest latency.

### 5.4 Failure Recovery Evaluation

We re-run MicroBench (skew factor=0.5), and each coordinator submits 10K txns/s (80K txns/s in total). We kill the leader in one shard and compare the performance (latency and throughput) before and after the leader failure. Figure 11a shows that Tiga takes only 3.8 seconds to complete the global view change and recover to the same level of throughput. After the recovery, the commit latency increases (Figure 11b) because one of the shards only has $f + 1 = 2$ remaining servers. When transactions involve the data from this shard, they can only be committed in the slow path. However, even in such cases, Tiga's coordinators in the remote region (Hong Kong) still yield lower latency than the other protocols under the same workload (Figure 8).

### 5.5 Leaders Separation vs. Leaders Colocation

When leaders cannot be located in the same region, Tiga prioritizes optimistic execution without waiting for timestamp agreement, to achieve 1-WRTT latency. To evaluate Tiga in this setting, we rotate the *shard-ids* and *replica-ids* for each server so that servers with the same *shard-id* are located in different regions. We continue to run MicroBench (skew factor=0.5). Table 2 summarizes the maximum throughput and the 50th percentile latency at this throughput, as well as the relative difference (+/−%) compared to the previous setting (Table 1) where leaders are co-located.

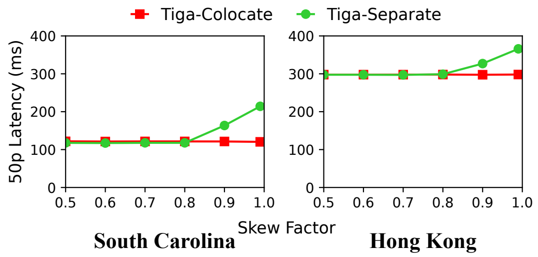


**Figure 12.** MicroBench latency performance with varying skew factors (per-coordinator rate=8K txns/s).

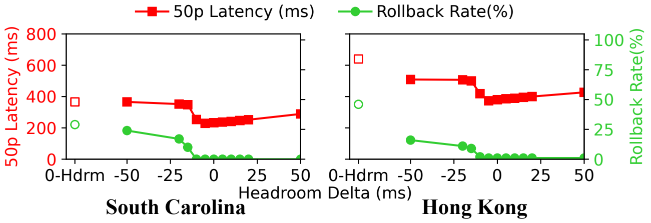


**Figure 13.** Tiga performance with varying headroom (per-coordinator rate=8K txns/s), 0-Hdrm (i.e., headroom=0 ms) directly uses sending time for proactive ordering.

Table 2 indicates that Tiga's throughput decreases by 9.7%, but it still outperforms the other protocols in both throughput and latency. Calvin+ achieves the highest throughput among baselines, but its latency increases significantly (+162%) after server rotation because each server needs to collect the epoch messages across regions, costing additional WAN overhead and exacerbating the straggler problem.

Figure 12 compares Tiga's performance in the two settings with varying skew factors, represented as Tiga-Separate and Tiga-Colocate. Tiga-Separate incurs higher latency than Tiga-Colocate, as the skew factor (i.e., contention) increases. This is because Tiga-Separate involves more complexity to manage transactions; some transactions also require an additional WRTT to roll back when the execution results prove to be non-serializable. Even so, Tiga-Separate still achieves much lower latency than the other protocols.

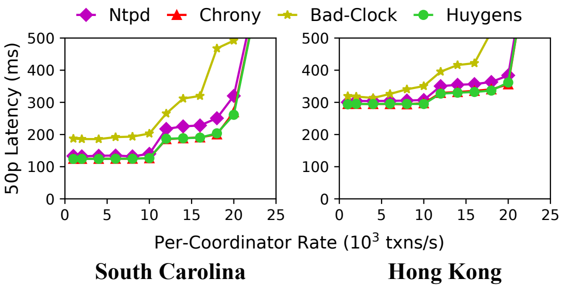
### 5.6 Sensitivity Analysis of Headroom

To evaluate the impact of headroom on Tiga's performance, we run MicroBench (skew factor=0.99) with leaders separated in different regions. Tiga continues to use the approach in §3.1 to estimate the headroom for transactions, but we further adjust the headroom by adding different offsets (Headroom Delta in Figure 13), ranging from −50 ms to 50 ms. We study Tiga's latency and rollback rate. As shown in Figure 13, Tiga's estimation approach (Headroom Delta=0 ms) yields a headroom that is close to optimal: Reducing headroom incurs more rollback and worse latency; increasing headroom eliminates rollback but still prolongs latency because transactions are held unnecessarily long at servers. We also evaluate a

**Table 3.** Throughput and clock synchronization errors with different clocks.

| | Ntpd | Chrony | Huygens | Bad-Clock |
|---|---|---|---|---|
| **Thpt ($10^3$ txns/s)** | 156.8 | 157.1 | 158.1 | 154.7 |
| **Clock errors (ms)** | 16.45 | 4.54 | 0.012 | 62.55 |

The stats of clock synchronization errors are collected by using Huygens' real-time monitor functionality [27].



**South Carolina** **Hong Kong**

**Figure 14.** Tiga latency using different clocks.

baseline that uses the sending time directly (0-Hdrm in Figure 13). This approach yields the worst latency and rollback rate, as it cannot tolerate network message reordering (illustrated in Figure 1), thereby highlighting the effectiveness of Tiga's headroom estimation based on synchronized clocks.

### 5.7 Tiga with Different Clocks

To understand the impact of synchronized clocks on Tiga's performance, we conduct an ablation study to compare Tiga's performance with different clocks. We use different synchronization algorithms and design the following variants.

(1) Tiga-Ntpd. We use `ntpd`, which manages time synchronization in most older Linux distributions in Google Cloud [43]. We configure `ntpd` to only use Google's internal NTP server as the reference clock.

(2) Tiga-Chrony. We use `chrony`, which is a newer implementation of NTP [24] as well as the current default NTP service in Google Cloud [44]. We configure `chrony` to only use Google's internal NTP server as the reference clock.

(3) Tiga-Huygens. We use the Huygens algorithm to synchronize the clocks for coordinators and servers.

(4) Tiga-Bad-Clock. We simulate the situation when the NTP service becomes unstable (e.g., due to network congestion and partition) by running a local NTP server as the reference clock. We keep periodically restarting and shutting down the NTP server. In this case, the clock synchronization becomes much worse than the previous variants (Table 3).

We run MicroBench (skew factor=0.99) to compare the performance of the four Tiga variants (Table 3 and Figure 14).

While `chrony` and Huygens yield different levels of synchronization errors (Table 3), Tiga's latency remains similar when equipped with either of them. This is because cross-region delays range from 60 ms to 150 ms; the synchronization error of `chrony`, though not as good as Huygens, is still negligible compared to the cross-region delay. As a result, both `chrony` and Huygens enable Tiga to accurately measure the one-way delay between coordinators and servers and decide a proper timestamp for the transaction at submission. By contrast, `ntpd`'s synchronization error is larger and causes extra holding time at servers due to the inaccurate measurement of one-way delay. In the worst case, when clocks are poorly synchronized (as in Tiga-Bad-Clock) and the error approaches the one-way delay between regions, Tiga 's latency inflates substantially.

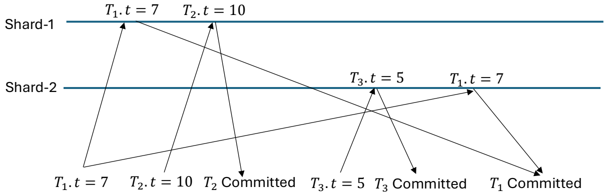


**Figure 15.** Undetectable timestamp inversion without inter-shard (leader) coordination.

## 6 Discussion

***Timestamp initialization.*** Tiga initializes transaction timestamps based on the maximum latency from the coordinator to a super quorum of servers in each shard (§3.1). This approach aims to increase the likelihood of fast-path commits. However, in certain deployments, the fast path may actually incur higher latency than the slow path. This situation arises when each shard has a simple quorum located close to the coordinator, while the remaining servers are geographically distant. In such scenarios, committing through the slow path may be more efficient. To accommodate this, the coordinator can estimate the latencies for both paths and then choose whether to use a super quorum or a simple quorum, based on which option can yield better performance.

***Dynamic sharding.*** Dynamic sharding [3, 8, 52, 82] allows OLTP systems to distribute heavy-hitter keys and co-locate frequently accessed data. We believe it could further enhance Tiga's performance, and we plan to support it in future versions. Because single-shard transactions do not require timestamp agreement, dynamic sharding can convert multi-shard transactions into single-shard transactions, thereby reducing the overhead of timestamp agreement and rollback.

***Clock accuracy and timestamp inversion.*** To prevent timestamp inversion and ensure strict serializability, Tiga introduces timestamp agreement (§3.5), which requires leaders of different shards to coordinate and confirm that their transactions respect real-time ordering. Without this inter-leader coordination, a shard cannot detect timestamp inversion when it occurs. Figure 15 illustrates such a case: Shard-1's servers run with faster clocks than Shard-2's. As a result, Shard-1 commits a single-shard transaction $T_2$ with a larger timestamp ($T_2.t = 10$), while Shard-2 later commits another single-shard transaction $T_3$ with a smaller timestamp ($T_3.t = 5$). Although $T_2$ and $T_3$ are processed independently, both conflict with the multi-shard transaction $T_1$. This yields a serializable schedule $T_2 \longrightarrow T_1 \longrightarrow T_3$, which contradicts the real-time order $T_2 \longrightarrow T_3$. Since all transactions arrive at their shards before their assigned timestamps, both shards treat them as valid, leaving the timestamp inversion undetected. To avoid such violations of strict serializability, the shards (leaders) must coordinate.

However, such inter-leader coordination incurs 0.5–1 RTT of blocking latency for subsequent transactions: if the transaction at the head of the priority queue has not completed timestamp agreement, any conflicting transactions behind it cannot be executed or released. This blocking latency can be costly when leaders are distributed across regions and workloads exhibit high contention. This raises a natural question: *Can we avoid coordination by leveraging synchronized clocks?*

In fact, if we could assume synchronized clocks with a bounded error $\epsilon$, Tiga can eliminate inter-leader coordination while still avoiding timestamp inversion. The coordination-free approach works as follows:

(1) Each leader updates an incoming transaction's timestamp to its local clock time if the initial timestamp is smaller.

(2) Each leader defers the release of the transaction $T$ until its local clock exceeds $T.t+\epsilon$, ensuring that all leaders' clocks have passed $T.t$ before $T$ is released.

Then, we revisit the example in Figure 15. Suppose the local clock time of Shard-1's leader is $clock_1$. Then the local clock time of Shard-2's leader $clock_2 \in [clock_1-\epsilon, clock_1+\epsilon]$. When Shard-1 receives $T_2$, it defers release until $clock_1 > T_2.t+\epsilon$, ensuring that every shard's clock has already passed $T_2.t$. Meanwhile, when Shard-2 receives $T_3$ and $T_1$, it updates their timestamps if they are smaller than $clock_2$. Two outcomes follow: (1) if $\epsilon \longrightarrow 0$, Shard-2 updates the timestamps for both $T_3$ and $T_1$ to values greater than $T_2.t = 10$, yielding the order $T_2 \longrightarrow T_3 \longrightarrow T_1$; (2) if $\epsilon \longrightarrow \infty$, Shard-1 defers $T_2$'s release until after Shard-2 releases $T_3$ and $T_1$, yielding the order $T_3 \longrightarrow T_1 \longrightarrow T_2$. In both cases, the serializable order remains consistent with the real-time order. Thus, the prior knowledge of $\epsilon$ provides a straightforward way to prevent timestamp inversion without inter-leader coordination, allowing more transactions to commit in 1 RTT.

We do not assume a deterministic error bound in Tiga's design due to the probabilistic nature of Huygens. Nonetheless, several clock synchronization systems provide deterministic guarantees. For example, Spanner [30] achieves millisecond-level error bounds, and Sundial [59] further reduces them to ~100 ns. While such synchronization requires specialized hardware, we expect these solutions to become increasingly deployable in the future, offering promising opportunities for Tiga to preserve strict serializability more efficiently.

**Table 4.** Summary of protocol comparison.

| System | Consistency | Aborts | WRTTs | | Require co-locating leaders for best latency? |
|---|---|---|---|---|---|
| | | | Best | Worst | |
| Spanner [30] | Strict Ser. | High | 3 | ≥4 | Required |
| AOCC [2] | Strict Ser. | High | 2 | ≥3 | Not Required |
| MVTO [81] | Ser. | Med | 2 | ≥3 | Not Required |
| MDCC [54] | Ser. | High | 2 | ≥3 | Required |
| Calvin [88] | Strict Ser. | None | 2 | 2.5 | Required |
| Tapir [95] | Ser. | High | 1 | ≥2 | Not Required |
| Janus [70] | Strict Ser. | None | 2 | 3 | Required |
| OceanVista [36] | Strict Ser. | None | 2 | 2.5 | Required |
| Natto [93] | Strict Ser. | Med | 2 | ≥3 | Not Required |
| Detock [74] | Strict Ser. | None | 2 | 2.5 | Required |
| NCC [63] | Strict Ser. | Med | 2 | ≥3 | Required |
| Mako [84] | Strict Ser. | Med | 2 | ≥5 | Required |
| Tiga | Strict Ser. | None | 1 | 2 | Not Required |

We discuss the commit latency for each system assuming no co-location requirement between coordinators and servers. For AOCC, MVTO and NCC, we assume they achieve geo-distributed fault tolerance via replication. Some systems incur $\geq x$ WRTTs because of aborts and retries.

## 7 Related Work

Table 4 compares Tiga with state-of-the-art protocols. While several existing protocols can achieve 1-WRTT commit latency, this optimal performance typically holds only under narrow conditions—such as co-locating servers and/or coordinators. Moreover, they often sacrifice correctness guarantees or incur costly aborts. In contrast, Tiga achieves 1-WRTT latency in more general deployments, and ensures strict serializability with few or no transaction aborts.[2]

***Ordering guarantees in multicast.*** Several network primitives have been proposed to accelerate distributed protocols. Ordered Unreliable Multicast (OUM) [58] and Multi-Sequencing Groupcast (MSG) [57] both leverage a single sequencer to establish ordering, which can incur centralized bottlenecks with a software-based sequencer in cloud settings. Hydra [23] extends OUM and MSG with multiple sequencers. However, it requires all sequencers to continually send flush messages to receivers. The slowdown of any sequencer can impede the progress of all receivers. Tiga's design is inspired by the deadline-ordered multicast (DOM) primitive of Nezha [41], but DOM does not consider inter-shard timestamp agreement of transactions. Moreover, Nezha, as a pure consensus protocol, cannot be easily extended to work in the multi-shard setting that Tiga targets.

***Tiga vs. Mako.*** The recent protocol, Mako [84], advocates for decoupling consensus and concurrency control to improve throughput. In contrast, Tiga prioritizes latency optimization. Accordingly, we argue that a consolidated design is better suited for minimizing transaction latency. In geo-distributed settings, Mako needs multiple WRTTs to commit transactions when they are issued from followers, or when leaders are not co-located. In contrast, Tiga can consistently commit transactions in the 1-WRTT fast path.

[2] Tiga is abort-free for one-shot transactions when leaders are co-located.

# 8 Conclusion

The rapid advancement of accurate clock synchronization enables new protocols that exploit timestamp ordering to accelerate geo-distributed transaction processing. In this paper, we have presented the design, implementation, and evaluation of Tiga, a consolidated protocol that uses synchronized clocks to proactively order transactions at predesignated timestamps and efficiently resolve inconsistencies among servers. Compared with conventional layered designs (e.g., 2PL/OCC+Paxos, Calvin+, Detock, NCC) and state-of-the-art consolidated designs (e.g., Tapir and Janus), Tiga can achieve significantly higher throughput and lower latency.

This work does not raise any ethical issues.

# Acknowledgments

We thank the anonymous reviewers for suggestions that improved our work. This project was funded in part by NSF awards CNS-2321725, CNS-2238768, CNS-2130590 and NSF CAREER award 2340748. We also appreciate the support from Google Cloud Research Credits program.

# References

[1] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Technical Report. USA. https://hdl.handle.net/1721.1/149899

[2] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. 1995. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. *SIGMOD Record* 24, 2 (May 1995), 23–34. https://doi.org/10.1145/568271.223787

[3] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. 2016. Slicer: Auto-Sharding for Datacenter Applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016).* USENIX Association, Savannah, GA, 739–753. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/adya

[4] Saif Al-Kuwari, James H. Davenport, and Russell J. Bradford. 2011. Cryptographic Hash Functions: Recent Design Trends and Security Notions. (2011). https://eprint.iacr.org/2011/565.pdf

[5] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *2008 IEEE 24th International Conference on Data Engineering.* 576–585. https://doi.org/10.1109/ICDE.2008.4497466

[6] Andy Pavlo and et al. [n. d.]. AuctionMark Benchmark. https://hstore.cs.brown.edu/projects/auctionmark. ([n. d.]).

[7] Andy Pavlo and et al. [n. d.]. SEATS Benchmark. https://github.com/apavlo/h-store/tree/master/src/benchmarks/edu/brown/benchmark/seats/. ([n. d.]).

[8] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. 2018. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018).* USENIX Association, Carlsbad, CA, 445–460. https://www.usenix.org/conference/osdi18/presentation/annamalai

[9] Apache Software Foundation. 2021. ZooKeeper. https://zookeeper.apache.org. (2021). Accessed: 2025-08-31.

[10] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13).* Association for Computing Machinery, New York, NY, USA, 1185–1196. https://doi.org/10.1145/2463676.2465296

[11] AWS. 2019. Global Tables: Multi-Region Replication for DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GlobalTables.html. (2019). Accessed: 2025-08-31.

[12] AWS. 2024. Amazon Time Sync Service expands Microsecond-Accurate time to 87 additonal EC2 instance types. https://aws.amazon.com/about-aws/whats-new/2024/04/amazon-time-sync-service-microsecond-accurate-time-additonal-ec2-instance-types/. (2024). Accessed: 08/31/2024.

[13] Baidu. 2022. Server Push. https://brpc.apache.org/docs/server/server-push/. (2022). Accessed: 2025-08-31.

[14] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. 1994. Incremental Cryptography: The Case of Hashing and Signing. In *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '94).* Springer-Verlag, Berlin, Heidelberg, 216–233.

[15] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. 1995. Incremental Cryptography and Application to Virus Protection. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC 1995).*

[16] Mihir Bellare, Roch Guérin, and Phillip Rogaway. 1995. XOR MACs: New Methods for Message Authentication Using Finite Pseudorandom Functions. In *Proceedings of the Annual International Cryptology Conference (CRYPTO 1995).*

[17] Mihir Bellare and Phillip Rogaway. 1997. Collision-Resistant Hashing: Towards Making UOWHFs Practical. In *Proceedings of the Annual International Cryptology Conference (CRYPTO 1997).*

[18] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, USA.

[19] Philip A. Bernstein and Eric Newcomer. 2009. Chapter 6: Locking. In *Principles of Transaction Processing (Second Edition).* Morgan Kaufmann, Burlington, MA, USA.

[20] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable Isolation for Snapshot Databases. *ACM Trans. Database Syst.* 34, 4, Article 20 (dec 2009), 42 pages. https://doi.org/10.1145/1620585.1620587

[21] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving Low Tail-Latency and High Scalability for Serializable Transactions in Edge Computing. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys 2021).* 1–16. https://doi.org/10.1145/3447786.3456238

[22] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. 2022. TAOBench: An End-to-End Benchmark for Social Network Workloads. *Proc. VLDB Endow.* 15, 9 (jul 2022), 1965–1977. https://doi.org/10.14778/3538598.3538616

[23] Inho Choi, Ellis Michael, Yunfan Li, Dan Ports, and Jialin Li. 2023. Hydra: Serialization-Free Network Ordering for Strongly Consistent Distributed Applications. In *Proceedings of the 20th USENIX Conference on Networked Systems Design and Implementation (NSDI 2023).* 1–16. https://www.usenix.org/conference/nsdi23/presentation/choi

[24] Chrony Team. 2024. Chrony. https://chrony-project.org/index.html. (2024). Accessed: 09/11/2024.

[25] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. 2003. Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking. In *Advances in Cryptology – Proceedings of CRYPTO 2003.* 1–18.

[26] Clockwork.io. 2022. Cloud Clocksync Showdown: Ntpd vs Chrony vs Clockwork. https://www.clockwork.io/cloud-clocksync-showdown-

ntpd-vs-chrony-vs-clockwork/. (2022).

[27] Clockwork.io. 2024. Clockwork Latency Sensei. https://www.clockwork.io/latency-sensei/. (2024). Accessed: 09/11/2024.

[28] Clockwork.io. 2024. Why One-Way Latency Measures Are Critical for Distributed Databases, Microservices, and AI Workloads. https://www.clockwork.io/why-one-way-latency-measures-are-critical-for-distributed-databases-microservices-and-ai-workloads/. (2024). Accessed: 08/31/2025.

[29] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment* 1, 2 (August 2008), 1277–1288. https://doi.org/10.14778/1454159.1454167

[30] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, and et al. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012)*. 251–264. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[31] Transaction Processing Performance Council. 2022. TPC-C. https://www.tpc.org/tpcc/. (2022). Accessed: 08/31/2025.

[32] Volt Active Data. 2025. How VoltDB Works. https://docs.voltdb.com/UsingVoltDB/IntroHowVoltDBWorks.php. (2025). Accessed: 08/31/2025.

[33] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (dec 2013), 277–288. https://doi.org/10.14778/2732240.2732246

[34] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633. https://doi.org/10.1145/360363.360369

[35] European Union. 2018. GDPR Personal Data – What Information Does This Cover? https://www.gdpreu.org/the-regulation/key-concepts/personal-data/. (2018). Accessed: 08/31/2025.

[36] Hua Fan and Wojciech Golab. 2019. Ocean Vista: Gossip-Based Visibility Control for Speedy Geo-Distributed Transactions. *Proceedings of the VLDB Endowment* 12, 6 (2019), 1471–1484. https://doi.org/10.14778/3342263.3342627

[37] Marc Fischlin. 1997. Incremental Cryptography and Memory Checkers. In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT 1997)*. 275–291.

[38] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. Strong and Efficient Consistency with Consistency-Aware Durability. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST 2020)*. 1–16. https://doi.org/10.1145/3423138

[39] Jinkun Geng. 2025. TLA+ Specification of Tiga. https://github.com/New-Consensus-Concurrency-Control/Tiga-TLA-plus. (2025).

[40] Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. 2022. Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks [Technical Report]. (2022). https://arxiv.org/abs/2206.03285

[41] Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. 2023. Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks. *Proceedings of the VLDB Endowment* 16 (2023), 629–642. https://doi.org/10.14778/3574245.3574250

[42] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 81–94. https://www.usenix.org/conference/nsdi18/presentation/geng

[43] Google. 2025. Configure NTP on a VM. https://cloud.google.com/compute/docs/instances/configure-ntp#linux-ntpd. (2025). Accessed: 2025-08-31.

[44] Google. 2025. Configure NTP on a VM (Chrony). https://cloud.google.com/compute/docs/instances/configure-ntp#linux-chrony. (2025). Accessed: 2025-08-31.

[45] Jim Gray and Leslie Lamport. 2006. Consensus on Transaction Commit. *ACM Transactions on Database Systems* 31, 1 (March 2006), 133–160. https://doi.org/10.1145/1132863.1132867

[46] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. *Proceedings of the International Conference on Management of Data (SIGMOD 1994)* (1994), 243–252. https://doi.org/10.1145/191839.191886

[47] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[48] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. 2023. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. *Proceedings of the VLDB Endowment* 17, 3 (November 2023), 469–482. https://doi.org/10.14778/3632093.3632109

[49] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[50] IBM Software Group. [n. d.]. Telecommunication Application Transaction Processing (TATP) Benchmark Description. https://tatpbenchmark.sourceforge.net/TATP_Description.pdf. ([n. d.]).

[51] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, and Yang Zhang. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[52] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. 2021. Zeus: Locality-Aware Distributed Transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys 2021) (EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 145–161. https://doi.org/10.1145/3447786.3456234

[53] Jan Kończak, Paweł T. Wojciechowski, Nuno Santos, Tomasz Żurkowski, and André Schiper. 2021. Recovery Algorithms for Paxos-Based State Machine Replication. *IEEE Transactions on Dependable and Secure Computing* 18, 4 (July–August 2021), 1234–1247. https://doi.org/10.1109/TDSC.2021.3051234

[54] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys 2013) (EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2465351.2465363

[55] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (2001), 51–58. https://doi.org/10.1145/568425.568433

[56] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19, 2 (October 2006), 79–103. https://doi.org/10.1007/s00446-006-0016-x

[57] Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*. ACM, Shanghai, China, 17. https://doi.org/10.1145/3132747.3132751

[58] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*.

USENIX Association, Savannah, GA, 395–410. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li

[59] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. 2020. Sundial: Fault-Tolerant Clock Synchronization for Datacenters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*. USENIX Association, Santa Clara, CA, 611–630. https://www.usenix.org/conference/osdi20/presentation/li

[60] Barbara Liskov. 1991. Practical Uses of Synchronized Clocks in Distributed Systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*. https://doi.org/10.1145/112600.112601

[61] Barbara Liskov and James Cowling. 2012. Viewstamped replication revisited. (2012).

[62] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, Cascais, Portugal, 401–416. https://doi.org/10.1145/2043556.2043593

[63] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. 2023. NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2023)*. USENIX Association, Santa Clara, CA, 821–839. https://www.usenix.org/conference/osdi23/presentation/lu

[64] Meta. 2021. ZippyDB: Facebook's Key-Value Store. https://engineering.fb.com/2021/08/06/core-infra/zippydb/. (2021). Accessed: 2025-08-31.

[65] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. 2017. *Recovering Shared Objects Without Stable Storage [Extended Version]*. Technical Report. University of Washington. https://www.microsoft.com/en-us/research/publication/recovering-shared-objects-without-stable-storage-extended-version/ Accessed: 2025-08-31.

[66] Microsoft. 2022. Global Data Distribution with Azure Cosmos DB — Under the Hood. https://docs.microsoft.com/en-us/azure/cosmos-db/global-dist-under-the-hood. (2022). Accessed: 2025-08-31.

[67] Microsoft. 2025. Partitioning and Horizontal Scaling in Azure Cosmos DB. https://learn.microsoft.com/en-us/azure/cosmos-db/partitioning-overview. (2025). Accessed: 2025-08-31.

[68] D. L. Mills. 1991. Internet Time Synchronization: The Network Time Protocol. *IEEE Transactions on Communications* 39, 10 (1991), 1482–1493. https://doi.org/10.1109/26.103043

[69] Shuai Mu and et al. 2016. Janus Repo. https://github.com/NYU-NEWS/janus. (2016). Accessed: 2025-08-31.

[70] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*. USENIX Association, Savannah, GA, 409–425. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/mu

[71] Ali Najafi and Michael Wei. 2022. Graham: Synchronizing Clocks by Leveraging Local Clock Properties. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2022)*. USENIX Association, Renton, WA, USA, 453–466. https://www.usenix.org/conference/nsdi22/presentation/najafi

[72] National Congress of the People's Republic of China. 2021. Data Security Law of the People's Republic of China. https://digichina.stanford.edu/work/translation-data-security-law-of-the-peoples-republic-of-china/. (2021). Accessed: 2025-08-31.

[73] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. 2015. Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores. In *Proceedings of the 33rd ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, 1279–1294. https://doi.org/10.1145/2723372.2723729

[74] Cuong D. T. Nguyen, Johann K. Miller, and Daniel J. Abadi. 2023. Detock: High Performance Multi-Region Transactions at Scale. *Proc. ACM Manag. Data* 1, 2, Article 148 (June 2023), 27 pages. https://doi.org/10.1145/3589293

[75] ObjectDB Software Ltd. [n. d.]. JPA Performance Benchmark (JPAB). https://www.jpab.org/Benchmark_FAQ.html. ([n. d.]).

[76] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 8–17. https://doi.org/10.1145/62546.62549

[77] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[78] Seo Jin Park and John Ousterhout. 2019. Exploiting Commutativity for Practical Fast Replication. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2019)*. USENIX Association, Boston, MA, USA, 183–198. https://www.usenix.org/conference/nsdi19/presentation/park

[79] PingCap. 2024. Three Availability Zones in Two Regions Deployment. https://docs.pingcap.com/tidb/stable/multi-data-centers-in-one-city-deployment. (2024). Accessed: 2025-08-31.

[80] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015) (NSDI '15)*. USENIX Association, Oakland, CA, USA, 43–57. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ports

[81] David P. Reed. 1983. Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems* 1, 1 (February 1983), 3–23. https://doi.org/10.1145/357353.357355

[82] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-Latency, Geo-Replicated Transactions. *Proc. VLDB Endow.* 12, 11 (July 2019), 1747–1761. https://doi.org/10.14778/3342263.3342647

[83] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis. 1978. System-Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems* 3, 2 (1978), 178–198. https://doi.org/10.1145/320080.320083

[84] Weihai Shen, Yang Cui, Siddhartha Sen, Sebastian Angel, and Shuai Mu. 2025. Mako: Speculative Distributed Transactions with Geo-Replication. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2025)*. USENIX Association, Santa Clara, CA, USA, 1–16. https://www.usenix.org/conference/osdi25/presentation/shen-weihai

[85] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[86] Robert H. Thomas. 1979. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems* 4, 2 (June 1979), 180–209. https://doi.org/10.1145/320071.320076

[87] Alexander Thomson and Daniel J. Abadi. 2010. The case for determinism in database systems. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 70–80. https://doi.org/10.14778/1920841.1920855

[88] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[89] Sarah Tollman, Seo Jin Park, and John Ousterhout. 2021. EPaxos Revisited. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2021)*. https://www.usenix.org/conference/nsdi21/presentation/tollman

[90] VoltDB. [n. d.]. Voter Benchmark. https://github.com/VoltDB/voltdb/tree/master/examples/voter. ([n. d.]).

[91] Yahoo! [n. d.]. YCSB Workload. https://github.com/brianfrankcooper/YCSB/tree/master/workloads. ([n. d.]).

[92] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. https://dl.acm.org/doi/10.1145/3183713.3196924

[93] Linguan Yang, Xinan Yan, and Bernard Wong. 2022. Natto: Providing Distributed Transaction Prioritization for High-Contention Workloads. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/3514221.3526161

[94] YugabyteDB. 2025. Isolation Levels. (2025). https://docs.yugabyte.com/preview/explore/transactions/isolation-levels/ Accessed: 2025-08-31.

[95] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP 2015)*. https://doi.org/10.1145/2815400.2815404

[96] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_wenting

[97] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, et al. 2021. FoundationDB: A Distributed Unbundled Transactional Key-Value Store. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/3448016.3457559

# Appendices

In this appendix, we include the following:

- Appendix A explains more details of Tiga's protocol in normal transaction processing and includes a more comprehensive version of pseudo-code.
- Appendix B discusses how Tiga handles different failures. Besides, it describes the detailed procedures (in pseudo-code) to replace leaders, recover logs and let failed servers rejoin the system.
- Appendix C presents the proof of Tiga's correctness properties (per-shard linearizability and strict serializability).
- Appendix D explains the details about how Tiga uses incremental hash to facilitate the quorum check.
- Appendix E describes Tiga's optimization of slow path.
- Appendix F illustrates how Tiga decomposes dependent transactions into one-shot transactions, so that dependent transactions can be processed by Tiga.
- The TLA+ specification of Tiga is available at https://github.com/New-Consensus-Concurrency-Control/Tiga-TLA-plus.

## A Normal Processing in Tiga

In §3, we have included a simplified version of the pseudo-code (Algorithm 1) to describe the actions at servers to process transactions. Here, we continue to explain more details of Tiga's protocol design and implementation. Algorithm 2 and Algorithm 3 describe the actions at servers and coordinators in more detailed pseudo-code. In the algorithm description, we omit the details of implementing message retransmission and reliable delivery (e.g., SEND-MESSAGE).

In general, Tiga extends the view-based approach [61, 76]. In each shard, every server maintains a local view identified by *l-view*, and a global view identified by *g-view*. In Tiga's implementation, *g-view* is associated with an $m$-length vector *g-vec*, recording the *l-view* of each shard under this global view. More specifically, $shard_i$'s *l-view* equals *g-vec*$[i]$.

***View manager.*** Tiga uses a stand-alone view manager to manage all the view information for Tiga's servers and coordinators. The view manager maintains a simple replicated state machine (i.e., <*g-view, g-vec*> ) that is resilient to failures. The view manager can be built with ZooKeeper [9], or backed by canonical consensus protocols like Paxos [55], Raft [77], and Viewstamped Replication [61, 76]. In our implementation, we implement the view manager by using Viewstamped Replication protocol. Besides, we also equip Viewstamped Replication with the crash vector technique [65], so that Viewstamped Replication can maintain its state machine in the memory, without writing disks every time it updates <*g-view, g-vec*>. The view manager initiates global view change for Tiga's servers in case of server failure or reconfiguration. After every global view change, each server's *g-view* is incremented by 1; their *l-views* may remain unchanged or be incremented by at least 1. Note that the view manager is off the critical path of transaction processing in the common cases so its performance has no significant impact.

***Server action.*** During normal processing, all servers share the same *g-view*, as well as the same *g-vec* because *g-vec* is associated with *g-view*. Besides, servers in the same shard also share the same *l-view*, which decides the leader's *replica-id* in this shard (*replica-id* = *l-view*%$(2f+1)$), but servers in different shards may have different *l-views*. (1) Inside each shard, servers communicate with the others under the same local view (e.g., the leader calls SEND-LOG-SYNC in Algorithm 2 to communicate with its followers); (2) Across shards, only leaders communicate with each other under the same global view (e.g., the leader calls SEND-TIMESTAMP-NOTIFICATION in Algorithm 2 to communicate with the other leaders).

***Coordinator action.*** The coordinator needs to collect fast-replies/slow-replies from servers to conduct quorum check for its submitted transactions. Initially, the coordinator needs to contact the view manager to acquire the view information <*g-view, g-vec*>. Then, the coordinator only accepts the reply messages with the matching *g-view* and *l-view* (line 9-10 in Algorithm 3).

<*g-view, l-view*> is the key to guarantee both serializability across multiple shards and the linearizability in each shard. To commit a transaction, the coordinator must receive sufficient replies from every participating shard, and (1) all replies should have the same *g-view*; (2) replies belonging to the same shard should also have the same *l-view*.

## B Failure Recovery in Tiga

***Message drop.*** Message drop does not affect Tiga's correctness. Message drop triggers the timeout at the coordinator, which resubmits the transaction with the same identifier but different timestamps. Tiga servers maintain at-most-once semantics to avoid duplicate execution of the same transaction.

***Coordinator failure.*** Coordinator failure does not affect Tiga's correctness. Whenever the coordinator crashes and is rebooted, the coordinator first contacts the view manager to acquire the fresh view information, i.e., <*g-view, g-vec*>, and them resume its service.

One case to note is that the coordinator can fail in the middle of multicasting transactions to multiple servers. In this case, the timestamp agreement (② in Figure 3) can be blocked because some leaders receive the transaction whereas the other leaders not. To handle such case, each leader will launch a timer after it has received the first notification message from the other leaders but it has not received the corresponding transaction from the coordinator. Then when it is timeout and the transaction does not come, the leader will fetch the missing transaction from the other leaders which have sent it the notification message (because these leaders must have received the transaction).

**Algorithm 2** Server action

```
Local State:                                          ▷
s,                                  ▷ the server's shard-id
r,                                  ▷ the server's replica-id,
                                    ▷ <s,r> uniquely identify the server
l-view,                             ▷ the local view of this server
g-view,                             ▷ the global view of this server
pq,                                 ▷ the server's priority queue
g-vec,                              ▷ g-vec records the local view of the each shard
sync-point,                         ▷ the point to which the server's log is consistent
                                    ▷ with the leader's log
upon receiving txn T do
    $T'$ ← last released txn that conflicts with $T$
    if CONFLICT-DETECTION($T$) then $pq$.insert($T$)
    else if AM-LEADER() then        ▷ Only leaders can update $T.t$
        $T.t$ ← CLOCK-TIME()
        $pq$.insert($T$)
▷ Periodically check the clock time
upon clock time progressing do
    $nowTime$ ← CLOCK-TIME()
    $releaseTxns$ ← []
    ▷ Enumerate txns based on timestamp order
    for $T \in pq$ do
        if $T.t > nowTime$ then break        ▷ $T$ has not expired
        if $\nexists T' \in pq : T'.t < T.t$ and $T'$ conflicts with $T$ then
            $releaseTxns$.append($T$)
    for $T \in releaseTxns$ do
        $\forall$ $key \in T.readSet$, $rMap[key] \leftarrow T.t$
        $\forall$ $key \in T.writeSet$, $wMap[key] \leftarrow T.t$ ▷ Conflict detection
        if AM-LEADER() then
            $ret$ ← EXECUTE($T$)        ▷ Only leaders execute $T$
            $hash$ = CALCULATE-HASH($log$)
            SEND-FAST-REPLY($T$, $hash$, $ret$)
            ▷ $tSet$ contains $T$'s timestamps used by each leader
            $tSet$ ←TIMESTAMP-AGREEMENT($T$)
            if $T.t = \max\{t : t \in tSet\}$ then
            ▷ This leader used correct (max) timestamp
                if $tSet$.size()>1 then
                ▷ Some leaders used incorrect $T.t$
                ▷ After completing second round, leaders agree
                  on $T$'s timestamp, i.e., $T.t$
                    TIMESTAMP-AGREEMENT($T$)
                Append $T$ to $log$ and syncs $T.t$ with followers
                    $pq$.erase($T$)
                else        ▷ This leader used smaller timestamp
                    $T.t \leftarrow \max\{t : t \in tSet\}$
                    $pq$.reposition($T$)
        else        ▷ Follower sends fast-reply without execution
            SEND-FAST-REPLY($T$, $hash$, $null$)
            $pq$.erase($T$)
▷ Only leaders send/receive TIMESTAMP-NOTIFICATION to/from
each other
upon leader's receiving TIMESTAMP-NOTIFICATION, $msg$ do
    if $m.g\text{-}view \neq g\text{-}view$ then
        return ▷ $msg$ does not come from the same global view
    if $m.r \neq g\text{-}vec[msg.s]\%(2f+1)$ then
        return ▷ $msg$ does not come from the leader of $shard_s$
    Choose $T \in pq : T.txn\text{-}id = msg.txn\text{-}id$
    ▷ Maintain a quorum set for each txn
    $T.timestampQ \leftarrow T.timestampQ \cup \{msg\}$
    if $T.timestampQ$.size()=$T.shards$.size() then
    ▷ All involved leaders' timestamps have been collected
        $T.t \leftarrow \max\{mm.t : mm \in T.t\}$
        ▷ The updated timestamp is commonly agreed
▷ Only followers receive LOG-SYNC
upon FOLLOWER'S RECEIVING LOG-SYNC, $msg$ do
    Modify $log$ to keep consistent with leader's $log$
    Update $sync\text{-}point$
    SEND-SLOW-REPLY($T$)
function TIMESTAMP-AGREEMENT($T$)
    if $T.timestampQ$.size()<$T.shards$.size() then
    ▷ This is the first round of timestamp agreement
        BROADCAST-TIMESTAMP-NOTIFICATION($T$)
        Wait until $T.timestampQ$.size()=$T.shards$.size()
        $tSet \leftarrow \{mm.t | mm \in T.timestampQ\}$
        return $tSet$
    else        ▷ We have previously collected $T$'s timestamps
    ▷ This is the second round of timestamp agreement,
    because the first round detects inconsistent timestamps
        $t_{agreed} \leftarrow \max\{mm.t : mm \in T.timestampQ\}$
        $T.timestampQ \leftarrow \{mm : mm \in T.timestampQ$
                                  and $mm.t = t_{agreed}\}$
        BROADCAST-TIMESTAMP-NOTIFICATION($T$)
        Wait until $T.timestampQ$.size()=$T.shards$.size()
        return $tSet$
```

***Server failure.*** There are two types of server failure in Tiga, i.e., follower failure and leader failure. Follower failure does not interrupt the service availability, but it may cause the transactions unable to be committed in the fast path, because some shards may have insufficient servers to establish a super quorum. Leader failure is more serious and it makes the shard fail to commit any transactions until (1) a new leader is elected from the followers in the same shard and (2) the new leader has recovered and executed the committed transactions before server failure.

***Leader election and log recovery.*** Tiga uses the standalone view manager to detect server failure. We implement our view manager by using Viewstamped Replication protocol [61, 65]. Algorithm 4 describes the action of view manager to detect Tiga server failure and issue global view change among Tiga servers. Algorithm 5 describes the servers' action during the global view change (i.e., selecting new leaders and recovering logs). Here we only focus on how the view manager supports the failure recovery of Tiga servers, and omit the details on how Viewstamped Replication supports the failure recovery of our view manager. For example, when the VR protocol [61] is running in diskless mode, its correctness can be damaged by the *stray messages* (i.e., messages

```
function BROADCAST-TIMESTAMP-NOTIFICATION($T$)
    $msg.type \leftarrow$ TIMESTAMP-NOTIFICATION
    $msg.g\text{-}view \leftarrow g\text{-}view$
    $msg.l\text{-}view \leftarrow l\text{-}view$
    $msg.s \leftarrow s$
    $msg.r \leftarrow r$
    $msg.txn\text{-}id \leftarrow T.txn\text{-}id$
    $msg.t \leftarrow T.t$
    ▷ Notify the leaders of each shard that executes $T$
    for $ss \in T.shards$ do
        $rr \leftarrow g\text{-}vec[ss]\%(2f+1)$
        ▷ <$ss, rr$> is the leader in $shard_{ss}$
        SEND-MESSAGE($msg$, <$ss, rr$>)        ▷ Send $msg$ to <$ss, rr$>
function SEND-FAST-REPLY($T$, $hash$,$result$)
    $msg.type \leftarrow$ FAST-REPLY
    $msg.g\text{-}view \leftarrow g\text{-}view$
    $msg.l\text{-}view \leftarrow l\text{-}view$
    $msg.s \leftarrow s$
    $msg.r \leftarrow r$
    $msg.txn\text{-}id \leftarrow T.txn\text{-}id$
    $msg.t \leftarrow T.t$
    $msg.hash \leftarrow hash$
    $msg.ret \leftarrow result$
    ▷ Send the reply to the coordinator (indexed by $T.coord\text{-}id$) which mutlicasts the txn
    SEND-MESSAGE($msg$, $T.coord\text{-}id$)
function SEND-LOG-SYNC($T$)
    $msg.type \leftarrow$ LOG-SYNC
    $msg.l\text{-}view \leftarrow l\text{-}view$
    $msg.s \leftarrow s$
    $msg.r \leftarrow r$
    $msg.log\text{-}pos \leftarrow$ log position of $T$ in the leader's log list
    $msg.t \leftarrow T.t$
    $msg.txn\text{-}id \leftarrow T.txn\text{-}id$
    for $rr \leftarrow 0$ to $2f$ do
        if $rr = r$ then
            continue
        ▷ <$s, rr$> is a follower in $shard_s$
        SEND-MESSAGE($msg$, <$s, rr$>)
function SEND-SLOW-REPLY($T$)
    $msg.type \leftarrow$ SLOW-REPLY
    $msg.g\text{-}view \leftarrow g\text{-}view$
    $msg.l\text{-}view \leftarrow l\text{-}view$
    $msg.s \leftarrow s$
    $msg.r \leftarrow r$
    $msg.txn\text{-}id \leftarrow T.txn\text{-}id$
    $msg.t \leftarrow T.t$
    $msg.sync\text{-}point \leftarrow sync\text{-}point$
    SEND-MESSAGE($msg$, $T.coord\text{-}id$)
```

**Algorithm 3** Coordinator action

```
upon receiving a txn $T$ from client do
    $T.shards \leftarrow$ the $shard\text{-}id$s of the servers that execute $T$
    $T.coord\text{-}id \leftarrow coord\text{-}id$
    ▷ Refer to §3.1 for initializing the timestamp
    $T.t \leftarrow$ CLOCK-TIME()+$estimated\text{-}delay$
    for $s \in T.shards$ do
        for $r \leftarrow 0$ to $2f$ do
            SEND-MESSAGE($T$, <$s, r$>)
▷ Periodically, coordinator contacts view manager (§4) to get the current global view $g\text{-}view$, and the vector $g\text{-}vec$, which records each shard's local view.
upon receiving FAST-REPLY or SLOW-REPLY, $msg$ do
    if $msg.g\text{-}view \neq g\text{-}view$
     or $msg.l\text{-}view \neq g\text{-}vec[msg.shardId]$ then
        return            ▷ Views mismatch, ignore the reply
    $replySet \leftarrow replySet \cup \{msg\}$
    $quorum \leftarrow \{mm \in replySet : mm.txn\text{-}id = T.txn\text{-}id\}$
    $leaderReplies \leftarrow \{\}$
    for each $shard_s$ that executes $T$ do
        if $quorum$ has no $shard_s$'s leader's fast reply then
            return                 ▷ $T$ is not committed
        Choose $mm \in quorum$ : $mm$ is $shard_s$'s leader's fast reply
        $leaderReplies \leftarrow leaderReplies \cup \{mm\}$
        $fastQ, slowQ \leftarrow 0, 0$
        for $r \leftarrow 0$ to $2f$ do
            if $quorum$ contains server <$s,r$>'s slow-reply then
                $slowQ \leftarrow slowQ + 1$
                ▷ Slow-reply can be used as fast-reply
                $fastQ \leftarrow fastQ + 1$
            else if $quorum$ has server <$s,r$>'s fast-reply and the fast-reply has the same hash as $mm.hash$ then
                $fastQ \leftarrow fastQ + 1$
        if $fastQ < 1 + f + \lceil f/2 \rceil$ and $slowQ < f$ then
            return                 ▷ $T$ is not committed
    $tSet \leftarrow \{msg.t : msg \in leaderReplies\}$
    if $tSet$.size()>1 then ▷ Leaders used different timestamps
        ▷ Filter out those replies with incorrect (smaller) timestamp
        $invalidSet \leftarrow \{msg \in quorum : msg.t < \max tSet\}$
        $replySet \leftarrow replySet - invalidSet$
        return                     ▷ $T$ is not committed.
    $results \leftarrow \{mm.ret : mm \in leaderReplies\}$
    Deliver $results$ to the client          ▷ $T$ is committed
```

that are sent by some servers before crash but are forgotten by the servers after they recover) [53, 65] in the system. To tackle this, VR needs to be equipped with the crash vector technique [65] to preserve correctness. Here we do not discuss how to use crash vector to preserve VR correctness, and readers can refer to [40, 65] for more details of crash vector application.

***Server rejoins.*** A failed server can rejoin the its shard as a follower. After running the rejoining procedure (described in Algorithm 5), a server switches from RECOVERING status to NORMAL status with the proper log list, and then work as a follower to continue processing transactions.

**Algorithm 4** View manager (VMgr) action during global view change

```
Local State:                                          ▷
v-view,                                   ▷ the view-id of VMgr
v-rid,                        ▷ the replica-id of the VMgr replica
g-view,                  ▷ the current global view of Tiga servers
g-vec,                         ▷ the current g-vec of Tiga servers
g-mode,      ▷ the current mode for cross-shard coordination,
                          i.e., preventive or detective (§3.8)
prepare-g-view,        ▷ the prepared global view to be committed
prepare-g-vec,              ▷ the prepared g-vec to be committed
prepare-mode,  ▷ the prepared mode for cross-shard coordination

  ▷ VMgr leader detects whether any leader servers in Tiga fails
upon VMgr leader's failing to hear the heartbeat of some
        leader server(s) in Tiga do
    ▷ Before issuing global view change, VMgr leader first persists
    the tentative g-view and g-vec
    prepare-g-view ← g-view + 1
    new-leader-rids ← FIND-NEW-LEADERS()
    ▷ Decide the new l-views based on the newly selected
    leaders
    for $r \leftarrow 0$ to $2f$ do
        $r_{old} \leftarrow$ g-vec[$r$]%($2f+1$)          ▷ Old leader's replica-id
        $r_{new} \leftarrow$ new-leader-rids[$r$]       ▷ New leader's replica-id
        prepare-g-vec[r] ← g-vec[$r$] + ($r_{new} - r_{old}$)%($2f+1$)
        ▷ Preventive mode is adopted when leaders can be
         co-located, and detective mode is used otherwise.
        prepare-mode ← PREVENTIVE or DETECTIVE
    ▷ Initialize a quorum set to receive PREPARE-REPLYS
    prepareQuorum ← {}
    ▷ Send prepare message to VMgr followers
    SEND-CM-PREPARE()
upon receiving CM-PREPARE, msg do
    if msg.v-view ≠ v-view then
        return ▷ Only process messages from the same v-view
    prepare-g-view ← m.prepare-g-view
    prepare-g-vec ← m.prepare-g-vec
    SEND-CM-PREPARE-REPLY(msg.v-rid)
upon receiving CM-PREPARE-REPLY, msg do
    if msg.v-view ≠ v-view then
        return                         ▷ Views mismatch, ignore msg
    if msg.prepare-g-view ≠ prepare-g-view
       or msg.prepare-g-vec ≠ prepare-g-vec then
        return
    prepareQuorum ← prepareQuorum ∪ {m}
    if prepareQuorum.size() ≥ $f + 1$ then
        ▷ The prepared info has been persisted and can be used
        g-view ← prepare-g-view
        g-vec ← prepare-g-vec
        g-mode ← prepare-g-mode
        ▷ Broadcast VIEW-CHANGE-REQ to every Tiga server
        SEND-VIEW-CHANGE-REQ()
        ▷ In parallel, VMgr leader also asks each VMgr follower to
        commit the new <g-view,g-vec>
        SEND-CM-COMMIT()
upon receiving CM-COMMIT, msg do
    if msg.v-view ≠ v-view then
        return
    if msg.g-view ≠ prepare-g-view
       or msg.g-vec ≠ prepare-g-vec then
        return
    ▷ VMgr follower commits the new state: <g-view, g-vec>
    g-view ← prepare-g-view
    g-vec ← prepare-g-vec
    g-mode ← prepare-g-mode
function FIND-NEW-LEADERS()
    for $r \leftarrow 0$ to $2f$ do
        if $\forall s \in [0, m)$: server <$s, r$> is alive then
            ▷ Servers <*, r> are co-located in the same region; we
            can choose these servers as new leaders for their shards
            new-leaders ← $\underbrace{[r \ldots r]}_{m}$
            return new-leaders
    ▷ If every region has failed servers, then we first choose the
    region with most alive servers as leaders
    Choose $r \in [0, 2f]$: region-$r$ has the most alive servers
    for $s \leftarrow 0$ to $(m-1)$ do
        if server <$s, r$> is alive then
            new-leaders[$s$] ← $r$
        else
            Choose $r' \in [0, 2f]$: server $< s, r' >$ is alive
            new-leaders[$s$] ← $r'$
    return new-leaders
```

## C Correctness Proof of Tiga

Tiga guarantees two correctness properties, i.e., per-shard linearizability and strict serializability across shards.

- (Per-shard linearizability) All committed transactions always satisfy linearizability in each shard.
- (Strict Serializability) The execution results for all committed transactions across multiple shards are strictly serializable.

### C.1 Proof of Per-Shard Linearizability

**Lemma C.1** (Durability). *If a transaction $T$ is committed in $shard_i$ under the global view $gv_1$, then $T$ can always be recovered in any follow-up views $gv_2 \geq gv_1$.*

First of all, a minority ($\leq f$) of follower failure does not interrupt service availability of Tiga, so it does not trigger global view change. As a result, there is no violation of durability. We focus on the cases when the leader fails or encounters network partition. In both cases, the view manager will launch a global view change and leads to the election of new leaders and the reconstruction of log lists in the new global view. Based on which path is used to commit $T$ in $shard_i$, we discuss two different cases.

***Case-1:** $T$ **is committed in the fast path.*** According to the super quorum check in the fast path (§3.4), $T$ can be committed

```
50: function SEND-CM-PREPARE()
51:     $msg.type \leftarrow$ CM-PREPARE
52:     $msg.v\text{-}view \leftarrow v\text{-}view$
53:     $msg.v\text{-}rid \leftarrow v\text{-}rid$
54:     $msg.prepare\text{-}g\text{-}view \leftarrow prepare\text{-}g\text{-}view$
55:     $msg.prepare\text{-}g\text{-}vec \leftarrow prepare\text{-}g\text{-}vec$
        ▷ Broadcast the prepare message to every VMgr replica
56:     for $rid \leftarrow 0$ to $2f$ do
57:         SEND-MESSAGE($msg$, $rid$)
58: function SEND-CM-PREPARE-REPLY(DEST)
59:     $msg.type \leftarrow$ CM-PREPARE-REPLY
60:     $msg.v\text{-}view \leftarrow v\text{-}view$
61:     $msg.v\text{-}rid \leftarrow v\text{-}rid$
62:     $msg.prepare\text{-}g\text{-}view \leftarrow prepare\text{-}g\text{-}view$
63:     $msg.prepare\text{-}g\text{-}vec \leftarrow prepare\text{-}g\text{-}vec$
64:     $msg.prepare\text{-}mode \leftarrow prepare\text{-}mode$
        ▷ Broadcast the prepare message to every VMgr replica
65:     for $rid \leftarrow 0$ to $2f$ do
66:         SEND-MESSAGE($msg$, $dest$)
67: function SEND-CM-COMMIT()
68:     $msg.type \leftarrow$ CM-COMMIT
69:     $msg.v\text{-}view \leftarrow v\text{-}view$
70:     $msg.v\text{-}rid \leftarrow v\text{-}rid$
71:     $msg.g\text{-}view \leftarrow g\text{-}view$
72:     $msg.g\text{-}vec \leftarrow g\text{-}vec$
        ▷ Broadcast the commit message to every VMgr replica
73:     for $rid \leftarrow 0$ to $2f$ do
74:         SEND-MESSAGE($msg$, $rid$)
75: function SEND-VIEW-CHANGE-REQ()
      ▷ Broadcast the global view change requests to all Tiga servers
76:     $msg.type \leftarrow$ VIEW-CHANGE-REQ
77:     $msg.g\text{-}view \leftarrow g\text{-}view$
78:     $msg.g\text{-}vec \leftarrow g\text{-}vec$
79:     for $s \leftarrow 0$ to $m - 1$ do
80:         for $r \leftarrow 0$ to $2f$ do
81:             SEND-MESSAGE($msg$, <s,r>)
```

through the fast path if the leader and $f + \lceil f/2 \rceil$ followers have the **same** log lists containing $T$. "**Same** log lists" indicates that $T$ has the same timestamp on these $1 + f + \lceil f/2 \rceil$ servers. Because at most $f$ servers simultaneously fail in $shard_i$, then based on quorum intersection, among the $f + 1$ servers that participate in the recovery, there are at least $1 + f + \lceil f/2 \rceil - f = \lceil f/2 \rceil + 1$ servers containing $T$. Therefore, $T$ can be recovered from any $f + 1$ servers that are selected to rebuild the log list (refer to line 51–61 in Algorithm 5).

***Case-2:*** $T$ ***is committed in the slow path.*** According to the quorum check in the slow path (§3.7), Servers' advancing their sync-points *happens before* sending the slow-replies. In other words, when $T$ is committed in the slow path, there have been at least $f + 1$ servers whose sync-points surpasses the position of $T$ in the servers' log lists. Based on quorum intersection, among the $f + 1$ server that participate in the recovery, there is at least $(f + 1) + (f + 1) - (2f + 1) = 1$ server's sync-point that surpasses the position of $T$ in the server's log list. Since the new leader will pick the server holding largest sync-point among the participating servers, and copy all the log entries from the corresponding server up to sync-point, so $T$ will be included in the recovered log list (refer to line 48–50 in Algorithm 5).

Combing Case-1 and Case-2, we have proved that the committed transactions can always be recovered after any view change (due to server failure or network partition).

**Lemma C.2** (Consistency). *If a transaction $T$ is committed in $shard_i$ under a global view $gv_1$, then its execution result will remain the same in any follow-up views $gv_2 \geq gv_1$.*

(1) Given a committed transaction $T$ in the global view $gv_1$, we can also prove that any transaction $T'$ executed before $T$ on the leader is also committed in $gv_1$. More specifically, if $T$ is committed through the fast path, then there are $f + \lceil f/2 \rceil + 1$ servers (including the leader) have consistent log lists up to $T$. Since the log list up to $T'$ is a sub-sequence, these $f + \lceil f/2 \rceil + 1$ servers also have the consistent log list up to $T'$, so $T'$ is also a committed transaction through the fast path. If $T$ is committed through the slow path, then the recovered sync-point, which surpasses the log position of $T$, will also surpass the log position of $T'$. Therefore, $T'$ is also a committed transaction in the slow path.

(2) Because the timestamp is a property of the transaction, the timestamp will also be recovered if the committed transaction is recovered in the new view (Durability). Therefore, all the committed transactions in the global view $gv_1$ will remain the same timestamp in the new view $gv_2$. Since the leader executes the transactions based on the timestamp order, then for any transaction $T'$, which is committed in the view $gv_1$ and executed before $T$ (i.e., $T'$ has a smaller timestamp), the transaction $T'$ will still be recovered with the same timestamp and executed before $T$.

Combining (1) and (2), we conclude that given a transaction $T$ committed in the global view $gv_1$, all the transactions executed before $T$ by the leader in $gv_1$ will are also committed and be executed before $T$ by the new leader in the follow-up global view $gv_2$.

(3) Therefore, the only case that can cause the execution inconsistency to $T$ is that, there exists another conflicting transaction $T''$, which has not been executed by the leader in the global view $gv_1$, but is recovered on the new leader in $gv_2$ with a smaller timestamp than $T$, making $T''$ executed before $T$ in the new global view $gv_2$. However, we will prove by contradiction that such cases can never happen.

***Case-1:*** $T$ ***is committed in the fast path.*** Assume there exists another $T''$ with a smaller timestamp than the committed transaction $T$, but has not been executed by the leader in the old global view $gv_1$. In order for $T''$ to be recovered in the new view $gv_2$, $T''$ must exist in the log lists of at least $\lceil f/2 \rceil + 1$ servers which have survived from $gv_1$ to $gv_2$. However, if

**Algorithm 5** Server action during global view change

```
Local State:                                              ▷
s,                                        ▷ the server's shard-id
r,                                        ▷ the server's replica-id
status,                         ▷ NORMAL, RECOVERING or VIEWCHANGE
l-view,                             ▷ the local view of this server
g-view,                            ▷ the global view of this server
g-vec,                  ▷ g-vec records the local view of each shard
g-mode,          ▷ the current mode for cross-shard coordination
sync-point,                  ▷ the point to which the server's log
                              ▷ is consistent with the leader's log
last-normal-view,              ▷ The most recent l-view in which
                                  ▷ the server's status is NORMAL
vQuorum,                 ▷ the set to collect VIEW-CHANGE messages
tQuorum,  ▷ the set to collect TIMESTAMP-VERIFICATION messages
upon receiving VIEW-CHANGE-REQ, msg, FROM VMGR do
    if msg.g-view ≤ g-view then
        return                            ▷ msg is stale, ignore it
    if status = RECOVERING then
    ▷ Recovering server does not participate view change
        return
    ▷ Switch server status
    status ← VIEWCHANGE
    ▷ Empty pq and append these txns to log list
    pendingEntries ← sort(pq)
    pq.clear()
    log ← log.append(pendingEntries)
    g-view ← msg.g-view
    g-vec ← msg.g-vec
    g-mode ← msg.g-mode
    l-view ← g-vec[s]
    leader-r ← l-view%$(2f + 1)$   ▷ The new leader's replica-id
    ▷ Send VIEW-CHANGE message to the new leader
    SEND-VIEW-CHANGE(<s, leader-r>)
upon receiving VIEW-CHANGE, msg do
    if msg.g-view < g-view then
        return
    if status=RECOVERING then
        return
    status ← VIEWCHANGE
    if msg.g-view > g-view then               ▷ Restart a new view
        g-view ← msg.g-view
        g-vec ← msg.g-vec
        g-mode ← msg.g-mode
        l-view ← g-vec[s]
        vQuorum ← ∅
    vQuorum ← vQuorum ∪ {m}
    if |vQuorum| = $f + 1$ then
        REBUILD-LOG()
        VERIFY-TIMESTAMP-ACROSS-SHARDS()
        ▷ Broadcast the new log list to followers
        SEND-START-VIEW()
upon receiving TIMESTAMP-VERIFICATION, msg do
    if status ≠ VIEWCHANGE or msg.l-view ≠ l-view then
        return
    tQuorum ← tQuorum ∪ {msg}
upon receiving START-VIEW, msg do
    if status ≠ VIEWCHANGE or msg.l-view ≠ l-view then
        return
    log ← msg.log
    sync-point ← msg.log.length() − 1
    last-normal-view ← l-view
    status ← NORMAL
function REBUILD-LOG()
    new-log ← []                ▷ Initialize a new log list to build
    largest-normal-view ← max{msg.lnv|msg ∈ vQuorum}
    largest-sync-point ← max{msg.sp|msg ∈ vQuorum
                          and msg.lnv = largest-normal-view}
    ▷ Recover Part (a) logs (refer to the third step in §4)
    Choose msg ∈ vQuorum : msg.sp = largest-sync-point
    for i ← 0 to largest-sync-point do
        new-log.append(msg.log[i])
    ▷ Recover Part (b) logs
    t ← max{msg.t : msg ∈ new-log}
    candidateLogs ← {}
    for msg ∈ vQuorum do
        for i ← msg.sync-point + 1 to msg.log.length()-1 do
            candidateLogs ← candidateLogs ∪ {msg.log[i]}
    committedLogs ← {}
    for e ∈ candidateLogs do
    ▷ Check how many servers in vQuorum have e in their logs
        Q ← {msg ∈ vQuorum : e ∈ msg.log}
        if |Q| ≥ $\lceil f/2 \rceil + 1$ then
            committedLogs ← committedLogs ∪ {e}
    ▷ Sort logs by their timestamps
    logList ← sort(committedLogs)       ▷ This is Part (b) logs
    new-log.append(logList)           ▷ Concat two parts of logs
    log ← new-log                     ▷ Replace the old log list
function VERIFY-TIMESTAMP-ACROSS-SHARDS()
    SEND-TIMESTAMP-VERIFICATION()
    Wait until |tQuorum| = $m − 1$
    logSet ← {log[i]|i ∈ 0 ··· log.length() − 1}
    allTxnIds ← {}
    for msg ∈ tQuorum do
        for i ← 0 to msg.info.length()-1 do
            allTxnIds ← allTxnIds ∪ {msg.info[i].id}
    missingTxnIds ← allTxnIds − {txn.txn-id|txn ∈ logSet}
    Fetch missing transactions from other shards based on
    missingTxnIds, and add to logSet
    for txn ∈ logSet do
    ▷ If multiple shards have this txn with different timestamps,
    pick the largest one as the agreed timestamp
        for mm ∈ tQuorum do
            if ∃e ∈ mm.info : e.id = txn.txn-id then
                txn.t ← max{e.t, txn.t}
    ▷ Sort logSet by (verified) timestamps
    log ← Sort(logSet)
```

```
function SEND-VIEW-CHANGE(<ss,rr>)
    ▷ Send VIEW-CHANGE message to server <ss,rr>
    msg.type ← VIEW-CHANGE
    msg.g-view ← g-view
    msg.l-view ← l-view
    msg.g-mode ← g-mode
    msg.lnv ← last-normal-view
    msg.log ← log
    msg.sp ← sync-point
    SEND-MESSAGE(msg, <ss,rr>)
function SEND-TIMESTAMP-VERIFICATION()
    for ss ← 0 to $m - 1$ do
        if $ss = s$ then
            continue  ▷ No need to send the message to itself
        msg.type ← TIMESTAMP-VERIFICATION
        msg.g-view ← g-view
        msg.l-view ← l-view
        msg.s ← s
        msg.r ← r
        msg.info ← []
        for i ← 0 to log.length()-1 do
            txn ← logs[i]
            if txn involves $shard_s$ then
                msg.info.append({id : txn.txn-id, t : txn.t})
        ▷ Calculate replica-id of $shard_s$s's leader
        rr ← g-vec[ss]%$(2f + 1)$
        SEND-MESSAGE((msg, <ss,rr>)
function SEND-START-VIEW()
    msg.type ← START-VIEW
    msg.g-view ← g-view
    msg.l-view ← l-view
    msg.g-mode ← g-mode
    msg.s ← s
    msg.r ← r
    msg.log ← log
    for rr ← 0 to $2f$ do
        if $rr = r$ then
            continue
        SEND-MESSAGE((msg, <s,rr>)
```

**Algorithm 6** Server action during rejoining Tiga

```
Local State:                                        ▷
s,                                       ▷ the server's shard-id
r,                                       ▷ the server's replica-id
status,                   ▷ NORMAL, RECOVERING or VIEWCHANGE
l-view,                          ▷ the local view of this server
g-view,                         ▷ the global view of this server
g-vec,                  ▷ g-vec records the local view of each shard
g-mode,           ▷ the current mode for cross-shard coordination
sync-point,                ▷ the point to which the server's log
                           ▷ is consistent with the leader's log
last-normal-view,            ▷ The most recent l-view in which
                                ▷ the server's status is NORMAL
upon status =RECOVERING do
    msg′ ←INQUIRE-VIEW-MANAGER()
    g-view ← msg′.g-view
    g-vec ← msg′.g-vec
    g-mode ← msg′.g-mode
    l-view ← g-vec[s]
    ▷ Compute the leader server's replica-id of $shard_s$
    leader-r ← g-vec[s] % $(2f + 1)$
    STATE-TRANSFER(<s, leader-r>)
function INQUIRE-VIEW-MANAGER()
    msg.type←INQUIRE-REQ
    msg.s ← s
    msg.r ← r
    Send the inquiry msg to any replica of the view manager
    (if the replica is a follower, it will forward the inquiry to the
    leader of the view manager.)
    Wait until receiving reply msg′ containing <g-view,g-vec>
    return msg′
function STATE-TRANSFER(<ss,rr>)
    msg.type ←STATE-TRANSFER-REQ
    msg.g-view ← g-view
    msg.l-view ← l-view
    msg.s ← s
    msg.r ← r
    SEND-MESSAGE(msg, <ss,rr>)
    Wait until status = NORMAL
    return
upon receiving STATE-TRANSFER-REQ, msg do
    if status ≠ NORMAL then
        return
    if g-view ≠ msg.g-view or l-view ≠ msg.l-view then
        return
    msg′.type ← STATE-TRANSFER-REP
    msg′.log ← log
    msg′.g-view ← g-view
    msg′.l-view ← l-view
    msg′.sp ← sync-point
    SEND-MESSAGE(msg′, <msg.s, msg.r>)
upon receiving STATE-TRANSFER-REP, msg do
    if status ≠ RECOVERING then
        return
    if g-view ≠ msg.g-view or l-view ≠ msg.l-view then
        return
    log ← msg.log
    last-normal-view ← l-view
    sync-point ← msg.sp
    status ← NORMAL                ▷ Rejoin as a NORMAL follower
```

these $\lceil f/2 \rceil + 1$ servers contains $T''$ during the global view $gv_1$, then their log lists will be inconsistent with the leader in the view $gv_1$ because the leader's log list does not contain $T''$ ahead of $T$. As a result, the coordinator can only obtain at most $(2f + 1) - (\lceil f/2 \rceil + 1) = f + \lfloor f/2 \rfloor$ fast-replies containing the same hash (i.e., indicating the servers have the same log lists). Thus, $T$ cannot be committed in the fast path, which contradicts the assumption of Case-1.

***Case-2:** $T$ **is committed in the slow path.*** Still assume there exists such a $T''$ with a smaller timestamp than $T$, but has not been executed by the leader in the old global view $gv_1$. Since $T$ is committed in the slow path, $T$ and the other committed transactions before $T$ are recovered by using the largest sync-point from the remaining $f + 1$ servers that participate in the recovery. If $T''$ is recovered after the sync-point, then $T''$ has a larger timestamp than $T$ because transactions are appended to log lists according to their timestamp order. On the other hand, if $T''$ is also recovered prior to the sync-point, then it indicates there exists at least one follower in the previous global view $gv_1$, whose log list contains $T''$ even after it has ensured the log list is consistent with the leader's log list. Further, it can be derived that the leader's log list also contains $T''$ with a smaller timestamp than $T$. As a result, the leader in the global view $gv_1$ should have executed $T''$ before $T$, which contradicts the assumption that the old leader has not executed $T''$.

Combining Case-1 and Case-2, we have proved that there does not exist such a transaction $T''$ which has not been executed before a committed transaction $T$ in an old global view but is executed before $T$ in a new global view.

Finally, combining (1), (2) and (3), we have proved the consistency property in each shard of Tiga.

**Theorem C.3** (Per-Shard Linearizability). *All committed transactions always satisfy linearizability in each shard.*

According to [78], the linearizability property can reworded as: Given two committed transactions $T_1$ and $T_2$, if the execution effect of $T_1$ is observed by $T_2$, then no contrary observation can occur afterwards (i.e., it should not appear to revert or be reordered).

Based on the design of Tiga, for any shard in Tiga, if $T_2$ can observe $T_1$'s execution effect before $T_2$ is executed, then $T_2$ must have a larger timestamp than $T_1$.

Based on the proved durability property (Lemma C.1), $T_1$ and $T_2$ will be recovered in the new global view with the same timestamps as in the previous global view that they have been committed, so their execution will not be reordered, i.e., $T_2$ will still observe $T_1$'s execution effect, rather than that $T_1$ observes $T_2$'s execution effect.

Based on the proved consistency property (Lemma C.2), $T_1$'s execution result will remain unchanged in the new global view, so $T_2$ will continue to observe the *same* execution effect (result) of $T_1$ and there is no contrary observation. Therefore, linearizability is guaranteed in each shard of Tiga.

## C.2 Proof of Strict Serializability

. We decompose the proof of strict serializability into two steps. First, we prove Tiga guarantees serializability for all transactions and the serializable order is the order of transactions timestamp order. Second, we continue to prove the timestamp order is strictly serializable, i.e., transactions' real-time order does not contradict the serializable order.

**Lemma C.4** (Serializability). *The execution of all committed transactions in Tiga is serializable.*

We use the serializability graph [18, 19] to prove serializability. Serializability graph considers every transaction as one vertex in the graph, and adds directional edges between conflicting (non-commutative) transactions to represent the *happened-before* order: Given two conflicting transactions $T_1$ and $T_2$, if $T_1$ is executed earlier than $T_2$ on one leader server, then there is an edge from $T_1$ to $T_2$. According to the Serializability Theorem [19], the execution of transactions is serializable if the serializability graph is acyclic. We prove it by contradiction.

Suppose there is a cycle in the serializability graph representing Tiga's execution. Without the loss of generality, we assume a pair of conflicting transactions are involved in the cycle, denoted as $T_1$ and $T_2$. Thus the serializability graph can be described as $T_1 \rightarrow \cdots T_2 \rightarrow \cdots T_1$.

According to Tiga's protocol design, each leader executes the transactions following the order of their timestamps after timestamp agreement (§3.5). From $T_1 \rightarrow \cdots T_2$, we know that there must exist one leader server, denoted as $leader_1$, which executes $T_1$ earlier than $T_2$, so $T_1$'s timestamp is smaller than $T_2$'s timestamp on $leader_1$, denoted as $t_1^1 < t_2^1$. But on the other hand, from $T_2 \rightarrow \cdots T_1$, we know there is another leader server, denoted as Leader-2, where $T_2$'s timestamp is smaller than $T_1$'s timestamp, denoted as $t_2^2 < t_1^2$ (the subscripts indicate transaction identifier and the superscripts indicate the shard identifier).

For any committed transaction, the coordinator considers it committed after the coordinator has verified that all shards are using the same timestamp (Line 28-32 in Algorithm 3).[3] Therefore, we have $t_1^1 = t_1^2$ and $t_2^1 = t_2^2$. Then we come to the contradiction: both $t_1^1 < t_1^2$ and $t_1^1 > t_1^2$ hold simultaneously.

Therefore, the assumption is not true. The serializability graph is acyclic, which indicates the execution by Tiga's leader servers is serializable, which follows the transactions' timestamp order after timestamp agreement.

**Theorem C.5** (Strict Serializability). *The execution of all committed transactions in Tiga is strictly serializable. In other words, the execution of all committed transactions respects their real-time ordering.*

***Notation.*** For every transaction $T_i$, we use $\textsc{start}_i$ to represent the real time that $T_i$ is started, and use $\textsc{complete}_i$ to represent the real time that $T_i$ is completed. $T_i$ involves a set of shards for joint execution, so we use $ShardSet_i$ to represent the *shard-id*s of $T_i$'s involved shards. For an involved shard $shard_s$, we use $\textsc{exec}_i^s$ to represent the real

[3] The coordinator might know this agreement earlier than leaders: At this point, leaders themselves might be still uncertain whether they already used the agreed timestamp if they have not completed timestamp agreement.

time when $T_i$ is optimistically executed on $shard_s$'s leader $L_s$ ($s \in ShardSet_i$), and we use $\textsc{deq}_i^s$ to represent the real time when $T_i$ dequeued from $L_s$'s queue. Some transactions might be re-executed if its first optimistic execution is invalid and revoked. Here we do not consider the invalid execution, and only consider the $T_i$'s final optimistic execution that is eventually committed. In this execution, $T_i$ used the agreed timestamps across all leaders, $T_i.t = T_i.t_{agreed}$. Thus, we have $\forall s \in ShardSet_i : \textsc{start}_i < \textsc{exec}_i^s < \textsc{complete}_i$.

From Lemma C.4, we know Tiga has a valid serializability schedule according to the transactions' agreed timestamp order. Therefore, given two transactions in the serializable schedule, i.e., $T_i$ and $T_j$ ($i < j$ and $T_i.t_{agreed} < T_j.t_{agreed}$), we represent the dependency relationship between them as $T_i \longrightarrow T_{i+1} \longrightarrow \cdots \longrightarrow T_{j-1} \longrightarrow T_j$, with each two consecutive transactions conflict on some data items. To prove by contradiction, we assume $T_i$ and $T_j$'s real-time ordering violates the serializable schedule, i.e. $\textsc{complete}_j < \textsc{start}_i$.

Based on Tiga's protocol design:

(1) $T_i$ can be dequeued from $L_s$'s queue only after timestamp agreement succeeds and $L_s$ confirms all leaders have used the same timestamp for $T_i$. Besides, on each leader, $T_i$'s optimistic execution happens before timestamp agreement, so we have $\forall s_1, s_2 \in ShardSet_i : \textsc{exec}_i^{s_1} < \textsc{deq}_i^{s_2}$

(2) On any leader $L_s$, transactions can only be executed when there is no conflicting transactions ahead of it in the queue. Consider two consecutive transactions $T_i \longrightarrow T_{i+1}$, and $s \in ShardSet_i \cap ShardSet_j$, since $T_i.t_{agreed} < T_{i+1}.t_{agreed}$ , $T_{i+1}$ can only be executed after $T_i$ has dequeued, then we have $\textsc{exec}_i^s < \textsc{deq}_i^s < \textsc{exec}_{i+1}^s$.

Since each two consecutive transactions in the schedule conflict on some data items, there are at least 1 common shard shared by the two transactions, we use $s_{i,i+1}$ to represent any common shard that is shared by $T_i$ and $T_{i+1}$, so we have

$s_{i,i+1} \in ShardSet_i \cap ShardSet_{i+1}$,
$s_{i+1,i+2} \in ShardSet_{i+1} \cap ShardSet_{i+2}$,
$\cdots$,
$s_{j-1,j} \in ShardSet_{j-1} \cap ShardSet_j$

Combing (1) and (2), we have $\textsc{exec}_i^{s_{i,i+1}} < \textsc{deq}_i^{s_{i,i+1}} < \textsc{exec}_{i+1}^{s_{i,i+1}} < \textsc{deq}_{i+1}^{s_{i+1,i+2}} < \textsc{exec}_{i+2}^{s_{i+1,i+2}} < \cdots < \textsc{exec}_j^{s_{j-1,j}}$

However, since we assume $\textsc{complete}_j < \textsc{start}_i$, then we have $\textsc{exec}_j^{s_{j-1,j}} < \textsc{complete}_j < \textsc{start}_i < \textsc{exec}_i^{s_{i,i+1}}$, which contradicts with the derived relation $\textsc{exec}_i^{s_{i,i+1}} < \textsc{exec}_j^{s_{j-1,j}}$ based on Tiga's design. Therefore, our assumption is not correct, there does not exists such $T_i$ and $T_j$, whose real-time ordering violates the serializable schedule, i.e., the schedule is strictly serializable.

## D Tiga's Incremental Hash

Tiga's incremental hash is computed based on the current log list and is included in servers' fast-replies. The coordinator can compare the hash values from different servers' fast-replies. If the hash values are the same, then the coordinator knows the two servers have the same log list.

The incremental hash is computed as follows.

First, for every log entry, a hash can be calculated by concating the *client-id*, *txn-id*, and timestamp as a string, and then converting the string into a hash value. In Tiga's implementation, we use SHA1 to do this, but the choice of hash function is orthogonal to Tiga's protocol design: SHA1 can be replaced by the other alternatives (e.g., SHA256) for stronger collision resistance.

Second, the server aggregates the hashes of the log entries. Initially, the server maintains a zeroed hash value $h$. Since we use SHA1 in Tiga's implementation, $h$ is composed of 80 bits and its initial value is 80 bits of 0. For every log entry $e$ appended to the log list, we use the logical operation, exclusive-or $\oplus$, to aggregate its hash $\texttt{SHA1}(e)$ with $h$, i.e.,

$$h \longleftarrow h \oplus \texttt{SHA1}(e)$$

On the other hand, during log synchronization (§3.7), some log entries might be removed from the log list, which also leads to a hash update. Due to the nature of exclusive-or operation, removing the log entry $e$ from the log list does the same operation as the aggregation, i.e., $h \longleftarrow h \oplus \texttt{SHA1}(e)$.

The benefit of using incremental hash is to avoid recomputing the hash value from scratch every time, because adding or removing a log entry only incurs one exclusive-or operation.

In Tiga, we further extend the incremental hash approach to support commutativity optimization. First of all, we realize that read-only transactions do not modify the server state. Therefore, the hash does not need to encode read-only transactions. Then, for write transactions, we consider commutativity when updating the hash. Instead of maintaining one single hash value, each server maintains a table of per-key hashes. For every newly appended write transaction, the sever will XOR its hash to update the corresponding per-key hashes in the table according to the transaction's read-set and write-set. For example, if a transaction $T$ is appended to the log list and $T$ needs to access two keys $x$ and $y$, then in the table, both $x$ and $y$'s hashes will aggregate $\texttt{SHA1}(T)$, i.e., $hash_x \longleftarrow hash_x \oplus \texttt{SHA1}(T)$ and $hash_y \longleftarrow hash_y \oplus \texttt{SHA1}(T)$. Meanwhile, when sending fast-reply for $T$, the server only encodes the $x$'s and $y$'s hashes instead of the hashes for all log entries. Specifically, the server first concat the key with the *hash* as a string, and then convert the per-key string into a SHA1 hash, and finally aggregate them into the final hash, i.e.,

$$h_T \longleftarrow \texttt{SHA1}(<x, hash_x>) \oplus \texttt{SHA1}(<y, hash_y>)$$

## E Optimizations in Slow Reply

In Tiga's workflow (⑤-⑥ in Figure 3), the followers send a slow reply for each entry after the follower's sync-point has surpassed the entry, i.e., the follower has confirmed that its log is consistent with the leader's log up to this entry. Such design cuts down the length of the slow path and allows the transaction to be committed only 1 message delay later if the transaction fails to be committed in the fast path. However, we find two issues in real implementation.

First, the slow-replies can be redundant and increase the load for coordinators. Even when the transaction is committed in the fast path ④, the servers are unaware of that, so they continue to send the unnecessary slow-replies. Processing these slow-replies adds more burden to the coordinators, especially when the coordinator machine is not powerful.

Second, when implementing Tiga, we find many ordinary RPC libraries do not support "server push" [13] (i.e., the server can proactively send messages to the client instead of passively replying the the client's request), including the RPC library used by Janus and Tiga.

Therefore, we choose not to let servers proactively send slow-replies for each transaction. Instead, the coordinator should actively ask for slow-replies from followers when the coordinator finds the followers' fast-replies contain different hash values from the leader. Besides, we also notice that slow-replies can be batched, especially when the coordinator maintains many outstanding transactions. Instead of requesting the slow-replies one by one, we let the coordinators periodically (e.g., every 10 ms when Tiga works in WAN) inquire each follower's <*g-view*, *l-view*> and sync-points.

We use an example to explain how the periodic inquiry works. Consider the coordinator multicasts a transaction to a targeting shard but fails to commit it in the fast path because the leader and followers have inconsistent log lists. However, followers will later receive the cross-region synchronization message (⑤ in Figure 3), and then modify their log lists, and also advance their sync-points to indicate to which point their log lists have become consistent with the *leader*'s log list. In the leader's fast-reply, we let the leader include its *view-id* and the *log-id* assigned to this entry (transaction). Through the periodic inquiry, the coordinator finally confirms the transaction is committed in the targeting shard if (1) At least $f$ followers in this shard have the same view as their leader, and (2) these followers have advanced their sync-points to be larger than the *log-id* in the leader's fast-reply.

# F Decomposing Dependent Transactions

## F.1 Common Existence of One-Shot Transactions

Tiga targets improving the performance for OLTP systems to process one-shot transactions. One-shot transaction is a constrained category of transactions, but commonly exists in many real-world workloads. We have examined 12 benchmarks widely used and publicized in literature. Table 5 summarizes the characteristics of these benchmarks. We find that almost all the transactions included in these workloads are one-shot transactions, which verifies the common existence of one-shot transactions in practice.

## F.2 Example of Decomposing Transactions

Although Tiga targets one-shot transactions, it can also support handling dependent transactions by using the decomposition technique. Figure 16 (left side) illustrates a typical dependent transaction. We use $a$, $b$, $c$, $d$ to represent different keys distributed across shards, and use $v_a$, $v_b$, $v_c$, $v_d$ to represent their values.

Different from one-shot transactions, the read set (i.e., $a$ and $b$) and write set (i.e., $c$ and $d$) of the transaction are not completely known in advance. In this example, the write set (the keys and/or the values) depends on the values of the read set and some functions ($F_1$ and $F_2$). Therefore, this transaction $U(a, b)$ cannot be executed in one shot. In order for Tiga to execute such transactions in strict serializability, we decompose the transaction into three one-shot transactions, i.e., $U_1$, $U_2$, $U_3$, shown on the right side of Figure 16.

The decomposition technique enables Tiga to handle dependent transactions because each of the three transactions, $U_1$, $U_2$ and $U_3$, are one-shot transactions. However, the decomposition inevitably introduces aborts: If $U_2$ fails because some other transactions come in between $U_1$ and $U_2$, causing $U_2$'s lock failure or dirty read (i.e., $v_a$ and $v_b$), then the transaction will be aborted and we need to retry from $U_1$ after releasing all locks. Fortunately, as previous works [70, 87] observed, real-life OLTP workloads rarely involve key-dependencies on frequently updated data, leading to very few occurrence of such aborts (e.g., $U_2$ fails).

```
$U(a,b)$:
    $v_a, v_b \leftarrow$ Read $a, b$
    $c, d \leftarrow F_1(v_a, v_b)$
    $v_c, v_d \leftarrow F_2(v_a, v_b)$
    Write $c \leftarrow v_c, d \leftarrow v_d$
```

➡

```
$U_1(a,b)$:
    $v_a, v_b \leftarrow$ Read $a, b$
    $c, d \leftarrow F_1(v_a, v_b)$
    $v_c, v_d \leftarrow F_2(v_a, v_b)$

$U_2(a, b, c, d, v_a, v_b)$:
    $lockOK$ = (SharedLock($a, b$)&& ExclusiveLock($c, d$))
    $v'_a, v'_b \leftarrow$ Read $a, b$
    If (! $lockOK$)||($v'_a, v'_b \neq v_a, v_b$):
        Release all locks, aborts and retry from $U_1$
    Else:
        Continue to execute $U_3$

$U_3(a, b, c, d, v_a, v_b)$:
    Write $c \leftarrow v_c, d \leftarrow v_d$
    Release all locks
```

**Figure 16.** Example of decomposing dependent transaction

**Table 5.** Common Benchmarks

| Benchmark | Domain | Tables | Columns | Types of Txns | Read Ratio | One-Shot? |
|---|---|---|---|---|---|---|
| AuctionMark [6] | Online auctions | 16 | 125 | 9 | 55.0% | Yes |
| JPAB [75] | Java performance bench | 7 | 68 | 4 | 25.0% | Yes |
| LinkBench [10] | Social network | 3 | 7 | 10 | 69.05% | Yes |
| SEATS [7] | Online airline ticketing | 10 | 189 | 6 | 45.0% | Yes |
| SIBench [20] | Snapshot isolation | 1 | 2 | 2 | 50.0% | Yes |
| SmallBank [5] | Banking system | 3 | 6 | 6 | 15.0% | Yes |
| TATP [50] | Caller location system | 4 | 51 | 7 | 40.0% | Yes |
| TPC-C [31] | data warehouse | 9 | 92 | 5 | 8% | Yes |
| Twitter [33] | Social network | 5 | 18 | 5 | 0.9% | Yes |
| Voter [90] | Phone-based electon system | 3 | 9 | 1 | 92.2% | Yes |
| YCSB [91] | NoSQL benchmark | 1 | 11 | 6 | 50.0% | Yes |
| TAOBench [22] | Social network | 2 | 5 | 8 | 99.8% | >99.99% [1] |

[1] In TAOBench, only the `edge_add` transaction are not one-shot because it needs check the eligibility before insertion. However, such transactions only occupy <0.01% according to `workload_o.json`, and such transactions can be also processed by Tiga using the decomposition technique described in §F.