Building Consistent Transactions with Inconsistent Replication

UW Technical Report UW-CSE-14-12-01

Irene Zhang Naveen Kr. Sharma Adriana Szekeres
Arvind Krishnamurthy Dan R. K. Ports
University of Washington
{iyzhang, naveenks, aaasz, arvind, drkp}@cs.washington.edu

Abstract

Application programmers increasingly prefer distributed storage systems with distributed transactions and strong consistency (e.g., Google's Spanner) for their strong guarantees and ease of use. Unfortunately, existing transactional storage systems are expensive to use because they rely on expensive replication protocols like Paxos for fault-tolerance. In this paper, we take a new approach to make transactional storage systems more affordable; we eliminate consistency from the replication protocol, while still providing distributed transactions with strong consistency to applications.

This paper presents TAPIR *– the Transaction Application Protocol for Inconsistent Replication – the first transaction protocol to use a replication protocol,* inconsistent replication*, that provides fault-tolerance with no consistency. By enforcing strong consistency only in the transaction protocol, TAPIR is able to commit transactions in a single round-trip and schedule distributed transactions with no centralized coordination. We demonstrate the use of TAPIR in TAPIR-KV, a key-value store that provides high-performance transactional storage. Compared to system using conventional transaction protocols that require replication with strong consistency, TAPIR-KV has 2× better latency* and *throughput.*

1. Introduction

Distributed storage systems provide fault-tolerance and availability for large-scale web applications like Google and Amazon. Increasingly, application programmers prefer systems that support distributed transactions with strong consistency[1] to help them manage application complexity and concurrency in a distributed environment. Several recent systems [3, 7, 15, 19] reflect this trend, most notably Google's Spanner system [8], which supports linearizable transactions.

For application programmers, distributed transactional storage with strong consistency comes at a price. These systems commonly use replication for fault-tolerance; unfortunately, replication protocols with strong consistency, like Paxos [21], impose a high performance cost. They require cross-replica coordination on every operation, increasing the latency of the system, and typically need a designated leader, reducing the throughput of the system.

A number of transactional storage systems have addressed some of these performance limitations. Some systems reduce throughput limitations [15, 33] or wide-area latency [19, 30], while others tackle latency and throughput for read-only transactions [8], commutative transactions [19] or independent transactions [9]. Even more have sought to provide better performance at the cost of a limited transaction model [2, 44] or weaker consistency model [29, 40]. However, none of these systems have been able to improve latency and throughput for general-purpose, replicated read-write transactions with strong consistency.

In this paper, we use a new approach to reduce the cost of replicated, read-write transactions and make transactional storage more affordable for application programmers. Our key insight is that existing transactional storage systems waste work and performance by integrating a distributed transaction protocol and a replication protocol that *both* enforce strong consistency. Instead, we show that it is possible to provide distributed transactions with better performance and the same transaction and consistency model using replication with *no consistency.*

To demonstrate our approach, we designed *TAPIR* – the Transactional Application Protocol for Inconsistent Replication. TAPIR uses a new replication technique, called *inconsistent replication* (IR), which pro-

[1] Strong consistency for transactions is sometimes referred to as linearizable isolation or external consistency. We use the terms interchangeably in this paper.

vides *fault-tolerance without consistency*. IR requires no synchronous cross-replica coordination or designated leaders. Using IR, TAPIR is able to commit a distributed read-write transaction in a *single round-trip* and schedule transactions across partitions and replicas with *no centralized coordination.*

Conventional transaction protocols cannot be easily combined with inconsistent replication. IR only guarantees that successful operations execute at a majority of replicas; however, replicas can execute operations in different orders and can be missing any operation at any time. To support IR's weak consistency model, TAPIR integrates several novel techniques:

- *Loosely synchronized clocks for optimistic transaction ordering at clients.* TAPIR clients schedule transactions by proposing a transaction ordering, which is then accepted or rejected by TAPIR's replicated storage servers.
- *New use of optimistic concurrency control to detect conflicts with only a partial transaction history.* TAPIR servers use quorum intersection with optimistic concurrency control to ensure that any transactions that violate the linearizable transaction ordering are detected and aborted by at least one server.
- *Multi-versioning for executing transactions out-of-order.* TAPIR servers use multi-versioned storage to install updates out-of-order and still converge on a single, consistent storage state.

We implement TAPIR in a new distributed transactional key-value store called *TAPIR-KV*, which supports linearizable ACID transactions over a partitioned set of keys. We compare TAPIR-KV's protocol to several conventional transactional storage systems, including one based on the Spanner system, and a non-transactional key-value store with eventual consistency. We find that TAPIR-KV reduces commit latency by 50% and increases throughput by more than 2x compared to standard systems and achieves performance close to that of the system with eventual consistency.

In this paper, we make the following significant contributions to our understanding of designing distributed, replicated transaction systems:

- We define inconsistent replication, a new replication technique that provides fault-tolerance without consistency. (Section 3)
- We design TAPIR, a new distributed transaction protocol that provides transactions with strong consistency using replication with no consistency. (Sections 4–6)
- We design and evaluate TAPIR-KV, a key-value store that uses inconsistent replication and TAPIR to provide high-performance transactional storage. (Section 7)

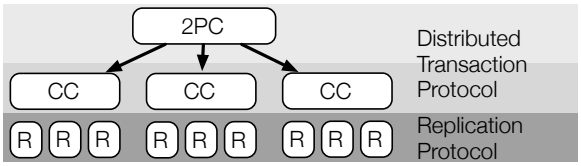

Figure 1: *Architecture of a distributed transactional storage system using replication.* Distributed transaction protocols coordinate transactions across partitioned data, or shards. They typically consist of two components: (1) an atomic commitment protocol like two-phase commit to coordinate distributed transactions across shards and (2) a concurrency control mechanism like strict two-phase locking to schedule transactions within each shard. A replication protocol with strong consistency like Paxos is responsible for keeping replicas in each shard synchronized.

2. Background

Replication protocols have become an important component in distributed storage systems. While classic storage systems use local disk for durability, modern storage systems commonly incorporate replication for better fault-tolerance and availability. Some new systems [35, 41] replace on-disk storage altogether with in-memory replication.

Replicated, transactional storage systems must implement both a distributed transaction protocol and a replication protocol. Liskov [27] and Gray [17] described one of the earliest architectures for these systems that is still used today many systems [3, 8, 15, 19, 27, 36]. This architecture places the distributed transaction protocol on top of the replication protocol, as shown in Figure 1. Other alternatives have been proposed as well (e.g., Replicated Commit [30]).

Conventional transaction protocols [4] assume the availability of an ordered, fault-tolerant log. This ordered log abstraction is easy to provide with disk; in fact, sequential writes are more efficient for disk-based storage than random writes. Replicating this log abstraction is more complicated and expensive. To enforce a serial ordering of operations to the replicated log, transactional storage systems must use a replication protocol with strong consistency, like Paxos [21], Viewstamped Replication (VR) [34] or virtual synchrony [5].

Replication protocols with strong consistency present a fundamental performance limitation for storage systems. To enforce a strict serial ordering of operations across replicas, they require cross-replica coordination on every operation. To avoid additional coordination when replicas disagree on operation ordering, most implementations use a designated leader to decide operation ordering and coordinate each operation. This leader presents a throughput bottleneck, while cross-replica coordination adds latency.

Combining these replication protocols with a distributed transaction protocol results in a large amount of complex distributed coordination for transactional storage systems. Figure 2 shows the large number of messages needed to coordinate a single read-write transaction in a system like Spanner. Significant work [23, 25, 31] has been done to reduce the cost of these replication protocols; however, much of it centers on commutative operations [6, 22, 32] and do not apply to general-purpose transaction protocols.

Maintaining the ordered log abstraction means that replicated transactional storage systems use expensive distributed coordination to enforce strict serial ordering in two places: the transaction protocol enforces a serial ordering of transactions across data partitions or *shards*, and the replication protocol enforces a serial ordering of operations within a shard. TAPIR eliminates this redundancy and its associated performance cost. As a result, TAPIR is able avoid the performance limitations of consistent replication protocols.

Conventional transaction protocols depend on operations to the log to determine the ordering of transactions in the system, so they cannot easily support a replication protocol without strong consistency. TAPIR carefully avoids this constraint to support inconsistent replication. Section 4 discusses some of the challenges that we faced in designing TAPIR to enforce a strict serial ordering of transactions without a strict serial ordering of operations across replicas.

3. Inconsistent Replication

The main objective of TAPIR is to build a protocol for consistent, linearizable transactions on top of a replication layer with *no consistency guarantees*. To achieve this, we must carefully design the weak guarantees of the replication protocol to support a higher-level protocol with stronger guarantees. This section defines *inconsistent replication*, a new replication protocol with fault-tolerance, but no consistency guarantees.

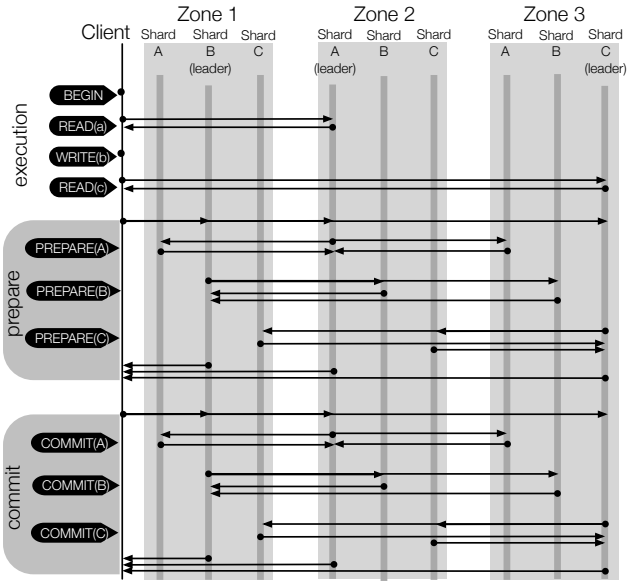

Figure 2: *Example read-write transaction using two-phase commit, Viewstamped Replication and strict two-phase locking.* Each zone represents an availablity region, which could be a cluster, datacenter or geographic region. Each shard holds a partition of the data stored in the system and is replicated across zones for fault-tolerance. For each read-write transaction, there is a large amount of distributed coordination. The transaction protocol must coordinate reads with the designated leader in each shard to acquire locks. To commit a transaciton, the transaction protocol coordinates across shards and then the replication protocol must coordinate within each shard.

3.1 IR Overview

Inconsistent replication provides *unordered*, fault-tolerant operations using a state machine replication model. Applications invoke operations through IR for fault-tolerant execution across replicas, and replicas execute operations in the order they receive them. Operations might be the puts and gets of a replicated key-value storage system or the prepares and commits of a transactional system. Unlike typical state machine replication, inconsistent replication treats application operations as *unordered*. Each replica executes operations in a different order; thus, unless all operations are commutative, replicas will be in an *inconsistent* state.

Because IR replicas are inconsistent and execute operations in different orders, IR does not expect replicas to always return the same result. Executing an operation at enough replicas to provide fault-tolerance (e.g., $f + 1$ replicas to survive f simultaneous failures) does

not ensure that the results from each replica will agree. Therefore, IR provides two types of guarantees: (1) fault-tolerance for an operation and (2) both consensus and fault-tolerance for an operation result. For example, IR must ensure consensus and fault-tolerance for TAPIR operations whose *result* determines transaction ordering.

We define two types of IR operations with these different guarantees. When applications invoke *inconsistent operations*, IR only ensures that the operation will not be lost for up to f simultaneous failures. When applications invoke *consensus operations*, IR preserves the operation and the result given by the majority of the replicas *if enough replicas return the same result*. While both operation types are fault-tolerant, consensus operations serve as the basic block in satisfying applications' consistency guarantees. TAPIR prepares are consensus operations, while commits and aborts are inconsistent operations.

IR provides these guarantees for up to f simultaneous server failures and any number of client failures. IR supports only fail-stop failures, not Byzantine faults or actively malicious servers. We assume an asynchronous network where messages can be lost or delivered out of order; however, we assume that messages sent repeatedly will be eventually delivered if the replica is up.

IR *does not* rely on synchronous disk writes; it ensures guarantees are maintained even if clients or replicas lose state on failure. This property allows IR to provide better performance, especially within a datacenter, compared to Paxos and its variants, which require synchronous disk writes and recovery from disk on failure. IR also provide better fault-tolerance because this property allows it to tolerate disk failures at replicas.

3.2 IR Protocol

Figure 3 summarizes the IR interfaces and state at clients/replicas. Applications invoke their operations with a Client Interface call. IR clients communicate with IR replicas to execute operations and collect their responses. IR uses $2f + 1$ replicas to tolerate f simultaneous failures. Using a larger group size to tolerate fewer than f failures is possible, but does not make sense because it requires a larger quorum size.

Inconsistent operations - replicated but unordered - return every result from all replicas that execute the operations. *Successful* inconsistent operations have executed at enough replicas to ensure that they are fault-tolerant.

Client Interface:
`InvokeInconsistentOperation`(*op*) →*results*
`InvokeConsensusOperation`(*op*) →*finalized,result*

Client State:
- *client id* - unique identifier for the client
- *operation counter* - Number of sent operations

Replica Upcalls:
`Execute`(*op*) →*result*
`Recover`(*op*)
`Recover`(*op,res*)

Replica State:
- *state* - denotes current replica state which is either NORMAL (processing operations) or VIEW-CHANGING (participating in recovery)
- *record* - unordered set of operations and their results

Figure 3: Summary of inconsistent replication interfaces and client/replica state.

Successful consensus operations have executed *with the same result* at a majority of replicas. A successful consensus operation is considered *finalized* when the client received enough *matching* results to also ensure that the result of the operation is not lost (i.e. the operation was executed with the same outcome at a supermajority of replicas). Consensus operations may fail because the replicas do not return enough matching results, e.g., if there are conflicting concurrent operations.

Each client keeps an *operation counter*, which, combined with the *client id*, uniquely identifies operations to replicas. Each IR replica keeps an unordered *record* of executed operations.

IR uses four sub-protocols - *operation processing*, *replica recovery*, *client recovery* and *group membership change*. This section details the first two only; the last two are identical to the VR [28] protocol.

3.2.1 Operation Processing Protocol

To begin, we describe IR's normal-case operation processing protocol without failures. Due to IR's weak guarantees, replicas *do not* communicate on each operation. Instead, the IR client simply contacts IR replicas and collects their responses. The protocol follows:

1. The client sends $\langle \text{REQUEST}, id, op \rangle$ to all replicas, where id is the message id (a tuple of the *client id* and *message counter*) and op is the operation.

2. Each replica writes *op* and its result to its record, then responds $\langle$REPLY, id, $res\rangle$, where res is the result of the operation.
3. For inconsistent operations, once the client receives $f+1$ responses from replicas, it returns all results to the application.
4. For consensus operations, once the client receives $f+1$ responses from replicas *with matching results*, it returns the result to the application. Once the client receives $\lceil \frac{3}{2} f \rceil + 1$ matching responses, it *finalizes* the result.

IR clients retry inconsistent operations until they succeed and consensus operations until they finalize or time out. Replicas that receive operations they have already executed can safely ignore them. A successful operation requires at least a single round-trip to $f+1$ replicas. However, IR needs $\lceil \frac{3}{2} f \rceil + 1$ matching responses to finalize a consensus operation. Otherwise, IR cannot guarantee the majority result will persist at the majority across failures. This quorum requirement is the same as Fast Paxos [23] and related protocols because IR must be able to determine both that a consensus operation succeeded *and* what was the result the replicas agreed upon.

Consensus operations can time out, either because the group does not reach consensus or because more than $\lfloor \frac{f}{2} \rfloor$ replicas are unreachable. TAPIR is designed to cope with this using a slow path that only relies on inconsistent operations.

3.2.2 IR Recovery Protocol

IR does not rely on synchronous disk writes, so failed replicas may lose their records of previously executed operations. IR's recovery protocol is carefully designed to ensure that a failed replica recovers any operation that it *may have executed previously and can still succeed*.

Without this property, successful IR operations could be lost. For example, suppose an IR client receives a quorum of responses and reports success to the application. Then, each replica in the quorum fails in sequence and each lose the operation, leading to the previously successful operation being lost by the group.

Ensuring this property is challenging because IR does not have a leader like VR. The VR leader tracks every operation that the group processes and every response from replicas. While recovering IR replicas can poll a quorum of other replicas to find successful operations, they cannot poll all of the clients to find operations they may have executed and their response. To deal with this situation, the recovering replica must instead ensure any operation that it executed but now cannot recover *does not succeed.*

For this purpose, we introduce *viewstamps* into the IR protocol. Each replica maintains a current *view number* and includes this view number in responses to clients[2]. Replicas also divide their records into *views* based on the view number when the replica executed the operation. Each IR replica can be in one of two states: either NORMAL or VIEW-CHANGING. IR replicas only process operations in the NORMAL state.

For an operation to be considered successful (or finalized), the IR client must receive responses with matching view numbers. On recovery, failed replicas force a view change, ensuring that any operation that has not already achieved a quorum in the previous view will not succeed. The recovery protocol follows:

1. The recovering replica sends $\langle$START-VIEW-CHANGE$\rangle$ to all replicas.
2. Each replica that receives a START-VIEW-CHANGE increments its current view number by 1 and then responds with $\langle$DO-VIEW-CHANGE, r, $v\rangle$, where r is its unordered record of executed operations, and v is the newly updated view number. The replica sets its state to VIEW-CHANGING and stops responding to client requests.
3. Once the recovering replica receives $f+1$ responses, it updates its record using the received records:
   (a) For any consensus operation with matching results in at least $\lceil \frac{f}{2} \rceil + 1$ of the received records (i.e., operations that were finalized), the replica calls `Recover`(*op,res*), where *res* is the operation with the majority result. It then adds the operation to its record.
   (b) For any consensus operation in a received record without a matching majority result (i.e., successful but not finalized), the replica recovers the operation using `Recover`(op) without a result. It then adds the operation to its record.
   (c) For any inconsistent operation in a received record, the replica calls `Recover`(*op*) and adds the operation to its record.

[2] If clients receive responses with different view numbers, they send back the largest view number received. This ensures that replicas are not forever stuck in different views, leaving the group unable to process operations.

4. The replica then sends a $\langle$START-VIEW, $v_{new}\rangle$, where v_{new} is the max of the view numbers from other replicas.
5. Any replica that receives START-VIEW checks whether v_{new} is higher than or equal to its current view number. If so, the replica updates its current view number and enters the NORMAL state; otherwise, it stays in its current state. The replica then always replies with a $\langle$START-VIEW-REPLY, $v_{new}\rangle$ message.
6. After the recovering replica receives f START-VIEW-REPLY responses, it enters the NORMAL state and resumes processing client requests. At this point, the replica is considered to be recovered.

If a non-recovering replica receives a START-VIEW-CHANGE without a matching START-VIEW, after a timeout it assumes the role of the recovering replica to force the group into the new view and back into the NORMAL state to continue processing operations.

3.3 Correctness

We give a brief *sketch of correctness*. Correctness requires that (1) successful operations are not lost, and (2) finalized consensus operation *results* are not lost. To ensure these guarantees, we show that IR maintains the following properties, to tolerate up to f simultaneous failures out of a total of $2f + 1$ replicas:

P1. Successful operations are always in at least one replica record out of any set of $f + 1$ replicas.

P2. Finalized consensus operation results are always in the records of a majority out of any set of $f + 1$ replicas.

In the absence of failures, the operation processing protocol ensures that any successful operation is in the record of at least $f + 1$ replicas and at least $\lceil \frac{3}{2}f \rceil + 1$ replicas executed any finalized consensus operation and recorded the same result. P1 follows from the fact that any two sets of size $f + 1$ share at least one replica. P2 follows from the fact that any two sets of sizes $\lceil \frac{3}{2}f \rceil + 1$ and $f + 1$, respectively, share at least $\lceil \frac{f}{2} \rceil + 1$ replicas, which constitute a majority of any set of size $f + 1$.

However, since replicas can lose records on failures, we must prove that the recovery protocol maintains these properties after failures as well. The recovery protocol ensures the following necessary properties will be met after recovery:

P3. A majority of the replicas, including the recovered replica, are in a view greater than that of any previously successful operation.

P4. The recovered replica gets all operations that were successful in previous views and any finalized results from previous views. Any ongoing operations are never successful.

These properties are sufficient to ensure the general properties, P1 and P2, stated above.

P3 is ensured by the view change protocol. Since at least 1 of the $f+1$ replicas that respond to START-VIEW-CHANGE must have the maximum view number in the group, and since every replica that responds to a START-VIEW-CHANGE will increment its view, the new view must be larger than any previous view.

P4 is ensured because any operations that were successful in views lower than the new view must appear in the joined records of any $f + 1$ replicas by quorum intersection. No further operations will become successful in any previous view because at least $f + 1$ replicas have incremented their view numbers prior to sending a DO-VIEW-CHANGE message. Any in-progress operations will therefore be unable to achieve a quorum. The equivalent holds for consensus operations with a finalized result because the quorum intersection for these is at least $\lceil \frac{f}{2} \rceil + 1$.

3.4 Building Atop IR

Due to its weak consistency guarantees, IR is an efficient protocol for replication. It has no leader to order operations, which increases throughput, and it has no cross-replica coordination, which reduces latency for each operation. In this respect, IR resembles protocols like Generalized Paxos [22] and EPaxos [32], which also do not require ordering for commutative operations. However, the goal in these protocols is still to maintain replicas in a consistent state while executing operations in different orders. IR has a completely different goal: to ensure fault-tolerance for operations and their results instead of consistent replication.

With replicas in inconsistent states, applications or protocols built on top of IR may be inconsistent, as well. IR does not attempt to synchronize inconsistent application state; the application must reconcile any inconsistencies caused by replicas executing operations in different orders. Section 4 describes how TAPIR does so. In addition, IR offers weak liveness guarantees for consensus operations.

For checkpointing purposes, replicas can periodically synchronize or gossip to find missed operations; however, this is not necessary for correctness. Without a

leader, no replica is guaranteed to have all of the successful operations. Thus, to get a complete view, applications must join the records of $f + 1$ replicas.

By carefully selecting the quorum size for inconsistent and consensus operations, IR provides sufficient fault-tolerance guarantees to be able to construct a higher-level protocol with strong consistency. Other systems could use IR with different quorum sizes to provide weaker consistency guarantees. For example, Dynamo could use IR with a quorum of all replicas for put operations to provide eventual consistency. We took advantage of this flexibility when building comparison systems for our evaluation.

4. TAPIR Overview

TAPIR is a distributed transaction protocol that relies on inconsistent replication. This section describes protocol and the techniques that TAPIR uses to guarantee strong consistency for transactions using IR's weak guarantees.

Figure 4 shows the messages sent during a sample transaction in TAPIR. Compared to the protocol shown in Figure 2, TAPIR has three immediately apparent advantages:

1. **Reads go to the closest replica.** Unlike protocols that must send reads to the leader, TAPIR sends reads to the replica closest to the client.
2. **Successful transactions commit in one round-trip.** Unlike protocols that use consistent replication, TAPIR commits most transactions in a single round-trip by eliminating cross-replica coordination.
3. **No leader needed.** Unlike protocols that order operations at a leader, TAPIR replicas all process the same number of messages, eliminating a bottleneck.

4.1 System Model

TAPIR is designed to provide distributed transactions for a scalable storage architecture. We assume a system that partitions data into *shards* and replicates each shard across a set of storage servers for availability and fault-tolerance. Clients are front-end application servers located in the same or another datacenter as the storage servers; they are *not* end-hosts or user machines. They can access a directory of storage servers and directly map data to them using a technique like consistent hashing [18].

TAPIR provides a general storage and transaction interface to clients through a client-side library. Clients begin a transaction, then read and write during the transaction's *execution period*. During this period, the client

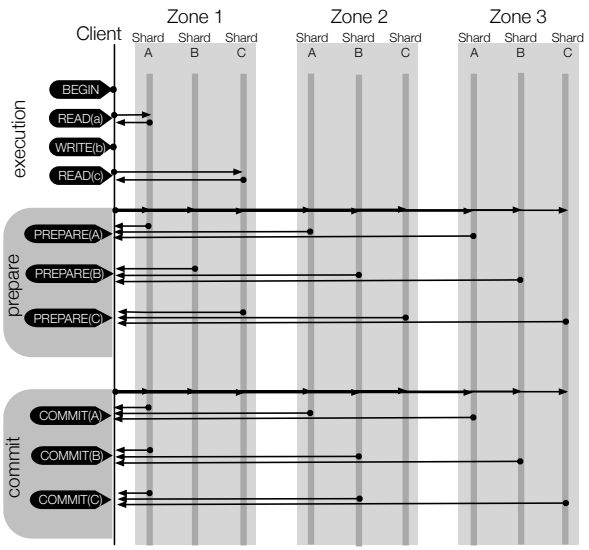

Figure 4: **TAPIR protocol.**

is free to abort the transaction. Once the client finishes execution, it commits the transaction, atomically and durably committing the executed operations across all shards that participated in the transaction. TAPIR ensures a linearizable ordering and fault-tolerance for the committed transaction for up to f simultaneous server failures and any client failures.

4.2 Protocol Overview

Figure 5 shows TAPIR's interfaces and state at both the client and the replica. Applications perform reads and writes within a transaction and then durably and atomically commit them.

Each TAPIR client supports one ongoing transaction at a time. In addition to the client's *client id*, the client stores the state for the ongoing *transaction*, including the transaction id and read and write set. TAPIR clients communicate with TAPIR replicas to read, commit, and abort transactions. TAPIR executes `Begin` operations locally and buffers `Write` operations at the client until commit; these operations need not contact replicas.

TAPIR replicas are grouped into shards; each shard stores a subset of the key space. `Read` operations are sent directly to any TAPIR replica in the appropriate shard because they do not require durability. TAPIR uses IR to replicate `Prepare`, `Commit` and `Abort` operations across replicas in each shard participating in the transaction (i.e., holding one of the keys read or written in the transaction).

Client Interface:
`Begin`()
`Read`(*key*)→*object*
`Write`(*key,object*)
`Commit`()
`Abort`()

Client State:
- *client id* - Unique client identifier
- *transaction* - ongoing transaction *id*, *read set*, *write set*

Replica Interface:
`Read`(*key*)→*object,version*
`Read`(*key,version*)→*object*
`Prepare`(*transaction,timestamp*)→
⟨PREPARE-OK||ABSTAIN||ABORT||RETRY,*timestamp*⟩
`Commit`(*transaction,timestamp*)
`Abort`(*transaction,timestamp*)

Replica State:
- *prepare transactions* - list of transactions that this replica is prepared to commit
- *transaction log* - log of committed and abort transactions in timestamp order
- *store* - versioned data store
- *log table* - table of transactions for which this replica group is the backup coordinator; includes the currently designated coordinator and the outcome of the transaction

Figure 5: Summary of TAPIR interfaces and client and replica state.

TAPIR uses *timestamp ordering* for transactions. A transaction's timestamp reflects its place in the global linearizable order. TAPIR replicas keep transactions in a *transaction log* in timestamp order; they also maintain a multi-versioned *data store*, where each version of an object is identified by the timestamp of the transaction that wrote the version. TAPIR replicas serve reads from the versioned data store and maintain the transaction log for synchronization and checkpointing. Like other two-phase commit protocols, TAPIR replicas maintain a list of *prepared transactions*.

TAPIR uses two-phase commit (2PC) with optimistic concurrency control (OCC) to commit transactions at a single timestamp. The TAPIR client also serves as the two-phase commit coordinator. TAPIR uses an *optimistic ordering* approach for committing transactions, similar to CLOCC [1]. Clients choose a *propose timestamp* for the committing transaction in `Prepare`. TAPIR replicas then use OCC to validate that the transaction is consistent with the proposed timestamp.

Unlike other two-phase commit protocols, TAPIR replicas can respond to `Prepare` operations in four ways. PREPARE-OK and ABORT are the usual two-phase commit responses, indicating that no conflicts were found or a conflict with a committed transaction was found, respectively. The other two responses are designed to reduce the number of aborts in TAPIR. ABSTAIN indicates that the replica cannot prepare the transaction at this time, which usually occurs when finding a conflict with a *prepared* transaction. RETRY indicates that the transaction might succeed if the client uses the *retry timestamp* returned with the result.

`Prepare` is a consensus operation since replicas can return different results; `Commit` and `Abort` are inconsistent operations. The latter are carefully designed to enable TAPIR replicas to consistently commit or abort transactions at any time.

Because `Prepare` is a consensus operation, it may not finalize, meaning the result will not survive failures. However, TAPIR must still be able to make progress and eventually commit or abort the transaction. TAPIR copes with this situation using a *slow path*, which writes the transaction outcome to a *backup coordinator group* before reporting it to the application. Once TAPIR replicates the outcome across the backup coordinator group, replicas can determine the result of the `Prepare` from the transaction outcome.

4.3 Building TAPIR on an IR Foundation

Numerous aspects of TAPIR are explicitly designed to cope with inconsistent replication. These design techniques make it possible for TAPIR to provide linearizable transactions from IR's weak guarantees. TAPIR must: (1) order transactions using unordered operations, (2) detect conflicts between transactions with an incomplete transaction history, and (3) reconcile different operation results from inconsistent replicas and application state at inconsistent replicas. This section highlights how TAPIR tackles these challenges.

4.3.1 Ordering Transactions.

The first challenge is how to order transactions when the replication layer provides only unordered operations. Conventional transaction protocols use the order of operations at storage servers to order transactions, but this cannot work with inconsistent replication.

Instead, TAPIR uses an optimistic transaction ordering technique, similar to the one employed in CLOCC [1], where *clients* propose timestamps to order their transactions. To help clients pick timestamps, TAPIR uses loosely synchronized clocks. As other work has noted [38], the Precision Time Protocol (PTP) [39] makes clock synchronization easily achievable in today's datacenters. With hardware support, this protocol can achieve submicrosecond accuracy. Even without hardware support, PTP lets us achieve microsecond accuracy, far less than TAPIR requires. Across the wide-area, we found that sub-millisecond accuracy was possible in VMs on the widely available Google Compute Engine [16].

TAPIR's performance depends on clients proposing closely clustered timestamps. Increasing clock skew increases the latency for committing transactions since clients must retry their transactions. Retrying does not force the transaction to re-execute; it incurs only an additional round-trip to the replicas. As our evaluation shows, TAPIR is robust even to high clock skews; we measured a retry rate that was well below 1% even with clock skews of several milliseconds and a high-contention Zipf distribution of accesses. As a rough metric, TAPIR sees no negative performance impact as long as the clock skew stays within a few percent of the latency between clients and servers.

It is important to note that TAPIR does not depend on clock synchronization for correctness, only performance. This differs from Spanner [8], where consistency guarantees can be violated if the clock skew exceeds the uncertainty bound given by TrueTime. Thus, TAPIR's performance depends only on the actual clock skew, unlike Spanner, which must wait for a conservatively estimated uncertainty bound on every transaction.

4.3.2 Detecting Conflicts.

Conventional distributed transaction protocols use concurrency control to prevent conflicts between concurrent transactions. However, most concurrency control protocols check a transaction for conflicts against *all previous transactions*. Every TAPIR replica might have a different set of transactions, so TAPIR must detect conflicts between transactions with an incomplete transaction history.

TAPIR solves this problem using optimistic concurrency control and quorum intersection. OCC validation checks occur between the committing transaction and *one* previous transaction at a time. Thus, it not necessary for a single server to perform all the checks. Since IR ensures that every `Commit` executes at at least 1 replica in any set of $f+1$ replicas, and every `Prepare` executes at at least $\lceil \frac{f}{2} \rceil$ replicas in any set of $f+1$, at least one replica will detect any possible OCC conflict between transactions, thus ensuring correctness.

4.3.3 Reconciling Inconsistency.

TAPIR must reconcile both inconsistent replica state and inconsistent operation results.

The result of `Commit` and `Abort` is the same regardless of their execution order. The replica state also stays consistent. For `Commit`, the replica will always commit the transaction to the multi-versioned data store using the transaction timestamp and then log the transaction. For `Abort`, the replica will abort the transaction if it is prepared and then log the transaction.

Both the result and replica state after a `Prepare` differ if they are executed in a different order at each replica. We cope with this inconsistency as follows. First, replica state diverges only until the replicas all execute the `Commit` or `Abort`, so it is not important to reconcile replica state. Second, TAPIR manages inconsistent results by using inconsistent operations in IR. Inconsistent operations always return the *majority* result and preserve that result across failures. Thus, TAPIR can rely on the majority result to decide whether to commit or abort a transaction. As a further optimization, TAPIR replicas give an ABSTAIN result when they are unsure of the outcome of the conflicting transaction; this prevent transactions from aborting unnecessarily due to inconsistency.

5. TAPIR Protocol

This section details the TAPIR protocol, which provides linearizable transactions using inconsistent replication. TAPIR provides the usual atomicity, consistency, isolation and durability guarantees for transactions. It also supports the strictest level of isolation: external consistency or linearizability. More specificially, TAPIR's transaction timestamps reflect an externally consistent ordering, ensuring that if transaction A commits before transaction B, then A will have a lower timestamp than B.

TAPIR provides these guarantees using a *transaction processing protocol* layered on top of inconsistent replication. It also relies on inconsistent replication for replica recovery; however, TAPIR uses a *coordinator recovery protocol* for two-phase commit coordinator fail-

ures. The remainder of this section describes these protocols in detail.

5.1 Transaction Processing Protocol

TAPIR clients communicate with TAPIR replicas to coordinate linearizable transactions. TAPIR replicas process transactions unless replicas in the shard are in IR recovery mode. This section describes the protocols for reading objects in a transaction and then committing.

5.1.1 Read Protocol

TAPIR's read protocol is identical to other optimistic concurrency control protocols for sharded multi-versioned data stores. The TAPIR client sends read requests to any replica server in the shard for that data object. Its server returns the latest version of the object along with the timestamp of the transaction that committed that version, which serves as the version ID. The client adds the key and the version ID to the transaction's *read set*. If the client has previously read the object in the transaction, it can retrieve the same version by sending the version ID and with the read request to the server.

Unlike other optimistic concurrency control protocols, TAPIR servers are *inconsistent.* Any replica could miss any transaction at any time. However, the OCC validation process detects any inconsistencies on commit and the transaction will abort.

5.1.2 Commit Protocol

During the execution period, the TAPIR client accumulates the read and write sets for the transaction. Once the application decides to commit, TAPIR starts the commit protocol to find a timestamp at which it can serialize the transaction's reads and writes.

The TAPIR client starts the protocol by selecting a *proposed timestamp*. Proposed timestamps must be unique, so clients use a tuple of their local time and their *client id*. The TAPIR client serves as the two-phase commit coordinator, sendind the `Prepare` with the proposed timestamp to all participants.

The result of `Prepare` at each replica depends on the outcome of the *TAPIR validation checks.* As noted, TAPIR validation checks have four possible outcomes.

1. The replica checks for conflicts with other transactions. For each read, it checks that the version remains valid (i.e. that it has not accepted a write for that key before the proposed timestamp. For each write, it checks that it has not accepted a read or write for that key after the proposed timestamp.[3]
2. If there are no conflicts, the participant replica adds the transaction to the list of prepared transactions and replies PREPARE-OK.
3. If a read conflict exists with a previously committed transaction, then the participant replica cannot commit the transaction; it sends an ABORT.
4. If a write conflict exists with a previously committed transaction, the participant replies RETRY with the timestamp of the conflicting transaction. If the client retries the transaction at a timestamp after the conflicting transaction, it may commit.
5. If a conflict exists with a *prepared* transaction, the participant replica replies ABSTAIN.

If `Prepare` succeeds in each participant shard, the TAPIR client proceeds as follows:

1. If the result is PREPARE-OK at every shard, then the client waits for IR to finalize the `Prepare` result. As an optimization, while it waits, it can begin a slow-path commit (described below).
2. If the result at any shard is ABORT, then the client sends `Abort` to all participants.
3. If the result at any shard is RETRY, then client retries the transaction with a new proposed timestamp: the maximum of the returned retry timestamps and the local time.
4. Otherwise, the client retries the transaction with a new proposed timestamp.

Once IR declares the `Prepare` result to be finalized in every participant shard, the TAPIR client reports the outcome to the application. If the `Prepare` times out without finalizing the result in one of the participant shards, then the TAPIR client must take a slow path to finish the transaction. If `Prepare` succeeded with PREPARE-OK in every shard, the client commits the transaction, otherwise it aborts. To complete the transaction, the client first logs the outcome of the transaction to the backup coordinator group. It then notifies the client, and sends a `Commit` or `Abort` to all participant replicas. The client uses the same slow path to abort the transaction if `Prepare` does not succeed in any shard because there were not enough matching responses.

[3] These conflict checks ensure serializability and external consistency; they could be relaxed slightly (e.g., following the Thomas Write Rule [42]) for higher performance if external consistency is not required.

5.1.3 Out-of-order Execution

Since TAPIR relies on inconsistent replication, it must be able to execute operations out-of-order. For example, TAPIR may receive the `Commit` for a transaction before the `Prepare`. Thus, in addition to normal-case execution, TAPIR performs the following checks for each operation:

- `Prepare`: If the transaction has been committed or aborted (logged in the transaction log), ignore. Otherwise, TAPIR validation checks are run.
- `Commit`: Commit the transaction to the transaction log and update the data store. If prepared, remove from prepared transaction list.
- `Abort`: Log abort in the transaction log. If prepared, remove from prepared list.

5.2 Recovery Protocols

TAPIR supports up to f simultaneous failures per shard as well as arbitrary client failures. Because it uses the client as the two-phase commit coordinator, TAPIR must have protocols for recovering from both replica and coordinator failures.

5.2.1 Replica Recovery

On replica recovery, IR recovers, and executes operations in an arbitrary order, and gives any finalized consensus operation the result. TAPIR already supports out-of-order execution of `Commit` and `Abort`, so it need not implement additional support for these operations.

For finalized `Prepare` operations, TAPIR skips the validation checks and simply prepares any transaction where the finalized result was PREPARE-OK.

For successful, but unfinalized `Prepare` operations, TAPIR adds the transaction to the prepared list, but it returns ABSTAIN if queried about the transaction. These transactions exist in an in-between state until either committed or aborted. TAPIR does this because the transaction might still commit; however, the recovering replica does not know if it returned PREPARE-OK originally, so it cannot participate in committing the transaction.

5.2.2 Coordinator Recovery

TAPIR uses the client as coordinator for two-phase commit. Because the coordinator may fail, TAPIR uses a group of $2f + 1$ backup coordinators for every transaction. Coordinator recovery uses a *coordinator change protocol*, conceptually similar to Viewstamped Replication's view change protocol. The currently active backup coordinator is identified by indexing into the list with a coordinator-view number; it is the only coordinator permitted to log an outcome for the transaction.

If the current coordinator is suspected to have failed, a backup coordinator executes a view change in the coordinator group. In doing so, it receives any slow-path outcome that was logged by a previous coordinator. If such an outcome has been logged, the new coordinator must follow that decision; it notifies the client and all replicas in every shard.

If the new coordinator does not find a logged outcome, it sends a `CoordinatorChange`(*transaction*, *view-num*) message to all replicas in participating shards. Upon receiving this message, replicas agree not to process messages from the previous coordinator; they also reply to the new coordinator with any previous `Prepare` result for the transaction. Once the `CoordinatorChange` is successful (at $f+1$ replicas in each participating shard), the new coordinator determines the outcome of the transaction in the following way:

- If any replica in any shard has recorded a `Commit` or `Abort`, it must be preserved.
- If any shard has less than $\lceil \frac{f}{2} \rceil + 1$ PREPARE-OK responses, the transaction could not have committed on the fast path, so the new coordinator aborts it.
- If at least $\lceil \frac{f}{2} \rceil + 1$ replicas in every shard have PREPARE-OK responses, the outcome of the transaction is uncertain: it may or may not have committed on the fast path. However, every conflicting transaction must have used the slow path. The new coordinator polls the coordinator (or backup coordinators) of each of these transactions until they have completed. If those transactions committed, it aborts the transaction; otherwise, it sends `Prepare` operations to the remaining replicas until it receives a total of $f + 1$ PREPARE-OK responses and then commits.

The backup coordinator then completes the transaction in the normal slow-path way: it logs the outcome to the coordinator group, notifies the client, and sends a `Commit` or `Abort` to all replicas of all shards.

5.3 Correctness

We give a brief sketch of how TAPIR maintains the following properties given up to f failures in each replica group and any number of client failures:

- **Isolation.** There exists a global linearizable ordering of committed transactions.
- **Atomicity.** If a transaction commits at any participating shard, it commits at them all.

- **Durability.** Committed transactions stay committed, maintaining the original linearizable order.

Isolation. Because `Prepare` is a consensus operation, a transaction X can be committed only if $f + 1$ replicas return PREPARE-OK. We show that this means the transaction is consistent with the serial timestamp ordering of transactions.

If a replica returns PREPARE-OK, it has not prepared or committed any conflicting transactions. If a conflicting transaction Y had committed, then there is one common participant shard where at least $f + 1$ replicas responded PREPARE-OK to Y. However, those replicas would not return PREPARE-OK to X. Thus, by quorum intersection, X cannot obtain $f + 1$ PREPARE-OK responses.

By the same reasoning, ABORT is just an optimization. If a replica detects a conflict with committed transaction Y, then it may safely return ABORT and know that at least $f + 1$ replicas will give matching results. At least $f+1$ replicas must have voted to commit Y, and at least $f + 1$ replicas processed the `Commit` operation. Thus, it knows that X will never get a PREPARE-OK result.

This property remains true even in the presence of replica failures and recovery. IR ensures that any `Prepare` that succeeded with a PREPARE-OK persists for up to f failures. It does not ensure that the result is maintained. However, our recovery protocol for `Prepare` without a result is conservative and keeps the transaction in the prepared state in case it commits. This ensures that other new transactions that conflict cannot acquire the necessary quorum.

Atomicity. A transaction commits at a replica only if a coordinator has sent the `Commit` operation after a majority of replicas in each participating shard have agreed to `Prepare` the transaction. In this case, the coordinator will send `Commit` operations to all shards.

If the coordinator fails after deciding to commit, the coordinator recovery protocol ensures that the backup coordinator makes the same decision. If the previous coordinator committed the transaction on the slow path, the commit outcome was logged to the backup coordinators; if it was committed on the fast path, the new coordinator finds the quorum of PREPARE-OK results by polling $f + 1$ replicas in each shard.

Durability. If there are no coordinator failures, a transaction is eventually finalized through an IR inconsistent operation (Commit/Abort), which ensures that the decision is never lost. As described above, for coordinator failures, the coordinator recovery protocol ensures that the coordinator's decision is not lost.

6. TAPIR Extensions

We now describe two useful extensions to TAPIR.

6.1 Read-only Transactions.

Since it uses a multi-versioned store, TAPIR easily supports interactive read-only transactions at a snapshot. However, since TAPIR replicas are inconsistent, it is important to ensure that: (1) reads are up-to-date and (2) later write transactions do not invalidate the reads. To achieve these properties, TAPIR replicas keep a read timestamp for each object.

TAPIR's read-only transactions have a single round-trip fast path that sends the `Read` to only one replica. If that replica has a *validated version* of the object where the write timestamp precedes the snapshot timestamp and the read timestamp follows the snapshot timestamp we know that the returned object is valid, because it is up-to-date, and will remain valid, because it will not be overwritten later. If the replica lacks a validated version, TAPIR uses the slow path and executes a `QuorumRead` through IR as an inconsistent operation. A `QuorumRead` updates the read timestamp, ensuring that at least $f + 1$ replicas do not accept writes that would invalidate the read.

The protocol for read-only transactions follows:

1. The TAPIR client chooses a *snapshot timestamp* for the transaction; for example, the client's the local time.
2. The client sends `Read`(*key,version*), where *key* is what the application wants to read and *version* is the snapshot timestamp.
3. If the replica has a validated version of the object, it returns it. Otherwise, it returns a failure.
4. If the client was not able to get the value from the replica, the client executes a `QuorumRead`(*key,version* through IR as an inconsistent operation.
5. Any replica that receives `QuorumRead` returns the latest version of the object from the data store. It also writes the read to the transaction log and updates the data store to ensure that it will not prepare for transactions that would invalidate the read.
6. The client returns the object with the highest timestamp to the application.

As a quick sketch of correctness, it is always safe to read a version of the key that is *validated* at the snapshot

timestamp. Because the write timestamp for the version is before the snapshot timestamp, the version was valid at the snapshot timestamp, and because the read timestamp is after the snapshot timestamp, the version will *continue* being valid at the snapshot timestamp. If the replica does not have validated version, the replicated `QuorumRead` ensure that both that the client gets the latest version of the object (because it must be at at least 1 of any $f + 1$ replicas) and that a later write transaction cannot overwrite the version (because the `QuorumRead` is replicated at $f + 1$ replicas).

Since TAPIR also uses loosely synchronized clocks, TAPIR can be combined with Spanner's algorithm for providing linearizable read-only transactions. Like Spanner, this combination would require waits at the client, who acts as the coordinator in TAPIR, for the uncertainty bound and only provides linearizability guarantees as long as the clock skew does not exceed that bound.

6.2 Synchronization.

Individual TAPIR replicas may be missing some committed transactions. While these "holes" do not affect TAPIR's correctness, storage systems using TAPIR may prefer to have a full transaction log at certain points to checkpoint state to durable storage. Synchronization in TAPIR requires that replicas write *sync timestamps* into their transaction log, and track their latest *sync point*. Replicas do not accept prepares for any transaction with a timestamp before the sync point.

TAPIR implements synchronization by sending a To ensure that `Sync` and `FinishSync` operations are fault-tolerant, we send then through IR as inconsistent operations. Since IR can reorder `Sync` and `FinishSync` operations, we send timestamp ranges for the synchronization with each operation.

1. A synchronizing replica sends `Sync`(*start, end*) to the replica group through IR, where $t_{start} - t_{end}$ is the portion of time the replica would like to have the full log. For example, the replica can pick *start* to be the its local time at the last sync and *end* to be its current time.
2. Other replicas execute `Sync` by returning all of their transactions with timestamps between *start* and *end*. If *t_end* is bigger than the replica's last sync point, then the replica updates its sync point to *end* and stops taking transaction with smaller timestamps.
3. Once the synchronizing replica has received f results from IR, the replica joins the transaction logs with its own and sends the complete log back in `FinishSync`(*start,end,log*). It updates its transaction log and data store by committing any transaction in *log* not already its log.
4. Any replica that receives `FinishSync` updates their sync point if it has not already. The replica also updates its own transaction log and data store by committing every transaction in *log* that is not already in its log.

At this point, every replica that participated in the synchronization and has a new sync timestamp in its log is guaranteed to have all of the transactions that the group committed between *start* and *end*.

6.3 Serializability.

TAPIR is somewhat limited in its ability to accept transactions out of order because it provides linearizability. Thus, TAPIR replicas cannot accept writes that are older than the last write that they accepted for the same key and they cannot accept reads of older versions of the same key.

However, if TAPIR's guarantees are slightly weakened to provide *serializability*, TAPIR can then accept proposed timestamps any time in the past as long as they respect serializable transaction ordering. This optimization requires tracking the timestamp of the transaction that last *read* to each version, as well as the timestamp of the transaction that wrote each version.

With this optimization, TAPIR can now accept: (1) reads of past versions, as long as the read timestamp is before the write timestamp of the next version, and (2) writes in the past (Tomas Write Rule), as long as the write timestamp is after the read timestamp of the previous version and before the write timestamp of the next version.

6.4 Synchronous log writes.

Given the ability to synchronously log to durable storage (e.g. hard disk, NVRAM), we can reduce the quorum requirements for TAPIR. As long as we can recover the log after failures, we can reduce the replica group size to $2f + 1$ and reduce all consensus and synchronization quorums to $f + 1$.

6.5 Retry Timestamp Selection.

A client can increase the likelihood that the participant replicas will accept its proposed timestamp by proposing a very large timestamp, as this decreases the chance that the participant replicas have already accepted a higher

timestamp. Thus, to decrease the chances that clients will have to retry forever, clients can exponentially increase their proposed timestamp on each retry.

6.6 Tolerating Very High Skew.

If there is significant clock skew between servers and clients, TAPIR can use waits at the participant replicas to decrease the likelihood that transactions will arrive out of timestamp order. On receiving each prepare message, the participant replica can wait (for the error bound period) to see if other transactions with smaller timestamps arrive. After the wait, the replica can process the transaction in timestamp order with respect to the other transactions that have arrived. This wait increases the chances that the participant replica will receive transaction in timestamp order and decreases the number of transactions that it will have to reject for arriving out of order.

7. Evaluation

In order to evaluate the effectiveness of TAPIR, we measure an implementation of TAPIR against several conventional designs of both transactional and weakly consistent storage systems. Our evaluation compares and analyzes their performance in local-area and wide-area environments.

7.1 Experimental Setup

We implemented TAPIR in a transactional key-value store, called *TAPIR-KV*. TAPIR-KV consists of 4,921 lines of C++ code, not including the testing framework. We also built five comparison systems and a workload generator based on the Retwis [24] benchmark. We ran our experiments in three testing environments: single cluster, single datacenter and wide-area, cross-datacenter.

Test Systems We compare TAPIR-KV with five systems:

- *Txn-Locking* - standard two-phase commit with strict two-phase locking (S2PL) and Viewstamped Replication (VR).
- *Txn-OCC* - standard two-phase commit with optimistic concurrency control (OCC) and VR.
- *Span-Locking* - our implementation of the Spanner protocol.
- *Span-OCC* - our implementation of the Spanner protocol with OCC, instead of S2PL.
- *QWStore* - an eventual consistency system with no transactions (e.g., Cassandra [20], Dynamo [11]) built with new IR operations that implement a read anywhere, write everywhere policy.

The transactional systems buffer writes at the client until commit and use the client as the coordinator.

Test Environment We ran our TAPIR experiments with three setups:

- *Single cluster* - 10 quad-core servers with 24 GB RAM, running Ubuntu 12.04 connected to a fat-tree network with 12 switches.
- *Single datacenter* - 15 Google Compute Engine VMs with two cores running Debian 7 in three availability zones in the same geographic region (US-Central).
- *Cross-datacenter* - 6 Google Compute Engine VMs running Debian 7 in three geographic regions (US, Europe and Asia) connected over the wide-area network.

These three deployments offer round-trip times that differ by three orders of magnitude: 100 µs, 1 ms and 100-200 ms. For most experiments, we used 3 replicas per shard with replicas placed across availability zones or geographic regions.

Within our cluster, we synchronized clocks using the Linux PTP [39] kernel module without hardware support. For our two Google Compute Engine deployments, we use Google's existing VM clock synchronization mechanism. Lacking access to Spanner's TrueTime clock skew error bounds, we treated them as negligible compared to the network latency.

7.2 Latency and Clock Skew Measurements

We first present a profile of our different test environments. Since TAPIR depends on loosely synchronized clocks, we measured clock skews within a cluster using PTP, as well as between Google Compute Engine VMs around the world. We measure clock skew by sending a user-level to user-level ping message, with timestamps taken on either end. We calculate skew by by subtracting the timestamp taken at the destination from $\frac{rtt}{2}$ and assuming symmetric latencies.

Within our cluster, the machines had a 150 µs round-trip time between them and we were able to synchronize the servers with an average skew of 6 µs using PTP. We report the average skew because TAPIR's performance depends on the average, not worst-case, skew.

Table 1 reports the average skew and latency between Google Compute Engine's three geographic regions. Within each region, we average over the availability

Table 1: Cluster round-trip times and clock skews between Google Compute VMs.

	Latency (ms)			Clock Skew (ms)		
	US	**Europe**	**Asia**	**US**	**Europe**	**Asia**
US	1.2	111.3	166.5	3.4	1.3	1.86
Europe	111.0	0.8	261.8	2.1	0.1	1.9
Asia	166.7	263.2	10.8	2.6	1.8	.3

zones. Overall, the clock skew is low, demonstrating the feasibility of synchronizing clocks in the wide area.

Google gives Compute Engine VMs access to Google's reliable wide-area network infrastructure. We implement all communication using UDP and saw few packet drops and little variation in round-trip time. However, there was a long tail to the clock skew. The worst case clock skew that we saw was 27 ms, indicating that Spanner may sometimes violate consistency if they find TrueTime error bounds to only rise to 5 ms between synchronizations.

7.3 Microbenchmarks

For the microbenchmarks, we deployed our six test systems with a single shard with 3 replicas. We use the same number of replicas for all systems for a fair throughput comparison, but TAPIR-KV needs replies from all 3 replicas for its fast path commit, while the other transactional storage systems – TXN-LOCK, TXN-OCC, SPAN-LOCK, SPAN-OCC – only need replies from 2 out of 3 replicas. QWStore must wait for replies from all replicas for writes.

Our microbenchmark consists of single-key, read-modify-write transactions with a uniform distribution of accesses across 1 million keys. This workload stresses the transactional systems because the short transactions have high coordination overhead.

Latency. Figure 6 gives latencies for all six systems deployed in a single cluster, single datacenter and cross-datacenter. Within a cluster, TAPIR provides transactions at the same cost as the non-transactional QWStore; both systems require one round trip to read and another round trip to all replicas to either write or commit the transaction. The other transaction systems have lower performance because they need two round-trips to commit, with TXN-OCC being the slowest because it also needs a round-trip to a timestamp servers.

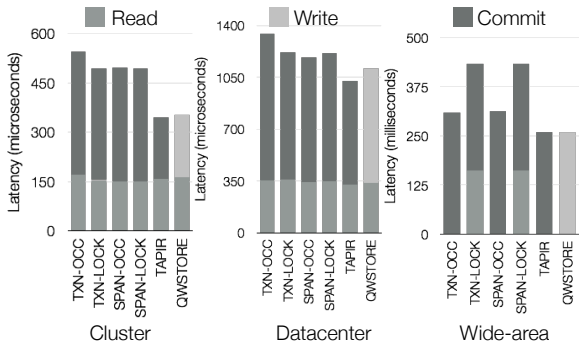

Figure 6: Latency for read-modify-write transaction in our three test environment. For the wide-area deployment, we place the leader replica in a different datacenter from the client.

Moving to VMs in a datacenter, the computation cost outweighs the communication cost, so TAPIR-KV's fewer round-trips have less impact on latency. TAPIR still matches the performance of QWStore, but, because the round-trip time is only 1ms between nodes, TAPIR and QWStore only offer slightly improved performance over the four transactional storage systems.

With VMs in different datacenters, the communication cost outweighs the computation cost. For these experiments, we place the leader for the shard in the US with the clients in Asia. The advantage of OCC compared to locking becomes visible. Being able to read at the closest replica, rather than having to go to the leader to acquire locks reduces the cost of the OCC systems. TAPIR-KV offers better performance than the other OCC systems because it only needs a single round-trip to all of the replicas to commit the transaction, compared to two round-trips to the two closest replicas for the other systems. And again, TAPIR-KV matches the performance of the non-transactional QWStore.

Throughput. While TAPIR's latency benefits are reduced In our datacenter deployment, TAPIR still provides throughput benefits. As shown in Figure 7, TAPIR provides 2.6× the throughput of other transactional systems because it is leader-less. Each TAPIR-KV replica processes the same number of messages, so all three replicas run at peak throughput. In comparison, the other transactional systems bottleneck on the leader, which must coordinate replication. We do not show QWStore, but it provides roughly 2x the throughput of TAPIR. QWStore has higher throughput because it does not provide atomicity.

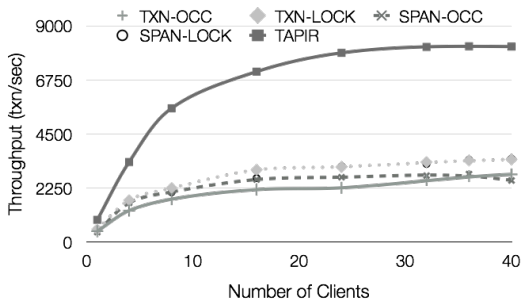

Figure 7: Throughput for read-modify-write microbenchmark deployed in Google VMs in a single datacenter.

7.4 Retwis Experiments

For our application experiments, we deploy 5 shards of 3 replicas using Google Compute VMs. We generate a synthetic workload based on the Retwis application and similar to YCSB-T [12]. Retwis is an open-source Twitter clone designed to use the Redis key-value store. Retwis has a number of Twitter functions (e.g., add user, post tweet, get timeline, follow user) that perform puts and gets on Redis. We treat each function as a transaction, and generate a synthetic workload based on the Retwis functions. For example, the `FollowUser` transaction has 2 reads and 2 puts. We use 1 million keys distributed across the 5 shards andvary the access distribution (e.g., uniform, Zipf) to vary the level of contention on different keys.

7.4.1 Datacenter Performance

To measure performance within a datacenter, we deploy our VMs across availability zones in the US-Central region. All experiments use the Retwis workload with a Zipf co-efficient of 0.6 to approximate the contention in real-life workloads.

Latency. Figure 8 reports the average latency for the `FollowUser` transaction at peak throughput for the six systems. Note again that the computation costs outweigh the communication costs in the datacenter with VMs, so TAPIR-KV provides less latency benefit than in other environments. TAPIR-KV still provides lower latency than other transaction systems, except for SPAN-OCC; however, SPAN-OCC achieves the same latency as TAPIR-KV, but with half the peak throughput. QWStore provides lower latency at higher peak throughput than all of the transactional systems however because it can avoid coordinating transactions with two-phase commit.

Throughput. As Figure 9 shows, the peak throughput for TAPIR-KV is twice that of the other transactional storage systems because TAPIR-KV does not need a

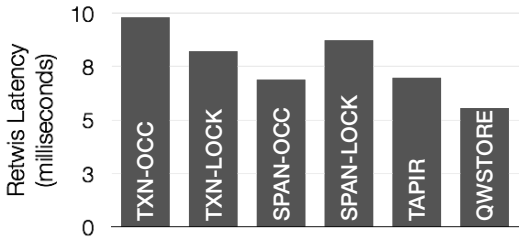

Figure 8: Latency for `FollowUser` transaction with Zipf distribution (coeff=0.6) at peak throughput for each system.

leader to coordinate between replicas in a shard. However, QWStore achieves twice the throughput of TAPIR-KV because it does not need to coordinate replicas within a shard or transactions across shards.

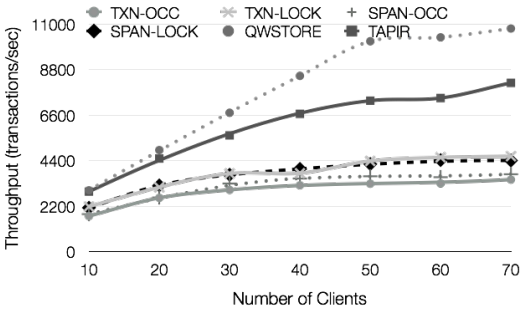

Figure 9: Peak Throughput for Retwis Transactions with Zipf distribution.

7.4.2 Cross-datacenter experiments

For wide-area experiments, we placed one replica from each shard in each geographic region. For the systems that use consistent replication and have a leader, we fix the location of the leader in the US and move the client between the US, Europe and Asia. Figure 10 gives latency results for the `FollowUser` transaction in the Retwis workload. As before, we use a Zipf distribution with a 0.6 co-efficient.

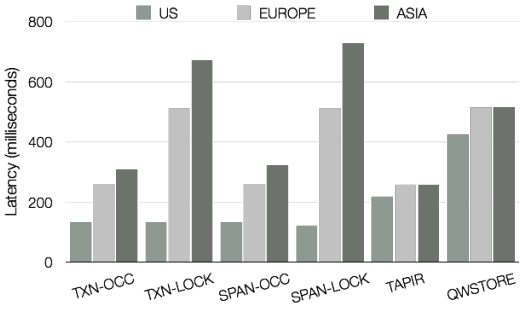

Figure 10: Wide-area latency for `FollowUser` transaction, with leader in the US and client in US, Europe or Asia.

When the client shares a datacenter with the leader, the latency of the other transactional systems is better

than TAPIR-KV and QWStore. The other transactional systems are faster because they can commit with a round-trip to the leader and another replica in the *closest* geographic region, while TAPIR-KV must contact all replicas to commit in the fast path. QWStore is the slowest of all because it does not buffer writes, so it needs a round-trip to all of the replicas on each of the two write operations in the `FollowUser` function.

When the client is in a different datacenter from the leader, TAPIR-KV provides the best performance of all of the comparison systems. The locking-based transactional systems suffer because they must go to the leader for read locks. QWStore has better performance because it can read at any replica; however, it still takes two round-trips to write. The OCC-based transactional storage systems provide the best performance in this case; they all buffer writes and read at a replica in the local datacenter. However, TAPIR can commit the transaction in a single round-trip to all replicas, compared to two round-trips to the two closest replica for the other OCC systems.

8. Related Work

TAPIR is the first distributed transaction protocol to support a replication protocol with *no consistency guarantees*. It uses no writes to disk, no replica leader and no coordination between replicas on commit to provide extremely high-performance read-write transactions. This section discusses other work with similar goals and mechanisms.

8.1 Replication

Many people [23, 25, 31] have worked on providing efficient replication with strong consistency guarantees. In particular, significant work has focused on supporting commutative operations in consistent replication protocols [6, 22, 32]. Commutative operations do not require a consistent ordering across replicas, so they can execute without a leader or cross-replica coordination. Inconsistent replication supports only unordered operations, so all operations commute by definition.

Dynamo [11] and other systems [10, 13, 14, 26, 40, 43] chose replication techniques with weak consistency guarantees for performance. However, weak consistency guarantees can be hard to understand and programmers find them increasingly difficult to use for satisfying many applications' complex requirements [8]. TAPIR offers the best of both worlds: inconsistent replication provide performance on par with weak consistency systems, but TAPIR provides strong consistency guarantees in the form of linearizable transactions to application programmers.

The inconsistent replication protocol shares many features with Viewstamped Replication [34]. Like VR, inconsistent replication is designed for in-memory replication without relying on synchronous disk writes. The possibility of data loss on replica failure, which does not happen in protocols like Paxos [21] that assume durable writes, necessitates the use of viewstamps for both replication protocols. Our decision to focus on in-memory replication is motivated by the popularity of recent in-memory systems like RAMCloud [35] and H-Base [41].

8.2 Distributed Transactions

There has been significant work in improving the performance of distributed transactions by limiting the transaction model [2, 44] or consistency guarantees [29, 40]. This work largely assumes durable writes and does not consider replication.

Like TAPIR, Granola [9] uses loosely synchronized clocks; however, it provide higher performance only for independent transactions and can only be used with a consistent replication protocol. Other transactional systems have looked at commutative transactions [19] or executing transactions out-of-order [33]. TAPIR naturally supports preparing and executing commutative or non-conflicting transactions in any order due to its use of optimistic transaction ordering and multi-versioned storage to support inconsistent replication.

The CLOCC [1] distributed transaction protocol also uses loosely synchronized clocks for optimistic transaction ordering; however, it did not consider replication. CLOCC was later combined with VR [27], but does not achieve the same performance as TAPIR because VR enforces strong consistency guarantees. Gray and Lamport [17] use of Paxos [21] with locking and two-phase commit has similar performance limitations.

More recently, numerous systems [3, 7, 8, 15, 19, 36, 37, 44] have implemented distributed transactions with different consistent replication protocols; however, they are all limited by their need for cross-replica coordination. Spanner [8] uses loosely synchronized clocks to address latency and throughput limitations in replicated transactional systems for read-only transactions, but uses a standard protocol for read-write transactions. Its approach for read-only transactions is complementary to TAPIR's high-performance read-write transac-

tions and the two could be easily combined. Other systems improve latency for read-write transactions (e.g., MDCC [36], Replicated Commit [30] for wide-area environments) or throughput (e.g., Warp [15] for datacenter environments), none address latency and throughput for a wide range of environments like TAPIR. TAPIR

9. Conclusion

This paper presented TAPIR, the first distributed transaction protocol designed to use inconsistent replication. TAPIR uses three techniques that use loosely synchronized clocks and the properties of optimistic concurrency control and two-phase commit to provide linearizable transactions with inconsistent replication. We designed and built TAPIR-KV, a distributed transactional key-value store, along with several comparison systems. Our experiments reveal that TAPIR-KV can lower the commit latency by 50%, increase throughtput by 2× relative to conventional transactional storage systems and, in many cases, can match the performance of a weakly consistent system without transaction support.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *Proc. of SIGMOD*, 1995.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proc. of SOSP*, 2007.

[3] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, 2011.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[5] K. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of SOSP*, 1987.

[6] L. Camargos, R. Schmidt, and F. Pedone. Multicoordinated Paxos. Technical report, University of Lugano Faculty of Informatics, 2007/02, Jan. 2007.

[7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. of VLDB*, 2008.

[8] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *Proceedings of OSDI*, Proc. of OSDI, 2012.

[9] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proc. of USENIX ATC*, 2012.

[10] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of SOSP*, 2007.

[12] A. Dey, A. Fekete, and U. Röhm. Scalable transactions across heterogeneous NoSQL key-value data stores. *Proc. of VLDB*, 2013.

[13] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, 2006.

[14] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007.

[15] R. Escriva, B. Wong, and E. G. Sirer. Warp: Multi-key transactions for key-value stores. Technical report, Cornell, Nov 2013.

[16] Google Compute Engine. https://cloud.google.com/products/compute-engine/.

[17] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 2006.

[18] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of STOC*, 1997.

[19] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *Proc. of EuroSys*, 2013.

[20] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

[21] L. Lamport. Paxos made simple. *ACM Sigact News*, 2001.

[22] L. Lamport. Generalized consensus and Paxos. Technical Report 33, Microsoft Research, 2005.

[23] L. Lamport. Fast paxos. *Distributed Computing*, 19(2), 2006.

[24] C. Leau. Spring Data Redis - Retwis-J, 2013. http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[25] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as

possible, consistent when necessary. In *Proc. of OSDI*, 2012.

[26] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, 2005.

[27] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proc. of ECOOP*, 1999.

[28] B. Liskov and J. Cowling. Viewstamped replication revisited, 2012.

[29] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[30] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. E. Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. of VLDB*, 2013.

[31] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proc. of OSDI*, 2008.

[32] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of SOSP*, 2013.

[33] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proc. of OSDI*, 2014.

[34] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

[35] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proc. of SOSP*, 2011.

[36] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *Proc. of VLDB*, 2012.

[37] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. of OSDI*, 2010.

[38] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *Proc. of SIGCOMM*, 2014.

[39] *IEEE 1588 Standard for A Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. http://www.nist.gov/el/isd/ieee/ieee1588.cfm.

[40] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proc. of SOSP*, 2011.

[41] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. of VLDB*, 2007.

[42] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[43] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the 21st International Conference on Data Engineering*, 2005.

[44] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proc. of SOSP*, 2013.