Building Consistent Transactions with Inconsistent Replication (Extended Version)

Technical Report UW-CSE-2014-12-01 v2 *

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 strong consistency and distributed transactions (e.g., Google's Spanner) for their strong guarantees and ease of use. Unfortunately, existing transactional storage systems are expensive to use – in part because they require costly replication protocols, like Paxos, for fault tolerance. In this paper, we present a new approach that makes transactional storage systems more affordable: we eliminate consistency from the replication protocol while still providing distributed transactions with strong consistency to applications.

We present TAPIR *– the Transactional Application Protocol for Inconsistent Replication – the first transaction protocol to use a novel replication protocol, called* inconsistent replication*, that provides fault tolerance without consistency. By enforcing strong consistency only in the transaction protocol, TAPIR can commit transactions in a single round-trip and order distributed transactions without centralized coordination. We demonstrate the use of TAPIR in a transactional key-value store,* TAPIR-KV*. Compared to conventional systems,* TAPIR-KV *provides better latency* and *throughput.*

1. Introduction

Distributed storage systems provide fault tolerance and availability for large-scale web applications. Increasingly, application programmers prefer systems that support distributed transactions with strong consistency to help them manage application complexity and concurrency in a distributed environment. Several recent systems [4, 11, 17, 23] reflect this trend, notably Google's Spanner system [13], which guarantees linearizable transaction ordering.[1]

For application programmers, distributed transactional storage with strong consistency comes at a price. These systems commonly use replication for fault-tolerance, and replication protocols with strong consistency, like Paxos, impose a high performance cost, while more efficient, weak consistency protocols fail to provide strong system guarantees.

Significant prior work has addressed improving the performance of transactional storage systems – including systems that optimize for read-only transactions [4, 13], more restrictive transaction models [2, 14, 23], or weaker consistency guarantees [3, 34, 43]. However, none of these systems have addressed both 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 programmers. Our key insight is that existing transactional storage systems waste work and performance by incorporating 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), that provides fault tolerance without consistency. Rather than an ordered operation log, IR presents an *unordered operation set* to applications. Successful operations execute at a majority of the replicas and survive failures, but replicas can execute them in any order. Thus, IR needs no cross-replica coordination or designated leader for operation processing. However, unlike eventual consistency, IR allows applications to enforce higher-level invariants when needed.

Thus, despite IR's weak consistency guarantees, TAPIR provides *linearizable read-write transactions* and supports globally-consistent reads across the database at a timestamp – the same guarantees as Spanner. TAPIR efficiently leverages IR to distribute read-write transactions in a *single round-trip* and order transactions globally across partitions and replicas *with no centralized coordination.*

* This document is an extended version of the paper by the same title that appeared in SOSP 2015. An overview of the additional content is provided in Section 1.1.

[1] Spanner's linearizable transaction ordering is also referred to as strict serializable isolation or external consistency.

We implemented TAPIR in a new distributed transactional key-value store called TAPIR-KV, which supports linearizable transactions over a partitioned set of keys. Our experiments found that TAPIR-KV had: (1) 50% lower commit latency and (2) more than 3× better throughput compared to systems using conventional transaction protocols, including an implementation of Spanner's transaction protocol, and (3) comparable performance to MongoDB [36] and Redis [40], widely-used eventual consistency systems.

This paper makes the following contributions to the design of distributed, replicated transaction systems:

- We define *inconsistent replication*, a new replication technique that provides fault tolerance without consistency.
- We design *TAPIR*, a new distributed transaction protocol that provides strict serializable transactions using inconsistent replication for fault tolerance.
- We build and evaluate TAPIR-KV, a key-value store that combines inconsistent replication and TAPIR to achieve high-performance transactional storage.

1.1 Technical Report Overview

This technical report includes the following additions to our paper:

1. A more detailed description of the IR view change protocol. (Section 3.2.2)
2. A description of the IR client recovery protocol. (Section 3.2.3)
3. A full proof and TLA+ [27] specification for IR. (Section 3.3 and Appendix A)
4. Additional pseudocode for TAPIR-EXEC-CONSENSUS, which executes TAPIR's `Prepare` operation, and TAPIR-SYNC, which synchronizes replicas with missed IR operations and consensus results. (Section 5)
5. The complete coordinator recovery protocol for TAPIR. (Section 5.2.3)
6. A full proof and TLA specification for TAPIR on IR. (Section 5.3 and Appendix B)
7. Extensions to the TAPIR protocol for:
   (a) Supporting read-only transactions at a consistent timestamp and Spanner-style linearizable read-only transactions. (Section 6.1)
   (b) Relaxing from linearizable transaction ordering to serializable. (Section 6.2)
   (c) Reducing the fast quorum size using synchronous disk writes. (Section 6.3)
   (d) Coping with very high clock skews. (Section 6.4)
8. A full latency and clock skew profile of our Google Compute Engine testbed. (Section 7.1.1)

2. Over-Coordination in Transaction Systems

Replication protocols have become an important component in distributed storage systems. Modern storage systems commonly partition data into *shards* for scalability and then replicate each shard for fault-tolerance and availability [4, 9, 13, 35]. To support transactions with strong consistency, they must implement both a *distributed transaction protocol* – to ensure atomicity and consistency for transactions across shards – and a *replication protocol* – to ensure transactions are not lost (provided that no more than half of the replicas in each shard fail at once). As shown in Figure 1, these systems typically place the transaction protocol, which combines an atomic commitment protocol and a concurrency control mechanism, on top of the replication protocol (although alternative architectures have also occasionally been proposed [35]).

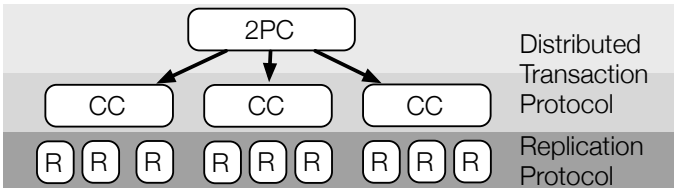

Figure 1: *A common architecture for distributed transactional storage systems today.* The distributed transaction protocol consists of an atomic commitment protocol, commonly Two-Phase Commit (2PC), and a concurrency control (CC) mechanism. This runs atop a replication (R) protocol, like Paxos.

Distributed transaction protocols assume the availability of an *ordered, fault-tolerant log*. This ordered log abstraction is easily and efficiently implemented with a spinning disk but becomes more complicated and expensive with replication. To enforce the serial ordering of log operations, transactional storage systems must integrate a costly replication protocol with strong consistency (e.g., Paxos [28], Viewstamped Replication [38] or virtual synchrony [7]) rather than a more efficient, weak consistency protocol [25, 41].

The traditional log abstraction imposes a serious performance penalty on replicated transactional storage systems, because it enforces strict serial ordering using expensive distributed coordination *in two places*: the replication protocol enforces a serial ordering of operations across replicas in each shard, while the distributed transaction protocol enforces a serial ordering of transactions across shards. This redundancy impairs latency and throughput for systems that integrate both protocols. The replication protocol must coordinate across replicas on every operation to enforce strong consistency; as a result, it takes *at least two round-trips* to order any read-write transaction. Further, to efficiently order operations, these protocols typically rely on a replica leader, which can introduce a throughput bottleneck to the system.

As an example, Figure 2 shows the redundant coordination required for a single read-write transaction in a system like Spanner. Within the transaction, `Read` operations go to the shard leaders (which may be in other datacenters), because the operations must be ordered across replicas, even though they are not replicated. To `Prepare` a transaction for commit, the transaction protocol must coordinate transaction ordering

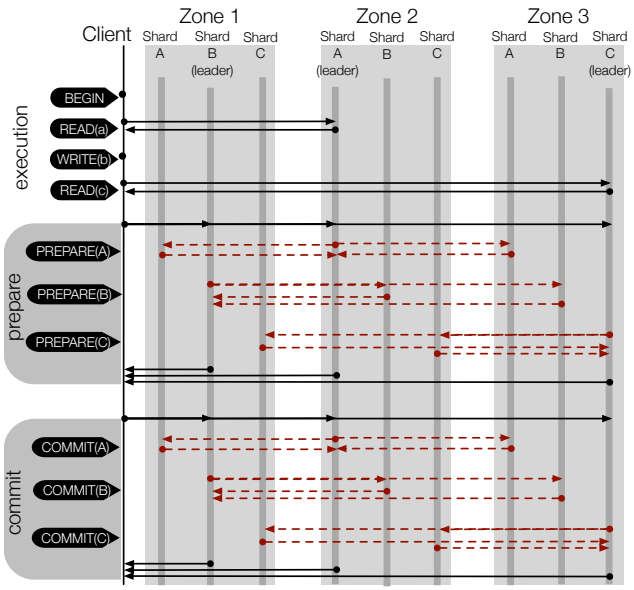

Figure 2: *Example read-write transaction using two-phase commit, viewstamped replication and strict two-phase locking.* Availability *zones* represent either a cluster, datacenter or geographic region. Each *shard* holds a partition of the data stored in the system and has replicas in each zone for fault tolerance. The red, dashed lines represent redundant coordination in the replication layer.

across shards, and then the replication protocol coordinates the `Prepare` operation ordering across replicas. As a result, it takes at least two round-trips to commit the transaction.

In the TAPIR and IR design, we eliminate the redundancy of strict serial ordering over the two layers and its associated performance costs. IR is the first replication protocol to provide *pure fault tolerance* without consistency. Instead of an ordered operation log, IR presents the abstraction of an *unordered operation set*. Existing transaction protocols cannot efficiently use IR, so TAPIR is the first transaction protocol designed to provide linearizable transactions on IR.

3. Inconsistent Replication

Inconsistent replication (IR) is an efficient replication protocol designed to be used with a higher-level protocol, like a distributed transaction protocol. IR provides fault-tolerance without enforcing any consistency guarantees of its own. Instead, it allows the higher-level protocol, which we refer to as the *application protocol*, to decide the outcome of conflicting operations and recover those decisions through IR's fault-tolerant, unordered operation set.

3.1 IR Overview

Application protocols invoke operations through IR in one of two modes:

- **inconsistent** – operations can execute in any order. Successful operations persist across failures.

Client Interface

`InvokeInconsistent`(op)
`InvokeConsensus`$(op, decide(results)) \rightarrow result$

Replica Upcalls

`ExecInconsistent`(op) `ExecConsensus`$(op) \rightarrow result$
`Sync`(R) `Merge`$(d,u) \rightarrow record$

Client State

- *client id* - unique identifier for the client
- *operation counter* - # of sent operations

Replica State

- *state* - current replica state; either NORMAL or VIEW-CHANGING
- *record* - unordered set of operations and consensus results

Figure 3: *Summary of IR interfaces and client/replica state.*

- **consensus** – operations execute in any order, but return a single *consensus result*. Successful operations and their consensus results persist across failures.

inconsistent operations are similar to operations in weak consistency replication protocols: they can execute in different orders at each replica, and the application protocol must resolve conflicts afterwards. In contrast, **consensus** operations allow the application protocol to *decide* the outcome of conflicts (by executing a *decide* function specified by the application protocol) and recover that decision afterwards by ensuring that the chosen result persists across failures as the consensus result. In this way, **consensus** operations can serve as the basic building block for the higher-level guarantees of application protocols. For example, a distributed transaction protocol can decide which of two conflicting transactions will commit, and IR will ensure that decision persists across failures.

3.1.1 IR Application Protocol Interface

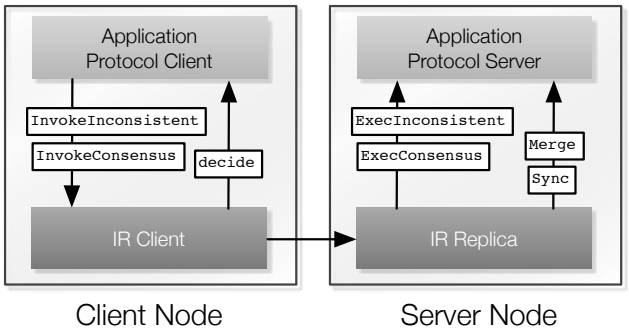

Figure 4: *IR Call Flow.*

Figure 3 summarizes the IR interfaces at clients and replicas. Application protocols invoke operations through a client-side IR library using `InvokeInconsistent` and `InvokeConsensus`, and then IR runs operations using the `ExecInconsistent` and `ExecConsensus` upcalls at the replicas.

If replicas return conflicting/non-matching results for a **consensus** operation, IR allows the application protocol to decide the operation's outcome by invoking the *decide* function

– passed in by the application protocol to `InvokeConsensus` – in the client-side library. The *decide* function takes the list of returned results (the candidate results) and returns a single result, which IR ensures will persist as the *consensus result*. The application protocol can later recover the consensus result to find out its decision to conflicting operations.

Some replicas may miss operations or need to reconcile their state if the consensus result chosen by the application protocol does not match their result. To ensure that IR replicas eventually converge, they periodically *synchronize*. Similar to eventual consistency, IR relies on the application protocol to reconcile inconsistent replicas. On synchronization, a single IR node first upcalls into the application protocol with `Merge`, which takes records from inconsistent replicas and merges them into a *master record* of successful operations and consensus results. Then, IR upcalls into the application protocol with `Sync` at each replica. `Sync` takes the *master record* and reconciles application protocol state to make the replica consistent with the chosen consensus results.

3.1.2 IR Guarantees

We define a *successful operation* to be one that returns to the application protocol. The *operation set* of any IR group includes all successful operations. We define an operation X as being *visible* to an operation Y if one of the replicas executing Y has previously executed X. IR ensures the following properties for the operation set:

P1. [Fault tolerance] At any time, every operation in the operation set is in the record of at least one replica in any quorum of $f+1$ non-failed replicas.

P2. [Visibility] For any two operations in the operation set, at least one is visible to the other.

P3. [Consensus results] At any time, the result returned by a successful **consensus** operation is in the record of at least one replica in any quorum. The only exception is if the consensus result has been explicitly modified by the application protocol through `Merge`, after which the outcome of `Merge` will be recorded instead.

IR ensures guarantees are met for up to f simultaneous failures out of $2f+1$ replicas[2] and any number of client failures. Replicas must fail by crashing, without Byzantine behavior. We assume an asynchronous network where messages can be lost or delivered out of order. IR *does not* require synchronous disk writesduring operation execution, ensuring guarantees are maintained even if clients or replicas lose state on failure. IR makes progress (operations will eventually become successful) provided that messages that are repeatedly resent are eventually delivered before the recipients time out.

[2] Using more than $2f+1$ replicas for f failures is possible but illogical because it requires a larger quorum size with no additional benefit.

3.1.3 Application Protocol Example: Fault-Tolerant Lock Server

As an example, we show how to build a simple lock server using IR. The lock server's guarantee is mutual exclusion: a lock cannot be held by two clients at once. We replicate `Lock` as a **consensus** operation and `Unlock` as an **inconsistent** operation. A client application acquires the lock only if `Lock` successfully returns OK as a consensus result.

Since operations can run in any order at the replicas, clients use unique ids (e.g., a tuple of client id and a sequence number) to identify corresponding `Lock` and `Unlock` operations and only call `Unlock` if `Lock` first succeeds. Replicas will therefore be able to later match up `Lock` and `Unlock` operations, regardless of order, and determine the lock's status.

Note that **inconsistent** operations *are not commutative* because they can have side-effects that affect the outcome of **consensus** operations. If an `Unlock` and `Lock` execute in different orders at different replicas, some replicas might have the lock free, while others might not. If replicas return different results to `Lock`, IR invokes the lock server's decide function, which returns OK if $f+1$ replicas returned OK and NO otherwise. IR only invokes `Merge` and `Sync` on recovery, so we defer their discussion until Section 3.2.2.

IR's guarantees ensure correctness for our lock server. P1 ensures that held locks are persistent: a `Lock` operation persists at one or more replicas in any quorum. P2 ensures mutual exclusion: for any two conflicting `Lock` operations, one is visible to the other in any quorum; therefore, IR will never receive $f+1$ matching OK results, precluding the *decide* function from returning OK. P3 ensures that once the client application receives OK from a `Lock`, the result will not change. The lock server's `Merge` function will not change it, as we will show later, and IR ensures that the OK will persist in the record of at least one replica out of any quorum.

3.2 IR Protocol

Figure 3 shows the IR state at the clients and replicas. Each IR client keeps an *operation counter*, which, combined with the *client id*, uniquely identifies operations. Each replica keeps an unordered *record* of executed operations and results for **consensus** operations. Replicas add **inconsistent** operations to their record as TENTATIVE and then mark them as FINALIZED once they execute. **consensus** operations are first marked TENTATIVE with the result of locally executing the operation, then FINALIZED once the record has the consensus result.

IR uses four sub-protocols – *operation processing*, *replica recovery/synchronization*, *client recovery*, and *group membership change*. We discuss the first three here; the last is identical to that of Viewstamped Replication [33].

3.2.1 Operation Processing

We begin by describing IR's normal-case **inconsistent** operation processing protocol without failures:

1. The client sends ⟨PROPOSE, *id*, *op*⟩ to all replicas, where *id* is the operation id and *op* is the operation.
2. Each replica writes *id* and *op* to its record as TENTATIVE, then responds to the client with ⟨REPLY, *id*⟩.
3. Once the client receives $f+1$ responses from replicas (retrying if necessary), it returns to the application protocol and asynchronously sends ⟨FINALIZE, *id*⟩ to all replicas. (FINALIZE can also be piggy-backed on the client's next message.)
4. On FINALIZE, replicas upcall into the application protocol with `ExecInconsistent`(*op*) and mark *op* as FINALIZED.

Due to the lack of consistency, IR can successfully complete an **inconsistent** operation with a *single round-trip* to $f+1$ replicas and no coordination across replicas. If the IR client does not receive a response to its PREPARE from $f+1$ replicas, it will retry until it does.

Next, we describe the normal-case **consensus** operation processing protocol, which has both a *fast path* and a *slow path*. IR uses the fast path when it can achieve a *fast quorum* of $\lceil \frac{3}{2} f \rceil + 1$ replicas that *return matching results* to the operation. Similar to Fast Paxos and Speculative Paxos [39], IR requires a fast quorum to ensure that a majority of the replicas in any quorum agrees to the consensus result. This quorum size is necessary to execute operations in a single round trip when using a replica group of size $2f+1$ [30]; an alternative would be to use quorums of size $2f+1$ in a system with $3f+1$ replicas.

When IR cannot achieve a fast quorum, either because replicas did not return enough matching results (e.g., if there are conflicting concurrent operations) or because not enough replicas responded (e.g., if more than $\frac{f}{2}$ are down), then it must take the slow path. We describe both below:

1. The client sends ⟨PROPOSE, *id*, *op*⟩ to all replicas.
2. Each replica calls into the application protocol with `ExecConsensus`(*op*) and writes *id*, *op*, and *result* to its record as TENTATIVE. The replica responds to the client with ⟨REPLY, *id*, *result*⟩.
3. If the client receives at least $\lceil \frac{3}{2} f \rceil + 1$ matching *result*s (within a timeout), then it takes the *fast path*: the client returns *result* to the application protocol and asynchronously sends ⟨FINALIZE, *id*, *result*⟩ to all replicas.
4. Otherwise, the client takes the *slow path*: once it receives $f+1$ responses (retrying if necessary), then it sends ⟨FINALIZE, *id*, *result*⟩ to all replicas, where *result* is obtained from executing the *decide* function.
5. On receiving FINALIZE, each replica marks the operation as FINALIZED, updating its record if the received *result* is different, and sends ⟨CONFIRM, *id*⟩ to the client.
6. On the slow path, the client returns *result* to the application protocol once it has received $f+1$ CONFIRM responses.

The fast path for **consensus** operations takes a single round trip to $\lceil \frac{3}{2} f \rceil + 1$ replicas, while the slow path requires two round-trips to at least $f+1$ replicas. Note that IR replicas can execute operations in different orders and *still* return matching responses, so IR can use the fast path without a strict serial ordering of operations across replicas. IR can also run the fast path and slow path in parallel as an optimization.

3.2.2 Replica Recovery and Synchronization

IR uses a single protocol for recovering failed replicas and running periodic synchronizations. On recovery, we must ensure that the failed replica applies all operations it may have lost or missed in the operation set, so we use the same protocol to periodically bring all replicas up-to-date.

To handle recovery and synchronization, we introduce *view changes* into the IR protocol, similar to Viewstamped Replication (VR) [38]. These maintain IR's correctness guarantees across failures. Each IR view change is run by a leader; leaders coordinate only view changes, *not* operation processing. During a view change, the leader has just one task: to make at least $f+1$ replicas up-to-date (i.e., they have applied all operations in the operation set) and consistent with each other (i.e., they have applied the same consensus results). IR view changes require a leader because polling inconsistent replicas can lead to conflicting sets of operations and consensus results. Thus, the leader must decide on a *master record* that replicas can then use to synchronize with each other.

To support view changes, each IR replica maintains a current *view*, which consists of the identity of the leader, a list of the replicas in the group, and a (monotonically increasing) *view number* uniquely identifying the view. Each IR replica can be in one of the three states: NORMAL, VIEW-CHANGING or RECOVERING. Replicas process operations only in the NORMAL state. We make four additions to IR's operation processing protocol:

1. IR replicas send their current view number in every response to clients. For an operation to be considered successful, the IR client must receive responses with matching view numbers. For **consensus** operations, the view numbers in REPLY and CONFIRM must match as well. If a client receives responses with different view numbers, it notifies the replicas in the older view.
2. On receiving a message with a view number that is higher than its current view, a replica moves to the VIEW-CHANGING state and requests the master record from any replica in the higher view. It replaces its own record with the master record and upcalls into the application protocol with `Sync` before returning to NORMAL state.
3. On PROPOSE, each replica first checks whether the operation was already FINALIZED by a view change. If so, the replica responds with ⟨REPLY, *id*, FINALIZED, *v*, [*result*]⟩, where *v* is the replica's current view number and *result* is the consensus result for **consensus** operations.
4. If the client receives REPLY with a FINALIZED status for **consensus** operations, it sends ⟨FINALIZE, *id*, *result*⟩ with the received *result* and waits until it receives $f+1$ CONFIRM responses in the same view before returning *result* to the application protocol.

IR's view change protocol is similar to VR's. Each view change is coordinated by a leader, which is unique per view and deterministically chosen. There are three key differences. First, in IR the leader *merges* records during a view change rather than simply taking the longest log from the latest view. The reason for this is that, with inconsistent replicas and unordered operations, any single record could be incomplete. Second, in VR, the leader is used to process operations in the normal case, but IR uses the leader *only* for performing view changes. Finally, on recovery, an IR replica performs a view change, rather than simply interrogating a single replica. This makes sure that the recovering replica either receives all operations it might have sent a reply for, or prevents them from completing.

The full view change protocol follows:

1. A replica that notices the need for a view change advances its view number and sets its status to either VIEW-CHANGING or RECOVERING – if the replica just started a recovery. A replica notices the need for a view change either based on a timeout, because it is a recovering replica, or because it received a DO-VIEW-CHANGE message for a view with a larger number than its own current view-number. It records the new view number to disk.
2. The replica then sends a $\langle$DO-VIEW-CHANGE, *rec*, v, $v'\rangle$ message to the new leader, *except when the sending replica is a recovering replica*. It also sends the same message, without the *rec* field, to the other replicas. Here v identifies the new view, v' is the latest view in which the replica's status was NORMAL, and *rec* is its unordered record of executed operations.
3. Once the new leader receives f records from f other replicas, it considers all records with the highest value of v'. It uses a a merge function, shown in Figure 5, to join these into a master record R.
4. The leader updates its view number to v_{new}, where v_{new} is the view number from the received messages, and its status to NORMAL. It then informs the other replicas of the completion of the view change by sending a $\langle$START-VIEW, v_{new}, $R\rangle$, where R is the master record.
5. When a replica receives a START-VIEW message with v_{new} greater than or equal to its current view number, it replaces its own record with R and upcalls into the application protocol with `Sync`.
6. Once `Sync` is complete, the replica updates its current view number to v_{new}, records this to disk, and enters the NORMAL state.

Merging Records. The IR-MERGE-RECORDS function is used by the new leader to merge the set of received records. This function is shown in Figure 5. IR-MERGE-RECORDS starts by adding all **inconsistent** operations and **consensus** operations marked FINALIZED to R and calling `Sync` into the application protocol. These operations must persist in the next view, so we first apply them to the leader, ensuring that they are visible to any operations for which the leader will decide

```
IR-MERGE-RECORDS(records)
 1  R,d,u = ∅
 2  for ∀op ∈ records
 3      if op.type == inconsistent
 4          R = R∪op
 5      elseif op.type == consensus and op.status == FINALIZED
 6          R = R∪op
 7      elseif op.type == consensus and op.status == TENTATIVE
 8          if op.result in more than f/2 + 1 records
 9              d = d∪op
10          else
11              u = u∪op
12  Sync(R)
13  return R∪Merge(d,u)
```

Figure 5: *Merge function for the master record.* We merge all records from replicas in the latest view, which is always a strict super set of the records from replicas in lower views.

consensus results next in `Merge`. As an example, `Sync` for the lock server matches up all corresponding `Lock` and `Unlock` by id; if there are unmatched `Locks`, it sets *locked* = TRUE; otherwise, *locked* = FALSE.

IR asks the application protocol to decide the consensus result for the remaining TENTATIVE **consensus** operations, which either: (1) have a matching result, which we define as the *majority result*, in at least $\lceil \frac{f}{2} \rceil + 1$ records or (2) do not. IR places these operations in d and u, respectively, and calls `Merge`(d,u) into the application protocol, which must return a consensus result for every operation in d and u.

IR must rely on the application protocol to decide consensus results for several reasons. For operations in d, IR cannot tell whether the operation succeeded with the majority result on the fast path, or whether it took the slow path and the application protocol *decide*'d a different result that was later lost. In some cases, it is not safe for IR to keep the majority result because it would violate application protocol invariants. For example, in the lock server, OK could be the majority result if only $\lceil \frac{f}{2} \rceil + 1$ replicas replied OK, but the other replicas might have accepted a conflicting lock request. However, it is also possible that the other replicas *did* respond OK, in which case OK would have been a successful response on the fast-path.

The need to resolve this ambiguity is the reason for the caveat in IR's consensus property (P3) that consensus results can be changed in `Merge`. Fortunately, the application protocol can ensure that successful consensus results *do not change* in `Merge`, simply by maintaining the majority results in d on `Merge` *unless they violate invariants*. The merge function for the lock server, therefore, does not change a majority response of OK, *unless* another client holds the lock. In that case, the operation in d could not have returned a successful consensus result to the client (either through the fast or the slow path), so it is safe to change its result.

For operations in u, IR needs to invoke *decide* but cannot without at least $f+1$ results, so uses `Merge` instead. The application protocol can decide consensus results in `Merge` without $f+1$ replica results and still preserve IR's visibility

property because IR has already applied all of the operations in R and d, which are the only operations definitely in the operation set, at this point.

The leader adds all operations returned from `Merge` and their consensus results to R, then sends R to the other replicas, which call `Sync`(R) into the application protocol and *replace their own records with R*. The view change is complete after at least $f+1$ replicas have exchanged and merged records and SYNC'd with the master record. A replica can only process requests in the new view (in the NORMAL state) *after* it completes the view change protocol. At this point, any recovering replicas can also be considered recovered. If the leader of the view change does not finish the view change by some timeout, the group will elect a new leader to complete the protocol by starting a new view change with a larger view number.

3.2.3 Client Recovery

We assume that clients can lose some or all of their state on failure. On recovery, a client must ensure that: (1) it recovers its latest operation counter, and (2) any operations that it started but did not finish are FINALIZED. To do so, the recovering client requests the *id* for its latest operation from a majority of the replicas. This poll gets the client the largest *id* that the group has seen from it, so the client takes the largest returned *id* and increments it to use as its new operation counter.

A view change finalizes all TENTATIVE operation on the next synchronization, so the client does not need to finish previously started operations and IR does not have to worry about clients failing to recover after failure.

3.3 Correctness

For correctness, we show that IR provides the following properties for operations in the *operation set*:

P1. [Fault tolerance] At any time, every operation in the operation set is in the record of at least one replica in any quorum of $f+1$ non-failed replicas.

P2. [Visibility] For any two **consensus** operations in the operation set, at least one is visible to the other.

P3. [Consensus results] At any time, every successful consensus result is in the record of at least one replica in any quorum. Again, the only exception being that the application protocol modified the result through `Merge`.

P4. [Eventual Consistency] Given a sufficiently long period of synchrony, any operation in the operation set (and its consensus result, if applicable) will eventually have executed or `Sync`ed at every non-faulty replica.

In Appendix A, we give a TLA+ specification, which we have model-checked. In addition, we have added an *eventual consistency* property, which is not necessary for correctness, but is useful for application protocols. As this is a liveness property, it holds only during periods of synchrony, when messages that are repeatedly resent are eventually delivered before the recipient times out [18].

We begin our proof of correctness by defining the following terms:

D1. An operation is *applied* at a replica if that replica has executed (through `ExecInconsistent` or `ExecConsensus`) or synchronized (through `Sync`) the operation.

D2. An operation X is *visible* to a **consensus** operation Y if one of the replicas providing candidate results for Y has previously applied X.

D3. The *persistent operation set* is the set of operations applied at at least one replica in any quorum of $f+1$ non-failed replicas.

We first prove a number of invariants about the persistent operation set. Given these invariants, we can show that the IR properties hold.

I1. *The size of persistent operation set is monotomically increasing.*

I1 holds at every replica during normal operation because replicas never roll back executed operations. I1 also hold across view changes. The leader merges all operations from the records of $f+1$ non-faulty replicas into the master record, so by quorum intersection, the master record contains every operation in the persistent operation set. Then, at least $f+1$ non-faulty replicas replace their record with the master record and applies the master record (through `Sync`), so any persistent operations before the view change will continue to persist after the view change.

I2. *All operations in the persistent operation set are visible to any* **consensus** *operation added to the set.*

consensus operations are added to the persistent set by either: (1) executing at at least a quorum of $f+1$ replicas or (2) being merged by the leader into the master record. In case 1, by definition, every operation already in the persistent operation set must be applied at at least 1 replica out of the quorum and will be visible to the added **consensus** operation. In case 2, the leader applies all operations in the persistent operation set (through `Sync`) before running `Merge`, ensuring that every operation already in the persistent operation set is visible to operations added to the persistent operation set through `Merge`.

I3. *The result of any* **consensus** *operation in the persistent operation set is* either *the successful consensus result or the* `Merge` *result.*

The result of any **consensus** operations in the persistent set is either: (1) a matching result from executing the operation (through `ExecConsensus`) at a fast quorum of $\lceil \frac{3}{2} f \rceil + 1$ replicas, (2) a result from executing the application protocol-specific *decide* function in the client-side library, or (3) a result from executing `Merge` at the leader during a view change. In case 1, the matching result will be both the result in the persistent operation set and the successful consensus result. The same holds for the result returned from *decide* in case 2. During a view change, the leader may get an operation that has already fulfilled either case 1 or case 2, and change the

result in `Merge`. The result from `Merge` will be in the record and applied to at least $f+1$ replicas. Thus, either the successful consensus result or, if the application protocol changed the result in `Merge`, the `Merge` result, will continue to persist in the persistent operation set.

I4. *All operations and consensus results in the persistent operation set in all previous view must be applied at a replica before it executes any operations in the new view.*

IR clients require that all responses come from replicas in the same view. Thus, if any replica is in view v and at least $f+1$ other replicas are in a higher view $V > v$, that replica cannot successfully complete an operation until it joins the higher view. In order to join the higher view, the replica in the lower view must obtain the master record from a replica in the higher view, and `Sync` with that master record. The master record contains all operations in the persistent operation set, so the replica will apply all operations from the persistent operation set before processing operations in the new view.

Given these four invariants for the persistent operation set, we can show that the four properties of IR hold. Any operation in the operation set must have executed at (and received responses from) $f+1$ of $2f+1$ replicas, so by quorum intersection, all operations in the operation set must be in the persistent operation set. Thus, I1 directly implies P1, as any operation in the persistent operation set will continue to be in the set. I1 and I2 imply P2 because, for any **consensus** operation X, all operations added to the persistent operation set before X are visible to X and X will be visible to all operations added to the persistent operation set after it. I1 and I3 implies P3 because either the successful consensus result will remain in the persistent operation set or the `Merge` result will. I4 implies P4 because, if all replicas are non-faulty for long enough, they will eventually all attempt to participate in processing operations, which will cause them to `Sync` all operations in the persistent operation set.

4. Building Atop IR

IR obtains performance benefits because it offers weak consistency guarantees and relies on application protocols to resolve inconsistencies, similar to eventual consistency protocols such as Dynamo [15] and Bayou [44]. However, unlike eventual consistency systems, which expect applications to resolve conflicts *after they happen*, IR allows application protocols to *prevent conflicts before they happen*. Using **consensus** operations, application protocols can enforce higher-level guarantees (e.g., TAPIR's linearizable transaction ordering) across replicas despite IR's weak consistency.

However, building strong guarantees on IR requires careful application protocol design. IR cannot support certain application protocol invariants. Moreover, if misapplied, IR can even provide applications with *worse* performance than a strongly consistent replication protocol. In this section, we discuss the properties that application protocols need to have to correctly and efficiently enforce higher-level guarantees with IR and TAPIR's techniques for efficiently providing linearizable transactions.

4.1 IR Application Protocol Requirement: *Invariant checks must be performed pairwise.*

Application protocols can enforce certain types of invariants with IR, but not others. IR guarantees that in any pair of **consensus** operations, at least one will be visible to the other (P2). Thus, IR readily supports invariants that can be safely checked by examining *pairs* of operations for conflicts. For example, our lock server example can enforce mutual exclusion. However, application protocols cannot check invariants that require the entire history, because each IR replica may have an incomplete history of operations. For example, tracking bank account balances and allowing withdrawals only if the balance remains positive is problematic because the invariant check must consider the entire history of deposits and withdrawals.

Despite this seemingly restrictive limitation, application protocols can still use IR to enforce useful invariants, including lock-based concurrency control, like Strict Two-Phase Locking (S2PL). As a result, distributed transaction protocols like Spanner [13] or Replicated Commit [35] would work with IR. IR can also support optimistic concurrency control (OCC) [24] because OCC checks are pairwise as well: each committing transaction is checked against every previously committed transaction, so **consensus** operations suffice to ensure that *at least one replica sees any conflicting transaction* and aborts the transaction being checked.

4.2 IR Application Protocol Requirement: *Application protocols must be able to change* **consensus** *operation results.*

Inconsistent replicas could execute **consensus** operations with one result and later find the group agreed to a different consensus result. For example, if the group in our lock server agrees to reject a `Lock` operation that one replica accepted, the replica must later free the lock, and vice versa. As noted above, the group as a whole continues to enforce mutual exclusion, so these temporary inconsistencies are tolerable and are always resolved by the end of synchronization.

In TAPIR, we take the same approach with distributed transaction protocols. 2PC-based protocols are always prepared to abort transactions, so they can easily accommodate a `Prepare` result changing from PREPARE-OK to ABORT. If ABORT changes to PREPARE-OK, it might temporarily cause a conflict at the replica, which can be correctly resolved because the group as a whole could not have agreed to PREPARE-OK for two conflicting transactions.

Changing `Prepare` results does sometimes cause unnecessary aborts. To reduce these, TAPIR introduces two `Prepare` results in addition to PREPARE-OK and ABORT: ABSTAIN and RETRY. ABSTAIN helps TAPIR distinguish between conflicts with *committed* transactions, which will not abort, and conflicts with *prepared* transactions, which may later abort.

Replicas return RETRY if the transaction has a chance of committing later. The client can retry the `Prepare` *without* re-executing the transaction.

4.3 IR Performance Principle: *Application protocols should not expect operations to execute in the same order.*

To efficiently achieve agreement on consensus results, application protocols should not rely on operation ordering for application ordering. For example, many transaction protocols [4, 20, 23] use Paxos operation ordering to determine transaction ordering. They would perform worse with IR because replicas are unlikely to agree on which transaction should be next in the transaction ordering.

In TAPIR, we use *optimistic timestamp ordering* to ensure that replicas agree on a single transaction ordering despite executing operations in different orders. Like Spanner [13], every committed transaction has a timestamp, and committed transaction timestamps reflect a linearizable ordering. However, TAPIR clients, not servers, propose a timestamp for their transaction; thus, if TAPIR replicas agree to commit a transaction, they have all agreed to the same transaction ordering.

TAPIR replicas use these timestamps to order their transaction logs and multi-versioned stores. Therefore, replicas can execute `Commit` in different orders but still converge to the same application state. TAPIR leverages loosely synchronized clocks at the clients for picking transaction timestamps, which improves performance but is not necessary for correctness.

4.4 IR Performance Principle: *Application protocols should use cheaper* **inconsistent** *operations whenever possible rather than* **consensus** *operations.*

By concentrating invariant checks in a few operations, application protocols can reduce **consensus** operations and improve their performance. For example, in a transaction protocol, any operation that decides transaction ordering must be a **consensus** operation to ensure that replicas agree to the same transaction ordering. For locking-based transaction protocols, this is any operation that acquires a lock. Thus, every `Read` and `Write` must be replicated as a **consensus** operation.

TAPIR improves on this by using optimistic transaction ordering and OCC, which reduces **consensus** operations by concentrating all ordering decisions into a single set of validation checks at the proposed transaction timestamp. These checks execute in `Prepare`, which is TAPIR's only **consensus** operation. `Commit` and `Abort` are **inconsistent** operations, while `Read` and `Write` are not replicated.

5. TAPIR

This section details TAPIR – the Transactional Application Protocol for Inconsistent Replication. As noted, TAPIR is designed to efficiently leverage IR's weak guarantees to provide high-performance linearizable transactions. Using IR, TAPIR can order a transaction in a *single round-trip* to all replicas in all participant shards without *any centralized coordination.*

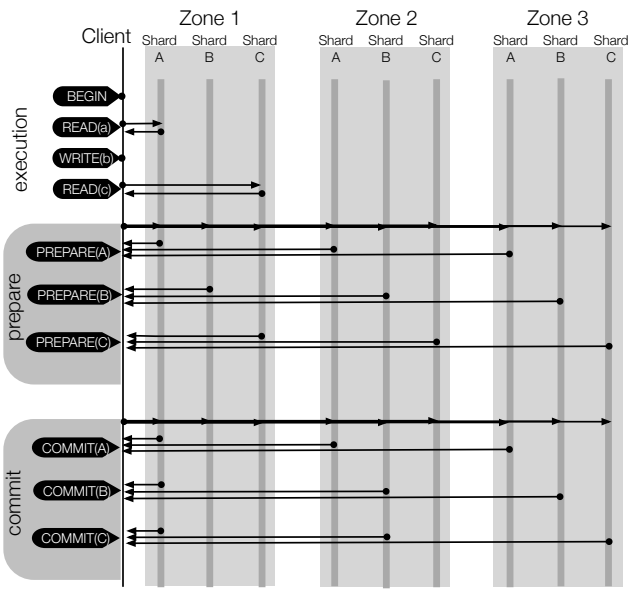

Figure 6: *Example read-write transaction in TAPIR.* TAPIR executes the same transaction pictured in Figure 2 with less redundant coordination. `Reads` go to the closest replica and `Prepare` takes a single round-trip to all replicas in all shards.

TAPIR is designed to be layered atop IR in a replicated, transactional storage system. Together, TAPIR and IR eliminate the redundancy in the replicated transactional system, as shown in Figure 2. As a comparison, Figure 6 shows the coordination required for the same read-write transaction in TAPIR with the following benefits: (1) TAPIR does not have any leaders or centralized coordination, (2) TAPIR `Reads` always go to the closest replica, and (3) TAPIR `Commit` takes a single round-trip to the participants in the common case.

5.1 Overview

TAPIR is designed to provide distributed transactions for a scalable storage architecture. This architecture 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, not end-hosts or user machines. They have access to a directory of storage servers using a service like Chubby [8] or ZooKeeper [21] and directly map data to servers using a technique like consistent hashing [22].

TAPIR provides a general storage and transaction interface for applications via a client-side library. Note that TAPIR is the application protocol for IR; applications using TAPIR do not interact with IR directly.

Client Interface

`Begin()` `Read(`*key*`)`→*object* `Abort()`
`Commit()`→TRUE/FALSE `Write(`*key*,*object*`)`

Client State

- *client id* - unique client identifier
- *transaction* - ongoing transaction *id*, *read set*, *write set*

Replica Interface

`Read(`*key*`)`→*object*,*version* `Commit(`*txn*,*timestamp*`)`
`Abort(`*txn*,*timestamp*`)`
`Prepare(`*txn*,*timestamp*`)`→PREPARE-OK/ABSTAIN/ABORT/(RETRY, *t*)

Replica State

- *prepared list* - list of transactions replica is prepared to commit
- *transaction log* - log of committed and aborted transactions
- *store* - versioned data store

Figure 7: Summary of TAPIR interfaces and client and replica state.

A TAPIR application `Begin`s a transaction, then executes `Read`s and `Write`s during the transaction's *execution period*. During this period, the application can `Abort` the transaction. Once it finishes execution, the application `Commit`s the transaction. Once the application calls `Commit`, it can no longer abort the transaction. The 2PC protocol will run to completion, committing or aborting the transaction based entirely on the decision of the participants. As a result, TAPIR's 2PC coordinators cannot make commit or abort decisions and do not have to be fault-tolerant. This property allows TAPIR to use clients as 2PC coordinators, as in MDCC [23], to reduce the number of round-trips to storage servers.

TAPIR provides the traditional ACID guarantees with the strictest level of isolation: strict serializability (or linearizability) of committed transactions.

5.2 Protocol

TAPIR provides transaction guarantees using a *transaction processing protocol*, *IR functions*, and a *coordinator recovery protocol*.

Figure 7 shows TAPIR's interfaces and state at clients and replicas. Replicas keep committed and aborted 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 2PC-based protocols, each TAPIR replica also maintains a *prepared list* of transactions that it has agreed to commit.

Each TAPIR client supports one ongoing transaction at a time. In addition to its *client id*, the client stores the state for the ongoing *transaction*, including the *transaction id* and *read and write sets*. The transaction id must be unique, so the client uses a tuple of its client id and *transaction counter*, similar to IR. TAPIR does not require synchronous disk writes at the client or the replicas, as clients do not have to be fault-tolerant and replicas use IR.

```
TAPIR-EXEC-CONSENSUS(op)
 1  txn = op.args.txn
 2  timestamp = op.args.timestamp
 3  if txn.id ∈ txn-log
 4      if txn − log[txn.id].status == COMMITTED
 5          return PREPARE-OK
 6      else
 7          return ABORT
 8      elseif txn.id ∈ prepared-list
 9          return PREPARE-OK
10      else
11          return TAPIR-OCC-CHECK(txn, timestamp)
```

Figure 8: Since `Prepare` is TAPIR's only **consensus** operations, TAPIR-EXEC-CONSENSUS simply runs TAPIR's prepare algorithm at replicas.

5.2.1 Transaction Processing

We begin with TAPIR's protocol for executing transactions.

1. For `Write`(*key*, *object*), the client buffers *key* and *object* in the write set until commit and returns immediately.
2. For `Read`(*key*), if *key* is in the transaction's write set, the client returns *object* from the write set. If the transaction has already read *key*, it returns a cached copy. Otherwise, the client sends `Read`(*key*) to the replica.
3. On receiving `Read`, the replica returns *object* and *version*, where *object* is the latest version of *key* and *version* is the timestamp of the transaction that wrote that version.
4. On response, the client puts (*key, version*) into the transaction's read set and returns *object* to the application.

Once the application calls `Commit` or `Abort`, the execution phase finishes. To commit, the TAPIR client coordinates across all *participants* – the shards that are responsible for the keys in the read or write set – to find a single timestamp, consistent with the strict serial order of transactions, to assign the transaction's reads and writes, as follows:

1. The TAPIR client selects a *proposed timestamp*. Proposed timestamps must be unique, so clients use a tuple of their local time and their *client id*.
2. The TAPIR client invokes `Prepare`(*txn*, *timestamp*) as an IR **consensus** operation, where *timestamp* is the proposed timestamp and *txn* includes the transaction id (*txn.id*) and the transaction read (*txn.read_set*) and write sets (*txn.write_set*). The client invokes `Prepare` on all participants through IR as a **consensus** operations.
3. Each TAPIR replica that receives `Prepare` (invoked by IR through `ExecConcensus`) first checks its transaction log for *txn.id*. If found, it returns PREPARE-OK if the transaction committed or ABORT if the transaction aborted.
4. Otherwise, the replica checks if *txn.id* is already in its *prepared list*. If found, it returns PREPARE-OK.
5. Otherwise, the replica runs TAPIR's *OCC validation checks*, which check for conflicts with the transaction's read and write sets at *timestamp*, shown in Figure 9.
6. Once the TAPIR client receives results from all shards, the client sends `Commit`(*txn*, *timestamp*) if all shards replied

```
TAPIR-OCC-CHECK(txn, timestamp)
 1  for ∀key, version ∈ txn.read-set
 2      if version < store[key].latest-version
 3          return ABORT
 4      elseif version < MIN(prepared-writes[key])
 5          return ABSTAIN
 6  for ∀key ∈ txn.write-set
 7      if timestamp < MAX(PREPARED-READS(key))
 8          return RETRY, MAX(PREPARED-READS(key))
 9      elseif timestamp < store[key].latestVersion
10          return RETRY, store[key].latestVersion
11  prepared-list[txn.id] = timestamp
12  return PREPARE-OK
```

Figure 9: Validation function for checking for OCC conflicts on `Prepare`. PREPARED-READS and PREPARED-WRITES get the proposed timestamps for all transactions that the replica has prepared and read or write to *key*, respectively.

PREPARE-OK or `Abort`(*txn*, *timestamp*) if any shards replied ABORT or ABSTAIN. If any shards replied RETRY, then the client retries with a new proposed timestamp (up to a set limit of retries).

7. On receiving a `Commit`, the TAPIR replica: (1) commits the transaction to its transaction log, (2) updates its versioned store with *w*, (3) removes the transaction from its prepared list (if it is there), and (4) responds to the client.
8. On receiving a `Abort`, the TAPIR replica: (1) logs the abort, (2) removes the transaction from its prepared list (if it is there), and (3) responds to the client.

Like other 2PC-based protocols, TAPIR can return the outcome of the transaction to the application as soon as `Prepare` returns from all shards (in Step 6) and send the `Commit` operations asynchronously. As a result, using IR, TAPIR can commit a transaction with a *single round-trip* to all replicas in all shards.

5.2.2 IR Support

Because TAPIR's `Prepare` is an IR **consensus** operation, TAPIR must implement a client-side *decide* function, shown in Figure 10, which merges inconsistent `Prepare` results from replicas in a shard into a single result. TAPIR-DECIDE is simple: if a majority of the replicas replied PREPARE-OK, then it commits the transaction. This is safe because no conflicting transaction could also get a majority of the replicas to return PREPARE-OK.

TAPIR also supports `Merge`, shown in Figure 11, and `Sync`, shown in Figure 12, at replicas. TAPIR-MERGE first removes any prepared transactions from the leader where the `Prepare` operation is TENTATIVE. This step removes any inconsistencies that the leader may have because it executed a `Prepare` differently – out-of-order or missed – by the rest of the group.

The next step checks *d* for any PREPARE-OK results that might have succeeded on the IR fast path and need to be preserved. If the transaction has not committed or aborted already, we re-run TAPIR-OCC-CHECK to check for conflicts

```
TAPIR-DECIDE(results)
1  if ABORT ∈ results
2      return ABORT
3  if count(PREPARE-OK, results) ≥ f + 1
4      return PREPARE-OK
5  if count(ABSTAIN, results) ≥ f + 1
6      return ABORT
7  if RETRY ∈ results
8      return RETRY, max(results.retry-timestamp)
9  return ABORT
```

Figure 10: *TAPIR's decide function.* IR runs this if replicas return different results on `Prepare`.

```
TAPIR-MERGE(d, u)
 1  for ∀op ∈ d ∪ u
 2      txn = op.args.txn
 3      if txn.id ∈ prepared-list
 4          DELETE(prepared-list, txn.id)
 5  for op ∈ d
 6      txn = op.args.txn
 7      timestamp = op.args.timestamp
 8      if txn.id ∉ txn-log and op.result == PREPARE-OK
 9          R[op].result = TAPIR-OCC-CHECK(txn, timestamp)
10      else
11          R[op].result = op.result
12  for op ∈ u
13      txn = op.args.txn
14      timestamp = op.args.timestamp
15      R[op].result = TAPIR-OCC-CHECK(txn, timestamp)
16  return R
```

Figure 11: *TAPIR's merge function.* IR runs this function at the leader on synchronization and recovery.

with other previously prepared or committed transactions. If the transaction *conflicts*, then we know that its PREPARE-OK did not succeed at a fast quorum, so we can change it to ABORT; otherwise, for correctness, we must preserve the PREPARE-OK because TAPIR may have moved on to the commit phase of 2PC. Further, we know that it is safe to preserve these PREPARE-OK results because, if they conflicted with another transaction, the conflicting transaction *must* have gotten its consensus result on the IR slow path, so if TAPIR-OCC-CHECK did not find a conflict, then the conflicting transaction's `Prepare` must not have succeeded.

Finally, for the operations in *u*, we simply decide a result for each operation and preserve it. We know that the leader is now consistent with $f+1$ replicas, so it can make decisions on consensus result for the majority.

TAPIR's sync function, shown in Figure 12, runs at the other replicas to reconcile TAPIR state with the master records, correcting missed operations or consensus results where the replica did not agree with the group. It simply applies operations and consensus results to the replica's state: it logs aborts and commits, and prepares uncommitted transactions where the group responded PREPARE-OK.

```
TAPIR-SYNC(R)
 1  for ∀op ∈ R
 2      if op ∉ r or op.result ≠ r[op].result
 3          txn = op.args.txn
 4          timestamp = op.args.timestamp
 5          if op.func == Prepare
 6              if op.result == PREPARE-OK
 7                  if txn.id ∉ prepared-list and txn.id ∉ txn-log
 8                      prepared-list[txn.id] = timestamp
 9              elseif txn.id ∈ prepared-list
10                  DELETE(prepared-list, txn.id)
11          else
12              txn-log[txn.id].txn = txn
13              txn-log[txn.id].timestamp = timestamp
14              if op.func == Commit
15                  txn-log[txn.id].status = COMMITTED
16              else
17                  txn-log[txn.id].status = ABORTED
18              if txn.id ∈ prepared-list
19                  DELETE(prepared-list, txn.id)
```

Figure 12: *TAPIR's function for synchronizing inconsistent replica state.* IR runs this on each replica except the leader during synchronization. r is the replica's local record.

5.2.3 Coordinator Recovery

If a client fails while in the process of committing a transaction, TAPIR ensures that the transaction runs to completion (either commits or aborts). Further, the client may have returned the commit or abort to the application, so we must ensure that the client's commit decision is preserved. For this purpose, TAPIR uses the *cooperative termination protocol* defined by Bernstein [6] for coordinator recovery and used by MDCC [23]. TAPIR designates one of the participant shards as a *backup shard*, the replicas in which can serve as a backup coordinator if the client fails. As observed by MDCC, because coordinators cannot unilaterally abort transactions (i.e., if a client receives $f+1$ PREPARE-OK responses from each participant, it must commit the transaction), a backup coordinator can safely complete the protocol without blocking. However, we must ensure that no two coordinators for a transaction are active at the same time.

Coordinator Changes. We use a coordinator change protocol, similar to IR's view change protocol to ensure that only one coordinator is active at a time.[3] For each transaction, we designate one of the participant shards as a *backup shard*. The initial coordinator for every transaction is the client. In every subsequent view, the currently active backup coordinator is a replica from the backup shard.

For every transaction in its *prepared-list*, each TAPIR replica keeps the transaction's backup shard and a current *coordinator view*. Replicas only process and respond to `Prepare`, `Commit` and `Abort` operations from the active coordinator designated by the current view, identified by indexing into the list of backup shard replicas with the coordinator view number. Replicas also keep a *no-vote list* with transactions that the replica knows a backup coordinator may abort.

If the current coordinator is suspected to have failed, any of the participants can initiate a coordinator change. In doing so, it keeps the client or any previous backup coordinator from sending operations to the participating replicas. The new coordinator can then poll the participant using `Prepare`, and make a commit decision without interference from other coordinators. The election protocol for a new backup coordinator progresses as follows:

1. Any replica in any participant shard calls `CoordinatorChange` through IR as a **consensus** operation on the backup shard.
2. Each replica that executes `CoordinatorChange` through IR, increments and returns its current coordinator view number v. If the replica is not already in the COORDINATOR-VIEW-CHANGE state, it sets its state to COORDINATOR-VIEW-CHANGE and stops responding to operations for that transaction.
3. The *decide* function for `CoordinatorChange` returns the highest v returned by the replicas.
4. Once `CoordinatorChange` returns successfully, the replica sends `StartCoordinatorView`(v_{new}), where v_{new} is the returned view number from `CoordinatorChange`, as an IR **inconsistent** operation to *all* participant shards, including its own.
5. Any replica that receives `StartCoordinatorView` checks if v_{new} is higher or equal to its current view. If so, the replica updates its current view number and begins accepting `Prepare`, `Commit` and `Abort` from the active backup coordinator designated by the new view. If the replica is in the backup shard, it can set its state back to NORMAL.
6. When a replica executes `StartCoordinatorView` for the view where it is the designated backup coordinator, it begins the cooperative termination protocol.

The `Merge` function for `CoordinatorChange` preserves the consensus result if it is greater than or equal to the current view number at the leader during synchronization. The `Sync` function for `CoordinatorChange` sets the replica state to COORDINATOR-VIEW-CHANGE if the consensus result is larger than the replica's current view number. The `Sync` function for `StartCoordinatorView` just executes the function: it updates the replica's current view number if v_{new} is greater than or equal to it and sets the state back to NORMAL if the replica is in the backup shard.

Cooperative Termination. The backup coordination protocol executed by the active coordinator is similar to the cooperative termination protocol described by Bernstein [6], with changes to accommodate IR and TAPIR. The most notable changes are that the backup coordinators do not propose timestamps. If the client successfully prepared the transaction at a timestamp t (i.e., achieved at least $f+1$ PREPARE-OK in every participant shard), then the transaction will commit at t.

[3] Other possible ways to achieve this goal include logging the currently active backup coordinator to a service like Chubby [8] or ZooKeeper [21], or giving each backup coordinator a lease in turn.

```
TAPIR-RECOVERY-DECIDE(results)
1  if ABORT ∈ results
2      return ABORT if count(NO-VOTE, results) ≥ f + 1
3      return ABORT
4  if count(PREPARE-OK, results) ≥ f + 1
5      return PREPARE-OK
6  return RETRY
```

Figure 13: *TAPIR's decide function for* `Prepare` *on coordinator recovery.* IR runs this if replicas return different results on `Prepare`. This *decide* function differs from the normal case execution decide because it is not safe to return ABORT unless it is sure the original coordinator did not receive PREPARE-OK.

Otherwise, the backup coordinator will eventually abort the transaction.

Next, in Bernstein's algorithm, any single participant can abort the transaction if they have not yet voted (i.e., replied to a coordinator). However, with IR, no single replica can abort the transaction without information about the state of the other replicas in the shard. As a result, replicas return a NO-VOTE response and add the transaction to their *no-vote-list*. Once a replica adds a transaction to the NO-VOTE-LIST, it will always respond NO-VOTE to `Prepare` operations. Eventually, all replicas in the shard will either converge to a response (i.e., PREPARE-OK, ABORT) to the original coordinator's `Prepare` or to a NO-VOTE response. TAPIR's modified cooperative termination protocol proceeds as follows:

1. The backup coordinator polls the participants with `Prepare` with no proposed timestamp by invoking `Prepare` as a **consensus** operation in IR with the *decide* function outlined in Figure 13.
2. Any replica that receives `Prepare` with no propose timestamp, returns PREPARE-OK if it has committed or prepared the transaction, ABORT if it has received an `Abort` for the transaction or committed a conflicting transaction and NO-VOTE if it does not have the transaction in its *prepared-list* or *txn-log*. If the replica returns NO-VOTE, it adds the transaction to its *no-vote-list*.
3. The coordinator continues to send `Prepare` as an IR operation until it either receives a ABORT or PREPARE-OK from all participant shards. Note that the result will be ABORT if a majority of replicas respond NO-VOTE.
4. If all participant shards return PREPARE-OK, the coordinator sends `Commit`; otherwise, it sends `Abort`.

Assuming $f+1$ replicas are up in each participant shard and shards are able to communicate, this process will eventually terminate with a backup coordinator sending `Commit` or `Abort` to all participants.

We must also incorporate the NO-VOTE into our `Merge` and `Sync` handlers for `Prepare`. We make the following changes to `Merge` for the final function shown in Figure 14: (lines 5-6) delete any tentative NO-VOTES from the *no-vote-list* at the leader for consistency, (lines 10-11) return NO-VOTE without running TAPIR-OCC-CHECK if the transaction is already in the *no-vote-list* because any result to the original `Prepare`

```
TAPIR-MERGE(d, u)
 1  for ∀op ∈ d ∪ u
 2      txn = op.args.txn
 3      if txn.id ∈ prepared-list
 4          DELETE(prepared-list, txn.id)
 5      if txn.id ∈ no-vote-list
 6          DELETE(no-vote-list, txn.id)
 7  for op ∈ d
 8      txn = op.args.txn
 9      timestamp = op.args.timestamp
10      if txn.id ∈ no-vote-list
11          R[op].result = NO-VOTE
12      elseif txn.id ∉ txn-log and op.result == PREPARE-OK
13          R[op].result = TAPIR-OCC-CHECK(txn, timestamp)
14      else
15          R[op].result = op.result
16  for op ∈ u
17      txn = op.args.txn
18      if txn.id ∈ no-vote-list
19          R[op].result = NO-VOTE
20      else
21          timestamp = op.args.timestamp
22          R[op].result = TAPIR-OCC-CHECK(txn, timestamp)
23  return R
```

Figure 14: *TAPIR's merge function.* IR runs this function at the leader on synchronization and recovery. This version handles NO-VOTE results.

```
TAPIR-SYNC(R)
 1  for ∀op ∈ R
 2      if op ∉ r or op.result ≠ r[op].result
 3          txn = op.args.txn
 4          timestamp = op.args.timestamp
 5          if op.func == Prepare
 6              if op.result == PREPARE-OK
 7                  if txn.id ∉ prepared-list and txn.id ∉ txn-log
 8                      prepared-list[txn.id] = timestamp
 9              elseif txn.id ∈ prepared-list
10                  DELETE(prepared-list, txn.id)
11                  if op.result == NO-VOTE and txn.id ∉ txn-log
12                      no-vote-list[txn.id] = timestamp
13          else
14              txn-log[txn.id].txn = txn
15              txn-log[txn.id].timestamp = timestamp
16              if op.func == Commit
17                  txn-log[txn.id].status = COMMITTED
18              else
19                  txn-log[txn.id].status = ABORTED
20              if txn.id ∈ prepared-list
21                  DELETE(prepared-list, txn.id)
```

Figure 15: *TAPIR's function for synchronizing inconsistent replica state.* IR runs this on each replica except the leader during synchronization. r is the replica's local record.

could not have succeeded, (lines 18-19) do the same for operations without majority result where the original coordinator's `Prepare` definitely did not succeed. If the consensus result to the `Prepare` is NO-VOTE in `Sync`, we add transactions to the *no-vote-list* and remove it from the *prepared-list*, as shown in lines 11-12 of Figure 15.

5.3 Correctness

To prove correctness, we show that TAPIR maintains the following properties[4] 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.

Appendix B gives a TLA+ [27] specification for TAPIR with IR, which we have model-checked for correctness.

5.3.1 Isolation

For correctness, we must show that any two conflicting transactions, A and B, that violate the linearizable transaction ordering cannot both commit. If A and B have a conflict, then there must be at least one common shard that is participating in both A and B. We show that, in that shard, `Prepare`(A) and `Prepare`(B) cannot both return PREPARE-OK, so one transaction must abort.

In the common shard, IR's visibility property (P2) guarantees that `Prepare`(A) must be *visible* to `Prepare`(B) (i.e., executes first at one replica out of every $f+1$ quorum) *or* `Prepare`(B) is visible to `Prepare`(A). Without loss of generality, suppose that `Prepare`(A) is visible to `Prepare`(B) and the group returns PREPARE-OK to `Prepare`(A). Any replica that executes TAPIR-OCC-CHECK for both A and B will not return PREPARE-OK for both, so at least one replica out of any $f+1$ quorum will not return PREPARE-OK to `Prepare`(B). IR will not get a fast quorum of matching PREPARE-OK results for `Prepare`(B), and TAPIR's *decide* function will not return PREPARE-OK because it will never get the $f+1$ matching PREPARE-OK results that it needs. Thus, IR will never return a consensus result of PREPARE-OK for `Prepare`(B). The same holds if `Prepare`(B) is visible to `Prepare`(A) and the group returns PREPARE-OK to `Prepare`(B). Thus, IR will never return a successful consensus result of PREPARE-OK to executing both `Prepare`(A) and `Prepare`(B) in the common participant shard and TAPIR will not be able to commit both transactions.

Further, once decided, the successful consensus results for `Prepare`(A) and `Prepare`(B) will persist in the record of at least one replica out of every quorum, unless it has been modified by the application through `Merge`. TAPIR will never change another result to a PREPARE-OK, so the shard will never respond PREPARE-OK to both transactions. IR will ensure that the successful consensus result is eventually `Sync`'d at all replicas. Once a TAPIR replica prepared a transaction, it will continue to return PREPARE-OK until it receives a `Commit` or `Abort` for the transaction. As a result, if the shard returned PREPARE-OK as a successful consensus result to `Prepare`(A), then it will never allow `Prepare`(B) to also return PREPARE-OK (unless A aborts), ensuring that B is never able to commit. The opposite also holds true.

5.3.2 Atomicity

If a transaction commits at any participating shard, the TAPIR client must have received a successful PREPARE-OK from every participating shard on `Prepare`. Barring failures, it will ensure that `Commit` eventually executes successfully at every participant. TAPIR replicas always execute `Commit`, even if they did not prepare the transaction, so `Commit` will eventually commit the transaction at every participant if it executes at one participant.

If the coordinator fails, then at least one replica in a participant shard will detect the failure and initiate the coordinator recovery protocol. Assuming no more than f simultaneous failures in the backup shard, the coordinator change protocol will eventually pick a new active backup coordinator from the backup shard. At this point, the participants will have stopped processing operations from the client and any previous backup coordinators.

Backup coordinators do not propose timestamps, so if any replica in a participant shard received a `Commit`, then the client's `Prepare` must have made it into the operation set of every participant shard with PREPARE-OK as the consensus result. IR's consensus result and eventual consistency properties (P3 and P4) ensure that the PREPARE-OK will eventually be applied at all replicas in every participant shard and TAPIR ensures that successful PREPARE-OK results are not changed in `Merge` (as shown above). Once a TAPIR replica applies PREPARE-OK, it will continue to return PREPARE-OK, so once replicas in participant groups have stopped processing operations from previous coordinators, all non-failed replicas in all shards will eventually return PREPARE-OK. As a result, the backup coordinator must eventually receive PREPARE-OK as well from all participants.

In the meantime, the backup coordinator is guaranteed to not receive an ABORT from a participant shard. A participant shard will only return an ABORT if: (1) a conflicting transaction committed, (2) a majority of the replicas return NO-VOTE because they did not have a record of the transaction, or (3) the transaction was aborted on the shard. Case (1) is not possible because the conflicting transaction could not have also received a successful consensus result of PREPARE-OK (based on our isolation proof) and IR's consensus result property (P3) ensures that the conflicting transaction could never get a PREPARE-OK consensus result, so the conflicting transaction cannot commit. Case (2) is not possible because the client could not have received PREPARE-OK as a consensus result if a majority of the replicas do not have the transaction in their *prepared-list* and IR's P3 and P4 ensures the transaction eventually makes its way into the *prepared-list* of every replica. Case (3) is not possible because the client could not have sent `Abort` if it got PREPARE-OK from all participant shards and no previous backup coordinator could have sent

[4] We do not prove database consistency, as it depends on application invariants; however, strict serializability is sufficient to enforce consistency.

`Abort` because cases (1) and (2) will never happen. As a result, the backup coordinator will not abort the transaction.

5.3.3 Durability

For all committed transactions, either the client or a backup coordinator will eventually execute `Commit` successfully as an IR **inconsistent** operation. IR guarantees that the `Commit` will never be lost (P1) and every replica will eventually execute or synchronize it. On `Commit`, TAPIR replicas use the transaction timestamp included in `Commit` to order the transaction in their log, regardless of when they execute it, thus maintaining the original linearizable ordering. If there are no coordinator failures, a transaction would eventually be finalized through an IR inconsistent operation (`Commit/Abort`), which ensures that the decision would never be lost. As described above, for coordinator failures, the coordinator recovery protocol ensures that a backup coordinator would eventually send `Commit` or `Abort` to all participants.

6. TAPIR Extensions

We now describe four useful extensions to TAPIR.

6.1 Read-only Transactions

Since it uses a multi-versioned store, TAPIR easily supports globally-consistent read-only transactions at a timestamp. 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.

More precisely, the protocol for read-only transactions follows:

1. The TAPIR client chooses a *snapshot timestamp* for the transaction; for example, the client's 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 could not get the value from the replica, it 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 brief sketch of correctness, it is always safe to read a version of the key that is *validated* at the snapshot timestamp. The version will always be valid at the snapshot timestamp because the write timestamp for the version is earlier than the snapshot timestamp and the read timestamp is after the snapshot timestamp. If the replica does not have a validated version, the replicated `QuorumRead` ensures that: (1) the client gets the latest version of the object (because at least 1 of any $f+1$ replicas must have it), and (2) a later write transaction cannot overwrite the version (because at least 1 of the $f+1$ `QuorumRead` replicas will block it).

Since TAPIR also uses loosely synchronized clocks, it could be combined with Spanner's algorithm for providing externally consistent read-only transactions as well. This combination would require Spanner's TrueTime technology and waits at the client for the TrueTime uncertainty bound. Note that while TAPIR itself provides external consistency for read-write transactions regardless of clock skew, this read-only protocol would provide linearizability guarantees only if the clock skew did not exceed the TrueTime bound, like Spanner [13].

6.2 Serializability

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

However, if TAPIR's guarantees were weakened to (non-strict) *serializability*, then it can then accept proposed timestamps any time in the past as long as they respect the serializable transaction ordering. This optimization requires tracking the timestamp of the transaction that last read and wrote each version.

With this optimization, TAPIR can now accept: (1) reads of past versions, as long as the read timestamp precedes the write timestamp of the next version, and (2) writes in the past (per the Thomas Write Rule [45]), as long as the write timestamp follows the read timestamp of the previous version and precedes the write timestamp of the next version.

6.3 Synchronous Log Writes

Given the ability to synchronously log to durable storage (e.g. hard disk, NVRAM), we can reduce TAPIR's quorum requirements. 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 $f+1$.

6.4 Retry Timestamp Selection

A client can increase the likelihood that participant replicas will accept its proposed timestamp by proposing a very large timestamp; this decreases the likelihood that the participant replicas have already accepted a higher timestamp. Thus, to decrease the chances of retrying forever, clients can exponentially increase their proposed timestamp on each retry.

6.5 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 will arrive. After the wait, the replica can process transactions in timestamp order. This wait increases the chances that the participant replica can process transactions in timestamp order and decreases the number of transactions that it will have to reject for arriving out of order.

7. Evaluation

In this section, our experiments demonstrate the following:

- TAPIR provides better latency *and* throughput than conventional transaction protocols in both the datacenter and wide-area environments.
- TAPIR's abort rate scales similarly to other OCC-based transaction protocols as contention increases.
- Clock synchronization sufficient for TAPIR's needs is widely available in both datacenter and wide-area environments.
- TAPIR provides performance comparable to systems with weak consistency guarantees and no transactions.

7.1 Experimental Setup

We ran our experiments on Google Compute Engine [19] (GCE) with VMs spread across 3 geographical regions – US, Europe and Asia – and placed in different availability zones within each geographical region. Each server has a virtualized, single core 2.6 GHz Intel Xeon, 8 GB of RAM and 1 Gb NIC.

7.1.1 Testbed Measurements

As TAPIR's performance depends on clock synchronization and round-trip times, we first present latency and clock skew measurements of our test environment. As clock skew increases, TAPIR's latency increases and throughput decreases because clients may have to retry more `Prepare` operations. It is important to note that TAPIR's performance depends on the *actual* clock skew, not a worst-case bound like Spanner [13].

We measured the clock skew by sending a ping message with timestamps taken on either end. We calculate skew by comparing the timestamp taken at the destination to the one taken at the source plus half the round-trip time (assuming that network latency is symmetric). Table 1 reports the average skew and latency between the three geographic regions. Within each region, we average over the availability zones. Our VMs benefit from Google's reliable wide-area network infrastructure; although we use UDP for RPCs over the wide-area, we saw negligible packet drops and little variation in round-trip times.

Table 1: Measured RTTs 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	–	0.8	261.8	–	0.1	1.9
Asia	–	–	10.8	–	–	.3

The average RTT between US-Europe was 110 ms; US-Asia was 165 ms; Europe-Asia was 260 ms. We found the clock skew to be low, averaging between 0.1 ms and 3.4 ms, demonstrating the feasibility of synchronizing clocks in the wide area. However, there was a long tail to the clock skew, with the worst case clock skew being around 27 ms – making it significant that TAPIR's performance depends on actual rather than worst-case clock skew. As our measurements show, the skew in this environment is low enough to achieve good performance.

7.1.2 Implementation

We implemented TAPIR in a transactional key-value storage system, called TAPIR-KV. Our prototype consists of 9094 lines of C++ code, not including the testing framework.

We also built two comparison systems. The first, OCC-STORE, is a "standard" implementation of 2PC and OCC, combined with an implementation of Multi-Paxos [28]. Like TAPIR, OCC-STORE accumulates a read and write set with read versions at the client during execution and then runs 2PC with OCC checks to commit the transaction. OCC-STORE uses a centralized timestamp server to generate transaction timestamps, which we use to version data in the multi-versioned storage system. We verified that this timestamp server was not a bottleneck in our experiments.

Our second system, LOCK-STORE is based on the Spanner protocol [13]. Like Spanner, it uses 2PC with S2PL and Multi-Paxos. The client acquires read locks during execution at the Multi-Paxos leaders and buffers writes. On `Prepare`, the leader replicates these locks and acquires write locks. We use loosely synchronized clocks at the leaders to pick transaction timestamps, from which the coordinator chooses the largest as the commit timestamp. We use the client as the coordinator, rather than one of the Multi-Paxos leaders in a participant shard, for a more fair comparison with TAPIR-KV. Lacking access to TrueTime, we set the TrueTime error bound to 0,

Table 2: Transaction profile for Retwis workload.

Transaction Type	# gets	# puts	workload %
Add User	1	3	5%
Follow/Unfollow	2	2	15%
Post Tweet	3	5	30%
Load Timeline	rand(1,10)	0	50%

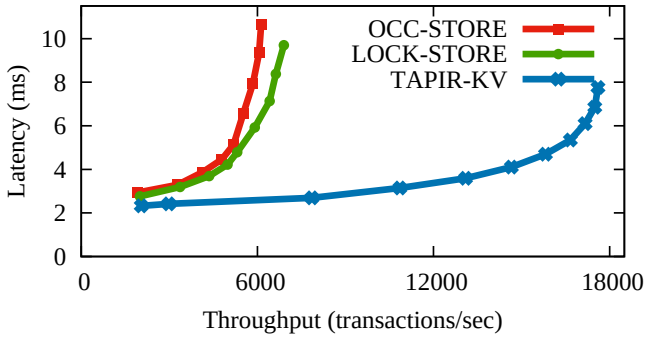

Figure 16: Average Retwis transaction Latency (Zipf coefficient 0.75) versus throughput within a datacenter.

eliminating the need to wait out clock uncertainty and thereby giving the benefit to this protocol.

7.1.3 Workload

We use two workloads for our experiments. We first test using a synthetic workload based on the Retwis application [31]. Retwis is an open-source Twitter clone designed to use the Redis key-value storage system [40]. 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 as shown in Table 2.

Our second experimental workload is YCSB+T [16], an extension of YCSB [12] – a commonly-used benchmark for key-value storage systems. YCSB+T wraps database operations inside simple transactions such as read, insert or read-modify-write. However, we use our Retwis benchmark for many experiments because it is more sophisticated: transactions are more complex – each touches 2.5 shards on average – and longer – each executes 4-10 operations.

7.2 Single Datacenter Experiments

We begin by presenting TAPIR-KV's performance within a single datacenter. We deploy TAPIR-KV and the comparison systems over 10 shards, all in the US geographic region, with 3 replicas for each shard in different availability zones. We populate the systems with 10 million keys and make transaction requests with a Zipf distribution (coefficient 0.75) using an increasing number of closed-loop clients.

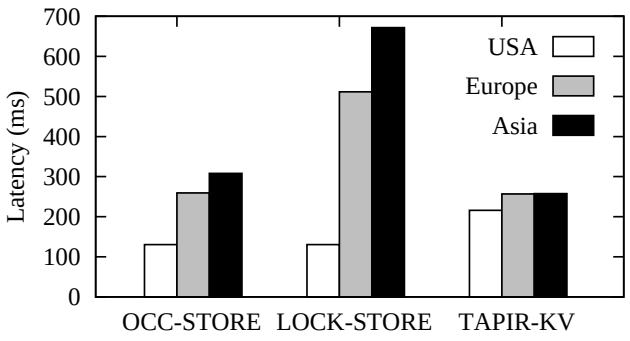

Figure 17: Average wide-area latency for Retwis transactions, with leader located in the US and client in US, Europe or Asia.

Figure 16 shows the average latency for a transaction in our Retwis workload at different throughputs. At low offered load, TAPIR-KV has lower latency because it is able to commit transactions in a single round-trip to all replicas, whereas the other systems need two; its commit latency is thus reduced by 50%. However, Retwis transactions are relatively long, so the difference in *transaction* latency is relatively small.

Compared to the other systems, TAPIR-KV is able to provide roughly 3× the peak throughput, which stems directly from IR's weak guarantees: it has no leader and does not require cross-replica coordination. Even with moderately high contention (Zipf coefficient 0.75), TAPIR-KV replicas are able to inconsistently execute operations and still agree on ordering for transactions at a high rate.

7.3 Wide-Area Latency

For wide-area experiments, we placed one replica from each shard in each geographic region. For systems with leader-based replication, we fix the leader's location in the US and move the client between the US, Europe and Asia. Figure 17 gives the average latency for Retwis transactions using the same workload as in previous section.

When the client shares a datacenter with the leader, the comparison systems are faster than TAPIR-KV because TAPIR-KV must wait for responses from all replicas, which takes 160 ms to Asia, while OCC-STORE and LOCK-STORE can commit with a round-trip to the local leader and one other replica, which is 115 ms to Europe.

When the leader is in a different datacenter, LOCK-STORE suffers because it must go to the leader on `Read` for locks, which takes up to 160 ms from Asia to the US, while OCC-STORE can go to a local replica on `Read` like TAPIR-KV. In our setup TAPIR-KV takes longer to `Commit`, as it has to contact the *furthest* replica, and the RTT between Europe and Asia is more expensive than two round-trips between US to Europe (likely because Google's traffic goes through the US). In fact, in this setup, IR's slow path, at two RTT to the two closest replicas, is *faster* than its fast path, at one RTT to the furthest

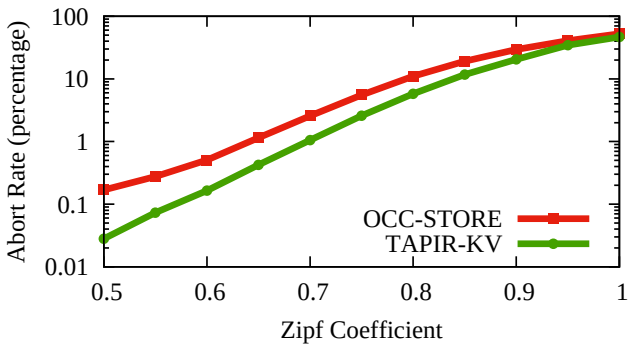

Figure 18: Abort rates at varying Zipf co-efficients with a constant load of 5,000 transactions/second in a single datacenter.

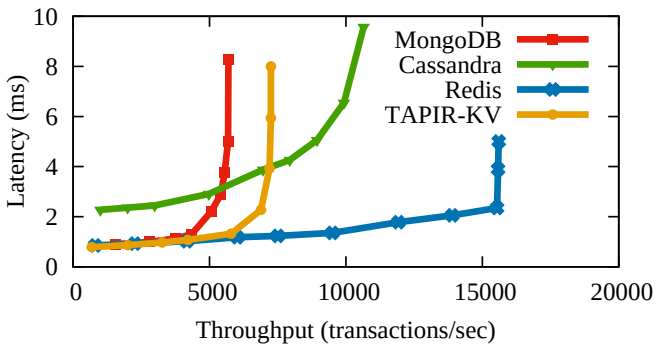

Figure 19: Comparison with weakly consistent storage systems.

replica. We do not implement the optimization of running the fast and slow paths in parallel, which could provide better latency in this case.

7.4 Abort and Retry Rates

TAPIR is an optimistic protocol, so transactions can abort due to conflicts, as in other OCC systems. Moreover, TAPIR transactions can also be forced to abort or retry when conflicting timestamps are chosen due to clock skew. We measure the abort rate of TAPIR-KV compared to OCC-STORE, a conventional OCC design, for varying levels of contention (varying Zipf coefficients). These experiments run in a single region with replicas in three availability zones. We supply a constant load of 5,000 transactions/second.

With a uniform distribution, both TAPIR-KV and OCC-STORE have very low abort rates: 0.005% and 0.04%, respectively. Figure 18 gives the abort rate for Zipf co-efficients from 0.5 to 1.0. At lower Zipf co-efficients, TAPIR-KV has abort rates that are roughly an order of magnitude lower than OCC-STORE. TAPIR's lower commit latency and use of optimistic timestamp ordering reduce the time between `Prepare` and `Commit` or `Abort` to a single round-trip, making transactions less likely to abort.

Under heavy contention (Zipf coefficient 0.95), both TAPIR-KV and OCC-STORE have moderately high abort rates: 36% and 40%, respectively, comparable to other OCC-based systems like MDCC [23]. These aborts are primarily due to the most popular keys being accessed very frequently. For these workloads, locking-based systems like LOCK-STORE would make better progress; however, clients would have to wait for extended periods to acquire locks.

TAPIR rarely needs to retry transactions due to clock skew. Even at moderate contention rates, and with simulated clock skew of up to 50 ms, we saw less than 1% TAPIR retries and negligible increase in abort rates, demonstrating that commodity clock synchronization infrastructure is sufficient.

7.5 Comparison with Weakly Consistent Systems

We also compare TAPIR-KV with three widely-used eventually consistent storage systems, MongoDB [36], Cassandra [26], and Redis [40]. For these experiments, we used YCSB+T [16], with a single shard with 3 replicas and 1 million keys. MongoDB and Redis support master-slave replication; we set them to use synchronous replication by setting `WriteConcern` to `REPLICAS_SAFE` in MongoDB and the `WAIT` command [42] for Redis. Cassandra uses `REPLICATION_FACTOR=2` to store copies of each item at any 2 replicas.

Figure 19 demonstrates that the latency and throughput of TAPIR-KV is comparable to these systems. We do not claim this to be an entirely fair comparison; these systems have features that TAPIR-KV does not. At the same time, the other systems do not support distributed transactions – in some cases, not even single-node transactions – while TAPIR-KV runs a distributed transaction protocol that ensures strict serializability. Despite this, TAPIR-KV's performance remains competitive: it outperforms MongoDB, and has throughput within a factor of 2 of Cassandra and Redis, demonstrating that strongly consistent distributed transactions are not incompatible with high performance.

8. Related Work

Inconsistent replication shares the same principle as past work on commutativity, causal consistency and eventual consistency: operations that do not require ordering are more efficient. TAPIR leverages IR's weak guarantees, in combination with optimistic timestamp ordering and optimistic concurrency control, to provide semantics similar to past work on distributed transaction protocols but with both lower latency and higher throughput.

8.1 Replication

Transactional storage systems currently rely on strict consistency protocols, like Paxos [28] and VR [38]. These protocols enforce a strict serial ordering of operations and no divergence of replicas. In contrast, IR is more closely related to

Table 3: Comparison of read-write transaction protocols in replicated transactional storage systems.

Transaction System	Replication Protocol	Read Latency	Commit Latency	Msg At Bottleneck	Isolation Level	Transaction Model
Spanner [13]	Multi-Paxos [28]	2 (leader)	4	$2n$ + reads	Strict Serializable	Interactive
MDCC [23]	Gen. Paxos [29]	2 (any)	3	$2n$	Read-Committed	Interactive
Repl. Commit [35]	Paxos [28]	$2n$	4	2	Serializable	Interactive
CLOCC [1, 32]	VR [38]	2 (any)	4	$2n$	Serializable	Interactive
Lynx [47]	Chain Repl. [46]	–	$2n$	2	Serializable	Stored procedure
TAPIR	IR	2 (to any)	2	2	Strict Serializable	Interactive

eventually consistent replication protocols, like Bayou [44], Dynamo [15] and others [25, 26, 41]. The key difference is that applications resolve conflicts after they happen with eventually consistent protocols, whereas IR **consensus** operations allow applications to decide conflicts and recover that decision later. As a result, applications can enforce higher-level guarantees (e.g., mutual exclusion, strict serializability) that they cannot with eventual consistency.

IR is also related to replication protocols that avoid coordination for *commutative operations* (e.g., Generalized Paxos [29], EPaxos [37]). These protocols are more general than IR because they do not require application invariants to be pairwise. For example, EPaxos could support invariants on bank account balances, while IR cannot. However, these protocols consider two operations to commute if their order does not matter when applied to *any* state, whereas IR requires only that they produce the same results *in a particular execution*. This is a form of state-dependent commutativity similar to SIM-commutativity [10]. As a result, in the example from Section 3.1.3, EPaxos would consider any operations on the same lock to conflict, whereas IR would allow two unsuccessful `Lock` operations to the same lock to commute.

8.2 Distributed Transactions

A technique similar to optimistic timestamp ordering was first explored by Thomas [45], while CLOCC [1] was the first to combine it with loosely synchronized clocks. We extend Thomas's algorithm to: (1) support multiple shards, (2) eliminate synchronous disk writes, and (3) ensure availability across coordinator failures. Spanner [13] and Granola [14] are two recent systems that use loosely synchronized clocks to improve performance for read-only transactions and independent transactions, respectively. TAPIR's use of loosely synchronized clocks differs from Spanner's in two key ways: (1) TAPIR depends on clock synchronization only for performance, not correctness, and (2) TAPIR's performance is tied to the *actual* clock skew, not TrueTime's worst-case estimated bound. Spanner's approach for read-only transactions complements TAPIR's high-performance read-write transactions, and the two could be easily combined.

CLOCC and Granola were both combined with VR [32] to replace synchronous disk writes with in-memory replication. These combinations still suffer from the same redundancy – enforcing ordering both at the distributed transaction and replication level – that we discussed in Section 2. Other layered protocols, like the examples shown in Table 3, have a similar performance limitation.

Some previous work included in Table 3 improves throughput (e.g., Warp [17], Transaction Chains [47], Tango [5]), while others improve performance for read-only transactions (e.g., MegaStore [4], Spanner [13]) or other limited transaction types (e.g., Sinfonia's mini-transactions [2], Granola's independent transactions [14], Lynx's transaction chains [47], and MDCC's commutative transactions [23]) or weaker consistency guarantees [34, 43]. In comparison, TAPIR is the first transaction protocol to provide better performance (both throughput and latency) for general-purpose, read-write transactions using replication.

9. Conclusion

This paper demonstrates that it is possible to build distributed transactions with better performance and strong consistency semantics by building on a replication protocol with *no* consistency. We present inconsistent replication, a new replication protocol that provides fault tolerance without consistency, and TAPIR, a new distributed transaction protocol that provides linearizable transactions using IR. We combined IR and TAPIR in TAPIR-KV, a distributed transactional key-value storage system. Our experiments demonstrate that TAPIR-KV lowers commit latency by 50% and increases throughput by 3× relative to conventional transactional storage systems. In many cases, it matches the performance of weakly-consistent systems while providing much stronger guarantees.

Acknowledgements

We thank the SOSP anonymous reviewers and our shepherd Miguel Castro for their helpful feedback. We thank Jialin Li, Neha Narula, and Xi Wang for early feedback on the paper. This work was supported by the National Science Foundation under grants CNS-0963754, CNS-1217597, CNS-1318396, CNS-1420703, and CNS-1518702, by NSF GRFP and IBM Ph.D. fellowships, and by Google. We also appreciate the support of our local zoo tapirs, Ulan and Bintang.

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] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. In *Proc. of VLDB*, 2014.

[4] 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.

[5] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proc. of SOSP*, 2013.

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

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

[8] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI*, 2006.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 2008.

[10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proc. of SOSP*, 2013.

[11] 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.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of SOCC*, 2010.

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

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

[15] 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.

[16] A. Dey, A. Fekete, R. Nambiar, and U. Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *Proc. of ICDEW*, 2014.

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

[18] M. J. Fischer, N. A. Lynch, and M. S. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

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

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

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. of USENIX ATC*, 2010.

[22] 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.

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

[24] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 1981.

[25] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 1992.

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

[27] L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 1994.

[28] L. Lamport. Paxos made simple. *ACM SIGACT News*, 2001.

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

[30] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, Oct. 2006.

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

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

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

[34] 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 *Proc. of SOSP*, 2011.

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

[36] MongoDB: A open-source document database, 2013. http://www.mongodb.org/.

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

[38] 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.

[39] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proc. of NSDI*, 2015.

[40] Redis: Open source data structure server, 2013. http://redis.io/.

[41] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 2005.

[42] S. Sanfilippo. WAIT: synchronous replication for Redis. http://antirez.com/news/66, Dec. 2013.

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

[44] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of SOSP*, 1995.

[45] 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.

[46] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of OSDI*, 2004.

[47] 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.

A. IR TLA+ Specification

MODULE *IR_consensus*

This is a TLA+ specification of the *Inconsistent* Replication algorithm. (And a mechanically-checked proof of its correctness using *TLAPS*)

EXTENDS *FiniteSets*, *Naturals*, *TLC*, *TLAPS*

Constants

Constant parameters: Replicas: the set of all replicas (Replica *IDs*)
Clients: the set of all clients (Client *IDs*)
Quorums: the set of all quorums *SuperQuorums*: the set of all super quorums Results: the set of all possible result types *OperationBody*: the set of all possible operation bodies (with arguments, etc. - can be infinite)
S: shard *id* of the shard *Replicas* constitute
f: maximum number of failures allowed (half of n)

Constants used to bound variables, for model checking (*Nat* is bounded) *max_vc*: maximum number of View-Changes allowed for each replicas *max_req*: maximum number of *op* requests performed by clients

CONSTANTS *Replicas*, *Clients*, *Quorums*, *SuperQuorums*, *Results*, *OpBody*,
AppClientFail, *AppReplicaFail*,
SuccessfulInconsistentOp(_), *SuccessfulConsensusOp*(_, _),
Merge(_, _),
Sync(_),
ExecInconsistent(_),
ExecConsensus(_),
Decide(_),
f,
S, *Shards*, $S =$ shard *id*
max_vc, *max_req*

ASSUME *IsFiniteSet*(*Replicas*)

ASSUME $QuorumAssumption \triangleq$
$\land\ Quorums \subseteq \text{SUBSET}\ Replicas$
$\land\ SuperQuorums \subseteq \text{SUBSET}\ Replicas$
$\land\ \forall\, Q1,\ Q2 \in Quorums : Q1 \cap Q2 \neq \{\}$
$\land\ \forall\, Q \in Quorums,\ R1,\ R2 \in SuperQuorums :$
$Q \cap R1 \cap R2 \neq \{\}$

ASSUME $FailuresAssumption \triangleq$
$\forall\, Q \in Quorums : Cardinality(Q) > f$

The possible states of a replica and the two types of operations currently defined by *IR*.

$ReplicaState \triangleq \{\text{“NORMAL”}, \text{“FAILED”}, \text{“RECOVERING”}, \text{“VIEW-CHANGING”}\}$
$ClientState \triangleq \{\text{“NORMAL”}, \text{“FAILED”}\}$

$OpType \triangleq \{\text{“Inconsistent”}, \text{“Consensus”}\}$
$OpStatus \triangleq \{\text{“TENTATIVE”}, \text{“FINALIZED”}\}$

Definition of operation space

$MessageId \triangleq [cid : Clients, msgid : Nat]$
$Operations \triangleq [type : OpType, body : OpBody]$

Message is defined to be the set of all possible messages

TODO: Assumptions
Assume unique message ids
Assume no more than f replica failures
We use shart to specify for what shard this message was
(we share the variables)

$$
\begin{array}{ll}
Message \triangleq & \\
 & [type : \{\text{“REQUEST”}\}, \\
 & \ id : MessageId, \\
 & \ op : Operations] \\
\cup & [type : \{\text{“REPLY”}\}, \quad \text{reply no result} \\
 & \ id : MessageId, \\
 & \ v : Nat, \\
 & \ src : Replicas] \\
\cup & \\
 & [type : \{\text{“REPLY”}\}, \quad \text{reply with result} \\
 & \ id : MessageId, \\
 & \ v : Nat, \\
 & \ res : Results, \\
 & \ src : Replicas] \\
 & \ v = \text{ view num.} \\
\cup & \\
 & [type : \{\text{“START-VIEW-CHANGE”}\}, \\
 & \ v : Nat, \\
 & \ src : Replicas] \\
\cup & \\
 & [type : \{\text{“DO-VIEW-CHANGE”}\}, \\
 & \ r : \text{SUBSET } ([msgid : MessageId, \\
 & \qquad\qquad op : Operations, \\
 & \qquad\qquad res : Results] \\
 & \qquad \cup [msgid : MessageId, \\
 & \qquad\quad op : Operations]), \\
 & \ v : Nat, \\
 & \ src : Replicas, \\
 & \ dst : Replicas] \\
\cup &
\end{array}
$$

$[type : \{\text{“START-VIEW”}\},$
$v : Nat,$
$src : Replicas]$
$\cup$
$[type : \{\text{“START-VIEW-REPLY”}\},$
$v : Nat,$
$src : Replicas,$
$dst : Replicas]$
$\cup$
$[type : \{\text{“FINALIZE”}\},$ finalize with no result
$id : MessageId,$
$op : Operations,$
$res : Results]$
$\cup$
$[type : \{\text{“FINALIZE”}\},$ finalize with result
$id : MessageId,$
$op : Operations,$
$res : Results]$
$\cup$ $[type : \{\text{“CONFIRM”}\},$
$v : Nat,$
$id : MessageId,$
$op : Operations,$
$res : Results,$
$src : Replicas]$

Variables and State Predicates

Variables: 1. State at each replica:

rState = Denotes current replica state. Either:
- *NORMAL* (processing operations)
- VIEW-CHANGING (participating in recovery)

rRecord = Unordered set of operations and their results *rViewNumber* = current view number

2. State of communication medium: *sentMsg* = sent (but not yet received) messages
3. State at client: *cCurrentOperation* = *crt* operation requested by the client *cMmessageCounter* = the message I must use for the next operation

VARIABLES *rState*, *rRecord*, *rViewNumber*, *rViewReplies*, *sentMsg*, *cCrtOp*, *cCrtOpToFinalize*, *cMsgCounter*, *cCrtOpReplies*, *cCrtOpConfirms*, *cState*, *aSuccessful*, *gViewChangesNo*

Defining these tuples makes it easier to express which varibles remain unchanged

$rVars \triangleq \langle rState, rRecord, rViewNumber, rViewReplies\rangle$ Replica variables.

$cVars \triangleq \langle cCrtOp,$ current operation at a client
$cCrtOpToFinalize,$
$cCrtOpReplies,$ current operation replies
$cCrtOpConfirms,$
$cMsgCounter,$
$cState\rangle$ Client variables.

$aVars \triangleq \langle aSuccessful\rangle$ Application variables

$oVars \triangleq \langle sentMsg,\ gViewChangesNo\rangle$ Other variables.

$vars \triangleq \langle rVars,\ cVars,\ oVars\rangle$ All variables.

$$
\begin{array}{l}
TypeOK \triangleq \\
\quad \land\ rState[S] \in [Replicas \to ReplicaState] \\
\quad \land\ rRecord[S] \in [Replicas \to \ \text{SUBSET}\ ([msgid : MessageId, \\
\qquad\qquad\qquad\qquad\qquad\qquad op : Operations, \\
\qquad\qquad\qquad\qquad\qquad\qquad res : Results, \\
\qquad\qquad\qquad\qquad\qquad\qquad status : OpStatus] \\
\qquad\qquad\qquad\qquad\qquad \cup\ [msgid : MessageId, \\
\qquad\qquad\qquad\qquad\qquad\quad op : Operations, \\
\qquad\qquad\qquad\qquad\qquad\quad status : OpStatus])] \\
\quad \land\ rViewNumber[S] \in [Replicas \to Nat] \\
\quad \land\ rViewReplies[S] \in [Replicas \to \text{SUBSET}\ [type : \{\text{“do-view-change”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{“start-view-reply”}\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad viewNumber : Nat, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad r : \text{SUBSET}\ ([msgid : MessageId, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad op : Operations, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad res : Results, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad status : OpStatus] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cup\quad [msgid : MessageId, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad op : Operations, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad status : OpStatus]), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad src : Replicas]] \\
\quad \land\ sentMsg[S] \in \text{SUBSET}\ Message \\
\quad \land\ cCrtOp[S] \in [Clients \to Operations \cup \{\langle\rangle\}] \\
\quad \land\ cCrtOpToFinalize \in [Clients \to Operations \cup \{\langle\rangle\}] \\
\quad \land\ cCrtOpReplies[S]\ \in [Clients \to \text{SUBSET}\ ([viewNumber : Nat, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad res : Results, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad src : Replicas] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cup\quad [viewNumber : Nat, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad src : Replicas])] \\
\quad \land\ cCrtOpConfirms[S] \in [Clients \to \text{SUBSET}\ [viewNumber : Nat, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad res : Results, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad src : Replicas]] \\
\quad \land\ cMsgCounter[S] \in [Clients \to Nat] \\
\quad \land\ cState \in [Clients \to ClientState] \\
\quad \land\ aSuccessful \in \text{SUBSET}\ ([mid : MessageId,
\end{array}
$$

$op : Operations,$
$res : Results]$
$\cup\ [mid \quad : MessageId,$
$op : Operations])$
$\land\ gViewChangesNo[S] \in Nat$

$Init \triangleq$
$\land\ rState = [r \in Replicas \mapsto \text{“NORMAL”}]$
$\land\ rRecord = [r \in Replicas \mapsto \{\}]$
$\land\ rViewNumber = [r \in Replicas \mapsto 0]$
$\land\ rViewReplies = [r \in Replicas \mapsto \{\}]$
$\land\ sentMsg = \{\}$
$\land\ cCrtOp = [c \in Clients \mapsto \langle\rangle]$
$\land\ cCrtOpToFinalize = [c \in Clients \mapsto \langle\rangle]$
$\land\ cCrtOpReplies = [c \in Clients \mapsto \{\}]$
$\land\ cCrtOpConfirms = [c \in Clients \mapsto \{\}]$
$\land\ cMsgCounter = [c \in Clients \mapsto 0]$
$\land\ cState = [c \quad \in Clients \mapsto \text{“NORMAL”}]$
$\land\ aSuccessful \quad = \{\}$
$\land\ gViewChangesNo = 0$

Actions

$Send(m) \triangleq sentMsg' = [sentMsg \text{ EXCEPT } ![S] = @ \cup \{m\}]$

Client Actions

Note: CHOOSE does not introduce nondeterminism (the same value is chosen each time)

Client sends a request

$ClientRequest(c,\ op) \triangleq$
$\land\ cCrtOp[S][c] = \langle\rangle$ the client is not waiting for a result of another operation
$\land\ cCrtOpToFinalize[S][c] = \langle\rangle$
$\land\ cMsgCounter' = [cMsgCounter \text{ EXCEPT } ![S][c] = @ + 1]$
$\land\ cCrtOp' = [cCrtOp \text{ EXCEPT } ![S][c] = op]$
$\land\ Send([type \mapsto \text{“REQUEST”},$
$id \mapsto [cid \mapsto c,\ msgid \mapsto cMsgCounter[S][c] + 1],$
$op \mapsto op])$
$\land$ UNCHANGED $\langle rVars,\ aVars,\ cCrtOpReplies,\ cCrtOpToFinalize,$
$cCrtOpConfirms,\ cState,\ gViewChangesNo\rangle$
$\land\ cMsgCounter[S][c] < max\_req$ BOUND the number of requests a client can make

```
 Client received a reply
ClientReceiveReply(c) ≜
 ∃ msg ∈ sentMsg[S] :
   ∧ msg.type = "REPLY"
   ∧ cCrtOp[S][c] ≠ ⟨⟩
   ∧ msg.id = [cid ↦ c, msgid ↦ cMsgCounter[S][c]]  reply to c's request for crt op
    TODO: if already reply from src, keep the most recent one (biggest view Number)
    ∧ Assert(Cardinality(cCrtOpReplies[c]) < 10, "cCrtOpReplies cardinality bound")
   ∧ ∨ ∧ cCrtOp[S][c].type = "Inconsistent"
       ∧ cCrtOpReplies' = [cCrtOpReplies EXCEPT ![S][c] = @ ∪
                                          {[viewNumber ↦ msg.v,
                                            src        ↦ msg.src]}]
     ∨ ∧ cCrtOp[S][c].type = "Consensus"
       ∧ cCrtOpReplies' = [cCrtOpReplies EXCEPT ![S][c] = @ ∪
                                          {[viewNumber ↦ msg.v,
                                            res        ↦ msg.res,
                                            src        ↦ msg.src]}]
   ∧ UNCHANGED ⟨cCrtOp, cCrtOpToFinalize, cCrtOpConfirms,
                cMsgCounter, cState, rVars, aVars, oVars⟩


 "Helper" formulas
__matchingViewNumbers(Q, c) ≜
     a (super)quorum of replies with matching view numbers
   ∧ ∀ r ∈ Q :
       ∧ ∃ reply ∈ cCrtOpReplies[S][c]: reply.src = r
     ∧ ∀ p ∈ Q : ∃ rr, pr ∈ cCrtOpReplies[S][c] :
                    ∧ rr.src = r
                    ∧ pr.src = p
                    ∧ rr.viewNumber = pr.viewNumber

__matchingViewNumbersAndResults(Q, c) ≜
     a (super)quorum of replies with matching view numbers
     and results
   ∧ ∀ r ∈ Q :
       ∧ ∃ reply ∈ cCrtOpReplies[S][c]: reply.src = r
     ∧ ∀ p ∈ Q : ∃ rr, pr ∈ cCrtOpReplies[S][c] :
                    ∧ rr.src = r
                    ∧ pr.src = p
                    ∧ rr.viewNumber = pr.viewNumber
                    ∧ rr.res = pr.res

 IR Client received enough responses to decide
 what to do with the operation
ClientDecideOp(c) ≜
  ∧ cCrtOp[S][c] ≠ ⟨⟩
```

```
I. The IR Client got a simple quorum of replies
∧ ∨ ∃ Q ∈ Quorums :
    ∧ ∀ r ∈ Q :
        ∃ reply ∈ cCrtOpReplies[S][c] : reply.src = r

    ∧ ∨ ∧ cCrtOp[S][c].type = "Inconsistent"
      ∧ __matchingViewNumbers(Q, c)
      ∧ aSuccessful' = aSuccessful ∪
                       {[mid ↦ [cid ↦ c,
                                msgid ↦ cMsgCounter[S][c]],
                         op ↦ cCrtOp[S][c]]}
      ∧ SuccessfulInconsistentOp(cCrtOp[S][c])
      ∧ Send([type ↦ "FINALIZE",
              id ↦ [cid ↦ c, msgid ↦ cMsgCounter[S][c]],
              op ↦ cCrtOp[S][c]])
      ∧ UNCHANGED ⟨cCrtOpToFinalize⟩

      ∨ ∧ cCrtOp[S][c].type = "Consensus"
        ∧ LET res ≜ IF __matchingViewNumbersAndResults(Q, c)
                    THEN
                        CHOOSE result ∈
                            {res ∈ Results :
                                ∃ reply ∈ cCrtOpReplies[S][c] :
                                    ∧ reply.src ∈ Q
                                    ∧ reply.res = res} : TRUE
                    ELSE
                        Decide(cCrtOpReplies[S][c])
          IN
            ∧ Send([type ↦ "FINALIZE",
                    id ↦ [cid ↦ c, msgid ↦ cMsgCounter[S][c]],
                    op ↦ cCrtOp[S][c],
                    res ↦ res])
        ∧ cCrtOpToFinalize' = [cCrtOp EXCEPT ![S][c] = cCrtOp[S][c]]
        ∧ UNCHANGED ⟨aSuccessful⟩

II. The IR Client got super quorum of responses
∨ ∃ SQ ∈ SuperQuorums :
    ∧ ∀ r ∈ SQ :
        ∃ reply ∈ cCrtOpReplies[S][c] : reply.src = r
    ∧ cCrtOp[S][c].type = "Consensus"  only care if consensus op
    ∧ __matchingViewNumbersAndResults(SQ, c)
    ∧ LET res ≜ CHOOSE result ∈
                    {res ∈ Results :
                        ∃ reply ∈ cCrtOpReplies[S][c] :
                            ∧ reply.src ∈ SQ
                            ∧ reply.res = res} : TRUE
```

```
        IN
          ∧ Send([type ↦ "FINALIZE",
                  id   ↦ [cid ↦ c, msgid ↦ cMsgCounter[S][c]],
                  op   ↦ cCrtOp[S][c],
                  res  ↦ res])
          ∧ aSuccessful' = aSuccessful ∪
                           {[mid ↦ [cid ↦ c,
                                    msgid ↦ cMsgCounter[S][c]],
                             op  ↦ cCrtOp[S][c],
                             res ↦ res]}
          ∧ SuccessfulConsensusOp(cCrtOp[S][c], res)
      ∧ UNCHANGED ⟨cCrtOpToFinalize⟩
  ∧ cCrtOp' = [cCrtOp EXCEPT ![S][c] = ⟨⟩]
  ∧ cCrtOpReplies' = [cCrtOpReplies EXCEPT ![S][c] = {}]
  ∧ UNCHANGED ⟨cMsgCounter, cState, cCrtOpConfirms, rVars, gViewChangesNo⟩

 Client received a confirm
ClientReceiveConfirm(c) ≜
  ∃ msg ∈ sentMsg[S] :
    ∧ msg.type = "CONFIRM"
    ∧ cCrtOpToFinalize[S][c] ≠ ⟨⟩
    ∧ msg.id = [cid ↦ c, msgid ↦ cMsgCounter[S][c]]  reply to c's request for crt op
    ∧ cCrtOpConfirms' = [cCrtOpConfirms EXCEPT ![S][c] = @ ∪
                                               {[viewNumber ↦ msg.v,
                                                 res        ↦ msg.res,
                                                 src        ↦ msg.src]}]
    ∧ UNCHANGED ⟨cCrtOp, cCrtOpReplies, cCrtOpToFinalize, cMsgCounter,
                 cState, rVars, aVars, oVars⟩


 An operation is finalized by a client and result returned to the application
ClientFinalizedOp(c) ≜
  ∧ cCrtOpToFinalize[S][c] ≠ ⟨⟩
  ∧ ∃ Q ∈ Quorums :
      IR client received a quorum of responses
    ∧ ∀ r ∈ Q :
        ∃ reply ∈ cCrtOpConfirms[S][c] : reply.src = r
    ∧ LET
          take the result in the biggest view number
        reply ≜ CHOOSE reply ∈ cCrtOpConfirms[S][c] :
                          ¬∃ rep ∈ cCrtOpConfirms[S][c] :
                               rep.viewNumber > reply.viewNumber
      IN
        ∧ aSuccessful' = aSuccessful ∪
                         {[mid ↦ [cid ↦ c,
                                  msgid ↦ cMsgCounter[S][c]],
```

```
                                    op ↦ cCrtOpToFinalize[S][c],
                                    res ↦ reply.res]}
          ∧ SuccessfulConsensusOp(cCrtOp[S][c], reply.res)   respond to app
  ∧ cCrtOpToFinalize' = [cCrtOpToFinalize EXCEPT ![S][c] = ⟨⟩]
  ∧ cCrtOpConfirms' = [cCrtOpConfirms EXCEPT ![S][c] = {}]
  ∧ UNCHANGED ⟨rVars, cCrtOp, cCrtOpReplies, cMsgCounter, cState, oVars⟩

 Client fails and looses all data
ClientFail(c) ≜
  ∧ cState' = [cState EXCEPT ![S][c] = "FAILED"]
  ∧ cMsgCounter' = [cMsgCounter EXCEPT ![S][c] = 0]
  ∧ cCrtOp' = [cCrtOp EXCEPT ![S][c] = ⟨⟩]
  ∧ cCrtOpReplies' = [cCrtOpReplies EXCEPT ![S][c] = {}]
  ∧ AppClientFail
  ∧ UNCHANGED ⟨rVars, aVars, oVars⟩

 Client recovers
ClientRecover(c) ≜ FALSE
```

Replica Actions

```
 Replica sends a reply
ReplicaReceiveRequest(r) ≜
  ∃ msg ∈ sentMsg[S] :
    ∧ msg.type = "REQUEST"
    ∧ ¬∃ rec    ∈ rRecord[S][r] : rec.msgid = msg.id
        not alredy replied for this op
    ∧ ∨ ∧ msg.op.type = "Inconsistent"
        ∧ Send([type ↦ "REPLY",
                id   ↦ msg.id,
                v    ↦ rViewNumber[S][r],
                src  ↦ r])
        ∧ rRecord' = [rRecord EXCEPT ![S][r] = @ ∪ {[msgid ↦ msg.id,
                                                     op ↦ msg.op,
                                                     status ↦ "TENTATIVE"]}]

      ∨ ∧ msg.op.type = "Consensus"
        ∧ LET res ≜ ExecConsensus(msg.op)
          IN
            ∧ Send([type ↦ "REPLY",
                    id   ↦ msg.id,
                    v    ↦ rViewNumber[S][r],
                    res  ↦ res,
                    src  ↦ r])
            ∧ rRecord' = [rRecord EXCEPT ![S][r] = @ ∪ {[msgid ↦ msg.id,
```

$op \mapsto msg.op,$
$res \mapsto res,$
$status \mapsto$ "TENTATIVE"]}]
$\land$ UNCHANGED $\langle rState,\ rViewNumber,\ rViewReplies,\ cVars,\ aVars,\ gViewChangesNo\rangle$

Replica receive a message from an *IR* Client to finalize an *op*
For inconsistent oprations the replica sends $< CONFIRM >$ and executes the operation.

TODO: Write this more compact

$ReplicaReceiveFinalize(r) \triangleq$
$\exists\, msg \in sentMsg[S]:$
$\land\ msg.type =$ "FINALIZE"
$\land\ \lor\ \land\ msg.op.type =$ "Inconsistent"
$\land\ Send([type \mapsto$ "CONFIRM",
$v \mapsto rViewNumber[S][r],$
$id \mapsto msg.id,$
$op \mapsto msg.op,$
$src \mapsto r])$
$\land\ \lor \exists\, rec \in rRecord[S][r]:$
$\land\ rec.msgid = msg.id$
$\land\ rec.op = msg.op$ Replica knows of this *op*
$\land$ IF $rec.status \neq$ "FINALIZED"
THEN $ExecInconsistent(msg.op)$
ELSE TRUE
$\land\ rRecord' = [rRecord$ EXCEPT $![S][r] = (@ \setminus \{rec\}) \cup$
$\{[msgid \mapsto msg.id,$
$op \mapsto msg.op,$
$status \mapsto$ "FINALIZED"]}]
$\lor\ \land \neg\exists\, rec \in rRecord[S][r]:$ Replica didn't hear of this *op*
$\land\ rec.msgid = msg.id$
$\land\ rec.op\ = msg.op$
$\land\ rRecord' = [rRecord$ EXCEPT $![S][r] = @ \cup$
$\{[msgid \mapsto msg.id,$
$op \mapsto msg.op,$
$status \mapsto$ "FINALIZED"]}]
$\land\ ExecInconsistent(msg.op)$
$\lor\ \land\ msg.op.type =$ "Consensus"
$\land\ \lor\ \land \exists\, rec \in rRecord[S][r]:$
$\land\ rec.msgid = msg.id$
$\land\ rec.op = msg.op$ Replica knows of this *op*
$\land\ \lor\ \land\ rec.status =$ "TENTATIVE" Operation tentative
$\land\ rRecord' = [rRecord$ EXCEPT $![S][r] = (@ \setminus \{rec\}) \cup$
$\{[msgid \mapsto msg.id,$
$op \mapsto msg.op,$

```
                                                    res ↦ msg.res,
                                                    status ↦ "FINALIZED"]}]
                    ∧ Send([type ↦ "CONFIRM",
                            v    ↦ rViewNumber[S][r],
                            id   ↦ msg.id,
                            op   ↦ msg.op,
                            res  ↦ msg.res,
                            src  ↦ r])
                      ∧ IF  rec.res ≠ msg.res
                        THEN UpdateConsensus(msg.op, msg.res)
                        ELSE TRUE
                  ∨ ∧ rec.status = "FINALIZED"  Operation already finalized (view change happened in the meantime)
                    ∧ Send([type ↦ "CONFIRM",
                            v    ↦ rViewNumber[S][r],
                            id   ↦ msg.id,
                            op   ↦ msg.op,
                            res  ↦ rec.res,
                            src  ↦ r])
                    ∧ UNCHANGED ⟨rRecord⟩
              ∨ ∧ ¬∃ rec ∈ rRecord[S][r] :  Replica didn't hear of this op
                ∧ rec.msgid = msg.id
                ∧ rec.op  = msg.op
                ∧ rRecord' = [rRecord EXCEPT ![S][r] = @ ∪
                                                    {[msgid ↦ msg.id,
                                                      op ↦ msg.op,
                                                      res ↦ msg.res,
                                                      status ↦ "FINALIZED"]}]
                ∧ Send([type ↦ "CONFIRM",
                        v    ↦ rViewNumber[S][r],
                        id   ↦ msg.id,
                        op   ↦ msg.op,
                        res  ↦ msg.res,
                        src  ↦ r])
                  ∧ ExecuteAndUpdateConsensus(msg.op, msg.res)
    ∧ UNCHANGED ⟨rState, rViewNumber, rViewReplies, cVars, aVars, gViewChangesNo⟩

A replica starts the view change procedure
supports concurrent view changes (id by src)
ReplicaStartViewChange(r) ≜
  ∧ Send([type ↦ "START-VIEW-CHANGE",
          v    ↦ rViewNumber[r],
          src  ↦ r])
  ∧ rState' = [rState EXCEPT ![r] = "RECOVERING"]
  ∧ UNCHANGED ⟨rViewNumber, rViewReplies, rRecord, cVars, aVars⟩
  ∧ gViewChangesNo < max_vc  BOUND on number of view changes
```

$\land\ gViewChangesNo' = gViewChangesNo + 1$

A replica received a message to start view change

$ReplicaReceiveStartViewChange(r) \triangleq$

$\land \exists\, msg \in sentMsg[S] :$
$\quad \land\ msg.type =$ "START-VIEW-CHANGE"
$\quad \land$ LET $v\_new \triangleq$
$\qquad$ IF $msg.v > rViewNumber[r]$ THEN $msg.v$
$\qquad$ ELSE $rViewNumber[S][r]$
$\quad$ IN
$\qquad \land \neg\exists\, m \in sentMsg[S] :$ not already sent (just to bound the model checker)
$\qquad\quad \land\ m.type =$ "DO-VIEW-CHANGE"
$\qquad\quad \land\ m.v \geq msg.v$
$\qquad\quad \land\ m.dst = msg.src$
$\qquad\quad \land\ m.src = r$
$\qquad \land\ Send([type \mapsto$ "DO-VIEW-CHANGE",
$\qquad\qquad\qquad v \mapsto v\_new + 1,$
$\qquad\qquad\qquad r \mapsto rRecord[r],$
$\qquad\qquad\qquad src \mapsto r,$
$\qquad\qquad\qquad dst \mapsto msg.src])$
$\qquad \land\ rViewNumber' = [rViewNumber$ EXCEPT $![S][r] = v\_new + 1]$
$\land\ rState' = [rState$ EXCEPT $![S][r] =$ "VIEW-CHANGING"$]$
$\land$ UNCHANGED $\langle cVars, rRecord, rViewReplies, aVars, gViewChangesNo\rangle$

Replica received DO-VIEW-CHANGE message

$ReplicaReceiveDoViewChange(r) \triangleq$

$\land \exists\, msg \in sentMsg[S] :$
$\quad \land\ msg.type =$ "DO-VIEW-CHANGE"
$\quad \land\ msg.dst = r$
$\quad \land\ msg.v > rViewNumber[r]$
$\quad \land\ rViewReplies' = [rViewReplies$ EXCEPT $![r] = @ \cup$
$\qquad\qquad \{[type \mapsto$ "do-view-change",
$\qquad\qquad\ viewNumber \mapsto msg.v,$
$\qquad\qquad\ r \mapsto msg.r,$
$\qquad\qquad\ src \mapsto msg.src]\}]$
$\land$ UNCHANGED $\langle cVars, rViewNumber, rRecord, rState, aVars, oVars\rangle$

A replica received enough view change replies to start processing in the new view

$ReplicaDecideNewView(r) \triangleq$

$\land \exists\, Q \in Quorums :$
$\quad \land \forall\, rep \in Q \quad : \exists\, reply \in rViewReplies[r] :\ \land\ reply.src = rep$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land\ reply.type =$ "do-view-change"
$\quad$ received at least a quorum of replies
$\quad \land$ LET $recoveredConensusOps\_a \triangleq$
$\qquad$ any consensus operation found in at least a majority of a Quorum
$\qquad\quad \{x \in$ UNION $\{y.r : y \in \{z \in rViewReplies[S][r] : z.src \in Q\}\} :$

$\land\ x[2].type = \text{“Consensus”}$
$\land\ \exists\, P \in SuperQuorums :$
$\forall\, rep \in Q \cap P \quad :$
$\exists\, reply \in rViewReplies[r] :$
$\land\ reply.src = rep$
$\land\ x \in reply.r\}$ same op, same result

$recoveredConensusOps\_b \triangleq$ $TODO$: what result? from the app?
the rest of consensus ops found in at least one record (discard the result)
$\{\langle z[1],\, z[2]\rangle :$
$z \in \{x \in \text{UNION}\ \{y.r : y \in \{z \in rViewReplies[S][r] : z.src \in Q\}\} :$
$\land\ x[2].type = \text{“Consensus”}$
$\land\ \neg x \in recoveredConensusOps\_a\}\}$

$recoveredInconsistentOps\_c \triangleq$
any inconsistent operation found in any received record (discard the result)
$\{\langle z[1],\, z[2]\rangle :$
$z \in \{x \in \text{UNION}\ \{y.r : y \in \{z \in rViewReplies[S][r] : z.src \in Q\}\} :$
$x[2].type = \text{“Inconsistent”}\}\}$

IN

$\land\ AppRecoverOpsResults(recoveredConensusOps\_a)$
$\land\ AppRecoverOps(recoveredConensusOps\_b)$
$\land\ AppRecoverOps(recoveredInconsistentOps\_c)$
$\land\ rRecord' = [rRecord\ \text{EXCEPT}\ ![S][r] = @ \cup recoveredConensusOps\_a$
$\cup\ recoveredConensusOps\_b$
$\cup\ recoveredInconsistentOps\_c]$

$\land$ LET $v\_new \triangleq$
max view number received
CHOOSE $v \in \{x.viewNumber : x \in rViewReplies[r]\} :$
$\forall\, y \in rViewReplies[r] :$
$y.viewNumber \leq v$

IN

$\land\ Send([type \mapsto \text{“START-VIEW”},$
$v \quad \mapsto v\_new,$
$src \ \mapsto r])$
$\land\ rViewNumber' = [rViewNumber\ \text{EXCEPT}\ ![r] = v\_new]$
$\land\ rViewReplies' = [rViewReplies\ \text{EXCEPT}\ ![r] = \{\}]$
$\land$ UNCHANGED $\langle rState,\, cVars,\, aVars,\, gViewChangesNo\rangle$

A replica receives a start view message
$ReplicaReceiveStartView(r) \triangleq$
$\land\ \exists\, msg \in sentMsg :$
$\land\ msg.type = \text{“START-VIEW”}$
$\land\ msg.v \geq rViewNumber[r]$
$\land\ msg.src \neq r$ don't reply to myself
$\land\ Send([type \mapsto \text{“START-VIEW-REPLY”},$

$v \mapsto msg.v,$
$src \mapsto r,$
$dst \mapsto msg.src])$
$\land\ rViewNumber' = [rViewNumber \text{ EXCEPT } ![r] = msg.v]$
$\land\ rState' = [rState \text{ EXCEPT } ![r] = \text{“NORMAL”}]$
$\land$ UNCHANGED $\langle rRecord, rViewReplies, cVars, aVars, gViewChangesNo\rangle$

$ReplicaReceiveStartViewReply(r) \triangleq$
$\land \exists\, msg \in sentMsg :$
$\land\ msg.type = \text{“START-VIEW-REPLY”}$
$\land\ msg.dst = r$
$\land\ msg.v > rViewNumber[r]$ receive only if bigger than the last view I was in
$\land\ rViewReplies' = [rViewReplies \text{ EXCEPT } ![S][r] = @ \cup$
$\{[type \mapsto \text{“start-view-reply”},$
$viewNumber \mapsto msg.v,$
$r \mapsto \{\},$
$src \mapsto msg.src]\}]$
$\land$ UNCHANGED $\langle rRecord, rState, rViewNumber, cVars, aVars, oVars\rangle$

$ReplicaRecover(r) \triangleq$ we received enough START-VIEW-*REPLY* messages
$\exists\, Q \in Quorums :$
$\land\ r \in Q$
$\land \forall\, p \in Q : \lor\ p = r$
$\lor\ \land\ p \neq r$
$\land \exists\, reply \in rViewReplies[S][r] : \land\ reply.src = p$
$\land\ reply.type = \text{“start-view-reply”}$
$\land\ rViewReplies' = [rViewReplies \text{ EXCEPT } ![S][r] = \{\}]$
$\land\ rState' = [rState \text{ EXCEPT } ![r] = \text{“NORMAL”}]$
$\land$ UNCHANGED $\langle rRecord, rViewNumber, cVars, aVars, oVars\rangle$

$ReplicaResumeViewChange(r) \triangleq$ *TODO*: On timeout
FALSE

A replica fails and looses everything
$ReplicaFail(r) \triangleq$ *TODO*: check cardinality
$\land\ rState' = [rState \text{ EXCEPT } ![S][r] = \text{“FAILED”}]$
$\land\ rRecord' = [rRecord \text{ EXCEPT } ![S][r] = \{\}]$
$\land\ rViewNumber' = [rViewNumber \text{ EXCEPT } ![r] = 0]$ * *TODO*: check what happens if we loose the view number
$\land\ rViewReplies' = [rViewReplies \text{ EXCEPT } ![S][r] = \{\}]$
$\land$ UNCHANGED $\langle rViewNumber, cVars, aVars, oVars\rangle$
$\land\ Cardinality(\{re \in Replicas :$
We assume less than f replicas are allowed to fail
$\lor\ rState[S][re] = \text{“FAILED”}$
$\lor\ rState[S][re] = \text{“RECOVERING”}\}) < f$

High-Level Actions

$ClientAction(c) \triangleq$
$\lor \land cState[c] = \text{“NORMAL”}$
$\land \quad \lor ClientRequest(c)$ * some client tries to replicate commit an operation
$\lor ClientReceiveReply(c)$ some client receives a reply from a replica
$\lor ClientReceiveConfirm(c)$ some client receives a confirm from a replica
$\lor ClientFail(c)$ * some client fails
$\lor ClientDecideOp(c)$ an operation is successful at some client
$\lor ClientFinalizedOp(c)$ * an operation was finalized at some client
$\lor \land cState[c] = \text{“FAILED”}$
$\land \lor ClientRecover(c)$

$ReplicaAction(r) \triangleq$
$\lor \land rState[S][r] = \text{“NORMAL”}$
$\land \lor ReplicaReceiveRequest(r)$ some replica sends a reply to a $REQUEST\ msg$
$\lor ReplicaReceiveFinalize(r)$
$\lor ReplicaReceiveStartViewChange(r)$
$\lor ReplicaReceiveStartView(r)$
$\lor ReplicaFail(r)$ * some replica fails
$\lor \land rState[S][r] = \text{“FAILED”}$
$\land \lor ReplicaStartViewChange(r)$ * some replica starts to recover
$\lor \land rState[r] = \text{“RECOVERING”}$ * just to make it clear
$\land \lor ReplicaReceiveDoViewChange(r)$
$\lor ReplicaDecideNewView(r)$
$\lor ReplicaReceiveStartViewReply(r)$
$\lor ReplicaRecover(r)$
$\lor \land rState[S][r] = \text{“VIEW-CHANGING”}$
$\land \lor ReplicaReceiveStartViewChange(r)$
$\lor ReplicaReceiveStartView(r)$
$\lor ReplicaResumeViewChange(r)$ * some timeout expired and view change not finished
$\lor ReplicaFail(r)$

$Next \triangleq$
$\lor \exists\, c \in Clients : ClientAction(c)$
$\lor \exists\, r \in Replicas : ReplicaAction(r)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$FaultTolerance \triangleq$
$\land \forall\, successfulOp \in aSuccessful,\ Q \in Quorums :$
$(\forall\, r \in Q : rState[S][r] = \text{“NORMAL”} \lor rState[S][r] = \text{“VIEW-CHANGING”})$
$\Rightarrow (\exists\, p \in Q : \exists\, rec \in rRecord[S][p] :$
$\land\ successfulOp.msgid = rec.msgid$
$\land\ successfulOp.op = rec.op)$ Not necessarily same result
$\land \forall\, finalizedOp \in aSuccessful,\ Q \in Quorums :$
$(\forall\, r \in Q : rState[r] = \text{“NORMAL”} \lor rState[r] = \text{“VIEW-CHANGING”})$
$\Rightarrow (\exists\, P \in SuperQuorums :$

$$\begin{array}{l}
\quad \forall\, p \in Q \cap P : \\
\quad\quad \exists\, rec \in rRecord[p] : \\
\quad\quad\quad finalizedOp = rec) \\
\\
Inv \triangleq TypeOK \wedge FaultTolerance
\end{array}$$

B. TAPIR TLA+ Specification

MODULE *TAPIR*

This is a TLA+ specification of the *TAPIR* algorithm.

EXTENDS *FiniteSets*, *Naturals*, *TLC*, *TLAPS*

$Max(S) \triangleq$ IF $S = \{\}$ THEN 0 ELSE CHOOSE $i \in S : \forall j \in S : j \leq i$

TAPIR constants:

1. *Shards*: function from shard id to set of replica ids in the shard
2. Transactions: set of all possible transactions
3. *nr_shards*: number of *shards*

CONSTANTS *Shards*, *Transactions*, *NrShards*

Note: assume unique number ids for replicas

IR constants & variables (description in the *IR* module)

CONSTANTS *Clients*, *Quorums*, *SuperQuorums*, *max_vc*, *max_req*, *f*

VARIABLES *rState*, *rRecord*, *rViewNumber*, *rViewReplies*, *sentMsg*, *cCrtOp*, *cCrtOpToFinalize*, *cMsgCounter*, *cCrtOpReplies*, *cCrtOpConfirms*, *cState*, *aSuccessful*, *gViewChangesNo*

$irReplicaVars \triangleq \langle rState, rRecord, rViewNumber, rViewReplies \rangle$
$irClientVars \triangleq \langle cCrtOp,$ current operation at a client
cCrtOpReplies, current operation replies
cMsgCounter,
cState,
cCrtOpToFinalize,
cCrtOpConfirms$\rangle$ Client variables.
$irAppVars \triangleq \langle aSuccessful \rangle$ Application variables
$irOtherVars \triangleq \langle sentMsg, gViewChangesNo \rangle$ Other variables.

TAPIR Variables/State: 1. State at each replica:
rPrepareTxns = List of txns this replica is prepared to commit
rTxnsLog = *Log* of committed and aborted txns in ts order *rStore* = *Versioned* store
rBkpTable = Table of txns for which this replica is the bkp coordinator

2. State of communication medium: *sentMsg* = sent (and duplicate) messages
3. State at client: *cCrtTxn* = *crt txn* requested by the client

TAPIR variables & data structures

VARIABLES *rPreparedTxns*, *rStore*, *rTxnsLogAborted*, *rTxnsLogCommited*, *rClock*, *cCrtTxn*, *cClock*

$tapirReplicaVars \triangleq \langle rPreparedTxns, rStore, rTxnsLogAborted, rTxnsLogCommited, rClock \rangle$
$tapirClientVars \triangleq \langle cCrtTxn, cClock \rangle$

$StoreEntry \triangleq [vs : Nat,\ val : Nat]$ $vs =$ version
$Store \triangleq [key : Nat,$
$\quad entries : \text{SUBSET } StoreEntry,$
$\quad latestVs : Nat,$
$\quad latestVal : Nat]$

$TransactionTs \triangleq [cid : Clients,\ clock : Nat]$ Timestamp
$ReadSet \triangleq [key : Nat,\ val : Nat,\ vs : Nat]$
$WriteSet \triangleq [key : Nat,\ val : Nat]$
$Transaction \triangleq [rSet : \text{SUBSET } ReadSet,$
$\quad wSet : \text{SUBSET } WriteSet,$
$\quad shards : \text{SUBSET } Nat]$

$TypeOK \triangleq$
$\quad \land \quad rStore \in [\text{UNION } \{Shards[i] : i \in 1 \,..\, NrShards\} \to \text{SUBSET } Store]$
$\quad \land \quad rPreparedTxns \in [\text{UNION } \{Shards[i] : i \in 1 \,..\, NrShards\} \to \text{SUBSET } Transaction]$
$\quad \land \quad rTxnsLogAborted \in [\text{UNION } \{Shards[i] : i \in 1 \,..\, NrShards\} \to \text{SUBSET } Transaction]$
$\quad \land \quad rTxnsLogCommited \in [\text{UNION } \{Shards[i] : i \in 1 \,..\, NrShards\} \to \text{SUBSET } Transaction]$

$TAPIRResults \triangleq \{\text{“Prepare-OK”},\ \text{“Retry”},\ \text{“Prepare-Abstain”},\ \text{“Abort”}\}$
$TAPIROpType \triangleq \{\text{“Prepare”},\ \text{“ABORT”},\ \text{“COMMIT”}\}$
$TAPIROpBody \triangleq [opType : TAPIROpType,\ txn : Transaction]$

$TAPIRClientFail \triangleq \text{TRUE}$ state we lose at the app level
$TAPIRReplicaFail \triangleq \text{TRUE}$ state we lose at the app level

TAPIR implementation of *IR* interface
$TAPIRExecInconsistent(op) \triangleq \text{TRUE}$
$TAPIRExecConsensus(op) \triangleq \text{IF } op.type = \text{“Consensus”} \text{ THEN “Prepare-OK” ELSE “Abort”}$
$TAPIRDecide(results) \triangleq \text{TRUE}$
$TAPIRMerge(d,\ u) \triangleq \text{TRUE}$
$TAPIRSync(records) \triangleq \text{TRUE}$
$TAPIRSuccessfulInconsistentOp(op) \triangleq \text{TRUE}$
$TAPIRSuccessfulConsensusOp(op,\ res) \triangleq \text{TRUE}$

Initialize for all *shards*
$InitIR \triangleq$
$\quad \land\ rState = [s \in 1 \,..\, NrShards \mapsto [r \in Shards[s] \mapsto \text{“NORMAL”}]]$
$\quad \land\ rRecord = [s \in 1 \,..\, NrShards \mapsto [r \in Shards[s] \mapsto \{\}]]$
$\quad \land\ rViewNumber = [s \in 1 \,..\, NrShards \mapsto [r \in Shards[s] \mapsto 0]]$
$\quad \land\ rViewReplies = [s \in 1 \,..\, NrShards \mapsto [r \in Shards[s] \mapsto \{\}]]$
$\quad \land\ sentMsg = [s \in 1 \,..\, NrShards \mapsto \{\}]$
$\quad \land\ cCrtOp = [s \in 1 \,..\, NrShards \mapsto [c \in Clients \mapsto \langle\rangle]]$

$\land\ cCrtOpToFinalize = [s \in 1\,..\,NrShards \mapsto [c \in Clients \mapsto \langle\rangle]]$
$\land\ cMsgCounter = [s \in 1\,..\,NrShards \mapsto [c \in Clients \mapsto 0]]$
$\land\ cCrtOpReplies = [s \in 1\,..\,NrShards \mapsto [c \in Clients \mapsto \{\}]]$
$\land\ cCrtOpConfirms = [s \in 1\,..\,NrShards \mapsto [c \in Clients \mapsto \{\}]]$
$\land\ cState = [c \in Clients \mapsto \text{“NORMAL”}]$
$\land\ aSuccessful = \{\}$
$\land\ gViewChangesNo = [s \in 1\,..\,NrShards \mapsto 0]$

IR instance per shard *TODO*: modify replica also

$IR(s) \triangleq$ INSTANCE *IR_consensus* WITH *AppClientFail* $\leftarrow$ *TAPIRClientFail*,
AppReplicaFail $\leftarrow$ *TAPIRReplicaFail*,
OpBody $\leftarrow$ *TAPIROpBody*,
ExecInconsistent $\leftarrow$ *TAPIRExecInconsistent*,
ExecConsensus $\leftarrow$ *TAPIRExecConsensus*,
Merge $\leftarrow$ *TAPIRMerge*,
Sync $\leftarrow$ *TAPIRSync*,
SuccessfulInconsistentOp $\leftarrow$ *TAPIRSuccessfulInconsistentOp*,
SuccessfulConsensusOp $\leftarrow$ *TAPIRSuccessfulConsensusOp*,
Decide $\leftarrow$ *TAPIRDecide*,
Results $\leftarrow$ *TAPIRResults*,
Replicas $\leftarrow$ *Shards*[*s*],
Quorums $\leftarrow$ *Quorums*[*s*],
SuperQuorums $\leftarrow$ *SuperQuorums*[*s*],
S $\leftarrow$ *s*

TAPIR messages

$Message \triangleq$
[*type* : {“READ”},
key : *Nat*,
dst : UNION *Shards*]
$\cup$
[*type* : {“READ-REPLY”},
key : *Nat*,
val : *Nat*,
vs : *Nat*, version
dst : *Clients*]
$\cup$
[*type* : {“READ-VERSION”},
key : *Nat*,
vs : *Nat*,
dst : UNION *Shards*]
$\cup$
[*type* : {“READ-VERSION-REPLY”},
key : *Nat*,
vs : *Nat*,

$dst : Clients]$

$$
\begin{aligned}
InitTAPIR \triangleq\ & \land\ cCrtTxn = [c \in Clients \mapsto \langle\rangle] \\
& \land\ cClock = [c \in Clients \mapsto 0] \\
& \land\ rPreparedTxns = \ [s \in 1\,..\,NrShards \mapsto [r \in Shards[s] \mapsto \{\}]] \\
& \land\ rStore = [r \in \text{UNION}\ \{Shards[i] : i \in 1\,..\,NrShards\} \mapsto \{\}] \\
& \land\ rTxnsLogAborted = [s \in 1\,..\,NrShards \mapsto [r \in Shards[s] \mapsto \{\}]] \\
& \\
& \land\ rClock = [s \in 1\,..\,NrShards \mapsto [r \in Shards[s] \mapsto 0]]
\end{aligned}
$$

$$Init \triangleq InitIR \land InitTAPIR$$

Tapir replica actions

$$TAPIRReplicaReceiveRead(r) \triangleq \text{TRUE}$$

$$
\begin{aligned}
& TAPIRReplicaAction(r) \triangleq \\
& \quad \lor\ \land\ rState[r] = \text{“NORMAL”} \\
& \qquad \land\ \lor\ TAPIRReplicaReceiveRead(r)
\end{aligned}
$$

Tapir client actions

$$
\begin{aligned}
& TAPIRClientExecuteTxn(c) \triangleq \\
& \quad \text{first, resolve all reads (read from any replica and get the } vs\text{)} \\
& \quad \text{then send prepares in all shard involved by seting the } cCrtOp \text{ in the} \\
& \quad \text{respective } IR \text{ shard instance} \\
& \quad TODO\text{: for now just simulate this, pick a transaction from} \\
& \qquad \text{transaction pool, get some versions from the replica} \\
& \qquad \text{stores} \\
& \land\ cCrtTxn[c] = \langle\rangle \\
& \land\ \exists\, t \in Transactions : \\
& \quad \text{LET}\ rSet \triangleq \{rse \in ReadSet : \\
& \qquad \land\ \exists\, trse \in t.rSet : rse = trse \\
& \qquad \land\ \text{LET} \\
& \qquad\quad r \triangleq Max(\{r \in Shards[(rse.key\%NrShards) + 1] : \\
& \qquad\qquad \exists\, se \in rStore[r] : rse.key = se.key\}) \\
& \qquad \text{IN} \\
& \qquad\quad \land\ r \neq 0 \\
& \qquad\quad \land\ \exists\, se \in rStore[r] : \\
& \qquad\qquad \land\ rse.key = se.key \\
& \qquad\qquad \land\ rse.val\ = se.latestVal \\
& \qquad\qquad \land\ rse.vs = se.latestVs \\
& \quad \} \\
& \quad shards \triangleq \{s \in 1\,..\,NrShards :
\end{aligned}
$$

$$\lor \exists\, trse \in t.rSet : s = (trse.key\%NrShards) + 1$$
$$\lor \exists\, twse \in t.wSet : s = (twse.key\%NrShards) + 1\}$$

IN

$\land\ Cardinality(rSet) = Cardinality(t.rSet)$ found all the reads

$\land\ cCrtTxn' = [cCrtTxn \text{ EXCEPT } ![c] = [rSet \mapsto rSet,$
$wSet \mapsto t.wSet,$
$shards \mapsto shards]]$

$\land$ UNCHANGED $\langle irReplicaVars,\ irClientVars,\ irOtherVars,\ irAppVars,$
$tapirReplicaVars,\ cClock\rangle$

$TAPIRClientPrepareTxn(c) \triangleq$

$\land\ cCrtTxn[c] \neq \langle\rangle$

$\land \exists\, s \in cCrtTxn[c].shards :$ prepare in shard s
- ok if already prepared

$\land\ IR(s)!ClientRequest(c, [type \mapsto$ "Consensus",
$body \mapsto \langle$ "Prepare", $cCrtTxn\rangle])$

$\land$ UNCHANGED $\langle irReplicaVars,\ irAppVars,$
$cCrtOpReplies,$
$cCrtOpConfirms,$
$cCrtOpToFinalize,$
$gViewChangesNo,$
$cState,\ tapirClientVars,\ tapirReplicaVars\rangle$

$TAPIRClientAction(c) \triangleq$

$\lor\ \land\ cState[c] =$ "NORMAL"

$\land\ \lor\ TAPIRClientExecuteTxn(c)$ for now just simulate this
(don't send explicit $READ$ messages)

$\lor\ TAPIRClientPrepareTxn(c)$

$\lor\ 2PC(c)$

High-Level Actions

$Next \triangleq$

$\land\ \lor \exists\, c \in Clients : TAPIRClientAction(c)$

$\lor\ \land \exists\, s \in 1\,..\,NrShards : IR(s)!Next$

$\land$ UNCHANGED $\langle tapirClientVars,\ tapirReplicaVars\rangle$

$Inv \triangleq\ Cardinality(aSuccessful) < 2$
