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.

---

\(^1\) 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 3x 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. 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. 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...
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.
- **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 3: Summary of IR interfaces and client/replica state.](image)

- **client id** - unique identifier for the client
- **operation counter** - # of sent operations
- **state** - current replica state; either NORMAL or VIEW-CHANGING
- **record** - unordered set of operations and consensus results

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 writes during 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 \( \langle \text{PROPOSE}, \ id, \ op \rangle \) 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 \( \langle \text{REPLY}, \ id \rangle \).
3. Once the client receives \( f+1 \) responses from replicas (retrying if necessary), it returns to the application protocol and asynchronously sends \( \langle \text{FINALIZE}, \ id \rangle \) to all replicas. \( \langle \text{FINALIZE} \rangle \) can also be piggy-backed on the client’s next message.
4. On \( \text{FINALIZE} \), replicas upcall into the application protocol with \( \text{ExecInconsistent}(op) \) 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 \( \text{PREPARE} \) 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 \( \left\lceil \frac{2}{3} f \right\rceil + 1 \) 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 \( f \) are down), then it must take the slow path. We describe both below:

1. The client sends \( \langle \text{PROPOSE}, \ id, \ op \rangle \) to all replicas.
2. Each replica calls into the application protocol with \( \text{ExecConsensus}(op) \) and writes \( id, op, \) and \( result \) to its record as TENTATIVE. The replica responds to the client with \( \langle \text{REPLY}, \ id, \ result \rangle \).
3. If the client receives at least \( \left\lceil \frac{2}{3} f \right\rceil + 1 \) matching results (within a timeout), then it takes the fast path: the client returns \( result \) to the application protocol and asynchronously sends \( \langle \text{FINALIZE}, \ id, \ result \rangle \) to all replicas.
4. Otherwise, the client takes the slow path: once it receives \( f+1 \) responses (retrying if necessary), then it sends \( \langle \text{FINALIZE}, \ id, \ result \rangle \) to all replicas, where \( result \) is obtained from executing the decide function.
5. On receiving \( \text{FINALIZE} \), each replica marks the operation as FINALIZED, updating its record if the received \( result \) is different, and sends \( \langle \text{CONFIRM}, \ id \rangle \) 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 \( \left\lceil \frac{2}{3} f \right\rceil + 1 \) 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 \( \text{REPLY} \) and \( \text{CONFIRM} \) 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 \( \text{Sync} \) before returning to \( \text{NORMAL} \) state.
3. On \( \text{PROPOSE} \), each replica first checks whether the operation was already \( \text{FINALIZED} \) by a view change. If so, the replica responds with \( \langle \text{REPLY}, \ id, \ \text{FINALIZED}, \ v, \ \{\text{result}\} \rangle \), where \( v \) is the replica’s current view number and \( result \) is the consensus result for consensus operations.
4. If the client receives \( \text{REPLY} \) with a \( \text{FINALIZED} \) status for consensus operations, it sends \( \langle \text{FINALIZE}, \ id, \ result \rangle \) 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 ⟨DO-VIEW-CHANGE, rec, v, v’⟩ 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 vnew, where vnew 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 ⟨START-VIEW, vnew, R⟩, where R is the master record.

5. When a replica receives a START-VIEW message with vnew 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 vnew, 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 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 ⌈f/2⌉ + 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 ⌈f/2⌉ + 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.

<table>
<thead>
<tr>
<th>Function: IR-MERGE-RECORDS(records)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. R, d, u = ∅</td>
</tr>
<tr>
<td>2. for v ∈ records</td>
</tr>
<tr>
<td>3. if op.type == inconsistent</td>
</tr>
<tr>
<td>4. R = R ∪ op</td>
</tr>
<tr>
<td>5. else if op.type == consensus and op.status == FINALIZED</td>
</tr>
<tr>
<td>6. R = R ∪ op</td>
</tr>
<tr>
<td>7. else if op.type == consensus and op.status == TENTATIVE</td>
</tr>
<tr>
<td>8. if op.result in more than ⌈f/2⌉ + 1 records</td>
</tr>
<tr>
<td>9. d = d ∪ op</td>
</tr>
<tr>
<td>10. else</td>
</tr>
<tr>
<td>11. u = u ∪ op</td>
</tr>
<tr>
<td>12. Sync(R)</td>
</tr>
<tr>
<td>13. return R ∪ Merge(d, u)</td>
</tr>
</tbody>
</table>

Figure 5: Merge function for the master record. We merge all records from replicas in the latest view, which is always a strict superset of the records from replicas in lower views.
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 Synced 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 monotonically 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 \textsc{Merge}. The result from \textsc{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 \textsc{Merge}, the \textsc{Merge} result, will continue to persist 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 \textsc{Prepare-OK} to \textsc{Abort}. If \textsc{Abort} changes to \textsc{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 \textsc{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 \textsc{Prepare-OK} and \textsc{Abort}: \textsc{Abstain} and \textsc{Retry}. \textsc{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.

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 begins a transaction, then executes Reads and Writes during the transaction’s execution period. During this period, the application can Abort the transaction. Once it finishes execution, the application commits 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.

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 ExecConsensus) 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.

Figure 8: Since Prepare is TAPIR’s only consensus operations, TAPIR-EXEC-CONSENSUS simply runs TAPIR’s prepare algorithm at replicas.
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 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.
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 still not already in the COORDINATOR-VIEW-CHANGE state, it returns 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.
This 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 the 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 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, \(\text{Prepare}(A)\) and \(\text{Prepare}(B)\) cannot both return \(\text{PREPARE-OK}\), so one transaction must abort.

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

Further, once decided, the successful consensus results for \(\text{Prepare}(A)\) and \(\text{Prepare}(B)\) will persist in the record of at least one replica out of every quorum, unless it has been modified by the application through \(\text{merge}\). TAPIR will never change another result to a \(\text{PREPARE-OK}\), so the shard will never respond \(\text{PREPARE-OK}\) to both transactions. IR will ensure that the successful consensus result is eventually \(\text{sync’d}\) at all replicas. Once a TAPIR replica prepared a transaction, it will continue to return \(\text{PREPARE-OK}\) until it receives a \(\text{commit}\) or \(\text{abort}\) for the transaction. As a result, if the shard returned \(\text{PREPARE-OK}\) as a successful consensus result to \(\text{Prepare}(A)\), then it will never allow \(\text{Prepare}(B)\) to also return \(\text{PREPARE-OK}\) (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 \(\text{PREPARE-OK}\) from every participating shard on \(\text{Prepare}\). Barring failures, it will ensure that \(\text{commit}\) eventually executes successfully at every participant. TAPIR replicas always execute \(\text{commit}\), even if they did not prepare the transaction, so \(\text{commit}\) 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 \(\text{commit}\), then the client’s \(\text{Prepare}\) must have made it into the operation set of every participant shard with \(\text{PREPARE-OK}\) as the consensus result. IR’s consensus result and eventual consistency properties (P3 and P4) ensure that the \(\text{PREPARE-OK}\) will eventually be applied at all replicas in every participant shard and TAPIR ensures that successful \(\text{PREPARE-OK}\) results are not changed in \(\text{merge}\) (as shown above). Once a TAPIR replica applies \(\text{PREPARE-OK}\), it will continue to return \(\text{PREPARE-OK}\), so once replicas in participant groups have stopped processing operations from previous coordinators, all non-failed replicas in all shards will eventually return \(\text{PREPARE-OK}\). As a result, the backup coordinator must eventually receive \(\text{PREPARE-OK}\) as well from all participants.

In the meantime, the backup coordinator is guaranteed to not receive an \(\text{ABORT}\) from a participant shard. A participant shard will only return an \(\text{ABORT}\) if: (1) a conflicting transaction committed, (2) a majority of the replicas return \(\text{NO-VOTE}\) 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 \(\text{PREPARE-OK}\) (based on our isolation proof) and IR’s consensus result property (P3) ensures that the conflicting transaction could never get a \(\text{PREPARE-OK}\) consensus result, so the conflicting transaction cannot commit. Case (2) is not possible because the client could not have received \(\text{PREPARE-OK}\) as a consensus result if a majority of the replicas do not have the transaction in their \(\text{prepared-list}\) and IR’s P3 and P4 ensures the transaction eventually makes its way into the \(\text{prepared-list}\) of every replica. Case (3) is not possible because the client could not have sent \(\text{ABORT}\) if it got \(\text{PREPARE-OK}\) 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.
We now describe four useful extensions to TAPIR.

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 \( \text{Read(key,version)} \), 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 (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.

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.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th># gets</th>
<th># puts</th>
<th>workload %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add User</td>
<td>1</td>
<td>3</td>
<td>5%</td>
</tr>
<tr>
<td>Follow/Unfollow</td>
<td>2</td>
<td>2</td>
<td>15%</td>
</tr>
<tr>
<td>Post Tweet</td>
<td>3</td>
<td>5</td>
<td>30%</td>
</tr>
<tr>
<td>Load Timeline</td>
<td>rand(1,10)</td>
<td>0</td>
<td>50%</td>
</tr>
</tbody>
</table>

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 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
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 REPLICA_SAFE in MongoDB and the WAIT command [42] for Redis. Cassandra uses REPLICA_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
A technique similar to optimistic timestamp ordering was implemented in TAPIR, which 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 combines 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.

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 –

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

<table>
<thead>
<tr>
<th>Transaction System</th>
<th>Replication Protocol</th>
<th>Read Latency</th>
<th>Commit Latency</th>
<th>Msg At Bottleneck</th>
<th>Isolation Level</th>
<th>Transaction Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDCC [23]</td>
<td>Gen. Paxos [29]</td>
<td>2 (any)</td>
<td>3</td>
<td>2n</td>
<td>Read-Committed</td>
<td>Interactive</td>
</tr>
<tr>
<td>CLOCC [1, 32]</td>
<td>VR [38]</td>
<td>2 (any)</td>
<td>4</td>
<td>2n</td>
<td>Serializable</td>
<td>Interactive</td>
</tr>
<tr>
<td>Lynx [47]</td>
<td>Chain Repl. [46]</td>
<td>–</td>
<td>2n</td>
<td>2</td>
<td>Serializable</td>
<td>Stored procedure</td>
</tr>
<tr>
<td>TAPIR</td>
<td>IR</td>
<td>2 (to any)</td>
<td>2</td>
<td>2</td>
<td>Strict Serializable</td>
<td>Interactive</td>
</tr>
</tbody>
</table>

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.

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


A. IR TLA+ Specification

```
MODULE IR_conservus

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,  
  max_vc, max_req

ASSUME IsFiniteSet(Replicas)

ASSUME QuorumAssumption ≜  
  ∧ Quorums ⊆ SUBSET Replicas  
  ∧ SuperQuorums ⊆ SUBSET Replicas  
  ∧ ∀ Q1, Q2 ∈ Quorums : Q1 ∩ Q2 ≠ {}  
  ∧ ∀ Q ∈ Quorums, R1, R2 ∈ SuperQuorums :  
    Q ∩ R1 ∩ R2 ≠ {}

ASSUME FailuresAssumption ≜  
  ∀ Q ∈ Quorums : Cardinality(Q) > f

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

ReplicaState ≜ {"NORMAL", "FAILED", "RECOVERING", "VIEW-CHANGING"}
ClientState ≜ {"NORMAL", "FAILED"}
```
\(\text{OpType} \triangleq \{\text{"Inconsistent"}, \text{"Consensus"}\}\)
\(\text{OpStatus} \triangleq \{\text{"TENTATIVE"}, \text{"FINALIZED"}\}\)

Definition of operation space

\(\text{MessageId} \triangleq [\text{cid} : \text{Clients}, \text{msgid} : \text{Nat}]\)
\(\text{Operations} \triangleq [\text{type} : \text{OpType}, \text{body} : \text{OpBody}]\)

Message is defined to be the set of all possible messages

\(\text{TODO: Assumptions}\)
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)
\(\text{Message} \triangleq \)
\[\text{[type} : \{\text{"REQUEST"}\},
\text{id} : \text{MessageId},
\text{op} : \text{Operations}]\]
\(\cup [\text{type} : \{\text{"REPLY"}\}, \text{reply no result}\]
\text{id} : \text{MessageId},
\text{v} : \text{Nat},
\text{src} : \text{Replicas}]\]
\(\cup [\text{type} : \{\text{"REPLY"}\}, \text{reply with result}\]
\text{id} : \text{MessageId},
\text{v} : \text{Nat},
\text{res} : \text{Results},
\text{src} : \text{Replicas}\]
\(\text{v} = \text{view num.}\]
\(\cup [\text{type} : \{\text{"START-VIEW-CHANGE"}\},
\text{v} : \text{Nat},
\text{src} : \text{Replicas}]\]
\(\cup [\text{type} : \{\text{"DO-VIEW-CHANGE"}\},
\text{r} : \text{SUBSET} ([\text{msgid} : \text{MessageId},
\text{op} : \text{Operations},
\text{res} : \text{Results}]
\cup [\text{msgid} : \text{MessageId},
\text{op} : \text{Operations}]),
\text{v} : \text{Nat},
\text{src} : \text{Replicas},
\text{dst} : \text{Replicas}]\]
\(\cup \)
[type : {"START-VIEW"},
v : Nat,
src : Replicas] ∪
[type : {"START-VIEW-REPLY"},
v : Nat,
src : Replicas,
dst : Replicas] ∪
[type : {"FINALIZE"}, finalize with no result
id : MessageId,
op : Operations,
res : Results] ∪
[type : {"FINALIZE"}, finalize with result
id : MessageId,
op : Operations,
res : Results] ∪
[type : {"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 = current operation requested by the client
   cMsgCounter = the message I must use for the next operation

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

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

rVars = (rState, rRecord, rViewNumber, rViewReplies) Replica variables.
\[
c\operatorname{Vars} \triangleq \{c\operatorname{CrtOp}, \text{current operation at a client}, c\operatorname{CrtOpToFinalize}, c\operatorname{CrtOpReplies}, \text{current operation replies, } c\operatorname{CrtOpConfirms}, c\operatorname{MsgCounter}, c\operatorname{State}\} \\
a\operatorname{Vars} \triangleq \{a\operatorname{Successful}\} \\
o\operatorname{Vars} \triangleq \{s\operatorname{entMsg}, g\operatorname{ViewChangesNo}\} \\
\operatorname{vars} \triangleq \{r\operatorname{Vars}, c\operatorname{Vars}, o\operatorname{Vars}\} \\
\operatorname{TypeOK} \triangleq \\
\wedge r\operatorname{State}[S] \in [\text{Replicas} \rightarrow \text{ReplicaState}] \\
\wedge r\operatorname{Record}[S] \in [\text{Replicas} \rightarrow \text{SUBSET} (\{\text{msgid} : \text{MessageId}, op : \text{Operations}, res : \text{Results}, status : \text{OpStatus}\} \\
\cup \{\text{msgid} : \text{MessageId}, op : \text{Operations}, status : \text{OpStatus}\}))] \\
\wedge r\operatorname{ViewNumber}[S] \in [\text{Replicas} \rightarrow \text{Nat}] \\
\wedge r\operatorname{ViewReplies}[S] \in [\text{Replicas} \rightarrow \text{SUBSET} \{\text{"do-view-change"}, \text{"start-view-reply"}, \text{type} : \{\text{"do-view-change"}, \text{"start-view-reply"}\}, \operatorname{viewNumber} : \text{Nat}, \operatorname{r} : \text{SUBSET} (\{\text{msgid} : \text{MessageId}, op : \text{Operations}, res : \text{Results}, status : \text{OpStatus}\} \\
\cup \{\text{msgid} : \text{MessageId}, op : \text{Operations}, status : \text{OpStatus}\}))], \operatorname{src} : \text{Replicas}] \\
\wedge s\operatorname{entMsg}[S] \in \text{SUBSET } \text{Message} \\
\wedge c\operatorname{CrtOp}[S] \in [\text{Clients} \rightarrow \text{Operations} \cup \{\emptyset\}] \\
\wedge c\operatorname{CrtOpToFinalize} \in [\text{Clients} \rightarrow \text{Operations} \cup \{\emptyset\}] \\
\wedge c\operatorname{CrtOpReplies}[S] \in [\text{Clients} \rightarrow \text{SUBSET} (\{\operatorname{viewNumber} : \text{Nat}, res : \text{Results}, \operatorname{src} : \text{Replicas}\} \\
\cup \{\operatorname{viewNumber} : \text{Nat}, \operatorname{src} : \text{Replicas}\}))] \\
\wedge c\operatorname{CrtOpConfirms}[S] \in [\text{Clients} \rightarrow \text{SUBSET} (\{\operatorname{viewNumber} : \text{Nat}, res : \text{Results}, \operatorname{src} : \text{Replicas}\})] \\
\wedge c\operatorname{MsgCounter}[S] \in [\text{Clients} \rightarrow \text{Nat}] \\
\wedge c\operatorname{State} \in [\text{Clients} \rightarrow \text{ClientState}] \\
\wedge a\operatorname{Successful} \in \text{SUBSET} (\{\operatorname{mid} : \text{MessageId}, \})
\]
\[
\begin{align*}
op &: \text{Operations}, \\
res &: \text{Results} \\
\cup [\text{mid} &: \text{MessageId}, \\
\op &: \text{Operations}]) \\
\& g\text{ViewChangesNo}[S] \in \text{Nat}
\end{align*}
\]

\textbf{Init} \triangleq
\begin{align*}
\& r\text{State} = [r \in \text{Replicas} \mapsto "NORMAL"] \\
\& r\text{Record} = [r \in \text{Replicas} \mapsto \{\}] \\
\& r\text{ViewNumber} = [r \in \text{Replicas} \mapsto 0] \\
\& r\text{ViewReplies} = [r \in \text{Replicas} \mapsto \{\}] \\
\& \text{sentMsg} = \{\} \\
\& \text{cCrtOp} = [c \in \text{Clients} \mapsto \{\}] \\
\& \text{cCrtOpToFinalize} = [c \in \text{Clients} \mapsto \{\}] \\
\& \text{cCrtOpReplies} = [c \in \text{Clients} \mapsto \{\}] \\
\& \text{cCrtOpConfirms} = [c \in \text{Clients} \mapsto \{\}] \\
\& \text{cMsgCounter} = [c \in \text{Clients} \mapsto 0] \\
\& \text{cState} = [c \in \text{Clients} \mapsto "NORMAL"] \\
\& a\text{Successful} = \{\} \\
\& g\text{ViewChangesNo} = 0
\end{align*}

\textbf{Actions}

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

\textbf{Client Actions}

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

Client sends a request
\textit{ClientRequest}(c, \op) \triangleq
\begin{align*}
\& \text{cCrtOp}[S][c] = \{\} \quad \text{the client is not waiting for a result of another operation} \\
\& \text{cCrtOpToFinalize}[S][c] = \{\} \\
\& \text{cMsgCounter'} = [\text{cMsgCounter EXCEPT } !S][c] = 0 + 1 \\
\& \text{cCrtOp'} = [\text{cCrtOp EXCEPT } !S][c] = \op \\
\& \text{Send}(\{\text{type} \mapsto "REQUEST",} \\
\quad \text{id} \mapsto [\text{cid} \mapsto c, \text{msgid} \mapsto \text{cMsgCounter}[S][c] + 1], \text{op} \mapsto \text{op}\}) \\
\& \text{UNCHANGED } (\text{rVars, aVars, cCrtOpReplies, cCrtOpToFinalize,} \\
\quad \text{cCrtOpConfirms, cState, gViewChangesNo}) \\
\& \text{cMsgCounter}[S][c] < \text{max req BOUND the number of requests a client can make}
\end{align*}
Client received a reply

\[
\text{ClientReceiveReply}(c) \triangleq \\
\exists \, \text{msg} \in \text{sentMsg}[S] : \\
\quad \land \, \text{msg.type} = \text{"REPLY"} \\
\quad \land \, \text{cCrtOp}[S][c] \neq \emptyset \\
\quad \land \, \text{msg.id} = [\text{cid} \mapsto c, \text{msgid} \mapsto \text{cMsgCounter}[S][c]] \quad \text{reply to c's request for crt op}
\]

\text{TODO: if already reply from src, keep the most recent one (biggest view Number)}

\land \, \text{Assert(Cardinality(cCrtOpReplies[c]) < 10, “cCrtOpReplies cardinality bound”)}

\land \, \lor \land \text{cCrtOp}[S][c].\text{type} = \text{"Inconsistent"} \\
\land \, \text{cCrtOpReplies}' = [\text{cCrtOpReplies EXCEPT } ![S][c] = \emptyset \cup \\
\quad \{[\text{viewNumber} \mapsto \text{msg.v}, \\
\quad \text{src} \mapsto \text{msg.src}]\}] \\
\lor \land \text{cCrtOp}[S][c].\text{type} = \text{"Consensus"} \\
\land \, \text{cCrtOpReplies}' = [\text{cCrtOpReplies EXCEPT } ![S][c] = \emptyset \cup \\
\quad \{[\text{viewNumber} \mapsto \text{msg.v}, \\
\quad \text{res} \mapsto \text{msg.res}, \\
\quad \text{src} \mapsto \text{msg.src}]\}]

\land \, \text{UNCHANGED } \langle \text{cCrtOp, cCrtOpToFinalize, cCrtOpConfirms,} \\
\text{cMsgCounter, cState, rVars, aVars, oVars} \rangle

"Helper" formulas

\text{matchingViewNumbers}(Q, c) \triangleq
\land \forall r \in Q : \\
\quad \land \exists \, \text{reply} \in \text{cCrtOpReplies}[S][c] : \text{reply.src} = r \\
\land \forall p \in Q : \exists rr, pr \in \text{cCrtOpReplies}[S][c] : \\
\quad \land \, \text{rr.src} = r \\
\quad \land \, \text{pr.src} = p \\
\quad \land \, \text{rr.viewNumber} = \text{pr.viewNumber}

\text{matchingViewNumbersAndResults}(Q, c) \triangleq
\land \forall r \in Q : \\
\quad \land \exists \, \text{reply} \in \text{cCrtOpReplies}[S][c] : \text{reply.src} = r \\
\land \forall p \in Q : \exists rr, pr \in \text{cCrtOpReplies}[S][c] : \\
\quad \land \, \text{rr.src} = r \\
\quad \land \, \text{pr.src} = p \\
\quad \land \, \text{rr.viewNumber} = \text{pr.viewNumber} \\
\quad \land \, \text{rr.res} = \text{pr.res}

\text{IR Client received enough responses to decide} \\
\text{what to do with the operation}

\text{ClientDecideOp}(c) \triangleq \\
\land \, \text{cCrtOp}[S][c] \neq \emptyset
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
\[\begin{align*}
& \wedge \text{Send}([\text{type} \mapsto \text{"FINALIZE"},]
\quad \text{id} \mapsto [\text{cid} \mapsto c, \text{msgid} \mapsto c\text{MsgCounter}[S][c]],
\quad \text{op} \mapsto c\text{CrtOp}[S][c],
\quad \text{res} \mapsto \text{res}]) \\
& \wedge a\text{Successful}' = a\text{Successful} \cup \\
\quad \{[\text{mid} \mapsto [\text{cid} \mapsto c,]
\quad \text{msgid} \mapsto c\text{MsgCounter}[S][c]],
\quad \text{op} \mapsto c\text{CrtOp}[S][c],
\quad \text{res} \mapsto \text{res}\}) \\
& \wedge \text{SuccessfulConsensusOp}(c\text{CrtOp}[S][c], \text{res}) \\
& \wedge \text{unchanged} \langle c\text{CrtOpToFinalize} \rangle \\
& \wedge c\text{CrtOp}' = [c\text{CrtOp} \text{EXCEPT } ![S][c] = \emptyset] \\
& \wedge c\text{CrtOpReplies}' = [c\text{CrtOpReplies} \text{EXCEPT } ![S][c] = \emptyset] \\
& \wedge \text{unchanged} \langle c\text{MsgCounter}, c\text{State}, c\text{CrtOpConfirms}, r\text{Vars}, g\text{ViewChangesNo} \rangle \\
\end{align*}\]

Client received a confirm
\[\begin{align*}
\text{ClientReceiveConfirm}(c) \triangleq & \\
\exists \text{msg} \in \text{sentMsg}[S] : \\
& \wedge \text{msg.type} = \text{"CONFIRM"} \\
& \wedge c\text{CrtOpToFinalize}[S][c] \neq \emptyset \\
& \wedge \text{msg.id} = [\text{cid} \mapsto c, \text{msgid} \mapsto c\text{MsgCounter}[S][c]] \text{ reply to c's request for crt op} \\
& \wedge c\text{CrtOpConfirms}' = [c\text{CrtOpConfirms} \text{EXCEPT } ![S][c] = \emptyset] \cup \\
\quad \{[\text{viewNumber} \mapsto \text{msg.v},
\quad \text{res} \mapsto \text{msg.res},
\quad \text{src} \mapsto \text{msg.src}]\} \\
& \wedge \text{unchanged} \langle c\text{CrtOp}, c\text{CrtOpReplies}, c\text{CrtOpToFinalize}, c\text{MsgCounter},
\text{cState, rVars, aVars, oVars} \rangle \\
\end{align*}\]

An operation is finalized by a client and result returned to the application
\[\begin{align*}
\text{ClientFinalizedOp}(c) \triangleq & \\
\wedge c\text{CrtOpToFinalize}[S][c] \neq \emptyset \\
\wedge \exists Q \in \text{Quorums} : \\
\quad \text{IR client received a quorum of responses} \\
\wedge \forall r \in Q : \\
\quad \exists \text{reply} \in c\text{CrtOpConfirms}[S][c] : \text{reply.src} = r \\
\wedge \text{LET} \\
\quad \text{take the result in the biggest view number} \\
\quad \text{reply} \triangleq \text{CHOOSE} \text{reply} \in c\text{CrtOpConfirms}[S][c] : \\
\quad \neg \exists \text{rep} \in c\text{CrtOpConfirms}[S][c] : \\
\quad \text{rep.viewNumber} > \text{reply.viewNumber} \\
\end{align*}\]

\[\begin{align*}
& \wedge a\text{Successful}' = a\text{Successful} \cup \\
\quad \{[\text{mid} \mapsto [\text{cid} \mapsto c,]
\quad \text{msgid} \mapsto c\text{MsgCounter}[S][c],
\quad \text{res} \mapsto \text{res}]\} \\
\end{align*}\]
Replica Actions

Replica sends a reply

\[
\text{ReplicaReceiveRequest}(r) \triangleq \\
\exists \text{msg} \in \text{sentMsg}[S] : \\
\land \text{msg.type} = \text{"REQUEST"} \\
\land \neg \exists \text{rec} \in \text{rRecord}[S][r] : \text{rec.msgid} = \text{msg.id} \\
\text{not alredy replied for this op} \\
\lor \lor \ \land \text{msg.op.type} = \text{"Inconsistent"} \\
\land \text{Send}([\text{type} \mapsto \text{"REPLY"}, \text{id} \mapsto \text{msg.id}, \text{v} \mapsto \text{rViewNumber}[S][r], \text{src} \mapsto r]) \\
\land \text{rRecord'} = [\text{rRecord EXCEPT } ![S][r] = @ \cup \{[\text{msgid} \mapsto \text{msg.id}, \text{op} \mapsto \text{msg.op}, \text{status} \mapsto \text{"TENTATIVE"}]\})
\]

\lor \land \text{msg.op.type} = \text{"Consensus"} \\
\land \text{LET res } \triangleq \text{ExecConsensus}(\text{msg.op}) \\
\text{IN} \\
\land \text{Send}([\text{type} \mapsto \text{"REPLY"}, \text{id} \mapsto \text{msg.id}, \text{v} \mapsto \text{rViewNumber}[S][r], \text{res} \mapsto \text{res}, \text{src} \mapsto r]) \\
\land \text{rRecord'} = [\text{rRecord EXCEPT } ![S][r] = @ \cup \{[\text{msgid} \mapsto \text{msg.id}, \text{op} \mapsto \text{msg.op}, \text{status} \mapsto \text{"TENTATIVE"}]\})
Replica receive a message from an IR Client to finalize an `op`. For inconsistent operations the replica sends `<CONFIRM>` and executes the operation.

**TODO:** Write this more compact

```plaintext
ReplicaReceiveFinalize(r) \equiv
\exists msg \in sentMsg[S] :
\land msg.type = "FINALIZE"
\land \forall \land msg.op.type = "Inconsistent"
\land Send([type → "CONFIRM",
    v → rViewNumber[S][r],
    id → msg.id,
    op → msg.op,
    src → r])
\land \forall \exists rec \in rRecord[S][r] :
    \land rec.msgid = msg.id
    \land rec.op = msg.op \quad \text{Replica knows of this op}
    \land \text{IF} rec.status \neq "FINALIZED"
    \text{THEN} ExecInconsistent(msg.op)
    \text{ELSE} TRUE
\land rRecord' = [rRecord \text{ EXCEPT } !(S)[r] = (@ \{ rec \}) \cup
    \{ [msgid → msg.id,
        op → msg.op,
        status → "FINALIZED" ] \})}

\lor \exists rec \in rRecord[S][r] :\quad \text{Replica didn’t hear of this op}
\land rec.msgid = msg.id
\land rec.op = msg.op
\land rRecord' = [rRecord \text{ EXCEPT } !(S)[r] = @ \cup
    \{ [msgid → msg.id,
        op → msg.op,
        status → "FINALIZED" ] \})}

\land \text{ExecInconsistent}(msg.op)
\lor \land msg.op.type = "Consensus"
\lor \exists rec \in rRecord[S][r] :
    \land rec.msgid = msg.id
    \land rec.op = msg.op \quad \text{Replica knows of this op}
\land \forall \land rec.status = "TENTATIVE" \quad \text{Operation tentative}
\land rRecord' = [rRecord \text{ EXCEPT } !(S)[r] = (@ \{ rec \}) \cup
    \{ [msgid → msg.id,
        op → msg.op,
        status → "FINALIZED" ] \})}
```

31
A replica starts the view change procedure

\begin{align*}
&\text{ReplicaStartViewChange}(r) \triangleq \\
&\wedge \text{Send}([\text{type} \mapsto \text{"START-VIEW-CHANGE"}], \\
&\quad v \mapsto r\text{ViewNumber}[r], \\
&\quad src \mapsto r) \\
&\quad \wedge r\text{State}' = [r\text{State} \setminus ![r] = \text{"RECOVERING"}] \\
&\quad \wedge \text{UNCHANGED} (r\text{ViewNumber}, r\text{ViewReplies}, r\text{Record}, c\text{Vars}, a\text{Vars}) \\
&\quad \wedge g\text{ViewChangesNo} < max_{vc} \ \text{BOUND on number of view changes}
\end{align*}
∧ gViewChangesNo' = gViewChangesNo + 1

A replica received a message to start view change
ReplicaReceiveStartViewChange(r) ≡
∧ ∃ msg ∈ sentMsg[S] :
   ∧ msg.type = "START-VIEW-CHANGE"
   ∧ LET v_new ≡
       IF msg.v > rViewNumber[r] THEN msg.v
       ELSE rViewNumber[S][r]
   IN
   ∧ ¬∃ m ∈ sentMsg[S] : not already sent (just to bound the model checker)
   ∧ m.type = "DO-VIEW-CHANGE"
   ∧ m.v ≥ msg.v
   ∧ m.dst = msg.src
   ∧ m.src = r
   ∧ Send([type ↦ "DO-VIEW-CHANGE",
            v ↦ v_new + 1,
            r ↦ rRecord[r],
            src ↦ r,
            dst ↦ msg.src])
∧ rViewNumber' = rViewNumber EXCEPT !(S)[r] = v_new + 1
∧ rState' = rState EXCEPT ![S][r] = "VIEW-CHANGING"
∧ UNCHANGED (cVars, rRecord, rViewReplies, aVars, gViewChangesNo)

Replica received DO-VIEW-CHANGE message
ReplicaReceiveDoViewChange(r) ≡
∧ ∃ msg ∈ sentMsg[S] :
   ∧ msg.type = "DO-VIEW-CHANGE"
   ∧ msg.dst = r
   ∧ msg.v > rViewNumber[r]
   ∧ rViewReplies' = rViewReplies EXCEPT ![r] = ∅ ∪
      {[type ↦ "do-view-change",
         viewNumber ↦ msg.v,
         r ↦ msg.r,
         src ↦ msg.src]}
∧ UNCHANGED (cVars, rViewNumber, rRecord, rState, aVars, oVars)

A replica received enough view change replies to start processing in the new view
ReplicaDecideNewView(r) ≡
∧ ∃ Q ∈ Quorums :
   ∧ ∀ rep ∈ Q : ∃ reply ∈ rViewReplies[r] : ∧ reply.src = rep
   ∧ reply.type = "do-view-change"
   received at least a quorum of replies
∧ LET recoveredConsensusOps_a ≡
   any consensus operation found in at least a majority of a Quorum
   { x ∈ UNION {y.r : y ∈ {z ∈ rViewReplies[S][r] : z.src ∈ Q}} :

33
∧ x[2].type = “Consensus”
∧ ∃ P ∈ SuperQuorums :
    ∀ rep ∈ Q ∩ P :
        ∃ reply ∈ rViewReplies[r] :
            ∧ reply.src = rep
            ∧ x ∈ reply.r same op, same result

recoveredConensusOps_b ≜ TODO: what result? from the app?
the rest of consensus ops found in at least one record (discard the result)
{⟨z[1], z[2]⟩ : z ∈ \{x ∈ UNION \{y.r : y ∈ \{z ∈ rViewReplies[S][r] : z.src ∈ Q\}\} :
    ∧ x[2].type = “Consensus”
    ∧ ¬ x ∈ recoveredConensusOps_a\}}

recoveredInconsistentOps_c ≜
any inconsistent operation found in any received record (discard the result)
{⟨z[1], z[2]⟩ : z ∈ \{x ∈ UNION \{y.r : y ∈ \{z ∈ rViewReplies[S][r] : z.src ∈ Q\}\} :
    x[2].type = “Inconsistent” \}}

IN
∧ AppRecoverOpsResults(recoveredConensusOps_a)
∧ AppRecoverOps(recoveredConensusOps_b)
∧ AppRecoverOps(recoveredInconsistentOps_c)
∧ rRecord’ = [rRecord EXCEPT ![S][r] = @ ∪ recoveredConensusOps_a
∪ recoveredConensusOps_b
∪ recoveredInconsistentOps_c]

∧ LET v_new ≜
max view number received
CHOOSE v ∈ \{x.viewNumber : x ∈ rViewReplies[r]\} :
    ∀ y ∈ rViewReplies[r] :
        y.viewNumber ≤ v

IN
∧ Send([type ↦ →“START-VIEW”,
    v ↦ v_new,
    src ↦ r])
∧ rViewNumber’ = [rViewNumber EXCEPT ![r] = v_new]
∧ rViewReplies’ = [rViewReplies EXCEPT ![r] = {}]
∧ UNCHANGED ⟨rState, cVars, aVars, gViewChangesNo⟩

A replica receives a start view message
ReplicaReceiveStartView(r) ≜
∧ ∃ msg ∈ sentMsg :
    ∧ msg.type = “START-VIEW”
    ∧ msg.v ≥ rViewNumber[r]
    ∧ msg.src ≠ r don’t reply to myself
∧ Send([type ↦ →“START-VIEW-REPLY”,
    34
\( v \mapsto \text{msg}.v, \)
\( \text{src} \mapsto r, \)
\( \text{dst} \mapsto \text{msg}.\text{src} \)
\( \land r\text{ViewNumber}' = [r\text{ViewNumber} \text{EXCEPT } ![r] = \text{msg}.v] \)
\( \land r\text{State}' = [r\text{State} \text{EXCEPT } ![r] = \text{“NORMAL”}] \)
\( \land \text{UNCHANGED} (r\text{Record}, r\text{ViewReplies}, c\text{Vars}, a\text{Vars}, g\text{ViewChangesNo}) \)

\( \text{ReplicaReceiveStartViewReply}(r) \triangleq \)
\( \land \exists \text{msg} \in \text{sentMsg} : \)
\( \land \text{msg.type} = \text{“START-VIEW-REPLY”} \)
\( \land \text{msg.dst} = r \)
\( \land \text{msg}.v > r\text{ViewNumber}[r] \) receive only if bigger than the last view I was in
\( \land r\text{ViewReplies}' = [r\text{ViewReplies} \text{EXCEPT } ![S][r] = \emptyset \cup \{[\text{type} \mapsto \text{“start-view-reply”}, \text{viewNumber} \mapsto \text{msg}.v, r \mapsto \emptyset, \text{src} \mapsto \text{msg}.\text{src}]\}] \)
\( \land \text{UNCHANGED} (r\text{Record}, r\text{ViewNumber}, c\text{Vars}, a\text{Vars}, o\text{Vars}) \)

\( \text{ReplicaRecover}(r) \triangleq \) we received enough START-VIEW-REPLY messages
\( \exists Q \in \text{Quorums} : \)
\( \land r \in Q \)
\( \land \forall p \in Q : \lor p = r \)
\( \lor \land p \neq r \)
\( \land \exists \text{reply} \in r\text{ViewReplies}[S][r] : \land \text{reply.src} = p \)
\( \land \text{reply.type} = \text{“start-view-reply”} \)
\( \land r\text{ViewReplies}' = [r\text{ViewReplies} \text{EXCEPT } ![S][r] = \emptyset] \)
\( \land r\text{State}' = [r\text{State} \text{EXCEPT } ![r] = \text{“NORMAL”}] \)
\( \land \text{UNCHANGED} (r\text{Record}, r\text{ViewNumber}, c\text{Vars}, a\text{Vars}, o\text{Vars}) \)

\( \text{ReplicaResumeViewChange}(r) \triangleq \) \text{TODO}: On timeout
\( \text{FALSE} \)

A replica fails and looses everything
\( \text{ReplicaFail}(r) \triangleq \text{TODO}: \text{check cardinality} \)
\( \land r\text{State}' = [r\text{State} \text{EXCEPT } ![S][r] = \text{“FAILED”}] \)
\( \land r\text{Record}' = [r\text{Record} \text{EXCEPT } ![S][r] = \emptyset] \)
\( \land r\text{ViewNumber}' = [r\text{ViewNumber} \text{EXCEPT } ![r] = 0] \) \text{TODO}: check what happens if we loose the view number
\( \land r\text{ViewReplies}' = [r\text{ViewReplies} \text{EXCEPT } ![S][r] = \emptyset] \)
\( \land \text{UNCHANGED} (r\text{ViewNumber}, c\text{Vars}, a\text{Vars}, o\text{Vars}) \)
\( \land \text{Cardinality} (\{re \in \text{Replicas} : \}) < f \)

\[ \text{High-Level Actions} \]
\textit{ClientAction}(c) \triangleq \forall \ W cState[c] = "NORMAL"
\quad \land \ W ClientRequest(c) \land \land some client tries to replicate commit an operation
\quad \lor ClientReceiveReply(c) \land \land some client receives a reply from a replica
\quad \lor ClientReceiveConfirm(c) \land \land some client receives a confirm from a replica
\quad \lor ClientFail(c) \land \land some client fails
\quad \lor ClientDecideOp(c) \land \land an operation is successful at some client
\quad \lor ClientFinalizedOp(c) \land \land an operation was finalized at some client
\forall \ W cState[c] = "FAILED"
\land \lor ClientRecover(c)

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

\textit{Next} \triangleq \forall \exists c \in \text{Clients} : \textit{ClientAction}(c)
\lor \exists r \in \text{Replicas} : \textit{ReplicaAction}(r)

\textit{Spec} \triangleq \text{Init} \land \Diamond [\textit{Next}]_{\textit{vars}}

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

Inv ≡ TypeOK ∧ FaultTolerance
B. TAPIR TLA+ Specification

```
MODULE TAPIR

This is a TLA+ specification of the TAPIR algorithm.

EXTENDS FiniteSets, Naturals, TLC, TLAPS

Max(S) ≜ IF S = {} THEN 0 ELSE CHOOSE i ∈ S : ∀ j ∈ S : j ≤ 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 ≜ (rState, rRecord, rViewNumber, rViewReplies)
irClientVars ≜ (cCrtOp,
  current operation at a client
  cCrtOpReplies,
  current operation replies
  cMsgCounter,
  cState,
  cCrtOpToFinalize,
  cCrtOpConfirms) Client variables.

irAppVars ≜ (aSuccessful) Application variables

irOtherVars ≜ (sentMsg, gViewChangesNo) 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 ≜ (rPreparedTxns, rStore, rTxnsLogAborted, rTxnsLogCommited,
  rClock)
tapirClientVars ≜ (cCrtTxn, cClock)
```
StoreEntry $\triangleq [vs : \text{Nat}, \text{val} : \text{Nat}] \quad \text{vs} = \text{version}$

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

TransactionTs $\triangleq [\text{cid} : \text{Clients}, \text{clock} : \text{Nat}] \quad \text{Timestamp}$

ReadSet $\triangleq [\text{key} : \text{Nat}, \text{val} : \text{Nat}, \text{vs} : \text{Nat}]$

WriteSet $\triangleq [\text{key} : \text{Nat}, \text{val} : \text{Nat}]$

Transaction $\triangleq [rSet : \text{SUBSET ReadSet},$
$$\quad wSet : \text{SUBSET WriteSet},$
$$\quad \text{shards} : \text{SUBSET Nat}]$

TypeOK $\triangleq$
$$\land r\text{Store} \in \{\text{Shards}[i] : i \in 1..\text{NrShards}\} \rightarrow \text{SUBSET Store}\right]$  
$$\land r\text{PreparedTxs} \in \{\text{Shards}[i] : i \in 1..\text{NrShards}\} \rightarrow \text{SUBSET Transaction}\right]$  
$$\land r\text{TxsnsLogAborted} \in \{\text{Shards}[i] : i \in 1..\text{NrShards}\} \rightarrow \text{SUBSET Transaction}\right]$  
$$\land r\text{TxsnsLogCommited} \in \{\text{Shards}[i] : i \in 1..\text{NrShards}\} \rightarrow \text{SUBSET Transaction}\right]$  

TAPIRResults $\triangleq \{\text{“Prepare-OK”}, \text{“Retry”}, \text{“Prepare-Abstain”}, \text{“Abort”}\}$

TAPIROpType $\triangleq \{\text{“Prepare”}, \text{“ABORT”}, \text{“COMMIT”}\}$

TAPIROpBody $\triangleq [\text{opType} : \text{TAPIROpType}, \text{txn} : \text{Transaction}]$

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

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

TAPIR implementation of IR interface

TAPIRExecInconsistent(op) $\triangleq \text{TRUE}$

TAPIRExecConsensus(op) $\triangleq \text{IF op.type = “Consensus” THEN “Prepare-OK” ELSE “Abort”}$

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 $\triangleq$
$$\land r\text{State} = [s \in 1..\text{NrShards} \mapsto [r \in \text{Shards}[s] \mapsto \text{“NORMAL”}]]$$
$$\land r\text{Record} = [s \in 1..\text{NrShards} \mapsto [r \in \text{Shards}[s] \mapsto \emptyset]]$$
$$\land r\text{ViewNumber} = [s \in 1..\text{NrShards} \mapsto [r \in \text{Shards}[s] \mapsto 0]]$$
$$\land r\text{ViewReplies} = [s \in 1..\text{NrShards} \mapsto [r \in \text{Shards}[s] \mapsto \emptyset]]$$
$$\land \text{sentMsg} = [s \in 1..\text{NrShards} \mapsto \emptyset]$$
$$\land c\text{CrtOp} = [s \in 1..\text{NrShards} \mapsto [c \in \text{Clients} \mapsto \emptyset]]$$
∧ cCrtOpToFinalize = \[s ∈ 1 . . NrShards ↦→ [c ∈ Clients ↦→ ⟨⟩]\]
∧ cMsgCounter = \[s ∈ 1 . . NrShards ↦→ [c ∈ Clients ↦→ 0]\]
∧ cCrtOpReplies = \[s ∈ 1 . . NrShards ↦→ [c ∈ Clients ↦→ {\{\}]}\]
∧ cCrtOpConfirms = \[s ∈ 1 . . NrShards ↦→ [c ∈ Clients ↦→ {\{\}]}\]
∧ cState = \[c ∈ Clients ↦→ "NORMAL""]
∧ aSuccessful = {\{\}}
∧ gViewChangesNo = \[s ∈ 1 . . NrShards ↦→ 0\]

IR instance per shard TODO: modify replica also

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

TAPIR messages

Message ≡
\[\{\text{type} : \{\text{"READ"}\},\]
key : Nat,
dst : UNION Shards\]
∪
\[\{\text{type} : \{\text{"READ-REPLY"}\},\]
key : Nat,
val : Nat,
vs : Nat,
version
        \]
dst : Clients\]
∪
\[\{\text{type} : \{\text{"READ-VERSION"}\},\]
key : Nat,
vs : Nat,
dst : UNION Shards\]
∪
\[\{\text{type} : \{\text{"READ-VERSION-REPLY"}\},\]
key : Nat,
vs : Nat,
\(dst : Clients\]

\[
\text{InitTAPIR} \triangleq \land cCrtTxn = [c \in Clients \mapsto \langle \rangle] \\
\land cClock = [c \in Clients \mapsto 0] \\
\land rPreparedTxns = [s \in 1 \ldots NrShards \mapsto [r \in Shards[s] \mapsto \{\}]] \\
\land rStore = [r \in \cup \{Shards[i] : i \in 1 \ldots NrShards\} \mapsto \{\}] \\
\land rTxnsLogAborted = [s \in 1 \ldots NrShards \mapsto [r \in Shards[s] \mapsto \{\}] ] \\
\land rClock = [s \in 1 \ldots NrShards \mapsto [r \in Shards[s] \mapsto 0] ]
\]

\[
\text{Init} \triangleq \text{InitIR} \land \text{InitTAPIR}
\]

### Tapir replica actions

\[
\text{TAPIRReplicaReceiveRead}(r) \triangleq \text{TRUE}
\]

\[
\text{TAPIRReplicaAction}(r) \triangleq \\
\land \land rState[r] = \text{"NORMAL"} \\
\land \land \text{TAPIRReplicaReceiveRead}(r)
\]

### Tapir client actions

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

\[
\text{shards} \triangleq \{s \in 1 \ldots NrShards : \\
\}
\]
\[ \forall \exists \text{trse} \in t.rSet : s = (\text{trse}.key\%\text{NrShards}) + 1 \]
\[ \forall \exists \text{twse} \in t.wSet : s = (\text{twse}.key\%\text{NrShards}) + 1 \]

IN
\[ \land \text{Cardinality}(rSet) = \text{Cardinality}(t.rSet) \land \text{found all the reads} \]
\[ \land \text{cCrtTxn}' = [\text{cCrtTxn \ except} ![c] = [rSet \mapsto rSet, wSet \mapsto t.wSet, shards \mapsto shards]] \]
\[ \land \text{UNCHANGED} \ (\text{irReplicaVars, irClientVars, irOtherVars, irAppVars, tapirReplicaVars, cClock}) \]

\[ \text{TAPIRClientPrepareTxn}(c) \triangleq \]
\[ \land \text{cCrtTxn}[c] \neq \langle \rangle \]
\[ \land \exists s \in \text{cCrtTxn}[c].shards : \text{prepare in shard } s \]
\[ \quad \text{- ok if already prepared} \]
\[ \land \text{IR}(s)! \text{ClientRequest}(c, [\text{type} \mapsto \text{"Consensus"}, \text{body} \mapsto ([\text{"Prepare"}, \text{cCrtTxn}]))} \]
\[ \land \text{UNCHANGED} \ (\text{irReplicaVars, irAppVars, cCrtOpReplies, cCrtOpConfirms, cCrtOpToFinalize, gViewChangesNo, cState, tapirClientVars, tapirReplicaVars}) \]

\[ \text{TAPIRClientAction}(c) \triangleq \]
\[ \lor \land \text{cState}[c] = \text{"NORMAL"} \]
\[ \land \lor \text{TAPIRClientExecuteTxn}(c) \text{ for now just simulate this} \]
\[ \quad \text{(don’t send explicit READ messages)} \]
\[ \lor \text{TAPIRClientPrepareTxn}(c) \]
\[ \lor 2\text{PC}(c) \]

---

**High-Level Actions**

\[ \text{Next} \triangleq \]
\[ \land \lor \exists c \in \text{Clients} : \text{TAPIRClientAction}(c) \]
\[ \lor \land \exists s \in 1 \ldots \text{NrShards} : \text{IR}(s)! \text{Next} \]
\[ \land \text{UNCHANGED} \ (\text{tapirClientVars, tapirReplicaVars}) \]

\[ \text{Inv} \triangleq \text{Cardinality}(a\text{Successful}) < 2 \]