When Is Operation Ordering Required in Replicated Transactional Storage?

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

Abstract

Today's replicated transactional storage systems typically have a layered architecture, combining protocols for transaction coordination, consistent replication, and concurrency control. These systems generally require costly strongly-consistent replication protocols like Paxos, which assign a total order to all operations. To avoid this cost, we ask whether all replicated operations in these systems need to be strictly ordered. Recent research has yielded replication protocols that can avoid unnecessary ordering, e.g., by exploiting commutative operations, but it is not clear how to apply these to replicated transaction processing systems. We answer this question by analyzing existing transaction processing designs in terms of which replicated operations require ordering and which simply require fault tolerance. We describe how this analysis leads to our recent work on TAPIR, a transaction protocol that efficiently provides strict serializability by using a new replication protocol that provides fault tolerance but not ordering for most operations.

1 Introduction

Distributed storage systems for today's large-scale web applications must meet a daunting set of requirements: they must offer high performance, graceful scalability, and continuous availability despite the inevitablity of failures. Increasingly, too, 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 [11, 3, 8, 5, 6] reflect this trend. One notable example is Google's Spanner system [6], which guarantees strictly-serializable (aka linearizable or externally consistent) transaction ordering [6].

Generally, distributed transactional storage with strong consistency comes at a heavy performance price. These systems typically integrate several expensive mechanisms, including concurrency control schemes like strict two-phase locking, strongly consistent replication protocols like Paxos, and atomic commitment protocols like two-phase commit. The costs associated with these mechanisms often drive application developers to use more efficient, weak consistency protocols that fail to provide strong system guarantees.

In conventional designs, these protocols – concurrency control, replication, and atomic commitment – are implemented in separate layers, with each layer providing a subset of the guarantees required for distributed

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

ACID transactions. For example, a system may partition data into shards, where each shard is replicated using Paxos, and use two-phase commit and two-phase locking to implement serializable transactions across shards. Though now frequently implemented together, these protocols were originally developed to address separate issues in distributed systems. In our recent work [23, 24], we have taken a different approach: we consider the storage system architecture as a whole and identify opportunities for cross-layer optimizations to improve performance.

To improve the performance of replicated transactional systems, we ask the question: *do all replicated operations need to be strictly ordered*? Existing systems treat the replication protocol as an implementation of a persistent, ordered log. Ensuring a total ordering, however, require cross-replica coordination on every operation, increasing system latency, and is typically implemented with a designated leader, introducing a system bottleneck. A recent line of distributed systems research has developed more efficient replication protocols by identifying cases when operations do not need to be ordered – typically by having the developer express which operations commute with others [4, 14, 18, 15]. Yet it is not obvious how to apply these techniques to the layered design of transactional storage systems: the operations being replicated are not transactions themselves, but coordination operations like PREPARE or COMMIT records.

To answer this question, we analyzed the interaction of atomic commitment, consistent replication, and concurrency control protocols in common combinations. We consider both their requirements on layers below and the guarantees they provide for layers above. Our analysis leads to several key insights about the protocols implemented in distributed transactional storage systems:

- The concurrency control mechanism is the *application* for the replication layer, so its requirements impact the replication algorithm.
- Unlike disk-based durable storage, replication can *separate* ordering/consistency and fault-tolerance guarantees with different costs for each.
- Consensus-based replication algorithms like Paxos *are not* the most efficient way to provide ordering.
- Enforcing consistency at every layer *is not necessary* for maintaining global transactional consistency.

We describe how these observations led us to design a new transactional storage system, *TAPIR* [23, 24]. TAPIR provides strict serializable isolation of transactions, but relies on a weakly consistent underlying replication protocol, *inconsistent replication* (IR). IR is designed to efficiently support fault-tolerant but unordered operations, and TAPIR uses an *optimistic timestamp ordering* technique to make most of its replicated operations unordered. This article does not attempt to provide a complete explanation of the TAPIR protocol, its performance, or its correctness. Rather, it analyzes it along with additional protocols to demonstrate the value of separating operation ordering from fault tolerance in replicated transactional systems.

2 Coordination Protocols

This paper is about the protocols used to build scalable, fault-tolerant distributed storage system, like distributed key-value stores or distributed databases. In this section, we specify the category of systems we are considering, and review the standard protocols that they use to coordinate replicated, distributed transactions.

2.1 Transactional System Model

The protocols that we discuss in this paper are designed for scalable, distributed transactional storage. We assume the storage system divides stored data across several *shards*. Within a shard, we assume the system uses replication for availability and fault tolerance. Each shard is replicated across a set of storage servers organized

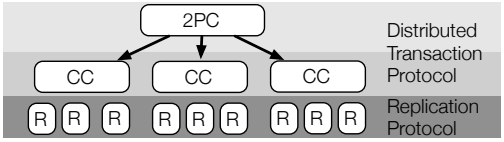


Figure 1: Standard protocol architecture for transactional distributed storage systems.

into a *replica group*. Each replica group consists of $2f + 1$ storage servers, where f is the number of faults that can be tolerated by one replica group. Transactions may read and modify data that spans multiple shards; a distributed transaction protocol ensures that transactions are applied consistently to all shards.

We consider system architectures that layer the distributed transaction protocol atop the replication protocol, as shown in Figure 1. In the context of Agrawal et al.'s recent taxonomy of partitioned replicated databases [2], these are *replicated object systems*. This is the traditional system architecture, used by several influential systems including Spanner [6], MDCC [11], Scatter [9] and Granola [7]. However, other architectures have been proposed that run the distributed transaction protocol within each site, and use replication across sites [17].

Our analysis does not consider the cost of disk writes. This may seem an unconventional choice, as synchronous writes to disk are the traditional way to achieve durability in storage systems – and a major source of their overhead. Rather, we consider architectures where durability is achieved using in-memory replication, combined with asynchronous checkpoints to disk. Several recent systems (e.g., RAMCloud [20], H-Store [21]) have adopted this approach, which offers two advantages over disk. First, it ensures that the data remains continuously available when one system fails, minimizing downtime. Second, recording an operation in memory on multiple machines can have far lower latency than synchronously writing to disk, while still tolerating common failures.

We assume that clients are application servers that access data stored in the system. Clients have access to a directory of storage servers and are able to directly map data to storage servers, using a technique like consistent hashing [10].

Transaction Model We assume a general transaction model. Clients begin a transaction, then execute operations during the transaction's *execution period*. During this period, the client is free to abort the transaction. Once the client finishes execution, it can commit the transaction, atomically and durably committing the executed operations to the storage servers.

2.2 Standard Protocols

The standard architecture for these distributed storage systems, shown in Figure 1 is layering an atomic commitment protocol, like two-phase commit, and a concurrency control mechanism, like strict two-phase locking, atop a consensus-based replication protocol, like Paxos. We briefly review each protocol.

Atomic Commitment Two-Phase Commit (2PC) is the standard atomic commit protocol; it provides all-or-nothing agreement to a single operation across a number of machines, even in the presence of failures. Distributed storage systems use 2PC to atomically commit transactions involving operations at different storage servers.

2PC achieves atomicity by having participants first *prepare* to commit the transaction, then wait until they hear a commit or abort decision from the coordinator to finish the transaction. To maintain correctness in the presence of participant or coordinator failures, 2PC requires three durable writes to an on-disk log: on prepare and commit at the participants and at the commit decision point on the coordinator.

Concurrency Control In order to support concurrent transactions at participants, distributed storage systems typically integrate 2PC with a concurrency control mechanism. Concurrency control mechanisms enable partic-

Table 1: Comparison of read-write transaction protocols in replicated transactional storage systems.

Transaction System	Replication Protocol	Read Latency	Commit Latency	Msg At Bottleneck	Isolation Level	Transaction Model
Spanner [6]	Multi-Paxos [13]	2 (leader)	4	$2n$ + reads	Strict Serializable	Interactive
MDCC [11]	Gen. Paxos [14]	2 (any)	3	$2n$	Read-Committed	Interactive
Repl. Commit [17]	Paxos [13]	$2n$	4	2	Serializable	Interactive
CLOCC [1, 16]	VR [19]	2 (any)	4	$2n$	Serializable	Interactive
Lynx [25]	Chain Repl. [22]	–	$2n$	2	Serializable	Stored procedure
TAPIR [23, 24]	Inconsistent Rep.	2 (to any)	2	2	Strict Serializable	Interactive

ipants to prepare for concurrent transactions as long as they do not violate linearizable ordering. Concurrency control mechanisms can be either pessimistic or optimistic; a standard pessimistic mechanism is strict two-phase locking (S2PL), while optimistic concurrency control [12] (OCC) is a popular optimistic mechanism. S2PL depends on durable log writes to ensure that locks persist in the presence of participant failures. OCC relies on writes to log to keep the list of accepted (prepared or committed) transactions, in order to check optimistically executed transactions for conflicts.

Consistent Replication Distributed systems have widely adopted consensus-based replication protocols like Paxos [13] or Viewstamped Replication [19]. Replication protocols execute operations at a number of replicas to ensure that the effects of successful operations persist even after a fraction of the replicas have failed. To achieve strict single-copy consistency across replicas, replicas coordinate to ensure a single order of operations across all replicas.

Distributed storage systems use these protocols to provide consistent replicated logs for two-phase commit and concurrency control. These replication protocols are often used to replace disk-based logs for durable storage; however, as we will demonstrate, they provide guarantees in a different way from disks: in a distributed system, it is possible to order operations without making them durable, and vice versa.

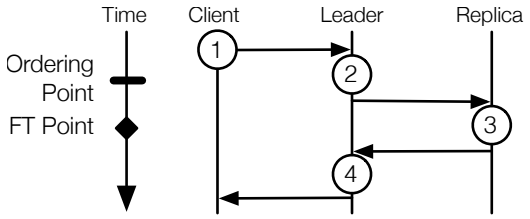
Examples Table 1 gives examples of transactional storage systems and describes the replication protocols they use in order to support particular transaction models. The table compares these systems based on the latency they require to complete a read or commit a transaction, as well as the number of messages processed by a bottleneck replica – typically the determining factor for replication system throughput.

3 Separating Ordering and Fault-Tolerance in Distributed Protocols

Traditional distributed storage systems use disk-based logs to provide two guarantees: ordering and fault tolerance. These two concepts are fundamentally intertwined in a disk-based log: the order in which operations are written to disk, in addition to marking the point at which an operation becomes fault-tolerant, defines a single global order of operations.

Modern distributed storage systems replace disk-based logs with consensus-based replication protocols to provide the same guarantees. The key difference in replicated systems is that ordering and fault tolerance each impose different costs. We argue that, as a result, they *should be separated* whenever possible. Therefore, it is important to distinguish between the following operations:

- **Logged operations** are guaranteed to be both ordered and fault-tolerant, and can be provided with a consensus-based replication protocol.
- **Fault-tolerant (FT) operations** are guaranteed to be fault-tolerant, and can be provided by quorum writes to $f + 1$ replicas.



1. Client sends an operation to the leader.
2. Leader orders the operation in the serial ordering and sends the operation to the replicas in serial order.
3. Replica records or executes the operation.
4. Leader collects responses from at least f replicas.

Figure 2: **Viewstamped Replication (VR) protocol.** The *ordering point* (black line) indicates the point in the algorithm where the ordering of the log operation is determined and the *FT point* (black diamond) is the point where the operation has been replicated and can be considered fault-tolerant. In VR, step 2 at the leader provides ordering, and step 3 at the other replicas provide durability.

- **Ordered operations** are guaranteed to have a single serial ordering. These can be provided in a number of ways, including serialization through a single server or timestamps based on loosely synchronized clocks like Spanner.

Researchers have traditionally analyzed the complexity of distributed protocols using metrics like the number of messages processed by each replica or the number of message delays per operation. We conduct our analysis by studying when logged, ordered and FT operations are required, as this ultimately gives greater insight into how to co-design the layered protocols used in distributed transactional storage systems.

At the same time, the number of logged, ordered and FT operations in each protocol gives insight into the basic complexity (i.e., message delays) and performance. FT operations require a single round-trip to multiple replicas. Ordered operations vary in cost, depending on how the ordering is provided. Logged operations are the most expensive because they provide both ordering and fault tolerance.

Two-phase commit and concurrency control mechanisms were designed for disk-based logs so they use logged operations for fault tolerance. In the following two sections, we analyze whether logged operations in these protocols require ordering or can be replaced with the cheaper FT operations.

3.1 Analysis of Standard Protocols

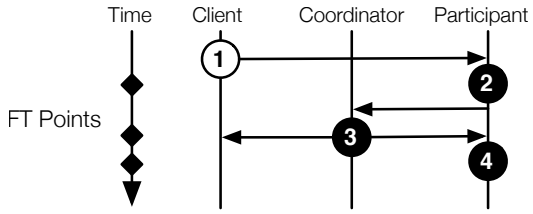
Consistent Replication We first briefly analyze the basic replication protocol used to provide logged operations. The commonly-used consensus protocols, like Multi-Paxos or VR, are leader-based. In the common, non-failure case, shown in Figure 2, these protocols provide consensus in two steps: the first step provides ordering, while the second ensures fault tolerance. Essentially, these protocols provided logged operations by combining an ordered operation followed by an FT operation. The ordered operation is expensive; it is provided by serialization through the leader, incurring a round-trip and leading to a bottleneck at the leader.

Previous work has noted the cost of providing ordering in consensus-based replication protocols. Our following analysis is orthogonal and complementary to recent work [18, 14, 15] on providing replication protocols where *some* of the operations do not require ordering.

3.1.1 Distributed Transactions

2PC uses logged operations to the replication layer in three places, shown as black numbered circles in Figure 3. The operations need to be fault-tolerant but not necessarily ordered.

To ensure that prepared transactions are not lost due to failures at the participants, the prepare operation must be an FT operation. The commit/abort operation must be an FT operation as well; it requires durability to move the participant out of the prepared state and to persist the effects of the transaction. At the coordinator, the commit/abort decision must be an FT operation to ensures that, if there is a failure at the coordinator, the



1. Client sends prepare to participants.
2. Participant prepares to commit transaction and responds to coordinator.
3. Once all participants respond, coordinator makes the commit decision and sends decision to participants and client.
4. Participant commits/aborts the transaction.

Figure 3: **Two-phase Commit (2PC) protocol**. Participants are servers involved in the transaction. Logged operations are marked as black circles with a white operation number. The blocking period represents the period when participants block on a decision from the coordinator.

participants can still find the outcome of the transactions[1]. Otherwise, participants could be blocked in the prepared state forever.

Whether the 2PC operations require ordering, besides fault tolerance, depends on whether the underlying concurrency control mechanism relies on ordered prepare and commit operations to maintain transaction ordering. In the next subsection we further analyze the ordering requirements for 2PC operations when integrating different concurrency control mechanisms.

Observation: 2PC requires at least 3 FT operations, but ordering depends on the underlying concurrency control mechanism.

3.1.2 Distributed Concurrency Control

While 2PC relies on replication for durability, the concurrency control mechanism is the application being replicated. Thus, its requirements dictate the ordering guarantees needed from the replication layer.

The number of logged operations remains the same in the integrated 2PC/concurrency control protocol. Concurrency control mechanisms do not require additional logged operations during the execution period because the transaction can always abort on failure. However, during 2PC, the concurrency control state must be logged on prepare and commit.

Different concurrency control mechanisms have different requirements for the replication layer. For S2PL, shown in Figure 4, acquiring locks during the execution phase is not a logged operation, however, *it is an ordered operation*. This ordered operation may be handled by the replication layer; a common way to implement distributed S2PL is to keep the locks at the leader of the replica group and send all operations to the leader during the execution period. The prepare operation for S2PL/2PC does not depend on ordering, so it only needs an FT operation for durability. The commit operation releases locks, so it must be an ordered and FT operation.

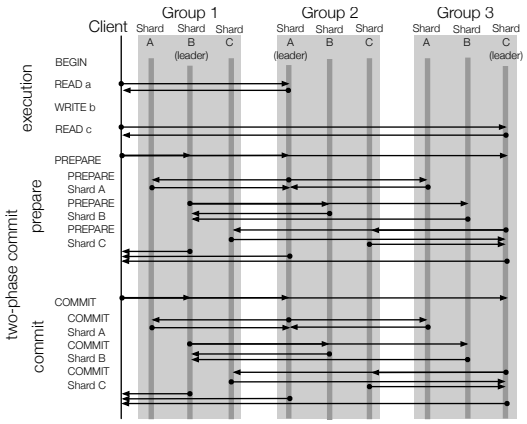
OCC/2PC, shown in Figure 5, has a different set of requirements. During execution, there are no ordered operations. Thus, distributed OCC can send reads to any replica and support buffered writes at the client[2]. Instead, OCC/2PC's prepare operation must be an ordered and FT operation. OCC requires ordering on prepare because it checks for conflicts against previously accepted (prepared or committed) transactions. The commit operations do not need ordering, but aborts do because they affect conflict checks.

S2PL and OCC maintain consistency in fundamentally different ways, therefore, they have different ordering requirements. At the participants, S2PL requires one ordered operation per lock, one FT operation and one logged operation, while OCC requires two logged operations. The coordinator still may require either an FT or a logged operation, depending on how it makes the commit/abort decision.

Observation: Optimistic concurrency control mechanisms limit the number of ordered operations.

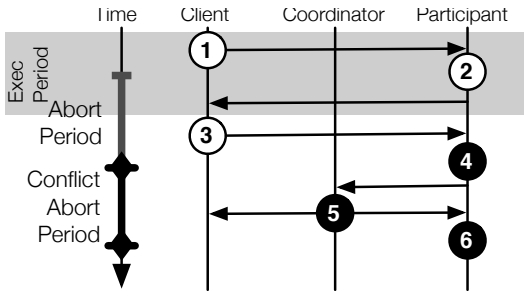
[1] If the coordinator cannot vote to abort after a successful prepare, then the coordinator does not require a logged operation

[2] A common optimization, used by Spanner [6], is to buffer writes for S2PL as well and acquire write locks on prepare. Then, the prepare for S2PL/2PC would also require ordering.



1. Client sends reads/writes to participant.
2. Participant acquires lock (and returns the value).
3. Client sends prepare to all participants.
4. Participant records locks.
5. Once all participants respond, coordinator makes the commit decision and sends decision to participants and client.
6. Participant commits the transaction and releases locks.

Figure 4: **2PC with Strict Two-Phase Locking (S2PL)**. The execution period represents when client runs the transaction, sending reads and writes to the server. With S2PL, locks block other transactions until they are released, so acquiring the lock is an ordered operation. That makes the prepare an FT operation, while the commit/abort decision at the coordinator and the commit/abort at the participants is a logged operation.



1. Client sends reads to participant.
2. Participant returns the current version.
3. Client sends prepare to all participants.
4. Participant runs OCC validation checks.
5. Once all participants respond, coordinator makes commit decision and sends decision to participants and client.
6. Participant commits the transaction and removes transaction from prepared list.

Figure 5: **2PC with Optimistic Concurrency Control (OCC)**. Instead of blocking other transactions, OCC executes optimistically, then checks for conflicts on prepare. This makes the prepare a logged operation, along with the coordinator commit/abort decision and the commit/abort at the participants.

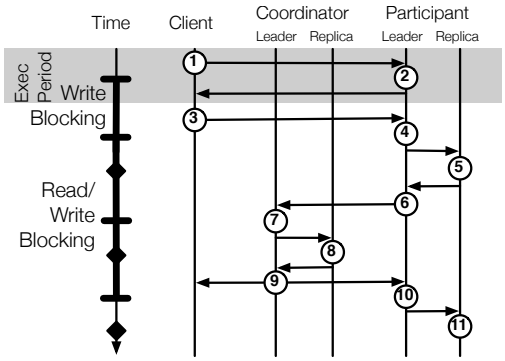
3.1.3 The Integrated Transaction Protocol

In this section, we combine 2PC and concurrency control with consistent replication into a full, integrated protocol. This integration allows us to see the consequences of requirements in the 2PC/concurrency control protocol on the replication protocol.

For simplicity, we only analyze the pessimistic, S2PL/2PC/VR protocol shown in Figure 6. We add a common optimization that Spanner uses to the protocol from the last section. We buffer writes at the client and only acquire write locks on prepare, making the prepare operation an ordered (and FT) operation. Altogether, we believe this protocol represents the typical way a distributed storage system today might provide replicated, distributed transactions.

There are a large number of ordered operations in this protocol. Reads during execution, prepares, and commits at the coordinator and the participants, all require ordered operations. These represent extra network delays and throughput bottlenecks. The protocol also has 3 FT operations.

Observation: The standard integration of S2PL/2PC/VR requires a large number of ordered (and FT) operations to provide transactional consistency.



1. Client sends reads to participant leader.
2. Participant leader acquires lock and returns value.
3. Client sends prepare to the each participant leaders.
4. Participant leader acquires write locks.
5. Participant replica acquires read and write locks.
6. Participant leader responds to coordinator.
7. Once participant leaders respond, coordinator makes commit decision.
8. Coordinator replica commits.
9. Coordinator sends commit to participant leaders and client.
10. Participant leader commits and releases locks.
11. Participant replica commits and releases locks.

Figure 6: **S2PL/VR/2PC**. We add the write-buffering optimization that Spanner uses, where write locks are acquired on prepare. Together, we believe this integrated protocol represents the typical way to provide distributed replicated transactions.

3.2 Analysis of Spanner

The recent Spanner protocol, shown in Figure 7, is Google's solution to replicated, distributed transactions. Its key contribution is TrueTime, which uses atomic clocks to provide loosely synchronized clocks with error bounds. Spanner uses TrueTime to provide ordering for read-only transactions without acquiring locks. Since read-only transactions require only ordering, and not durability, using TrueTime for ordering eliminates the need to serialize read-only transactions through a single server.

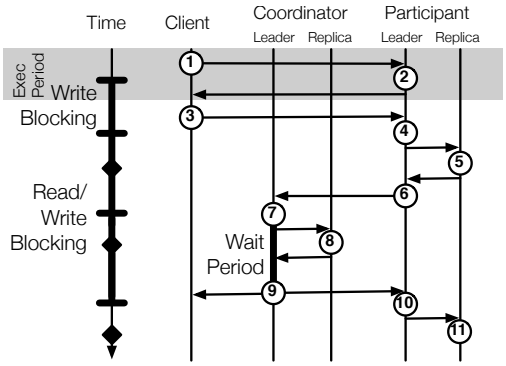
Spanner avoids serializing read-only transactions through a single server because it incurs an extra round-trip and the server can become a bottleneck. Systems that rely only on the replication layer to provide ordering, like MegaStore [3], *always incur these overheads,*. Spanner demonstrates that there are other ways to provide ordering in a distributed system that can be more effective in some cases. Of course, there is a trade-off. TrueTime requires waits and only provides consistency as long as clock skews are within error bounds.

To support read-only transactions using TrueTime, Spanner must serialize each read/write transaction at a single timestamp. While Spanner makes effective use of TrueTime for read-only transactions, it uses the standard S2PL/2PC/VR protocol for read-write transactions, with a wait on commit to accommodate clock skew. Spanner still uses a leader-based Paxos protocol and serialization through the leader for locking. Thus, for each read-write transaction, Spanner uses the same amount of distributed coordination as standard S2PL/2PC/VR to *make a single ordering decision.*

Observation: Spanner requires a large number of ordered (and FT) operations to order each transaction at a single timestamp.

3.2.1 Optimistic Spanner

As an example of how we can move and eliminate ordered operations in Spanner, we propose an alternative Spanner protocol, shown in Figure 8. This protocol has two changes from the basic Spanner protocol. First, we switch Spanner to OCC, which uses fewer ordered operations, but may have to abort. Next, we move the coordinator from one of the participants to the client, which we assume to be an application server. With these changes, we can eliminate ordered operations during execution, to allow reads from any replica for read-write transactions, and eliminate a ordered and replicated operation at the coordinator.



1. Client sends reads to participant leader.
2. Participant leader acquires lock and returns value.
3. Client sends prepare to participant leaders.
4. Participant leaders acquires write locks and select prepare timestamp.
5. Participant replicas acquire locks and record prepare timestamp.
6. Participant leaders respond with prepare timestamp.
7. Once participant leaders respond, coordinator selects a commit timestamp by taking the max of the prepare timestamps and its own local time.
8. Coordinator replicas commit at commit timestamp.
9. After waiting for the uncertainty bound, coordinator sends commit with commit timestamp to participant leaders and client.
10. Participant leaders commit at commit timestamp, and release locks.
11. Participant replicas commit at commit timestamp, and release locks.

Figure 7: **Spanner Commit Protocol.** This protocol is very similar to the S2PL/VR/2PC algorithm. The key difference is the use of timestamps from roughly synchronized clocks (TrueTime) and wait period (of double the uncertainty bound), which enables linearizable read-only transactions without locking or 2PC.

Spanner's use of TrueTime lends itself well to OCC. Like Spanner and unlike locking, OCC orders each transaction at a single timestamp, supplied by the coordinator on commit. In a distributed system, selecting a globally-relevant commit timestamp can be tricky; however, Spanner's commit timestamps work perfectly as OCC timestamps.

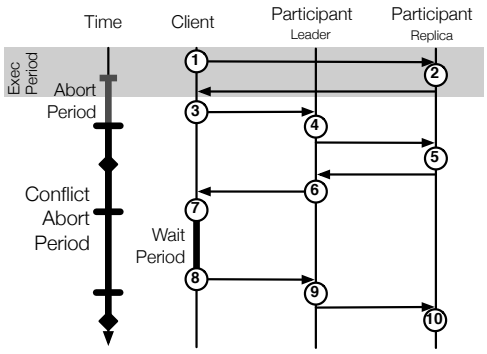
Observation: Ordered operations can be moved and eliminated in some cases while still maintaining transactional consistency.

4 Inconsistent Replication and TAPIR

Our analysis above shows that existing protocols require multiple ordered and fault tolerant operations to commit transactions in a replicated, partitioned database system. We argue that the inability to separate fault-tolerant and ordered operations is a major cause of wasted work in these cases, because extra protocol steps and increased blocking result in worse latency and throughput. Motivated by this observation, we recently designed a new transaction system based on the idea of co-designing the transaction coordination and replication protocols to support efficient, unordered operations [23, 24].

As part of this work, we have designed a new replication protocol, *inconsistent replication (IR)* that treats fault-tolerant and ordered operations separately. IR is not intended to be used by itself; rather, it is 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. It does so using two classes of operations:

- **inconsistent** operations are fault-tolerant but not ordered: successful operations persist across failures, but operations can execute in any order.
- **consensus** operations are allowed to execute in any order, but return a single *consensus result*. Successful operations and their consensus results persist across failures.



1. Client sends reads to participant replica.
2. Participant replica returns latest version.
3. Client sends prepare to participant leaders.
4. Participant leaders run OCC validation checks and select prepare timestamps.
5. Participant replicas record prepared transaction and timestamps.
6. Participant leaders respond to client with prepare timestamps.
7. Once participants respond, client selects a commit timestamp.
8. After waiting for the uncertainty bound, client sends commit with commit timestamp to participant leaders.
9. Participant leaders commit at commit timestamp, and remove transaction from list of prepared transactions.
10. Participant replicas commit at commit timestamp.

Figure 8: **Optimistic Spanner.** We switch the Spanner protocol to OCC instead of S2PL and eliminate the coordinator. The switch to OCC enables reads to be served by any replica. If the replica is not up-to-date, the transaction will abort during the prepare phase. Eliminating the coordinator lets the client pick (and know) the commit timestamp earlier, but leaves the client with nothing to do while waiting out the uncertainty.

Both types of operations are efficient: **inconsistent** operations complete in one round trip without coordination between replicas, as do **consensus** operations when replicas agree on their results (the common case). If replicas disagree on the result of a **consensus** operation, the application protocol on the client is responsible for *deciding* the outcome of the operation in an application-specific way.

TAPIR is designed to be layered atop IR in a replicated, transactional storage system. Figure 9 demonstrates how coordination in TAPIR works using the same transaction as Figure 6. TAPIR obtains greater performance because IR does not require any leaders or centralized coordination.

As we noted in our discussion of Spanner above, using optimistic concurrency control instead of locking makes it possible to reduce the number of ordered operations. TAPIR uses this approach, and takes it further using *optimistic timestamp ordering*. The client selects a timestamp using its local clock, and proposes that as the transaction's timestamp in its prepare operation. Participant replicas accept the transaction's prepare only if both of two conditions hold: they have not processed any transactions with a higher timestamp, and the transaction passes an OCC validation check with all previously prepared transactions. This reduces the number of ordered operations to one.

Realizing this technique requires careful protocol design. In particular, TAPIR must be able to handle inconsistent results from its replication layer, select timestamps in a way that ensures progress, and tolerate failures of client-coordinators. We do not attempt to describe these protocols here in detail; the interested reader is referred to our recent SOSP paper [23] and its accompanying technical report [24].

5 Conclusion

Many partitioned replicated databases treat the replicated storage much as they might use a traditional disk: as a persistent, ordered log. Unlike a disk, however, replicated distributed systems are able to achieve fault tolerance without ordering operations (and vice versa). We have presented a framework for analyzing transaction coordination protocols in terms of the number of fault tolerant and ordered operations, and argue that separating them in this way provides insight into the fundamental costs of different approaches like two-phase locking and optimistic concurrency control in distributed systems. As a concrete example, we discuss our recent work on