Distributed Transactions in Spanner 2

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 21

Marco Canini

Credits: Michael Freedman and Kyle Jamieson developed much of the original material.
Contents adapted from Haonan Lu, Wyatt Lloyd.

Recap: Spanner is Strictly Serializable

- Efficient read-only transactions in strictly serializable systems
  - Strict serializability is desirable but costly!
  - Reads are prevalent! (340x more than write txns)
  - Efficient rotxns → good overall performance

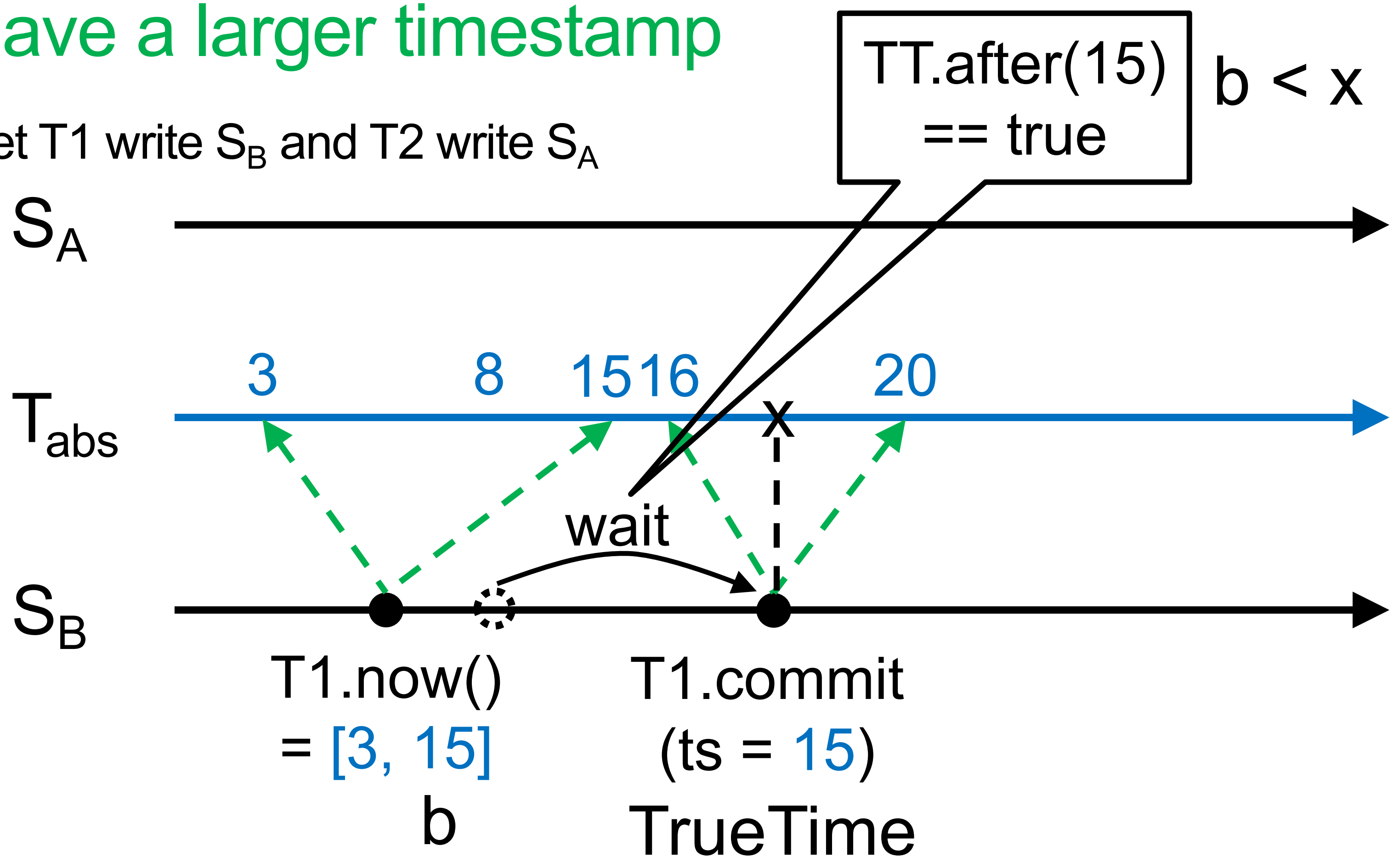
Recap: TrueTime

- Timestamping writes must enforce the invariant
  - If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp
- TrueTime: partially-synchronized clock abstraction
  - Bounded clock skew (uncertainty)
  - TT.now() → [earliest, latest]; earliest <= T_{abs} <= latest
  - Uncertainty (ε) is kept short
- TrueTime enforces the invariant by
  - Use at least TT.now().latest for timestamps
  - Commit wait

Enforcing the Invariant with TT

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

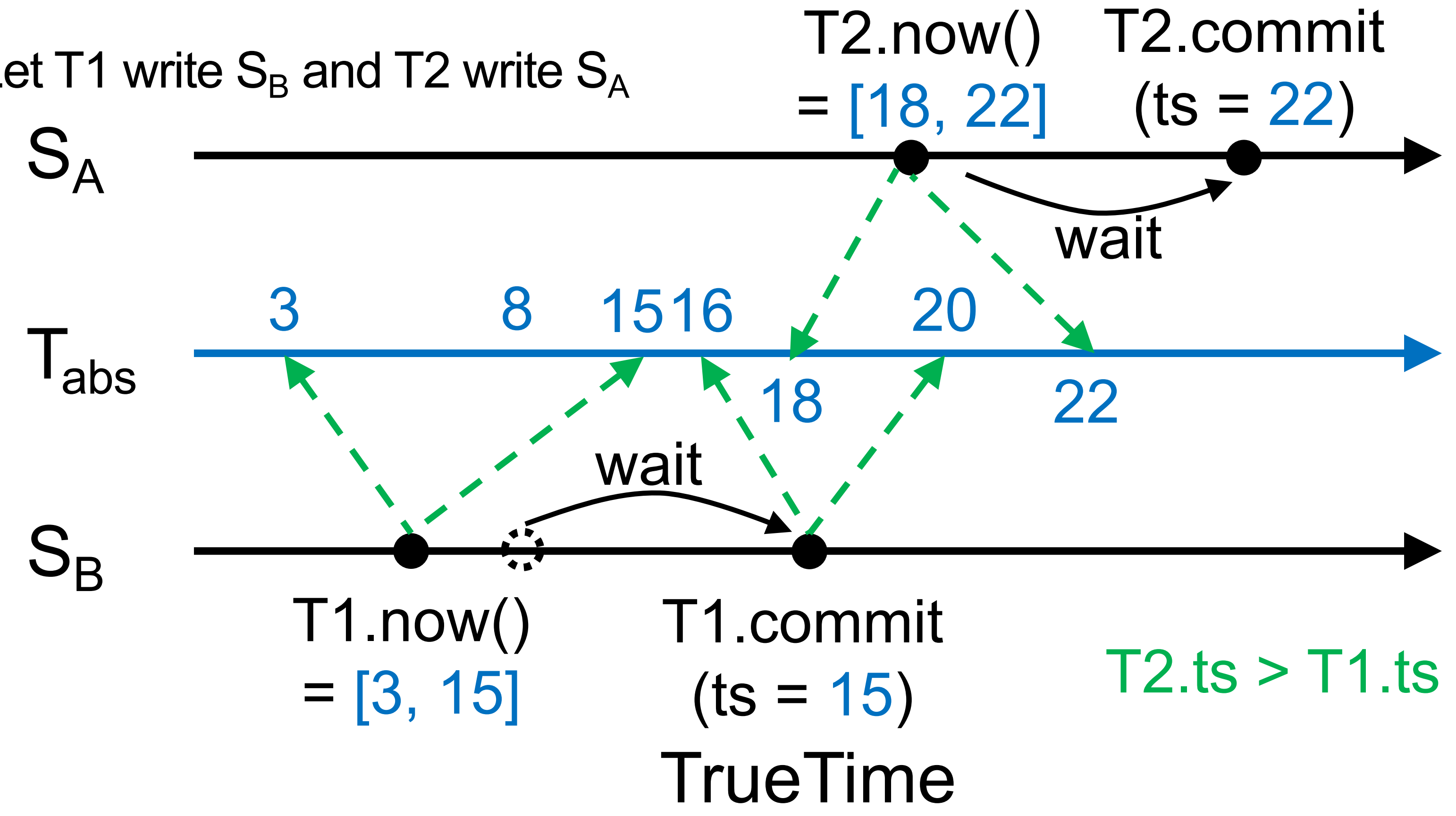
Let T1 write S_B and T2 write S_A



Enforcing the Invariant with TT

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Let T1 write S_B and T2 write S_A



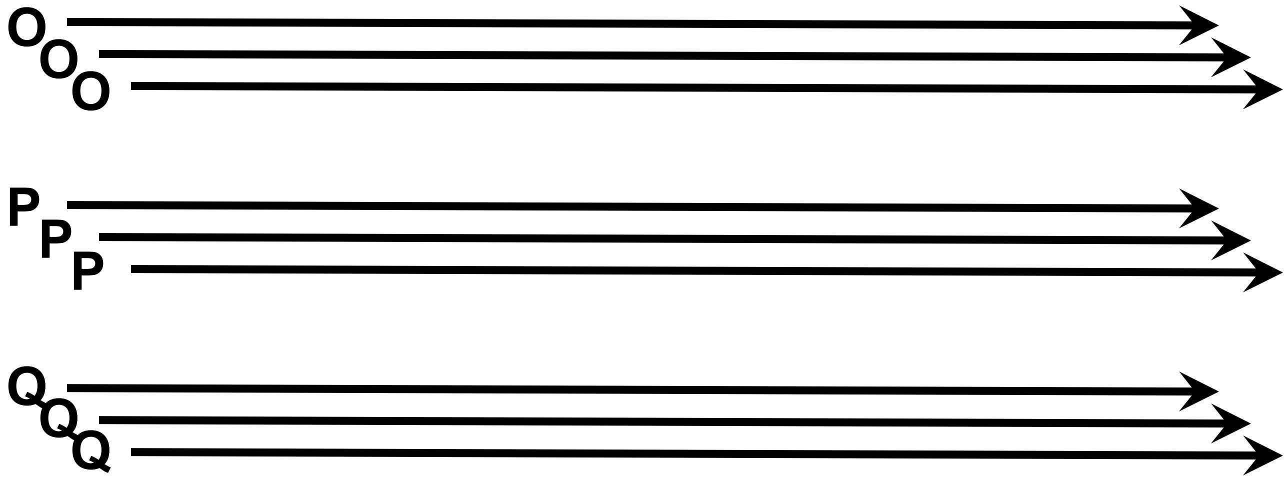
T2.ts > T1.ts

Strictly Serializable Multi-Shard Transactions

- How are clocks made “nearly perfect”?
  - TrueTime

- How does Spanner leverage these clocks?
  - How are writes done and tagged?
  - How read-only transactions are made efficient?

Scale-out vs. fault tolerance



Spanner mechanisms

- 2PL for concurrency control of read-write transactions
- 2PC for distributed transactions over tables
- (Multi)Paxos for replicating every tablet

This Lecture

- How write transactions are done
  - 2PL + 2PC (sometimes 2PL for short)
  - How they are timestamped
- How read-only transactions are done
  - How read timestamps are chosen
  - How reads are executed

Read-Write Transactions (2PL)

- Three phases

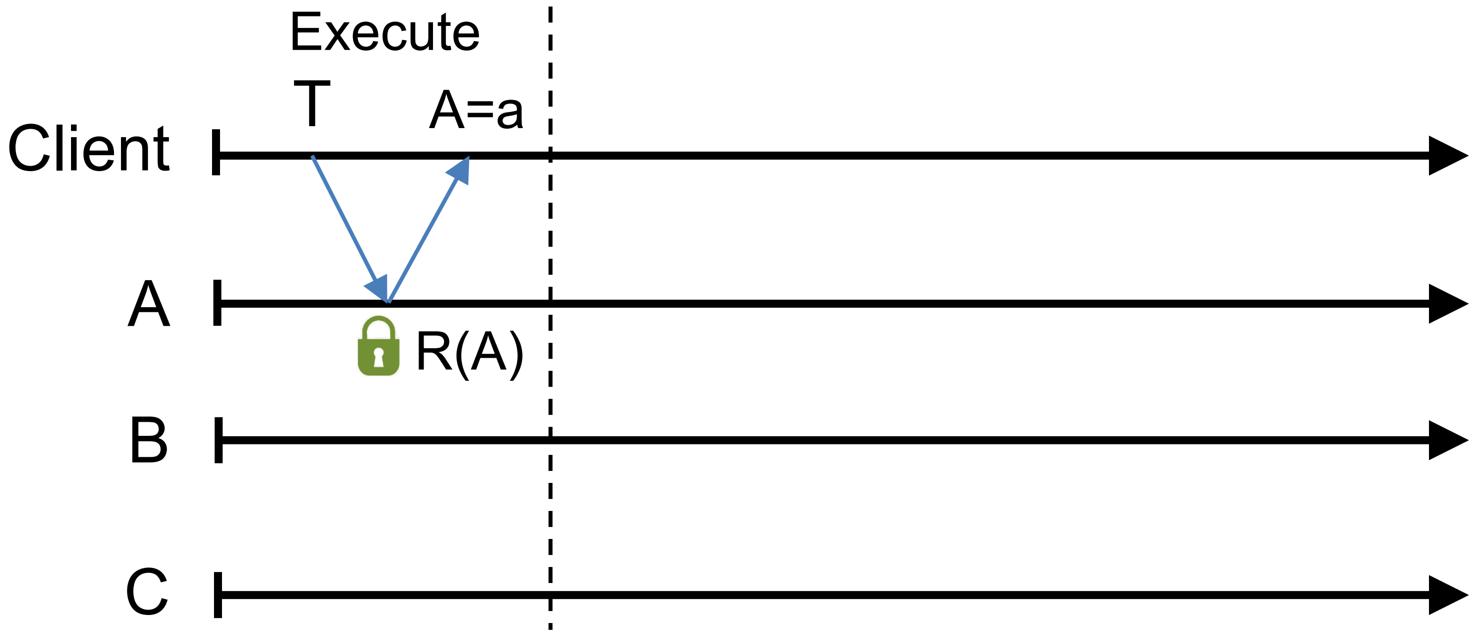
$$\text{Execute} \rightarrow \underbrace{\text{Prepare} \rightarrow \text{Commit}}_{\textbf{2PC: atomicity}}$$

Client-driven transactions (multi-shard)

Client: 2PL w/ 2PC

1. Issues reads to leader of each shard group, which acquires read locks and returns most recent data
2. Locally performs writes
3. Chooses coordinator from set of leaders, initiates commit
4. Sends commit message to each leader, include identity of coordinator and buffered writes
5. Waits for commit from coordinator

Read-Write Transactions (2PL)

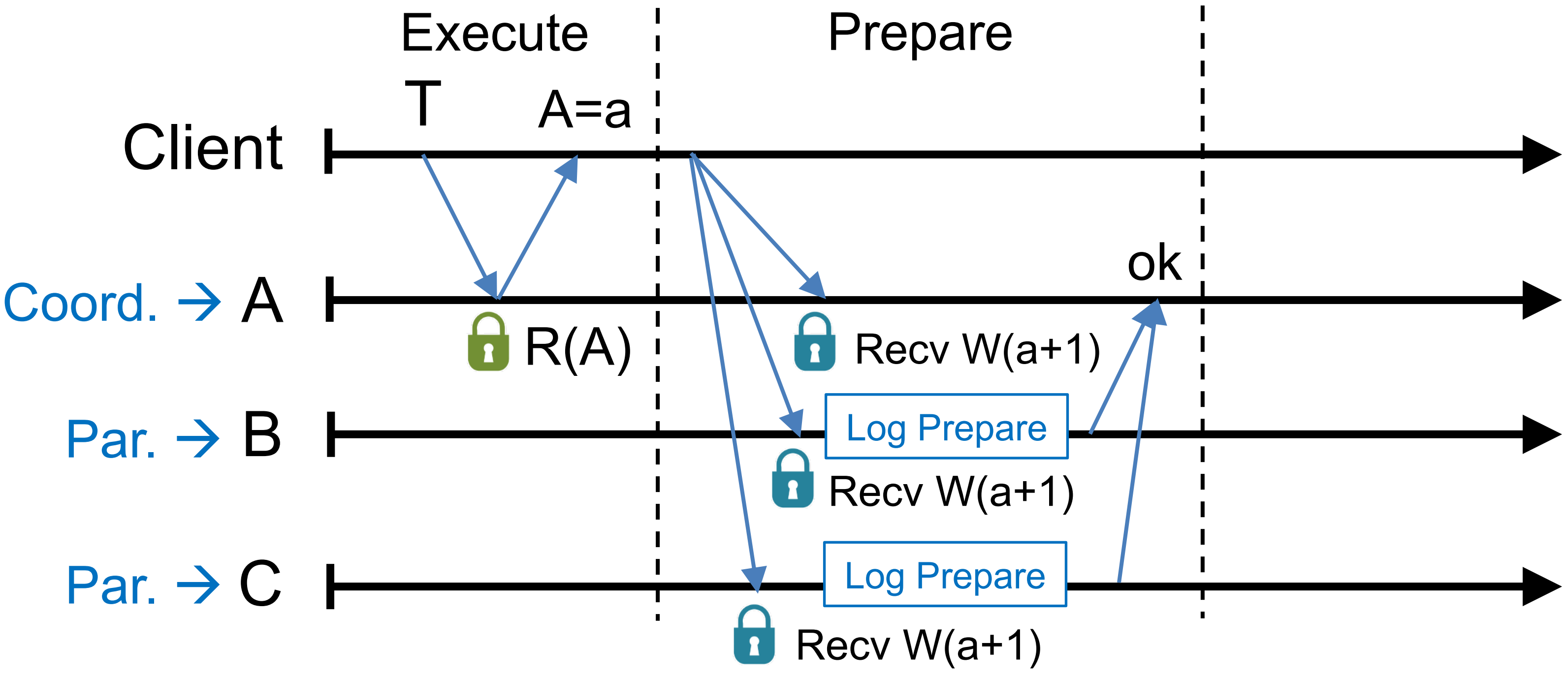


Txn T = {R(A=?), W(A=?+1), W(B=?+1), W(C=?+1)}

Execute:

- Does reads: grab read locks and return the most recent data, e.g., R(A=a)
- Client computes and buffers writes locally, e.g., A = a+1, B = a+1, C = a+1

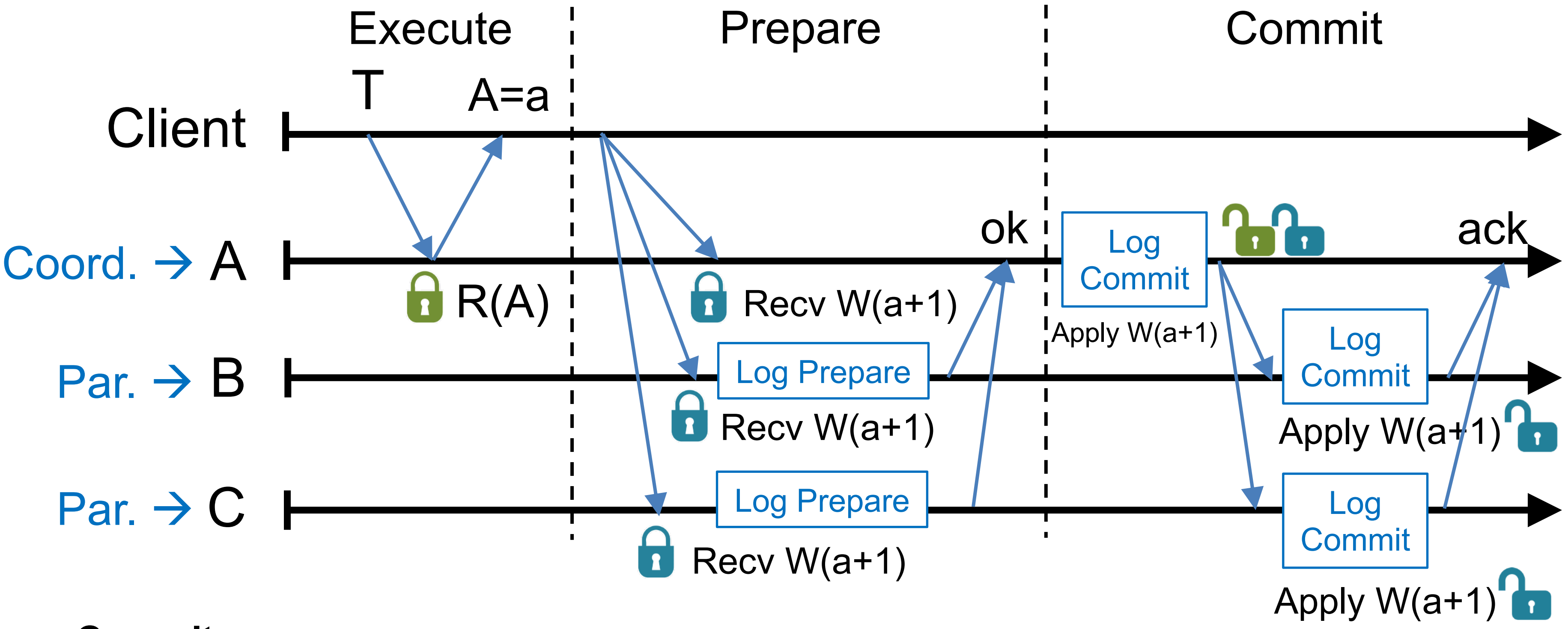
Read-Write Transactions (2PL)



Prepare:

- Choose a coordinator, e.g., A, others are participants
- Send buffered writes and the identity of the coordinator; grab write locks
- Each participant prepares T by logging a prepare record via Paxos with its replicas. Coord skips prepare (Paxos Logging)
- Participants send OK to the coord if lock grabbed and after Paxos logging is done

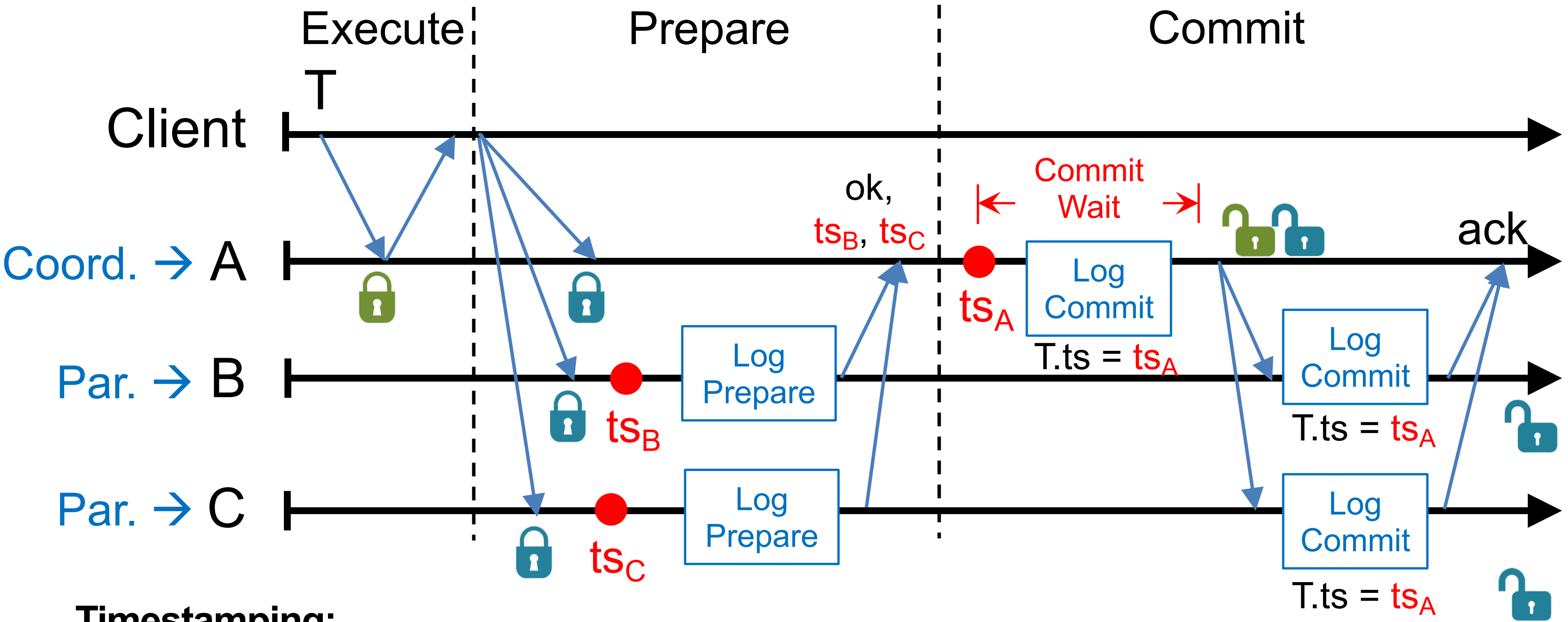
Read-Write Transactions (2PL)



Commit:

- After hearing from all participants, coord commits T if all OK; otherwise, abort T
- Coord logs a commit/abort record via Paxos, applies writes if commit, release all locks
- Coord sends commit/abort messages to participants
- Participants log commit/abort via Paxos, apply writes if commit, release locks
- Coord sends result to client either after its "log commit" or after ack

Timestamping Read-Write Transactions



Timestamping:

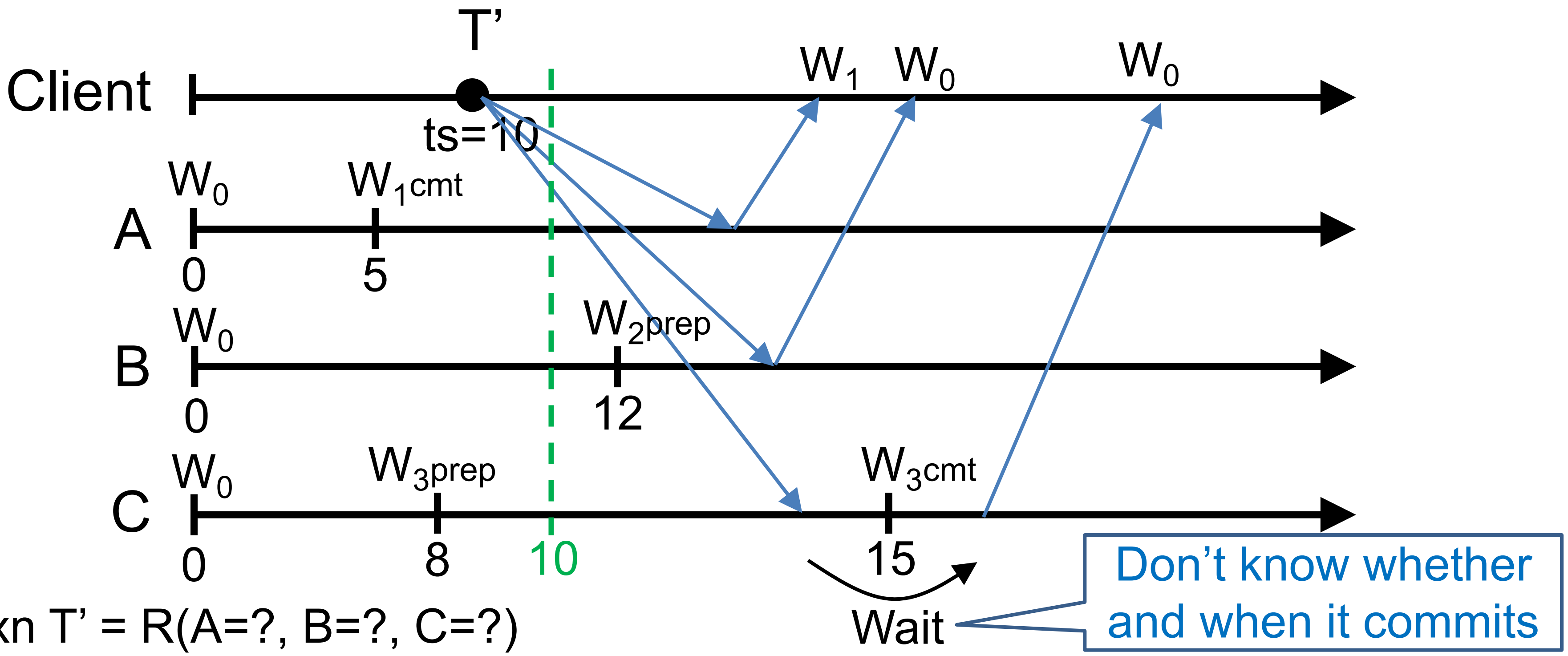
- Participant: choose a timestamp, e.g., ts_B and ts_C, larger than any writes it has applied
- Coordinator: choose a timestamp, e.g., ts_A, larger than
  - Any writes it has applied
  - Any timestamps proposed by the participants, e.g., ts_B and ts_C
  - Its current TT.now().latest
- Coord commit-waits: TT.after(ts_A) == true. Commit-wait overlaps with Paxos logging
- ts_A is T's commit timestamp

Ideas Behind Read-Only Txns

- Tag writes with physical timestamps upon commit
  - Write txns are strictly serializable, e.g., 2PL

- Read-only txns return the writes, whose commit timestamps precede the reads' current time
  - Rotxns are one-round, lock-free, and never abort

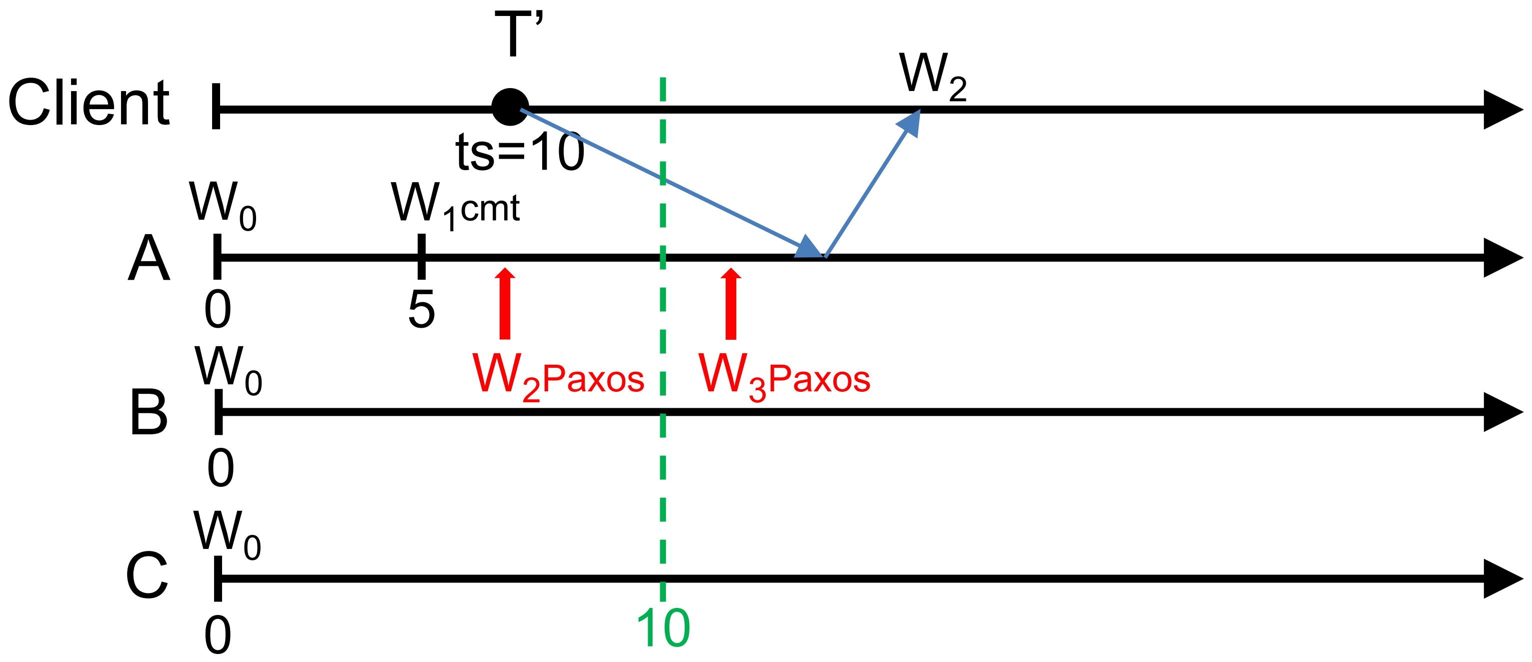
Read-Only Transactions (shards part)



Txn T' = R(A=?, B=?, C=?)

- Client chooses a read timestamp ts = TT.now().latest
- If no prepared write, return the preceding write, e.g., on A
- If write prepared with ts' > ts, no need to wait, proceed with read, e.g., on B
- If write prepared with ts' < ts, wait until write commits, e.g., on C

Read-Only Transactions (Paxos part)

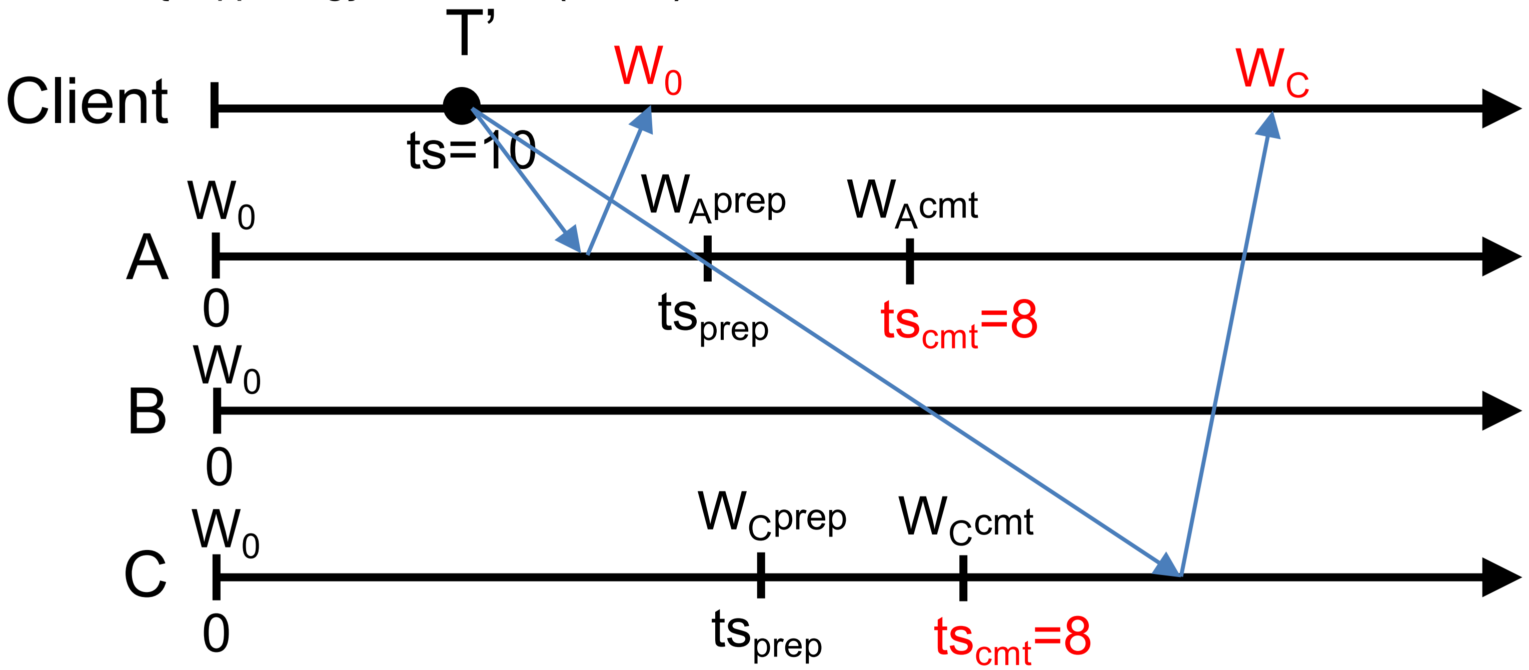


- Paxos writes are monotonic, e.g., writes with smaller timestamp must be applied earlier, W_2 is applied before W_3
- T' needs to wait until there exits a Paxos write with ts>10, e.g., W_3, so all writes before 10 are finalized
- Put it together: a shard can process a read at ts if ts <= t_{safe}
- $t_{safe} = \min(t_{safe}^{Paxos}, t_{safe}^{TM})$: before t_{safe}, all system states (writes) have finalized

A Puzzle to Help With Understanding

- What if no replication, only shards
  - Not in the paper, not realistic

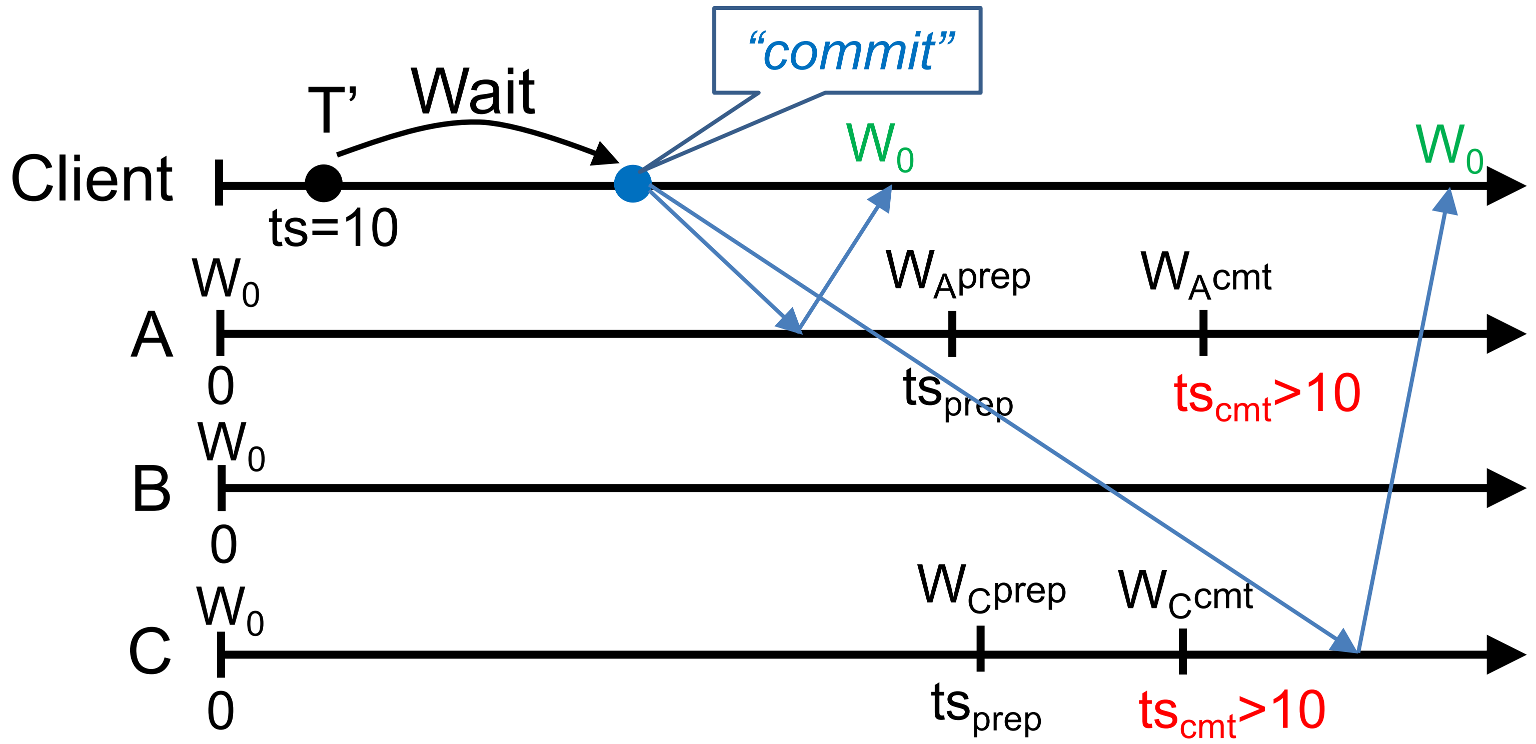
Txn T = {W_A, W_C}, T' = R (A, C)



T' sees partial effect of T, e.g., sees W_C but not W_A, and violates atomicity

A Puzzle to Help With Understanding

- Solution: uncertainty-wait



Uncertainty-wait ensures that ts_{cmt} must > readTS because

- W_1 starts after T' "commits," and
- T' waits out uncertainty before "commit", e.g., TT.after(10) == true

Serializable Snapshot Reads

- Client specifies a read timestamp way in the past
  - E.g., one hour ago
- Read shards at the stale timestamp
- Serializable
  - Old timestamp cannot ensure real-time order
- Better performance
  - No waiting in any cases
  - E.g., non-blocking, not just lock-free

Takeaway

- Strictly serializable (externally consistent)
  - Make it easy for developers to build apps!
- Reads dominant, make them efficient
  - One-round, lock-free
- TrueTime exposes clock uncertainty
  - Commit wait and at least TT.now.latest() for timestamps ensure real-time ordering
- Globally-distributed database
  - 2PL w/ 2PC over Paxos!