Spanner

Part II

COS 418: Distributed Systems
Lecture 18

Mike Freedman

Slides adapted from the Spanner OSDI talk

1

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 system overall performance

2

2

Recap: 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

3

3

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

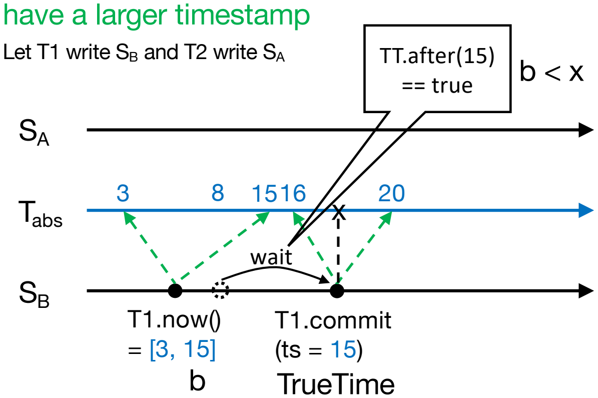
4

4

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



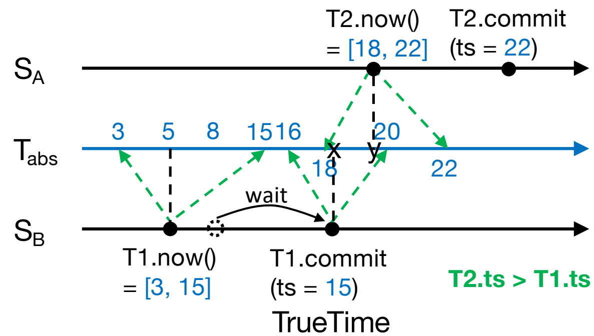
5

5

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

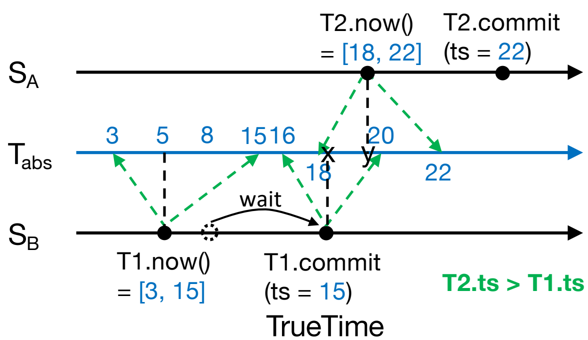


6

6

Enforcing the Invariant with TT

- What if T1.commit delayed, such that T2 happens after T1.now() but before T1.commit? Tricky as T1.commit.ts = T1.now().latest
  - Answer: T2 delayed until after T1 commits. Discussed later.



7

7

This Lecture

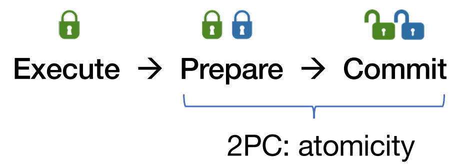
- How write transactions are done
  - 2PL + 2PC (sometimes 2PL for short)
  - How they are timestamped
- How read-only transactions are done
  - How read timestamps are chosen
  - How reads are executed

8

8

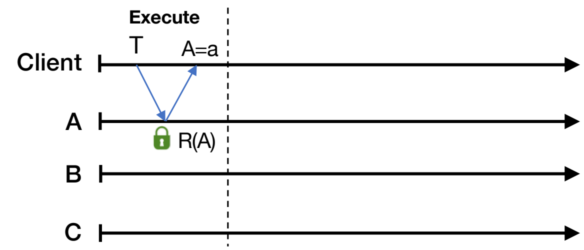
Read-Write Transactions (2PL)

- Three phases



9

Read-Write Transactions (2PL)



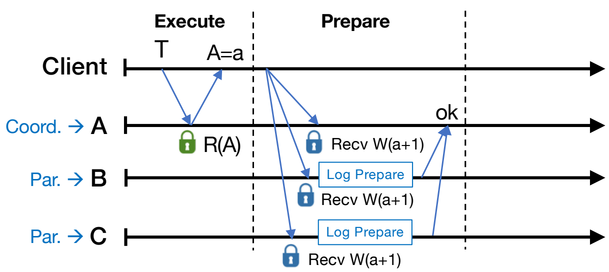
Txn T = {R(A=?), W(A=?+1), W(B=?+1), W(C=?+1)}

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

10

Read-Write Transactions (2PL)

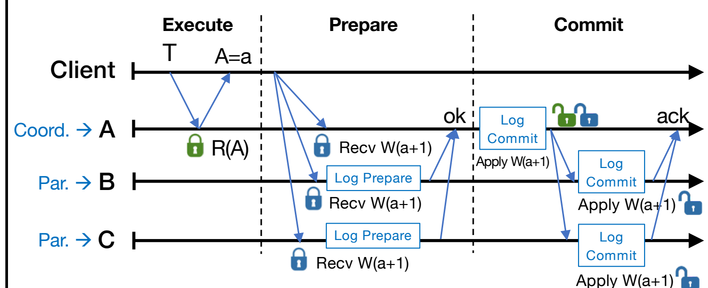


Prepare:

- 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 coord if lock grabbed and after Paxos logging is done

11

Read-Write Transactions (2PL)

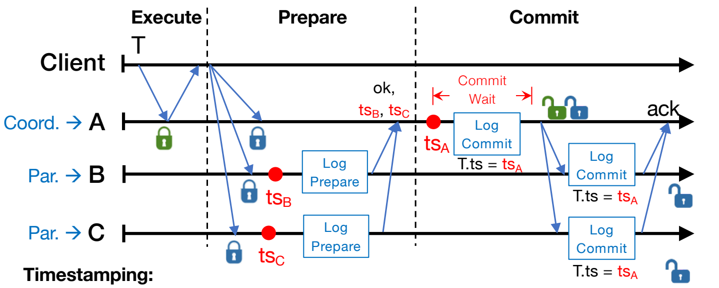


Commit:

- After hearing from all participants, coord commits T if all OK; o/w, abort T
- Coord logs commit/abort record via Paxos, applies writes if commit, release 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

12

Timestamping Read-Write Transactions



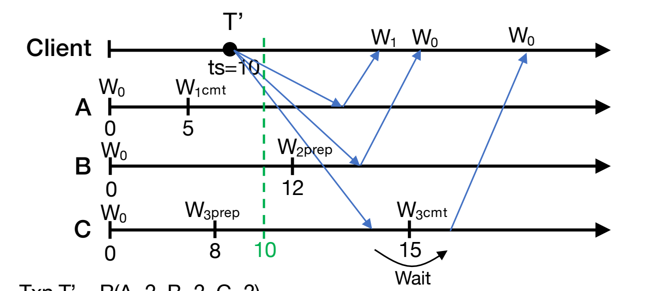
Timestamping:

- Participant: choose timestamp (eg, 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 w Paxos logging
  - ts_A is T's commit timestamp

13

13

Read-Only Transactions (shards part)



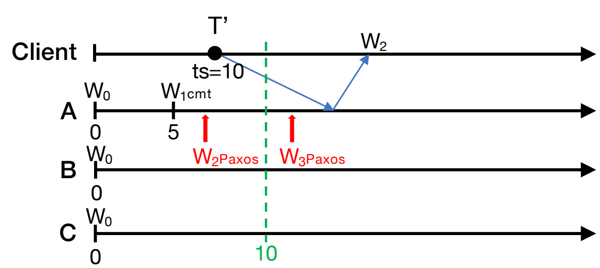
Txn T' = R(A=?, B=?, C=?)

- 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, eg, on B
- If write prepared with ts' < ts, wait until write commits, e.g., on C

14

14

Read-Only Transactions (Paxos part)

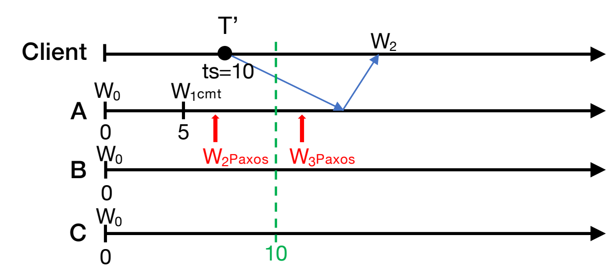


- Paxos writes are monotonic, e.g., writes with smaller timestamp must be applied earlier, W_2 is applied before W_3
- T' needs to wait until there exists a Paxos write with ts >10 (eg, W_3), so all writes before 10 are finalized
- Put it together: a shard can process a read at ts if ts <= t_{safe}
-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

15

15

Read-Only Transactions (Paxos part)



- A shard can process a read at ts if ts <= tsafe
- $t_{safe} = \min(t^{Paxos}_{safe}, t^{TM}_{safe})$: before t_{safe}, all system states (writes) have finalized
  - t^{Paxos}_{safe} = Timestamp of highest-applied Paxos write
  - t^{TM}_{safe} = infinity if zero prepared (but not commited) transactions
    Else, min of all prepare timestamps of any prepared txns

16

16

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
- Can have performance but still strictly serializable?
  - E.g., one-round, non-blocking, and strictly serializable
  - Coming in next lecture!

19

19

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!

20