Distributed Transactions in Spanner

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 17

Marco Canini

Recap: Distributed Storage Systems

- Concurrency control
  - Order transactions across shards
- State machine replication
  - Replicas of a shard apply transactions in the same order decided by concurrency control

Google’s Setting

- Dozens of zones (datacenters)
- Per zone, 100-1000s of servers
- Per server, 100-1000 partitions (tablets)
- Every tablet replicated for fault-tolerance (e.g., 5x)

Why Google built Spanner

2005 – BigTable [OSDI 2006]

- Eventually consistent across datacenters
- Lesson: “don’t need distributed transactions”

2008? – MegaStore [CIDR 2011]

- Strongly consistent across datacenters
- Option for distributed transactions
  - Performance was not great…

2011 – Spanner [OSDI 2012]

- Strictly Serializable Distributed Transactions
- “We wanted to make it easy for developers to build their applications”

A Deeper Look at Motivation
-- Performance-consistency tradeoff

- Strict serializability
  - Serializability + linearizability
  - As if coding on a single-threaded, transactionally isolated machine
  - Spanner calls it external consistency
- Strict serializability makes building correct application easier
- Strict serializability is expensive
  - Performance penalty in concurrency control + Replication
    - OCC/2PL: multiple round trips, locking, etc.

A Deeper Look at Motivation
-- Read-Only Transactions

- Transactions that only read data
  - Predeclared, i.e., developer uses READ_ONLY flag / interface
- Reads dominate real-world workloads
  - FB's TAO had 500 reads : 1 write [ATC 2013]
  - Google Ads (F1) on Spanner from 1? DC in 24h:
    - 31.2 M single-shard read-write transactions
    - 32.1 M multi-shard read-write transactions
    - 21.5 B read-only (~340 times more)
- Determines system overall performance

Can we design a strictly serializable, geo-replicated, sharded system with very fast (efficient) read-only transactions?

Before we get to Spanner …

- How would you design strictly serializable read-only transactions?
- 2PL (or OCC)
  - Multiple round trips and locking
- Can always read in local datacenters like COPS?
  - Maybe involved in Paxos agreement
  - Or must contact the leader
- Performance penalties
  - Round trips increase latency, especially in wide area
  - Distributed lock management is costly, e.g., deadlocks

Goal is to …

- Make read-only transactions efficient
  - One round trip
    - Could be wide-area
  - Lock-free
    - No deadlocks
    - Processing reads do not block writes, e.g., long-lived reads
  - Always succeed
    - Do not abort
- And strictly serializable

Leveraging the Notion of Time

- Strict serializability: a matter of real-time ordering
  - If txn T2 starts after T1 finishes, then T2 must be ordered after T1
    - If T2 is a ro-txn, then T2 should see the effects of all writes that finished before T2 started

Leveraging the Notion of Time

- Task 1: when committing a write, tag it with the current physical time
- Task 2: when reading the system, check which writes were committed before the time this read started
- How about the serializable requirement?
  - Physical time naturally gives a total order

Invariant:

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

Trivially provided by perfect clocks

Challenges

- Clocks are not perfect
  - Clock skew: some clocks are faster/slower
  - Clock skew may not be bounded
  - Clock skew may not be known a priori
- T2 may be tagged with a smaller timestamp than T1 due to T2’s slower clock
- Seems impossible to have perfect clocks in distributed systems. What can we do?

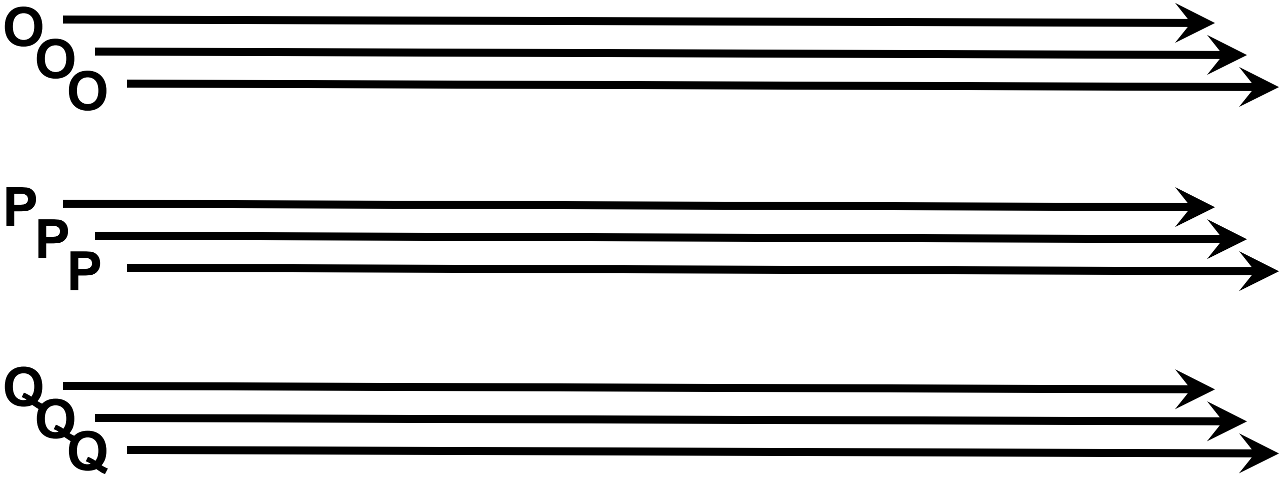
Nearly perfect clocks

- Partially synchronized
  - – Clock skew is bounded and known a priori
  - – My clock shows 1:30PM, then I know the absolute (real) time is in the range of 1:30 PM +/- X
    - e.g., between 1:20PM and 1:40PM if X = 10 mins
- Clock skew is short
  - – E.g., X = a few milliseconds
- Enable something special, e.g., Spanner!

Spanner: Google’s Globally-Distributed Database

OSDI 2012

Scale-out vs. fault tolerance



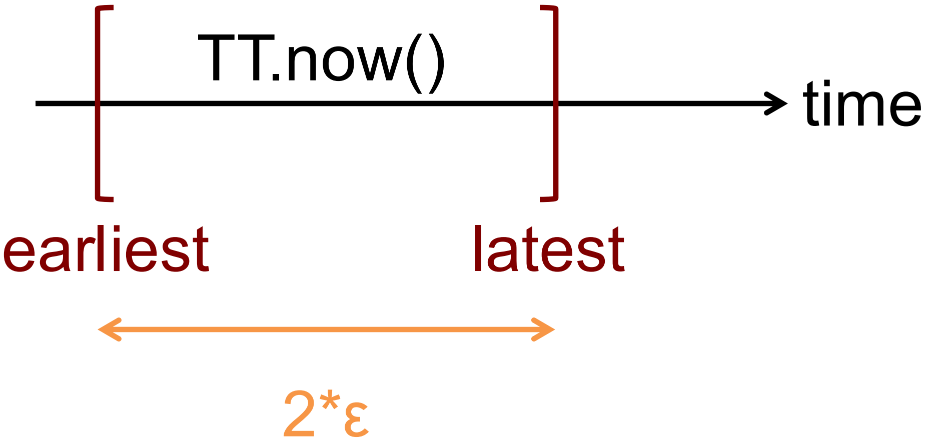
- Every tablet replicated via MultiPaxos
- So every “operation” within transactions across tablets actually is a replicated operation within Paxos RSM
- Paxos groups can stretch across datacenters!

Strictly Serializable Multi-Shard Transactions

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

TrueTime (TT)

- “Global wall-clock time” with bounded uncertainty
    - ε is worst-case clock divergence
    - Spanner’s time notion becomes intervals, not single values
    - ε is 4ms on average, 2 ε is about 10ms



Consider event e_{now} which invoked tt = TT.now():

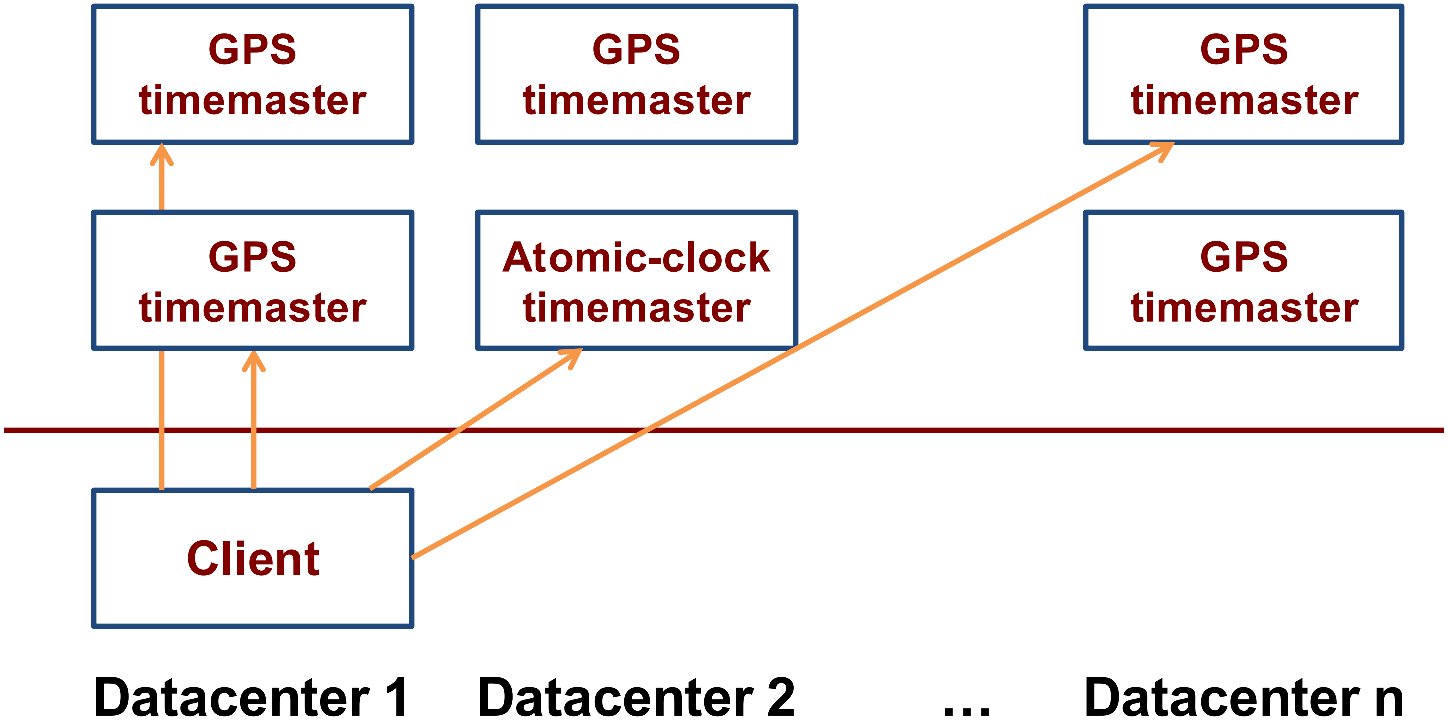
Guarantee: tt.earliest <= $t_{abs}(e_{now})$ <= tt.latest

TrueTime (TT)

- Interface
  - TT.now() = [earliest, latest] # latest – earliest = 2*ε
  - TT.after(t) = true if t has passed
    - TT.now().earliest > t (b/c t_{abs} >= TT.now().earliest)
  - TT.before(t) = true if t has not arrived
    - TT.now().latest < t (b/c t_{abs} <= TT.now().latest)

- Implementation
  - Relies on specialized hardware, e.g., GPS satellite and atomic clocks

TrueTime Architecture



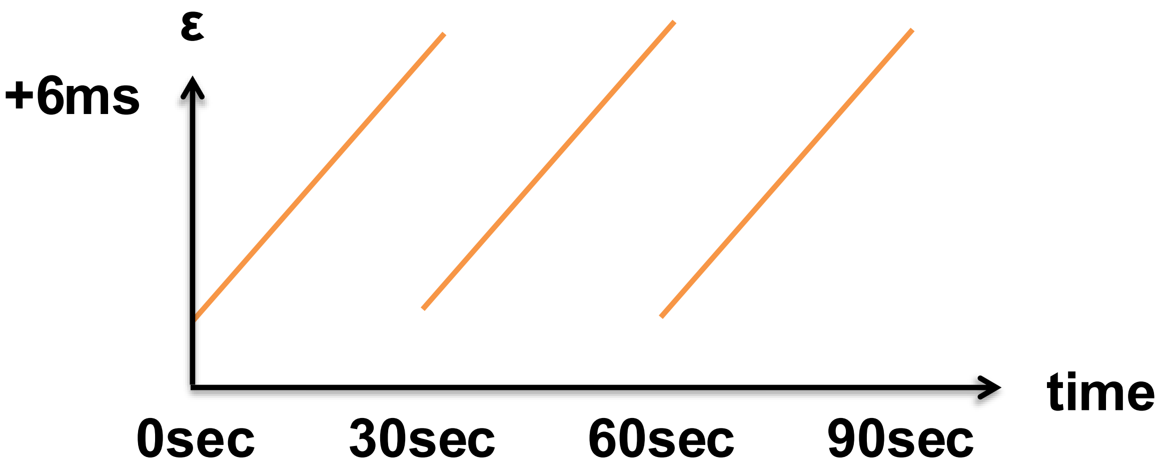
Compute reference [earliest, latest] = now ± ε

TrueTime implementation

now = reference now + local-clock offset

ε = reference ε + worst-case local-clock drift

= 1ms + 200 μs/sec

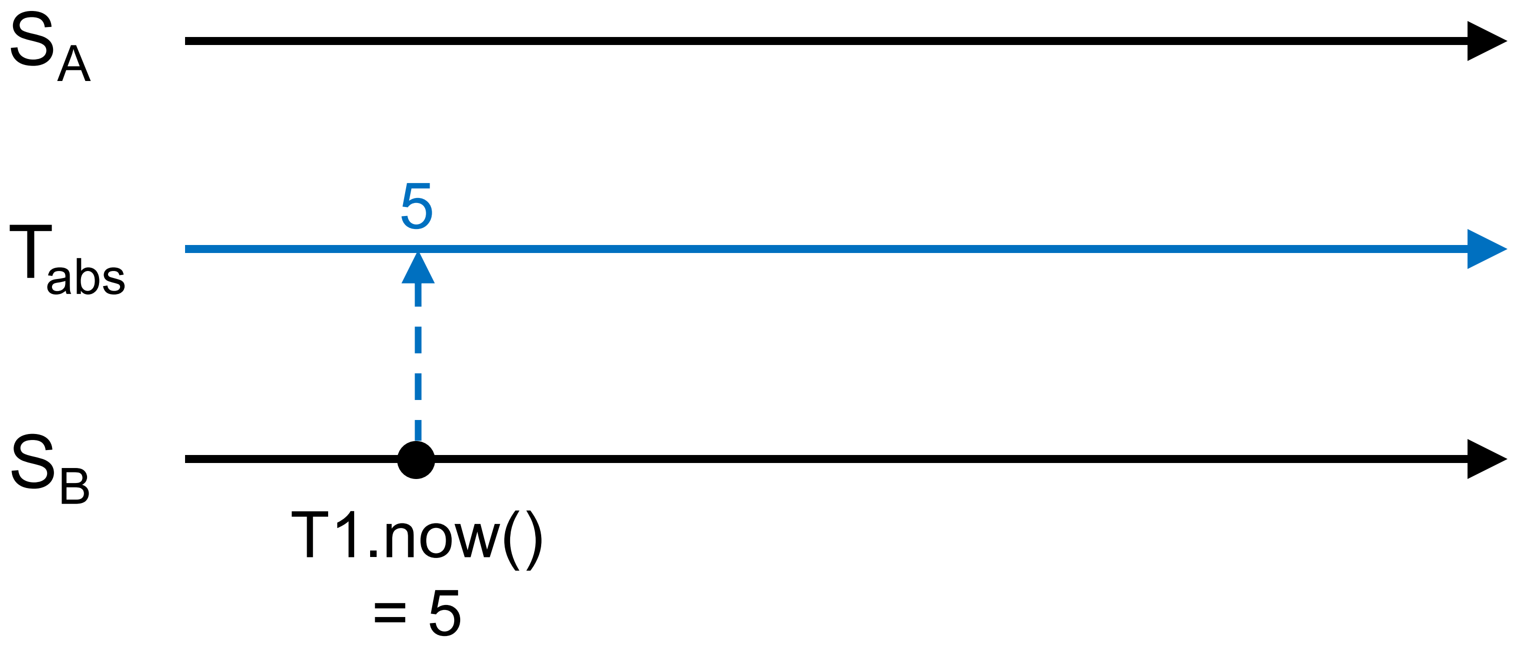


- What about faulty clocks?
  - Bad CPUs 6x more likely in 1 year of empirical data

Enforcing the Invariant

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

Let T1 write S_B and T2 write S_A

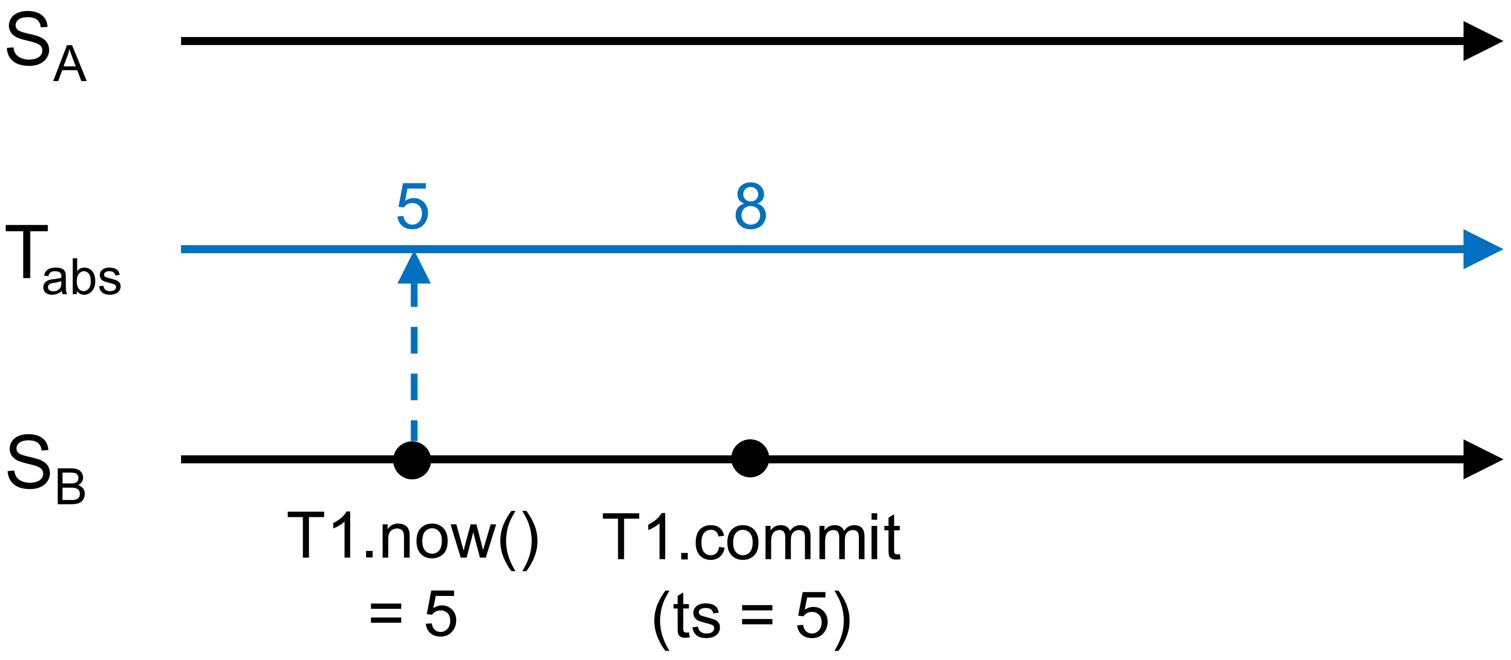


Perfect Clocks

Enforcing the Invariant

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

Let T1 write S_B and T2 write S_A

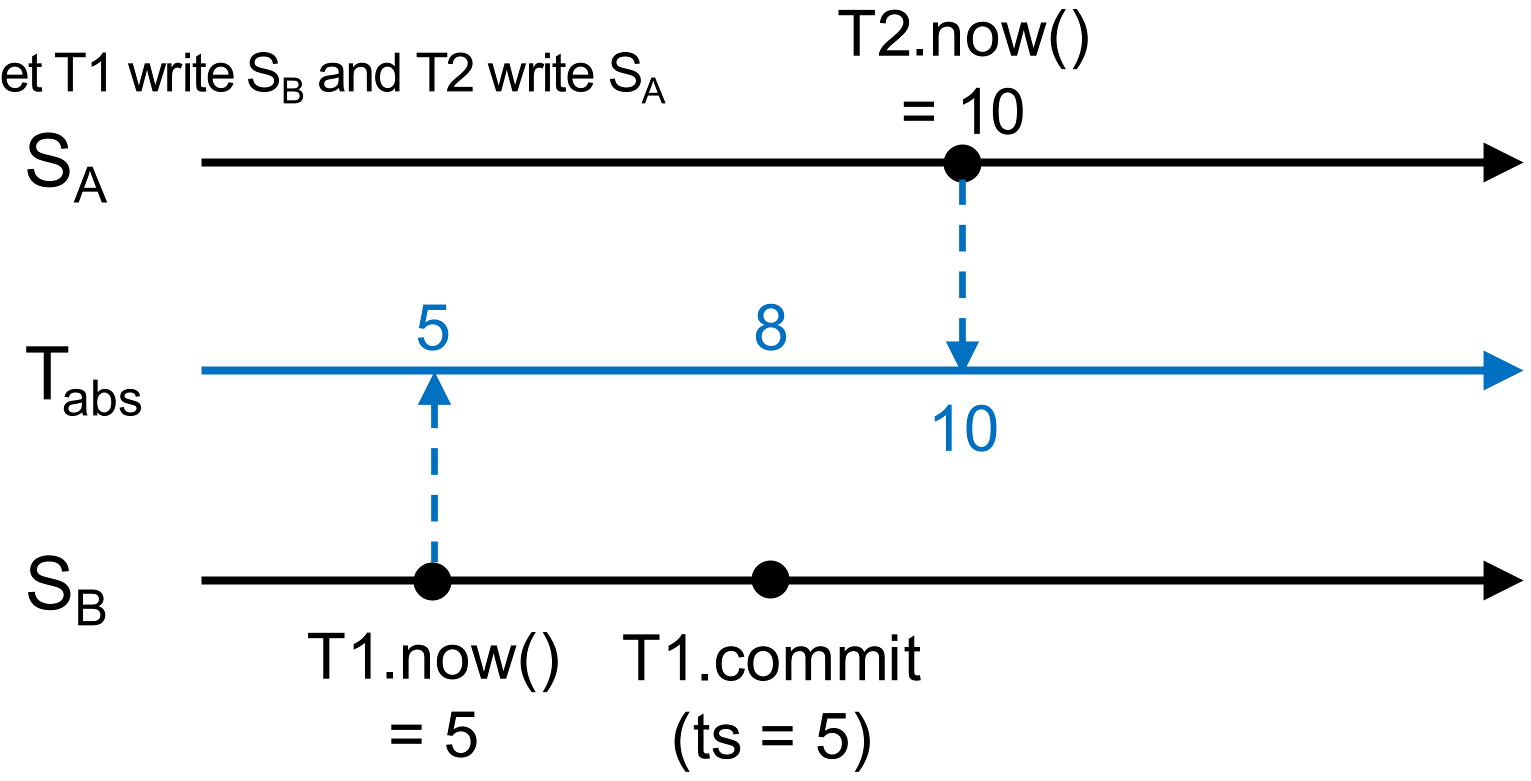


Perfect Clocks

Enforcing the Invariant

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

Let T1 write S_B and T2 write S_A

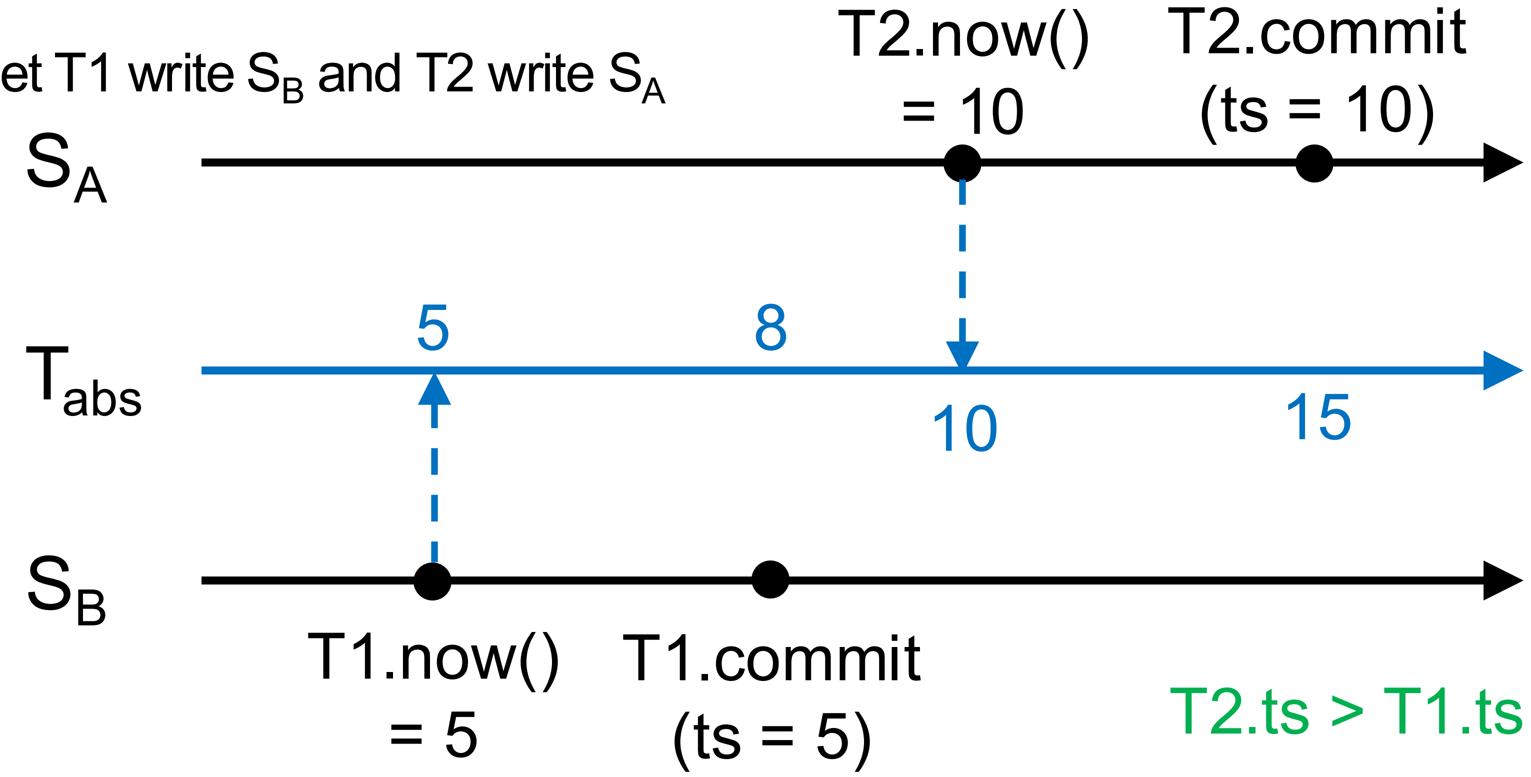


Perfect Clocks

Enforcing the Invariant

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

Let T1 write S_B and T2 write S_A



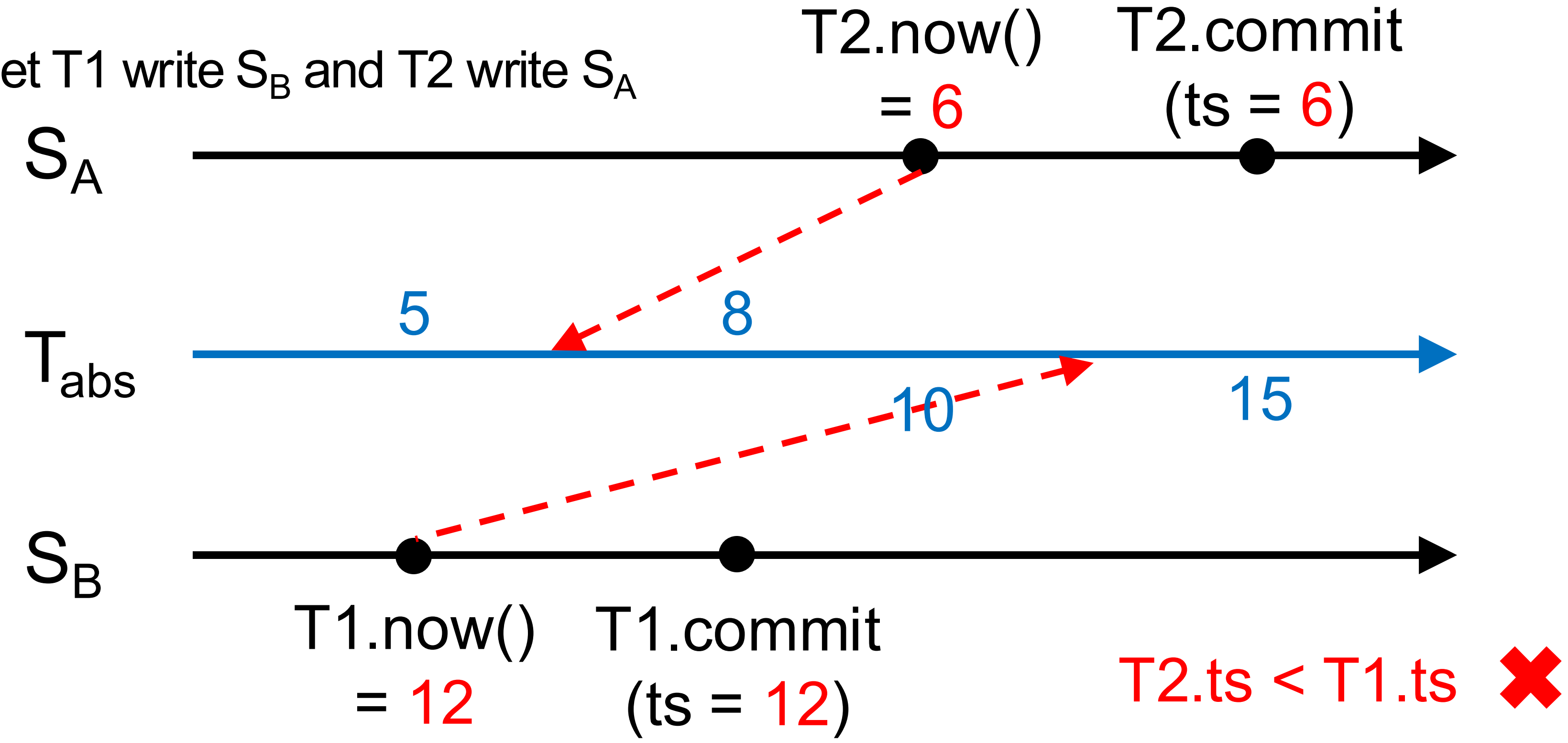
T2.ts > T1.ts

Perfect Clocks

Enforcing the Invariant

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.now() = 6

T2.commit (ts = 6)

S_A

5

8

T_{abs}

10

15

S_B

T1.now() = 12

T1.commit (ts = 12)

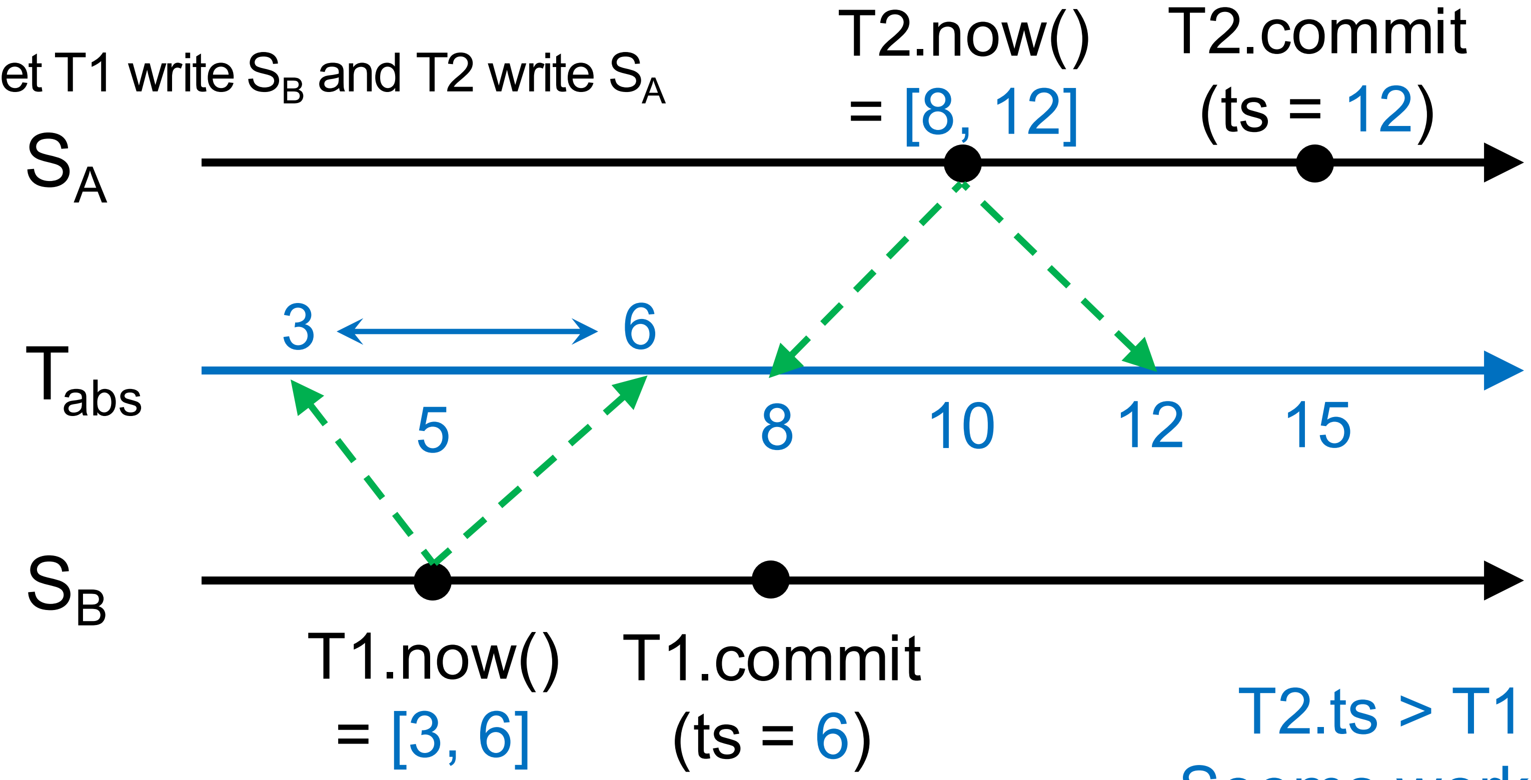
T2.ts < T1.ts ✖

Imperfect Clocks

Enforcing the Invariant

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.now() = [8, 12]

T2.commit (ts = 12)

S_A

T_{abs}

3

6

5

8

10

12

15

S_B

T1.now() = [3, 6]

T1.commit (ts = 6)

TrueTime

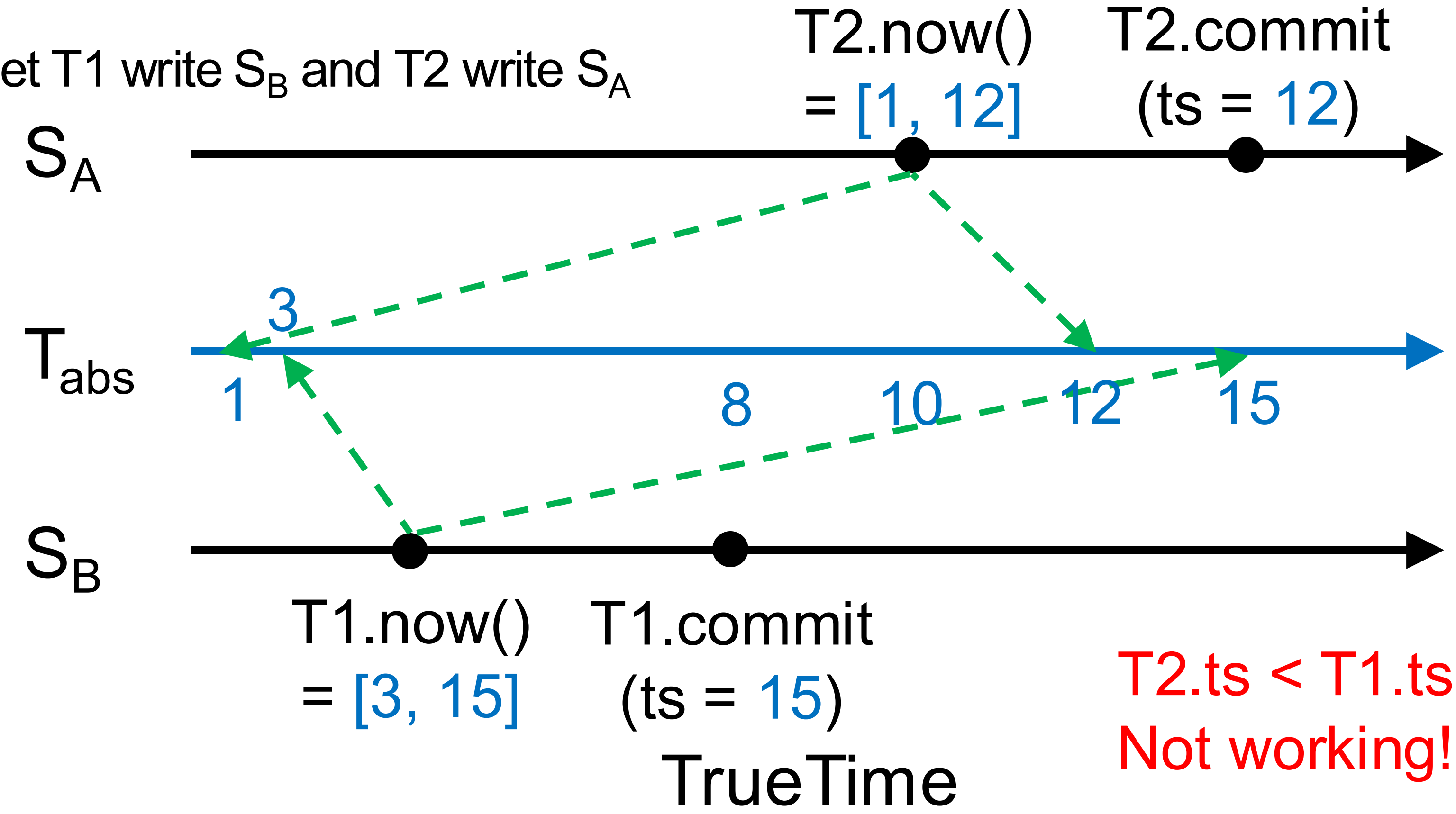
T2.ts > T1.ts

Seems working?

Enforcing the Invariant

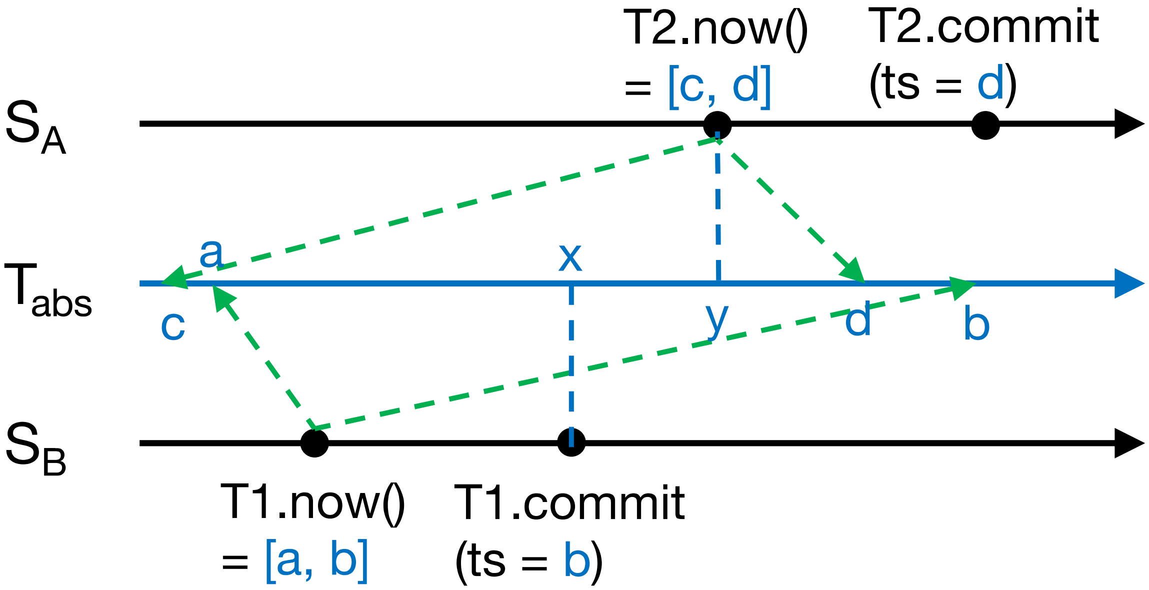
If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Let T1 write S_B and T2 write S_A



T2.ts < T1.ts
Not working!

A brain teaser puzzle

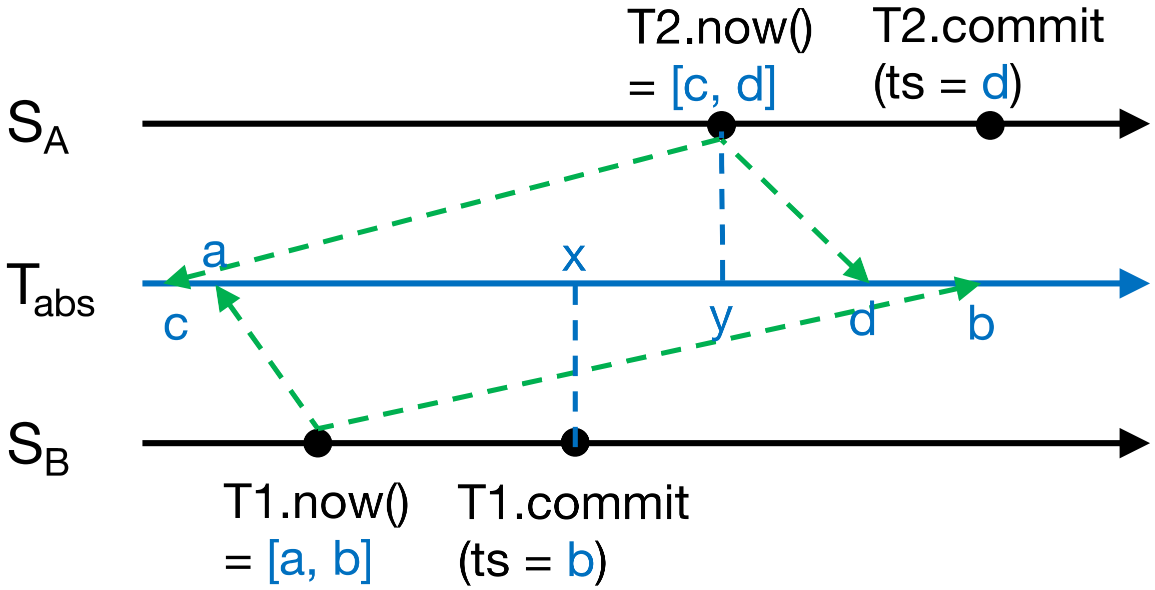


We know:

1. x < y, b/c T2 in real-time after T1 (the assumption)
2. c <= y <= d, b/c TrueTime
3. T1.ts = b, T2.ts = d, b/c how ts is assigned

We want: it is always true that b < d, how?

A brain teaser puzzle



We know:

1. x < y, b/c T2 in real-time after T1 (the assumption)
2. c <= y <= d, b/c TrueTime
3. T1.ts = b, T2.ts = d, b/c how ts is assigned

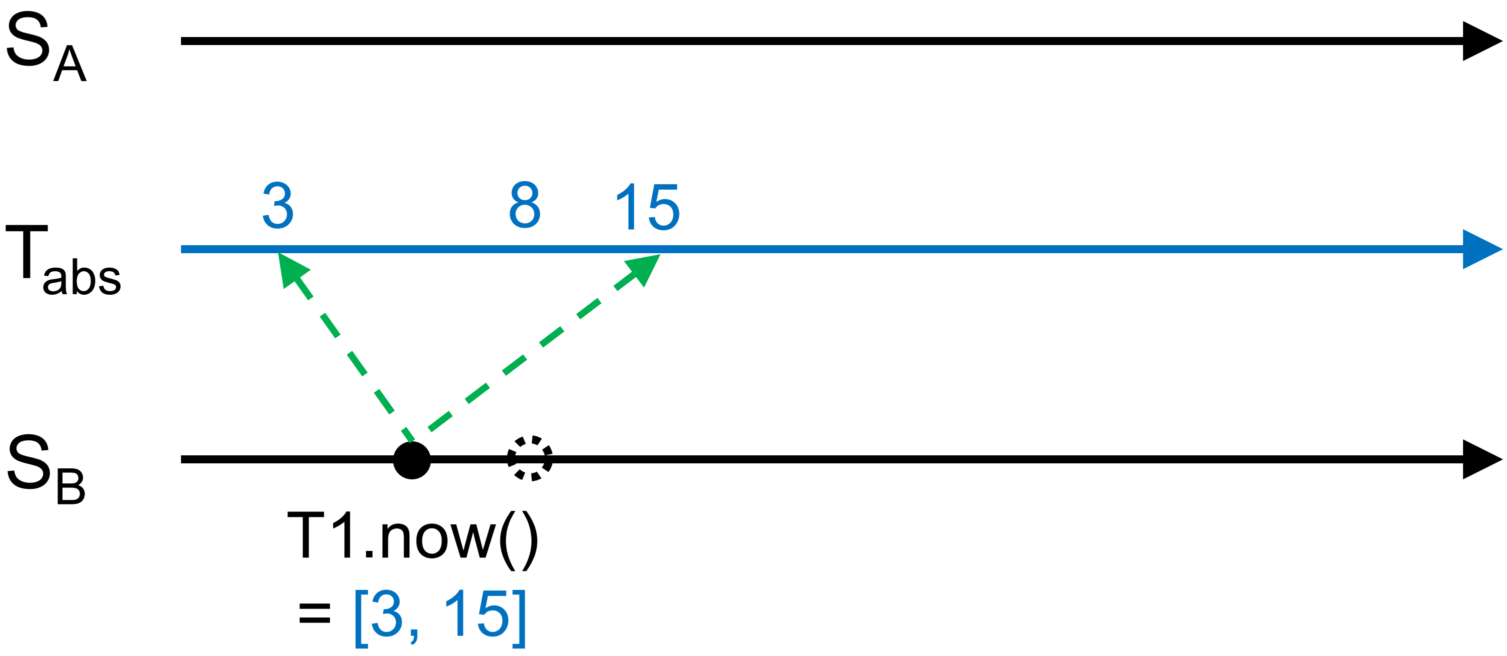
We want: it is always true that b < d, how?

1 and 2 → x < d; we need to ensure b < x; then b < x < d, done

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

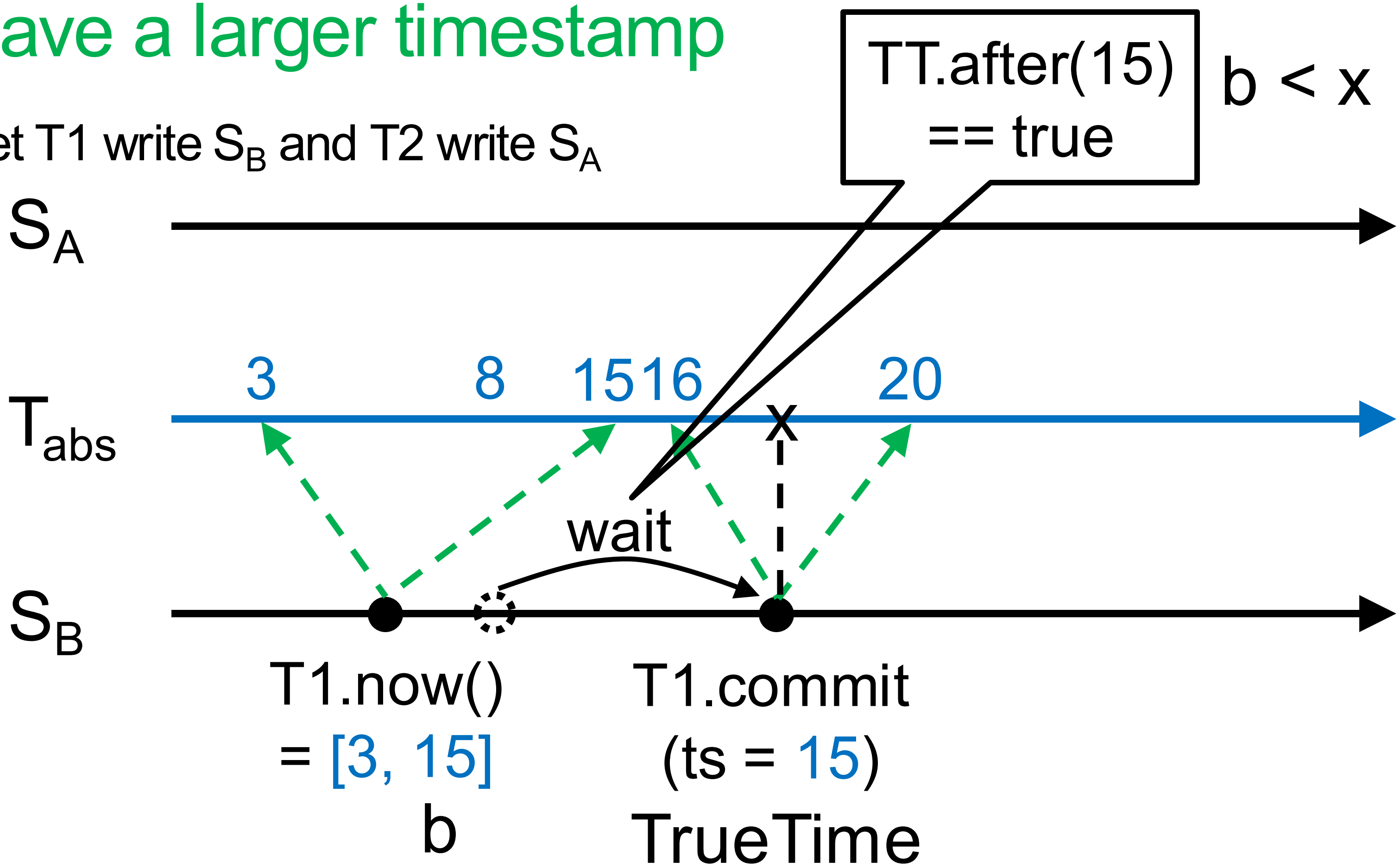


TrueTime

Enforcing the Invariant with TT

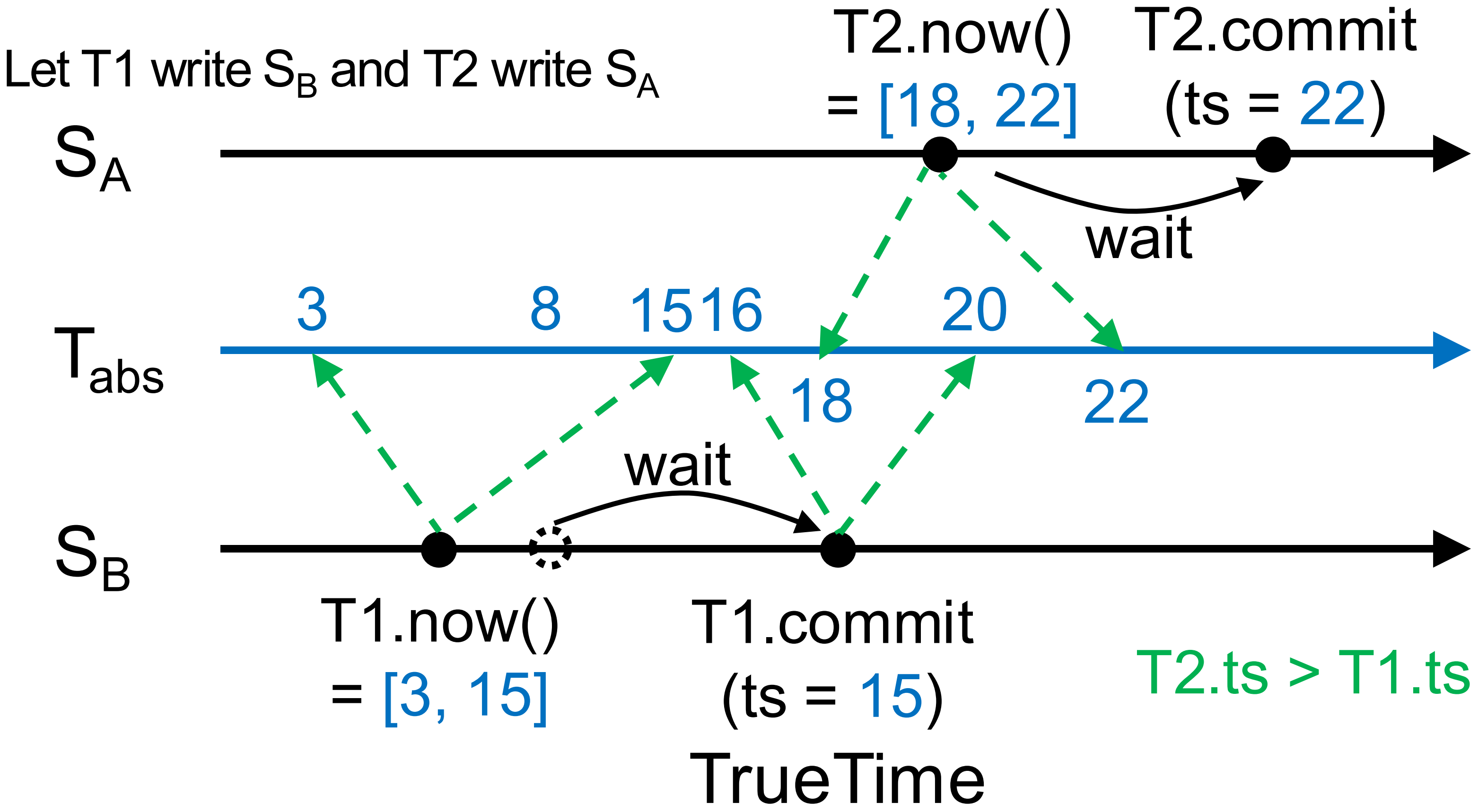
If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

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



Takeaways

- The invariant is always enforced: If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp
- How big/small ε is does not matter for correctness
- Only need to make sure:
  - TT.now().latest is used for ts (in this example)
  - Commit wait, i.e., TT.after(ts) == true
- ε must be known a priori and small so commit wait is doable!

After-class Puzzles

- Can we use TT.now().earliest for ts?

- Can we use TT.now().latest – 1 for ts?

- Can we use TT.now().latest + 1 for ts?

- Then what’s the rule of thumb for choosing ts?

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 ro-txns → good overall performance

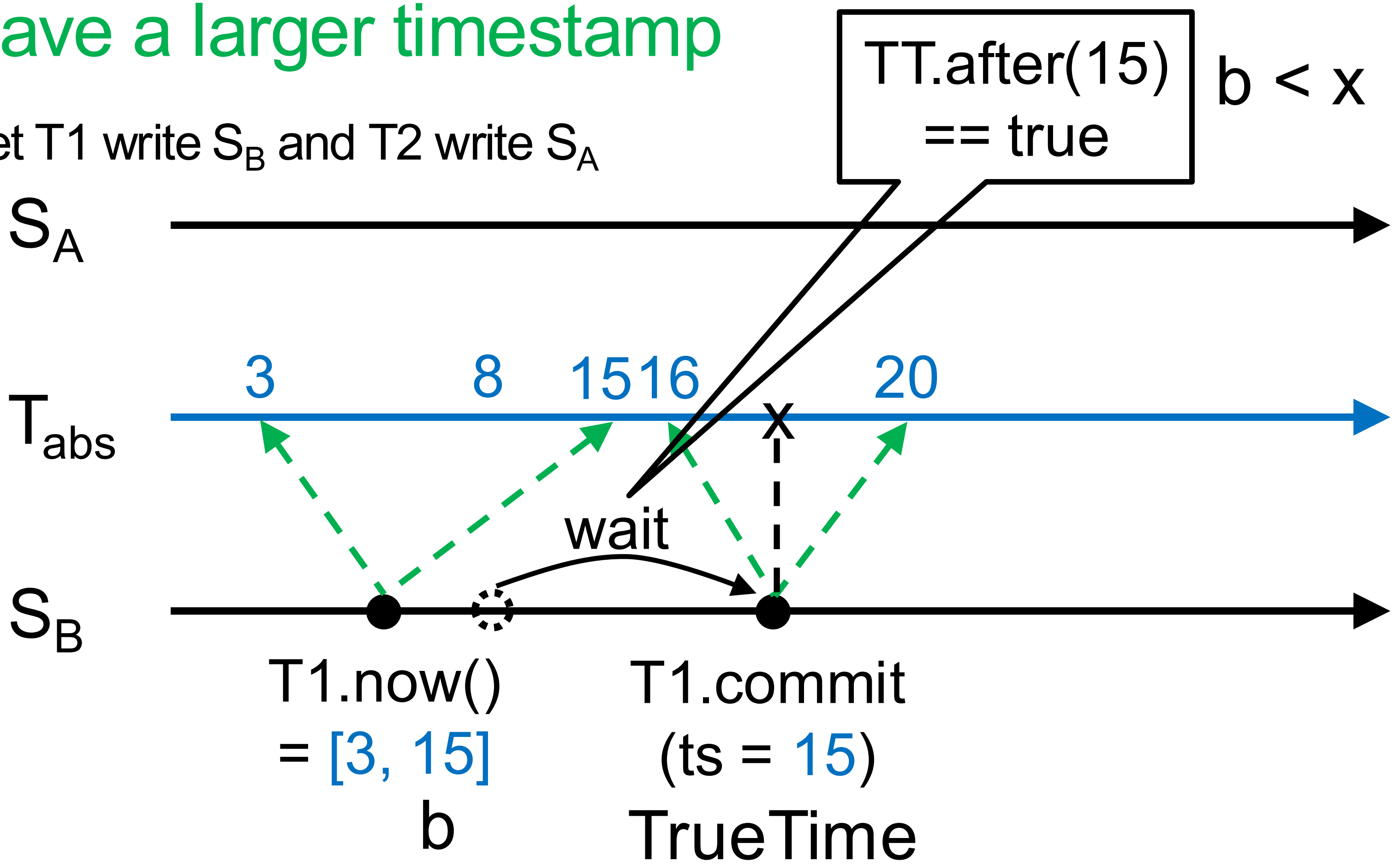
Recap: TrueTime

- 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

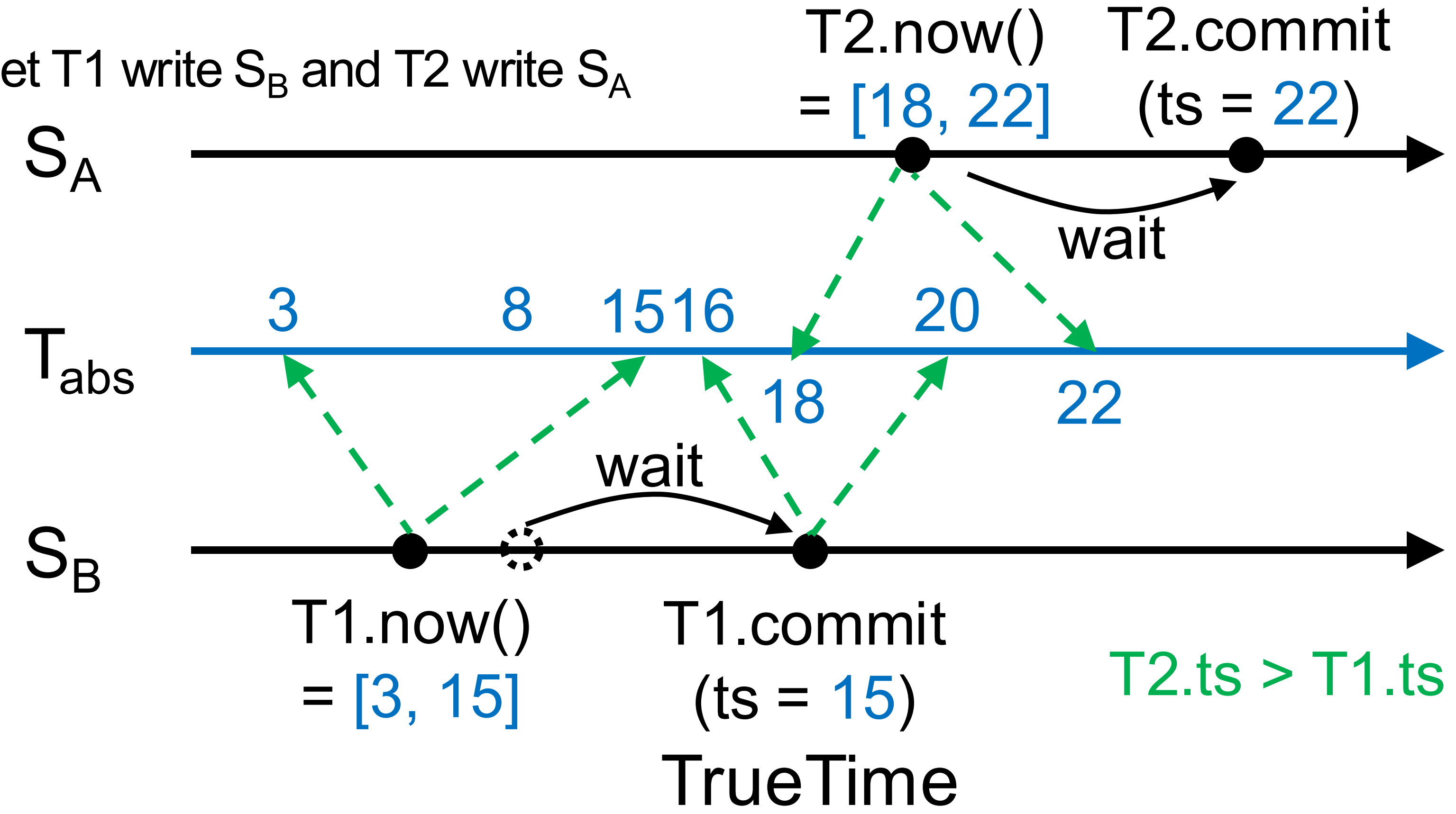
Let T1 write S_B and T2 write S_A



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



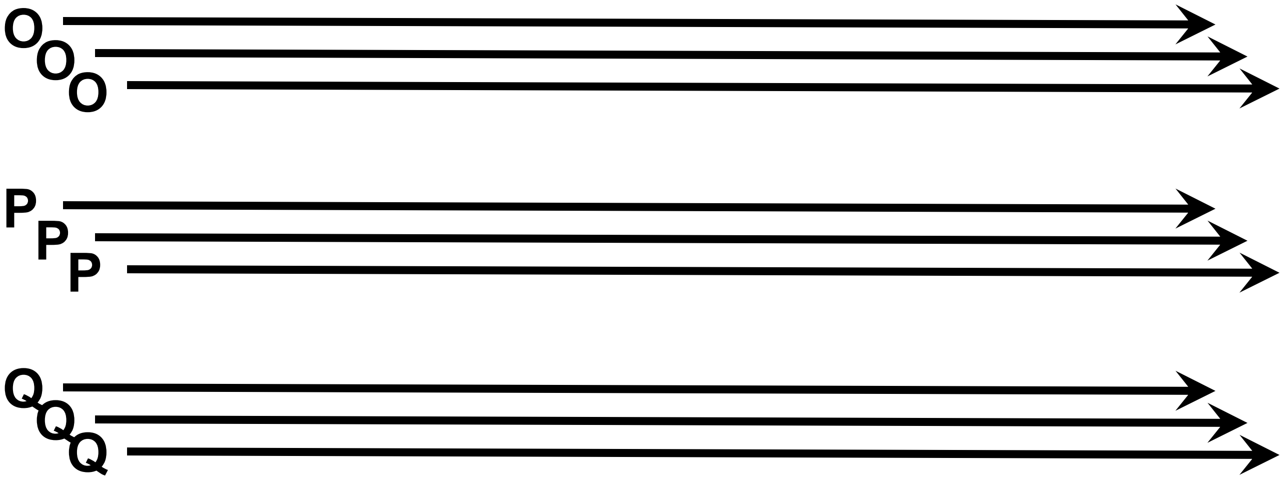
T2.ts > T1.ts

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

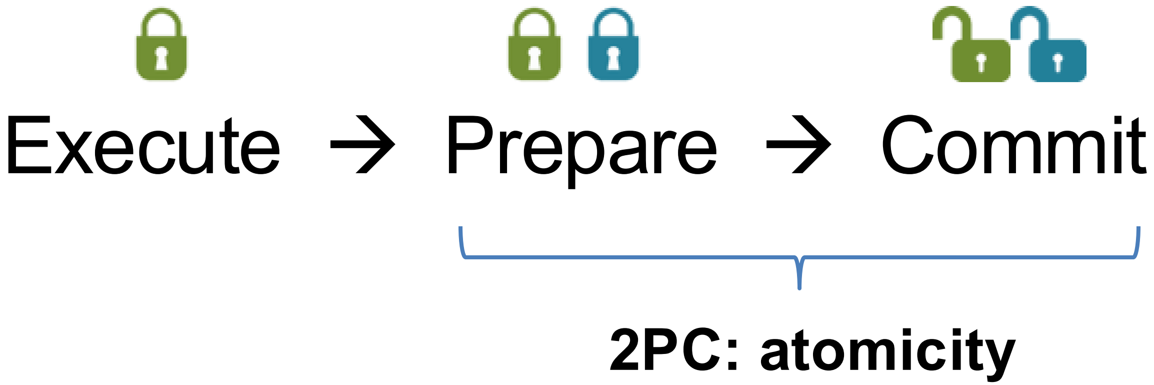
- 2PL for concurrency control of read-write transactions
- 2PC for distributed transactions over tables
- (Multi)Paxos for replicating every tablet

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



Execute → Prepare → Commit

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

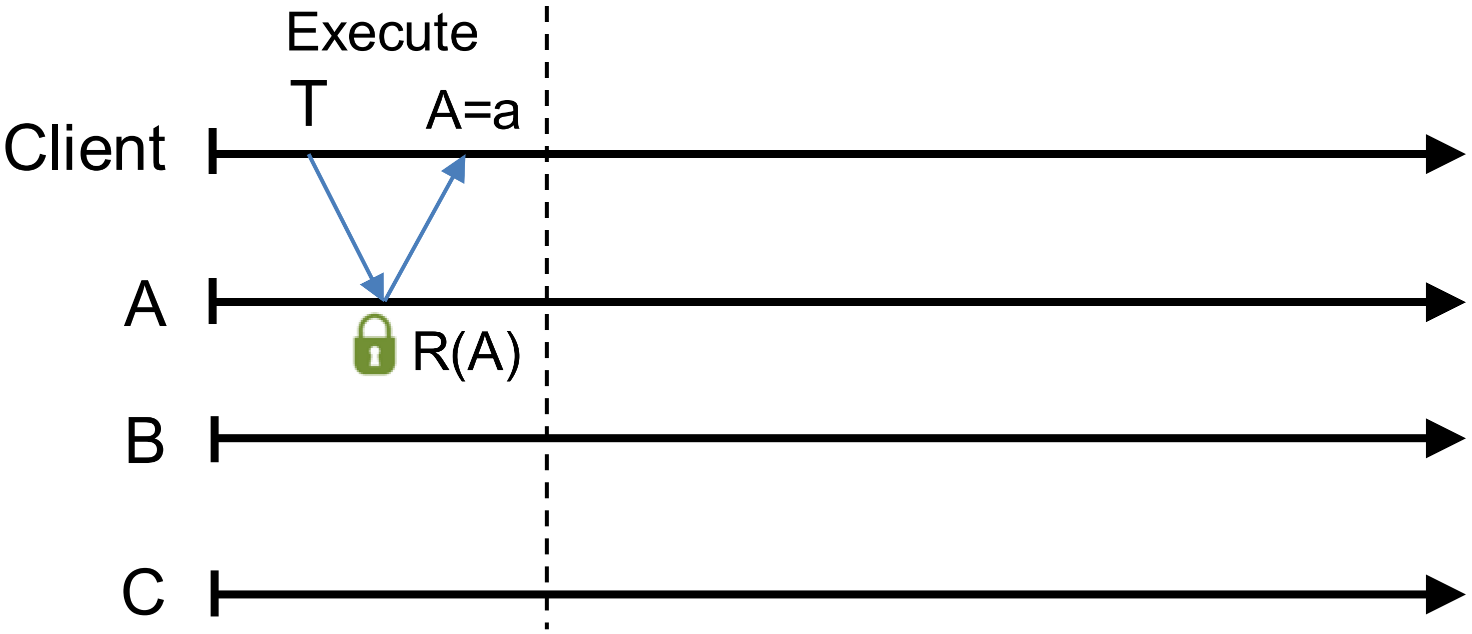
2. Locally performs writes

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)

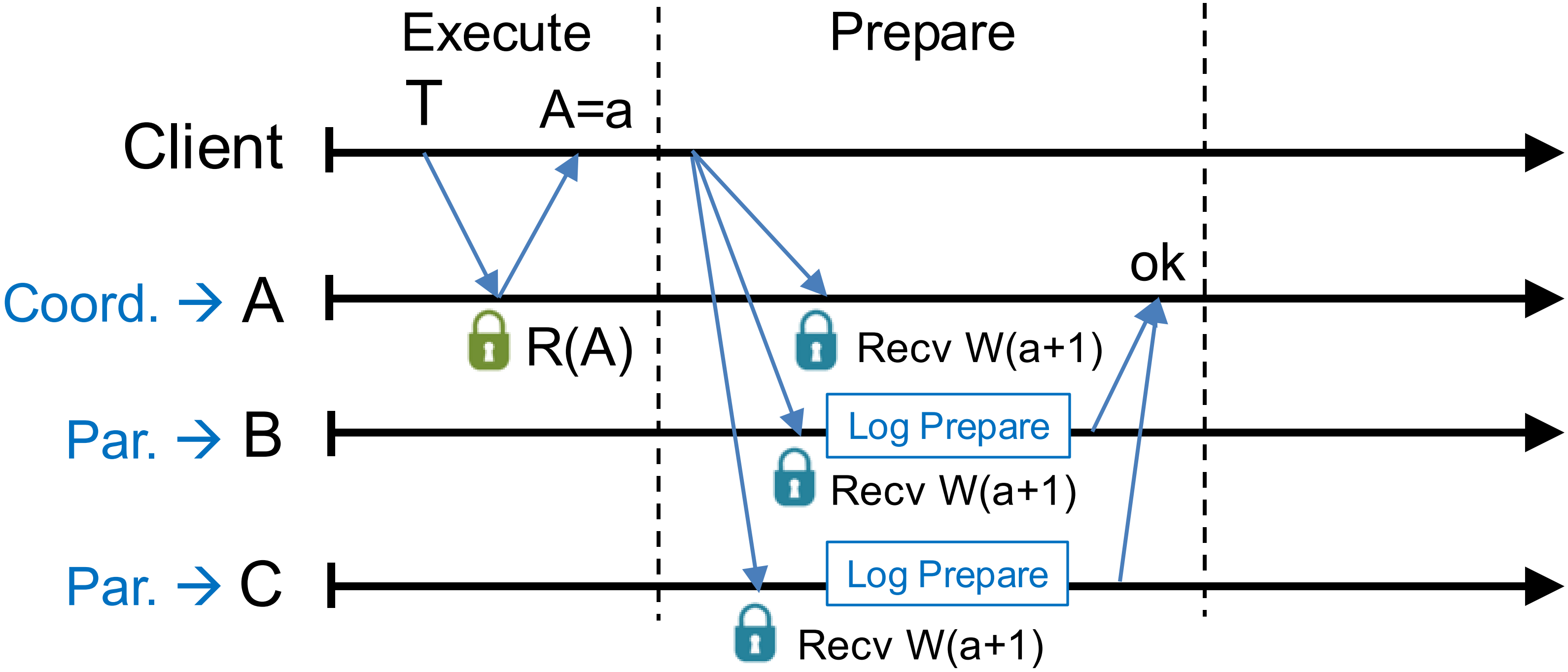


Txn T = {R(A=?), W(A=?+1), W(B=?+1), W(C=?+1)}

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

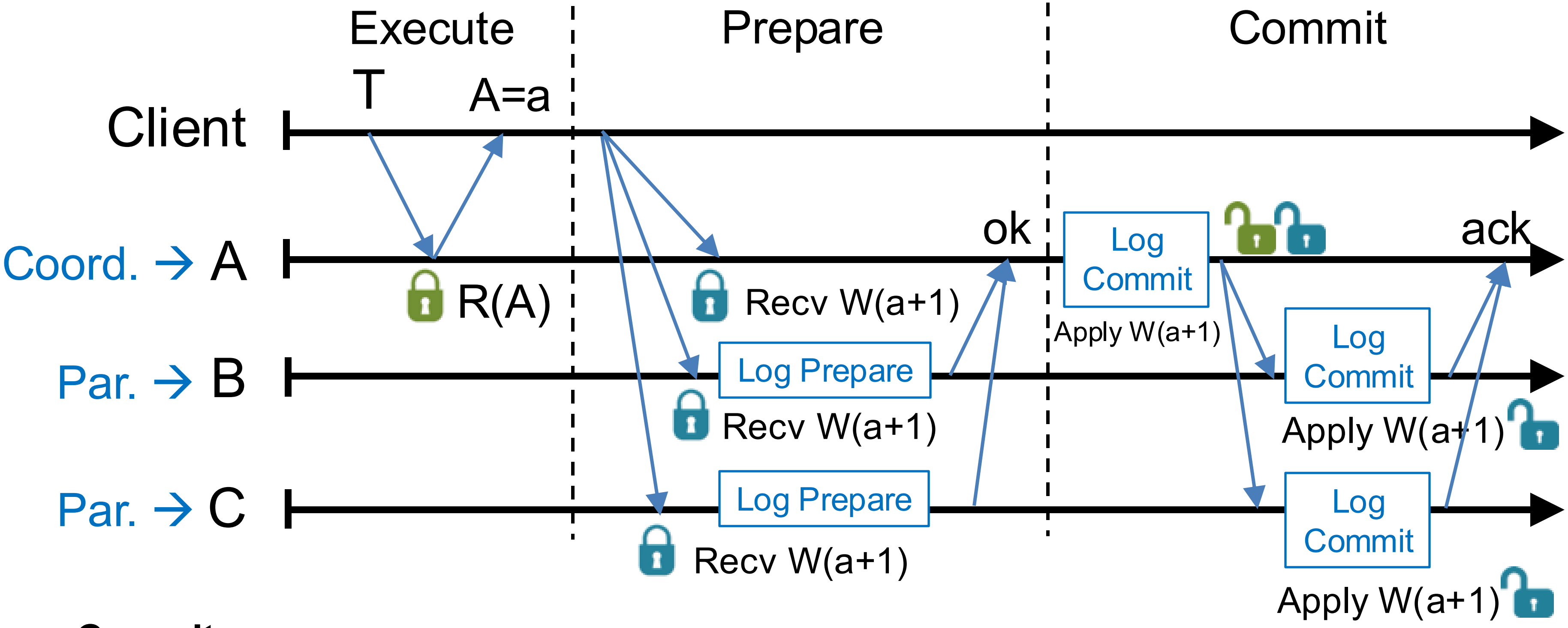
Read-Write Transactions (2PL)



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

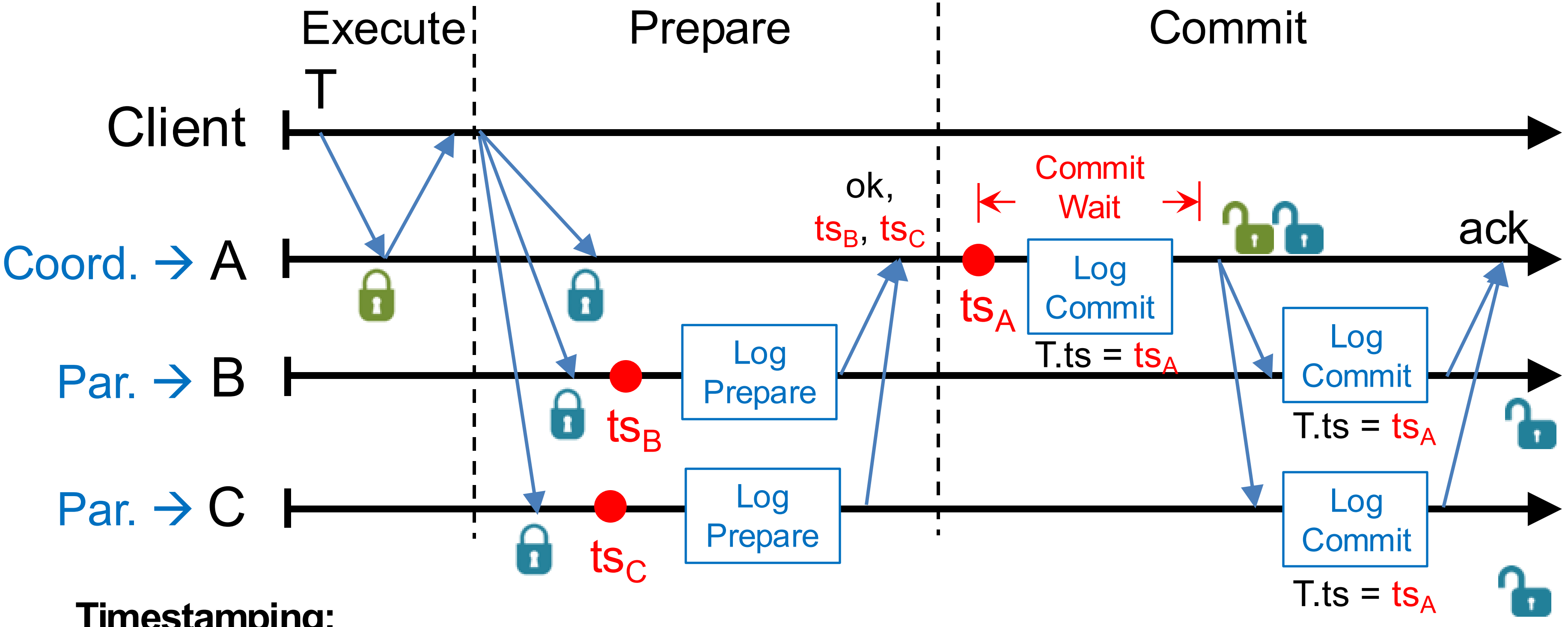
Read-Write Transactions (2PL)



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:

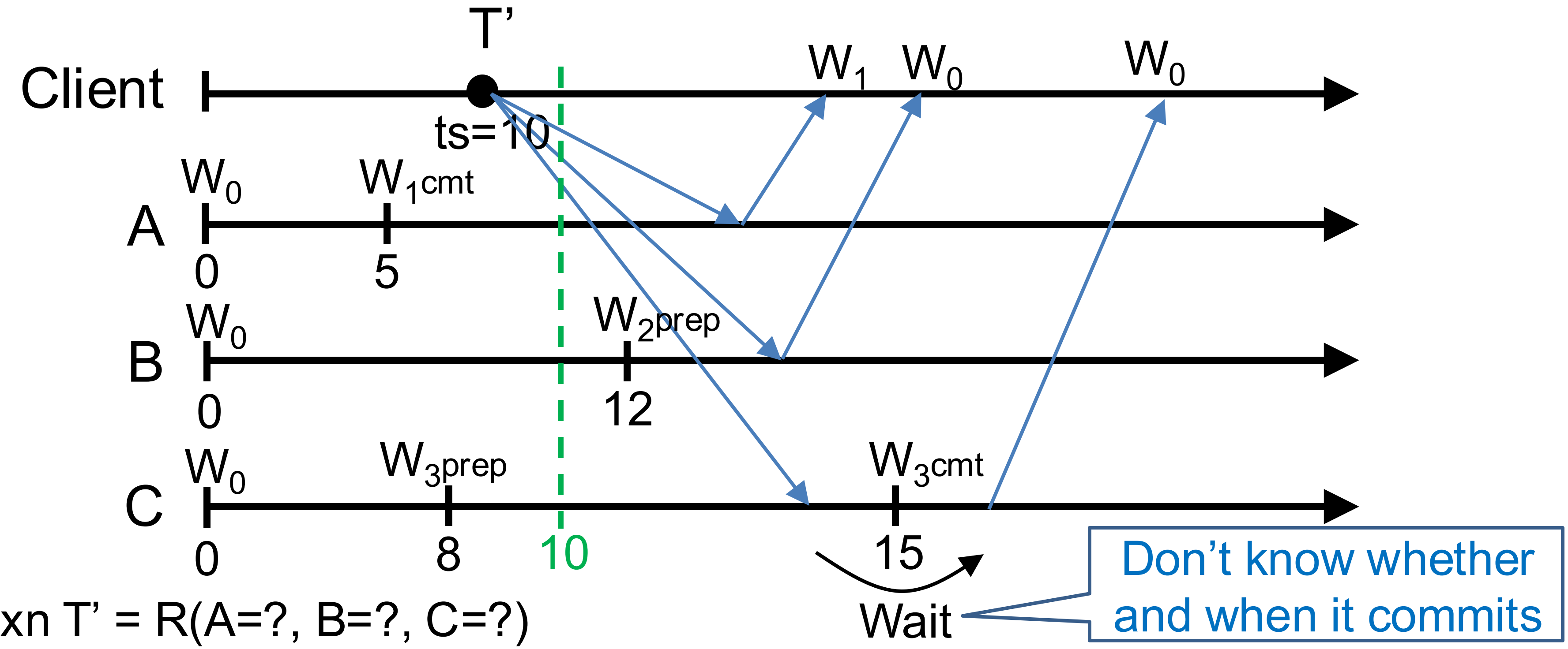
- Participant: choose a timestamp, e.g., ts_B and ts_C, larger than any writes it has applied
- Coordinator: choose a timestamp, e.g., ts_A, larger than
  - Any writes it has applied
  - Any timestamps proposed by the participants, e.g., ts_B and ts_C
  - Its current TT.now().latest
- Coord commit-waits: 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
  - Ro-txns are one-round, lock-free, and never abort

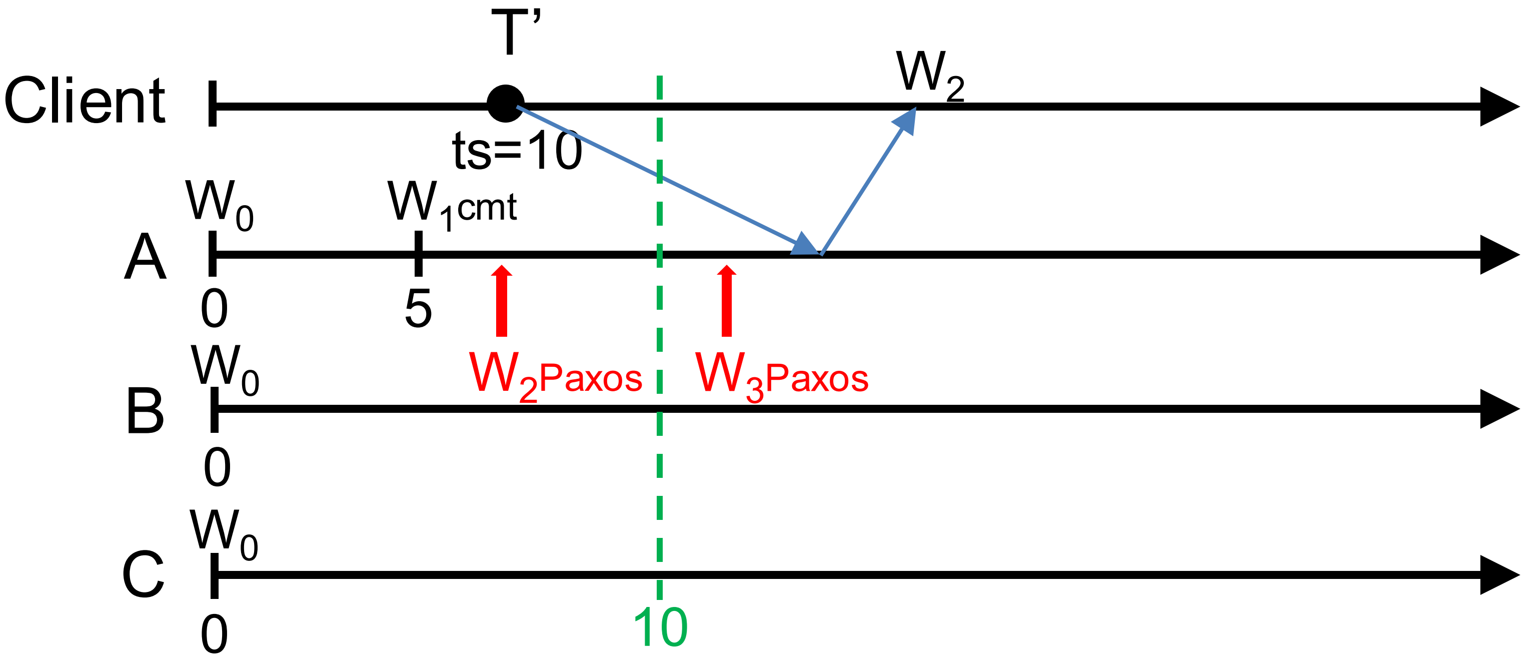
Read-Only Transactions (shards part)



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)

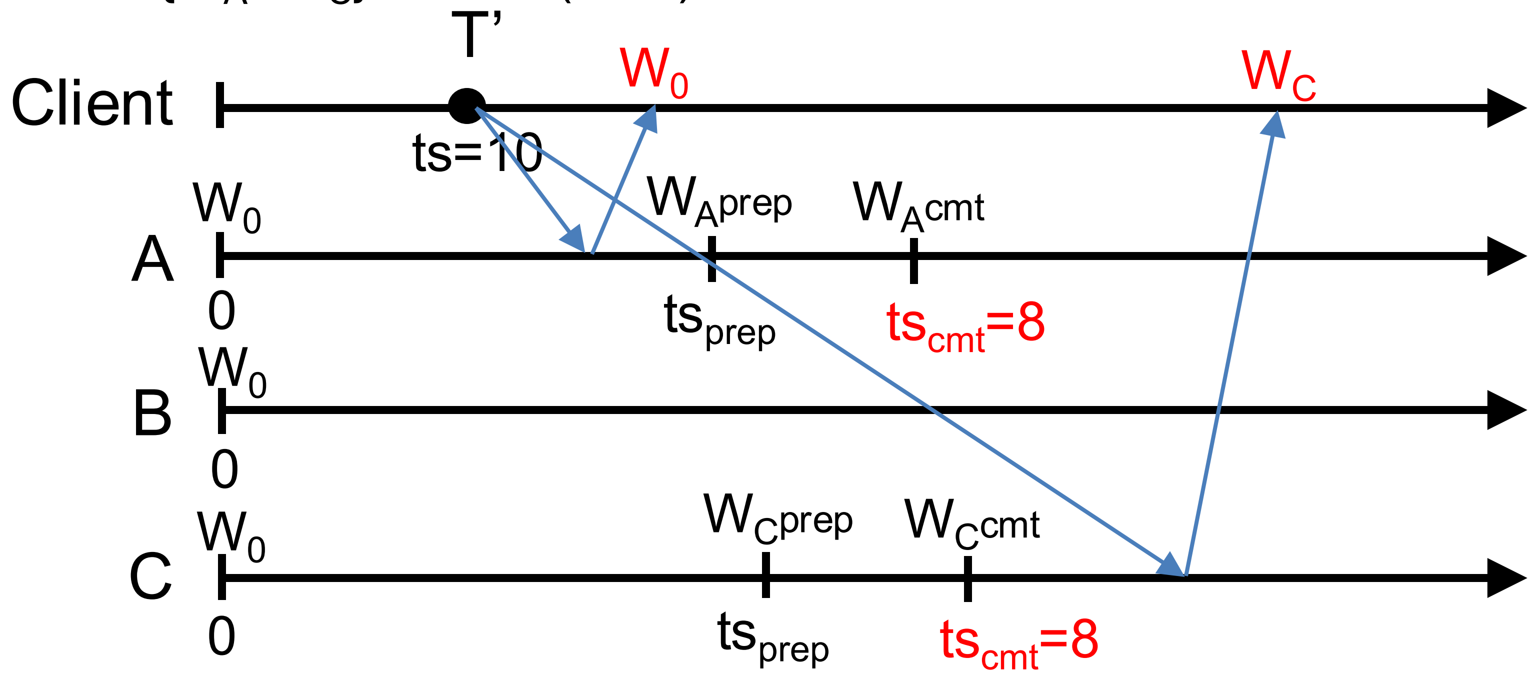


- Paxos writes are monotonic, e.g., writes with smaller timestamp must be applied earlier, W_2 is applied before W_3
- T' needs to wait until there exits a Paxos write with ts>10, e.g., W_3, so all writes before 10 are finalized
- Put it together: a shard can process a read at ts if ts <= t_{safe}
-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

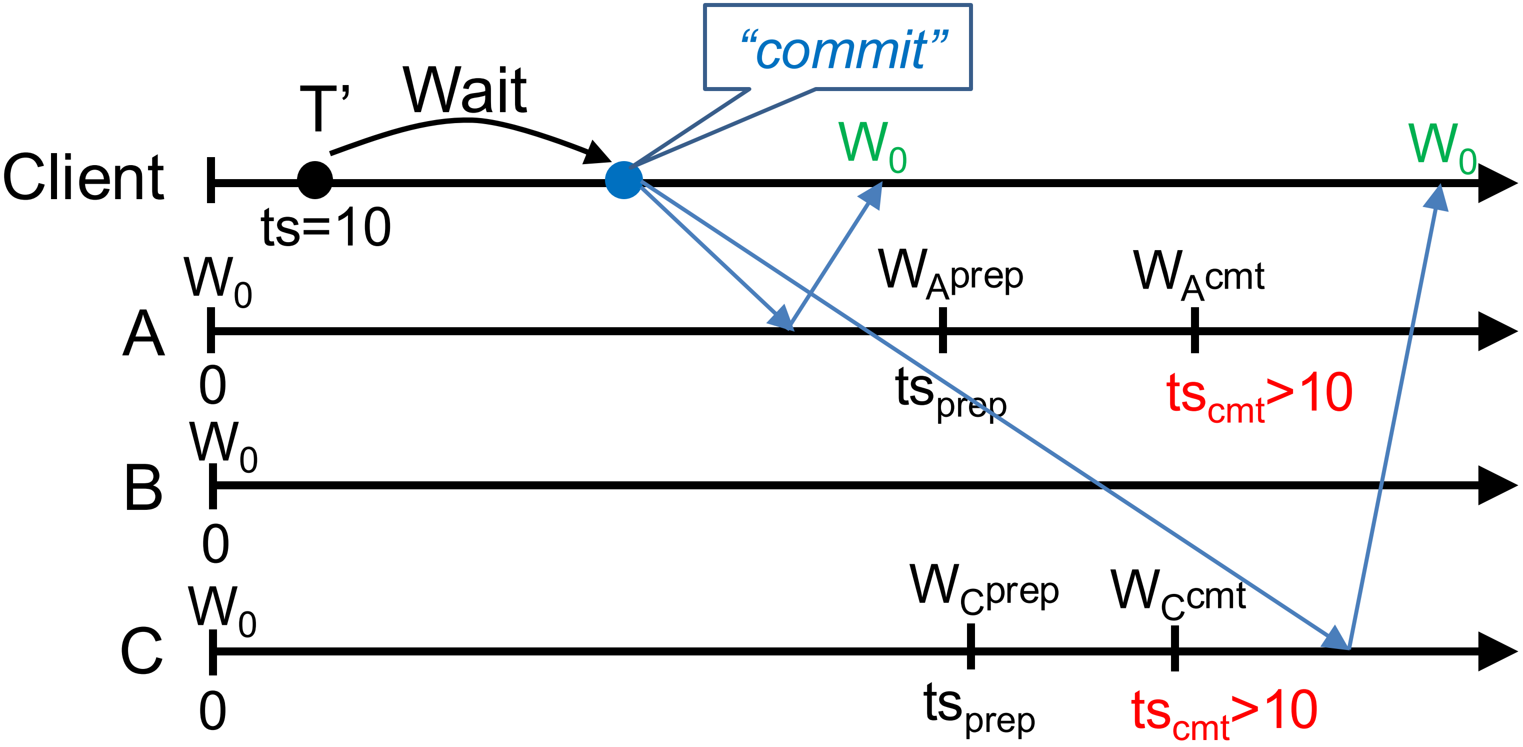
Txn T = {W_A, W_C}, T' = R (A, C)



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!