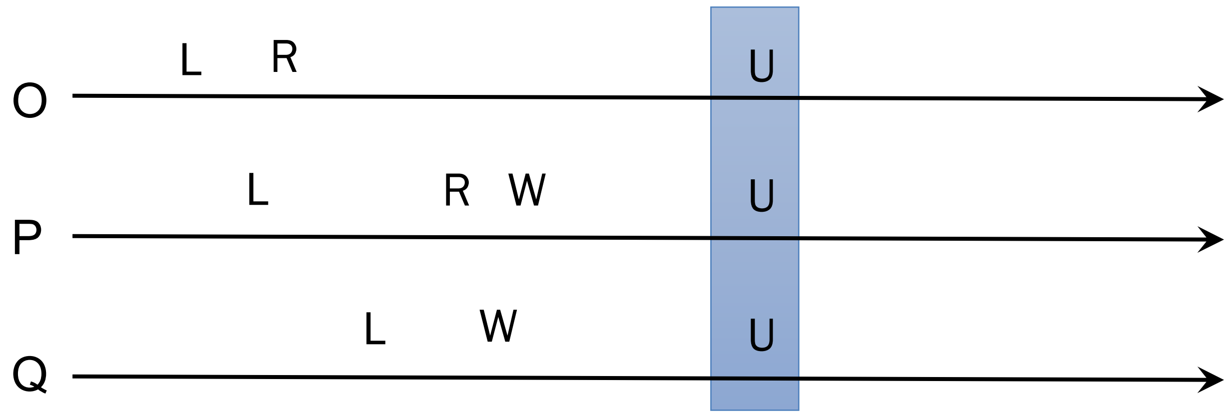
Distributed Transactions

CS6450: Distributed Systems

Lecture 17

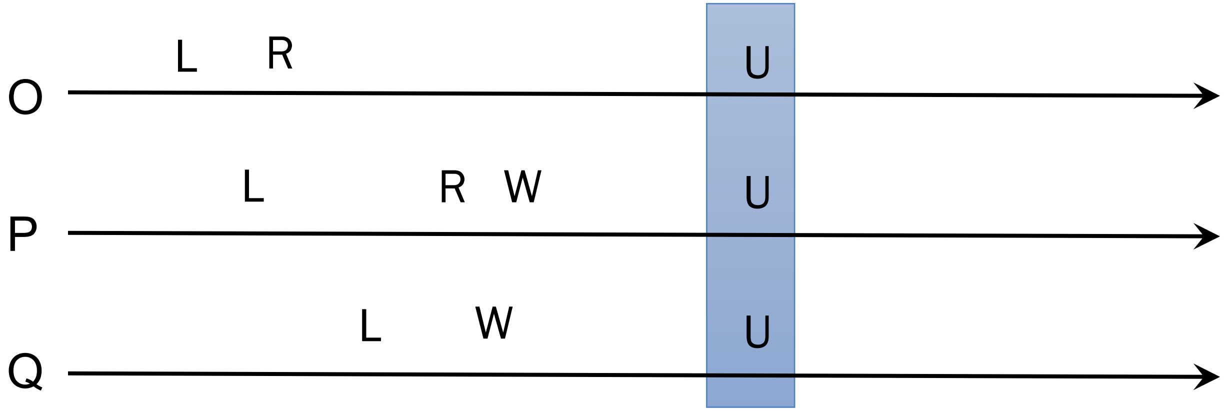
Ryan Stutsman

Consider partitioned data over servers



- Why not just use 2PL?
  - Grab locks over entire read and write set
  - Perform writes
  - Release locks (at commit time)
- Max throughput for hot item: 1/RTT
  - 10 txns/s? 100? 1000? Reader blocked by writers 😩

Consider partitioned data over servers



- How do you get serializability?
  - Need to generate a total order/timestamp? Hard to escape.
  - On one machine, often the COMMIT record in the WAL
  - Or a timestamp for timestamp-based CC
- Distributed: global timestamp per transaction
  - Centralized transaction manager?
  - Distributed consensus on timestamp?
  - Validation looks like an out, but its not really…

Problems with Partitioning Validation

- T1: R(x)=0 W(x)=1
- T2: R(x)=0 W(y)=1
- T3: R(y)=0 R(x)=1

Problems with Partitioning Validation

- T1: R(x)=0 W(x)=1
- T2: R(x)=0 W(y)=1
- T3: R(y)=0 R(x)=1

Problems with Partitioning Validation

- S1 validates just x information:
  - T1: R(x)=0 W(x)=1
  - T2: R(x)=0
  - T3: R(x)=1
  - Answer is "yes" (T2 T1 T3)
- S2 validates just y information:
  - T2: W(y)=1
  - T3: R(y)=0
  - Answer is "yes" (T3 T2)
- But, the real answer is "no"!

Q: What went wrong here?

Spanner: Google's Globally-Distributed Database

OSDI 2012

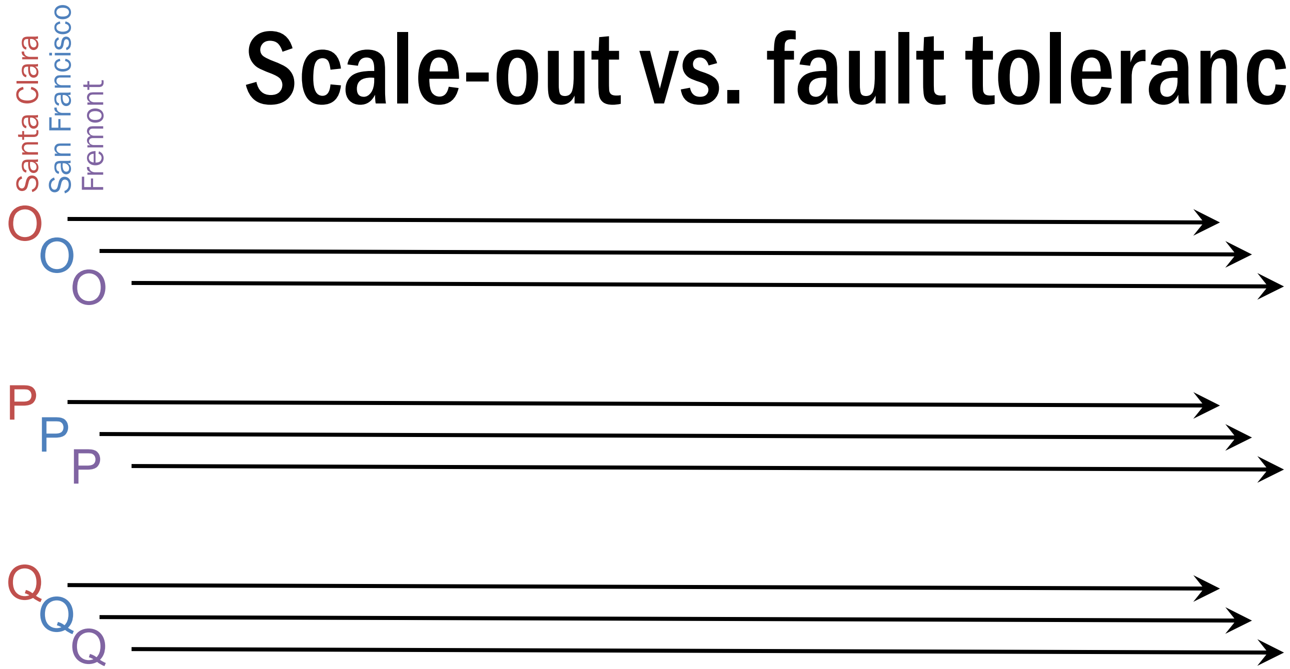
Google's Setting

- Dozens of zones across datacenters
- Per zone, 100-1000s of servers
  - Servers from different zones placed in replication groups
- Per server/group, 100-1000 partitions (tablets)
- Every tablet replicated for fault-tolerance (e.g., 5x)
  - Paper notes in *nearby* datacenters

#1 Rule: Never Lose a Sale

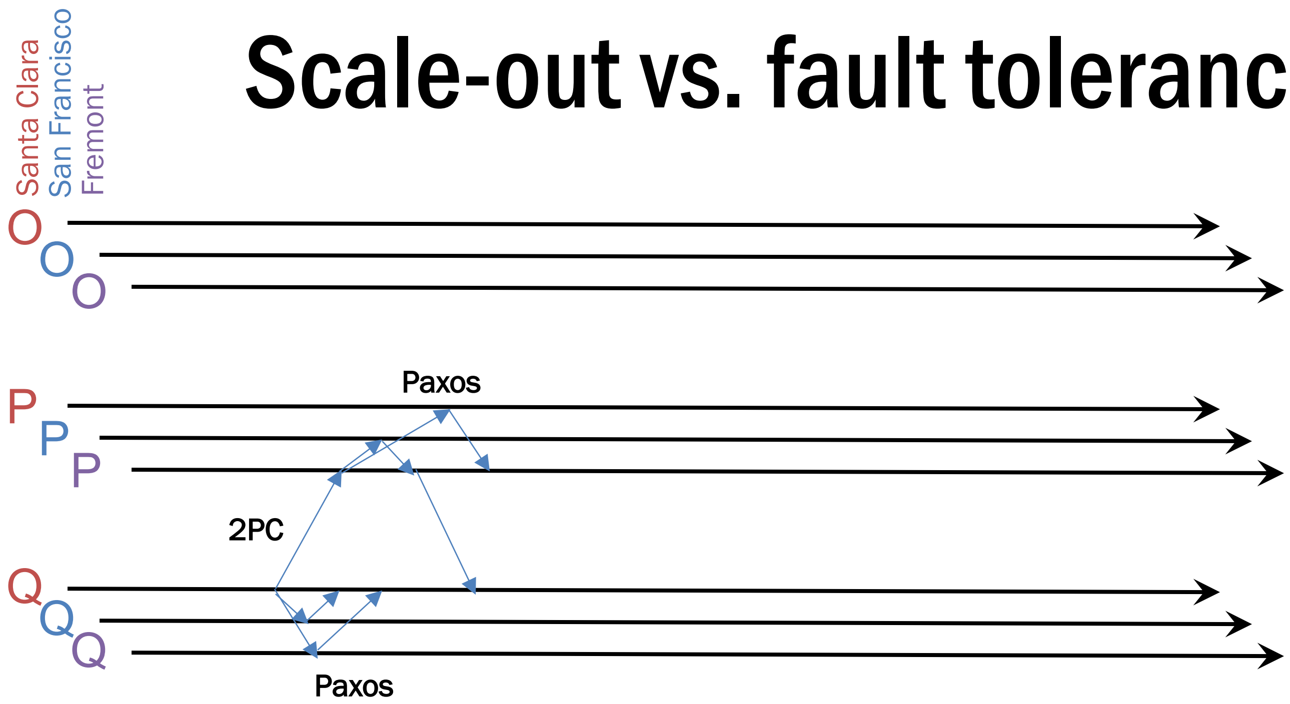
But can we do it with low(-ish) latency
and linearizbility/serializability+external consistency?

Scale-out vs. fault tolerance



- Every tablet replicated via Paxos (with leader election)
- So every "operation" within transactions across tablets actually a replicated operation within Paxos RSM
- Paxos groups can stretch across datacenters!

Scale-out vs. fault tolerance



- Every tablet replicated via Paxos (with leader election)
- So every "operation" within transactions across tablets actually a replicated operation within Paxos RSM
- Paxos groups can stretch across datacenters!
- **Question: what was a serious concern for 2PC at scale?**

Want it all

- All state replicated in multiple data centers
- Externally consistent transactions ongoing that span partitions (involve multiple Paxos groups)
  - 2PL at externally consistent timestamp overs all Paxos groups
- Externally consistent, non-transactional reads to the *local* Paxos replica without consensus or locks
  - Execute at externally consistent timestamp that comes after all possible ongoing transactions
- Non-blocking snapshot reads and consistent lock-free read-only transactions (and in the past)
  - Execute at externally consistent timestamp that comes after all possible ongoing transactions (even when spanning Paxos groups)
- **Hard: how can a Paxos replica guarantee a read saw up-to-date state?**
- **Hard: how can we ensure timestamp of t_2 that follows t_1 will have $ts(t_2) > ts(t_1)$?**

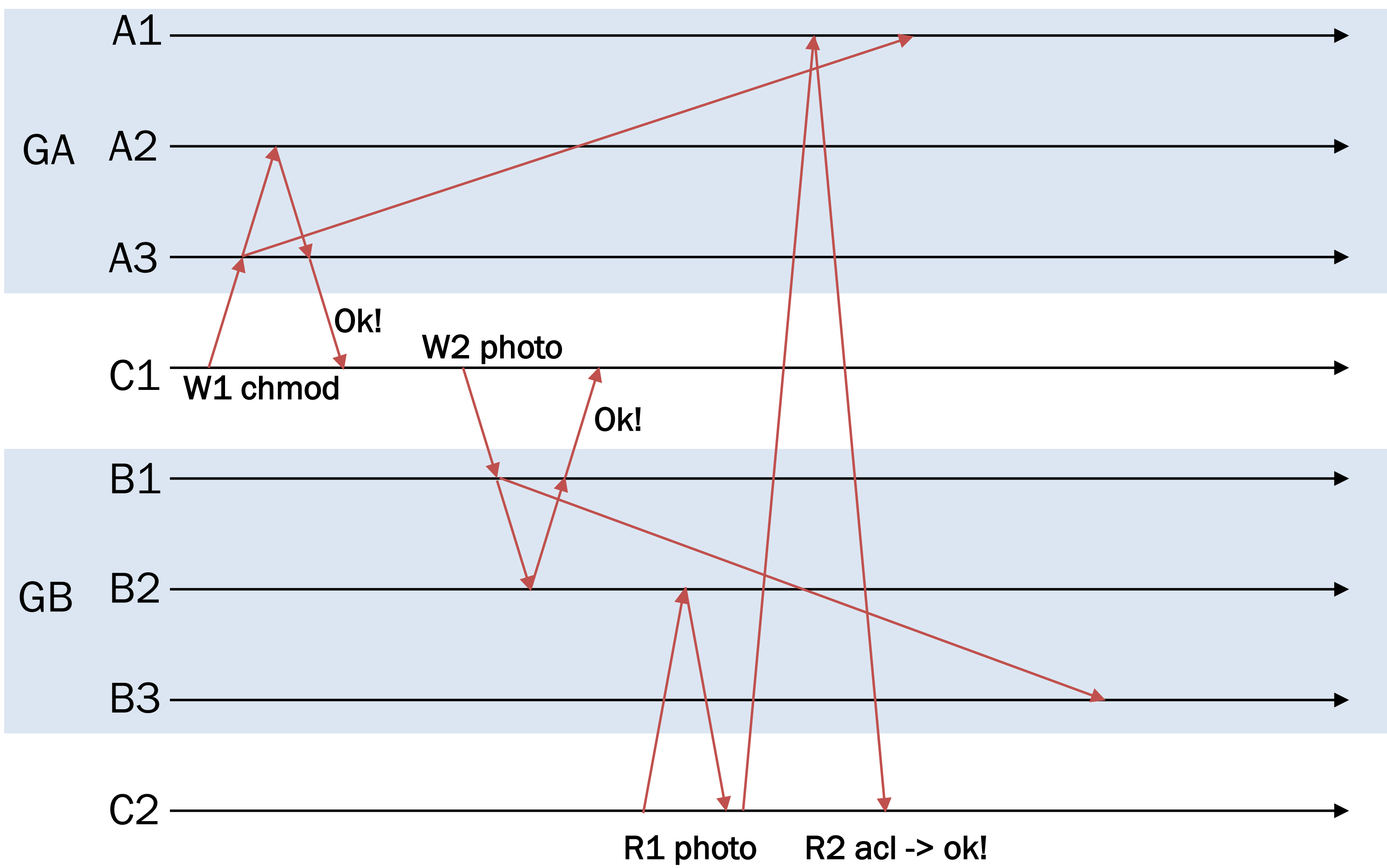
Idea: Physically Timestamp Everything

Physical timestamps + Multi-versioning

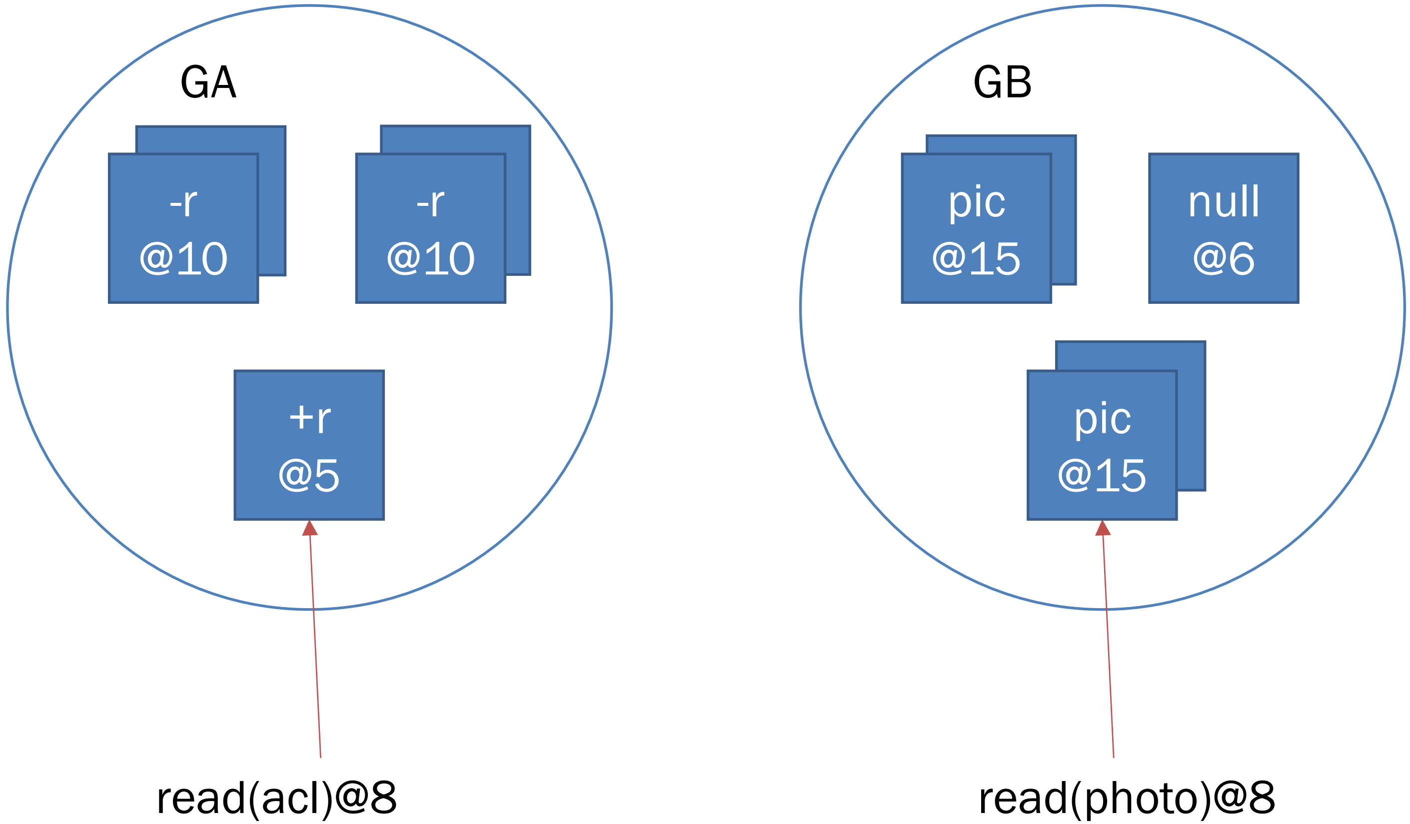
- All reads, writes, and transactions are physically timestamped
- If servers execute operations at their respective timestamps, things will be totally ordered
- and no operation that starts after the completion of another can appear to execute before it
  *** clock skew, though!

Race Due to Reading at Replicas

chmod -r album; upload secret.jpg

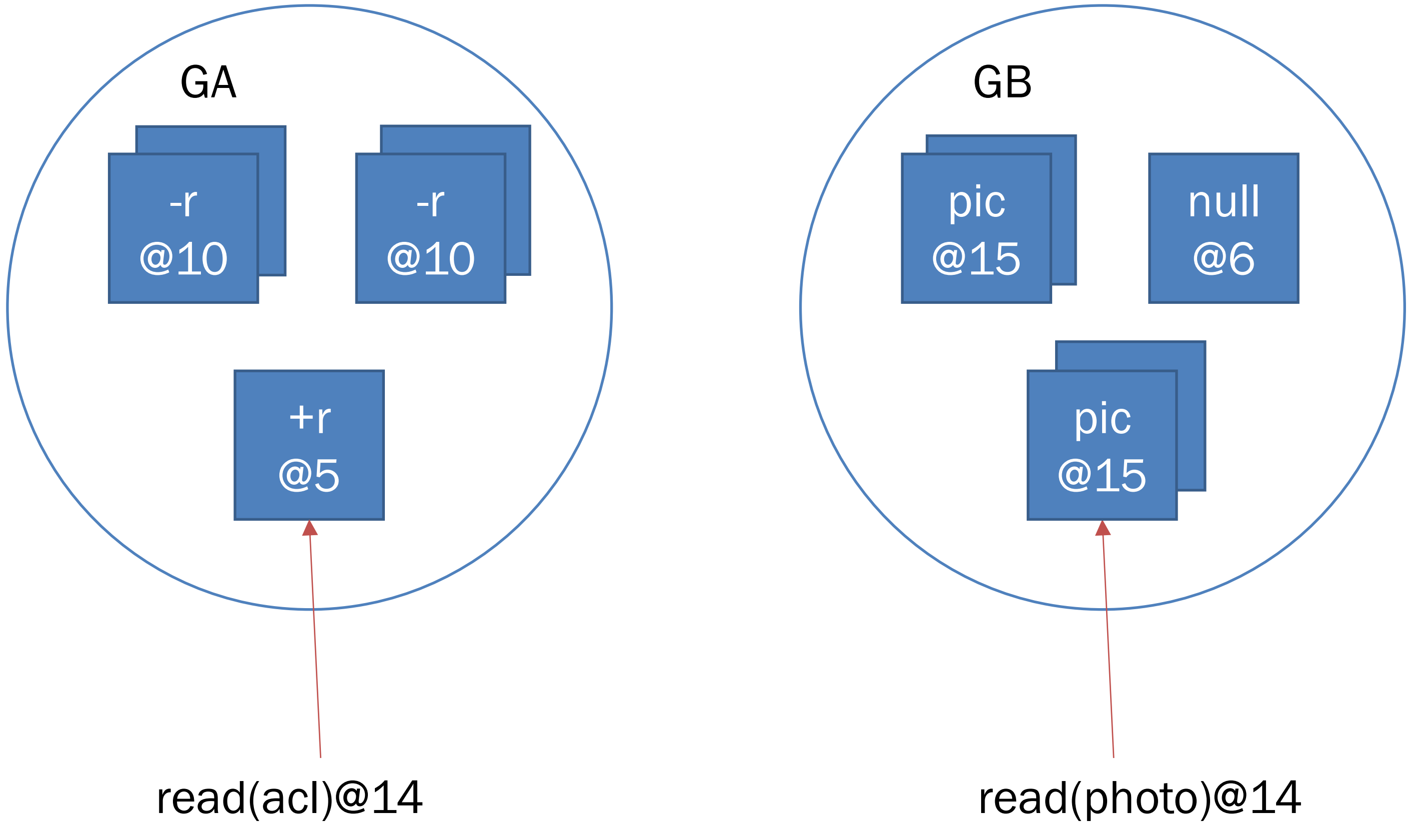


W1@10 chmod -r
W2@15 post photo



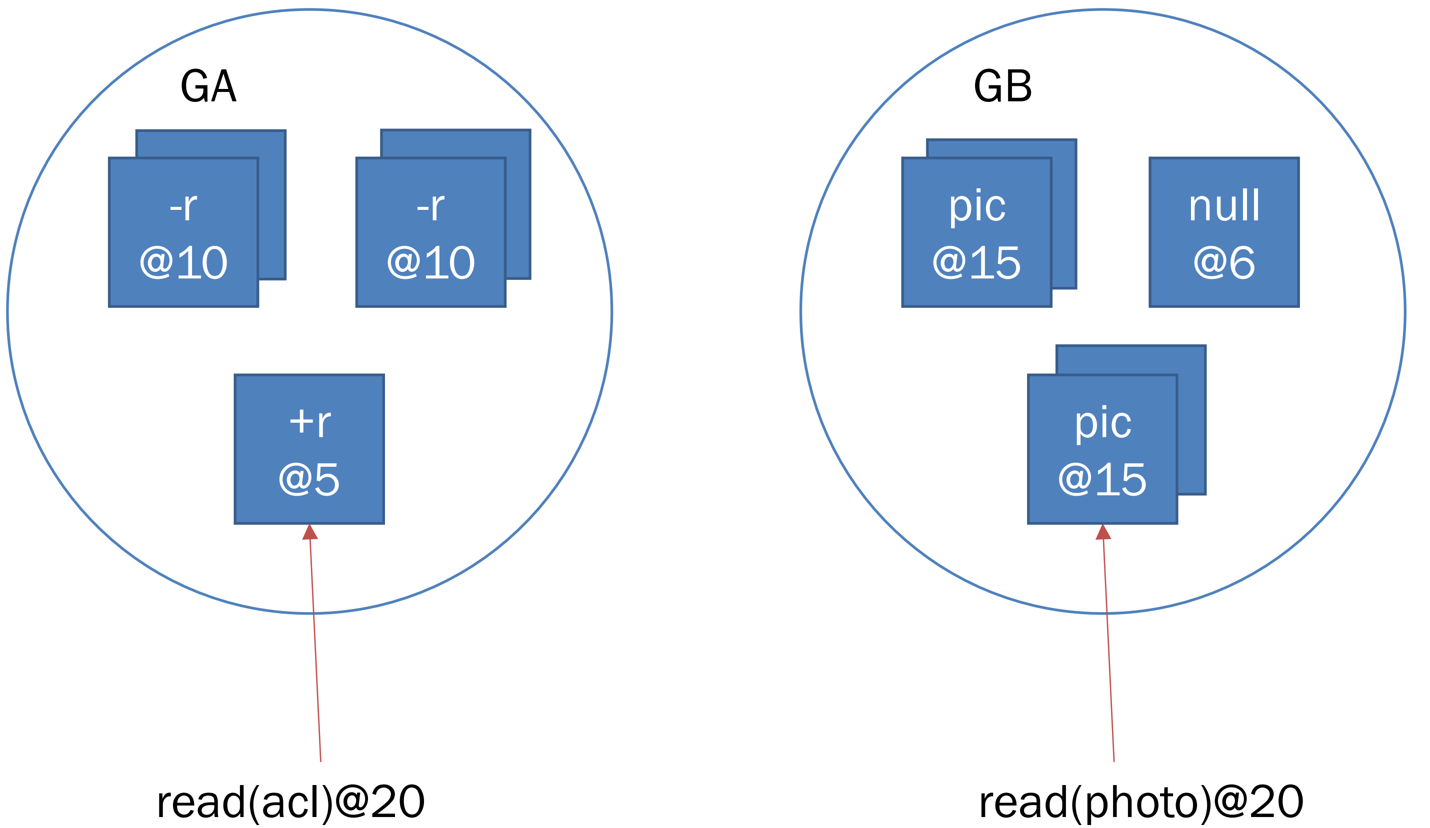
Can’t see photo @8. So, this is ok.

W1@10 chmod -r
W2@15 post photo



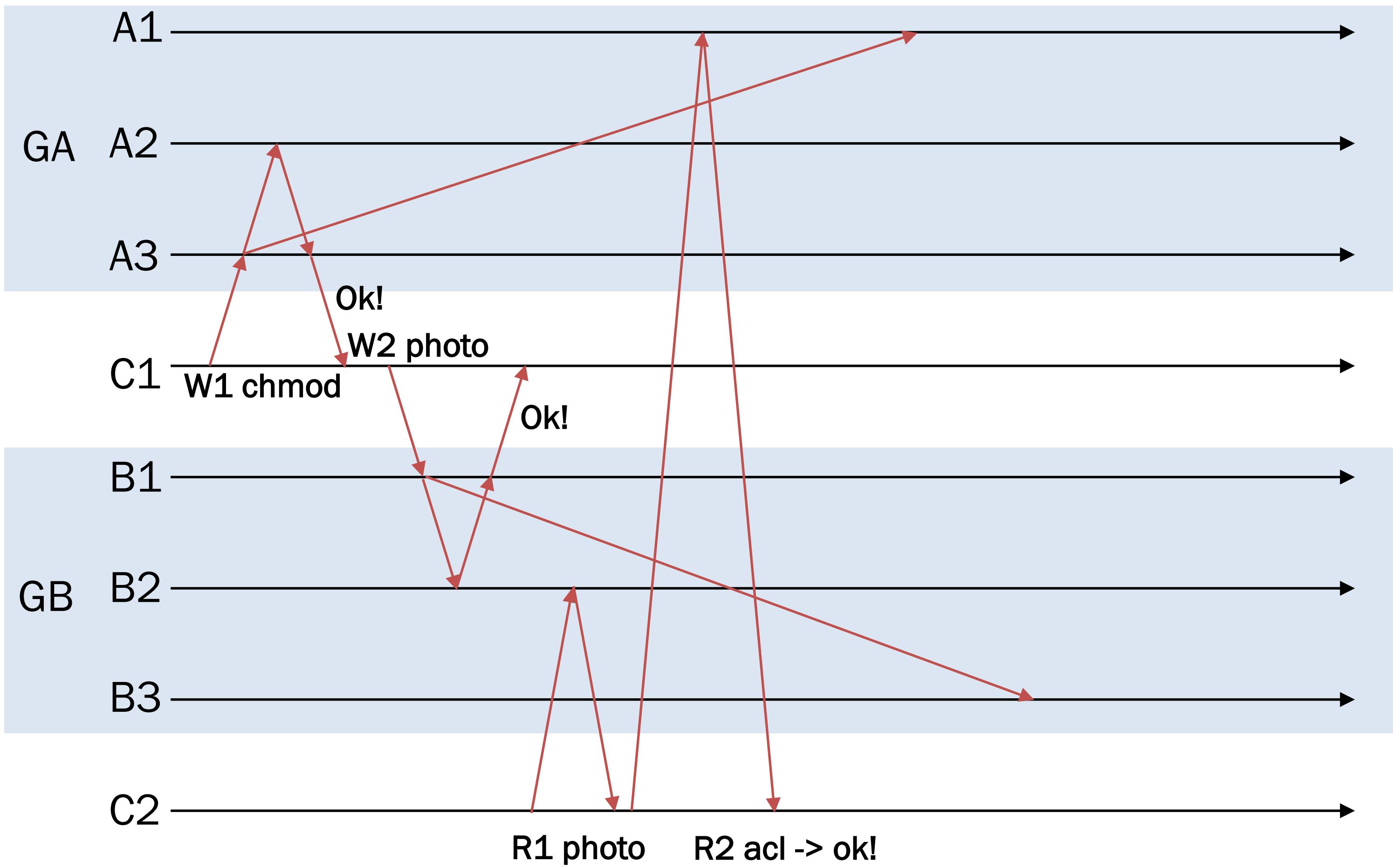
Can’t see photo @14. So this is ok.

W1@10 chmod -r
W2@15 post photo



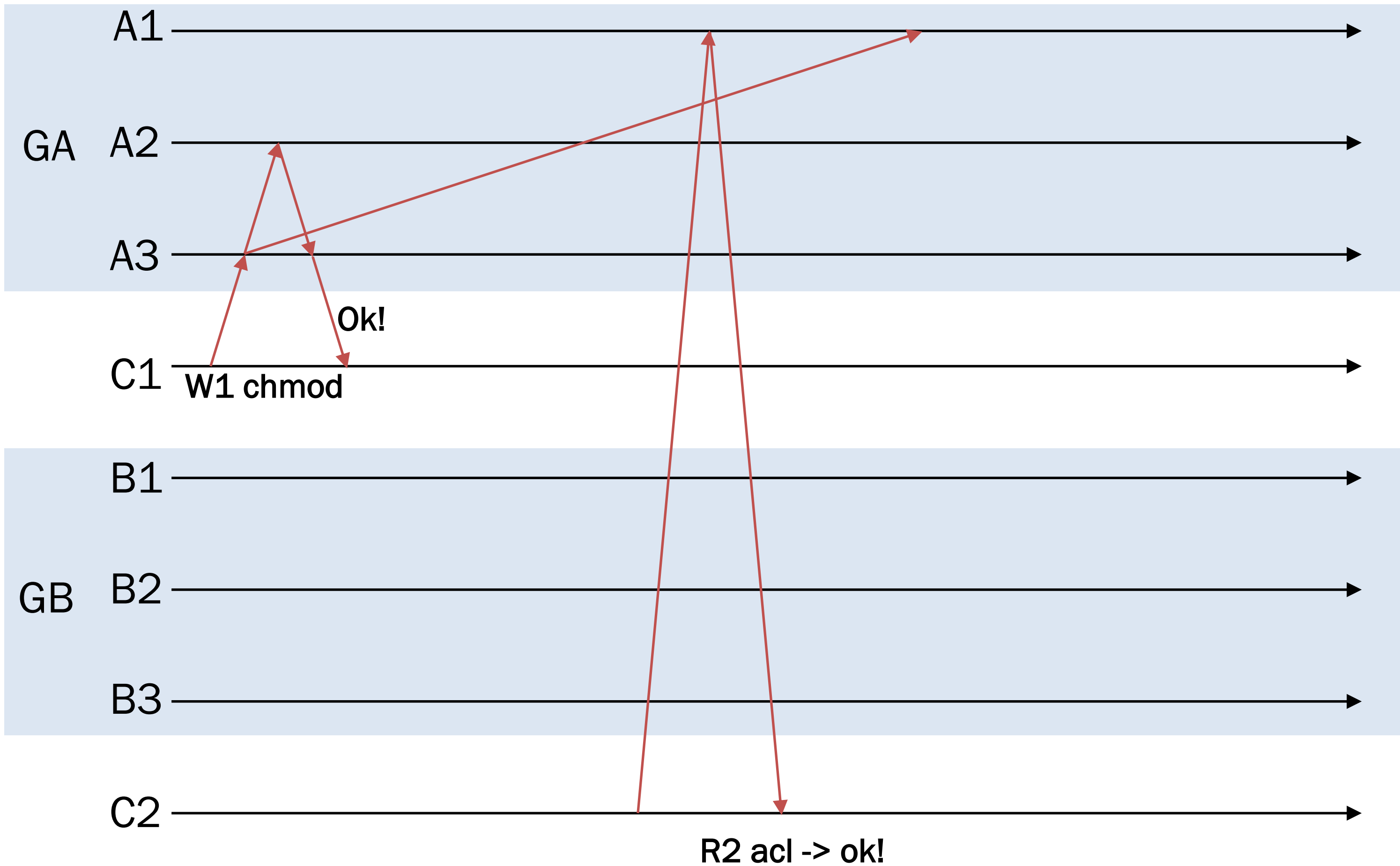
Can see photo @20, but then we must see +r as well, so access denied.

How can C2 tell A1 is out-of-date?



Tracking causality possible here!
(But, wouldn't be if chmod and photo add were by separate clients.)

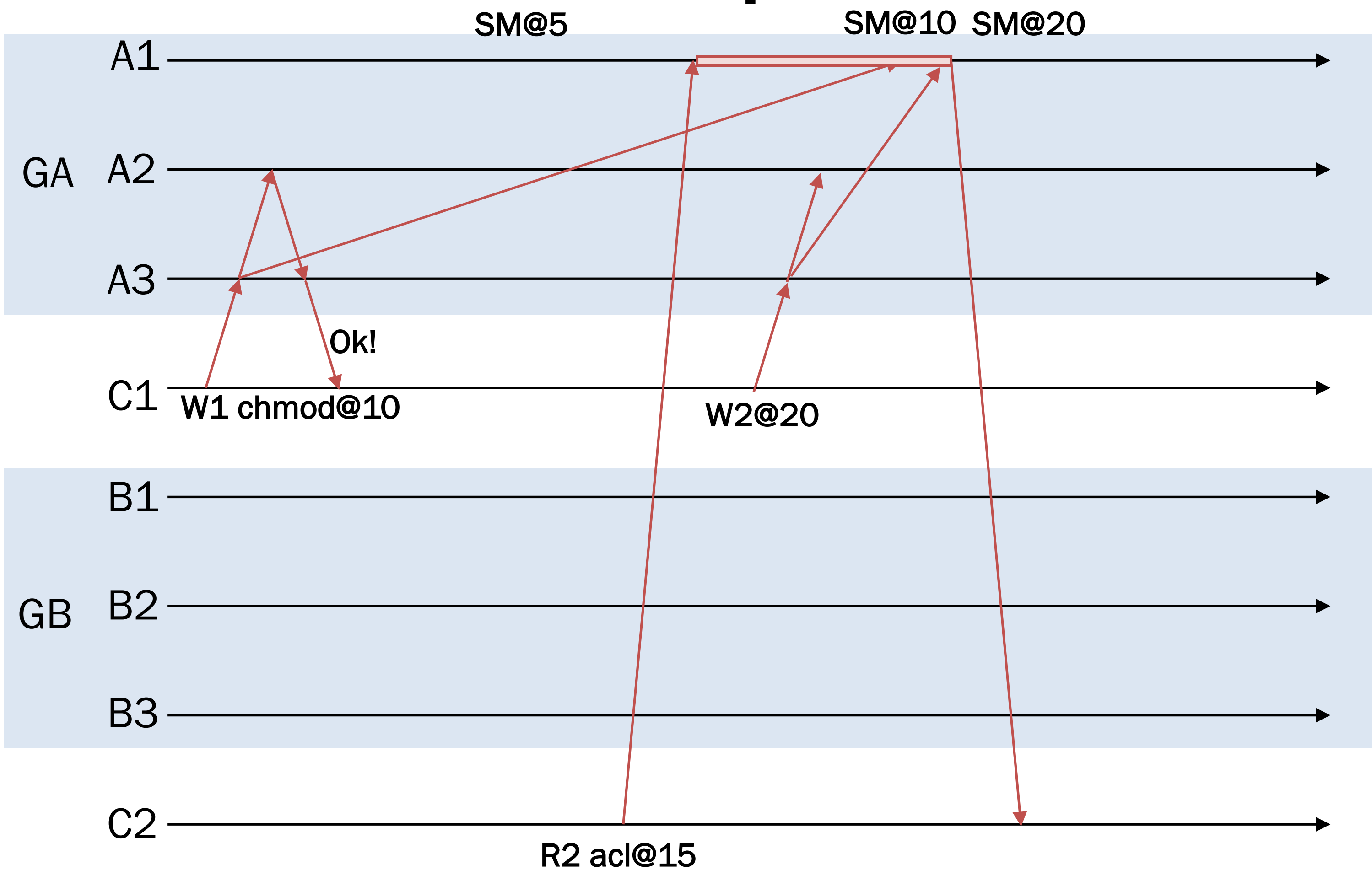
How can C2 tell A1 is out-of-date?



But R2 must see acked W1 even if no photo access!

Idea: delay reads at replica until sees a write with later ts

Stalls reads until SM up-to-date



But R2 must see acked W1 even if no photo access!

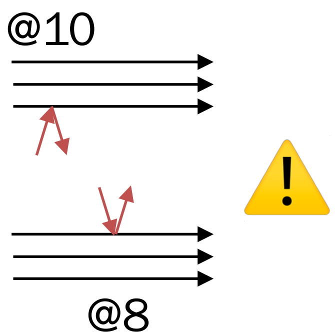
Idea: delay reads at replica until sees a write with later ts

What did we get?

- Externally consistent reads at all replicas, even followers
- Bonus: due to multi-versioning can pick timestamps in the past for (likely) non-blocking reads

One (big) problem

- Must also ensure all reads/writes (at any node) that don't overlap in wall time have increasing timestamps!



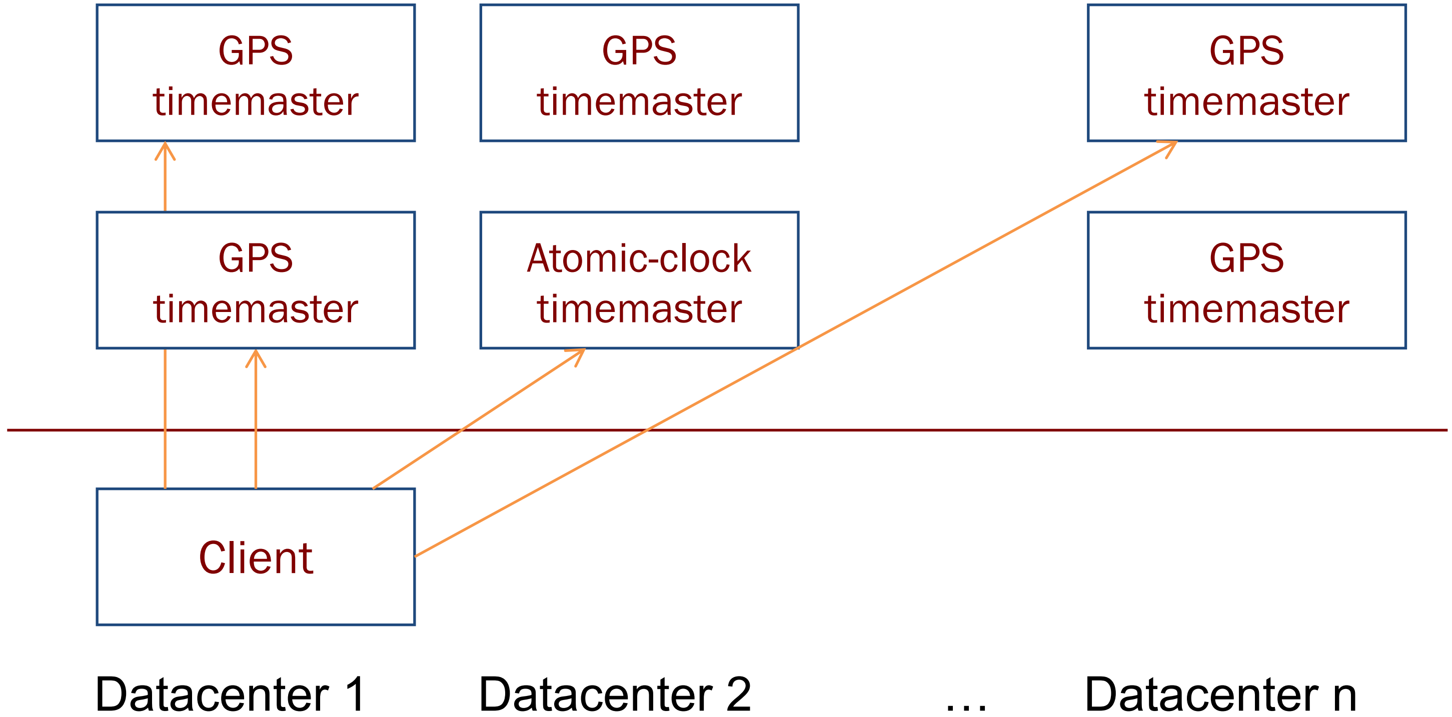
- Need ts(W1) < ts(W2) else photo may be visible
  - @8 here would assume SM with @9 ok to read
- How can we make this safe?
  - Communication, but possible with (uncertain) clocks?

Disruptive idea:

Do clocks really need to be arbitrarily unsynchronized?

Can you engineer some max divergence?

TrueTime Architecture



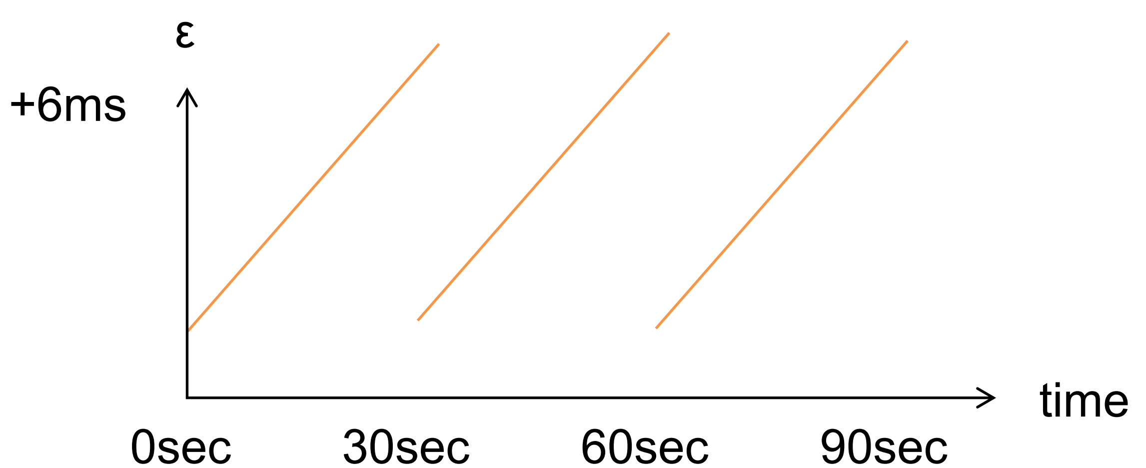
Compute reference [earliest, latest] = now ± ε

TrueTime implementation

now = reference now + local-clock offset

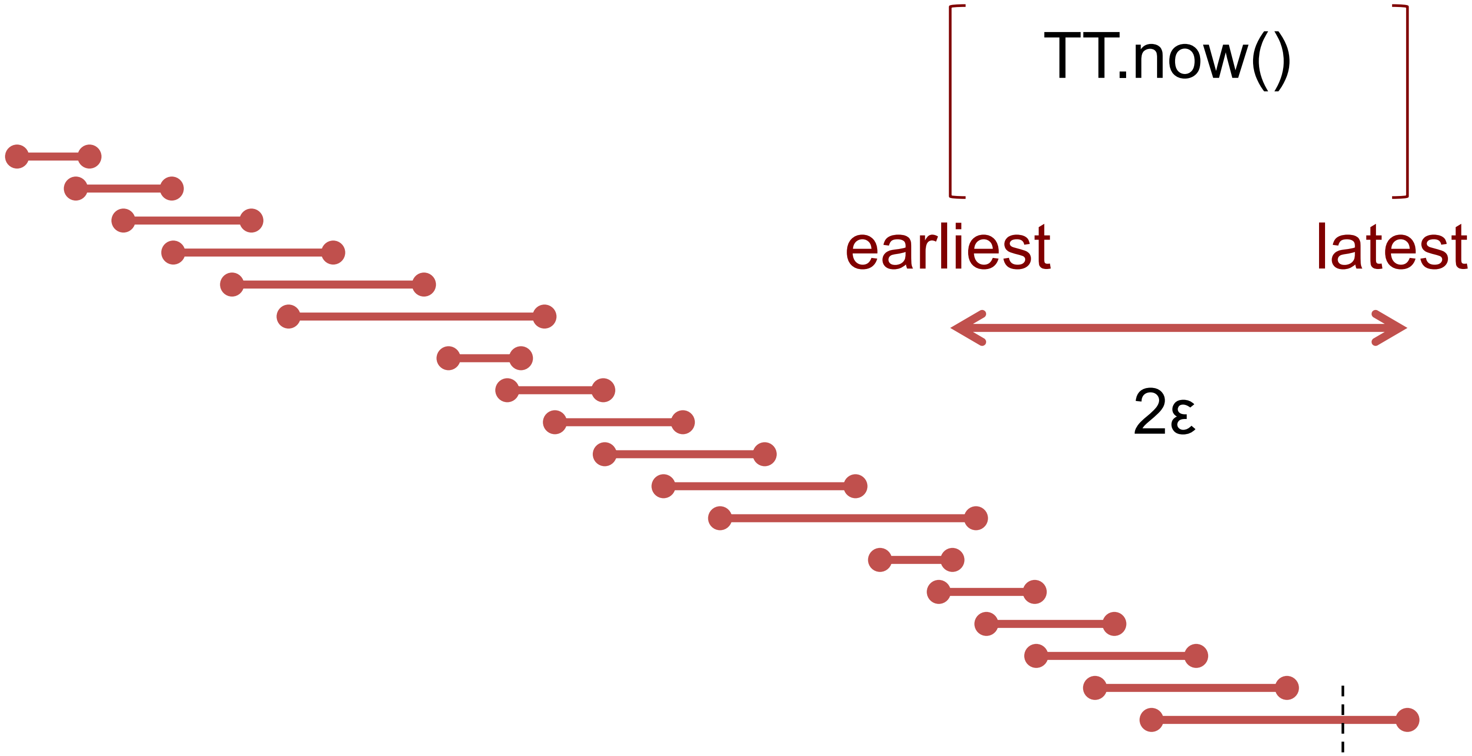
ε = reference ε + worst-case local-clock drift

= 1ms + 200 μs/sec



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

Progressive now() calls

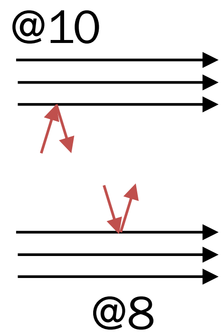


Consider event e_{now} which invoked tt = TT.now():

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

Remember the Goal

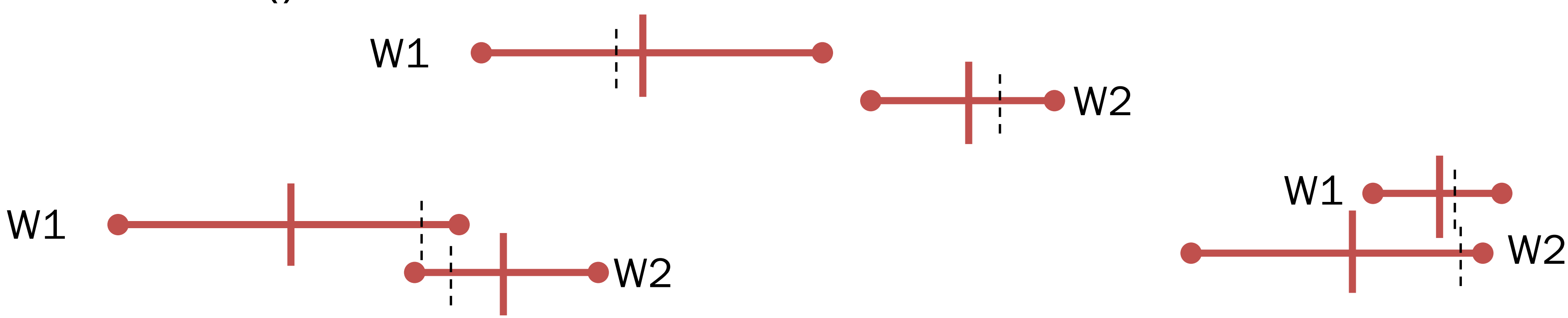
- Must also ensure all reads/writes (at any node) that don’t overlap in wall time have increasing timestamps!



- Need ts(W1) < ts(W2) else photo may be visible
  - @8 here would assume SM with @9 ok to read

Idea: Wait Out Uncertainty

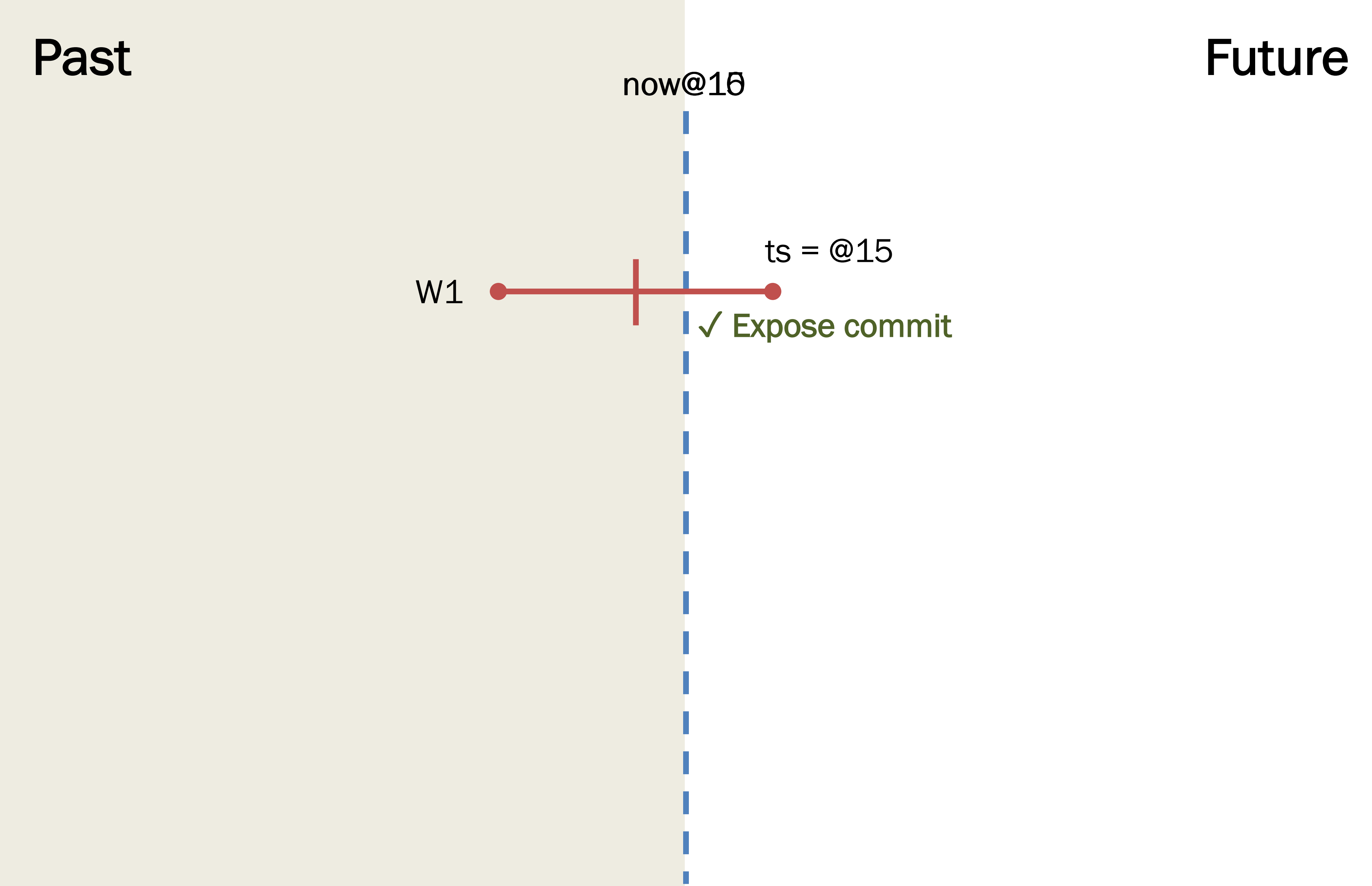
- TrueTime: TT.now() -> (earliest, latest)
- Guarantee: real-world time is in interval
- Imagine W1 and W2,
  W2 started after W1 finished
- now() return for both W1 and W2



- Choose ts(W1) and ts(W2)? Use latest? Earliest?

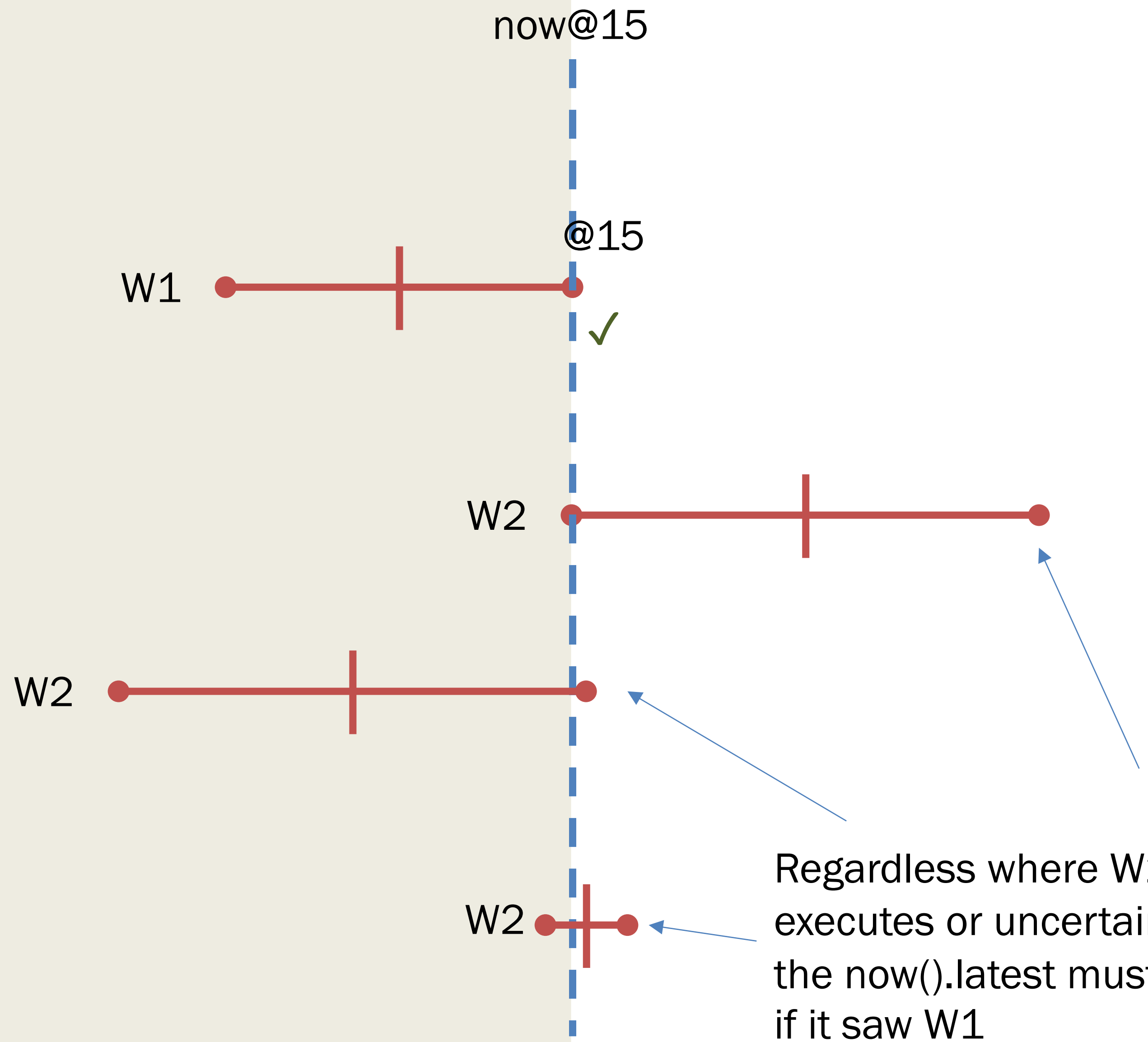
Commit Wait

- Use ts = now().latest
  - Enforce ts less than all op timestamps that start after it finishes?
- **Wait**; don’t expose write until ts *definitely* in past
  - Until ts < now().earliest
- *No* possible now().latest < ts on *any* machine
  - Remember: if a time is definitely in past it *cannot* be definitely in future (> now().latest) for any machine
  - Else violates invariant that for all server real time is in the range of now()


Past
Future
now@16
ts = @15
W1
✓ Expose commit

Past

Future

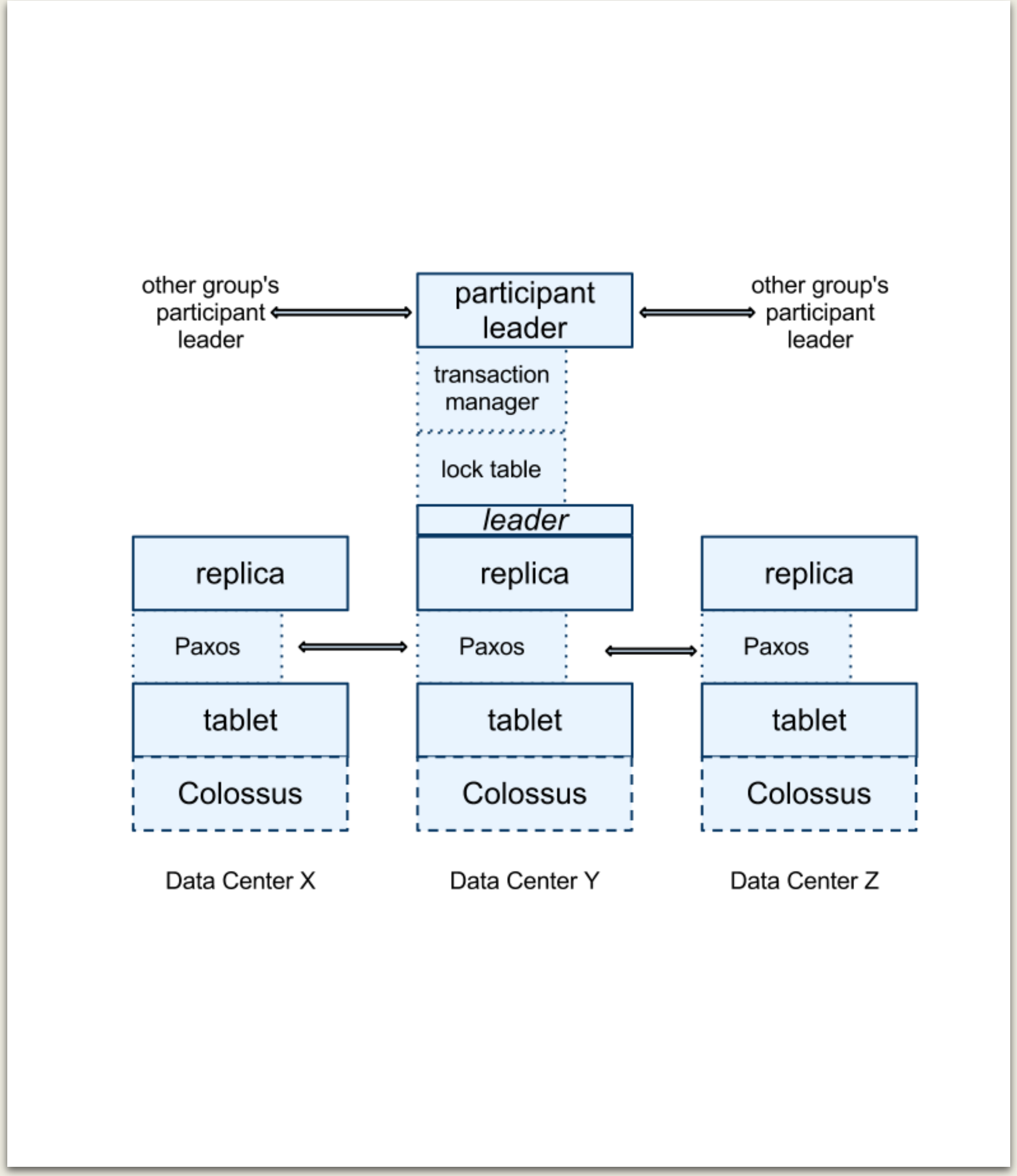


Regardless where W2 executes or uncertainty window, the now().latest must be > 15 if it saw W1

What did we get?

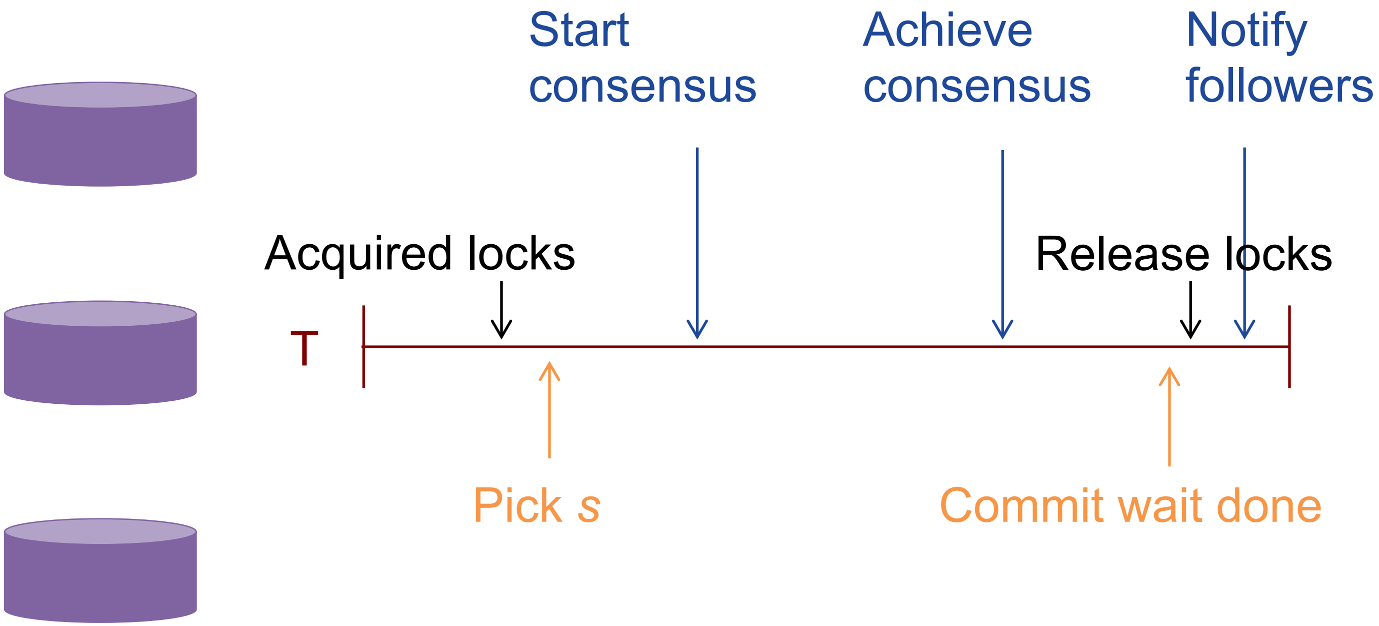
- Keep timestamped versions for all records
- Reads are similar
  - Pick ts = now().latest as timestamp to perform read
  - Wait until state machine has writes with higher timestamp and read there
- Also works to read at a consistent snapshot across partitions
- Read from Paxos replicas, 3 to 5x gain tput and consistent reads with no-x-DC comm
- Also, lock-free reads w/ongoing transactions

- All writes through leader
- Writes take locks at leader
- Both local and 2PC
- Non-transactional reads can skip leader/lock table
- Read-only txns can skip lock table too



Known unknowns →
Rethink algorithms to reason about uncertainty

Commit Wait and Replication



Client-driven cross-group transactions

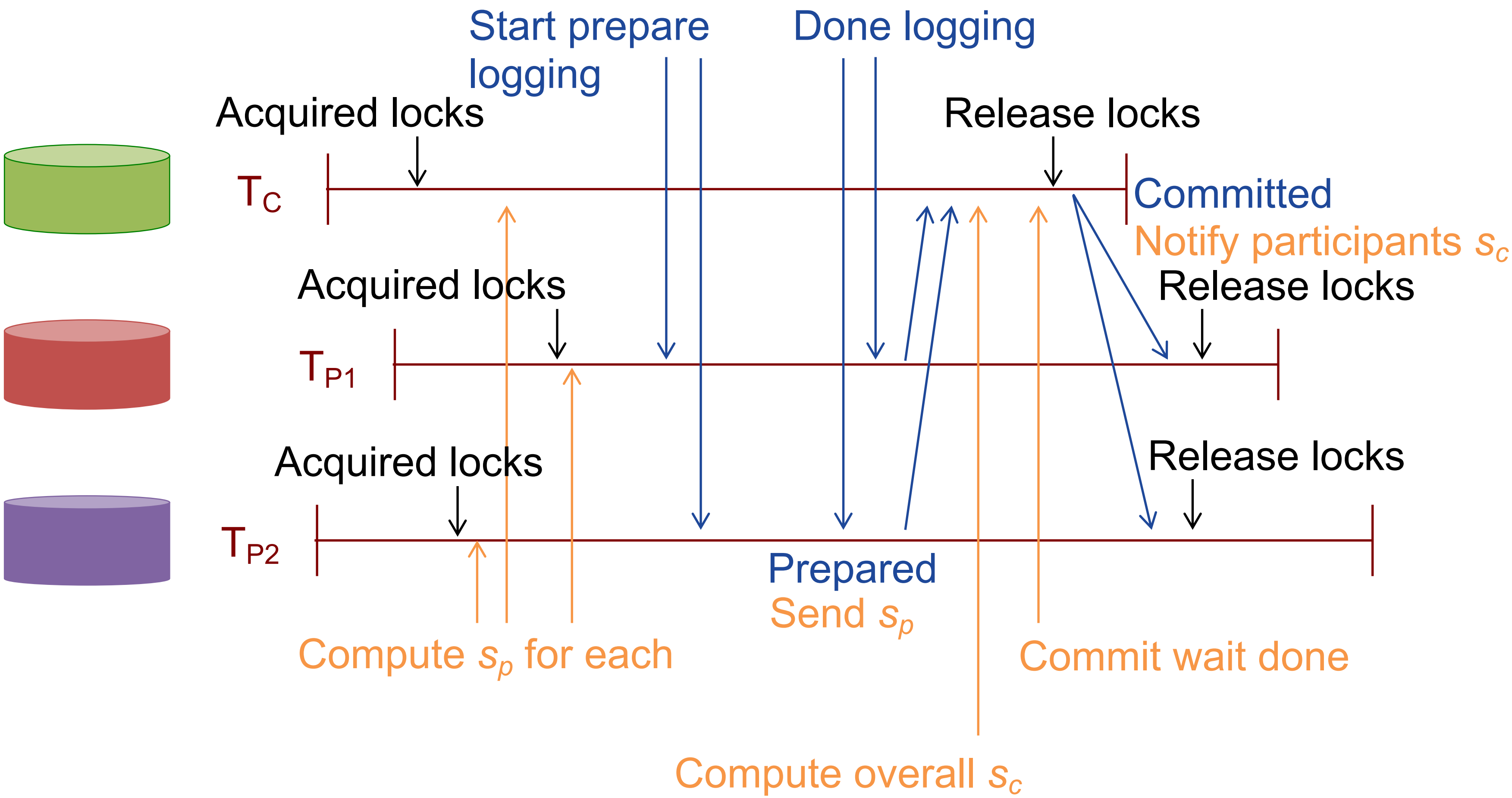
Client:

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

Commit Wait and 2-Phase Commit

- On commit msg from client, leaders acquire local write locks
  - If non-coordinator:
    - Choose prepare ts > previous local timestamps
    - Log prepare record through Paxos
    - Notify coordinator of prepare timestamp
  - If coordinator:
    - Wait until hear from other participants
    - Choose commit timestamp >= prepare ts, > local ts
    - Logs commit record through Paxos
    - Wait commit-wait period
    - Sends commit timestamp to replicas, other leaders, client
- All apply at commit timestamp and release locks

Commit Wait and 2-Phase Commit



Read-only optimizations

- Given global timestamp, can implement read-only transactions lock-free
- Step 1: Choose timestamp s_{read} = TT.now.latest()
- Step 2: Snapshot read (at s_{read}) to each tablet
  - Can be served by any up-to-date replica

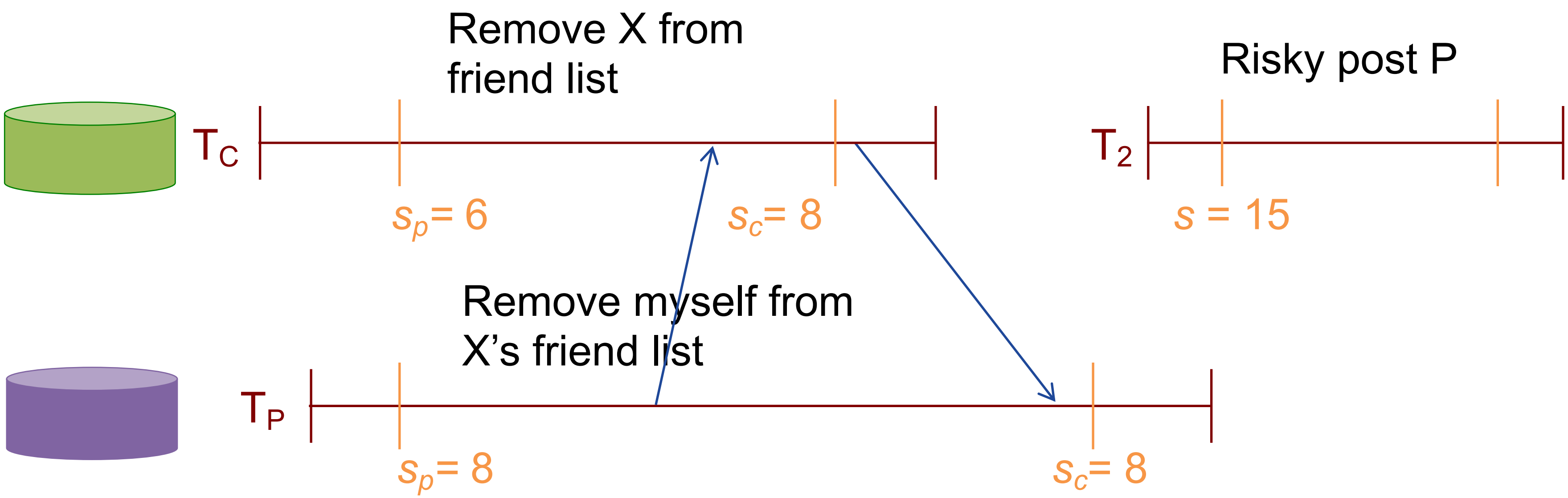
Disruptive ideas:

Do clocks really need to be arbitrarily unsynchronized?

Can you engineer some max divergence?

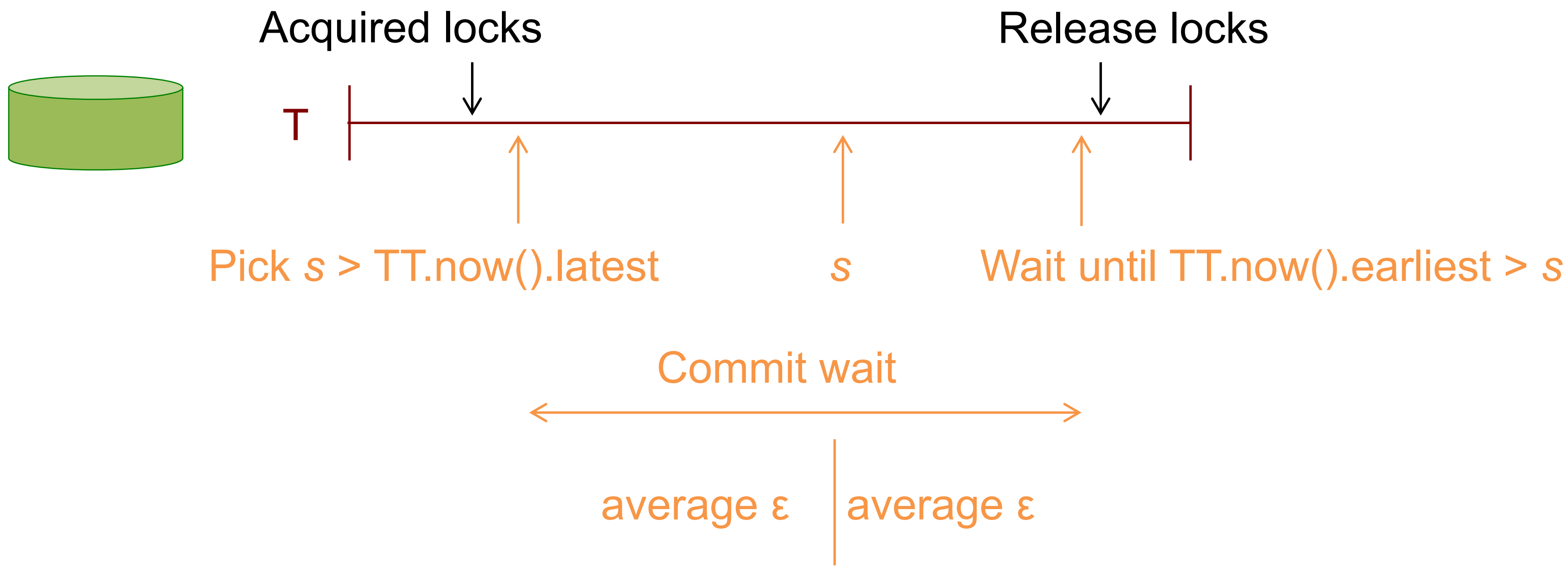
Known unknowns $\longrightarrow$
Rethink algorithms to reason about uncertainty

Example



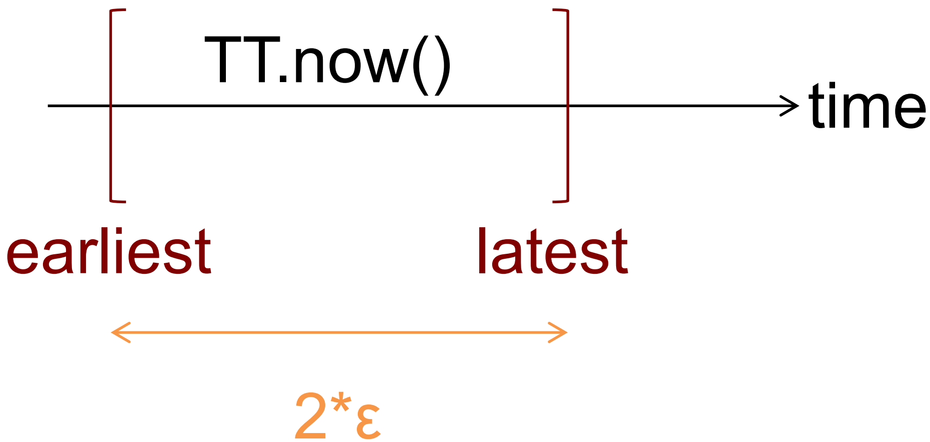
Time	<8	8	15
My friends	[X]	[]	
My posts			[P]
X's friends	[me]	[]	

Timestamps and TrueTime



TrueTime

- “Global wall-clock time” with bounded uncertainty



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

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