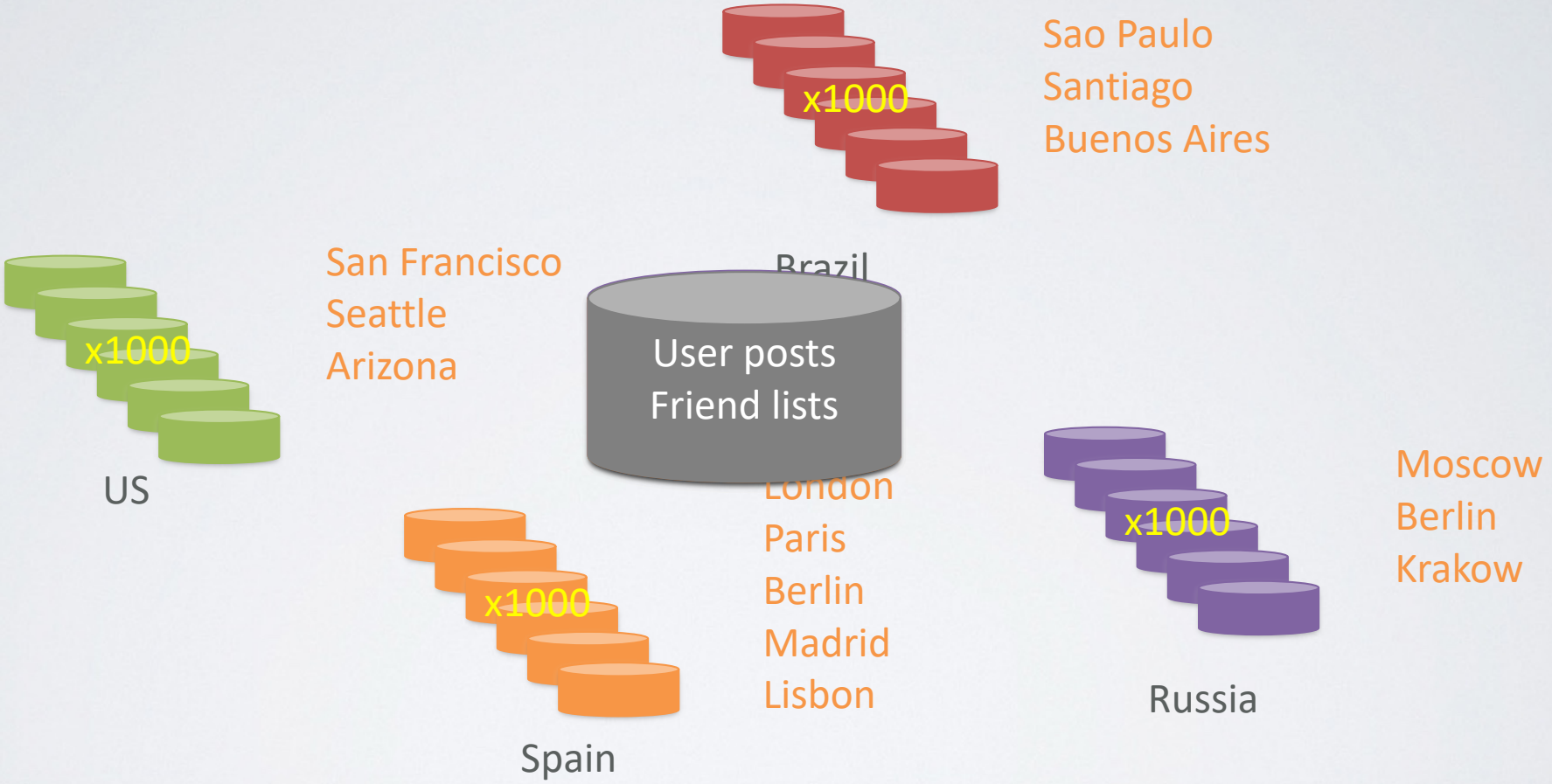


SPANNER

Corbett et al., OSDI 2012

- Distributed multiversion database
  - General purpose transactions
  - SQL / Semi-relational data model / Schemas / Tables
- Running in production
  - Storage for Google's ad data (!)

EXAMPLE: SOCIAL NETWORK



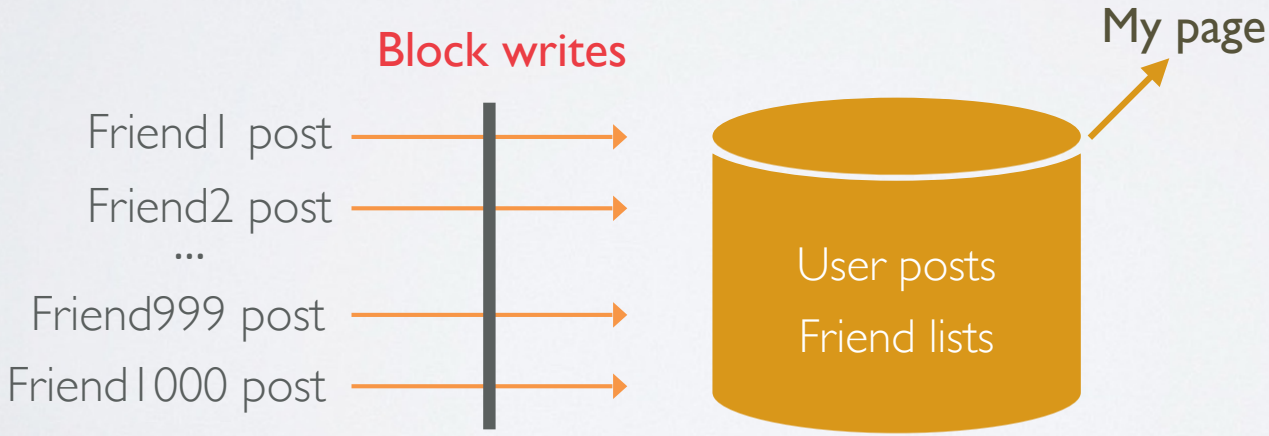
Figures adapted from [Wilson Hsieh and coauthors, OSDI 2012]

OVERVIEW

- Spanner offers:
  - Lock-free distributed read-only transactions
  - External consistency [strict serializability] for distributed transactions
    - Strictly serializable order consistent with real-time: transactions are "linearized" at their commit time
  - Integration of concurrency control, replication, and 2PC
- Enabling technology: TrueTime
  - exposes uncertainty about time by representing global time in terms of intervals

READ-ONLY TRANSACTIONS

- Suppose I want to generate a web page with my friends' recent posts
- From a single machine…



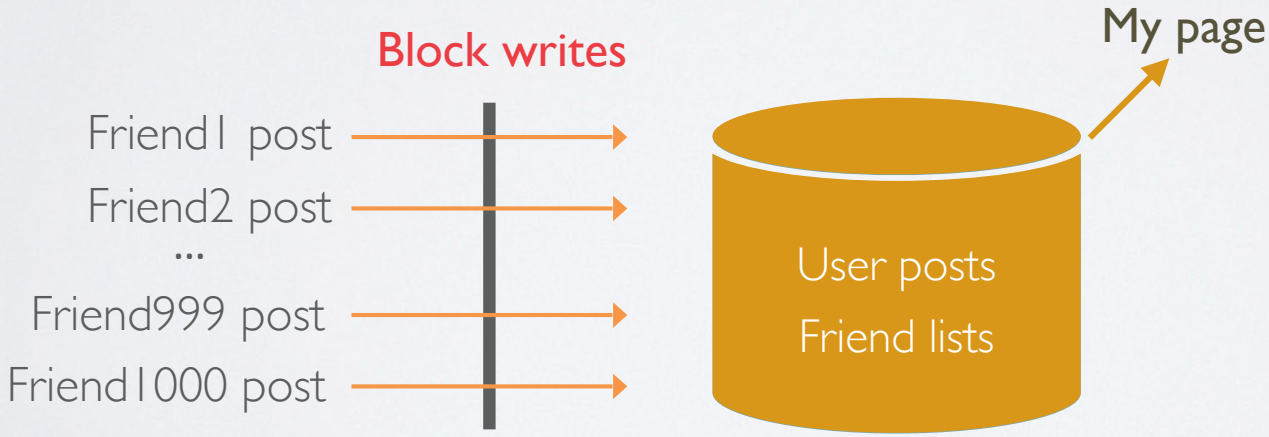
Read lock

Block all writes until read has finished

We'd rather not do that…

READ TRANSACTIONS

- Suppose I want to generate a web page with my friends' recent posts
- From a single machine…



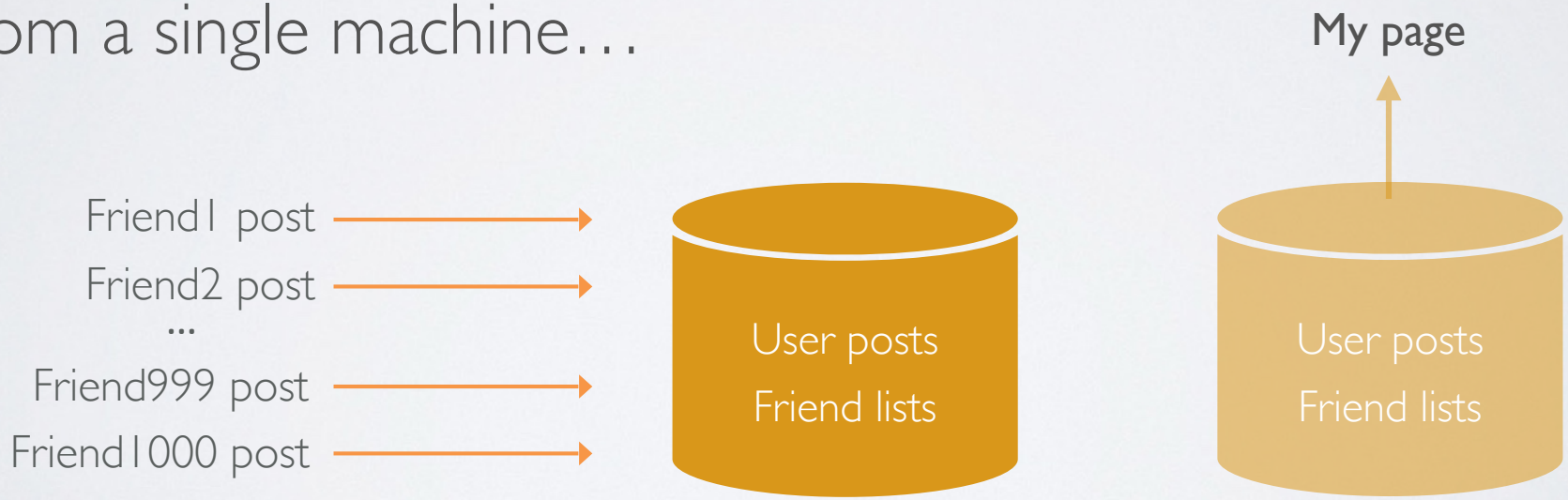
Read lock

Block all writes until read has finished

We'd rather not do that…

READ-ONLY TRANSACTIONS

- Suppose I want to generate a web page with my friends' recent posts
- From a single machine…



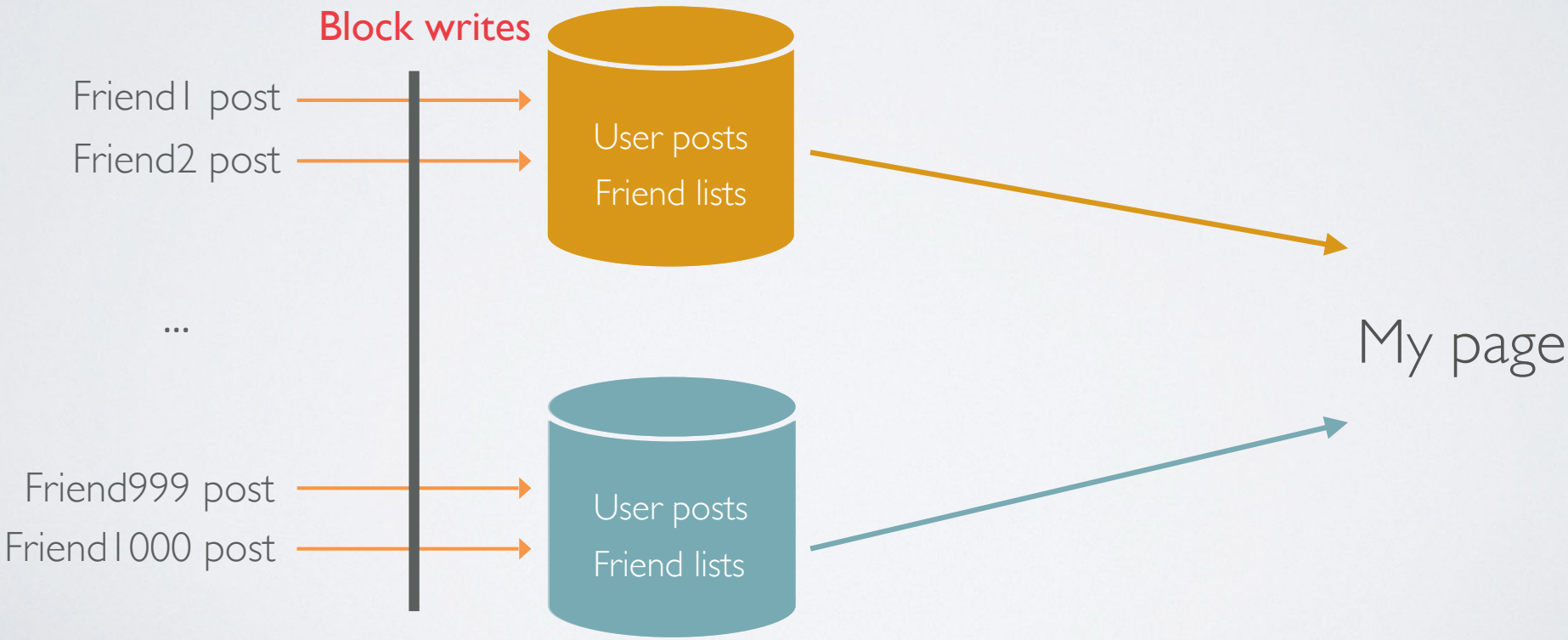
Better!

Snapshot

Read from DB copy,
writes continue to original DB

READ TRANSACTIONS

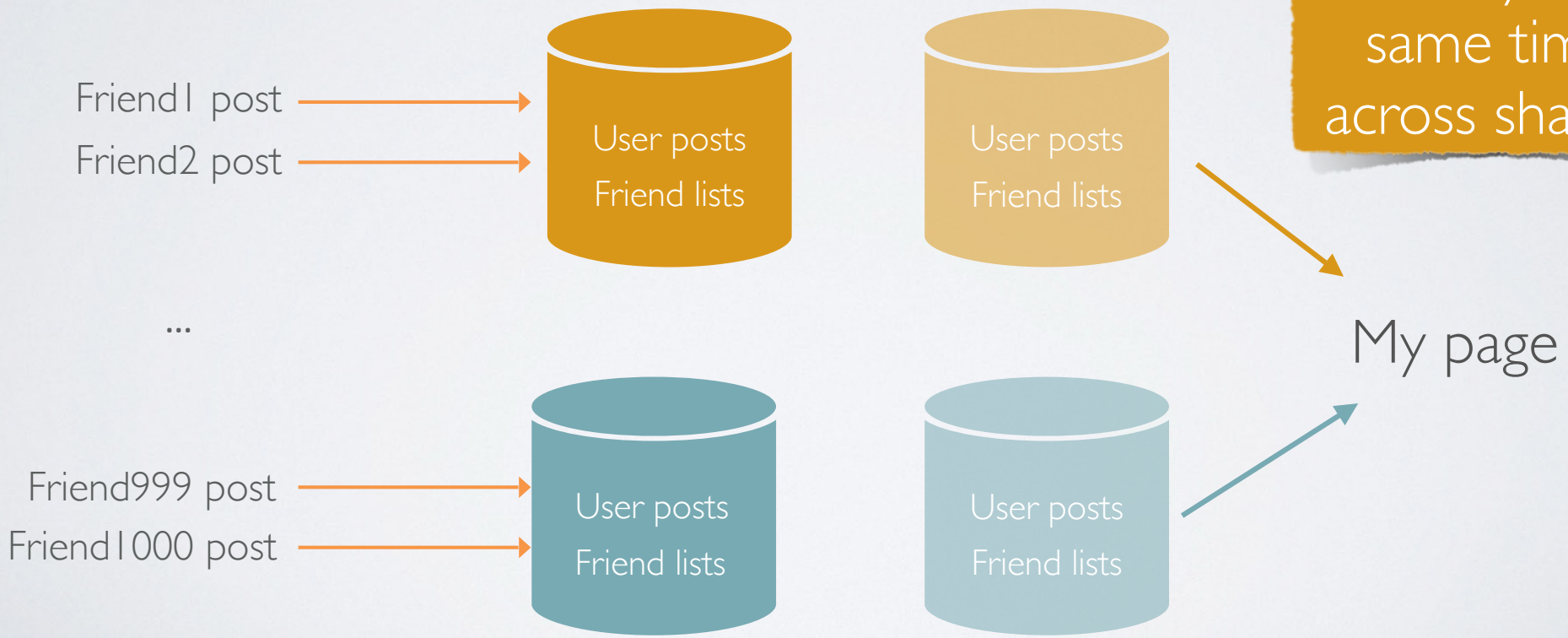
- From multiple machines…



READ TRANSACTIONS

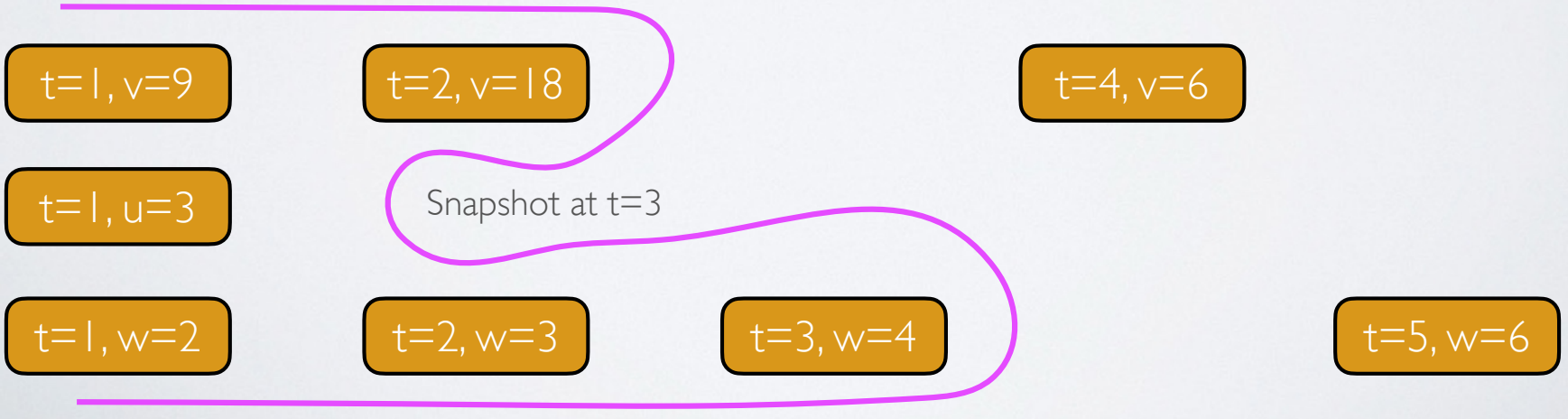
- From multiple machines…

Must create snapshot at exactly the same time across shards!



IMPLEMENTING SNAPSHOT READS

- Multi-version concurrency control
  - At commit time, create new versions of modified objects as tuples (timestamp, value)
  - Retain old (timestamp, value) tuples
  - Snapshot: read latest tuples with timestamp < now



HOW HARD IS TO SYNCHRONIZE CLOCKS?

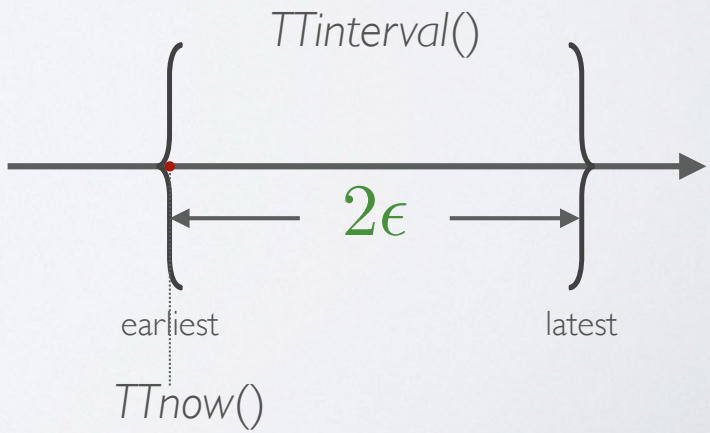
- Time synchronization error proportional to RTT!
- For geo-replicated data centers, that is in the hundreds of milliseconds!

TRUE TIME: TIME AS AN INTERVAL

- "Global wall-clock time" with bounded uncertainty

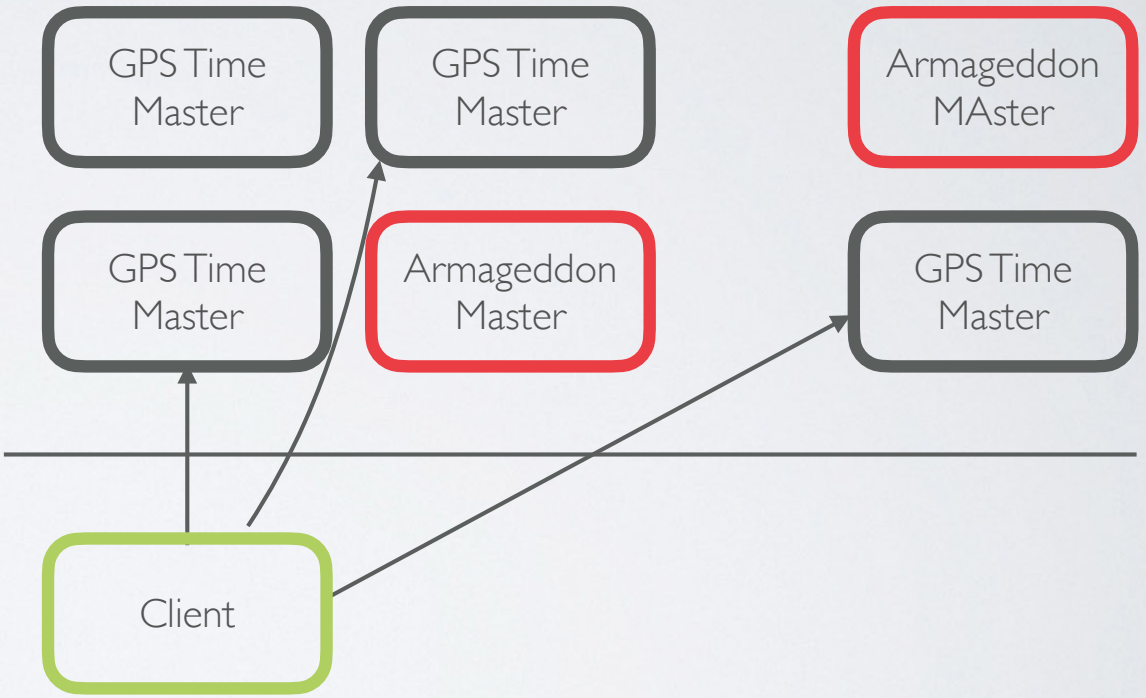
Method	Returns
TT.now()	*TTinterval*: [*earliest*, *latest*]
TT.after(*t*)	true if *t* has definitely passed
TT.before(*t*)	true if *t* has definitely not arrived

- The *TTinterval* returned by *TT.now*() is guaranteed to include the absolute time at which *TT.now*() was invoked
- Between synchronizations, ϵ grows from 1 to 7 ms (200μs/s drift, 30s poll, + 1 ms of communication delay from Time Master)



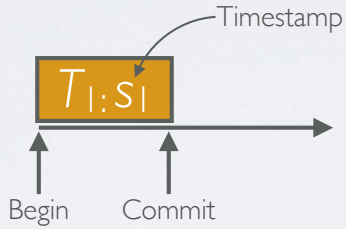
TIME-SYNCH IN SPANNER

- Each datacenter has a set of *time leader* (some with GPS antennas, others [the Armageddon leaders] with atomic clocks) and a *time follower daemon* per machine
- Each daemon *uses* a variant of *Marzullo's algorithm* (used also in NTP) to detect and reject liars, and synchronize to non liars



TRUE TIME AND STRICT SERIALIZABILTY

- Assign transactions a globally meaningful timestamp



- Commit transactions in timestamp order
- Ensure that if T_1 commits before T_2 begins, the $s_1 < s_2$
- Then reading a consistent state is easy:
  - to obtain a consistent snapshot at *t*, read state produced by transactions whose timestamp is at most *t*

HOW SHOULD WE PICK THE TIMESTAMP?

- Spanner uses strict two-phase locking for write transactions…
  - …assign timestamp while locks are held!



- But how? Let

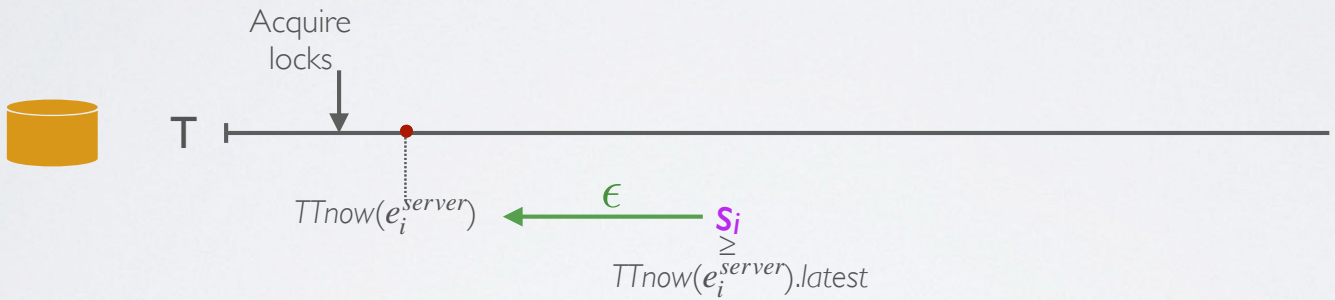
e_i^{begin} = T_i's begin event e_i^{commit} = T_i's commit event $t_{abs}(e)$ = e's absolute time

- Strict serializability requires:

$$t_{abs}(e_1^{commit}) < t_{abs}(e_2^{begin}) \Rightarrow s_1 < s_2$$

BUT TIME IS AN INTERVAL…

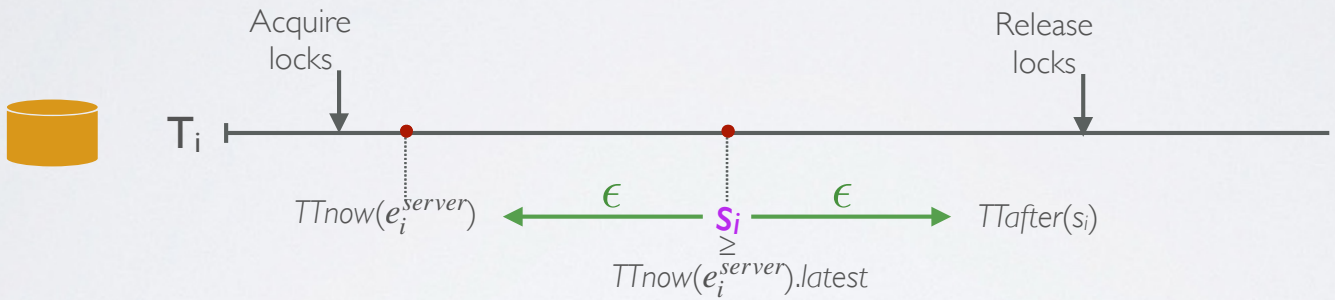
- Base T_i's timestamp on result of *TTnow*(e_i^{server}), where e_i^{server} is the event corresponding to T_i's coordinator receiving from the client the indication that it would like to commit T_i



- Choosing s_i at least as large *TTnow*().latest ensures that s is largest than the result of any *TTnow*() invoked before T_i begins

BUT TIME IS AN INTERVAL…

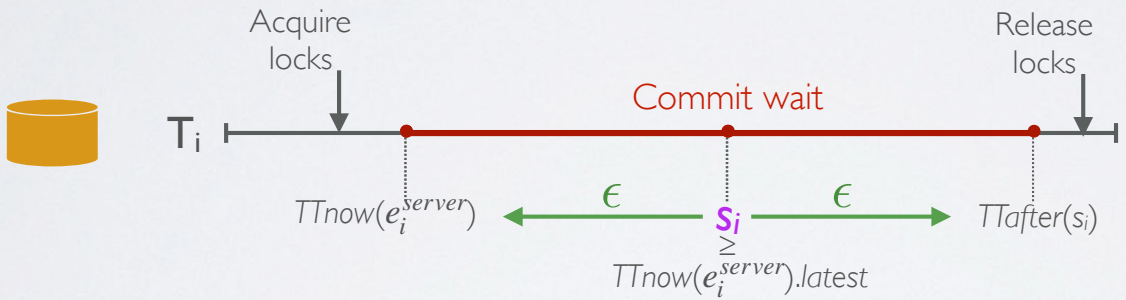
- Base T_i's timestamp on result of *TTnow*(e_i^{server}), where e_i^{server} is the event corresponding to T_i's coordinator receiving from the client the indication that it would like to commit T_i



- Choosing s_i at least as large *TTnow*().*latest* ensures that s is largest than the result of any *TTnow*() invoked before T_i begins

BUT TIME IS AN INTERVAL…

- Base T_i's timestamp on result of *TTnow*(e_i^{server}), where e_i^{server} is the event corresponding to T_i's coordinator receiving from the client the indication that it would like to commit T_i



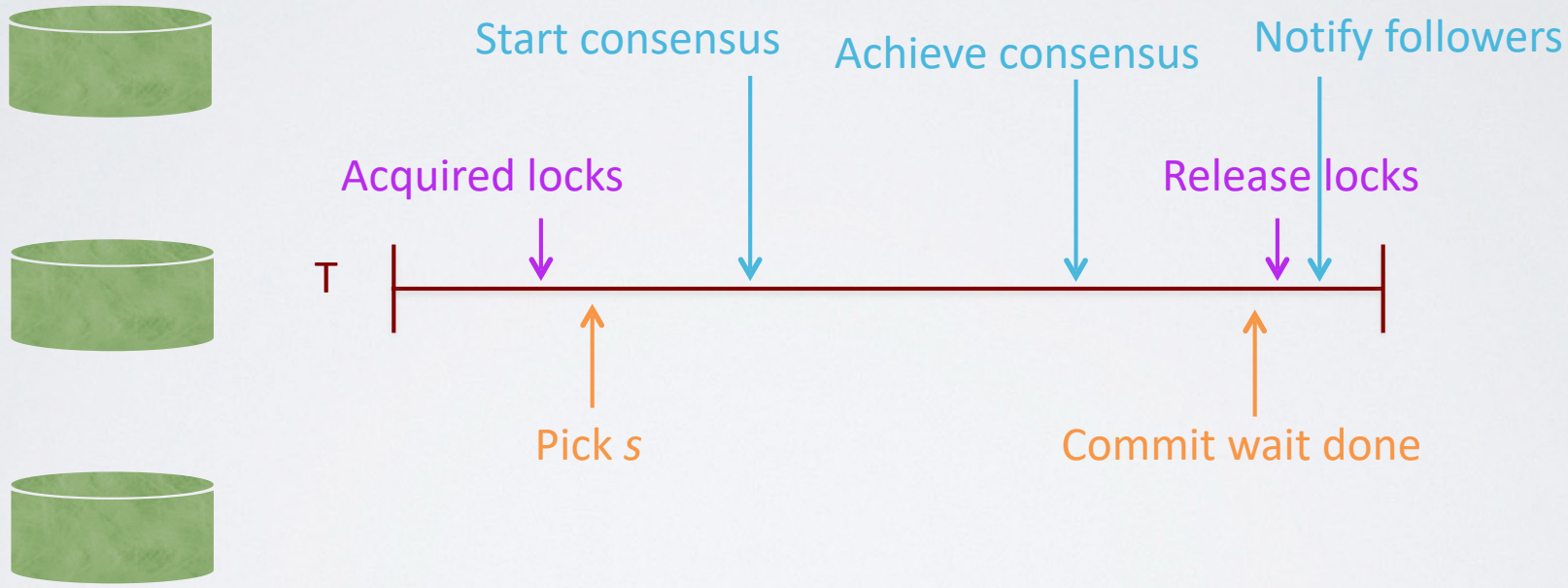
- Choosing s_i at least as large *TTnow*().*latest* ensures that s is largest than the result of any *TTnow*() invoked before T_i begins
- Releasing locks and committing after *TTafter*(s_i) ensures clients cannot see any data committed by T_i until s_i has definitely passed.

ENFORCING STRICT SERIALIZABILITY

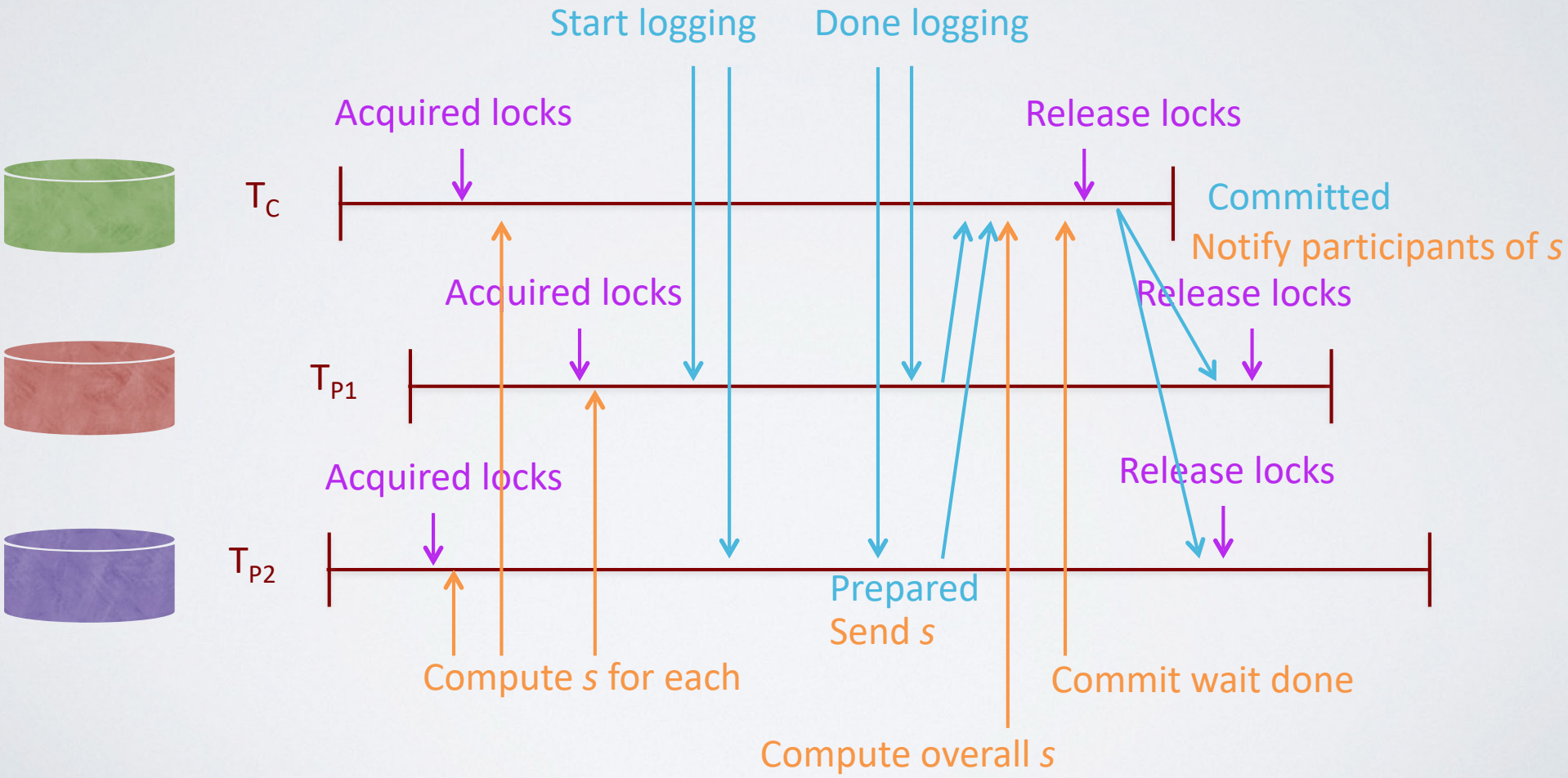
$$t_{abs}(e_1^{commit}) < t_{abs}(e_2^{begin}) \Rightarrow s_1 < s_2$$

$s_1 < t_{abs}(e_1^{commit})$	(commit wait)
$t_{abs}(e_1^{commit}) < t_{abs}(e_2^{begin})$	(assumption)
$t_{abs}(e_2^{begin}) \leq t_{abs}(e_2^{server})$	(causality)
$t_{abs}(e_2^{server}) \leq s_2$	(start)
$s_1 \leq s_2$	(transitivity)

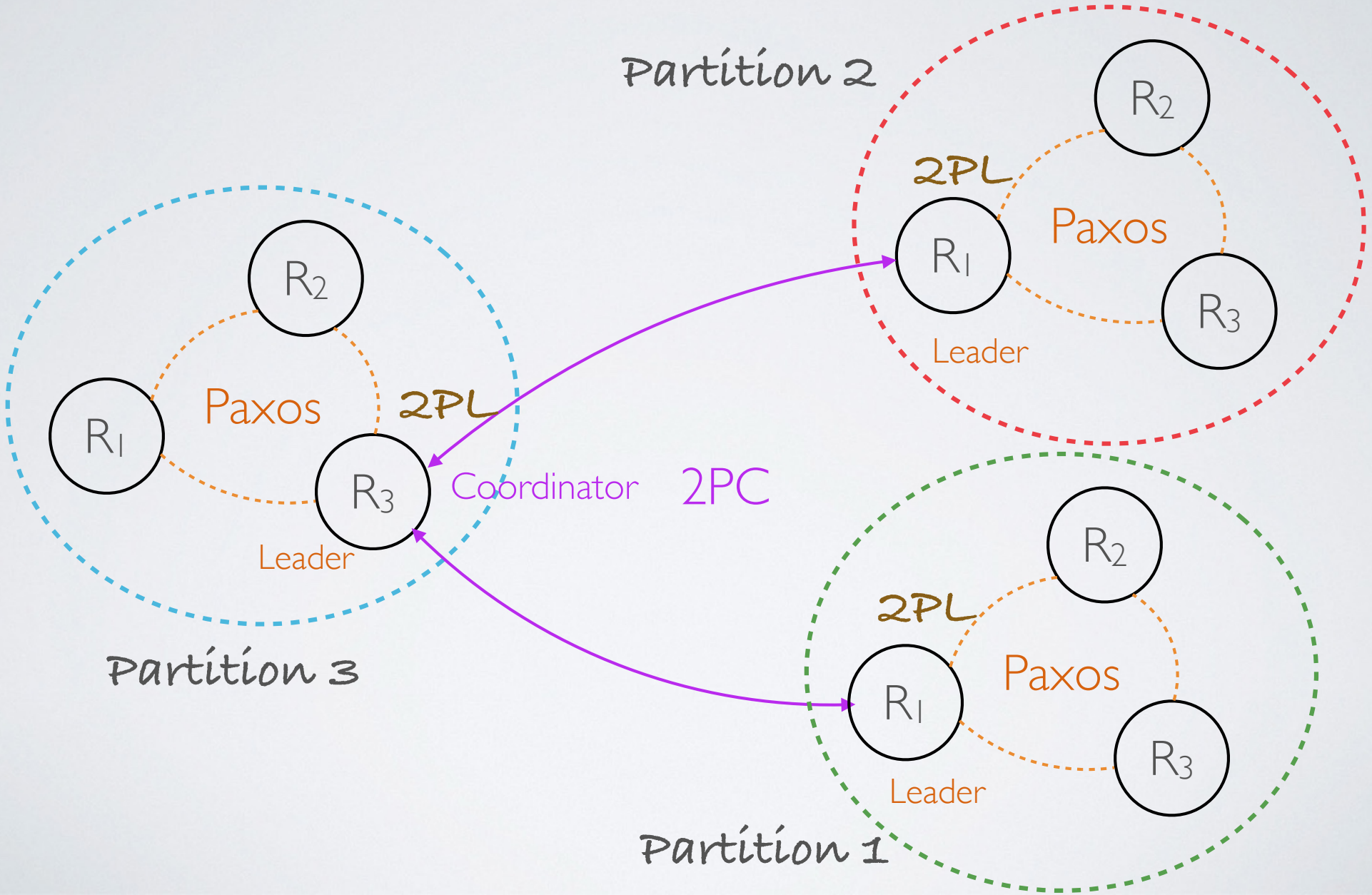
COMMIT WAIT AND REPLICATION



COMMIT WAIT AND 2-PHASE COMMIT



THE BIG PICTURE: RW



THE BIG PICTURE: RO

- Read from a snapshot read at time t
  - for scalability, go to any replica, not just the leader
- Use TrueTime to ensure that, no matter which replica we use, we are sure to see every version with timestamp at most t_{safe}
- Block reading until $t_{safe} \geq t$