▼ (Aside) Supporting our daily digital life: a case study of Google Spanner

1 ✦ (Aside) Supporting our daily digital life: a case study of Google Spanner

2 ✦ The topics of today's lecture

▼ Review: scaling an e-commerce website

3 ✦ Suppose we have an e-commerce website

4 ✦ First step: decide where the data lives

5 ✦ Second step: scale the data with sharding

6 ✦ Data scaling challenge #1: how to shard

7 ✦ Data scaling challenge #2: multi-shard transaction

8 ✦ Data scaling challenge #3: failures are the normal case

9 ✦ The challenge of replication: consistency

10 ✦ More issues: how do we serve a user?

11 ✦ Scaling the frontend with concurrency

12 ✦ Concurrency challenge: isolation (before-or-after atomicity)

13 ✦ Put it all together: from one machine to a distributed database

▼ Read-write transactions in Spanner

14 ✦ High-level overview of Spanner's read-write TX

15 ✦ Execution flow of read-write TX: the setting

16 ✦ Execution flow of read-write TX: 2PL

17 ✦ Execution flow of read-write TX: 2PC

18 ✦ Read-write TX: put it all together

19 ✦ Read-write TX: it gives a strong abstraction

20 ✦ Read-write TX of Spanner so far

▼ Read-only transactions and MVCC

21 ✦ Problem: 2PL does not fit read-only TX

22 ✦ Review: multi-version concurrency control

23 ✦ Idea: use 2PL for read-write TX, MVCC for read-only TX

24 ✦ MV-2PL: read-write TX (w/o 2PC & Paxos for simplicity)

25 ✦ MV-2PL: read-only TX

26 ✦ Question remains: how do we assign the time to TXs?

▼ Timestamps in geo-replicated transactions

27 ✦ Global time is inefficient for Spanner's use case

28 ✦ Possible solution: Cache the time locally to avoid querying the global counter

29 ✦ Cache the global counter to avoid frequent reads

30 ✦ Example revisit w/ cached time

31 ✦ Drawbacks of cached time

32 ✦ Challenge: implement external consistency without a global clock

▼ Spanner's TrueTime

33 ✦ Observation: an accurate bound (interval)

34 ✦ TrueTime API of Spanner

35 ✦ TrueTime gives external consistency: commit wait

36 ✦ TrueTime: how to achieve the bound?

37 ✦ TrueTime: how to calculate the bound?

38 ✦ Commit wait revisited

39 ✦ TrueTime adopts multiple time servers

40 ✦ Timemasters are backed by GPS & atomic clocks

41 ✦ Final takeaway: network-induced uncertainty

▼ Summary

42 ✦ Summary of Spanner

43 ✦ 参考文献

IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

(Aside) Supporting our daily digital life: a case study of Google Spanner

IPADS, Shanghai Jiao Tong University

The topics of today's lecture

Part 1 Review: scaling an e-commerce website

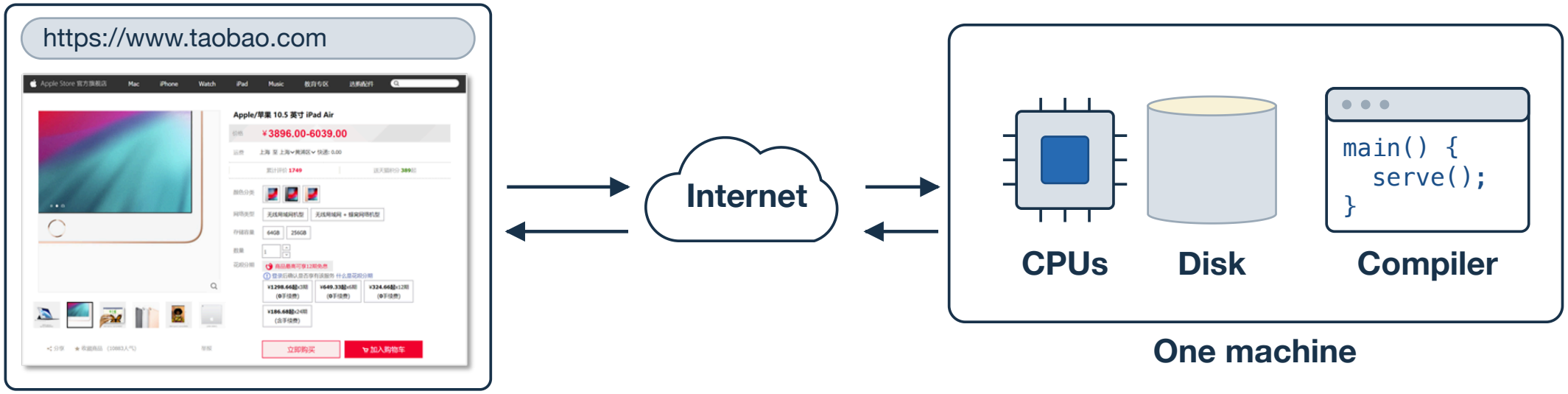
- All the techniques we have learned so far drive the scaling of large-scale e-commerce websites
- And the techniques apply to all large-scale websites
- We will further study a concrete example: Spanner

Part 2 — Google Spanner: one system that puts them all together

- How a read-write TX (placing an order) runs: 2PL + 2PC + Paxos
- How read-only TXs avoid locks: MVCC, and the TrueTime

Suppose we have an e-commerce website

At the start you only have a bare machine: disk, CPUs, and a compiler



How can you host such an e-commerce website?

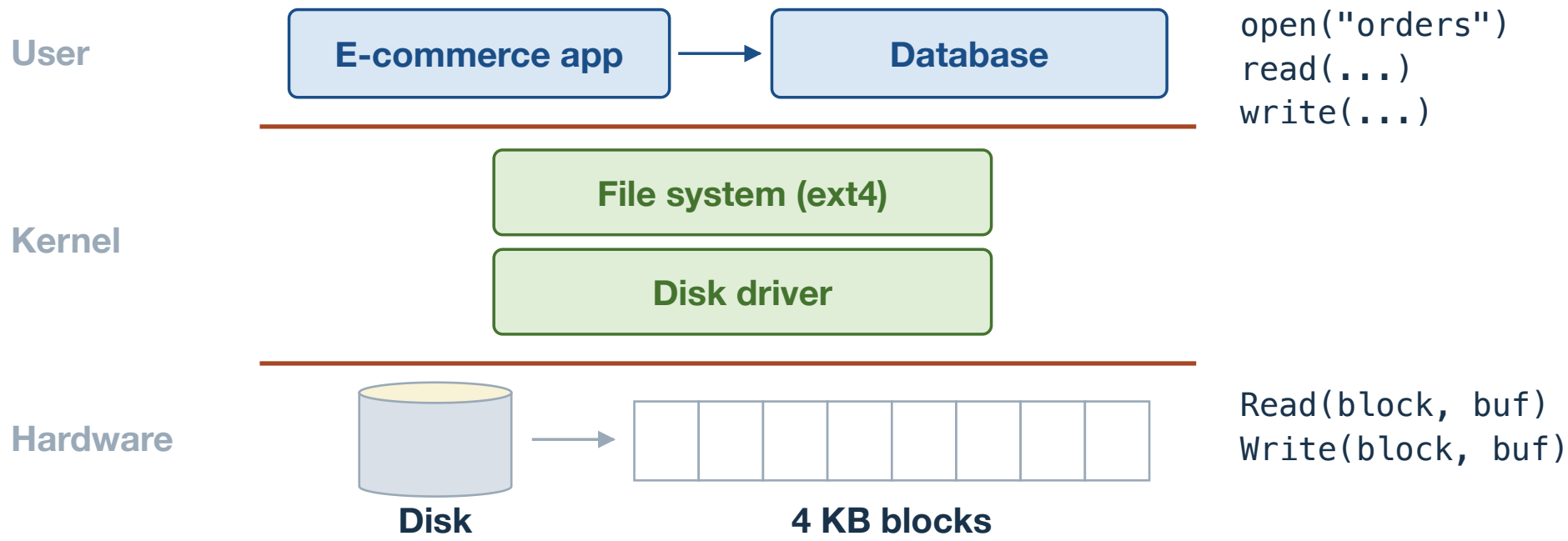
First step: decide where the data lives

The data is first organized into files

- Why files? The data must be stored durably
- Linux gives us files for free (the *Filesystem* lectures), e.g., `ext4`

A database is built on top of files

- Tables to organize the data, transactions to update it



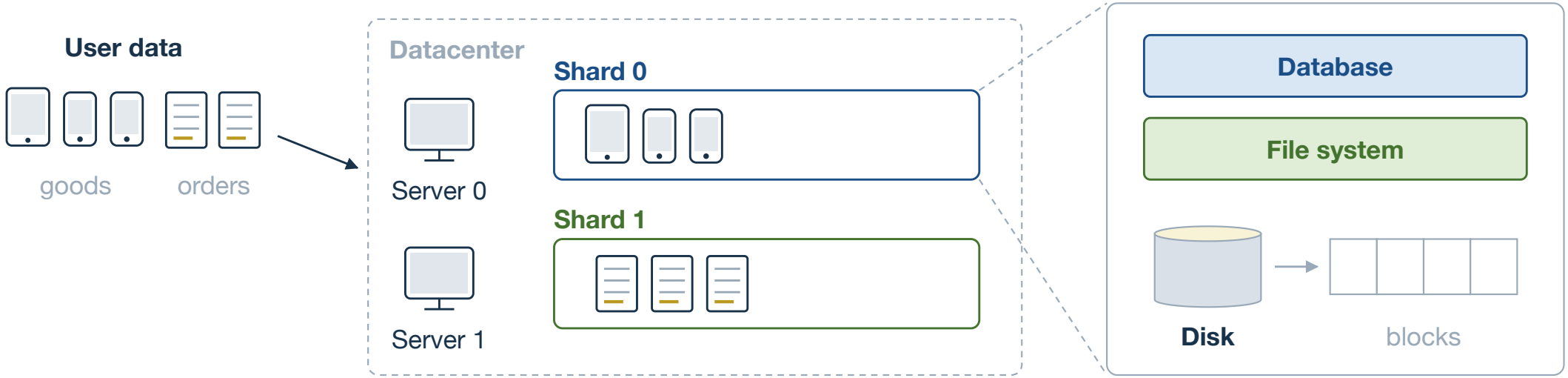
Second step: scale the data with sharding

At real scale, no single machine can hold the data

- Facebook: "people upload more than 350 million photos" every day (as of 2012)[1]

We also want transparent scaling: as the data grows, the system scales mostly without notice

- At least for the users
- Hopefully for the developers

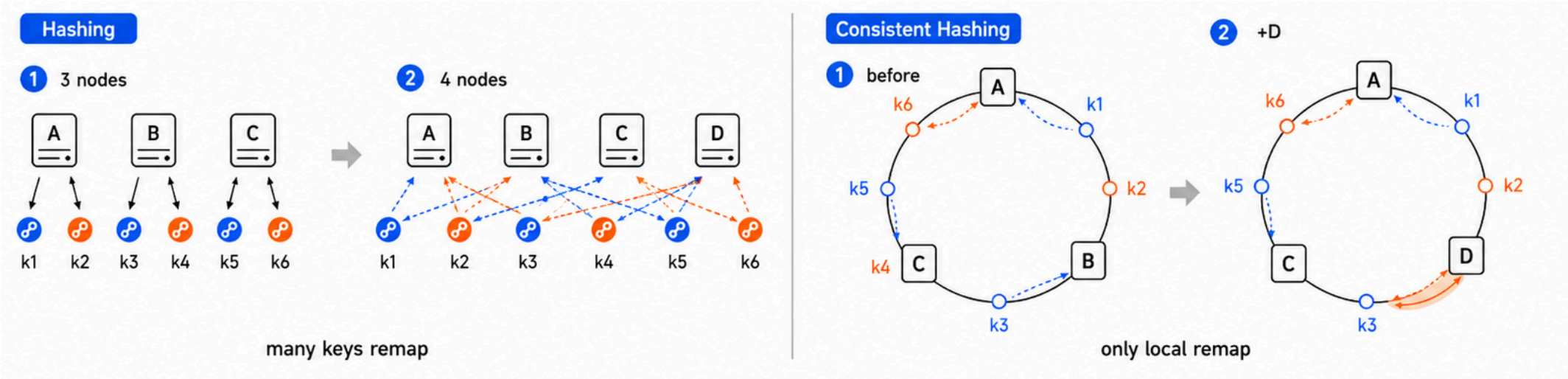


[1] Source: An Analysis of Facebook Photo Caching

Data scaling challenge #1: how to shard

Which shard holds a given key?

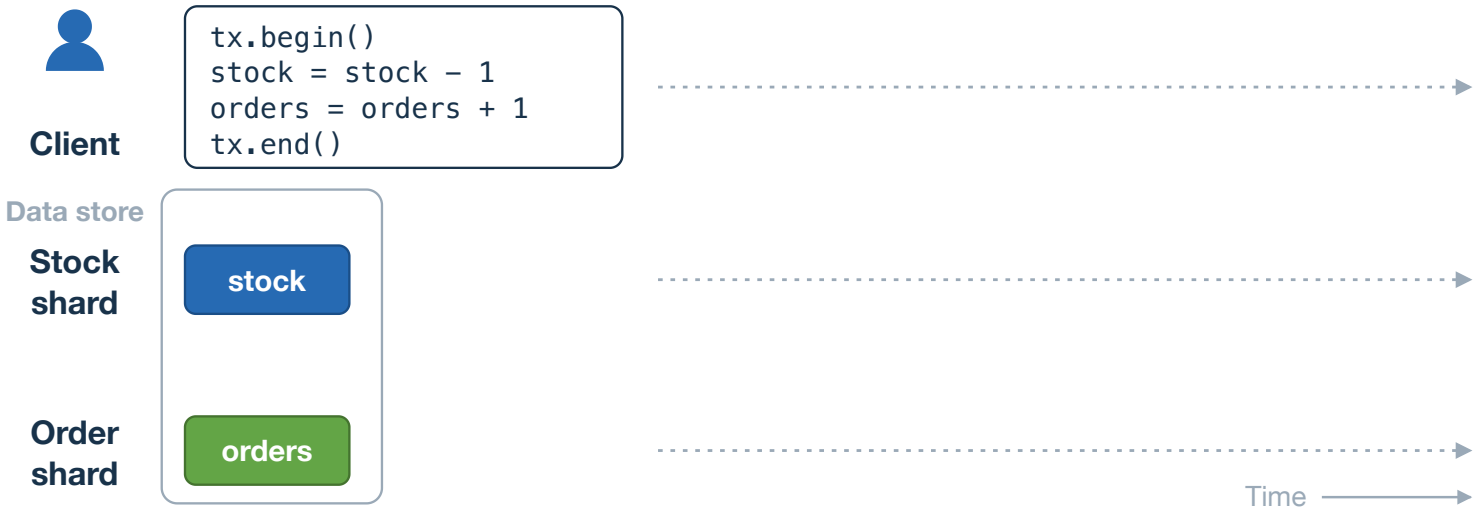
- Hashing: `node = hash(key) % N` — growing N reshuffles almost every key
- Consistent hashing fixes that (*A story of scaling*)
- Range partitioning: shard 0 holds keys 0–100, etc.



Data scaling challenge #2: multi-shard transaction

One order updates two shards: the stock on the Stock shard, the order on the Order shard

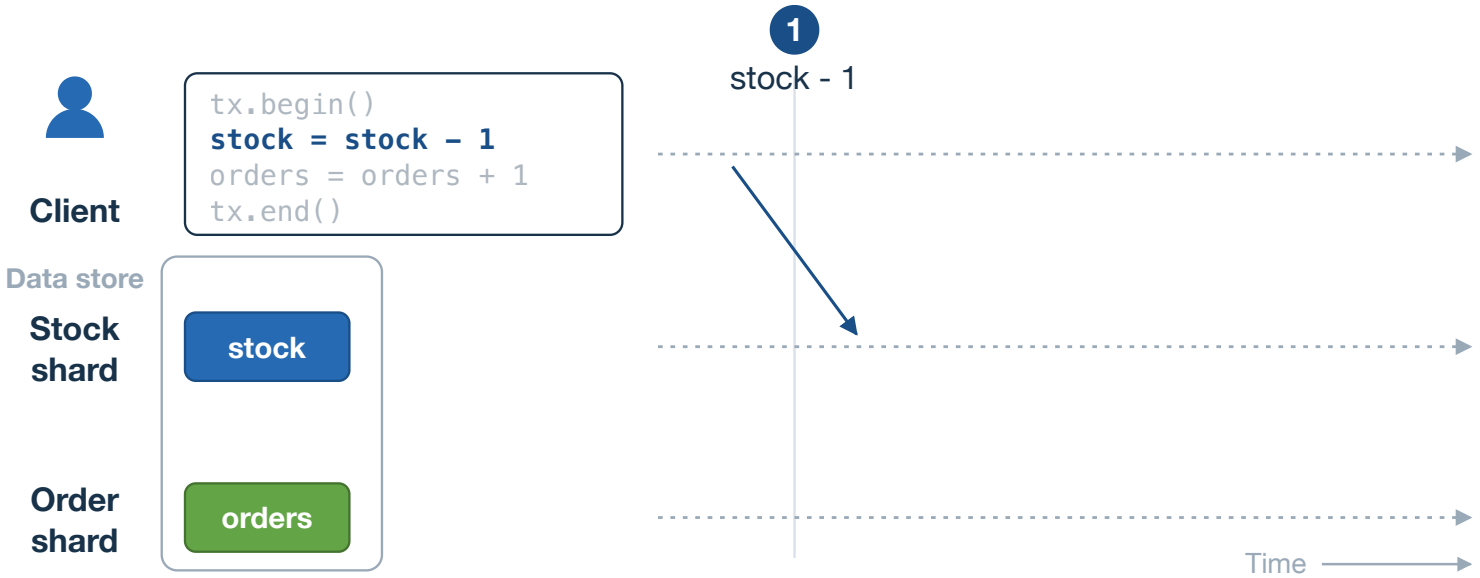
- What if one machine fails in the middle? A half-done order



Data scaling challenge #2: multi-shard transaction

One order updates two shards: the stock on the Stock shard, the order on the Order shard

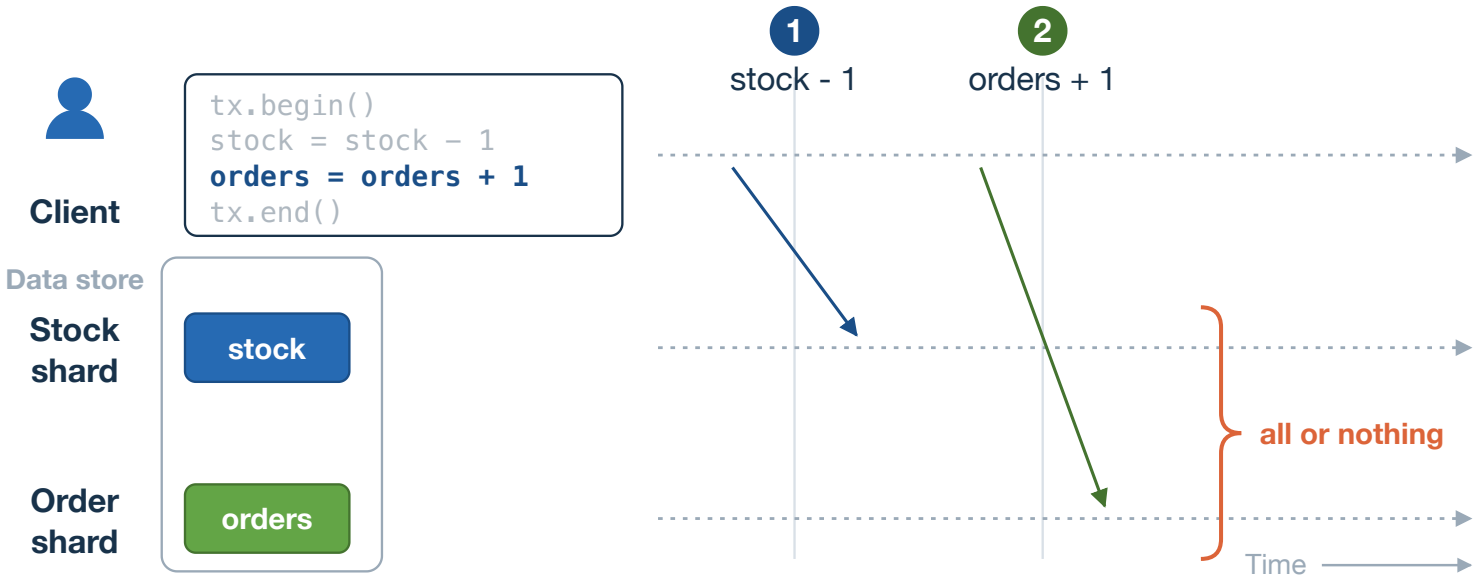
- What if one machine fails in the middle? A half-done order



Data scaling challenge #2: multi-shard transaction

One order updates two shards: the stock on the Stock shard, the order on the Order shard

- What if one machine fails in the middle? A half-done order

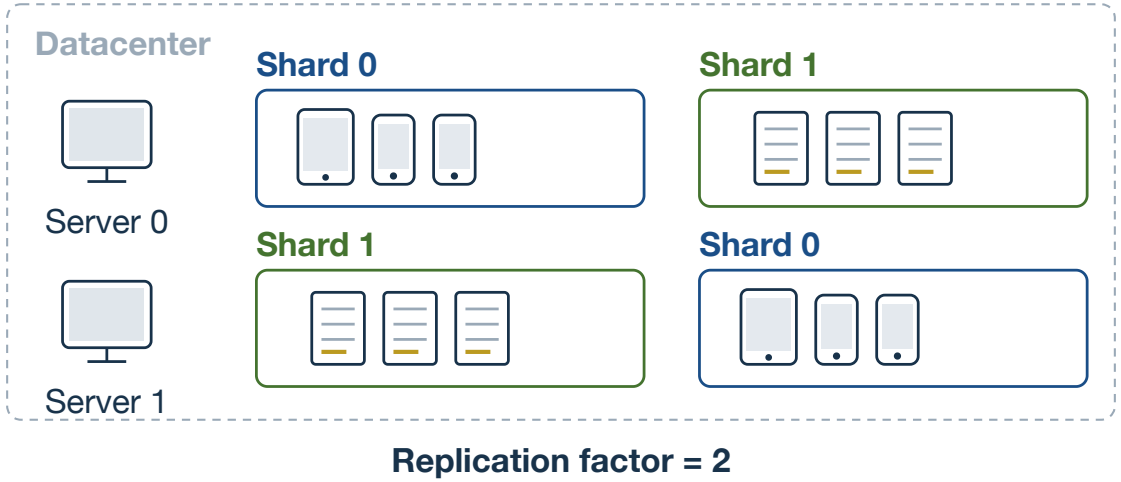


Solution: Two-phase commit

Data scaling challenge #3: failures are the normal case

A large-scale website runs on >10K servers; at that scale failures are the normal case

- "an MTBF of 30 years (10,000 days) … a cluster of 10,000 servers will see an average of one server failure per day" [1]



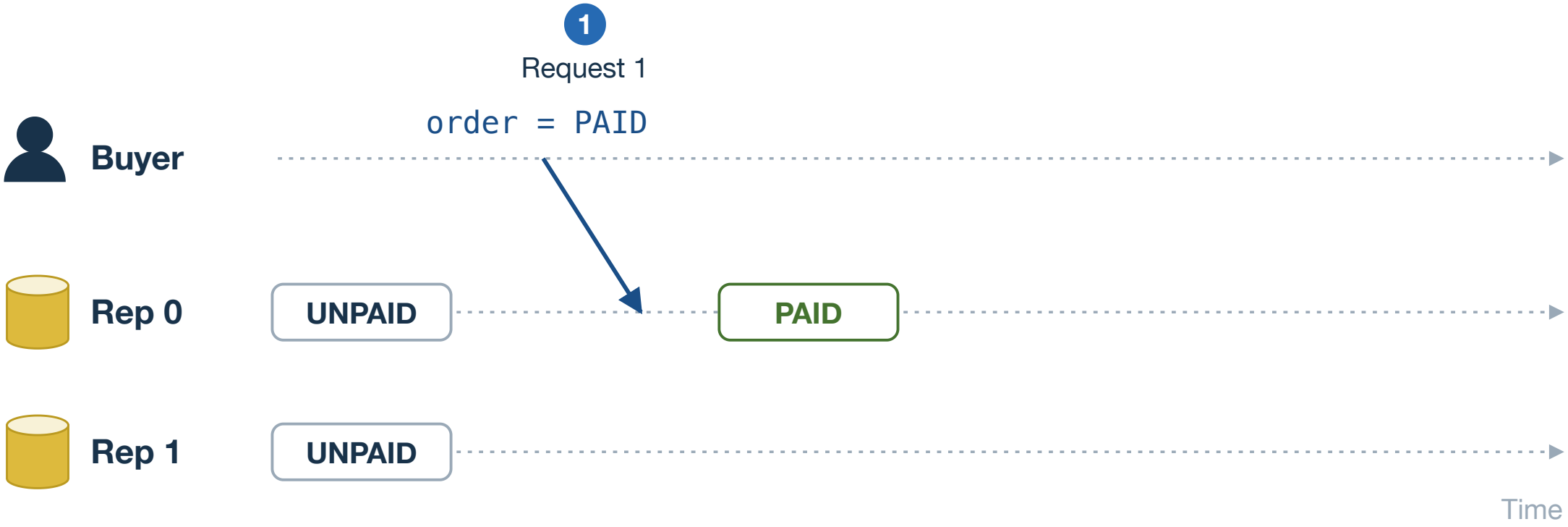
Solution: Logging + Replication

[1] Source: Barroso, Hölzle & Ranganathan, *The Datacenter as a Computer* 3e, §2.2

The challenge of replication: consistency

Allow arbitrary reads/writes on any replica: a buyer pays, then still sees UNPAID

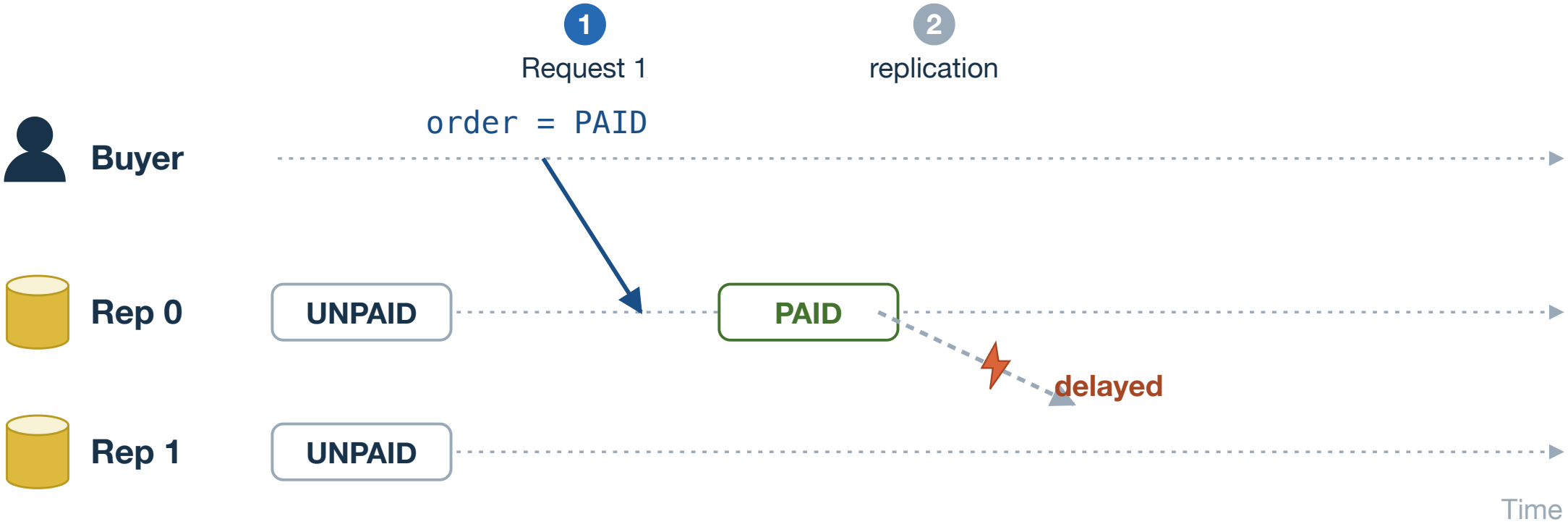
- Request 1 writes `order = PAID` to replica 0; Request 2 reads the stale replica 1



The challenge of replication: consistency

Allow arbitrary reads/writes on any replica: a buyer pays, then still sees UNPAID

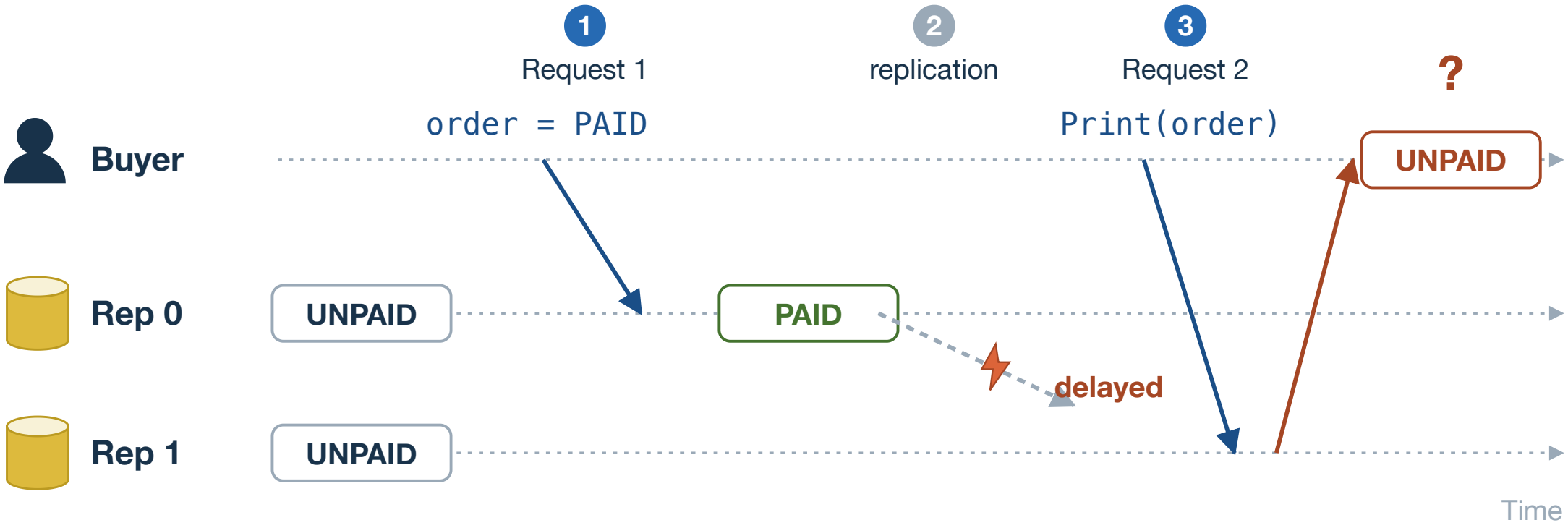
- Request 1 writes `order = PAID` to replica 0; Request 2 reads the stale replica 1



The challenge of replication: consistency

Allow arbitrary reads/writes on any replica: a buyer pays, then still sees UNPAID

- Request 1 writes `order = PAID` to replica 0; Request 2 reads the stale replica 1



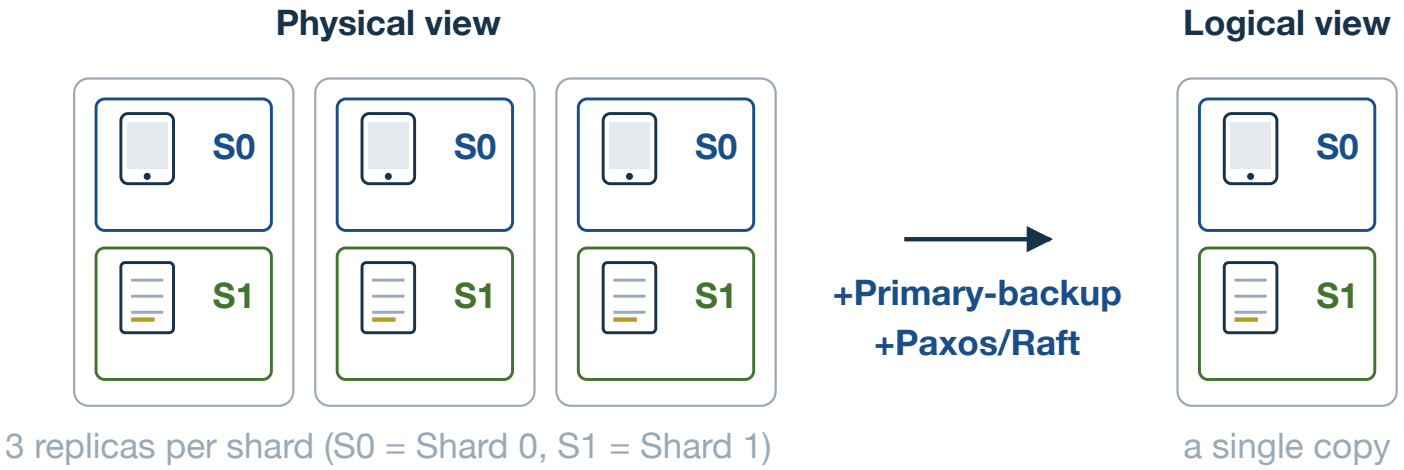
The challenge of replication: consistency

Two consistency models learned so far

- Weak: eventual consistency — simpler, faster, more available, but may reveal stale data
- Strong: linearizability — slower, less available, but behaves like one machine

Implement strong consistency: *Primary-backup + Paxos + Raft*): every write goes through a replicated log agreed on by consensus

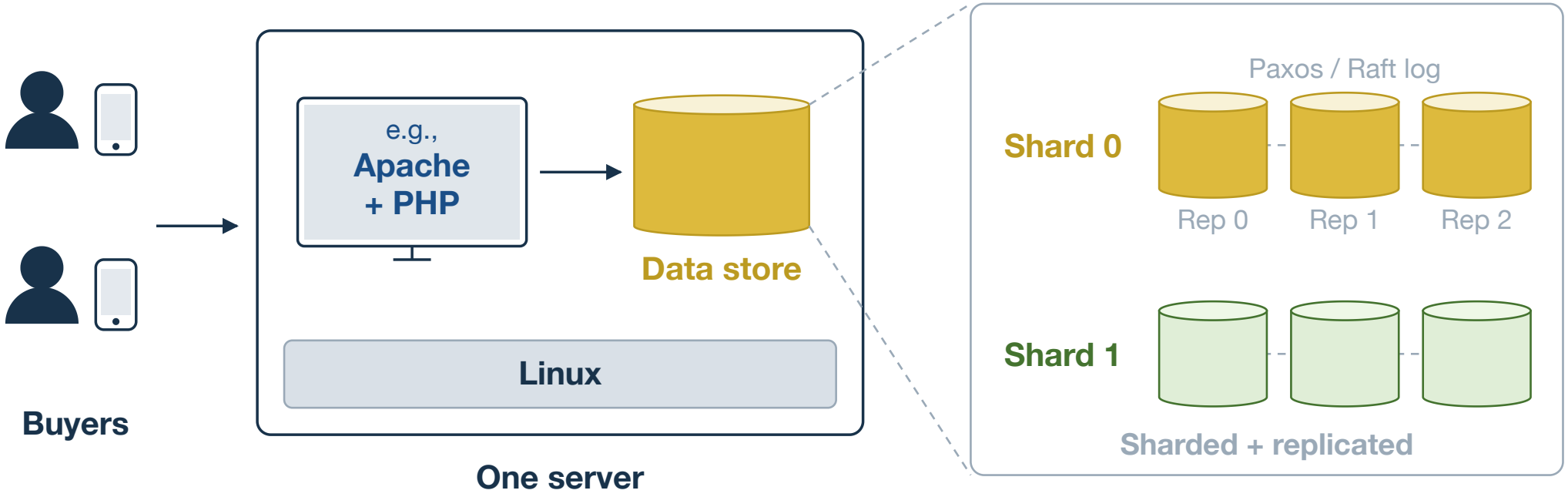
- We can use the log to recover from the failure, e.g., redo-logging



More issues: how do we serve a user?

We need a stack to turn a request into data access

- In the old days: the LAMP stack — **L**inux, **A**pache, data store (**M**ySQL), **P**HP

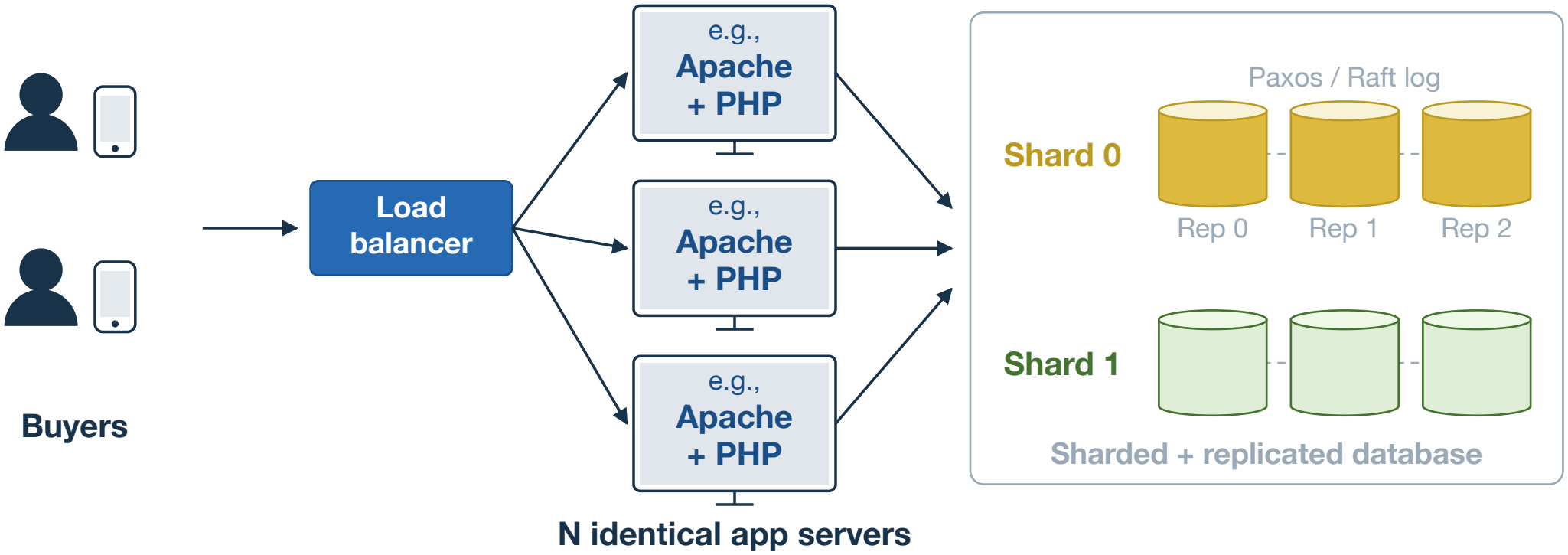


We have scaled the data store (MySQL). What about the others?

Scaling the frontend with concurrency

Good news: the frontend (e.g., Apache) is stateless

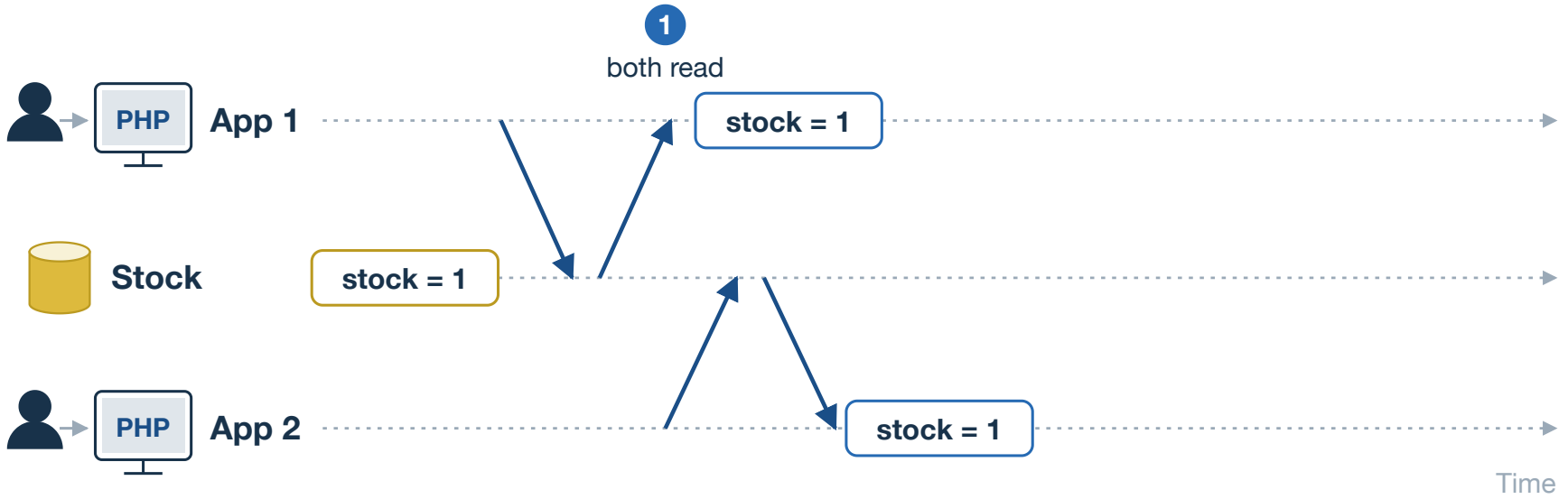
- Any server can serve any request; add servers and route requests to them



Concurrency challenge: isolation (before-or-after atomicity)

Apache is stateless, but the PHP handler it runs reads and writes the stateful data

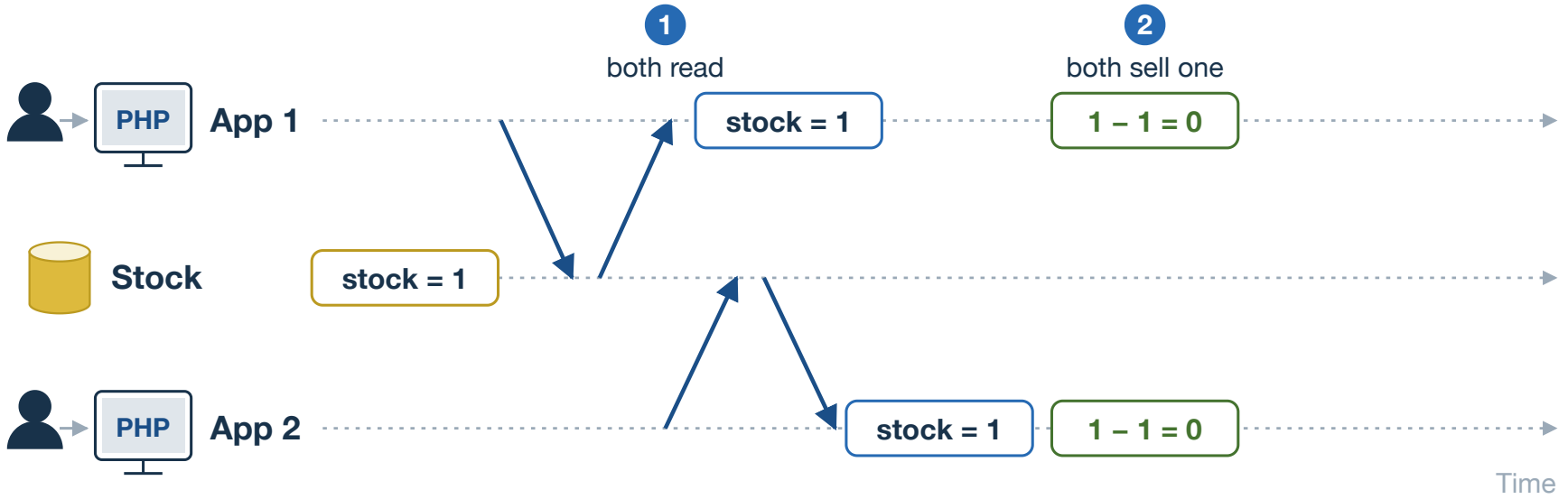
- e.g., two buyers' requests run on two app servers: both buy an item with stock == 1, which is inconsistent



Concurrency challenge: isolation (before-or-after atomicity)

Apache is stateless, but the PHP handler it runs reads and writes the stateful data

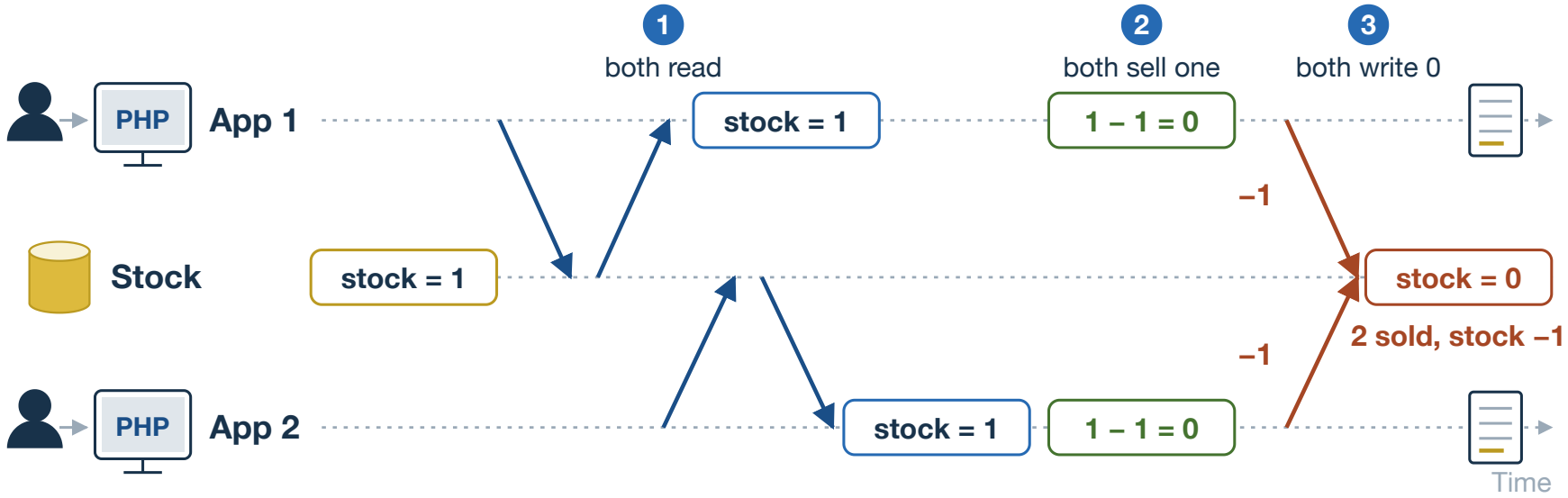
- e.g., two buyers' requests run on two app servers: both buy an item with stock == 1, which is inconsistent



Concurrency challenge: isolation (before-or-after atomicity)

Apache is stateless, but the PHP handler it runs reads and writes the stateful data

- e.g., two buyers' requests run on two app servers: both buy an item with stock == 1, which is inconsistent



Solution: 2PL, OCC and MVCC

Put it all together: from one machine to a distributed database

What we need	Techniques
Atomic commit across shards	Two-phase commit (2PC)
Isolation between transactions	2PL / OCC / MVCC
Consistent replicas	Primary-backup / Paxos / Raft
Data beyond one machine	Sharding: hash / range partitioning
Communication across machines	RPC (Remote Procedure Call)
Durable local storage	Filesystem + logging

Spanner combines these building blocks into one system

Case study: Spanner's read-write TX

High-level overview of Spanner's read-write TX

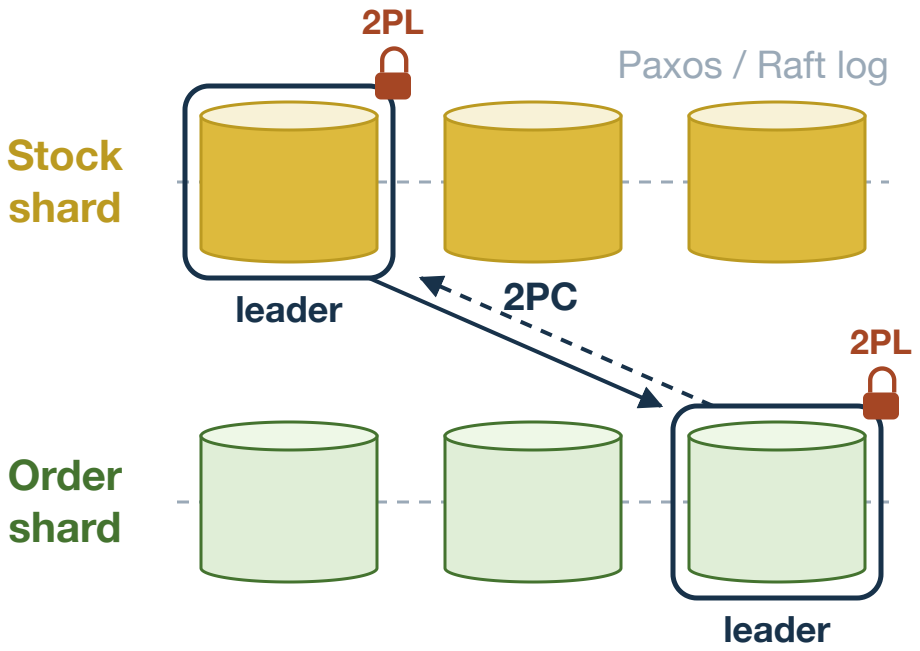
Read-write TX: a TX that both reads and writes the data

Goal:

1. Serializability
2. High availability

Techniques used:

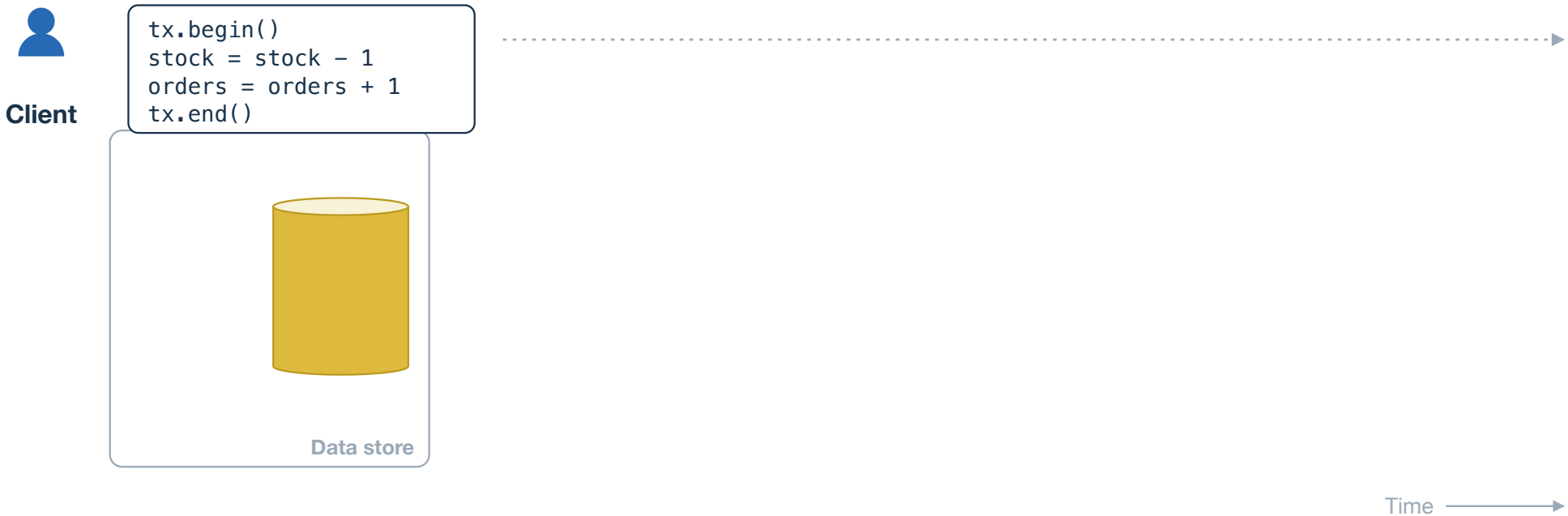
- 2PL
- 2PC
- Paxos



Execution flow of read-write TX: the setting

Read-write TX: a TX that both reads and writes the data

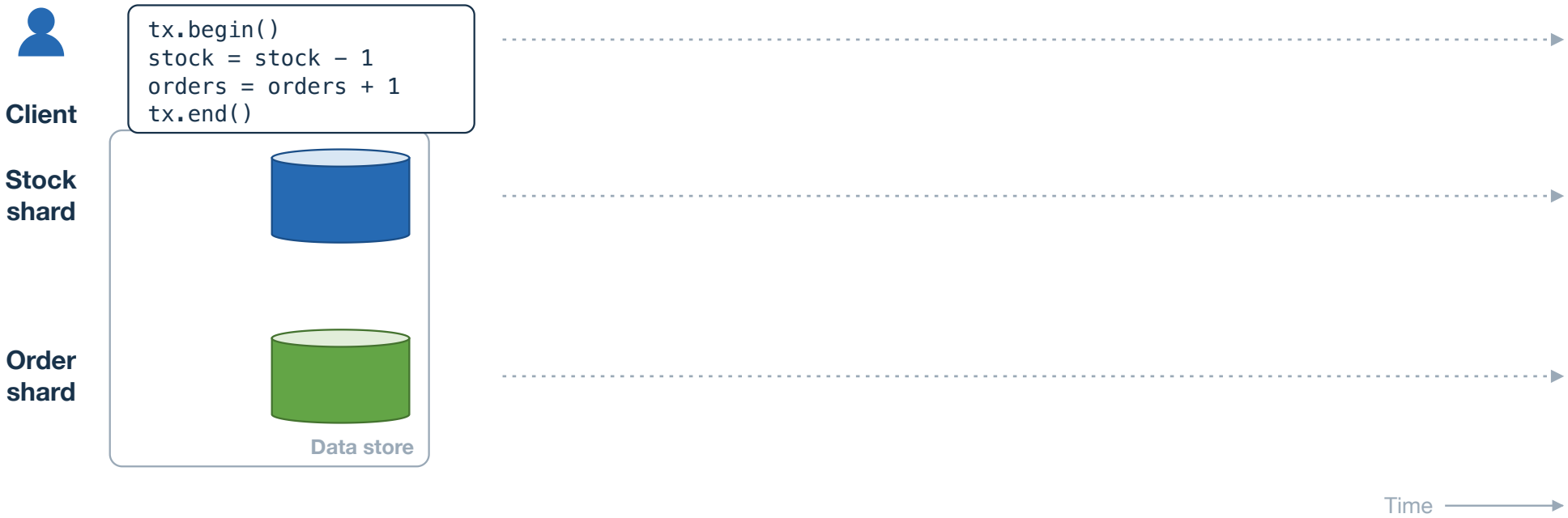
- The data is logically one store, but is physically partitioned across machines
- Our example: The client reads and writes stock and orders, then calls tx.end()



Execution flow of read-write TX: the setting

Read-write TX: a TX that both reads and writes the data

- The data is logically one store, but is physically partitioned across machines
- Our example: The client reads and writes stock and orders, then calls tx.end()

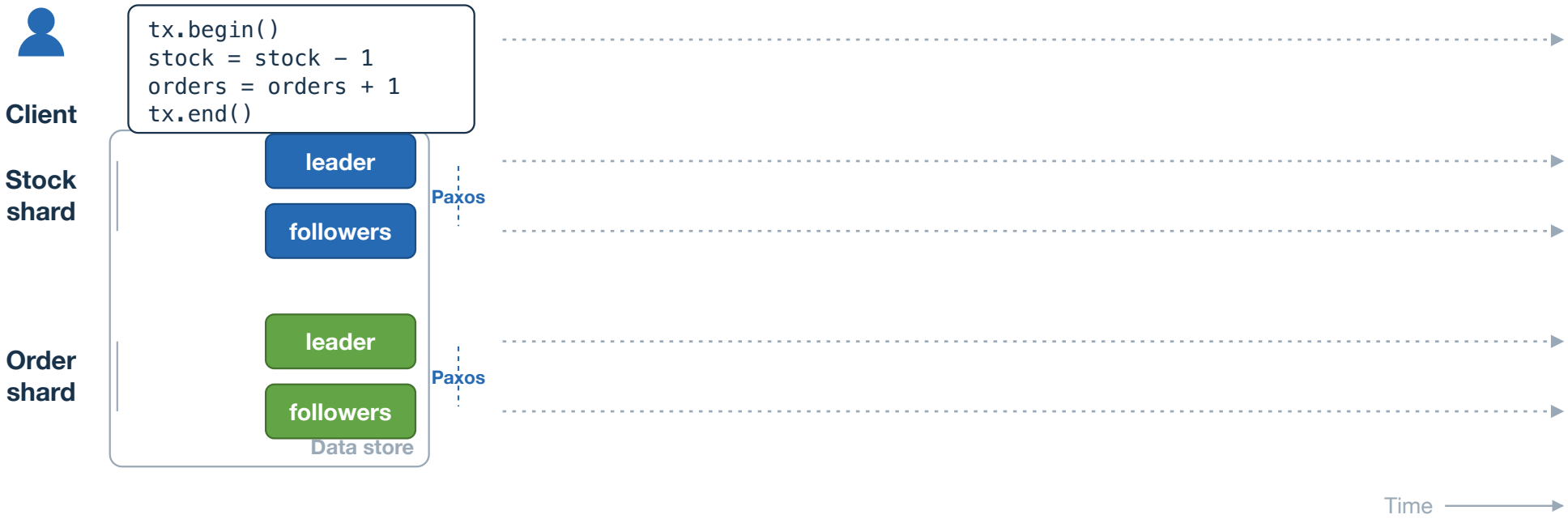


Execution flow of read-write TX: the setting

Spanner uses (multi-) Paxos for replication with consistency: one leader + followers

- The leader stores the data as well as the lock associated with it
- Note that the leader is a choice for performance, not correctness

1

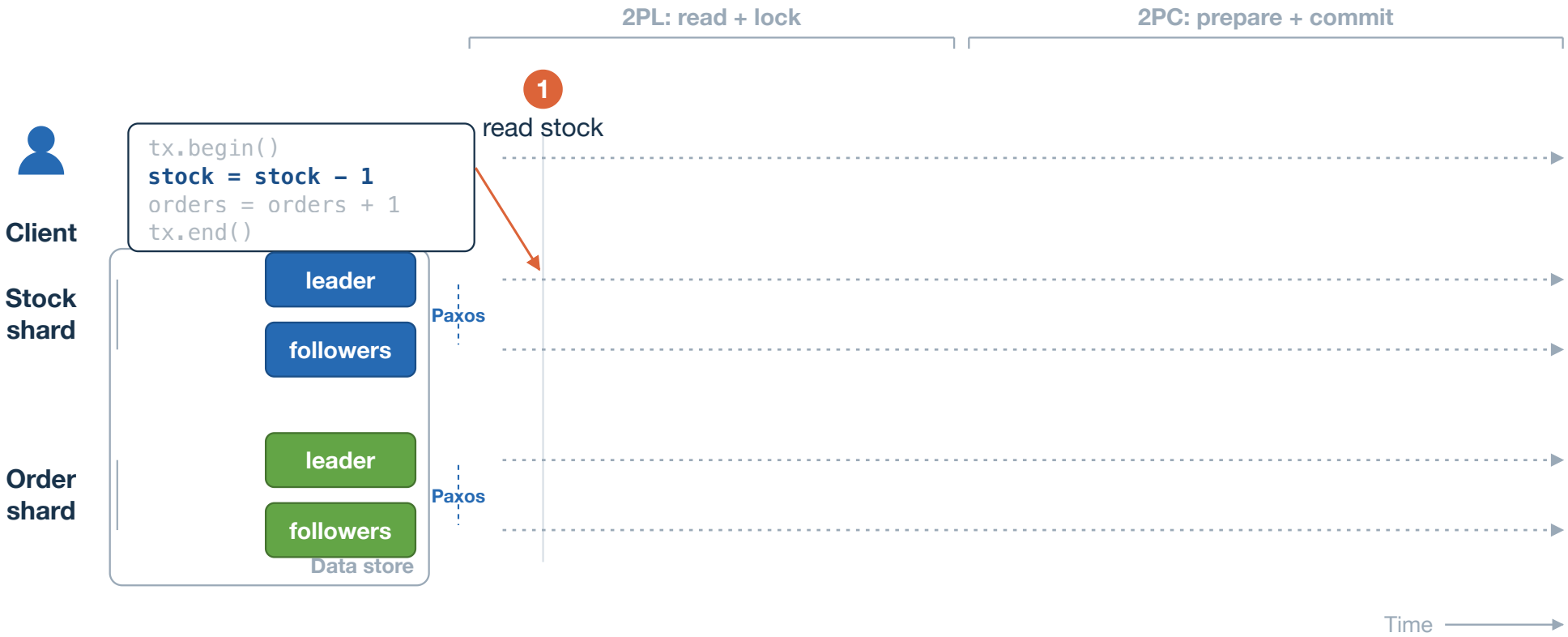


[1] Multi-Paxos with a stable leader: the prepare phase is skipped, one round = one RTT

Execution flow of read-write TX: 2PL

Read goes to the shard leader: to first grab the lock and read the latest values

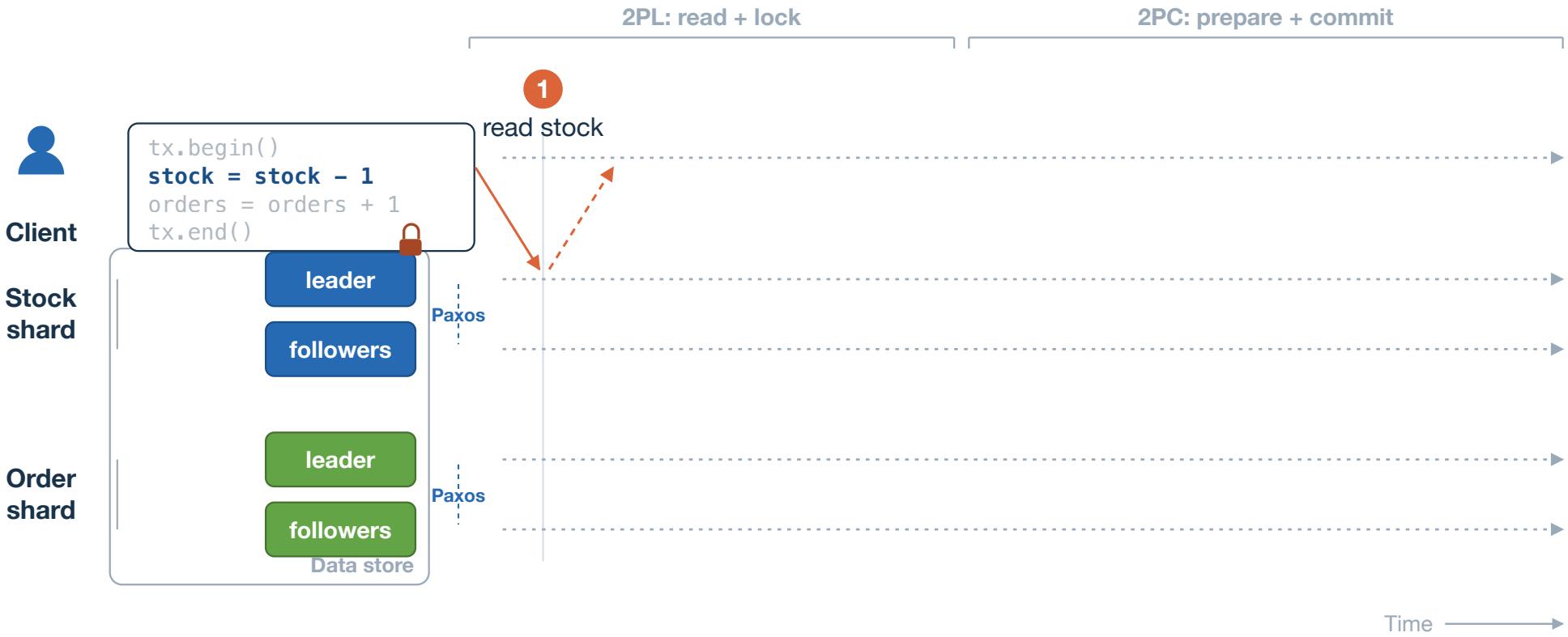
- The read lock lives on the leader only, it is not replicated (Think about it, why?)
- If the leader fails, the TX aborts and retries: nothing to recover



Execution flow of read-write TX: 2PL

Read goes to the shard leader: to first grab the lock and read the latest values

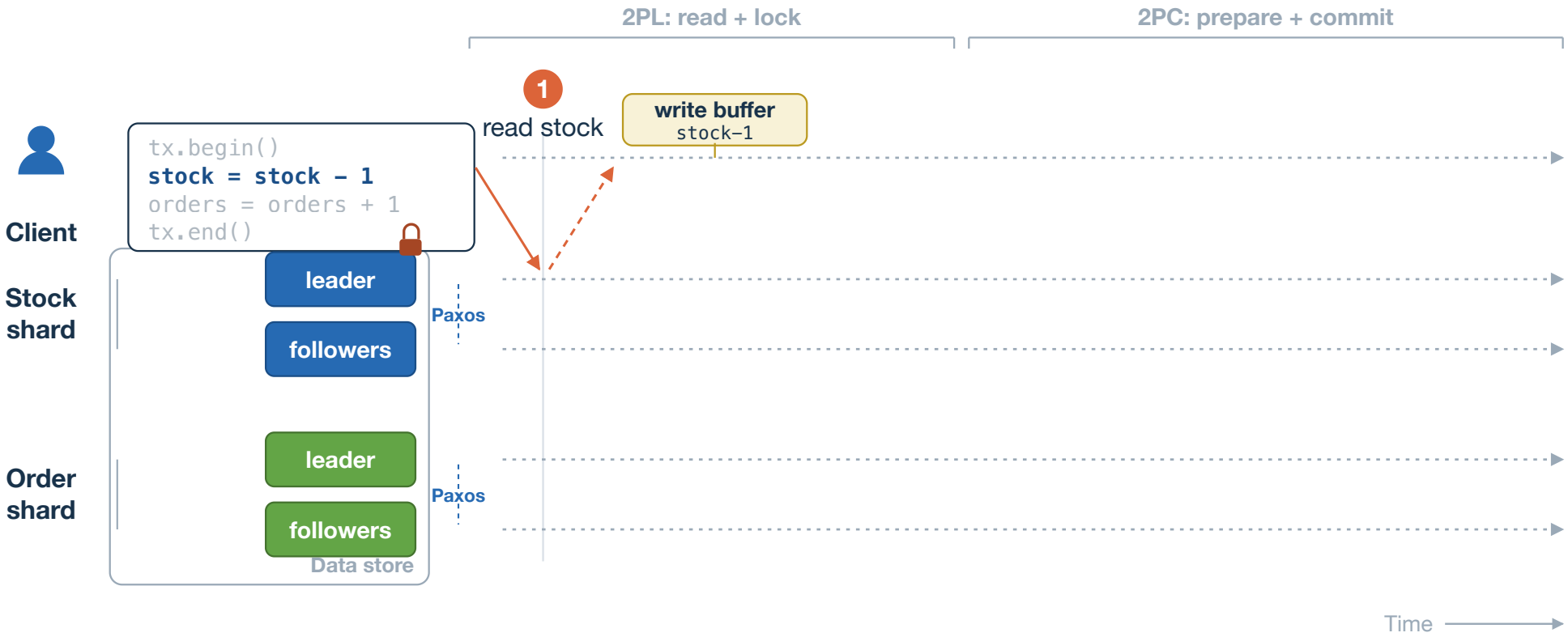
- The read lock lives on the leader only, it is not replicated (Think about it, why?)
- If the leader fails, the TX aborts and retries: nothing to recover



Execution flow of read-write TX: 2PL

The write is buffered at the client, not sent to the leader

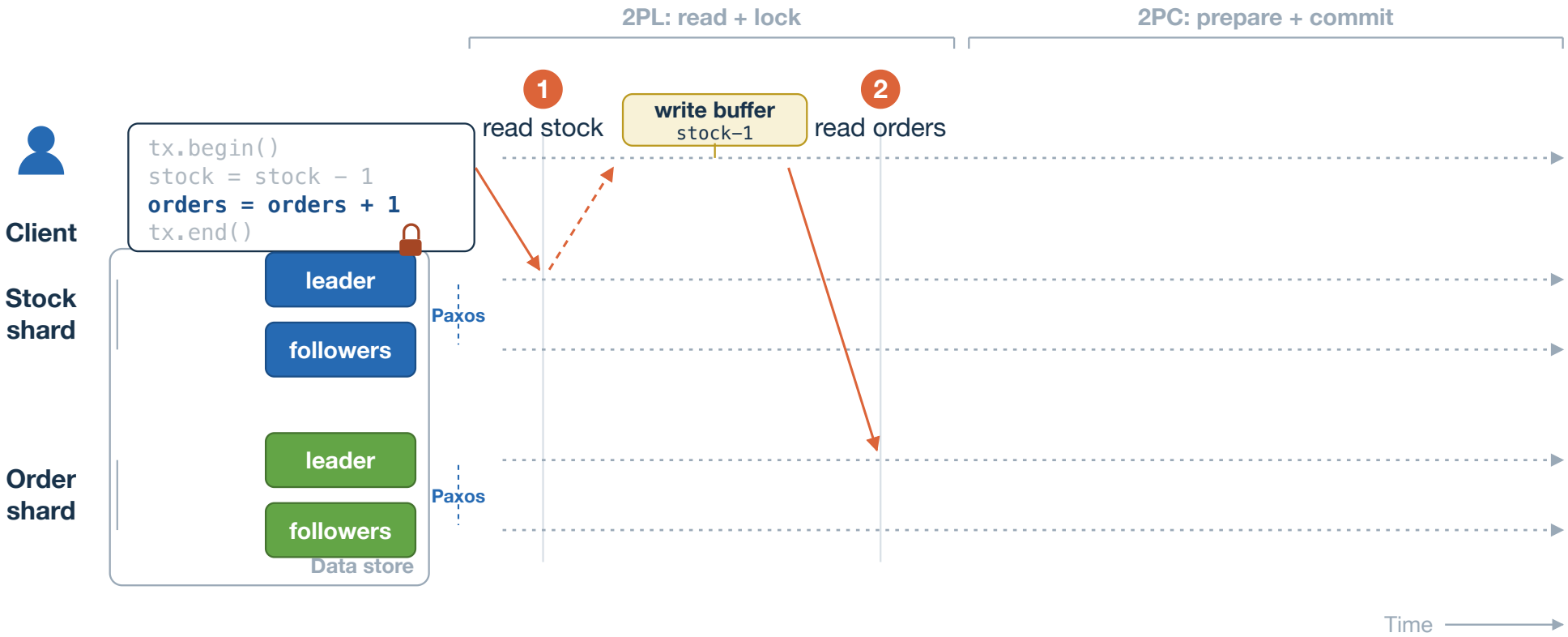
- An abort undoes nothing on the shard, and the write lock is held shorter
- At commit time, one Paxos round takes the write locks; the read locks are already held



Execution flow of read-write TX: 2PL

The write is buffered at the client, not sent to the leader

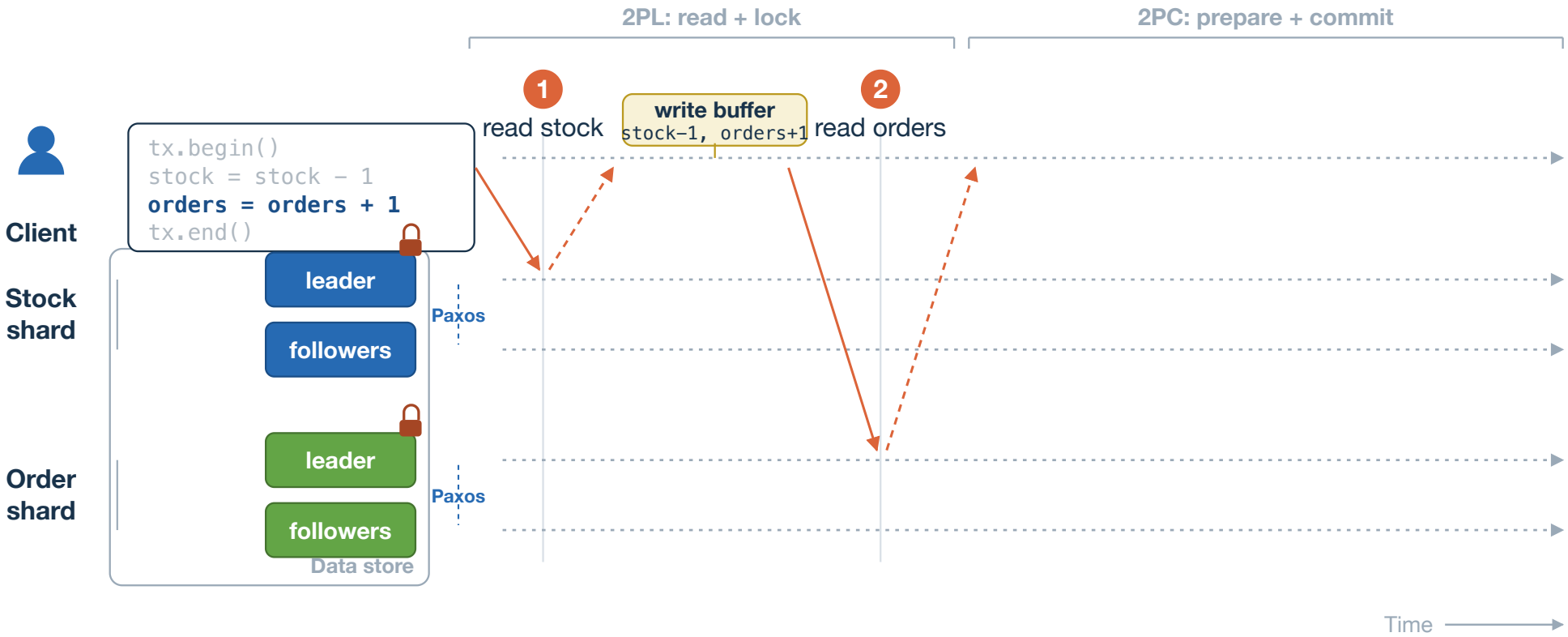
- An abort undoes nothing on the shard, and the write lock is held shorter
- At commit time, one Paxos round takes the write locks; the read locks are already held



Execution flow of read-write TX: 2PL

The write is buffered at the client, not sent to the leader

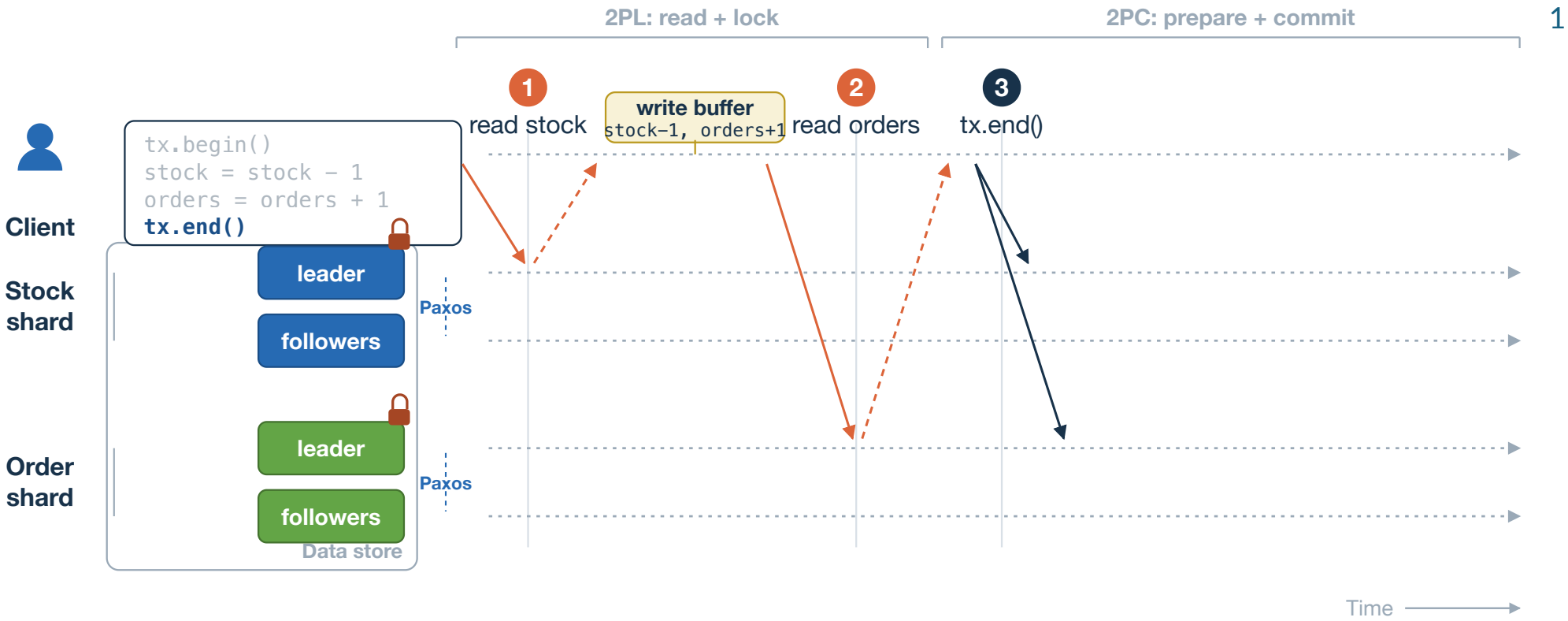
- An abort undoes nothing on the shard, and the write lock is held shorter
- At commit time, one Paxos round takes the write locks; the read locks are already held



Execution flow of read-write TX: 2PC

`tx.end()`: the client sends the write-set to every participant leader

- It chooses one of them the coordinator
- Why a shard leader? the coordinator can reuse its Paxos's state

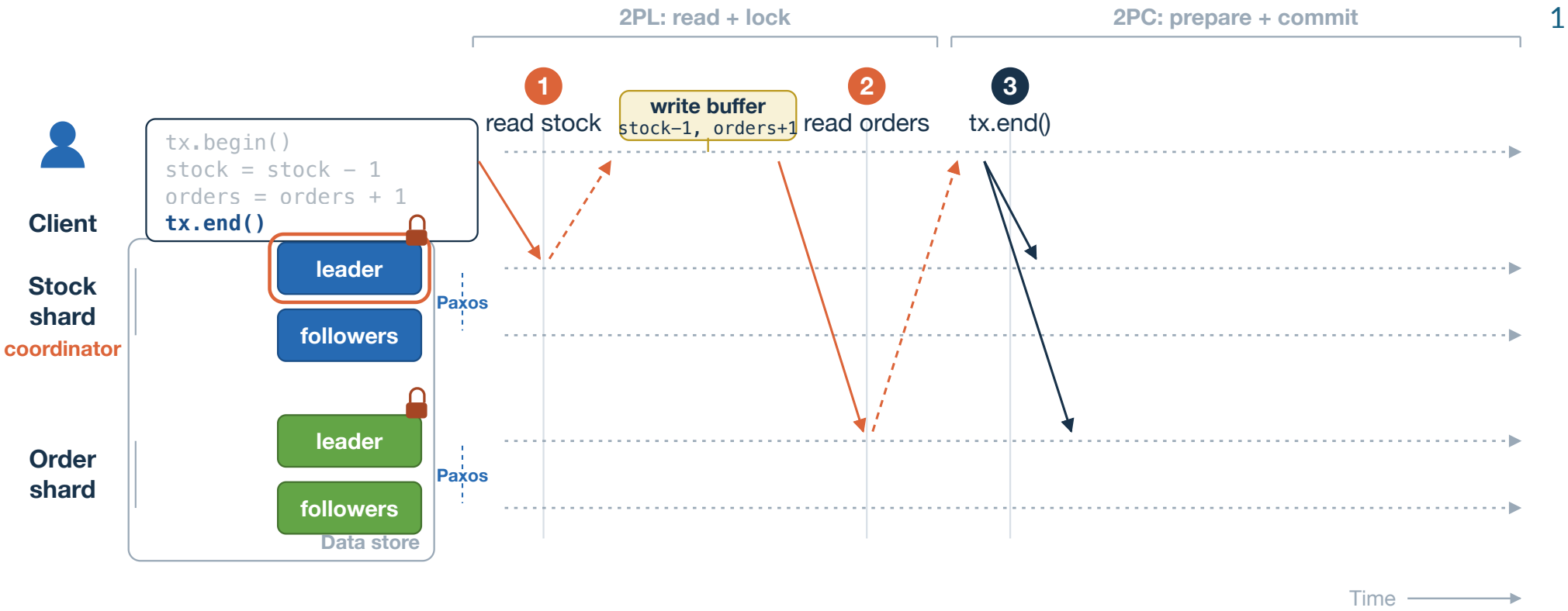


[1] Corbett et al., §4.2.1: the client drives 2PC — it "avoids sending data across wide-area links twice"

Execution flow of read-write TX: 2PC

`tx.end()`: the client sends the write-set to every participant leader

- It chooses one of them the coordinator
- Why a shard leader? the coordinator can reuse its Paxos's state

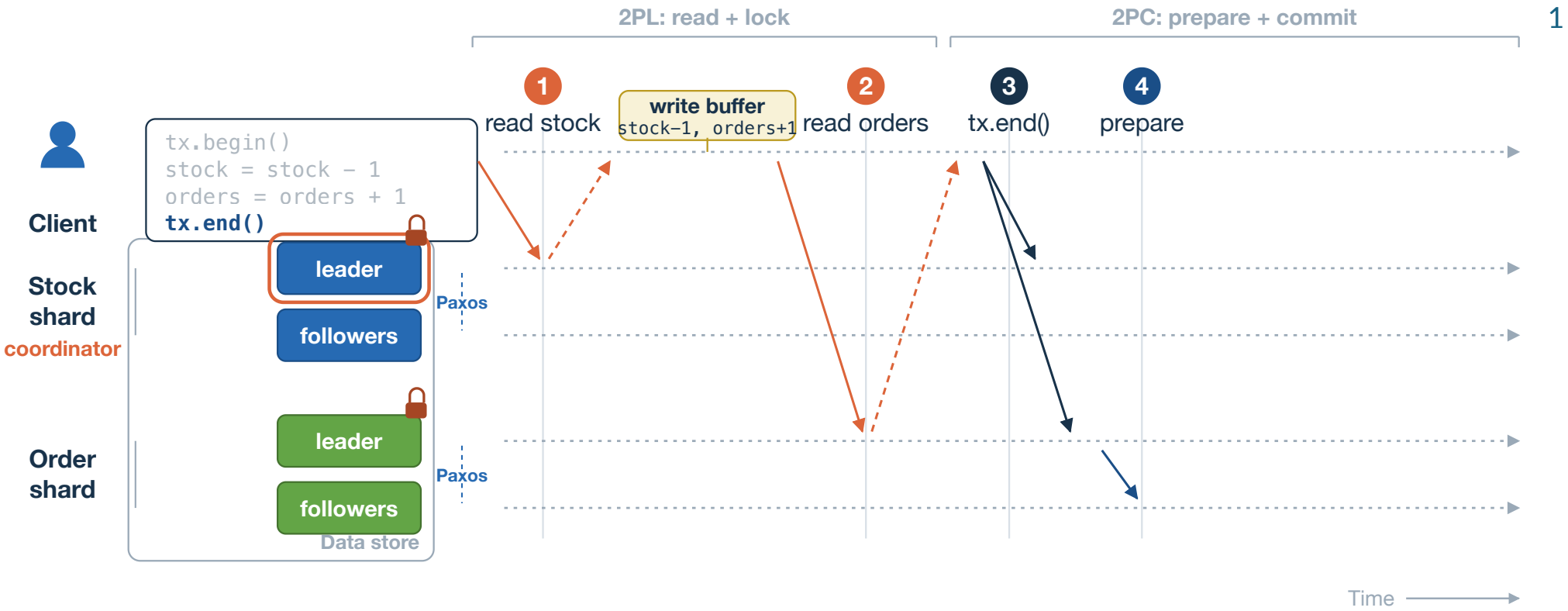


[1] Corbett et al., §4.2.1: the client drives 2PC — it "avoids sending data across wide-area links twice"

Execution flow of read-write TX: 2PC

prepare: each other leader takes the write lock, then one Paxos round

- It logs a prepare record + the lock, then reports back to the coordinator
- The coordinator skips its own prepare: it makes the decision, so it has nothing to promise

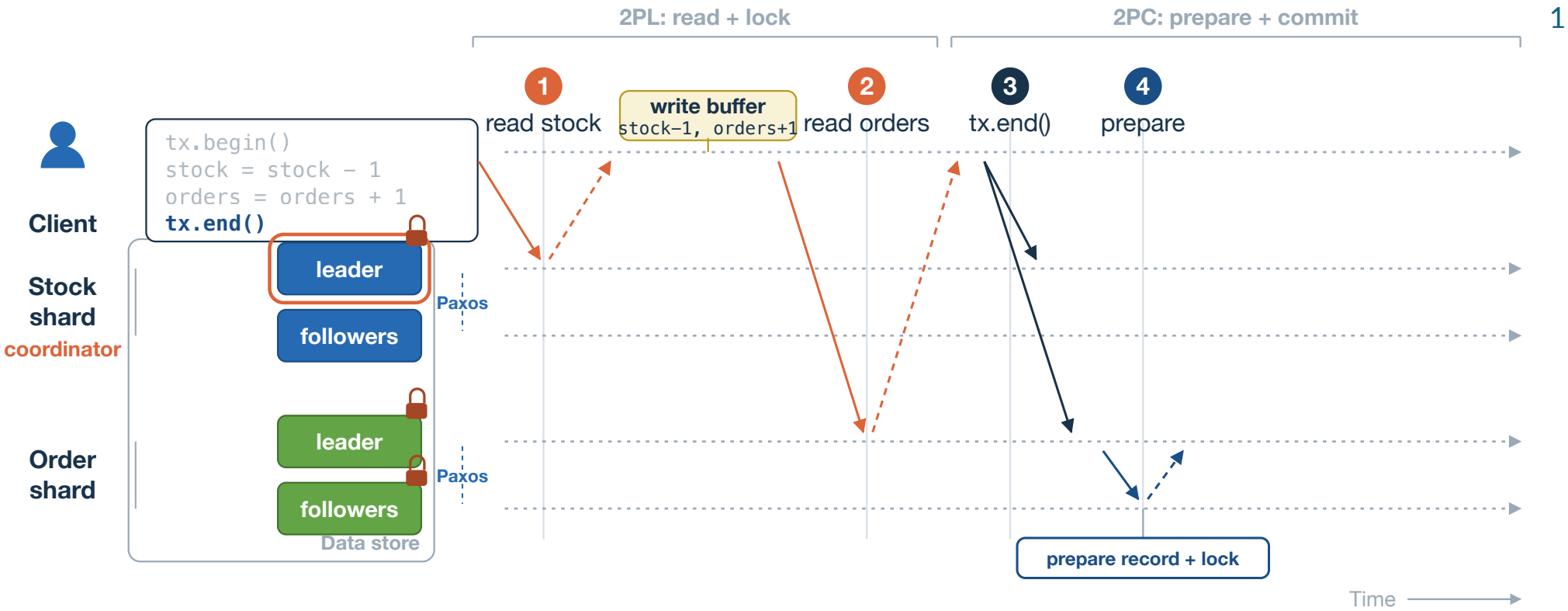


[1] Corbett et al., §4.2.1: the coordinator leader also skips the prepare phase

Execution flow of read-write TX: 2PC

prepare: each other leader takes the write lock, then one Paxos round

- It logs a prepare record + the lock, then reports back to the coordinator
- The coordinator skips its own prepare: it makes the decision, so it has nothing to promise

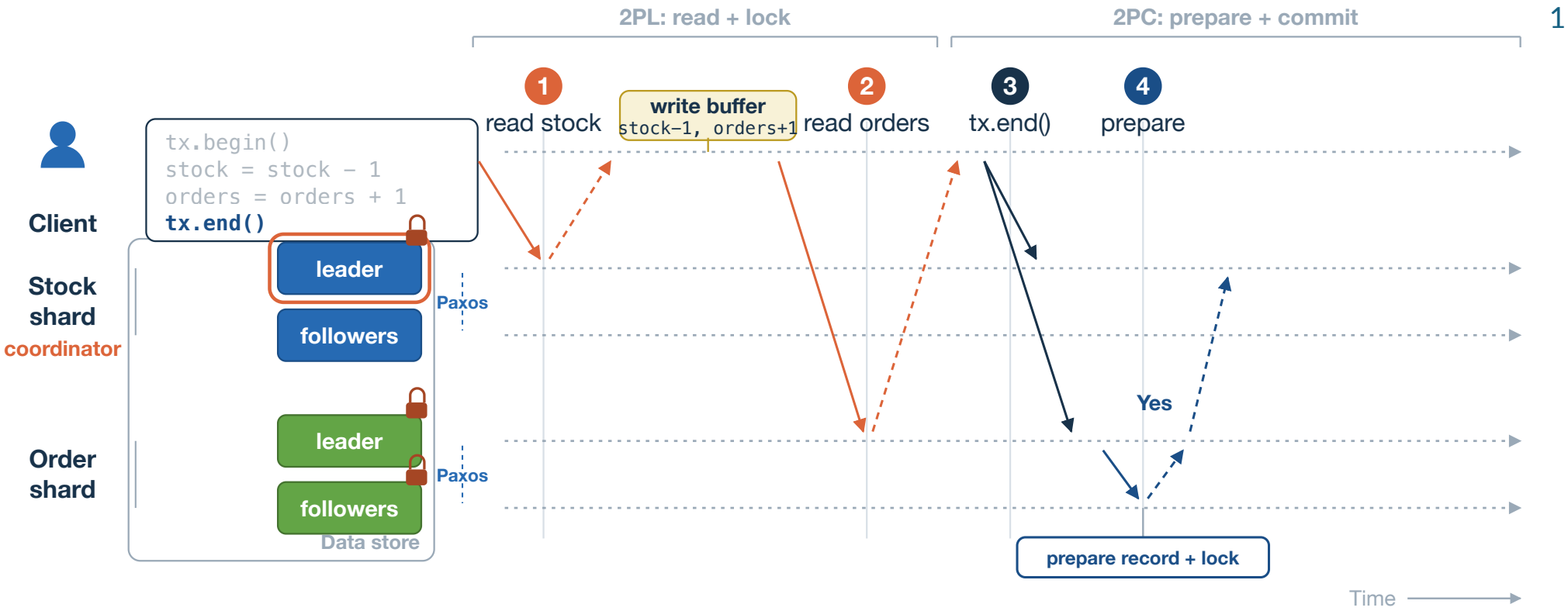


[1] Corbett et al., §4.2.1: the coordinator leader also skips the prepare phase

Execution flow of read-write TX: 2PC

prepare: each other leader takes the write lock, then one Paxos round

- It logs a prepare record + the lock, then reports back to the coordinator
- The coordinator skips its own prepare: it makes the decision, so it has nothing to promise

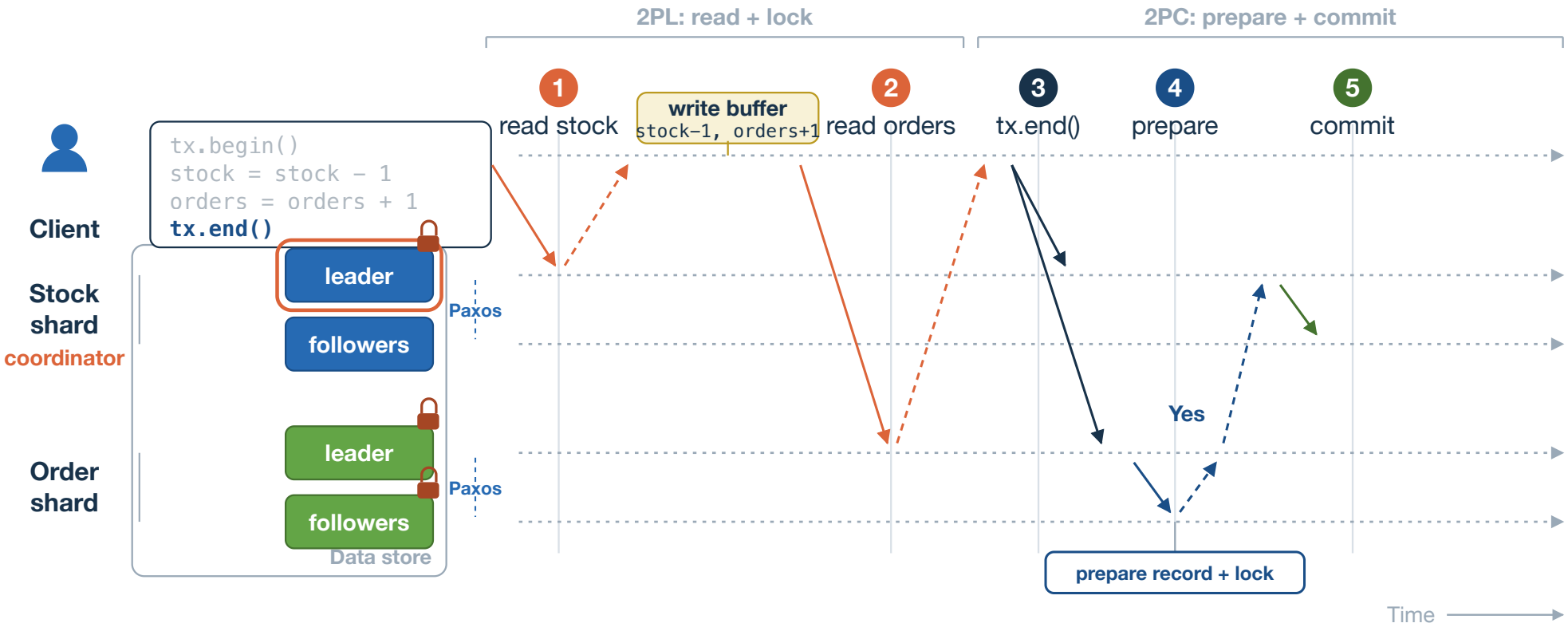


[1] Corbett et al., §4.2.1: the coordinator leader also skips the prepare phase

Execution flow of read-write TX: 2PC

All Yes: one Paxos round logs the commit record, then reply to the client

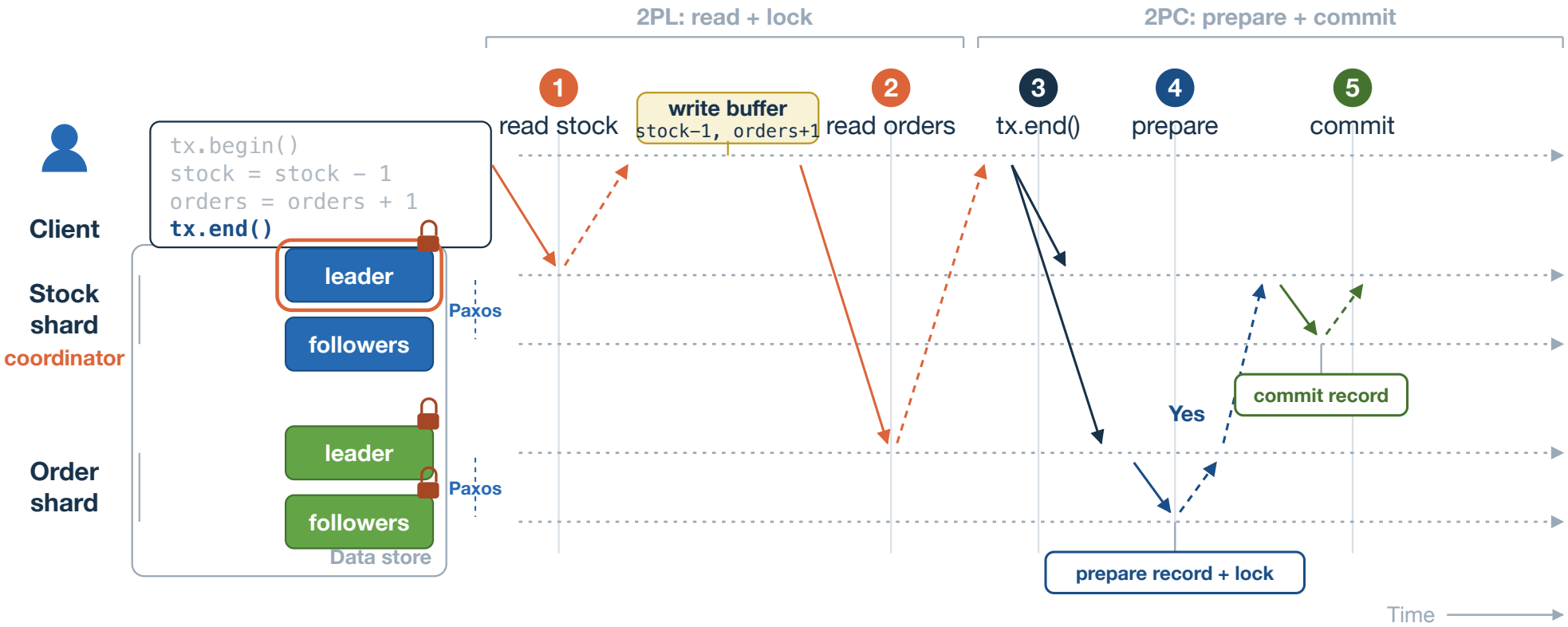
- The record also piggybacks the coordinator's own write (and lock)
- If the commit message fails, e.g., leader change, the TX is aborted



Execution flow of read-write TX: 2PC

All Yes: one Paxos round logs the commit record, then reply to the client

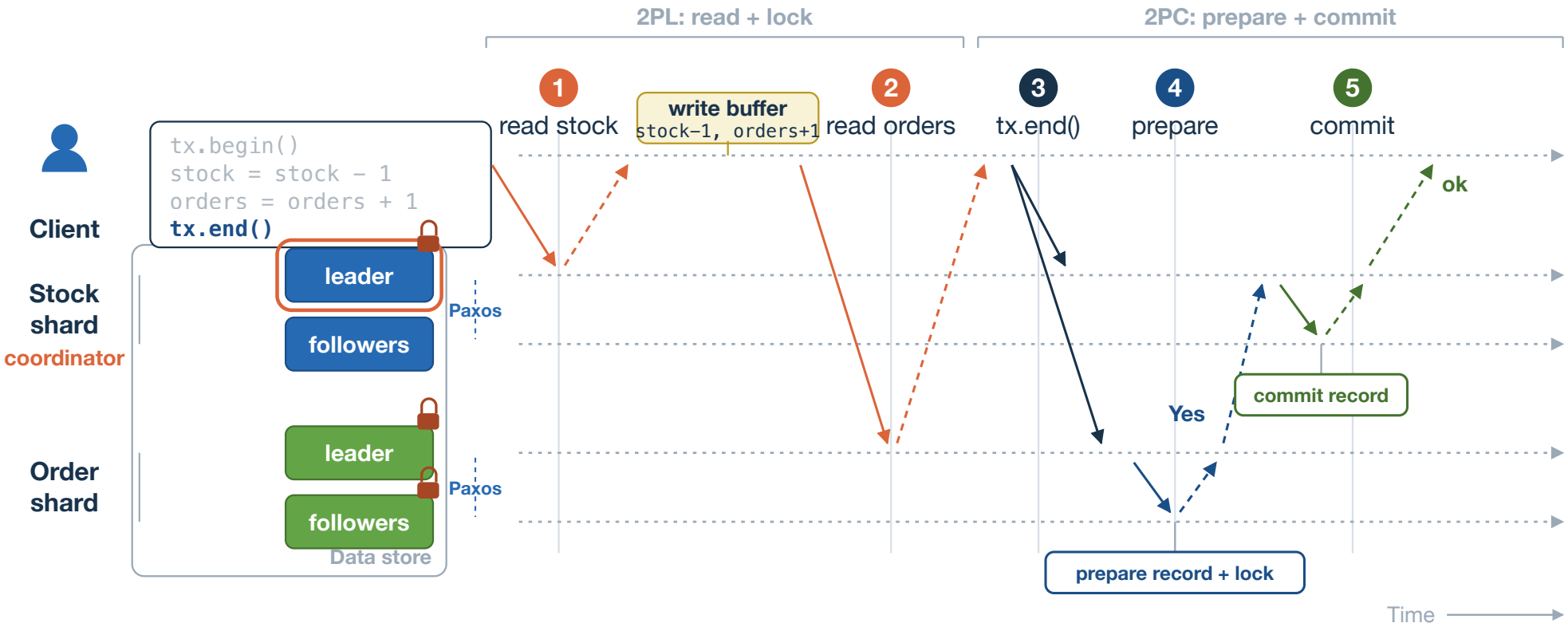
- The record also piggybacks the coordinator's own write (and lock)
- If the commit message fails, e.g., leader change, the TX is aborted



Execution flow of read-write TX: 2PC

All Yes: one Paxos round logs the commit record, then reply to the client

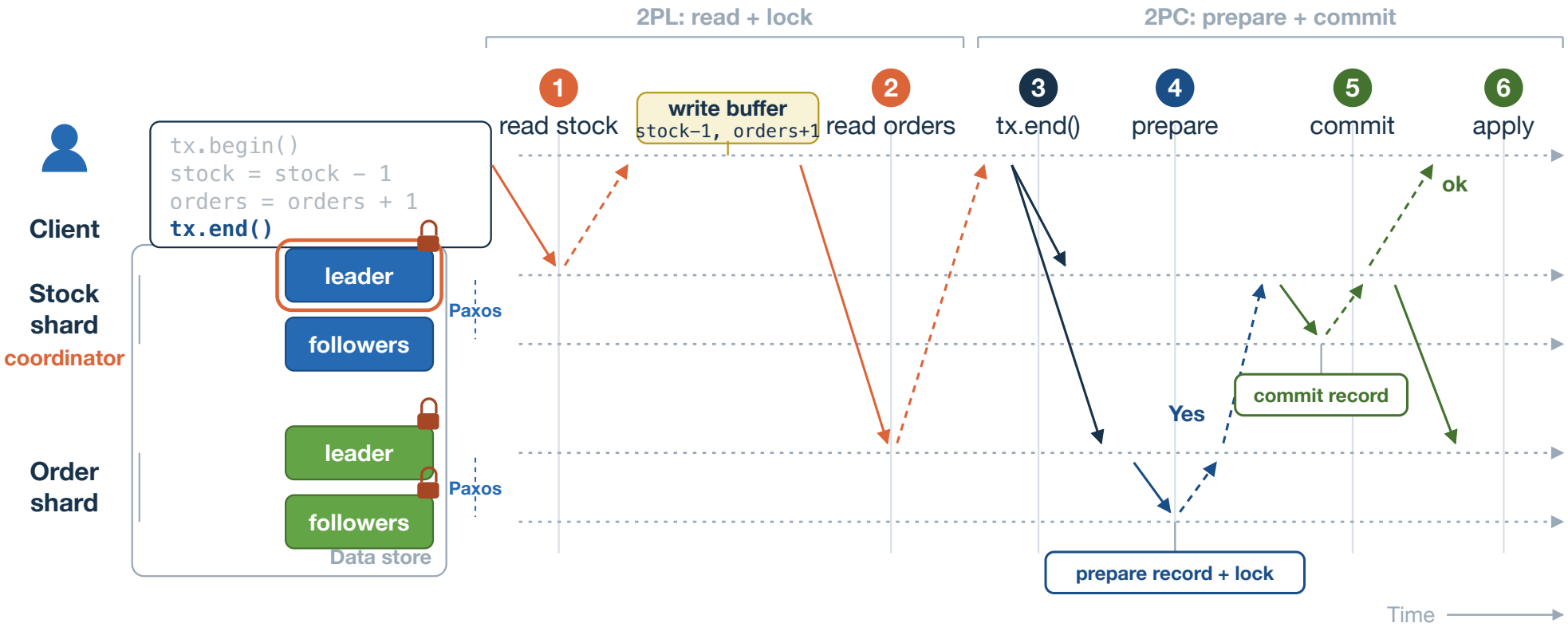
- The record also piggybacks the coordinator's own write (and lock)
- If the commit message fails, e.g., leader change, the TX is aborted



Execution flow of read-write TX: 2PC

Each participant applies the write in one more Paxos round, then unlocks

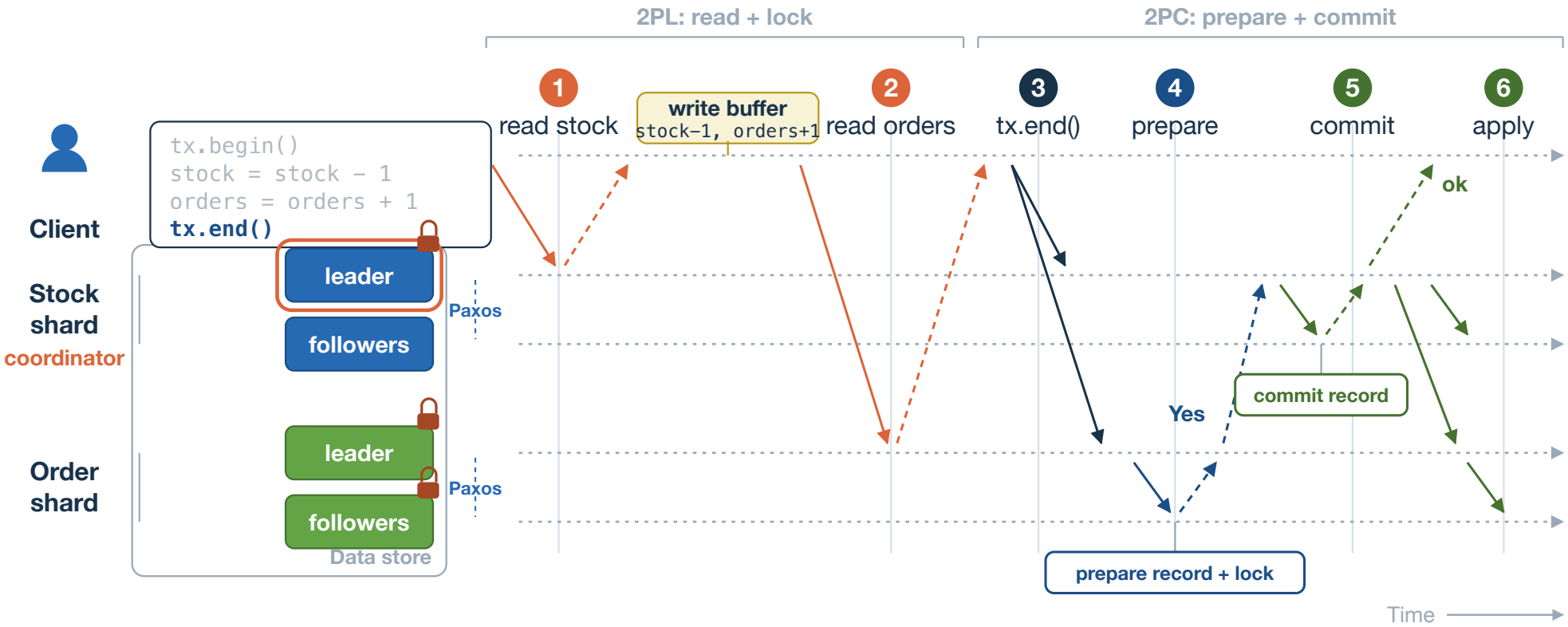
- Off the critical path: the client can get its reply even if this phase is not done
- Why? Followers apply from their own log, even under leader changes



Execution flow of read-write TX: 2PC

Each participant applies the write in one more Paxos round, then unlocks

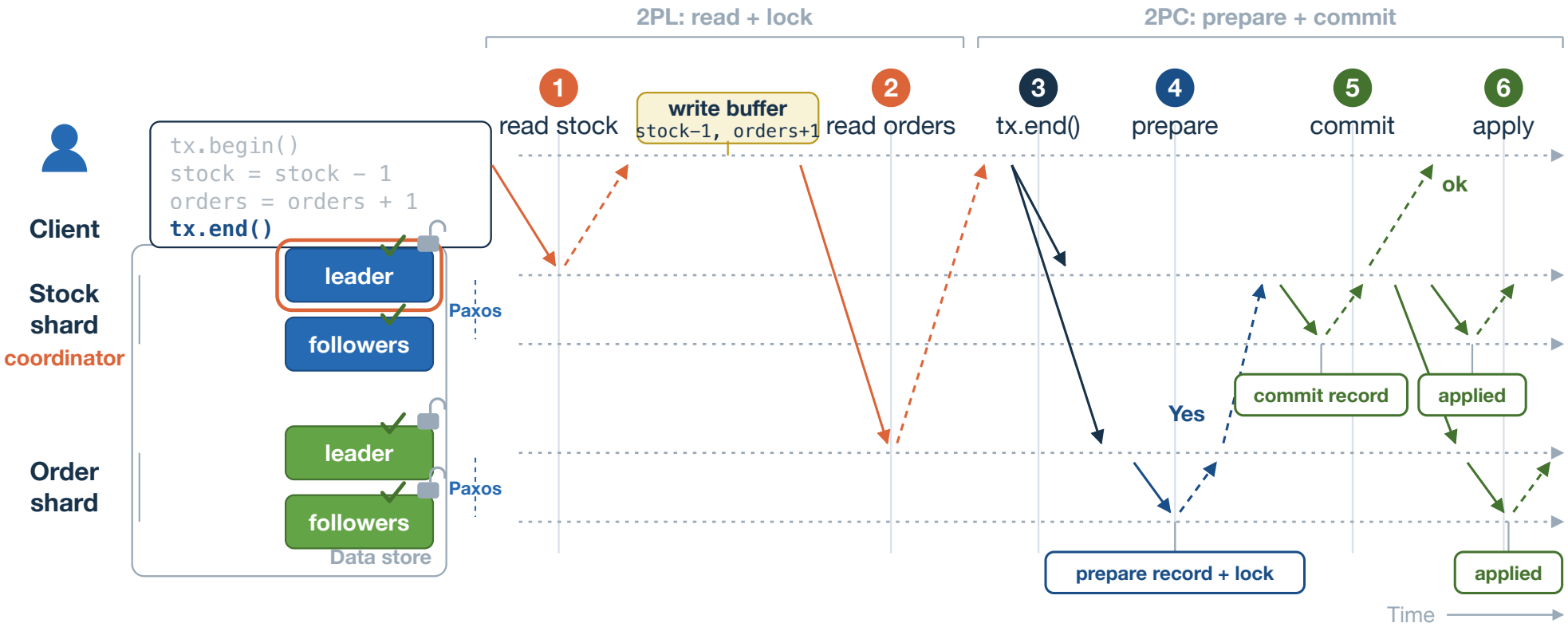
- Off the critical path: the client can get its reply even if this phase is not done
- Why? Followers apply from their own log, even under leader changes



Execution flow of read-write TX: 2PC

Each participant applies the write in one more Paxos round, then unlocks

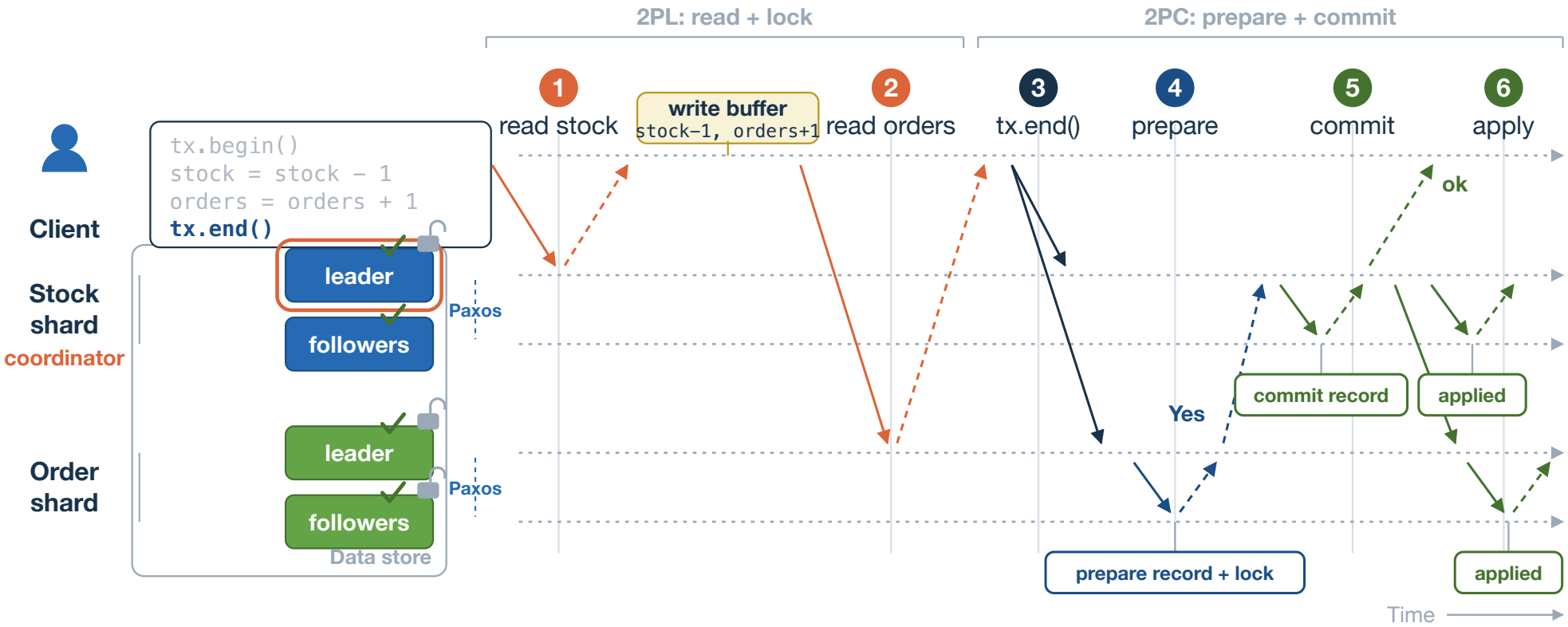
- Off the critical path: the client can get its reply even if this phase is not done
- Why? Followers apply from their own log, even under leader changes



Read-write TX: put it all together

From tx.begin() to the reply: 11 message delays

- 2 reads: 4 delays — 2 client → leader RTTs
- 2PC: tx.end() (both sent in parallel) + 2 Paxos rounds + the Yes + the reply = 7



Read-write TX: it gives a strong abstraction

Spanner's read-write TX gives a strong abstraction to the user

- A single-thread "machine" that never fails
- The "machine" has "unlimited" storage capacity
  - As long as the machines in the datacenters have enough capacity to store the data

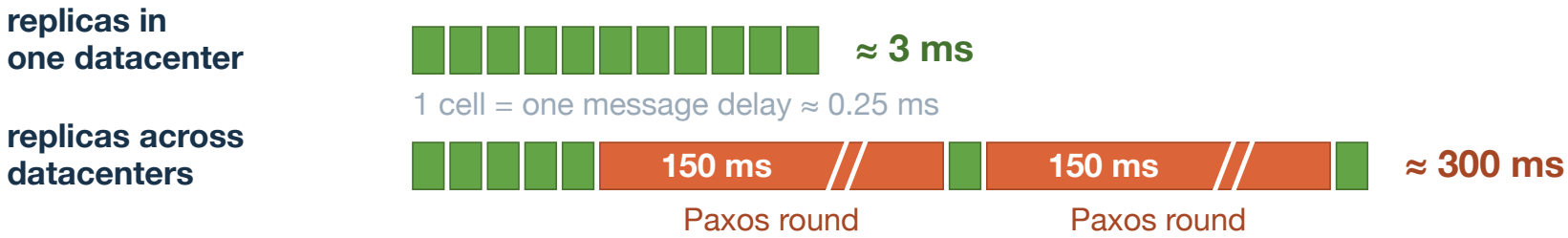
But, what are the costs?

Read-write TX of Spanner so far

2PL & 2PC & Paxos, not optimized — e.g., writes are typically less frequently then the reads

- One datacenter: the 11 message delays cost ≈ 3 ms
- Across datacenters (to survive a datacenter failure): the 2 Paxos rounds dominate

[1]



2PC participants	commit latency, mean (ms)	99th percentile (ms)
1	14.6 ± 0.2	26.550 ± 6.2
10	22.8 ± 0.4	45.931 ± 4.2
200	122.5 ± 6.7	206.443 ± 15.8

[2]

[1] Source: Colin Scott's interactive latency chart

[2] Table IV (two-phase commit scalability); §5.1: "Clients and zones were placed in a set of datacenters with network distance of less than 1ms." Source: Corbett et al., *Spanner: Google's Globally-Distributed Database*, ACM TOCS 31(3), 2013

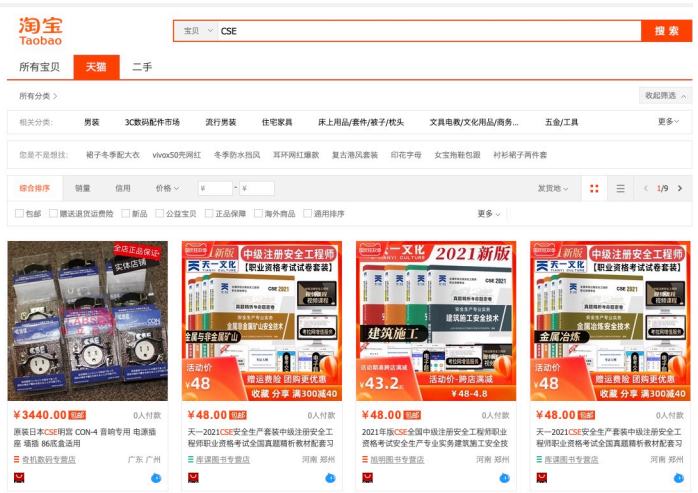
Problem: 2PL does not fit read-only TX

Read-only TXs are common

- Browsing order lists, reports: each read-only TX may touch many shards

Drawbacks of 2PL (in read-only TX)

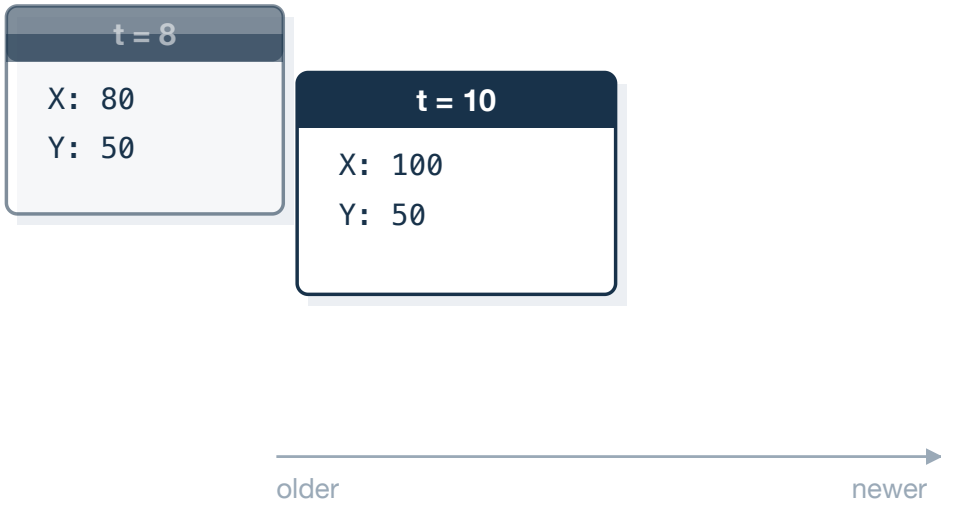
- Acquiring the read lock is costly (need Paxos!)
- May hold the item locks for a long time



Solution: multi-version concurrency control (MVCC)

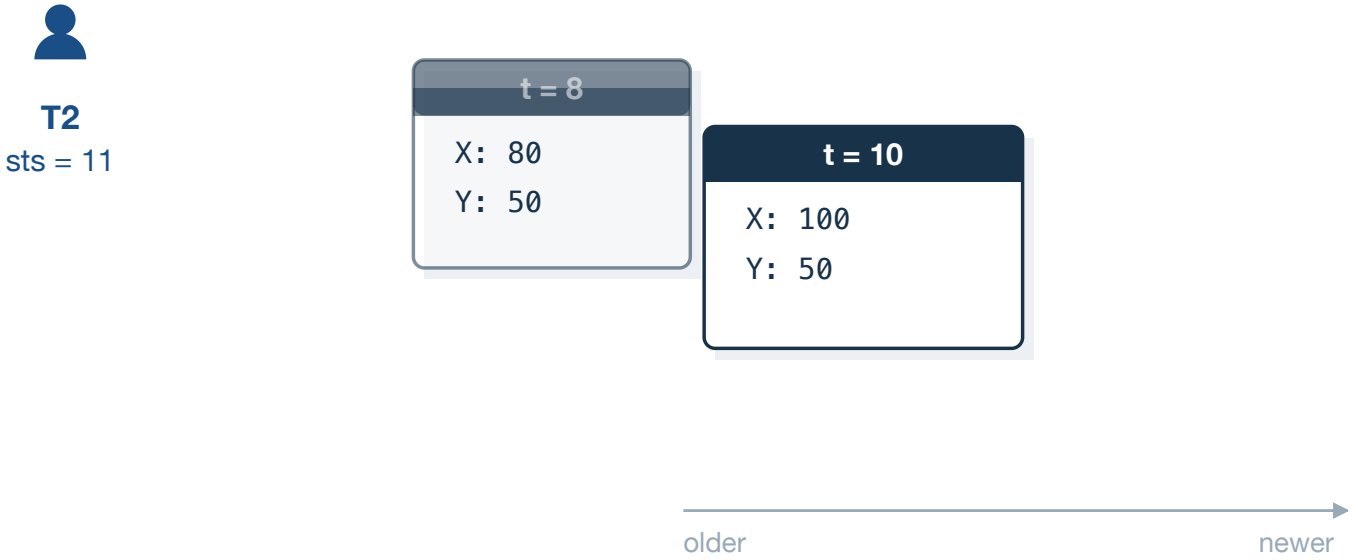
Review: multi-version concurrency control

Think of a time machine: every COMMIT leaves a snapshot of the whole data, stamped with its commit time



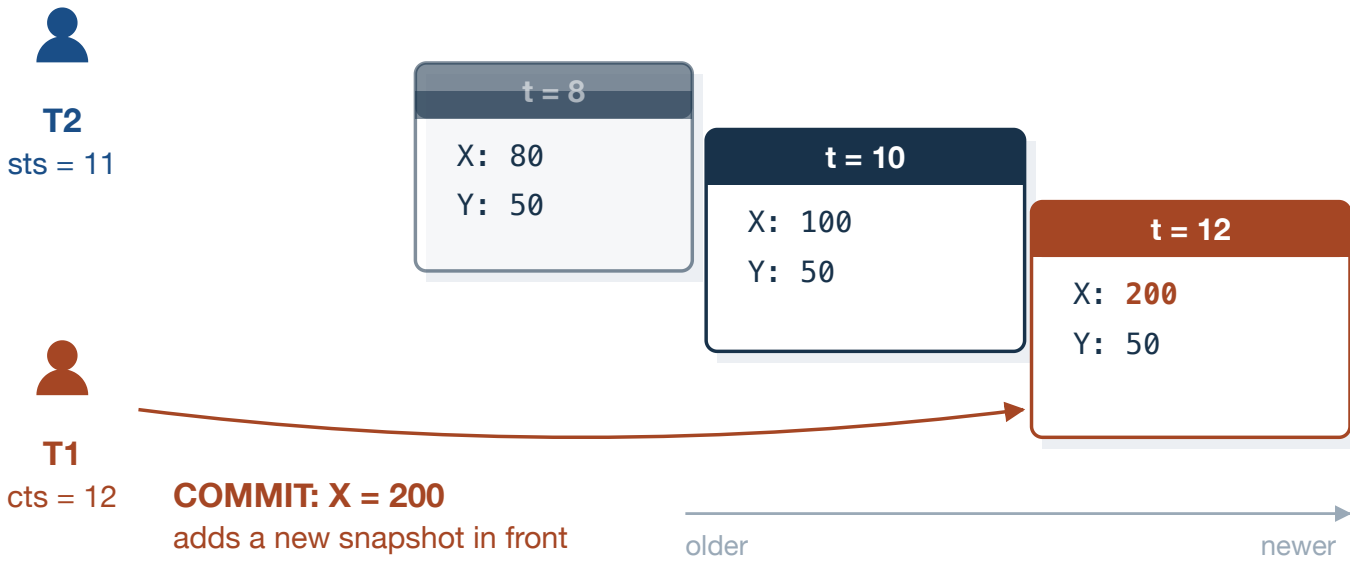
Review: multi-version concurrency control

Think of a time machine: every COMMIT leaves a snapshot of the whole data, stamped with its commit time



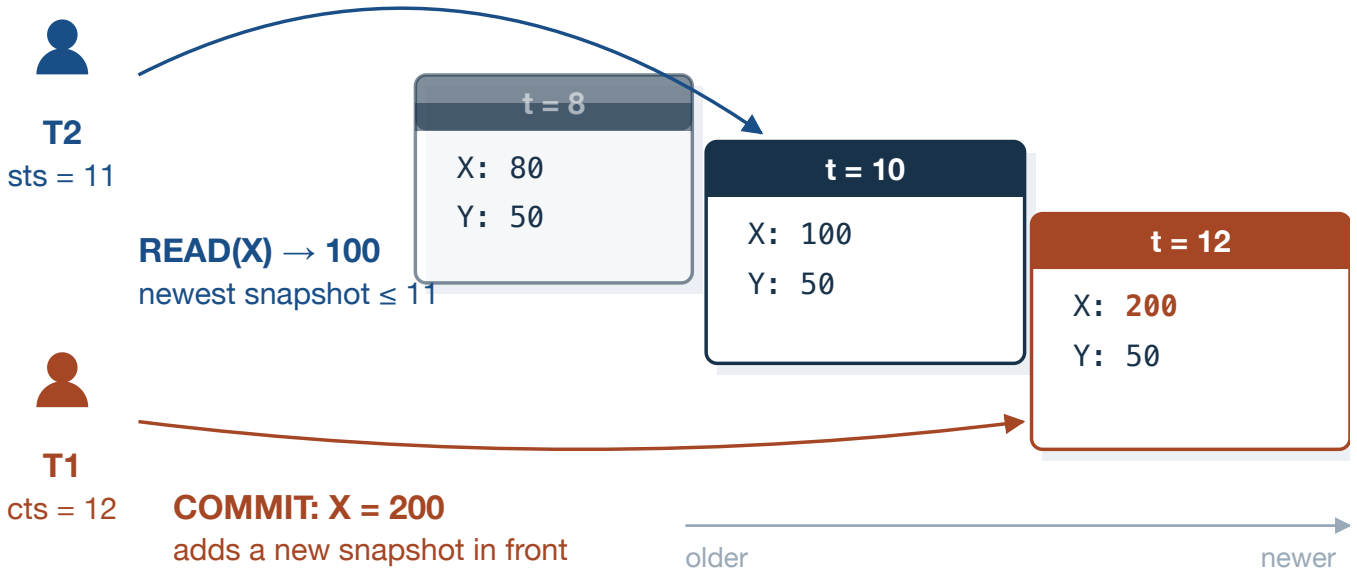
Review: multi-version concurrency control

Think of a time machine: every COMMIT leaves a snapshot of the whole data, stamped with its commit time



Review: multi-version concurrency control

Think of a time machine: every COMMIT leaves a snapshot of the whole data, stamped with its commit time



Writes don't overwrite; T1's COMMIT just adds a new snapshot in front

- A read-only TX reads from the newest snapshot <= its start time (T2: t = 10), so it never waits for a lock

Idea: use 2PL for read-write TX, MVCC for read-only TX

Still use 2PL for read-write TX

- The execution of TXs is always serializable!

Use MVCC to execute the read-only TX

- Read-only TX no longer acquires locks for the execution

Note that a similar technique also applies to OCC

- E.g., we can use OCC for read-write TX, and MVCC for read-only TX (out of the scope of this course)

MVCC for the reads, 2PL for the writes

MV-2PL: read-write TX (w/o 2PC & Paxos for simplicity)

The read-write TX runs as before

- COMMIT installs every item of `T.wset` as a new version stamped `T.cts`



MV-2PL: read-write TX (w/o 2PC & Paxos for simplicity)

The read-write TX runs as before

- COMMIT installs every item of `T.wset` as a new version stamped `T.cts`



MV-2PL: read-write TX (w/o 2PC & Paxos for simplicity)

The read-write TX runs as before

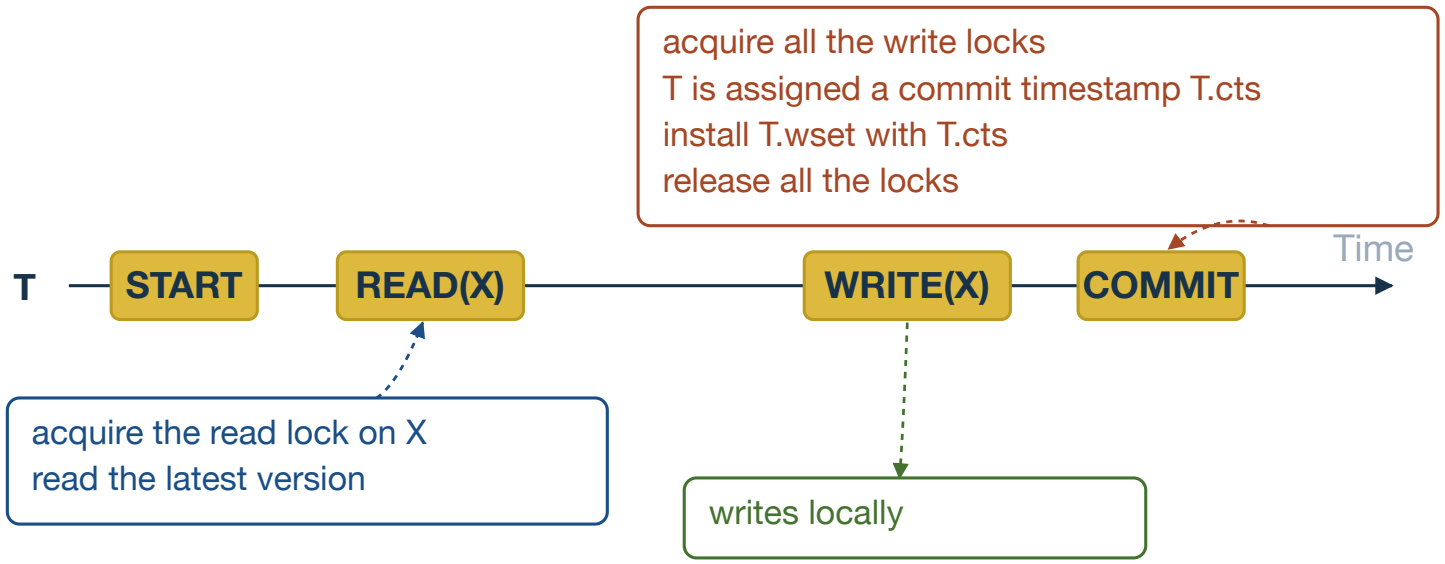
- COMMIT installs every item of `T.wset` as a new version stamped `T.cts`



MV-2PL: read-write TX (w/o 2PC & Paxos for simplicity)

The read-write TX runs as before

- COMMIT installs every item of `T.wset` as a new version stamped `T.cts`

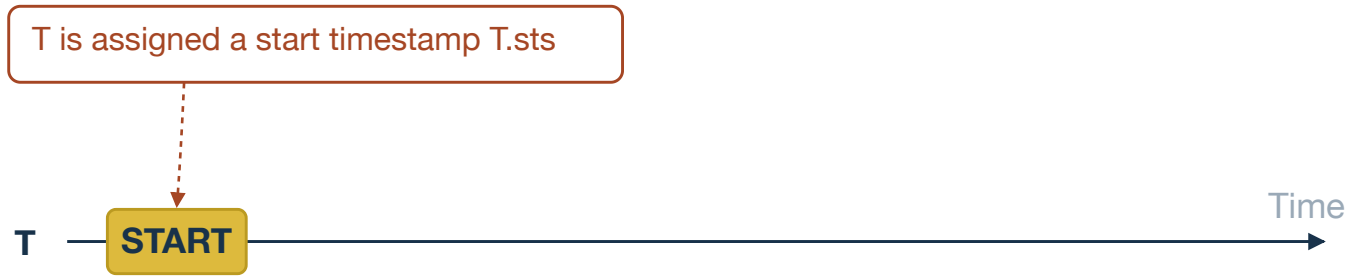


2PL is unchanged — commit now installs a new snapshot

MV-2PL: read-only TX

The read-only TX takes no lock at all — one timestamp is enough

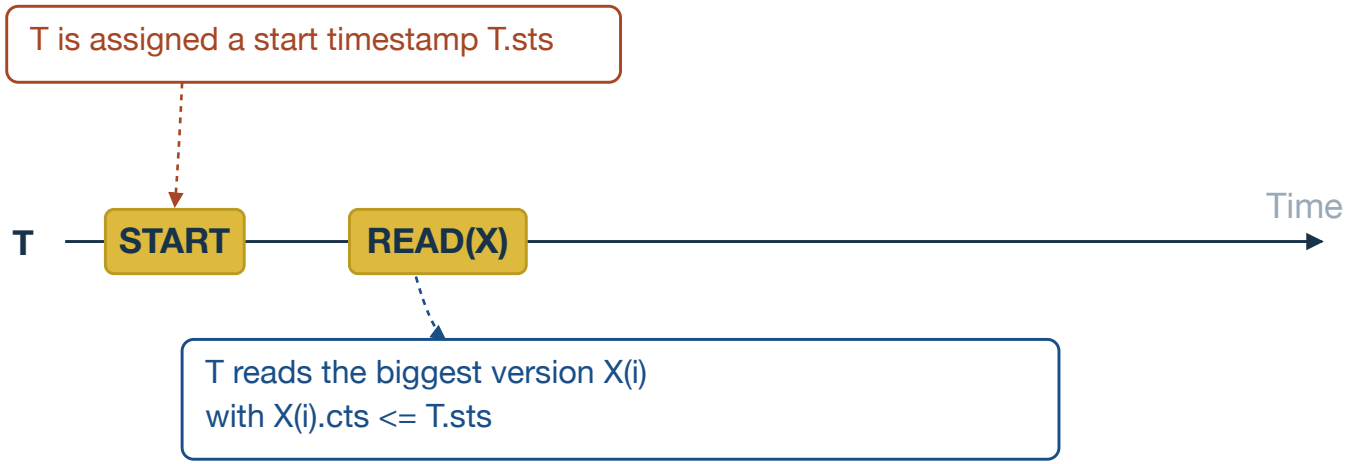
- At START, T is assigned a start timestamp `T.sts`
- Every READ returns the biggest version X(i) with `X(i).cts <= T.sts`



MV-2PL: read-only TX

The read-only TX takes no lock at all — one timestamp is enough

- At START, T is assigned a start timestamp `T.sts`
- Every READ returns the biggest version X(i) with `X(i).cts <= T.sts`



One timestamp pins a whole consistent snapshot

Question remains: how do we assign the time to TXs?

Commit timestamps must be unique and increasing: they number the snapshots

- Example implementation: a global counter on a single machine, `FAA` on every commit

A read-only TX only needs a start timestamp e.g., the latest counter

```
u64 global_time; // initially 0
struct TX {
   u64 start_time;
   u64 commit_time;
   set<...> write_set;
}
```

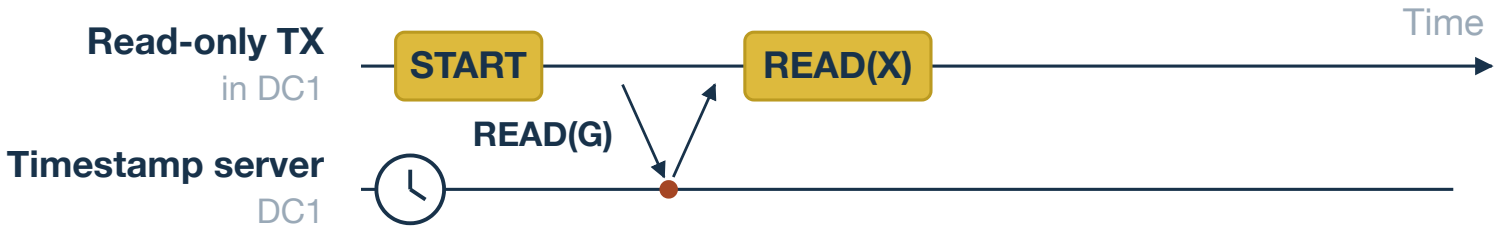
```
tx_begin(tx)  { // read-only: READ, never FAA
   tx.start_time = READ(global_time);
}
tx_commit(tx) { // read-write
   tx.commit_time = FAA(global_time);
}
```

Challenge: Spanner operates at the geo-wide, so a single counter may be insufficient

Global time is inefficient for Spanner's use case

The global counter lives on one machine — the timestamp server

- 1. Extra latency — one more round trip, and the TX may sit in a DC on another continent
- 2. Scalability bottleneck — one machine serves every TX in the system

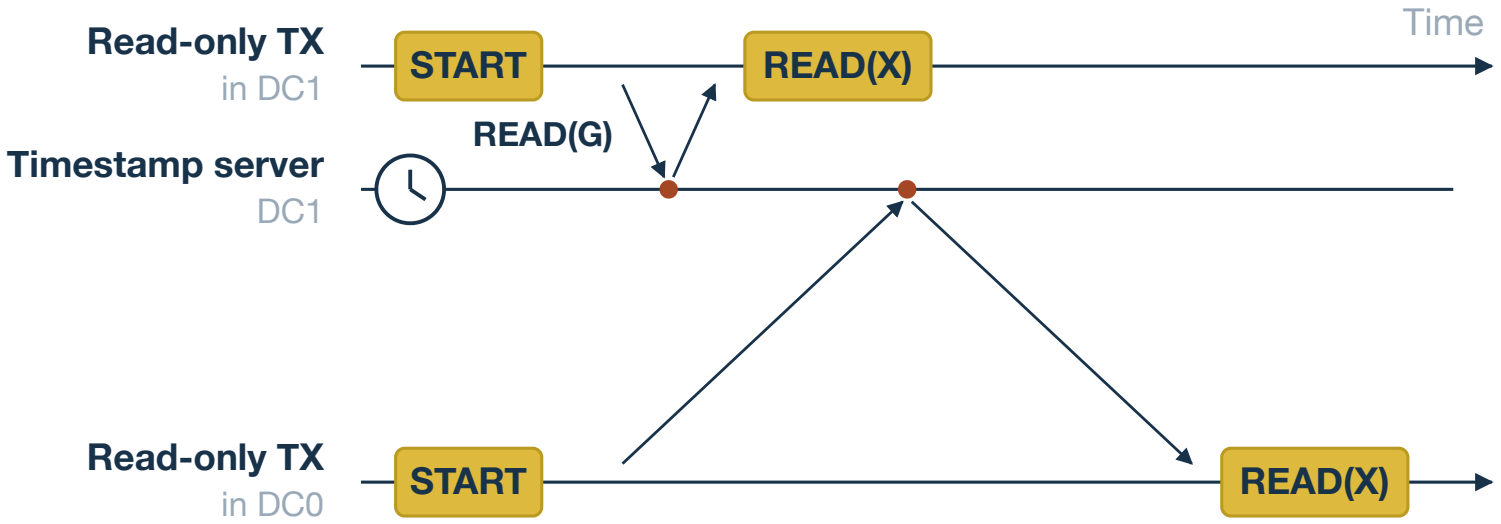


The same read-only TX, run in two places: the overseas one reads long after the local one

Global time is inefficient for Spanner's use case

The global counter lives on one machine — the timestamp server

- 1. Extra latency — one more round trip, and the TX may sit in a DC on another continent
- 2. Scalability bottleneck — one machine serves every TX in the system

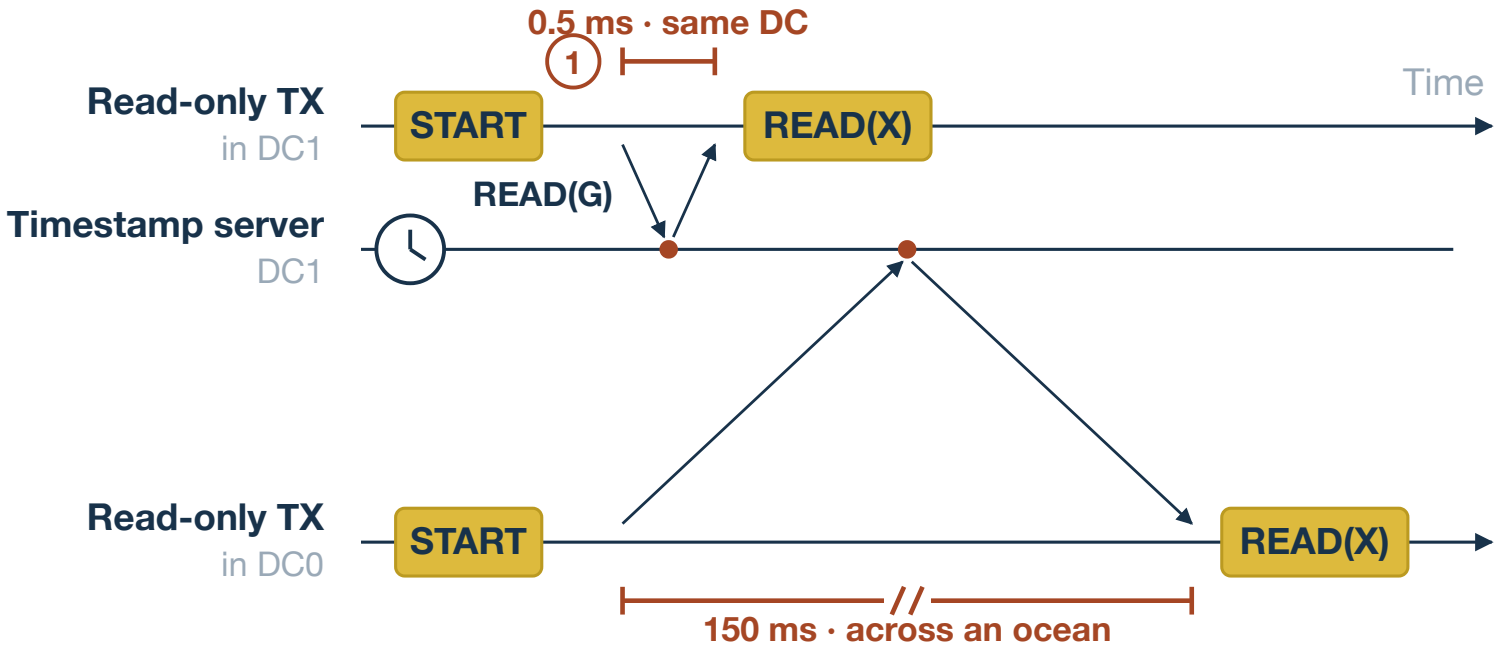


The same read-only TX, run in two places: the overseas one reads long after the local one

Global time is inefficient for Spanner's use case

The global counter lives on one machine — the timestamp server

- 1. Extra latency — one more round trip, and the TX may sit in a DC on another continent
- 2. Scalability bottleneck — one machine serves every TX in the system

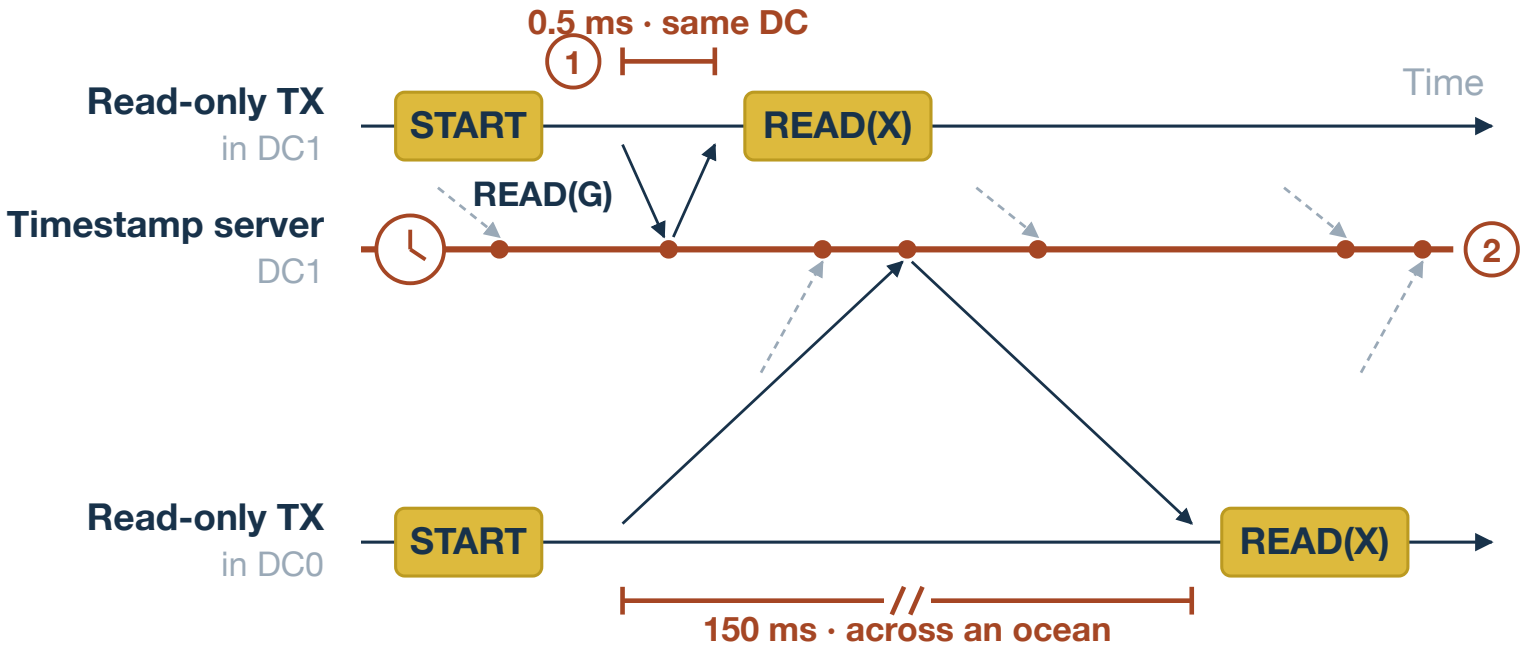


The same read-only TX, run in two places: the overseas one reads long after the local one

Global time is inefficient for Spanner's use case

The global counter lives on one machine — the timestamp server

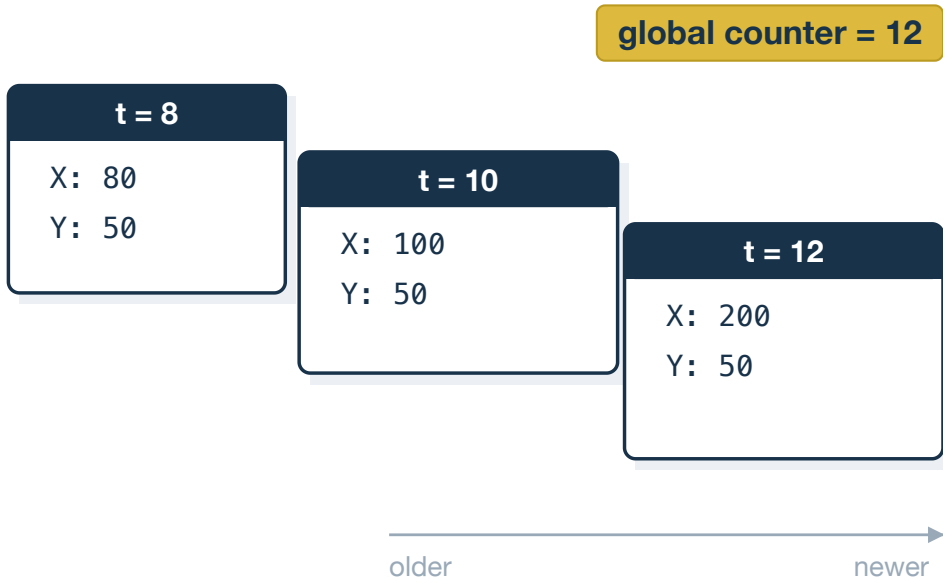
- 1. Extra latency — one more round trip, and the TX may sit in a DC on another continent
- 2. Scalability bottleneck — one machine serves every TX in the system



The same read-only TX, run in two places: the overseas one reads long after the local one

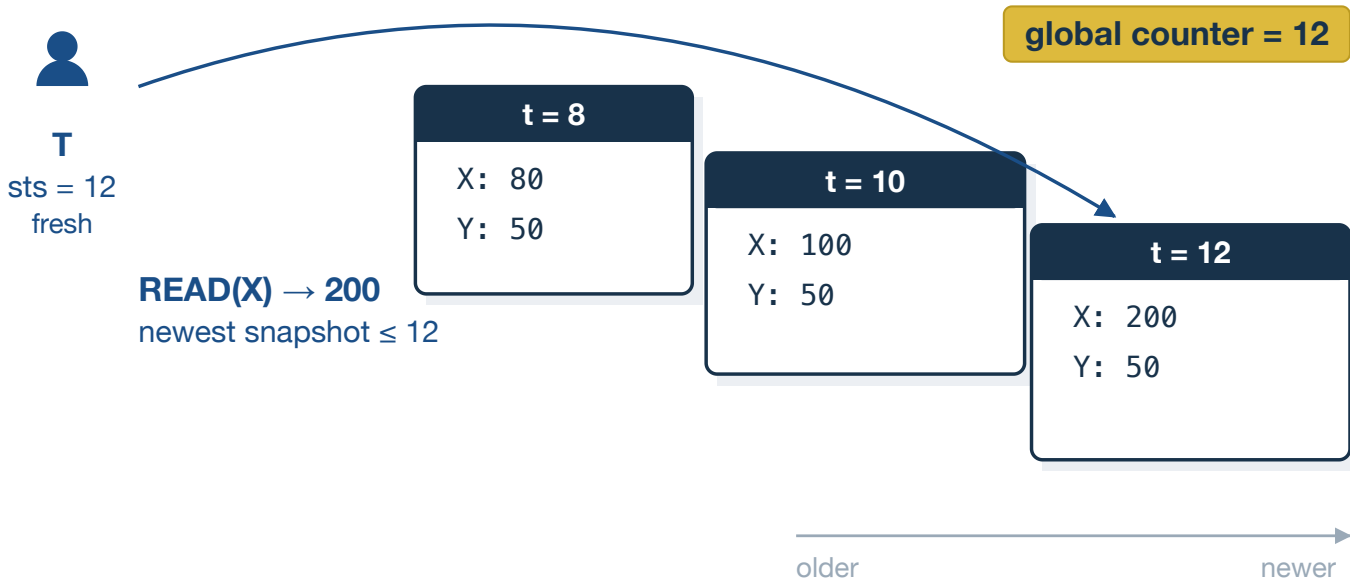
Possible solution: Cache the time locally to avoid querying the global counter

Observation: the read-only TX can read a stale timestamp, i.e., not the latest global counter



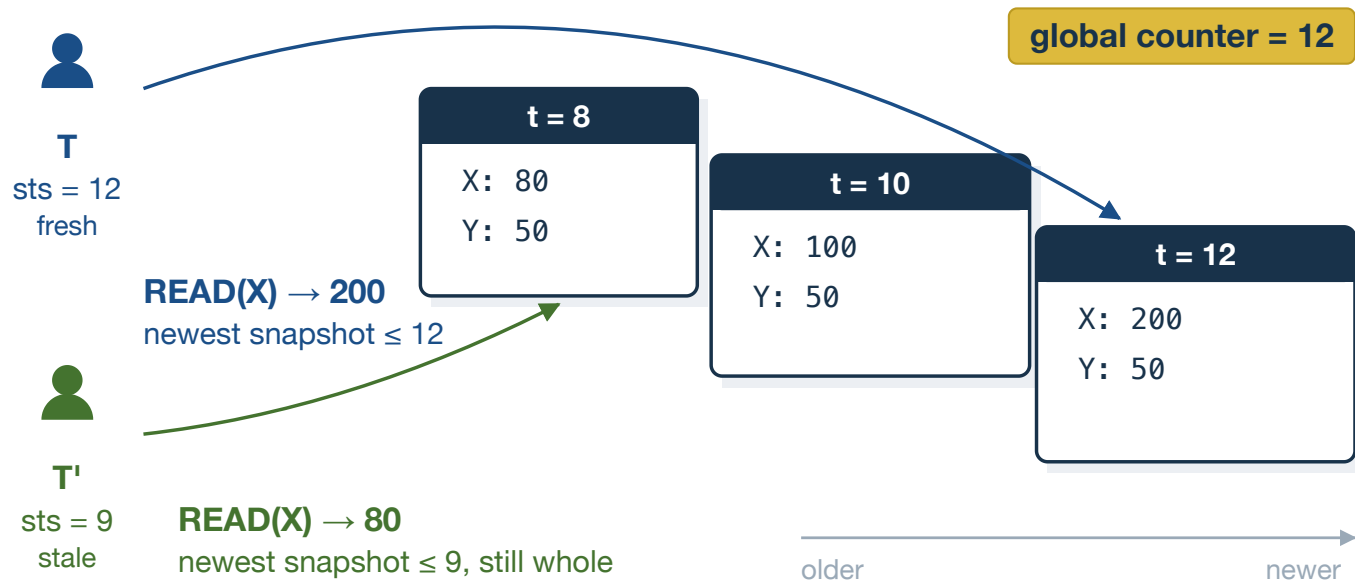
Possible solution: Cache the time locally to avoid querying the global counter

Observation: the read-only TX can read a stale timestamp, i.e., not the latest global counter



Possible solution: Cache the time locally to avoid querying the global counter

Observation: the read-only TX can read a stale timestamp, i.e., not the latest global counter



Still correct: a stale sts just picks an older (but consistent) snapshot

Read-only TXs may use a stale timestamp

Cache the global counter to avoid frequent reads

Each datacenter has a dedicated timestamp server (cached timestamp server)

- It periodically fetches the timestamp from the authoritative global counter

The read-only TX now only reads the timestamp from the cached server w/o cross-DC RTT

```
struct TX {
   ...
   u64 cached_time;                          // new; 0 = not cached yet
}

tx_begin(tx) {
   if (tx.cached_time == 0)
      tx.cached_time = READ(global_time);  // once, then reuse
   tx.start_time = tx.cached_time;
}
```

Example revisit w/ cached time

T2 commits at time 3 and installs A3 / B3; T1 starts with the cached 0

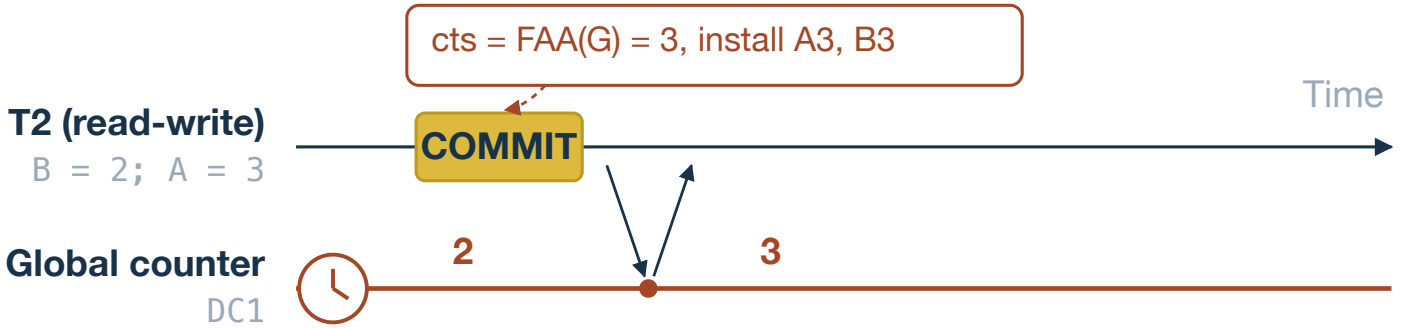


2 snapshots

t:	0	1
A:	A0	A1
B:	B0	B1

Example revisit w/ cached time

T2 commits at time 3 and installs A3 / B3; T1 starts with the cached 0

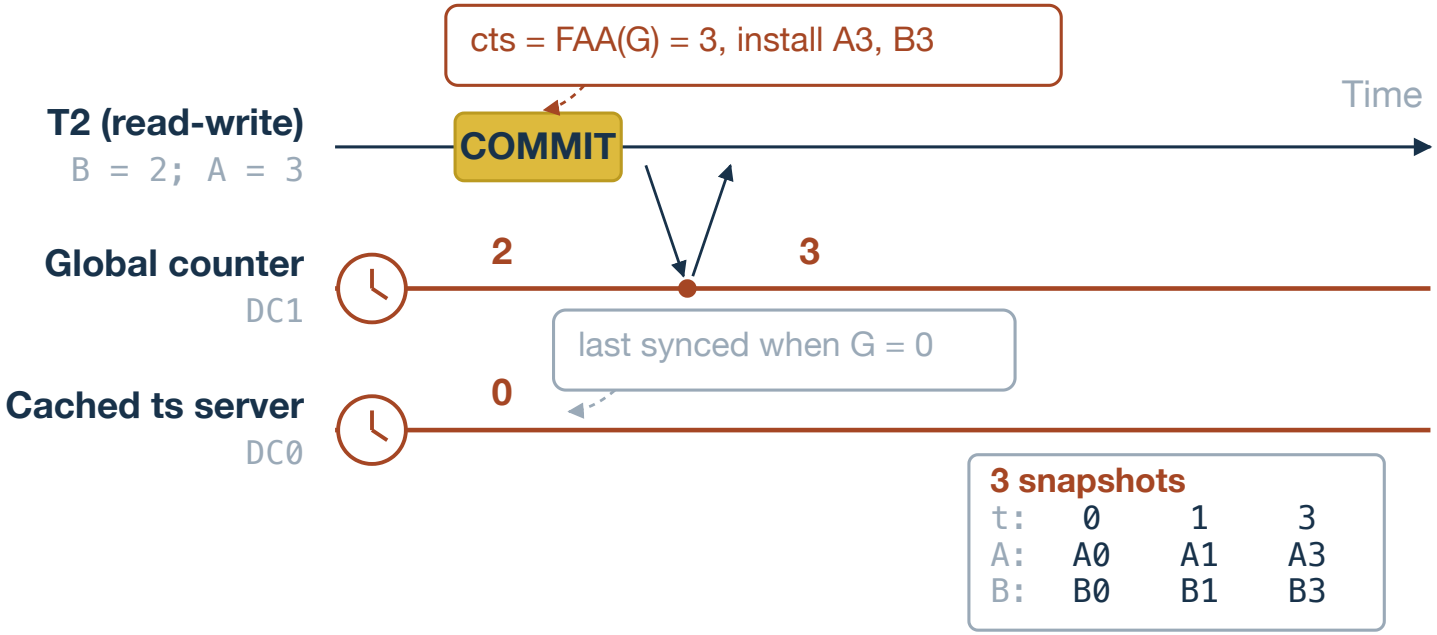


3 snapshots

t:	0	1	3
A:	A0	A1	A3
B:	B0	B1	B3

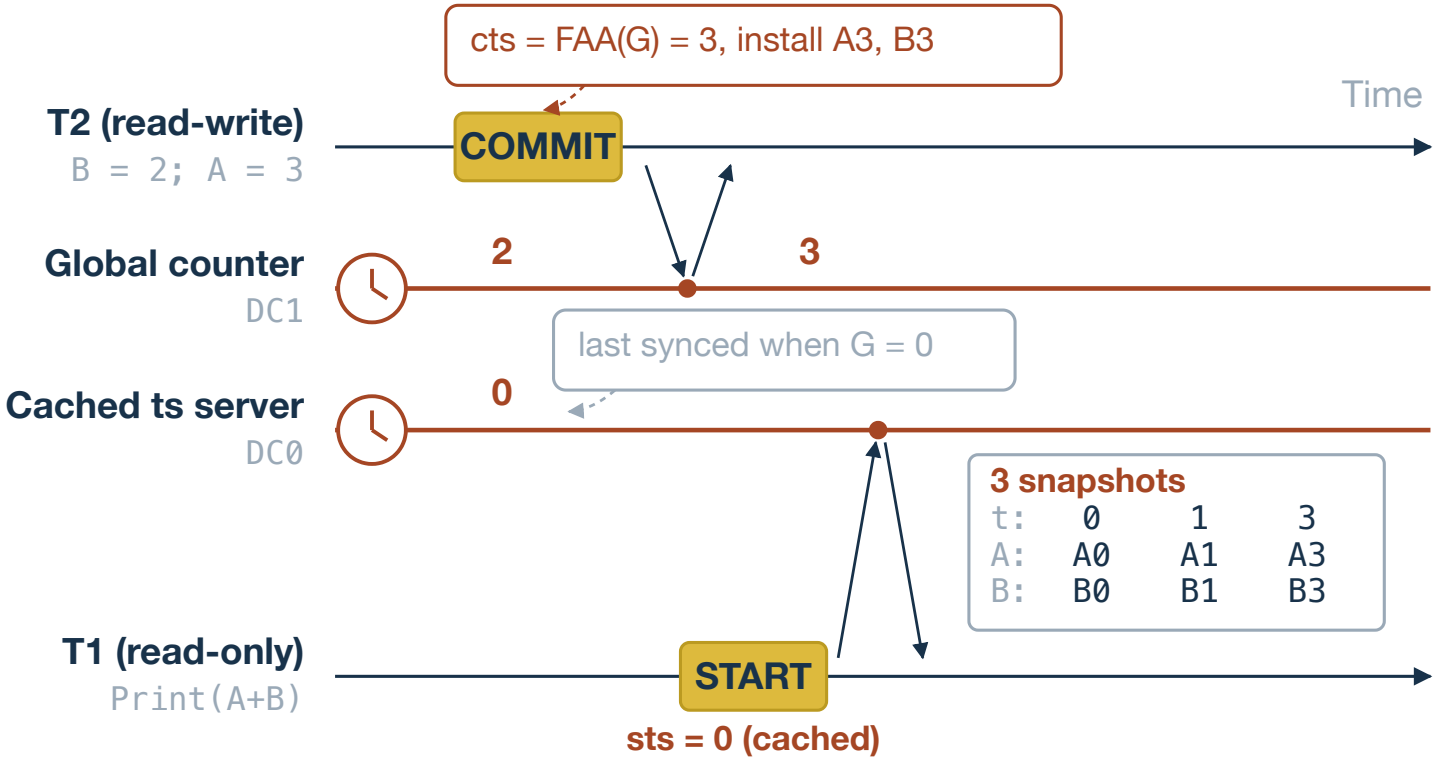
Example revisit w/ cached time

T2 commits at time 3 and installs A3 / B3; T1 starts with the cached 0



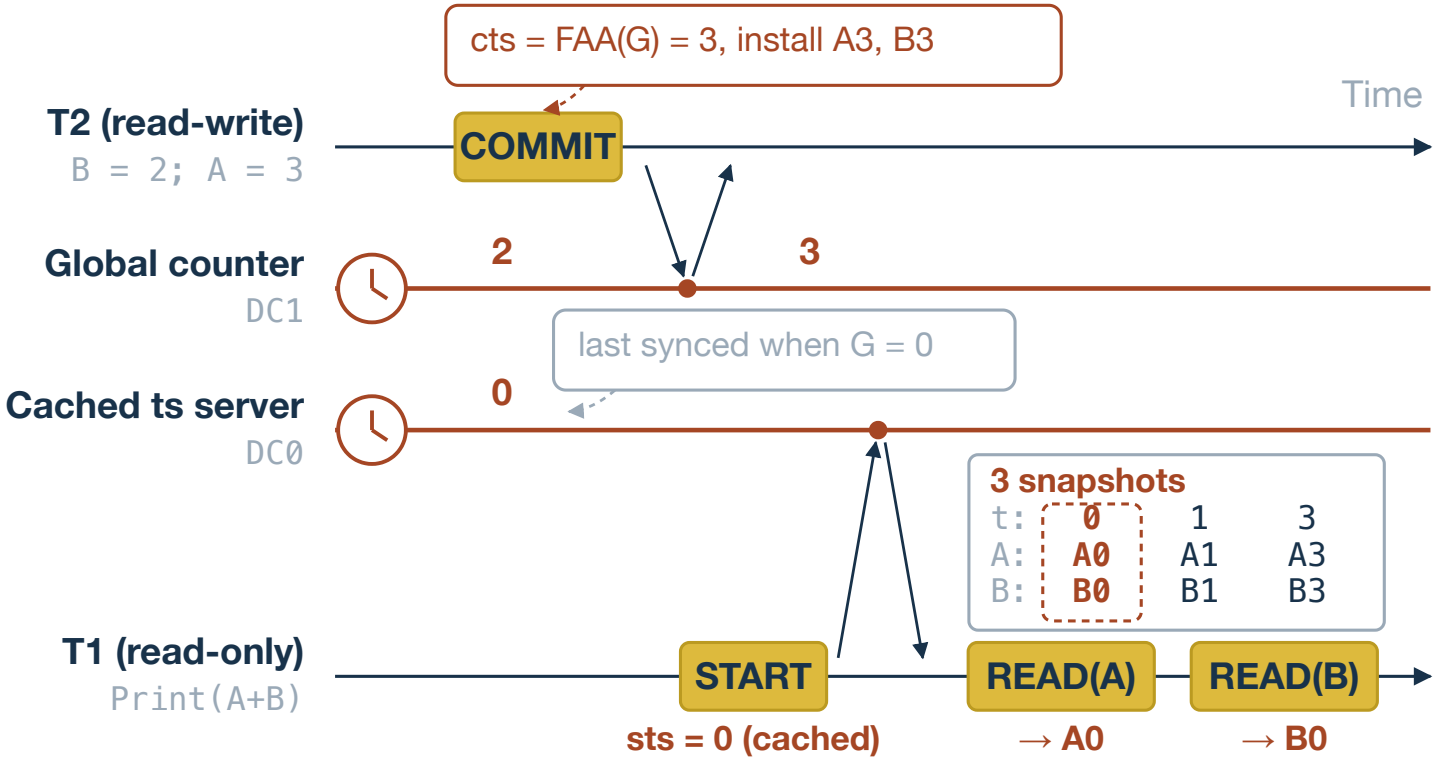
Example revisit w/ cached time

T2 commits at time 3 and installs A3 / B3; T1 starts with the cached 0



Example revisit w/ cached time

T2 commits at time 3 and installs A3 / B3; T1 starts with the cached 0



A cached timestamp reads an older but consistent snapshot

Drawbacks of cached time

No freshness guarantees — i.e., no external consistency

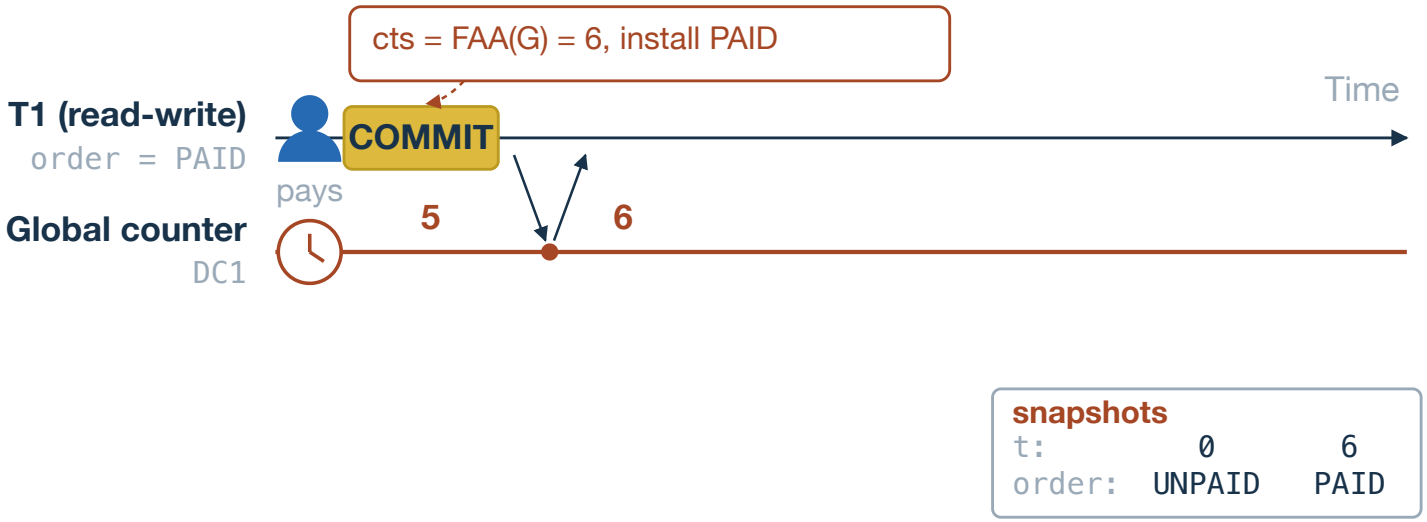
- Simplified definition: *If T1 completes before T2 starts, T2 must see T1's writes.*



Drawbacks of cached time

No freshness guarantees — i.e., no external consistency

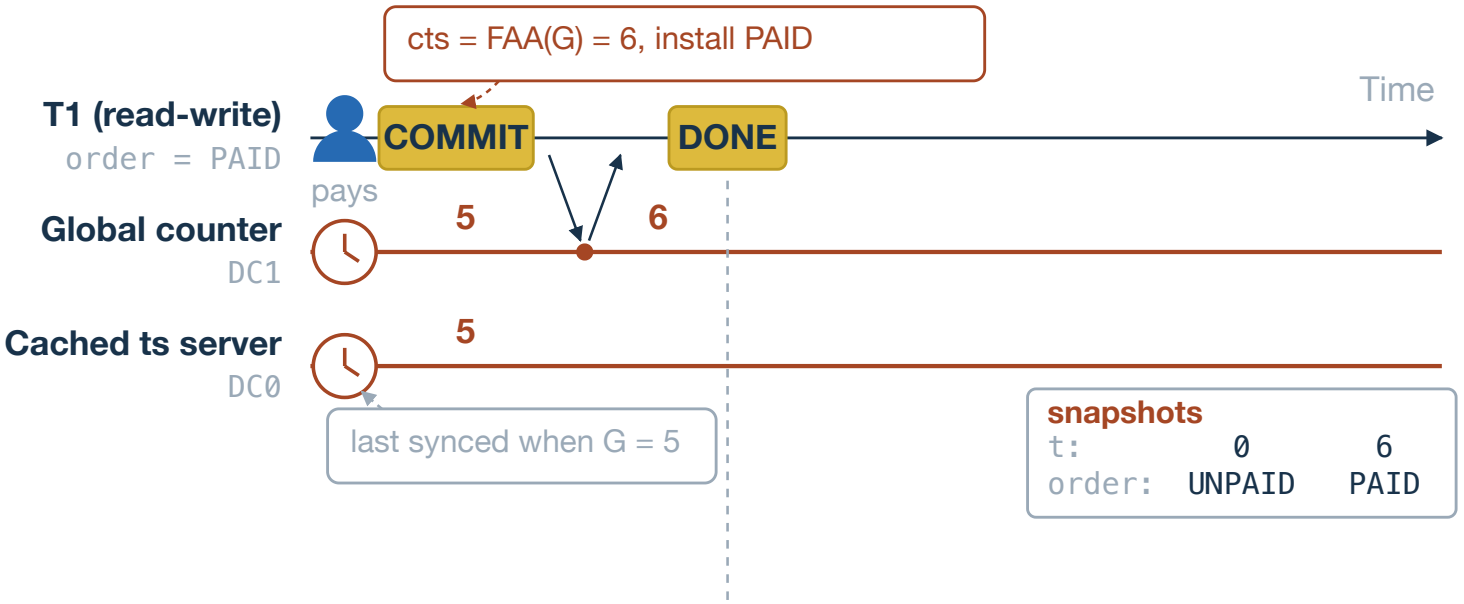
- Simplified definition: *If T1 completes before T2 starts, T2 must see T1's writes.*



Drawbacks of cached time

No freshness guarantees — i.e., no external consistency

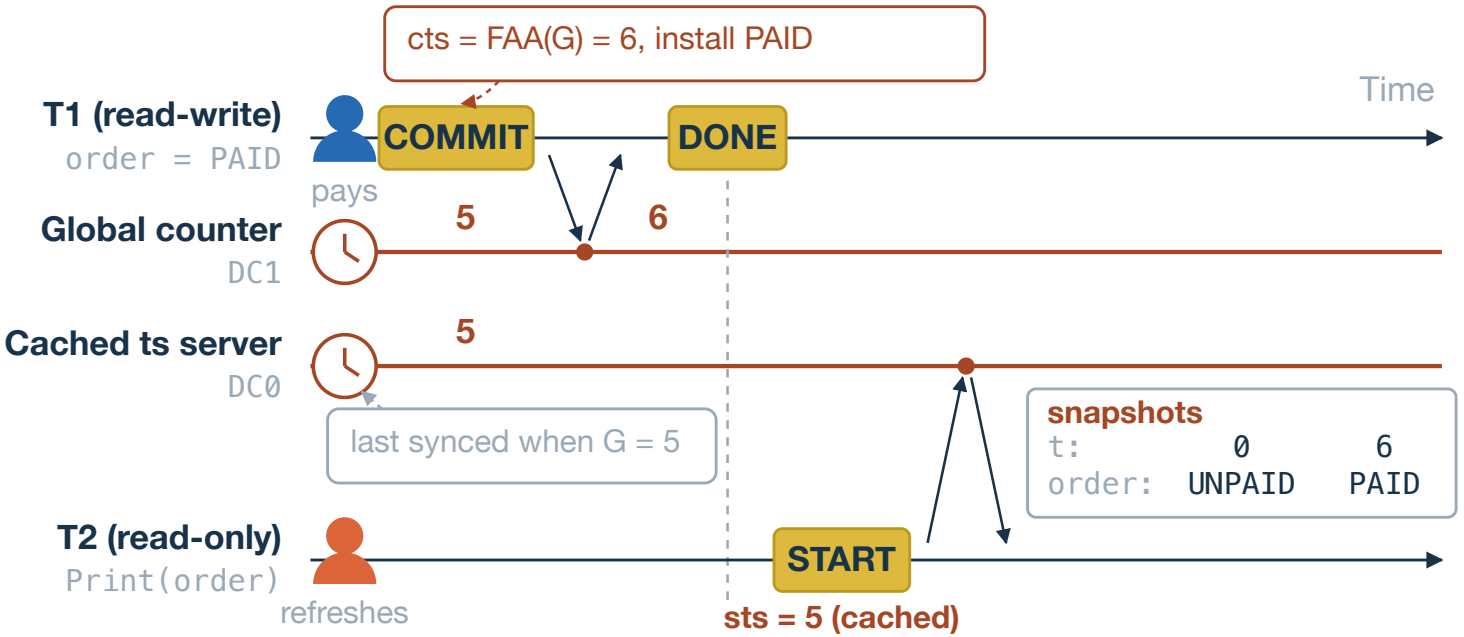
- Simplified definition: *If T1 completes before T2 starts, T2 must see T1's writes.*



Drawbacks of cached time

No freshness guarantees — i.e., no external consistency

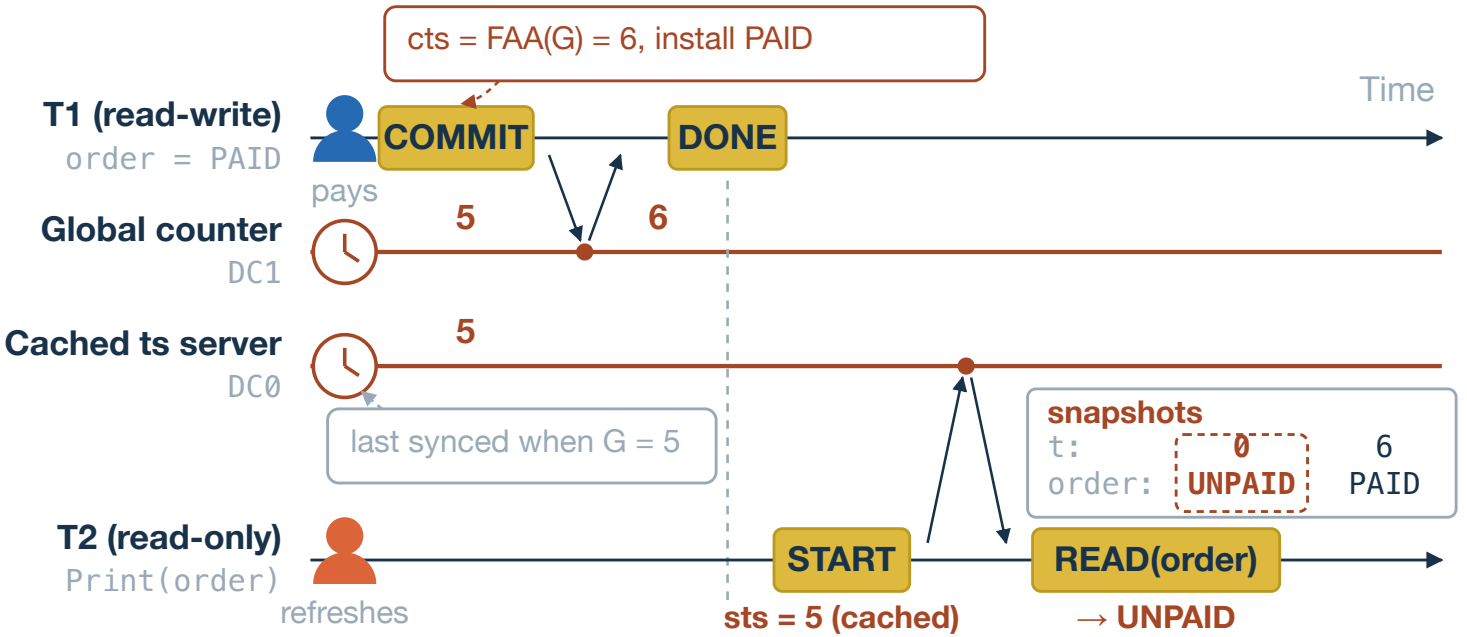
- Simplified definition: *If T1 completes before T2 starts, T2 must see T1's writes.*



Drawbacks of cached time

No freshness guarantees — i.e., no external consistency

- Simplified definition: *If T1 completes before T2 starts, T2 must see T1's writes.*



The buyer pays, refreshes, however, still reads UNPAID

Challenge: implement external consistency without a global clock

Time needs synchronization for external consistency (strict serializability)

- If a read-only TX starts after a read-write TX has committed, we must ensure:

$$\text{Time}_{ro} \geq \text{Time}_{rw}$$

Global counter trivially satisfies this requirement

- But requires a centralized time server, not suitable for geo-replicated databases

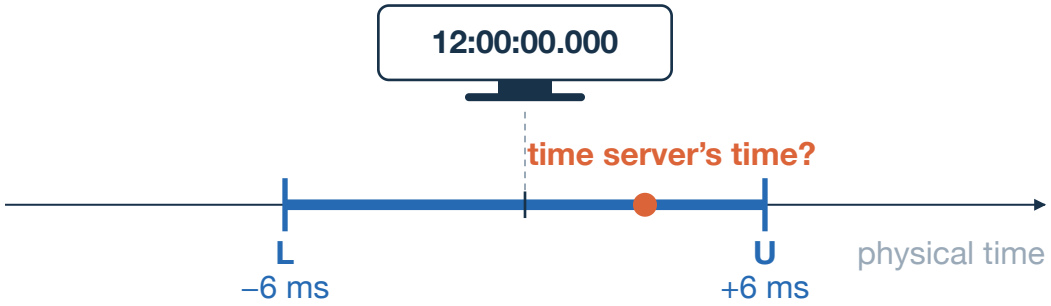
Can we use the physical clock of the machines?

- No. Different machines' clocks differ

Observation: an accurate bound (interval)

Example: a machine's clock reads 12:00:00.000

- It synced with the time server 30 s ago; its clock drifts at most 200 µs per second
- So the time server's time is within 30 × 200 µs = 6 ms of it [1]



We may not get the accurate physical time, but we can get an accurate bound

[1] Spanner's applied drift rate is 200 microseconds/second: Corbett et al., *Spanner*, OSDI 2012, §3

TrueTime API of Spanner

TrueTime returns a time interval instead of a single point of time

- The interval is a bound: the time server's time must be inside it
- Its endpoints are ordinary physical time, the unit users already know

```
// Linux time API
struct timeval { time_t tv_sec; suseconds_t tv_usec; };
int gettimeofday(struct timeval *restrict tv, ...);
```

```
// TrueTime API (simplified)
struct time_interval { timeval Lower; timeval Upper; };
int get_truetime(time_interval *interval);
// the time server's timeval must be in [L,U]
```

One call, two timestamps — the truth is somewhere in between

TrueTime gives external consistency: commit wait

For external consistency: If a rw TX has decided to commit, then $\mathrm{Time}_{ro} \geq \mathrm{Time}_{rw}$

- Commit wait: after acquiring $[L, U]$, the rw TX waits until U has passed, then commits with $\mathrm{Time}_{rw} = U$
- A ro TX simply uses its own U as Time_{ro}



TrueTime gives external consistency: commit wait

For external consistency: If a rw TX has decided to commit, then $\mathrm{Time}_{ro} \geq \mathrm{Time}_{rw}$

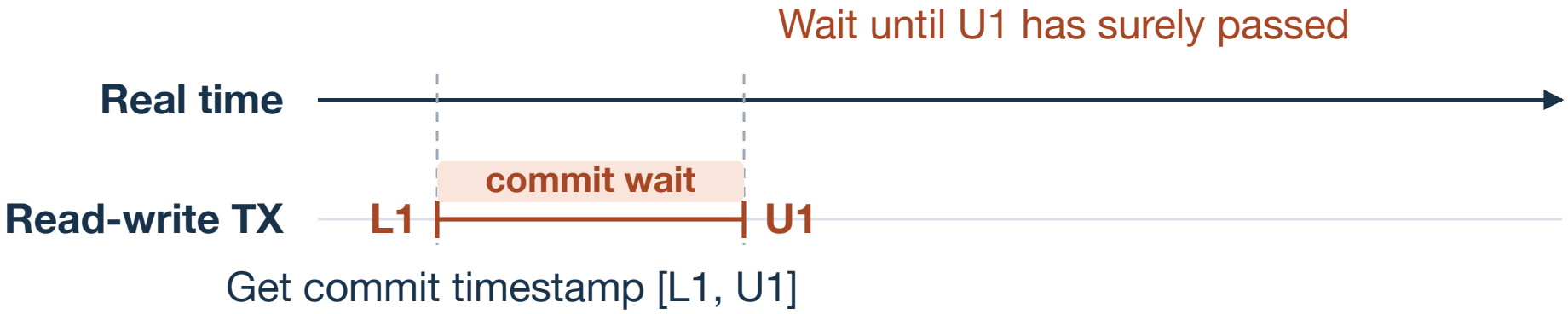
- Commit wait: after acquiring $[L, U]$, the rw TX waits until U has passed, then commits with $\mathrm{Time}_{rw} = U$
- A ro TX simply uses its own U as Time_{ro}



TrueTime gives external consistency: commit wait

For external consistency: If a rw TX has decided to commit, then $\mathrm{Time}_{ro} \geq \mathrm{Time}_{rw}$

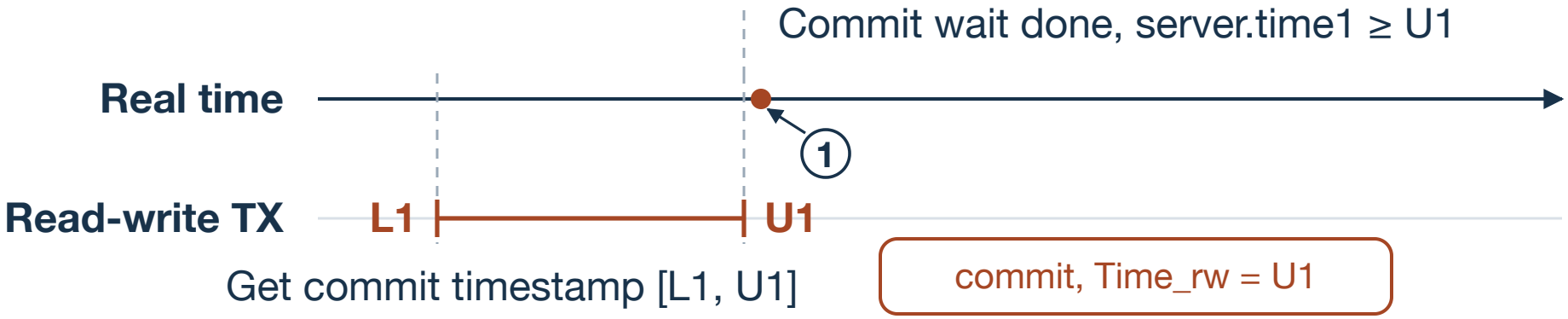
- Commit wait: after acquiring $[L, U]$, the rw TX waits until U has passed, then commits with $\mathrm{Time}_{rw} = U$
- A ro TX simply uses its own U as Time_{ro}



TrueTime gives external consistency: commit wait

For external consistency: If a rw TX has decided to commit, then $\mathrm{Time}_{ro} \geq \mathrm{Time}_{rw}$

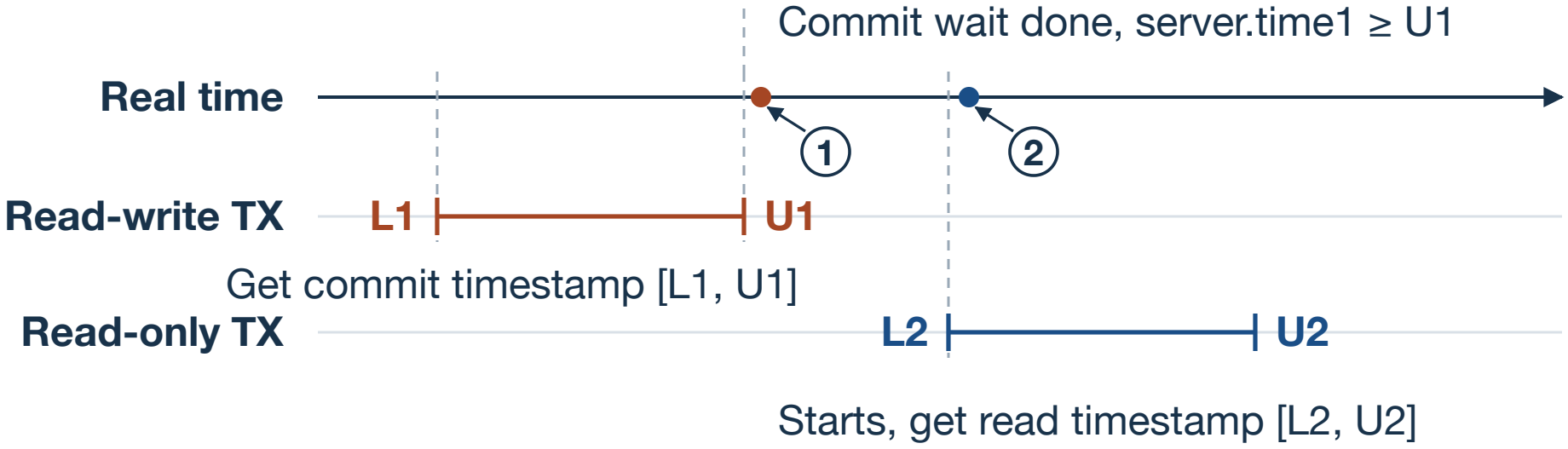
- Commit wait: after acquiring $[L, U]$, the rw TX waits until U has passed, then commits with $\mathrm{Time}_{rw} = U$
- A ro TX simply uses its own U as Time_{ro}



TrueTime gives external consistency: commit wait

For external consistency: If a rw TX has decided to commit, then $\mathrm{Time}_{ro} \geq \mathrm{Time}_{rw}$

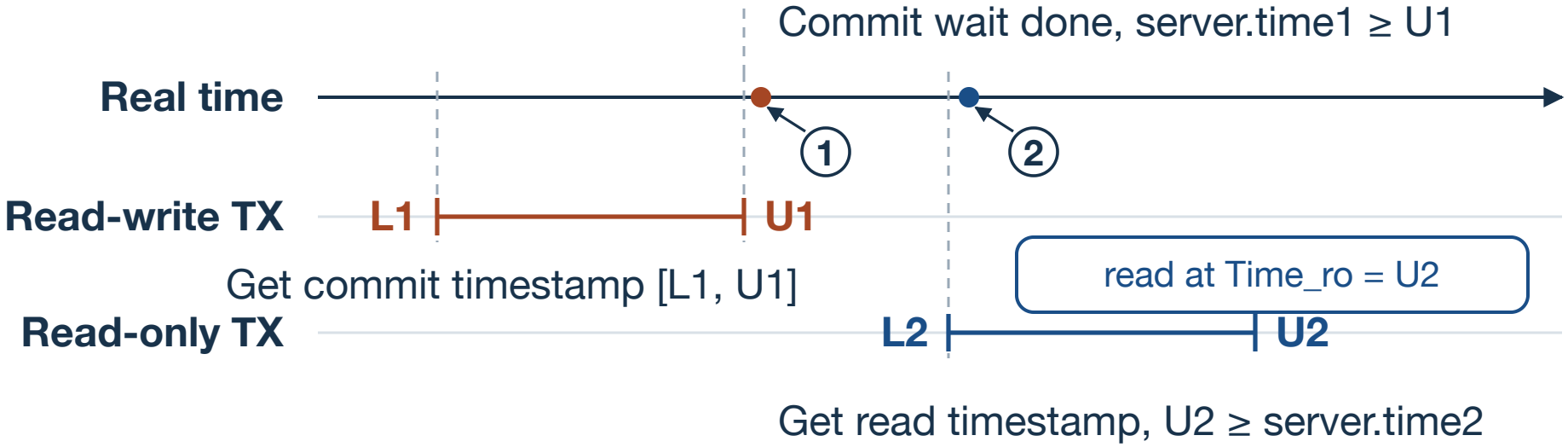
- Commit wait: after acquiring $[L, U]$, the rw TX waits until U has passed, then commits with $\mathrm{Time}_{rw} = U$
- A ro TX simply uses its own U as Time_{ro}



TrueTime gives external consistency: commit wait

For external consistency: If a rw TX has decided to commit, then $\mathrm{Time}_{ro} \geq \mathrm{Time}_{rw}$

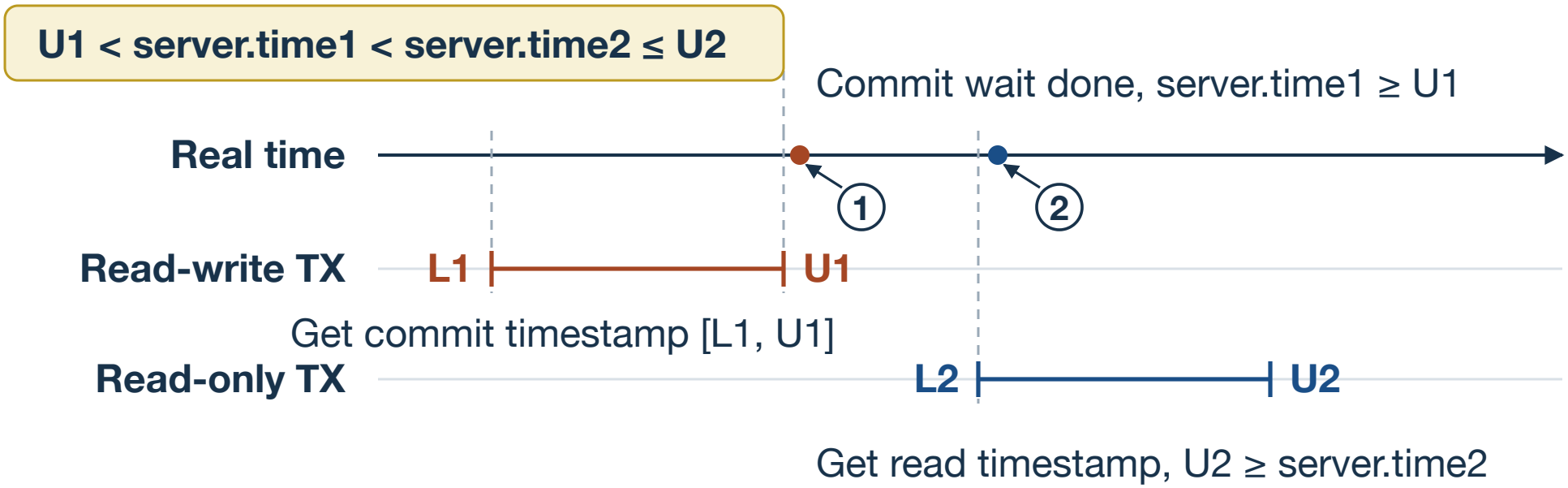
- Commit wait: after acquiring $[L, U]$, the rw TX waits until U has passed, then commits with $\mathrm{Time}_{rw} = U$
- A ro TX simply uses its own U as Time_{ro}



TrueTime gives external consistency: commit wait

For external consistency: If a rw TX has decided to commit, then $\mathrm{Time}_{ro} \geq \mathrm{Time}_{rw}$

- Commit wait: after acquiring $[L, U]$, the rw TX waits until U has passed, then commits with $\mathrm{Time}_{rw} = U$
- A ro TX simply uses its own U as Time_{ro}



Correctness: $\mathrm{Time}_{rw}\ (U_1) < \mathrm{server.time}_1 < \mathrm{server.time}_2 \leq U_2\ (\mathrm{Time}_{ro})$

So far the bound was a black box: get_truetime() hands back [L, U] and the real time is somewhere inside.

Where do L and U come from?

TrueTime: how to achieve the bound?

The bound is relative to the time servers as the ground truth

- For simplicity, we first assume there is only one time server

Spanner adopts a variant of Marzullo's algorithm

- Similar to NTP, sync w/ the time server to calculate the bound
- Even a single time server will not become the bottleneck: syncing is not on the critical path of the TX's execution

Sync happens in the background; a TX only reads the local bound

TrueTime: how to calculate the bound?

Problem statement: the client's clock reads T — what is the time server's Ts?

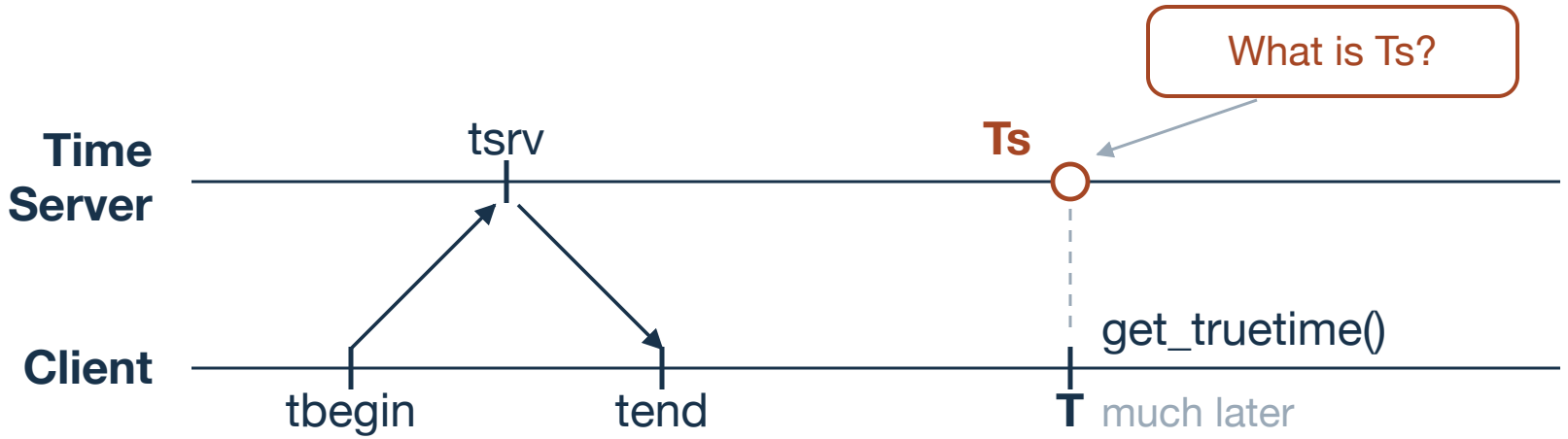
- The only measurements: tbegin, tsrv, tend from a background sync RPC
- For now, assume client and server tick at the same speed



TrueTime: how to calculate the bound?

Problem statement: the client's clock reads T — what is the time server's Ts?

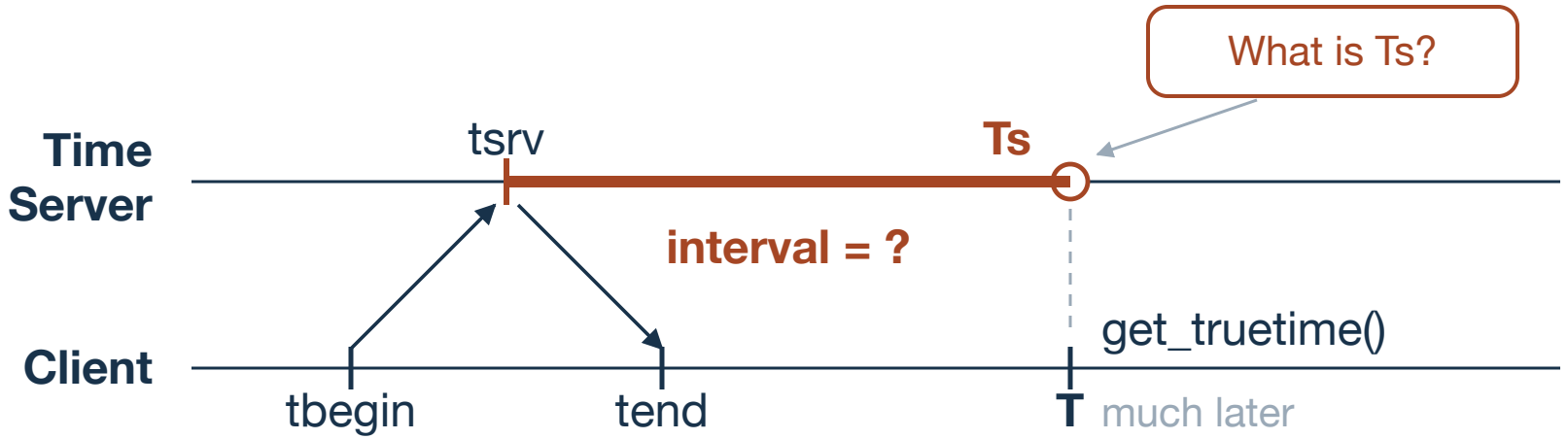
- The only measurements: tbegin, tsrv, tend from a background sync RPC
- For now, assume client and server tick at the same speed



TrueTime: how to calculate the bound?

Problem statement: the client's clock reads T — what is the time server's Ts?

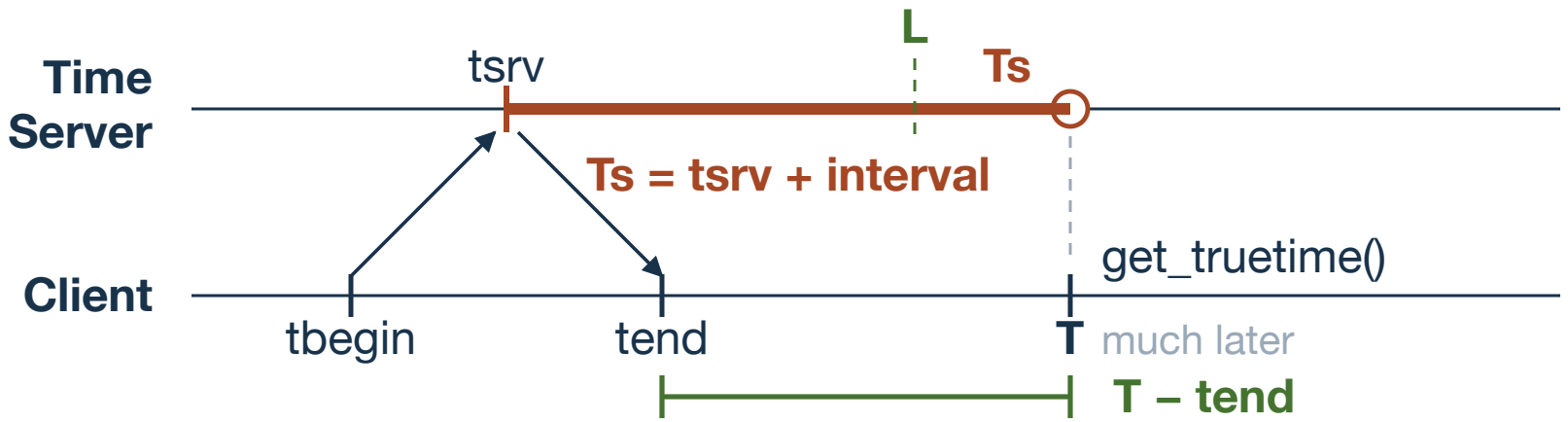
- The only measurements: tbegin, tsrv, tend from a background sync RPC
- For now, assume client and server tick at the same speed



TrueTime: how to calculate the bound?

Problem statement: the client's clock reads T — what is the time server's Ts?

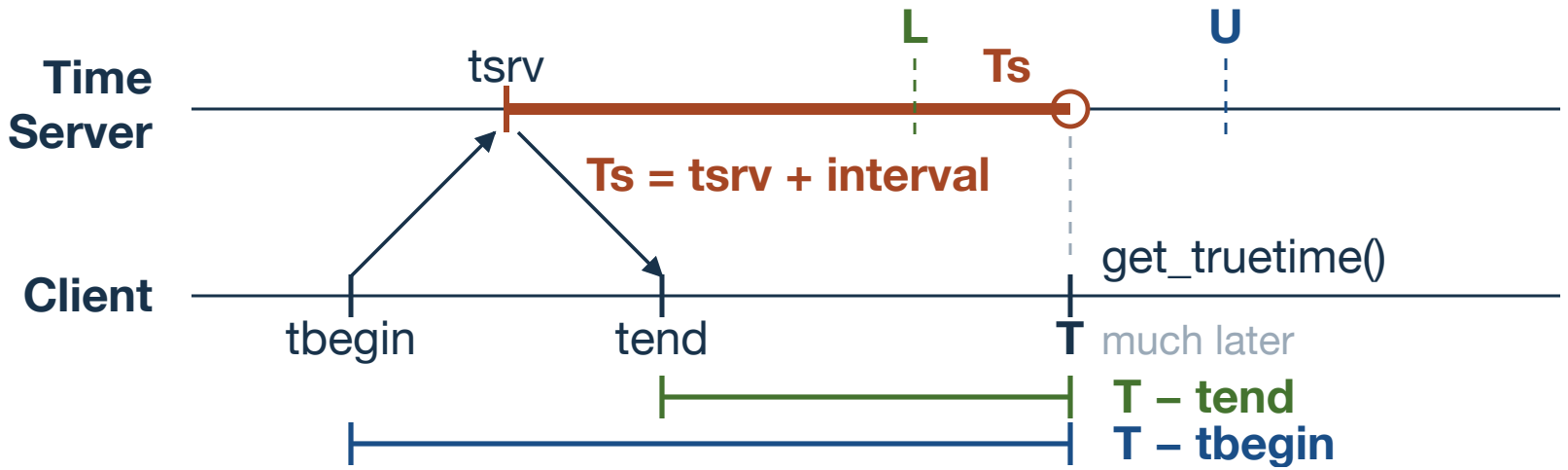
- The only measurements: tbegin, tsrv, tend from a background sync RPC
- For now, assume client and server tick at the same speed



TrueTime: how to calculate the bound?

Problem statement: the client's clock reads T — what is the time server's Ts?

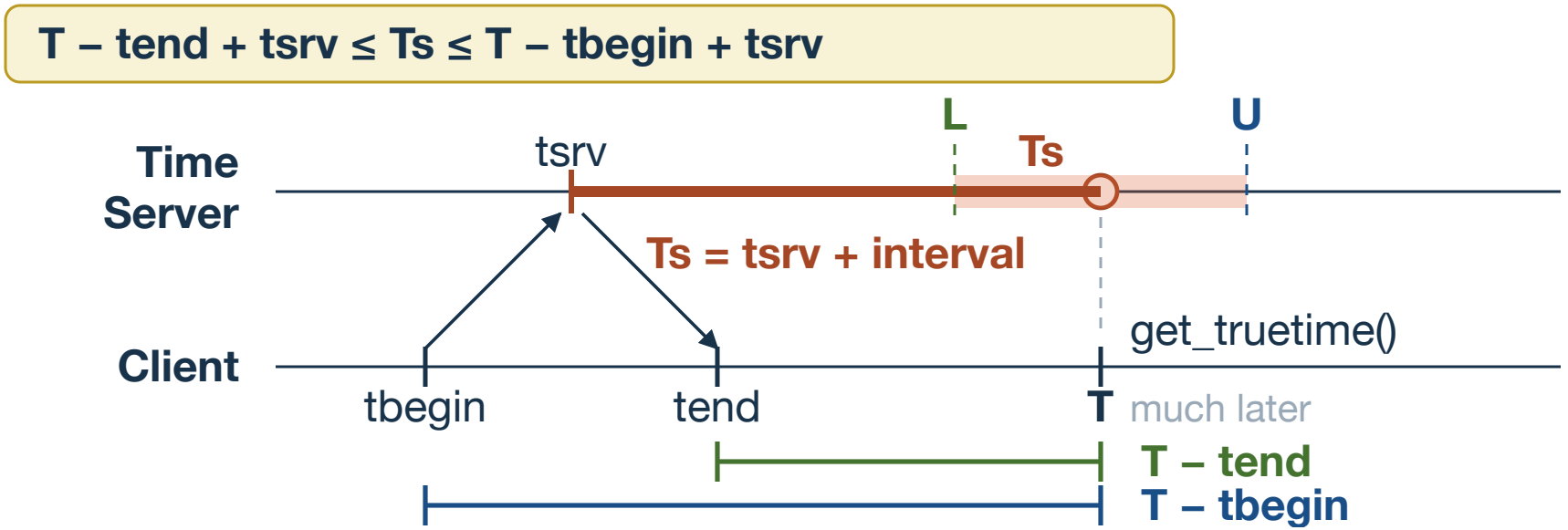
- The only measurements: tbegin, tsrv, tend from a background sync RPC
- For now, assume client and server tick at the same speed



TrueTime: how to calculate the bound?

Problem statement: the client's clock reads T — what is the time server's Ts?

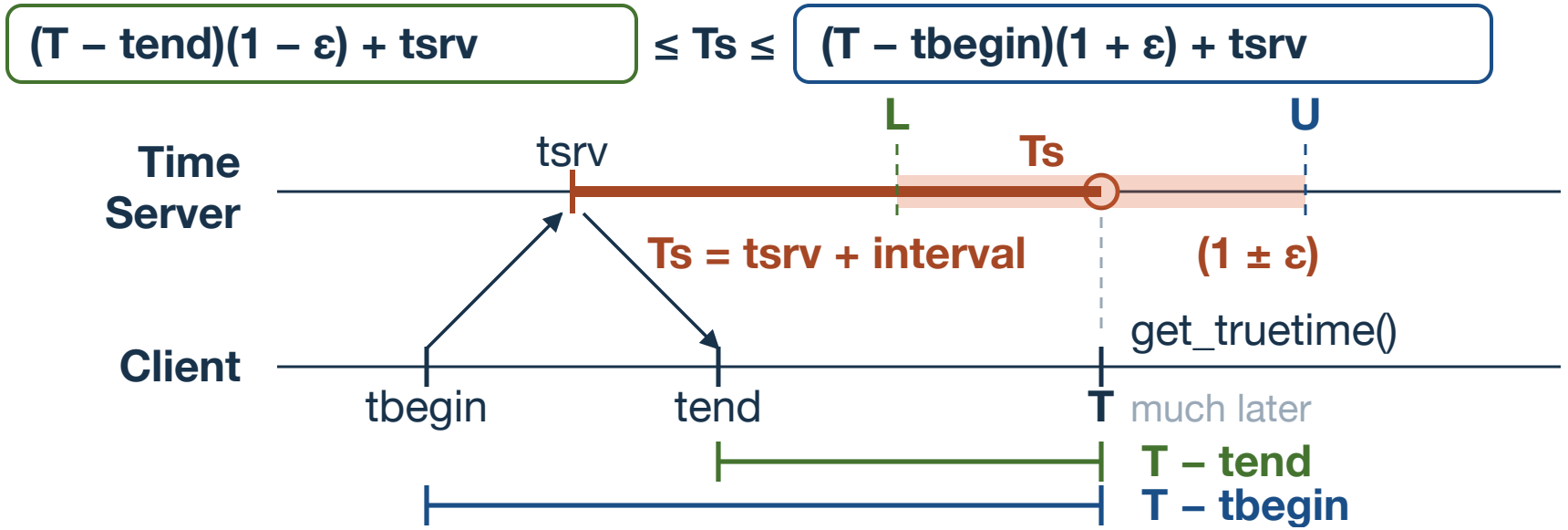
- The only measurements: tbegin, tsrv, tend from a background sync RPC
- For now, assume client and server tick at the same speed



TrueTime: how to calculate the bound?

Problem statement: the client's clock reads T — what is the time server's Ts?

- The only measurements: tbegin, tsrv, tend from a background sync RPC
- For now, assume client and server tick at the same speed

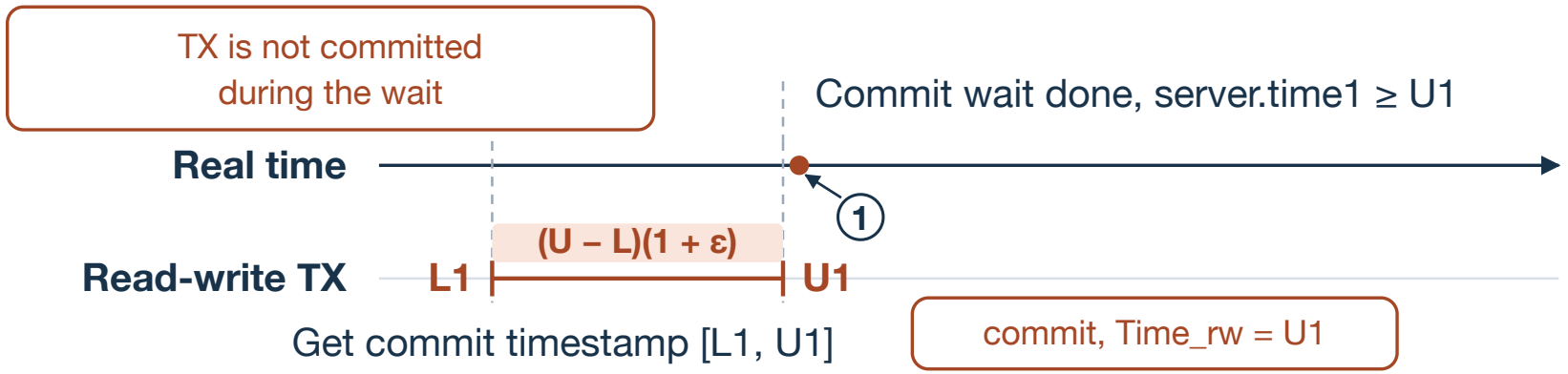


Widened by the drift rate ε, the two ends are exactly L and U

Commit wait revisited

For external consistency: If a rw TX has decided to commit, then $\mathrm{Time}_{ro} \geq \mathrm{Time}_{rw}$

- Commit wait for read-write TX: after acquiring the commit timestamp, the coordinator waits for $(U - L)(1 + \varepsilon)$ and uses U as Time_{rw}

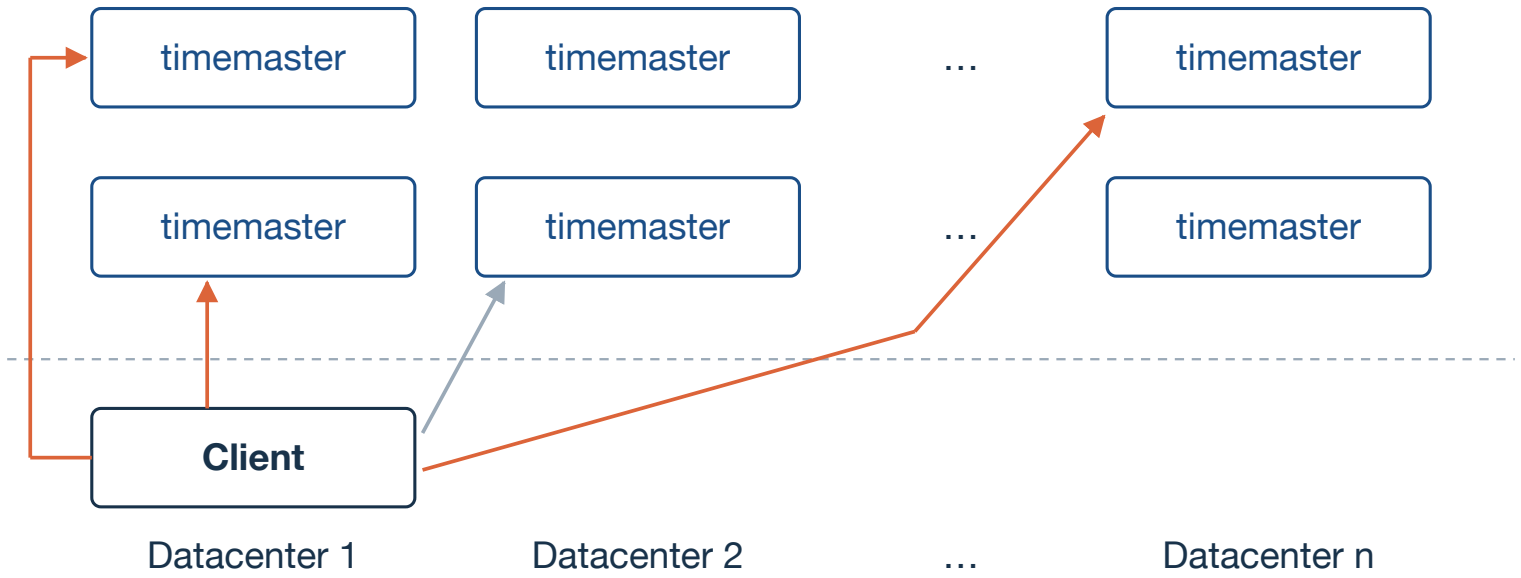


Every read-write TX pays one bound width of waiting

TrueTime adopts multiple time servers

Every datacenter deploys its own time servers — called timemasters in Spanner

- A client syncs with several timemasters: its own DC's, and other DCs'

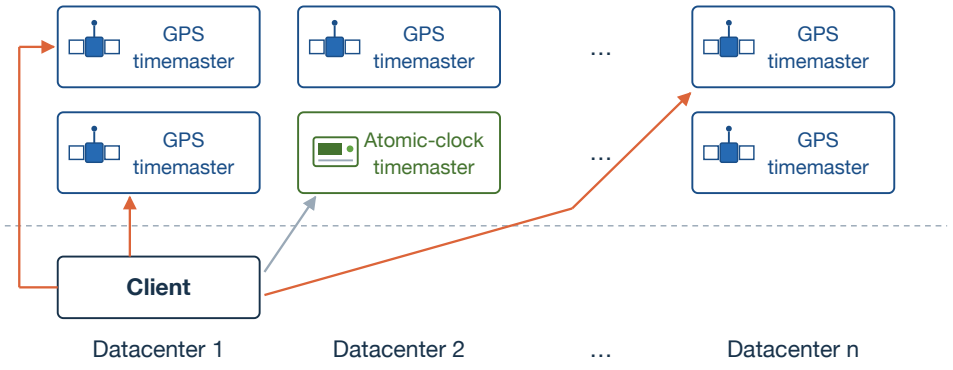


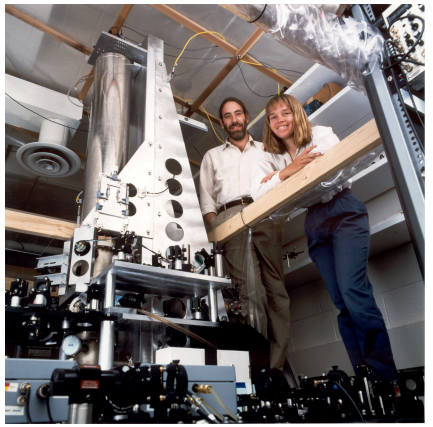
Question: how to keep different timemasters in sync?

Timemasters are backed by GPS & atomic clocks

High-precision clocks, with two independent sources

- A cesium clock neither gains nor loses a second in nearly 20,000,000 years
- vs. the 200 µs per second of drift Spanner assumes for an ordinary machine
- Atomic clocks are synchronized with each other [1, 2]





NIST-F1 (cesium)

3, 4

FS725 ... from $3995



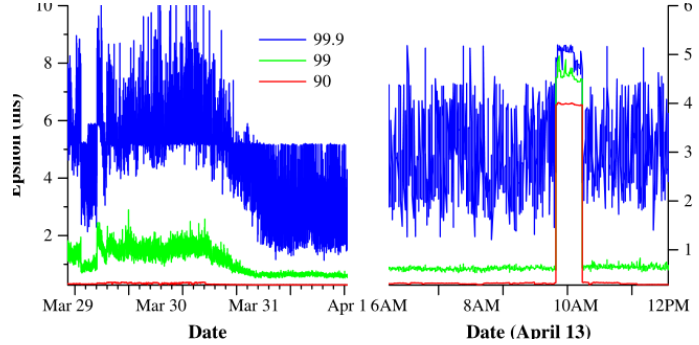
FS725 (rubidium)

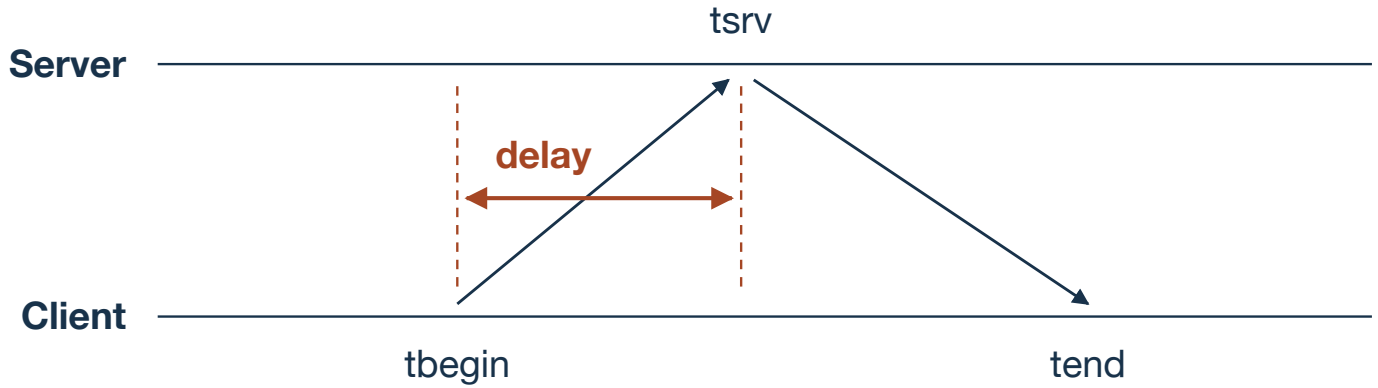
[1] NIST-F1 cesium fountain clock: "neither gain nor lose a second in nearly 20 million years"
[2] The 200 µs/second drift rate: Corbett et al., *Spanner*, OSDI 2012, §3
[3] Photo: NIST; an example of an atomic clock, not Spanner's deployed hardware
[4] Sales screenshot: SRS FS725, 2026-10-27; US$3,995 starting price is for the FS725, not NIST-F1

Final takeaway: network-induced uncertainty

Interval of Ts: [(T − tend)(1 − ε) + tsrv, (T − tbegin)(1 + ε) + tsrv]

- tsrv − tbegin is roughly estimated as the network delay
- Can have spikes if timemasters are out of service
- The spike here: 2 timemasters shut down for maintenance[1]





[1] ε measured at Google over a week; the April 13 spike "resulted from the shutdown of 2 time masters at a datacenter for routine maintenance". Source: Corbett et al., *Spanner*, OSDI 2012, Figure 6

Summary of Spanner

Builds on basic 2PL, 2PC and Paxos

But with tremendous innovations to make them work at scale!

Reify clock uncertainty in time APIs

- Known unknowns are better than unknown unknowns
- Rethink algorithms (TX's concurrency control) to make use of the uncertainty

Stronger semantics are achievable at scale

- Spanner keeps external consistency across shards, replicas, and datacenters

Greater scale ≠ weaker semantics

参考文献

[1] Spanner: Google's Globally-Distributed Database — Corbett et al., 2012. OSDI (P13)

[2] Maintaining the Time in a Distributed System — Marzullo & Owicki, 1983. PODC (P36)