

# Raft & Spanner

IPADS, Shanghai Jiao Tong University

<https://www.sjtu.edu.cn>

## Review: Replicated State Machines

A general approach to making consistent replicas of a server:

- Start with the **same initial state** on each server
- Provide each replica with the **same input** operations, in **same order**
- Ensure all operations are **deterministic**
  - E.g., no randomness, no reading of current time, etc.

These rules ensure each server will end up in the **same final state**

## Review: Single-decree Paxos vs. Multi-Paxos

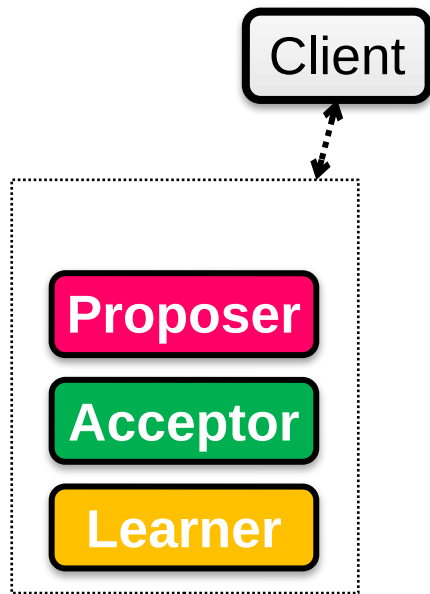
(Single-decree) Paxos allows us to ensure **a consistent value** over acceptors

- Value can be anything, e.g., an integer or a command

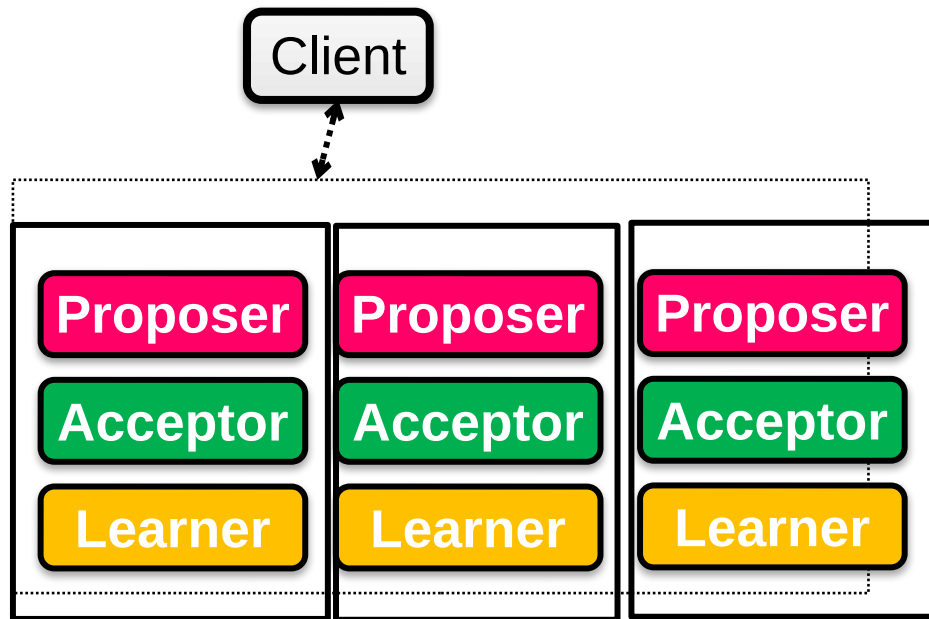
**Multi-paxos adopts multiple single-decree paxos instances to realize logs in replicated state machine**

# Review: Single-decree Paxos vs. Multi-Paxos

	0	1	2
Log	add	cmp	xxx



Single-decree Paxos



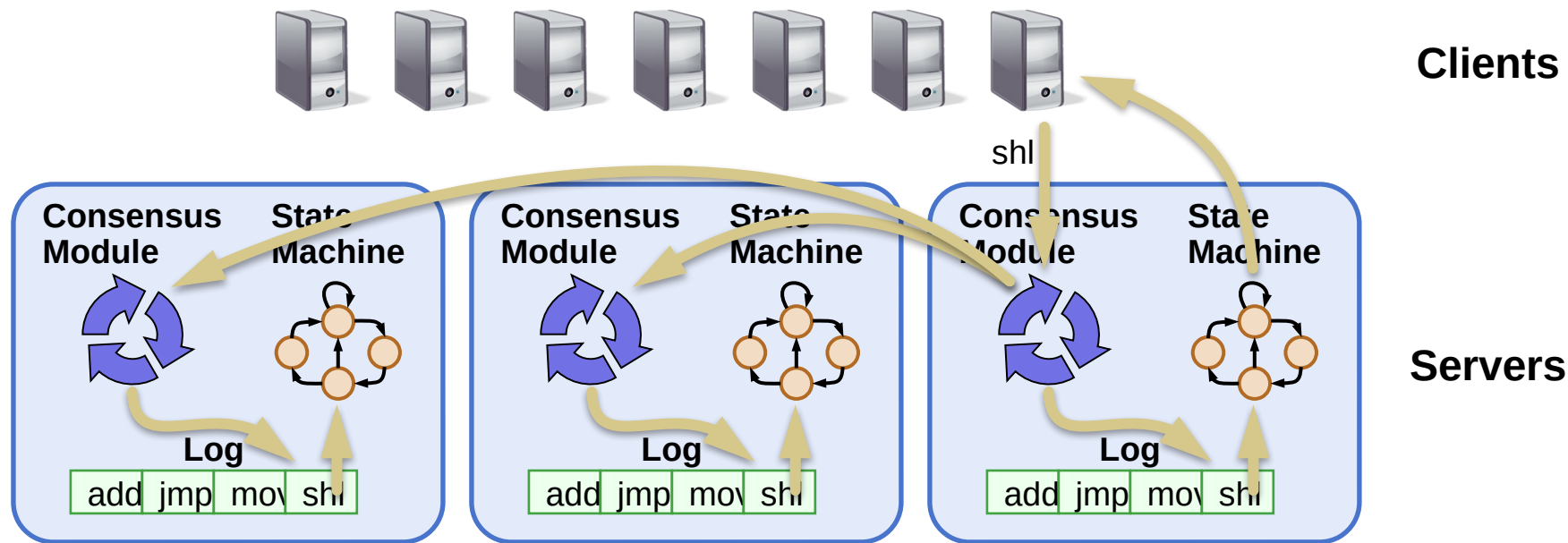
Single-decree  
instance for 0

Single-decree  
instance for 1

Single-decree  
instance for 2

Multi-Paxos

## Review: raft's replicated state machine based on log



Replicated log => **replicated state machine**

- All servers execute same (deterministic) commands in same order

Consensus module ensures **proper** logs are the same!

# Raft's high-level approach: problem decomposition

## 1. Leader election

Select one server as the leader

Detect crashes, choose new leader

## 2. Log replication (normal operation)

Leader accepts commands from clients, append to its log

Leader replicates its log to other servers (overwrites inconsistencies)

## 3. Safety

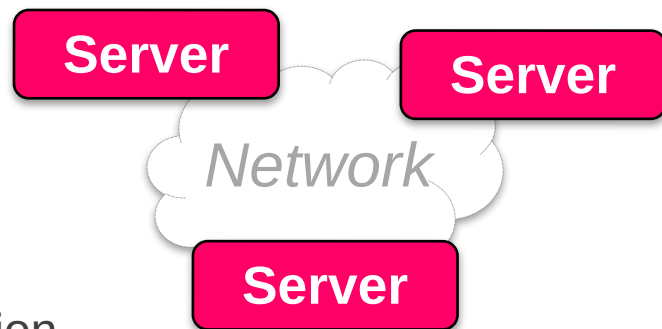
Keep logs consistent

Only servers with up-to-date logs can become the leader

## Review: Raft server states

At any time, each server is either:

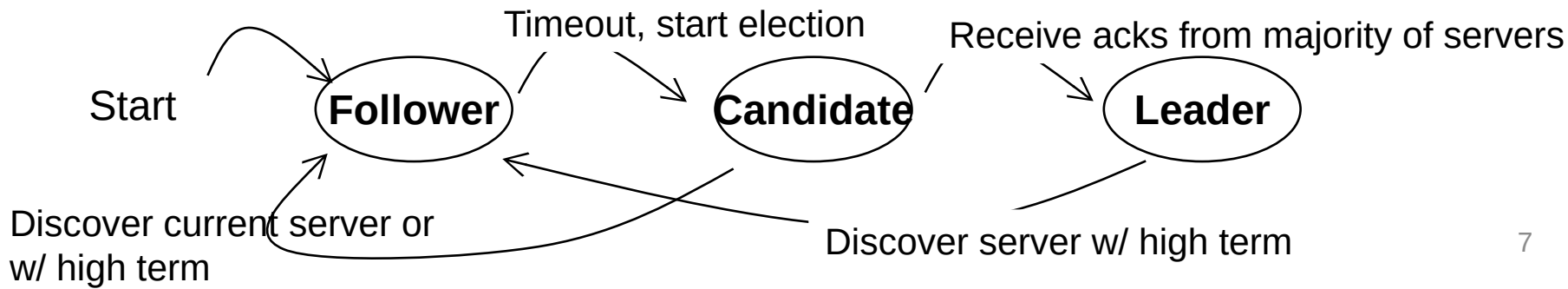
- **Leader**: handles all client interactions, log replication
  - Invariant: At most 1 viable leader at a time
- **Follower**: passive (only responds to incoming RPCs)
- **Candidate**: used to elect a new leader



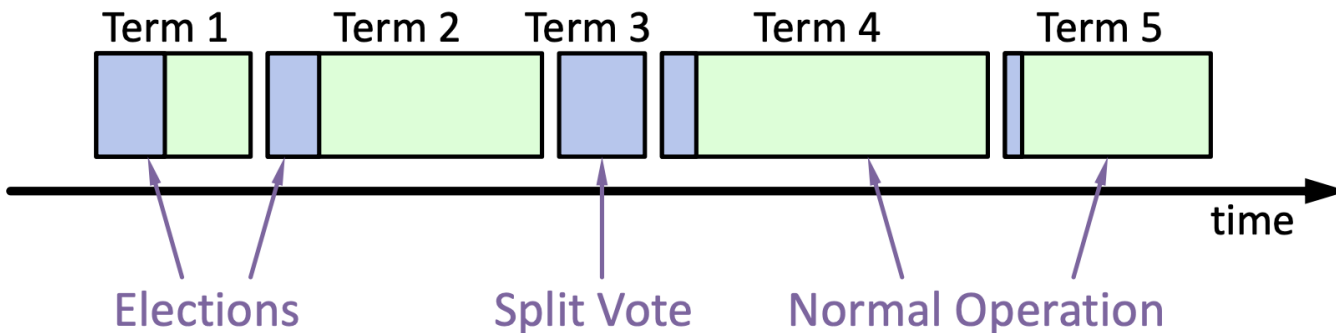
Servers communicates w/ RPCs

### Normal workloads

- 1 server is the leader, others are the followers



## Review of Raft basics: terms for one leader



**Raft divides time into terms (with arbitrary length):**

- Each term starts with an election
- Ends with one leader or no leader
- At most one leader per term

**Each leader is uniquely associated with a term**

- Key role: identify obsolete information



## Review: Basic request vote RPC so far

Invoked by candidates to gather votes.

### Arguments:

**candidateId**      candidate requesting vote

**term**      candidate's term

### Results:

**term**      currentTerm, for candidate to update itself

**voteGranted**      true means candidate received vote

### Implementation:

- 1.If  $\text{term} > \text{currentTerm}$ ,  $\text{currentTerm} \leftarrow \text{term}$   
(step down if leader or candidate)
- 2.If  $\text{term} == \text{currentTerm}$ ,  $\text{votedFor}$  is null or  $\text{candidateId}$ , grant vote and reset election timeout

# Raft's high-level approach: problem decomposition

## 1. Leader election

Select one server as the leader

Detect crashes, choose new leader

## 2. Log replication (normal operation)

Leader accepts commands from clients, append to its log

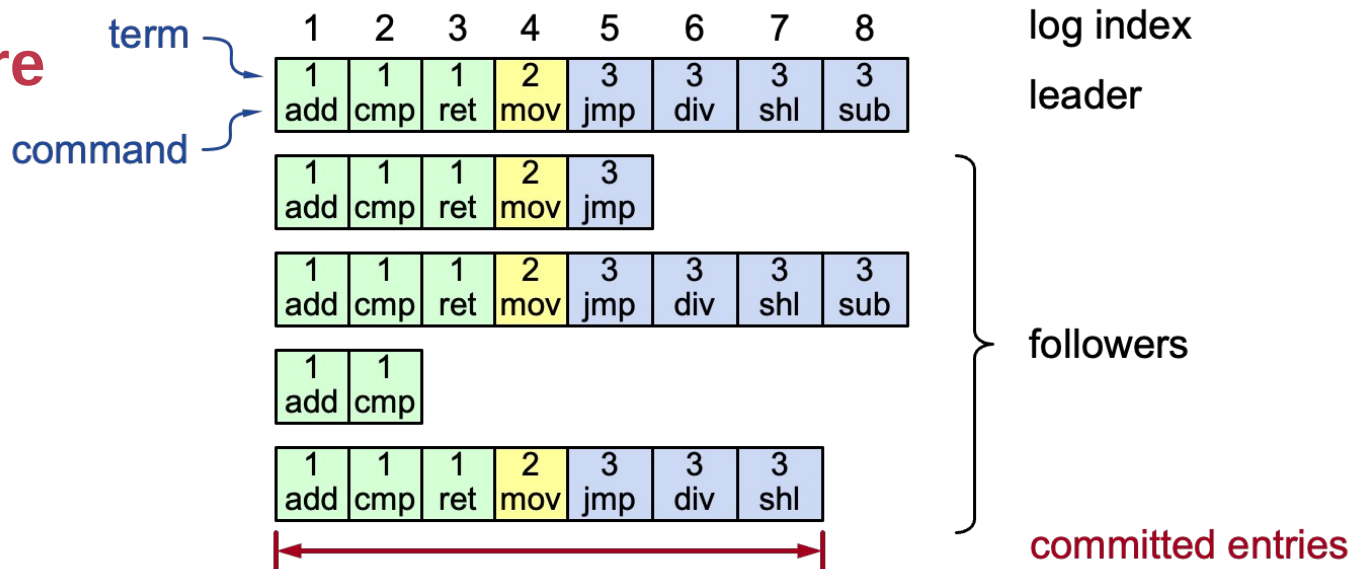
Leader replicates its log to other servers (overwrites inconsistencies)

## 3. Safety

Keep logs consistent

Only servers with up-to-date logs can become the leader

# Log structure



**Log entry = index, term, command**

- Stored on the disk to tolerate failures

**A log is committed if it can be safely applied to the state machine**

- i.e., eventually stored on all the servers with the same value

**Not all** entries are **committed** (will talk about later)

## Persistent state of each server + log

### **currentTerm**

Latest term server has seen (initialized to 0 on first boot)

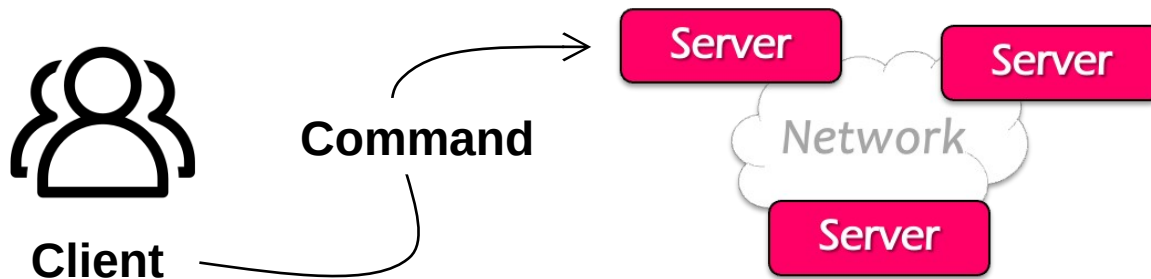
### **votedFor**

Candidate Id that received vote in current term (or null if none)

### **Log[]**

Log entries

## Normal operations to append the log

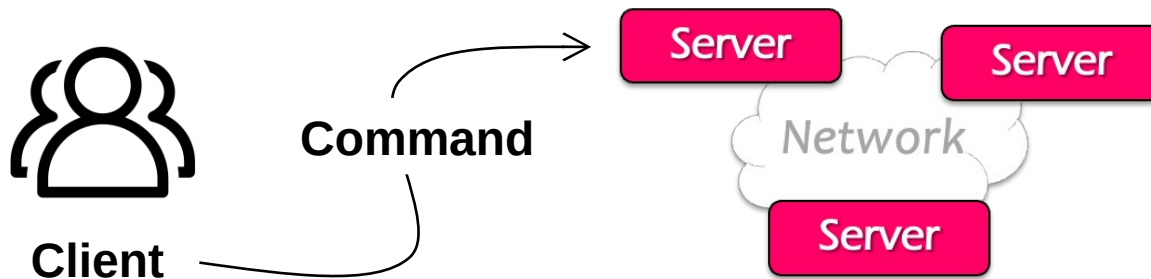


- ① Send command to the leader
- ② Leader appends command to its log
- ③ Leader sends **AppendEntries** RPCs to followers
- ④ Once a new entry (of log) **committed**:

Leader passes command to its state machine, returns results to client

Notifies followers of committed entries, Follower pass committed commands to their state machines (How to determine an entry is committed will be talked later)

## Normal operations to update the log



### Crashed/slow followers?

- Leader retries RPCs until they succeed (at least once)

### Performance is optimal in the common case

- One successful RPC to any majority of servers

# Challenge: crash can cause log to inconsistencies

Case: an old leader is unaware of the new leader due to network partition

- Yet, it can still get commands from the clients

**Leader**

S1	1 add	1 cmp
----	----------	----------

S2	1 add	1 cmp
----	----------	----------

S3	1 add	1 cmp
----	----------	----------

**Leader**

S1	1 add	1 cmp
----	----------	----------

S2	1 add	1 cmp
----	----------	----------

S3	1 add	1 cmp
----	----------	----------

**Leader (stale)**

S1	1 add	1 cmp	1 xxx
----	----------	----------	----------

S2	1 add	1 cmp	2 shi
----	----------	----------	----------

S3	1 add	1 cmp	2 shi	2 ret
----	----------	----------	----------	----------

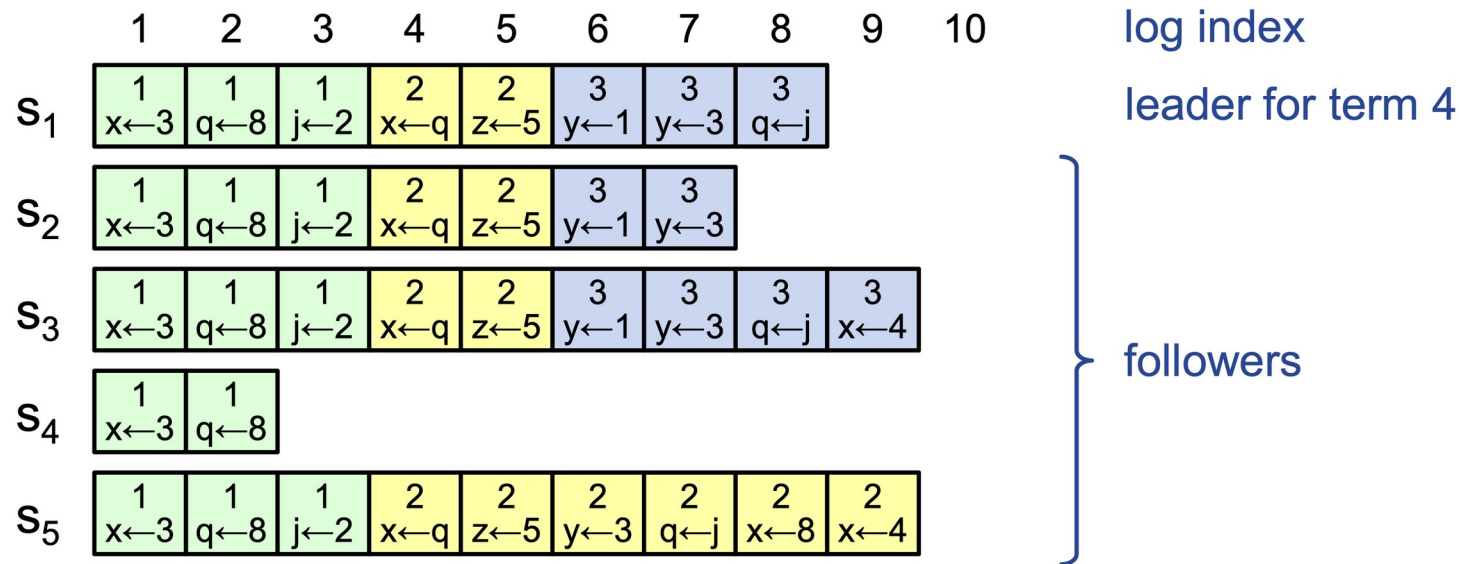
**Leader (term2)**

Term 1



Term 2

## Challenge: crash can cause log to inconsistencies



### Raft minimizes special code for repairing inconsistencies

- Leaders assume its log is correct
- Normal operation will repair all inconsistencies



# Consistency of the log

Question: how to achieve this property?

High level of **coherency** between logs **maintained by the raft**:

- If log entries on different servers have the same index & term
  - They store the same command
  - The logs are identical in all preceding entries

1	2	3	4	5	6
1 add	1 cmp	1 ret	2 mov	3 jmp	3 div
1 add	1 cmp	1 ret	2 mov	3 jmp	4 sub

If a given entry is committed, all preceding entries are also committed

- Note that not all log entries are committed

# AppendEntries consistency checks

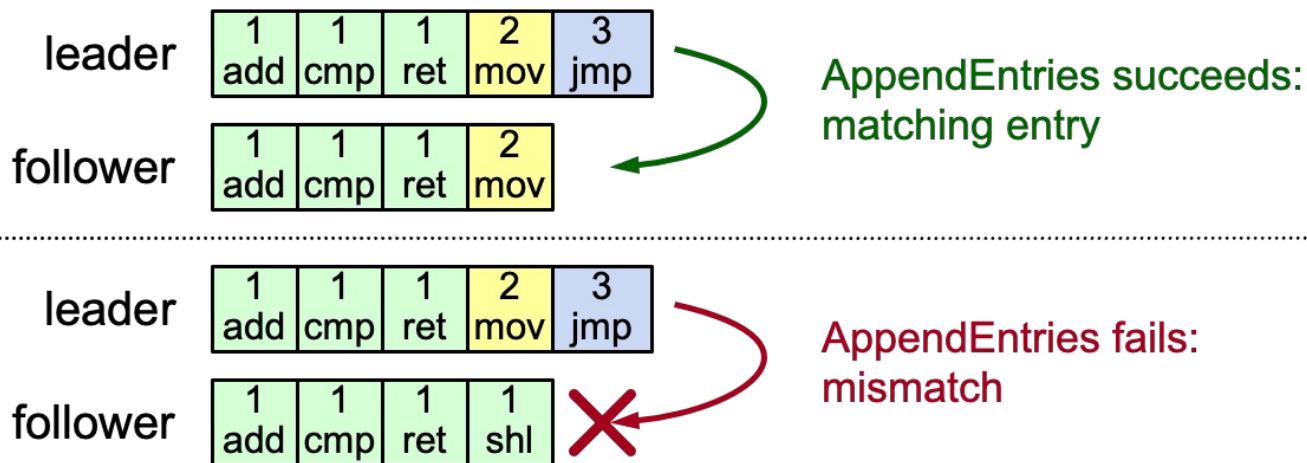
Implements an induction step, ensures coherency

Each RPC argument contains

- Append index, term, **term of entry preceding new ones**

Follower checks whether it has the **matching** entry

- Otherwise, it rejects the request



# **Inconsistent entries causes: leader changes or missing entries**

**At beginning of new leader's term:**

- Old leader may have left entries partially replicated

**Missing entries: a replica may be fall behind**

**No special steps by new leader: just start normal operation, w/ different position**

- Leader's log is "the truth"
- Will eventually make follower's logs identical to leader's (overwrite the divergent log entries with the leader's ones)

**The rejected/missing entries will be overwritten by the leader's log entry**

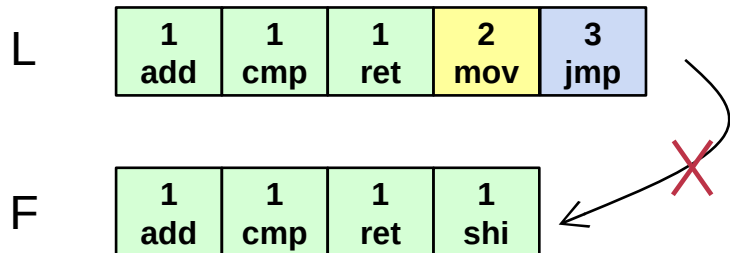
# Handle rejections

Implements an induction step, ensures coherency

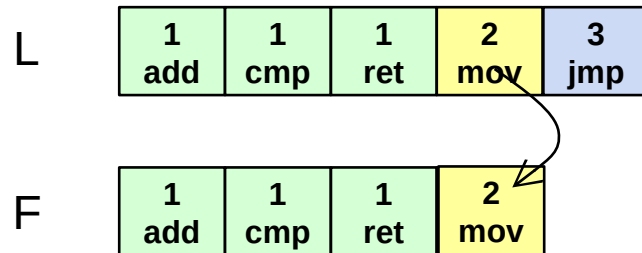
In this example, the leader will

- Overwrite index 4 with [2 mov] with AppendEntries
- Then append entry [3 jmp] to the end of the follower

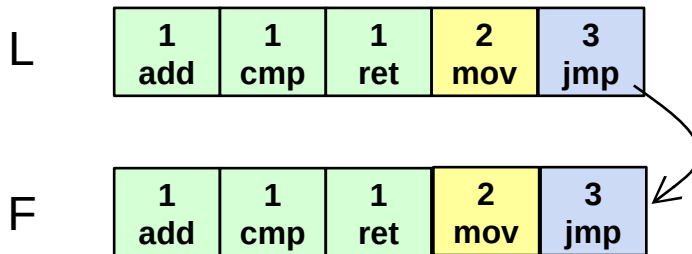
Fail to append case



Overwrite mismatched entry



Append the new entry



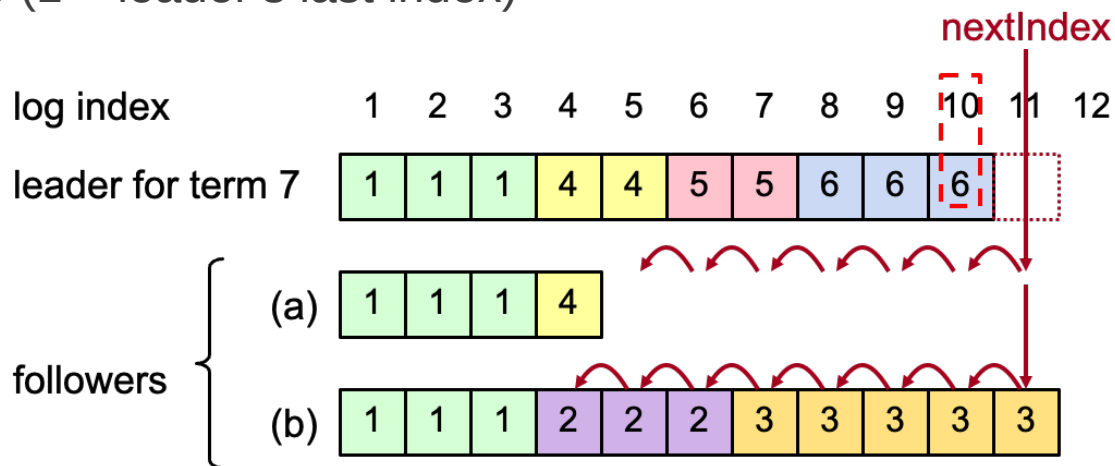
## More on Repairing Follower Logs

**New leader must make follower logs consistent with its own**

- Delete extraneous entries
- Fill in missing entries

**Leader keeps nextIndex for each follower:**

- Index of next log entry to send to that follower
- Initialized to (1 + leader's last index)



## AppendEntries RPC (Simplified)

Invoked by leader to replicate log entries and discover inconsistencies

### Arguments:

**term** leader's term  
**prevLogIndex** index of log entry immediately preceding new ones  
**prevLogTerm** term of prevLogIndex entry  
**entries[]** log entries to store (empty for heartbeat)

### Results:

**success** true if follower contained entry matching prevLogIndex and prevLogTerm

## AppendEntries RPC (Simplified)

### Implementation:

1. Return false if  $\text{term} < \text{currentTerm}$
2. If  $\text{term} > \text{currentTerm}$ ,  $\text{currentTerm} \leftarrow \text{term}$
3. If candidate or leader, step down
4. Reset election timeout
5. Return failure if log doesn't contain an entry at  $\text{prevLogIndex}$  whose term matches  $\text{prevLogTerm}$
6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry
7. Append any new entries not already in the log

**Overwrite makes a subtle issue:  
When can we commit a log entry? (Since an  
appended log entry may be overwritten)**

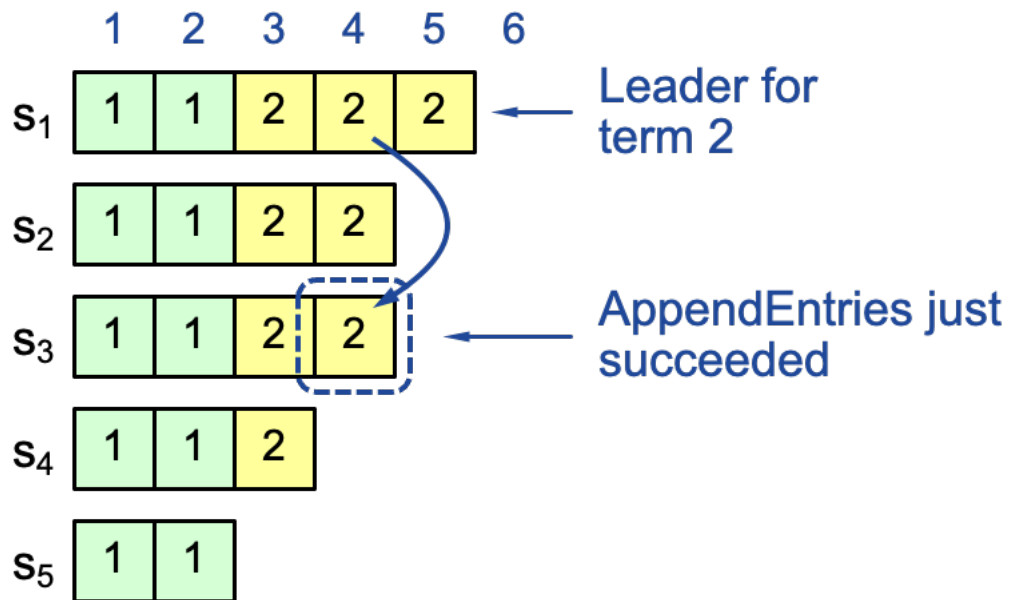


## Intuitive goal: replicating on a majority

If the log **is replicated on a majority of servers**, it can be replicated on the state machine

- Not always true on raft so far

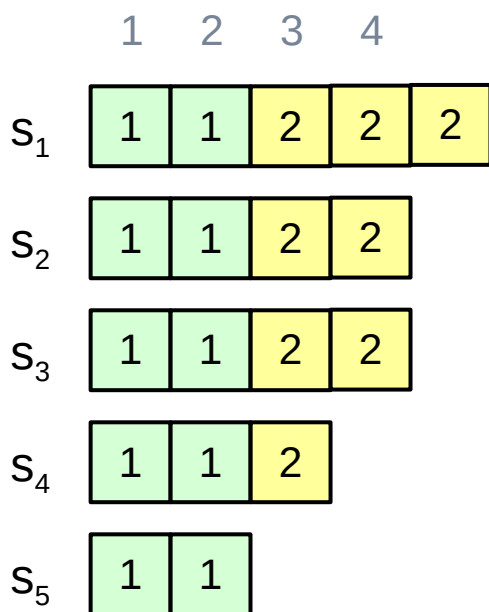
Question: can index 4 be overwritten after appending to S3?



## Case study: raft overwriting

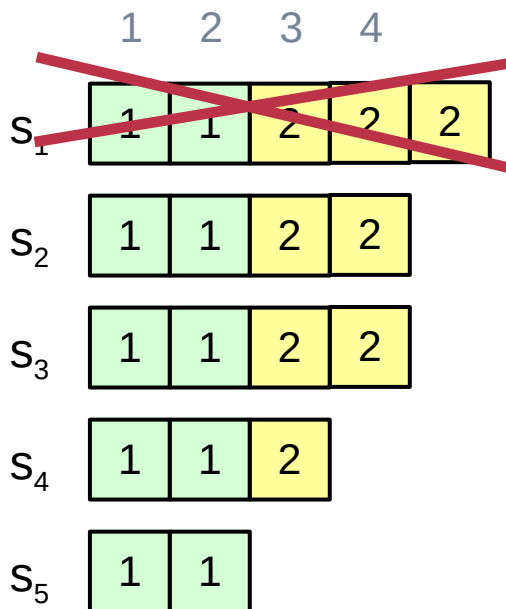
Question: can index 4 be overwritten?

Leader S1



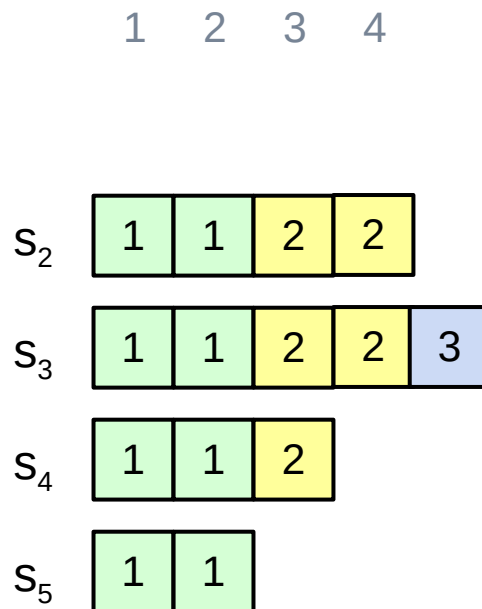
Term 2

Leader ?



S1 crashes

Leader S2

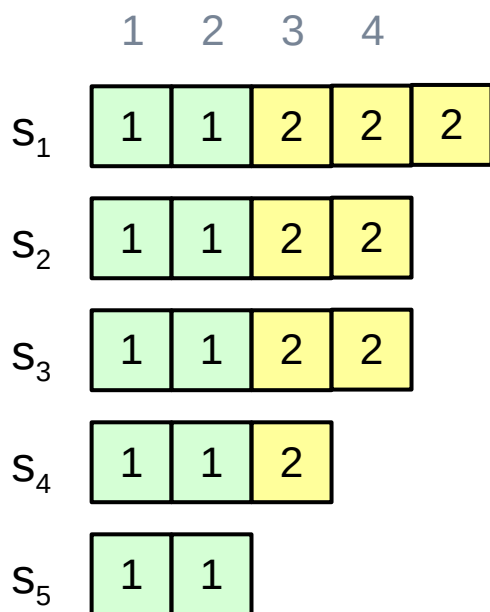


Term 3

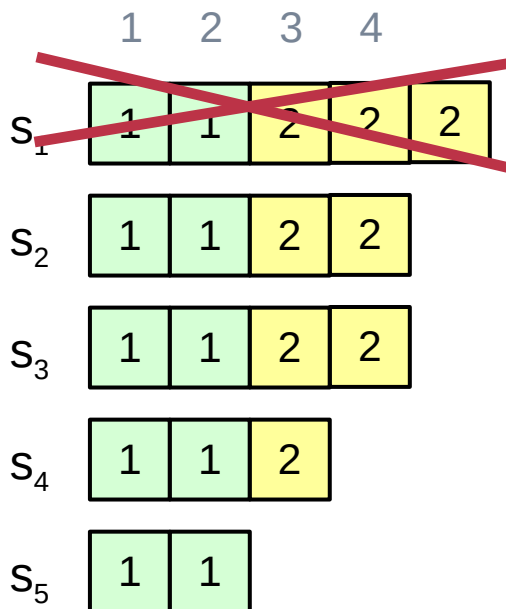
## Case study: raft overwriting

Question: can index 4 be overwritten?

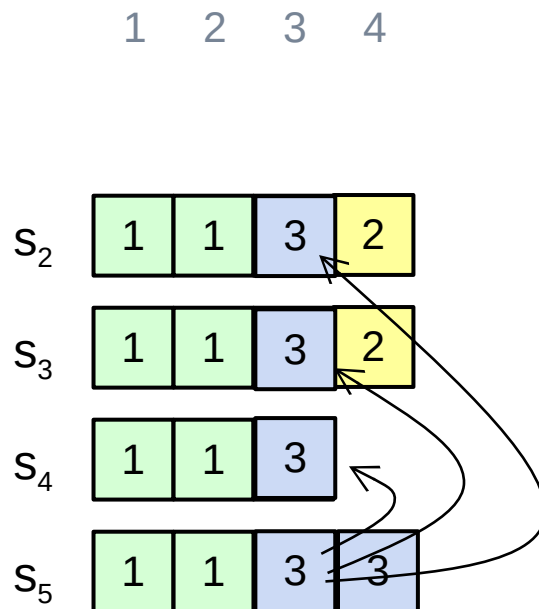
Leader S1



Leader ?



Leader **S5**



Term 2

S1 crashes

Term 3

## Safety requirement of the commit entry

Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry

### Raft safety property:

- If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders (no overwritten)

### This guarantees the safety requirement

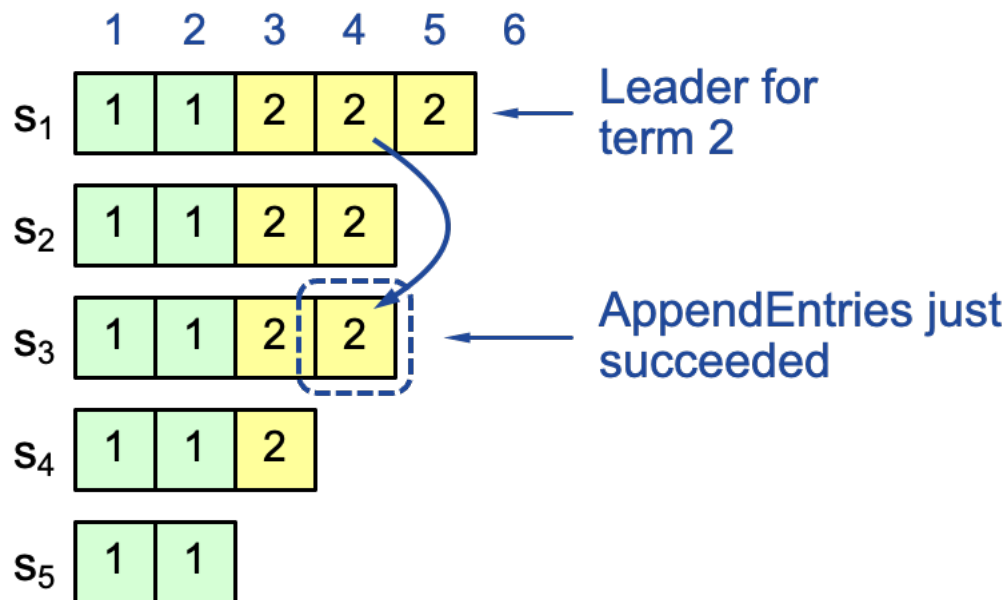
- Leaders never overwrite entries in their logs
- Only entries in the leader's log can be committed
- Entries must be committed before applying to state machine



**Goal: if a log entry has been replicated to majority followers, it is likely to commit**

# Committing Entry from the Current Term

Case #1/2: Leader decides entry in current term is committed



Question: how to prevent index 4 from being overwritten? Prevent S4 or S5 from becoming the leader

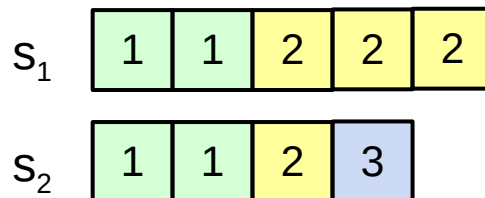
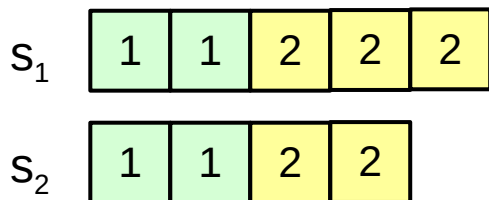
## Picking the Best Leader

**During elections, choose candidate with log most likely to contain all committed entries**

- Candidates include log info in RequestVote RPCs (index & term of last log entry)
- Voting server V denies vote if its log is “more complete”:  
`(lastTermV > lastTermC) ||`  
`(lastTermV == lastTermC) && (lastIndexV > lastIndexC)`
- Leader will have “most complete” log among electing majority

## Comparing the best leader

```
(lastTermV > lastTermC) ||  
(lastTermV == lastTermC) && (lastIndexV > lastIndexC)
```



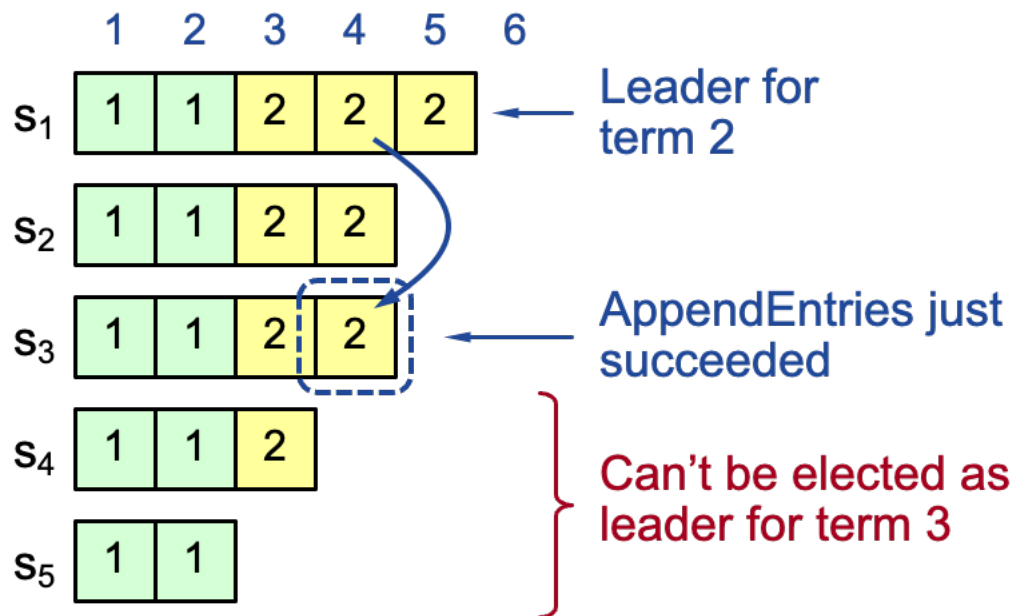
Question: which log is longer?

- $S_1$  &  $S_2$ , respectively



## Committing Entry from the Current Term

Leader decides entry in current term is committed



Safe: leader for term 3 must contain entry 4

# Complete picture of request vote RPC

Invoked by candidates to gather votes.

## Arguments:

**candidateId**      candidate requesting vote

**term**      candidate's term

**lastLogIndex**      index of candidate's last log entry

**lastLogTerm**      term of candidate's last log entry

## Results:

**term**      currentTerm, for candidate to update itself

**voteGranted**      true means candidate received vote

## Implementation:

- 1.If  $\text{term} > \text{currentTerm}$ ,  $\text{currentTerm} \leftarrow \text{term}$   
(step down if leader or candidate)
- 2.If  $\text{term} == \text{currentTerm}$ ,  $\text{votedFor}$  is null or  $\text{candidateId}$ , and **candidate's log is at least as complete as local log**, grant vote and reset election timeout

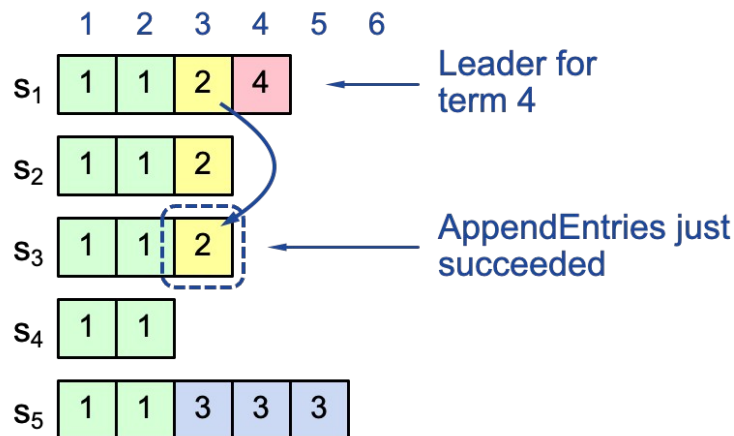
## Commit rule for raft so far

If the log entry from the leader term is replicated to a majority of followers

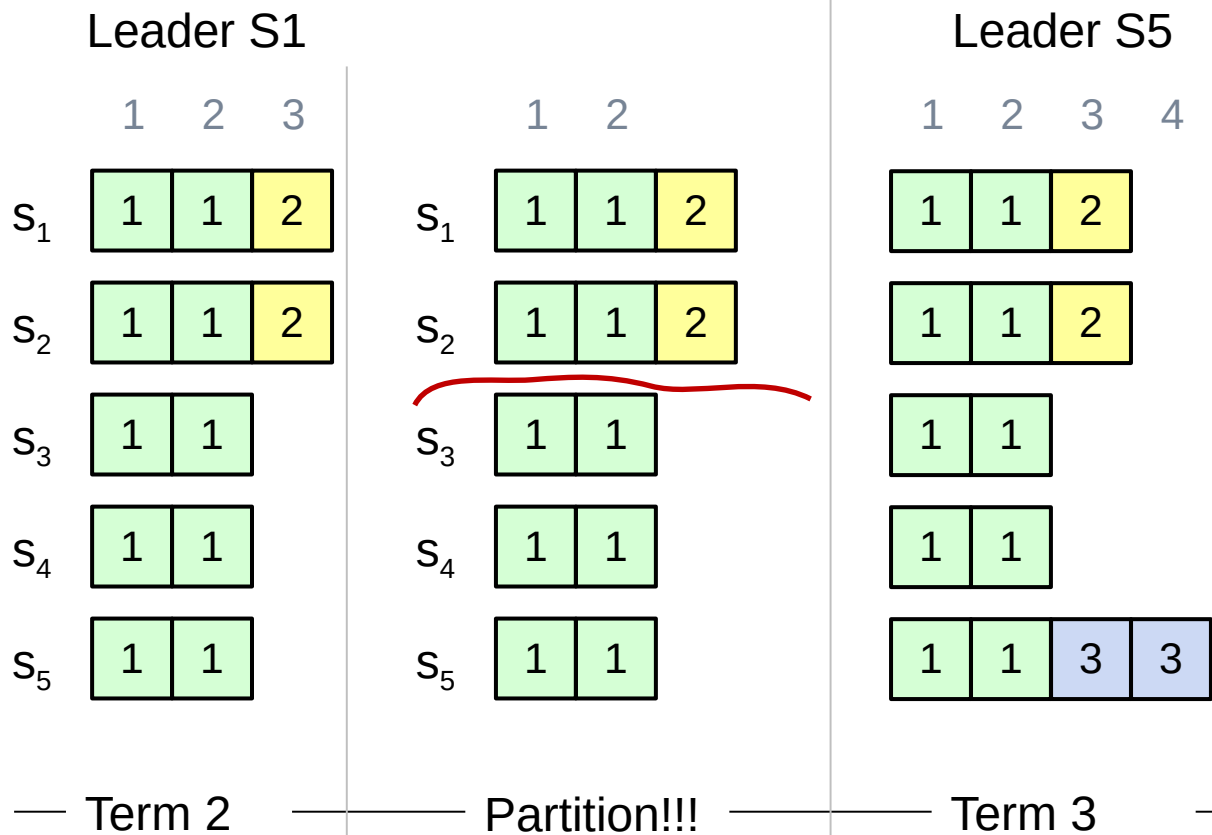
- Then we can treat it as committed
  - The later leader must contain the entry

But, what about replicating log entry from a previous term?

- The previous term's entry may fail to reach a majority

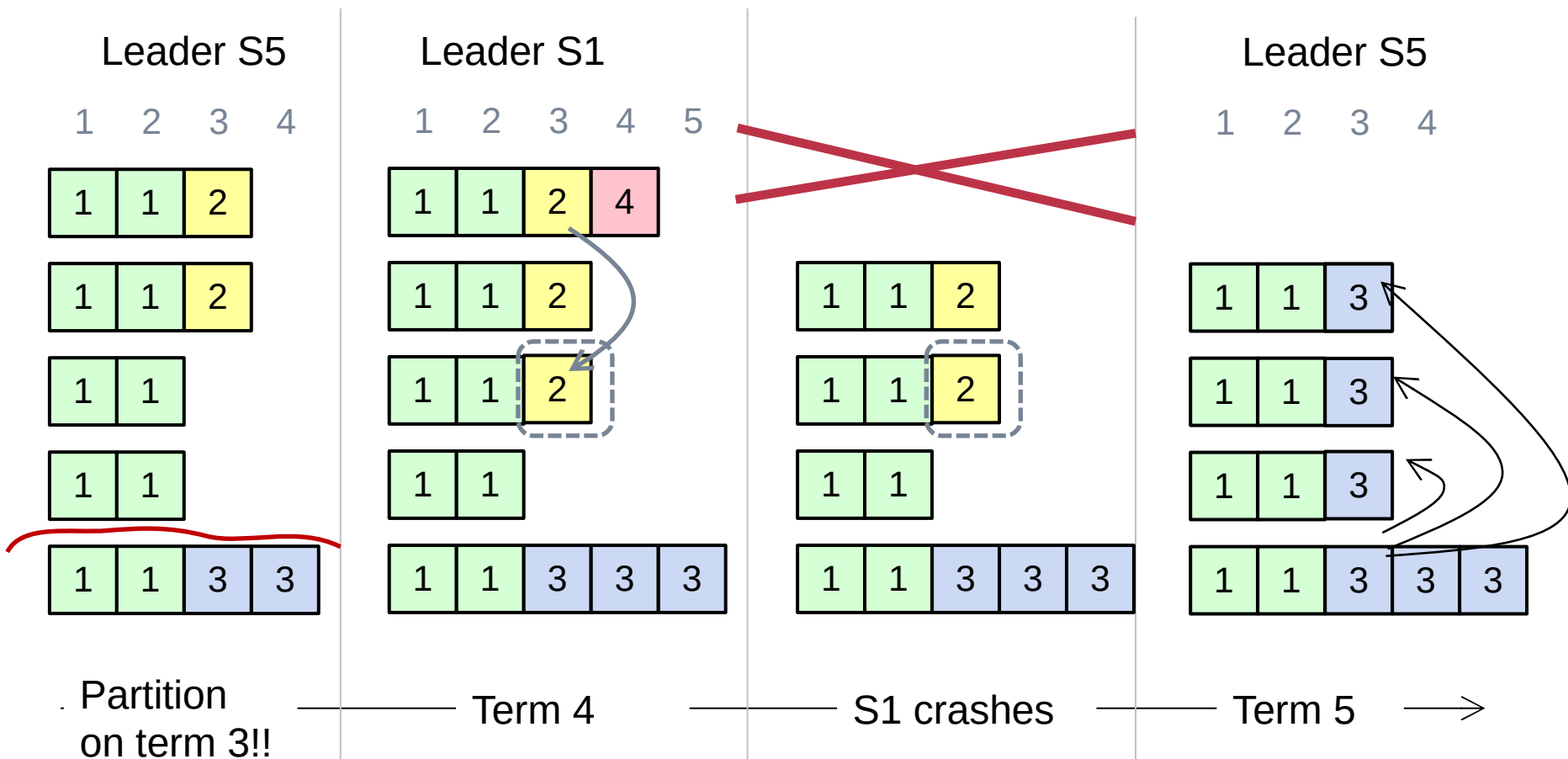


## Case study: a majority replicated entry can be overwritten



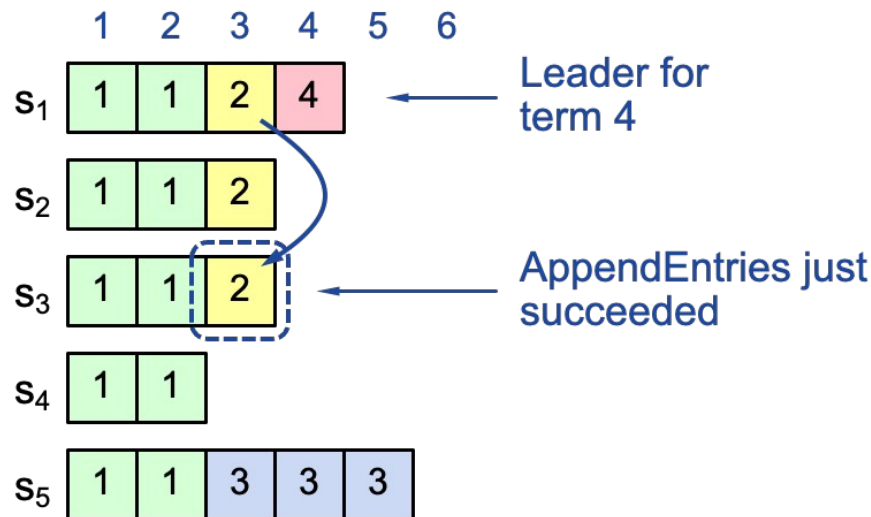
Question: who can become the leader for term 3?

## Case study: a majority replicated entry can be overwritten



## Committing Entry from Earlier Term

Case #2/2: Leader is trying to finish committing entry from an earlier term



**Entry 3 not safely committed:**

- S5 can still be elected as leader for term 5
- If elected, it will overwrite entry 3 on S1, S2 and S3 (recall our previous example)

# Commit rules

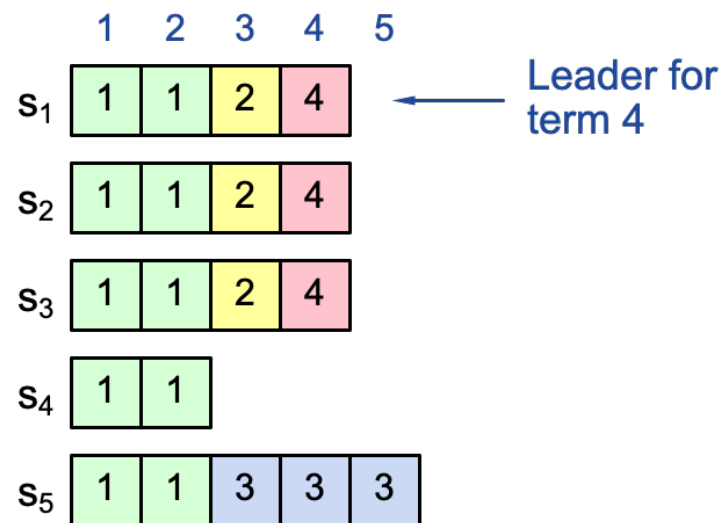
**For a leader to decide an (previous) entry is committed:**

- Must be stored on a majority of servers
- At least one new entry from leader's term must also be stored on majority of servers

**This is because once entry 4 committed:**

- $s_5$  cannot be elected leader for term 5
- Entries 3 and 4 both safe

Combination of election rules & commitment rules makes Raft safe



Detailed distributed database study:  
**Google Spanner**



# Problems faced by large-scale company

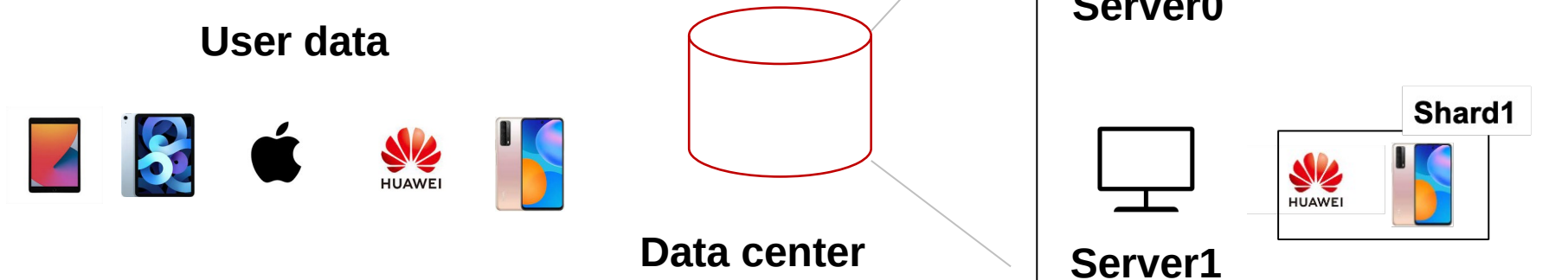
## Large-scale dataset

- **Facebook** has more than **1 billion** of images uploaded **weekly**, **Baidu** stores **tens of billions** of web pages

**Question: how to store such a large dataset?**

## Solution: sharding

- Each shard is stored in an LSM storage



# How to shard?

## A topic orthogonal to the design of Spanner

### Possible way:

- Hashing, node id =  $\text{hash}(\text{key}) \% \text{num\_of\_nodes}$
- Range partitioning
  - E.g., coordinators record that 0-100 keys are at shard 0, etc.

### A good partition need to consider many factors, e.g.,

- Reduce cross shard transaction (no 2PC for multi-site transactions)
  - Why? Shards are likely to span across two machines, which needs two-phase-commit for execution

We will not focus sharding in this lecture

# Problems faced by large-scale company

## Fault tolerance

- Large-scale companies are built over **large-scale datacenters** (>10K servers)
- Machine failures are particular common in large-scale datacenters

“Suppose a cluster has ultra-reliable server nodes with a stellar mean time between failures (MTBF) of 30 years (10,000 days)—a cluster of 10,000 servers will see an average of one server failure per day. ”

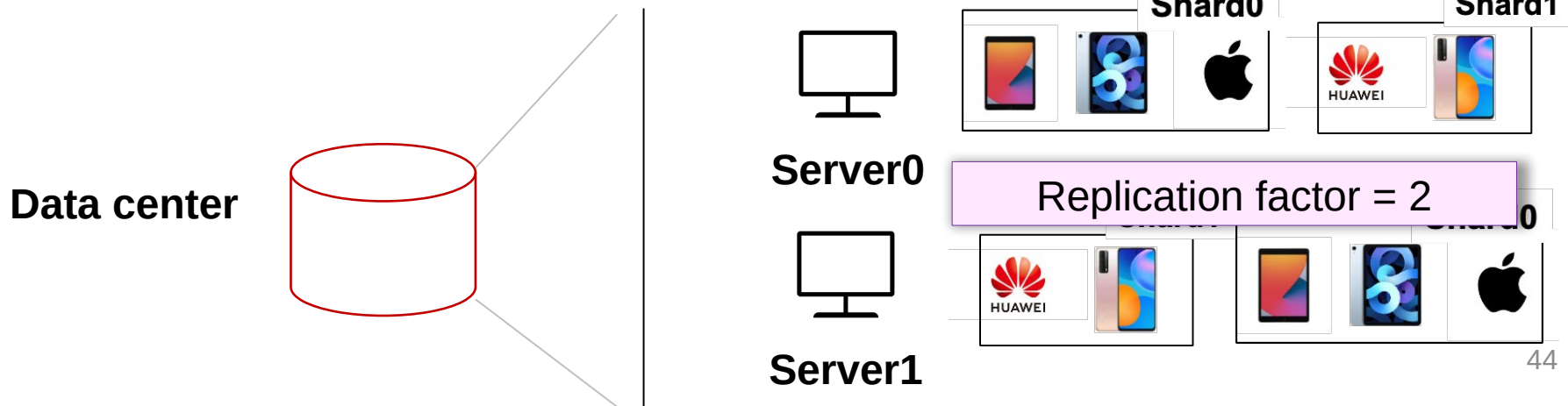
# Problems faced by large-scale company

## Fault tolerance

- Large-scale companies are built over large-scale datacenters (>10K servers)
- Machine failures are particular common in large-scale datacenters

## Solution: replication

- Each shard is replicated on multiple servers



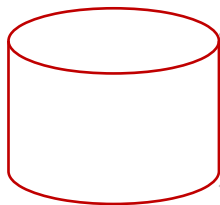
## Question: is a replication factor of 2 ok?

Ok means can achieve **single-copy consistency**

Whether OK depends on the scenarios (& setup)

- Under primary-backup replication, it is ok
  - We have a view server
- Under Paxos & Raft, it's not! (no majority)

Data center



Server0



Server1

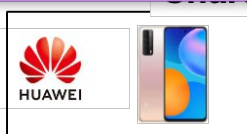
Shard0



Shard1



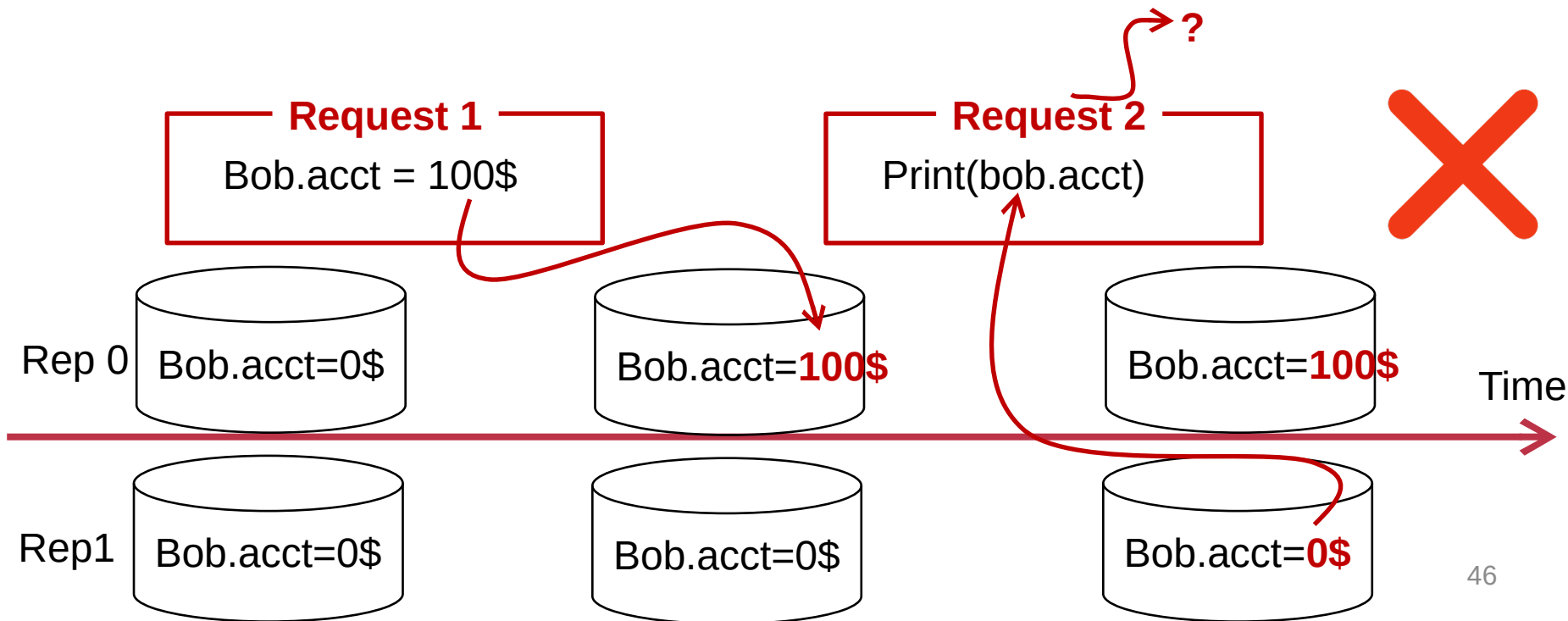
Replication factor = 2



# Replications causes consistency problem

Could not **achieve single-copy consistency** if allow client arbitrary reads/writes

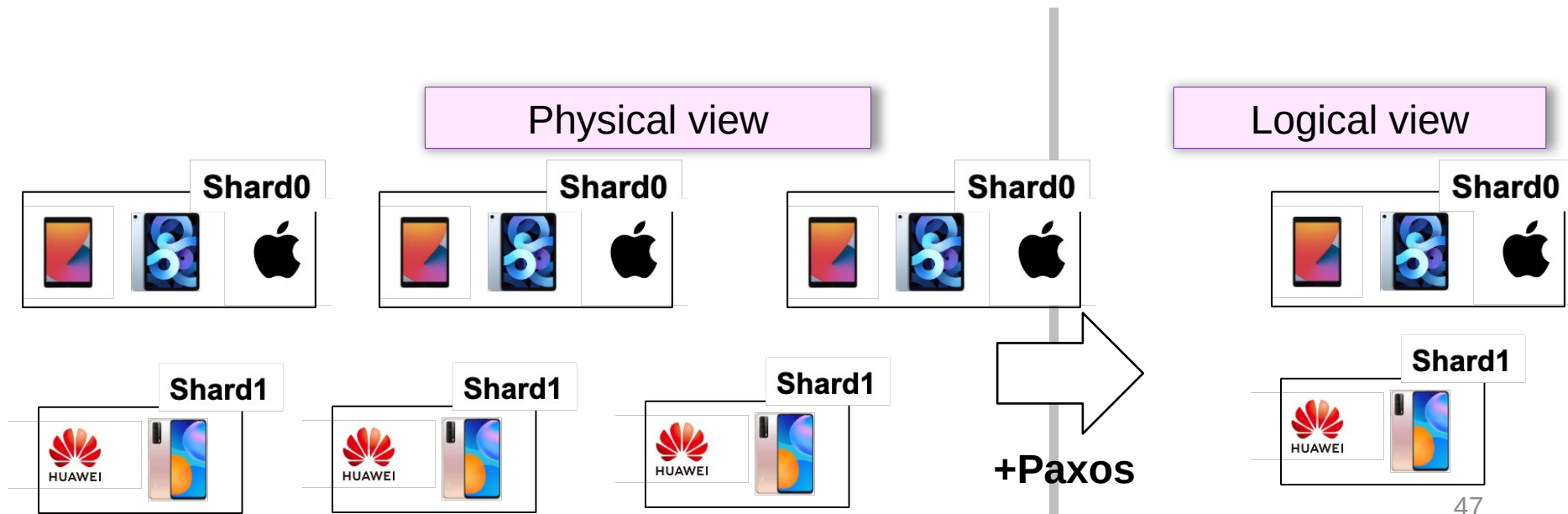
- E.g., consistency among replicas



# Solution: Paxos for single-copy consistency

Spanner uses Multi-Paxos w/ leaders to abstract the replicated shard

- On the developer's perspective, a shard w/ replication can be viewed as a single-shard (single-copy consistency)



# Geo-replicated datacenters

e.g., The **locations** of Alibaba datacenters



## ○ Regions outside Mainland China

Alibaba Cloud offers an expanding network of CDN nodes and deployment regions, including the first public cloud data center regions in the Middle East (Dubai) and Indonesia, a string of strategic data centers in Asia, and a strong presence in North America, Europe, and Australia.

## ○ Regions in Mainland China

Data center regions in China offer BGP backbone network lines providing high-quality coverage country-wide to ensure stable and fast access inside the Mainland. In general, we recommend customers to select the data center closest to their end-users to further speed up online access.

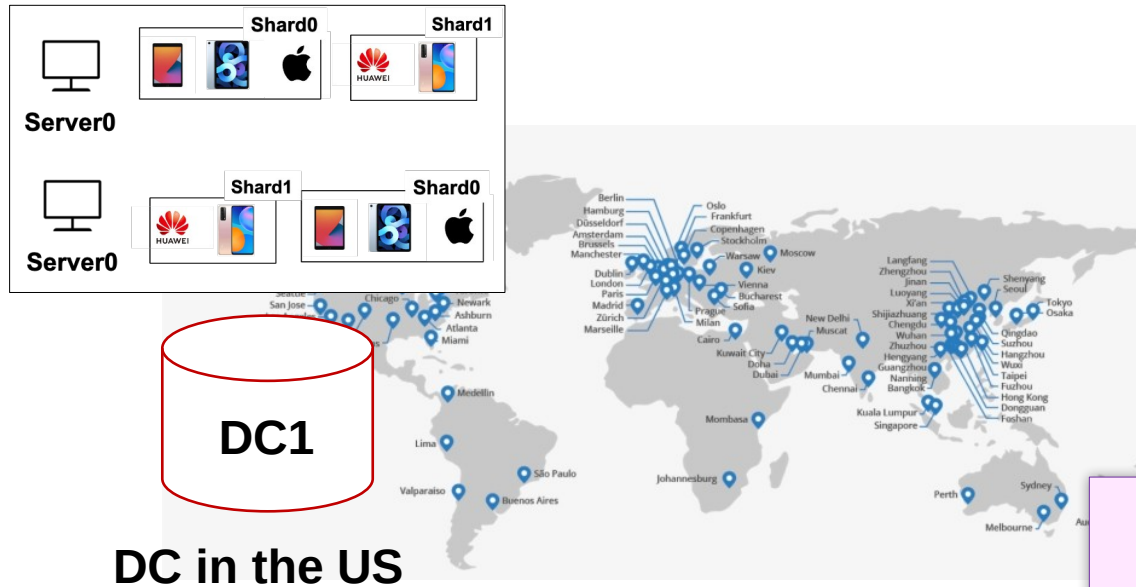
What can go wrong if we only replicate the data  
**only within one datacenter?**



# Replication within a DC is insufficient

A single datacenter (DC) can fail due to natural disasters

A DC cannot serve requests well (in low latency) across the planet



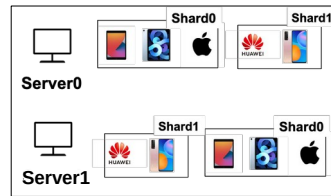
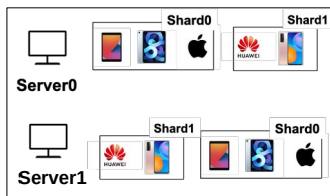
Users from china will suffer from long latency

# Spanner further replicates the data across datacenters

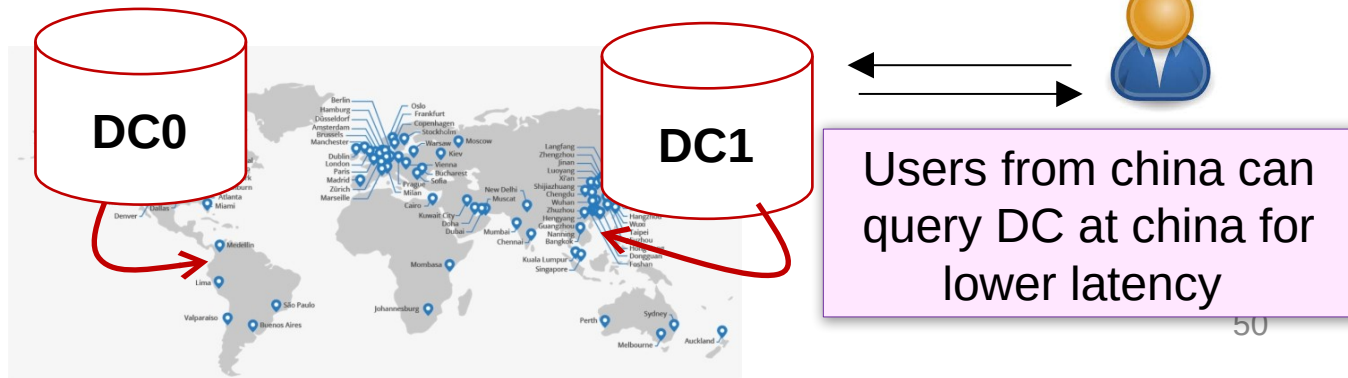
Data sharded & replicated over many machines & **datacenters**

- Tolerate machine failures across datacenters & lower request latency

User data



Data centers



## 2-phase commit + 2-phase locking

With Paxos, multiple replicas of a shard can be viewed as one

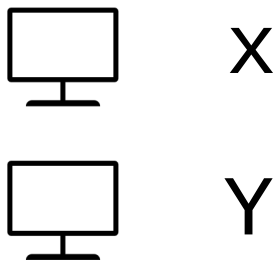
- E.g., the developer sees an X, but X is actually replicated multiple times

However, Paxos is insufficient

- A user request (TX) can touch multiple shards
- How to coordinate these accesses?

Using two-phase locking for **before-or-after atomicity** & two-phase commit for **multi-site atomicity**

```
tx.begin()  
x = x + 1  
y = y + 1  
tx.end()
```



# Big picture of Spanner

## How to scale to a larger dataset?

- Sharding

## How to tolerate failures across machines & datacenters?

- Replicated state machine w/ Paxos
  - Can also use raft (e.g., TiDB)

## How to execute read-write transaction to ensure before-or-after atomicity and multi-site atomicity?

- 2PL + 2PC

# Execution flow of read-write transaction(TX)

## Read-write TX:

- A TX that both reads and writes the data

```
tx.begin()  
x = x + 1  
y = y + 1  
tx.end()
```



**Client**

# Execution flow of read-write transaction(TX)

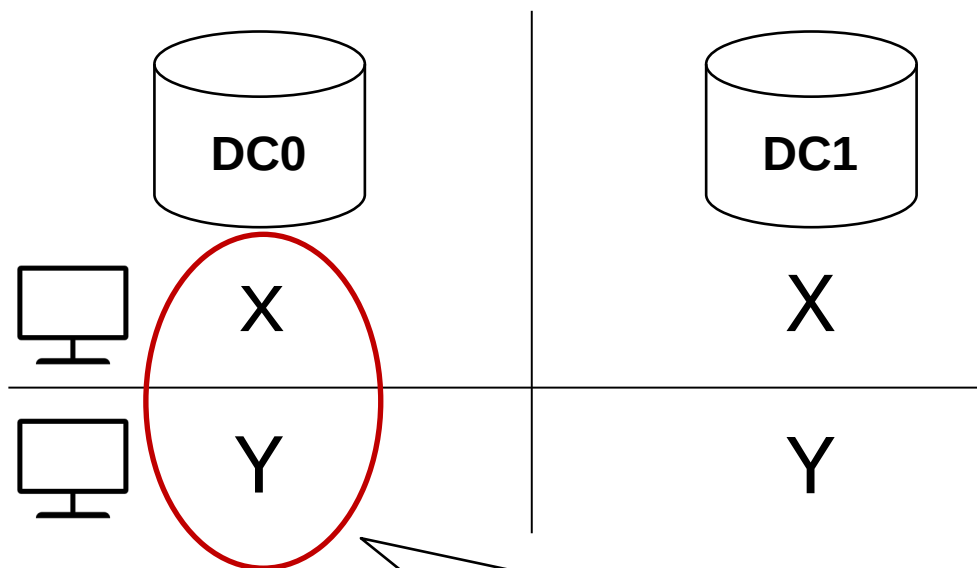
## Read-write TX:

- A TX that both reads and writes the data

```
tx.begin()  
x = x + 1  
y = y + 1  
tx.end()
```



Client



X,Y are sharded on two servers

# Execution flow of read-write transaction(TX)

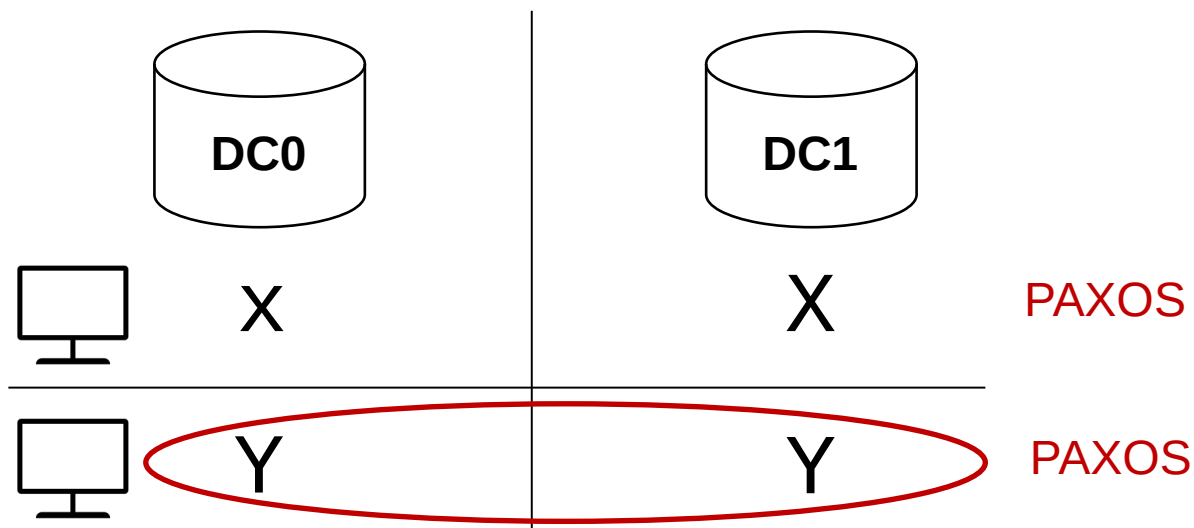
## Each shard is replicated

- The run PAXOS to behavior as a single (logical) shard

```
tx.begin()  
x = x + 1  
y = y + 1  
tx.end()
```



Client



Each shard is replicated on multiple DCs

# Execution flow of read-write transaction(TX)

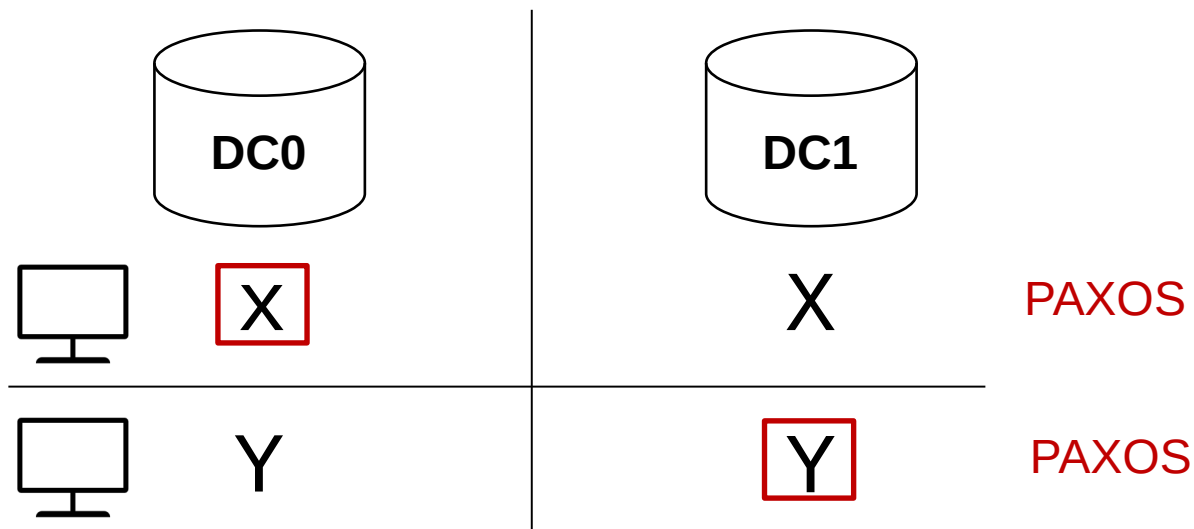
Each shard is replicated w/ PAXOS

- A replica is selected leader to simplify execution (no Raft introduced then)

```
tx.begin()  
x = x + 1  
y = y + 1  
tx.end()
```



Client



Leader will execute request  
Start a PAXOS proposal accordingly



## Execution flow of read-write transaction(TX)

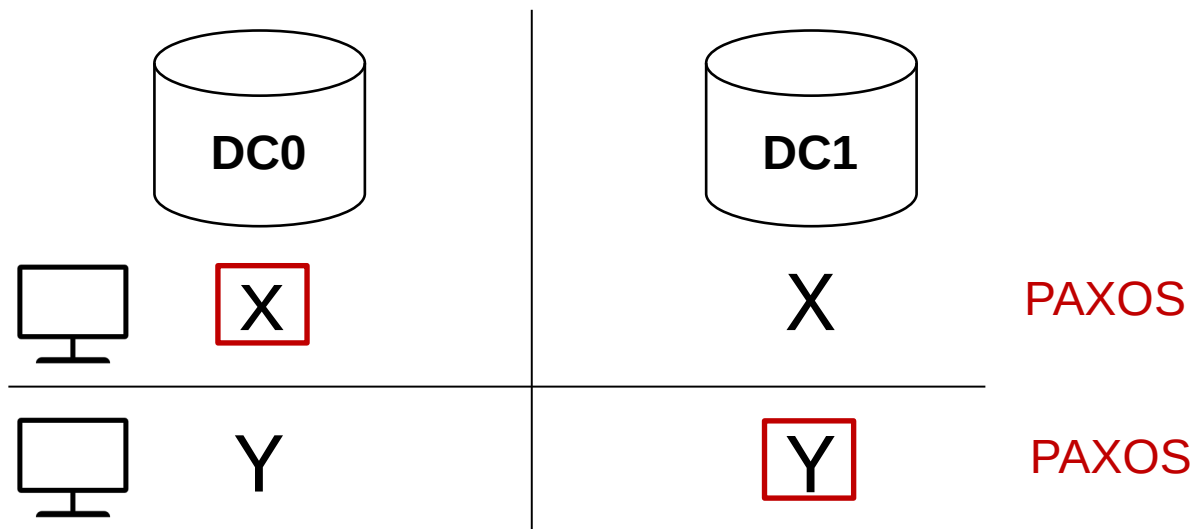
Spanner uses standard 2PC & 2PL for executing read-write TX

- Two-phase commit & two-phase locking

```
tx.begin()  
x = x + 1  
y = y + 1  
tx.end()
```



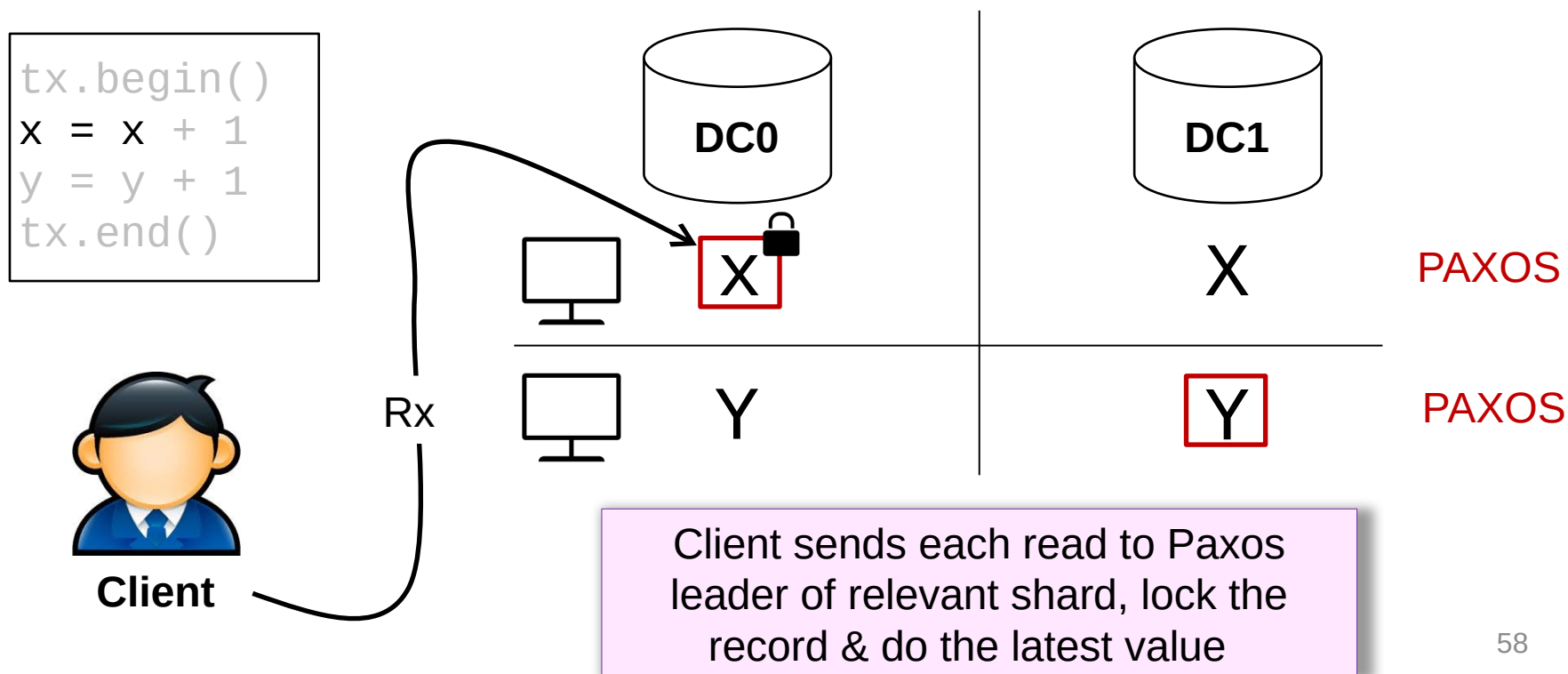
Client



## Execution flow of read-write transaction(TX)

Upon read, the leader will return the latest value of data

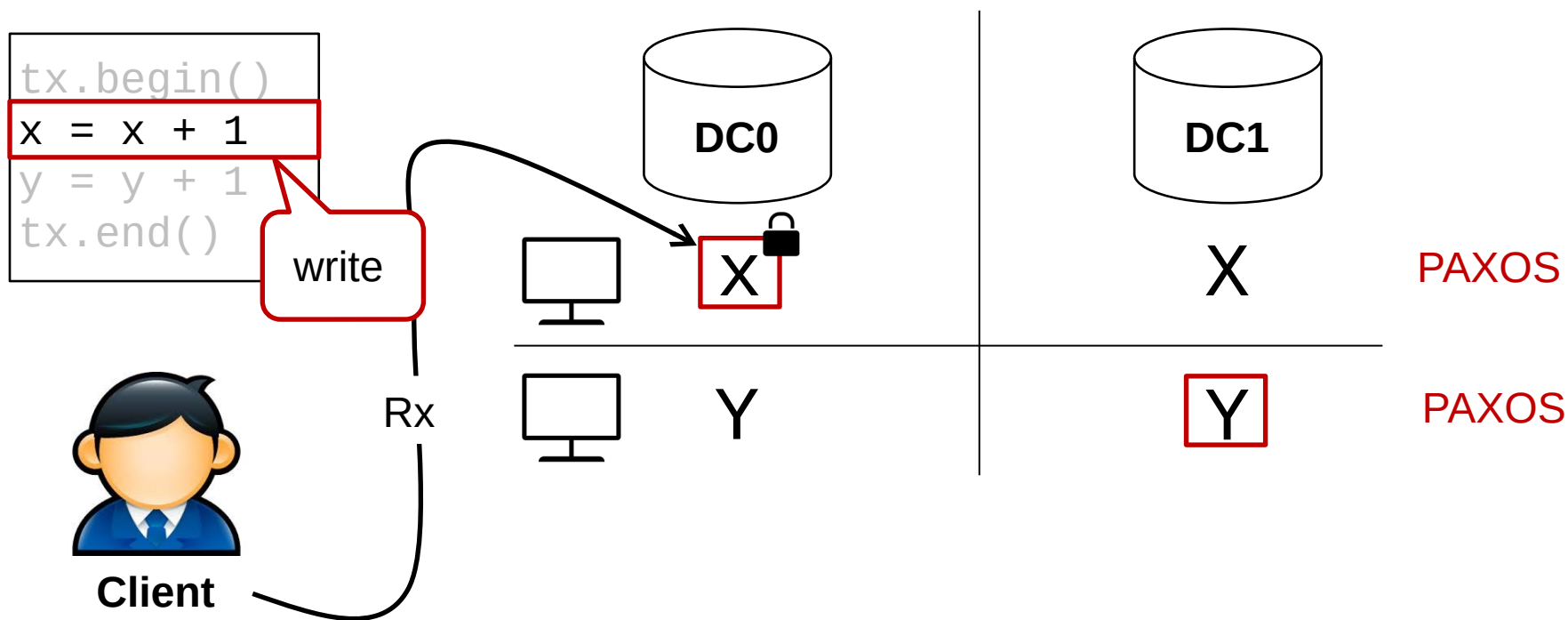
- Also hold the lock on it (2PL)



## Execution flow of read-write transaction(TX)

The write will be buffered at local client

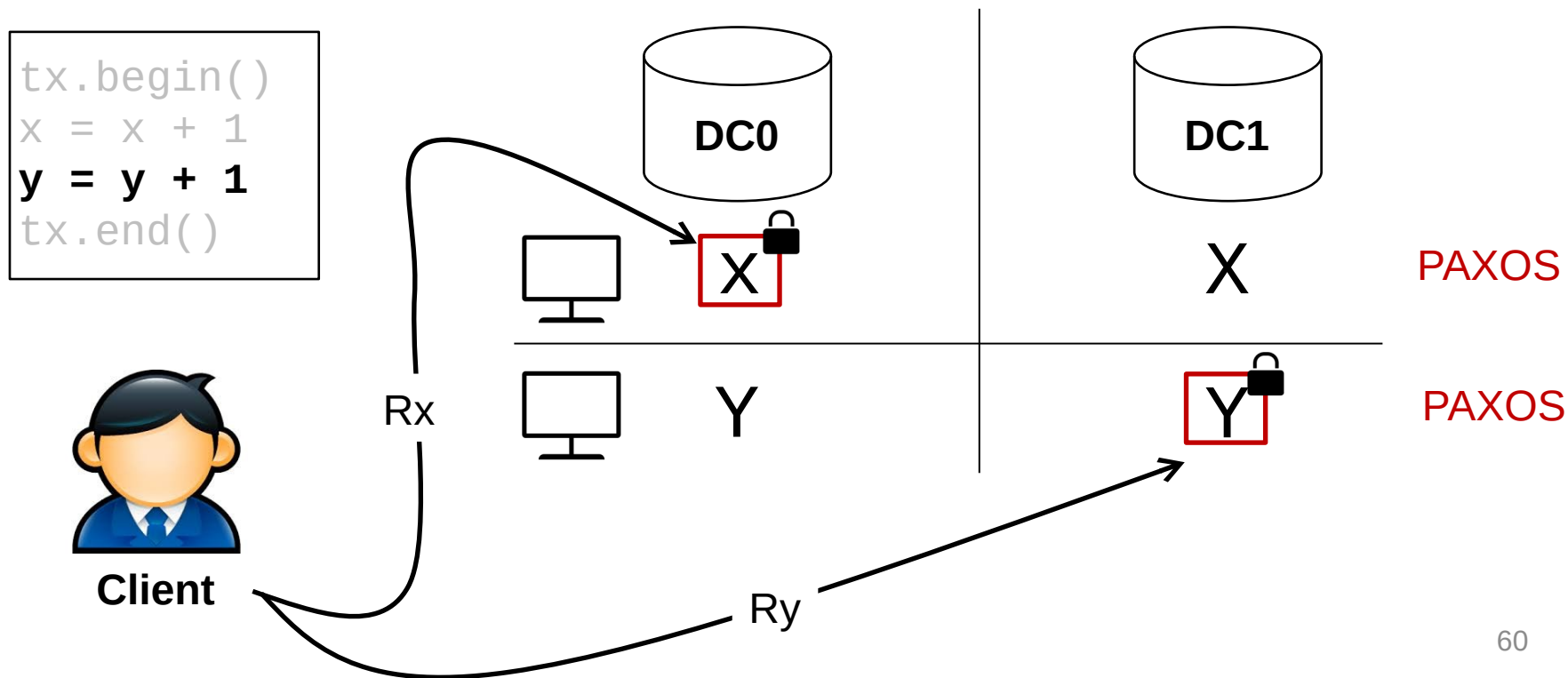
- They will be written back via 2PC at the TX's commit time



# Execution flow of read-write transaction(TX)

Commit will use 2PC to write all the updates back

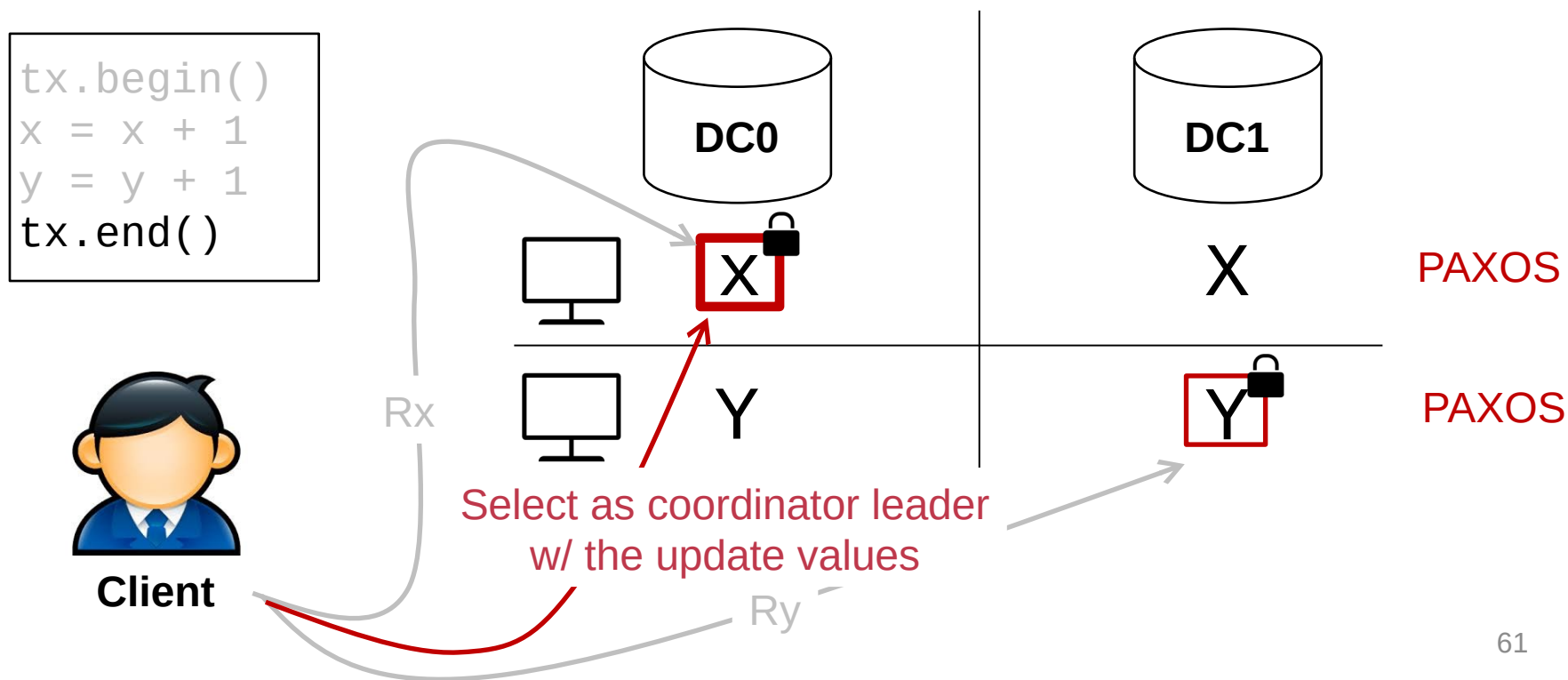
- It will choose a shard leader as the coordinator leader



## Execution flow of read-write transaction(TX)

The coordinator uses its PAXOS group to replicate the TX states

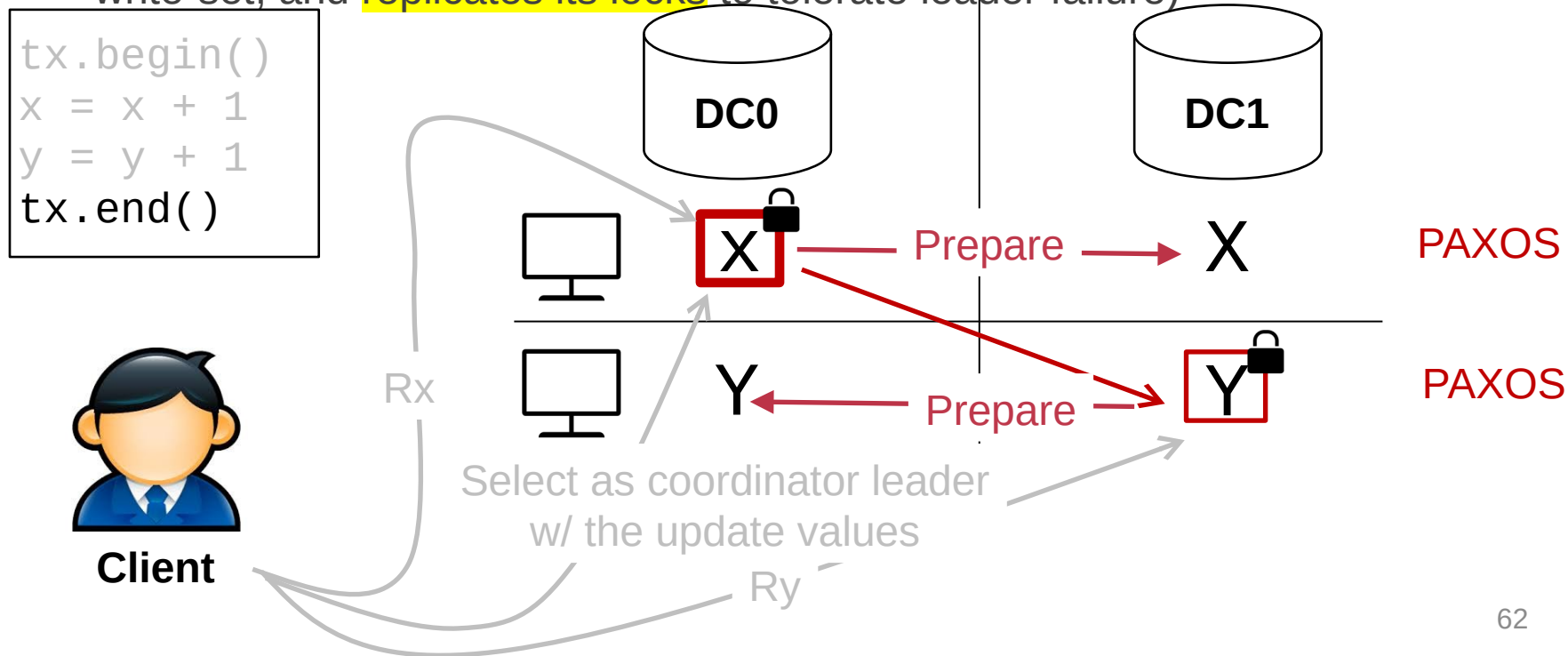
- Why? Otherwise, the 2PC may fail if the coordinator crashes!



## Execution flow of read-write transaction(TX)

The coordinator leader will send the update values to the host shard leader

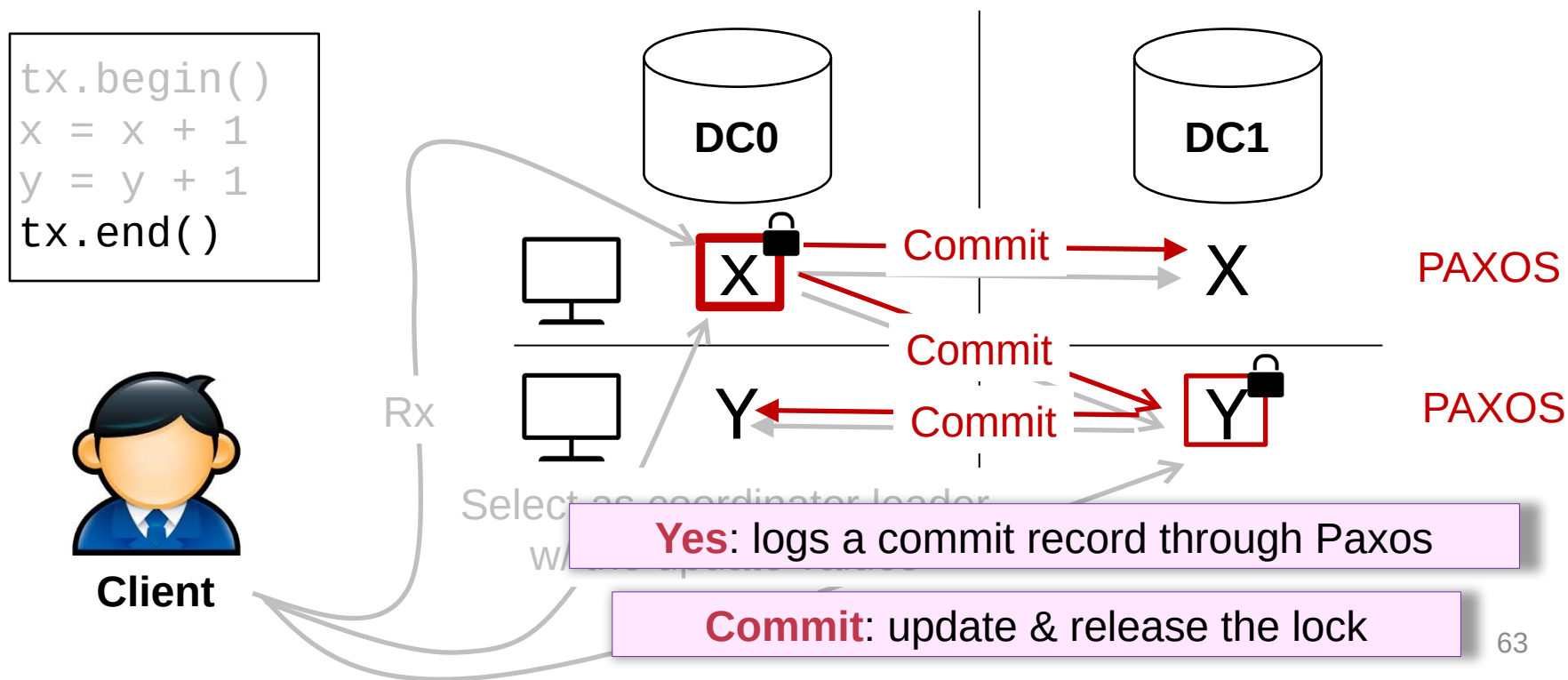
- And each shard leader execute the **prepare** with PAXOS (including locking the write-set, and replicates its locks to tolerate leader failure)



## Execution flow of read-write transaction(TX)

If all shard leader returns the **Yes** to the coordinator leader

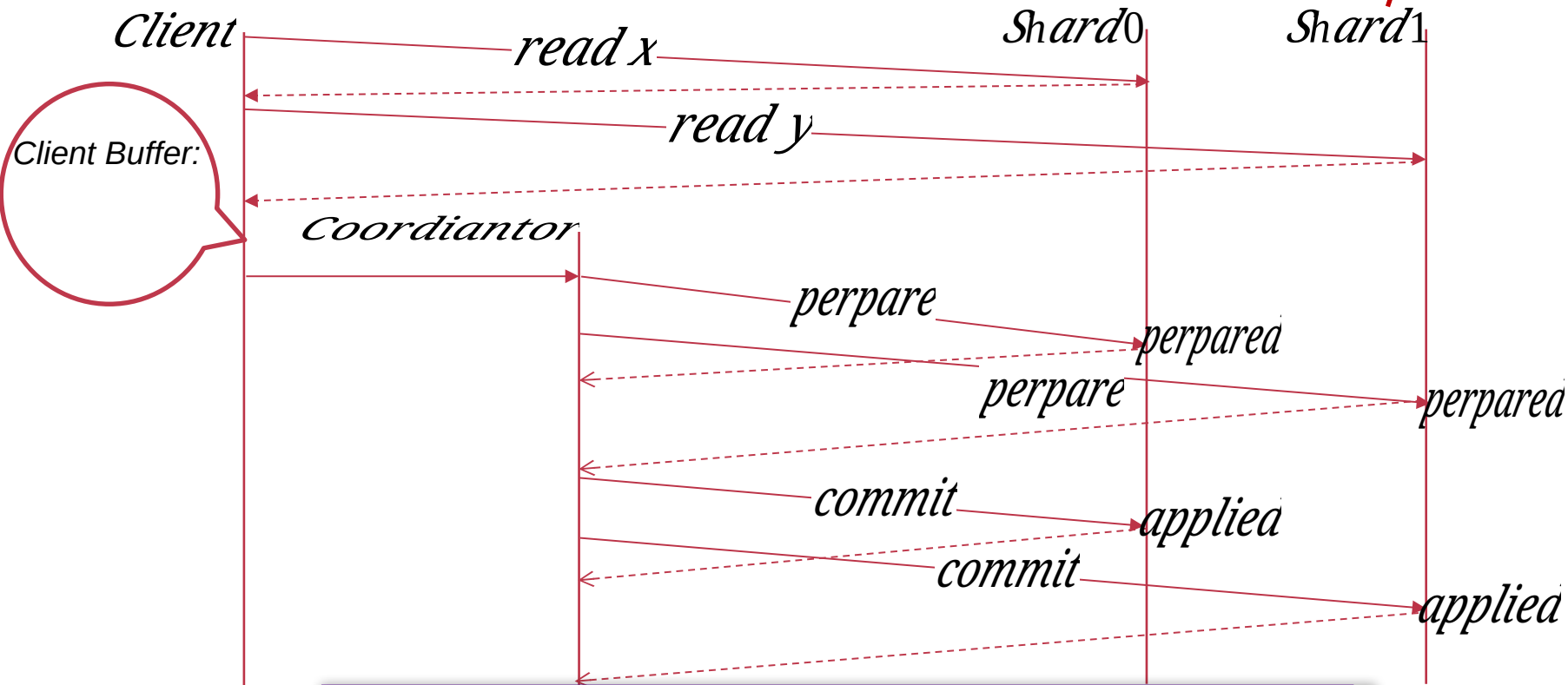
- The coordinator log the commit decision (via PAXOS) & commit others



## Read-write TX: put it all together

Multiple physical nodes  
backed by PAXOS

$2 PL + 2 PC + Paxos$



Note: we skip the network RTTs of Paxos



## Read-write TX: put it together

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


- A single-thread “machine” that never fails (even tolerate natural disasters)
- The “machine” has “unlimited” storage capacity
  - As long as the machines in the datacenters have enough capacity to store the data

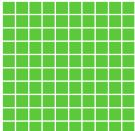

**But, what are the costs?**

# Read-write TX of Spanner so far

## Techniques: 2PL & 2PC & Paxos

- Not optimized– Spanner finds writes are infrequent at Google
- Maybe slow: many messages sent between machines
  - Also, across datacenters!

 Round trip in same datacenter: 500,000ns  $\approx$  500 $\mu$ s

 1,000,000ns = 1ms = 

 Packet roundtrip CA to Netherlands: 150,000,000ns  $\approx$  150ms

Table IV. Two-Phase Commit Scalability. Mean and Standard Deviations over 10 Runs

participants	latency (ms)	
	mean	99th percentile
1	14.6 $\pm$ 0.2	26.550 $\pm$ 6.2
2	20.7 $\pm$ 0.4	31.958 $\pm$ 4.1
5	23.9 $\pm$ 2.2	46.428 $\pm$ 8.0
10	22.8 $\pm$ 0.4	45.931 $\pm$ 4.2
25	26.0 $\pm$ 0.6	52.509 $\pm$ 4.3
50	33.8 $\pm$ 0.6	62.420 $\pm$ 7.1
100	55.9 $\pm$ 1.2	88.859 $\pm$ 5.9
200	122.5 $\pm$ 6.7	206.443 $\pm$ 15.8

Source:

[https://colin-scott.github.io/personal\\_website/research/interactive\\_latency.ht](https://colin-scott.github.io/personal_website/research/interactive_latency.ht)

# What about read-only TX?

## Problem: read-only TXs are massive faced by Google

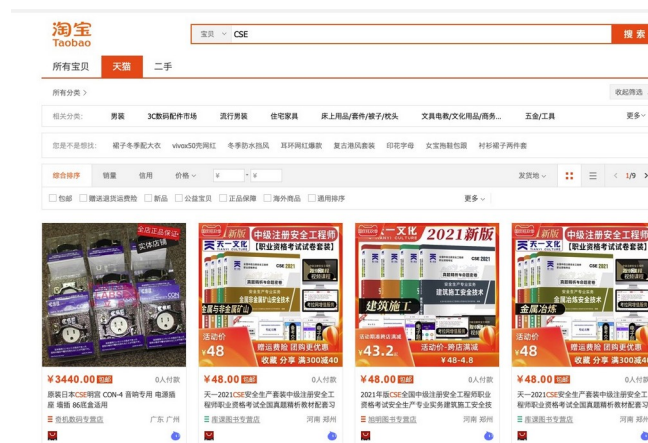
- Each read-only TX may touch many datasets

## Drawbacks of 2PL (in read-only TX)

- Acquire the read lock is costly (need Paxos!)
- May possibly lock the items very long

## Solution

- Multi-version concurrency control



# Review: Multi-version Concurrency Control

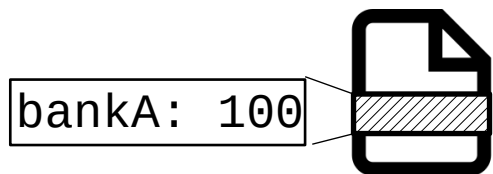
Each data item has **multiple** versions

- When accessing different versions of data, probably no conflict!

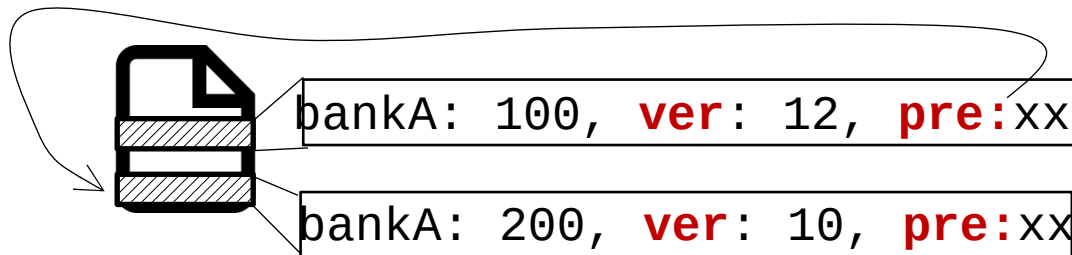
Key (high-level) idea

- Writes **don't overwrite** the original data
- Instead, writes **install new versions** of data
- Reads read from a "**snapshot**" of data

Benefits: no lock or validation during TX's execution



Single-version



Multi-version

## Review: Snapshot Isolation

A popular multi-version concurrency control (MVCC) scheme

- Transactions will get **start** and **commit** timestamp
- Use start timestamp to find the snapshot to read
- Use commit timestamp to install new versions

Transactions:

- **WRITES** a local **buffer** (similar to OCC)
- **READs** a “**snapshot**” of entire data image
- **COMMIT** only if no **write-write** conflict
  - Install new versions of data

Drawback: no serializability guarantee !

## 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!

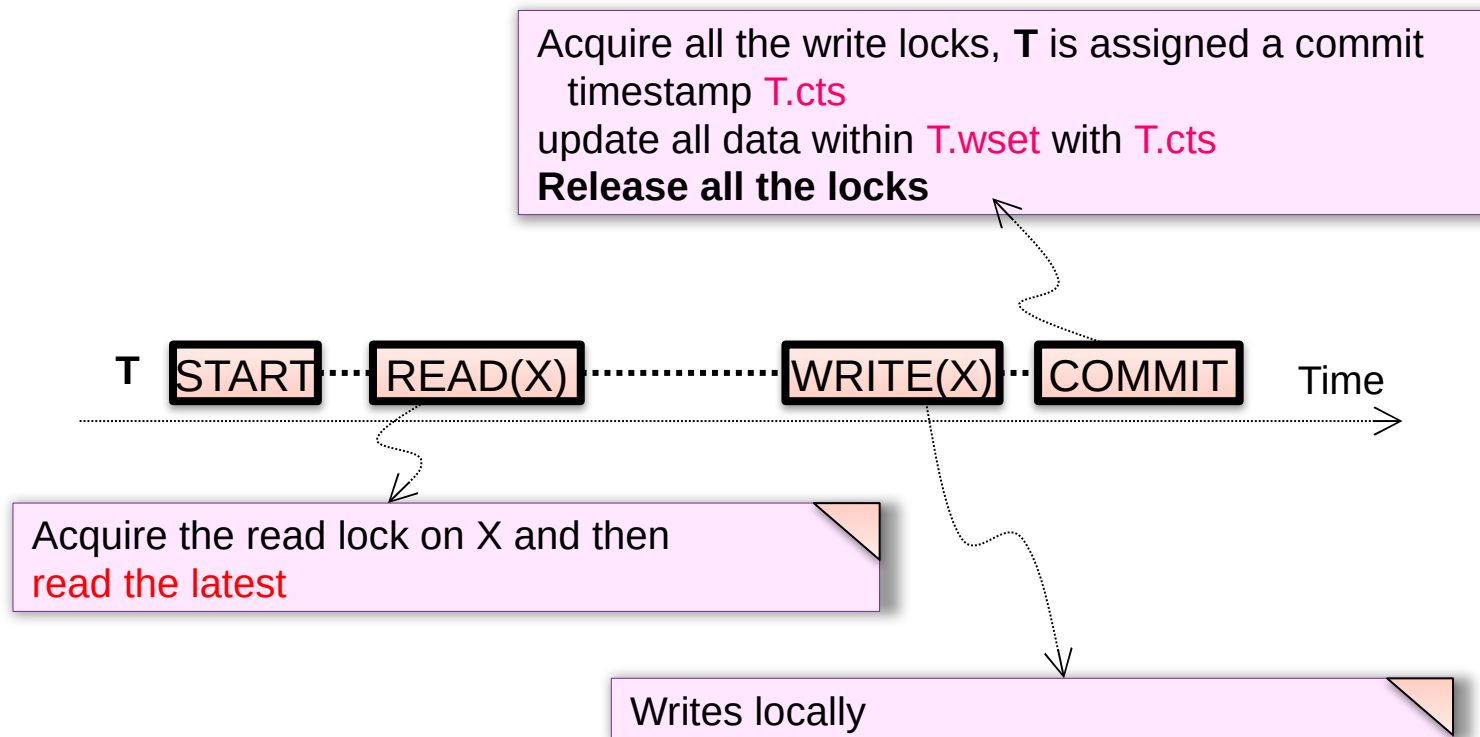
### Use MVCC to execute the read-only TX

- Read-only TX no-longer to acquire locks for the execution

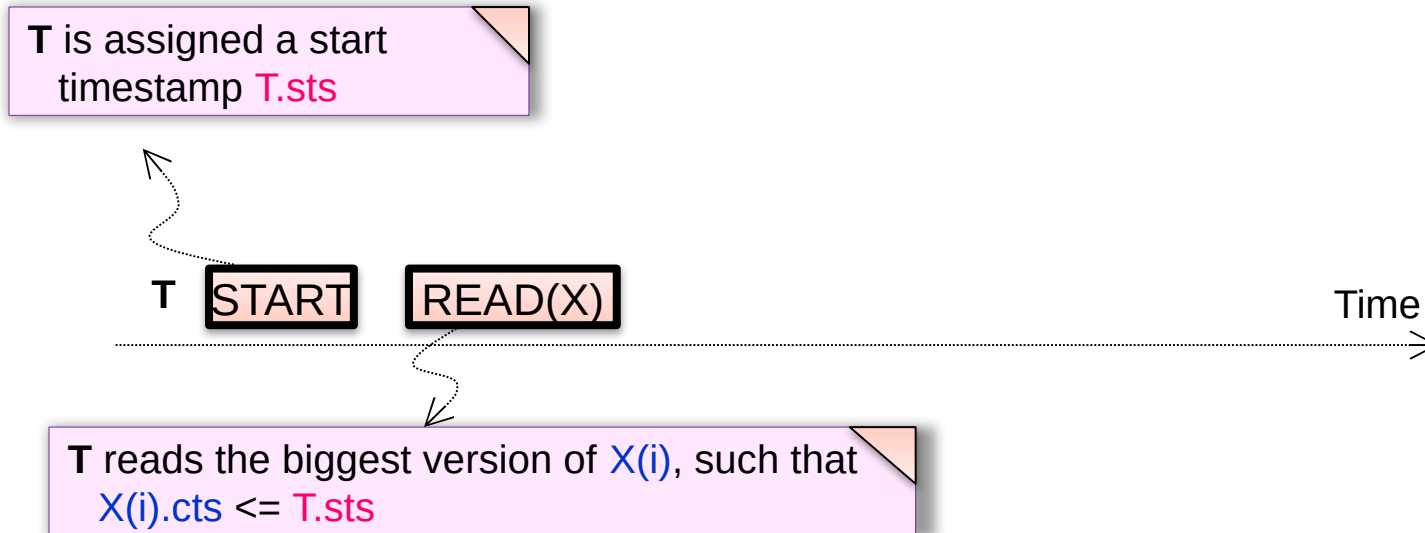
### Note that similar technique also applies to OCC

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

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



## MV-2PL: read-only TX





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

Timestamp of start & commit should be **assigned in an increasing order**

- E.g., global counters with atomic increase
- Note: read-only TX does not necessary increase the time

**Example implementation: global counter (on a single machine)**

```
u64 global_time; // initial 0
```

```
struct TX {  
    u64 start_time;  
    u64 commit_time;  
    set<...> write_set;  
    ...  
}
```

```
tx_begin(tx) { // read-only  
    tx.global_time =  
    READ(global_time);  
    ...  
}
```

```
tx_commit(tx) { // read-write  
    tx.commit_time =  
    FAA(global_time);
```

Question: is global counter suitable  
for Spanner's use case?

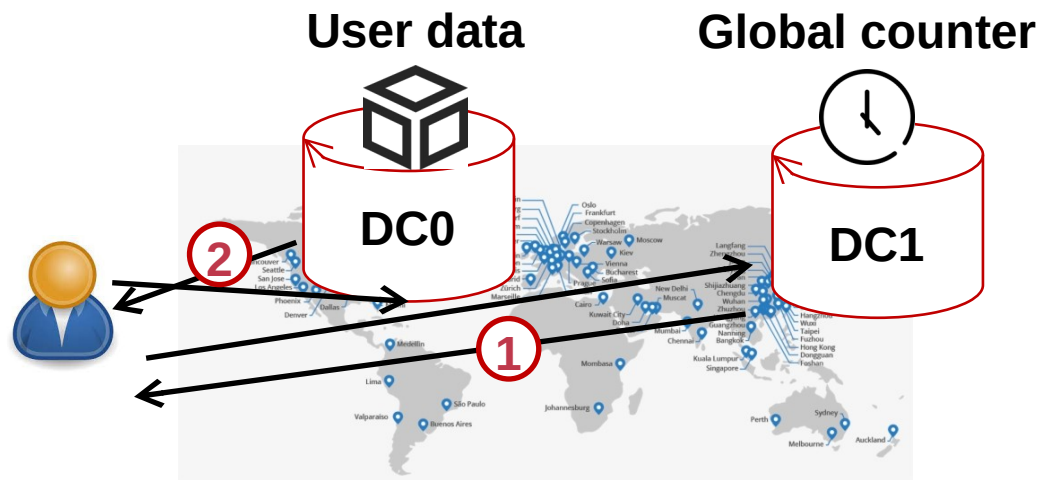
# Global time is inefficient for Spanner's use case

## Performance overhead

- 1. Extra latency overhead
- 2. Scalability bottleneck

## Question: can we avoid reading the global counter?

- Possible when considering the read-only TX



## Cache the time locally to avoid querying the global counter

**Observation: the read-only TX can read a stale timestamp**

- E.g., not reading from the latest global counter
- Still correct
  - TX only reads from a (possible) stale snapshot

# Cache global counter to avoid frequently reads

## Cache start timestamp for the read-only TX

- The read-only TX no longer needs to do the FAA all the time

**Note:** here FAA maybe implemented as a remote procedure call

- To the server that stores the global counter

```
u64 global_time; // initial 0
```

```
struct TX {  
    u64 start_time;  
    u64 commit_time;  
    Option<u64> cached_time;  
    set<...> write_set;  
    ...  
}
```

```
tx_begin(tx) {  
    if cached_time.is_none():  
        tx.cached_time=  
        Some(READ(global_time));  
    tx.start_time =  
    tx.cached_time.unwrap()  
    ...  
}
```

## Example revisit w/ cached time

Global counter

(initial 2)

3

T1 (cached\_time  
= 0)

T2

T1:  
Print(A+B  
)

T2:  
B = 2  
A = 3

A: A<sub>1</sub> A<sub>0</sub>  
B: B<sub>1</sub> B<sub>0</sub>

Commit\_time = FAA(G)

Write(B, B<sub>3</sub>)

Write(A, A<sub>3</sub>)

A: **A<sub>3</sub>** A<sub>1</sub> A<sub>0</sub>  
B: **B<sub>3</sub>** B<sub>1</sub> B<sub>0</sub>

Start\_time = 0 // cached

Read(A) = **A<sub>0</sub>**

Read(B) = **B<sub>0</sub>**

A: A<sub>3</sub> A<sub>1</sub> **A<sub>0</sub>**  
B: B<sub>3</sub> B<sub>1</sub> **B<sub>0</sub>**

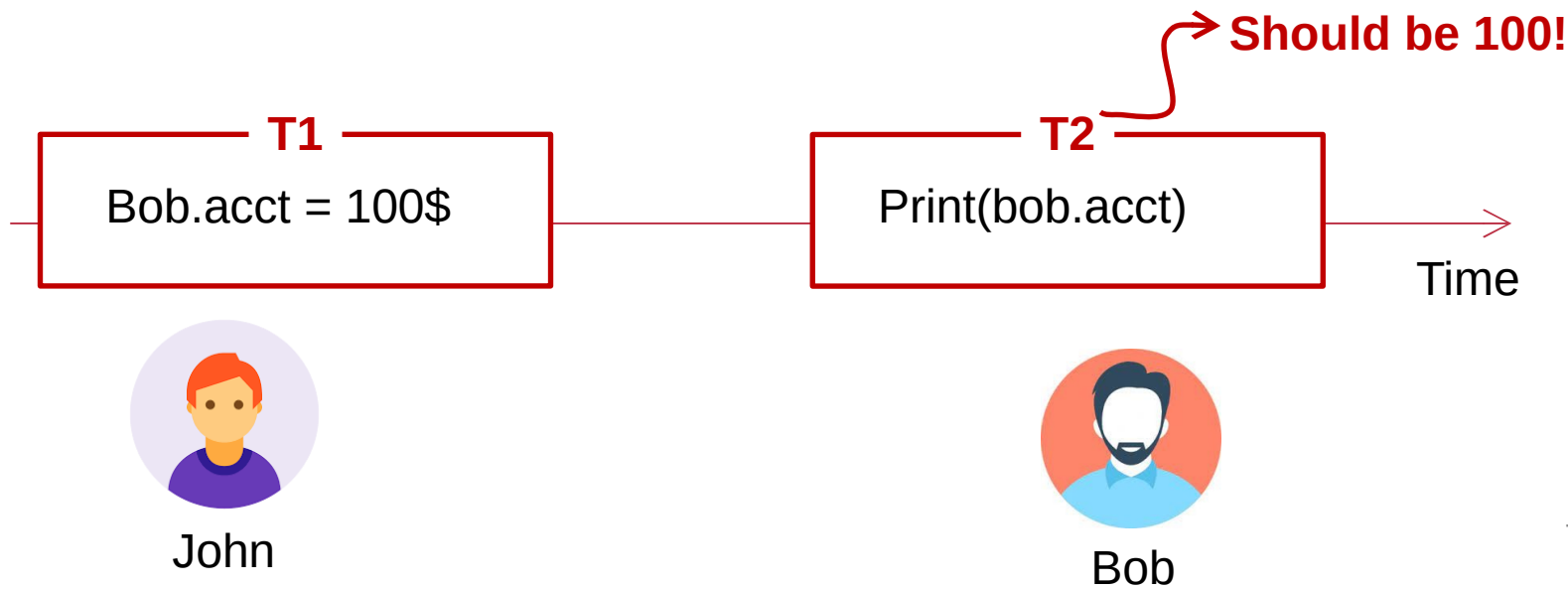
## Drawbacks of cached time

### No freshness guarantees

- i.e., no external consistency

### External consistency is the most desirable for the programmer

- 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

### **Read-write TX still need to acquire the global counter**

- Extra latency to communicate with the server that stores the global time
- Possible performance bottleneck



## Challenge of the timing in MVCC so far

Time needs synchronization for **external consistency (strict serializability)**

- If a read-write TX has decided to commit, then a read-only TX starts, then must ensure:

- $\text{Time}_{\text{read-only}} \geq \text{Time}_{\text{read-write}}$

**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' time are different

Observation: we may not get the accurate physical time, but we can get an accurate **bound**

# TrueTime API of Spanner

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

- The interval is **a bound**. i.e., the time **of the time server** must be in this bound
- The interval is the physical time. i.e., familiar to the user

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

Linux Time API

```
struct time_interval{
    timeval Lower;
    timeval Upper;
};

int get_truetime(time_interval
*interval);
// Server timeval must be in
[L,U]
```

TrueTime API (Simplified)

## Power of TrueTime API (return [L,U])

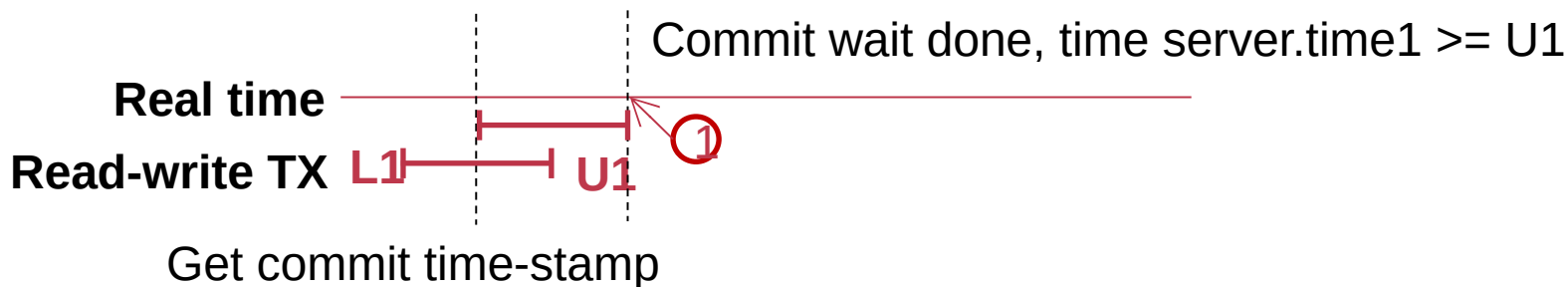
### Used to implement external consistency

- If a read-write TX has decided to commit, then:

- $\text{Time}_{\text{read-only}} \geq \text{Time}_{\text{read-write}}$

### How to achieve this?

- **Commit wait** for read-write TX: after acquire the commit timestamp, the coordinator wait until  $(U - L)$  and uses  $U$  as  $\text{Time}_{\text{read-write}}$



# Power of TrueTime API (return [L,U])

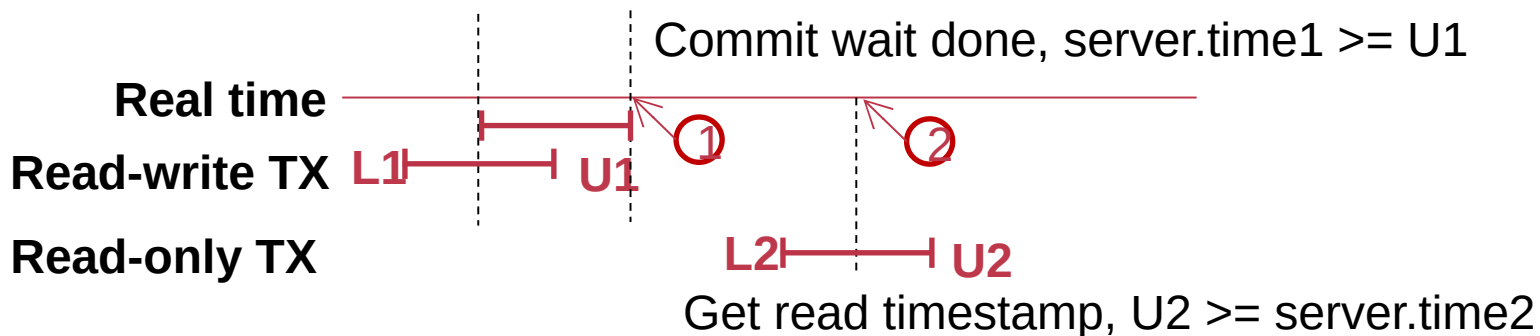
## Used to implement external consistency

– If a read-write TX has decided to commit, then:

- $\text{Time}_{\text{read-only}} \geq \text{Time}_{\text{read-write}}$

## How to achieve this?

- **Commit wait** for read-write TX: after acquire the commit timestamp, the coordinator wait until  $(U - L)$  and  $U$  as  $\text{Time}_{\text{read-write}}$
- For read-only TX, simply uses  $U$  as the read timestamp



# Power of TrueTime API (return [L,U])

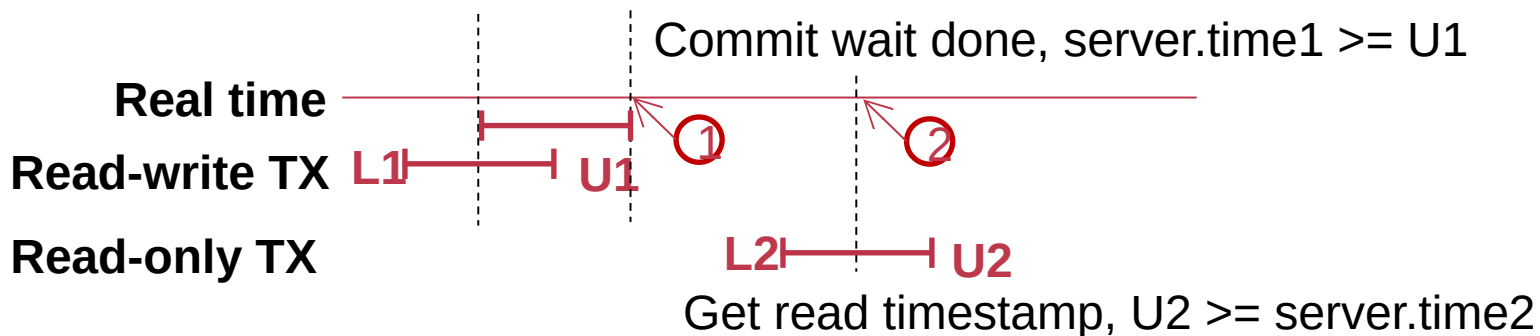
## Used to implement external consistency

– If a read-write TX has decided to commit, then:

- $\text{Time}_{\text{read-only}} \geq \text{Time}_{\text{read-write}}$

## Correctness

–  $\text{Time}_{\text{read-write}}(U1) < \text{server.time1} < \text{server.time2} < U2$  ( $\text{Time}_{\text{read-only}}$ )



How to achieve the bound of TrueTime?

## 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<sup>[1]</sup>**

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



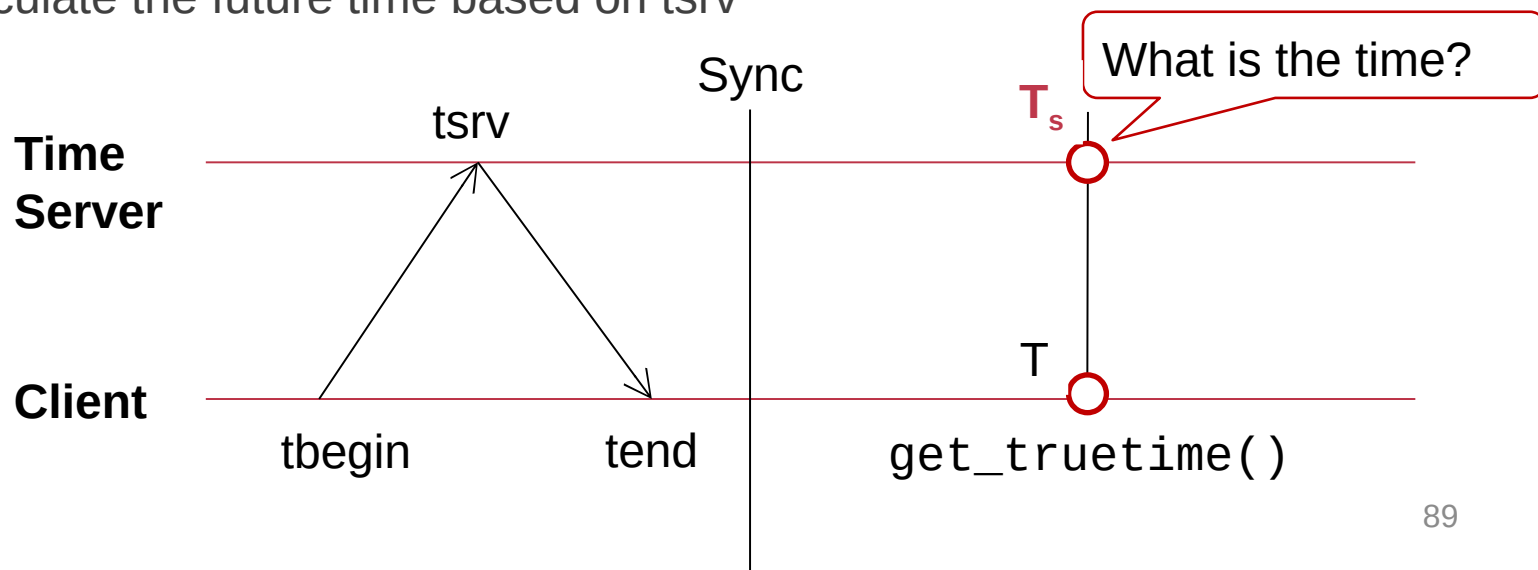
# TrueTime: how to calculate the bound?

## Problem statement

- If the local clock of client is  $T$ , what is the time interval of the time server?

**High-level idea: send RPC to the server for the query as the measurements!**

- But unlike NTP, we don't adjust local time according to  $tsrv$
- We calculate the future time based on  $tsrv$



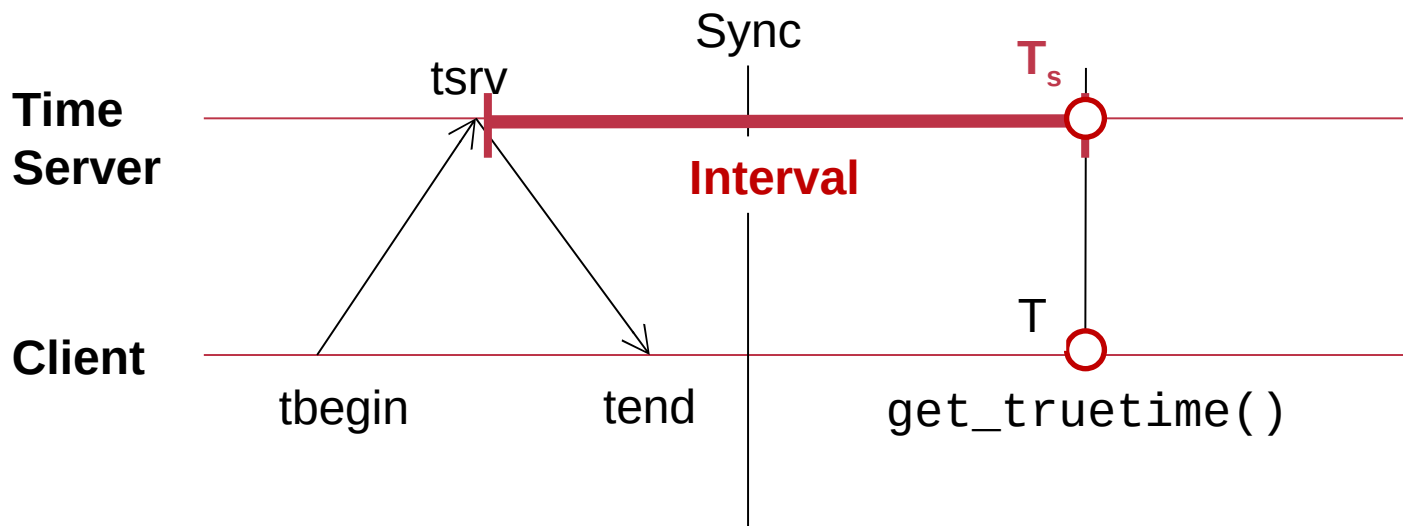
## TrueTime: how to calculate the bound?

**Simplification:** first assuming server advances in the same speed as clients

- Since we know the **tsrv**, the key question is how to calculate the **interval**

### Question

- How do we calculate the interval based on tsrv, tbegin & tend?



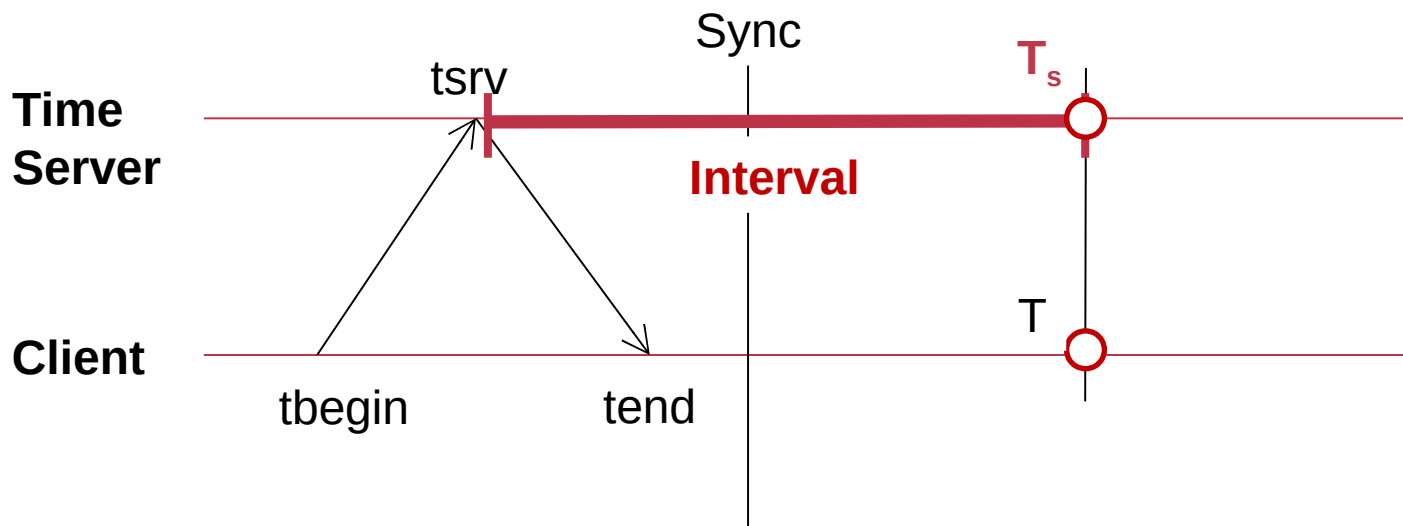
## TrueTime: how to calculate the bound?

Since we know the **tsrv**, the key question is how to calculate the **interval**

- $\text{Interval} \geq T - \text{tend}$
- $\text{Interval} \leq T - \text{tbegin}$
- $T_s = \text{tsrv} + \text{Interval}$

Question: what if there is a drift between client & server?

➤  $T - \text{tend} + \text{tsrv} \leq T_s \leq T - \text{tbegin} + \text{tsrv}$  (based on our simplification)



## TrueTime: how to calculate the bound?

~~Simplification: assuming server advances in the same speed as clients~~

- $T - t_{end} + t_{srv} \leq T_s \leq T - t_{begin} + t_{srv}$  (based on our simplification)

Solution: regulate with the drift rate

Assume a fixed  $\epsilon$  drift rate between client & server

- After  $t$  time, the drift between client & server is  $(1 + \epsilon)$  or  $(1 - \epsilon)$ 
  - Spanner assumes a fixed as 200us / second

Then the interval is regulated as

$$- \boxed{(T - t_{end}) * (1 - \epsilon) + t_{srv}} \leq T_s \leq \boxed{(T - t_{begin}) * (1 + \epsilon) + t_{srv}}$$

- Done

**L**

**U**

# Commit wait revisited

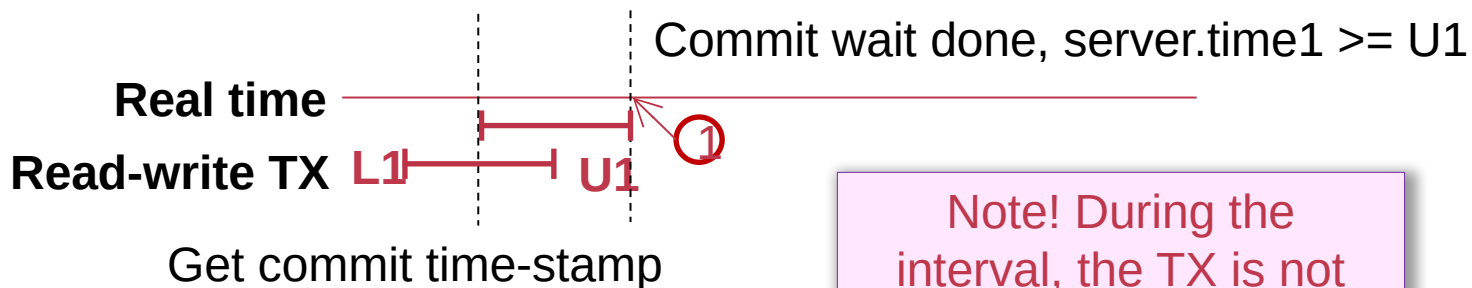
## Used to implement external consistency

– If a read-write TX has decided to commit, then:

- $\text{Time}_{\text{read-only}} \geq \text{Time}_{\text{read-write}}$

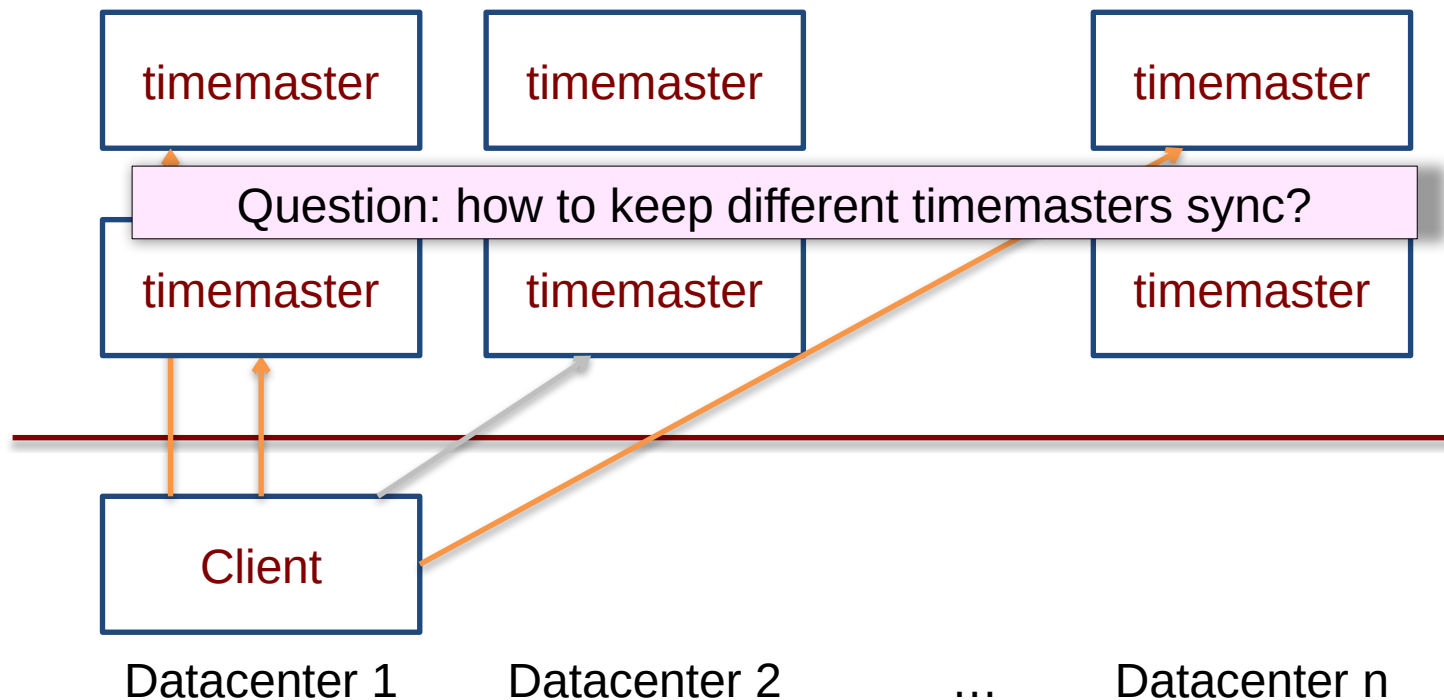
## How to achieve this?

– **Commit wait** for read-write TX: after acquire the commit timestamp, the coordinator wait until  $(U - L)(1 + \epsilon)$  and uses  $U$  as  $\text{Time}_{\text{read-write}}$



Note! During the interval, the TX is not committed

## TrueTime adopts multiple time servers



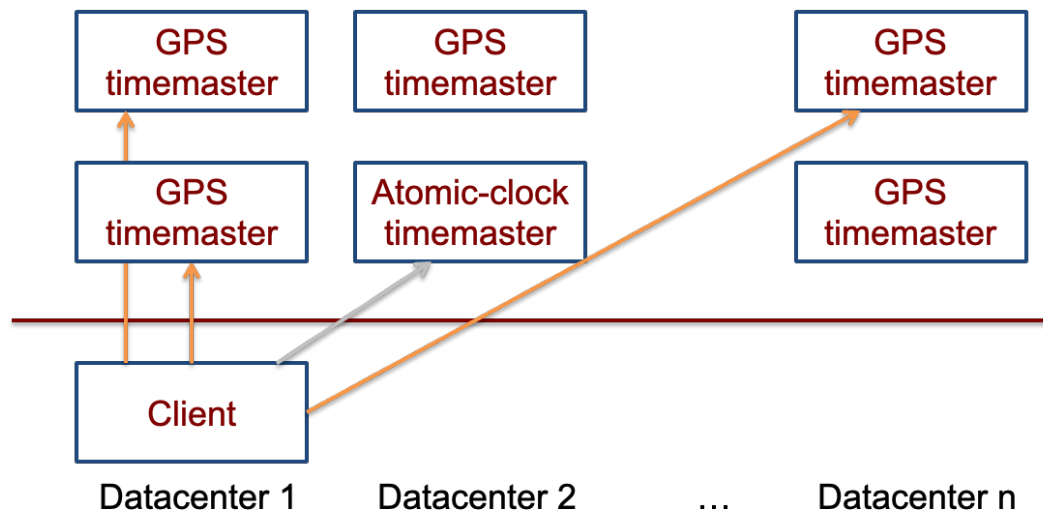
Called timemaster (◀◀) in Spanner

# TrueTime adopts multiple time servers

## Time servers are backed by GPS & atomic clocks



- High-precision clocks
- E.g., 1 second drift after 20,000,000 year (vs. 200us per second of CPU)
- Atomic clocks are synchronized with each other

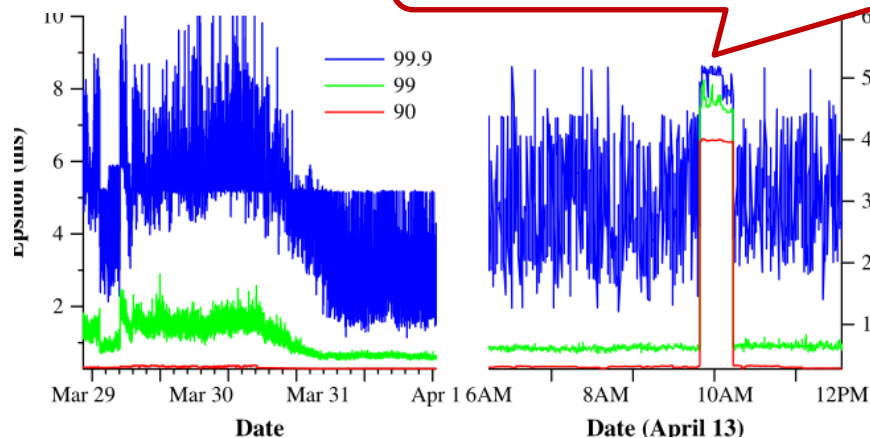
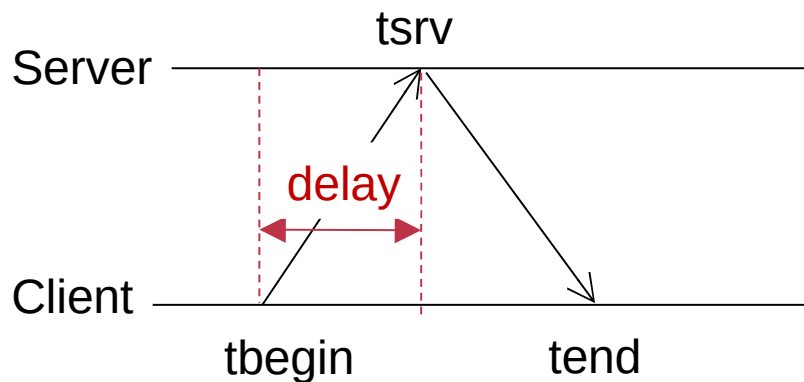


# Final takeaway of TrueTime: Network-Induced Uncertainty

## Interval of T

- $[(T - \text{tend}) * (1 - \varepsilon) + \text{tsrv}, (T - \text{tbegin}) * (1 + \varepsilon) + \text{tsrv}]$
- $\text{tsrv} - \text{tbegin}$  is roughly estimated as the network delay

Can have spikes if timemasters are out of services







## Summary

### **Reify clock uncertainty in time APIs**

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

### **Stronger semantics are achievable**

- Greater scale != weaker semantics