Spanner: Google's Globally-Distributed Database

COS 418: *Distributed Systems*
Precept 6

Themis Melissaris and Daniel Suo

Agenda

- Review {linear, serial, strict serial}izability
- Review concurrency controls
- Spanner
- Pro tips for Assignment 3

ACID properties of transactions

- **Atomicity:** Either **all** constituent operations of the transaction complete successfully, or **none** do
- **Consistency:** Each transaction in isolation preserves a set of **integrity constraints** on the data
- **Isolation:** Transactions' behavior not impacted by presence of **other concurrent transactions**
- **Durability:** The transaction's **effects survive failure** of volatile (memory) or non-volatile (disk) storage

Review *izability

Some context

- Terms come from two different communities (database and distributed systems). Overloaded!

- All refer to interleaving operations

- Definitions
  - **Operation**: typically refers to a single access operation (e.g., read, write)
  - **Transaction**: one or more operations that must be committed atomically

Linearizability

- Guarantee for a **single** operation on a **single** object

- Informally, writes should appear instantaneously within the system

- All later reads as defined by wall-clock time (i.e., real-time) reflect the written value or some later written value

- ‘Strong Consistency’ in CAP theorem
  - Yes, we use consistency in ACID to mean something different

Serializability

- Guarantee for **transactions**, or one or more operations on one or more objects

- A set of transactions over some objects should execute as though each transaction ran in *some* serial order (doesn't specify which one!)

- No real-time (i.e., world-clock) constraints; in other words, no deterministic order for transactions

- 'Isolation' in ACID properties

Strict serializability

- Linearizability + serializability
- Transactions have some serial behavior and that behavior corresponds to wall-clock time
- Straightforward to reason about for non-overlapping transactions
- What about overlapping transactions?

Review concurrency controls

ACID properties of transactions

- **Atomicity:** write-ahead logs and checkpoints
- **Consistency:** application logic
- **Isolation:** concurrency controls (locks, 2PL, OCC, MVCC)
- **Durability:** write-ahead logs and checkpoints

ACID properties of transactions

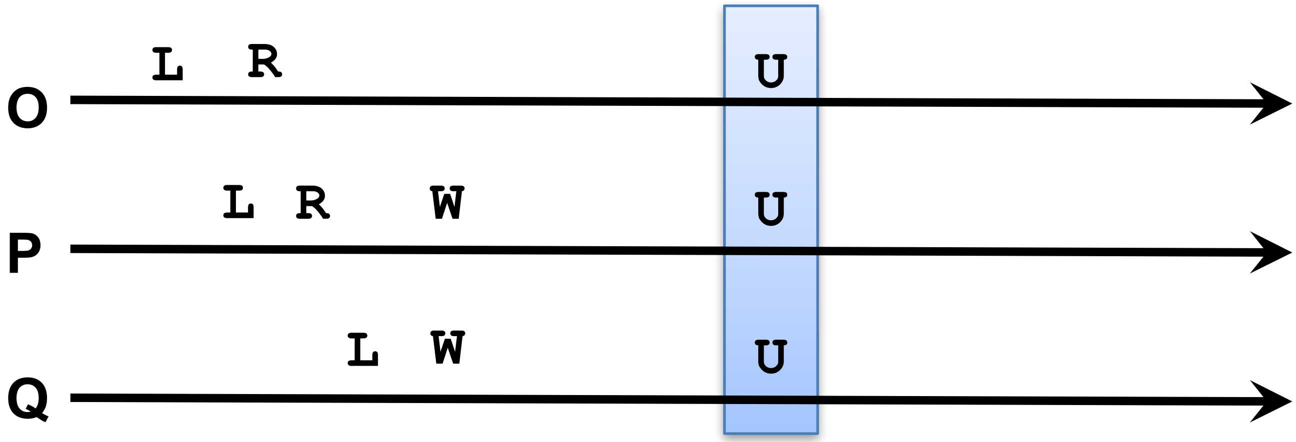
- **Atomicity:** write-ahead logs and checkpoints
- **Consistency:** application logic
- **Isolation:** concurrency controls (locks, 2PL, OCC, MVCC)
- **Durability:** write-ahead logs and checkpoints

Concurrency controls

- **Global lock**: simple, but slow
- **Per-object lock**: doesn’t guarantee serializability (isolation)
- **2PL**: gives serializability, but leaves opportunities on the table and can deadlock
- **OCC**: performs well if few conflicts, but poorly if many conflicts
- **MVCC**: snapshot isolation, not serializability

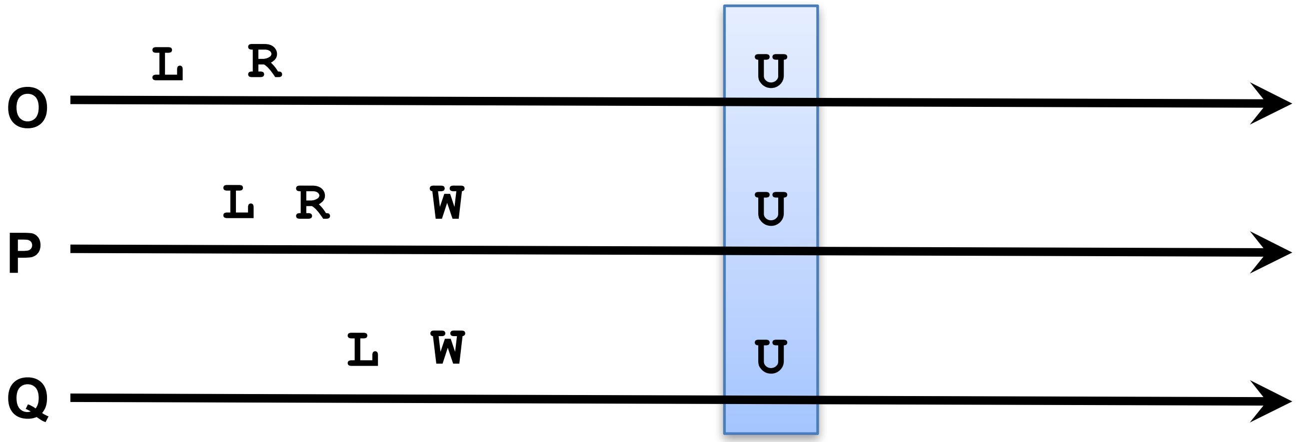
Distributed Transactions

Consider partitioned data over servers



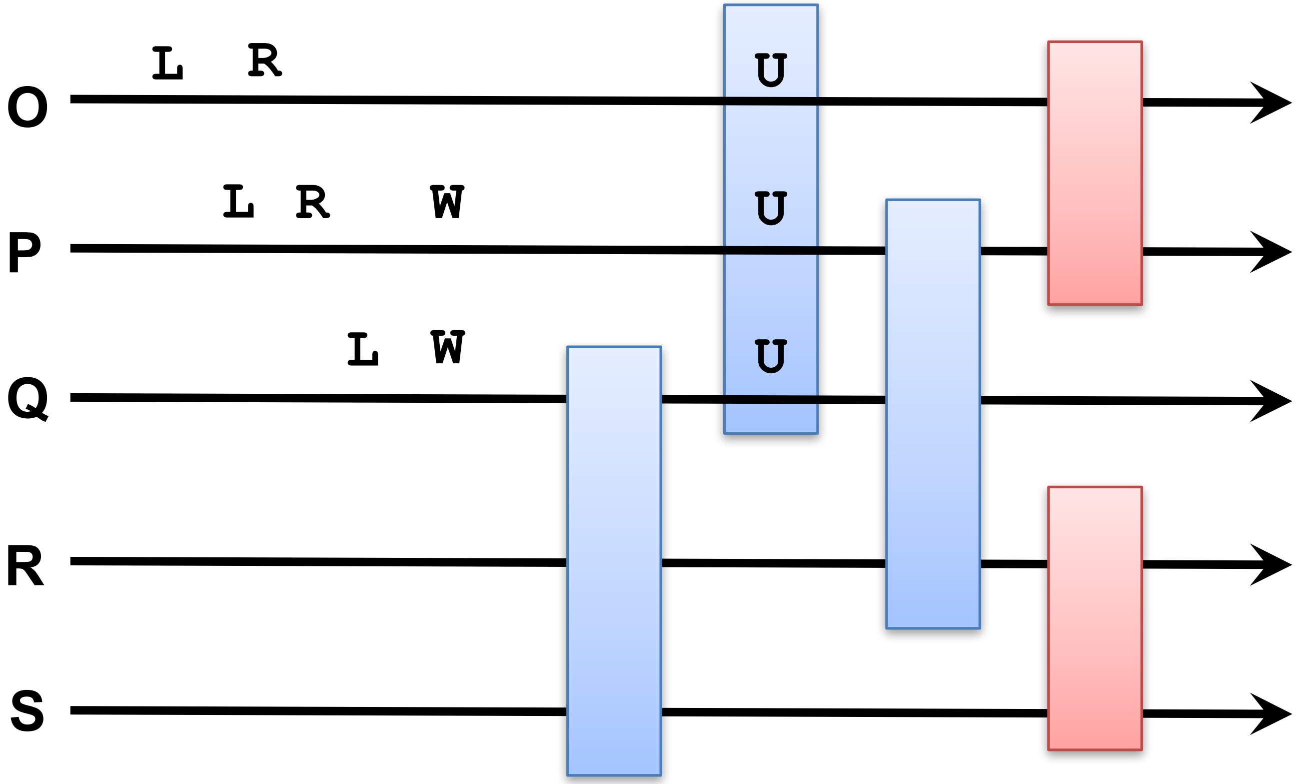
- Why not just use 2PL?
  - Grab locks over entire read and write set
  - Perform writes
  - Release locks (at commit time)

Consider partitioned data over servers



- How do you get serializability?
  - – On single machine, single COMMIT op in the WAL
  - – In distributed setting, assign global timestamp to txn (at sometime after lock acquisition and before commit)
    - Centralized txn manager
    - Distributed consensus on timestamp (not all ops)

Strawman: Consensus per txn group?



- Single Lamport clock, consensus per group?
  - Linearizability composes!
  - But doesn't solve concurrent, non-overlapping txn problem

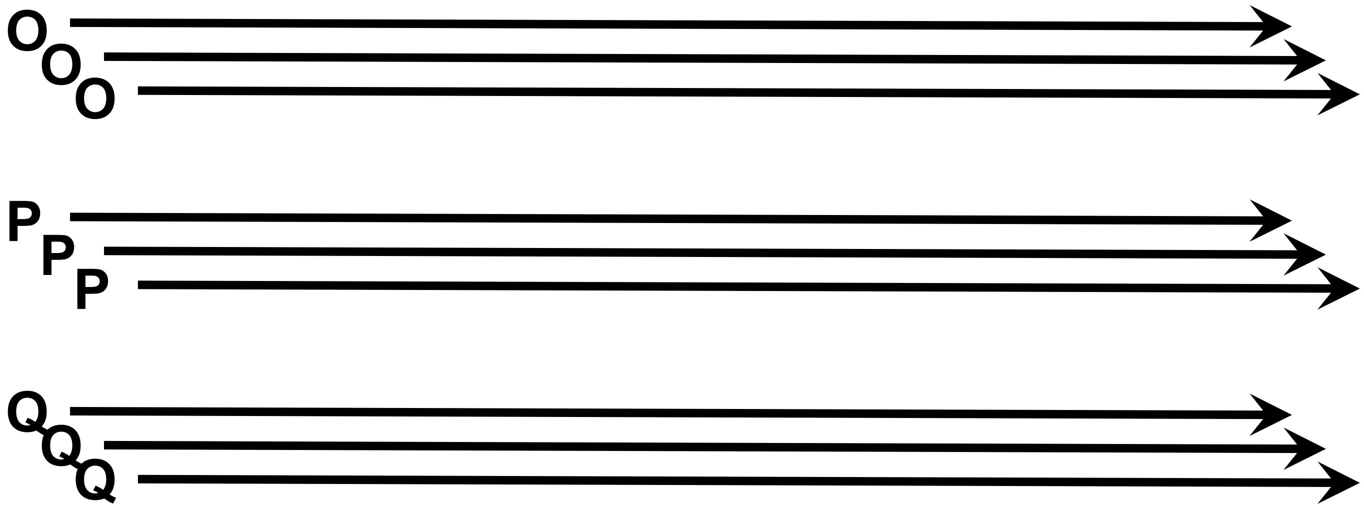
Spanner: Google’s Globally-Distributed Database

OSDI 2012

Google's Setting

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

Scale-out vs. fault tolerance



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

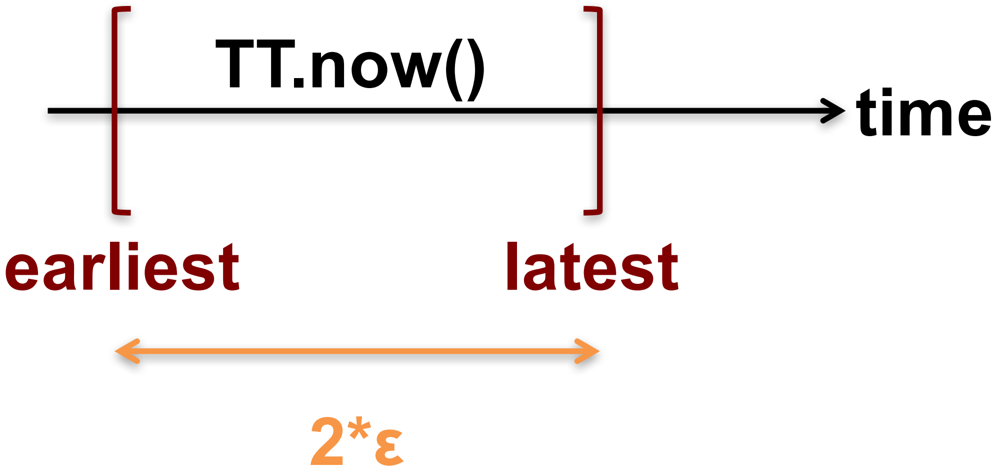
Disruptive idea:

Do clocks **really** need to be arbitrarily unsynchronized?

Can you engineer some max divergence?

TrueTime

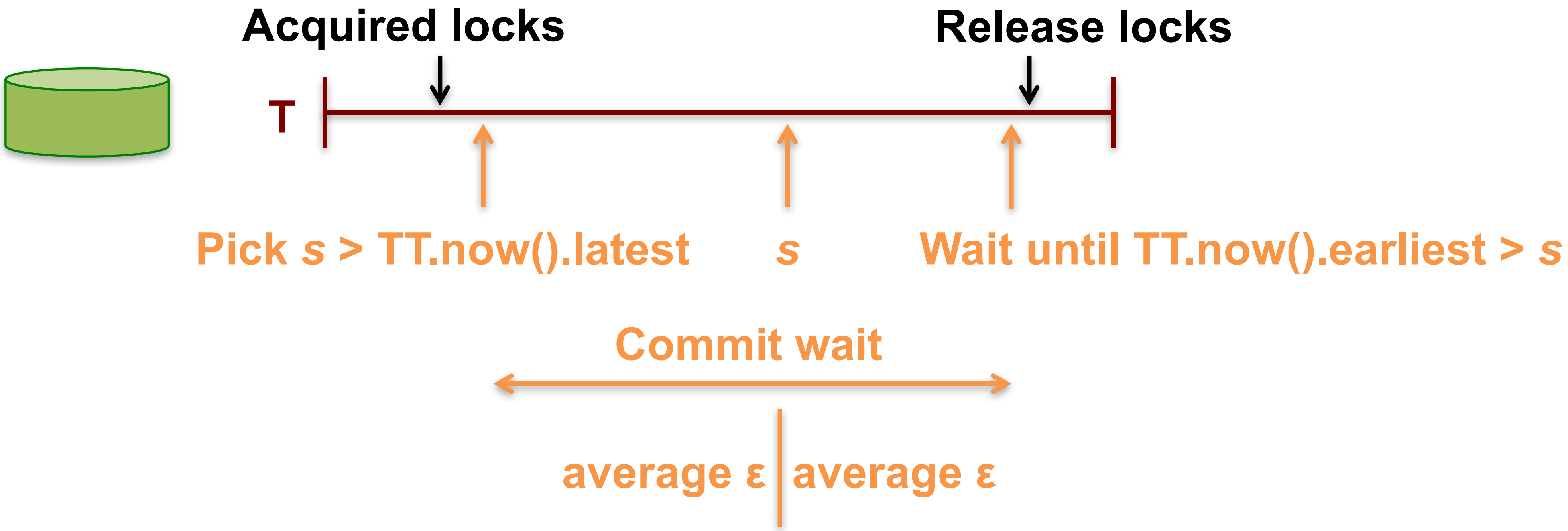
- “Global wall-clock time” with bounded uncertainty



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

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

Timestamps and TrueTime



Acquired locks

Release locks

T

Pick *s* > TT.now().latest

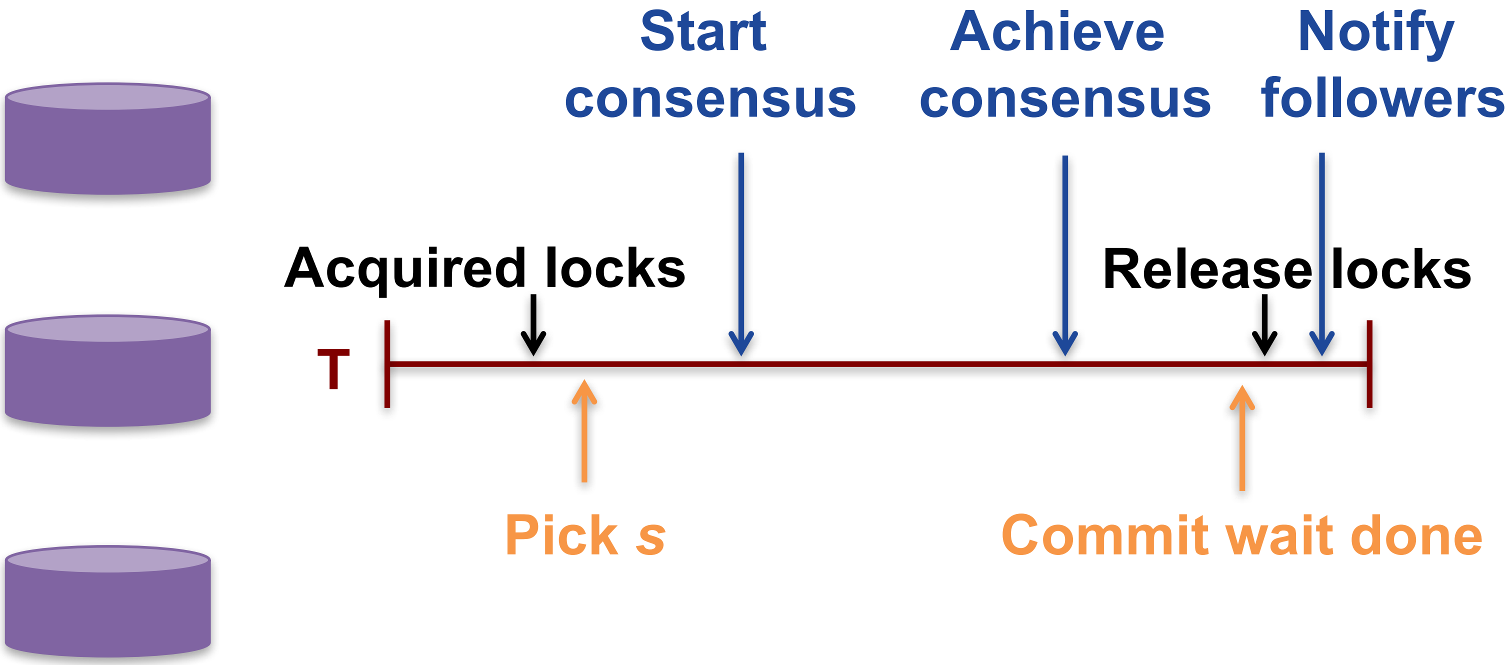
s

Wait until TT.now().earliest > *s*

Commit wait

average ε | average ε

Commit Wait and Replication



Client-driven transactions

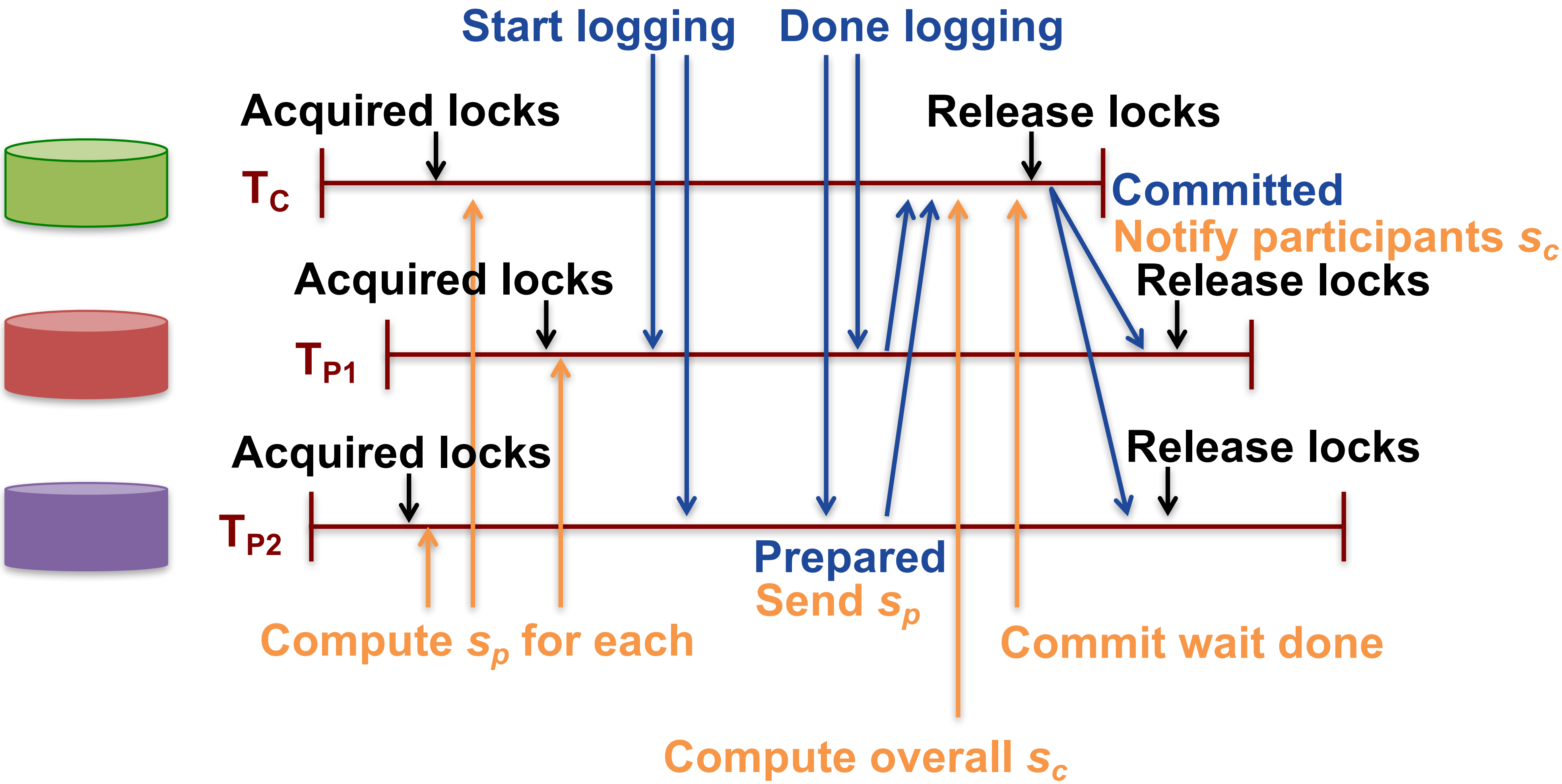
Client:

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

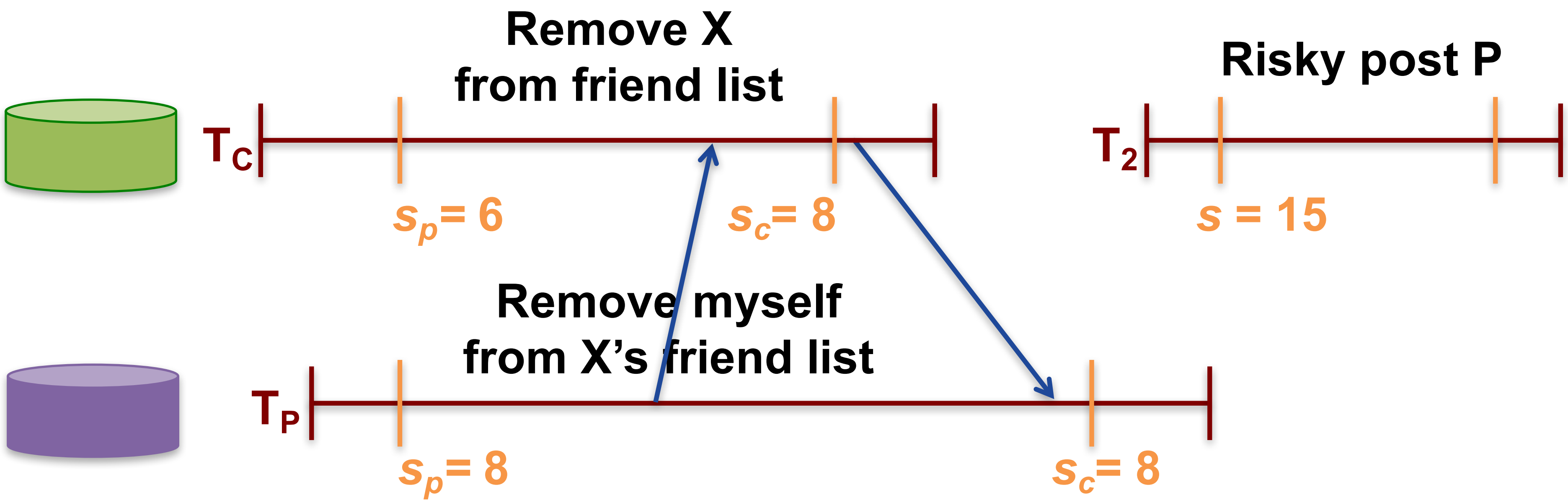
Commit Wait and 2-Phase Commit

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

Commit Wait and 2-Phase Commit



Example



	Time	<8	8	15
🟩	My friends	[X]	[]	
🟩	My posts			[P]
🟪	X’s friends	[me]	[]	

Read-only optimizations

- Given global timestamp, can implement read-only transactions lock-free (snapshot isolation)

- Step 1: Choose timestamp s_{read} = TT.now.latest()

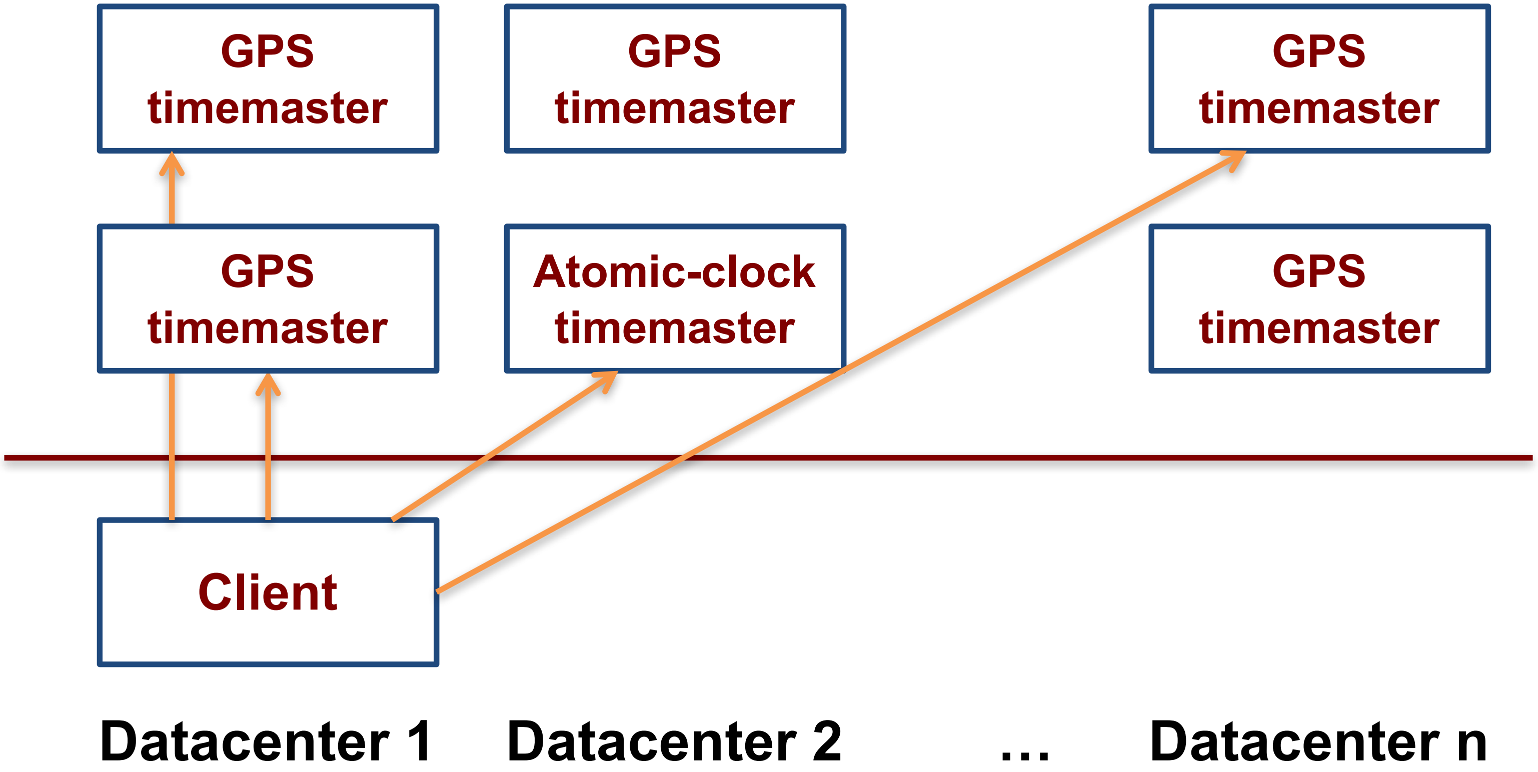
- Step 2: Snapshot read (at s_{read}) to each tablet
  - Can be served by any up-to-date replica

Disruptive idea:

Do clocks **really** need to be arbitrarily unsynchronized?

Can you engineer some max divergence?

TrueTime Architecture



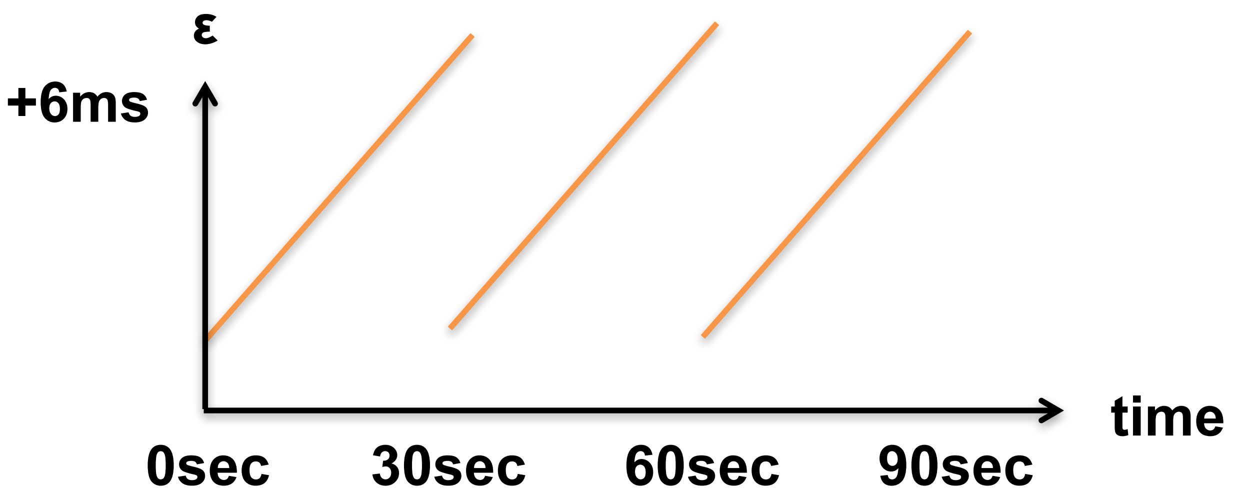
Compute reference [earliest, latest] = now ± ε

TrueTime implementation

now = reference now + local-clock offset

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

= 1ms + 200 μs/sec



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

Known unknowns > unknown unknowns

Rethink algorithms to reason about uncertainty

Pro tips

The single greatest source of head- and heartache

Not following Figure 2 (and, more generally, the paper) **exactly**

Ex. 1: heartbeat RPCs

- These are just empty AppendEntries RPCs!
- That means you **must** handle all the same checks as you would for AppendEntries
- Otherwise, bad things can happen
- If just return true, leader thinks that follower’s log matches the leader’s log up through prevLogIndex

Ex. 2: handling conflicts

- If the follower's log isn't as long as the leaders, conflict!
- Can't just truncate follower's log after prevLogIndex. Only do so if an existing entry conflicts with the leader's
- If all entries match, follower must keep any additional log entries it has. Why?

Ex. 3: reset timers precisely!

- There are only three scenarios
  - Receive AppendEntries RPC from **current** leader
  - Start election
  - Grant vote to another peer

- Tempting to reset timers everywhere; why not?

Ex. 4: (re)start elections

- We must start a new election if our election timer fires, **even if** we were already a candidate in the middle of an election

- What can happen if we don’t?

Ex. 5: abdicating the throne

- No matter what happens, if we receive a request with a higher term, convert to follower and update currentTerm

- Don’t forget to also change votedFor!

Ex. 6: when and when not to be lazy

- When checking whether a log is up to date, follow section 5.4! Checking length is insufficient

- If a step says ‘reply false’, return immediately and don’t execute subsequent steps

Ex. 7: applying log entries

- If the commitIndex (index of highest log entry known to be committed) is ever greater than lastApplied (index of highest log entry applied to state machine), apply!

- Don’t need to do right away, but should have dedicated way of handling so we don’t have multiple channels trying to apply the same entry

- P.S. – don’t forget to check commitIndex > lastApplied…

Ex. 8: matchIndex vs. nextIndex

- nextIndex is optimistic: assume that follower has all entries from previous interaction unless we received a negative response

- matchIndex is conservative: only update when we know a higher index log entry has been known to be replicated

Ex. 9: ignore RPCs from old terms!

- Yeah, don’t forget that

Monday lecture

Conflicting/concurrent writes in eventual/causal systems:

OT + CRDTs

(aka how Google Docs works)