Concurrency Control II

and Distributed Transactions

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 18

Marco Canini

Credits: Michael Freedman and Kyle Jamieson developed much of the original material.

Serializability

Execution of a set of transactions over multiple items is equivalent to *some* serial execution of txns

Lock-based concurrency control

- **Big Global Lock:** Results in a **serial** transaction schedule at the cost of performance

- **Two-phase locking with finer-grain locks:**
  - **Growing phase** when txn acquires locks
  - **Shrinking phase** when txn releases locks (typically commit)
  - Allows txn to execute concurrently, improving performance

Q: What if access patterns rarely, if ever, conflict?

Be optimistic!

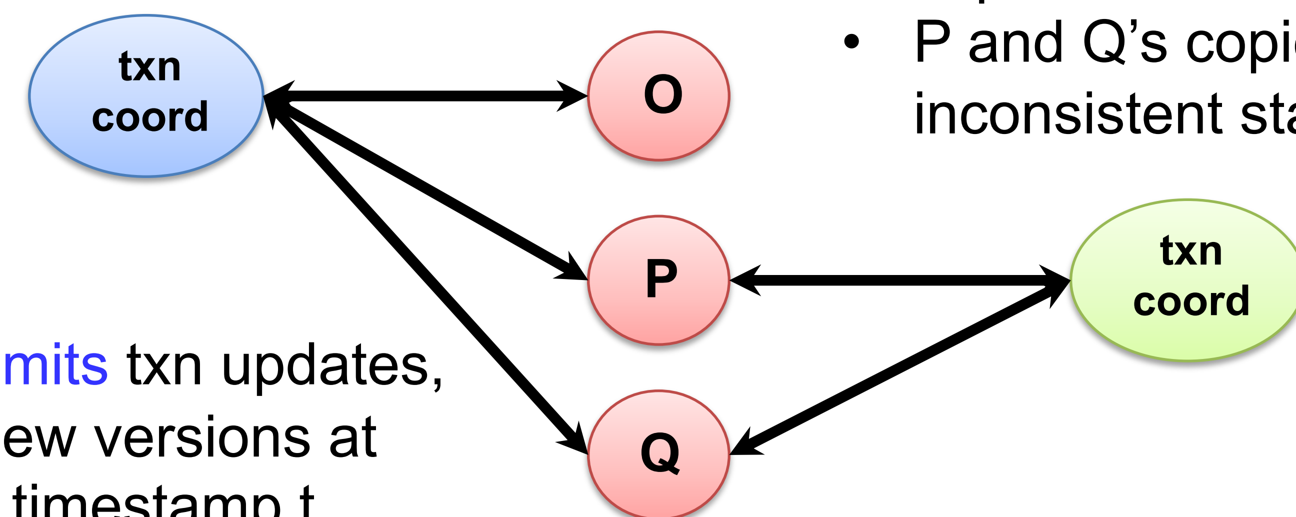
- Goal: Low overhead for non-conflicting txns
- Assume success!
  - Process transaction as if it would succeed
  - Check for serializability only at commit time
  - If fails, abort transaction
- Optimistic Concurrency Control (OCC)
  - Higher performance when few conflicts vs. locking
  - Lower performance when many conflicts vs. locking

OCC: Three-phase approach

- **Begin:** Record timestamp marking the transaction's beginning
- **Modify** phase
  - Txn can read values of committed data items
  - Updates only to local copies (versions) of items (in DB cache)
- **Validate** phase
- **Commit** phase
  - If validates, transaction's updates applied to DB
  - Otherwise, transaction restarted
  - Care must be taken to avoid "TOCTTOU" issues

OCC: Why validation is necessary

- New txn creates shadow copies of P and Q
- P and Q's copies at inconsistent state



When commits txn updates, create new versions at some timestamp t

OCC: Validate Phase

- Transaction is about to commit. System must ensure:
  - Initial consistency: Versions of accessed objects at start consistent
  - No conflicting concurrency: No other txn has committed an operation at object that conflicts with one of this txn's invocations

- Consider transaction 1. For all other txns N either committed or in validation phase, one of the following holds:
  A. N completes commit before 1 starts modify
  B. 1 starts commit after N completes commit, and ReadSet 1 and WriteSet N are disjoint
  C. Both ReadSet 1 and WriteSet 1 are disjoint from WriteSet N, and N completes modify phase.

- When validating 1, first check (A), then (B), then (C). If all fail, validation fails and 1 aborted.

2PL & OCC = strict serialization

- Provides semantics as if only one transaction was running on DB at time, in serial order

  + Real-time guarantees

- 2PL: Pessimistically get all the locks first

- OCC: Optimistically create copies, but then recheck all read + written items before commit

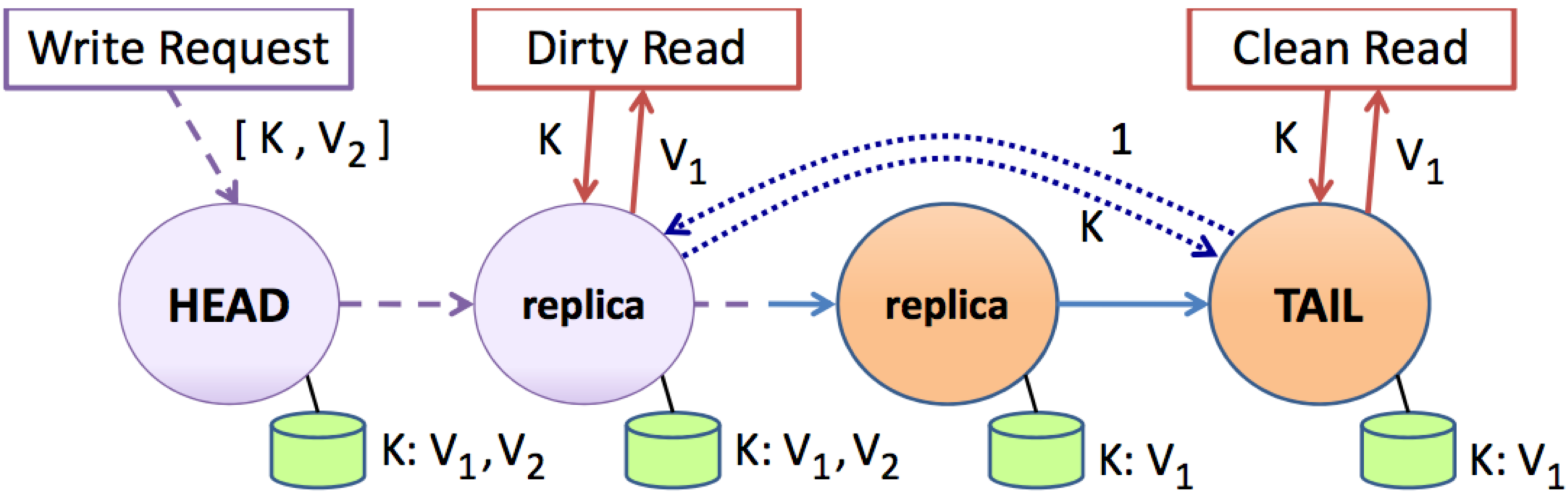
Multi-version concurrency control

Generalize use of multiple versions of objects

Multi-version concurrency control

- Maintain multiple versions of objects, each with own timestamp. Allocate correct version to reads.

- Prior example of MVCC:



Multi-version concurrency control

- Maintain multiple versions of objects, each with own timestamp. Allocate correct version to reads.
- Unlike 2PL/OCC, reads never rejected
- Occasionally run garbage collection to clean up

MVCC Intuition

- Split transaction into read set and write set
  - All reads execute as if one “snapshot”
  - All writes execute as if one later “snapshot”

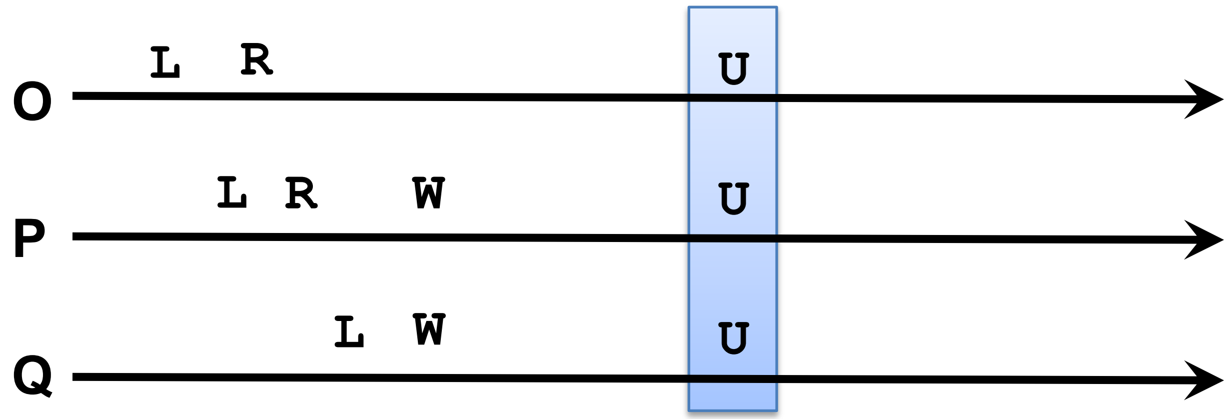
- Yields snapshot isolation < serializability

Serializability vs. Snapshot isolation

- Intuition: Bag of marbles: ½ white, ½ black
- Transactions:
  - T1: Change all white marbles to black marbles
  - T2: Change all black marbles to white marbles
- Serializability (2PL, OCC)
  - T1 → T2 or T2 → T1
  - In either case, bag is either ALL white or ALL black
- Snapshot isolation (MVCC)
  - T1 → T2 or T2 → T1 or T1 || T2
  - Bag is ALL white, ALL black, or ½ white ½ black

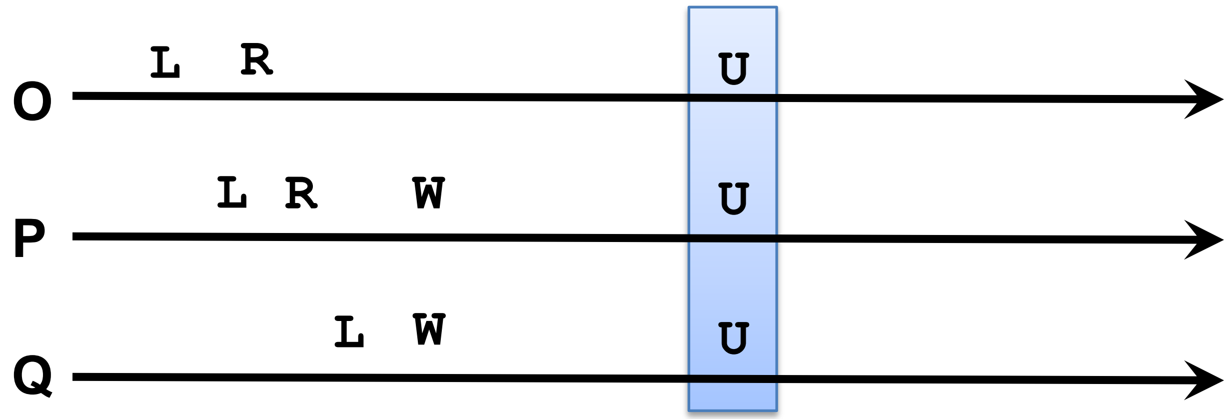
Distributed Transactions

Consider partitioned data over servers



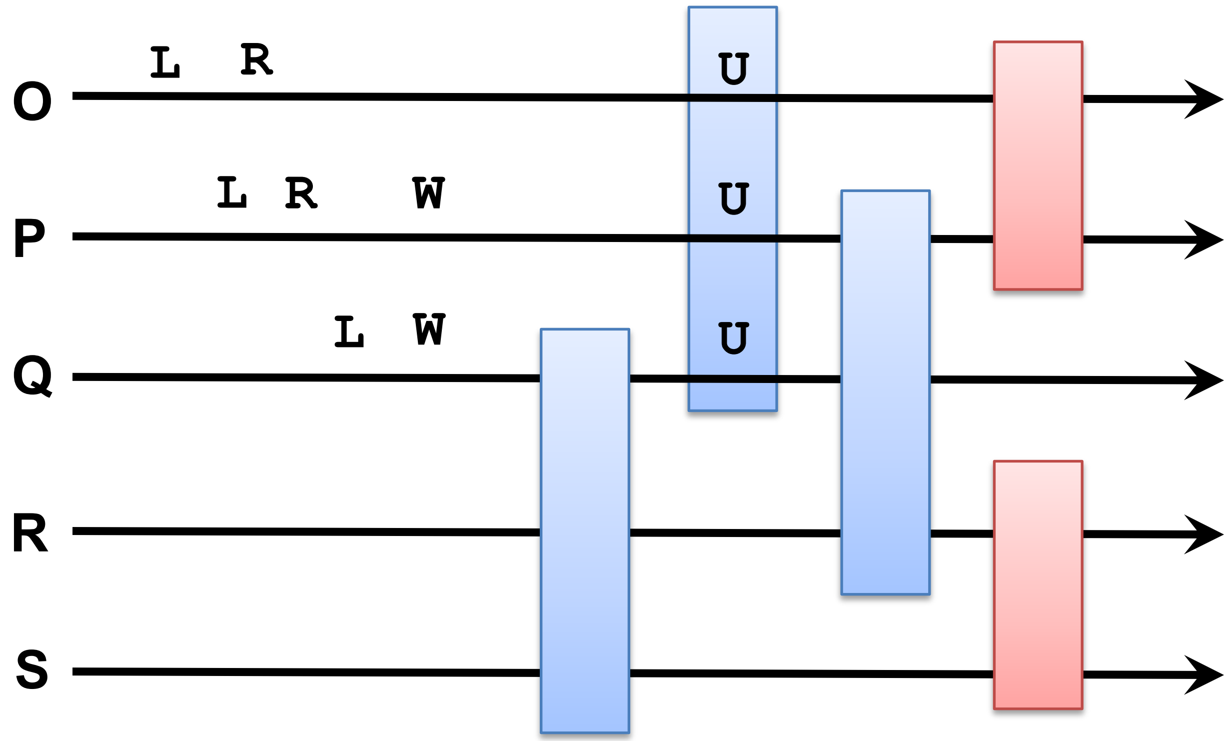
- Why not just use 2PL?
  - Grab locks over entire read and write set
  - Perform writes
  - Release locks (at commit time)

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

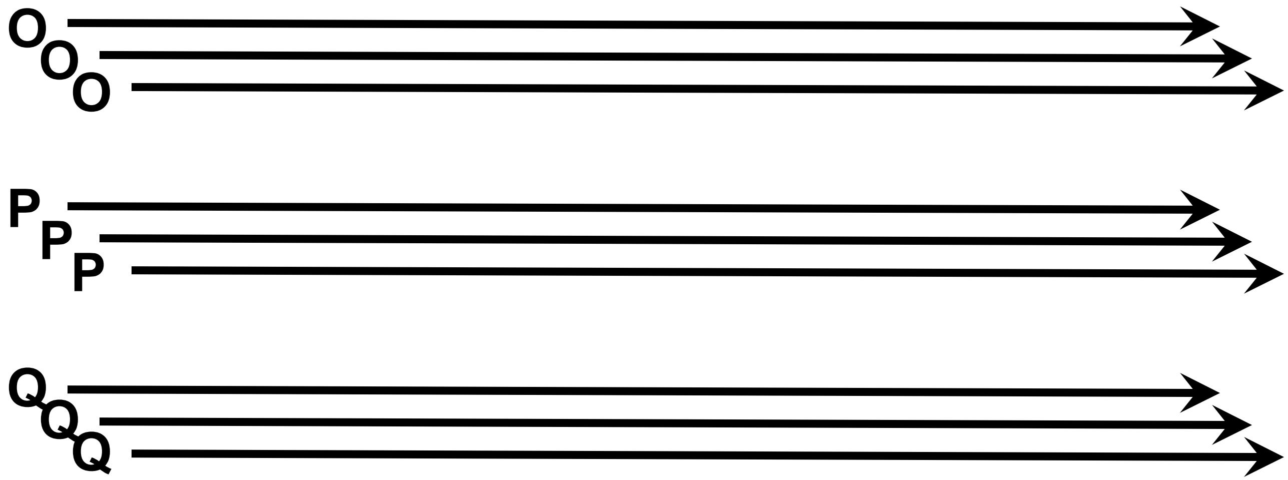
Spanner: Google’s Globally-Distributed Database

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!

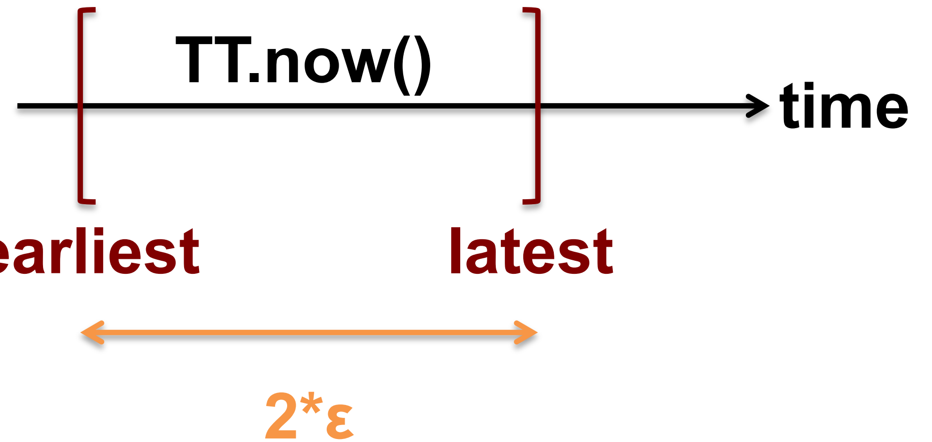
Disruptive idea:

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

Can you engineer some max divergence?

TrueTime

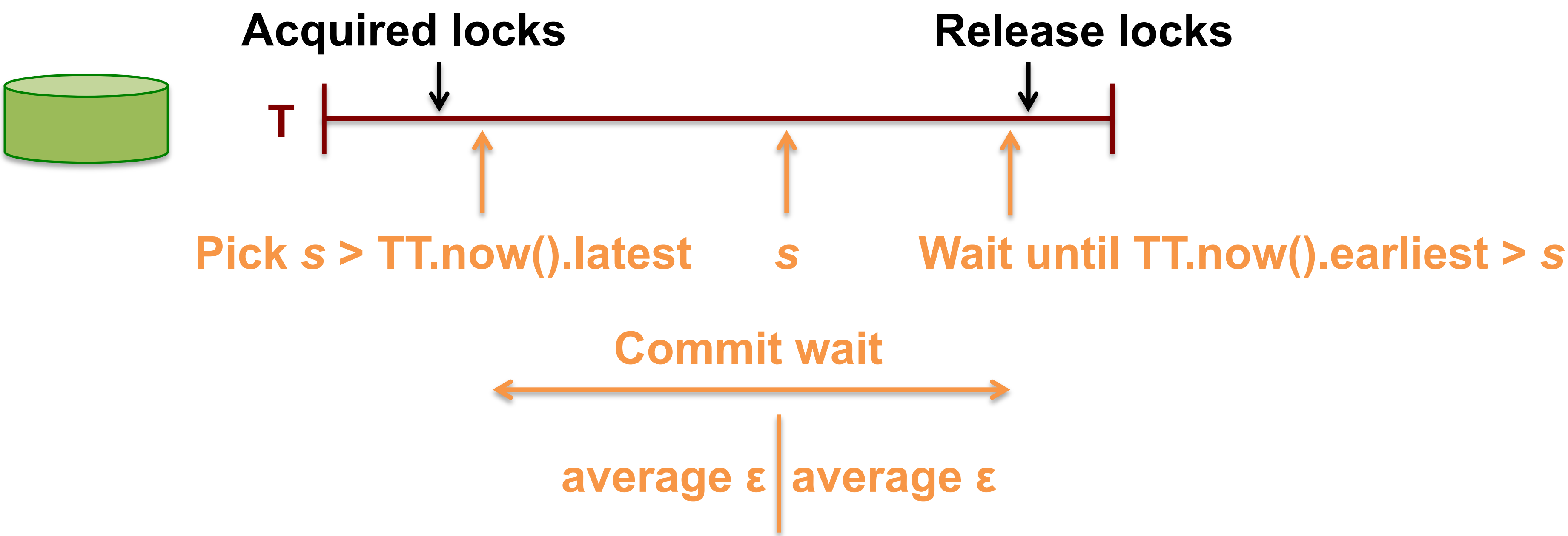
- “Global wall-clock time” with bounded uncertainty



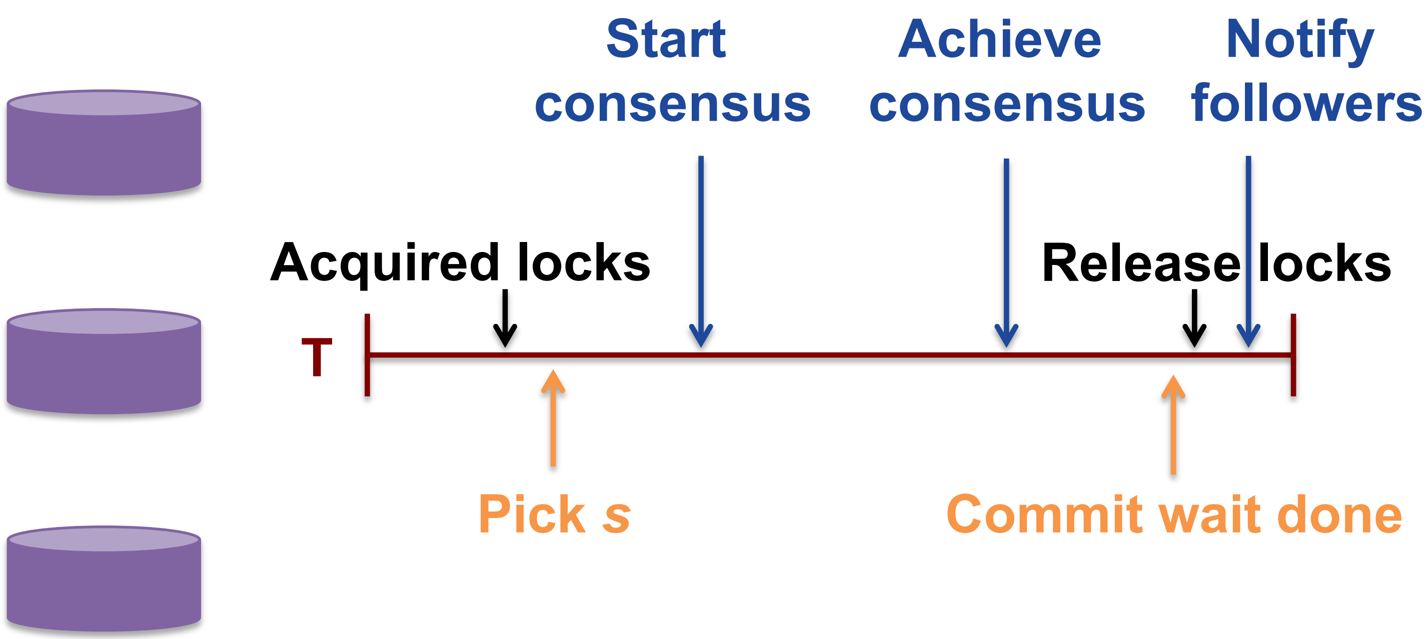
Consider event e_{now} which invoked tt = TT.new():

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

Timestamps and TrueTime



Commit Wait and Replication



Client-driven transactions

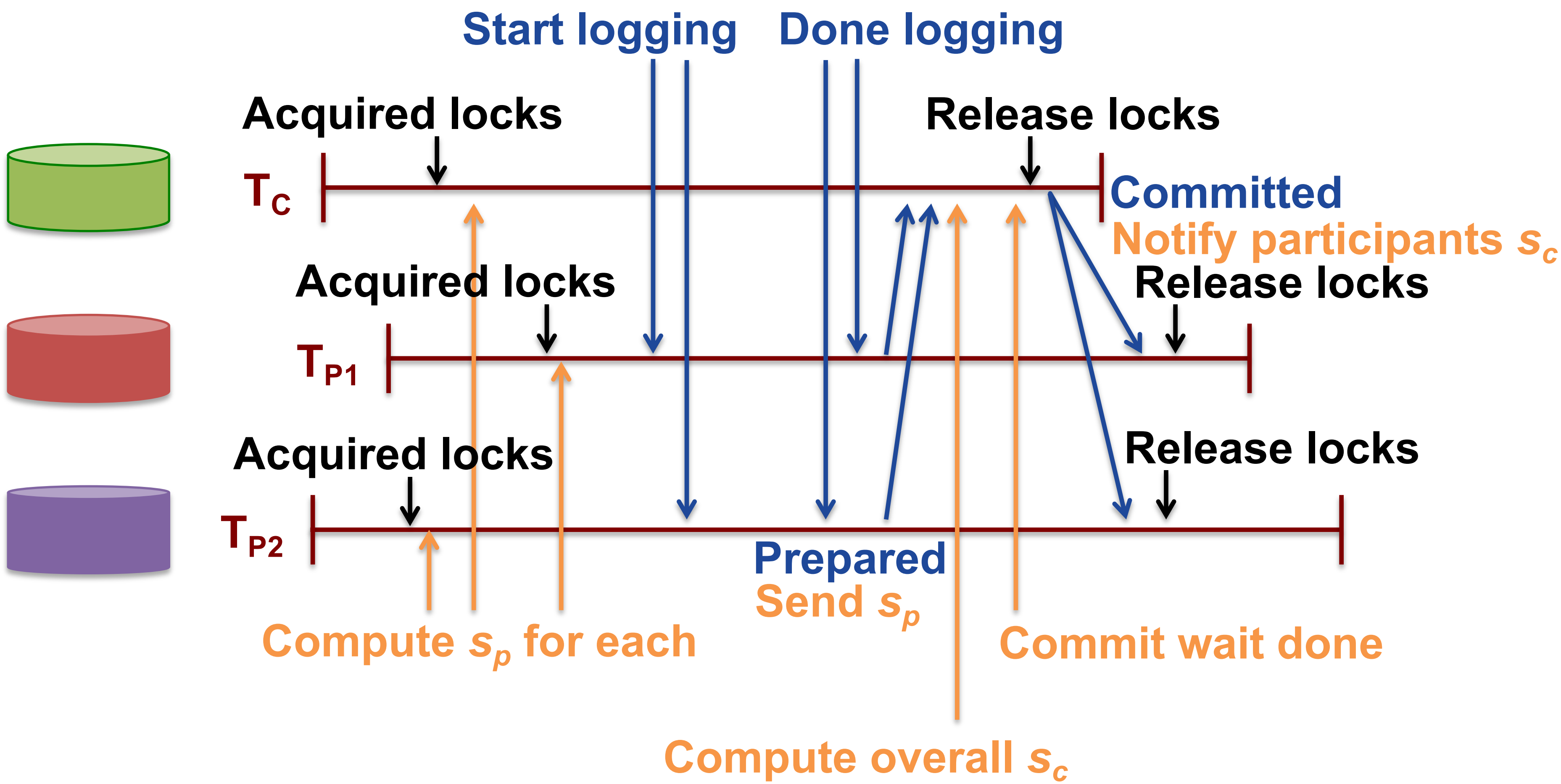
Client:

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

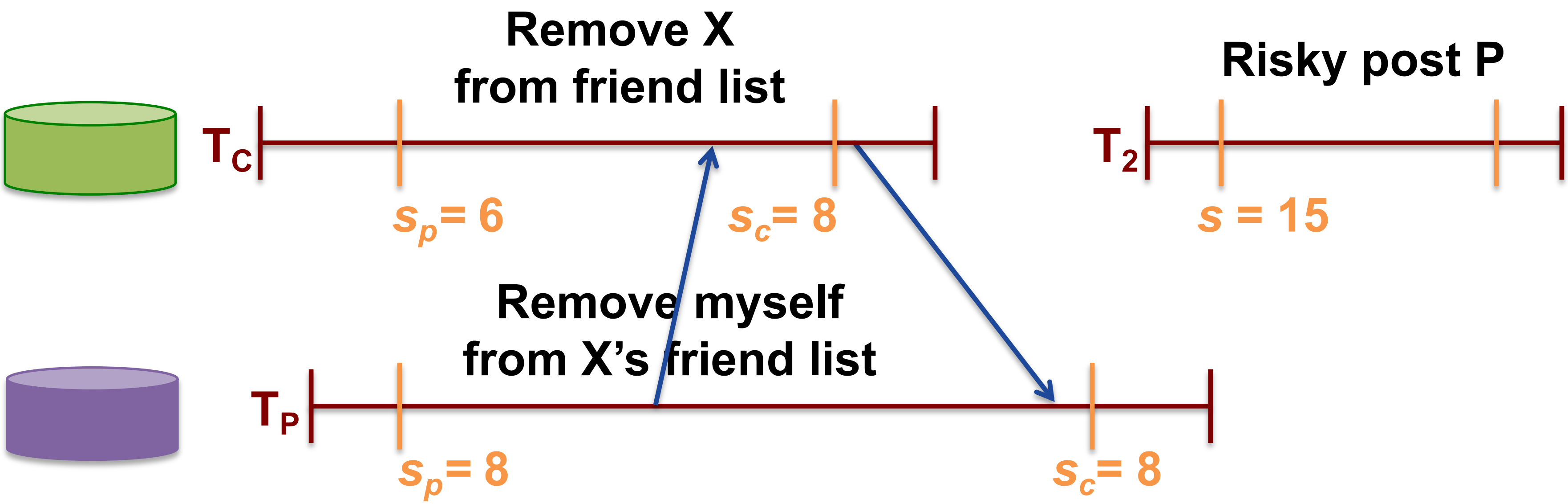
Commit Wait and 2-Phase Commit

- 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

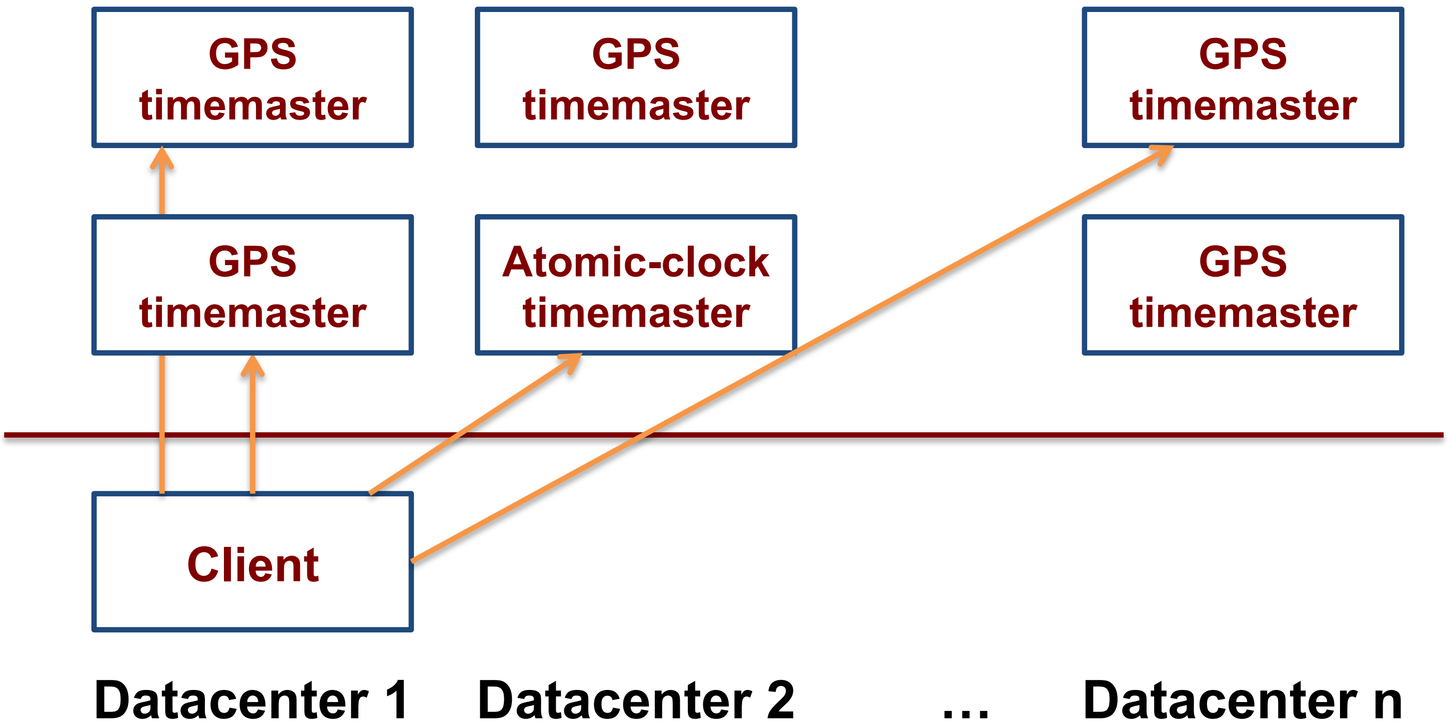
- Given global timestamp, can implement read-only transactions lock-free (snapshot isolation)
- Step 1: Choose timestamp s_{read} = TT.now.latest()
- Step 2: Snapshot read (at s_{read}) to each tablet
  - Can be served by any up-to-date replica

Disruptive idea:

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

Can you engineer some max divergence?

TrueTime Architecture



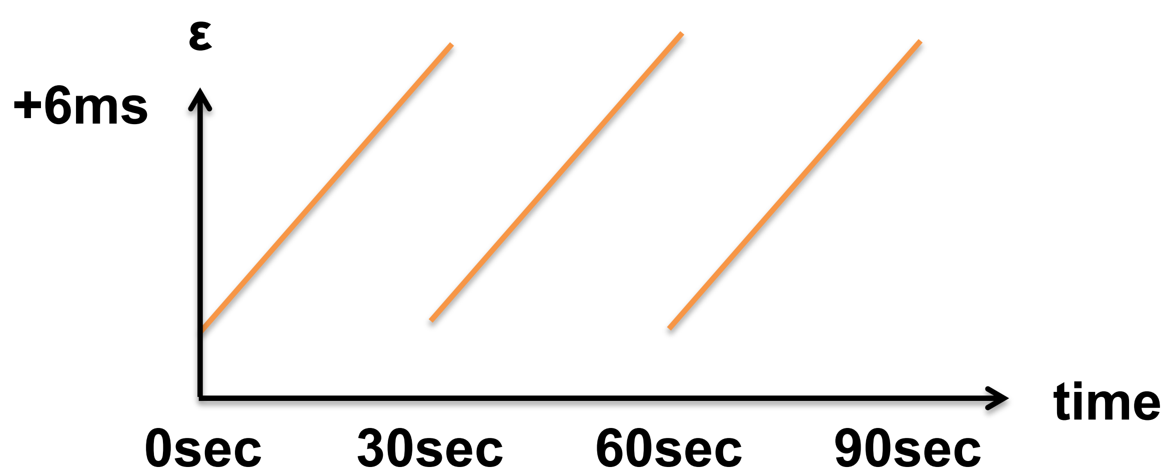
Compute reference [earliest, latest] = now ± ε

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

Next topic:
Virtualization and Cloud Computing