Spanner

COS 418: Distributed Systems
Lecture 16

Wyatt Lloyd

Slides adapted from Mike Freedman's, which are adapted from the Spanner OSDI talk

Why Google Built Spanner

2005 – BigTable [OSDI 2006]

- Eventually consistent across datacenters
- Lesson: “don’t need distributed transactions”

2008? – MegaStore [CIDR 2011]

- Strongly consistent across datacenters
- Option for distributed transactions
  - Performance was not great…

2011 – Spanner [OSDI 2012]

- Strictly Serializable Distributed Transactions
- “We wanted to make it easy for developers to build their applications”

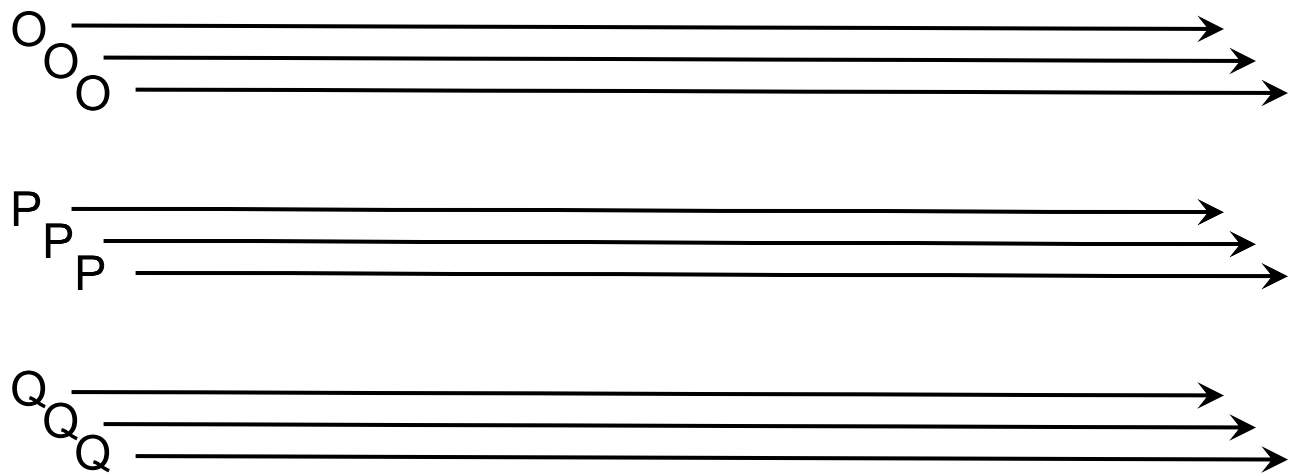
Spanner: Google's Globally-Distributed Database

OSDI 2012

Google’s Setting

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

Scale-out vs. Fault Tolerance



- Every shard replicated via MultiPaxos
- So every "operation" within transactions across tablets actually a replicated operation within Paxos RSM
- Paxos groups can stretch across datacenters!

Read-Only Transactions

- Transactions that only read data
  - Predeclared, i.e., developer uses READ_ONLY flag / interface

- Reads are dominant operations
  - e.g., FB's TAO had 500 reads : 1 write [ATC 2013]
  - e.g., Google Ads (F1) on Spanner from 1? DC in 24h:

    21.5B reads
    31.2M single-shard transactions
    32.1M multi-shard transactions

Make Read-Only Txns Efficient

- Ideal: Read-only transactions that are non-blocking
  - Arrive at shard, read data, send data back
  - Impossible with Strict Serializability
    - SNOW after the break!

- Goal 1: Lock-free read-only transactions

- Goal 2: Non-blocking stale read-only txns

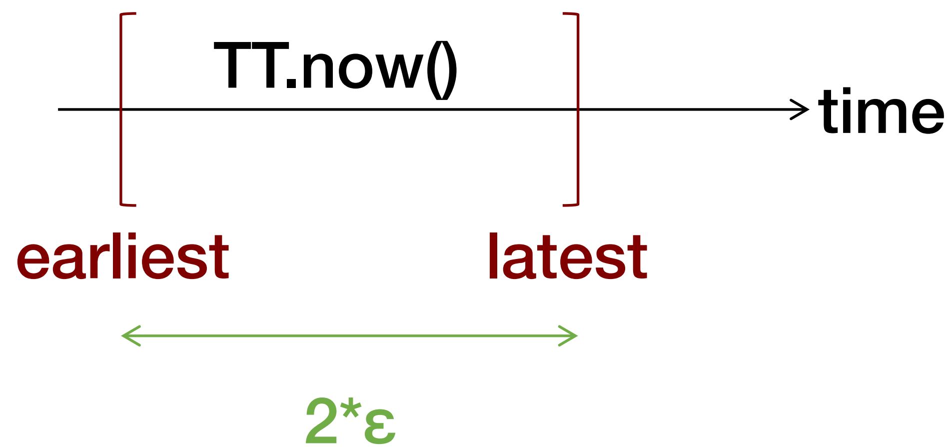
Disruptive idea:

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

Can you engineer some max divergence?

TrueTime

- "Global wall-clock time" with bounded uncertainty
  - ε is worst-case clock divergence
  - Timestamps become intervals, not single values



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

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

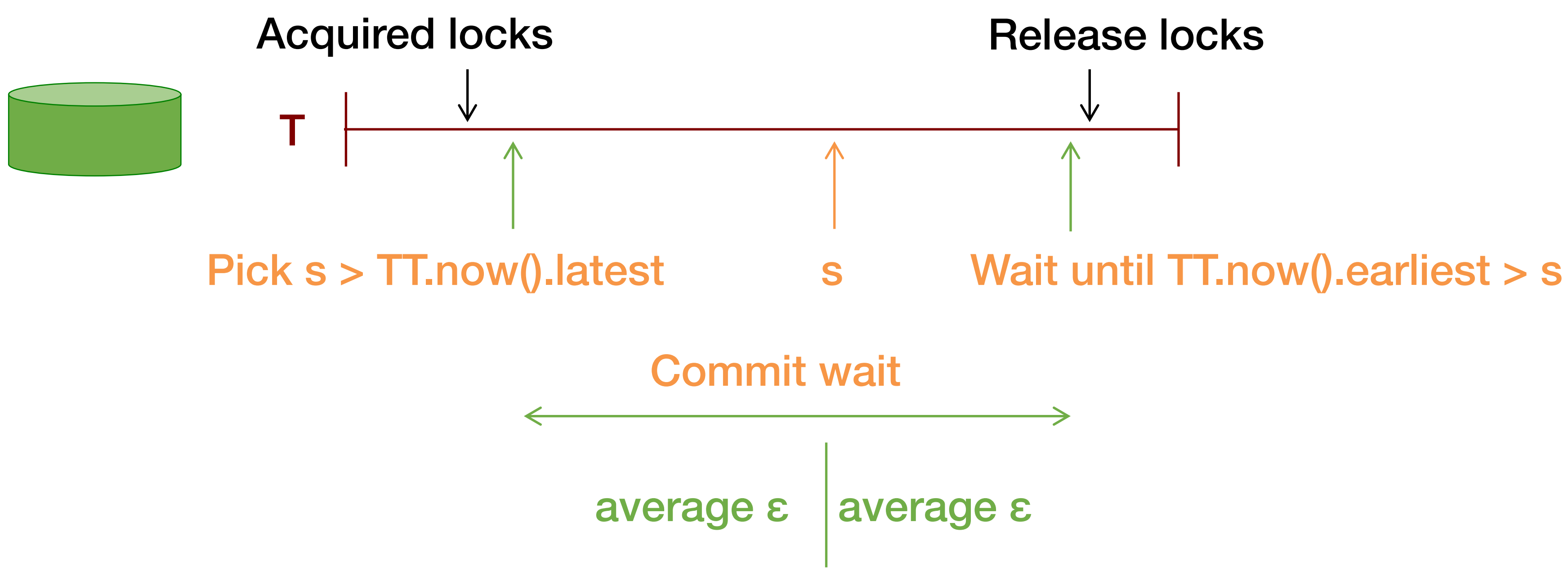
TrueTime for Read-Only Txns

- Assign all transactions a wall-clock commit time (s)
  - All replicas of all shards track how up-to-date they are with t_{safe}: all transactions with $s < t_{safe}$ have committed on this machine

- Goal 1: Lock-free read-only transactions
  - Current time ≤ TT.now.latest()
  - s_{read} = TT.now.latest()
  - wait until $s_{read} < t_{safe}$
  - Read data as of s_{read}

- Goal 2: Non-blocking stale read-only txns
  - Similar to above, except explicitly choose time in the past
  - (Trades away consistency for better perf, e.g., lower latency)

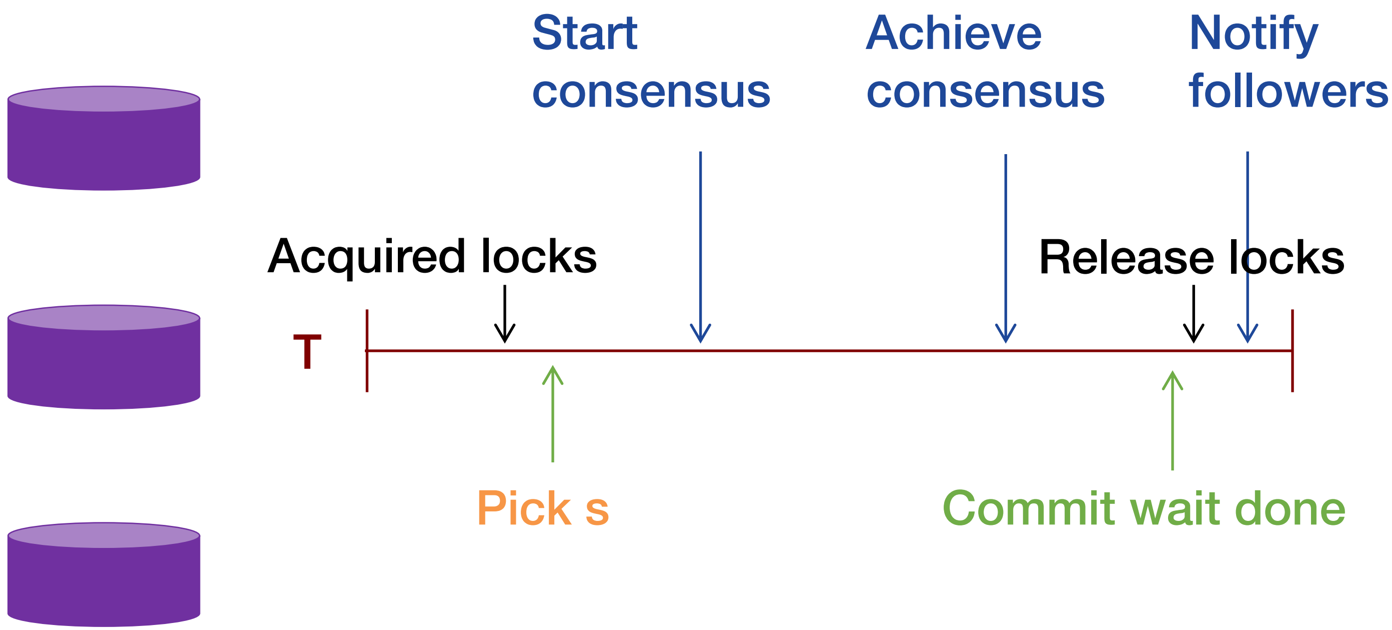
Timestamps and TrueTime



Commit Wait

- Enables efficient read-only transactions
- Cost: 2ε extra latency
- Reduce/eliminate by overlapping with:
  - Replication
  - Two-phase commit

Commit Wait and Replication



Client-Driven Transactions

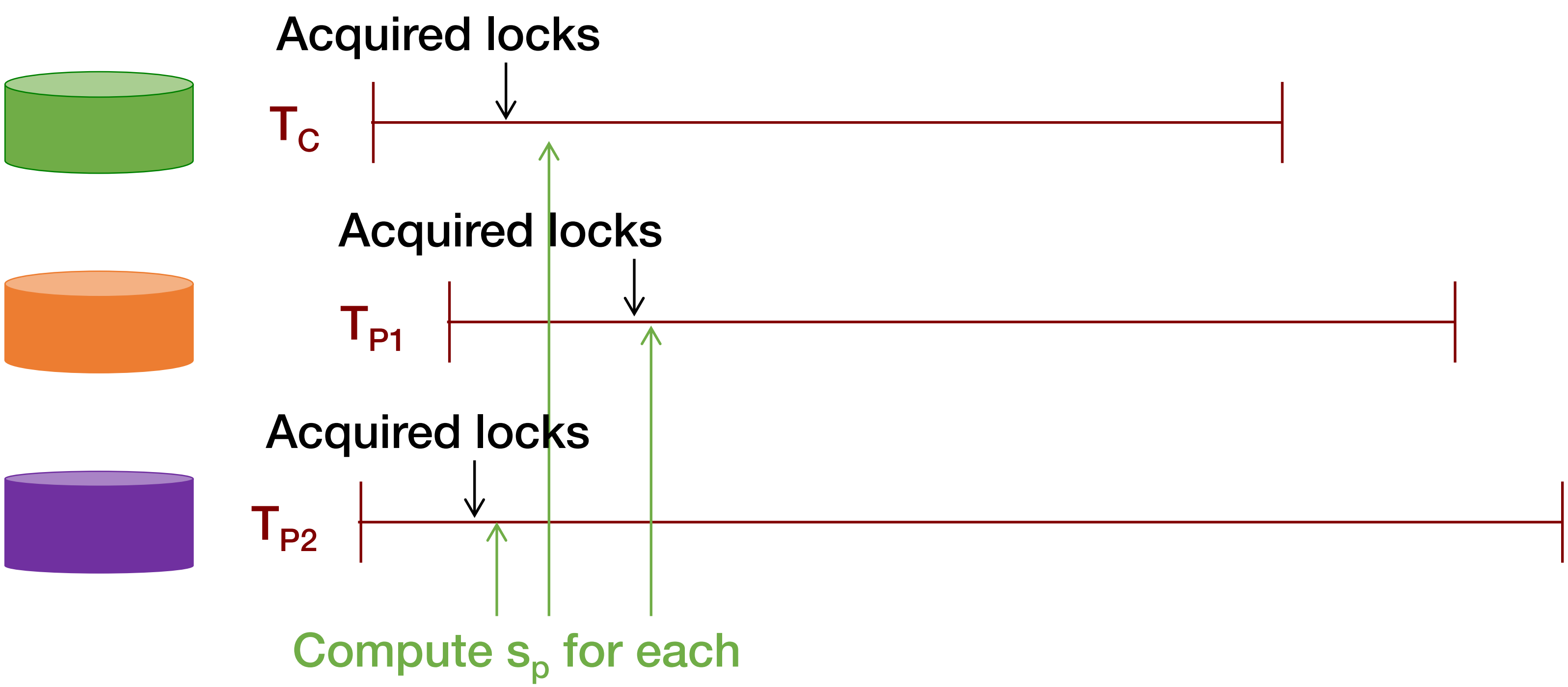
Client: 2PL w/ 2PC

1. Issues reads to leader of each shard 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 2PC

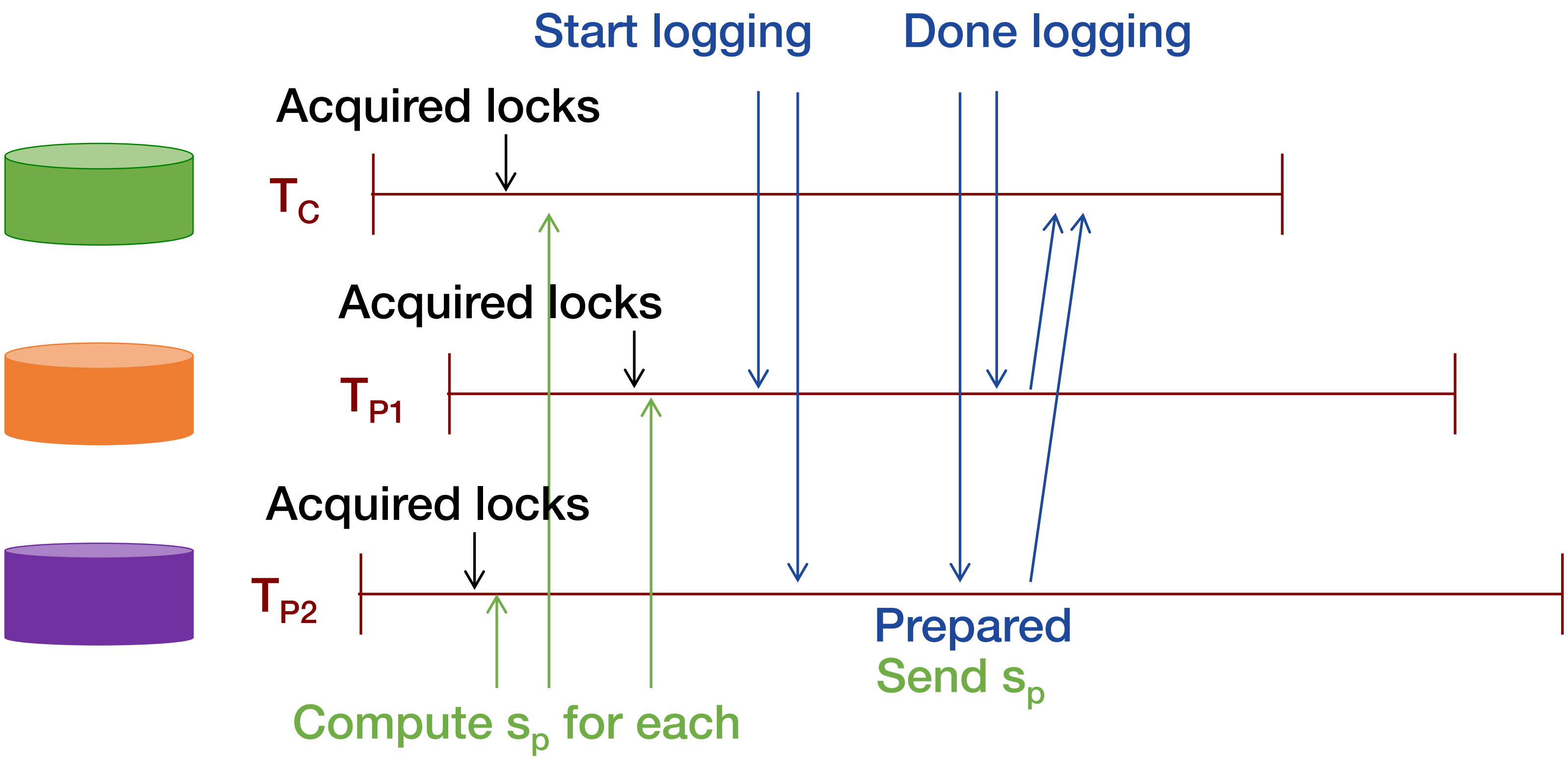
- On commit msg from client, leaders acquire local write locks
  - If non-coordinator:
    - Choose prepare ts > previous local timestamps
    - Log prepare record through Paxos
    - Notify coordinator of prepare timestamp
  - If coordinator:
    - Wait until hear from other participants
    - Choose 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 2PC



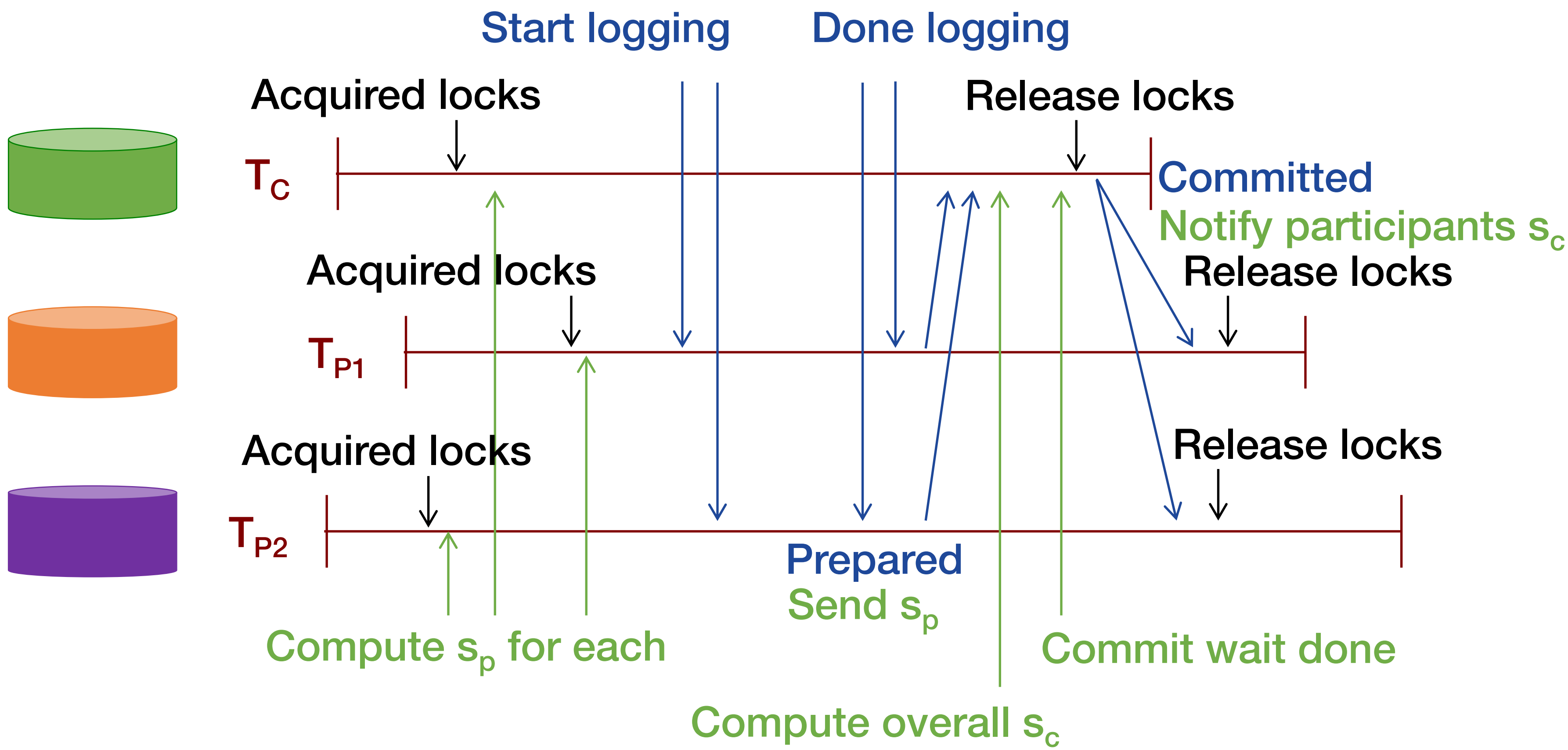
1. Client issues reads to leader of each shard group, which acquires read locks and returns most recent data

Commit Wait and 2PC



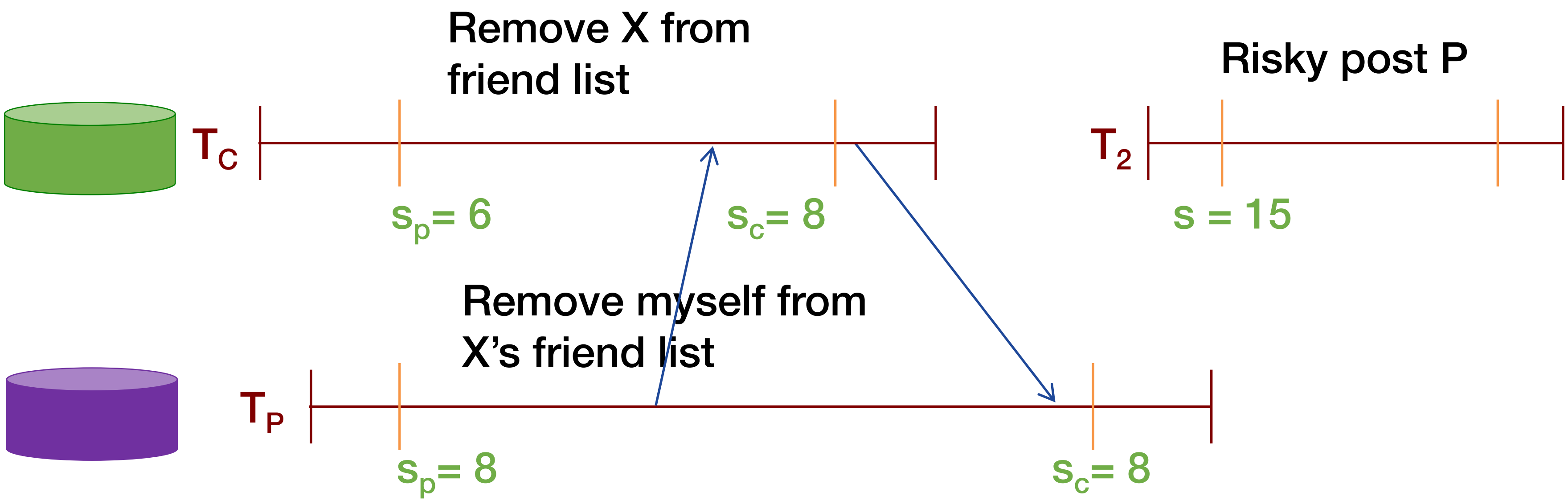
2. Locally performs writes
3. Chooses coordinator from set of leaders, initiates commit
4. Sends commit msg to each leader, incl. identity of coordinator

Commit Wait and 2PC



5. Client waits for commit from coordinator

Example



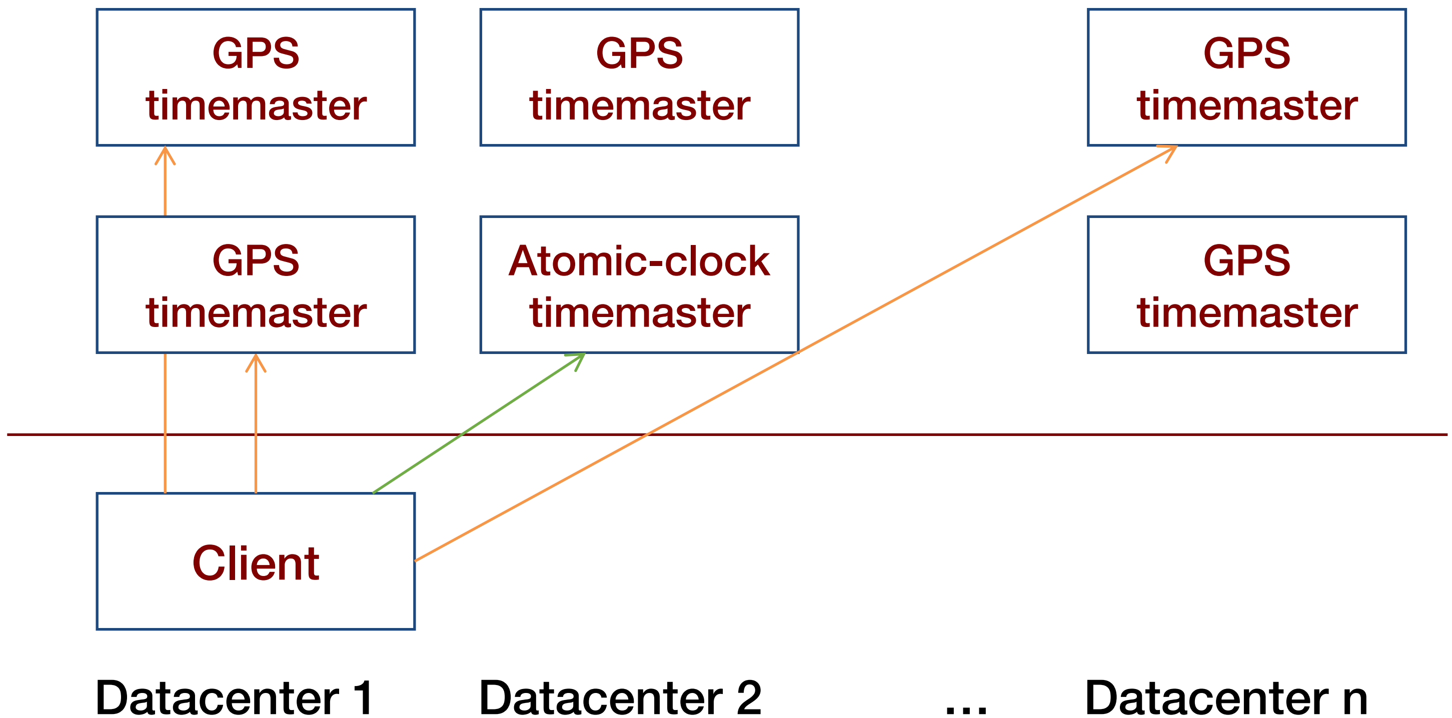
Time	<8	8	15
My friends	[X]	[]	
My posts			[P]
X's friends	[me]	[]	

Disruptive idea:

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

Can you engineer some max divergence?

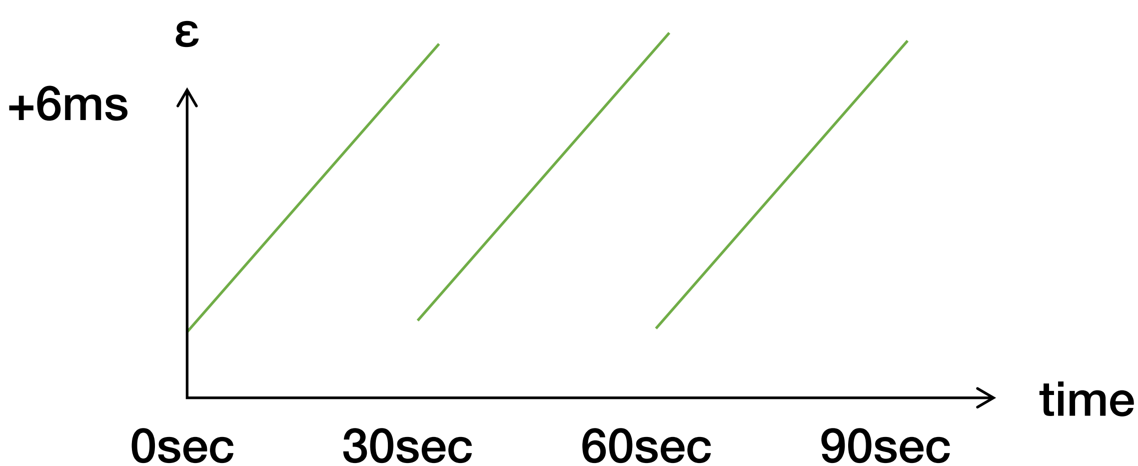
TrueTime Architecture



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

Spanner

- Make it easy for developers to build apps!
- Reads dominant, make them lock-free
- TrueTime exposes clock uncertainty
  - Commit wait ensures transactions end after their commit time
  - Read at TT.now.latest()
- Globally-distributed database
  - 2PL w/ 2PC over Paxos!