Spanner: Google's Globally-Distributed Database

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

ECE1724

Authors:

Many slides adapted from Wyatt Lloyd, Mike Freedman, Spanner OSDI talk

Why Built Spanner?

- BigTable [OSDI 2006]
  - Eventually consistent across datacenters
  - Lesson: Don't need distributed transactions…
- MegaStore [CIDR 2011]
  - Strongly consistent across datacenters
  - Supported distributed transactions, relational model
  - However, performance was not great…
- Spanner [OSDI 2012]
  - Strictly serializable distributed transactions at global scale
  - Goals: Make it easy for developers to build their applications, provide good performance

What is Spanner?

- Spanner is a globally distributed (multi-datacenter) and replicated storage system
- Spanner supports
  - General-purpose transactions with SQL interface
  - Strong consistency (strict serializability)
  - High availability with wide-area replication
- These properties ease app development
  - Behaves like a single-machine database

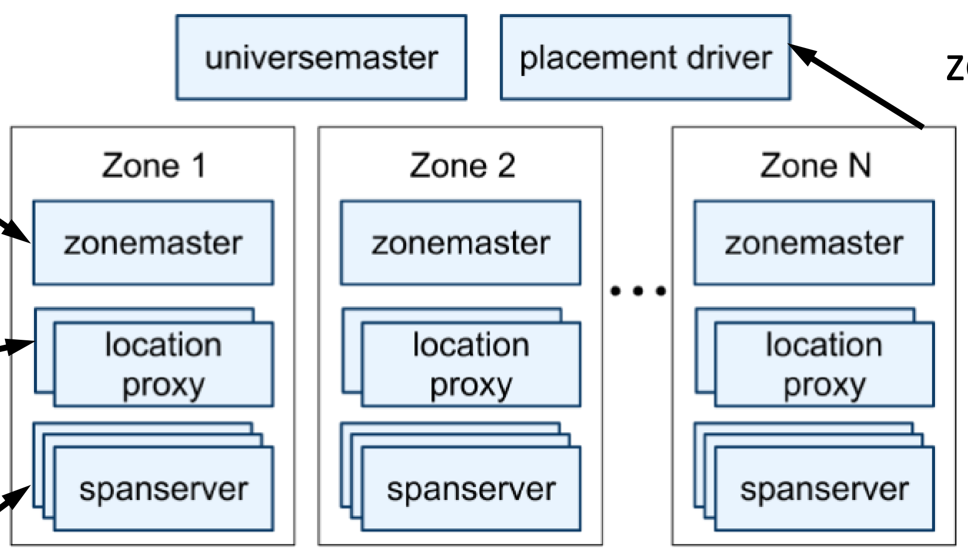
Spanner Architecture

- Spanner is deployed over multiple, geographically-distributed datacenters (zones)
- Each zone has a Bigtable style deployment
  - 100-1000s of servers per zone, 100-1000s of tablets per server

Assigns data to spanservers

Helps clients locate spanservers that serve their data

Serve data to clients



Moves data between zones for load balancing, meeting replication constraints, etc., in minutes timescale.

Figure 1: Spanner server organization.

Spanner Replication

- Each tablet is replicated using state machine replication (Paxos) for fault-tolerance
- Tablet replicas can cross data centers

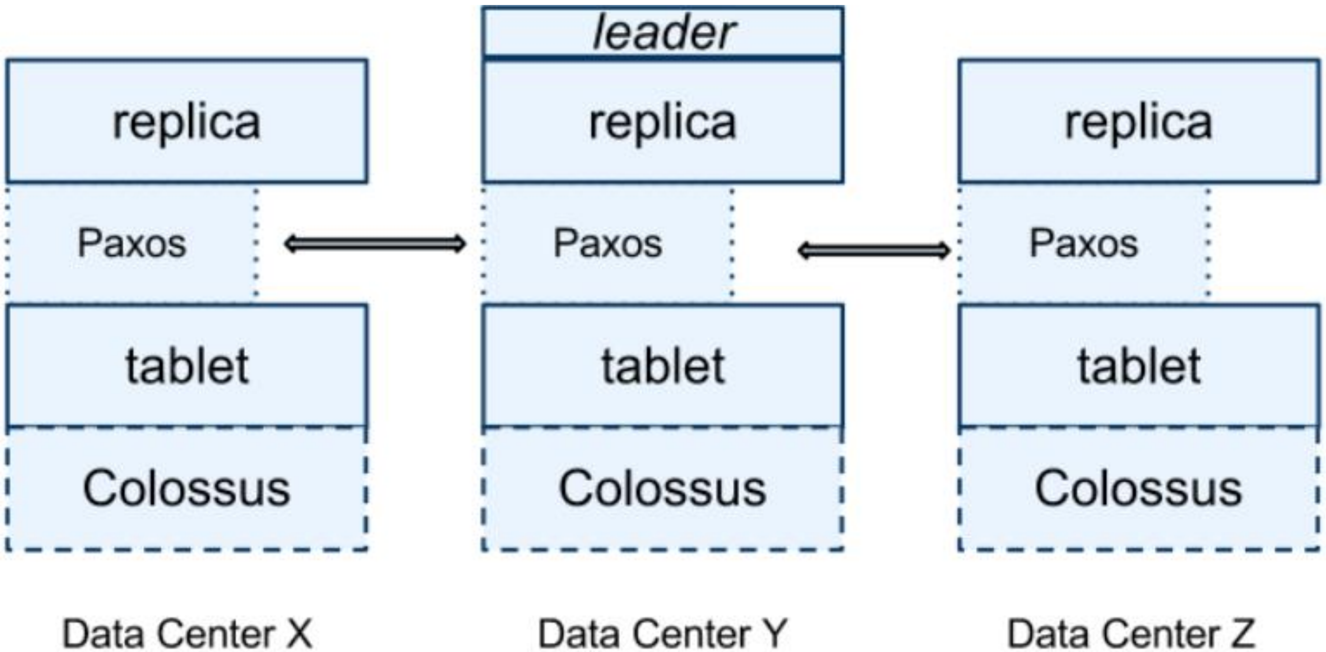
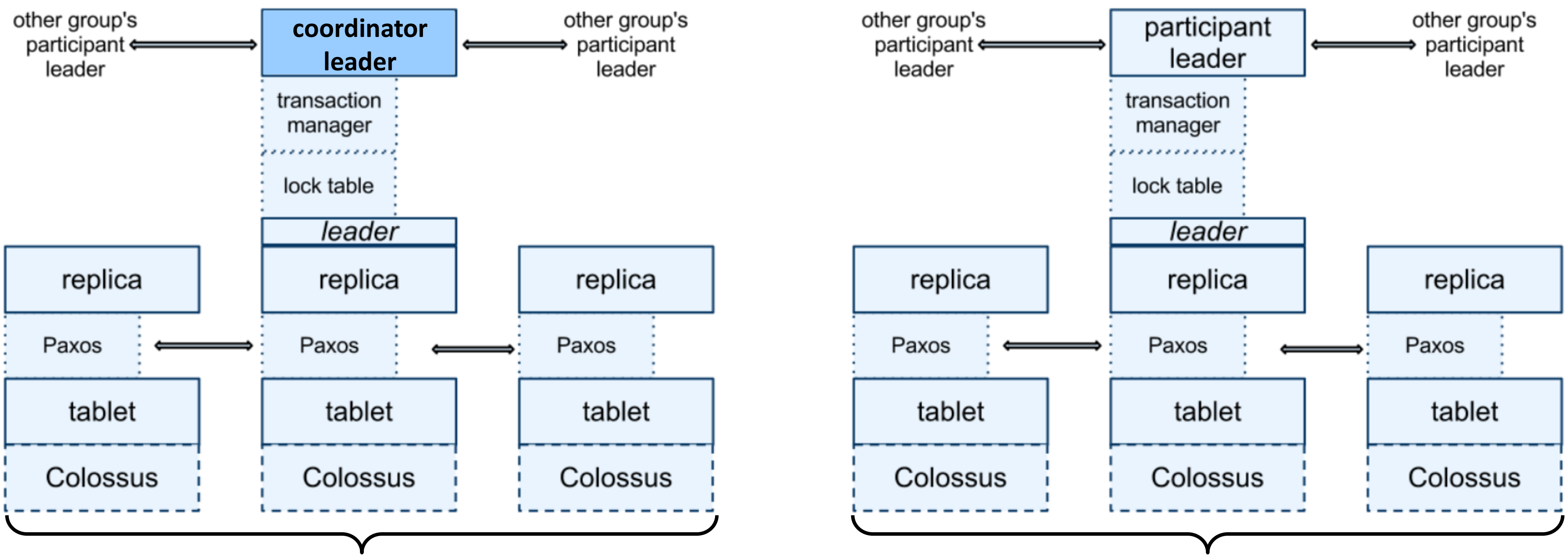


Figure 2: Spanserver software stack.

Spanner Transactions

- Uses strict two-phase locking and two-phase commit for read-write transactions, ensuring strict serializability
- Spanner provides external consistency → This is the same guarantee as strict serializability. So, what specific problem are they solving?



Replication of coordinator and participant transaction state ensures non-blocking commit, high availability

Read-Heavy Workloads

- Reads are dominant in many workloads
- Facebook's TAO had 500:1 reads-write ratio [ATC 2013]
- Google Ads (F1) on Spanner has 1000:1 read-write ratio
  - One data center in 24 hours had
    - 21.5B reads
    - 31.2M single-shard transactions
    - 32.1M multi-shard transactions

Fast Read-Only Transactions

- Transactions that only read data
  - Predeclared, e.g., developer uses READ_ONLY flag

- Spanner provides lock-free reads while ensuring strict serializability!
  - Reads don't acquire locks and thus don't block writers
  - Reads may block on writers but are consistent, i.e., read latest committed version
  - Snapshot reads (reads in the past) are supported

- How can we perform lock-free reads correctly?

Multi-versioning and Timestamps

- Lock-free reads can be performed by keeping multiple immutable versions of data and using timestamps
  - Writer: Each write creates a new immutable version with a timestamp of the transaction that issues the write
  - Reader: A read at a timestamp returns the value of the most recent version prior to that timestamp
  - Reader doesn't block writer
- The approach above allows lock-free reads,
  but how can we perform consistent reads?
  - i.e., after a read-write transaction completes, a later read-only transaction (in real-time order) returns the value written by the read-write transaction (or later read-write transaction)

Lock-Free Read-Only Transactions (Basic Idea)

assume
global wall-clock time

- Read-write transactions:
  - On commit, assign timestamp *Tw = current time* to transaction
  - All replicas track how up-to-date they are: *Tsafe*
    - => Replica has all committed transactions with timestamp *T* < *Tsafe*
- Read-only transactions:
  - Assign timestamp *Tr = current time* to transaction
  - Wait until *Tr* < *Tsafe* at any replica
  - Read data as of *Tr*
  - Guarantees read reflects
    all transactions committed before *Tr*,
    i.e., linearizable read-only transactions

assume
global wall-clock time

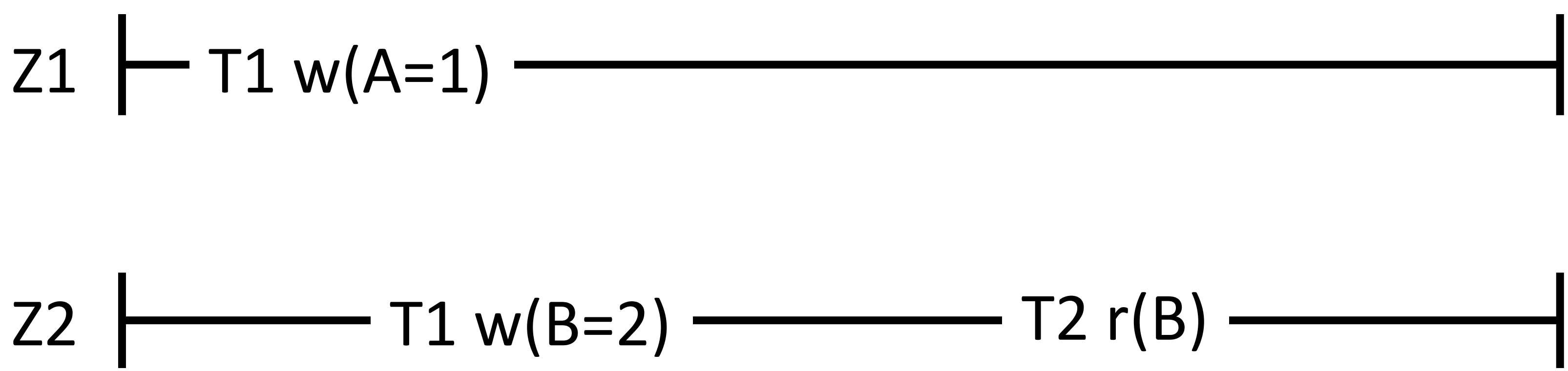
Timestamp Synchronization Problem

- Read-write transactions:
  - On commit, assign timestamp *Tw = current time*
  - All servers track how up-to-date they are *Tsafe*
    - => Replica has all committed transactions with timestamp *T < Tsafe*
- Read-only transactions:
  - Assign timestamp *Tr = current time*
  - Wait until *Tr < Tsafe*
  - Read data as of *Tr*
  - Guarantees read reflects
    all transactions committed before *Tr* ,
    i.e., linearizable read-only transactions
- Transactions are initiated and committed on different machines, so times may not be synchronized

How can the boxed operations be performed correctly?

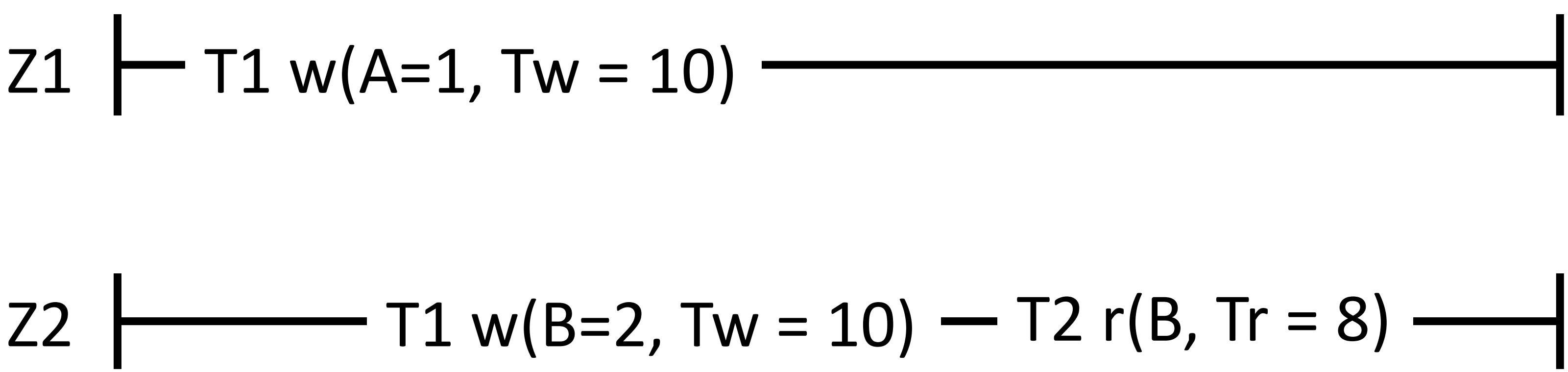
Timestamp Problem

- Say a person issues transaction T1 in Zone Z1
  - T1 writes A=1 at Z1, B=2 at Z2
- Then the same person issues transaction T2 in Zone Z2
  - T2 reads B at Z2
- Person expects that T2's read B will return 2



Timestamp Problem

- But what if Z2 is running much behind Z1?
- T1 is assigned timestamp based on Z1, e.g., Tw=10
- T2 is assigned timestamp Tr based on Z2, e.g., Tr=8
- Then, T2 reads previous version of B!

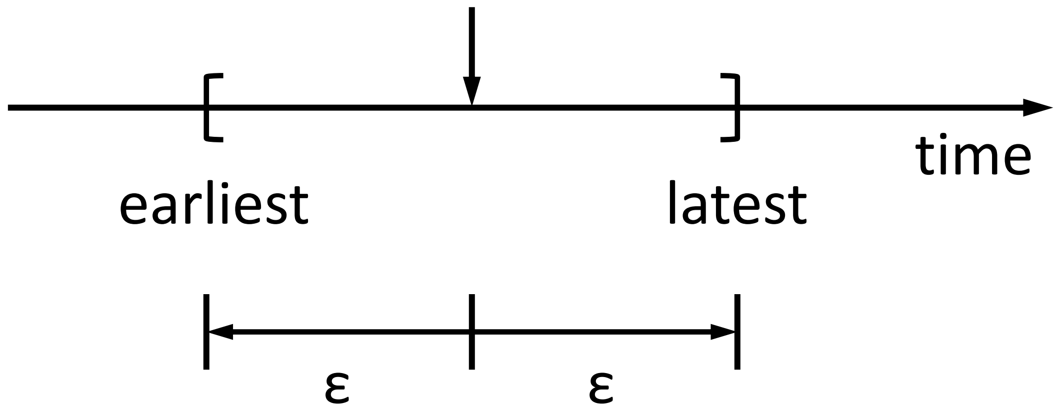


Key Innovation in Spanner

- Spanner provides a time API called TrueTime that provides bounded error
  - Clocks on all Spanner machines, across all data centers, are engineered to have a maximum divergence!
- TrueTime enables three innovations:
  1. Using the bounded error to ensure lock-free consistent reads
  2. Assigning commit timestamps to transactions that reflect serialization order in real time without global communication
  3. Allowing consistent reads for replicated data from any replica

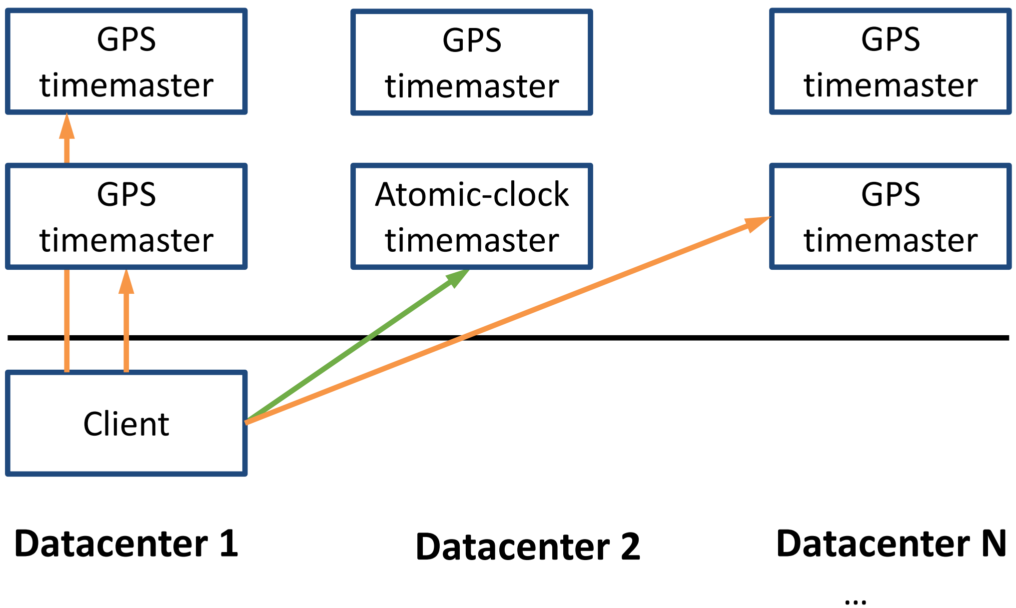
TrueTime

- A global wall-clock time with bounded uncertainty
- Consider event *e* that invokes tt = TT.now()
  - TrueTime tt is an interval (earliest, latest) with the guarantee:
  - tt.earliest <= $t_{abs}(e)$ <= tt.latest, t_{abs} is global wall-clock time



- Error bound ε is determined based on worst-case clock drift, communication delay to time masters

True Time Architecture



- Each client periodically synchronizes its clock:
  - Contacts multiple GPS and atomic-clock timeservers
  - Estimates **`reference now`**, **`reference error bound (ε)`**

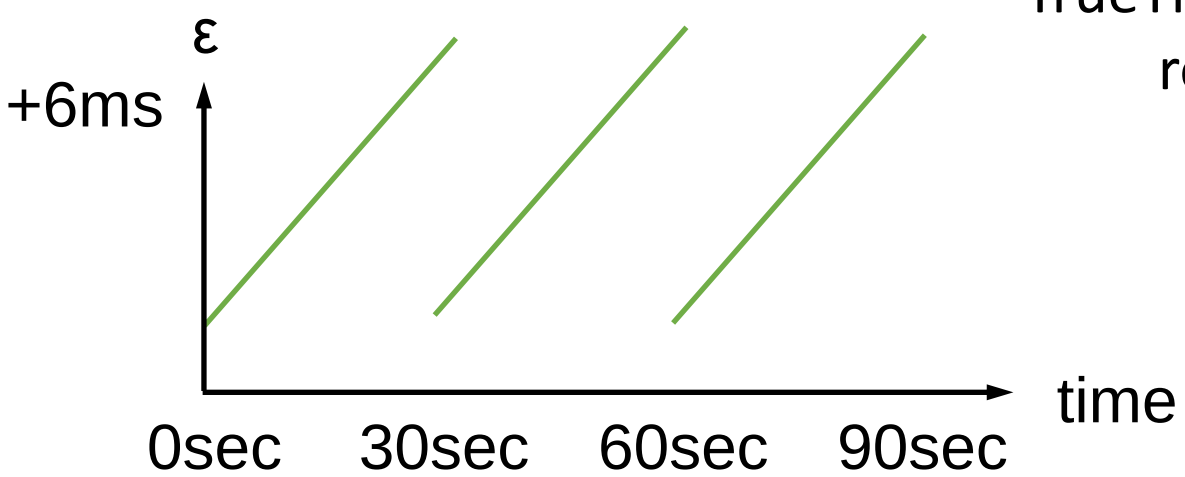
True Time Implementation

- TrueTime in between clock synchronizations:

```
now = reference now + local-clock offset
ε   = reference ε   + worst-case local-clock drift
TrueTime = now ± ε
```

Assumed to be 200 µs/sec for Google's machines

TrueTime interval is 2*ε, roughly 10 ms



Read-Only Txns with TrueTime

still assume
global wall-clock time

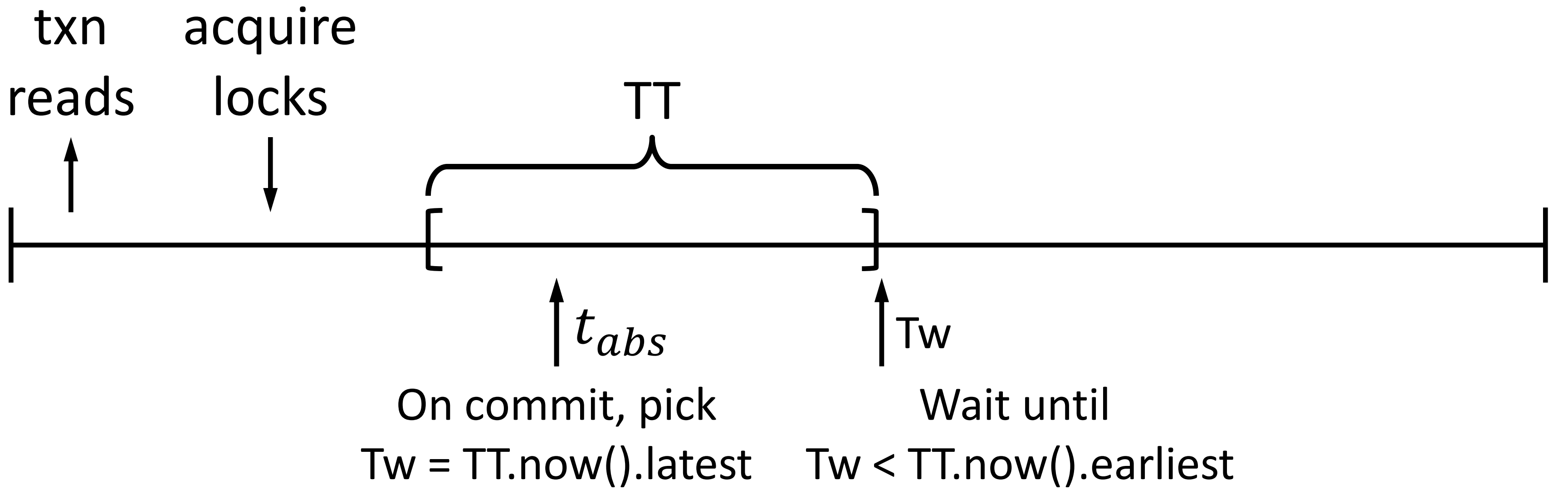
- Read-write transactions:
  - On commit, assign timestamp *Tw = current time* to transaction
  - All replicas track how up-to-date they are: *Tsafe*
    - => Replica has all committed transactions with timestamp *T* < *Tsafe*
- Read-only transactions:
  - Assign timestamp *Tr = TT.now().latest* to transaction
  - Wait until *Tr* < *Tsafe* at any replica
  - Read data as of *Tr*
  - Bounded error guarantees read reflects all transactions committed before *Tr*, i.e., linearizable read-only transactions

With TrueTime,
Tr >= global wall-clock time

Innovation 1

Read-Write Txns with TrueTime

- Read-write transactions:
  - On commit, assign timestamp *Tw = TT.now().latest* to transaction (similar to read-only transactions)
  - Wait until *Tw < TT.now().earliest* to commit



Commit Timestamp and Real-Time Serialization Order

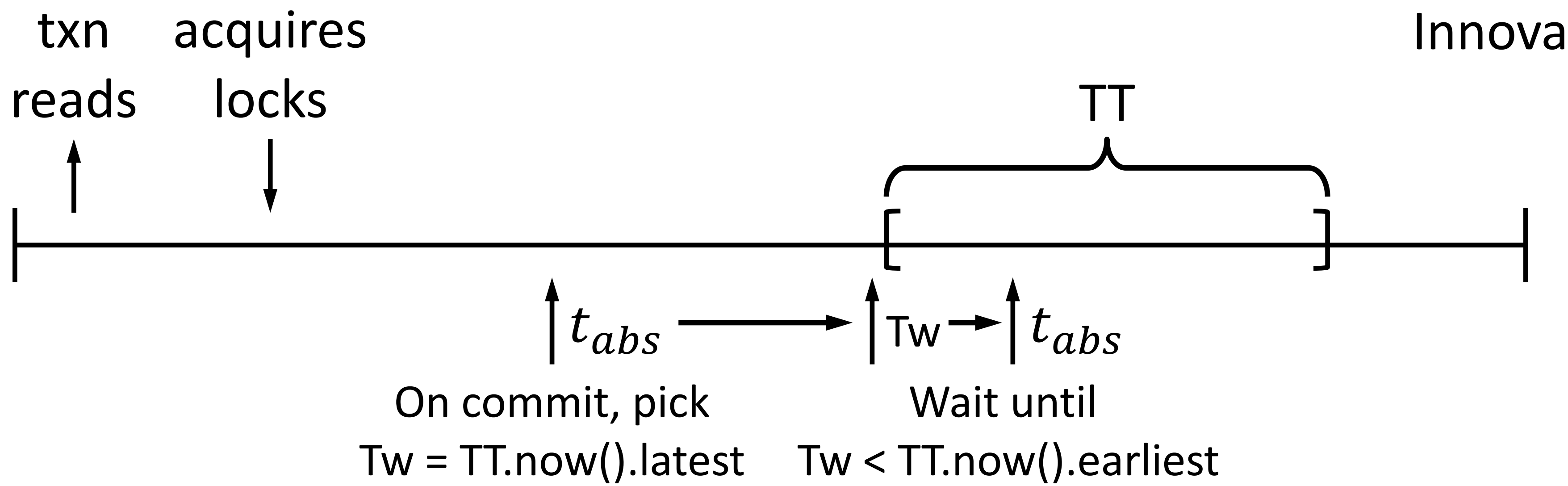
- Read-write transactions:
  - On commit, assign timestamp *Tw = TT.now().latest* to transaction (similar to read-only transactions)
  - Wait until *Tw < TT.now().earliest* to commit

With TrueTime,
Tw >= begin of commit

commit timestamp
respects real-time
serialization order

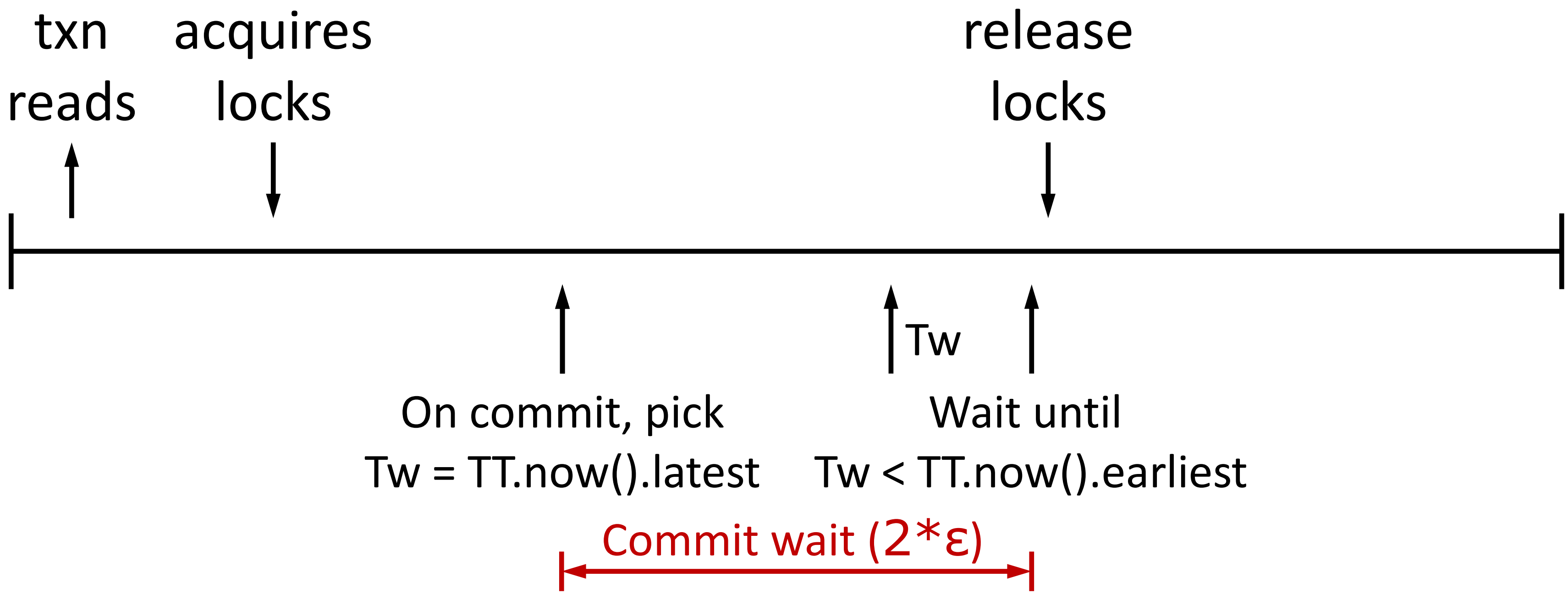
With TrueTime,
Tw < end of commit

Innovation 2



Commit Wait Time

- Read-write transactions:
  - On commit, assign timestamp *Tw = TT.now().latest* to transaction (similar to read-only transactions)
  - Wait until *Tw = TT.now().earliest* to commit
    - Expected wait is roughly 2*ε, TrueTime interval



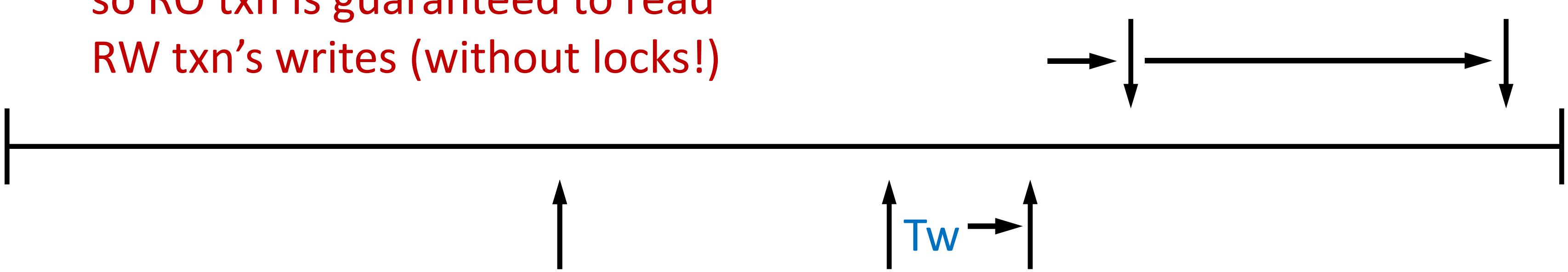
Consistent Lock-Free Reads

Innovation 1+2

- TrueTime guarantees consistent lock-free reads because commit timestamps reflect real-time serialization order

1. Tw < RW_txn ends (commit wait)
2. RW_txn ends < RO_txn starts (RO starts after RW ends)
3. RO_txn starts <= Tr (timestamp assignment)

⇒ Tw < Tr



Transaction Replication

- A read-write transaction runs at leader replica
- During commit, transaction log is replicated using consensus
  - Commit wait and consensus overlap in time!

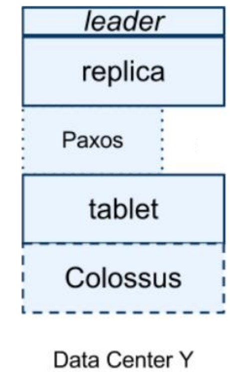
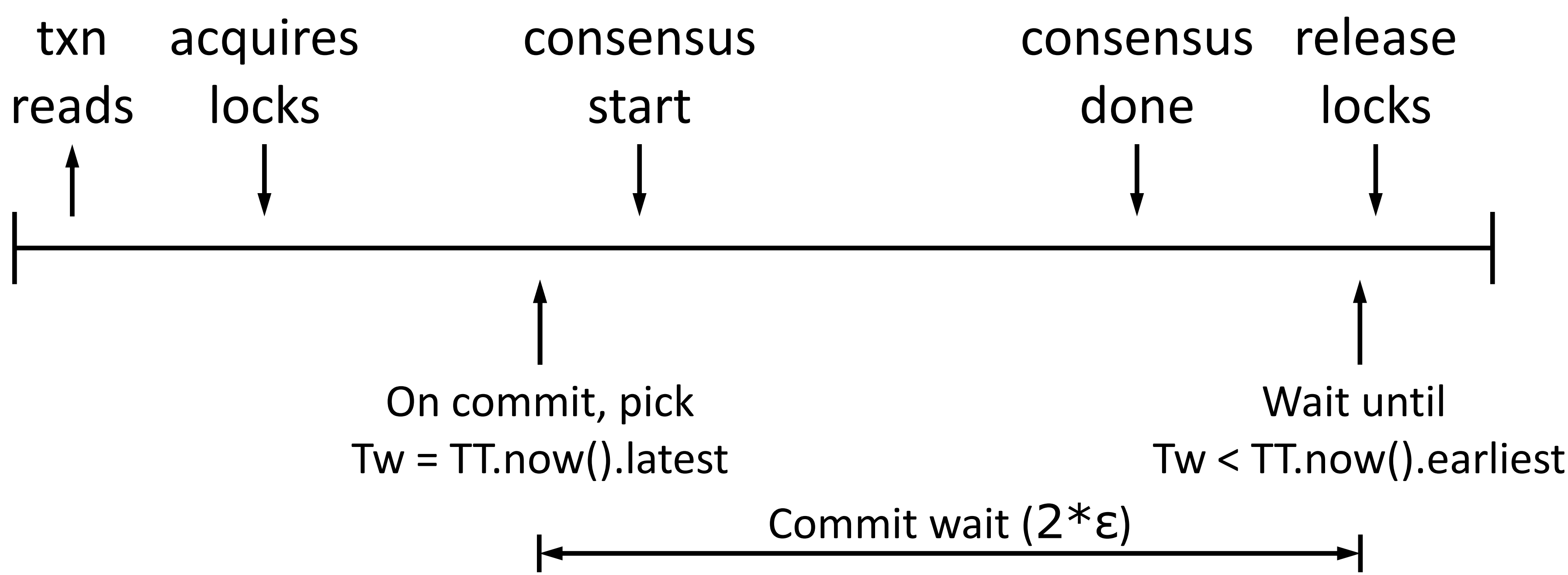
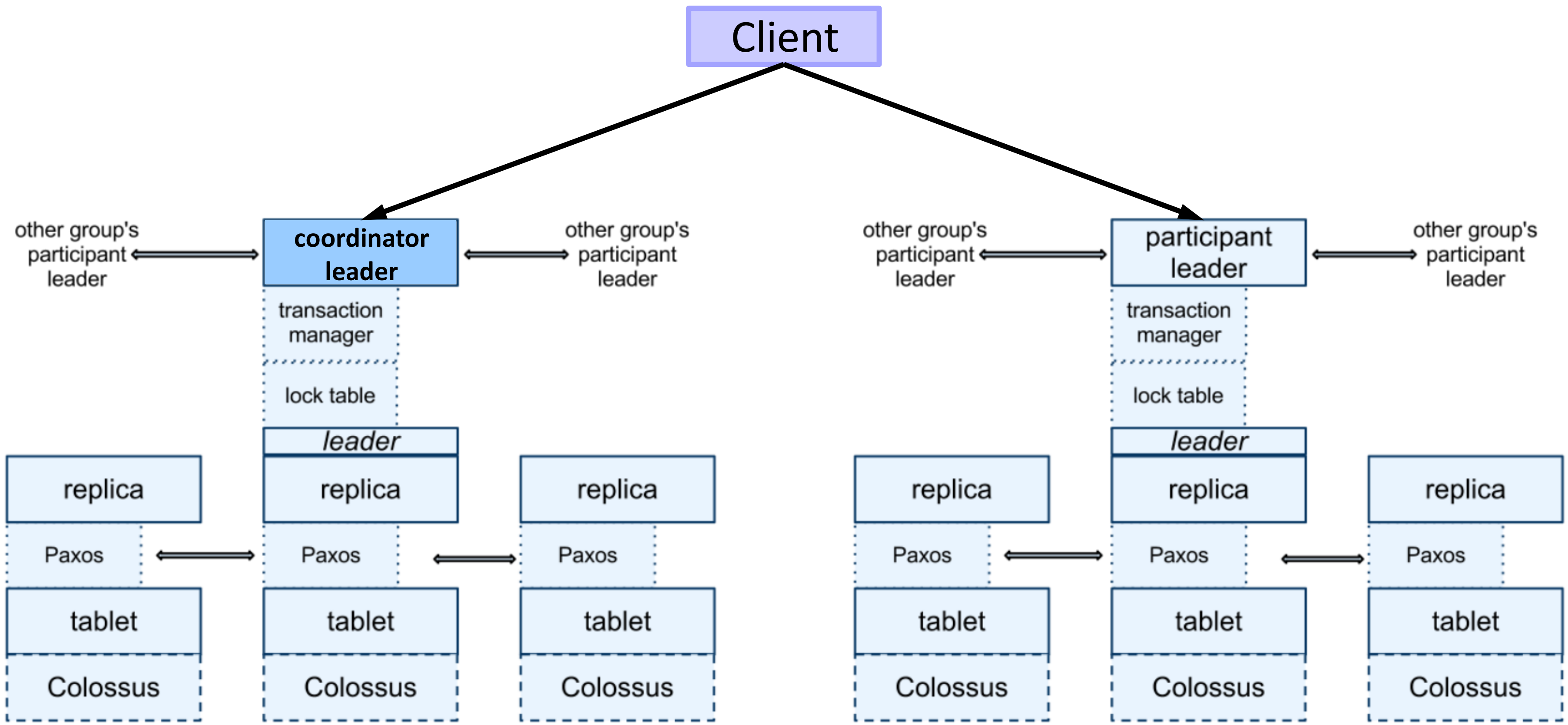


Figure 2: Spanserver software stack.



Distributed Transactions

- For read-write transactions, clients read data from leader replicas, drive two-phase commit



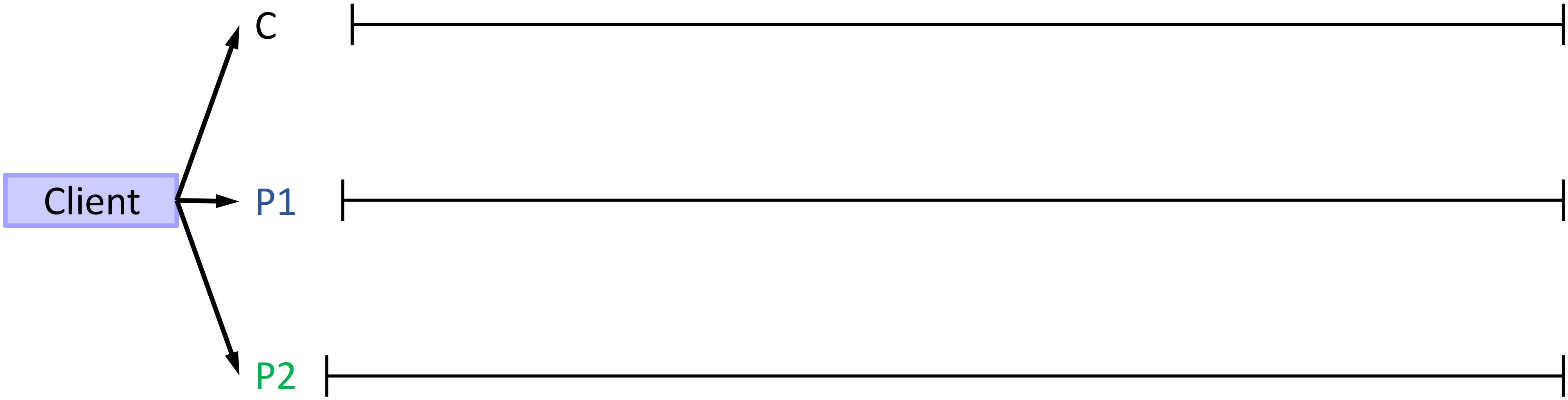
Distributed Transaction Execution

- Read-write transaction execution:
  - Client issues reads to leader (replica) of each tablet
  - Leaders acquire read locks, return most recent data
    - Recall, data is versioned
  - Client buffers writes
- Read-write transaction commit:
  - Client chooses coordinator from set of leaders
  - Client sends commit message to each leader, including identify of coordinator and buffered writes
  - Client waits for commit from coordinator

Two-Phase Commit

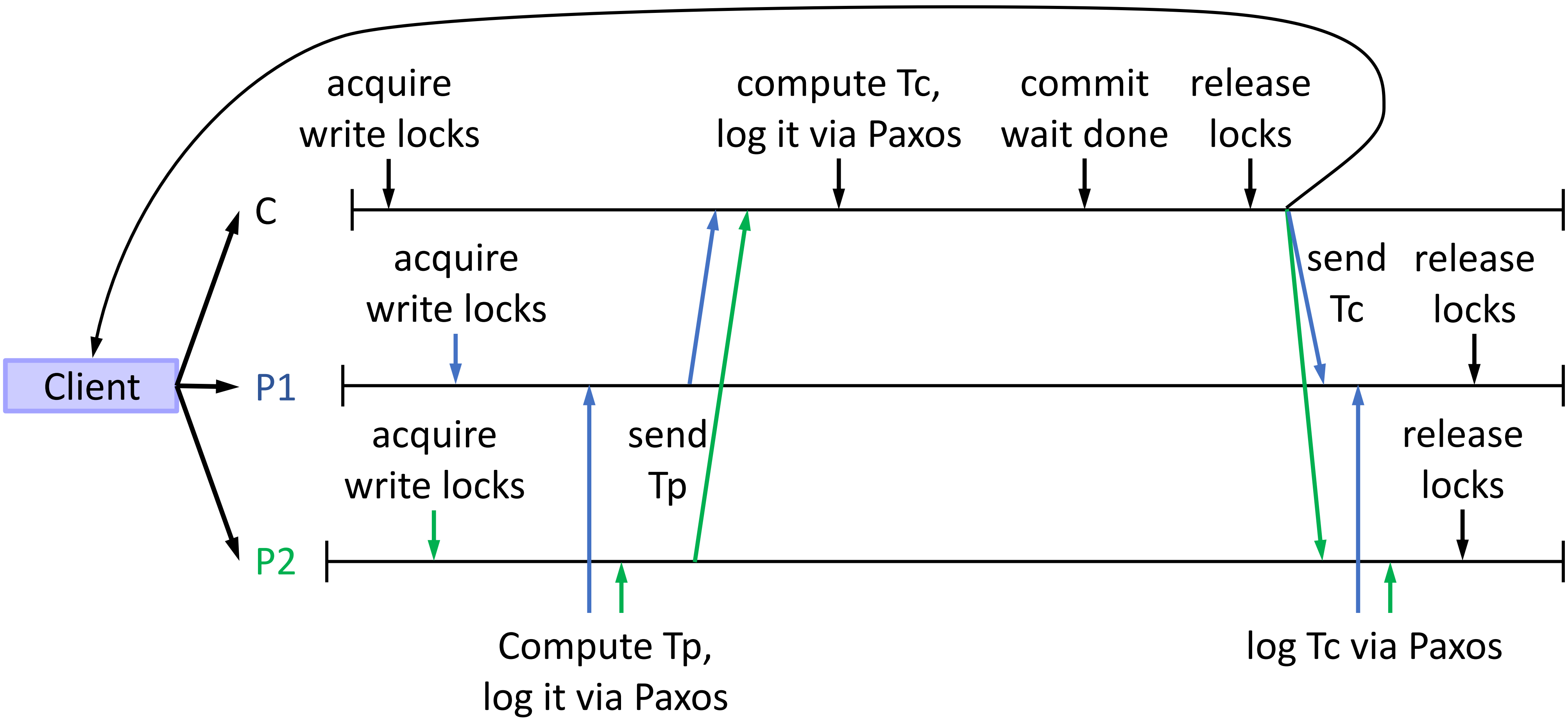
- On commit msg from client, participant leaders:
  - Acquire write locks
  - Choose increasing prepare timestamp (Tp) > all previously logged local ts
  - Log prepare record through Paxos
  - Notify coordinator of prepare timestamp
- On receiving all participant replies, coordinator leader:
  - Chooses monotonically increasing commit timestamp (Tc), such that:
    1) >= all Tp, 2) > previously logged local ts, 3) > TT.now.latest()
  - Logs commit record through Paxos
  - Waits until Tc < TT.now.earliest(), i.e., commit-wait period
  - Sends commit timestamp to other leaders, client
- All leaders log commit timestamp and transaction outcome through Paxos, and release locks

Two-Phase Commit



- Client chooses coordinator from set of leaders
- Client sends commit message to each leader (C, P1, P2), including identify of coordinator and buffered writes

Two-Phase Commit



- Client waits for commit from coordinator
- Client wait done

Tracking Progress at Replicas

- Recall that read-only transactions wait until *Tr* < *Tsafe*
  - All transactions with timestamp *T* < *Tsafe* have committed
  - But how is *Tsafe* determined?
- Spanner ensures
  1. Leaders use TrueTime to have disjoint lease intervals, assign timestamps to Paxos writes in monotonically increasing order
  2. Each replica assigns and logs prepare and commit timestamps via Paxos in monotonically increasing order

  ⇒ *Tsafe* is roughly the highest commit timestamp before which there are no prepare timestamps
- Each replica tracks *Tsafe,* so consistent reads can be performed at any replica ⟵ Innovation 3

Conclusions

- Spanner is a globally-distributed database that combines concurrency control (2PL) with atomic commit (2PC) and replication (Paxos)
  - Provides strong consistency and availability at global scale!
  - Makes it easy for developers to build apps
- Optimizes for reads, which are dominant
  - Enables strongly consistent, lock-free reads
- TrueTime exposes clock uncertainty
  - Transactions wait out this uncertainty to ensure real-time ordering of transactions
- CockroachDB, YugabyteDB build on Spanner

Discussion

Q1

- In what ways does Spanner use TrueTime?

Q2

- Databases generally use single-version, lock-based concurrency control, or multi-versioned concurrency control. Why does Spanner use both locking and multi-versioning?

Q3

- Spanner keeps a lock table at the leader replicas (of the tablets). Why is this table not replicated using Paxos at the participant replicas?

Q4

- How would a large TrueTime error bound affect Spanner?

Q5

- Compared to Dynamo, how may Spanner limit availability and performance for writes, reads?