Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Prof. Anna Arpaci-Dusseau**
and **Prof. Ioannis Liagouris** for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

0.1 Spanner: A Globally-Distributed Database

In 2012, Corbett et al. introduced Spanner, Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. The first system to support world-wide atomic transactions with its novel TrueTime API (Original Paper):

Definition 1.1: Spanner Design Goals

Google's Spanner is architected to be a planet-scale, strongly-consistent system. Its primary design goals are:

- **Global Scale & Distribution:** Span millions of machines and petabytes of data across multiple datacenters, automatically sharding and rebalancing data.
- **External Consistency:** Provide linearizable reads and writes across shards and datacenters via TrueTime; every transaction appears to occur at a single, globally-agreed timestamp.
- **Serializability:** Support serializable multi-row, multi-shard transactions using two-phase locking within Paxos-replicated groups (a consensus model comparable to raft) and two-phase commit across groups.
- **High Availability & Durability:** Synchronously replicate each shard with Paxos so that any replica group tolerates datacenter outages without losing committed data.
- **Low-Latency, Lock-Free Reads:** Enable read-only transactions to serve at a timestamp in the past without acquiring locks, minimizing read latency under contention.

In short,
Consistency Model: Strict serializability, with atomic transactions.

Note: We do not cover paxos, but every time it is said, just think of it as the consensus module in the system. I.e., one may synonymously think of it as raft.

The main motivation here is that when we want to do a database backup, or perform some enourmous read, placing a lock on the entire system would not be a great user experience. So we utilize snapshots instead, recall Definition (**??**):

Let's discuss how Spanner achieves these goals, with snapshots and timestamps:

Definition 1.2: Consistent Snapshots via MVCC

Spanner uses **Multi-Version Concurrency Control (MVCC)** and globally ordered timestamps to serve lock-free, causally-consistent read-only transactions:

- **Versioned Writes:** Each update transaction T_w is assigned a unique, monotonically increasing commit timestamp t_w. Every write to key k creates a new version tagged with t_w rather than overwriting.
- **Safe Time Rule:** Before T_r reads key k, it must see a write T_w that is $t_w > t_r$.
- **Causal Consistency:** Global timestamp ordering ensures that if $T_1 \to T_2$, any snapshot that reflects T_2's writes also reflects T_1's.
- **Lock-Free Reads:** Read-only transactions never acquire locks and run entirely against their fixed snapshot, eliminating read-write contention.

Definition 1.3: Commit Timestamp Ordering

Spanner assigns each transaction T_i a commit timestamp t_n s.t., for any two T_1 and T_2:

$$T_1 \text{ completes before } T_2 \quad \Longrightarrow \quad t_1 < t_2.$$

We use **physical clocks** (atomic) to assign timestamps. Lamport (logical) clocks capture causal dependencies but **do not guarantee** real-time precedence.

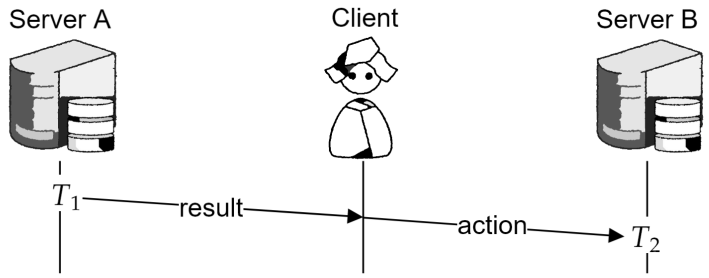


Figure 0.1: Server A may never communicate with server B, making logical clocks insufficient.

Google implements atomic clocks and GPS receivers to mitigate physical limitations of clocks:

Definition 1.4: TrueTime: Bounded Clock Uncertainty

Spanner's TrueTime API provides each server a timestamp interval $[t_{earliest}, t_{latest}]$ such that the actual absolute time (real-time) t_{abs} lies within the interval: $t_{earliest} \leq t_{abs} \leq t_{latest}$. On commit, Spanner enforces strict real-time ordering by:

- **Interval Return:** A call to `TT.now()` returns $[t_{earliest}, t_{latest}]$.
- **Uncertainty Wait:** Before finalizing a commit timestamp t, the coordinator waits until its local clock $\geq t_{latest}$, ensuring no future transaction can obtain a timestamp $\leq t$.
- **Monotonicity & Ordering:** This wait ensures that once a transaction reports commit at t, all subsequent `TT.now()` calls on any server will yield intervals with $t_{earliest} > t$, guaranteeing T_1 commit precedes T_2 timestamp.
- **Global Coverage:** By combining redundant GPS receivers and local atomic clocks in each datacenter, TrueTime keeps its uncertainty bound $\delta = t_{latest} - t_{earliest}$ small (typically a few milliseconds), even under network partitions (Kops)

E.g., If it's 12:32:01 pm we might see: [12:31:59pm, 12:32:04pm] where δ is 5 milliseconds.

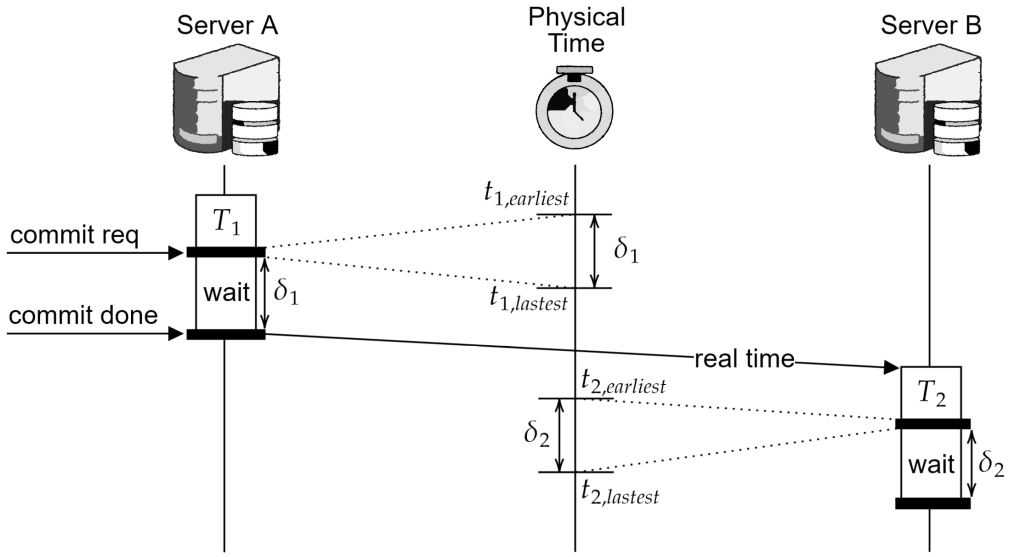


Figure 0.2: Upon receiving a commit request, each coordinator (e.g. A) calls `TT.now()` and obtains an uncertainty interval $[t_{1,\text{earliest}}, t_{1,\text{latest}}]$. It then delays for $\delta_1 = t_{1,\text{latest}} - t_{1,\text{earliest}}$ before finalizing its commit timestamp.

Finally we touch on the two-phase commit protocol:

Definition 1.5: Cross-Shard Transactions & Paxos Replication

Spanner provides strictly serializable, atomic transactions across independently-replicated shards by combining two-phase locking (2PL), two-phase commit (2PC), and Paxos state-machine replication:

- **Coordinator Selection:** After a client buffers all its reads and pending writes, it picks one of the involved shards' Paxos group leaders to act as the **2PC coordinator**. This is simply the leader replica for one of the keys in the transaction's shard set.
- **Intra-Shard Isolation:** Within each shard, that shard's Paxos leader acquires two-phase-locks on the keys being accessed, serializing concurrent read-write operations.
- **Atomic Cross-Shard Commit:** The coordinator issues a 2PC "`PREPARE`" to every participant leader. Each participant logs the prepare record via Paxos, acquires its write-locks, and returns a prepare-OK. Once the coordinator has all prepares, it logs the global "`COMMIT`" via Paxos to each shard, ensuring the transaction is made durable and atomic.
- **Paxos Fault Tolerance:** Let f be the maximum number of servers allowed to fail before system failure. Then each shard is a Paxos group of $n = 2f + 1$ replicas; as long as a majority $(f + 1)$ are up, Paxos ensures progress, automatically electing new leaders and replicating log entries even if up to f replicas fail.

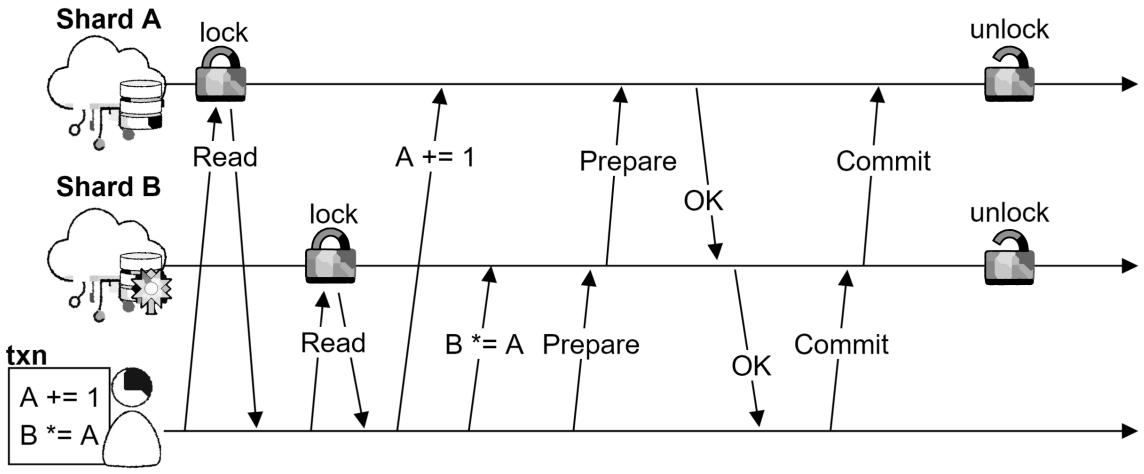
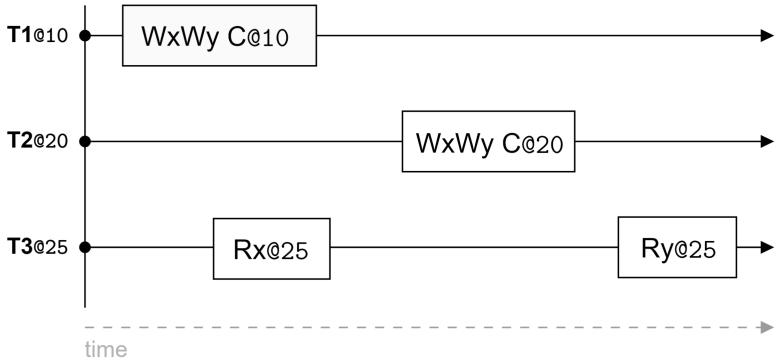


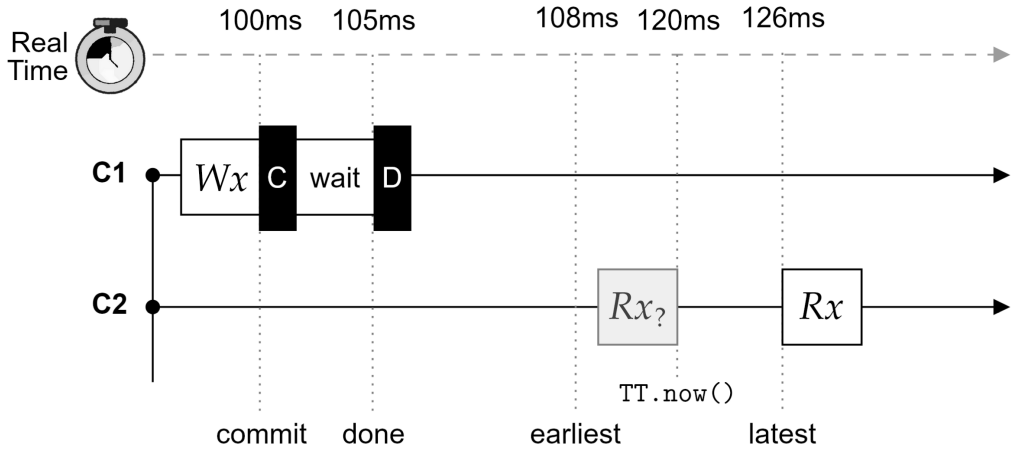
Figure 0.3: A Read-Write Transaction 2PC Commit in Spanner, where txn is the transaction, B's Paxos leader is the coordinator, and A another participant. First, reads acquire locks, then 2PC begins. Commands are sent, and then the `PREPARE` and `COMMIT` phases begin, ending with the release of locks.

Example 1.1: MVCC & TrueTime Evaluation

Consider the bellow transactions:



Here we have three transactions, T_1, T_2, and T_3. First we see that T_1 make a write request to both keys x and y at timestamp 10. Then T_2 makes the same request but at timestamp 20. Though it appears T_3 is a bit out of sync, reading at time 25. Hence, by the safe time rule, T_3 must see a write propagate with a time greater than 25. Now adding TrueTime intervals:



Here client C_1 has already prepared the write Wx. C_1 then sends a commit request at 100 ms, which returns a TrueTime interval of [100ms, 105ms]. The black boxes visualize possible uncertainty intervals. C_1 then waits for 5 ms before sending the commit request. On line 2, C_2 sends a read request to x at 120 ms. They receive a TrueTime interval of [108ms, 126ms]. C_2 then waits for 6 ms before sending before attempting to read x; However, since C_1's commit timestamp is 105ms, C_2 must wait for a later commit timestamp, which is typically an insignificant amount of time. ■

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.