Recap: Distributed Storage Systems

- Concurrency control
  - Order transactions across shards

- State machine replication
  - Replicas of a shard apply transactions in the same order decided by concurrency control

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)

Why Google Built Spanner

2005 – BigTable [OSDI 2006]
  - Eventually consistent across datacenters
  - Lesson: “don’t need distributed transactions”

  - Strongly consistent across datacenters
  - Option for distributed transactions
  - But performance was not great...

2011 – Spanner [OSDI 2012]
  - Strictly Serializable Distributed Transactions
  - “We wanted to make it easy for developers to build their applications”
Motivation: Performance-consistency tradeoff

• Strict serializability
  • Serializability + linearizability
  • As if coding on a single-threaded, transactionally isolated machine
  • Spanner calls it external consistency
• Strict serializability makes building correct application easier
• Strict serializability is expensive
  • Performance penalty in concurrency control + Repl.
    • OCC/2PL: multiple round trips, locking, etc.

Motivation: Read-Only Transactions

• Transactions that only read data
  • Predeclared, i.e., developer uses READ_ONLY flag / interface
• Reads dominate real-world workloads
  • FB’s TAO had 500 reads : 1 write [ATC 2013]
  • Google Ads (F1) on Spanner from 17 DC in 24h:
    • 31.2 M single-shard read-write transactions
    • 32.1 M multi-shard read-write transactions
    • 21.5 B read-only (~340 times more)
• Determines system overall performance

Can we design a strictly serializable, geo-replicated, sharded system with very fast (efficient) read-only transactions?

Before we get to Spanner ...

• How would you design SS read-only transactions?
• OCC or 2PL: Multiple round trips and locking
• Can always read in local datacenters like COPS?
  • Maybe involved in Paxos agreement
  • Or must contact the leader
• Performance penalties
  • Round trips increase latency, especially in wide area
  • Distributed lock management is costly, e.g., deadlocks
Goal is to ...

- Make read-only transactions efficient
  - One round trip (as could be wide-area)
  - Lock-free
    - No deadlocks
    - Processing reads do not block writes, e.g., long-lived reads
  - Always succeed (do not abort)

- And strictly serializable

Leveraging the Notion of Time

- Task 1: when committing a write, tag it with the current physical time
- Task 2: when reading the system, check which writes were committed before the time this read started.
- How about the serializable requirement?
  - Physical time naturally gives a total order

Invariant:
If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Trivially provided by perfect clocks
### Challenges
- Clocks are not perfect
  - Clock skew: some clocks are faster/slower
  - Clock skew may not be bounded
  - Clock skew may not be known a priori
- T2 may be tagged with a smaller timestamp than T1 due to T2’s slower clock
- Seems impossible to have perfect clocks in distributed systems. What can we do?

### Nearly perfect clocks
- Partially synchronized
  - Clock skew is bounded and known a priori
  - My clock shows 1:30PM, then I know the absolute (real) time is in the range of 1:30 PM +/- X.
    - e.g., between 1:20PM and 1:40PM if X = 10 mins
  - Clock skew is short (e.g., X = a few milliseconds)
- Enable something special, e.g., Spanner!

### Spanner: Google’s Globally-Distributed Database
**OSDI 2012**

- 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!
Strictly Serializable Multi-shard Transactions

• How are clocks made “nearly perfect”?

• How does Spanner leverage these clocks?
  • How are writes done and tagged?
  • How read-only transactions are made efficient?

TrueTime (TT)

• API (software interface)
  • TT.now() = [earliest, latest] # latest – earliest = 2ε
  • TT.after(t) = true if t has passed
    • TT.now().earliest > t (because tabs >= TT.now().earliest)
  • TT.before(t) = true if t has not arrived
    • TT.now().latest < t (because tabs <= TT.now().latest)

• Implementation
  • Relies on specialized hardware, e.g., satellite and atomic clocks

Enforcing the Invariant

If T2 starts after T1 commits, then T2 must have larger timestamp
Let T1 write S_B and T2 write S_A

Perfect Clocks
Enforcing the Invariant

If T2 starts after T1 commits, then T2 must have larger timestamp
Let T1 write $S_B$ and T2 write $S_A$

S_A  
\[ \uparrow \]
\[ \text{Tabs} \]
\[ \downarrow \]
S_B
\[ \text{T1.now} = 5 \]
\[ \text{T1.commit} \ (ts = 5) \]

Perfect Clocks

Enforcing the Invariant

If T2 starts after T1 commits, then T2 must have larger timestamp
Let T1 write $S_B$ and T2 write $S_A$

S_A  
\[ \uparrow \]
\[ \text{Tabs} \]
\[ \downarrow \]
S_B
\[ \text{T1.now} = 5 \]
\[ \text{T1.commit} \ (ts = 5) \]

Perfect Clocks

Enforcing the Invariant

If T2 starts after T1 commits, then T2 must have larger timestamp
Let T1 write $S_B$ and T2 write $S_A$

T2.commit
\[ ts = 6 \]

T1.now
\[ = 12 \]
\[ \text{T1.commit} \ (ts = 12) \]

Imperfect Clocks
Enforcing the Invariant

If T2 starts after T1 commits, then T2 must have larger timestamp
Let T1 write S_B and T2 write S_A

T_abs

S_A

T1.now() = [3, 6] (ts = 6)
T1.commit = 6
T2.now() = [8, 12] (ts = 12)
T2.commit = 12

T2.ts > T1.ts

Seems working?

A brain teaser puzzle

We know:
1. x < y, b/c T2 in real-time after T1 (the assumption)
2. c <= y <= d, b/c TrueTime
3. T1.ts = b, T2.ts = d, b/c how ts is assigned

We want: it is always true that b < d, how?

Enforcing the Invariant (Strawman)

If T2 starts after T1 commits, then T2 must have larger timestamp
Let T1 write S_B and T2 write S_A

T_abs

S_A

T1.now() = [1, 12] (ts = 12)
T1.commit = 12
T2.now() = [1, 12] (ts = 12)
T2.commit = 12

Not working!

A brain teaser puzzle

We know:
1. x < y, b/c T2 in real-time after T1 (the assumption)
2. c <= y <= d, b/c TrueTime
3. T1.ts = b, T2.ts = d, b/c how ts is assigned

We want: it is always true that b < d, how?
1 and 2 -> x < d; we need to ensure b < x; then b < x < d, done.
Enforcing the Invariant

If T2 starts after T1 commits, then T2 must have larger timestamp
Let T1 write $S_B$ and T2 write $S_A$

If T2 starts after T1 commits, then T2 must have larger timestamp
Let T1 write $S_B$ and T2 write $S_A$

Enforcing the Invariant

If T2 starts after T1 commits, then T2 must have larger timestamp
Let T1 write $S_B$ and T2 write $S_A$

Takeaways

- The invariant is always enforced: If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp
- How big/small $\epsilon$ is does not matter for correctness
- Only need to make sure:
  - TT.now().latest is used for $ts$ (in this example)
  - Commit wait, i.e., TT.after($ts$) == true
- $\epsilon$ must be known a priori and small so commit wait is doable!
After-class Puzzles

- Can we use `TT.now().earliest for ts`?
- Can we use `TT.now().latest - 1 for ts`?
- Can we use `TT.now().latest + 1 for ts`?
- Then what’s the rule of thumb for choosing ts?