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”

2008? — MegaStore [CIDR 2011]
- 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 1? 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

• Strict serializability: a matter of real-time ordering
  • If txn T2 starts after T1 finishes, then T2 must be ordered after T1
  • If T2 is ro-txn, then T2 should see effects of all writes finished before T2 started

• A similar scenario at a restaurant
  • Alice arrives, writes her name and time she arrives (e.g., 5pm) on waiting list
  • Bob then arrives, writes his name and the time (e.g., 5:10PM)
  • Then Bob is ordered after Alice on the waiting list
  • I arrive later at 5:15PM and check how many people are ahead of me by checking the waiting list by time

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

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!
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)

- “Global wall-clock time” with bounded uncertainty
  - $\epsilon$ is worst-case clock divergence
  - Spanner’s notion of time becomes intervals, not single values
  - $\epsilon$ is 4ms on average, $2\epsilon$ is about 10ms

\[
\text{Consider event } e_{\text{now}} \text{ which invoked } tt = \text{TT.now}():
\]

Guarantee: $\text{tt.earliest} \leq t_{abs(e_{\text{now}})} \leq \text{tt.latest}$

TrueTime (TT)

- API (software interface)
  - $\text{TT.now}() = [\text{earliest}, \text{latest}]$ # latest – earliest = $2\epsilon$
  - $\text{TT.after}(t) = \text{true if } t \text{ has passed}$
    - $\text{TT.now}().\text{earliest} > t$ (b/c $t_{abs} > TT.now().\text{earliest}$)
  - $\text{TT.before}(t) = \text{true if } t \text{ has not arrived}$
    - $\text{TT.now}().\text{latest} < t$ (b/c $t_{abs} < TT.now().\text{latest}$)
- 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 $S_A$

$S_A$  
\[ T_{abs} \]
\[ T1.now() = 5 \]

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

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

Perfect Clocks

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

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 $S_A$

$T_1.now() = [3, 6]$  $T_1.commit = [6, 8] \quad (ts = 6)$
$T_2.now() = [8, 12] \quad (ts = 12)$

$T_2.ts > T_1.ts$ Seems working?

A brain teaser puzzle

We know:
1. $x < y$, b/c T2 in real-time after T1 (the assumption)
2. $c \leq y \leq d$, b/c TrueTime
3. $T_1.ts = b$, $T_2.ts = d$, 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 $S_A$

$T_1.now() = [3, 15]$  $T_1.commit = [15, 12] \quad (ts = 15)$
$T_2.now() = [1, 12] \quad (ts = 12)$

$T_2.ts < T_1.ts$ Not working!

A brain teaser puzzle

We know:
1. $x < y$, b/c T2 in real-time after T1 (the assumption)
2. $c \leq y \leq d$, b/c TrueTime
3. $T_1.ts = b$, $T_2.ts = d$, b/c how ts is assigned

We want: it is always true that $b < d$, how?
1 and 2 $\Rightarrow x < d$; 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

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

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 $\varepsilon$ 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

• $\varepsilon$ 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?