Spanner

COS 418: Distributed Systems
Lecture 17

Jeffrey Helt

Slides adapted from Haonan Lu, Wyatt Lloyd, and Mike Freedman's, which are adapted from the Spanner OSDI talk
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
  • Performance was not great…

2011 – Spanner [OSDI 2012]
• Strictly Serializable Distributed Transactions
• “We wanted to make it easy for developers to build their applications”
A Deeper Look at 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

• But strict serializability is expensive
  • Performance penalty in concurrency control + Repl.
    • OCC/2PL: multiple round trips, locking, etc.
A Deeper Look at 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’s 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 you always read in local datacenters like COPS?
  - No but maybe can read from Paxos quorum?
  - 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
    • Could be wide-area
  • Lock-free
    • No deadlocks
    • Processing reads does 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 T2 starts after T1 finishes, then T2 must be ordered after T1
    • If T2 is a RO txn, then T2 should see the effects of all writes that
      finished before T2 started.

• A similar scenario at a restaurant
  • Alice arrives, writes her name and the time she arrives (e.g., 5pm) on the 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
Leveraging the Notion of Time

• Idea 1: when committing a write, tag it with the current physical time

• Idea 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
**Scale-out vs. Fault Tolerance**

- Every shard replicated via Multi-Paxos
- So every “operation” within transactions across tablets is actually a replicated operation within a Paxos RSM
- Paxos groups can span 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?
“Global wall-clock time” with bounded uncertainty
- $\varepsilon$ is worst-case clock divergence
- Spanner’s notion of time becomes intervals, not single values
- $\varepsilon$ is 4ms on average, $2\varepsilon$ is about 10ms

Consider event $e_{\text{now}}$ which invoked $tt = \text{TT.now}()$:
Guarantee: $tt.\text{earliest} \leq t_{\text{abs}}(e_{\text{now}}) \leq tt.\text{latest}$
TrueTime (TT)

• Interface
  • TT.now() = [earliest, latest]  # latest – earliest = 2*ε
  • TT.after(t) = true if t has passed
    • TT.now().earliest > t (b/c $t_{abs} \geq TT.now().earliest$)
  • TT.before(t) = true if t has not arrived
    • TT.now().latest < t (b/c $t_{abs} \leq TT.now().latest$)

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

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

Let T1 write $S_B$ and T2 write $S_A$

Perfect Clocks
Enforcing the Invariant

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

Let T1 write $S_B$ and T2 write $S_A$

---

**Perfect Clocks**

<table>
<thead>
<tr>
<th>$T_{abs}$</th>
<th>5</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_B$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_A$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$T1.now() = 5$

$T1.commit (ts = 5)$

Perfect Clocks
Enforcing the Invariant

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

Let T1 write $S_B$ and T2 write $S_A$

Perfect Clocks

$T_1.now() = 5$

$T_1.commit (ts = 5)$

$T_2.now() = 10$
Enforcing the Invariant

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

Let T1 write $S_B$ and T2 write $S_A$

Perfect Clocks

T1.now() = 5
T1.commit (ts = 5)

T2.now() = 10
T2.commit (ts = 10)

T2.ts > T1.ts
Enforcing the Invariant

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

Let T1 write $S_B$ and T2 write $S_A$

- $S_A$
- $T_{abs}$
- $S_B$

T1.now() = 12  
T1.commit (ts = 12)

T2.now() = 6
T2.commit (ts = 6)

T2.ts < T1.ts  

Imperfect Clocks
Enforcing the Invariant

If $T_2$ starts after $T_1$ commits (finishes), then $T_2$ must have a larger timestamp

Let $T_1$ write $S_B$ and $T_2$ write $S_A$

$T_1$.now() = [3, 6]  $T_1$.commit (ts = 6)

$T_2$.now() = [8, 12]  $T_2$.commit (ts = 12)

$T_2$.ts > $T_1$.ts

Seems to work?
Enforcing the Invariant

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

Let T1 write $S_B$ and T2 write $S_A$

$T_1$ now() = [3, 15] $T_1$.commit (ts = 15)

$T_2$.now() = [1, 12]

$T_2$.commit (ts = 12)

T2.ts < T1.ts
Not working!
A brain teaser

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. \( T1.ts = b \), \( T2.ts = d \), b/c how ts is assigned

We want: \( b < d \) to always be true, how?
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: $b < d$ to always be true, how?

1 and 2 $\rightarrow$ $x < d$; we need to ensure $b < x$; then $b < x < d$, done.
Enforcing the Invariant with TT

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

Let T1 write $S_B$ and T2 write $S_A$

$$T_{abs} \quad S_A \quad S_B \quad T1.now() = [3, 15]$$

TrueTime
Enforcing the Invariant with TT

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

Let T1 write $S_B$ and T2 write $S_A$

T1.commit ($ts = 15$)

TT.after(15) == true

$b < x$
Enforcing the Invariant with TT

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

Let T1 write $S_B$ and T2 write $S_A$

T1.now() = [3, 15]  
T1.commit (ts = 15)  
TrueTime

T2.now() = [18, 22]  
T2.commit (ts = 22)

$T2.ts > T1.ts$
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?