

Spanner



COS 418: Distributed Systems
Lecture 17

Mike Freedman, Wyatt Lloyd

Some slides from the Spanner OSDI talk

1

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

2

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)

3

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"

4

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.

5

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

6

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

7

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

8

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**

9

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

10

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

11

Invariant:

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

Trivially provided by perfect clocks

12

(Obvious) 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?

13

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!

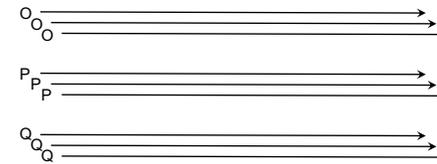
14

Spanner: Google's Globally-Distributed Database

OSDI 2012

15

Scale-out vs. Fault Tolerance



- Every shard replicated via MultiPaxos (akin to RAFT)
- So every "operation" within transactions across tablets actually a replicated operation within Paxos RSM
- Paxos groups can stretch across datacenters!

16

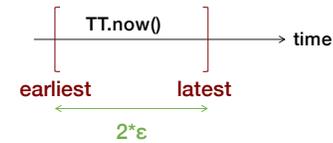
Strictly Serializable Multi-shard Transactions

- 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?

17

TrueTime (TT)

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



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

18

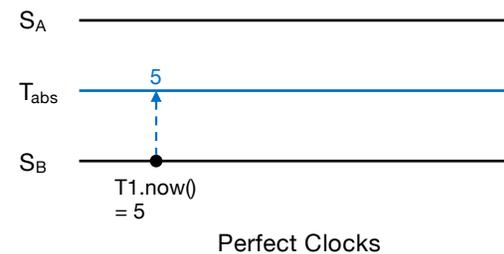
TrueTime (TT)

- API (software interface)
 - $TT.now() = [earliest, latest]$ # $latest - earliest = 2 * \epsilon$
 - $TT.after(t) = true$ if t has passed
 - $TT.now().earliest > t$ (because $t_{abs} \geq TT.now().earliest$)
 - $TT.before(t) = true$ if t has not arrived
 - $TT.now().latest < t$ (because $t_{abs} \leq TT.now().latest$)
- Implementation
 - Relies on specialized hardware, e.g., satellite and atomic clocks

19

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



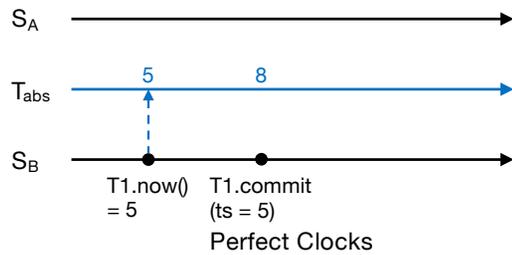
Perfect Clocks

20

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

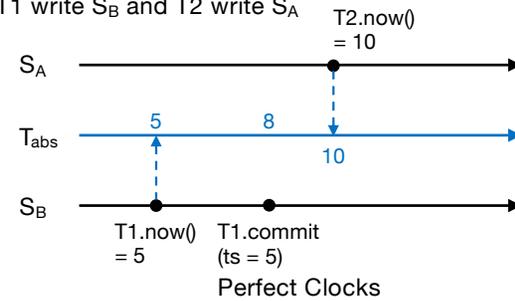


21

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

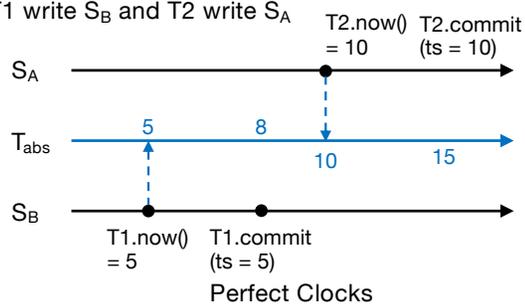


22

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

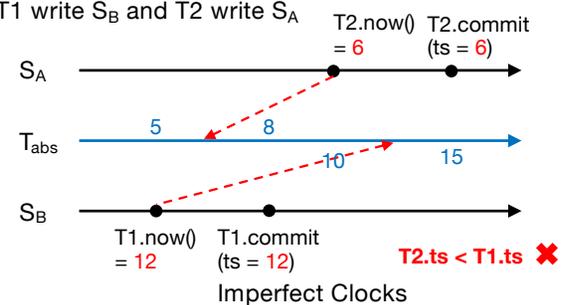


23

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

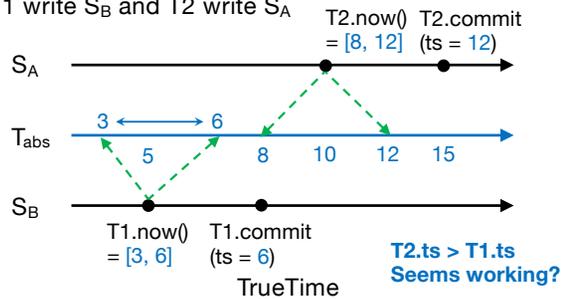


24

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

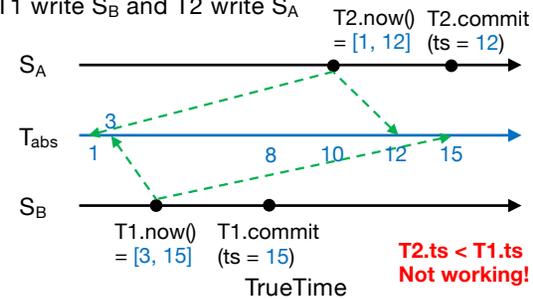


25

Enforcing the Invariant (Strawman)

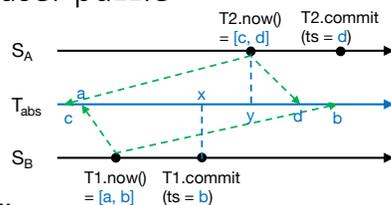
If T2 starts after T1 commits, then T2 must have larger timestamp

Let T1 write S_B and T2 write S_A



26

A brain teaser puzzle



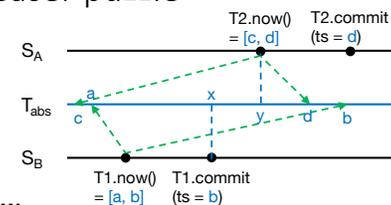
We know.

- $x < y$, because T2 in real-time after T1 (the assumption)
- $c \leq y \leq d$, because TrueTime
- $T1.ts = b$, $T2.ts = d$, because how ts is assigned

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

27

A brain teaser puzzle



We know.

- $x < y$, because T2 in real-time after T1 (the assumption)
- $c \leq y \leq d$, because TrueTime
- $T1.ts = b$, $T2.ts = d$, because how ts is assigned

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

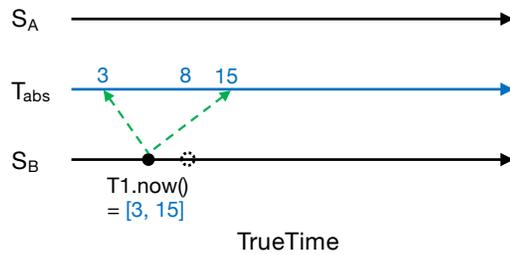
1 and 2 $\rightarrow x < d$; we need to ensure $b < x$; then $b < x < d$, done.

28

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

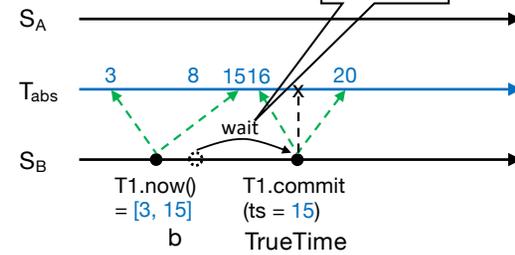


29

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

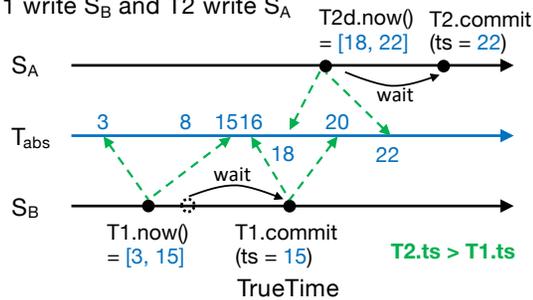


30

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



31

Takeaways

- The invariant is always enforced: If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp
- How big/small ϵ 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
- ϵ must be known a priori and small so commit wait is doable!

32

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`?