

External Consistency and Spanner

CS425/ECE428 — SPRING 2020

NIKITA BORISOV, UIUC



Transactions so far


Objects distributed / partitioned among different servers

- For load balancing (sharding)
- For separation of concerns / administration

Isolation enforced using two-phase locking (2PL)

- Each server maintains locks on own objects
- Deadlocks detected using e.g., edge-chasing

Atomic commit using 2PC

- Prepare to commit ensures durability
 - Recover from coordinator and participant crashes
- 

Dealing with Failures

Node failure

- Objects *unavailable* until recovery
- 2PC “stuck” after coordinator failure

But! Node failure is common

Drive failures => no recovery!

In each cluster's first year, it's typical that 1,000 individual machine failures will occur; thousands of hard drive failures will occur; one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours; 20 racks will fail, each time causing 40 to 80 machines to vanish from the network; 5 racks will "go wonky," with half their network packets missing in action; and the cluster will have to be rewired once, affecting 5 percent of the machines at any given moment over a 2-day span, Dean said. And there's about a 50 percent chance that the cluster will overheat, taking down most of the servers in less than 5 minutes and taking 1 to 2 days to recover.

Replication

Objects **distributed** among 1000's cluster nodes for load-balancing (sharding)


Objects **replicated** among a handful of nodes for availability / durability

- Replication across data centers, too

Two-level operation:

- Use transactions, coordinators, 2PC per object
- Use Paxos / Raft among object replicas

Note: can be expensive!

- Coordinator sends Prepare message to leaders of each replica group
 - Each leader uses Paxos / Raft to commit the Prepare to the group logs
 - Once commit succeeds, reply to coordinator
 - Coordinator uses Paxos / Raft to commit decision to *its* group log
- 

Example transaction

read A -> acquire read lock on A

read B -> acquire read lock on B

write A -> promote A's lock to write lock

commit -> perform 2PC

- Coordinator -> A, B: prepare
- A, B -> OK
- Coordinator -> A, B: commit

Read transactions

Read transactions often access many data items

- E.g., Facebook "news feed"
- E.g., Amazon front page
- E.g., balances across all accounts

Read transactions still need (read) locks (Why?)

Acquiring locks requires consensus (Why?)

Locks prevent write transactions from moving forward

Linearizability

Serial equivalence:

- Total effect on system is **equivalent** to a **run** that is serial and consistent with **each client's order**

Linearizability

- Total effect on system is **equivalent** to a **run** that is serial and consistent with **actual order** of events

E.g., buying a movie

- Client makes RPC to bank transfers \$3.99 to Amazon account
- Client requests video from Amazon
- Amazon makes RPC to bank, does not see transfer, rejects request!

Spanner: Google's Globally-Distributed Database

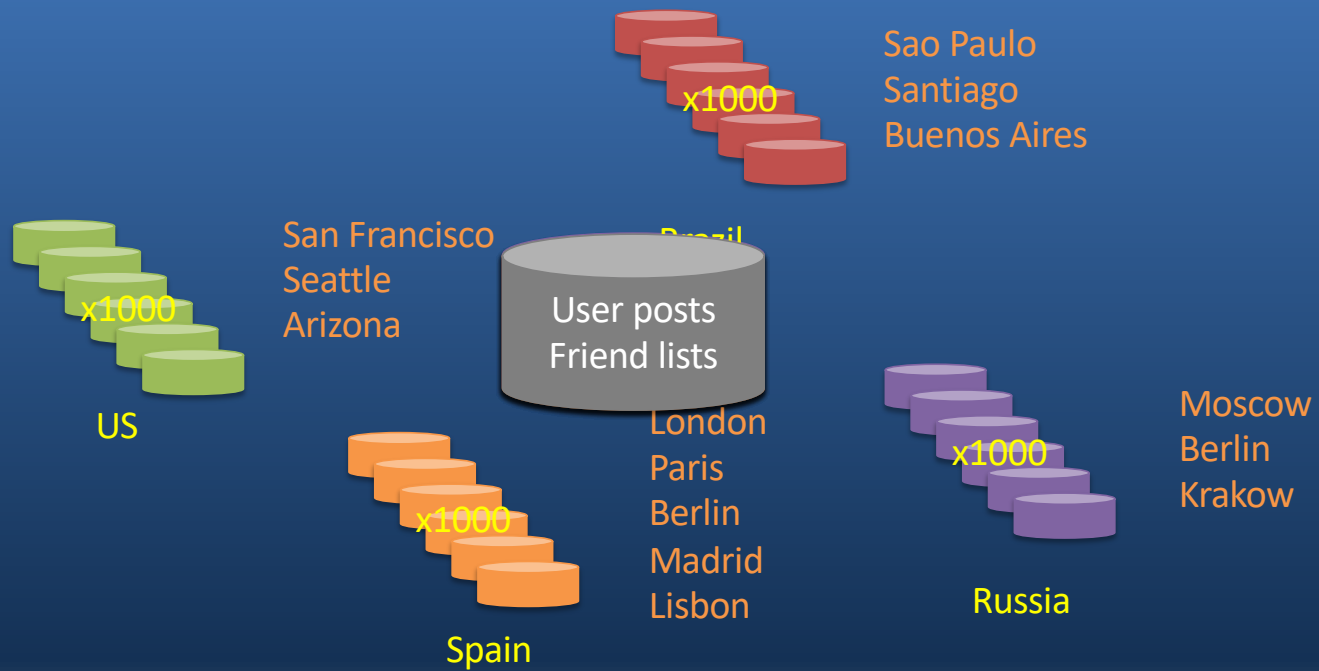
Wilson Hsieh
representing a host of authors
OSDI 2012



What is Spanner?

- Distributed multiversion database
 - General-purpose transactions (ACID)
 - SQL query language
 - Schematized tables
 - Semi-relational data model
- Running in production
 - Storage for Google's ad data
 - Replaced a sharded MySQL database

Example: Social Network



Overview

- Feature: Lock-free distributed read transactions
- Property: External consistency of distributed transactions
 - First system at global scale
- Implementation: Integration of concurrency control, replication, and 2PC
 - Correctness and performance
- Enabling technology: TrueTime
 - Interval-based global time

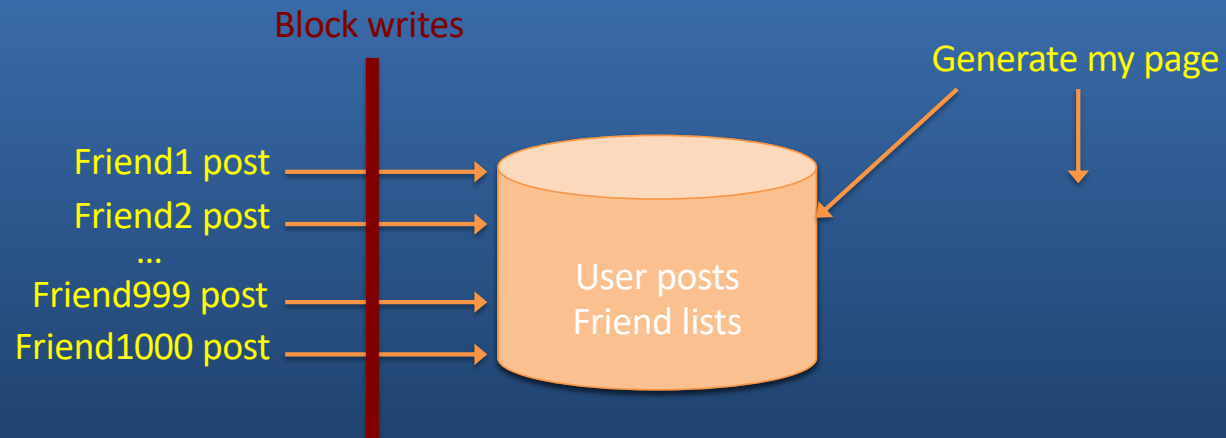
Read Transactions

- Generate a page of friends' recent posts
 - Consistent view of friend list and their posts

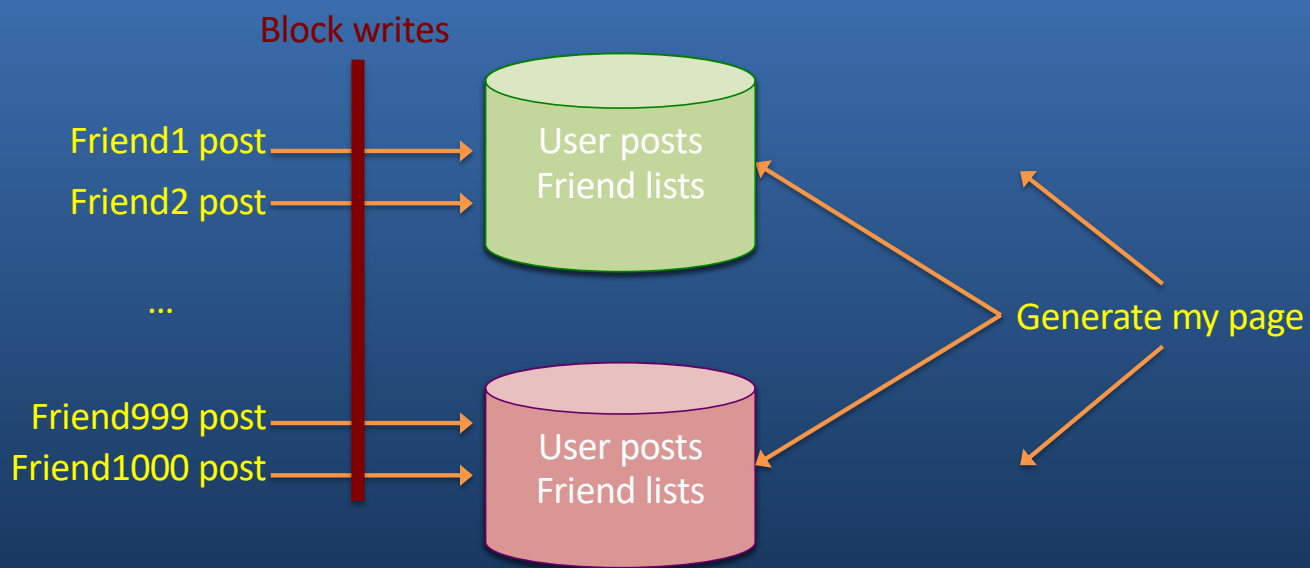
Why consistency matters

1. Remove untrustworthy person X as friend
2. Post P: "My government is repressive..."

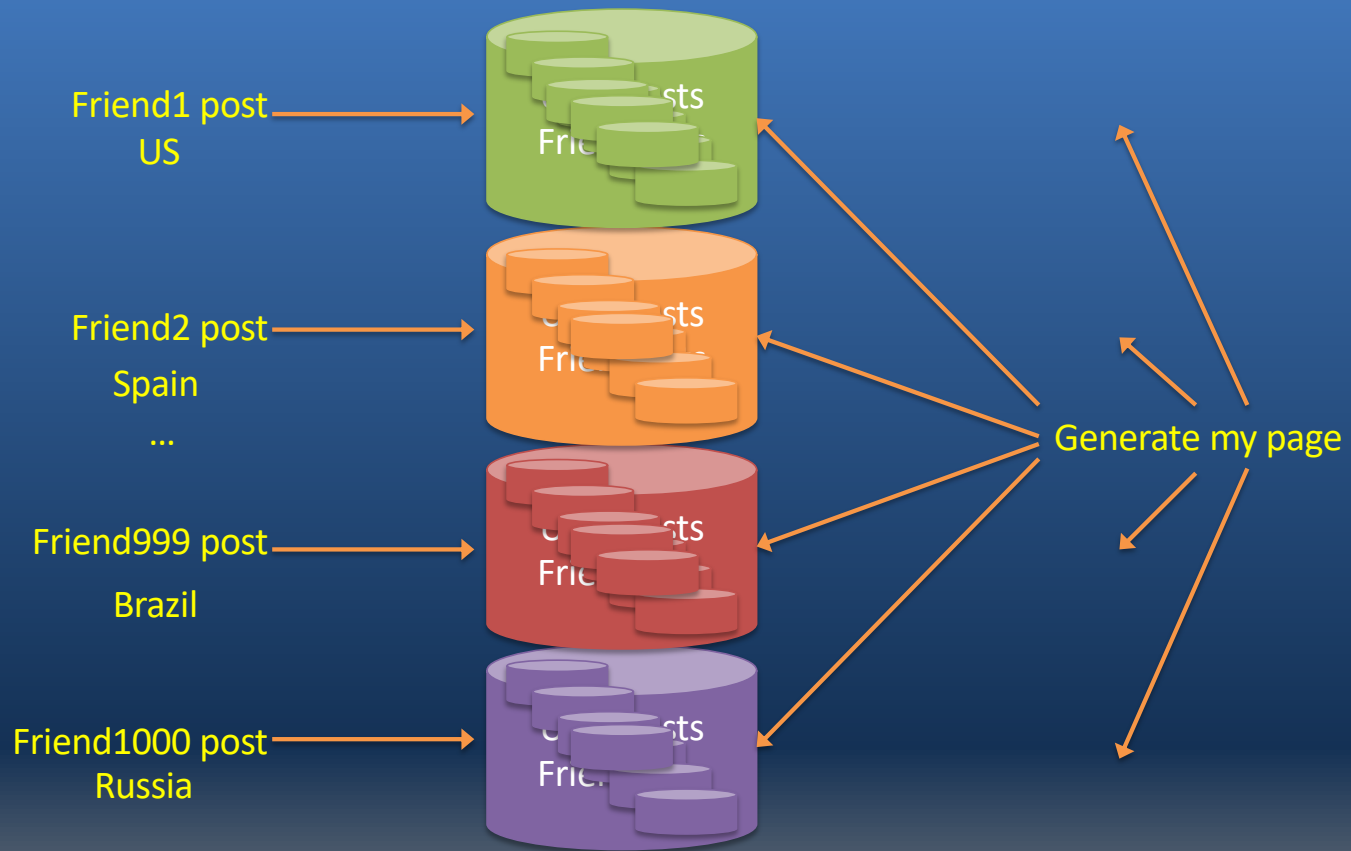
Single Machine



Multiple Machines



Multiple Datacenters



Version Management

- Transactions that write use strict 2PL
 - Each transaction T is assigned a timestamp s
 - Data written by T is timestamped with s

Time	<8	8	15
My friends	[X]	[]	
My posts			[P]
X's friends	[me]	[]	

Synchronizing Snapshots

Global wall-clock time

==

External Consistency:

Commit order respects global wall-time order

==

Timestamp order respects global wall-time order

given

timestamp order == commit order

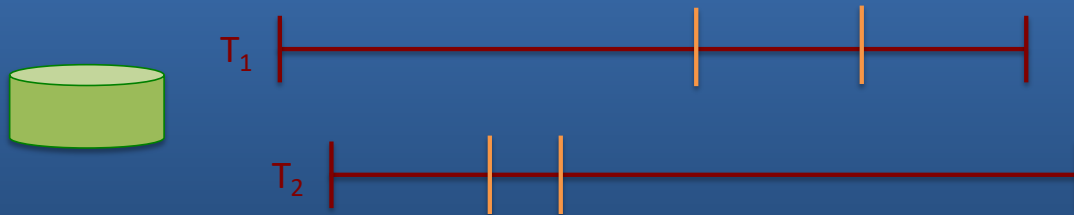
Timestamps, Global Clock

- Strict two-phase locking for write transactions
- Assign timestamp while locks are held

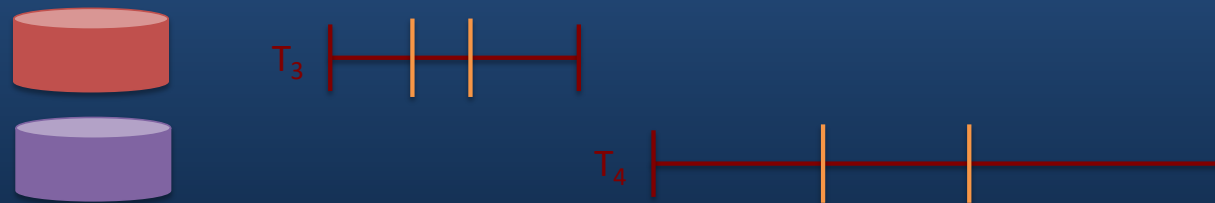


Timestamp Invariants

- Timestamp order == commit order

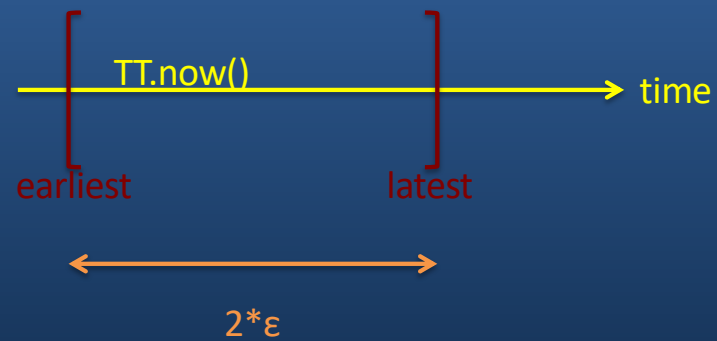


- Timestamp order respects global wall-time order

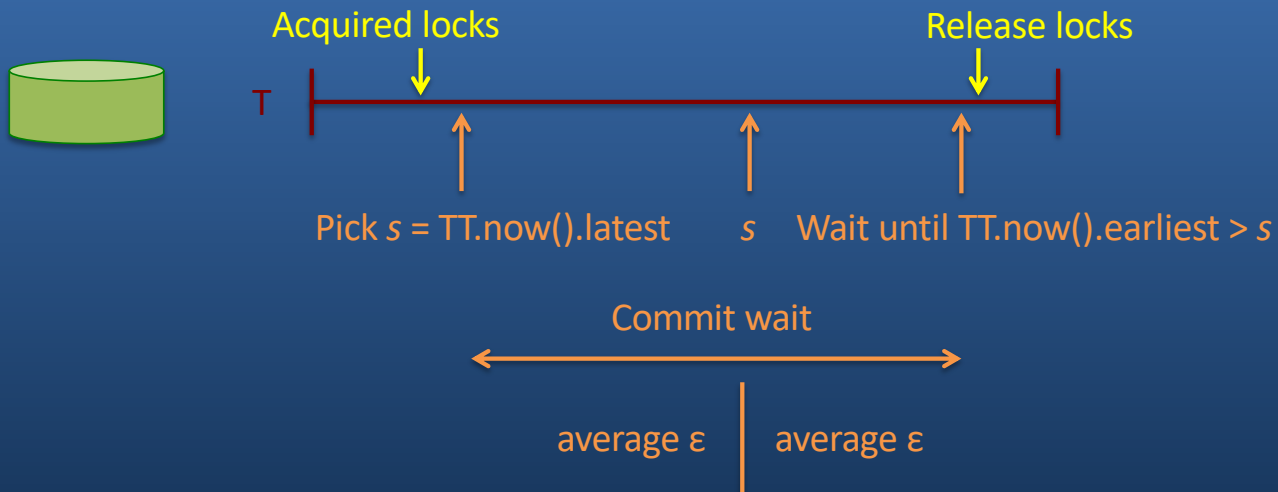


TrueTime

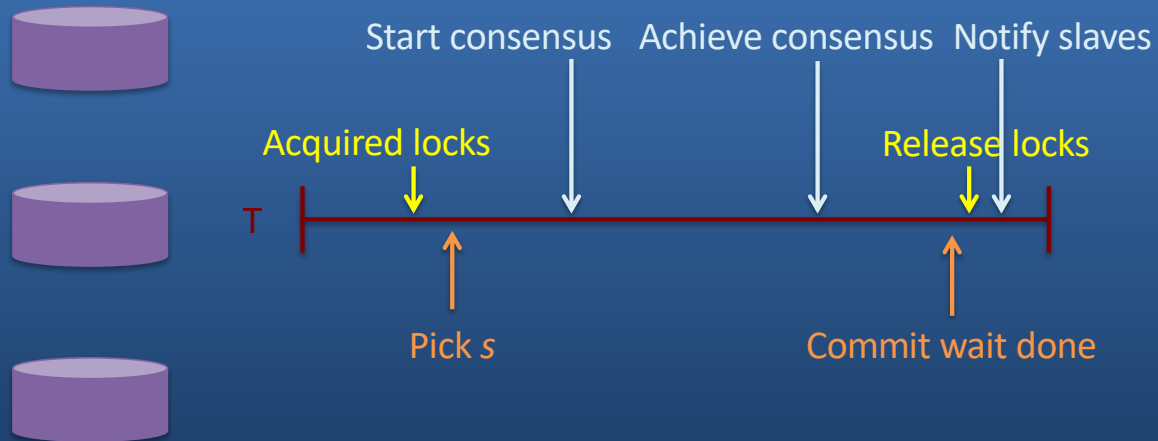
- “Global wall-clock time” with bounded uncertainty



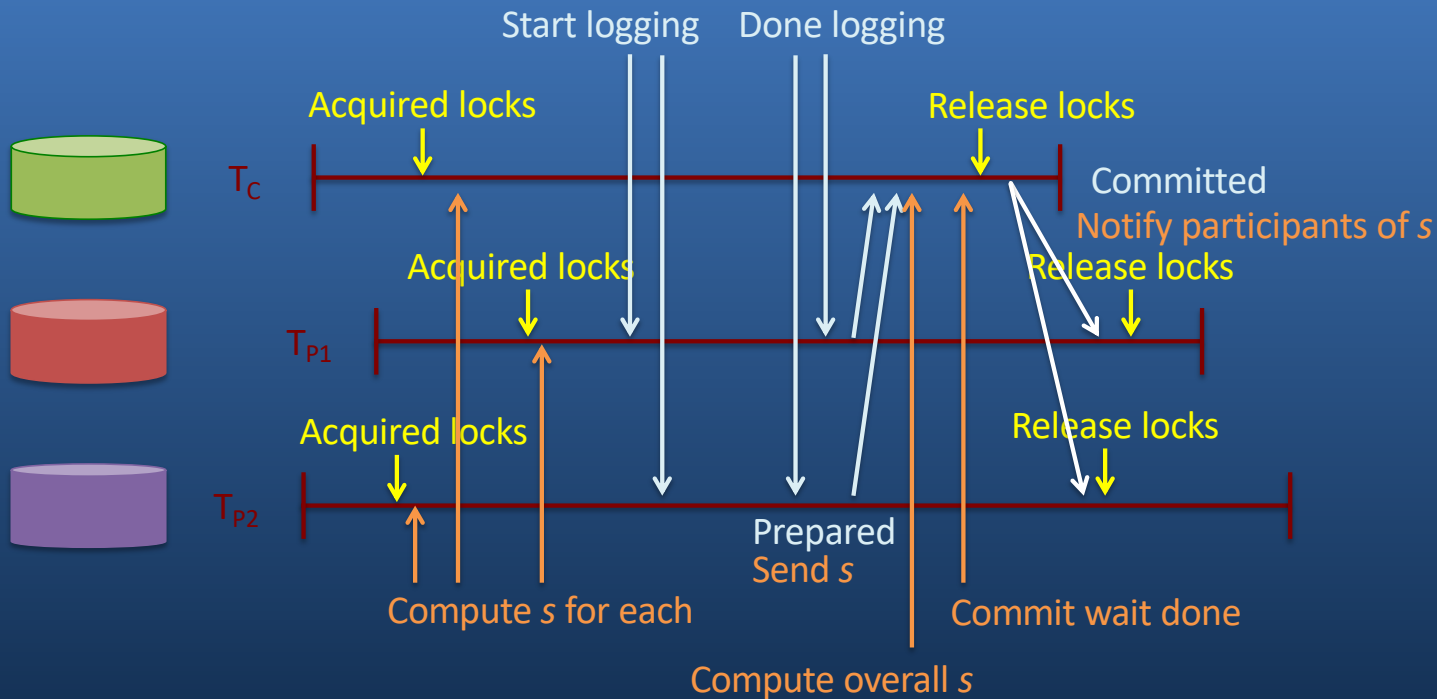
Timestamps and TrueTime



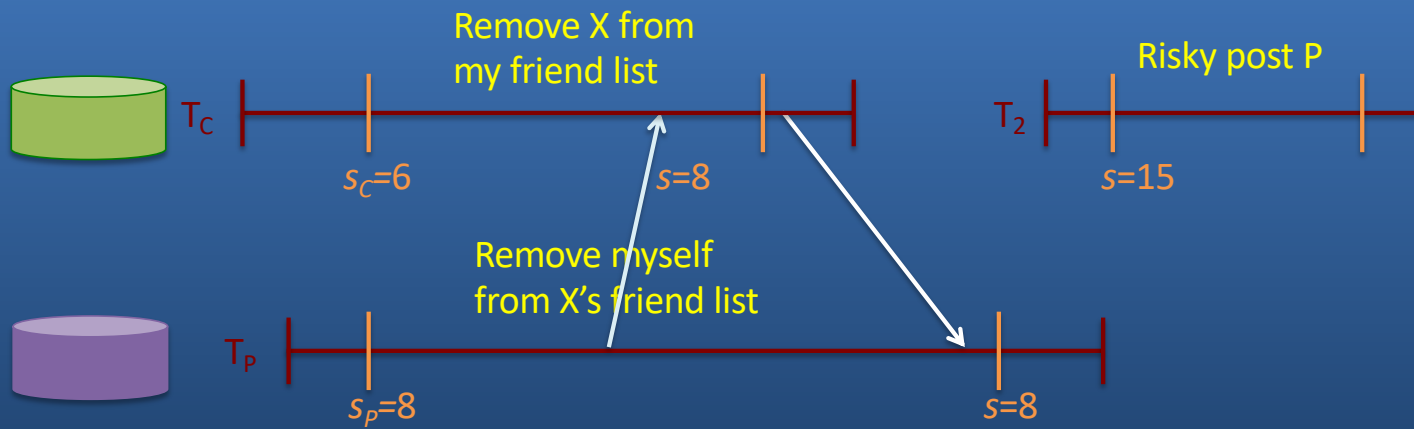
Commit Wait and Replication






Commit Wait and 2-Phase Commit



Example



	Time	<8	8	15
 My friends		[X]	[]	
 My posts				[P]
 X's friends		[me]	[]	

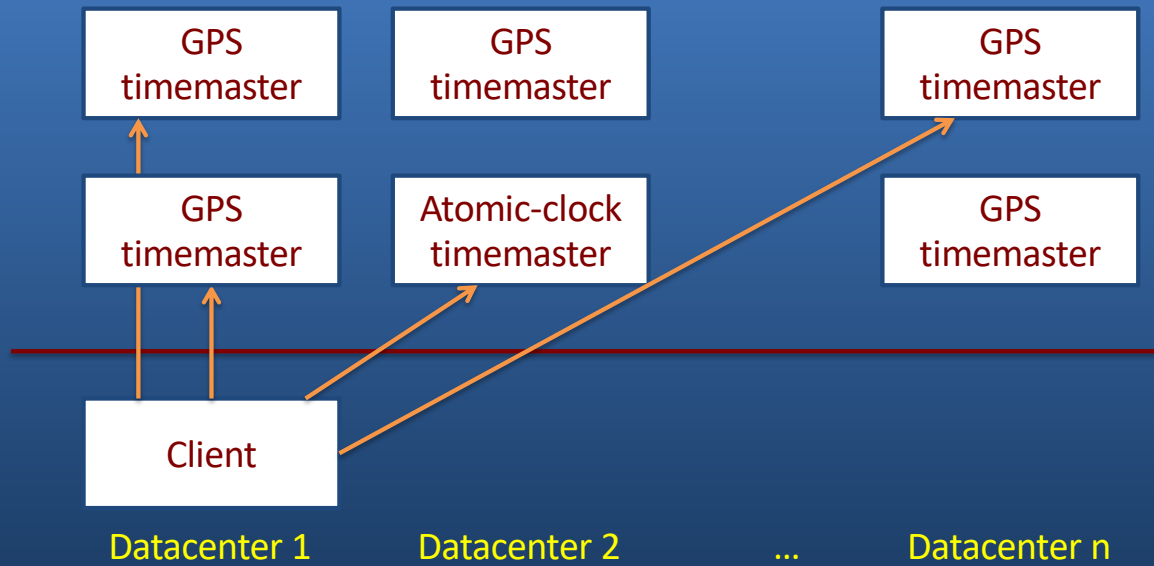
What Have We Covered?

- Lock-free read transactions across datacenters
- External consistency
- Timestamp assignment
- TrueTime
 - Uncertainty in time can be waited out

What Haven't We Covered?

- How to read at the present time
- Atomic schema changes
 - Mostly non-blocking
 - Commit in the future
- Non-blocking reads in the past
 - At any sufficiently up-to-date replica

TrueTime Architecture

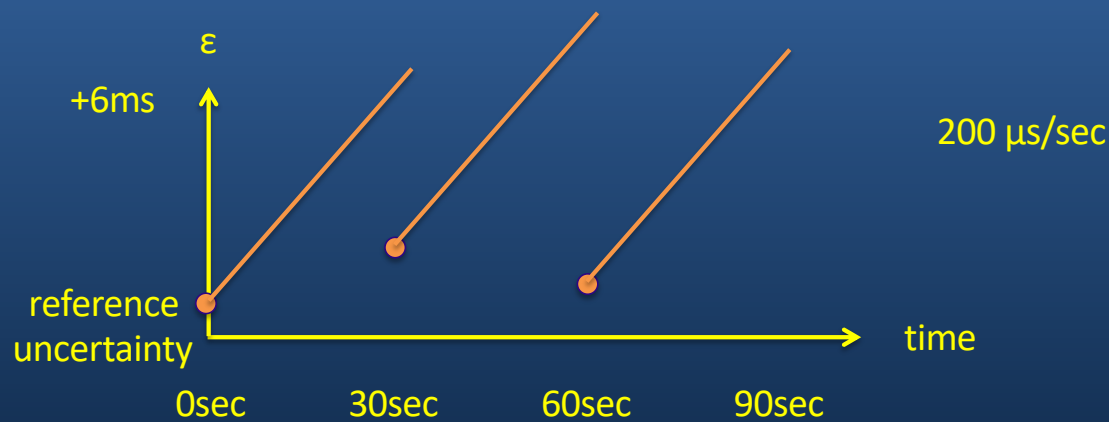


Compute reference [earliest, latest] = now $\pm \epsilon$

TrueTime implementation

$\text{now} = \text{reference now} + \text{local-clock offset}$

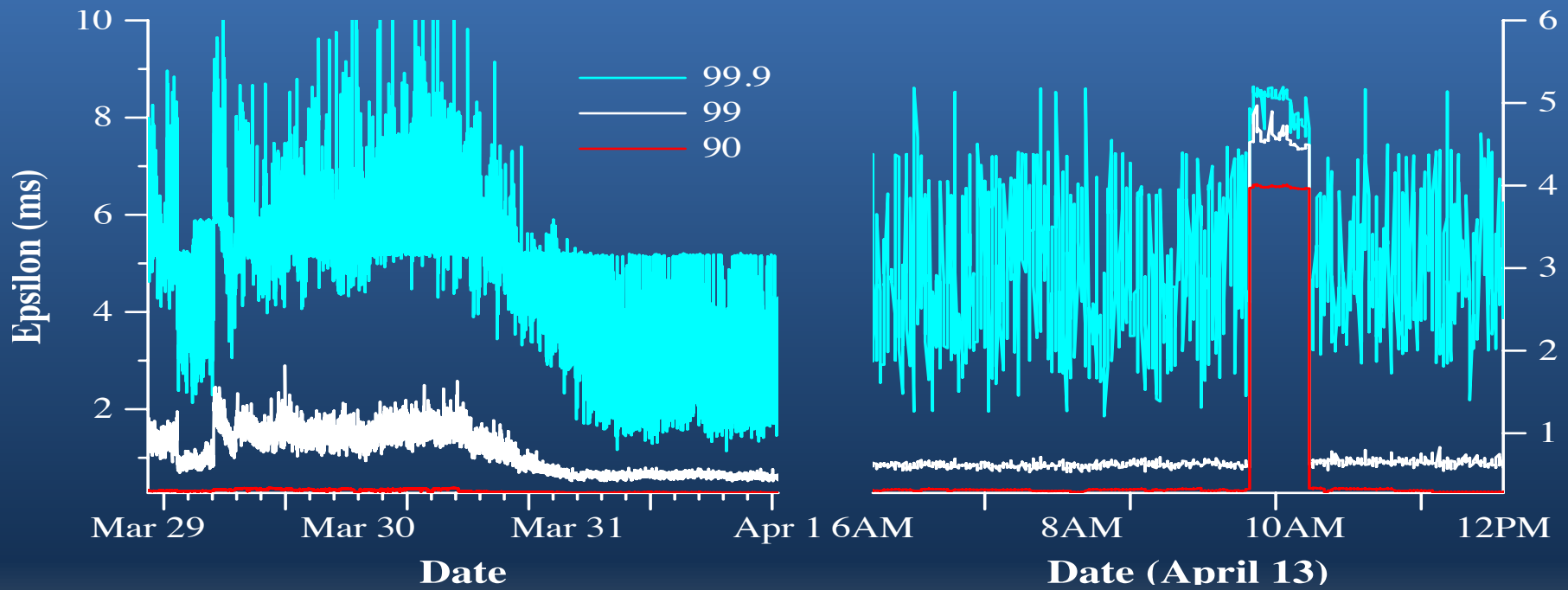
$\varepsilon = \text{reference } \varepsilon + \text{worst-case local-clock drift}$



What If a Clock Goes Rogue?

- Timestamp assignment would violate external consistency
- Empirically unlikely based on 1 year of data
 - Bad CPUs 6 times more likely than bad clocks

Network-Induced Uncertainty



Conclusions

- Reify clock uncertainty in time APIs
 - Known unknowns are better than unknown unknowns
 - Rethink algorithms to make use of uncertainty
- Stronger semantics are achievable
 - Greater scale != weaker semantics