

Distributed Systems

CS425/ECE428

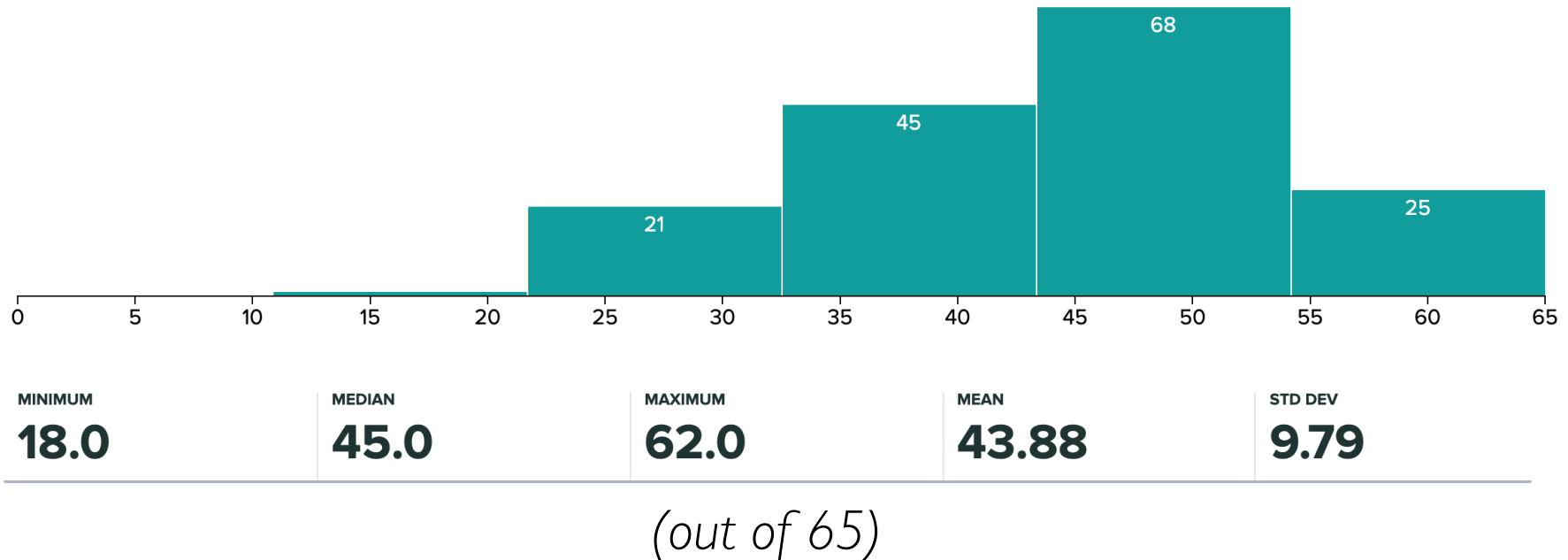
April 14 2021

Instructor: Radhika Mittal

Acknowledgements for the materials: Indy Gupta, Nikita Borisov, Spanner authors

Logistics

- Midterm 2 grades and solutions were released on Monday.



Logistics

- HW5 deadline has been extended to Friday, April 16th, 11:59pm.
- MP3 has been released.
- HW6 will be released on Friday.

Distributed Transactions

- Transaction processing can be *distributed* across multiple servers.
 - Different objects can be stored on different servers.
 - An object may be replicated across multiple servers.
 - Our focus today.
- Case study: Google's Spanner System

Distributed Transactions

- *Sharding*: objects can be distributed across multiple (1 000's of) servers
 - Primary reason: load balancing and scalability.
- *Replication*: the same object may be replicated among a handful of nodes.
 - Primary reason: fault-tolerance, availability, durability.

Replication: Natural way to handle failures

- Node failures are common.

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. 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.

-- Jeff Dean (Google), source: cnet.com

Replication: Natural way to handle failures

- Node failures are common.
- What could happen if a node fails?
 - Objects unavailable until recovery.
 - 2PC “stuck” after coordinator failure
- Even worse: what happens if the drive failures.
 - no recovery!
- Replication provides greater availability and robustness to failures.
 - Geo-replication (spanning datacenters across the world) for greater robustness.

Replication

- Replication = An object has identical copies, each maintained by a separate server.
 - Copies are called “replicas”
- With k replicas of each object, can tolerate failure of any $(k-1)$ servers in the system

Replication: Availability

- If each server is down a fraction f of the time
 - Server's failure probability
- With no replication, availability of object =
= Probability that single copy is up
= $(1 - f)$
- With k replicas, availability of object =
Probability that at least one replicas is up
= $1 - \text{Probability that all replicas are down}$
= $(1 - f^k)$

Replication: Availability

- With no replication, availability of object =
= Probability that single copy is up
= $(1 - f)$
- With k replicas, availability of object =
Probability that at least one replicas is up
= $1 - \text{Probability that all replicas are down}$
= $(1 - f^k)$

f=failure probability	No replication	$k=3$ replicas	$k=5$ replicas
0.1	90%	99.9%	99.999%
0.05	95%	99.9875%	6 Nines
0.01	99%	99.9999%	10 Nines

Replication: Challenges

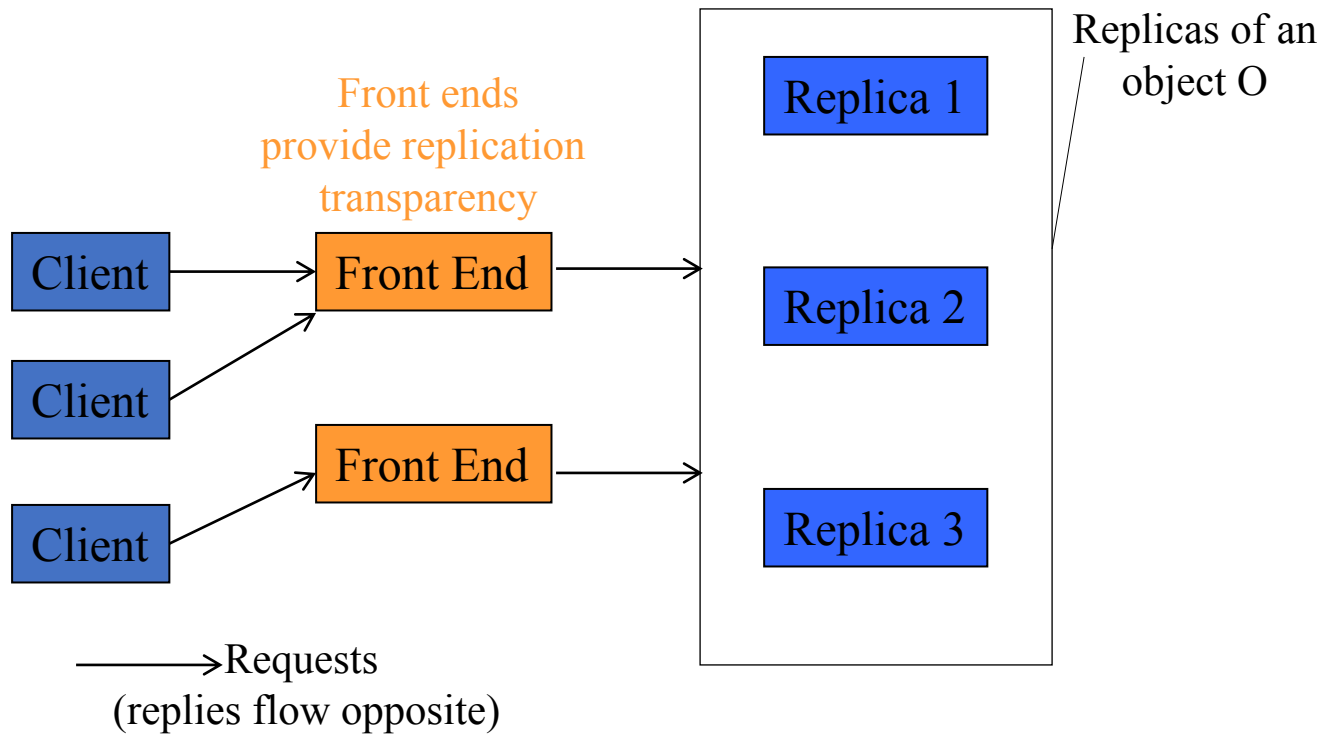
1. Replication Transparency

- A client ought not to be aware of multiple copies of objects existing on the server side

2. Replication Consistency

- All clients see single consistent copy of data, in spite of replication
- For transactions, guarantee ACID

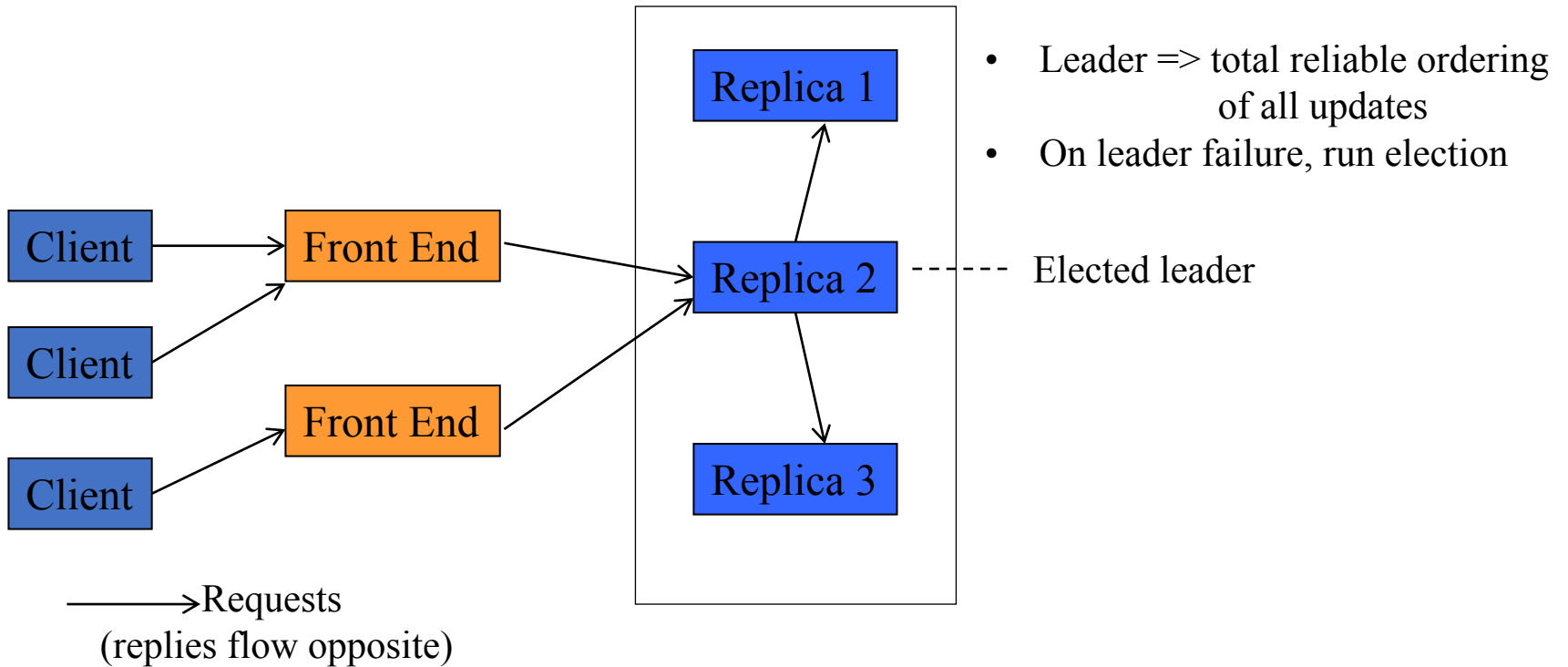
Replication Transparency



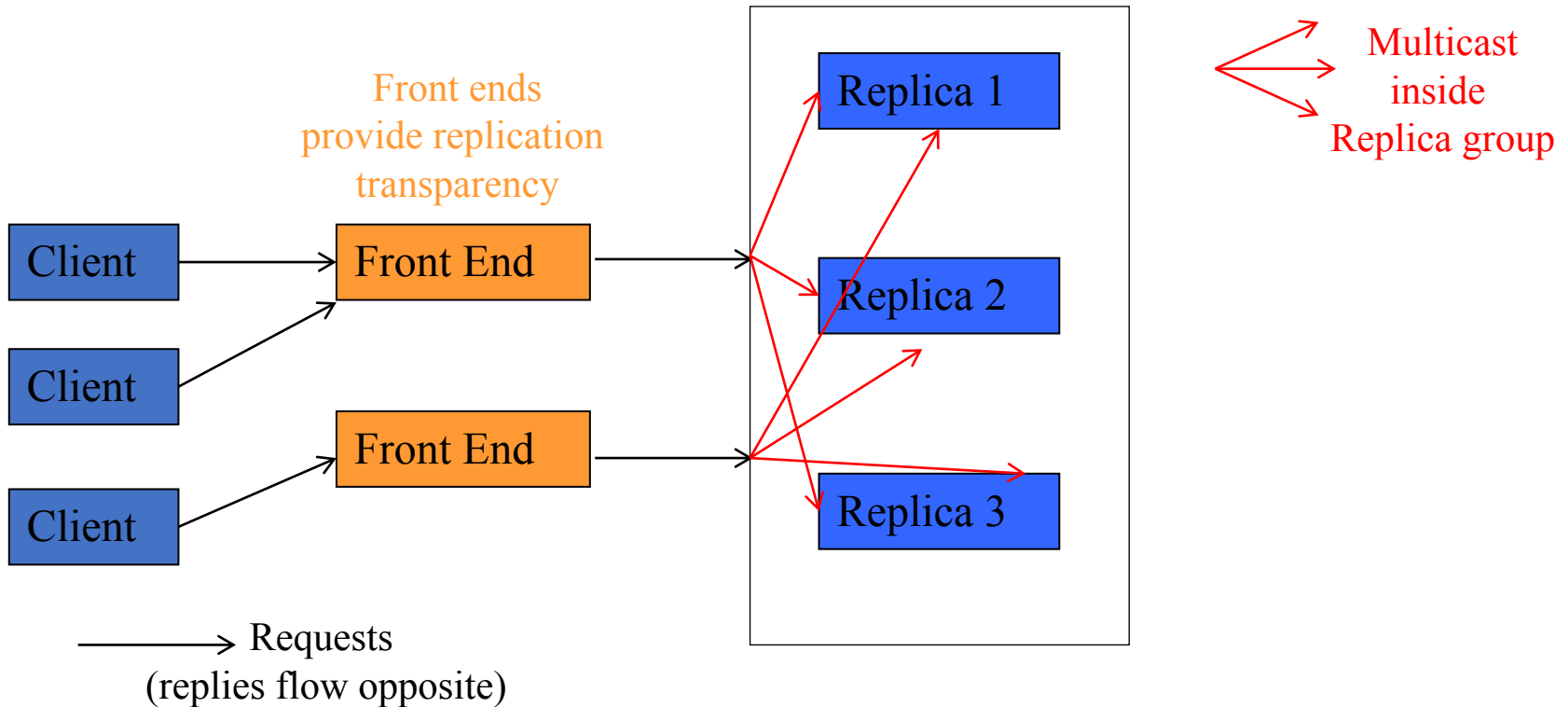
Replication Consistency

- Two ways to forward updates from front-ends (FEs) to replica group
 - **Passive Replication**: uses a primary replica (leader)
 - **Active Replication**: treats all replicas identically
- Both approaches use the concept of “**Replicated State Machines**”
 - Each replica's code runs the same state machine
 - *Multiple copies of the same State Machine begun in the Start state, and receiving the same Inputs in the same order will arrive at the same State having generated the same Outputs. [Schneider 1990]*

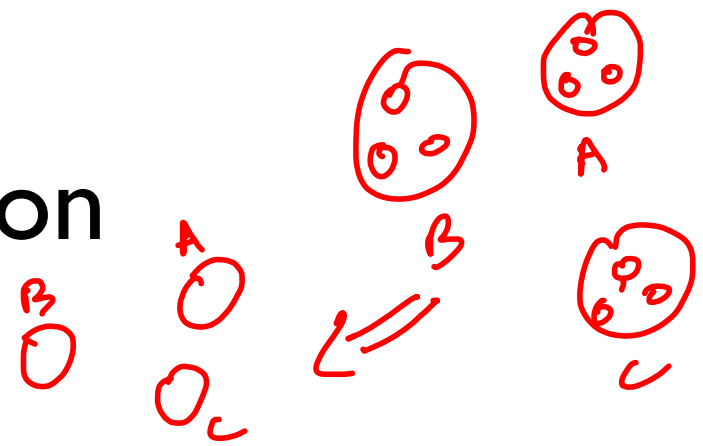
Passive Replication



Active Replication



Transactions and Replication



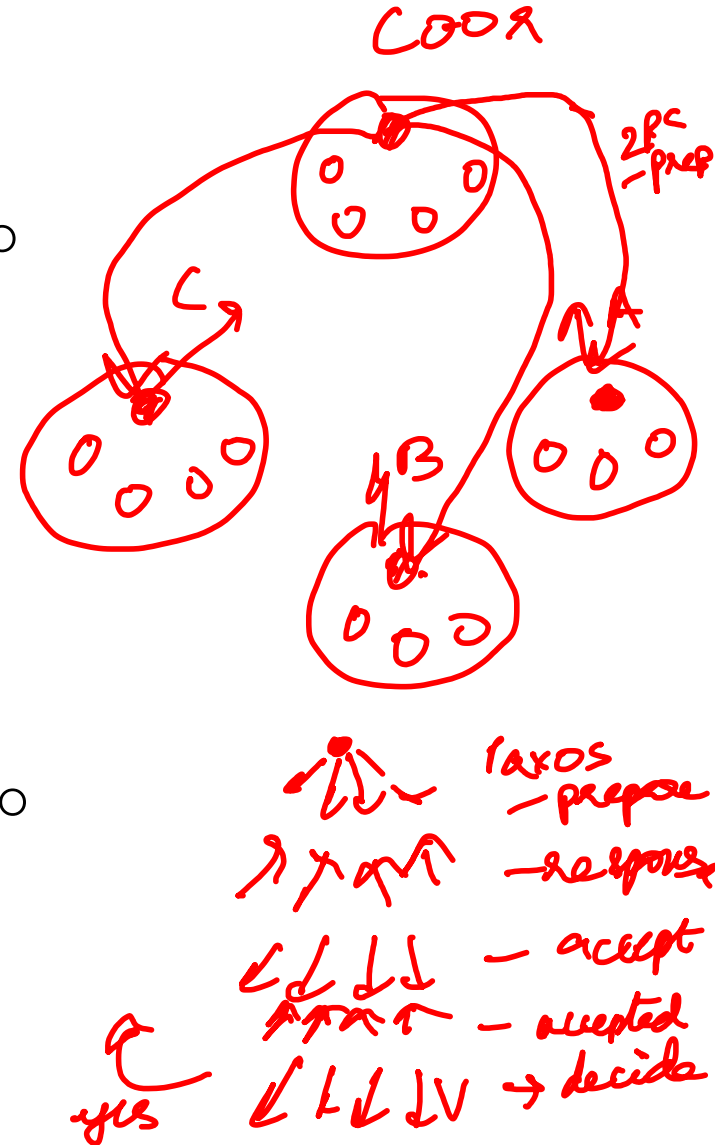
- One-copy serializability
 - A concurrent execution of transactions in a replicated database is one-copy-serializable if it is equivalent to a serial execution of these transactions over a single logical copy of the database.
 - (Or) The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at a time on a single set of objects (i.e., 1 replica per object).
- In a non-replicated system, transactions appear to be performed one at a time in some order.
 - Correctness means **serial equivalence** of transactions
- When objects are replicated, transaction systems for correctness need one-copy serializability.

Transactions and 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
- Consensus needed across object replicas, e.g.
 - When acquiring locks and executing operations
 - When committing transactions

2PC and Paxos

- E.g. workflow:
 - Coordinator leader sends Prepare message to leaders of each replica group
 - Each replica leader uses Paxos to commit the Prepare to the group logs
 - Once commit prepare succeeds, reply to coordinator leader
 - Coordinator leader uses Paxos to commit decision to its group log.
 - Coordinator leader sends Commit message to leaders of each replica group.
 - Each replica leader uses Paxos to process the final commit.
 - Replica leader send the “commit ok / have committed” message back to coordinator.



Spanner: Google's Globally-Distributed Database

- First three lines from the paper:
 - Spanner is a scalable, globally-distributed database designed, built, and deployed at Google.
 - At the highest level of abstraction, it is a database that shards data across many sets of Paxos state machines in datacenters spread all over the world.
 - Replication is used for global availability and geographic locality; clients automatically failover between replicas.

Spanner: Google's Globally-Distributed Database

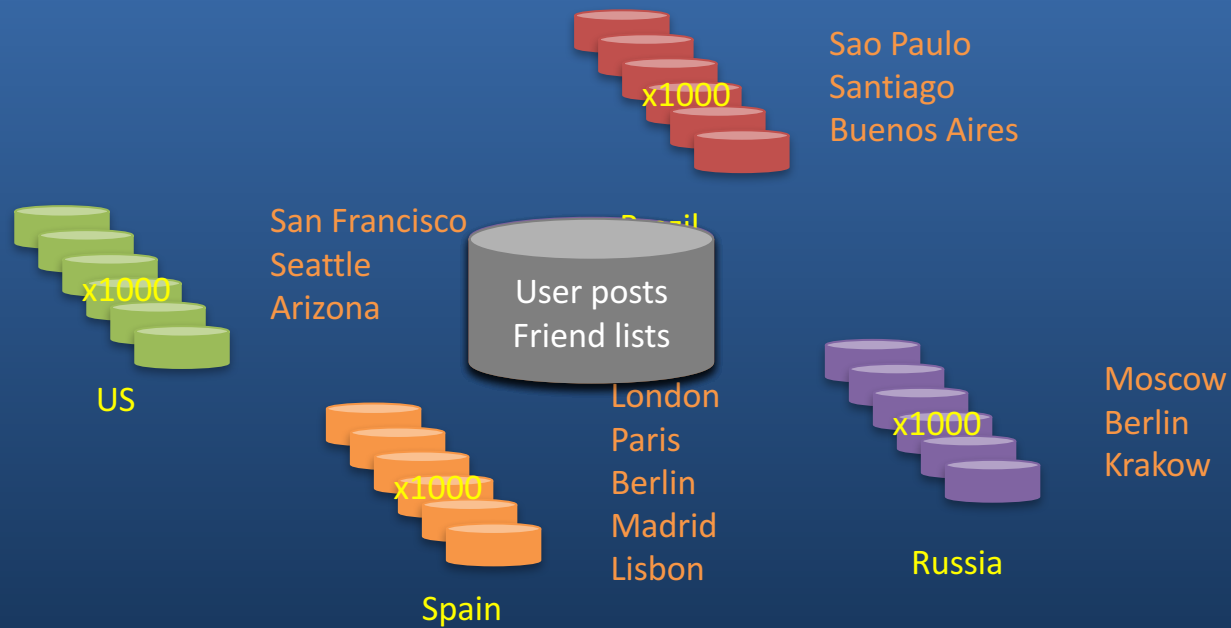
Wilson Hsieh
representing a host of authors
OSDI 2012



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

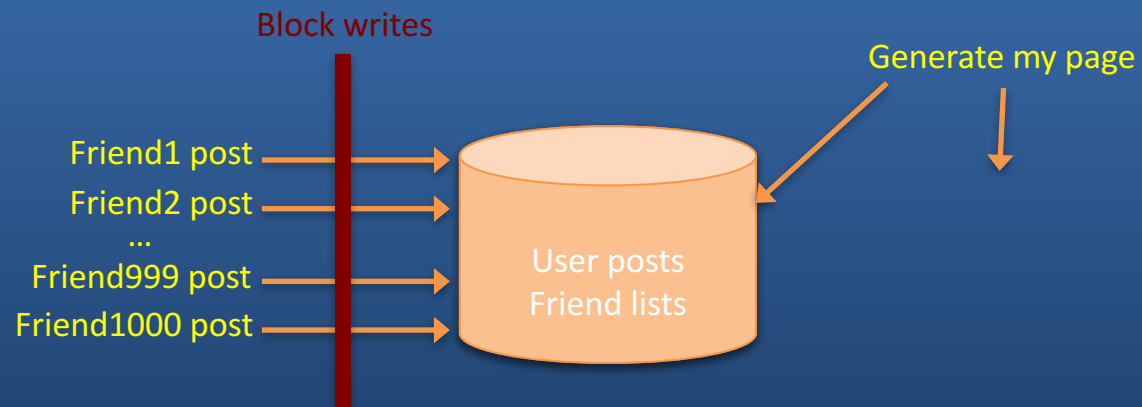
Read Transactions

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

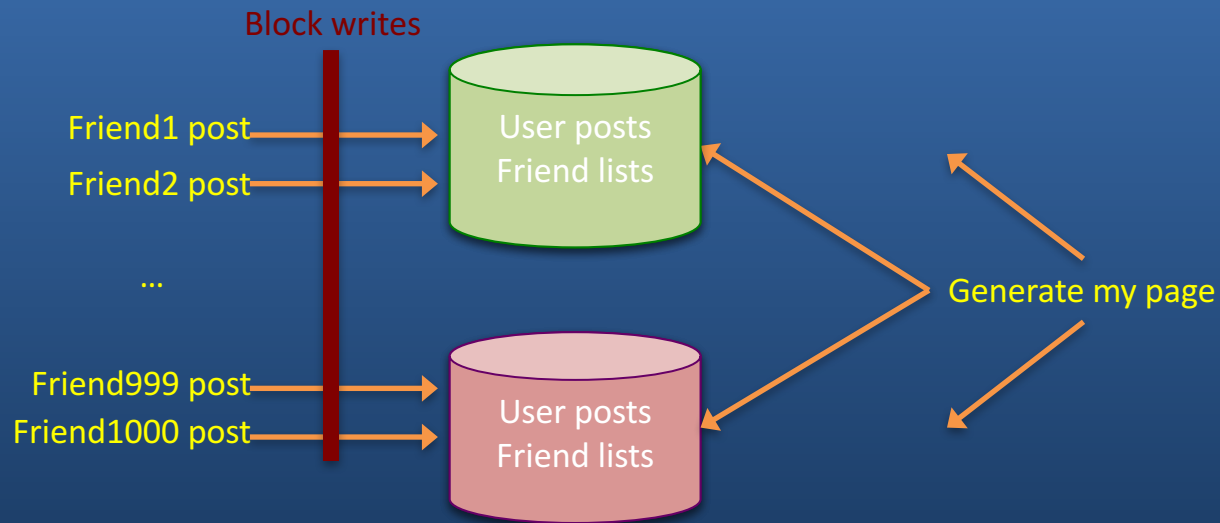
Why consistency matters

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

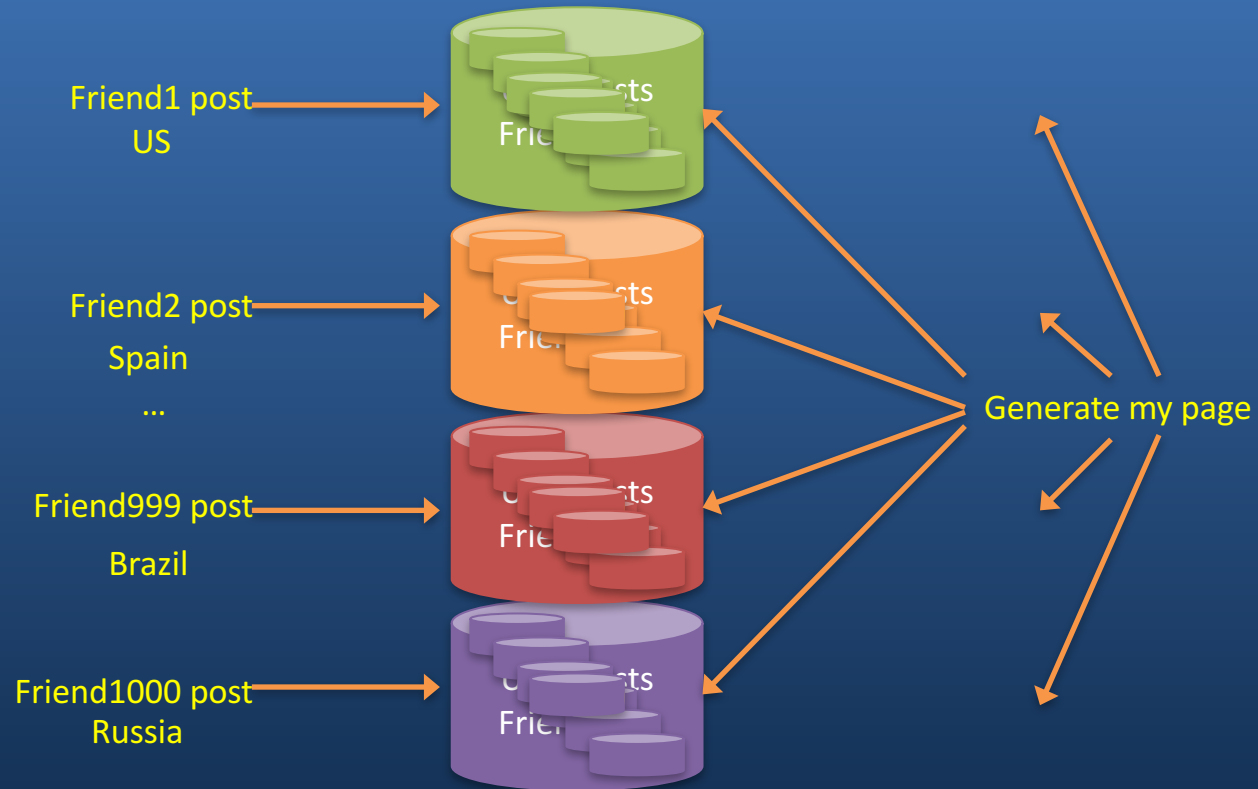
Single Machine



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

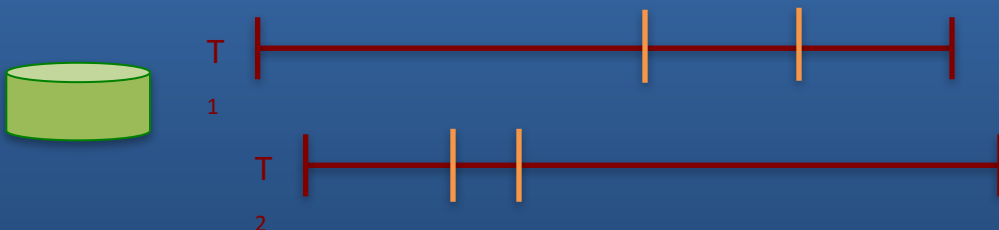
Timestamps, Global Clock

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



Timestamp Invariants

- Timestamp order == commit order

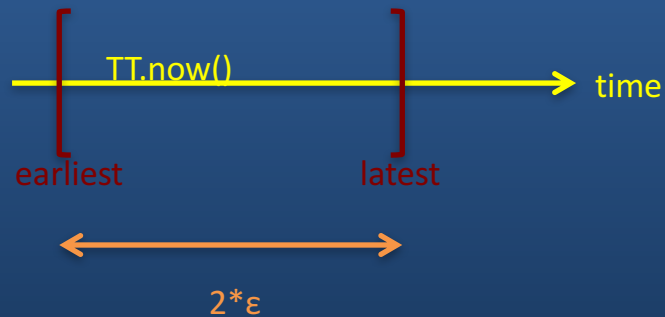


- Timestamp order respects global wall-time order

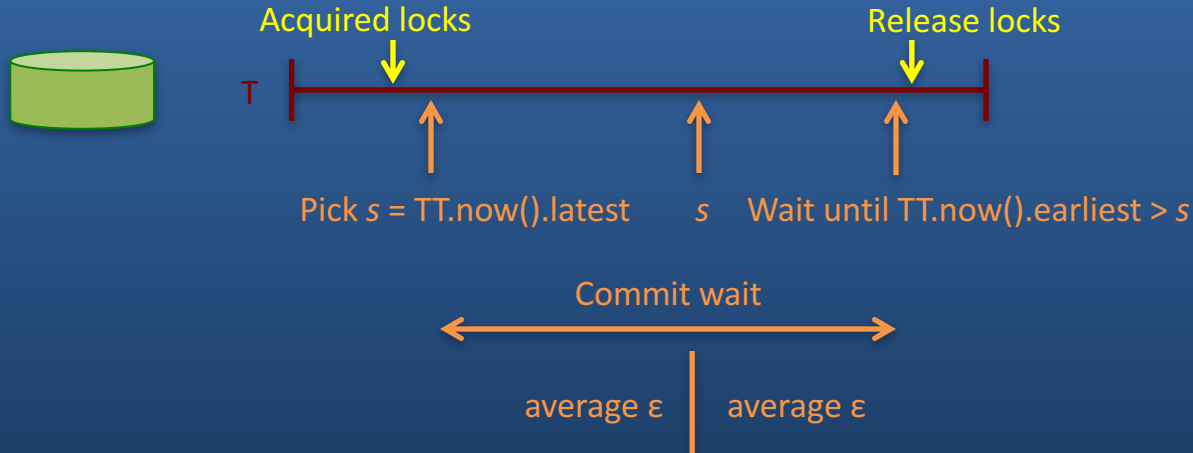


TrueTime

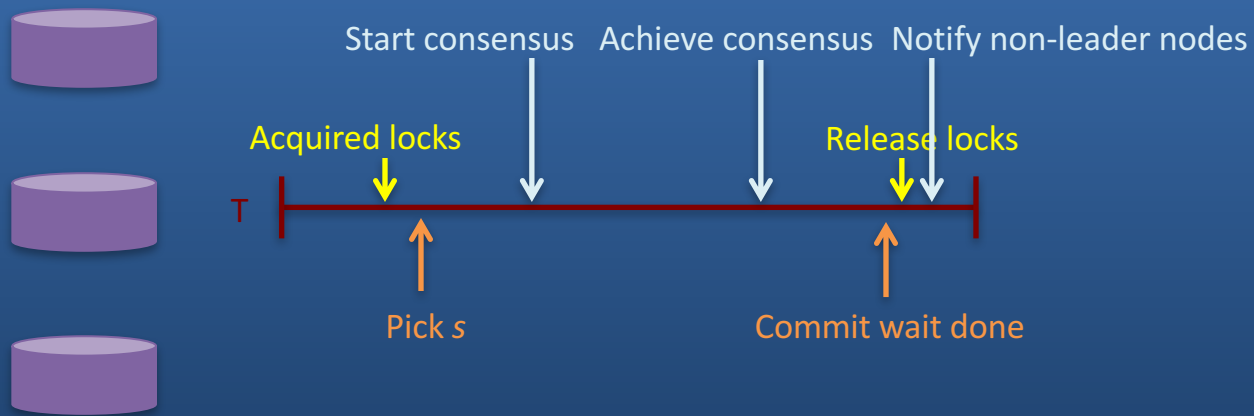
- “Global wall-clock time” with bounded uncertainty



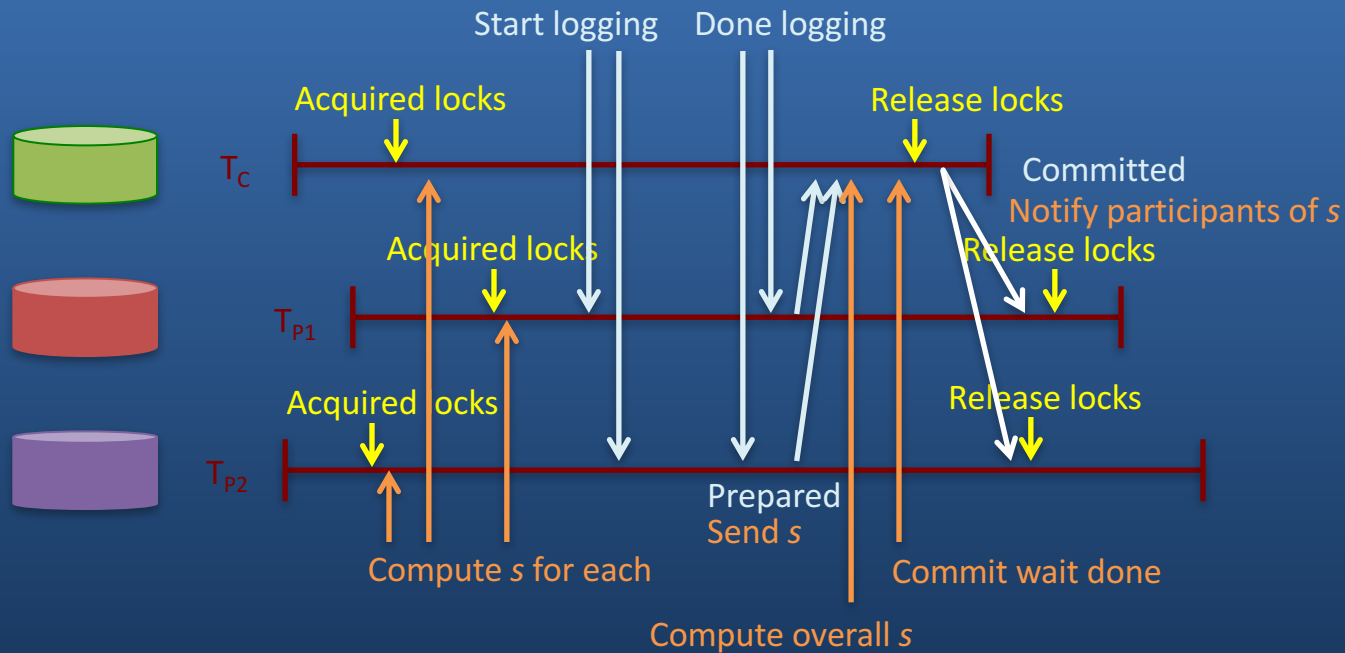
Timestamps and TrueTime



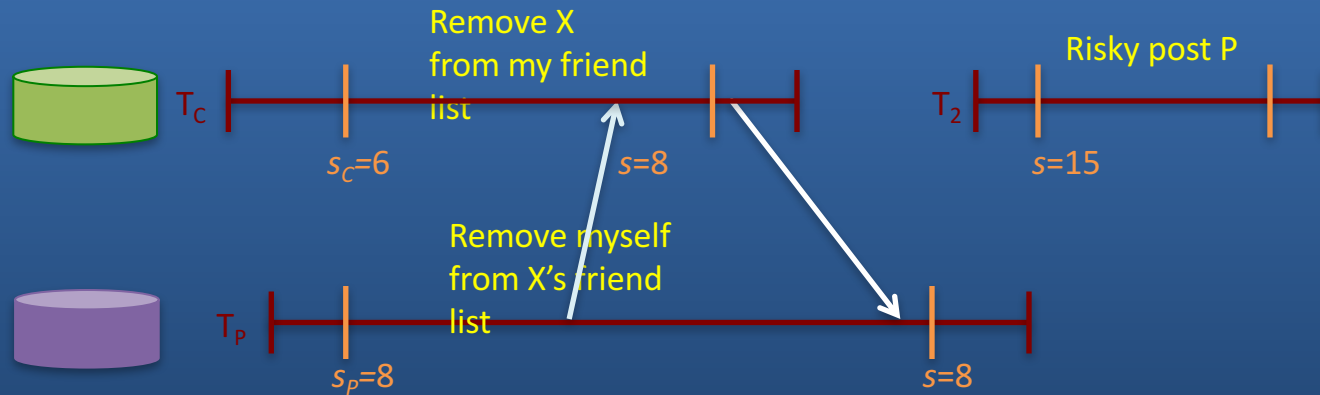
Commit Wait and Replication






Commit Wait and 2-Phase Commit



Example



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

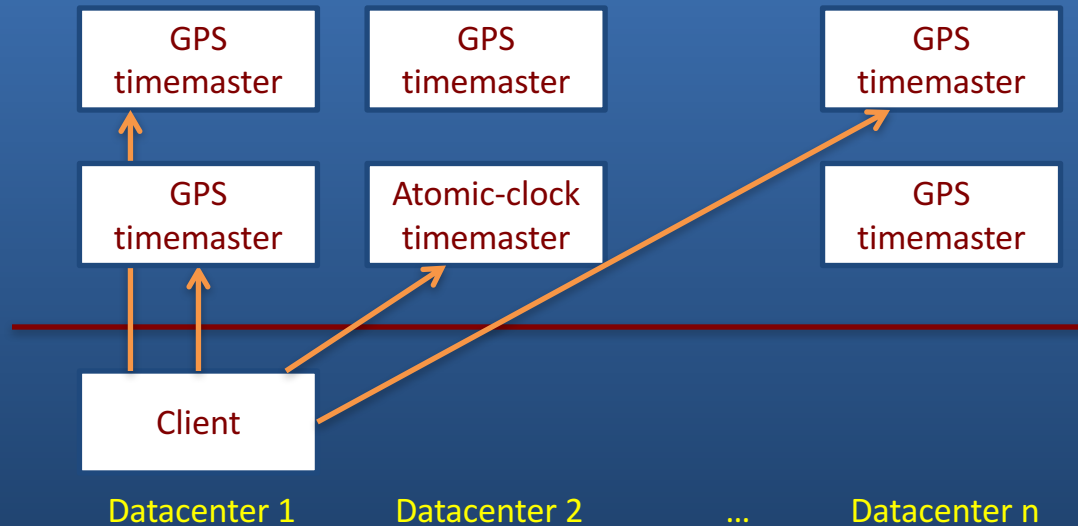
What Have We Covered?

- 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

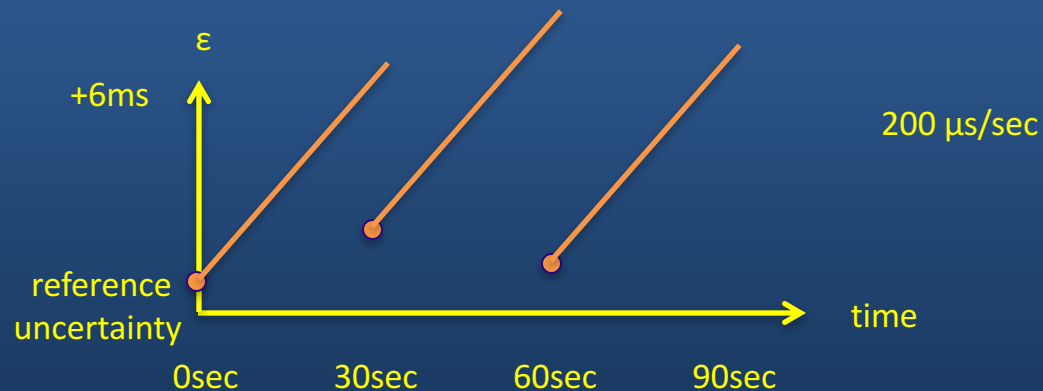


Compute reference [earliest, latest] = now $\pm \epsilon$

TrueTime implementation

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

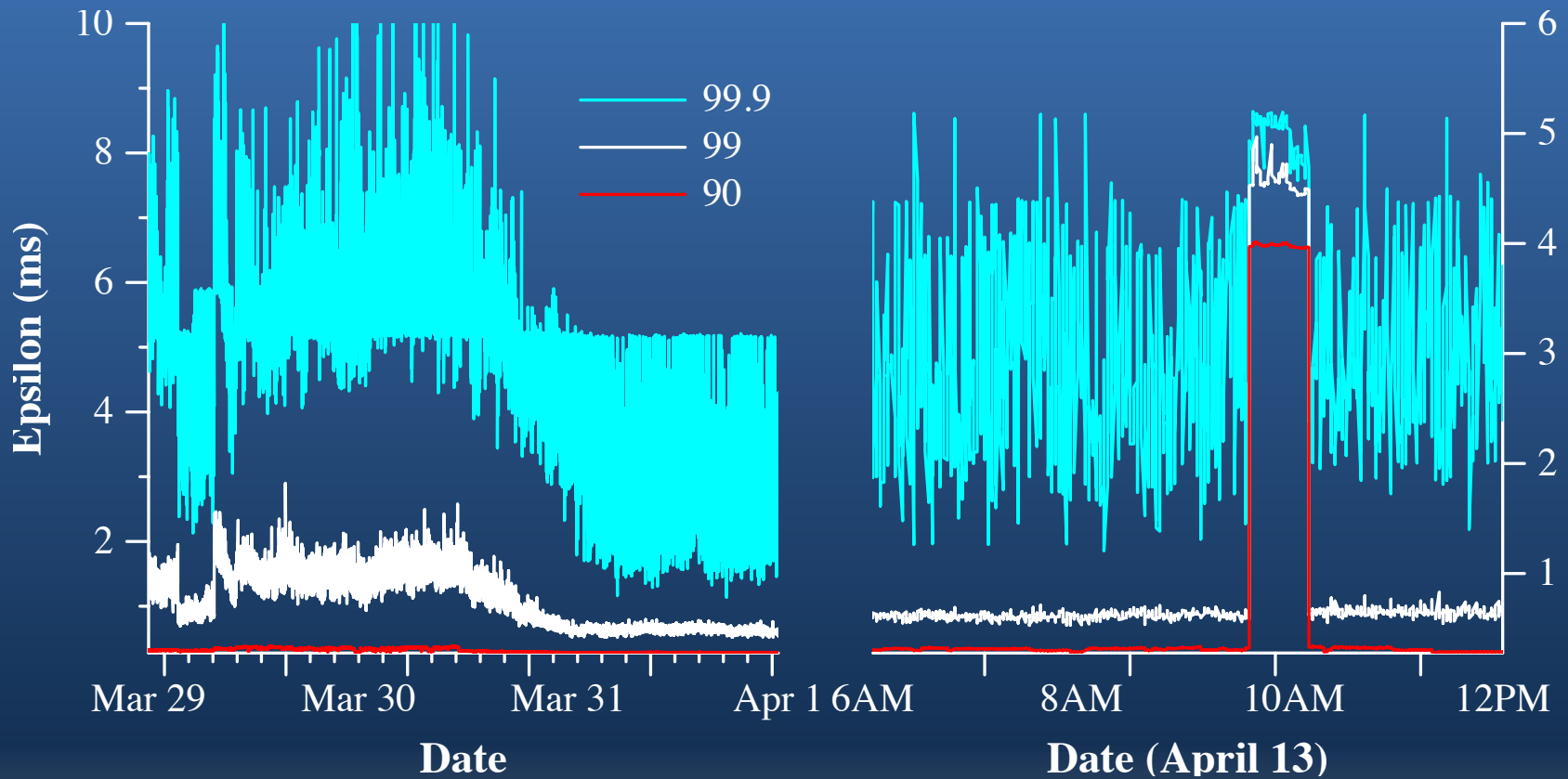
$\varepsilon = \text{reference } \varepsilon + \text{worst-case local-clock drift}$



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



What's in the Literature

- External consistency/linearizability
- Distributed databases
- Concurrency control
- Replication
- Time (NTP, Marzullo)

Future Work

- Improving TrueTime
 - Lower $\varepsilon < 1$ ms
- Building out database features
 - Finish implementing basic features
 - Efficiently support rich query patterns

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

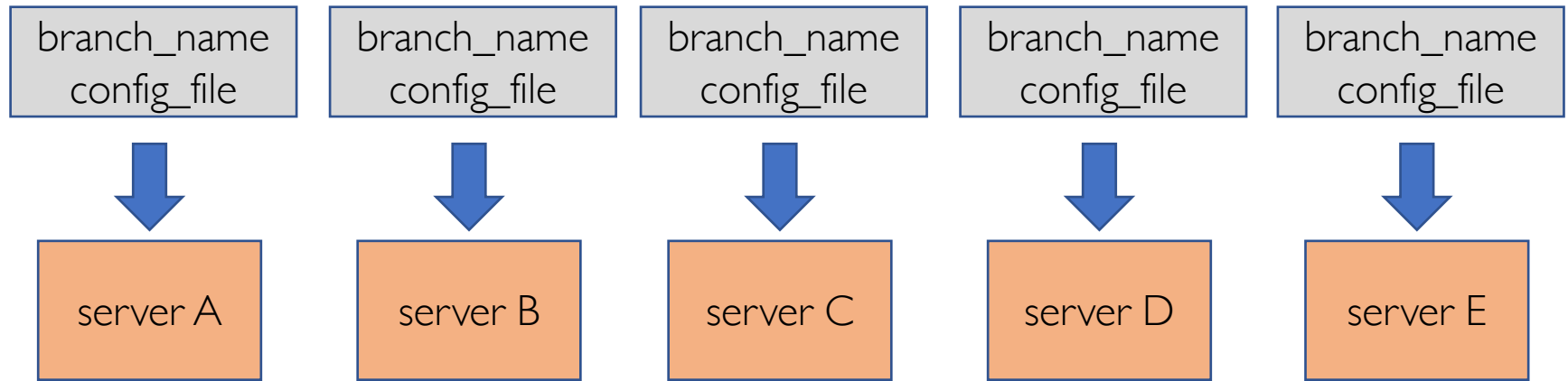
Thanks

- To the Spanner team and customers
 - To our shepherd and reviewers
 - To lots of Googlers for feedback
 - To you for listening!
-
- Questions?

MP3: Distributed Transactions

- <https://courses.grainger.illinois.edu/cs425/sp2021/mps/mp3.html>
- Lead TA: Dayue Bai
- Task:
 - Build a distributed transaction system that satisfies ACI properties (you do not need to handle Durability).
- Objective:
 - Think through and implement algorithms for achieving atomicity and consistency with distributed transactions (two-phase commit), concurrency control (two-phase locking / timestamped ordering), deadlock detection.

MP3: Distributed Transactions

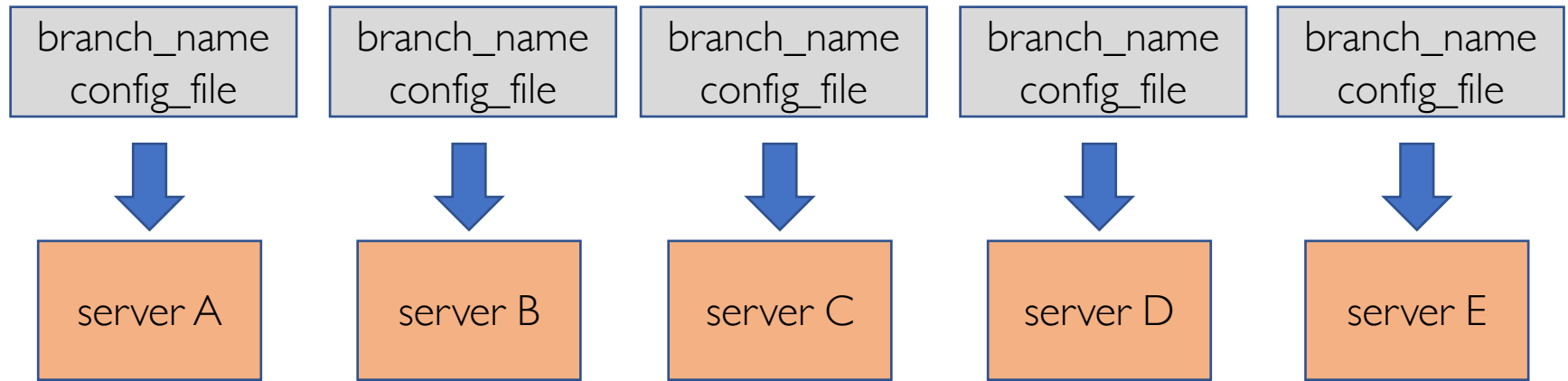


sample config_file

```
A sp21-cs425-g01-01.cs.illinois.edu 1234
B sp21-cs425-g01-02.cs.illinois.edu 1234
C sp21-cs425-g01-03.cs.illinois.edu 1234
D sp21-cs425-g01-04.cs.illinois.edu 1234
E sp21-cs425-g01-05.cs.illinois.edu 1234
```

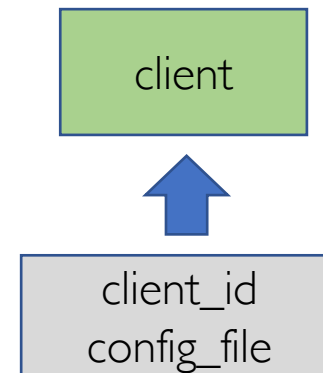
Use this information to establish communication across servers.

MP3: Distributed Transactions

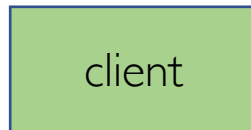


sample config_file

```
A sp21-cs425-g01-01.cs.illinois.edu 1234
B sp21-cs425-g01-02.cs.illinois.edu 1234
C sp21-cs425-g01-03.cs.illinois.edu 1234
D sp21-cs425-g01-04.cs.illinois.edu 1234
E sp21-cs425-g01-05.cs.illinois.edu 1234
```



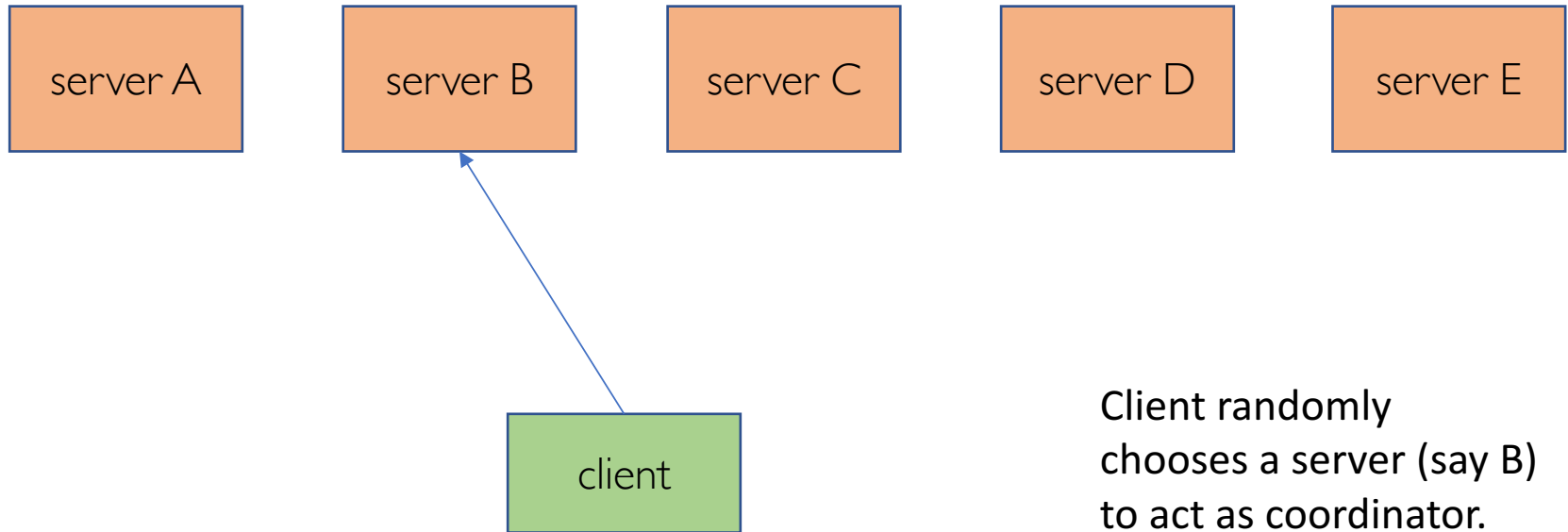
MP3: Distributed Transactions



Receives user input (command) from stdin.
Prints output of the command to stdout.

< BEGIN //start a new transaction

MP3: Distributed Transactions



Receives user input (command) from stdin.
Prints output of the command to stdout.

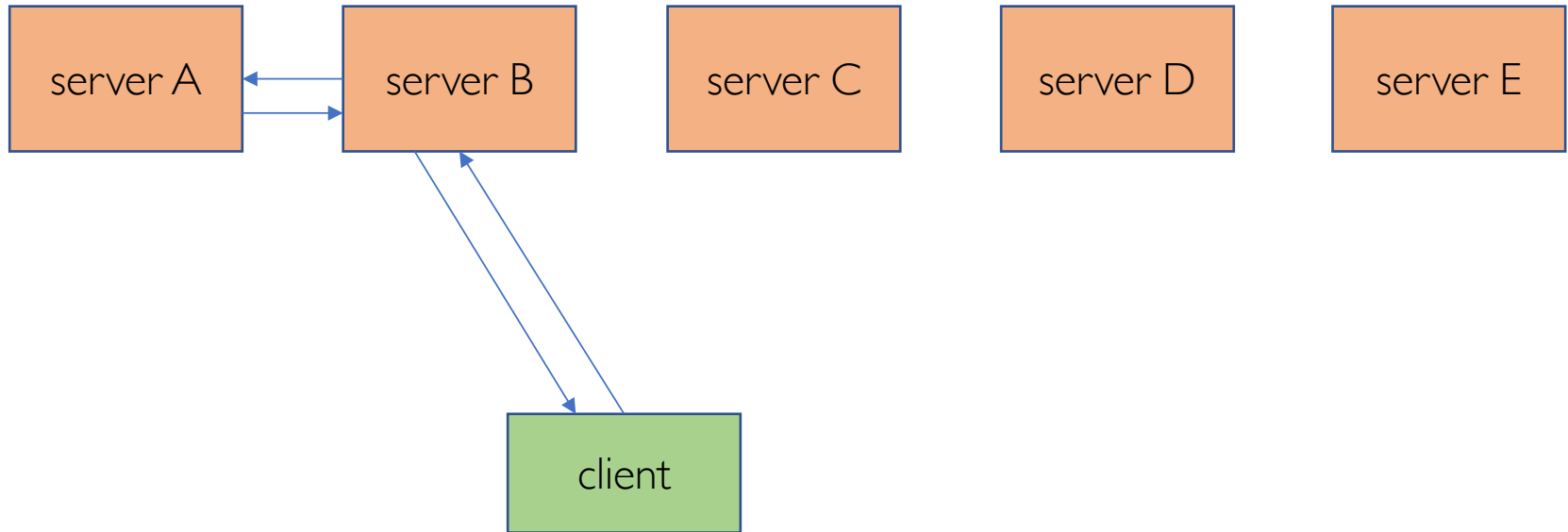
Client randomly
chooses a server (say B)
to act as coordinator.
**Only communicates
with the coordinator**

< **BEGIN** //start a new transaction

> **OK**

< **DEPOSIT A.foo 10** //deposit 10 units in account foo at branch A

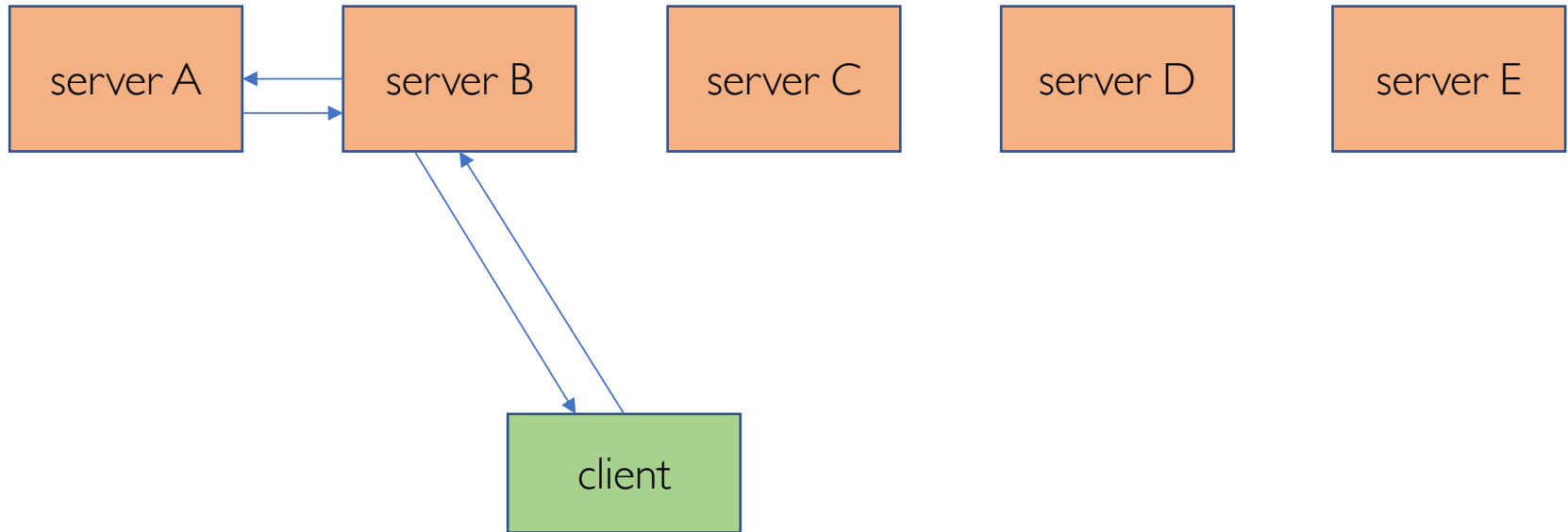
MP3: Distributed Transactions



Receives user input (command) from stdin.
Prints output of the command to stdout.

```
< BEGIN //start a new transaction  
> OK  
< DEPOSIT A.foo 10 //deposit 10 units in account foo at branch A  
> OK
```

MP3: Distributed Transactions

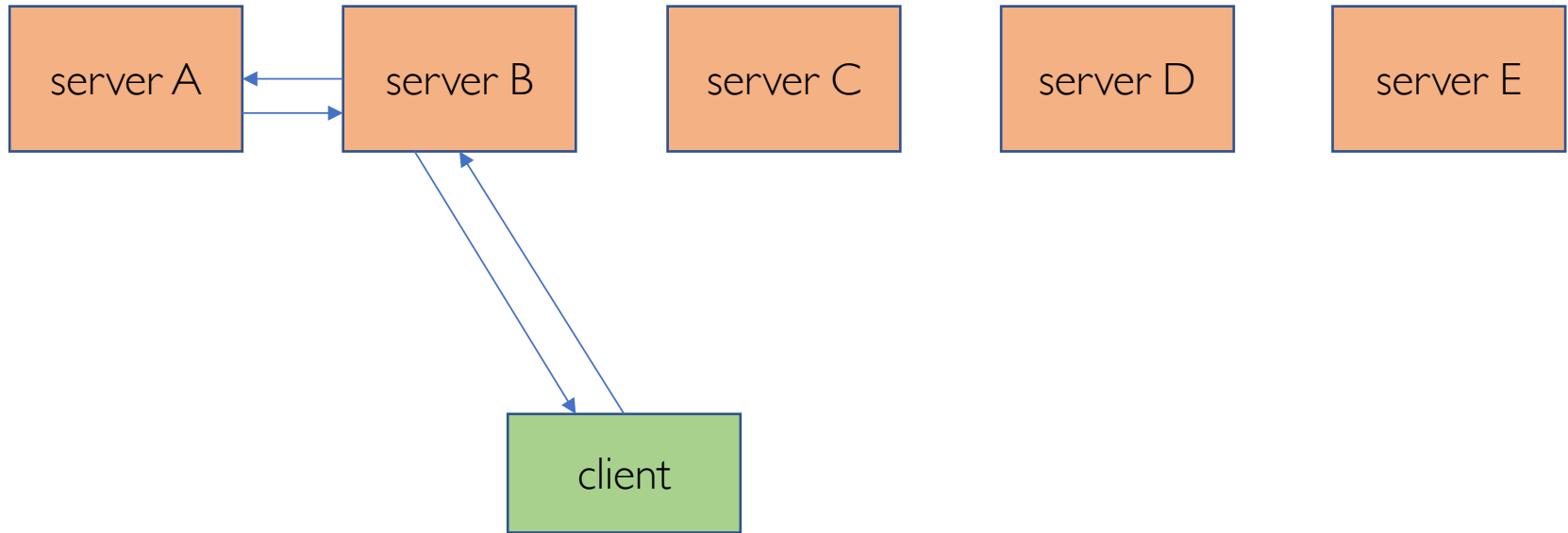


Receives user input (command) from stdin.
Prints output of the command to stdout.

```
< BEGIN //start a new transaction  
> OK  
< DEPOSIT A.foo 10 //deposit 10 units in account foo at branch A  
> OK
```

Other possible commands: WITHDRAW and BALANCE (only applicable if the account exists)

MP3: Distributed Transactions



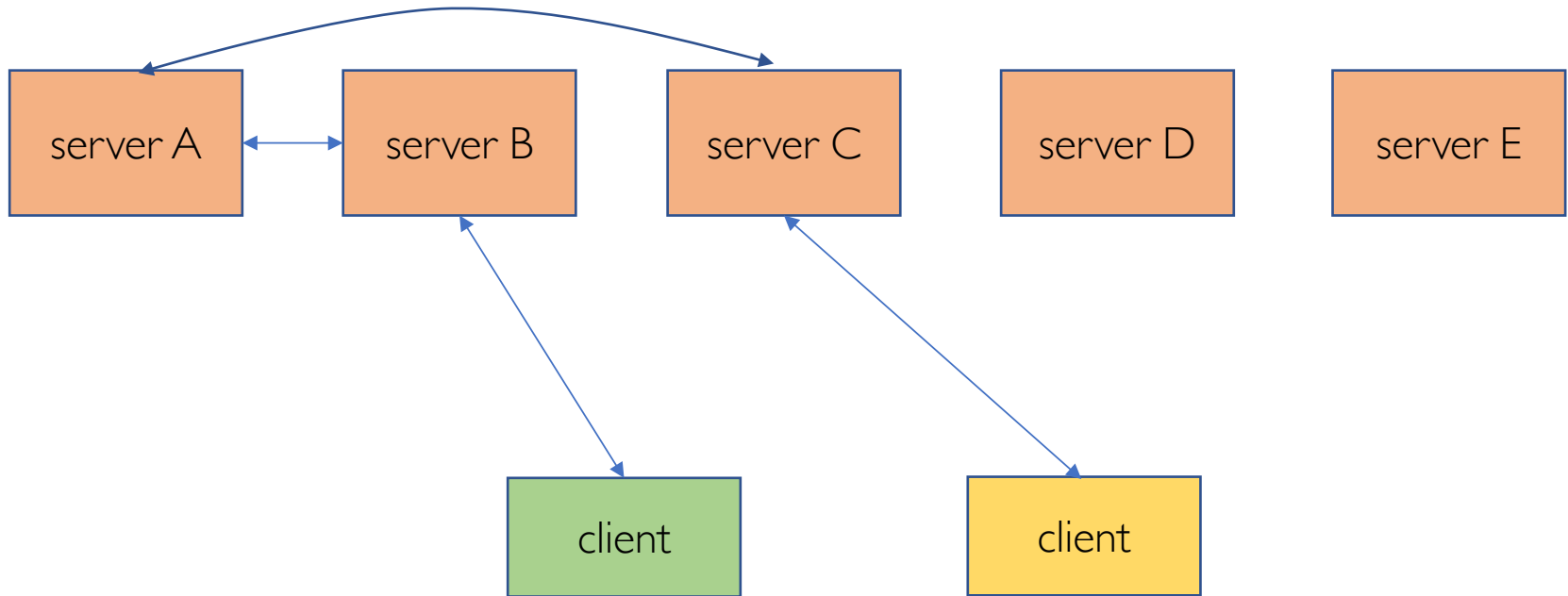
Receives user input (command) from stdin.
Prints output of the command to stdout.

User enters COMMIT or ABORT to end the transaction.

A server may also choose to ABORT a transaction (e.g. if consistency violated, or if needed for concurrency control).

Changes made by one transaction visible to others only after it successful commits.

MP3: Distributed Transactions

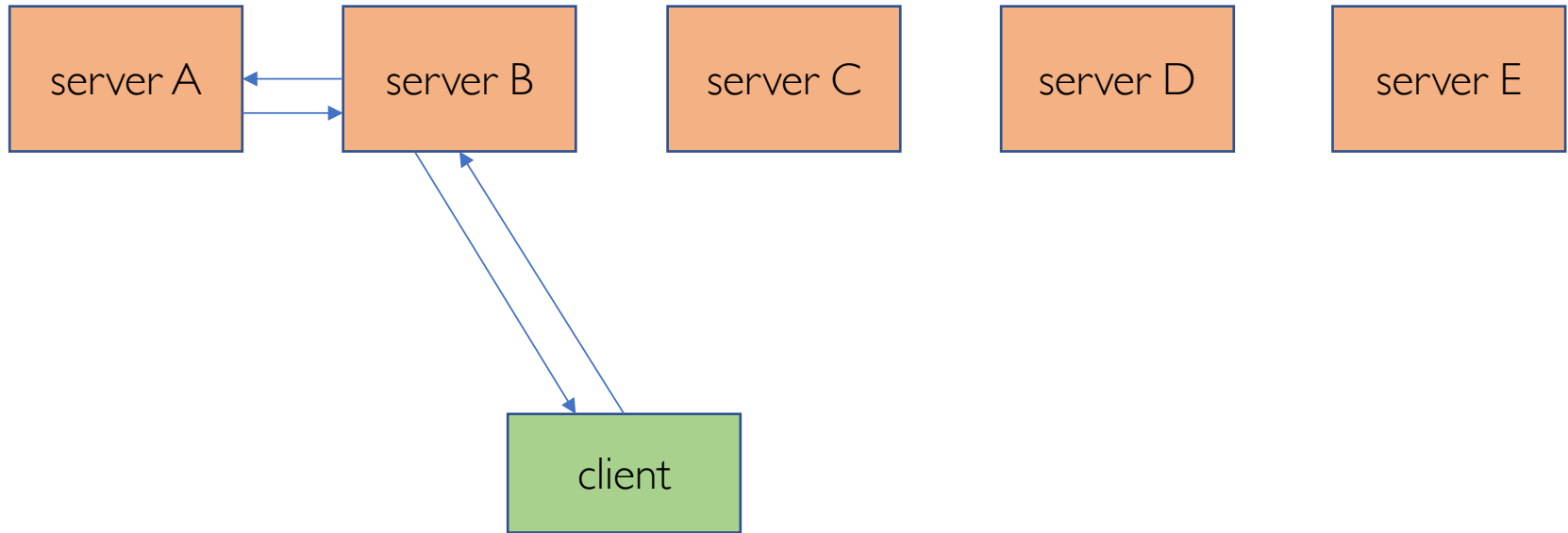


Receives user input (command) from stdin.
Prints output of the command to stdout.

Required properties:

- Isolation:
 - multiple clients may concurrently issue commands on the object.
 - Must provide serial equivalence.
- Deadlock avoidance.

MP3: Distributed Transactions



Receives user input (command) from stdin.
Prints output of the command to stdout.

Required properties:

- Atomicity:
 - all servers commit the entire transaction, or all rollback the entire transaction.
- Consistency:
 - cannot withdraw or read balance of a non-existent account.
 - a transaction cannot result in a negative account balance.

MP3: Distributed Transactions

- Due on Friday, May 5th.
 - Allowed to submit up to 50 hours late, but with 2% penalty for every late hour (rounded up).
- Read the specification fully and carefully.
 - Required semantics discussed more completely there.
- Start early!