Replicated State Machines

COS 316: Principles of Computer System Design
Lecture 13

Nicolaas Kaashoek

Motivation: Machines Fail

Machines have many sources of failure:

- Crashes
- Disk Failur
- Network is
- Fire/Natura

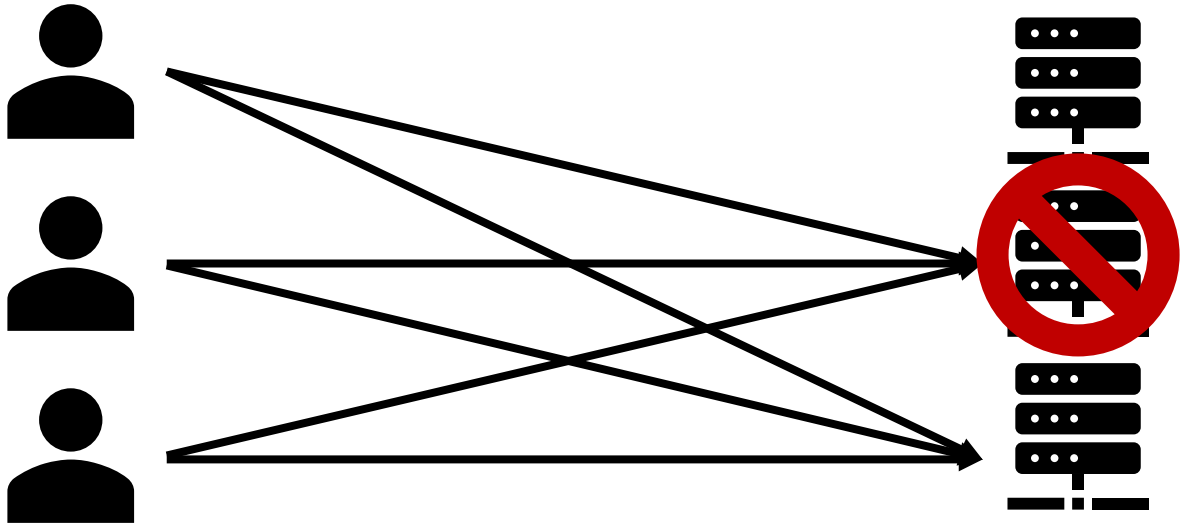
We need some kind of *fault tolerance*!

Individual Probability: Low (<)

Collective Probability? High!

- Google likely has 10^6 machines across many datacenters

Replication Provides Fault Tolerance



Machines may recover on own, may need manual assistance

How Fault Tolerant Should a System Be?

Goal: Minimize resource utilization

- Could use 100 machines…inefficient

Instead, figure out how many faults we can tolerate at once

- How? Measure. Time between failures, time to recover, etc
- Example: 2 machines, what happens when one fails?

Different protocols require different number of machines

- f+1 fault tolerance requires f+1 machines to handle f faults

The Replicated State Machine Model

The replicated state machine model is simple:

- All machines start from an initial state
- Then apply operations to move between states

Simple example: Counter

- Start state: 0
- Operations: Increment and Decrement

RSMs: Widely Used, Flexible Model

What is an operation?

- Updates to a file system
- Updates to a bank account
- Machine instructions

Ability to support *any* system that can be represented as start state and operations is very powerful!

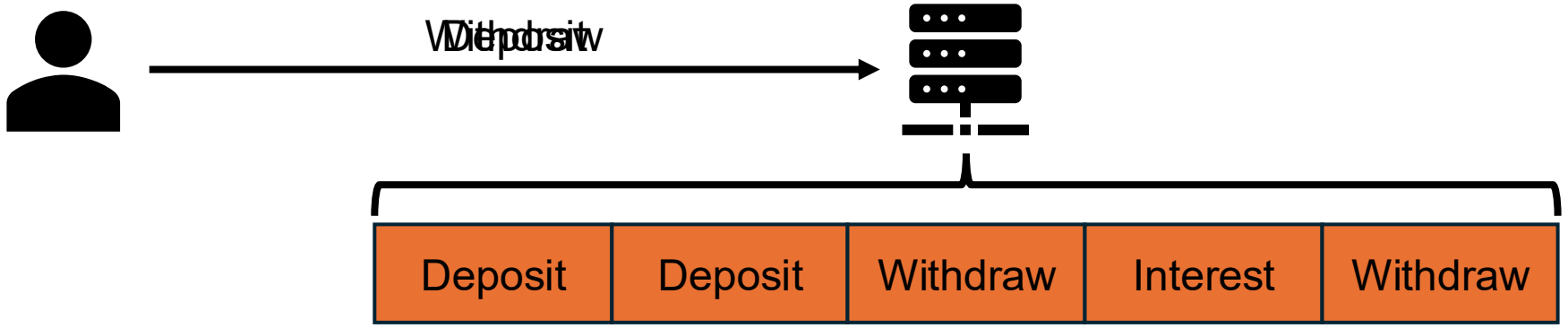
RSMs are widely used across distributed systems/databases

RSMs: Operation Log

Goal: Ensure that replicas get same end state

How? Enforce ordering of operations

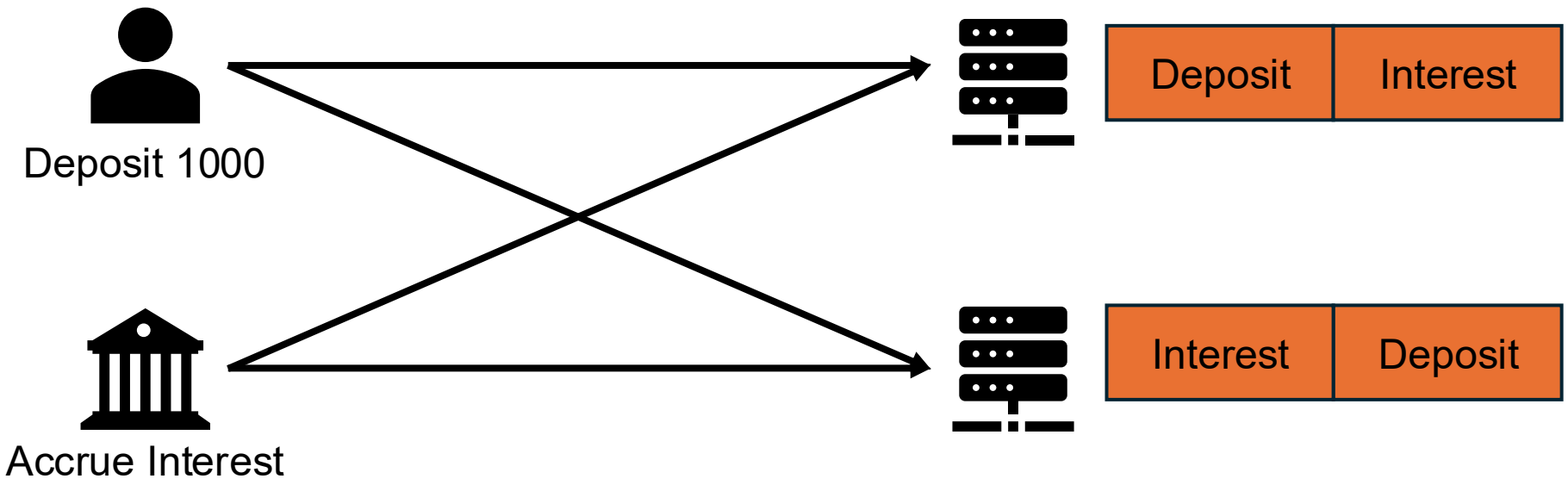
How? Use a log



Logs also allows for replay after failure

Challenge: Keeping Replicas in Sync

Problem: How do we keep the replicas *identical?*



Solution: Consistency! Remember the last few lectures.

How do we get consistency between many machines?

Primary-Backup Replication

Problem: How can we get the same log on all machines?

Simplest answer:

- 1 machine is the primary
- All other machines are backups
- Primary determines order of operations

Primary-Backup: Determinism

Potential problem: what if operations aren't deterministic?

`random.` **`seed`**(*a=None*)

Initialize internal state of the random number generator.

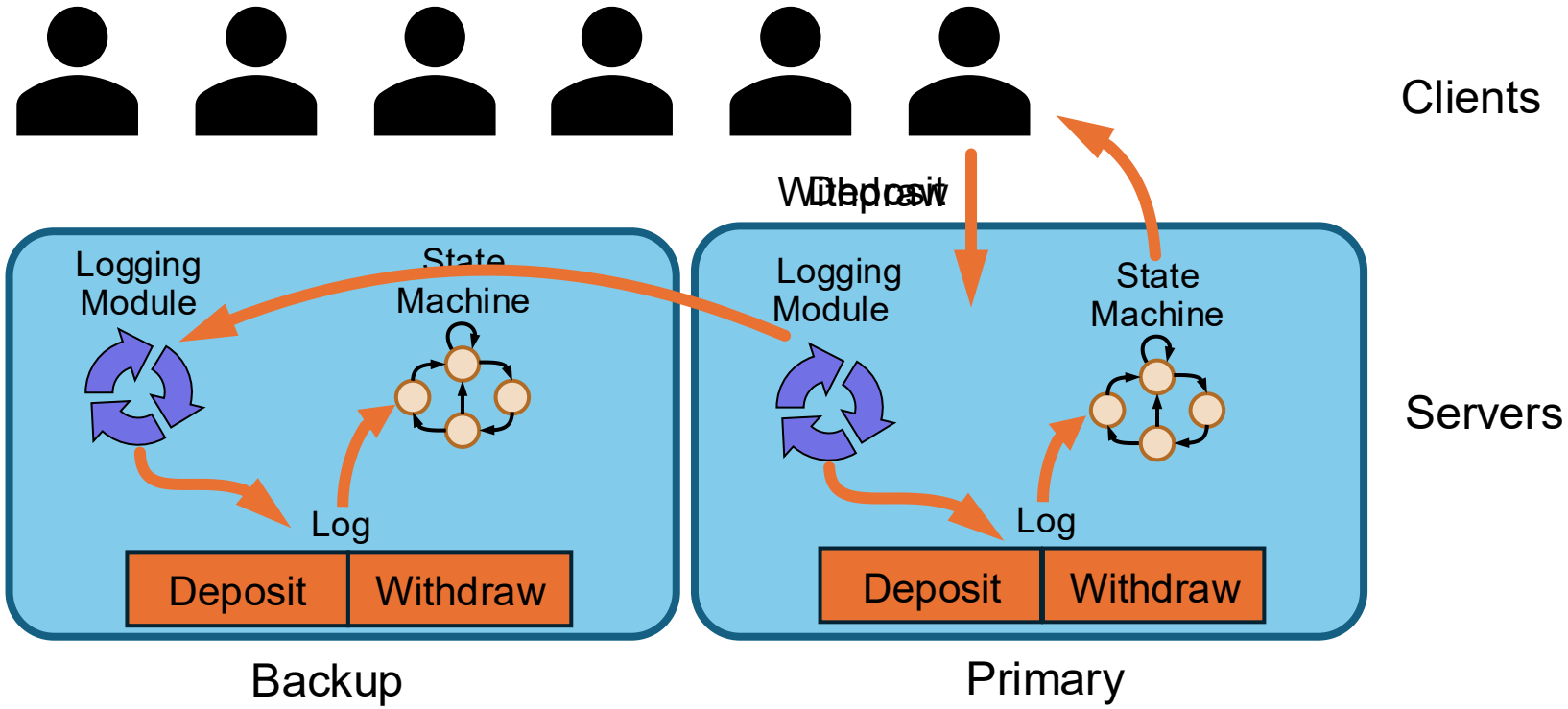
`None` or no argument seeds from current time or from an operating system specific randomness source if available (see the `os.urandom()` function for details on availability).

`random.` **`getstate`**()

Return an object capturing the current internal state of the generator. This object can be passed to `setstate()` to restore the state.

We can leverage the primary to provide ordering and make operations deterministic

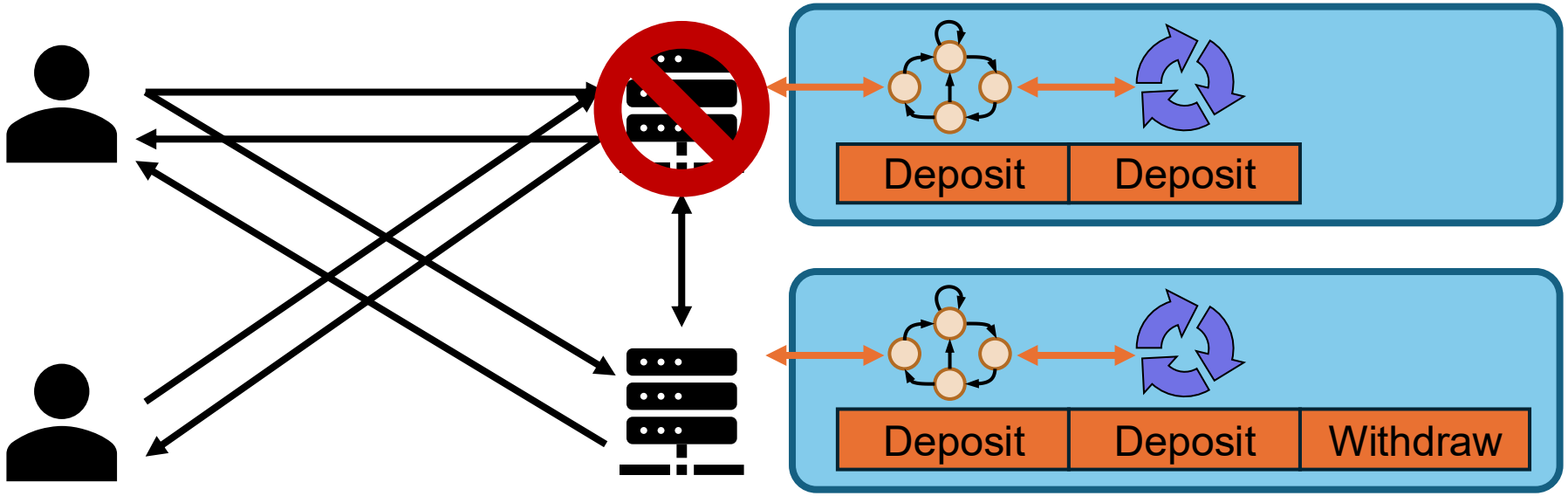
Primary-Backup Synch Replication



Synchronous Replication: Primary doesn't respond until all replication done

Reads can be handled by just the primary (ROWA)

Primary-Backup: Fault Tolerance



No matter which machine fails, we can easily recover!

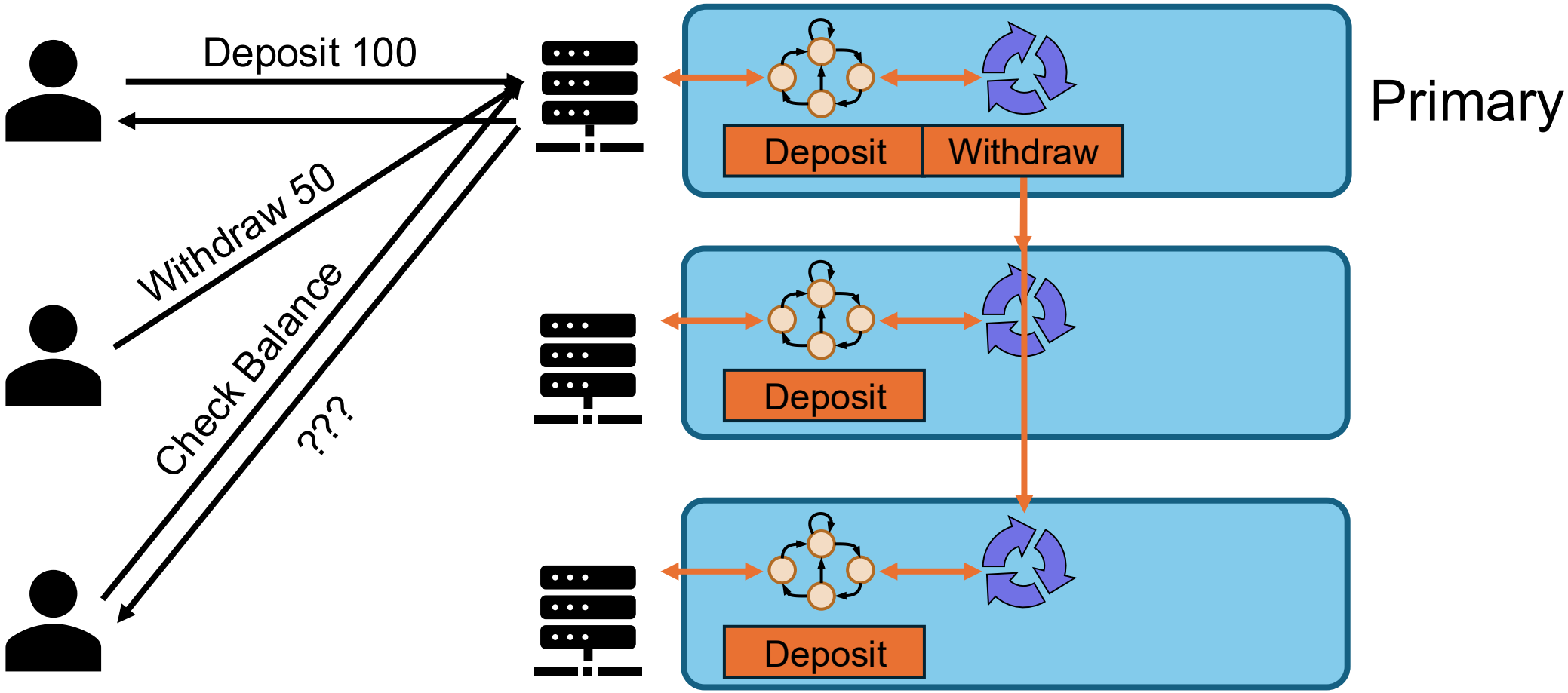
- Primary failure: Pick new primary
- Backup failure: Can just continue as normal

Primary-Backup: Consistency + FT

What level of consistency does primary-backup with synchronous replication give us?

What level of fault tolerance does primary-backup give us? How many machines to tolerate f failures?

Primary-Backup: Concurrent Ops



Primary-Backup: Log Entry States

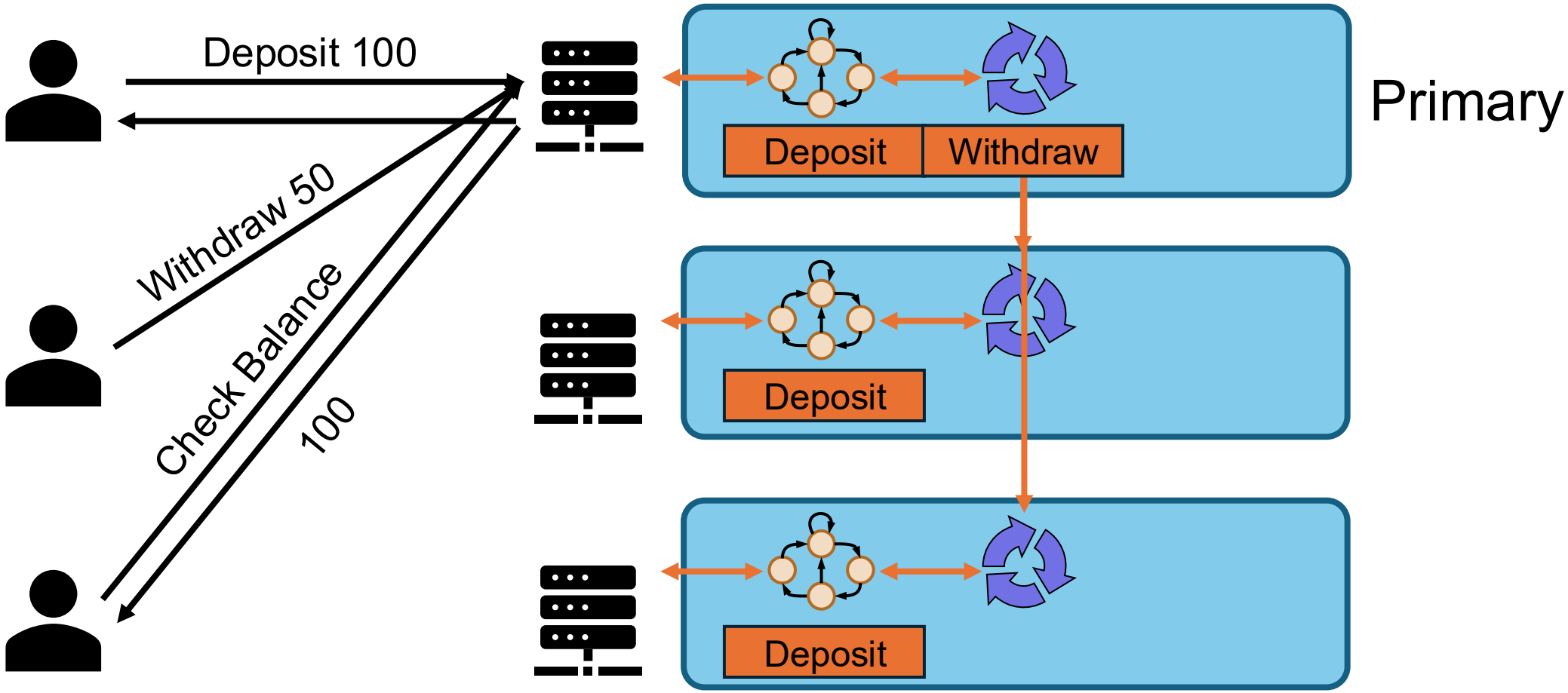
We can’t just blindly respond with what’s in the log

Instead, log entries have multiple states:

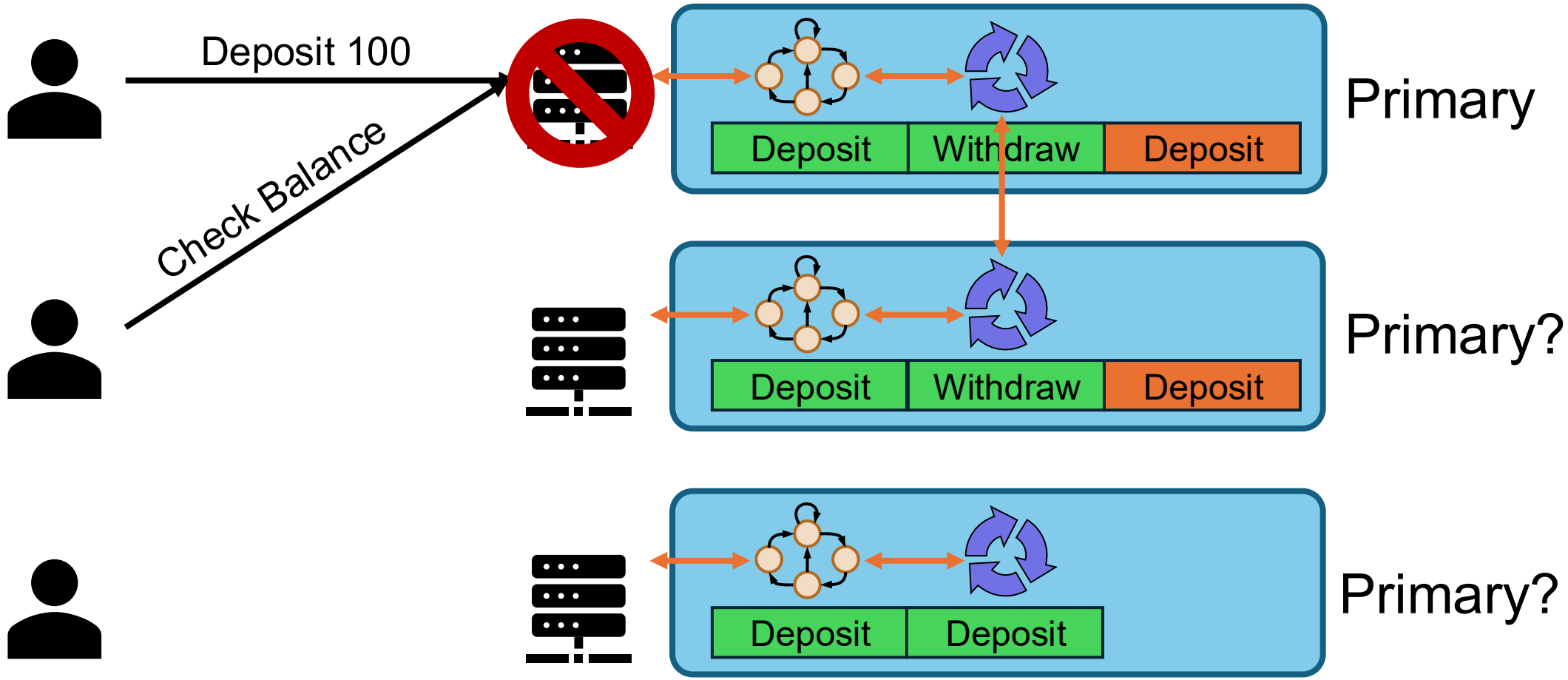
- Committed: All backups have confirmed they have the entry
- Uncommitted: Not all backups have confirmed the entry

When serving a read, only respond with committed data

Primary-Backup with Commits



Primary-Backup: Which Backup?



Primary piggy-backs commit status in future messages

Primary-Backup: Heartbeats

Need mechanism to know if replica is alive

Heartbeats: Regular check-ins to monitor service status

- Not 100% reliable (see in a few slides)
- Good mechanism for detecting and initiating failover

Primary-Backup: Failover

Problem: Backups have different logs. Which to pick?

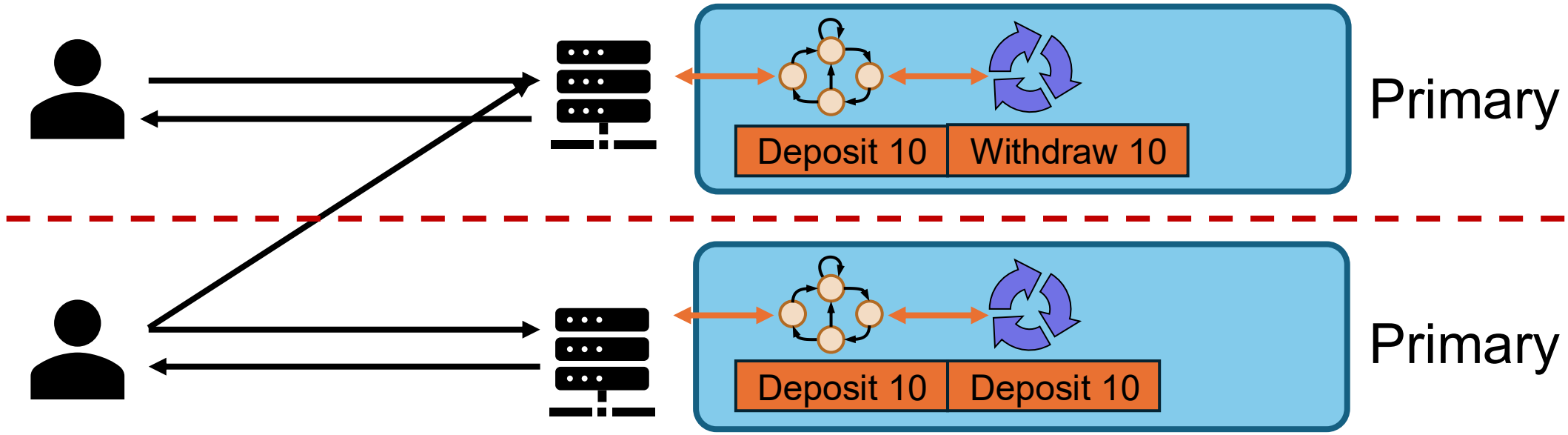
- Highest known operation number ?
- Lowest known operation number ?
- Select randomly ?

Once selected, new primary must enforce order on backups

- Failover Procedure

We'll come back to this in a few slides

Primary-Backup: Network Partitions



Network Partition: No traffic can cross, traffic flows both sides

- Occur in real life!

Danger: Having backups decide they should be new primary

Primary-Backup: Coordinator

Need a third party in the system that manages the configuration

- Coordinator/Configuration manager/View manager

Job:

- Determine which machine is the primary
- Tell the other replicas which machine is the primary
- Track health of the system and respond to failures

Primary-Backup: Coordinator 2

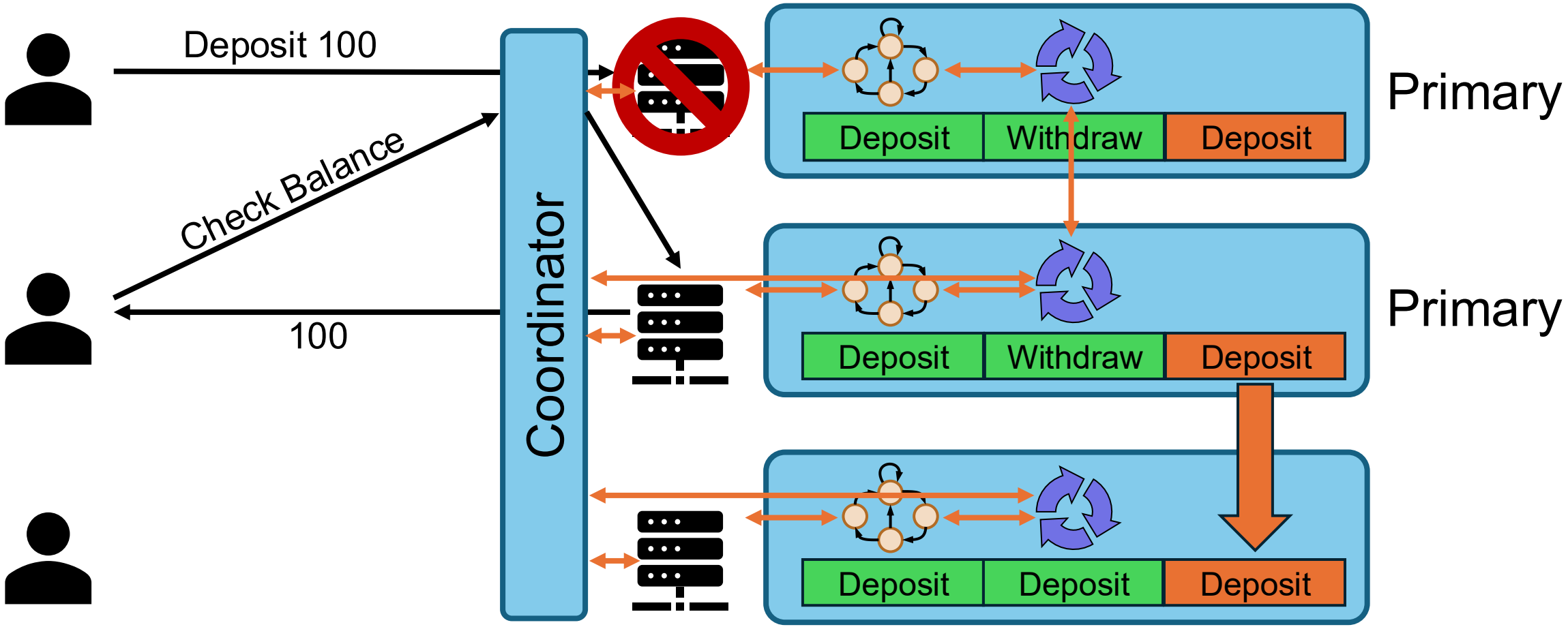
Split brain: Coordinator can’t talk to current primary

Solution:

- Primary doesn’t get to decide who it needs to hear back from
- Coordinator tells replicas who is current primary; don’t respond to old ones

What about reads?

Primary-Backup: With Coordinator



Coordinator Failures

This course: coordinator cannot fail, single entity

In practice: coordinator is replicated with complex protocol

- Paxos + its many derivatives
- Raft

Take 418!

Primary-Backup: Sharding for Scale

All requests go to single machine (primary)

Real systems: millions of users!

Solution: Sharding

- Split the data into pieces, allocate replica groups for each shard
- How to split? Depends on the data

Conclusion

Replicated state machines are a common model in distributed systems, and are used everywhere

Replication solves the problem of fault tolerance

Consistency solves the problems that replication creates

Even simple RSM protocols have lots of hidden complexity! Distributed systems are tricky