Fault Tolerance

SWE 622, Spring 2017
Distributed Software Engineering

GEORGE MASON UNIVERSITY

Review

- General agreement problems
- Fault tolerance limitations of 2PC
- 3PC
- Paxos + ZooKeeper

Properties of Agreement

- Safety (correctness)
  - All nodes agree on the same value (which was proposed by some node)
- Liveness (fault tolerance, availability)
  - If less than N nodes crash, the rest should still be OK

2PC Timeouts

- We can solve a lot (but not all of the cases) by having the participants talk to each other
- But, if coordinator fails, there are cases where everyone stalls until it recovers
- Can the coordinator fail?… yes
- Hence, 2PC does not guarantee **liveness**: a single node failing can cause the entire set to fail

3 Phase Commit

- Goal: Eliminate this specific failure from blocking liveness

Coordinator

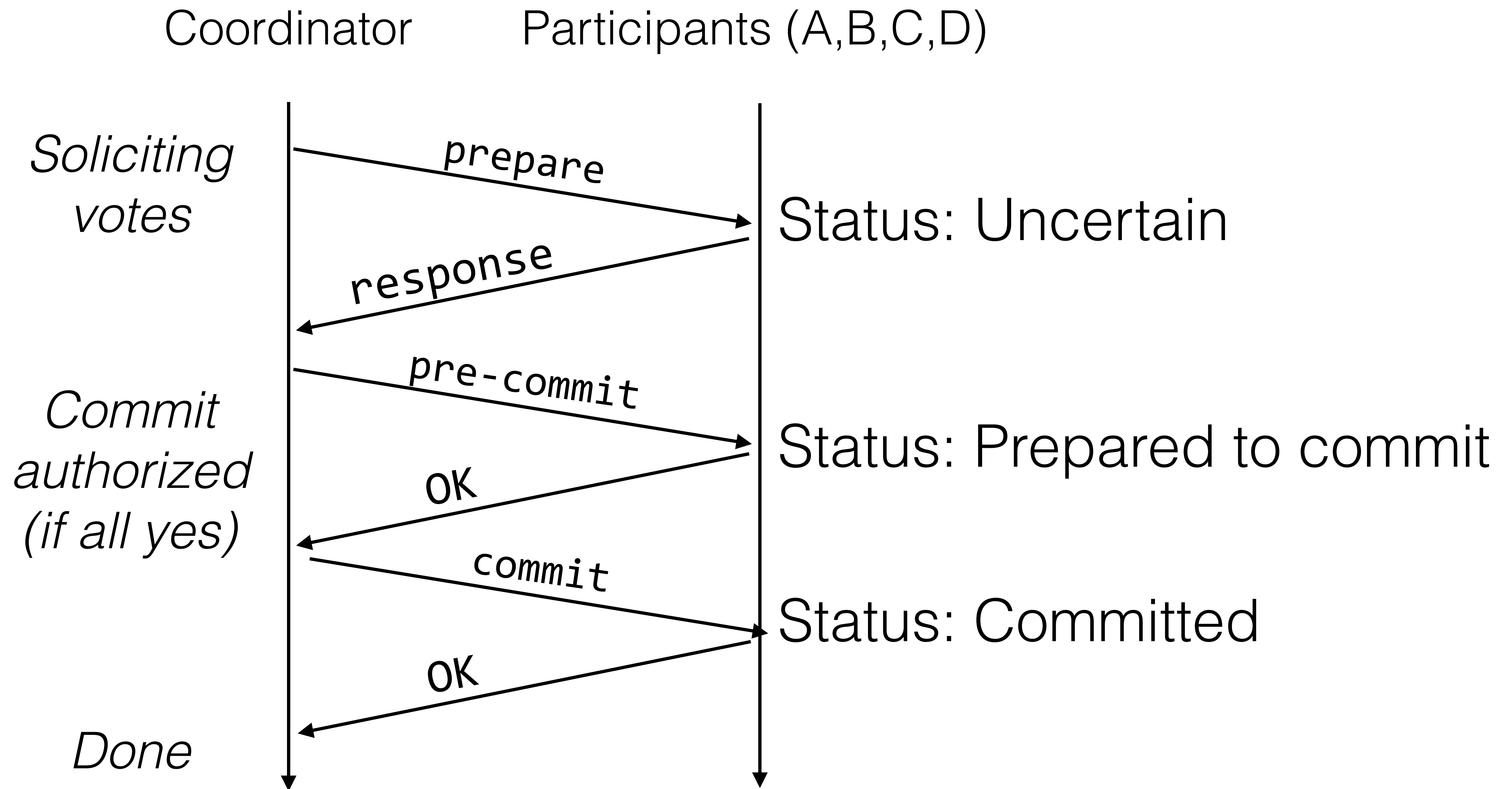
Participant A — Voted yes, Heard back "commit"

Participant B — Voted yes, **Did not hear result**

Participant C — Voted yes, **Did not hear result**

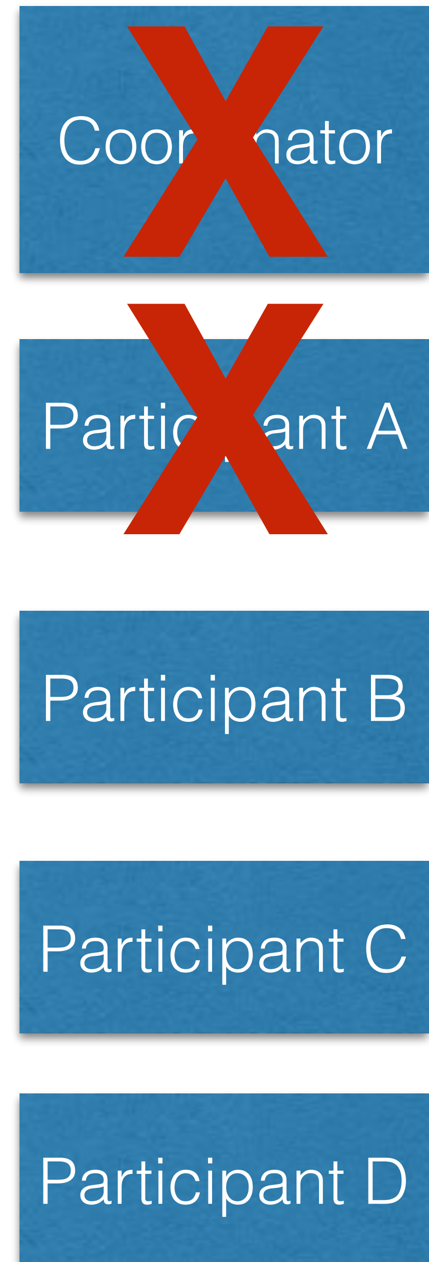
Participant D — Voted yes, **Did not hear result**

3PC Example



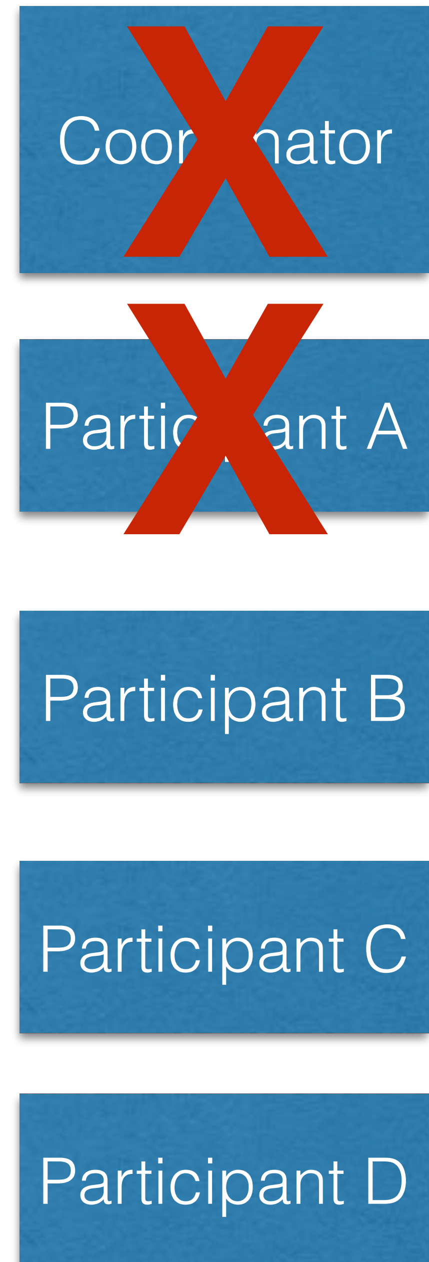
3PC Crash Handling

- Can B/C/D reach a safe decision…
  - If any one of them has received preCommit?
    - YES! Assume A is dead. When A comes back online, it will recover, and talk to B/C/D to catch up.
    - Consider equivalent to in 2PC where B/C/D received the "commit" message and all voted yes

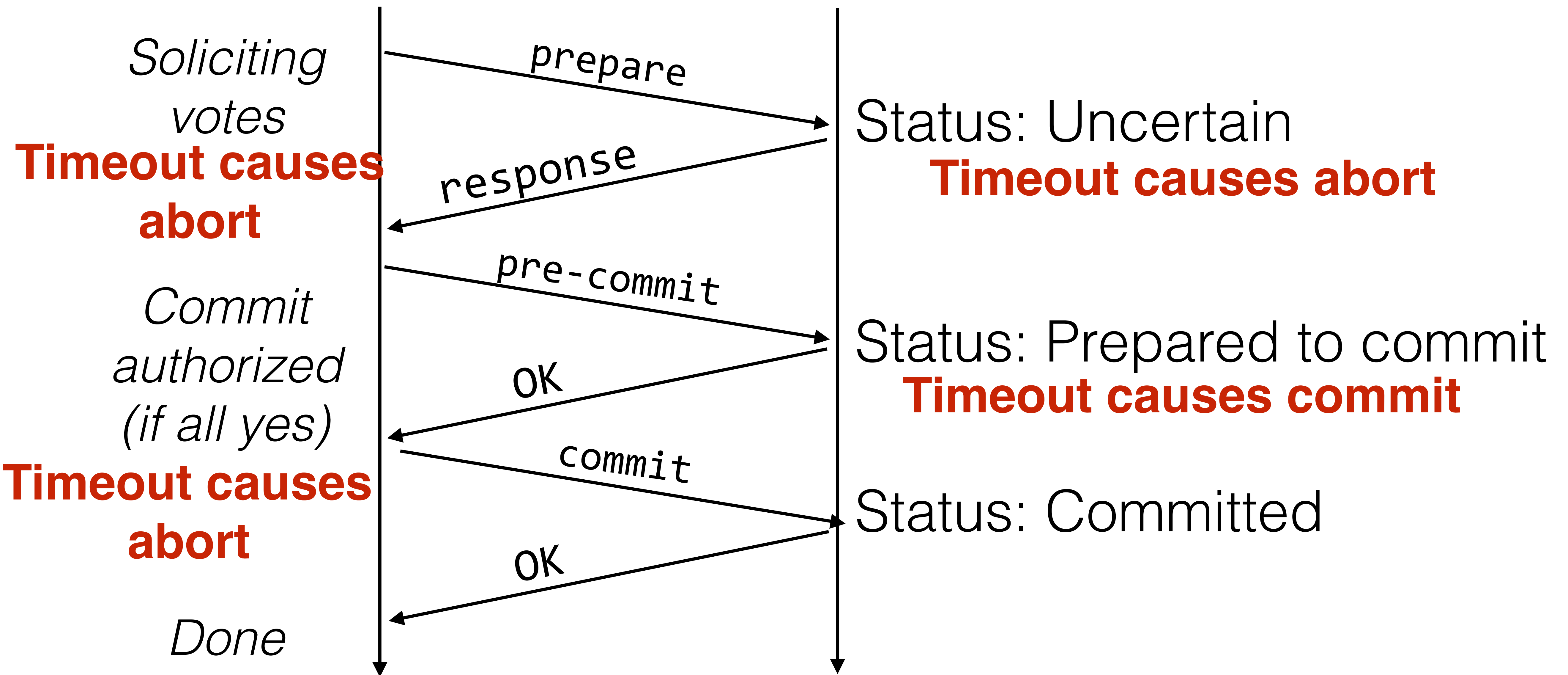


3PC Crash Handling

- Can B/C/D reach a safe decision…
  - If NONE of them has received preCommit?
    - YES! It is safe to abort, because A can not have committed (because it couldn’t commit until B/C/D receive and acknowledge the pre-commit)
    - This is the big strength of the extra phase over 2PC
- Summary: Any node can crash at any time, and we can always safely abort or commit.



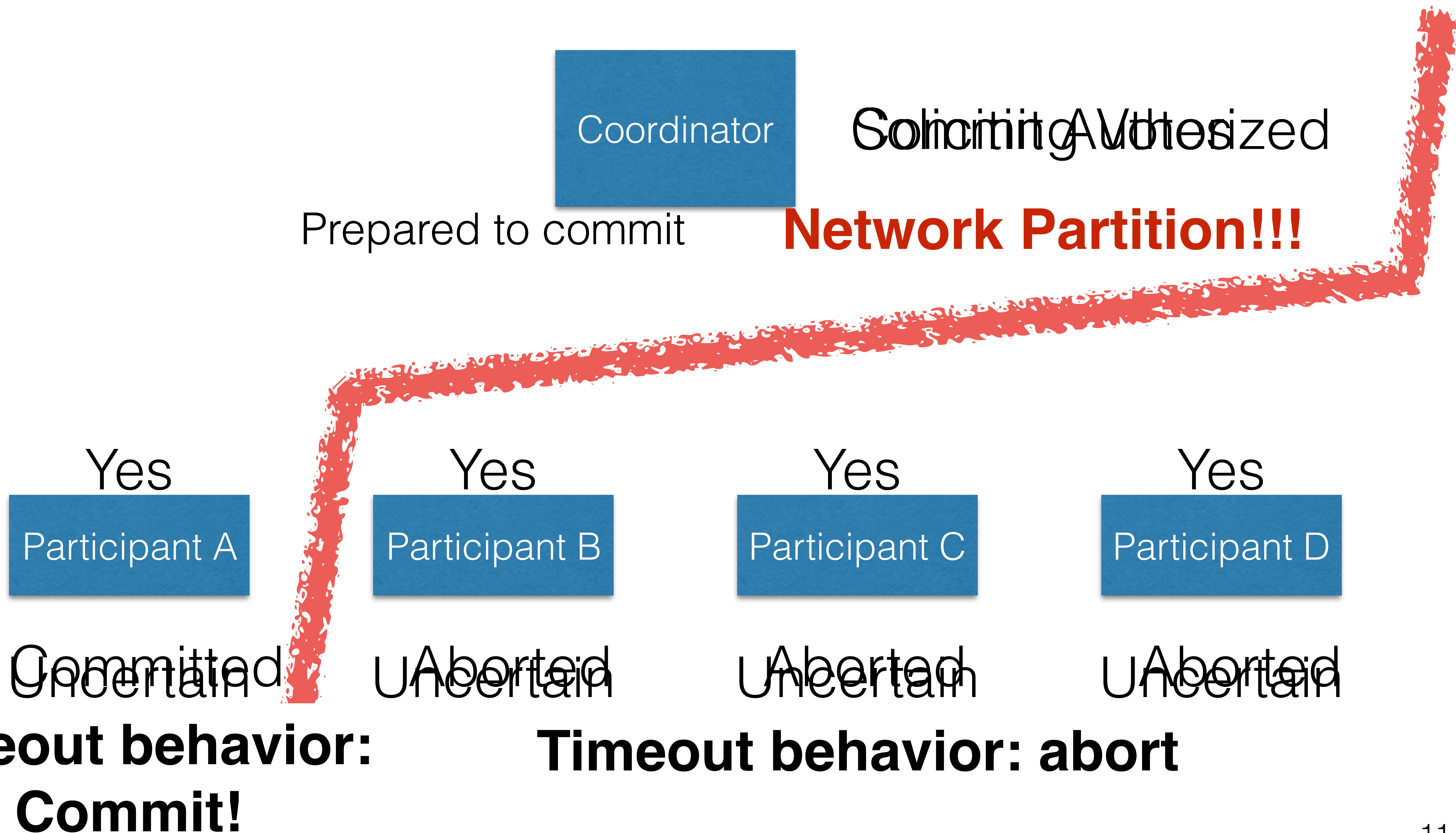
3PC Timeout Handling



Does 3PC guarantee consensus?

- Reminder, that means:
  - Liveness (availability)
    - **Yes!** Always terminates based on timeouts
  - Safety (correctness)
    - Hmm…

Partitions



Can we fix it?

- Short answer: No.
- Fischer, Lynch & Paterson (FLP) Impossibility Result:
  - Assume that nodes can only fail by crashing, network is reliable but can be delayed arbitrarily
  - Then, there can not be a deterministic algorithm for the consensus problem subject to these failures

Partition Tolerant Consensus Algorithms

- Decisions made by **majority**
- Typically a fixed coordinator (**leader**) during a time period (**epoch**)
- How does the leader change?
  - Assume it starts out as an arbitrary node
  - The leader sends a heartbeat
  - If you haven't heard from the leader, then you **challenge** it by advancing to the next epoch and try to elect a new one
  - If you don't get a **majority** of votes, you don't get to be leader
  - …hence no leader in a minority partition

ZooKeeper - Guarantees

- **Liveness guarantees**: if a majority of ZooKeeper servers are active and communicating the service will be available
- **Durability guarantees**: if the ZooKeeper service responds successfully to a change request, that change persists across any number of failures as long as a quorum of servers is eventually able to recover

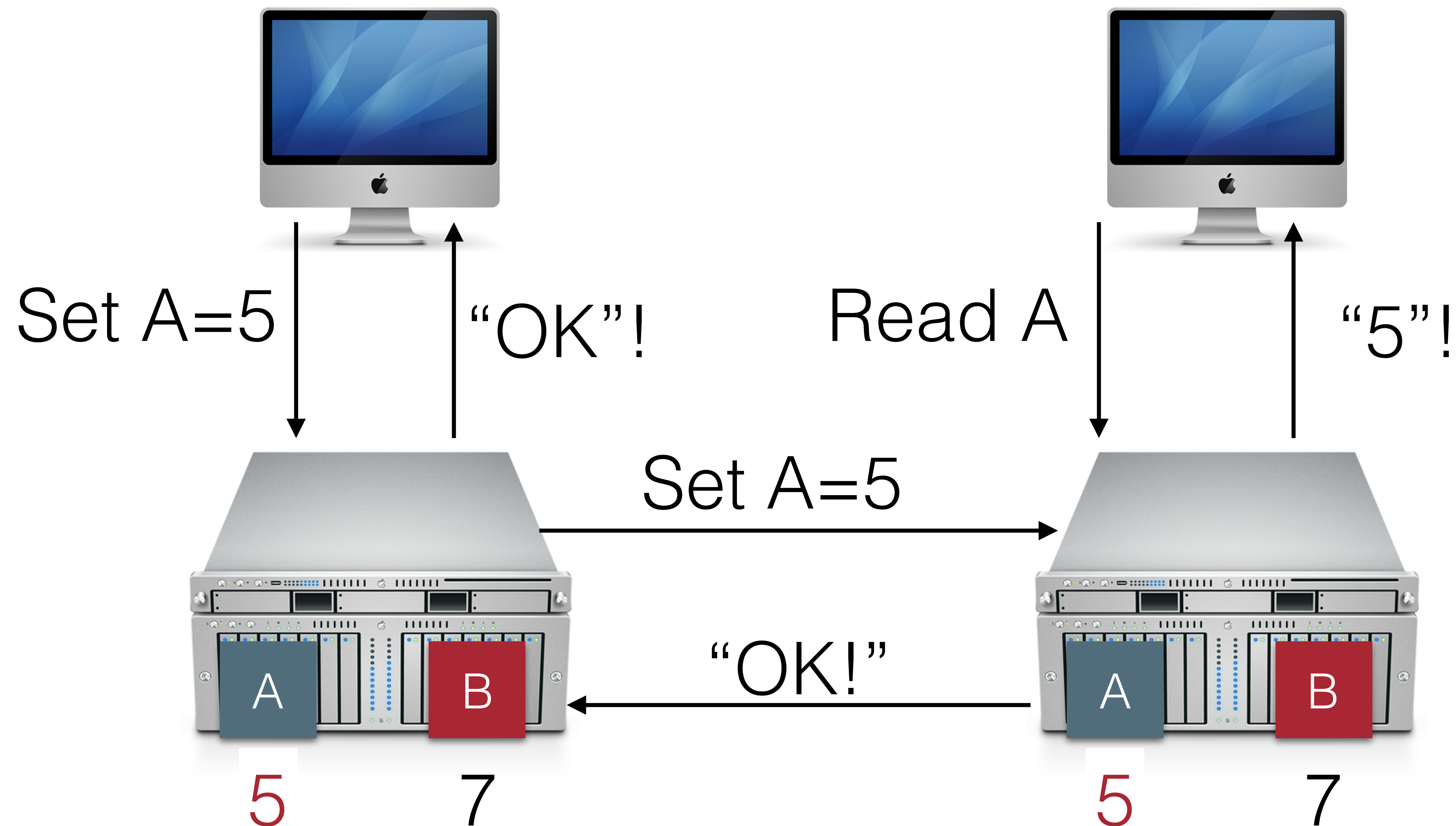
Today

- Replicated State Machines
- Fault Tolerance in ZooKeeper
- Case studies: Hypervisor, GFS, Spanner, MagicPocket

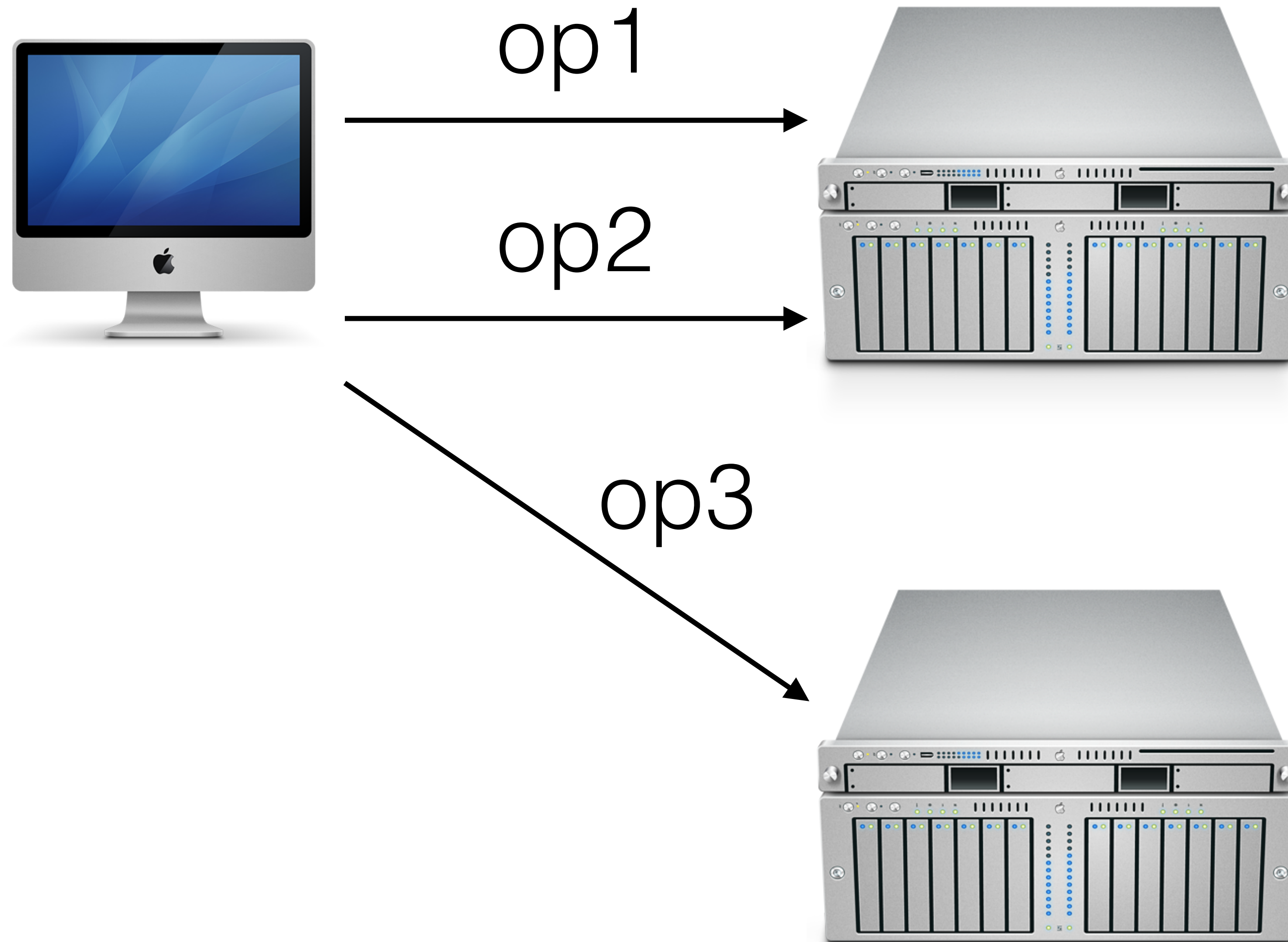
Fault Tolerance So Far

- Two goals:
  - Recoverability
    - Given a failure, can we detect it and eventually recover (maybe reboot or buy new hardware)
  - Availability
    - Given a failure, can we keep working
- So far, focused on recoverability (e.g. 2 phase commit), started to talk about availability (paxos)

Replication



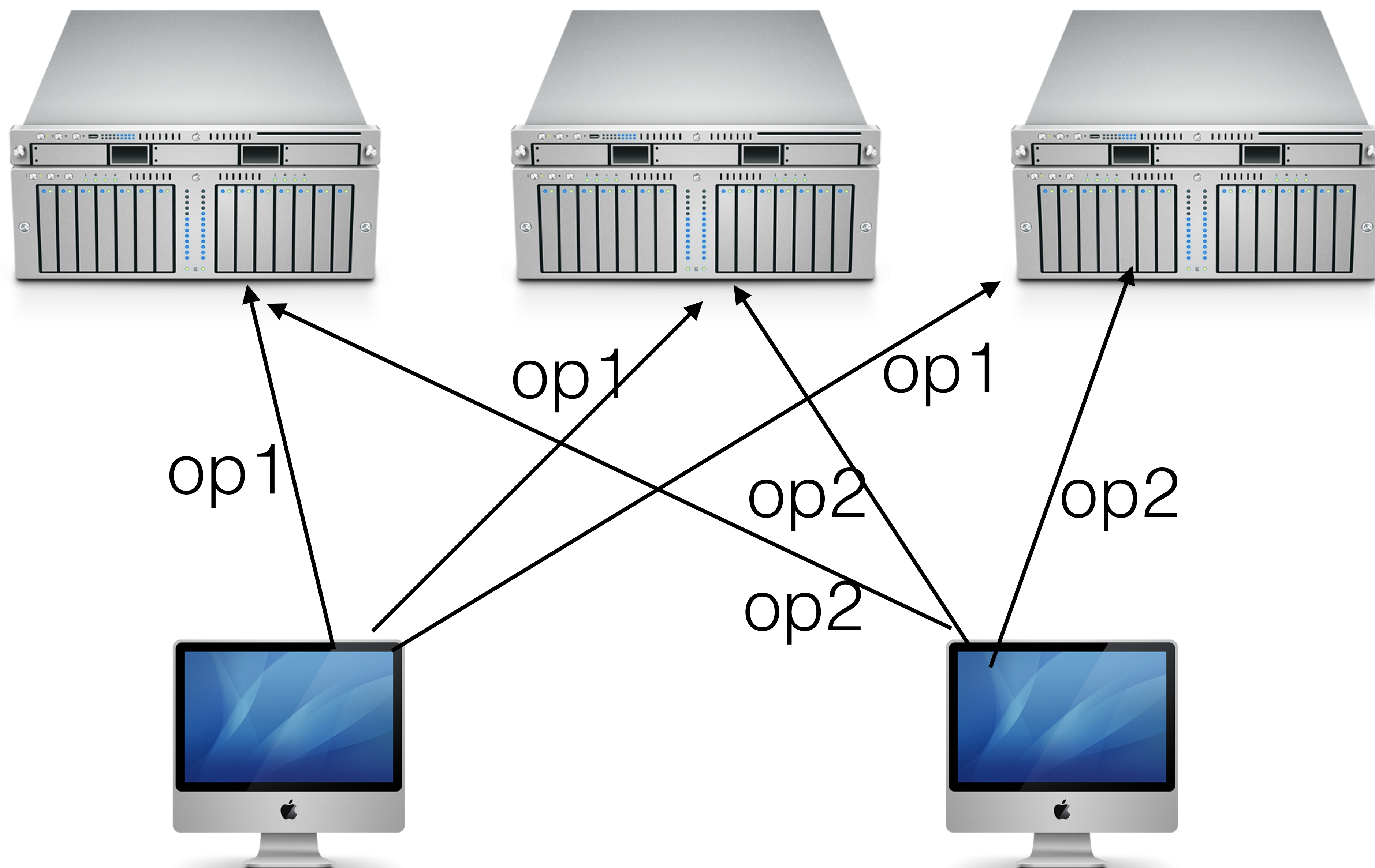
Replicating more than data



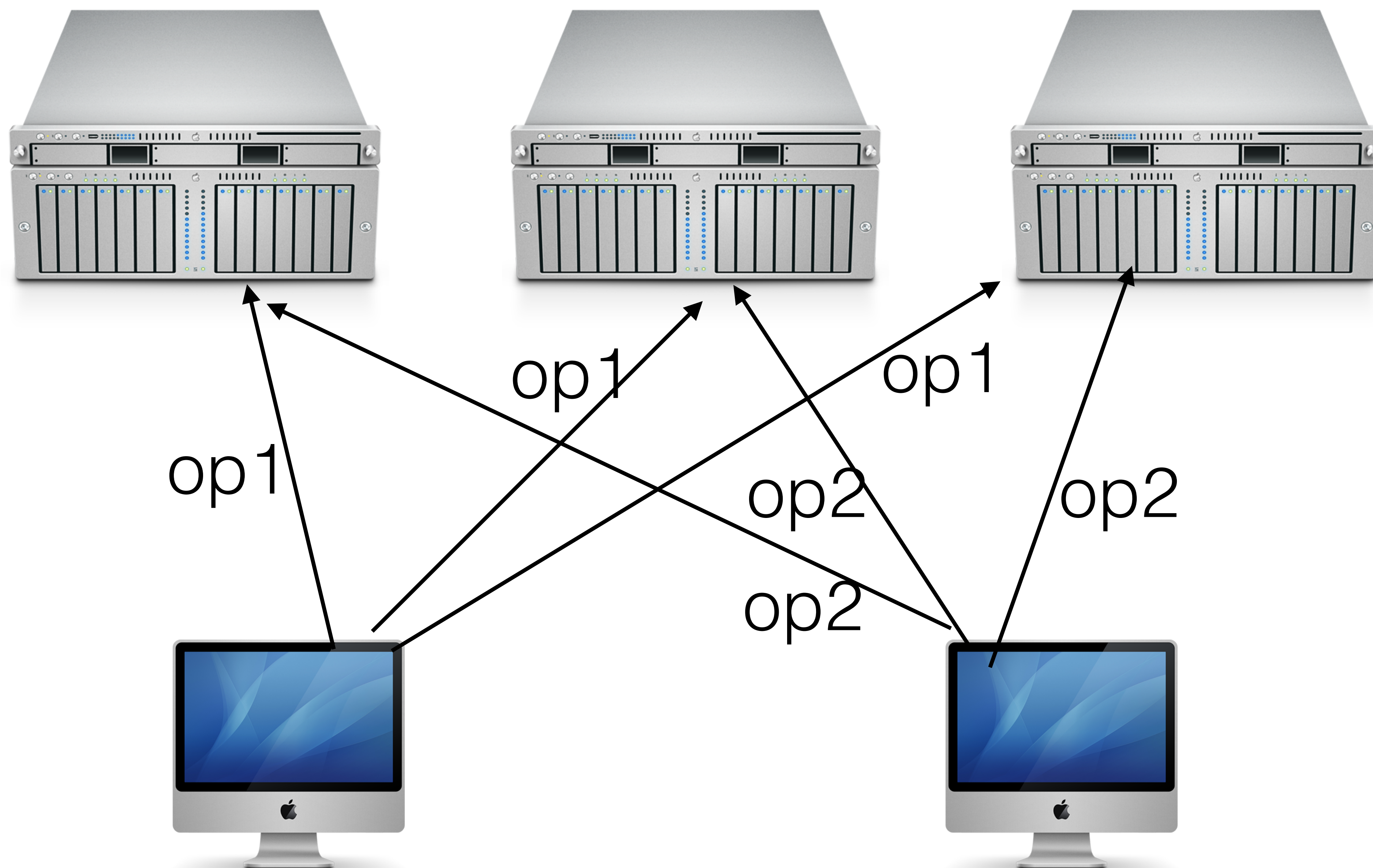
RSM: Replicated State Machines

- General replication method for protocols:
  - All replicas start in the same initial state
  - Every replica applies operations in the same order
  - All operations must be deterministic (same result on different replicas)
- All replicas end up in the same state

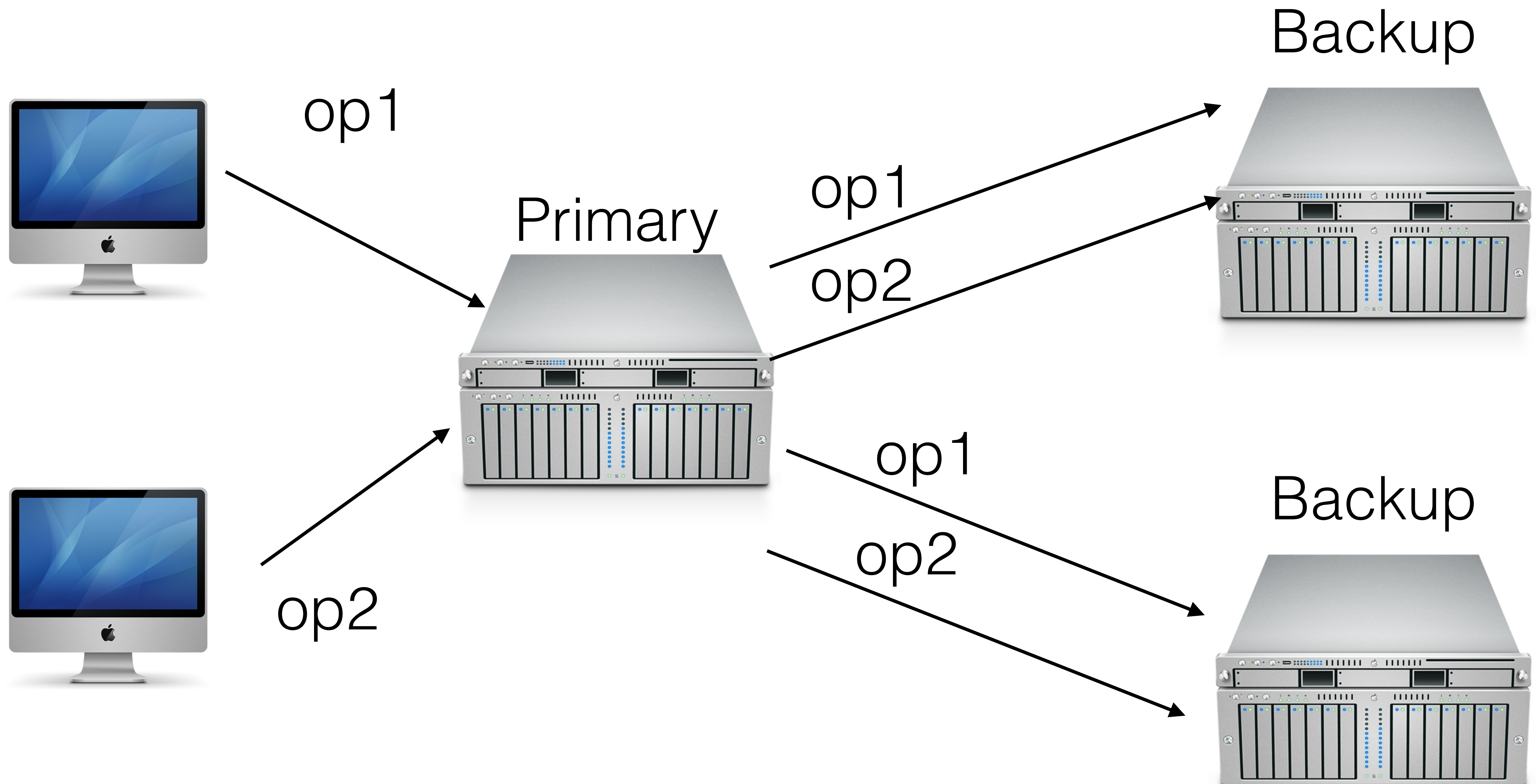
Strawman RSM implementation



Strawman RSM implementation

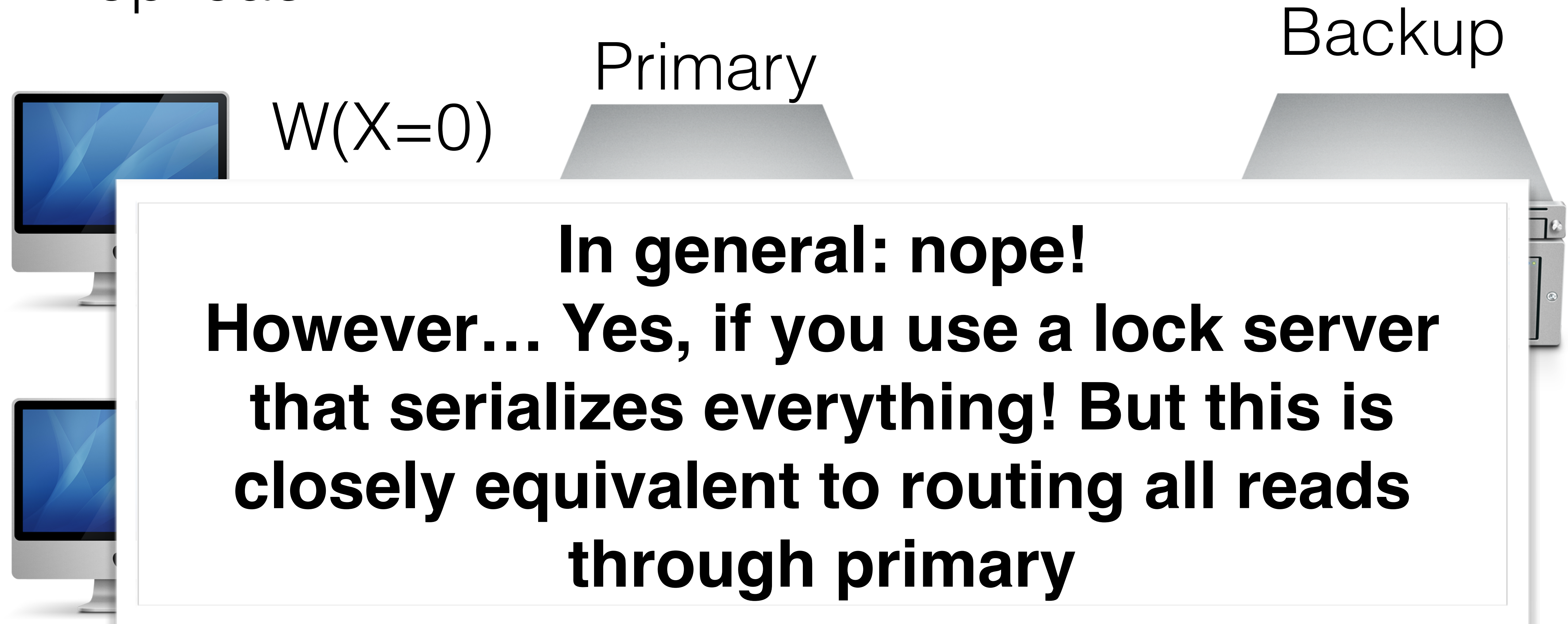


Primary/backup RSM



RSM: Read-Only Operations

- Consider Redis setup from homework. Obviously we don't need to broadcast reads to replicas if served by primary. Can reads be served by replicas?



In general: nope!
However… Yes, if you use a lock server that serializes everything! But this is closely equivalent to routing all reads through primary

Failure Handling

- If primary fails, one backup becomes new primary
- Challenges:
  - How to detect primary failed?
    - **Paxos/ZK: backup challenges the primary**
  - How to ensure only one primary?
    - **Paxos/ZK: voting**
  - How to preserve sequential consistency during transition between primaries?
    - **Paxos/ZK: voting**

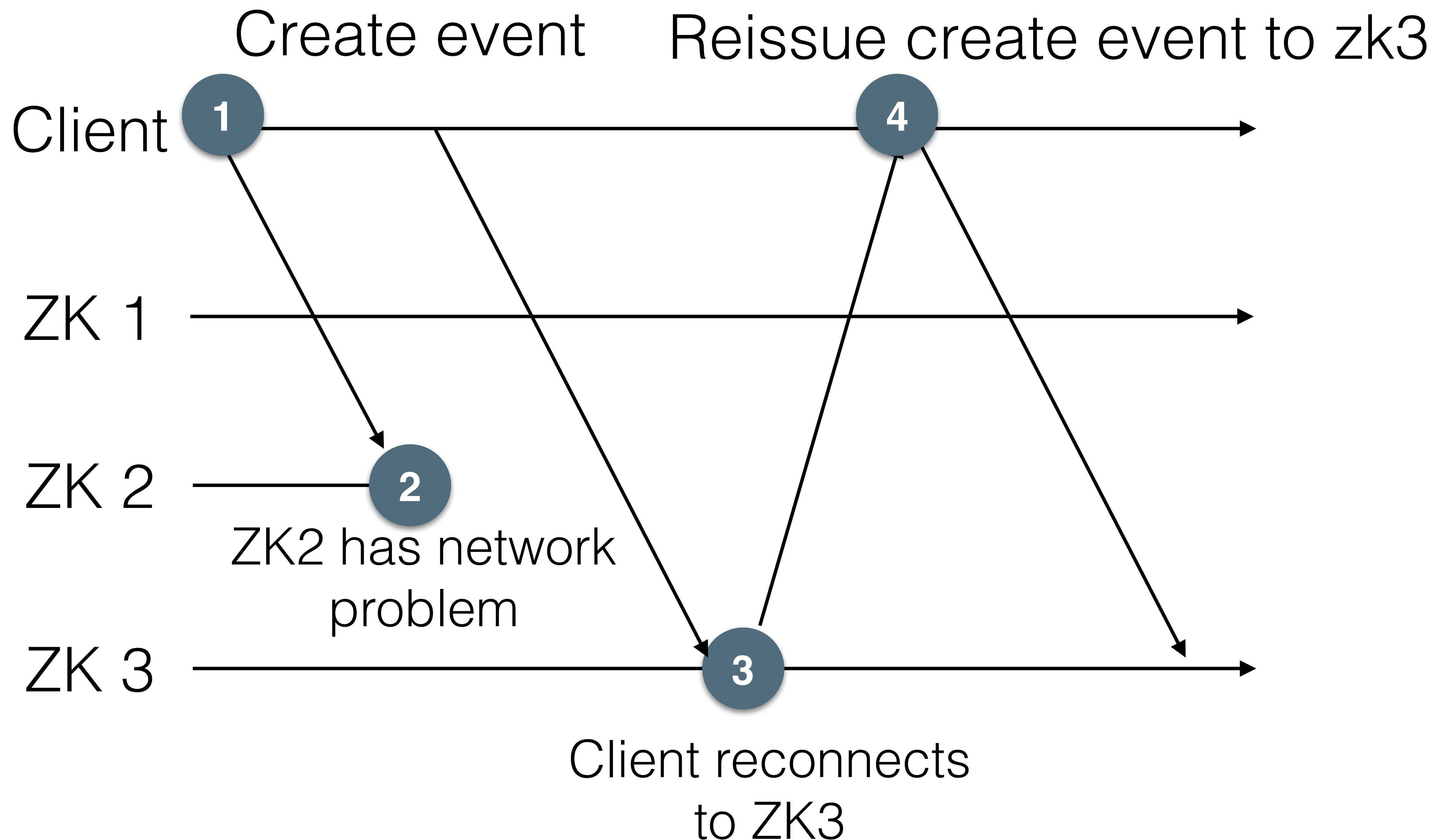
Paxos for RSM

- Track *group membership* (e.g. who is participating in the protocol). Any replica can join the group.
- Track *leadership* using an *election* - nodes vote on who is leader
- Track protocol state using voting

Failure Handling in ZK

- Just using ZooKeeper does not solve failures
- Apps using ZooKeeper need to be aware of the potential failures that can occur, and act appropriately
- ZK client will guarantee consistency **if it is connected to the server cluster**

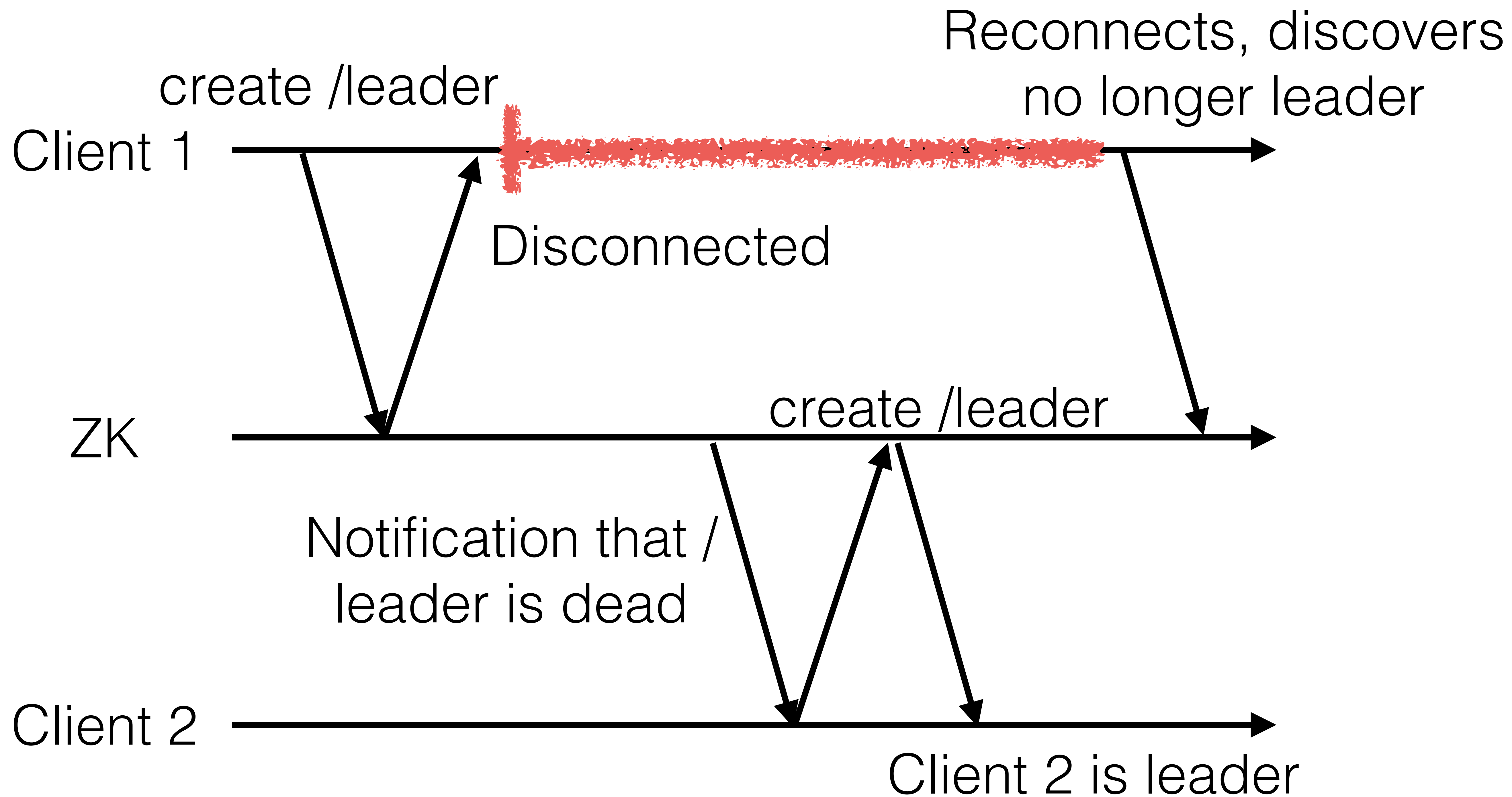
Failure Handling in ZK



Failure Handling in ZK

- If in the middle of an operation, client receives a **ConnectionLossException**
- Also, client receives a **disconnected message**
- Clients can't tell whether or not the operation was completed though - perhaps it was completed before the failure
- Clients that are disconnected can not receive any notifications from ZK

Dangers of ignorance



Dangers of ignorance

- Each client needs to be aware of whether or not its connected: when disconnected, can not assume that it is still included in any way in operations
- By default, ZK client will NOT close the client session because it's disconnected!
  - Assumption that eventually things will reconnect
  - Up to you to decide to close it or not

ZK: Handling Reconnection

- What should we do when we reconnect?
- Re-issue outstanding requests?
  - Can't assume that outstanding requests didn't succeed
  - Example: create /leader (succeed but disconnect), re-issue create /leader and fail to create it because you already did it!

Hypervisor (Case Study)

Hypervisor-Based Fault-Tolerance

THOMAS C. BRESSOUD
Isis Distributed Systems
and
FRED B. SCHNEIDER
Cornell University

Protocols to implement a fault-tolerant computing system are described. These protocols augment the hypervisor of a virtual-machine manager and coordinate a primary virtual machine with its backup. No modifications to the hardware, operating system, or application programs are required. A prototype system was constructed for HP's PA-RISC instruction-set architecture. Even though the prototype was not carefully tuned, it ran programs about a factor of 2 slower than a bare machine would.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart*; *fault tolerance*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Fault-tolerant computing system, primary/backup approach, virtual-machine manager

1. INTRODUCTION

One popular scheme for implementing fault tolerance involves replicating a computation on processors that fail independently. Replicas are coordinated so that they perform the same sequence of state transitions and, therefore, produce the same results. This article describes a novel implementation of that scheme. We interpose a software layer between the hardware and the operating system. The result is a fault-tolerant computing system whose implementation does not require modifications to the hardware, operating system, or any application software.

http://www.cs.cornell.edu/ken/book/2005sp/vft.tocs.pdf

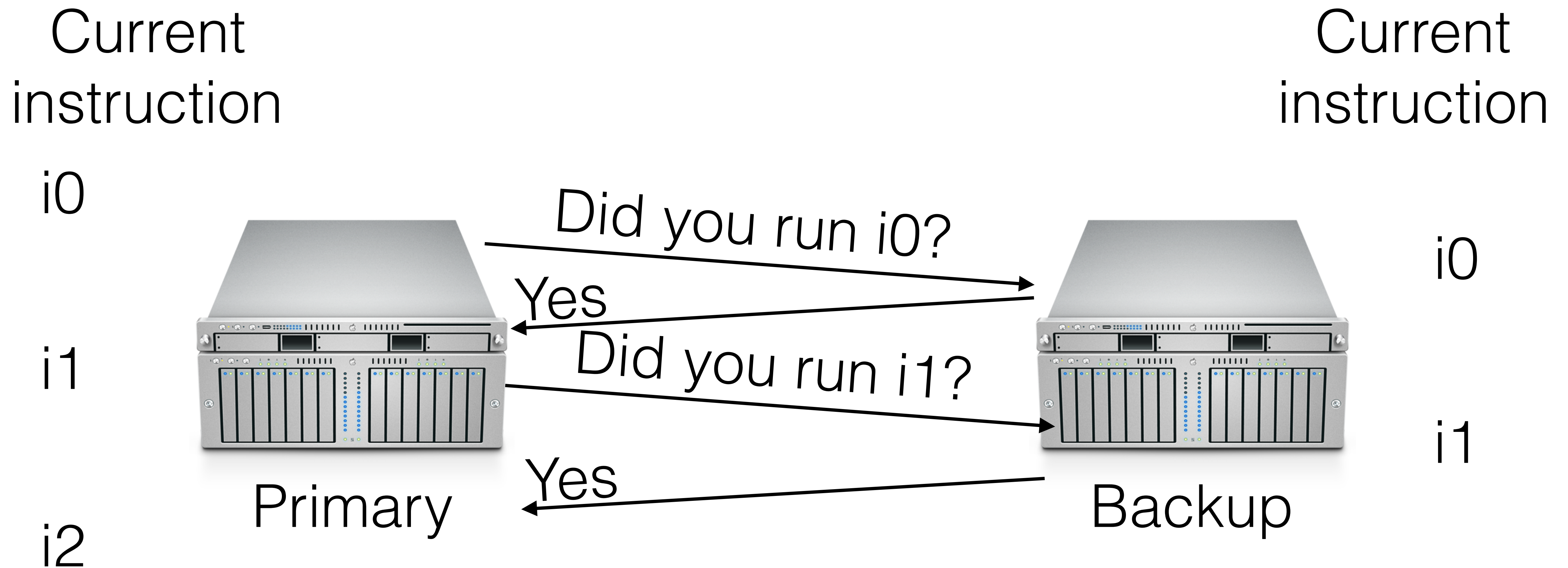
Hypervisor

- The year is 1996
- CPUs failed with enough frequency that it was really bad to be a bank, or NASA
- Primary/backup replication of entire VMs
- Avoid needing fault tolerant hardware, or fault tolerant software. Just make this VM fault tolerant.
- Replicates a state machine: the processor (i.e. replicates instructions)

Hypervisor: Strawman

- Start same program at same time on two CPUs
- Start with same memory/disk contents
- Machines are identical
- Will both get to the same result?

Hypervisor executes one instruction at a time



Hypervisor Challenges

- Synchronizing execution of single instructions: really slow
- How to handle I/O?
- What if an instruction is nondeterministic?

Grouping Instructions

- Observation: Do not need to synchronize EVERY instruction
- Only synchronize at externally-visible (I/O) instructions
- Intuition:
  - Do a bunch of deterministic instructions that do NOT interact with the outside world
  - If you crash, you can always re-run those
  - If you don't, replica can just acknowledge all instructions at once
- Helps a lot with limited I/O

I/O in Hypervisor

- I/O can be performed by either primary or backup
- But in normal operation, only primary does writes AND reads
  - Guarantees that backup never reads an old value
- Non-deterministic instructions work the same way: executed on primary and replicated to backup

Handling Failover

- If backup doesn't hear from primary in some amount of time (e.g. timeout), it takes over and becomes the primary
- What happens to pending I/O from the primary?
  - Backup always re-executes it
  - Assumes that I/O tolerates repeated operations (OK for network and disk, maybe not others?)

Caveats in Hypervisor

- Assumes perfect failure detection (uses simple timeout between two nodes)
- Assumes no network partitions
- Roughly factor of 2 slow down on evaluated hardware (50Mhz CPUs though)

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [4]: 3—*Distributed file systems*

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising

GFS (Google File System)

- Google apps observed to have specific R/W patterns (usually read recent data, lots of data, etc)
- Normal FS API (POSIX) is constraining (consider: CFS contains a ton of annoying glue to make it work)
- Hence, Google made their own FS

GFS

- Hundreds of thousands of regular servers
- Millions of regular disks
- Failures are normal
  - App bugs, OS bugs
  - Human Error
  - Disk failure, memory failure, network failure, etc
- Huge number of concurrent reads, writes

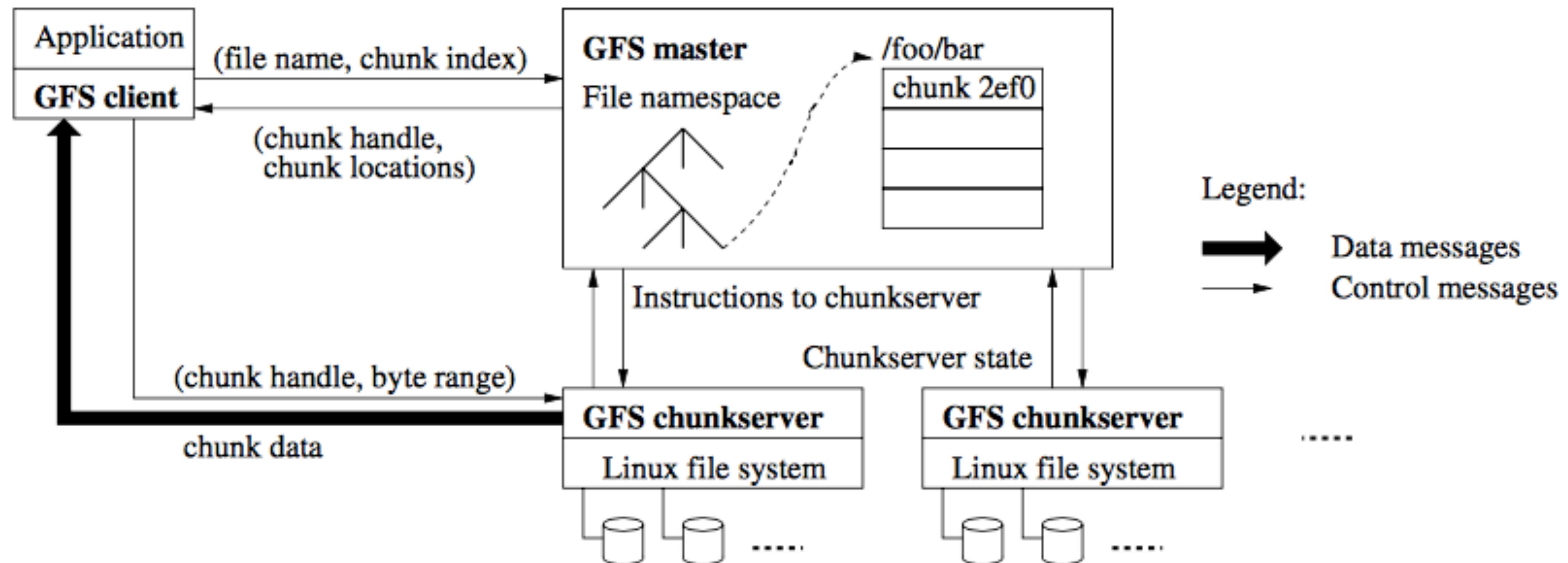
GFS Workload

- (Relatively) small total number of large files (>100MB) - millions
- Large, streaming reads (reading > 1MB at a time)
- Large, sequential writes that always append to end of a file
- Multiple clients might append concurrently

GFS Design Goals

- Unified FS for all google platforms (e.g. gmail, youtube)
- Data + system availability
- Graceful + transparent failure handling
- Low synchronization overhead
- Exploit parallelism
- High throughput and low latency

GFS Architecture



GFS Architecture

- Single master server (RSM replication to backups)
  - Holds all metadata (in RAM!) - namespace, ACL, file-chunk mapping
  - In charge of migrating chunks, GC'ing chunks
- Data stored in 64MB chunks each with some ID
  - Compare to EXT-4's 4KB block
- Thousands of chunk servers
  - Chunks are replicated
  - Chunk servers don't cache anything in RAM, store chunks as regular files

GFS Client

- Makes metadata requests to master server
- Makes chunk requests to chunk servers
- Caches metadata
- Does not cache data (chunks)
    - Google’s workload (streaming reads, appending writes) doesn’t benefit from caching, so why bother with consistency nightmare

GFS Reads

- Client asks master for chunk ID, chunk version number, and location of replicas given a file name
- By default, GFS replicates each chunk to 3 servers
- Client sends read request to closest (in network topology) chunk server

GFS Writes

- Client asks master for repliacs storing a chunk (one is arbitrarily declared primary)
- Client sends write request to all replicas
- Each replica acknowledges write to primary replica
- Primary coordinates commit between all of the replicas
- On success, primary replies to client

GFS Chunk Primaries

- There needs to be exactly one primary for each chunk
- GFS ensures this using *leases* (first time we talk about it, existed long before GFS)
  - Master selects a chunk server and grants it a lease
  - The chunk server holds the lease for T seconds, and is primary
  - Chunk server can *refresh* lease endlessly
  - If chunk server fails to refresh it, falls out of being primary
- Like a lock, but needs to be renewed (like with a heart beat)

GFS Fault Tolerance

- Master logs all client requests to disk
- Replicates log entries to remote backup server
- Only acknowledges changes to client after log entry is committed
- Only need to track the mapping from file-chunk ID
  - Recovery: ask all chunk servers what chunk IDs they have
- Master is in charge of replacing failed chunk servers

GFS Consistency

- Metadata changes are atomic. Occur only on a single machine, so no distributed issues.
- Changes to data are ordered as arbitrarily chosen by the primary chunk server for a chunk

GFS Summary

- Limitations:
  - Master is a huge bottleneck
  - Recovery of master is slow
- Lots of success at Google
- Performance isn't great for all apps
- Consistency needs to be managed by apps
- Replaced in 2010 by Google's Colossus system - eliminates master

Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

1 Introduction

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 [21] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automati-

tency over higher availability, as long as they can survive 1 or 2 datacenter failures.

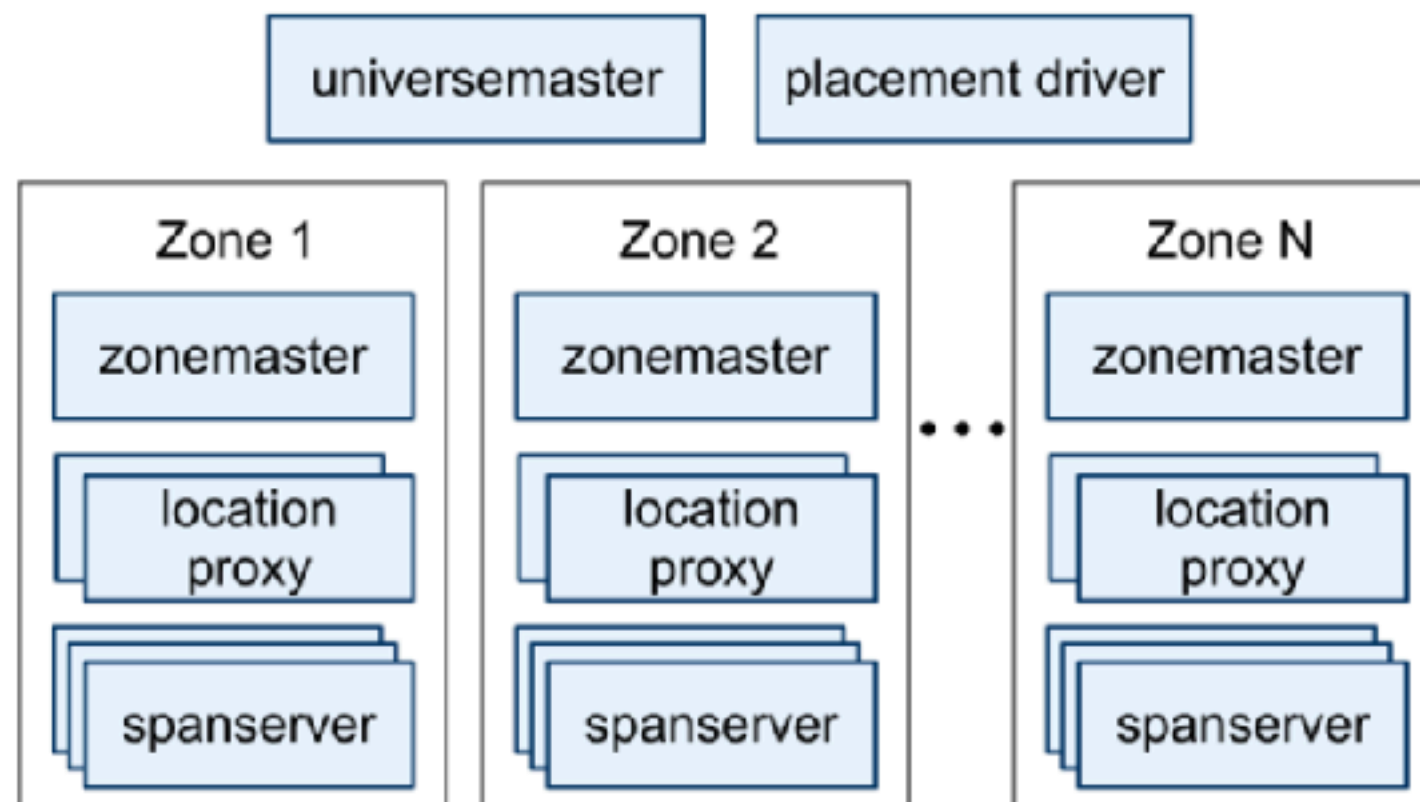
Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps.

Spanner

- Globally-distributed database system **with strong consistency**
- Replication used for availability and performance
- Paxos state machines used to manage sharded data
- Millions of machines across hundreds of data centers with trillions of rows of data

Spanner Deployment

- Entire deployment is called a universe
- Zone is a physical location
- Hundreds/thousands of spanservers in each zone



Spanservers

- Each spanserver stores a large amount of datastructures called tablets
- Each tablet runs a Paxos RSM
- Collection of Paxos RSMs managing a single tablet are called a Paxos group
- The Paxos group designates a leader for each tablet that maintains a lock table

Spanner concurrency control

- Key idea:
  - So far, it's been either "perfectly synchronize all clocks" (which we said we can't really do), or just order things relatively
  - What if we accept some degree of divergence in our clocks?
- TrueTime: global, wall-clock time, **but** with bounded uncertainty

Spanner Concurrency Control

- Uses GPS, atomic clocks to get reference time
- Compute worst-case drift... found to be ~ 1ms
- Rather than abandon "real" time stamps (because 1ms is "too much"), instead modify algorithms to accept the issue
- Only provide the consistency guarantees that you can: outside of this 1ms window, you can guarantee before/after ordering

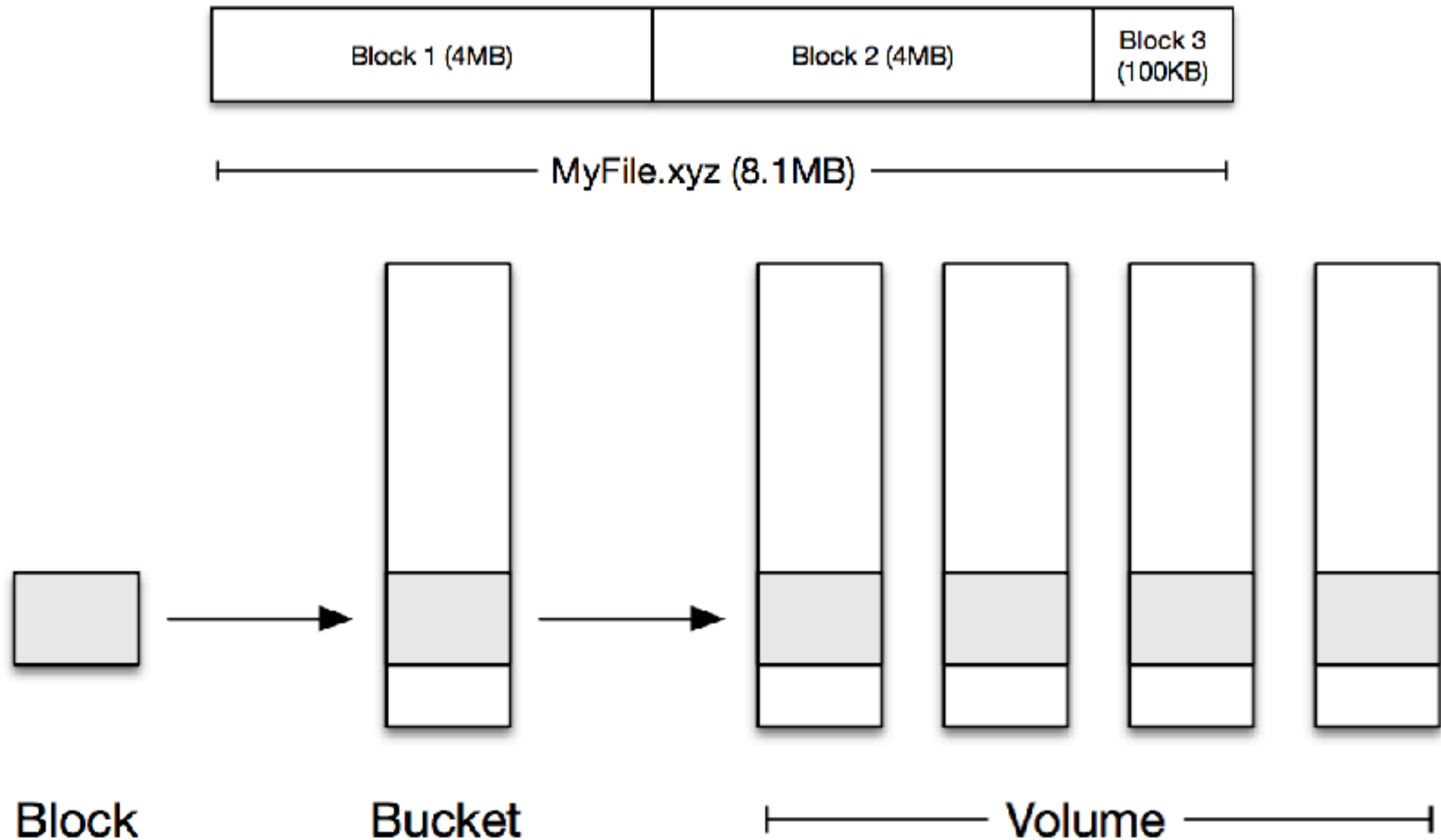
MagicPocket

- One last storage system
- Dropbox!
- https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire
- https://blogs.dropbox.com/tech/2016/05/inside-the-magic-pocket/
- http://www.se-radio.net/2017/03/se-radio-episode-285-james-cowling-on-dropboxs-distributed-storage-system/

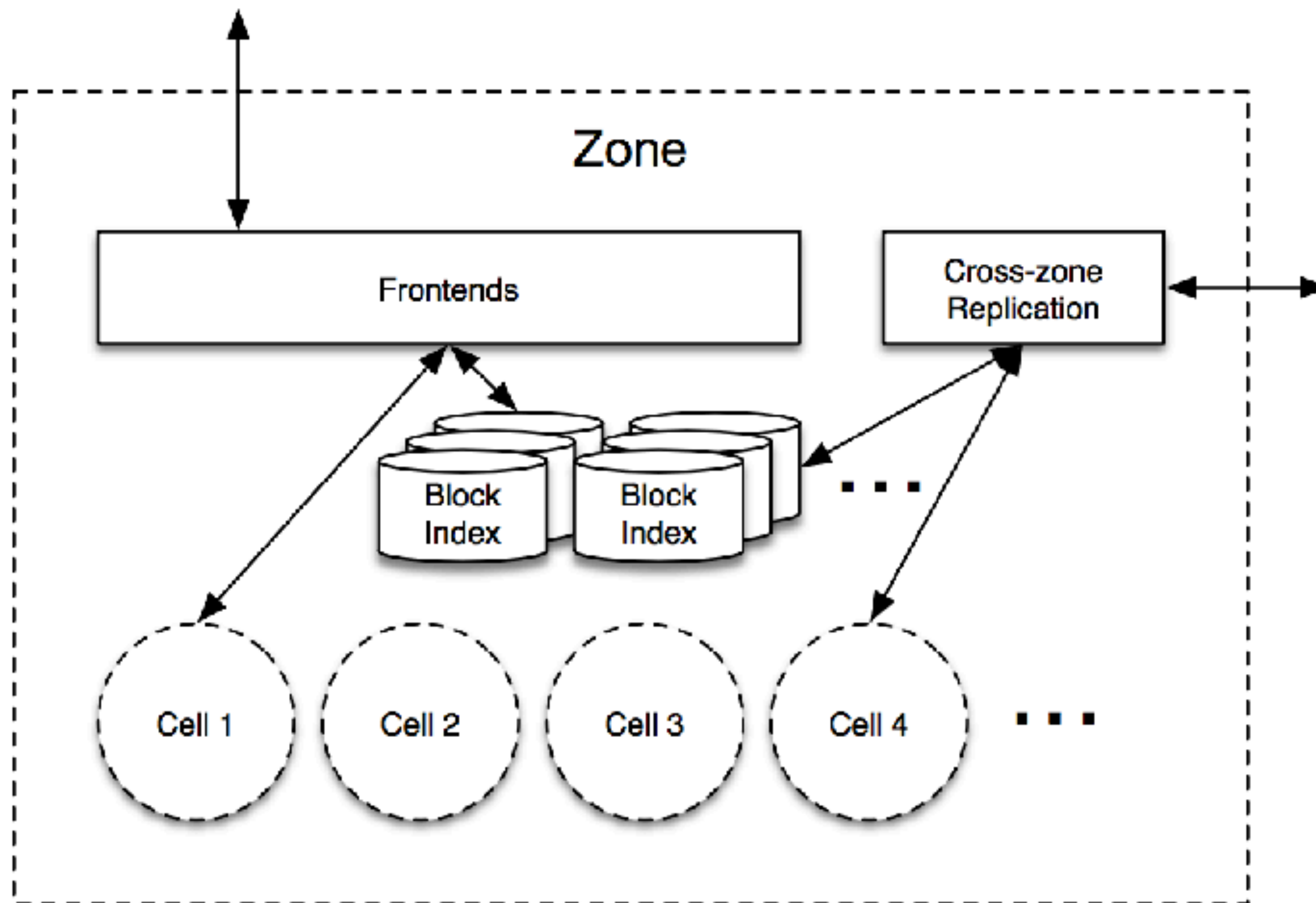
MagicPocket: High level

- Lots of similarities to GFS: split up metadata and chunks
- Chunks are immutable though - any changes result in new “files" -> only worry about consistency on metadata
- Weird workload: lots of writes, few reads; usually read only recent writes
  - Lots of caching and replication of new files, not so much for staler data
- Physically replicated across 3 data centers; each block stored in at least 2 data centers, then replicated within each zone

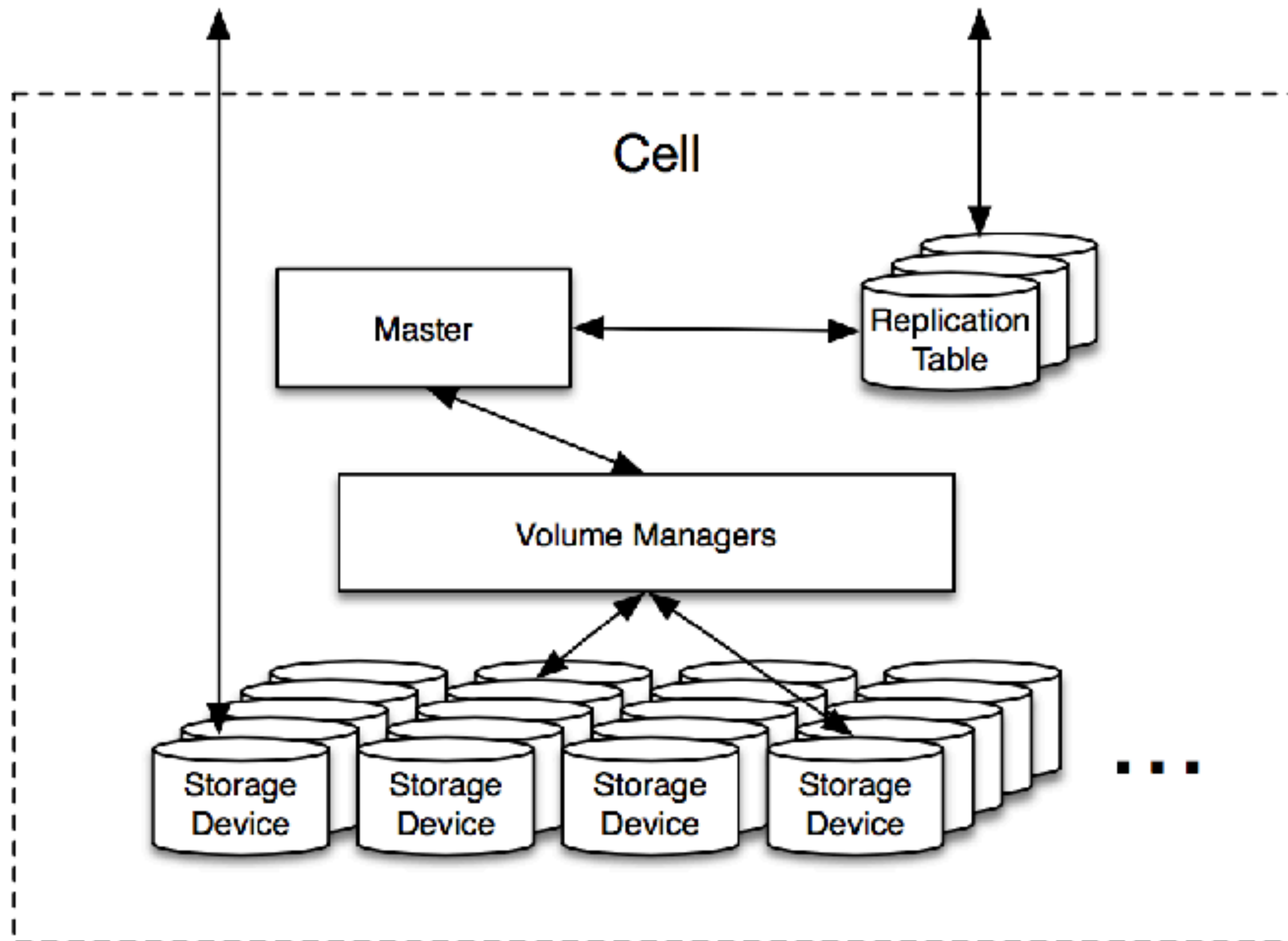
MagicPocket: Blocks



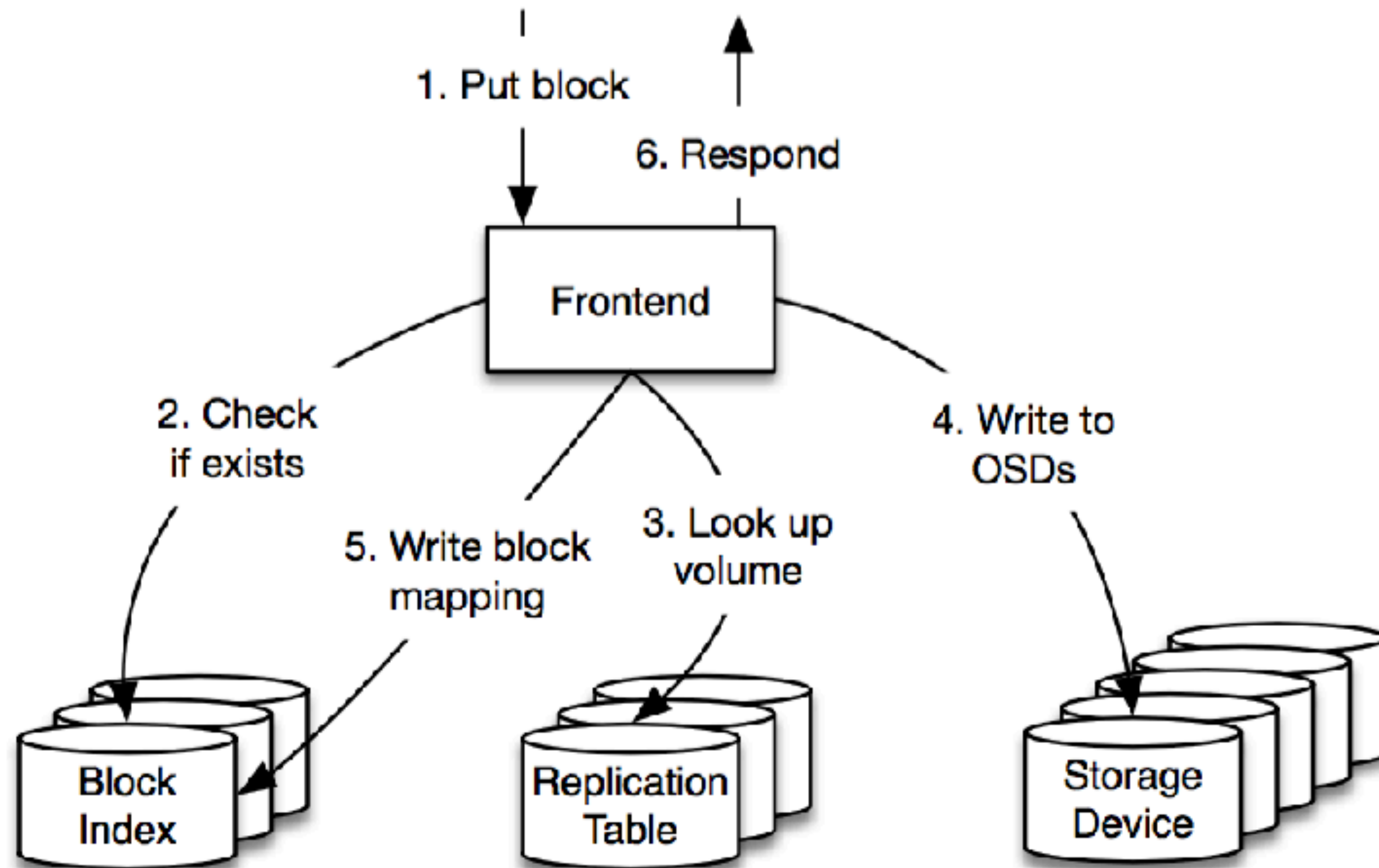
MagicPocket: Zones



MagicPocket: Cells



MagicPocket: Writes



MagicPocket: Recovery

- Master in a cell is responsible for repairing disk failures/moving data around
- Metadata is simply replicated using MySQL primary-backup mirroring (which has its own story for recovery)