CS4740
CLOUD COMPUTING

Case study: ZooKeeper

Prof. Chang Lou, UVA CS, Fall 2025

ZooKeeper: wait-free coordination for internet-scale systems

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, Benjamin Reed

USENIX ATC 2010

AGENDA

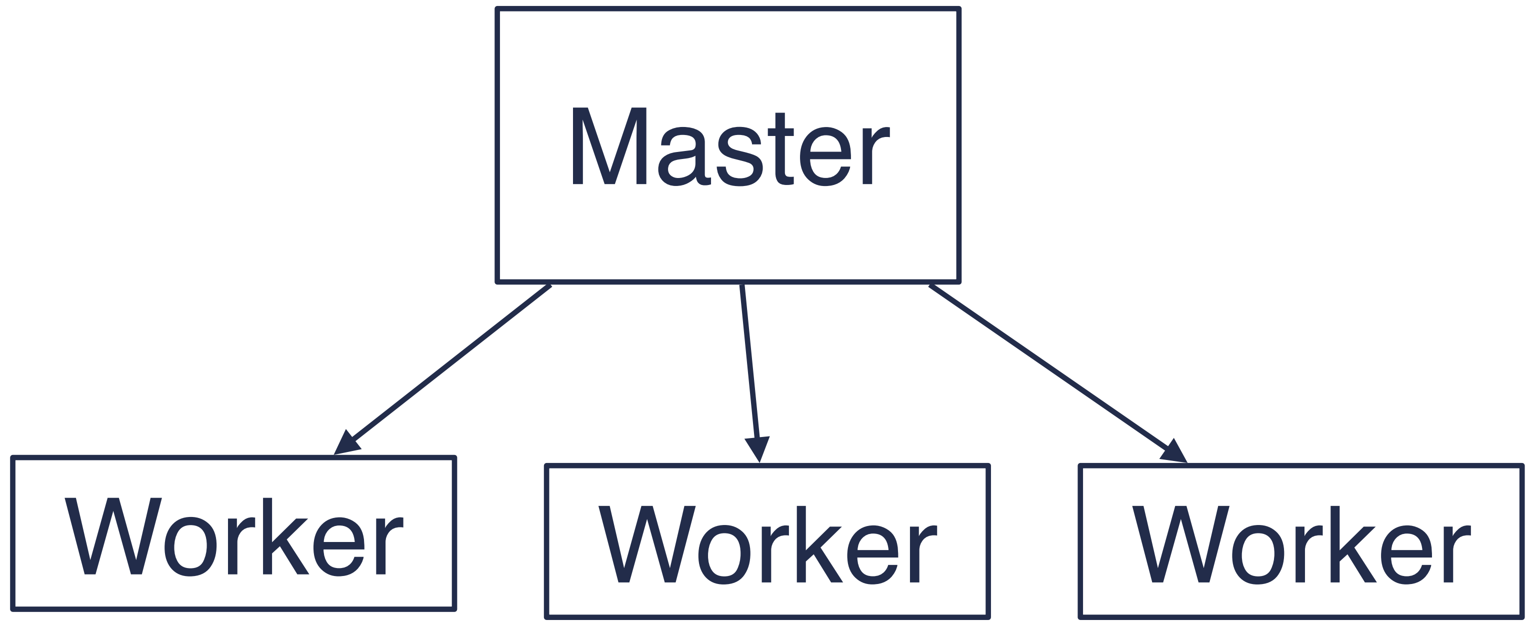
- Motivation: MapReduce Coordinator
- Programming with ZooKeeper
  - Interface
  - Code Example
- Performance

WHY ARE WE READING THIS PAPER?

- A simpler foundation for fault-tolerant applications.
  - the go-to application if you just start learning distributed systems
- High-performance in a real-life service built on Raft-like replication.
- Also because it's very widely used.

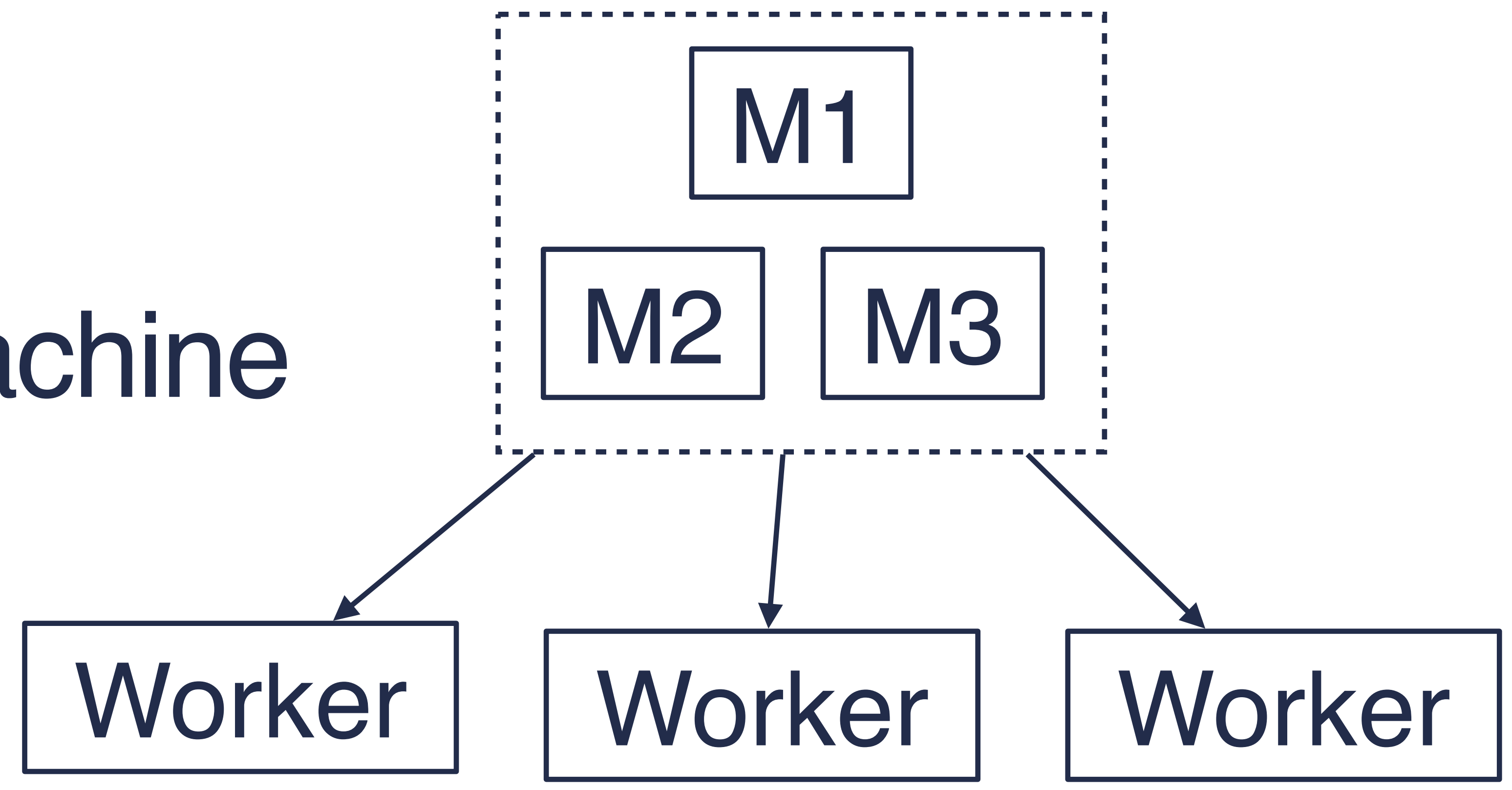
PROBLEM WITH SINGLE COORDINATOR

- if we wanted to make a fault-tolerant service like MR coordinator,
  - we could replicate with Raft, and that would be OK!



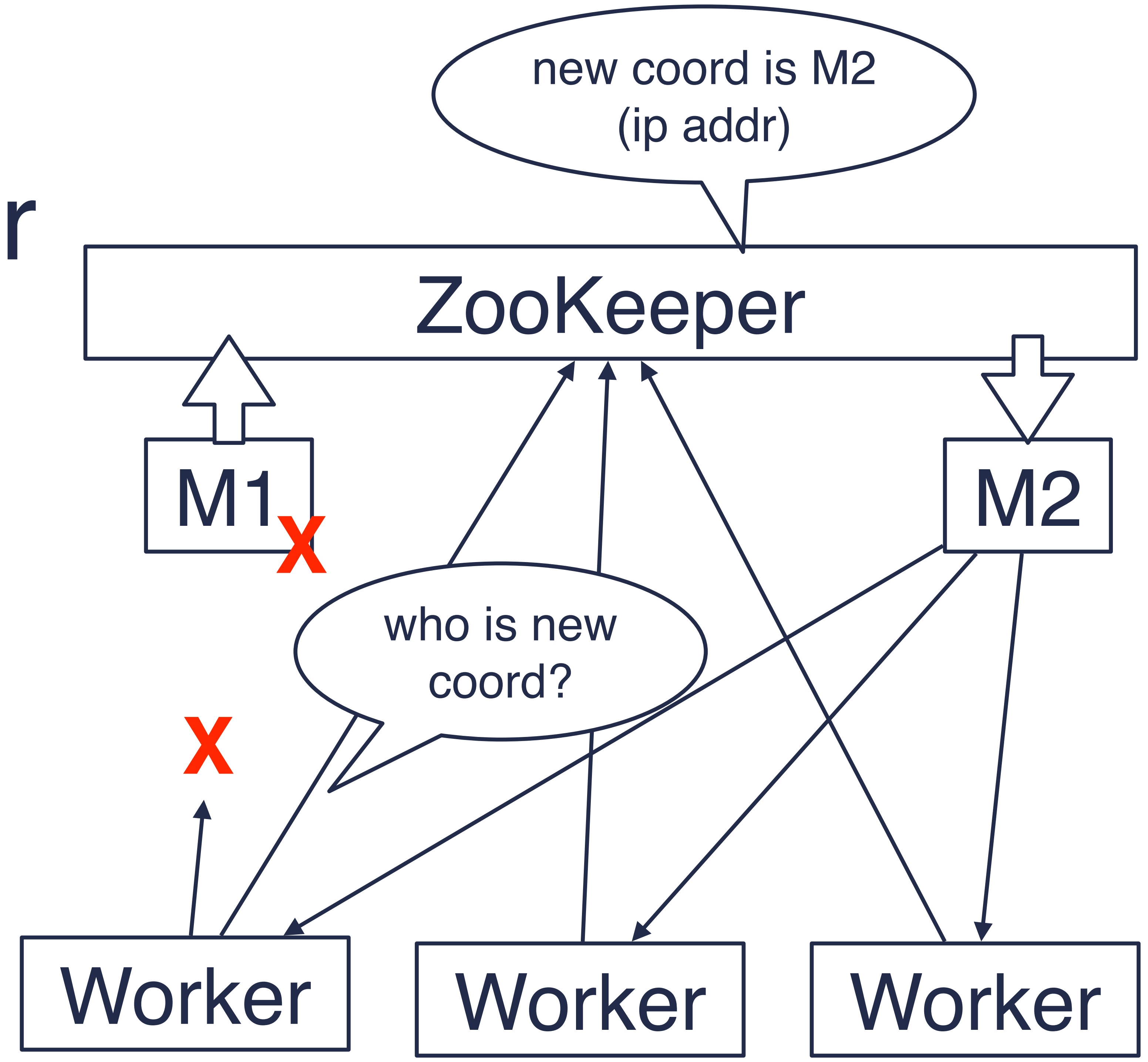
PROBLEM WITH SINGLE COORDINATOR

- but building directly on Raft is complex
  - a replicated state machine is awkward to program
- you can think of state machine replication (Raft) as replicating
  - the computation; the state is replicated as a side-effect.
- can we replicate state **without** replicating computation?
  - yes: use fault-tolerant storage, for example ZooKeeper
  - easier to write the MR coord than with replicated state machine
  - ordinary straight-line code, plus "save" calls



NEW PARADIGM W/ ZOOKEEPER

- now what if MR coord fails?
- we weren't replicating it on a backup coord server
  - but we don't need one!
- just pick any computer, start MR coord s/w on it,
  - have it read state from ZK.
- new coord can pick up where failed one left off.
- makes the most sense in a cloud
  - easy to allocate a replacement server



WHAT MIGHT MR COORD STORE IN ZK?

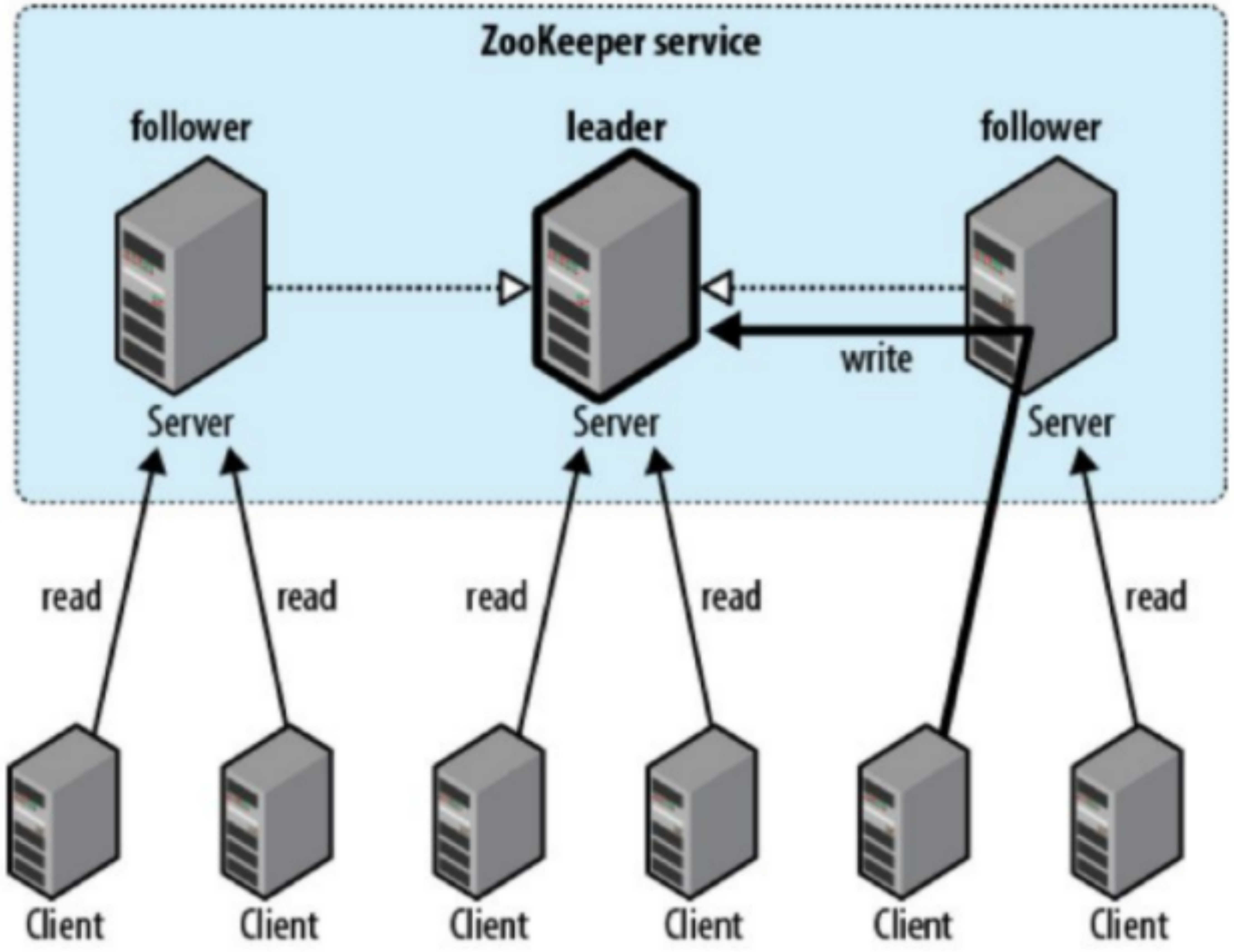
- coord's IP addr, set of jobs, status of tasks, set of workers, assignments
- update data in ZK on each change
- (but big data itself in GFS, not ZK)
- ZK acting as a "configuration service"
  - helps MR coord and worker find each other

NEXT: CHALLENGES

- detect MR coord failure
- elect new MR coord (one at a time! no split brain!)
- new coord needs to be able to recover/repair state read from ZK
  - what if old coord crashed midway through complex update?
- what if old coord doesn't realize it's been replaced
  - can it still read/write state in ZK?
  - can it affect other entities incorrectly? e.g. tell workers to do things?
- performance

ZooKeeper Architecture

ARCHITECTURE



ZooKeeper Interface

ZOOKEEPER DATA MODEL

- the state: a file-system-like tree of znodes
- file names, file content, directories, path names
  - directories help different apps avoid interfering
- each znode has a version number (?)
- types of znodes:
  - regular
  - ephemeral
  - sequential: name + seqno

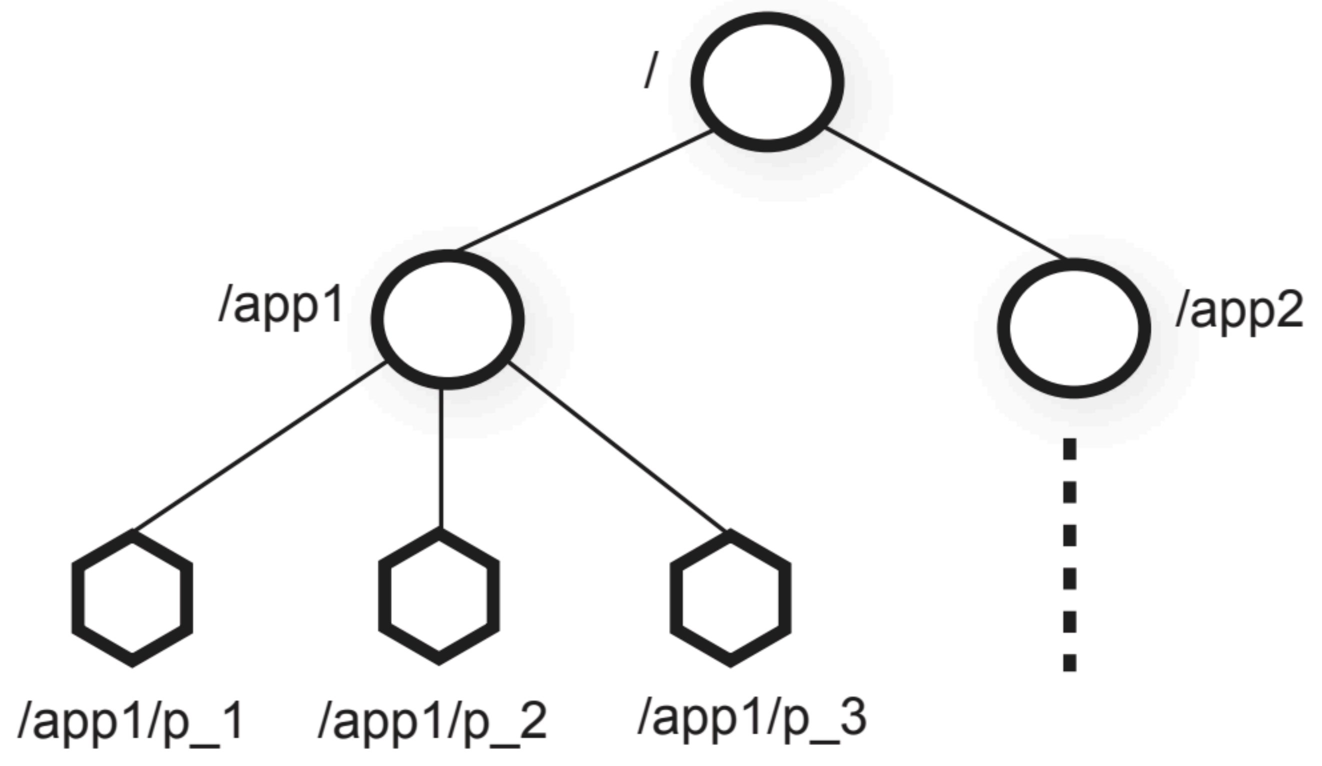


Figure 1: Illustration of ZooKeeper hierarchical name space.

OPERATIONS (CREATE)

- s = openSession()
- create(s, path, data, flags)
  - exclusive -- fails if path already exists
  - flags specify types: regular, ephemeral, sequential

```
s = openSession()
create(s, "/r")
getChildren(s, "/")
=> /r
create(s, "/r")
=> false, already exists
```

```
s = openSession()
create(s, "/r", ephemeral=true)
getChildren(s, "/")
=> /r
closeSession(s) // or crash
s2 = openSession()
getChildren(s2, "/")
=> null
```

```
s = openSession()
create(s, "/r", sequential=true)
create(s, "/r", sequential=true)
getChildren(s, "/")
=> /r0000000001, /r0000000002
```

OPERATIONS (OTHERS)

- exists(s, path, watch)
  - **watch**=true asks for notification if path is later created/deleted
- getData(s, path, watch) -> data, version
- setData(s, path, data, version)
  - if znode.version = version, then update
- getChildren(s, path, watch)
- delete(s, path, watch)
- these throw an exception if the ZK server says it has terminated the session
  - so that application won't continue

OPERATIONS

– ZooKeeper API well tuned for concurrency and synchronization:
  – + exclusive file creation; exactly one concurrent create returns success
  – + getData()/setData(x, version) supports mini-transactions
  – + sessions help cope with client failure (e.g. release locks)
  – + sequential files create order among multiple clients
  – + watches avoid costly repeated polling

Programming Example

EXAMPLE: SIMPLE LOCK

Lock

```
s = createSession
while true:
    if create(s, "/lock", ephemeral=true)
        // go ahead and do stuff
    else if exists(s, "/lock", watch=true)
        wait for watch event
```

Unlock

```
delete(s, "/lock")
```

Problem: Herd effect
If many clients wait for a lock, they will all vie for the lock when it is released but one client can get the lock.

EXAMPLE: LOCK WITHOUT HERD EFFECT

Lock

```
s = createSession
n = create(s, "/lock/lock-",
ephemeral=true, sequential=true)
while true:
   C= getChildren(s, "/lock", false)
   if n is lowest znode in C, break
   p = znode in C ordered just before n
   if exists(s, p, watch=true)
      wait for watch event
```

Unlock

```
delete(s, n)
```

Why this design works?

EXAMPLE: LOCK WITHOUT HERD EFFECT

- Q: could a lower-numbered file be created after getChildren()?
- Q: can watch fire before it is the client's turn?

```
lock-10 <- current lock holder
lock-11 <- next one
lock-12 <- my request
```

- A: yes
- if client that created lock-11 dies before it gets the lock, the
- watch will fire but it isn't my turn yet.

EXAMPLE: MAPREDUCE COORDINATOR ELECTION

```
s = openSession()
while true:
  if create(s, "/mr/c", ephemeral=true)
    // we are the coordinator!
    setData(s, "/mr/ip", ...)
  else if exists(s, "/mr/c", watch=true)
    // we are not the coordinator
    wait for watch event
```

- exclusive create
  - if multiple clients concurrently attempt, only one will succeed
- ephemeral znode
  - coordinator failure automatically lets new coordinator be elected
- watch
  - potential replacement coordinators can wait w/o polling

REQUIREMENTS FOR SOLUTION

- * want to elect a replacement
- * must cope with crash in the middle of updating state in ZK
- * must cope with possibility that the coordinator *didn't* fail!

REQUIREMENT 1: ELECT NEW COORD

- client failure -> client stops sending keep-alive messages to ZK
- no keep-alives -> ZK leader times out and terminates the session
- session termination -> ZK leader deletes session's ephemeral files
- and ignores further requests from that session
- ephemeral deletions are A-linearizable ZK ops
- now a new MR coordinator can elect itself

REQUIREMENT 2: ATOMICITY

- what if the MR coordinator crashes while updating state in ZK?
- maybe store all data in a single ZK file
  - individual setData() calls are atomic (all or nothing vs failure)
- what if there are multiple znodes containing state data?
  - use paper's "ready" file scheme

REQUIREMENT 3: DEAL WITH OLD COORD

- what if the coordinator is alive and thinks it is still coordinator, but ZK has decided it is dead and deleted its ephemeral /mr/c file?
- a new coordinator will likely be elected.
- will two computers think they are the coordinator?
  - this could happen.
- can the old coordinator modify state in ZK?
  - this cannnot happen!

REQUIREMENT 3: DEAL WITH OLD COORD

- when ZK times out a client's session, two things happen atomically:
  - ZK deletes the clients ephemeral nodes.
  - ZK stops listening to the session -- will reject all operations.
- so old coordinator can no longer modify or read data in ZK!
  - if it tries, its client ZK library will raise an exception
  - forcing the client to realize it is no longer coordinator

REQUIREMENT 3: DEAL WITH OLD COORD

- an important pattern in distributed systems:
- a single entity (e.g. ZK) decides which computers are alive or dead
  - sometimes called a failure detector
- it may not be correct, e.g. if the network drops messages
- but everyone **obeys** its decisions
- agreement is more important than being right, to avoid split brain
- but possibility of being wrong => may need to fence

REQUIREMENT 3: DEAL WITH OLD COORD

- what if coordinator interacts with entities (e.g., workers) other than ZK?

- that don't know about the coordinator's ZK session state.
- they may need to fence (i.e. ignore deposed coordinator) -- how?

- idea: worker could "watch" leader znode in ZK to learn of changes.
  - not perfect: window between change and watch notification arrival.

REQUIREMENT 3: DEAL WITH OLD COORD

- idea: each new coordinator gets an increasing "epoch" number.
  - from a file in ZK.
  - coordinator sends epoch in each message to workers.
  - workers remember highest epoch they have seen.
  - workers reject messages with epochs smaller than highest seen.
  - so they'll ignore a superseded coordinator once they
  - see a newer coordinator.

Performance

PERFORMANCE OPTIMIZATION

- Data must fit in memory, so reads are fast (no need to read disk).
  - So you can't store huge quantities of data in ZooKeeper.
- Writes (log entries) must be written to disk, and waited for.
  - So committed updates aren't lost in a crash or power failure.
  - Hurts latency; batching can help throughput.
- Periodically, complete snapshots are written to disk.
  - Fuzzy technique allows snapshotting concurrently with write operations.

PERFORMANCE OPTIMIZATION

- emphasis is on handling many reading/watching clients
- 1) many ZK follower servers; clients are spread over them for parallelism
- client sends all operations to its ZK follower
- ZK follower executes reads locally, from its replica of ZK data
- to avoid loading the ZK leader
- ZK follower forwards writes to ZK leader
- 2) watch, not poll
- the ZK follower (not the ZK leader) does the work

PERFORMANCE OPTIMIZATION

- 3) clients of ZK can launch async operations
- i.e. send request; completion notification delivered to code separately
- unlike RPC
- a client can launch many writes without waiting
- ZK processes them efficiently in a batch; fewer msgs, disk writes
- client library numbers them, ZK executes them in that order
- e.g. to update a bunch of znodes then create "ready" znode

HOW IS THE PERFORMANCE?

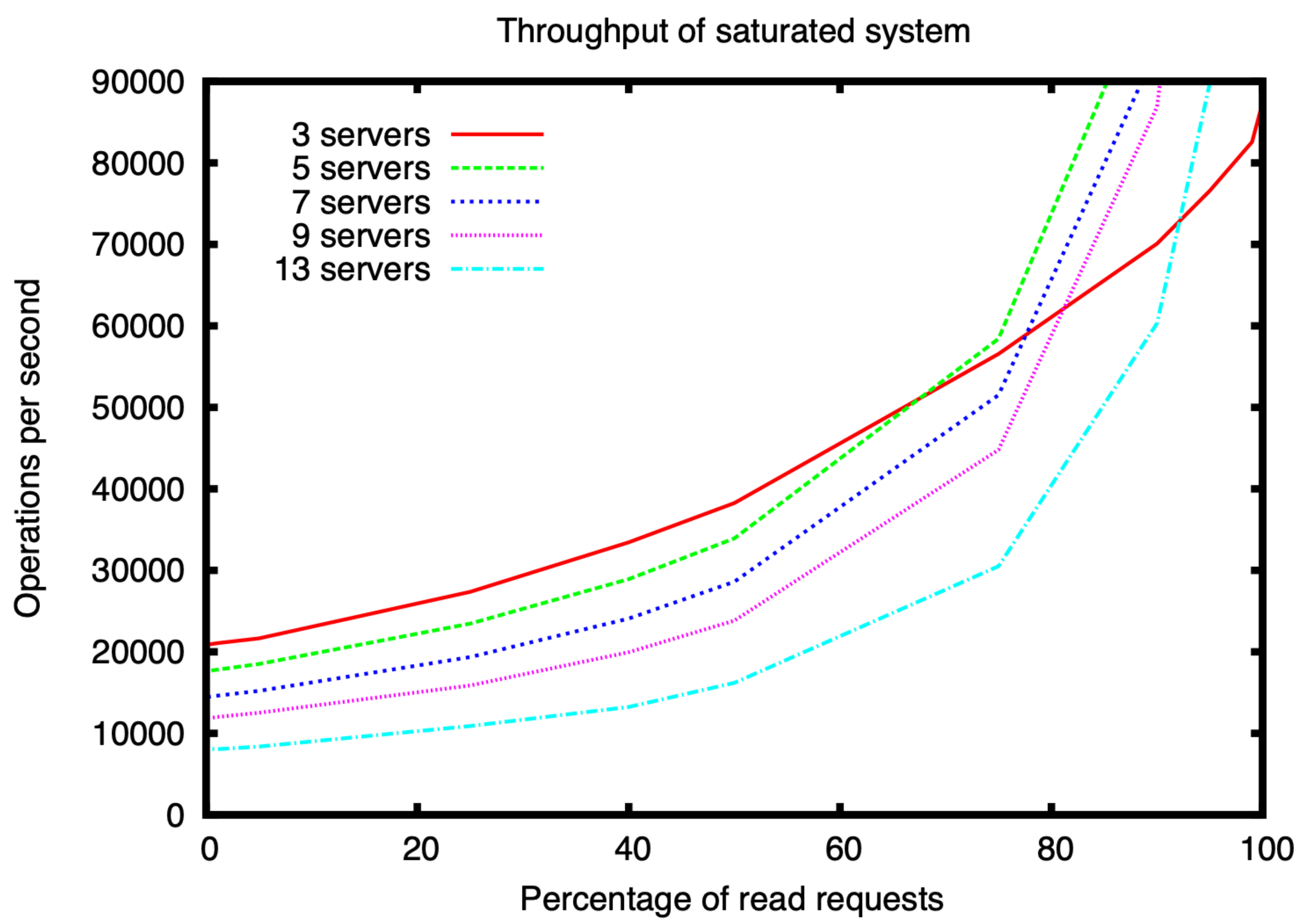


Figure 5: The throughput performance of a saturated system as the ratio of reads to writes vary.

- Overall, can handle 10s of thousands of operations / second.
  - Is this a lot? Enough?
- Why do the lines go up as they move to the right?
- Why does the x=0 performance go down as the number of servers increases?
- Why does the "3 servers" line change to be worst at 100% reads?
- What might limit it to 20,000? Why not 200,000?
  - Each op is a 1000-byte write...

WHAT ABOUT RECOVERY TIME?

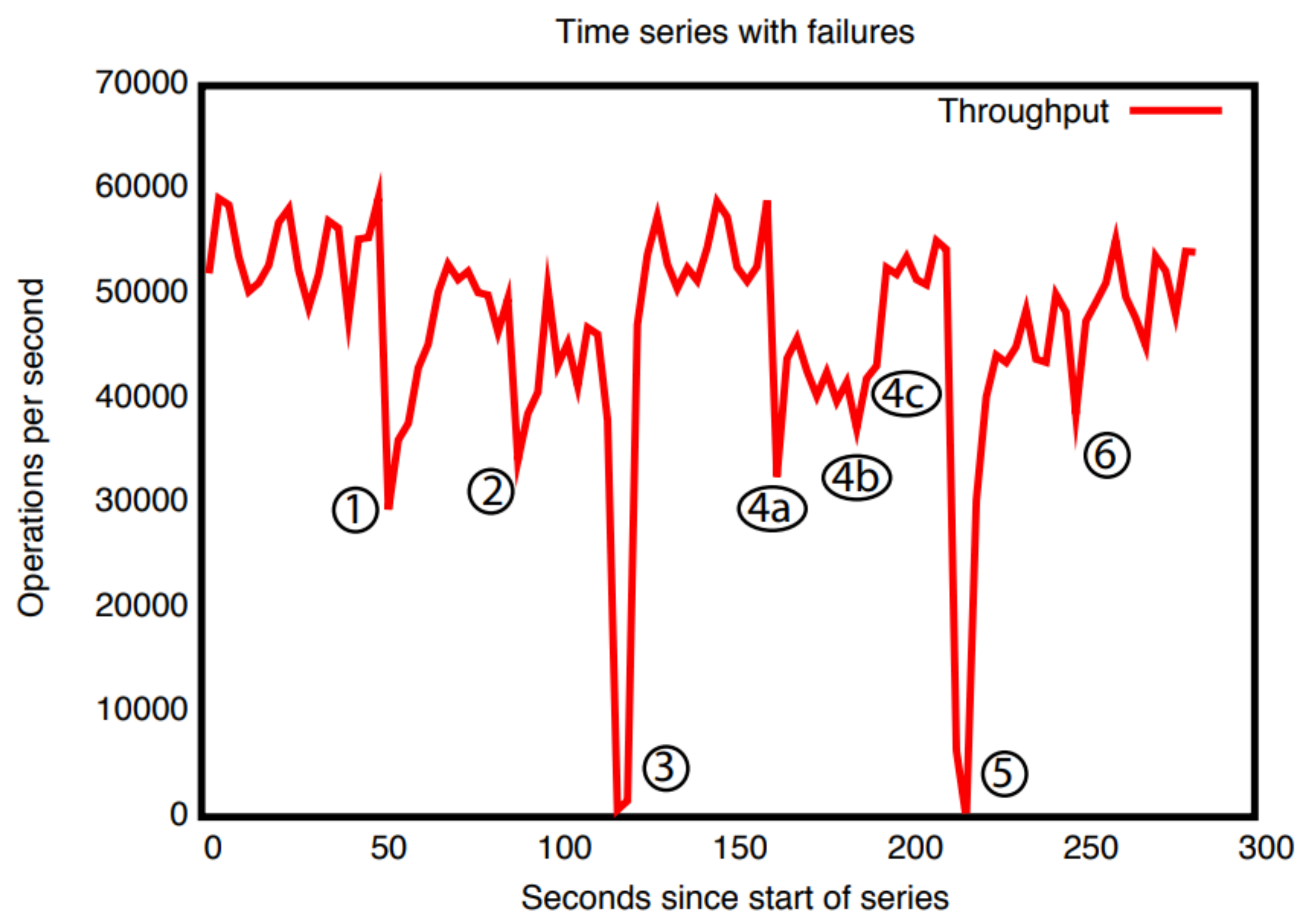


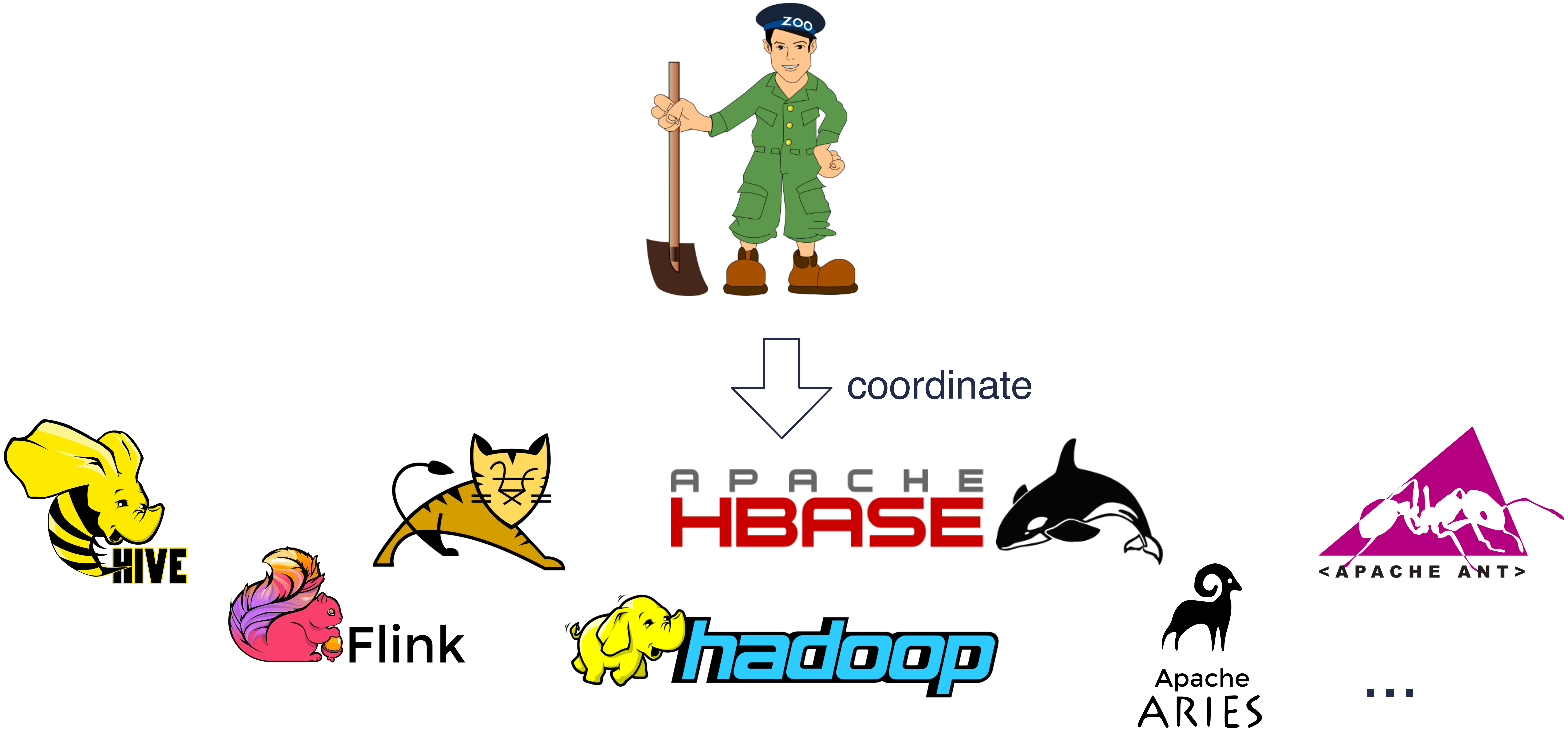
Figure 8: Throughput upon failures.

- Follower failure -> just a decrease in total throughput.
- Leader failure -> a pause for timeout and election.
  - Visually, on the order of a few seconds.

ZOOKEEPER IS VERY WIDELY USED

- see ZooKeeper's Wikipedia page for a list of projects that use it
- often used as a kind of fault-tolerant name service
  - what's the current coordinator's IP address? what workers exist?
- can be used to simplify overall fault-tolerance strategy
  - store all state in ZK e.g. MR queue of jobs, status of tasks
  - then service servers needn't themselves replicate

WHY IT IS CALLED ZOOKEEPER?



NEXT MONDAY: LAB DAY

- We will implement a Leader Election service with ZooKeeper
- Please setup the ZooKeeper on your laptop before class!

Process 1:	Process 2:	Process 3:
I joined the cluster. I became leader!		
	I joined the cluster. I am following node 1!	I joined the cluster. I am following node 1!
I left the cluster.		
	I became leader!	I am following node 2!
I joined the cluster. I am following node 2!		

TAKEAWAYS

–Zookeeper provides simple but convenient interface for coordination.

– Key concepts: session, znode types, watch..

–Efficiency and correctness guarantees depend on how clients use them.

–Next class: [**Lab Day] Implement Leader Election with ZooKeeper**

University of Virginia

ACKNOWLEDGEMENT

THIS COURSE IS DEVELOPED HEAVILY BASED ON COURSE MATERIALS SHARED BY PROF. INDRANIL GUPTA, PROF. ROBERT MORRIS, PROF. MICHAEL FREEDMAN, PROF. KYLE JAMIESON, PROF. WYATT LLOYD AND PROF. ROXANA GEAMBASU. MANY APPRECIATIONS FOR GENEROUSLY SHARING THEIR MATERIALS AND TEACHING INSIGHTS.