ZooKeeper: Wait-Free Coordination for Internet-Scale systems

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

Distributed Systems
ECE419

These slides are derived from the original slides by:

Patrick Hunt and Mahadev (Yahoo! Grid)
Flavio Junqueira and Benjamin Reed (Yahoo! Research)

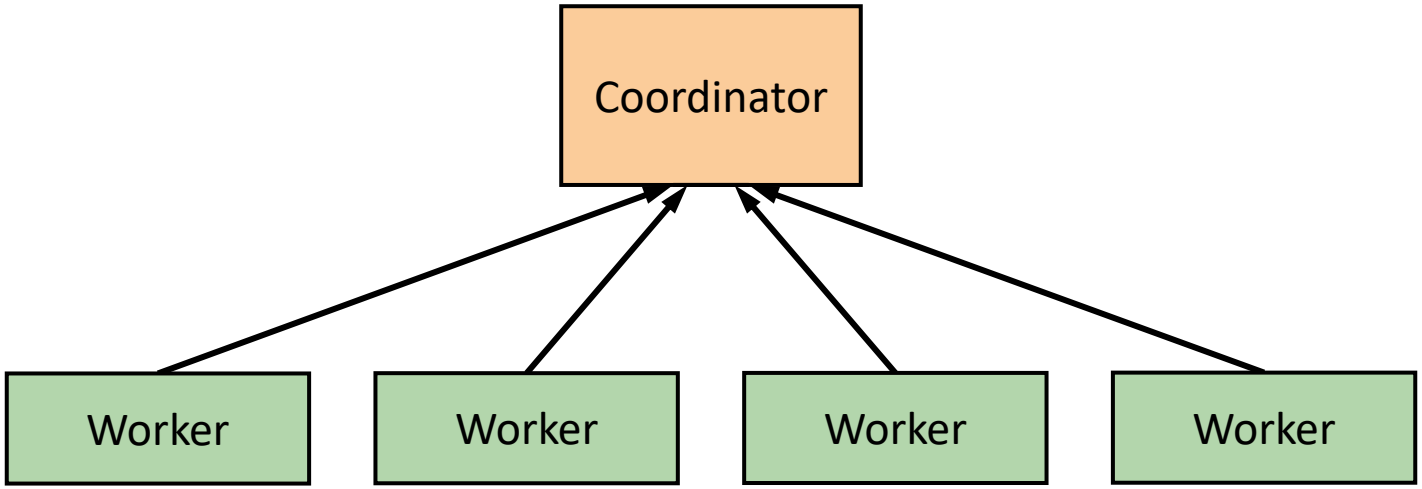
Overview

- What is ZooKeeper?
- ZooKeeper API
- Coordination recipes
- Consistency guarantees
- Handling failures
- Implementation

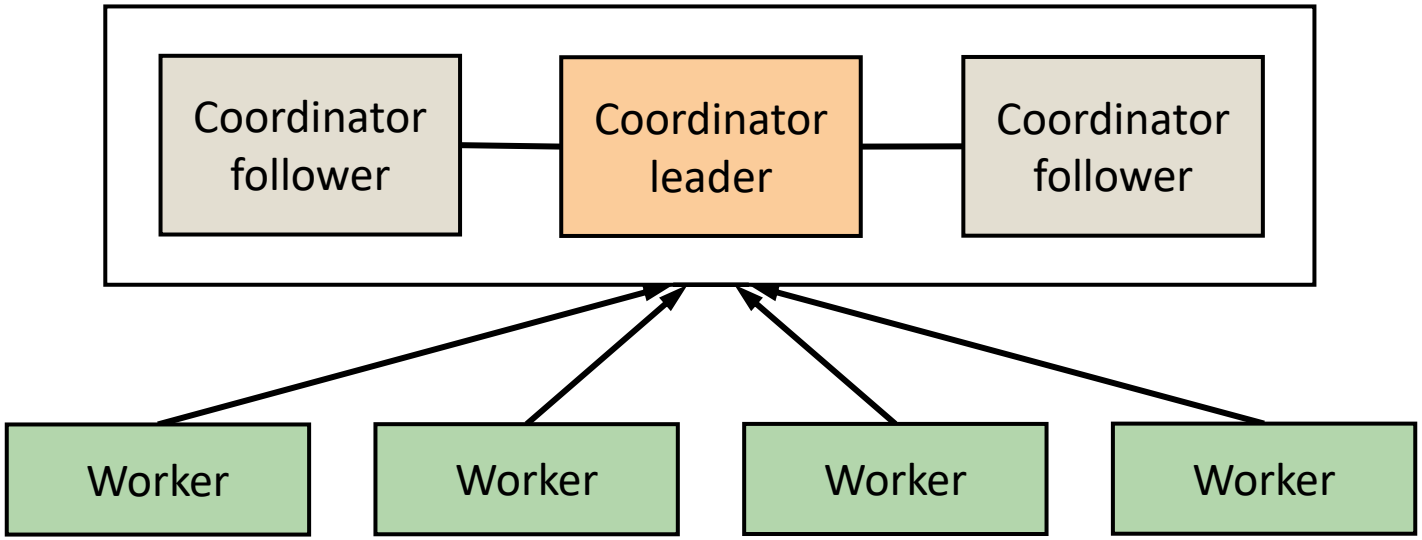
What is coordination?

- Group of nodes (or processes) need to manage their interactions to perform some common tasks
  - Configuration management: change, use configuration values
  - Synchronization: locking, barriers
  - Leader election: select leader, let others know about leader
  - Group membership: get list of current members
  - Status monitoring: monitor processes, machines, users, etc.

Classic coordination

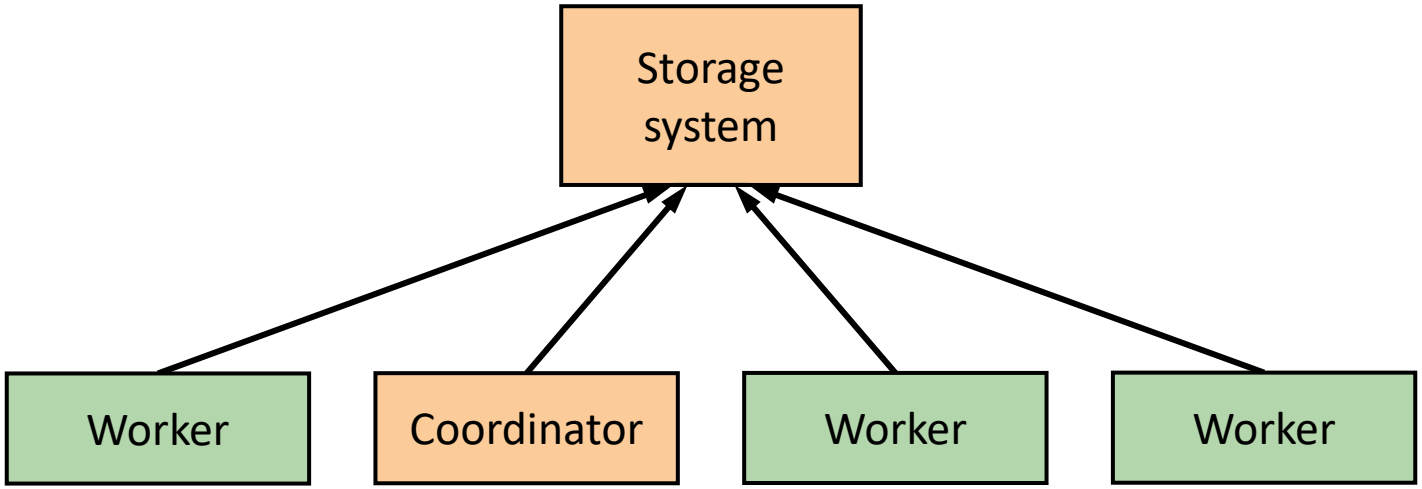


Fault-tolerant coordination



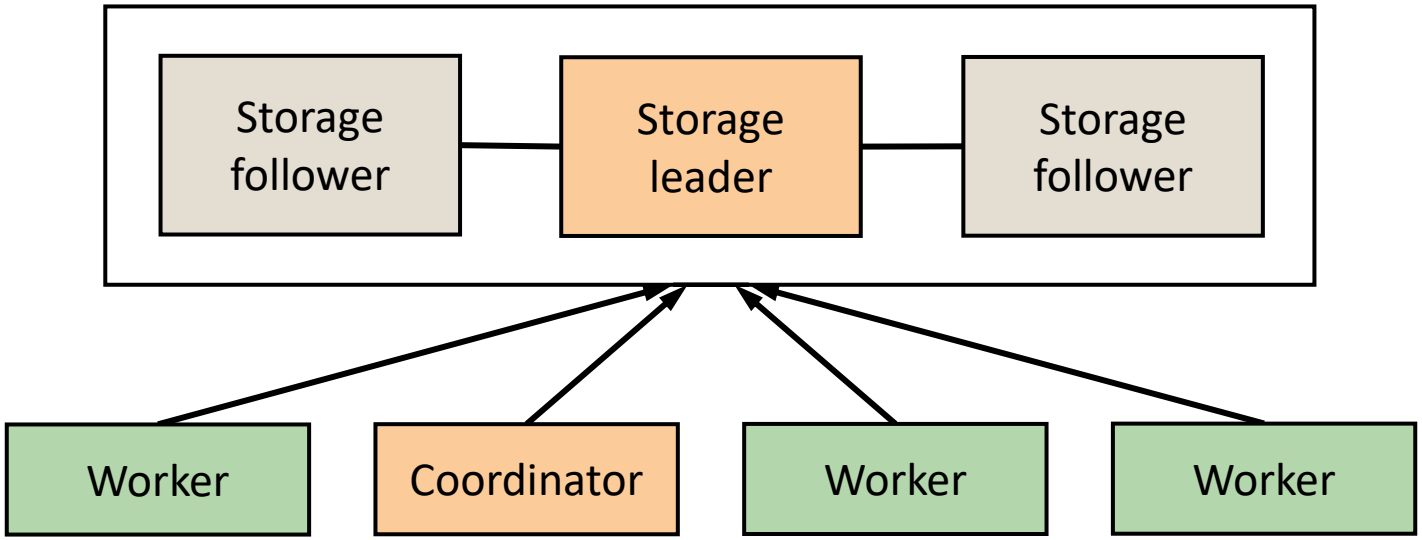
- Use state machine replication for fault tolerance
- Issues
  - Programming coordinator state machine is complicated
  - Computation is replicated, expensive
  - Coordinator can become bottleneck

Storage-based coordination



- Maintain coordinator state in separate storage system
  - E.g., IP of current coordinator, set of workers, task assignments
- Coordinator, workers coordinate via storage write/read
- Any worker can be coordinator

Fault-tolerant storage system



- Replicate storage (instead of computation) for fault tolerance
  - Coordinator code is simpler since no state machine needed
- What happens when coordinator fails?
  - Any another worker can take over, but this may take some time

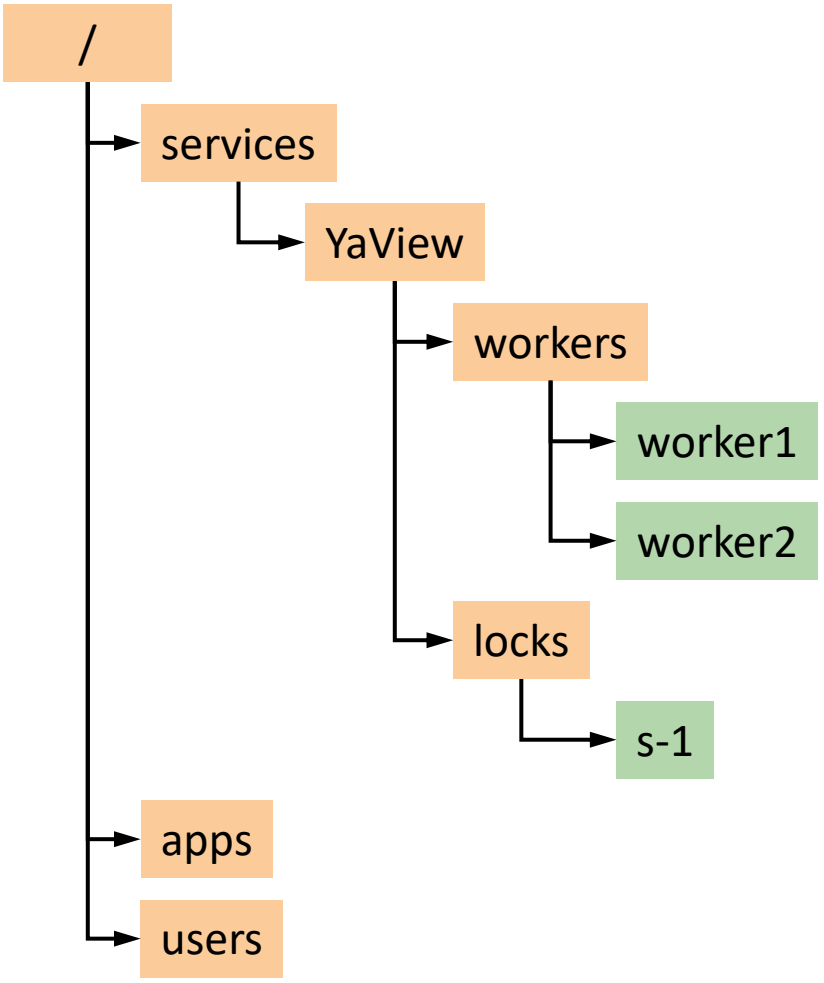
What is ZooKeeper?

- A fault-tolerant storage system that provides general coordination services, i.e., coordination kernel
  - E.g., group membership, locks, leader election, etc.
- Provides high performance
  - Allows multiple outstanding operations by a client
  - Reads are fast (although they may return stale data)
- Reliable and easy to use

ZooKeeper API

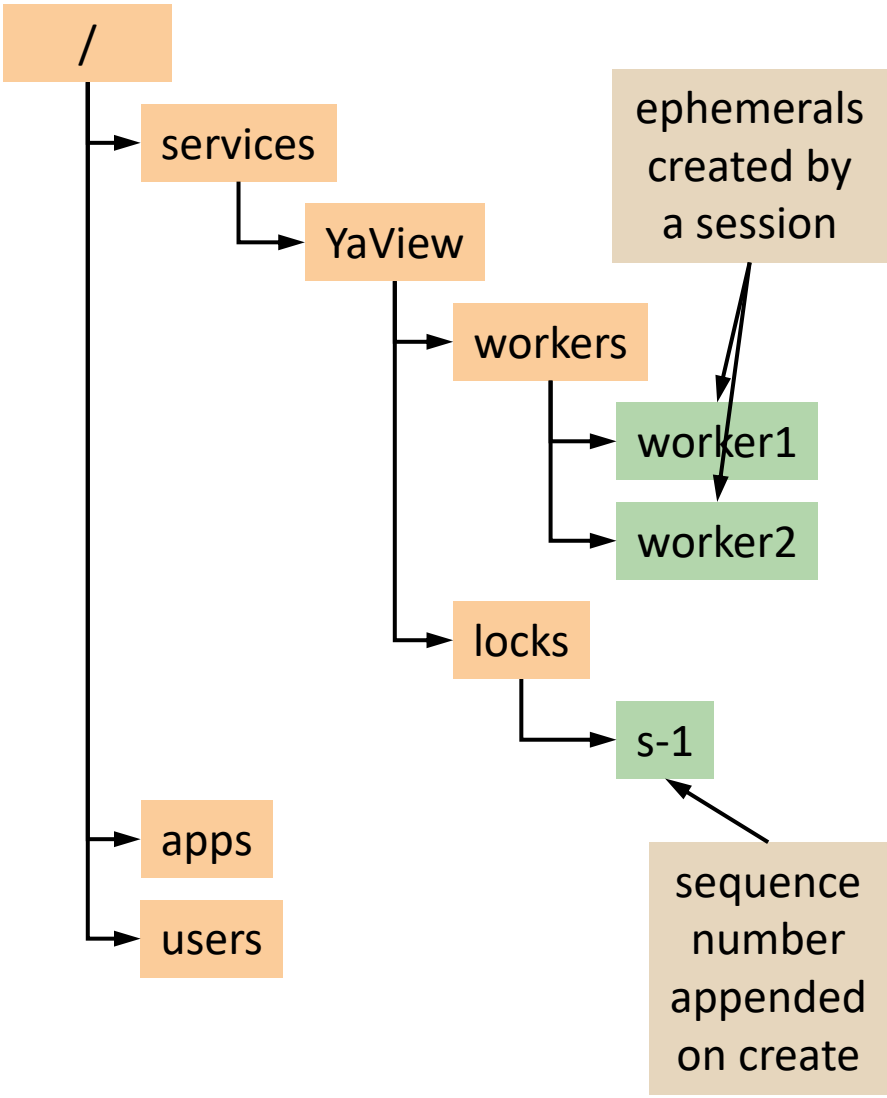
Data model

- Each node is called znode
  - Stores some data, including version
  - Data is read and written in its entirety
- znodes may have children
  - Hierarchal namespace
  - Like a file system, registry
- State maintained in memory



Znode types

- Two special types of znodes:
  - Ephemeral: znode deleted when explicitly deleted, or when client session that created the znode fails
  - Sequence: appends a (unique) monotonically increasing counter



Overview of API

- Operations look like file system operations
  - Take a path name to a znode, e.g., create("/app1/worker1", …)
- Operations are non-blocking (or wait-free)
  - Operations by one client do not block on another client
  - Slow and failed nodes cannot slow down fast ones
  - No deadlocks
- ZooKeeper uses API to provide "coordination recipes"
  - E.g., group membership recipe, locking recipe
- Some recipes need to wait on conditions, e.g., locking
  - ZooKeeper supports waiting for conditions efficiently

ZooKeeper API

- Clients open a session with (any) one ZK server, issue operations synchronously or asynchronously

```
s= openSession()
```

```
String create(path, data, acl, flags)
void delete(path, expectedVersion)
Stat setData(path, data, expectedVersion)
```

```
(data, Stat) getData(path, watch)
Stat exists(path, watch)
String[] getChildren(path, watch)
```

```
void sync(s)
```

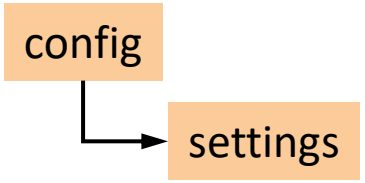
Key API Properties

- Async operations allow batching many operations
- Exclusive file creation: one concurrent create succeeds
- (d1, v) = getData()/setData(d2, v) support atomic ops
  - setData fails if data is modified since getData
- Sequence files allow ordering operations across clients
  - E.g., ordering lock operations
- Ephemeral files help cope with client session failure
  - E.g., group membership change, release locks, etc.
- Watches avoid costly repeated polling

Coordination Recipes

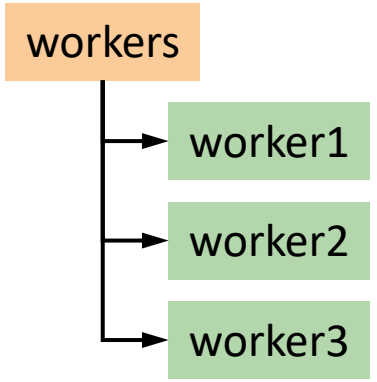
Configuration

- Workers read configuration
  - `getData("../config/settings", true)`
- Administrators change the configuration
  - `setData("../config/settings", newConfig, -1)`
- Workers notified of change, re-read new configuration
  - `getData("../config/settings", true)`

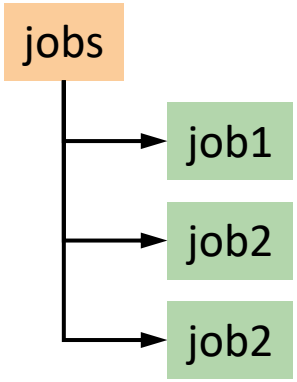


Group membership

- Register worker with host information in group
  - `create(".../workers/workerNR", hostInfo, EPHEMERAL)`
- List group members
  - `getChildren(".../workers", true)`



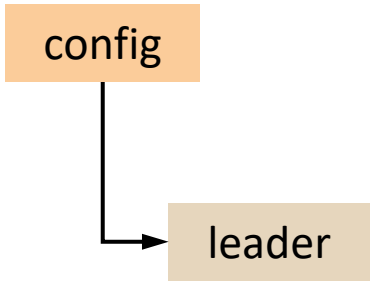
- `Job scheduling`



Leader election

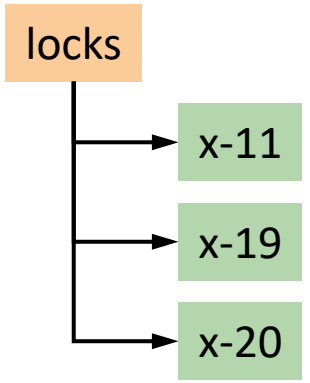
```
while true:
  if exists(“.../config/leader”, watch=true)
    follow the leader
    return

  if create(“.../config/leader”, hostname, EPHEMERAL)
    become leader
    return
```

```
If watch is triggered for “.../config/leader”
  restart leader election process
```



Locks

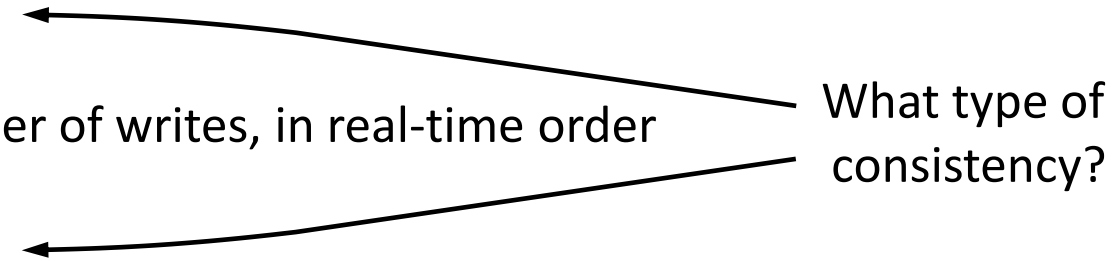
```
lock:
  id = create("../locks/x-", SEQUENCE|EPHEMERAL)

  restart:
  getChildren("../locks", false)
  if id is the 1st child // lock is acquired
    exit

  // wait for previous node
  if exists(name of last child before id, true)
    wait for event // no herd

  goto restart // why?


unlock:
  delete(id)
```



Consistency Guarantees

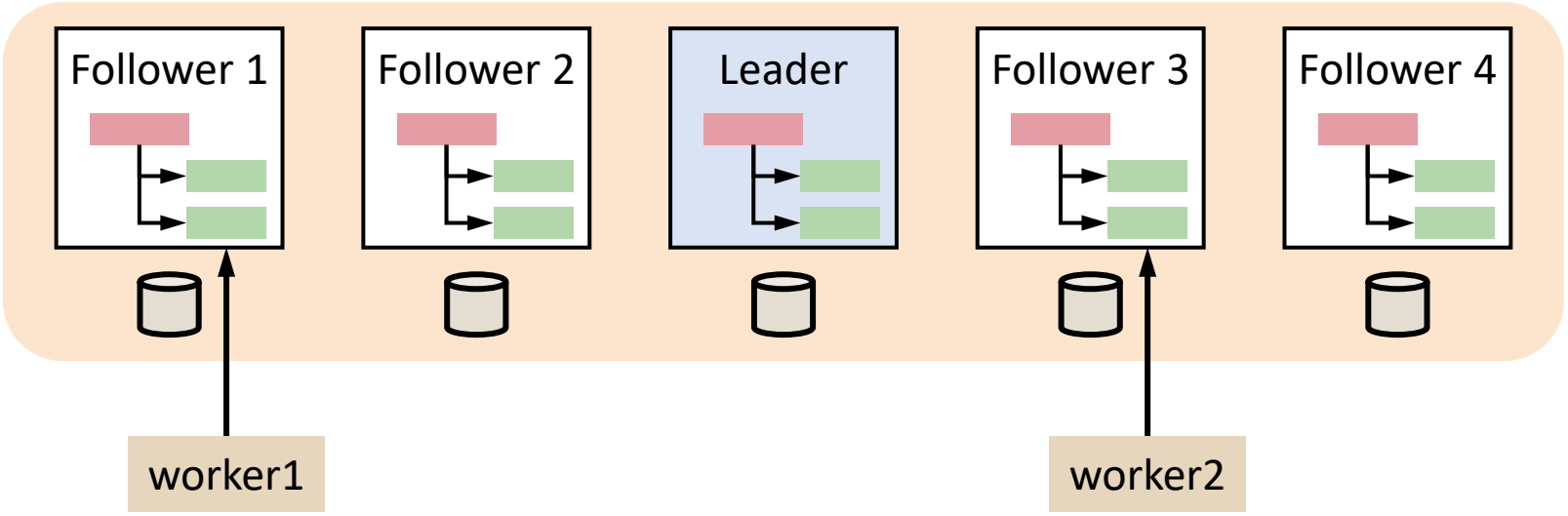
ZooKeeper consistency guarantees

- Linearizable writes
  - Clients see same order of writes, in real-time order
- FIFO client order
  - A client's operations are executed in order



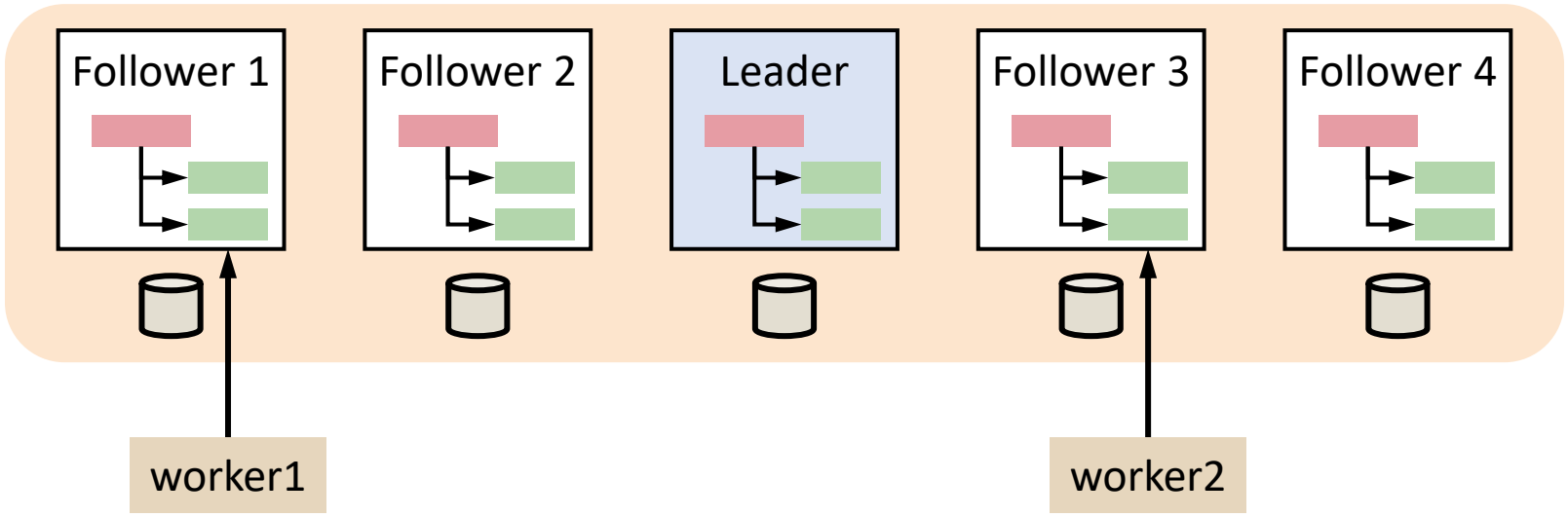
- Implications:
  - A client's read must wait for all its previous writes to be executed
  - A client's writes are applied in order
  - A client X reads another client Y's writes in order
- Hypothesis: wait-free synchronization + linearizable writes + FIFO client order enable high-performance coordination for read-heavy workloads

Providing good read performance



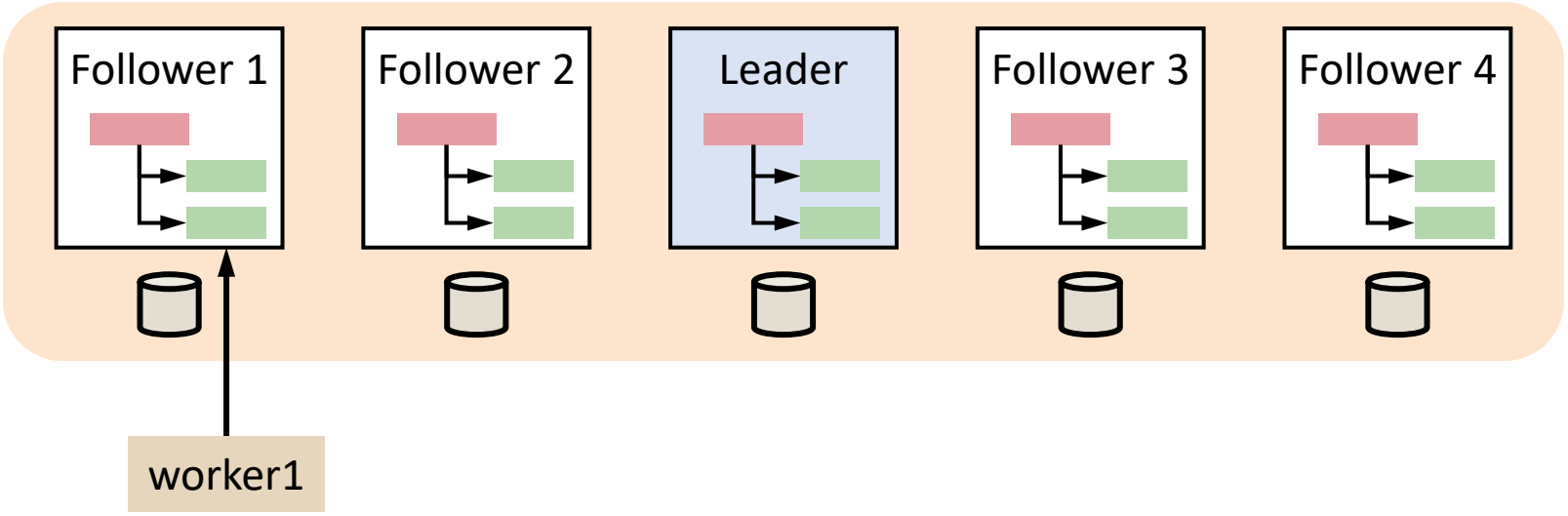
- Clients connect to different servers for parallelism
- Writes forwarded by followers to leader
  - Leader replicates writes using atomic total order broadcast
- Reads executed on replicas locally
  - Efficient but may return stale data when follower is lagging

Providing good read performance



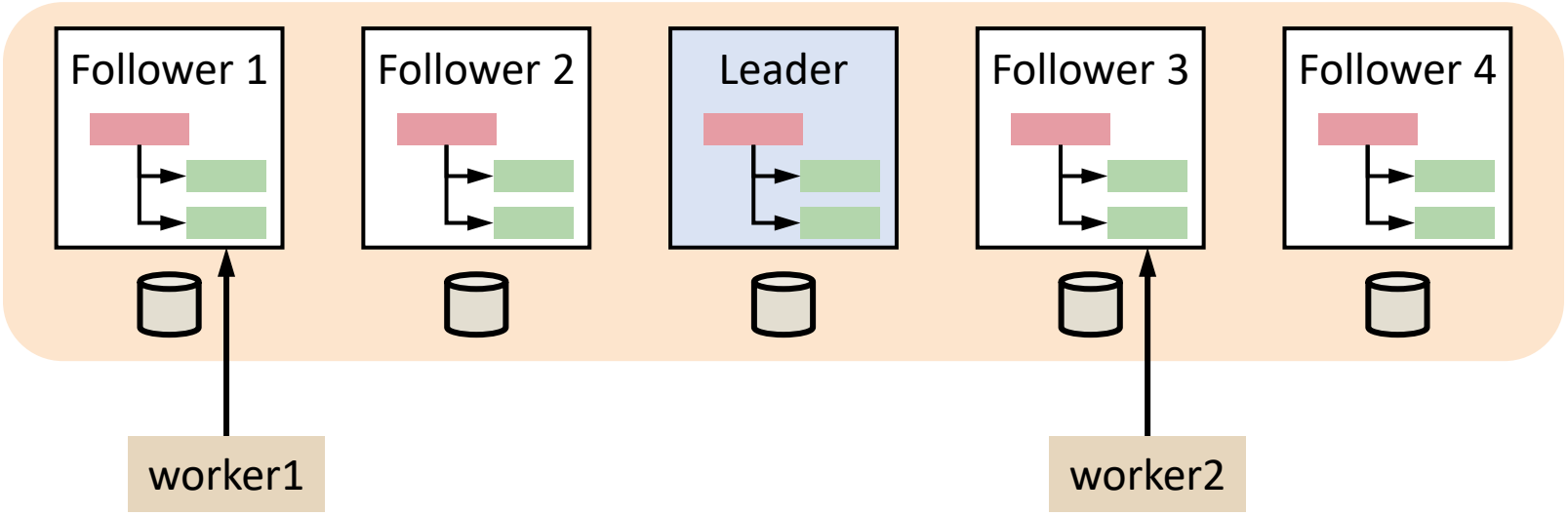
- Clients watch, rather than poll, for changes
  - Watches implemented efficiently on replicas locally
- Clients can issue many concurrent async operations
  - Clients number messages, ZK executes them in FIFO order
  - Completion notifications delivered asynchronously, unlike RPC
  - ZK can batch state updates, fewer messages & disk writes

Understanding FIFO client order



- Say worker1 issues W1, R2, W3 at Follower 1
  - Follower 1 forwards writes to leader
  - Leader delivers writes to Follower 1 in some order: W W W1 …
    - W are from other clients
- For FIFO client order, Follower 1 performs R2 after seeing W1, but before seeing W3

Understanding FIFO client order



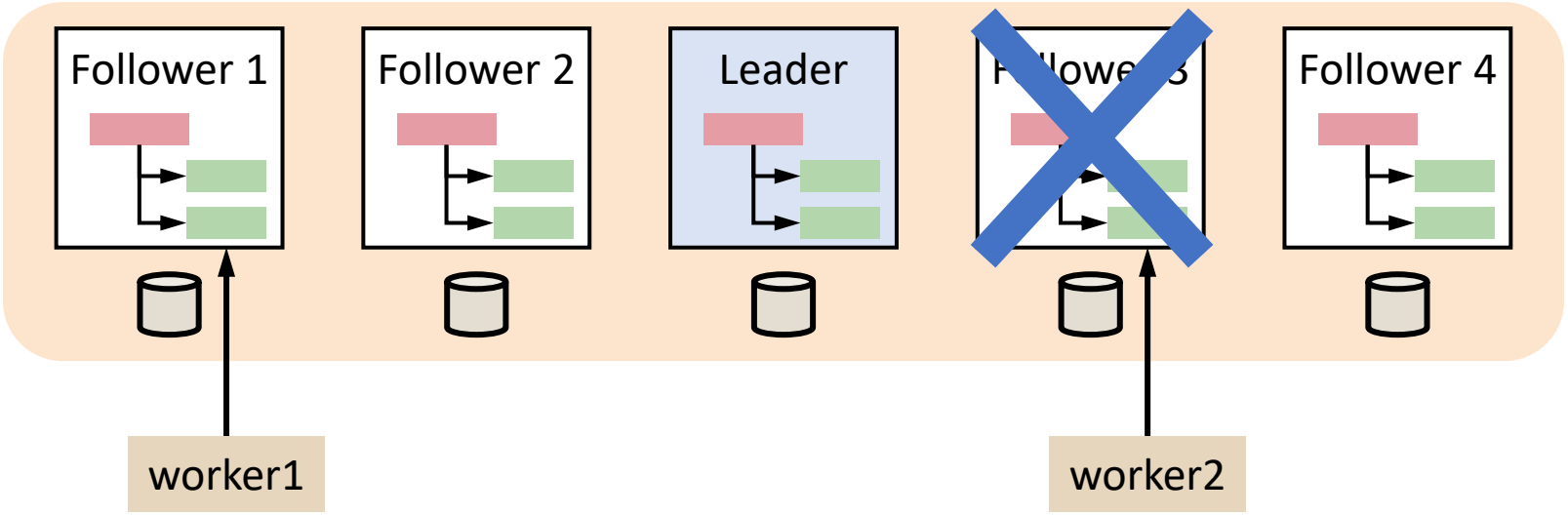
worker1 at Follower 1

```
setData("file", newdata, …)
create("ready", …)
```

worker2 at Follower 3

```
if (exists("ready", …))
  // guaranteed to read newdata
  getData("file", …)
```

Understanding FIFO client order



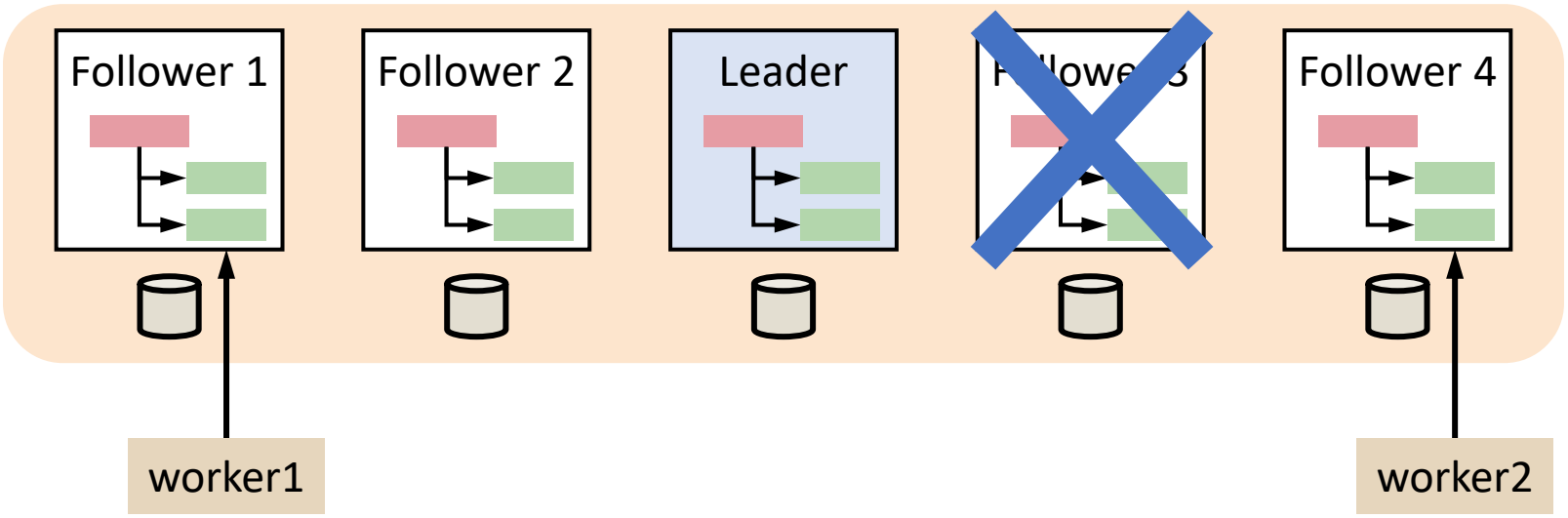
worker1 at Follower 1

```
setData("file", newdata, …)
create("ready", …)
```

worker2 at Follower 3

```
if (exists("ready", …))
  // guaranteed to read newdata
  // Follower 3 crashes
  getData("file", …)
```

Understanding FIFO client order



worker1 at Follower 1

```
setData("file", newdata, …)
create("ready", …)
```

worker2 at Follower 3

```
if (exists("ready", …))
  // guaranteed to read newdata
  // Follower 3 crashes
  // worker 2 connects to Follower 4
  // still guaranteed to read newdata
  getData("file", …)
```

FIFO client order across replicas

- ZK leader tags each change in its state with a unique id (zxid) in increasing (total) order
  - ZK replicas maintains last_zxid they have seen
- Client's read request at replica returns replica's last_zxid
  - Clients also maintain last_zxid they have seen
- When a client connects to another replica, replica delays responding to the client until replica's last_zxid >= client's last_zxid
- Ensures that replica's view is newer than client's view
- In previous example, worker2's connection to Follower 4 is delayed until Follower 4 has seen "ready" file

Handling Failures

Challenges

- How does ZooKeeper detect coordinator failure?
- What if coordinator fails midway during complex update?
- After a coordinator is elected, what if old coordinator is alive, thinks it is still the coordinator?

Coordinator failure

- A client establishes a session with a ZK server
  - Each session sends keep-alive messages to ZK server
- ZK leader decides a session has failed when no ZK server receives keep-alive message from session within a timeout
- Then, ZK leader
  - Terminates session
  - Deletes ephemeral files created by session
- When coordinator's session is terminated, clients use leader election to elect another coordinator

Failure while updating state

- A coordinator could fail while it is updating ZK state
  - Clients should not see coordinator's partial updates
- Option 1: store all coordinator data in one file
  - setData() calls are performed failure atomically
- Option 2: use a ready file scheme
  - Why does this scheme work?

Coordinator

```
delete("…/ready", …);
setData("…/config1", …);
setData("…/config2", …);
create("…/ready", …);
```

Worker

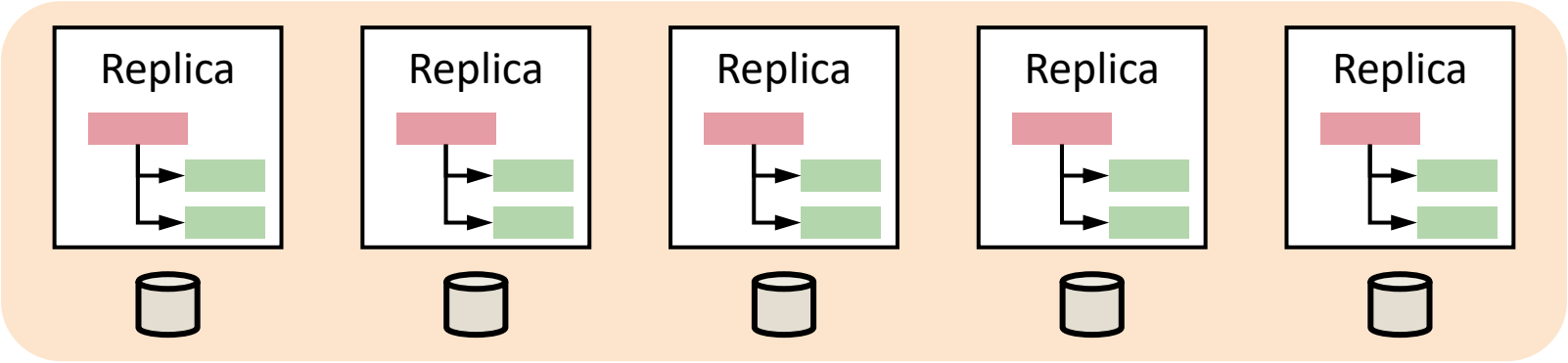
```
if (exists("…/ready", watch=true))
  getData("…/config1")
  getData("…/config2")
```

Old coordinator

- What if old coordinator is alive, thinks it is coordinator?
  - When ZooKeeper leader terminates coordinator's session, it also stops accepting requests from the coordinator's session
  - Old coordinator can no longer modify ZooKeeper state!
- What if old coordinator talks directly to a worker?
  - Worker needs to ignore requests from old coordinator
  - New coordinator reads zxid of newly created "leader" znode, sends zxid on each message to workers
  - If worker receives message with zxid < previously seen zxid, it ignores (old coordinator's) message
  - This method of ignoring requests from an old coordinator is called fencing

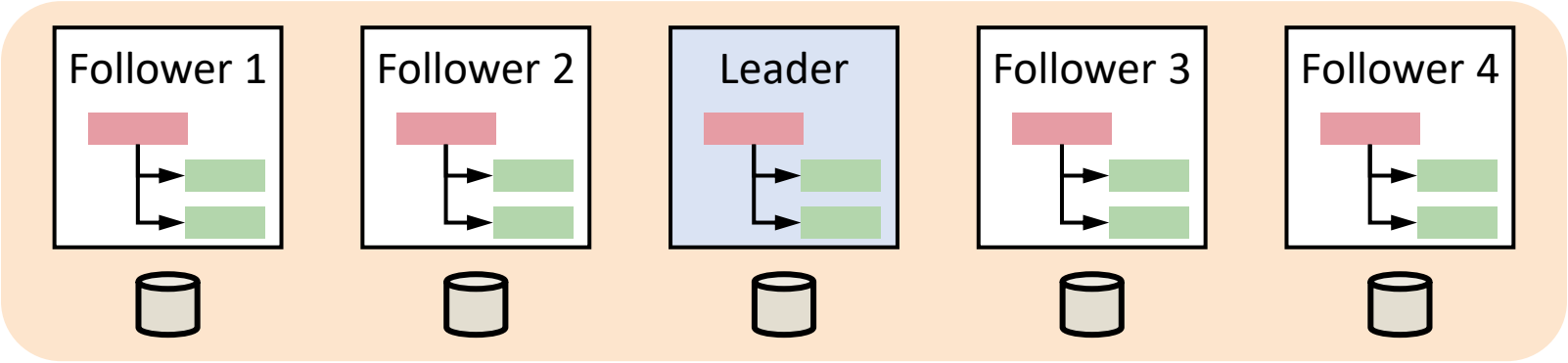
Implementation

ZooKeeper service



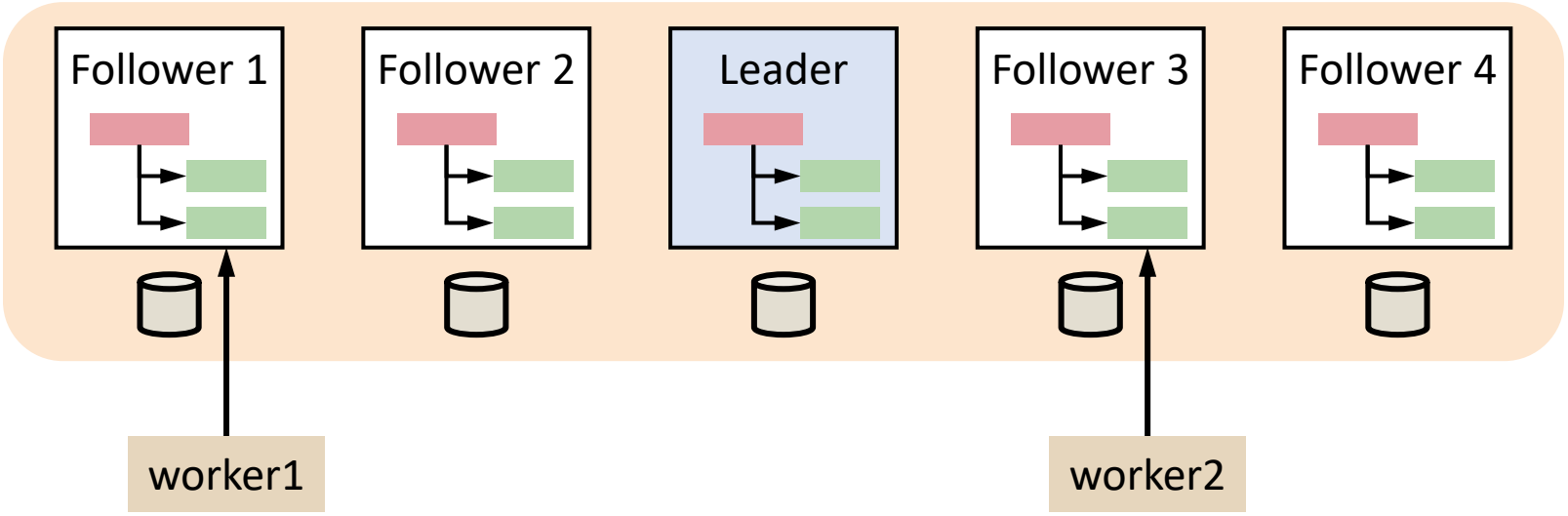
- ZooKeeper maintains a replicated database
- Each replica keeps a copy of ZooKeeper state in memory
  - Logs writes to ZooKeeper state in a write-ahead log on disk for recovering committed operations
  - Creates and stores snapshots of ZooKeeper state on disk for faster recovery

ZooKeeper leader



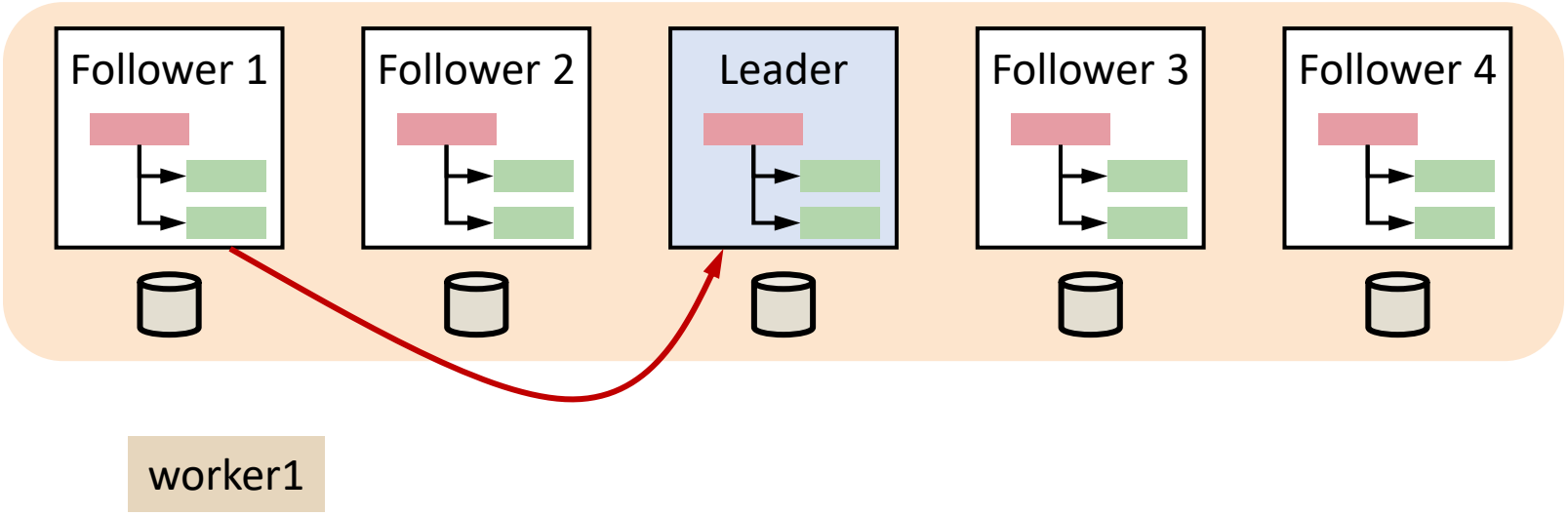
- Servers elect a leader at startup
- If a leader fails, they re-elect another leader using the ZAB leader-based atomic broadcast protocol
  - This election is different from clients electing a coordinator

ZooKeeper reads



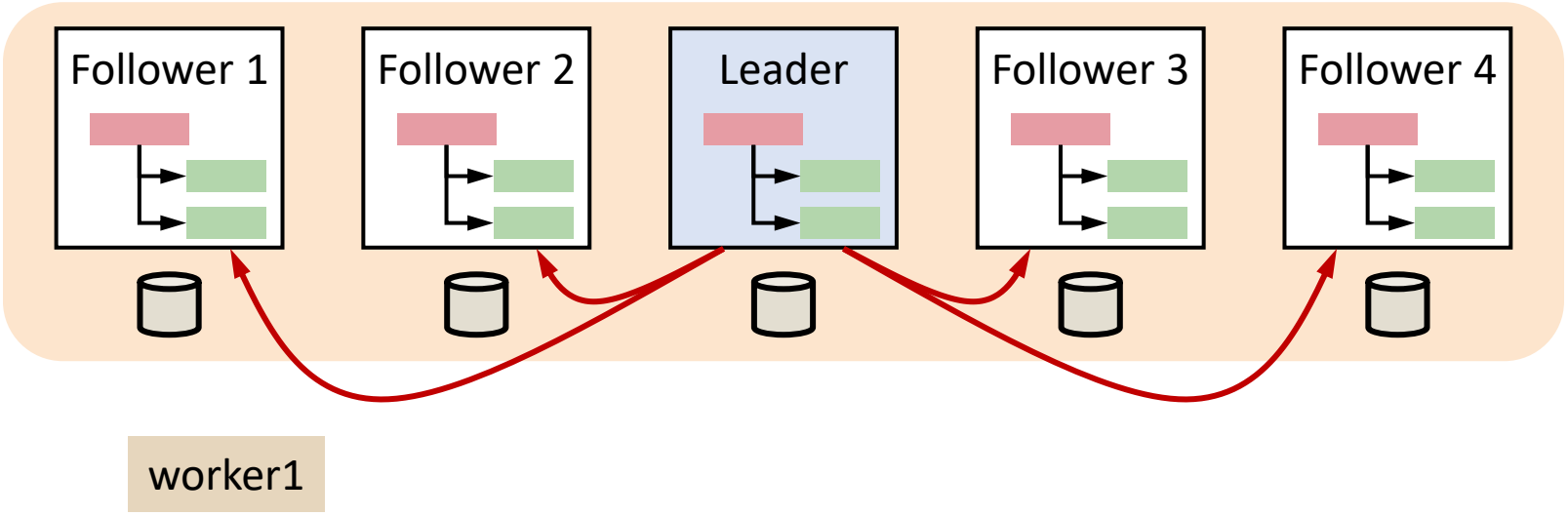
- Clients connect to any one server (follower or leader)
- Client's read (e.g., getData) performed by local server
  - E.g., When worker2 issues read, Follower 3 reads and returns data from its own copy
  - Reads may return stale results

ZooKeeper writes



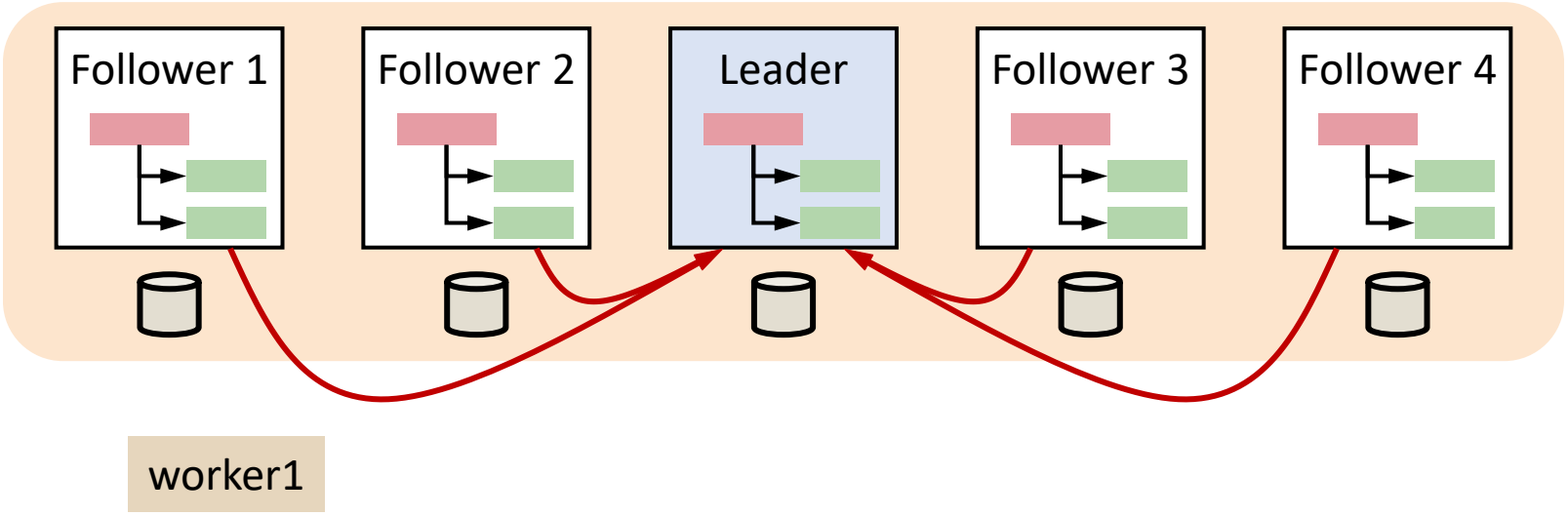
- worker1's write (e.g., setData) forwarded by local server (Follower 1) to leader

ZooKeeper: send writes



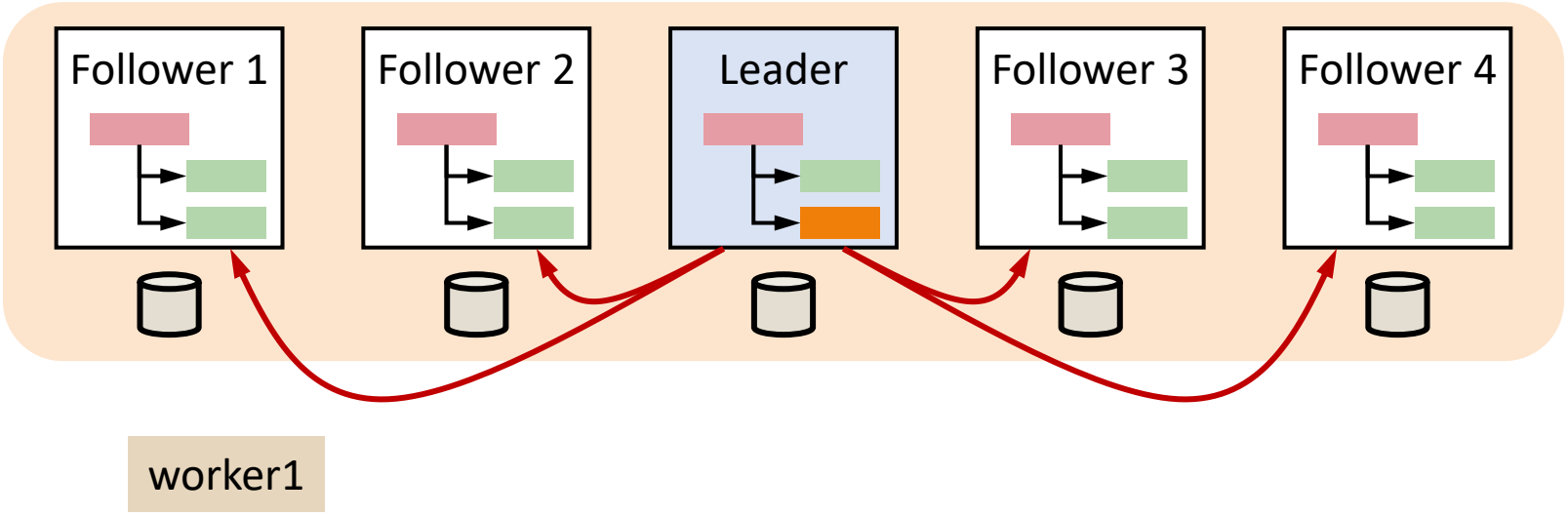
- Leader logs the write to its write-ahead log
- Leader sends write to all followers

ZooKeeper: receive acks



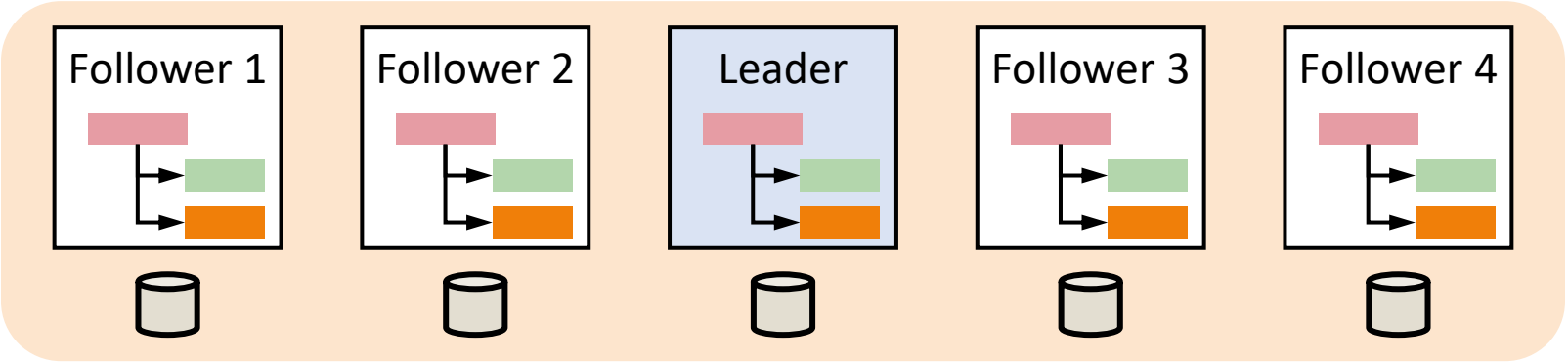
- Followers log the write to their write-ahead log
- Respond to the leader

ZooKeeper: commit write



- When leader receives acks from a majority of servers, it commits the write (need 2f+1 servers to handle f failures)
- Leader applies write to ZooKeeper state in memory
- Leader informs followers that write is committed

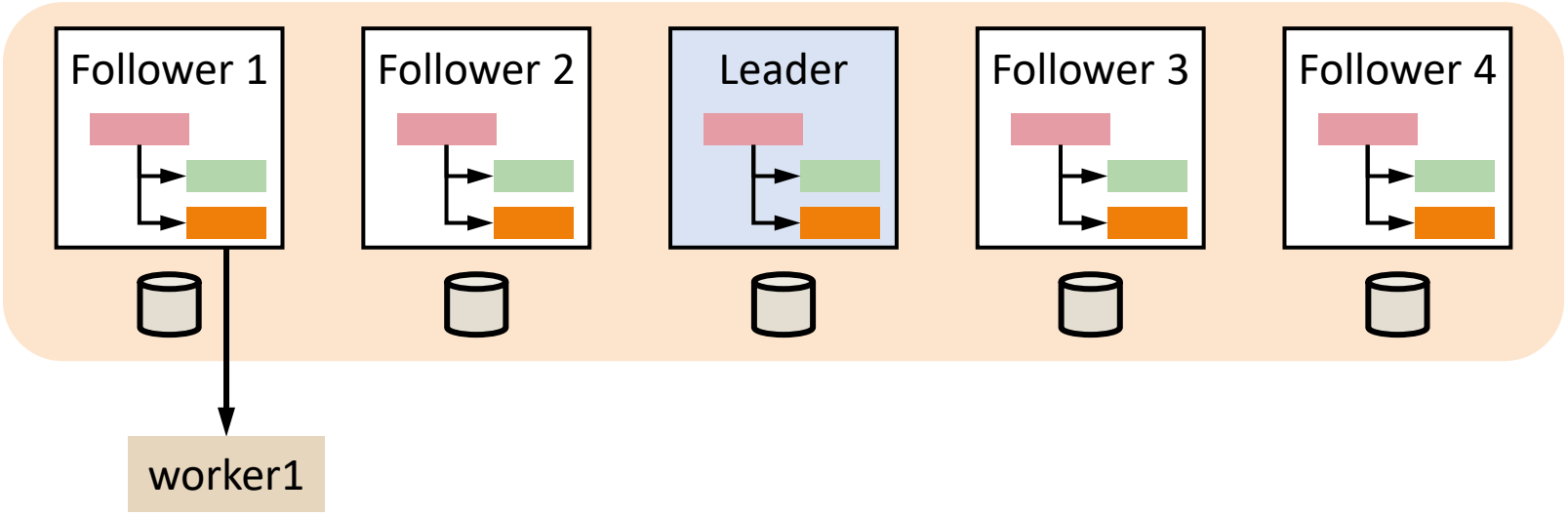
ZooKeeper: apply write



worker1

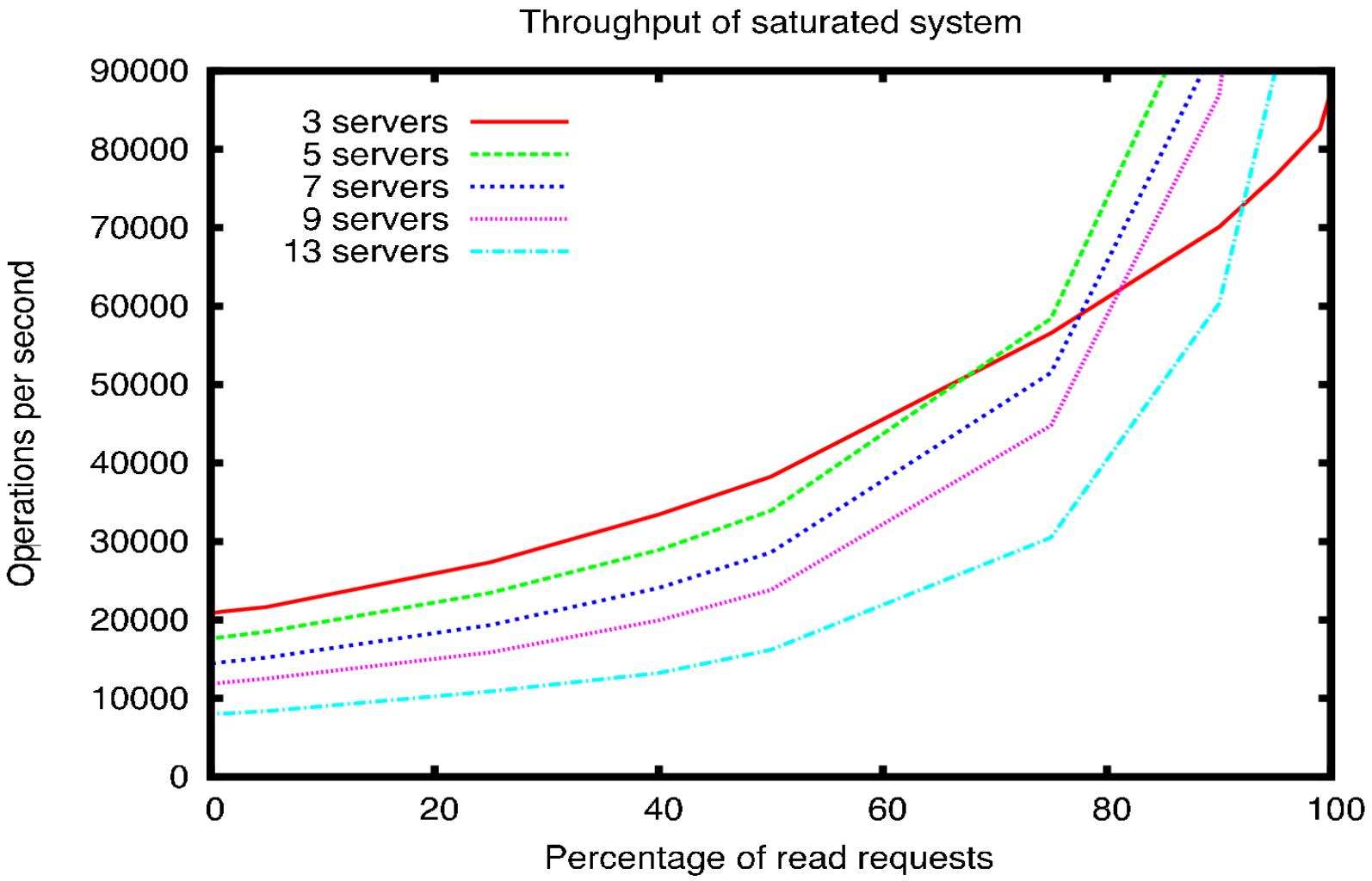
- Each follower:
  - Commits the write
  - Applies write to ZooKeeper state in memory
  - Issues watch notifications to clients connected to follower

ZooKeeper: write response

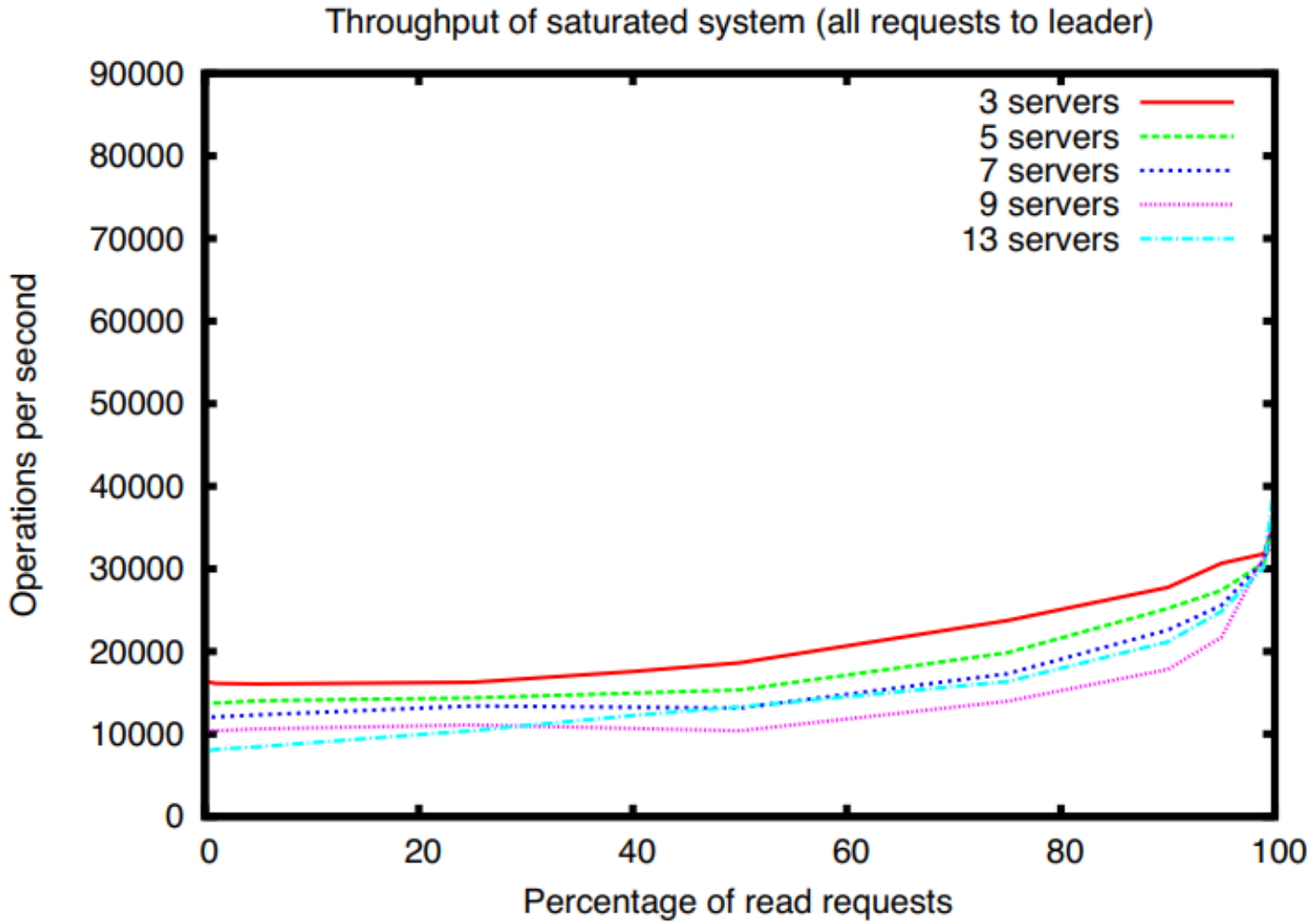


- Follower 1 delivers write response to worker1

ZooKeeper performance



Performance: clients connect to leader



Conclusions

- ZooKeeper is a coordination service with a wait-free API and a change notification service
  - Applications implement coordination recipes with this API
- ZooKeeper ensures linearizable writes and FIFO execution of client operations
  - These guarantees are sufficient for many applications
  - Enable good performance for read-heavy workloads
- Released as an Apache open-source project
  - Relatively easy to use
  - Today, used extensively for coordination functions