

ZooKeeper: Wait-Free Coordination for Internet-Scale systems

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

ECE1724

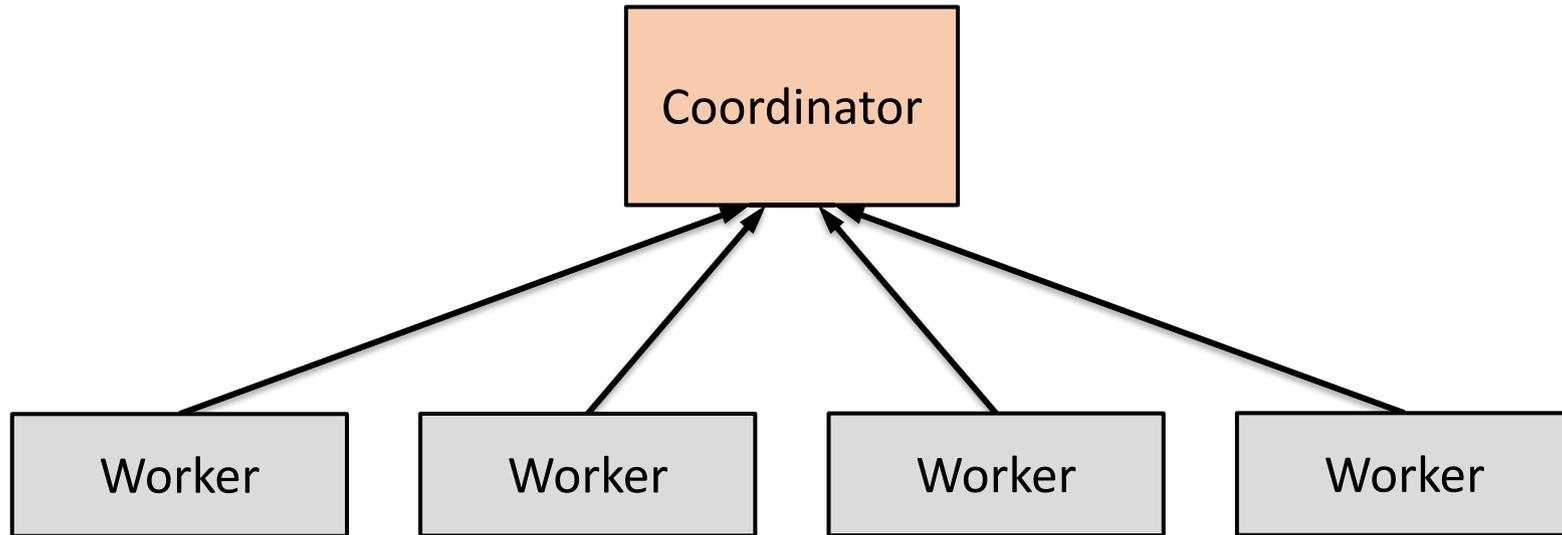
These slides are slightly modified versions from the original slides by:

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

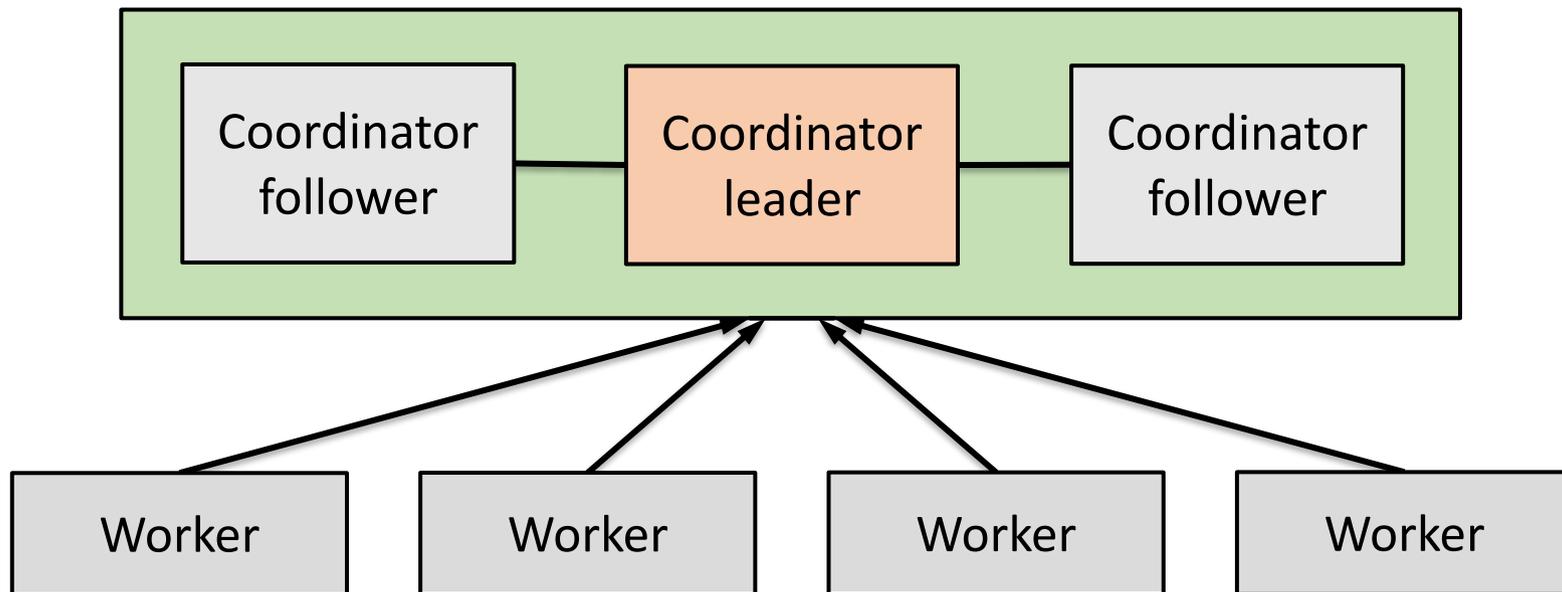
Why Coordination?

- Group membership
- Leader election
- Dynamic configuration
- Status monitoring
- Queuing
- Barriers
- Critical sections

Classic coordination

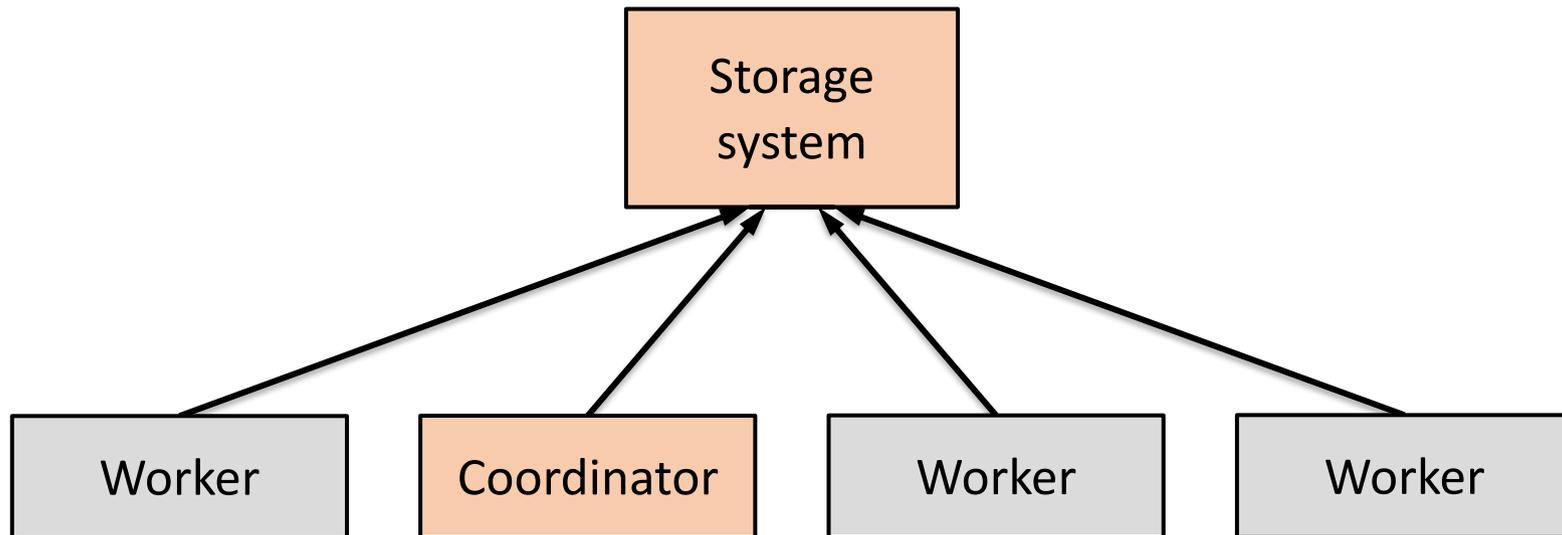


Fault-tolerant coordination



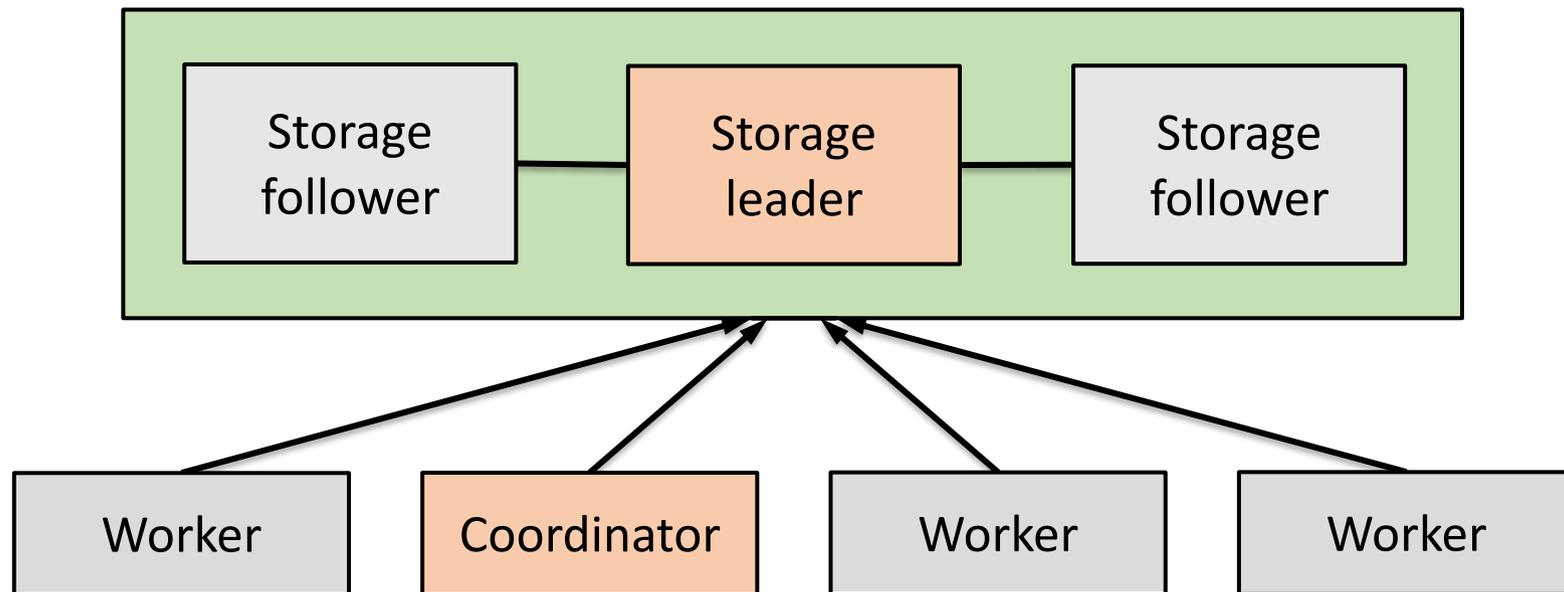
- Use state machine replication for fault tolerance
- Issues
 - Programming coordinator state machine is complicated
 - Coordinator can become bottleneck

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 accesses to storage
 - Any worker can be coordinator

Fault-tolerant storage system



- Replicate storage for fault tolerance
 - Coordinator code is simpler since no state machine needed
- What happens when coordinator fails?
 - Any another worker can take over

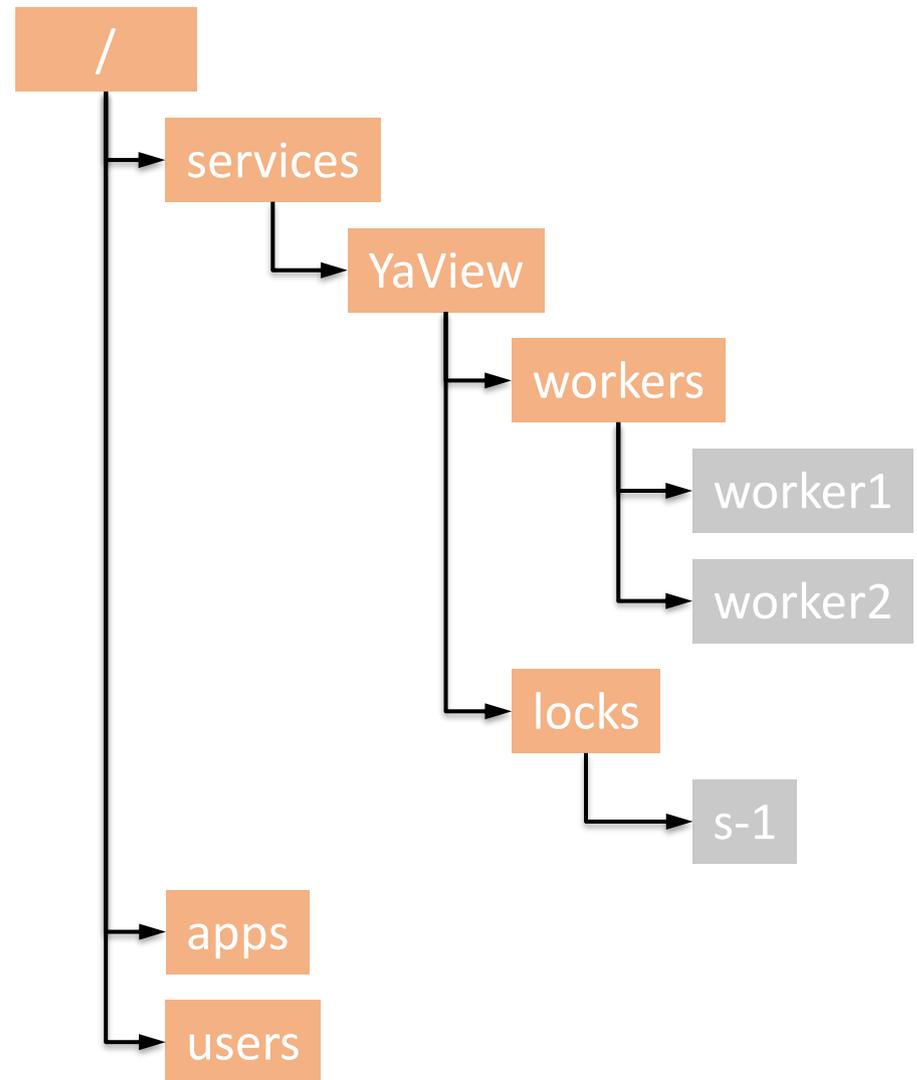
ZooKeeper

- 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

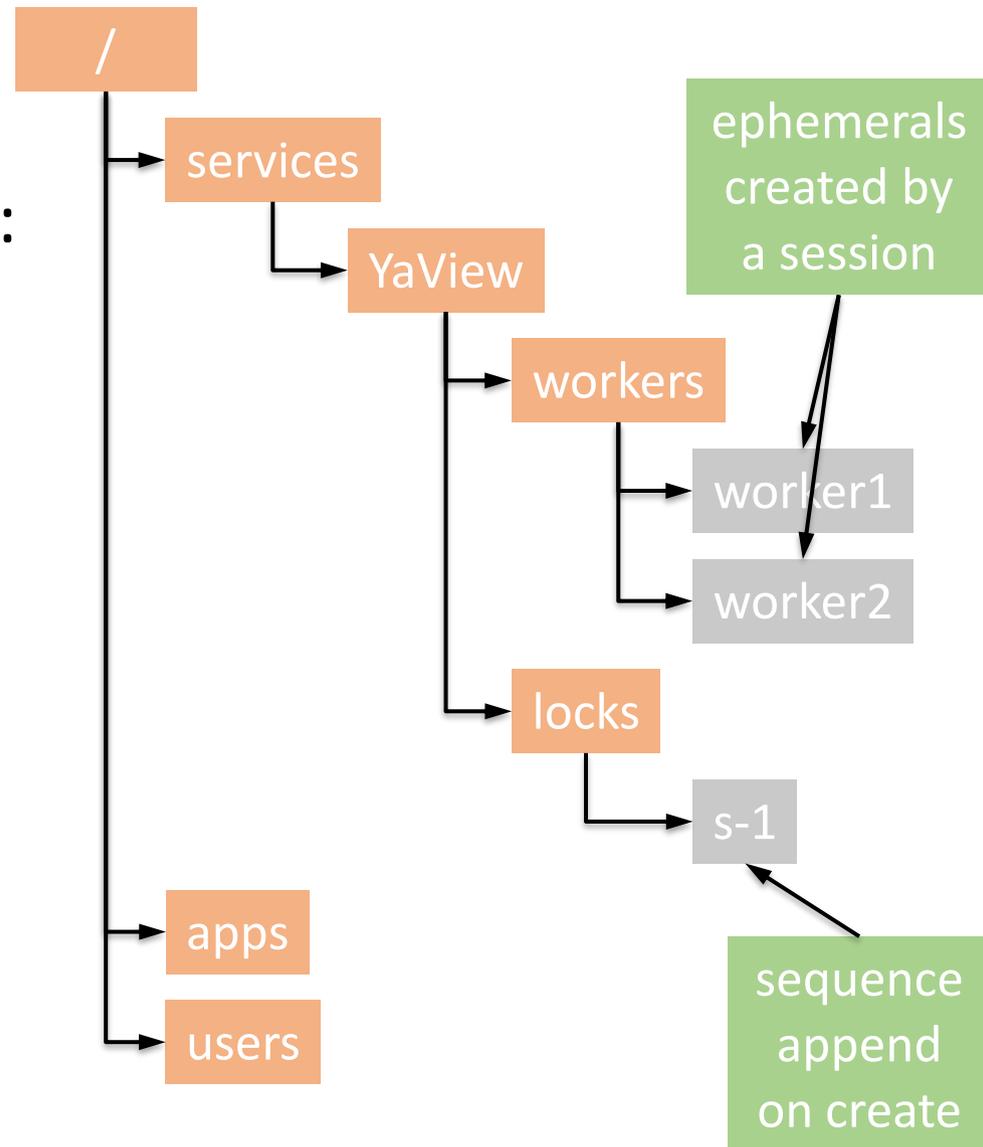
Data model

- 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 necessarily 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)
```

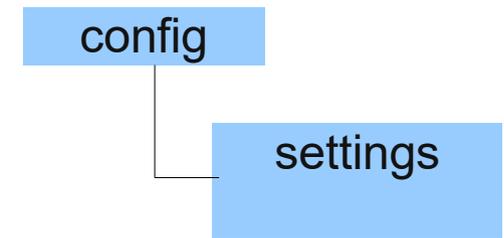
Key API Properties

- Asynchronous operations allow batching operations
- Exclusive file creation (one concurrent create succeeds)
- $(d0, v) = \text{getData}()$, $\text{setData}(d1, v)$ 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 (i.e., sessions) help with client failure
 - E.g., group membership change, release locks, etc.
- Watches avoid costly repeated polling

Coordination Recipes

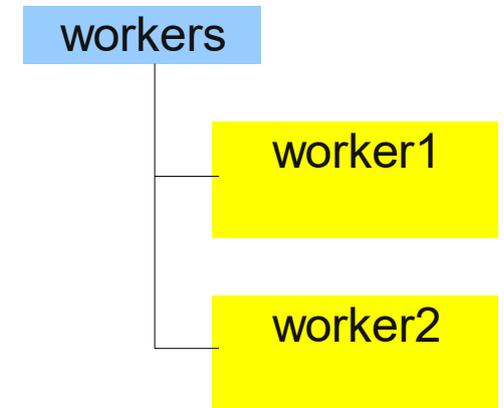
Configuration

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



Group Membership

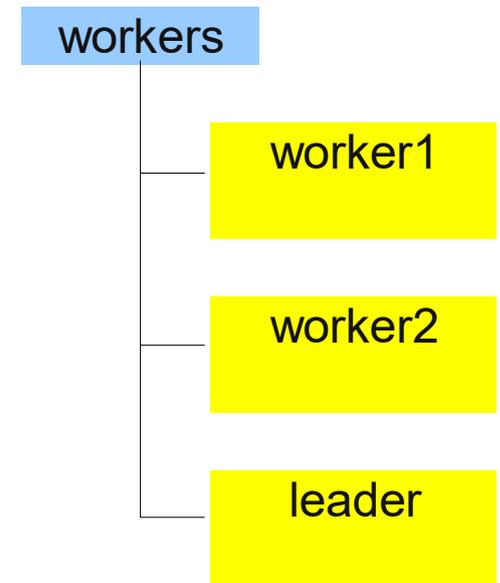
- Register worker with host information in group
 - `create(.../workers/worker1", hostInfo, EPHEMERAL)`
- List group members
 - `listChildren(.../workers", true)`



Leader Election

```
while true:  
    if exists("../workers/leader", watch=true)  
        follow the leader  
        return  
  
    if create("../workers/leader", hostname, EPHEMERAL)  
        become leader  
        return
```

If watch is triggered for “../workers/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
```

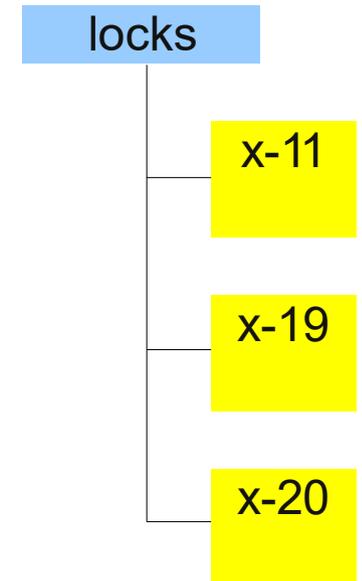
```
if exists(name of last child before id, true)
```

```
  wait for event // no herd
```

```
goto restart // why?
```

unlock:

```
delete(id)
```

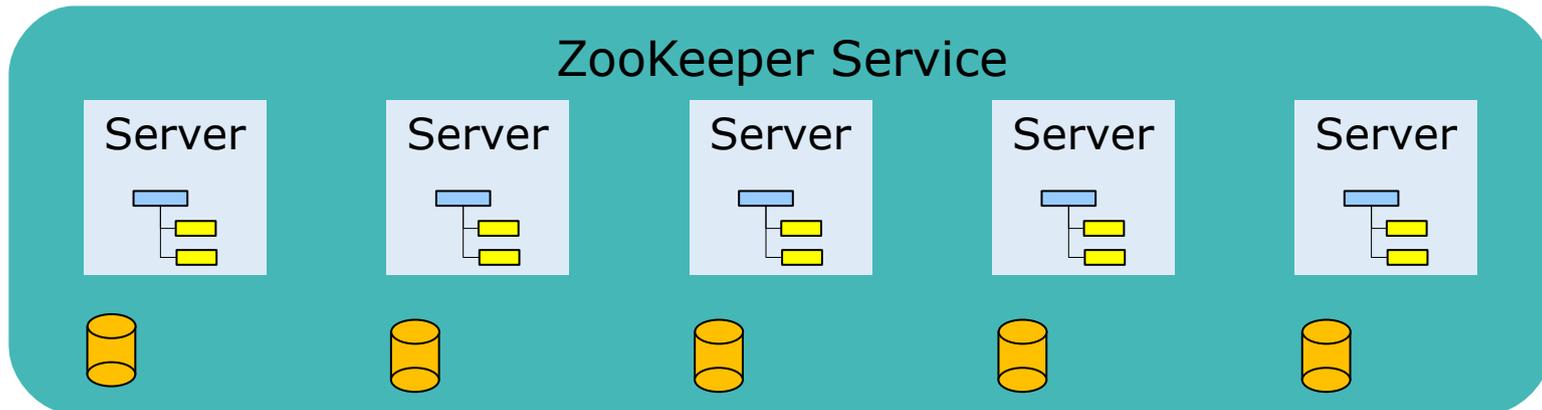


Implementation

ZooKeeper Guarantees

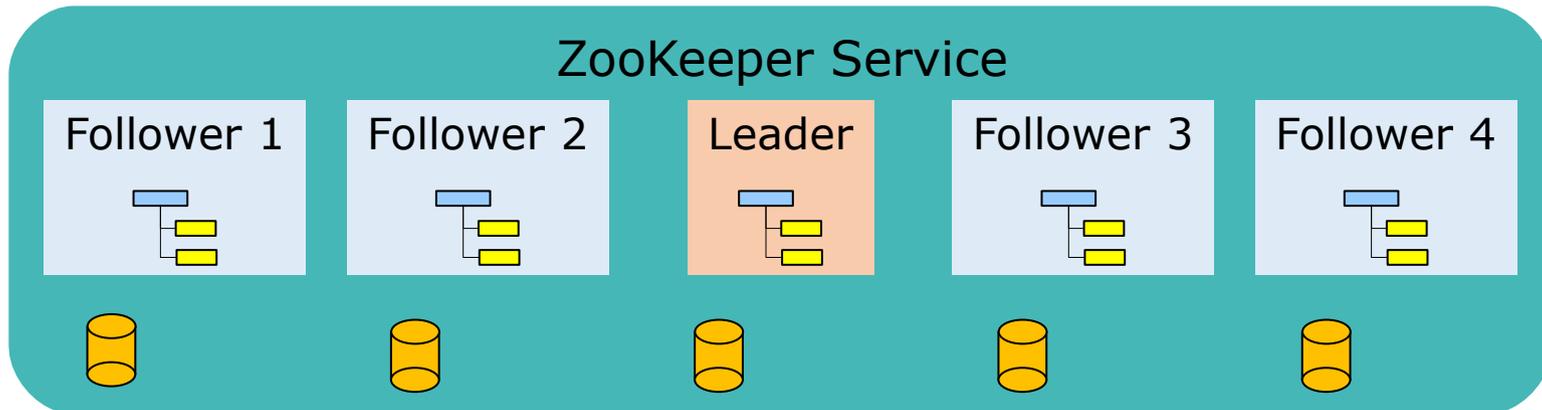
- **Linearizable writes**
 - Clients see same order of writes, in real-time order
- **FIFO client order**
 - A client's operations are executed in order
 - Implications:
 - Client A watching for Client B's changes sees them in order
 - A client's read must wait for all its previous writes to be executed
 - Reads may return **stale** values (see a prefix of writes)
- Hypothesis: **wait-free ops + linearizable writes + FIFO execution + watches** are sufficient for implementing efficient coordination services for read-heavy workloads 20

ZooKeeper Service



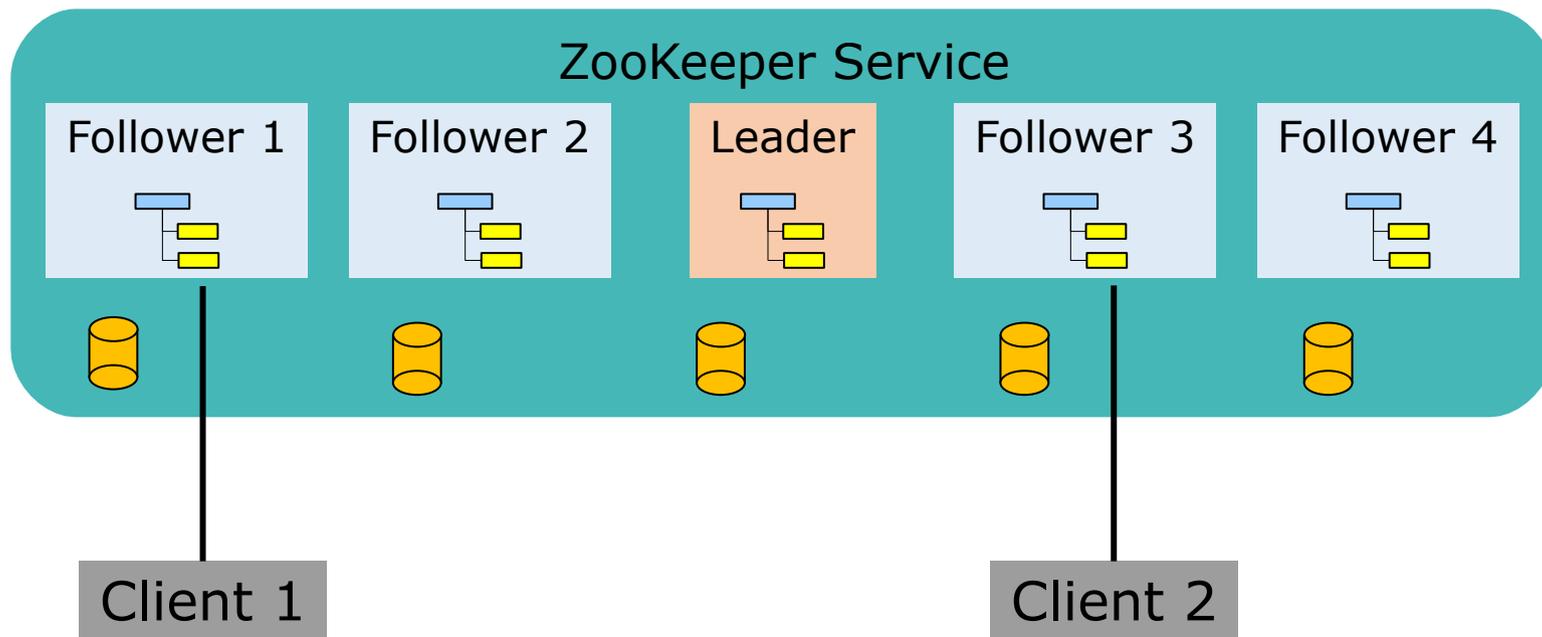
- ZooKeeper maintains a replicated database
- Each server
 - Keeps a copy of the ZooKeeper state in memory
 - Logs writes to ZooKeeper state in a write-ahead log on disk for recovering committed operations
 - Creates and stores snapshots of ZooKeeper state on disk for faster recovery

ZooKeeper Leader



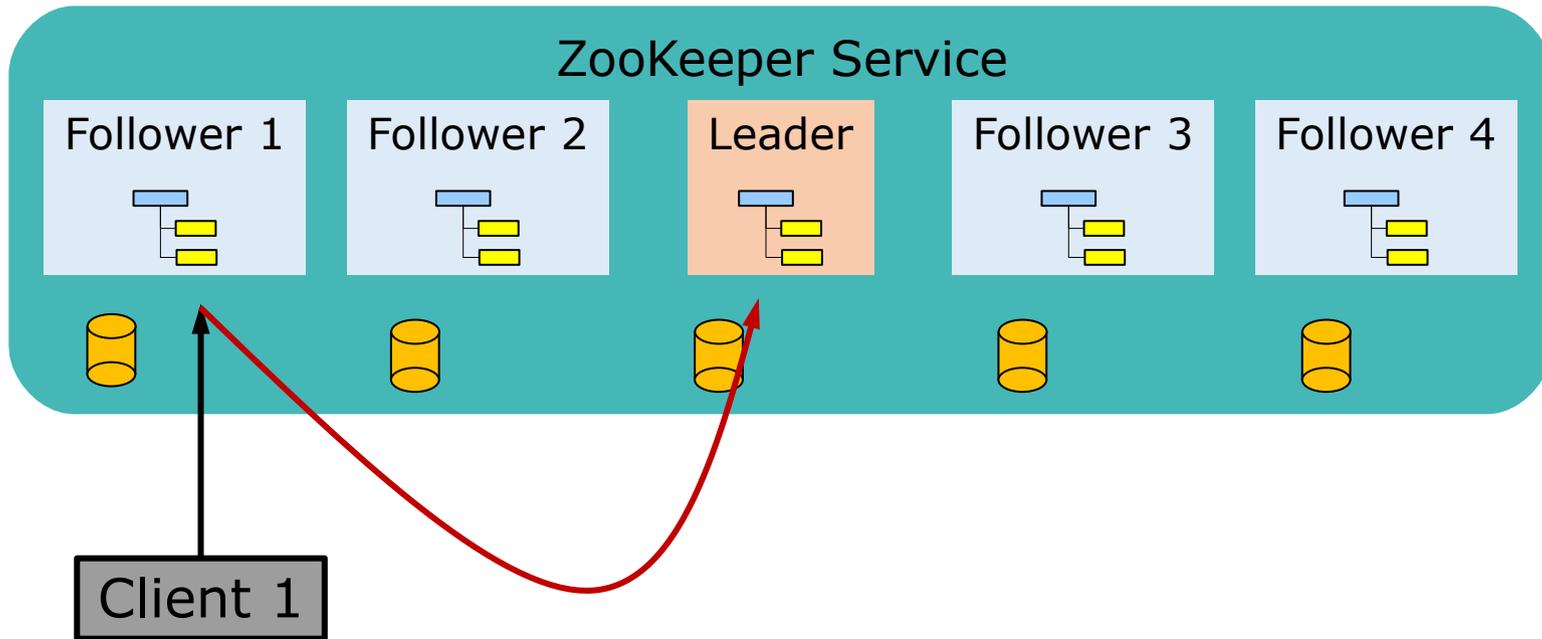
- Servers elect a leader at startup
- If a leader fails, they re-elect another leader using the ZAB leader-based atomic broadcast protocol

ZooKeeper Reads



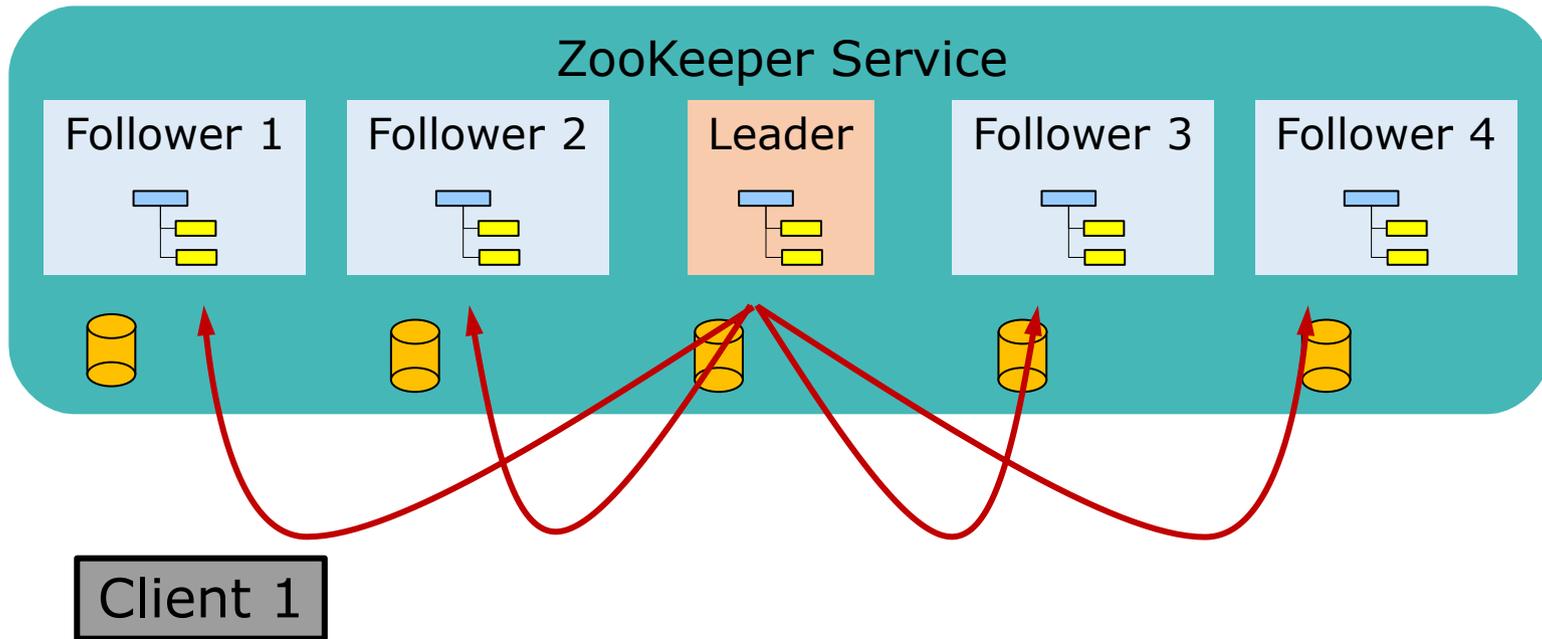
- Clients connect to any one server (follower or leader)
- Client's read (e.g., `getData`) performed by local server
 - E.g., When Client 2 issues read, Follower 3 reads and returns data from its own copy

ZooKeeper Writes



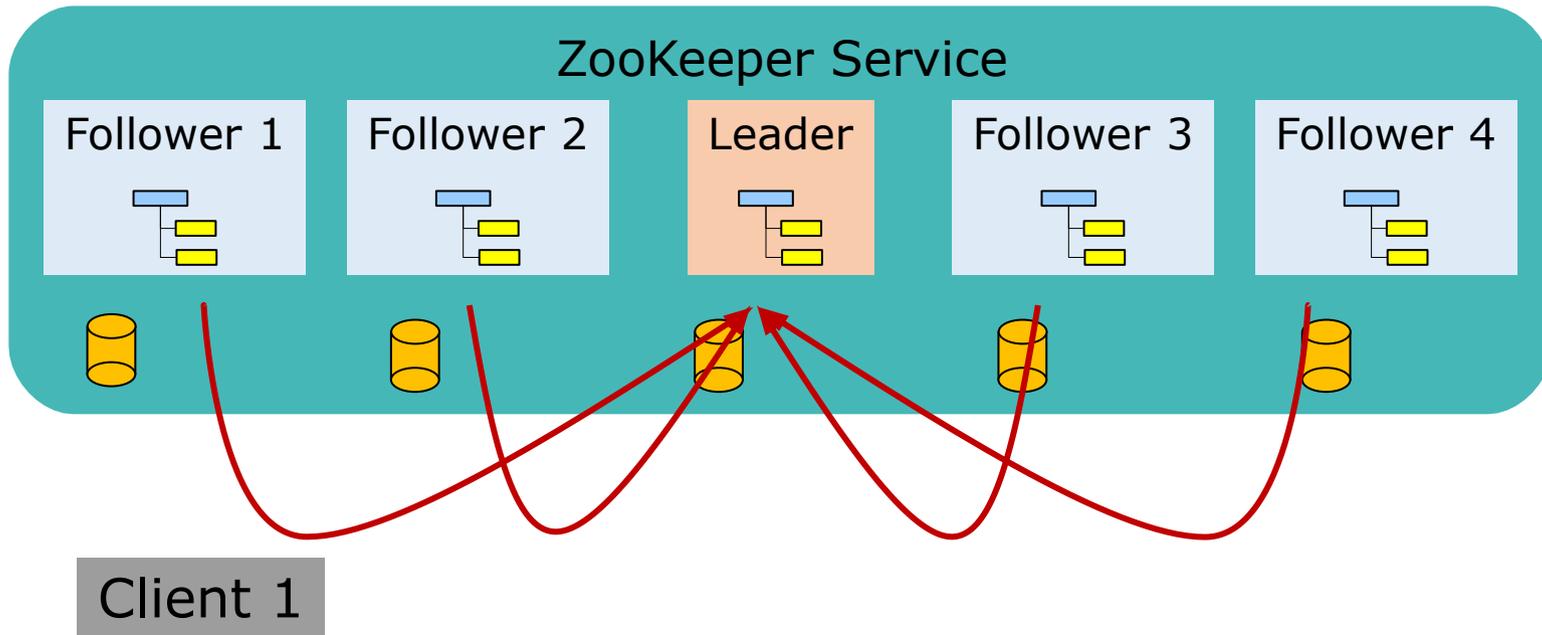
- Client 1's write (e.g., setData) forwarded by local server (Follower 1) to leader

ZooKeeper: Send Write



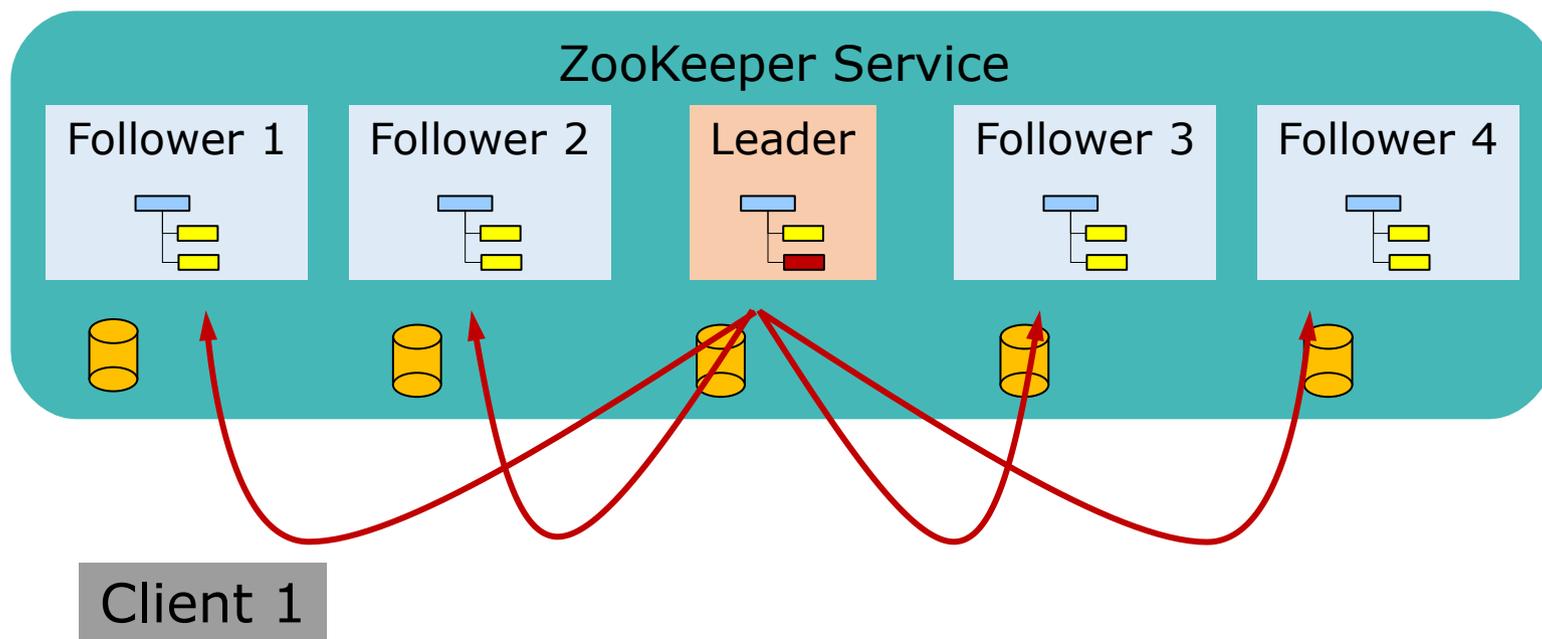
- Leader logs the write to its write-ahead log
- Leader sends write to all followers

ZooKeeper: Receive Acks



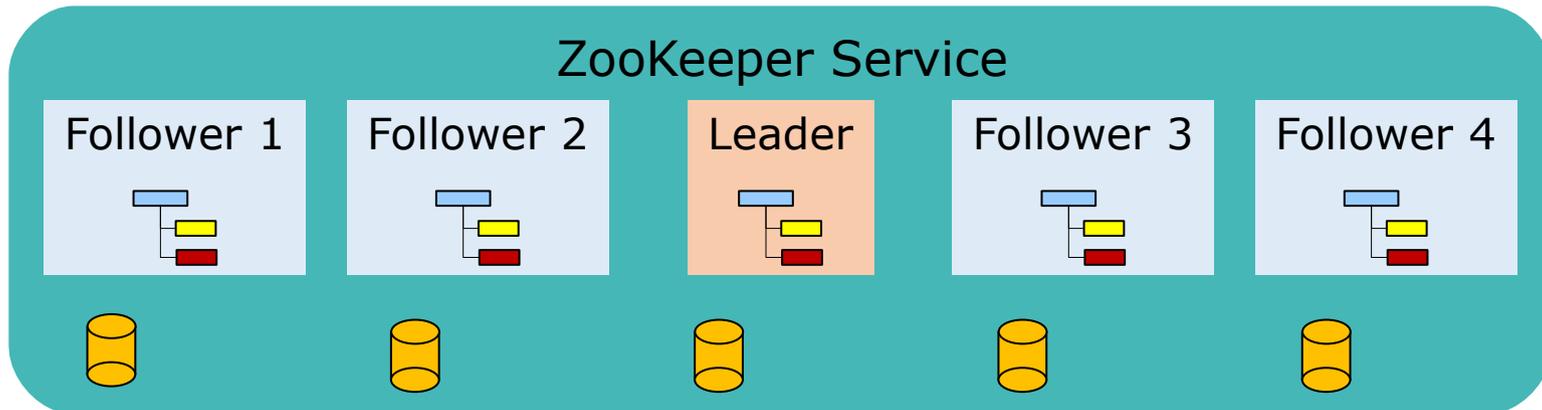
- Followers log the write to their write-ahead log
- Respond to the leader

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

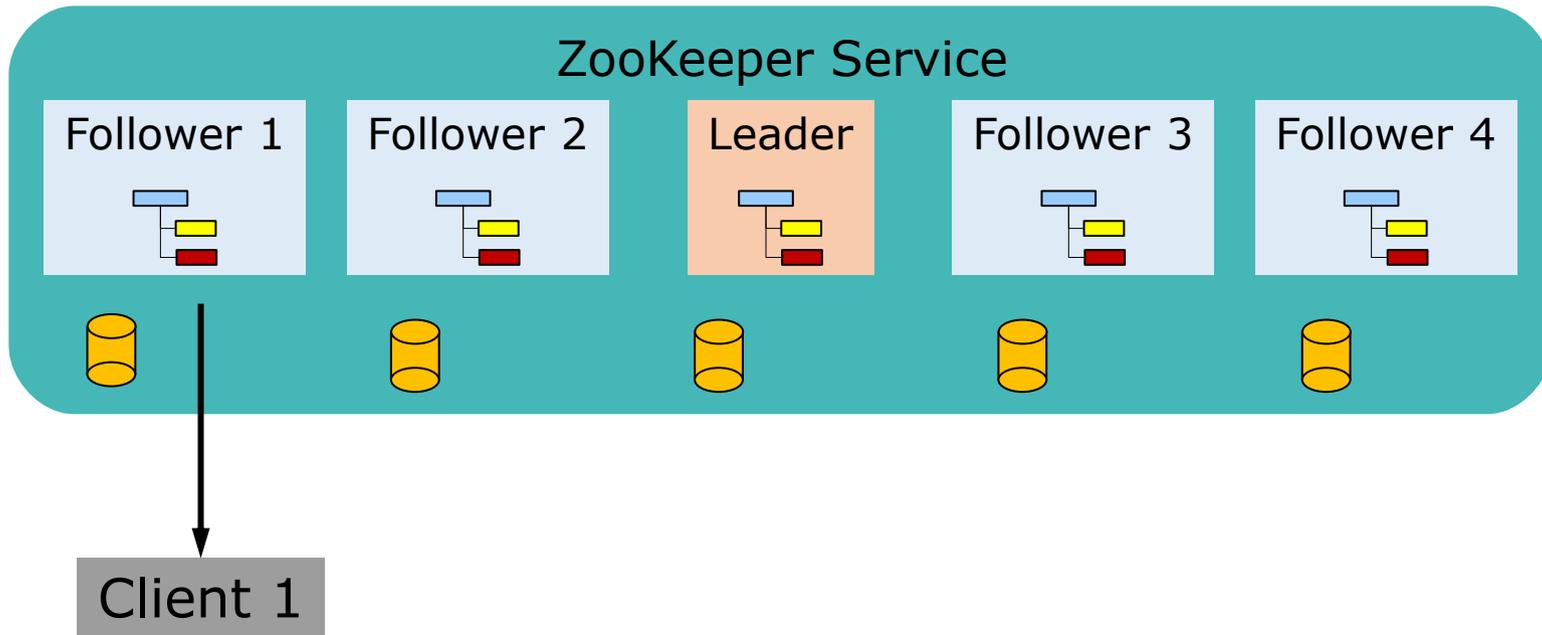
ZooKeeper: Apply Write



Client 1

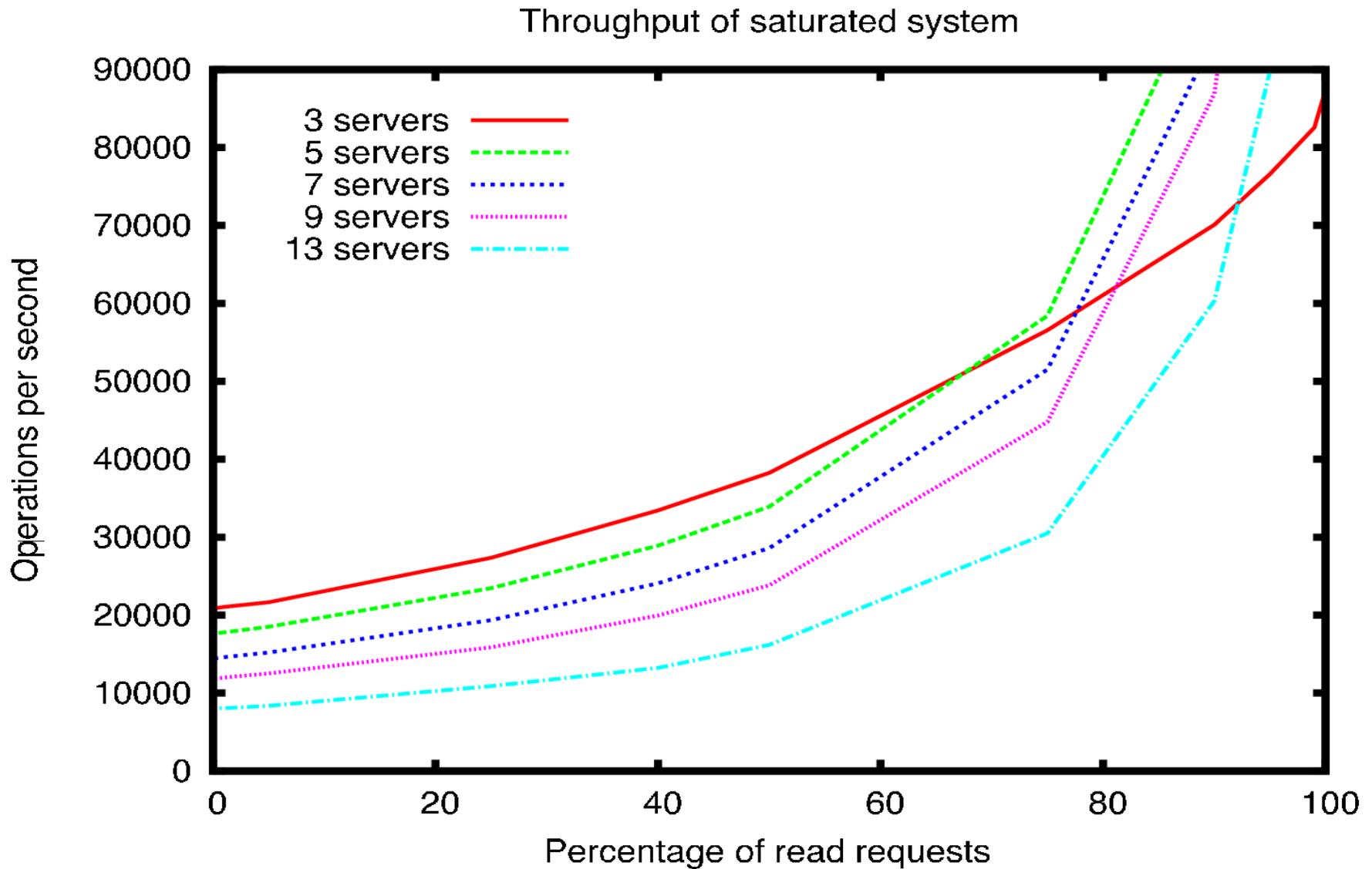
- Each follower:
 - Commits the write
 - Applies write to ZooKeeper state in memory
 - Issues watch notifications to clients connected to follower

ZooKeeper: Write Response

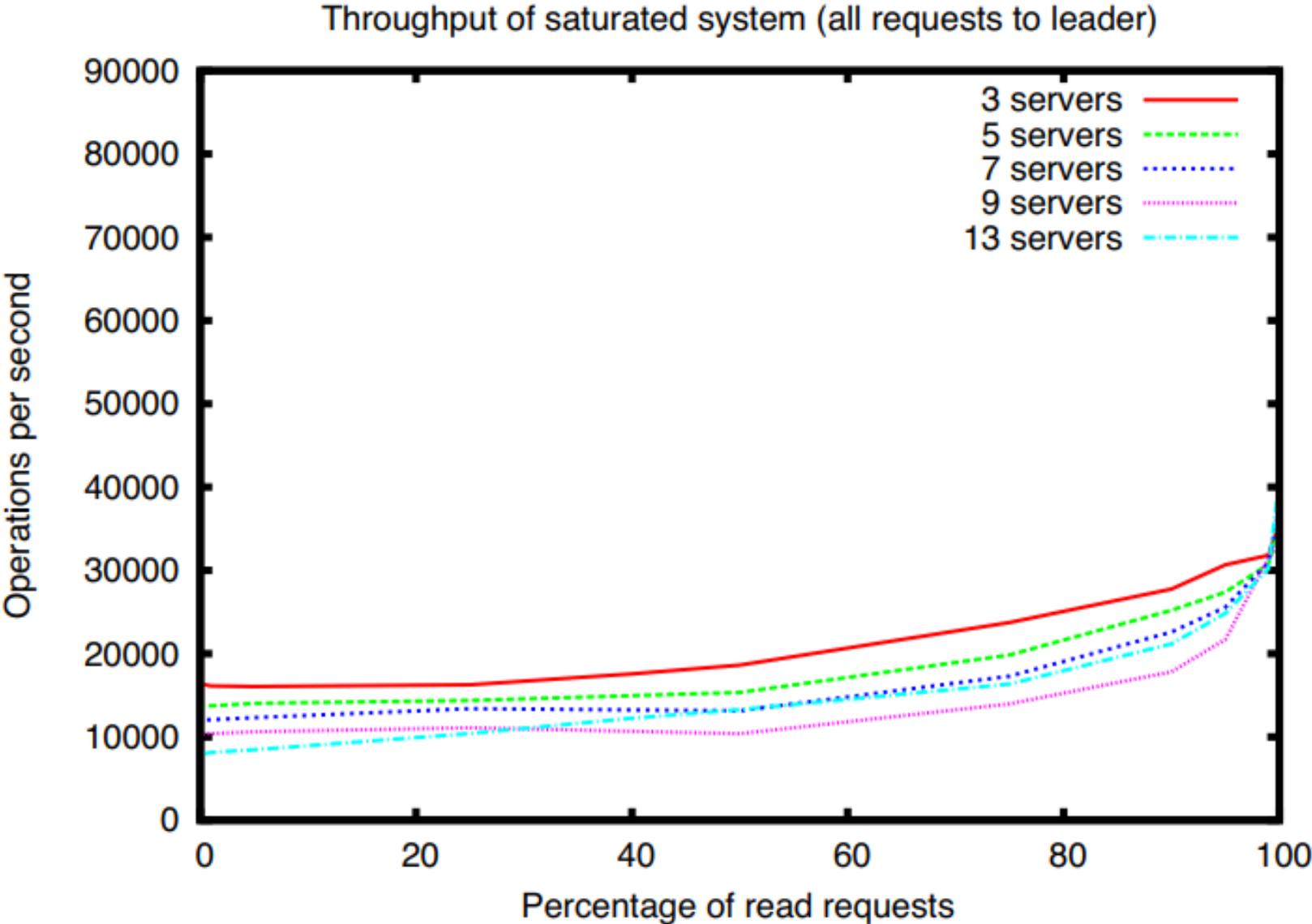


- Follower 1 delivers write response to Client 1

ZooKeeper Performance



Performance With Strong Consistency



Summary

- Easy to use
- High read performance
- General
- Reliable
- Released as an Apache open-source project
 - Relatively easy to use
 - Today, used extensively for coordination functions

Discussion

Q1

- What are wait-free operations? Why does the paper base the ZooKeeper design on wait-free operations?

Q2

- Compare ZooKeeper with RAFT in terms of
 - Functionality/purpose of the system
 - Replication method
 - Consistency guarantees and performance
 - Use of timeouts

Q3

- Why does ZooKeeper provide FIFO execution guarantees for each client's operations?

Coordinator

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

Worker

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

Q4

- The ZK locking implementation has no timeout. What happens if the lock holder dies? What if the lock holder is unresponsive for a while and then comes back?

Q5

- ZooKeeper converts write operations into **idempotent** transactions when applying them to all servers. What does idempotent mean? Why do these transactions need to be idempotent?