Big Data Systems

Djellel Difallah

Spring 2023

Lecture 13 – Cluster Coordination

Outline

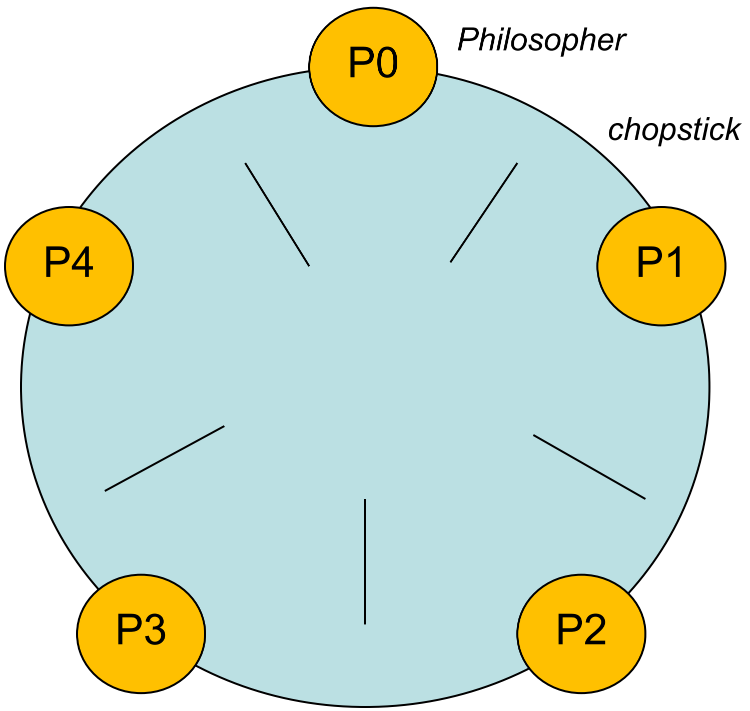
- Introduction
  - Coordination in Distributed Systems
  - The Dining Philosophers Problem
- Motivation and common issues
- Leader Election
  - Coordination in Master-Worker Architectures
- Apache Zookeeper
  - Architecture
  - Functionalities
  - Use Cases (aka recipes)

Coordination

Proper distributed system coordination is important but not easy

The Dining Philosophers Problem

- N Philosphers (5) seated around a circular table
- Each has a bowl of noodles in front of them
- Philosphers alternate between thinking and eating
- Chopstick (1 per Philospher)
- To eat, a Philospher must have both left and right chopsticks
- Rules:
  - No stealing
  - No communication
- Brainstorm!



Motivation

- Applications consist of independent programs running on a changing set of computers
- Developers have to think about the coordination logic in addition to the application logic (complex code)
- Question: Can we outsource this particular need to an external service?

Common Coordination Use Cases

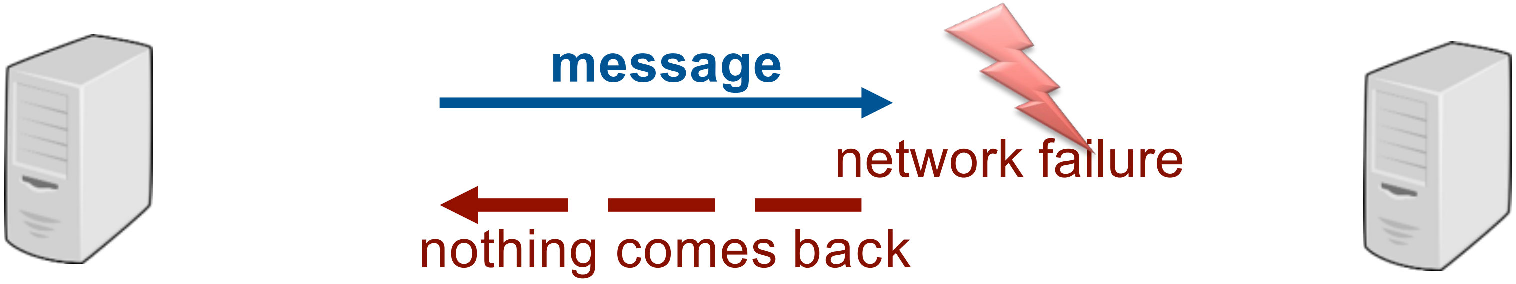
- **Static configuration**: a list of operational parameters for the system processes
- **Dynamic configuration**: parameter changes on the fly
- **Group membership**: who is alive?
- **Leader election**: who is in charge who is a backup?
- **Mutually exclusive access** to critical resources (locks)
- **Barriers** (e.g., steps in Spark)

Causes for Coordination Issues

- Message delays:
  - Network congestion can cause arbitrary delays, potentially leading to out-of-order message delivery.
- Processor speed:
  - OS scheduling and overload can induce message processing delays, increasing message latency.
- Clock drift:
  - Processor clocks can be unreliable and drift apart, causing potential errors when relying on system time.

Difficulties

Partial failures make application writing difficult

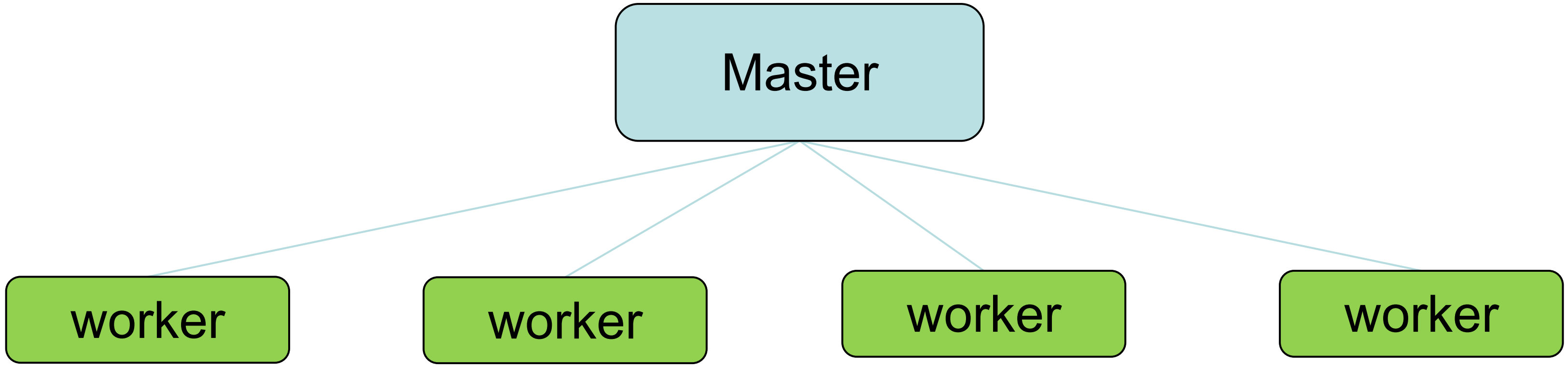


Sender does not know:

- whether the message was received
- whether the receiver's process died before/after processing the message

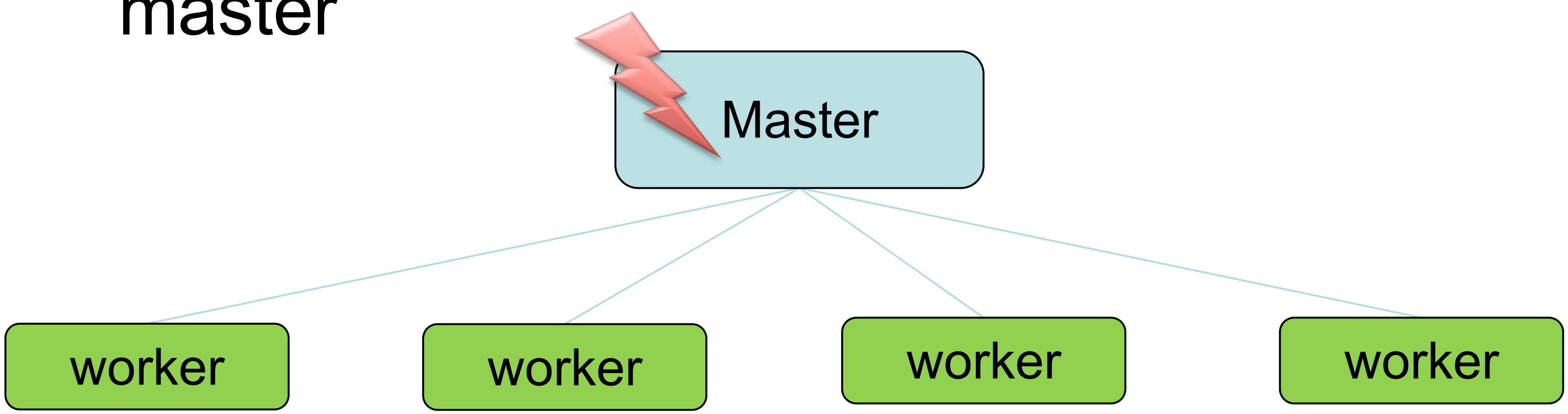
Simple Master/Worker Model

- Work assignment
  - Master assigns work
  - Workers execute tasks assigned by master



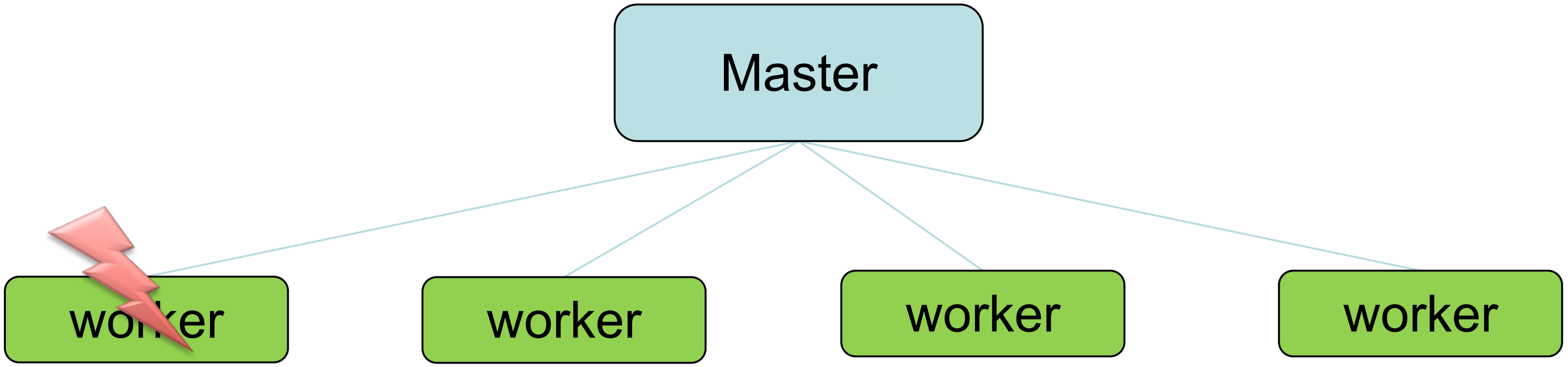
Master Crash

- Single point of failure
- No work is assigned
- Need to select a new master



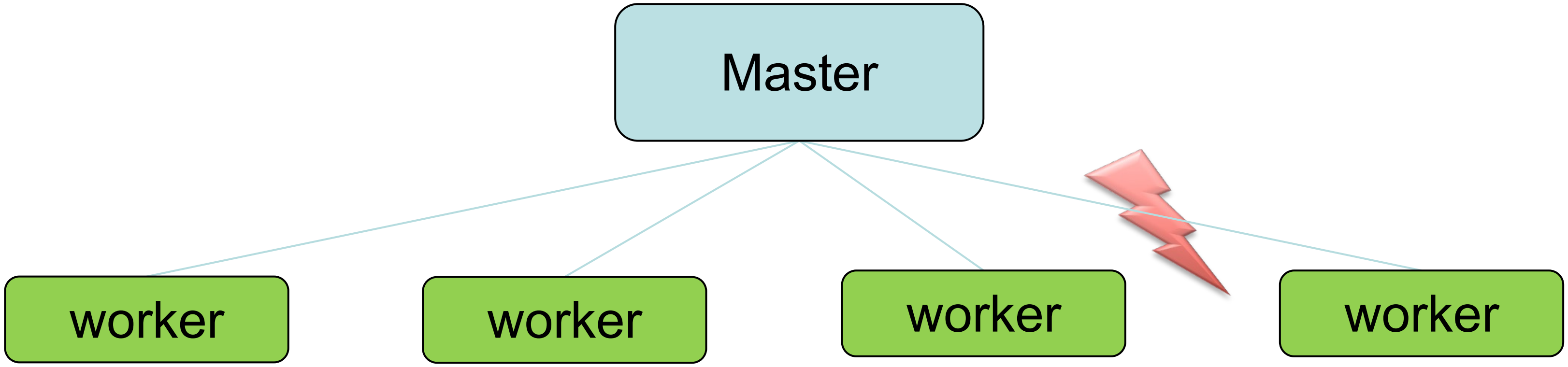
Worker Crash

- Overall system still works
  - – Does not work if there are dependencies
- Some tasks will never be executed
- Need to detect crashed workers

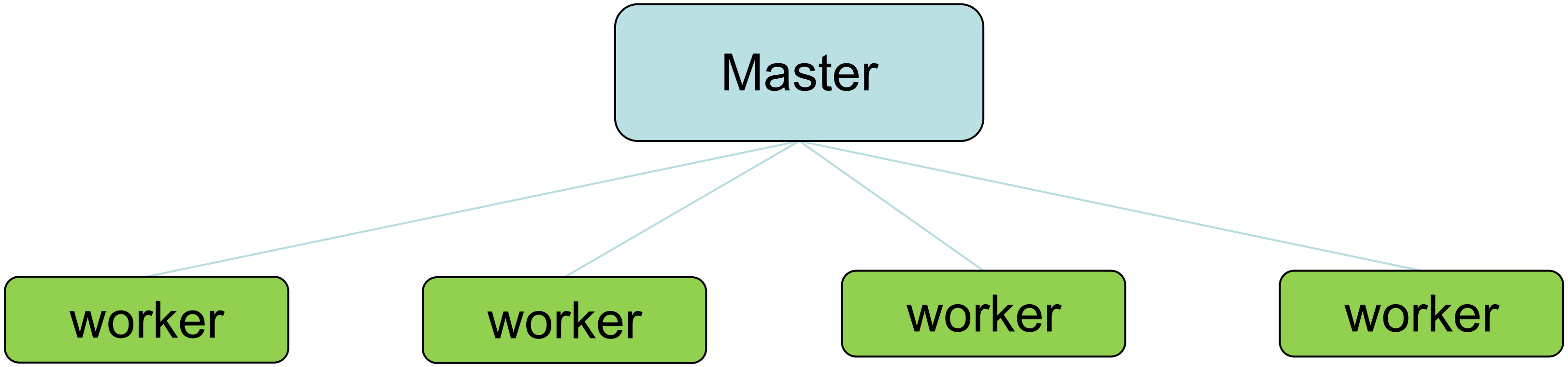


Worker does not receive assignment

- Same problem as before
- Some tasks may not be executed
- Need to guarantee that worker receives assignment

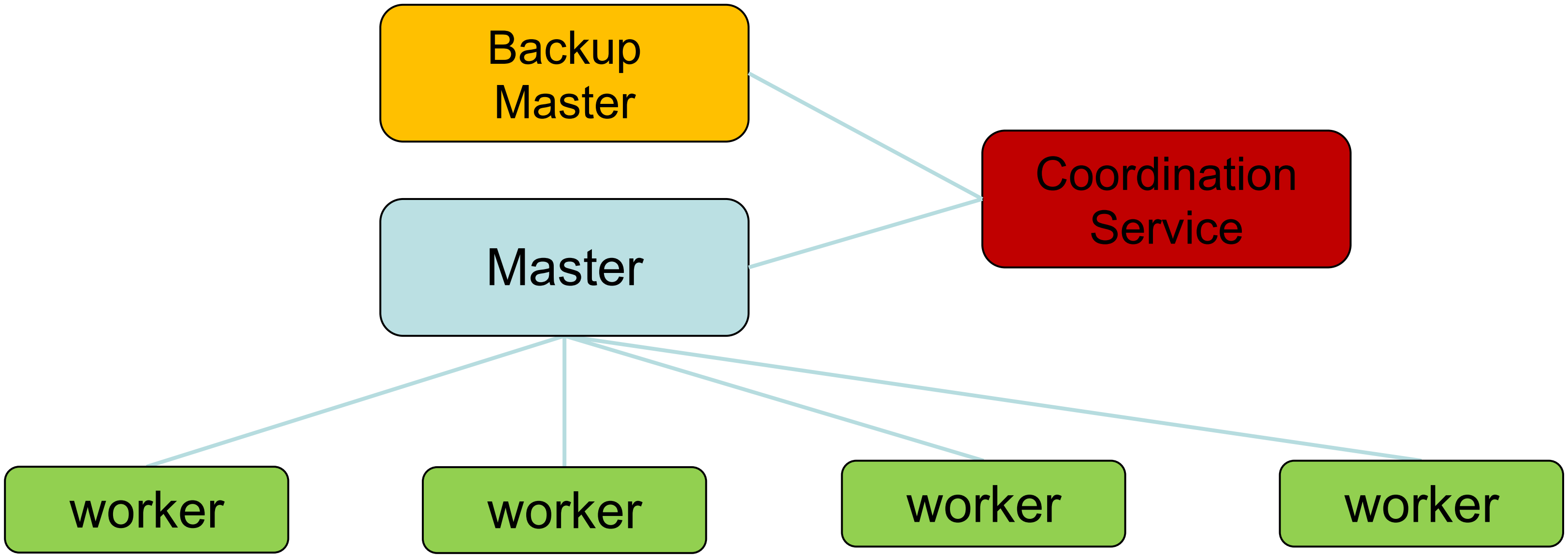


Distributed Systems



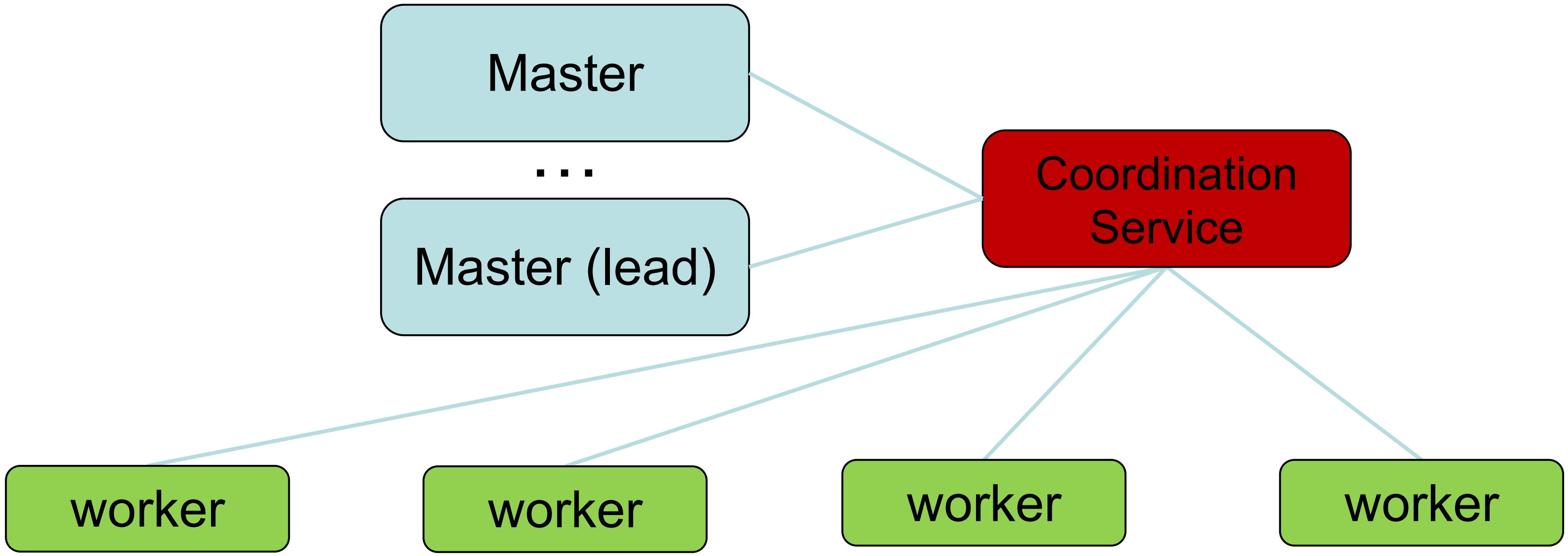
\+ simple
\- coordination performed by the master
\- single point of failure
\- scalability

Fault-tolerant Distributed Systems



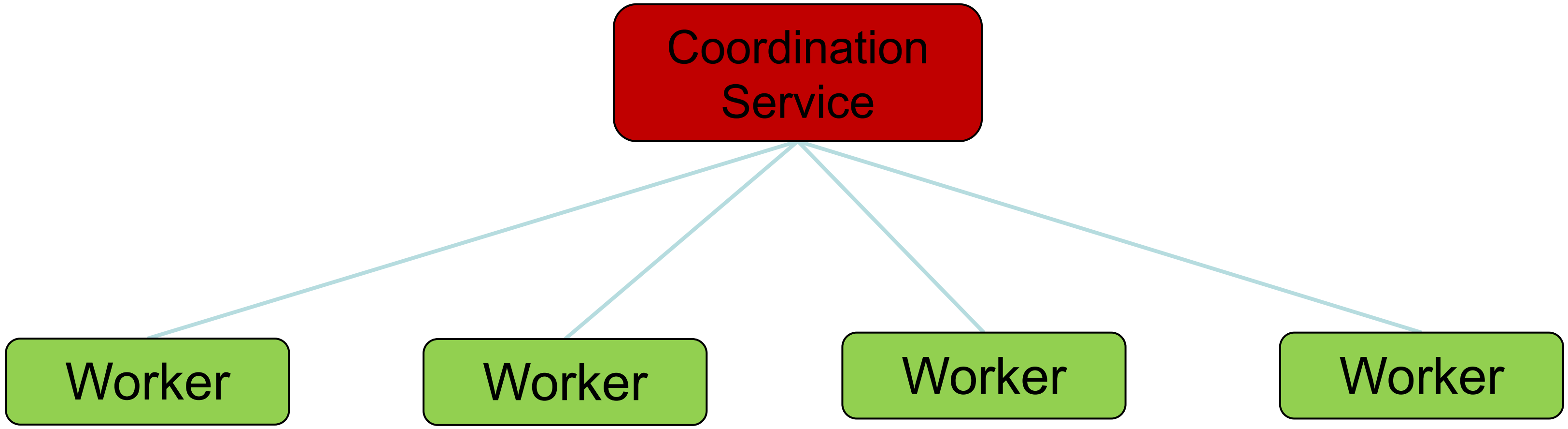
+ not a single point of failure anymore
- scalability is still an issue

Distributed Systems



+ Scalability

“Fully” Distributed Systems



++ Scalability

APACHE ZOOKEEPER

Apache Zookeeper

- A highly-availability service for coordinating processes in distributed systems

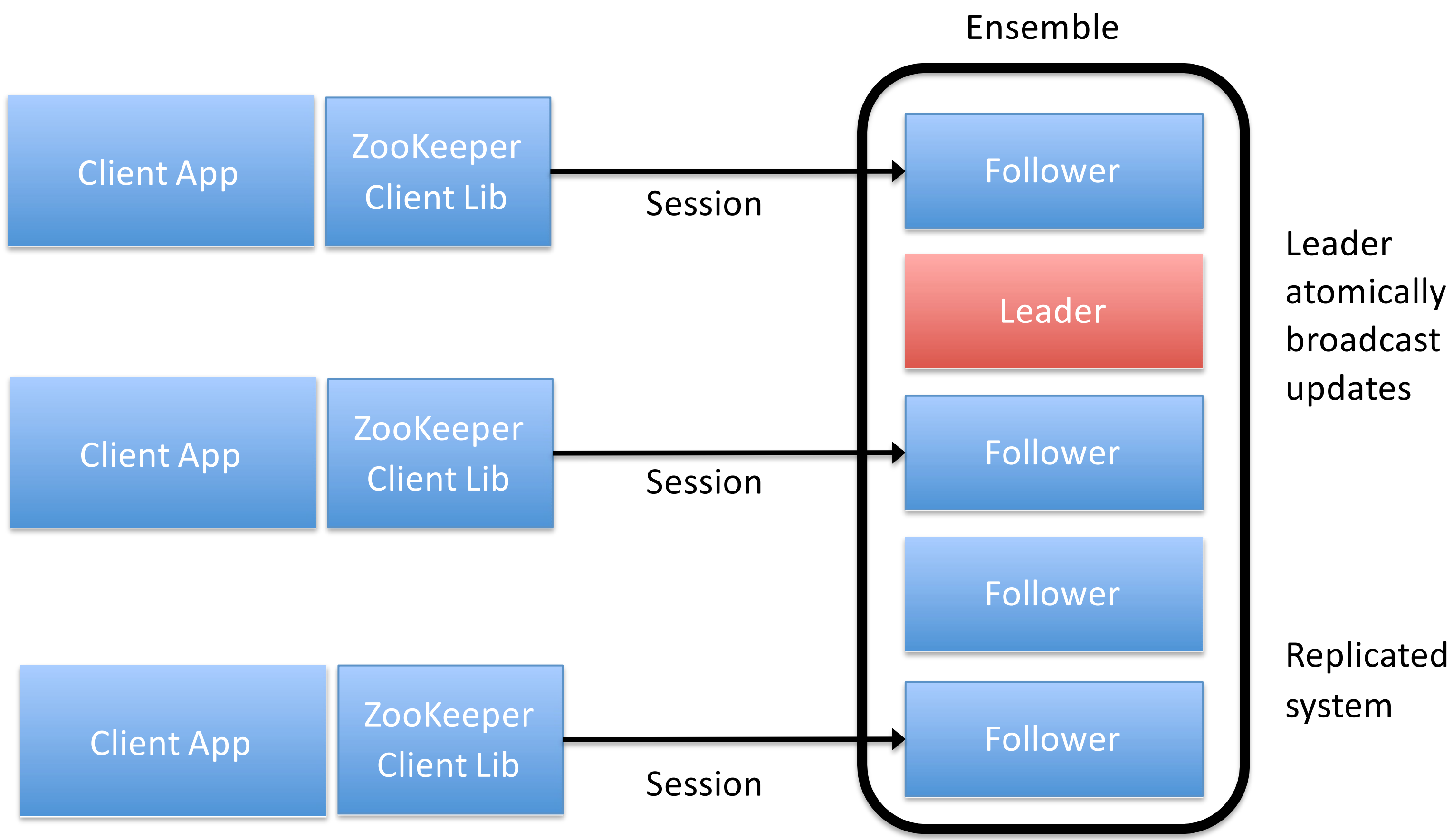
- Developped at Yahoo Research
  - *[Paper] "ZooKeeper: Wait-free coordination for Internet-scale systems"* by Hunt et al., 2010.

- Initially a subproject of Hadoop.
  - Now, it is a top-level project that can run independently.

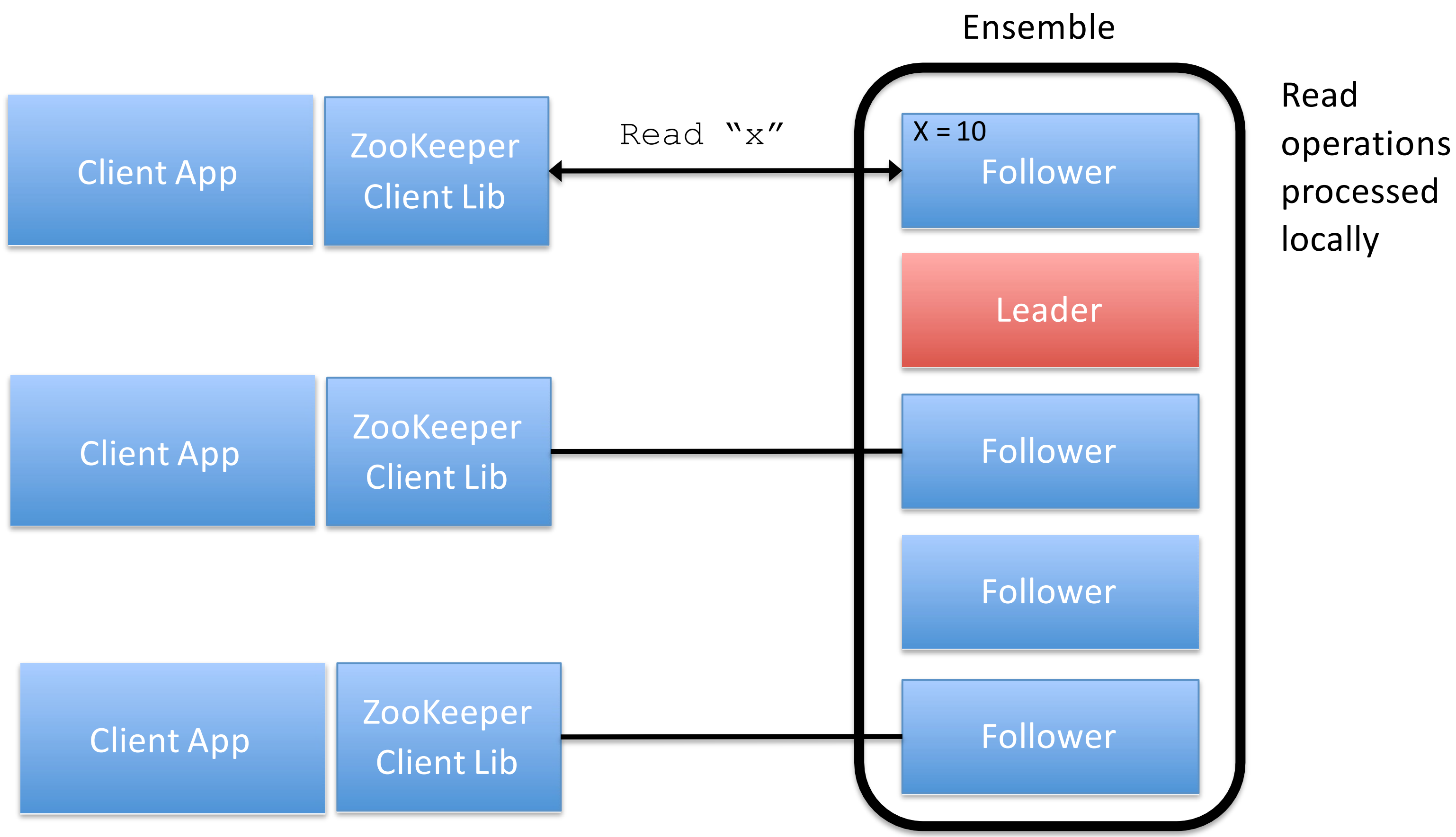
Availability

- A collection of Zookeeper servers called *Ensembles*
  - 1 Leader, multiple replicas
- If the leader fails then another replica takes over
- Ensemble available if majority servers running
- Ensembles should have an odd numbers of servers, e.g.
  - 3 servers supports 1 server failure (2 out of 3 running)
  - 5 servers support 2 server failures (3 out of 5 running)
  - 7 servers support 3 server failures (4 out of 7 running)
- Reads go to any server (succeed if >= 1)
- Writes go to leader only and succeed if more than half the servers running (Quorum)

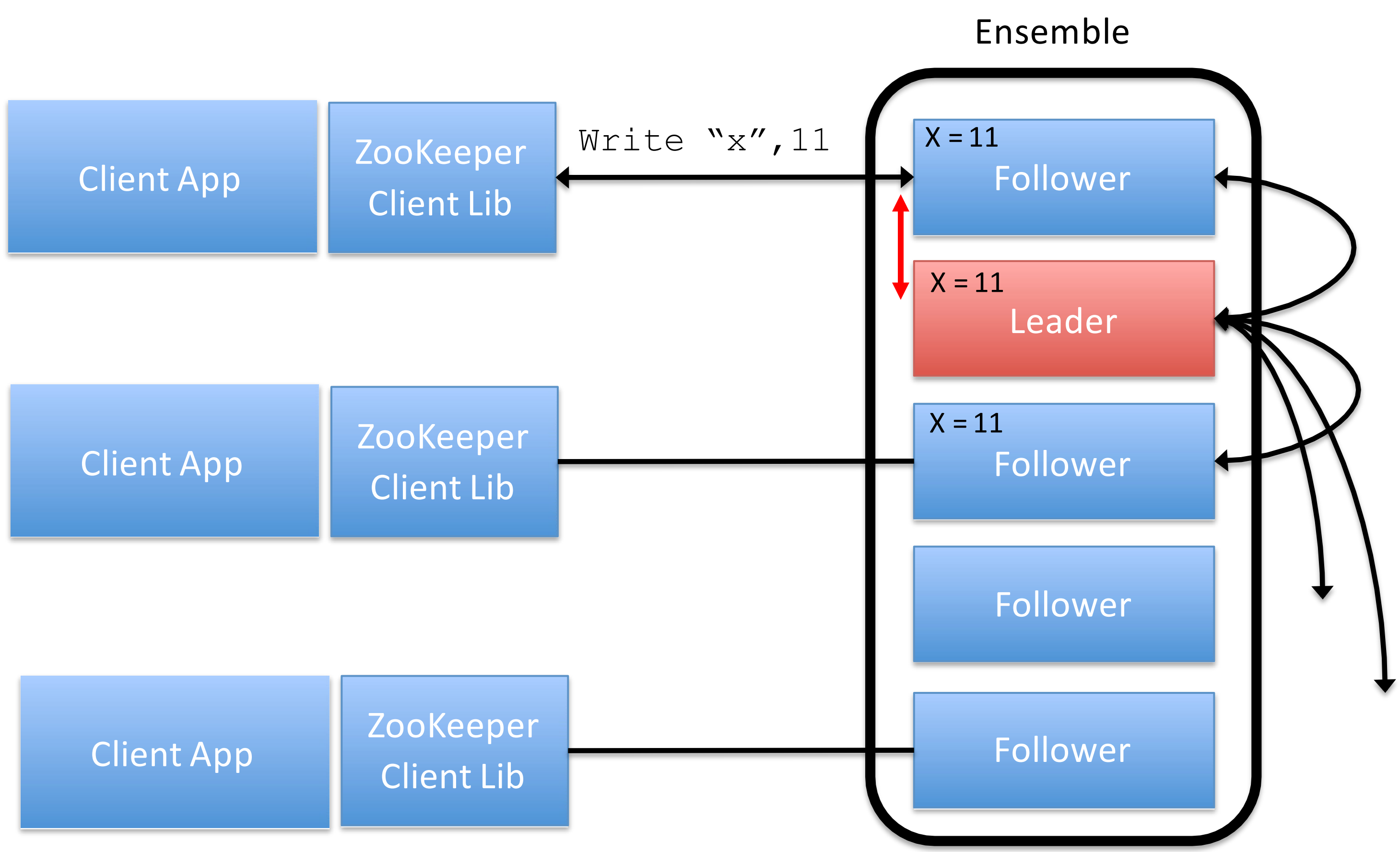
Zookeeper Architecture



Zookeeper: Read Ops



Zookeeper: Write Ops



Replicates across a quorum of nodes

ZooKeeper Design Principles

- API is wait-free
  - No blocking primitives in ZooKeeper
  - Blocking can be implemented by a client
  - No deadlocks
- Guarantees
  - Client requests are processed in FIFO order
  - Writes to ZooKeeper are linearizable
- Clients receive notifications of changes before the changed data becomes visible

ZooKeeper: Fast and Reliable

- ZooKeeper service is an ensemble of servers that use replication (high availability)
- Data is cached on the client side:
  - Example: a client caches the ID of the current leader instead of probing ZooKeeper every time.
- What if a new leader is elected?
  - Potential solution: polling (not optimal)
  - Watch mechanism: clients can watch for an update of a given data object
- ZooKeeper is optimized for read-mostly operations
  - In memory objects, with disk logs

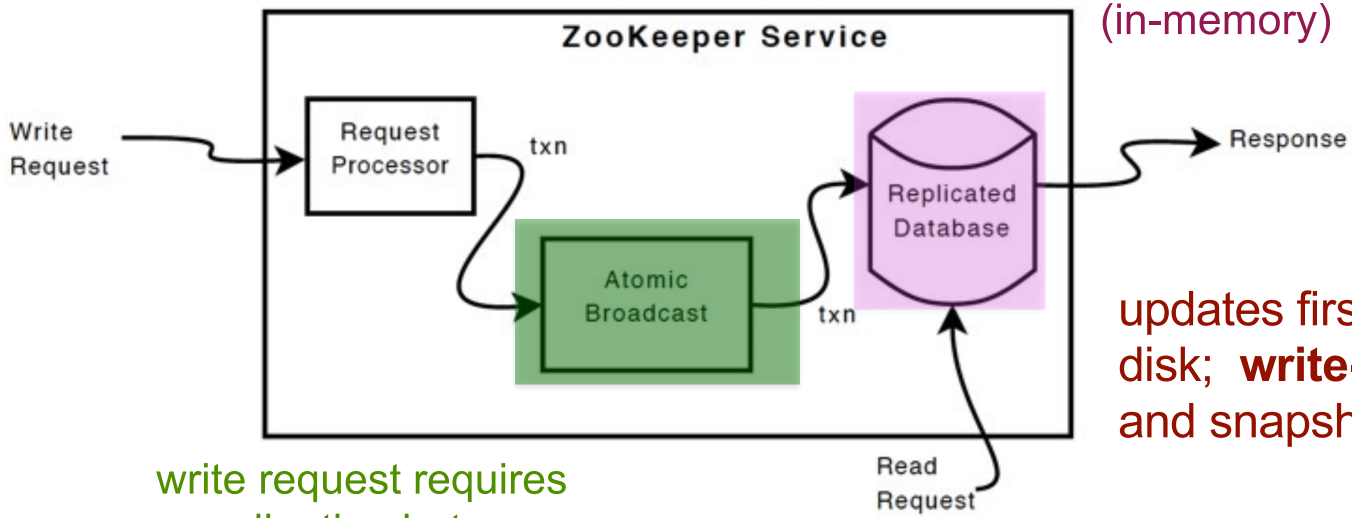
Implementation Details

- ZooKeeper server services clients
- Clients connect to exactly one server to submit requests
  - **read** requests served from the local replica
  - **write** requests are processed by an agreement protocol (an elected server leader initiates processing of the write request)

Implementation Details

Internal components: *Replicated database, request processor, leader, and followers.*

replicated across all servers (in-memory)



updates first logged to disk; **write-ahead log** and snapshot for recovery

write request requires coordination between servers

Zookeeper Components

- **Client**: user of the ZooKeeper service
- **Server**: process providing the ZooKeeper service
- Clients establish a **session** when connecting to ZooKeeper

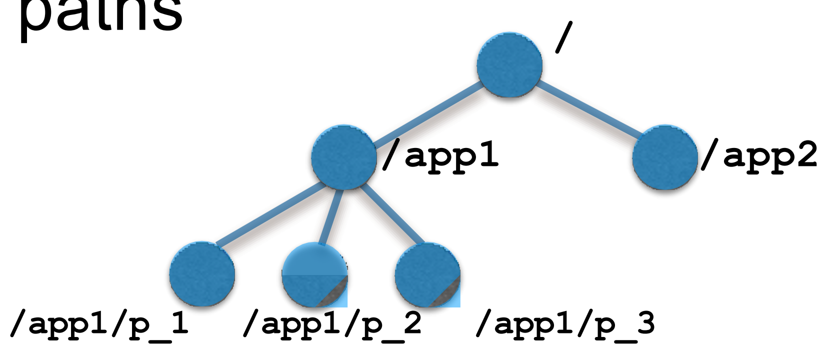
- **znode**: **in-memory** data node in ZooKeeper, organised in a hierarchical namespace (the data tree)
- **Operations:** which modifies the state of the data tree

Sessions

- When a client connects to ZooKeeper a new sessions is initiated
- Sessions have an associated **timeout**
- ZooKeeper considers a client faulty if it does not receive anything from its session for more than that timeout
- Session ends:
  - Faulty client
  - Ended by client

ZooKeeper's data model

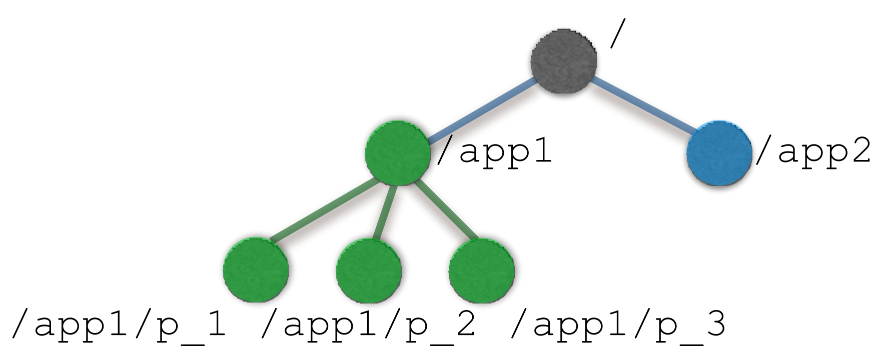
- znodes are organised in a hierarchical namespace
- znodes can be manipulated by clients through the ZooKeeper API
- znodes are referred to by UNIX style file system paths



All znodes store data (file like) & can have children (directory like).

Znodes

- znodes are not designed for general data storage (usually require storage in the order of kilobytes)
- znodes map to abstractions of the client application

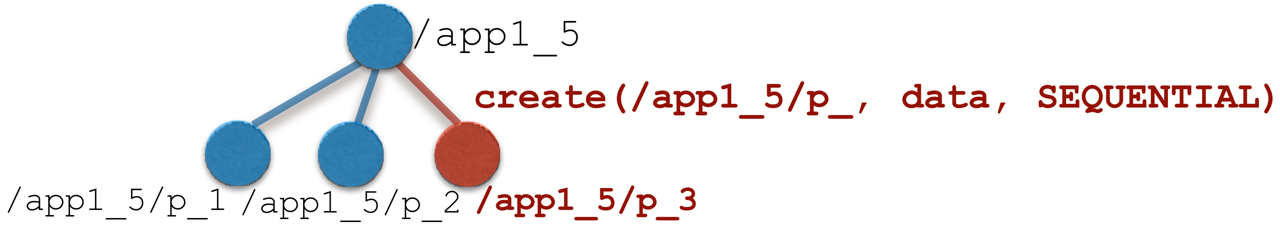


Zookeeper API

- Create znodes: create
  - Flags: *Persistent, sequential, ephemeral*
- Read and modify data: setData, getData
- Read the children of znode: getChildren
- Check if znode exists: exists
- Delete a znode: delete

Znode Flags

- Clients manipulate znodes by creating and deleting them
- EPHEMERAL flag:
  - clients create znodes which are deleted at the end of the client's session
- SEQUENTIAL flag:
  - monotonically increasing counter appended to a znode's path
  - counter value of a new znode under a parent is always larger than value of existing children



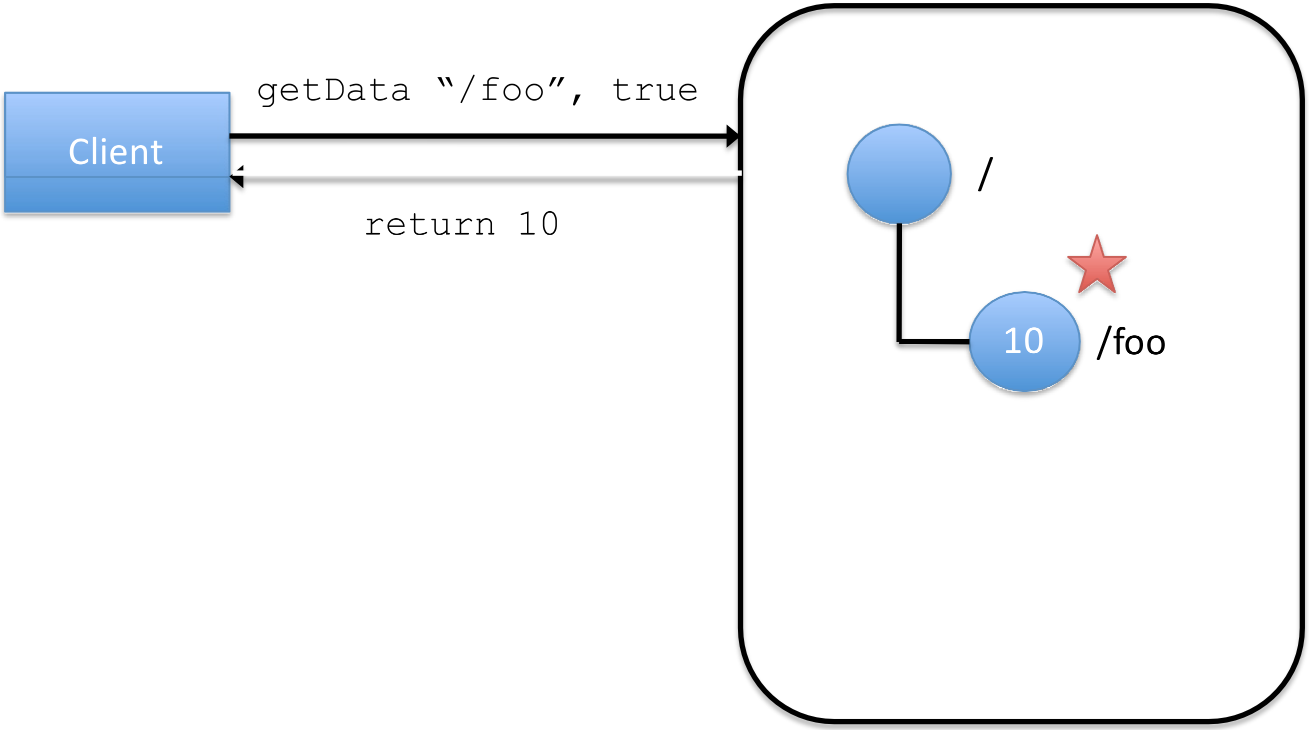
Znodes and Watch Flags

- To learn of znode changes clients can issue read operations on znodes with a **watch flag**
  - **Set a watch**
  - Upon chang in Znode, the Server notifies the client when the information on the znode has changed

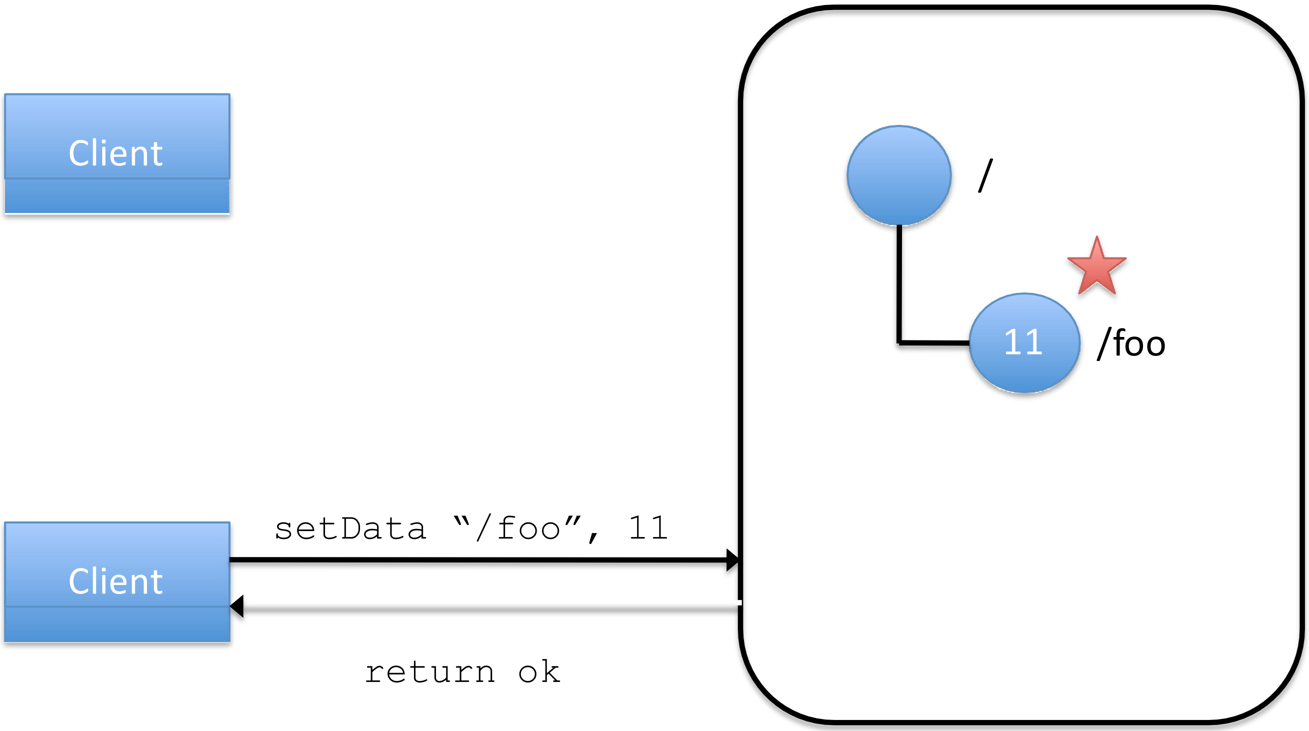
- Watches are one-time triggers associated with a session (unregistered once triggered or session closes)

- Watch notifications indicate the change, not the new data

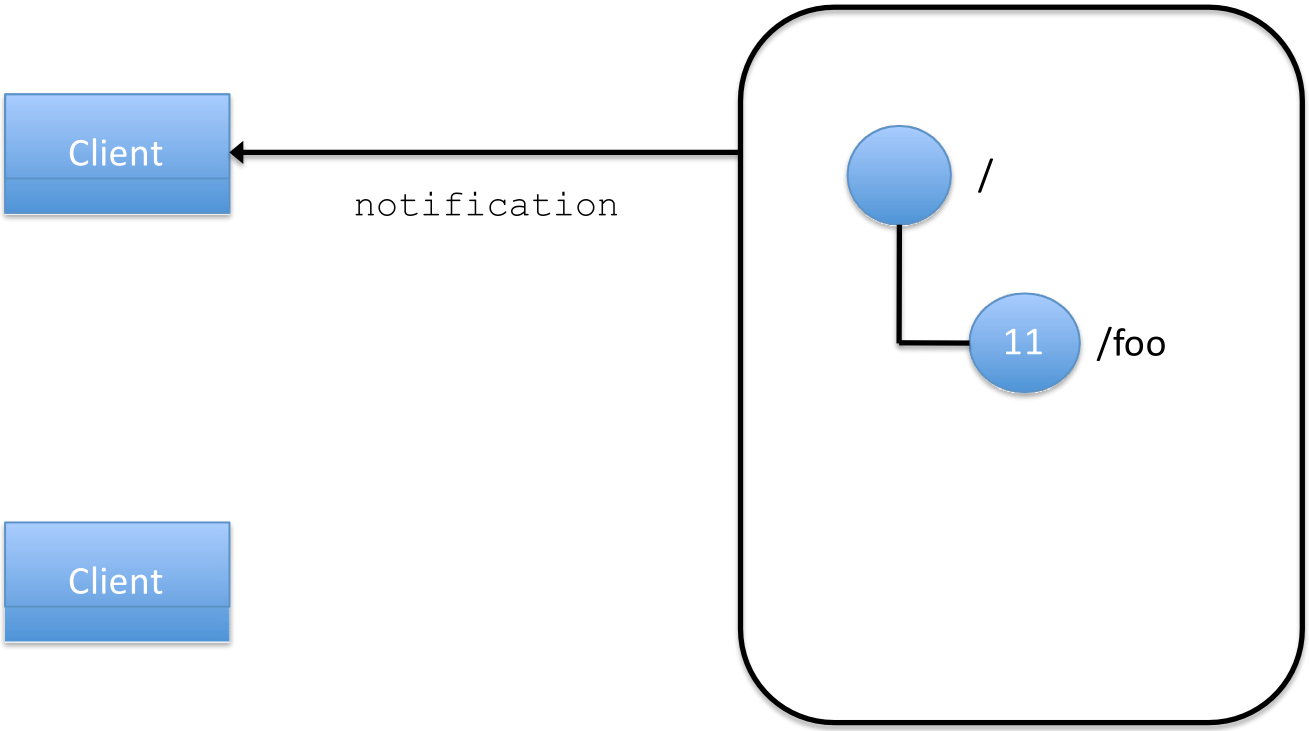
ZooKeeper: Watches



ZooKeeper: Watches



ZooKeeper: Watches

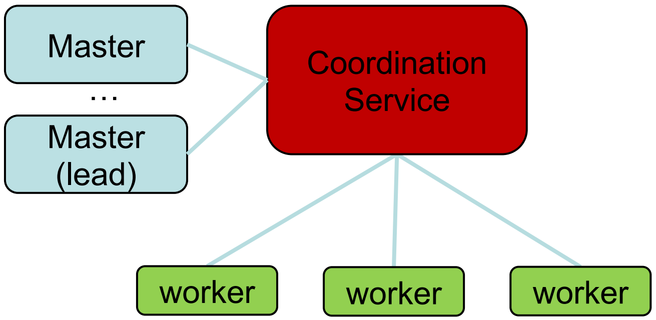


Use Case: Locks with ZooKeeper

- Implementing various lock types (e.g., read/write, global) for critical sections.
- Lock acquisition: Each process 'p' creates an ephemeral znode "/lock"; if successful, it holds the lock and enters the critical section.
  - Deadlock prevention: Ephemeral znodes ensure lock release upon process crashes, avoiding system deadlock.
  - Lock contention: Processes that fail to create "/lock" watch for znode deletion and retry lock acquisition when notified.
- Continuous attempts: If still interested, a process 'p'' will repeat lock acquisition steps, watching the znode if another process has already created it.

Master/Worker System

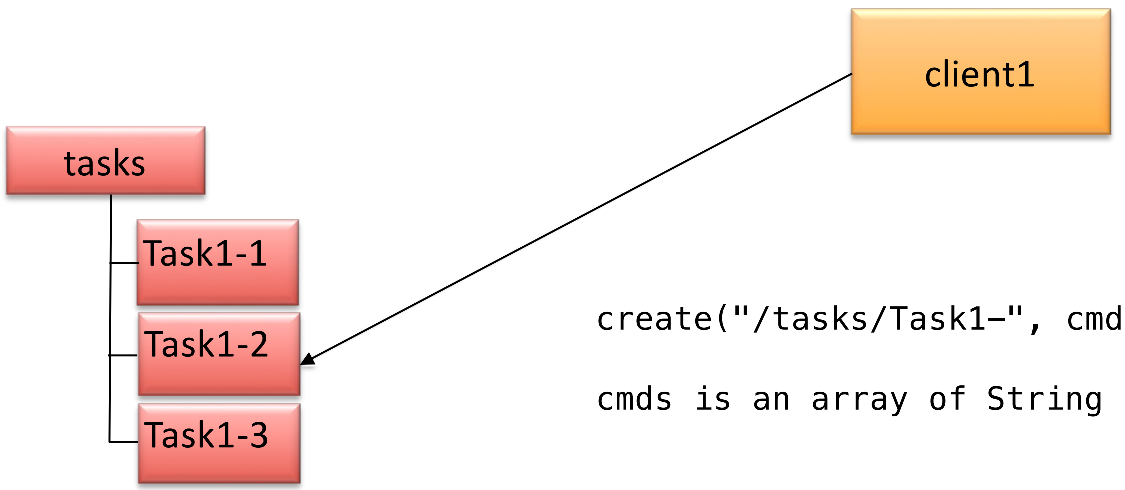
- Clients
  - Monitor the tasks
  - Queue tasks to be executed
- Masters
  - Assign tasks to workers
- Workers
  - Get tasks from the master
  - Execute tasks



APACHE ZOOKEEPER RECIPES

Curator https://curator.apache.org/

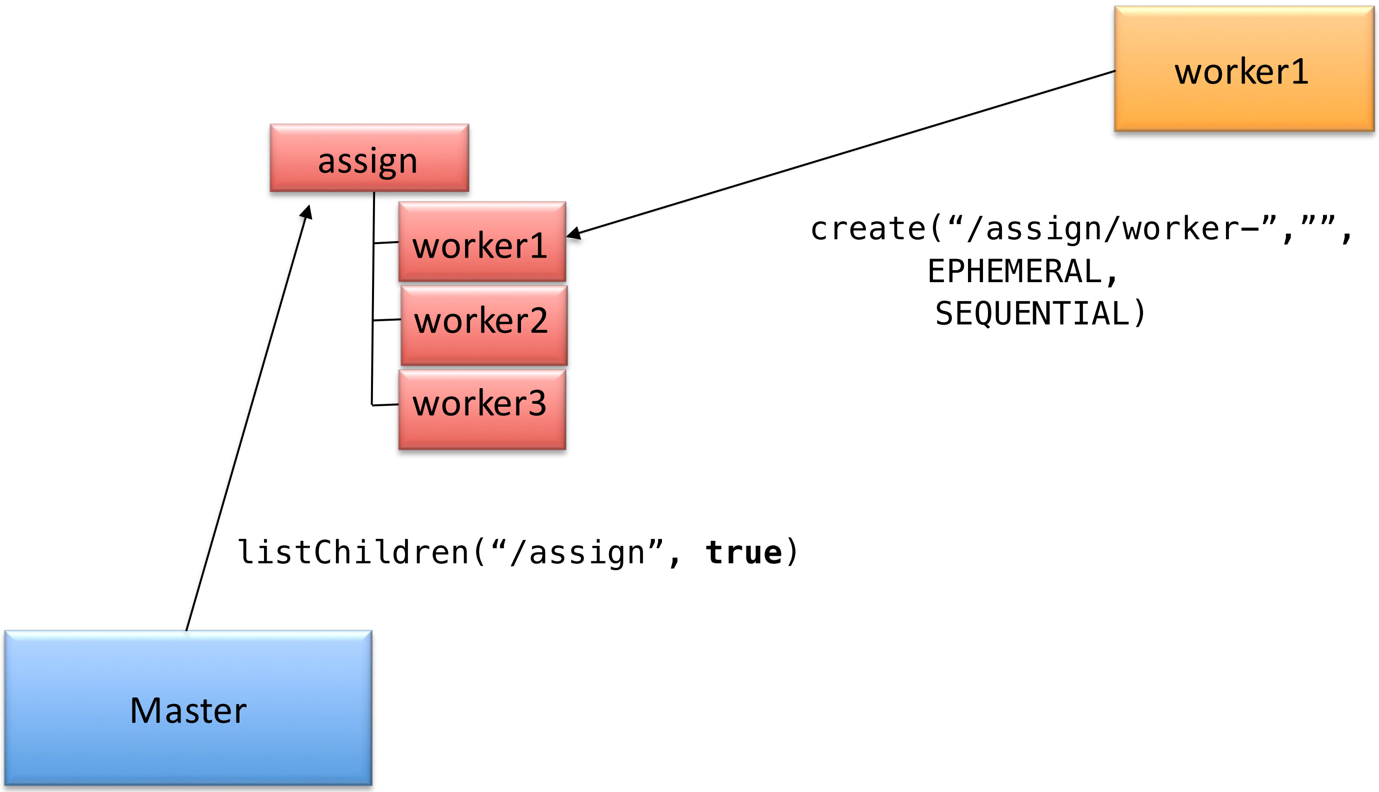
Task Queue



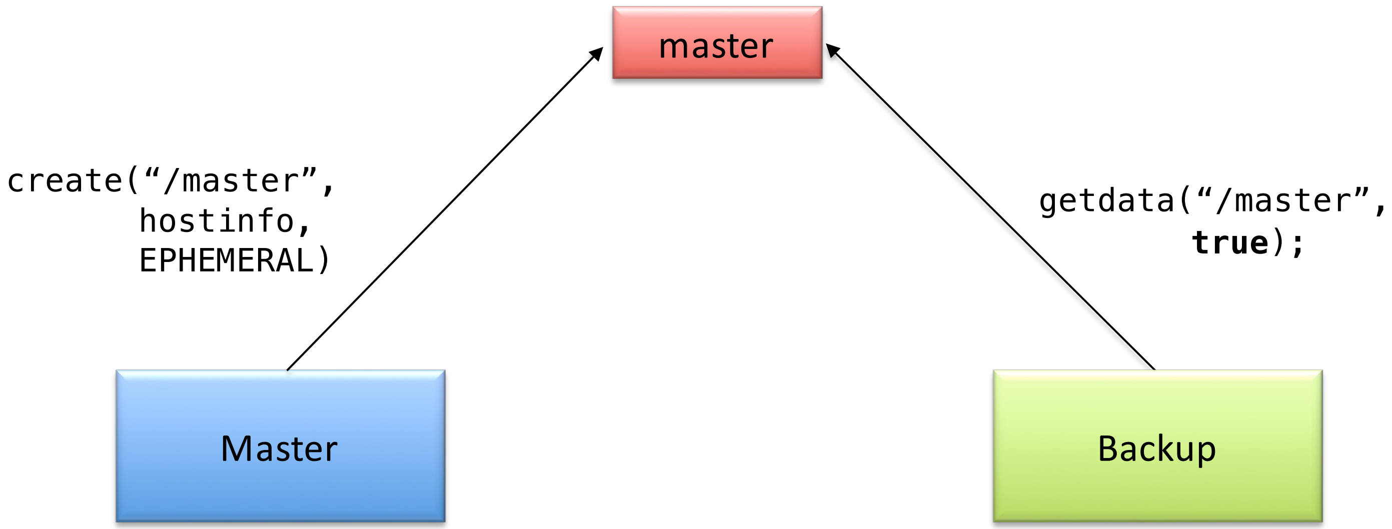

```
create("/tasks/Task1-", cmds, SEQUENTIAL)

cmds is an array of String
```

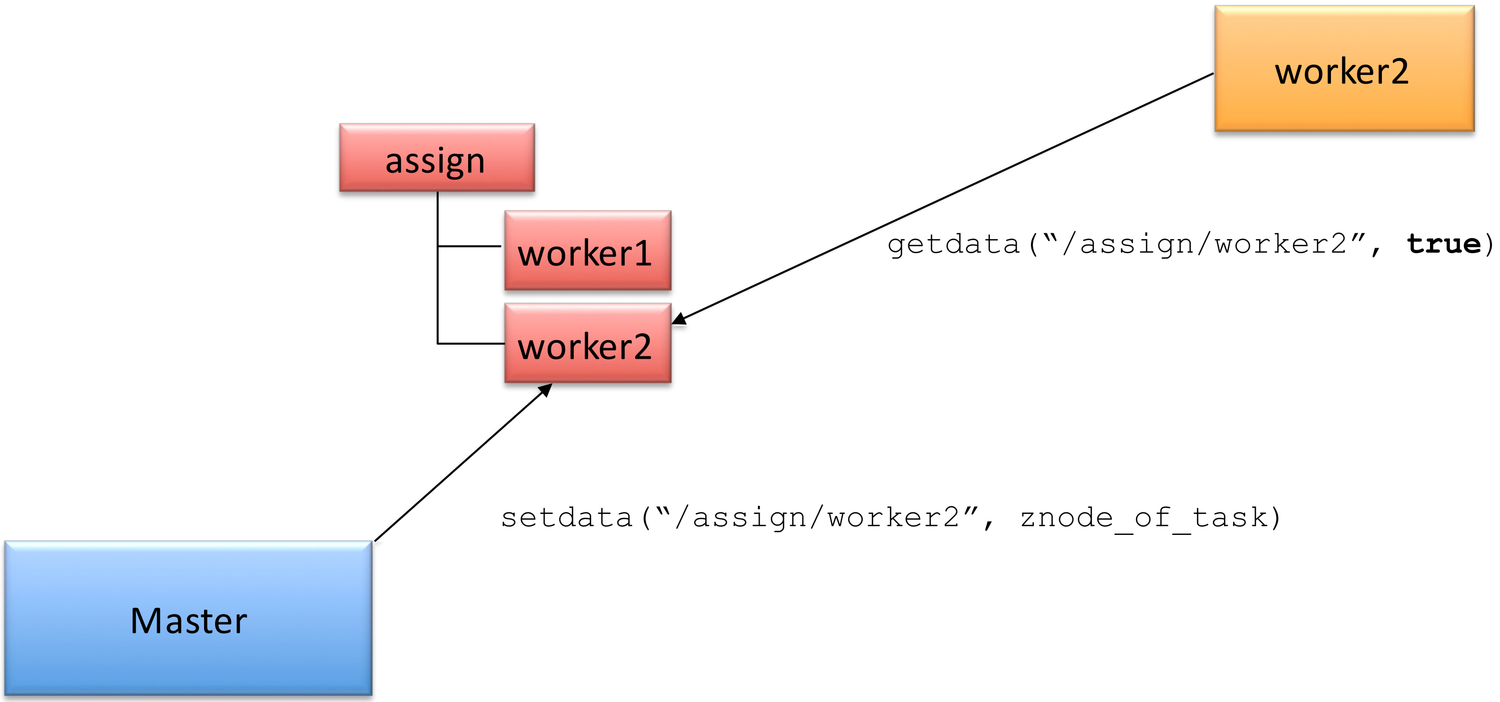
Group Membership



Leader Election



Configuration



ZooKeeper and Kafka

- Zookeeper manages Kafka brokers (keeps a list of them)
- Zookeeper helps in performing leader election for partitions
- Zookeeper sends notifications to Kafka in case of changes (e.g., new topic, broker dies, broker comes up, delete topics, etc)

Evolution of Kafka:

- ➢ Kafka 2.x can't work without Zookeeper
- ➢ Kafka 3.x can work without Zookeeper (KIP-500) - using Kraft instead
- ➢ Kafka 4.x will not have Zookeeper

Install Zookeeper

- Macos:
  - brew install zookeeper
- Windows:
  - [Tutorial]

- Let's do a demo using a console-based client **zkCli**