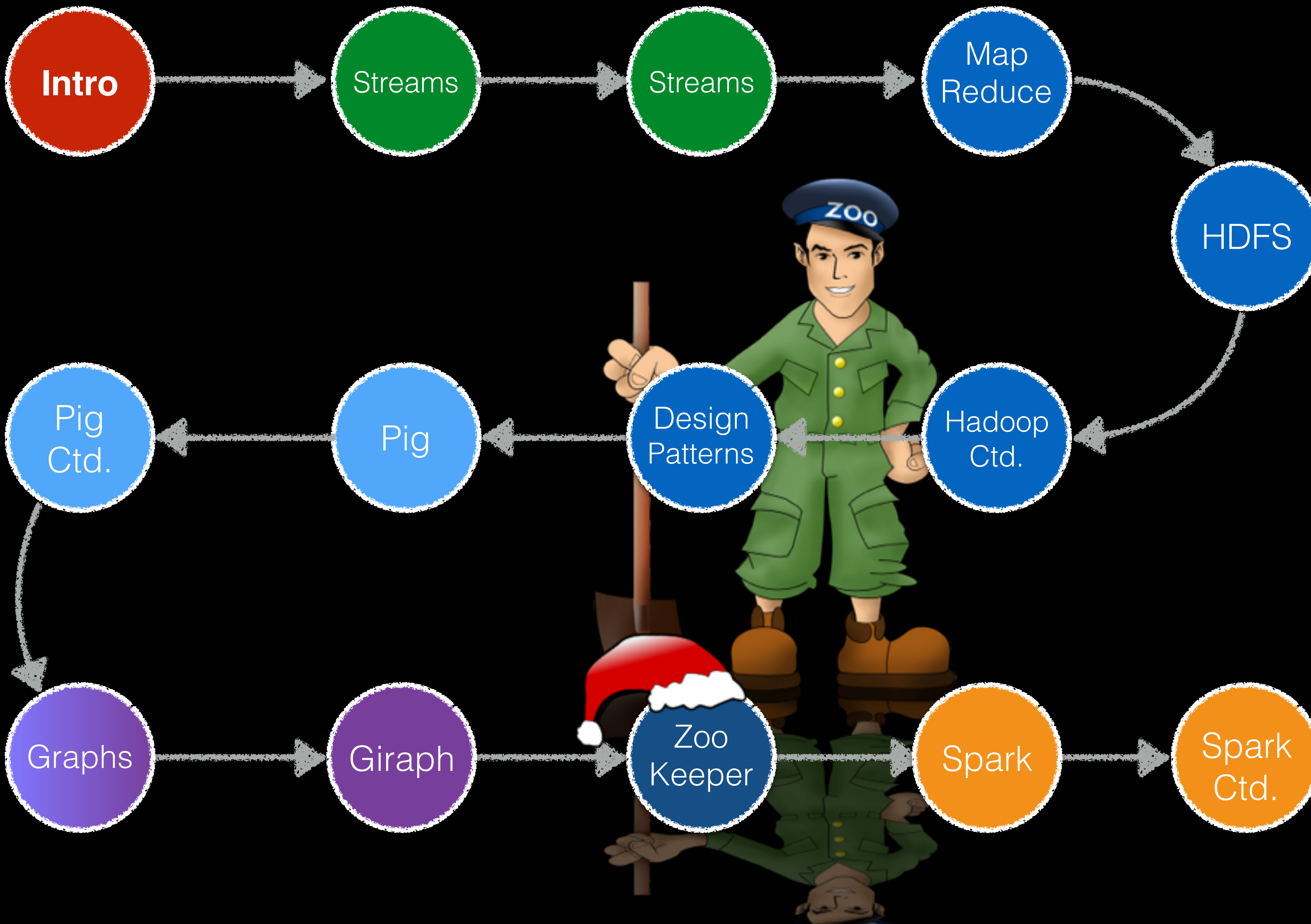
TI2736-B

Big Data Processing

Claudia Hauff
ti2736b-ewi@tudelft.nl

TU Delft


Intro
Streams
Streams
Map Reduce
HDFS
Hadoop Ctd.
Design Patterns
Pig
Pig Ctd.
Graphs
Giraph
Zoo Keeper
Spark
Spark Ctd.
ZOO

Learning objectives

- **Explain** and discuss the advantages of using ZooKeeper compared to a distributed system not using it
- **Explain** ZooKeeper’s data model
- **Derive** protocols to implement configuration tasks

Introduction

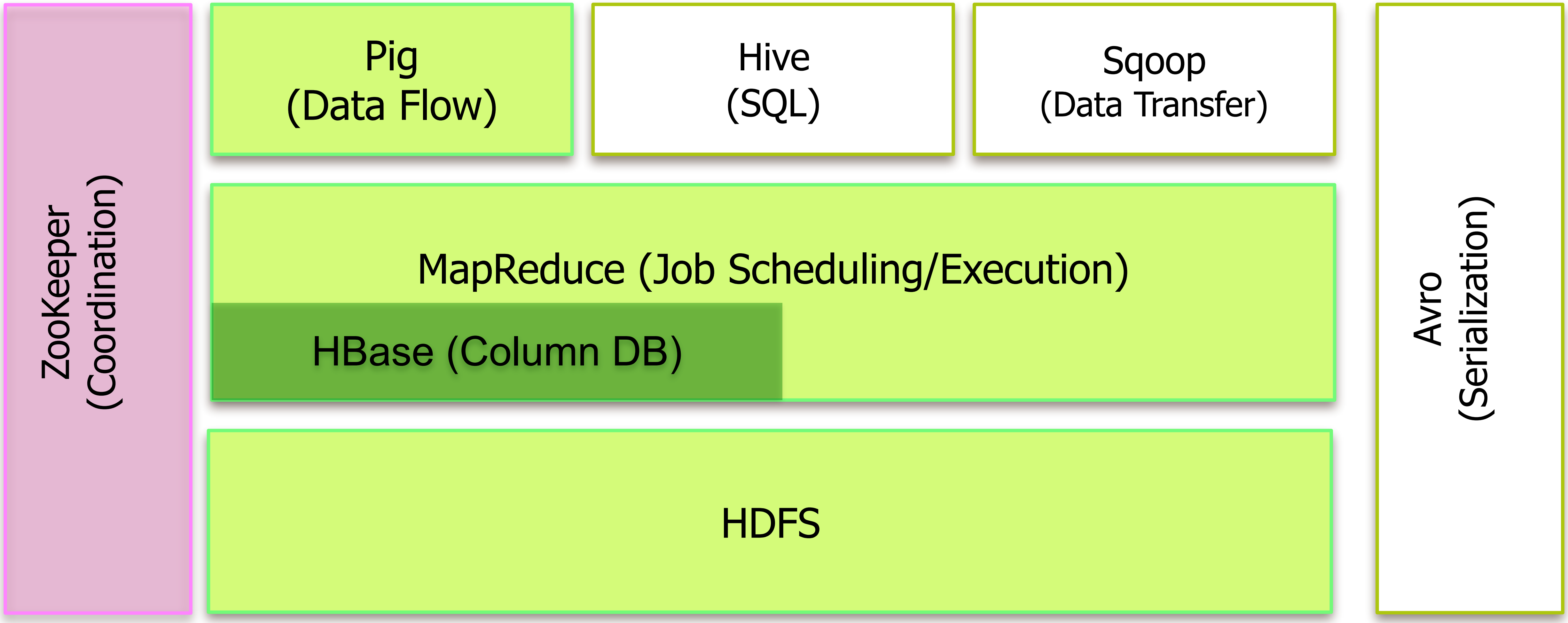
ZooKeeper

A highly-available service for coordinating processes of distributed applications.

- Developed at **Yahoo! Research** (2010 paper)
- Started as sub-project of Hadoop, since 2011 a **top-level Apache project**
- Development is driven by application needs

http://zookeeper.apache.org/

ZooKeeper in the Hadoop ecosystem



Coordination

Proper coordination is not easy.

Fallacies of distributed computing

- The network is **reliable**
- There is **no latency**
- The **topology** does not change
- The network is **homogeneous**
- The bandwidth is **infinite**
- …

Motivation

- In the past: a **single** program running on a **single** computer with a **single** CPU
- Today: applications consist of **independent** programs running on a **changing** set of computers
- Difficulty: coordination of those independent programs
- Developers have to deal with **coordination logic** and **application logic** at the same time

ZooKeeper: **designed** to relieve developers from writing coordination logic code.

Lets think ….

Question: how do you elect the leader?

application logic

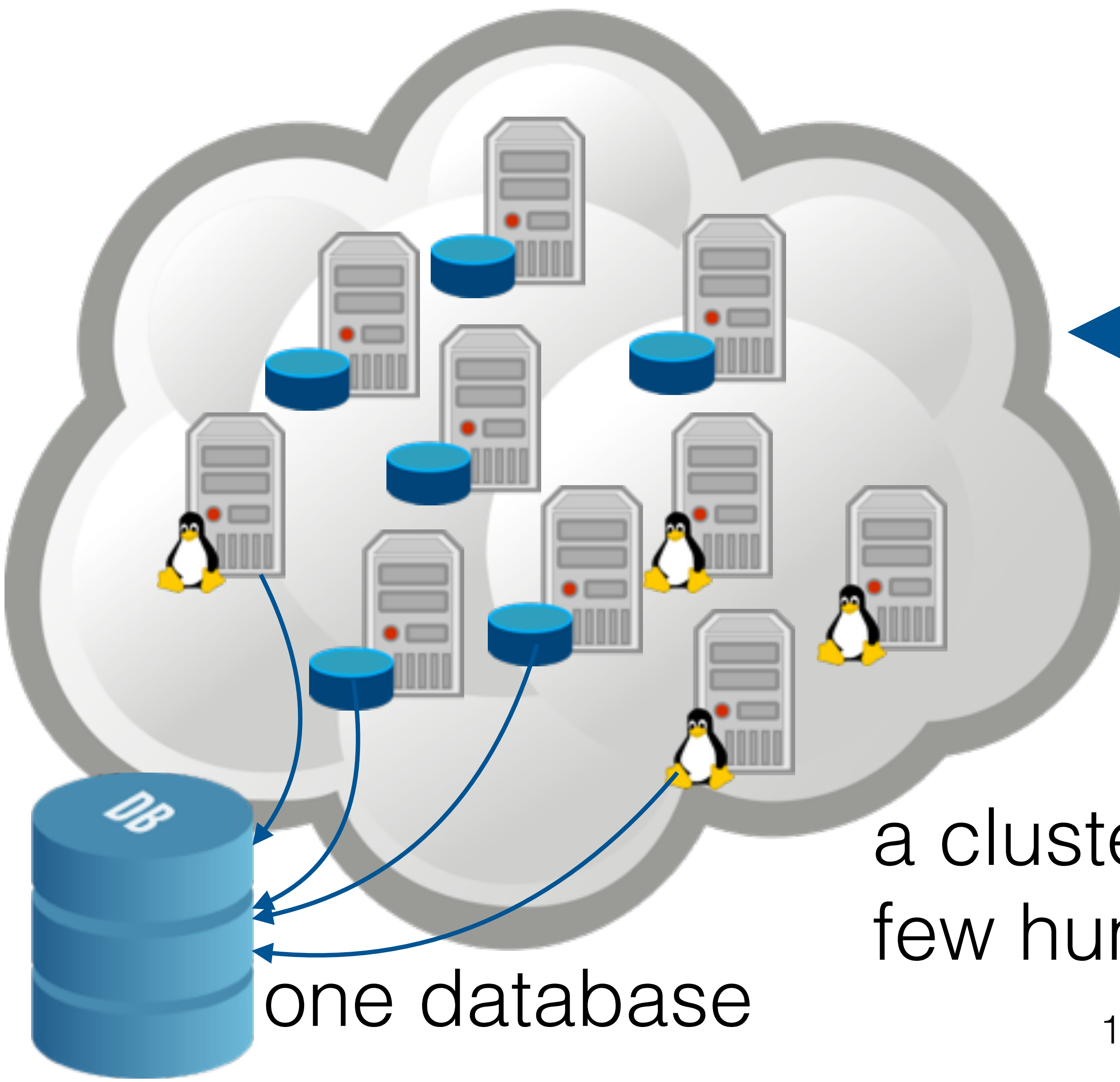
A program that crawls the Web

one machine (the leader) should coordinate the effort

coordination logic

a cluster with a few hundred machines

Question: how do you lock a service?



one database

a cluster with a few hundred machines

application logic

A program that crawls the Web

The progress of the crawl is stored in a DB: who accesses what & when?

coordination logic

Question: how can the configuration be distributed?

configuration file

application logic

A program that crawls the Web

Every worker should start with the same configuration

coordination logic

a cluster with a few hundred machines

Introduction contd.

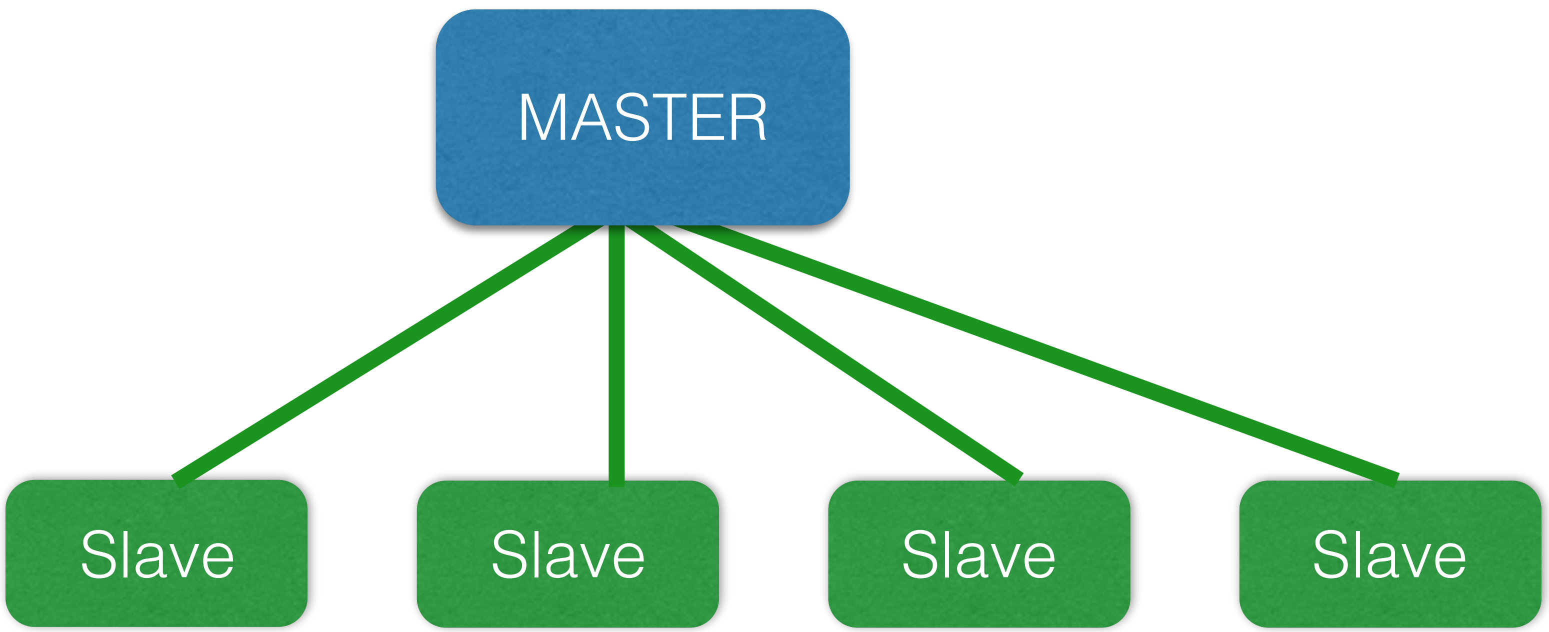
Solution approaches

- Be **specific**: develop a particular service for each coordination task
    - Locking service
    - Leader election
    - etc.

- Be **general**: provide an API to make many services possible

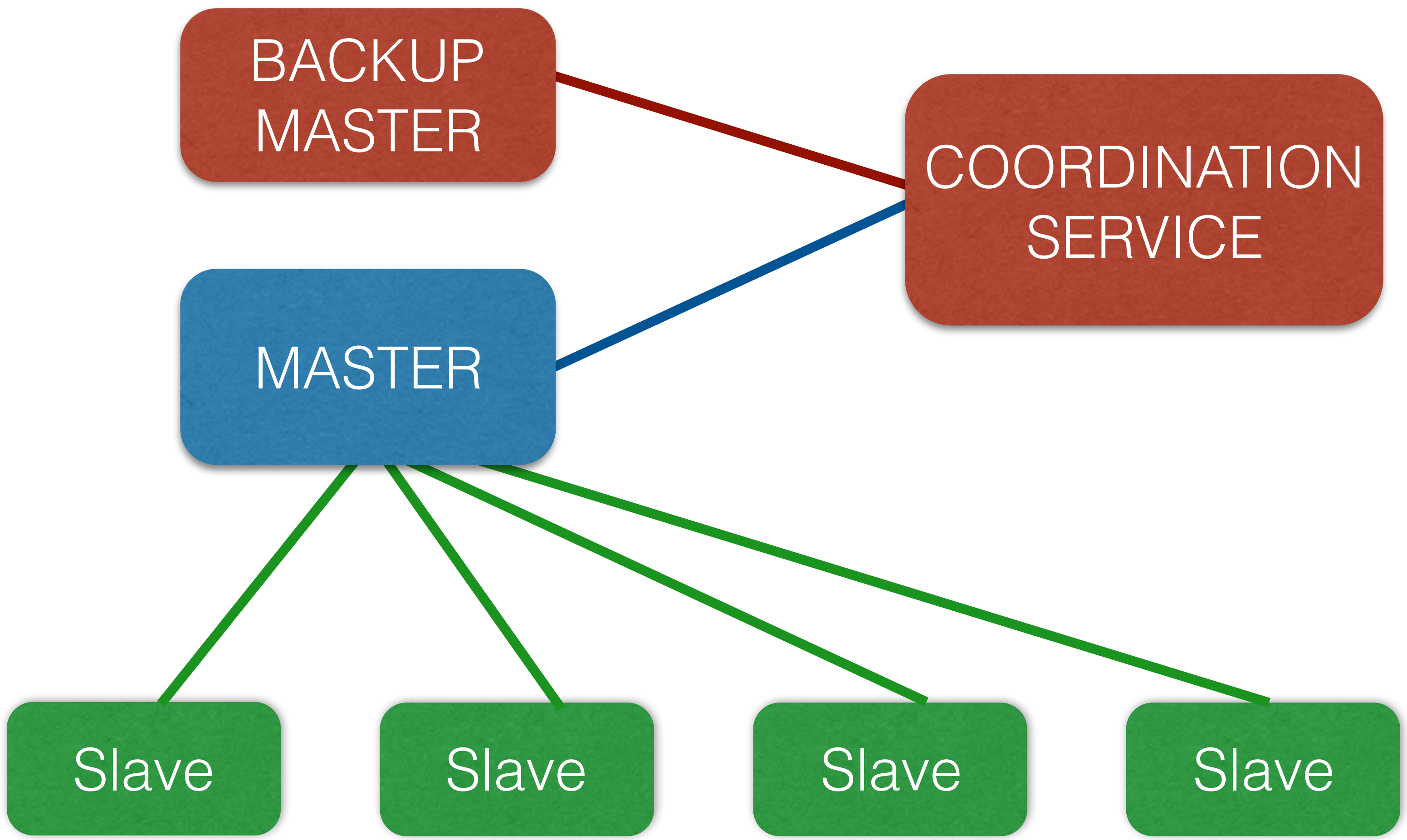
ZooKeeper	The Rest
API that enables application developers to **implement their own primitives easily**	specific primitives are implemented on the server side

How can a distributed system look like?



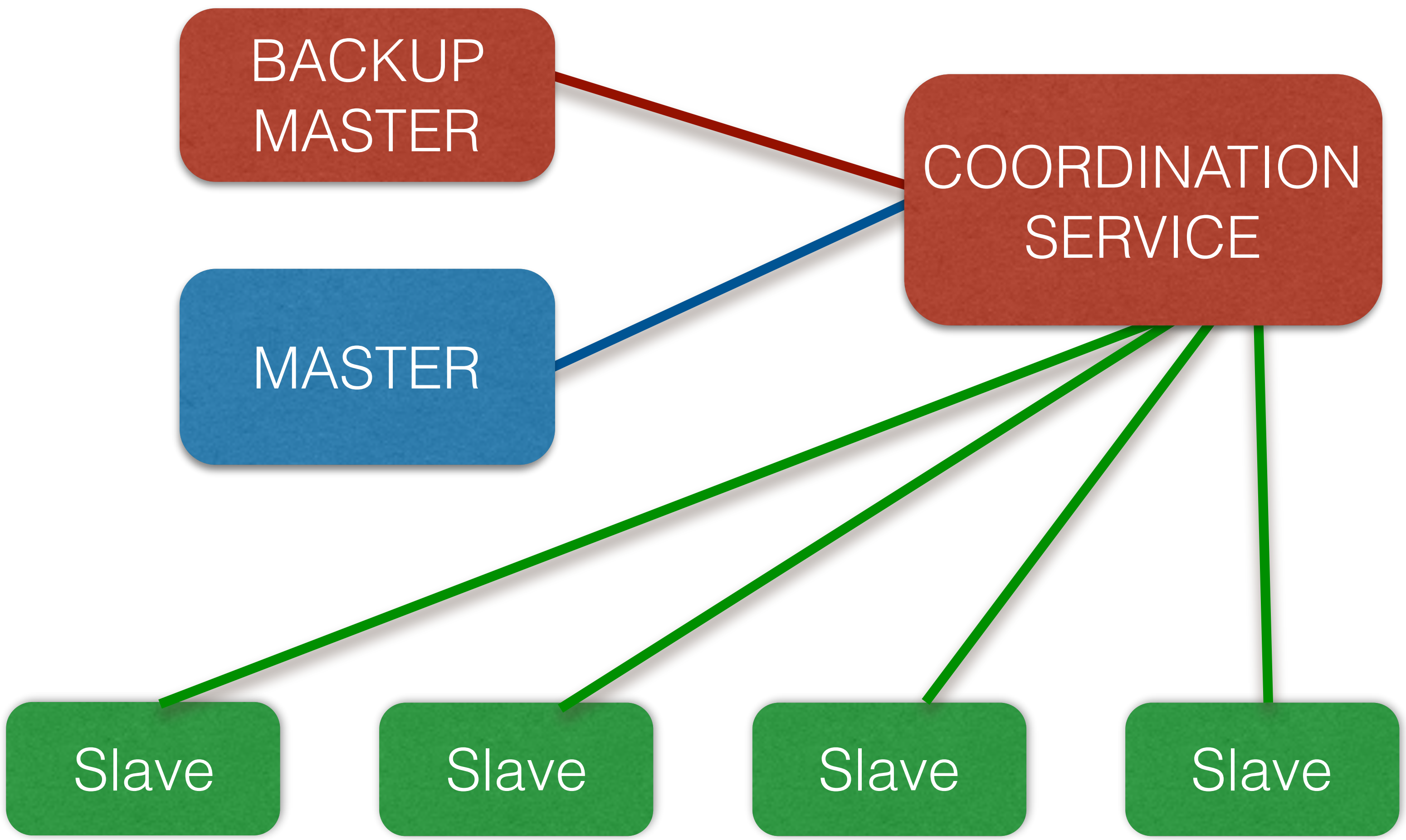
- \+ **simple**
- \- **coordination performed by the master**
- \- **single point of failure**
- \- **scalability**

How can a distributed system look like?



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

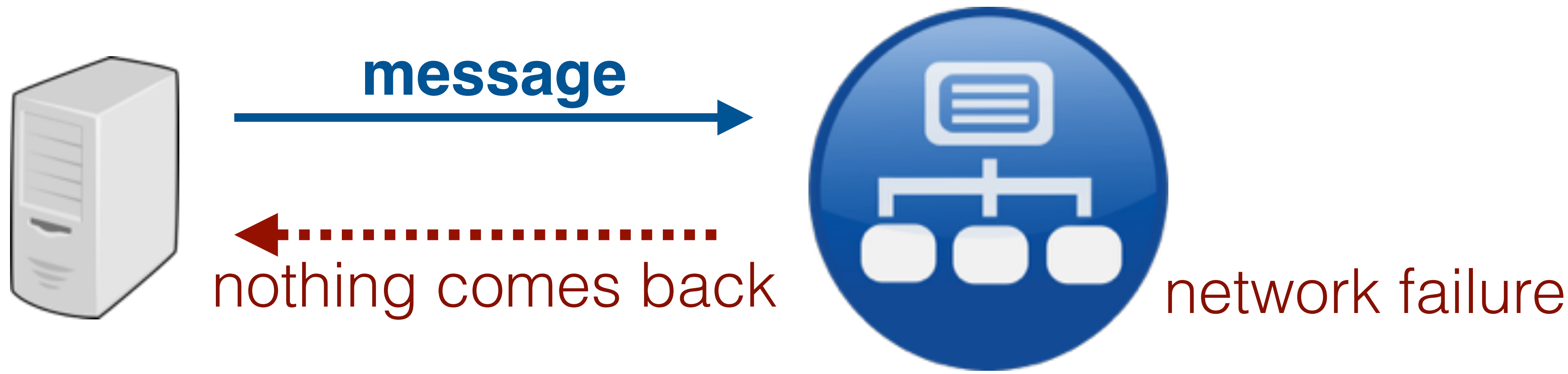
How can a distributed system look like?



+ scalability

What makes distributed system coordination difficult?

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

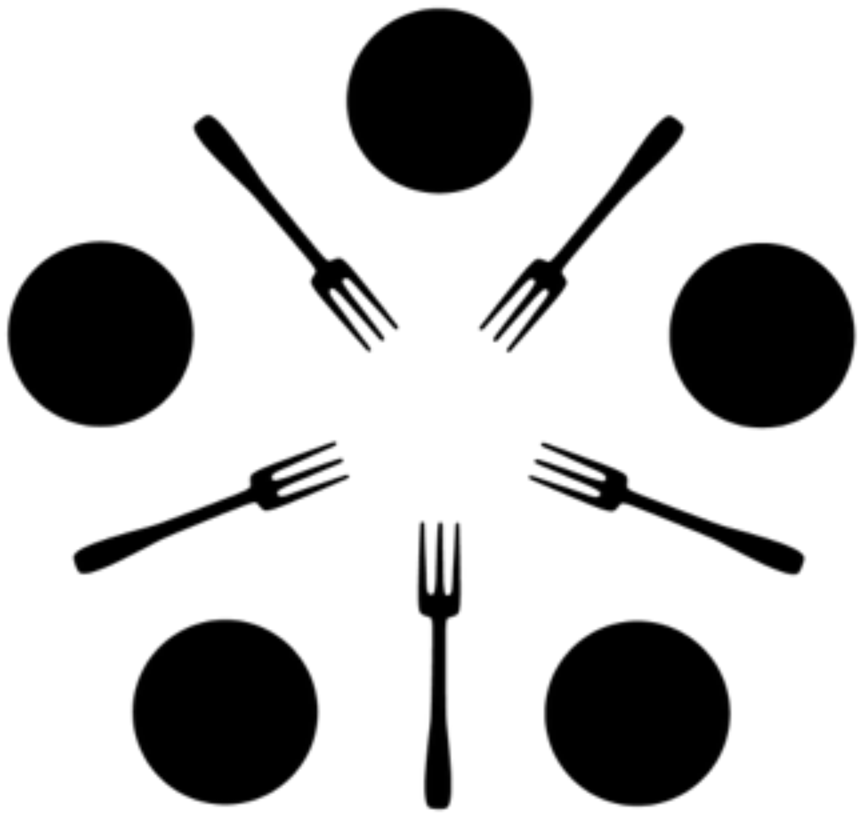
Typical coordination problems in distributed systems

- **Static configuration**: a list of operational parameters for the system processes
- **Dynamic configuration**: parameter changes on the fly
- **Group membership**: who is alive and part of the group?
- **Leader election**: who is in charge who is a backup?
- **Mutually exclusive access** to critical resources (locks)
- **Barriers** (supersteps in Giraph for instance)

The ZooKeeper API allows us to implement all these coordination tasks easily.

ZooKeeper principles

Dijkstra's dining philosophers

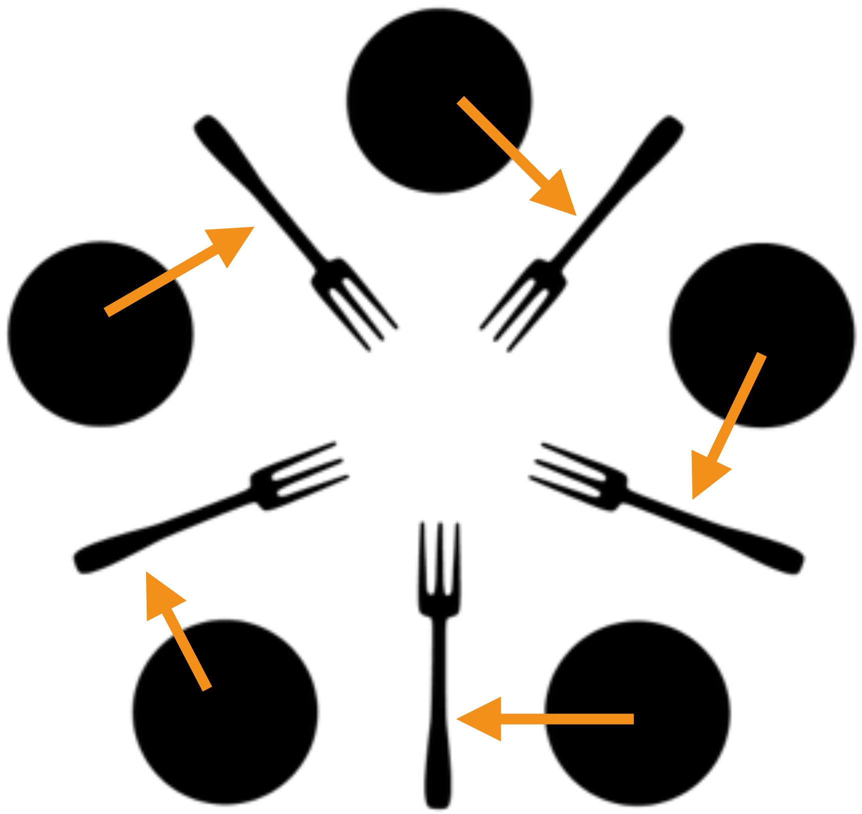


- N philosophers
- N forks
- To eat, both the left and right fork are needed
- Each philosopher can **pick up a fork** (if one is available), **eat** (when having both forks) and **think** (having no forks)

How do we ensure (=design an algorithm) that no philosopher starves?

Image source: http://zoo.cs.yale.edu/classes/cs422/2014fa/lectures/lec7.html#/25

Dijkstra's dining philosophers



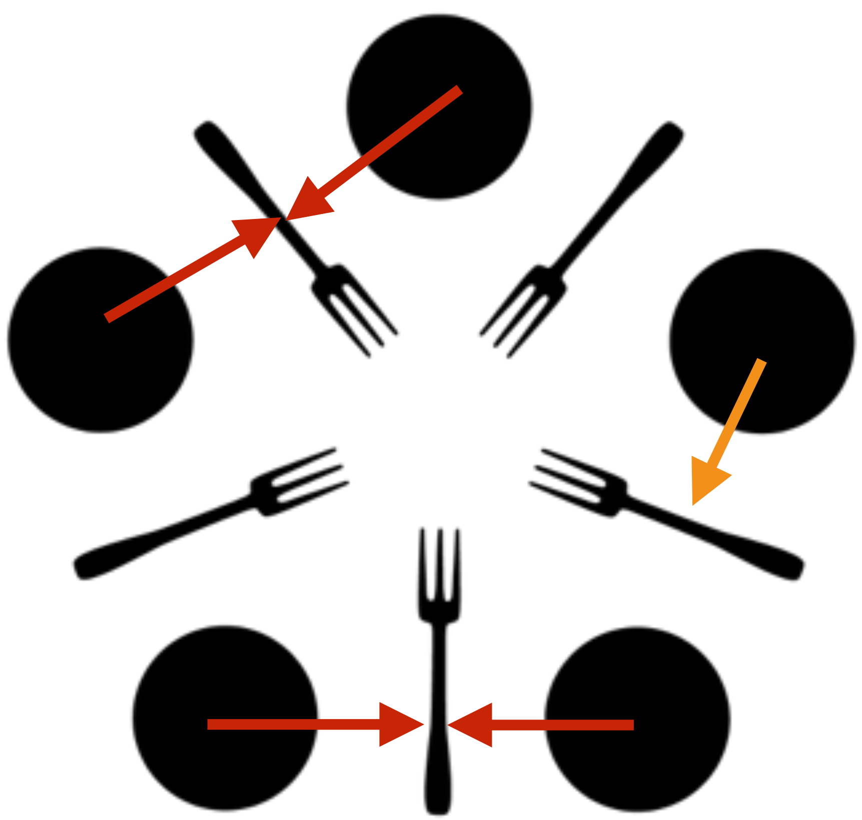
deadlock!

A simple algorithm:

- think until the left fork is available; pick it up
- think until the right fork is available; pick it up
- eat for time t when both forks are available
- after having eaten, put down the right fork
- put down the left fork
- repeat …

Image source: http://zoo.cs.yale.edu/classes/cs422/2014fa/lectures/lec7.html#/25

Dijkstra's dining philosophers



mutual exclusion!

A simple algorithm:

- think until the left fo… available; pick it …
- think until the … fork is available; … up
- eat for ti… when both forks are av…ble
- aft… …ving eaten, put down th… …ght fork
- put down the left fork
- repeat …

does not work!

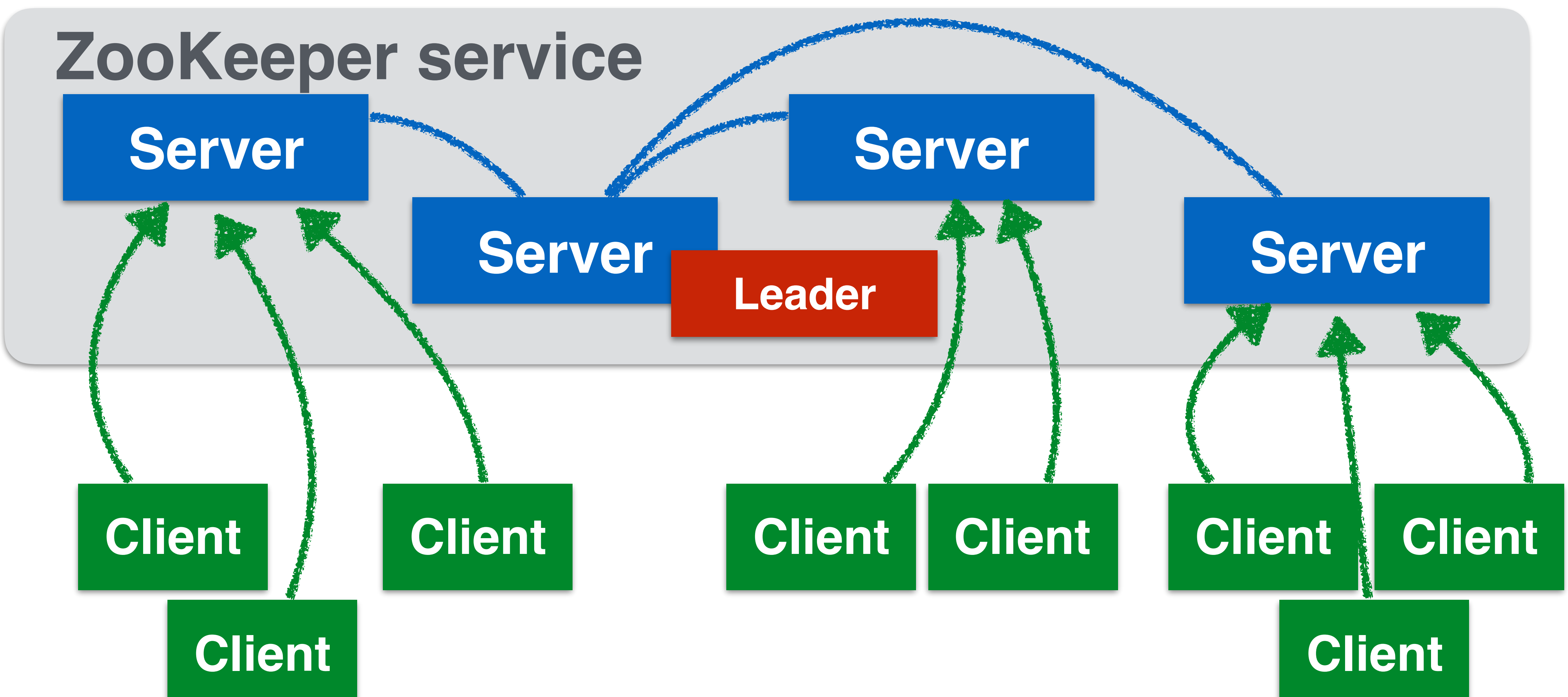
Image source: http://zoo.cs.yale.edu/classes/cs422/2014fa/lectures/lec7.html#/25

ZooKeeper's 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 **linearisable**
- Clients receive **notifications of changes** before the changed data becomes visible

ZooKeeper's strategy to be fast and reliable

- ZooKeeper service is an **ensemble of servers** that use replication (high availability)



ZooKeeper’s strategy to be 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

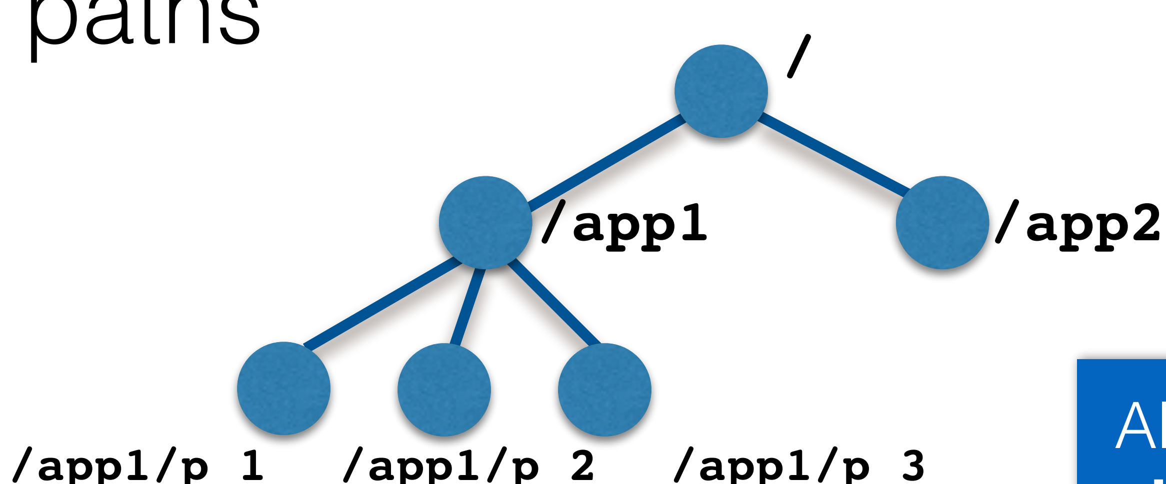
> ZooKeeper is optimised for read-dominant operations!

ZooKeeper terminology

- **Client**: user of the ZooKeeper service
- **Server**: process providing the ZooKeeper service
- **znode**: **in-memory** data node in ZooKeeper, organised in a hierarchical namespace (the **data tree**)
- **Update/write**: any operation which modifies the state of the data tree
- Clients establish a **session** when connecting to ZooKeeper

ZooKeeper's data model: filesystem

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



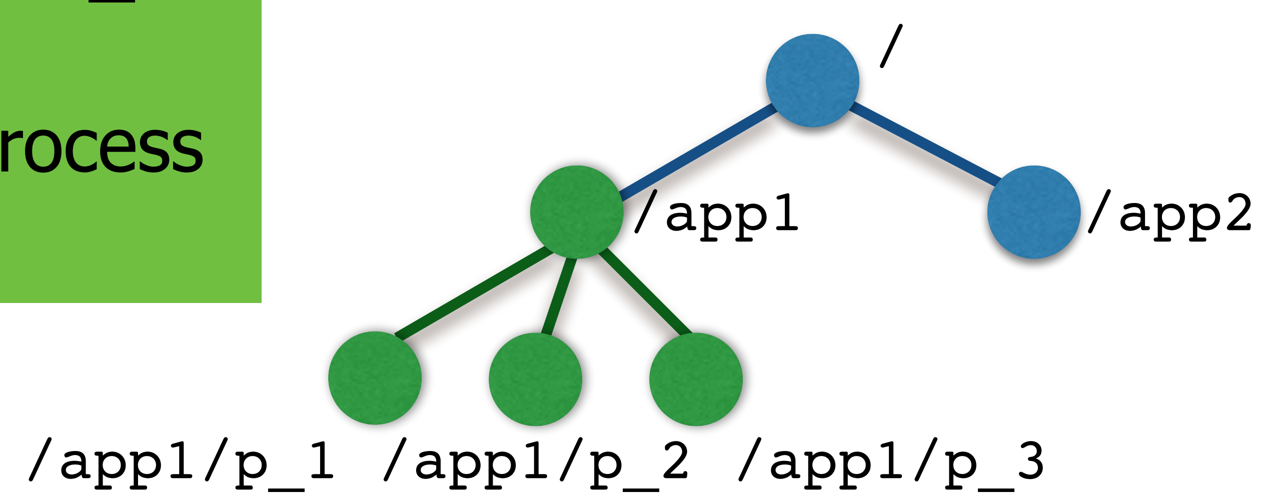
All znodes store **data (file like)** & can have **children (directory like)**.

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

Group membership protocol:
Client process `pi` creates znode `p_i` under `/app1`.
`/app1` persists as long as the process is running.



znode flags

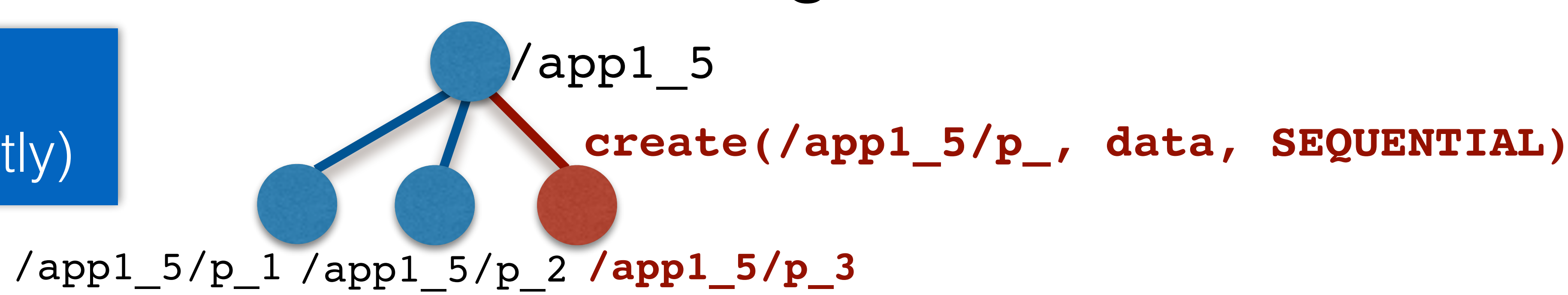
- Clients manipulate znodes by creating and deleting them

ephemeral (Greek): passing, short-lived

- **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

No flag: **regular znode**
(needs to be deleted explicitly)



znodes & watch flag

- Clients can issue read operations on znodes with a watch flag
- 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

Sessions

- A client connects to ZooKeeper and initiates a session
- 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 or explicitly ended by client

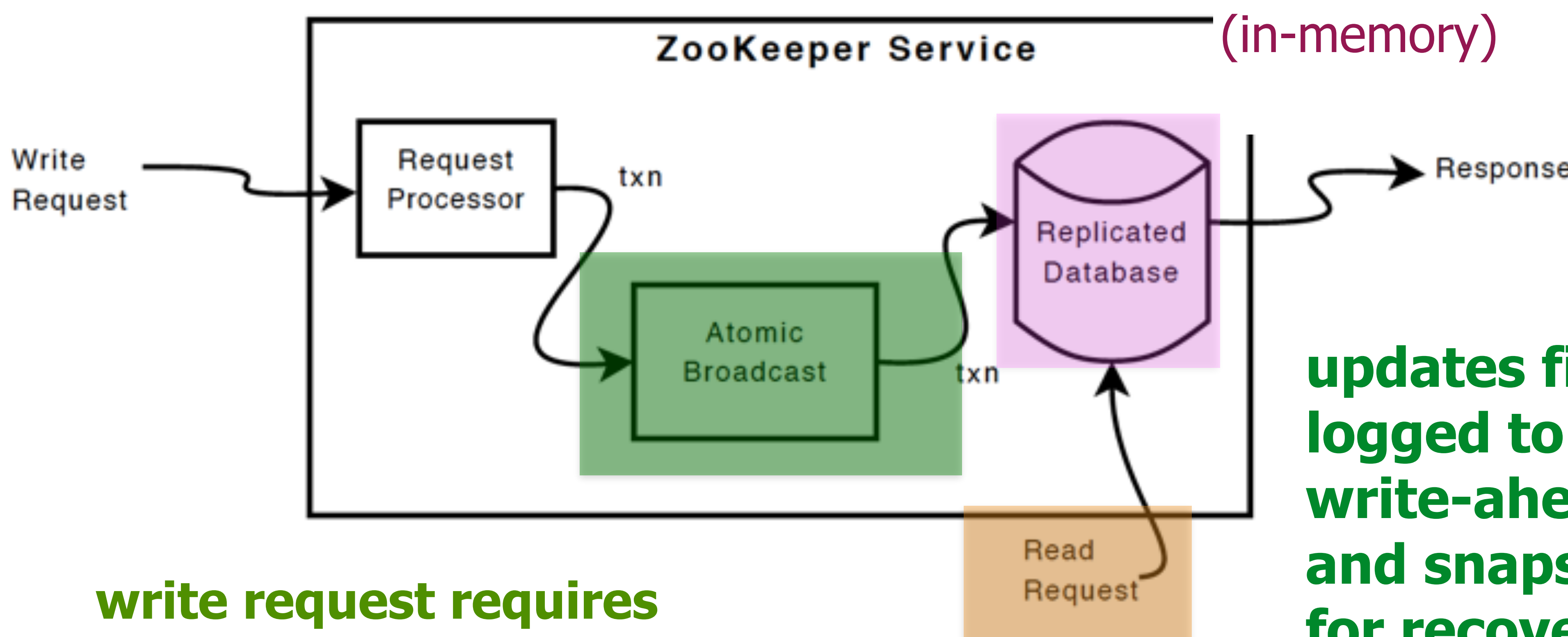
Versioning

- znodes have associated version counters
- Allow clients to execute conditional updates based on the version of the znode

A few implementation details

ZooKeeper data is replicated on each server that makes up the service

replicated across all servers (in-memory)



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

write request requires coordination between ZooKeeper servers (leader coordinates)

read requests served from individual server

Source: http://bit.ly/13VFohW

A few 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)

Lets work through some examples

ZooKeeper API

- `String create(path, data, flags)`
  - creates a znode with path name path, stores data in it and sets flags (ephemeral, sequential)
  - returns the name of the new znode
- `void delete(path, version)`
  - deletes the anode if it is at the expected version
- `Stat exists(path, watch)`
  - watch flag enables the client to set a watch on the znode
- `(data, Stat) getData(path, watch)`
  - returns the data and meta-data of the znode
- `Stat setData(path, data, version)`
  - writes data if the version number is the current version of the znode
- `String[] getChildren(path, watch)`

No partial read/writes (no **`open`**, **`seek`** or **`close`** methods). **Absolute path names** expected.

All methods exist as **synchronous** and as **asynchronous** versions.

`version` **set to -1**: execute method independent of current version

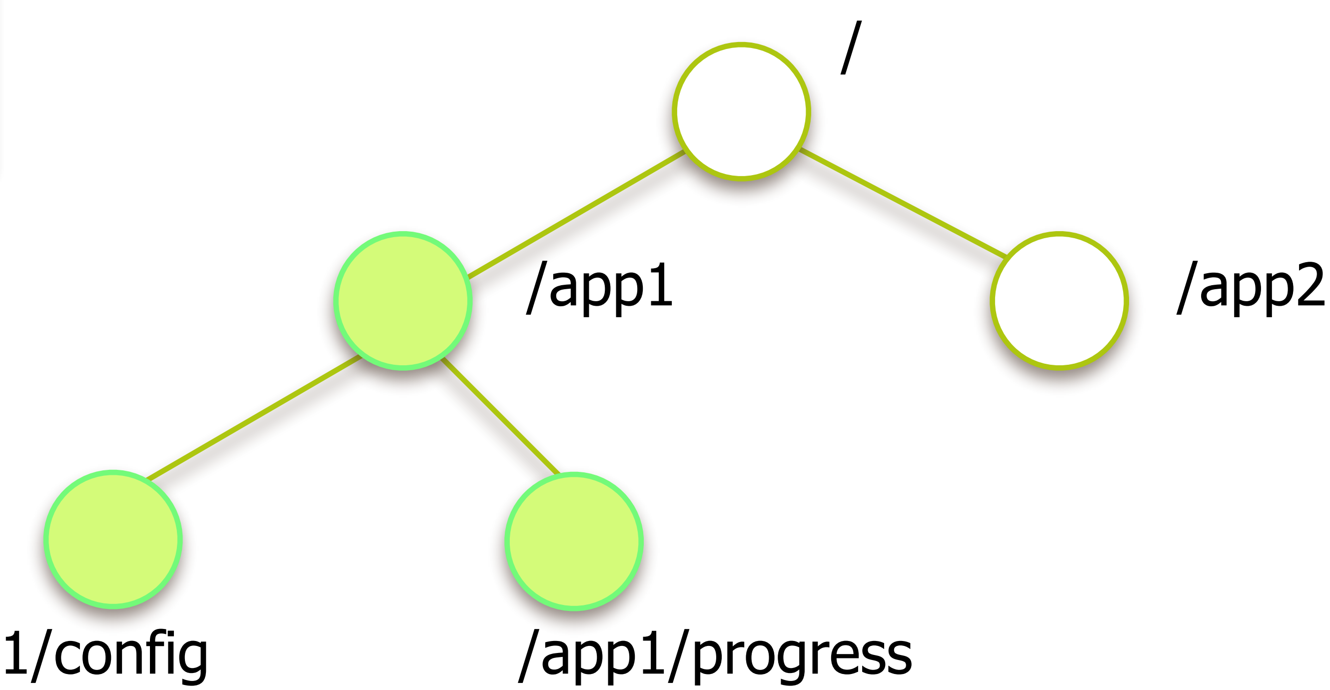
No `createLock()` or similar methods.

Example: configuration

- `String create(path, data, flags)`
- `void delete(path, version)`
- `Stat exists(path, watch)`
- `(data, Stat) getData(path, watch)`
- `Stat setData(path, data, version)`
- `String[] getChildren(path, watch)`

Questions:
1. How does a **new** worker query ZK for a configuration?
2. How does an administrator **change** the configuration **on the fly**?
3. How do the workers read the **new** configuration?

```
[configuration stored in /app1/config]
1. getData(/app1/config,true)
2. setData(/app1/config/config_data,-1)
[notify watching clients]
3. getData(/app1/config,true)
```

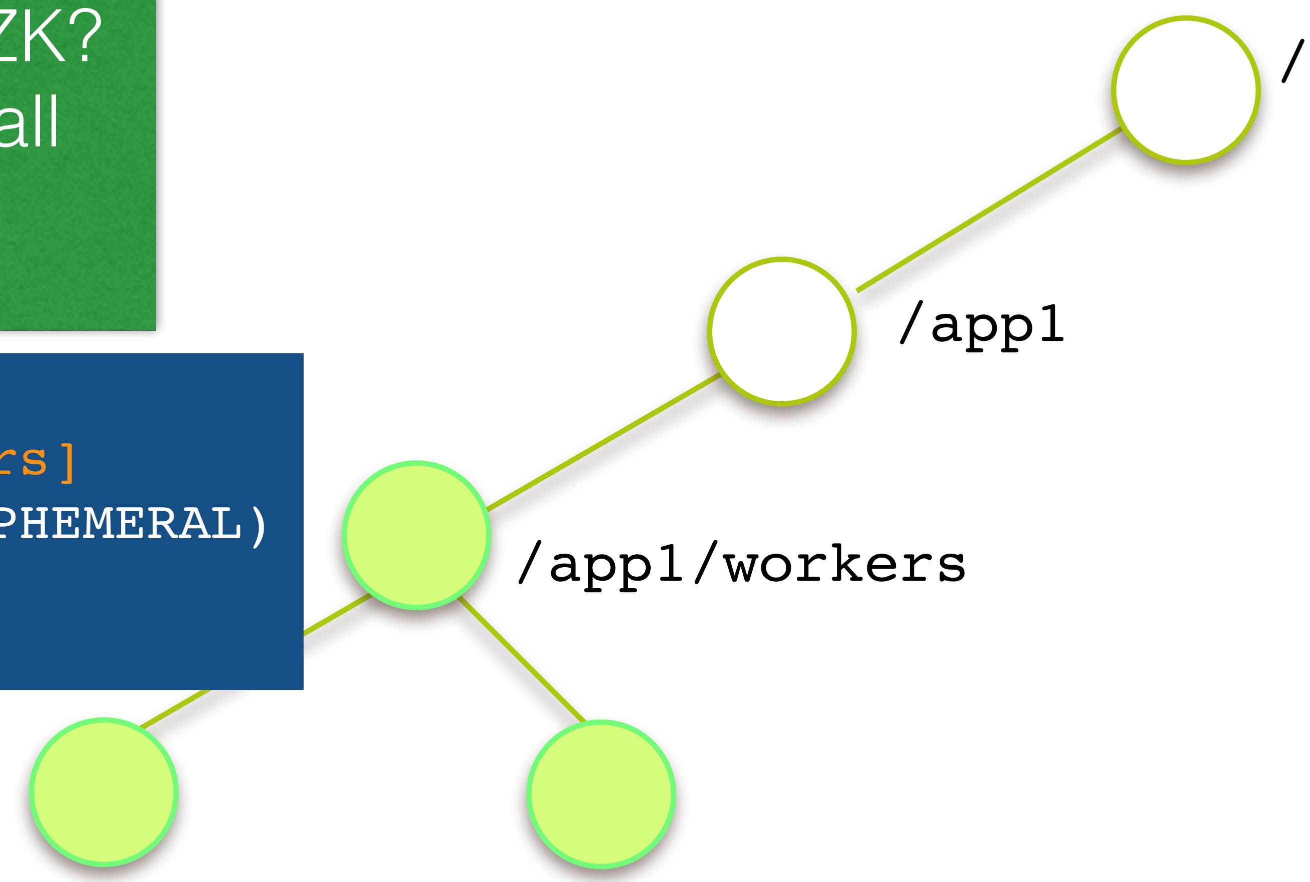


Example: group membership

```
• String create(path, data, flags)
• void delete(path, version)
• Stat exists(path, watch)
• (data, Stat) getData(path, watch)
• Stat setData(path, data, version)
• String[] getChildren(path, watch)
```

Questions:
1. How can all workers (slaves) of an application **register themselves** on ZK?
2. How can a process find out about all **active** workers of an application?

```
[a znode is designated to store workers]
1. create(/app1/workers/worker,data,EPHEMERAL)
2. getChildren(/app1/workers,true)
```

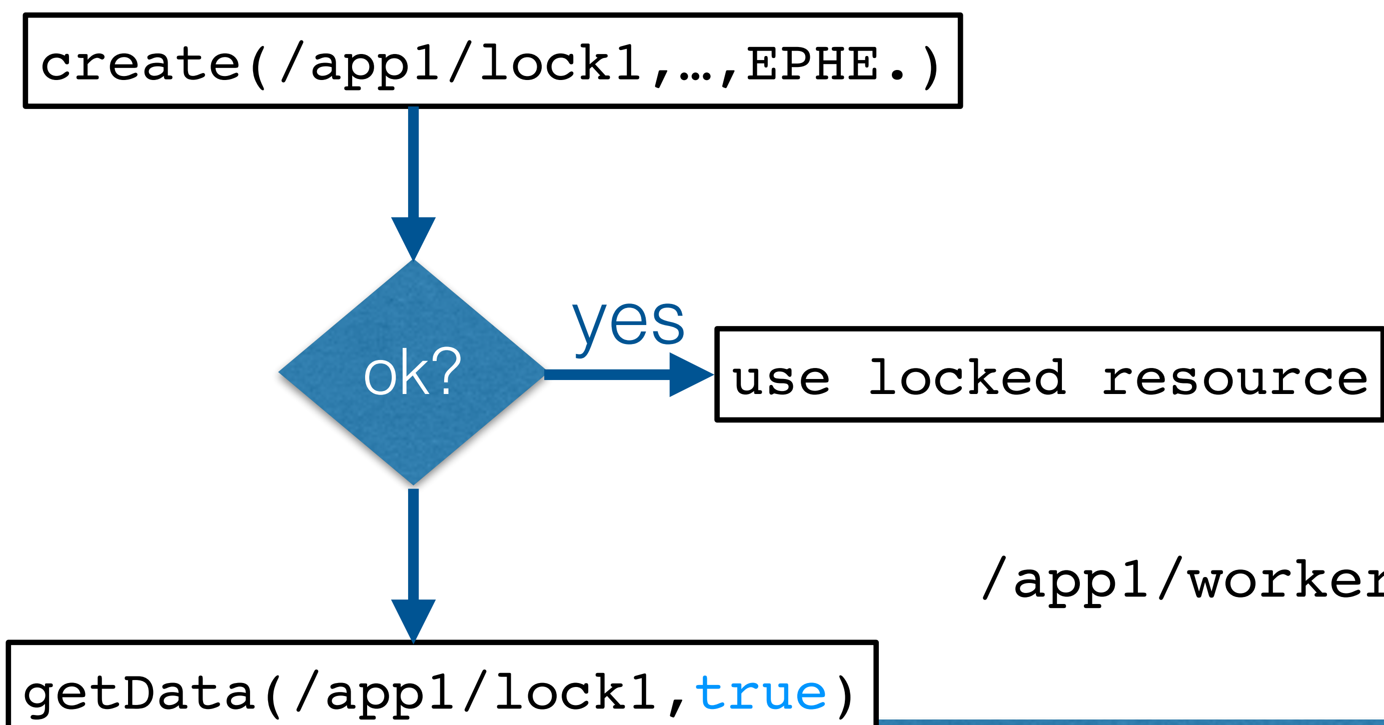


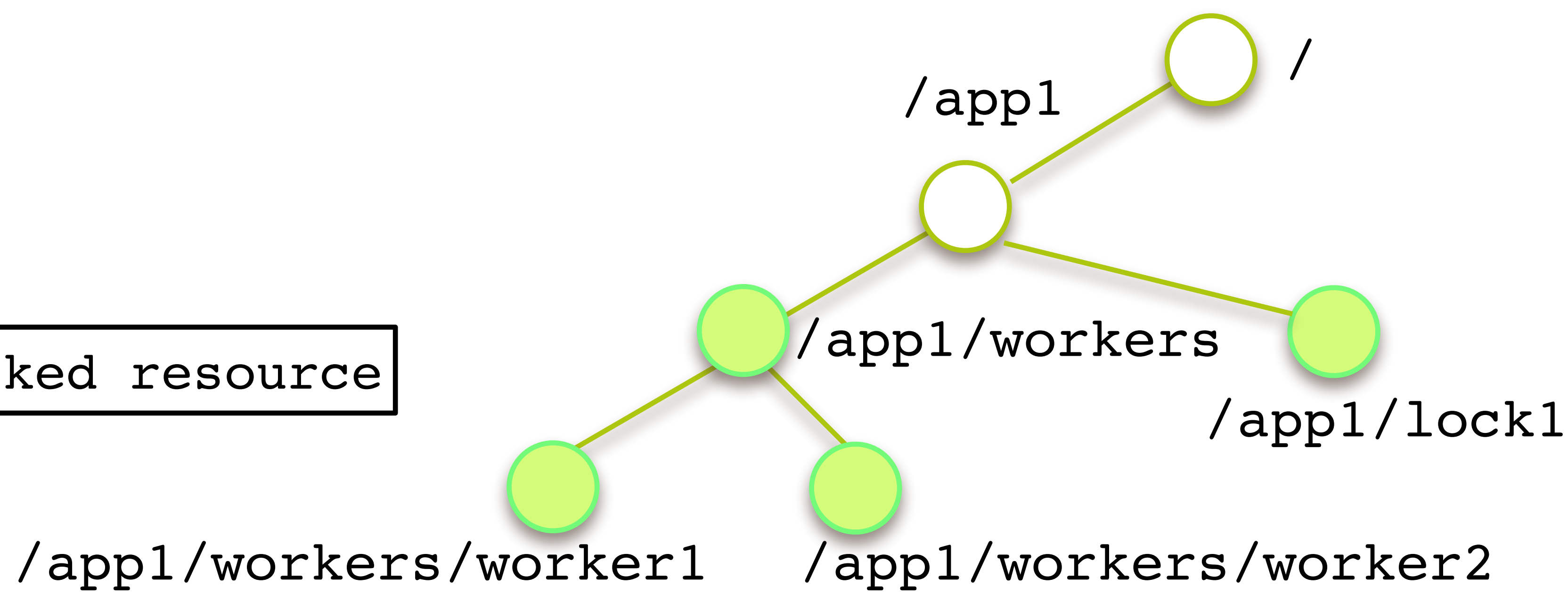
Example: simple locks

- `String create(path, data, flags)`
- `void delete(path, version)`
- `Stat exists(path, watch)`
- `(data, Stat) getData(path, watch)`
- `Stat setData(path, data, version)`
- `String[] getChildren(path, watch)`

a database, ….

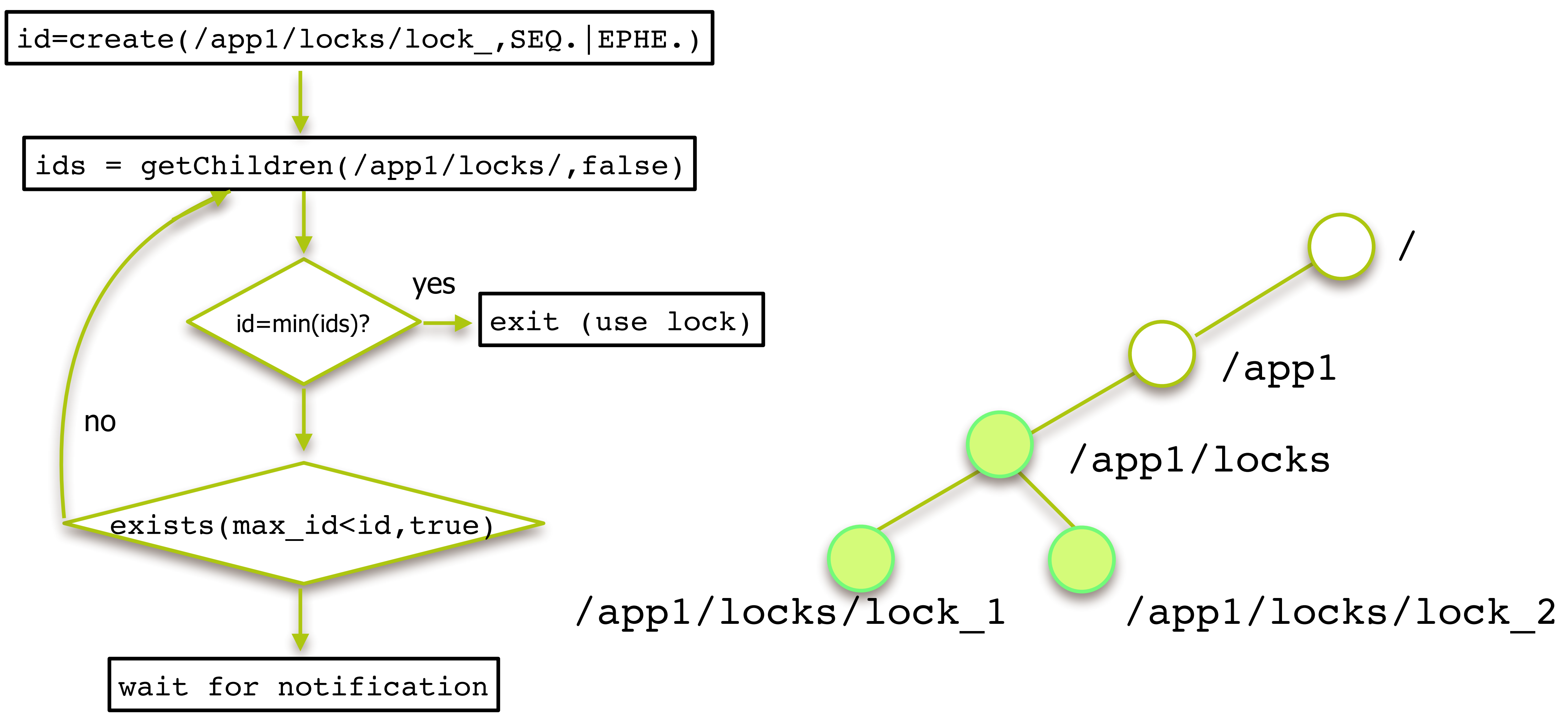
Question:
How can all workers of an application use a single resource through a **lock**?





all processes compete at all times for the lock

Example:
locking without herd effect

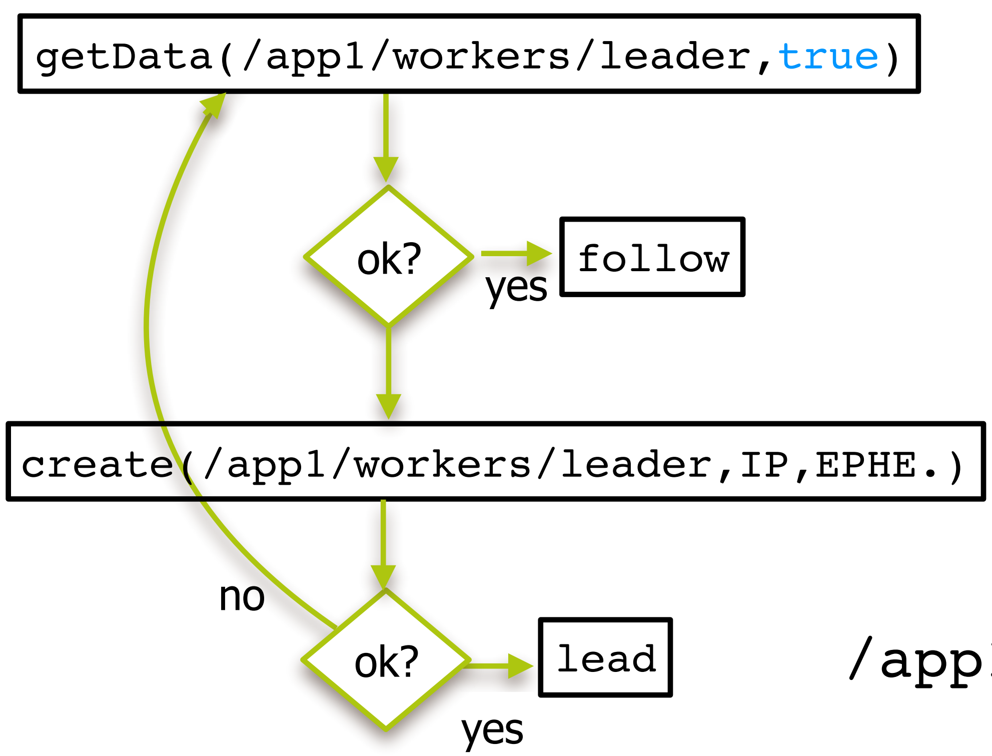


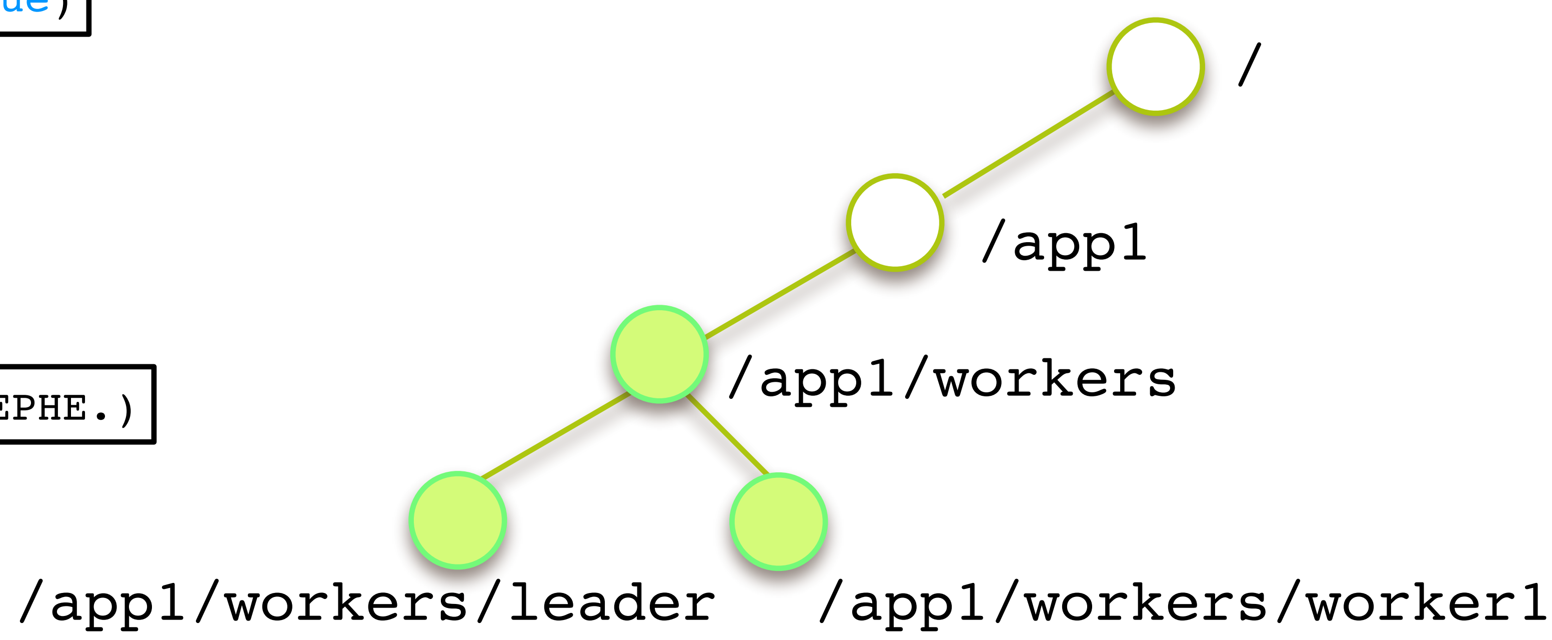
Question:
How can all workers of an application use a single resource through a lock?

Example: leader election

- `String create(path, data, flags)`
- `void delete(path, version)`
- `Stat exists(path, watch)`
- `(data, Stat) getData(path, watch)`
- `Stat setData(path, data, version)`
- `String[] getChildren(path, watch)`

Question:
How can all workers of an application elect a leader among themselves?

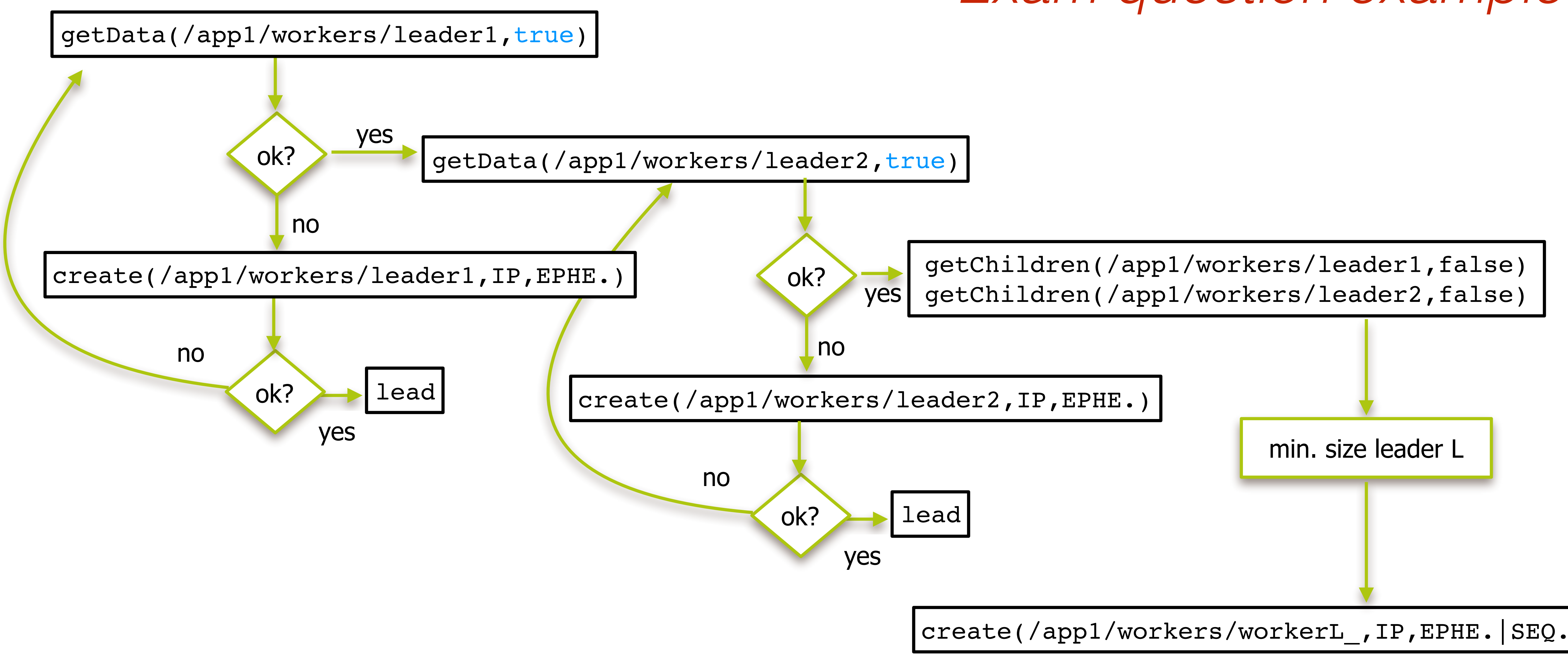




if the leader dies, elect again ("herd effect")

(3 points) **[ZooKeeper]** Describe how to implement the following coordination tasks using the ZooKeeper API: a program (e.g. a distributed WebCrawler) is assigned a number of workers in the cluster at program startup. These workers need to elect two leaders among themselves that both are responsible to assign tasks to the remaining workers. If one or both leaders crash, new leaders need to be chosen. Workers that are not elected as leaders should assign themselves to one of the two leaders in such a way that each leader is followed by roughly half the workers.

Exam question example



Example: rendezvous

- Scenario: a client wants to start a master and some worker processes
- Processes are started by a **scheduler**, the client does not know the IP addresses of the processes in advance
- **Rendezvous znode** z is created by the client
- Full pathname of z is passed as **startup parameter** of the master/worker processes
- Master starts and updates z with its IP/port information

ZooKeeper applications

The Yahoo! fetching service

- Fetching Service is part of Yahoo!'s crawler infrastructure
- Setup: master commands page-fetching processes
  - Master provides the fetchers with configuration
  - Fetchers write back information of their status and health
- Main advantage of ZooKeeper:
  - **Recovery** from master **failures**
  - Guaranteed **availability** despite failures
- Used primitives of ZK: configuration metadata, leader election

Yahoo! message broker

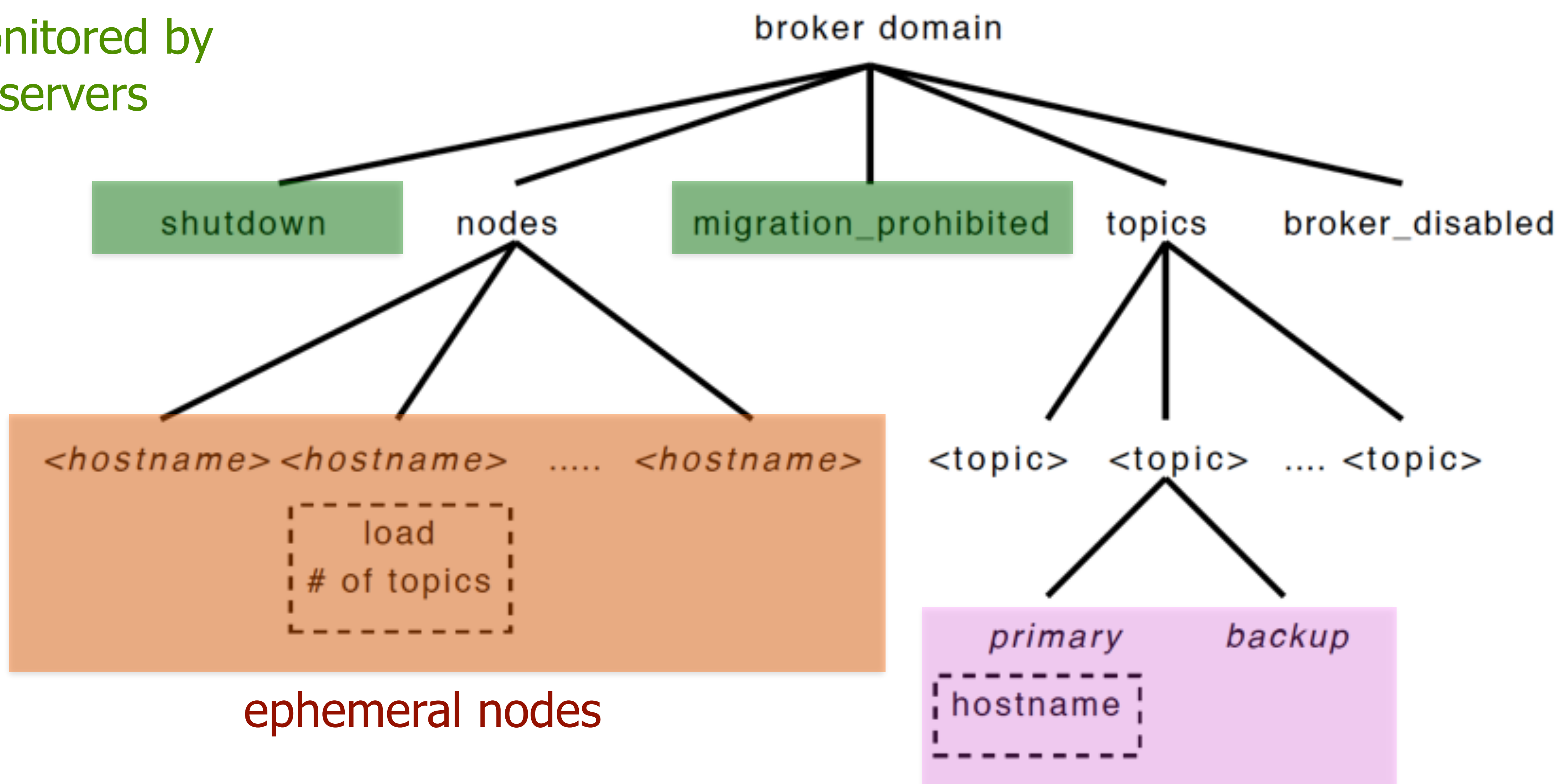
- A distributed publish-subscribe system
- The system manages thousands of topics that clients can publish messages to and receive messages from
- The topics are distributed among a set of servers to provide scalability
- Used primitives of ZK: configuration metadata (to distribute topics), failure detection and group membership

Yahoo! message broker

ZooKeeper filesystem setup

primary and backup
server per topic;
topic subscribers

monitored by
all servers

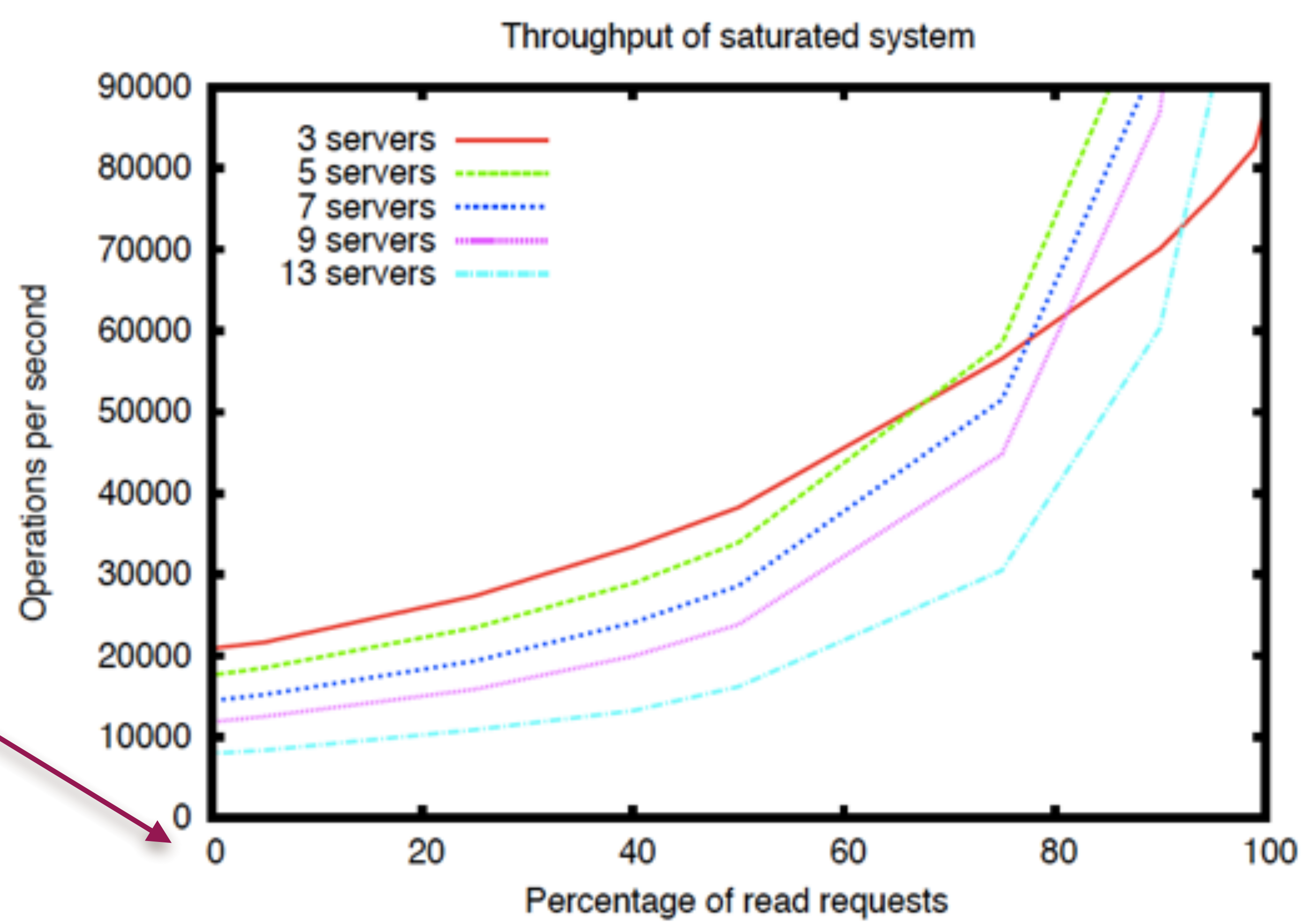


Source: http://bit.ly/13VFohW

Throughput

Setup: 250 clients, each client has at least 100 outstanding requests (read/write of 1K data)

only write requests

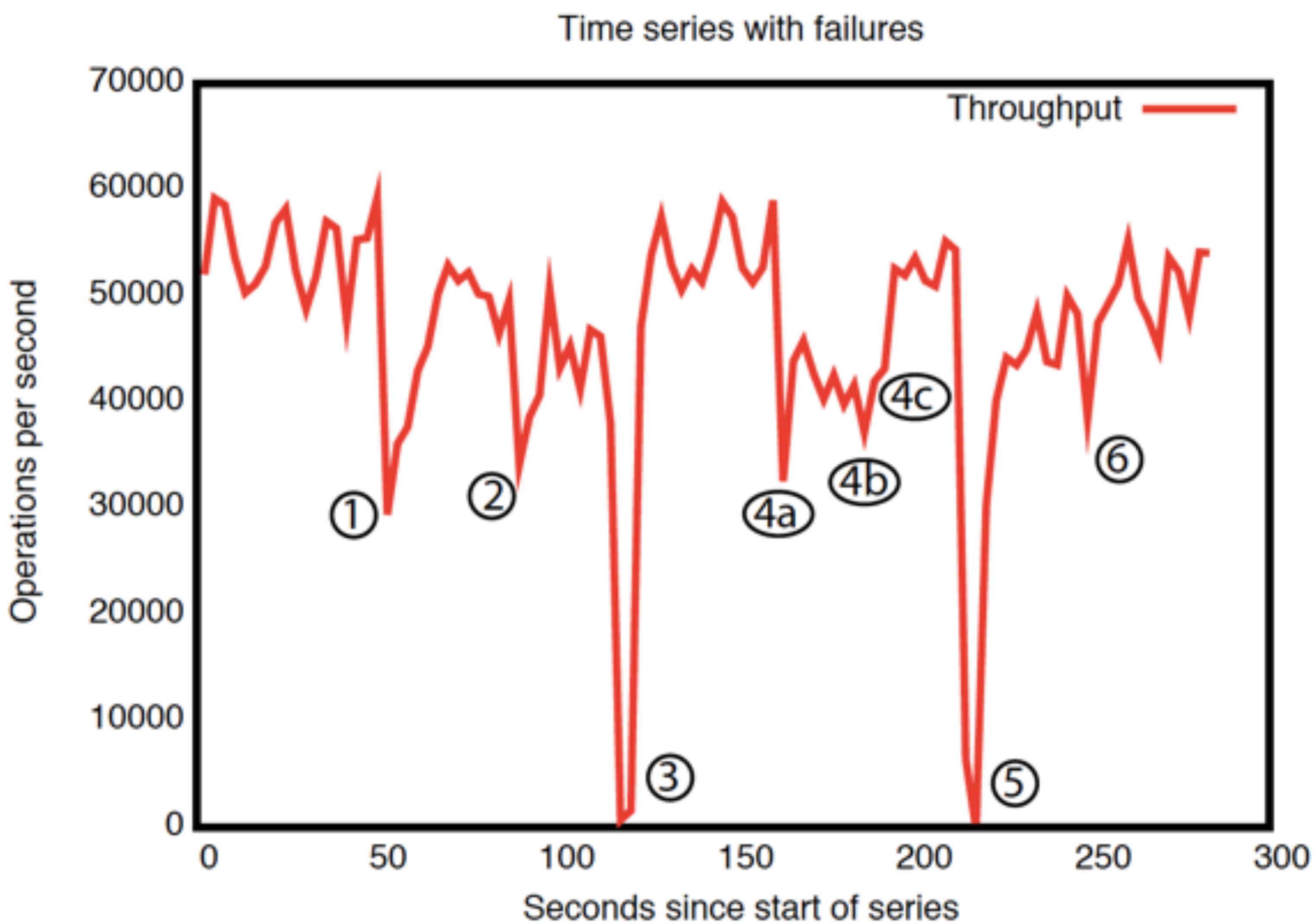


only read requests

Source: http://bit.ly/13VFohW

Recovery from failure

Setup: 250 clients, each client has at least 100 outstanding requests (read/write of 1K data);
5 ZK machines (1 leader, 4 followers), 30% writes



(1) failure & recovery of a follower
(2) failure & recovery of a different follower
(3) failure of the leader
(4) failure of followers (a,b), recovery at (c)
(5) failure of the leader
(6) recovery of the leader

Source: http://bit.ly/13VFohW

References

- [book] *ZooKeeper* by Junqueira & Reed, 2013 (available on the TUD campus network)

- [paper] *ZooKeeper: Wait-free coordination for Internet-scale systems* by Hunt et al., 2010; http://bit.ly/13VFohW

Summary

- Whirlwind tour of ZooKeeper
- Why do we need it?
- Data model of ZooKeeper: znodes
- Example implementations of different coordination tasks