

Lecture 14: ZooKeeper

Claudia Hauff (Web Information Systems)
ti2736b-ewi@tudelft.nl



Course content

- Introduction
- Data streams 1 & 2
- The MapReduce paradigm
- Looking behind the scenes of MapReduce: HDFS & Scheduling
- Algorithm design for MapReduce
- A high-level language for MapReduce: Pig Latin 1 & 2
- MapReduce is not a database, but HBase nearly is
- Lets iterate a bit: Graph algorithms & Giraph
- **Coordination in distributed systems**

Learning objectives

- Place ZooKeeper in the Hadoop ecosystem
- **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

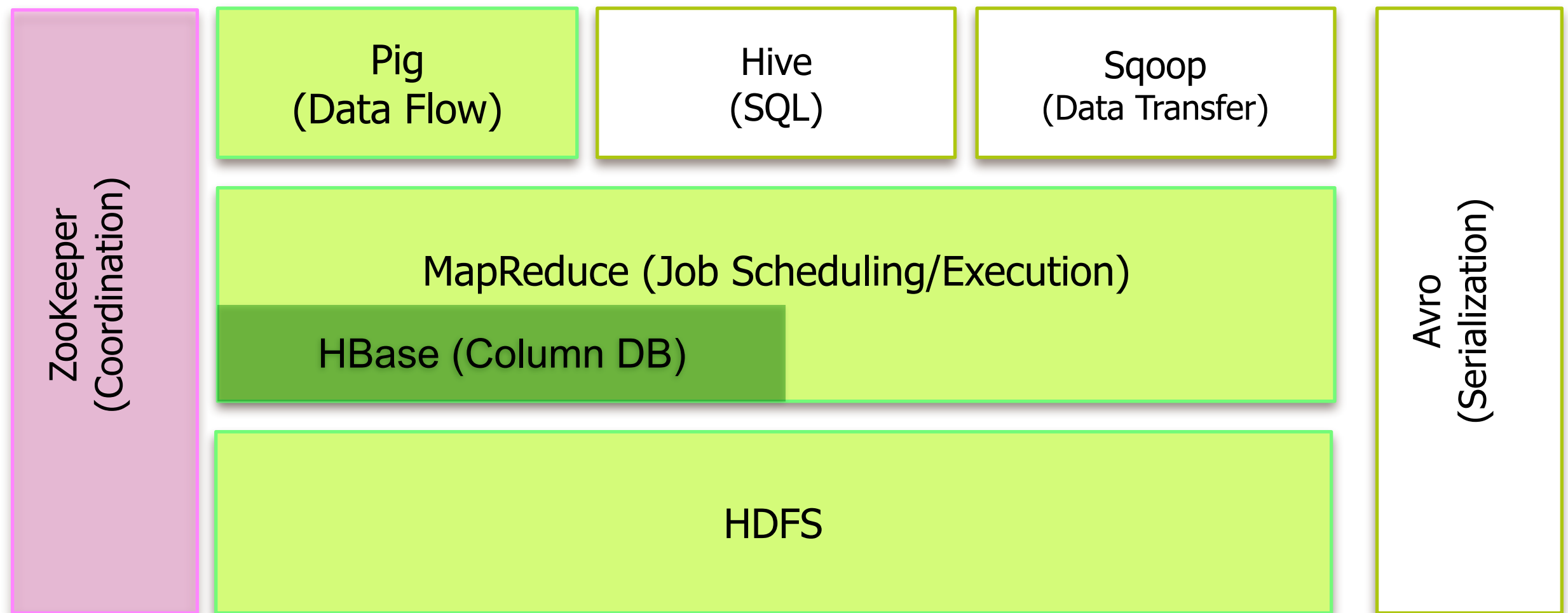
Introduction

ZooKeeper

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

- Developed at Yahoo! Research
- Started as sub-project of Hadoop, now a top-level Apache project
- Development is driven by application needs

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
- ...

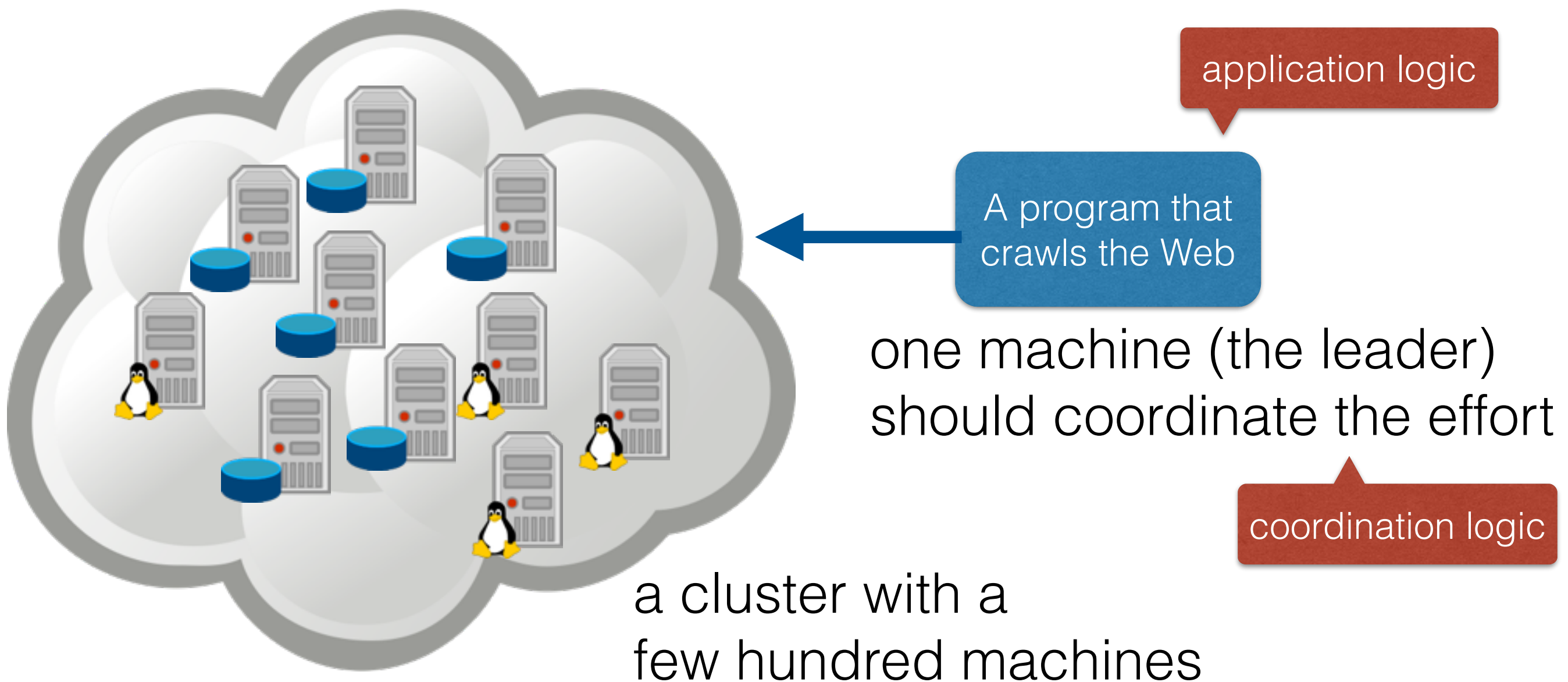
Motivation

- 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

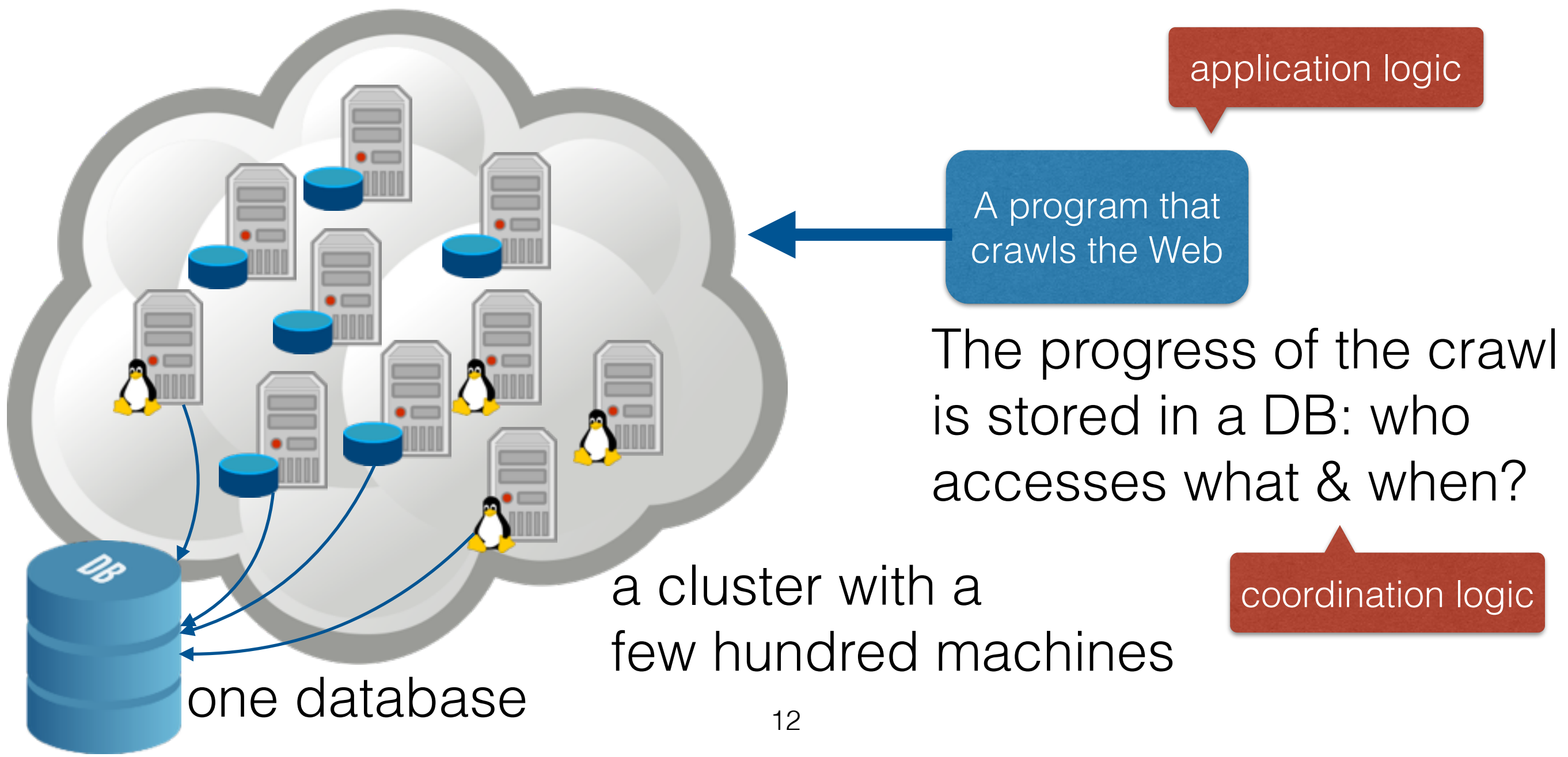
ZooKeeper: **designed** to relieve developers from writing coordination logic code.

Lets think

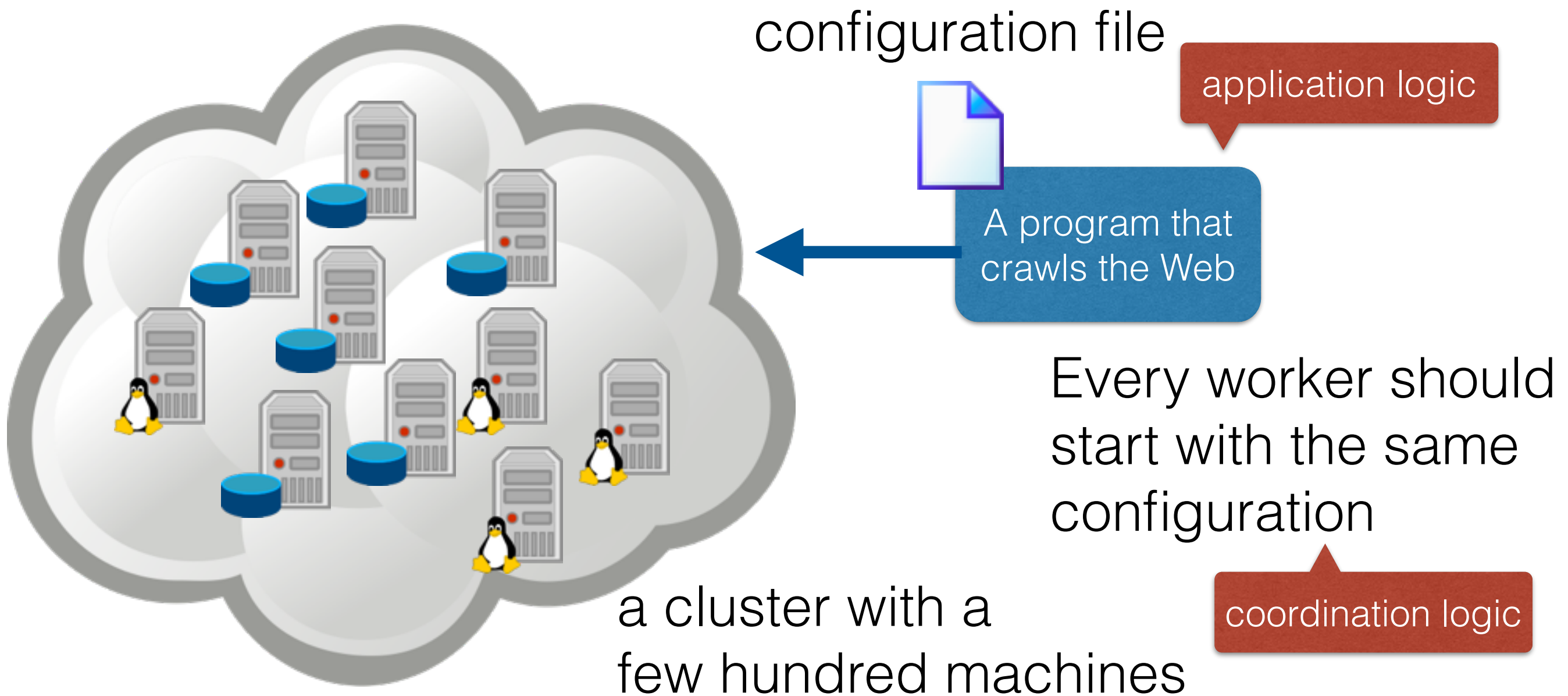
Question: how do you elect the leader?



Question: how do you lock a service?



Question: how can the configuration be distributed?



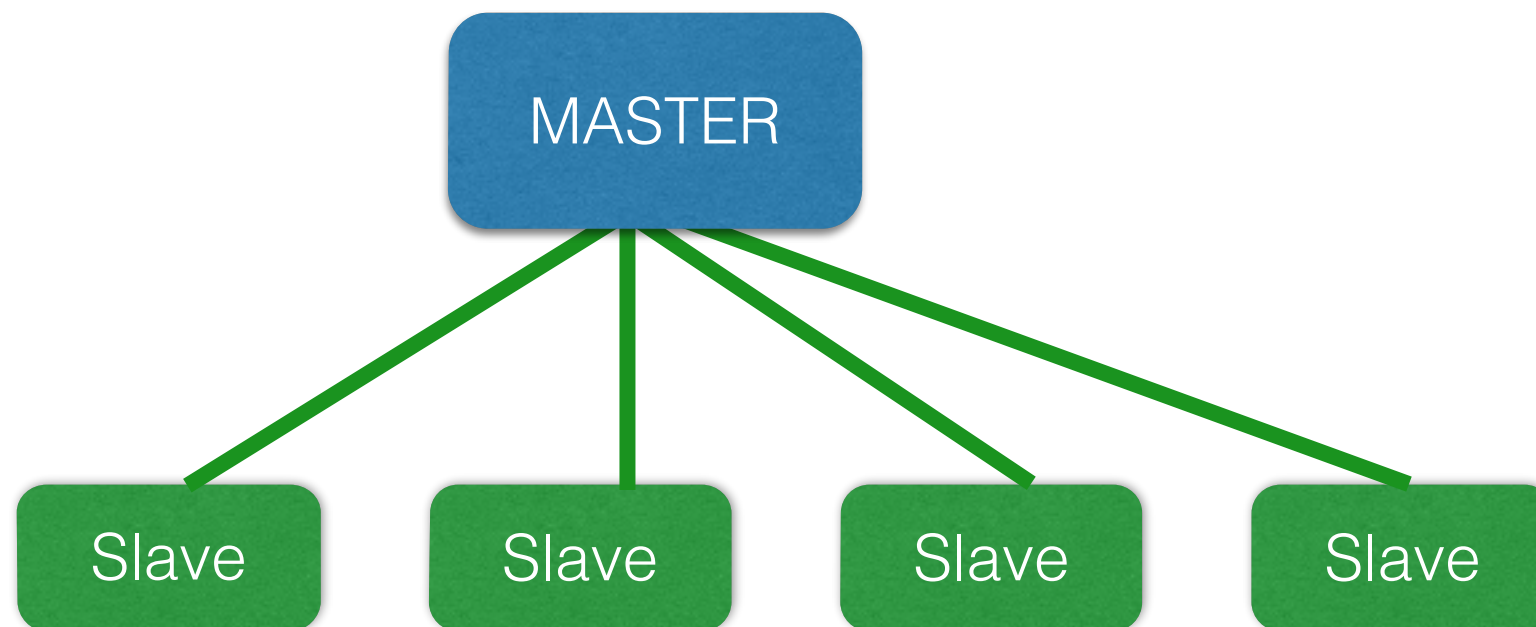
Introduction contd.

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

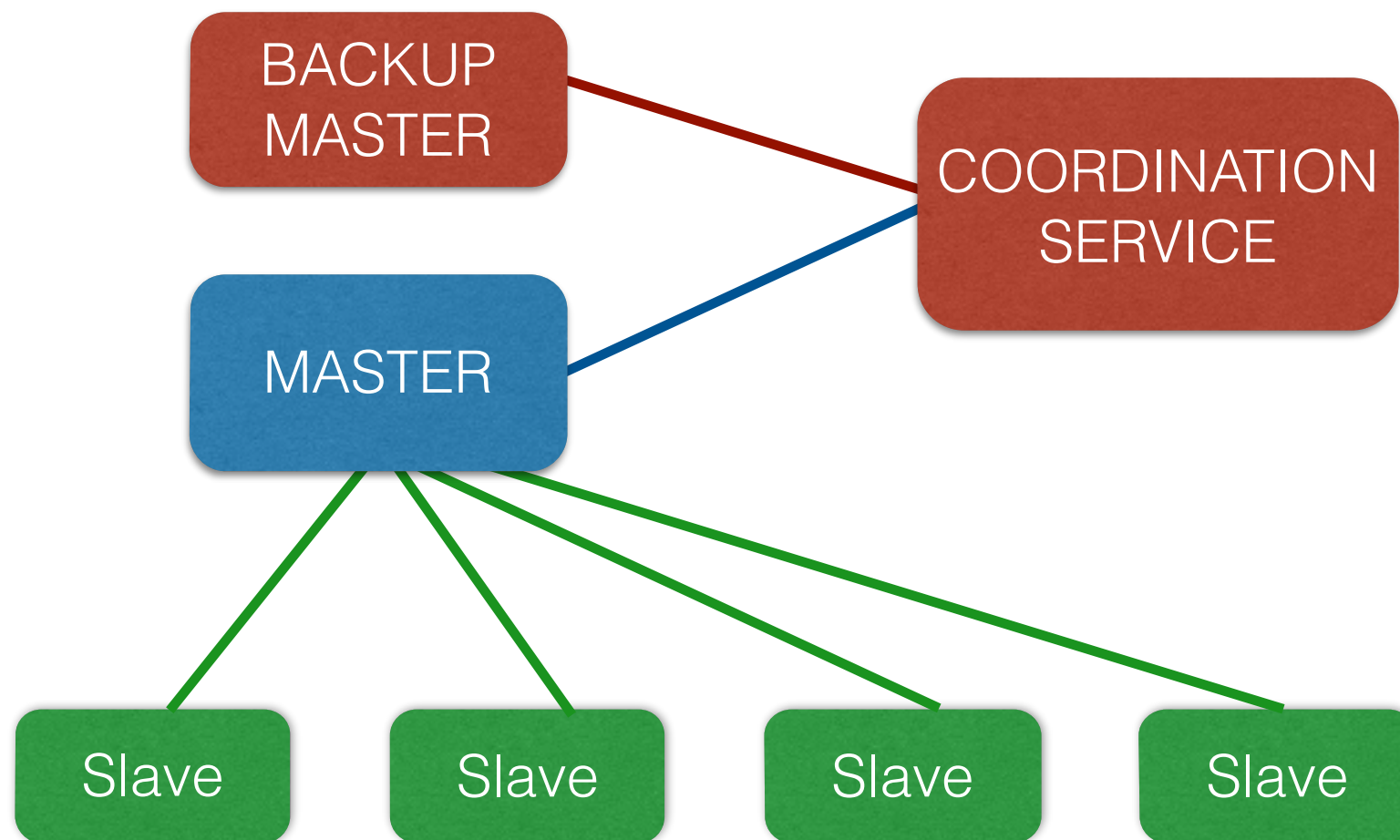
ZooKeeper	The Rest
API that enables application developers to implement their own primitives easily	specific primitives are implemented on the server side

How can a distributed system look like?



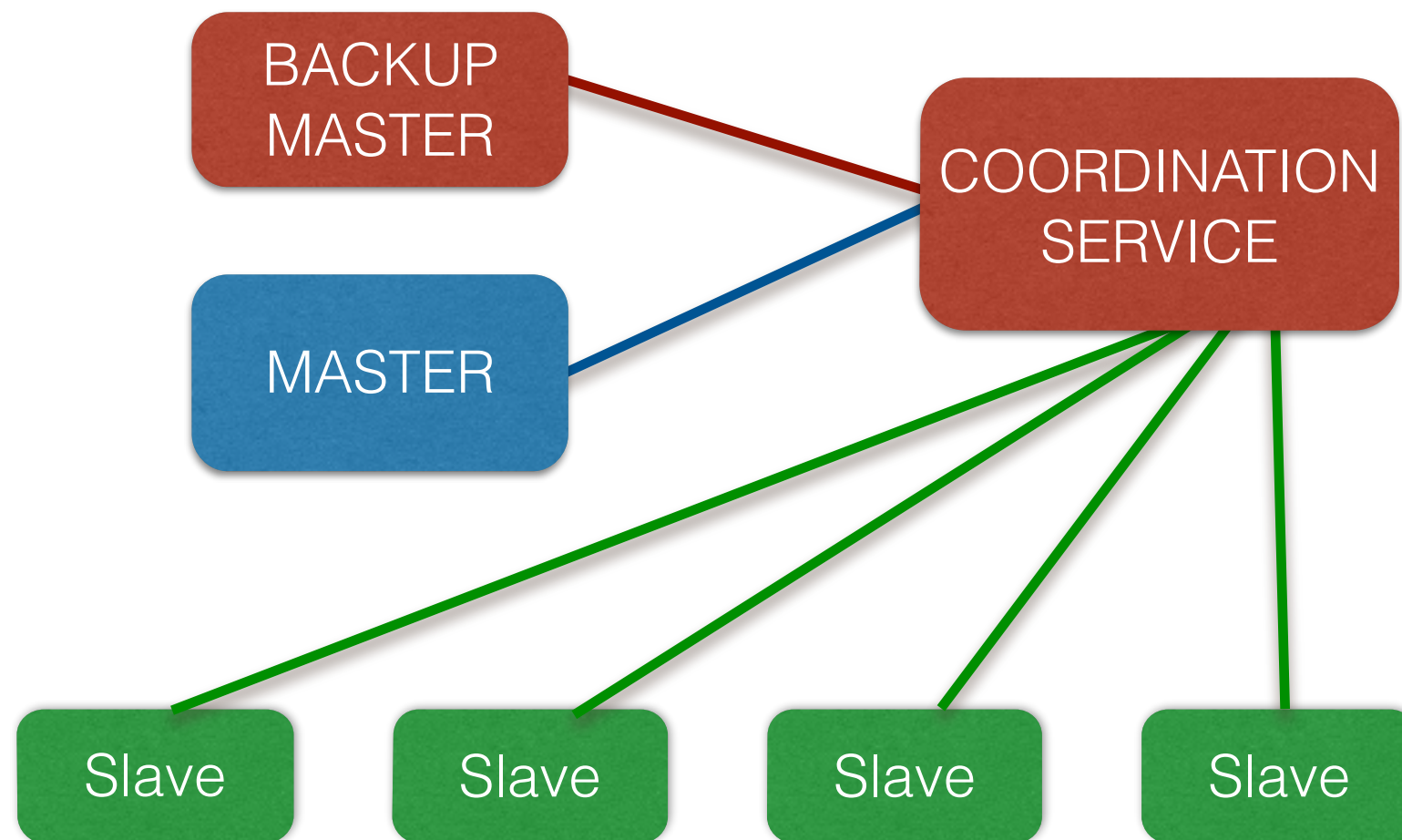
- + **simple**
- **coordination performed by the master**
- **single point of failure**
- **scalability**

How can a distributed system look like?



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

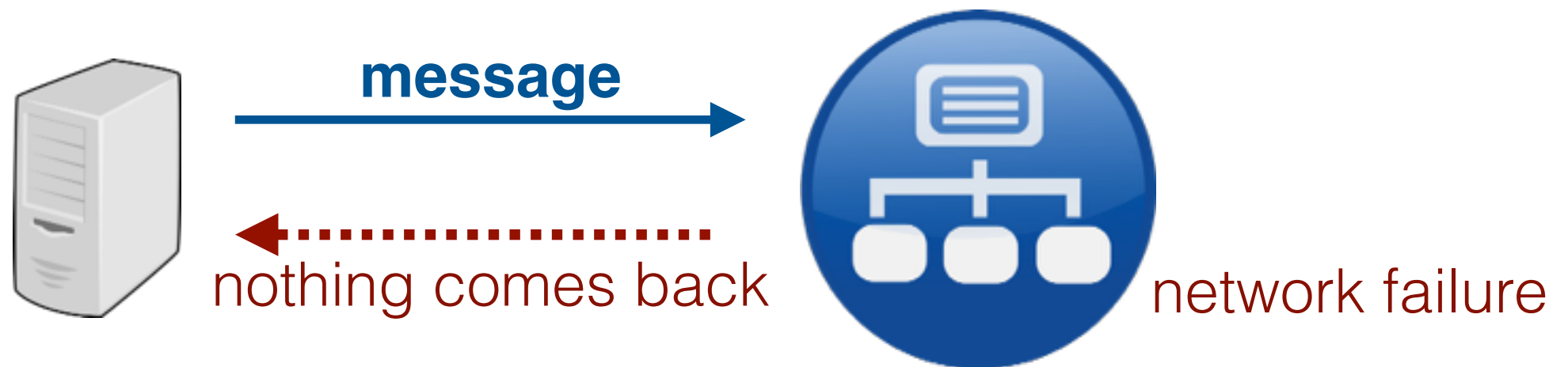
How can a distributed system look like?



+ 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?
- **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

ZooKeeper's design principles



Remember the dining philosophers, forks & deadlocks.

- 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)
- 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

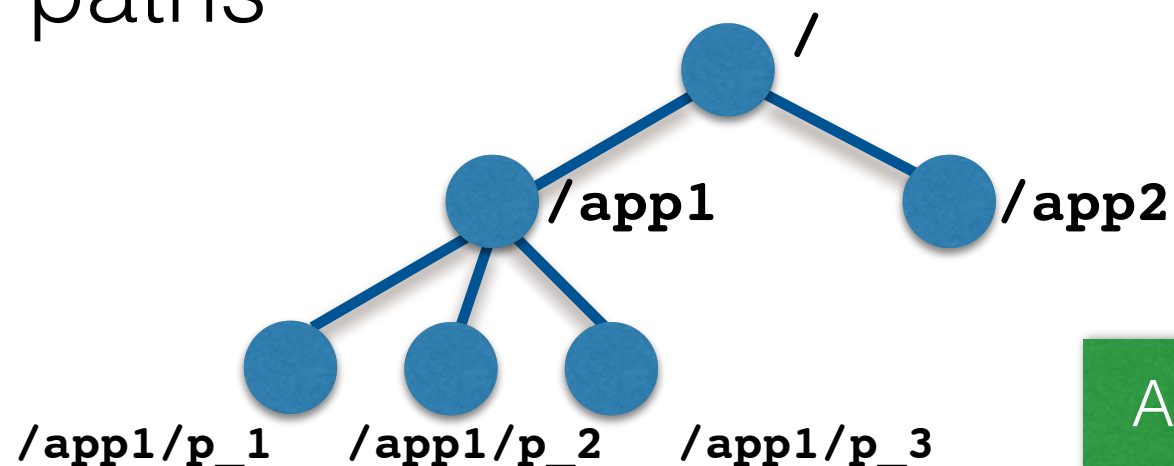
ZooKeeper is optimised for read-dominant operations!

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 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)**

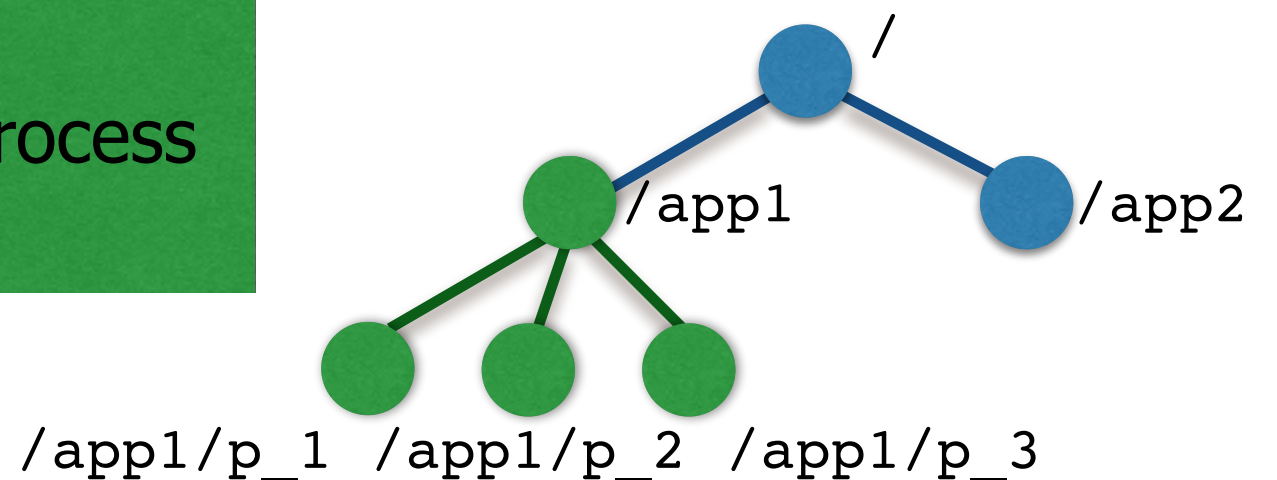
znodes

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

Group membership protocol:

Client process p_i creates znode p_i under $/app1$.

$/app1$ persists as long as the process is running.

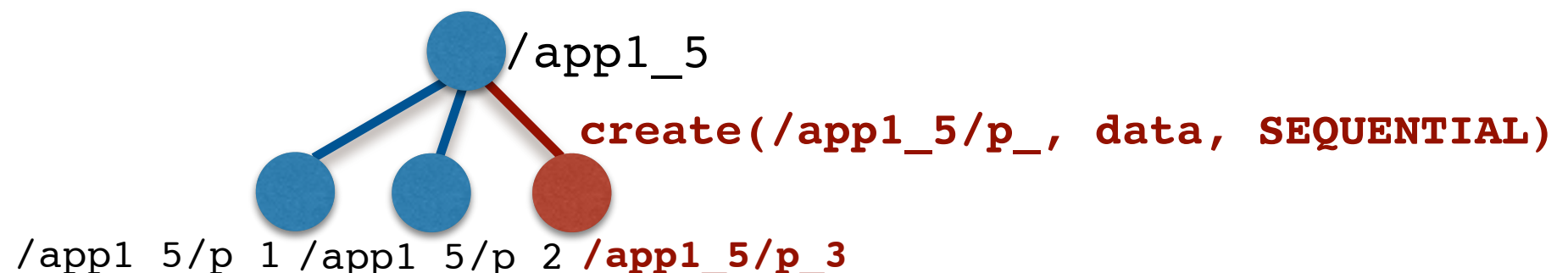


znode flags

- 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



znodes & watch flag

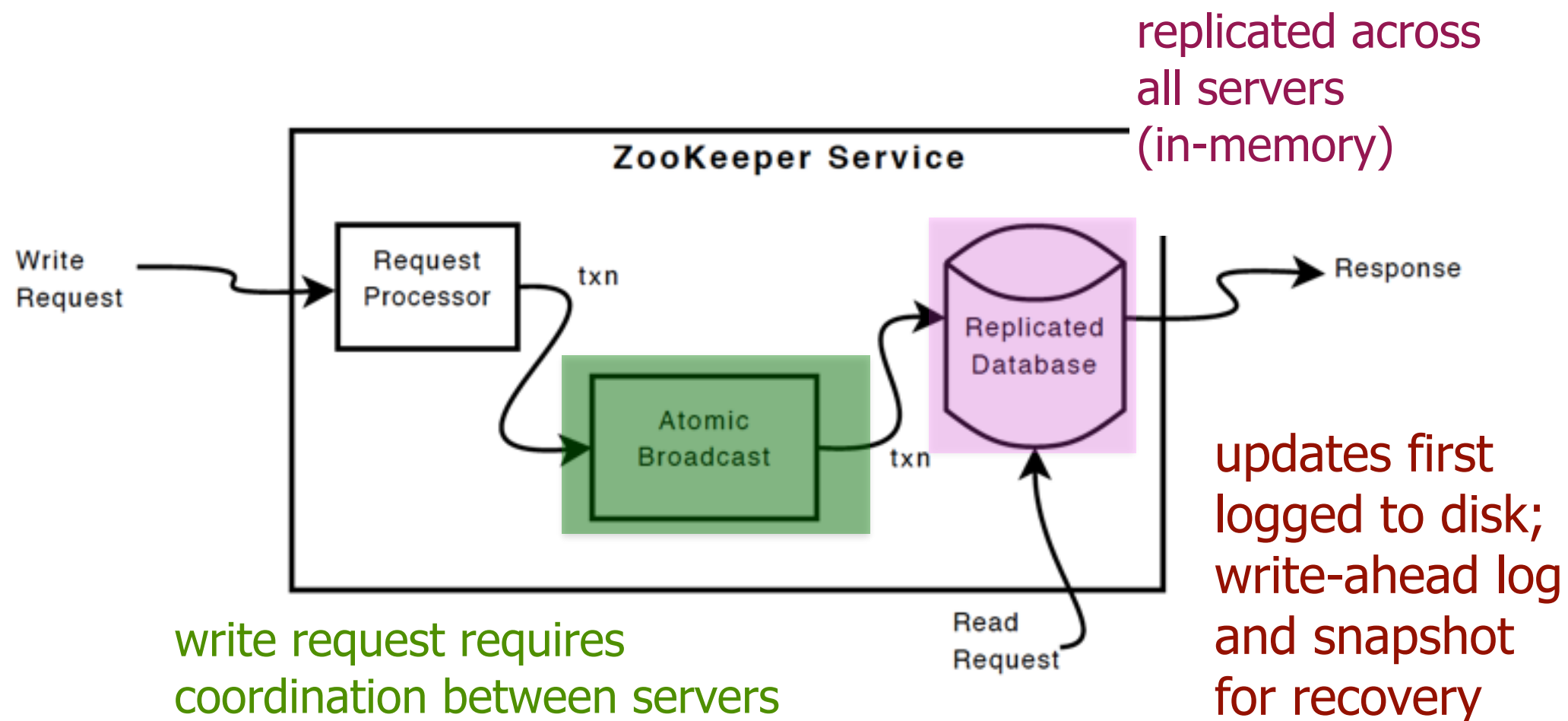
- Clients can issue read operations on znodes with a watch flag
- Server **notifies** the client when the information on the znode has changed
- Watches are **one-time** triggers associated with a session (unregistered once triggered or session closes)
- Watch notifications indicate the change, not the 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

A few implementation details

ZooKeeper data is replicated on each server that composes the service



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

No partial read/writes
(no **open**, **seek** or
close methods).

ZooKeeper API

- `String create(path, data, flags)`
 - creates a znode with path name path, stores data in it and sets flags (ephemeral, sequential)
- `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)`

Note: no `createLock()` or similar methods.

Example: configuration

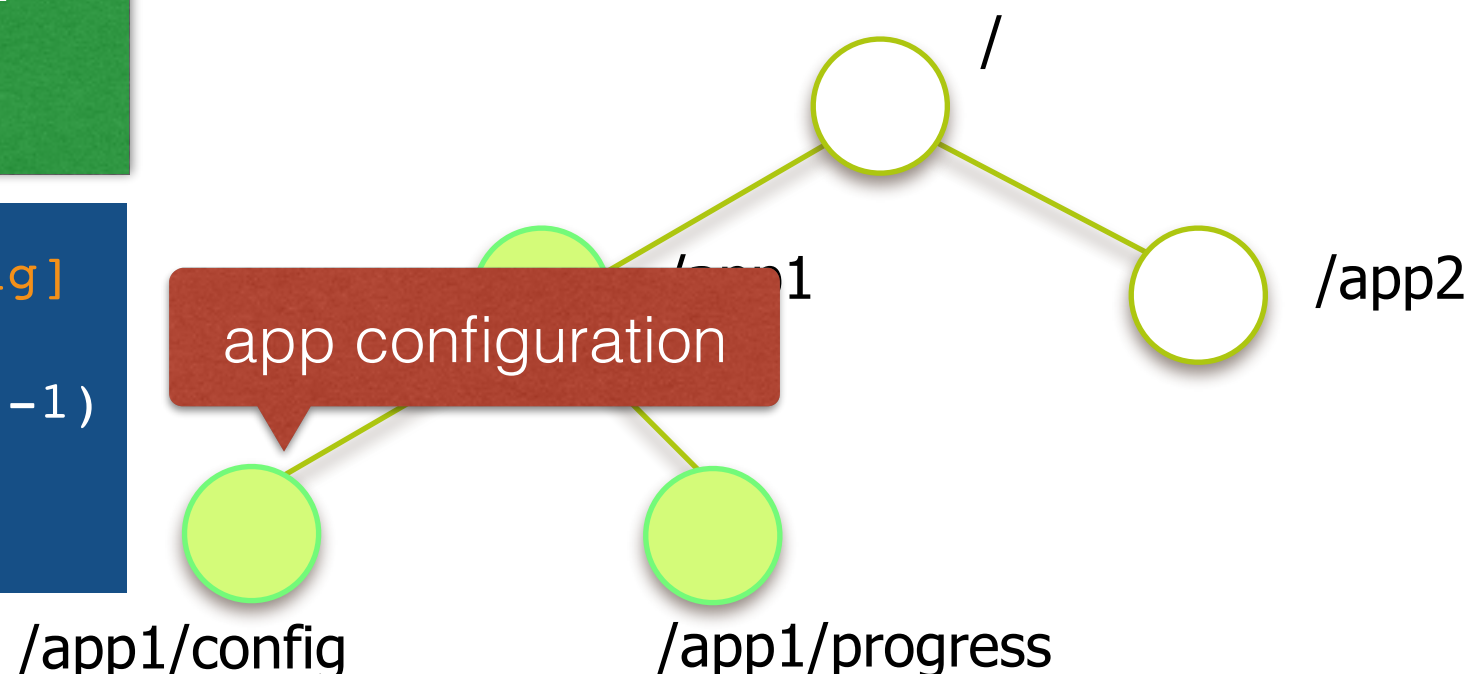
Questions:

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)`

- `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)`



Example: group membership

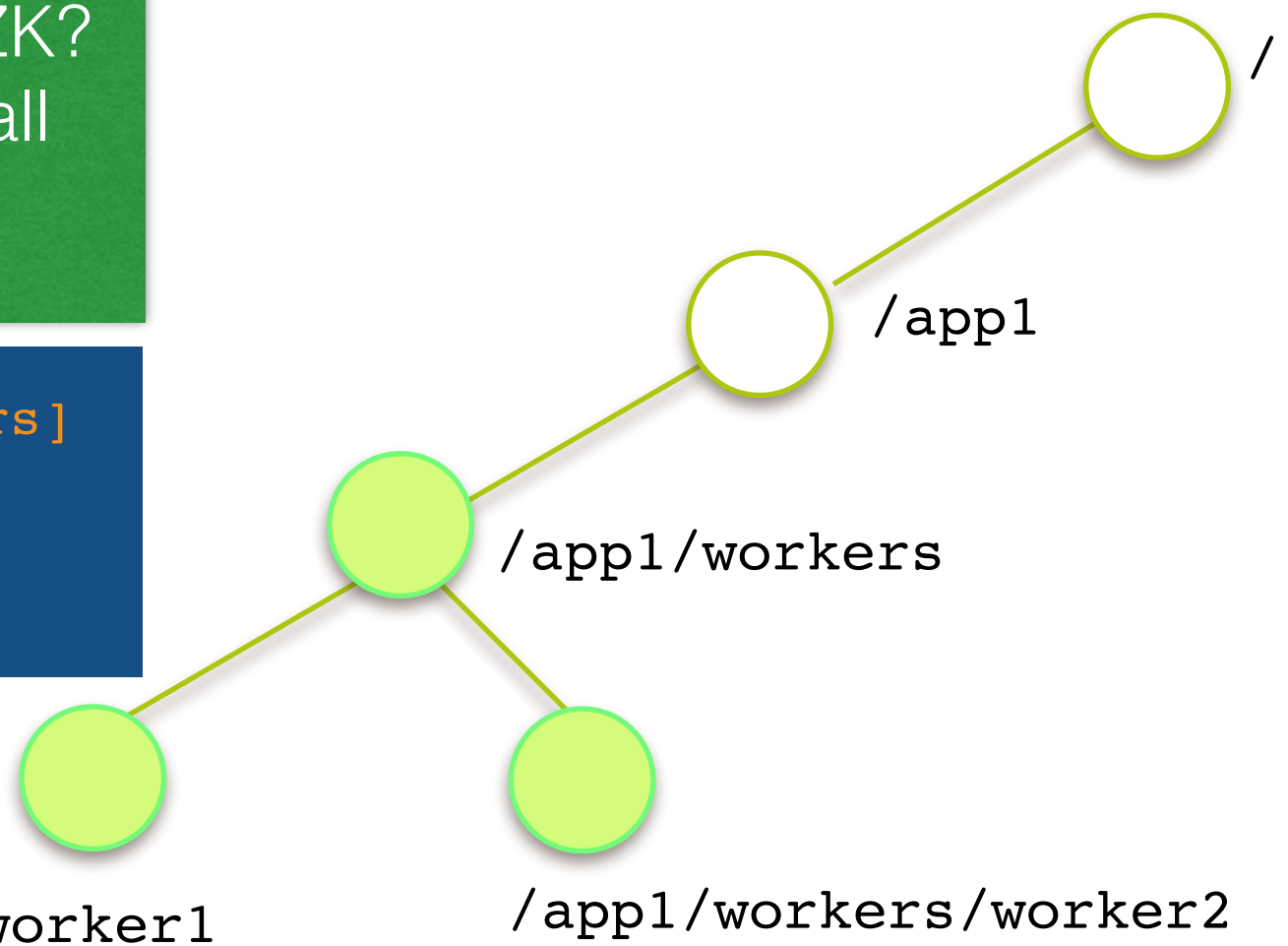
- `String create(path, data, flags)`
- `void delete(path, version)`
- `Stat exists(path, watch)`
- `(data, Stat) getData(path, watch)`
- `Stat setData(path, data, version)`
- `String[] getChildren(path, watch)`

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)`

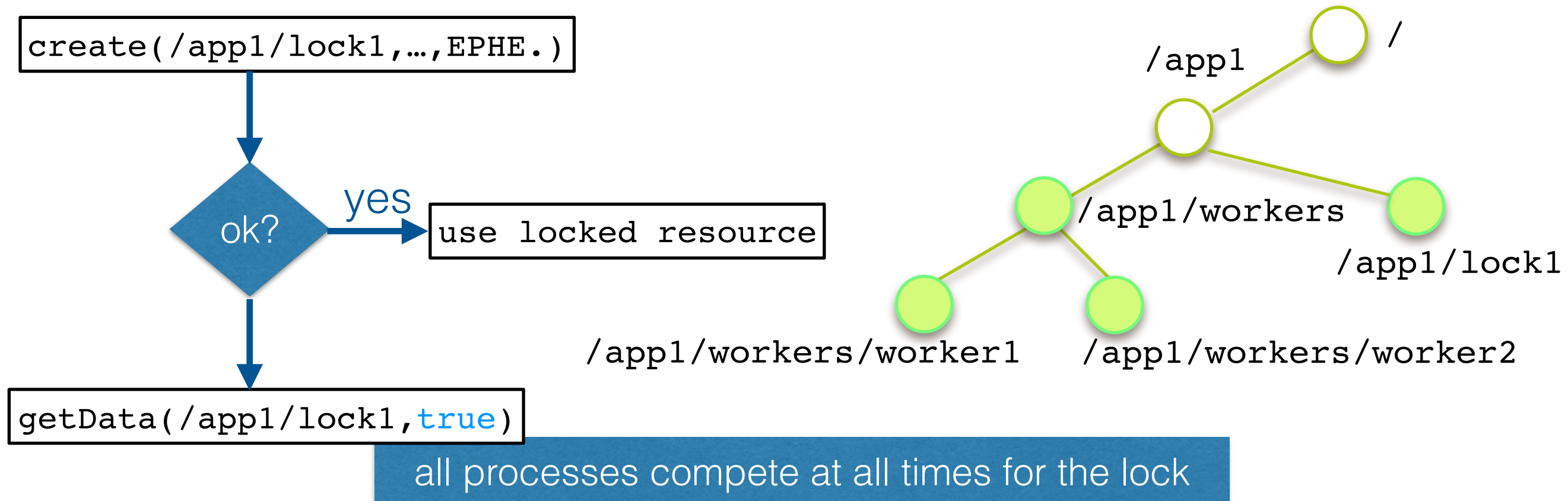


Example: simple locks

- `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)`

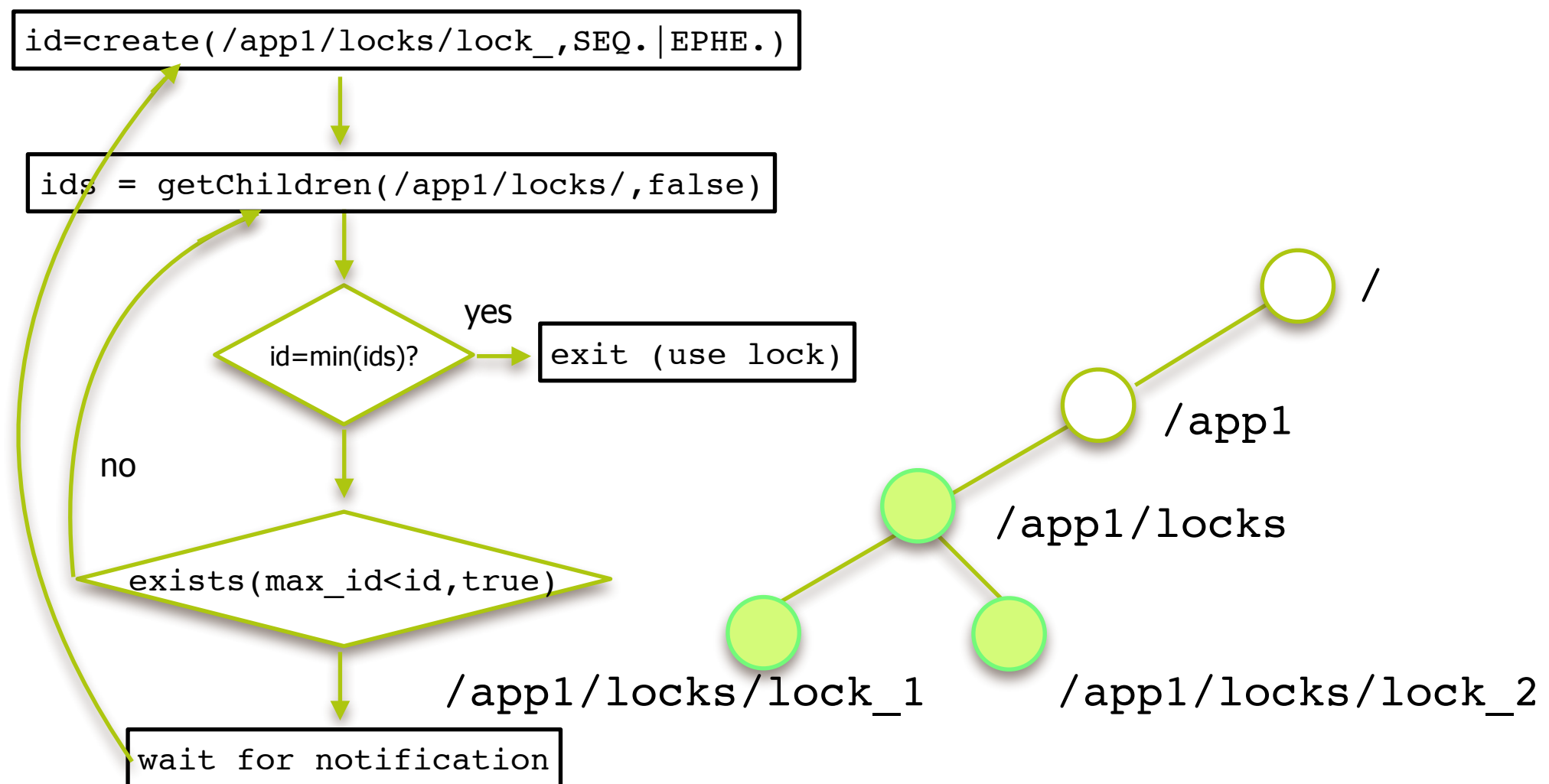
Question:

1. How can all workers of an application use a single resource through a lock?



Example:

locking without herd effect



Question:

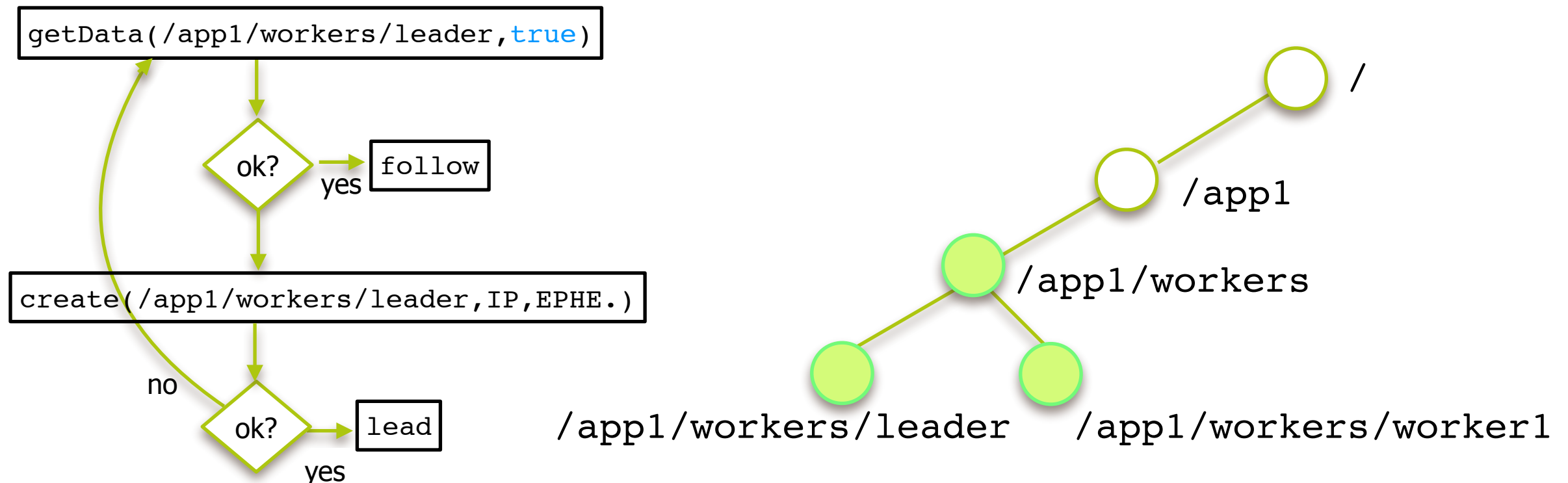
1. How can all workers of an application use a single resource through a lock?

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:

1. How can all workers of an application elect a leader among themselves?



if the leader dies, elect again (“herd effect”)

ZooKeeper applications

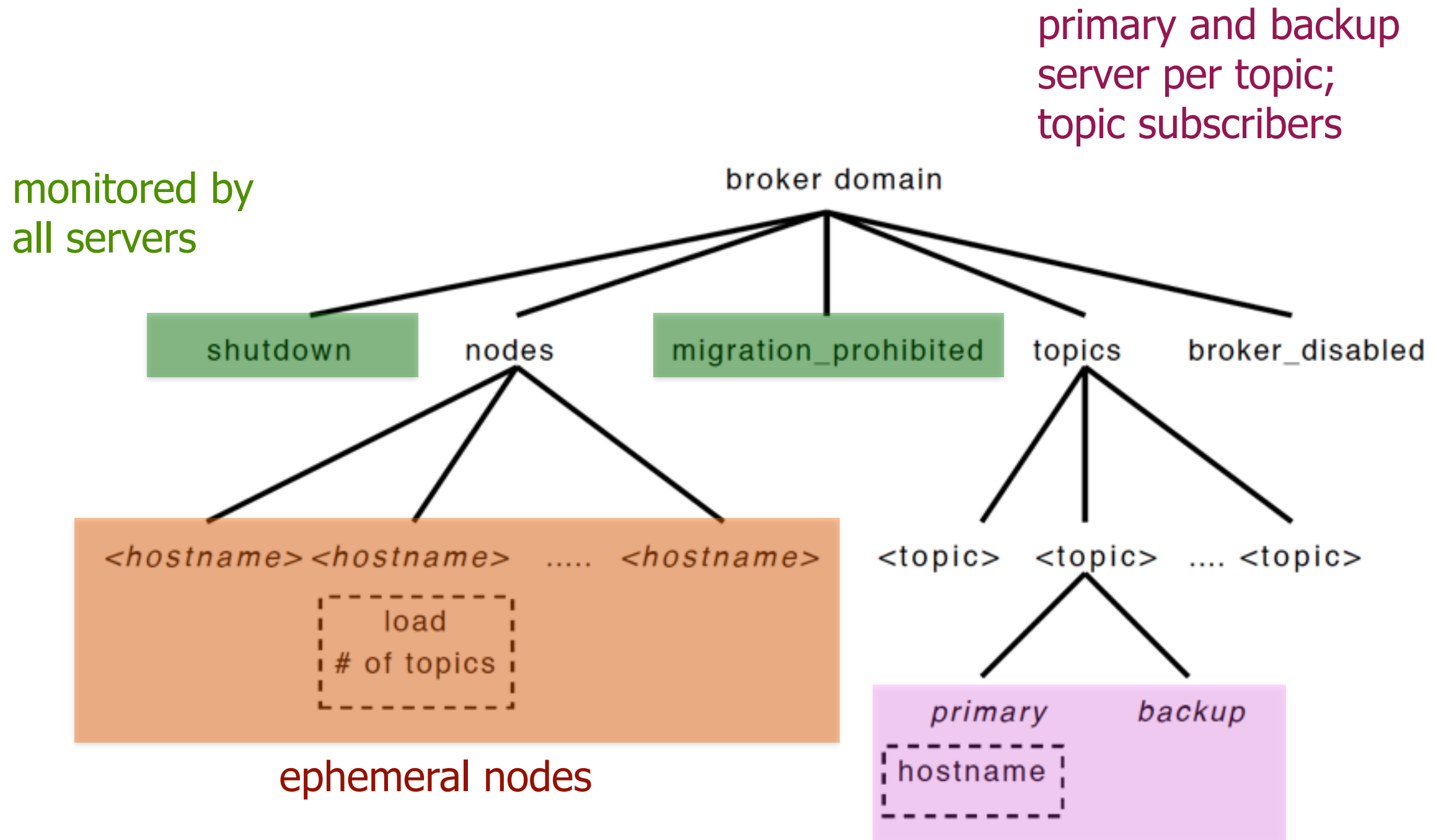
The Yahoo! fetching service

- 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

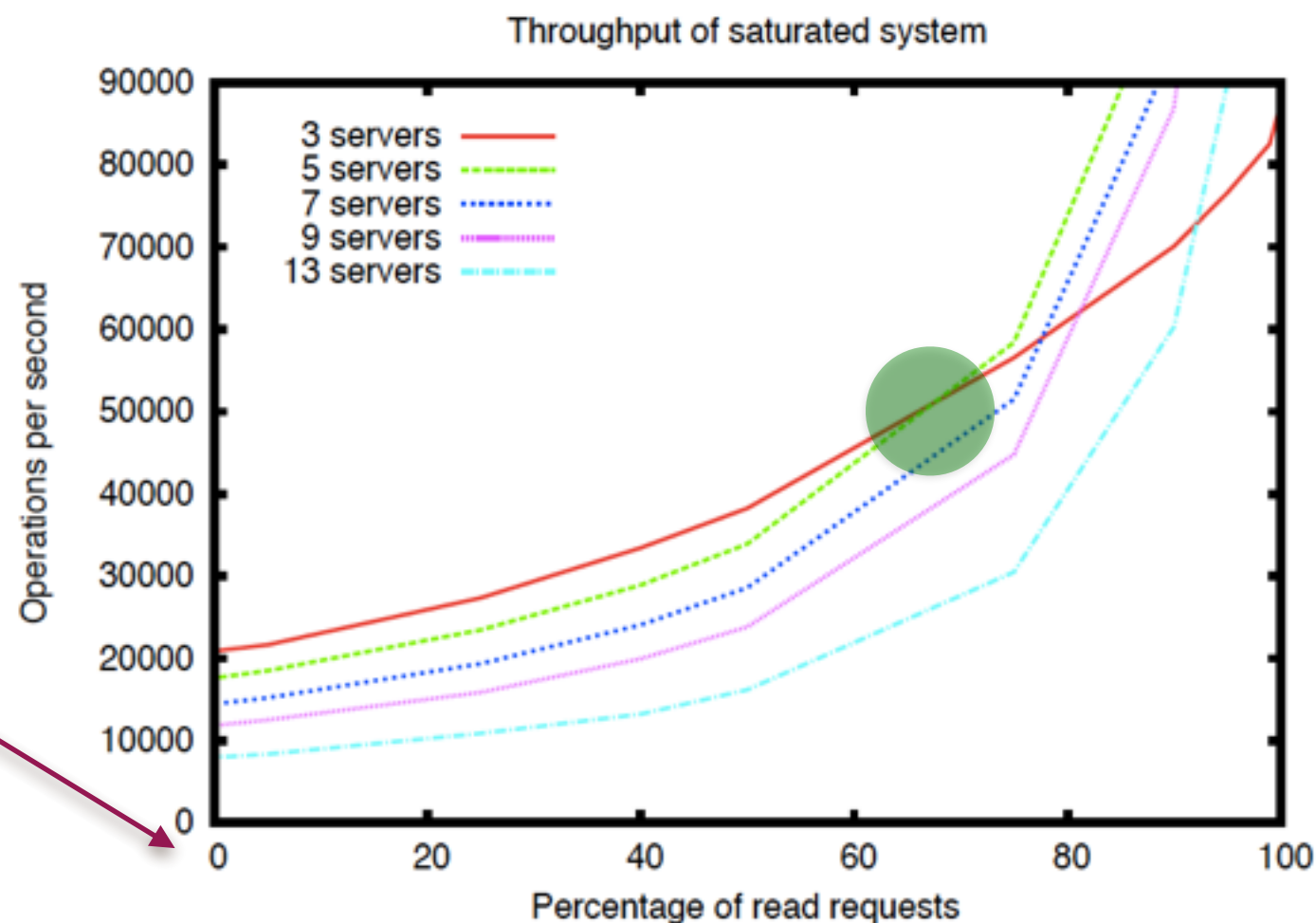
- A distributed publish-subscribe system
- The system manages thousands of topics that clients can publish messages to and receive messages from
- The topics are distributed among a set of servers to provide scalability
- Used primitives of ZK: configuration metadata (to distribute topics), failure detection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

Yahoo! message broker



Throughput

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



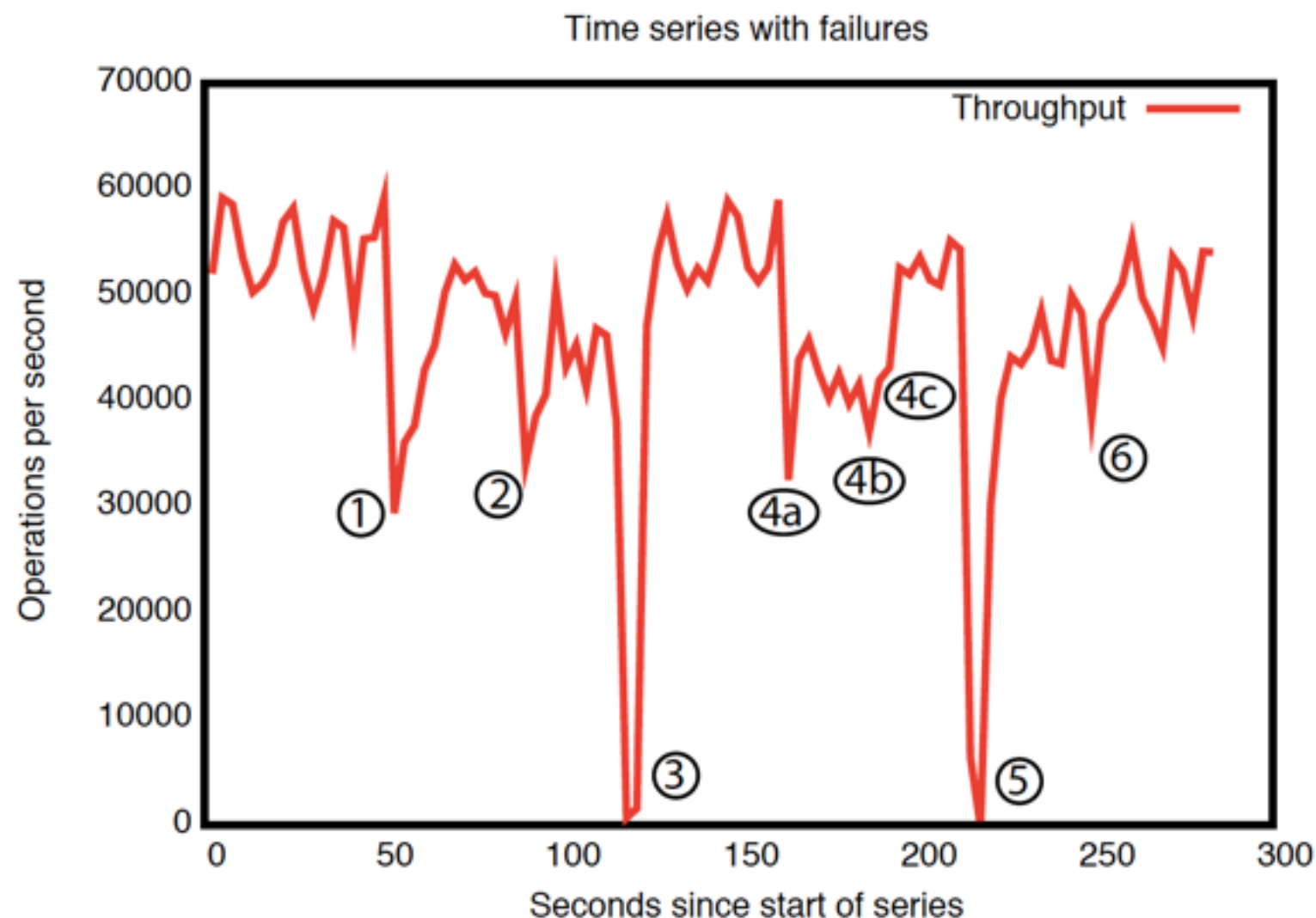
crossing eventually
always happens

only write
requests

only read
requests

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

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

That's it!

In 7 weeks we covered ...

- Data streaming
- MapReduce
- Pig
- HBase
- Giraph
- ZooKeeper

All major
“big data”
technologies
in use today.

These
technologies
are still
continuously
changing.

New ones
appear all
the time, e.g.
Dremel

What about the exam?

- Multiple choice & open questions
- Take a hint from the assignments, the quizzes and the in-lecture Q&A sessions
- Take a look at last year's exam and resit
- Nobody asked for a Dutch exam, answers are expected in English!

THE END