ZooKeeper

"Because Coordinating Distributed Systems is a Zoo"

Slides adapted from C. Hauff

Motivation

Why do we need a coordination service?

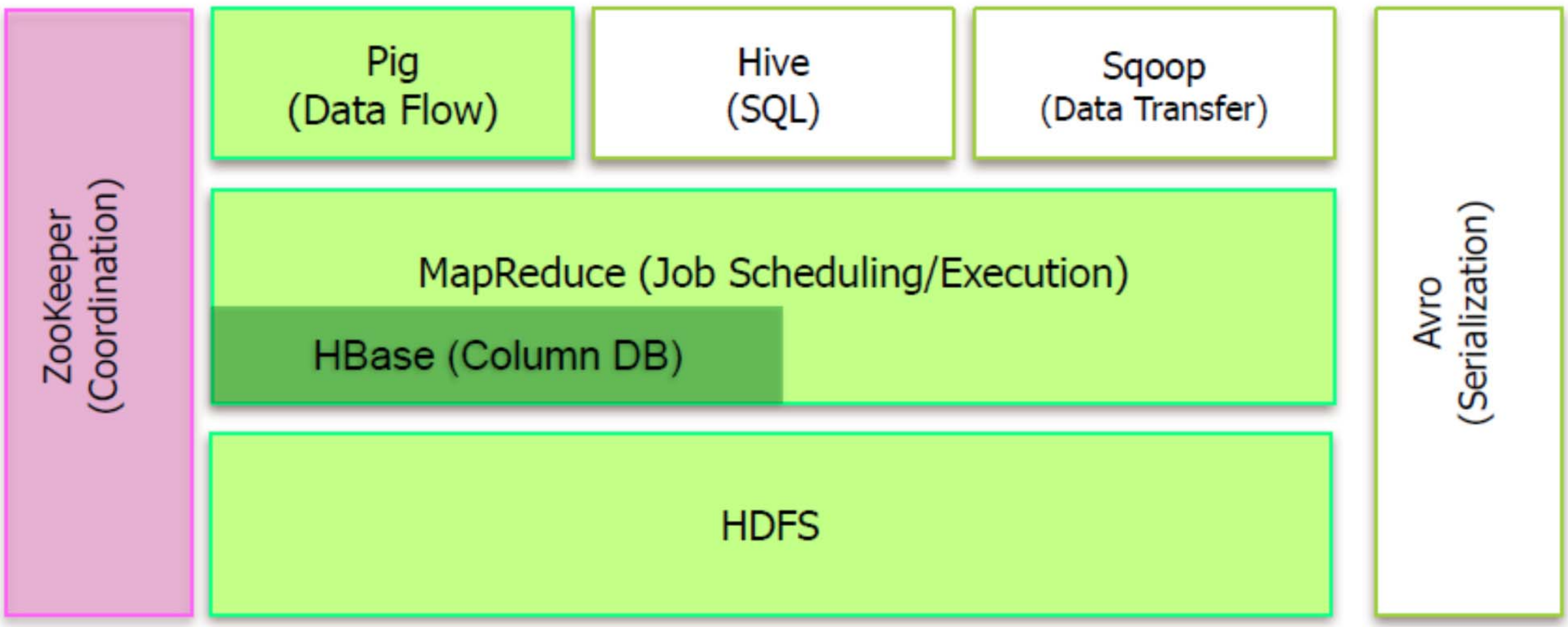
- Formerly, a **single** process running on a **single** node with a **single** CPU – no coordination required
- Today, applications, so called **services**, consist of **independent** processes running on a **changing** set of nodes
- Difficulty: **coordination** of those independent processes
- Developers have to deal with **coordination logic** and **application logic** at the same time

ZooKeeper

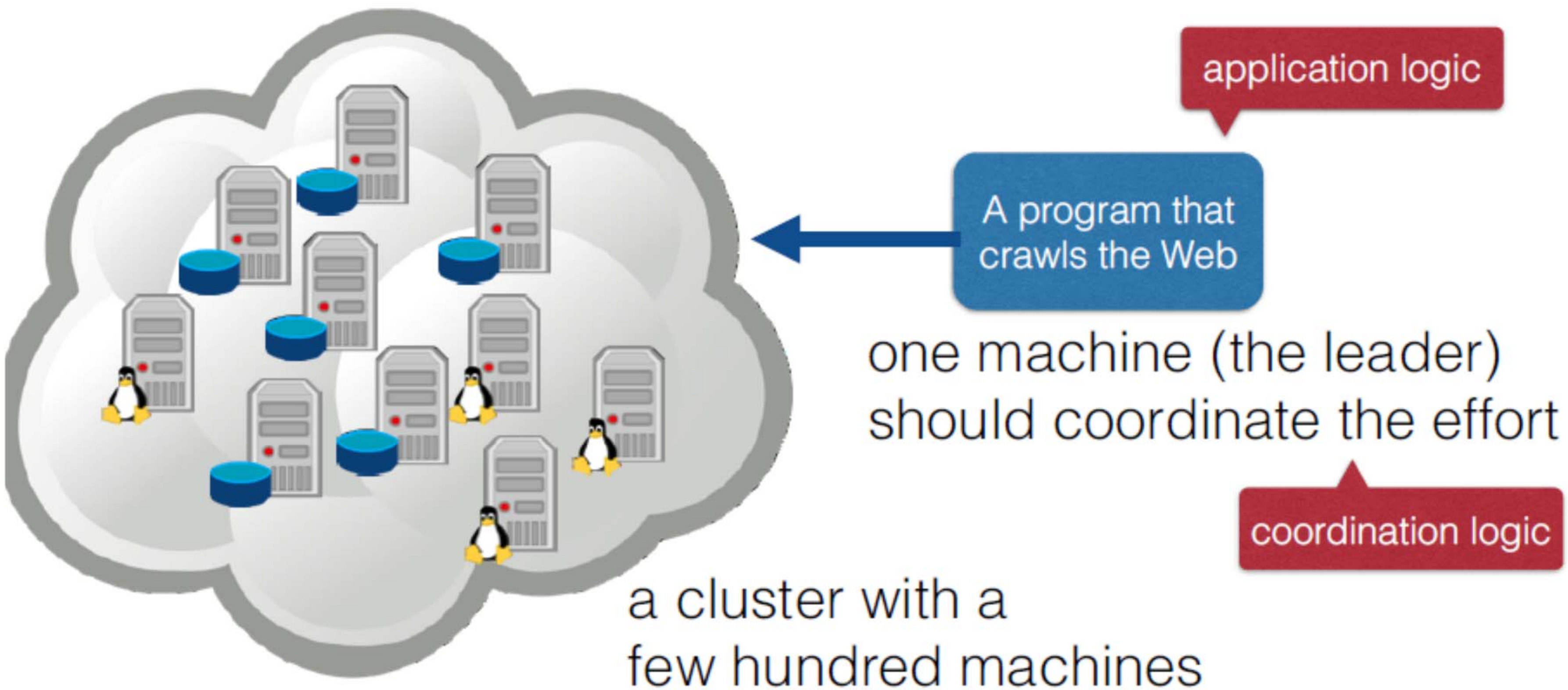
- ZooKeeper is designed to relieve developers from writing coordination logic code

- ZooKeeper is a highly-available service for coordinating processes of distributed applications

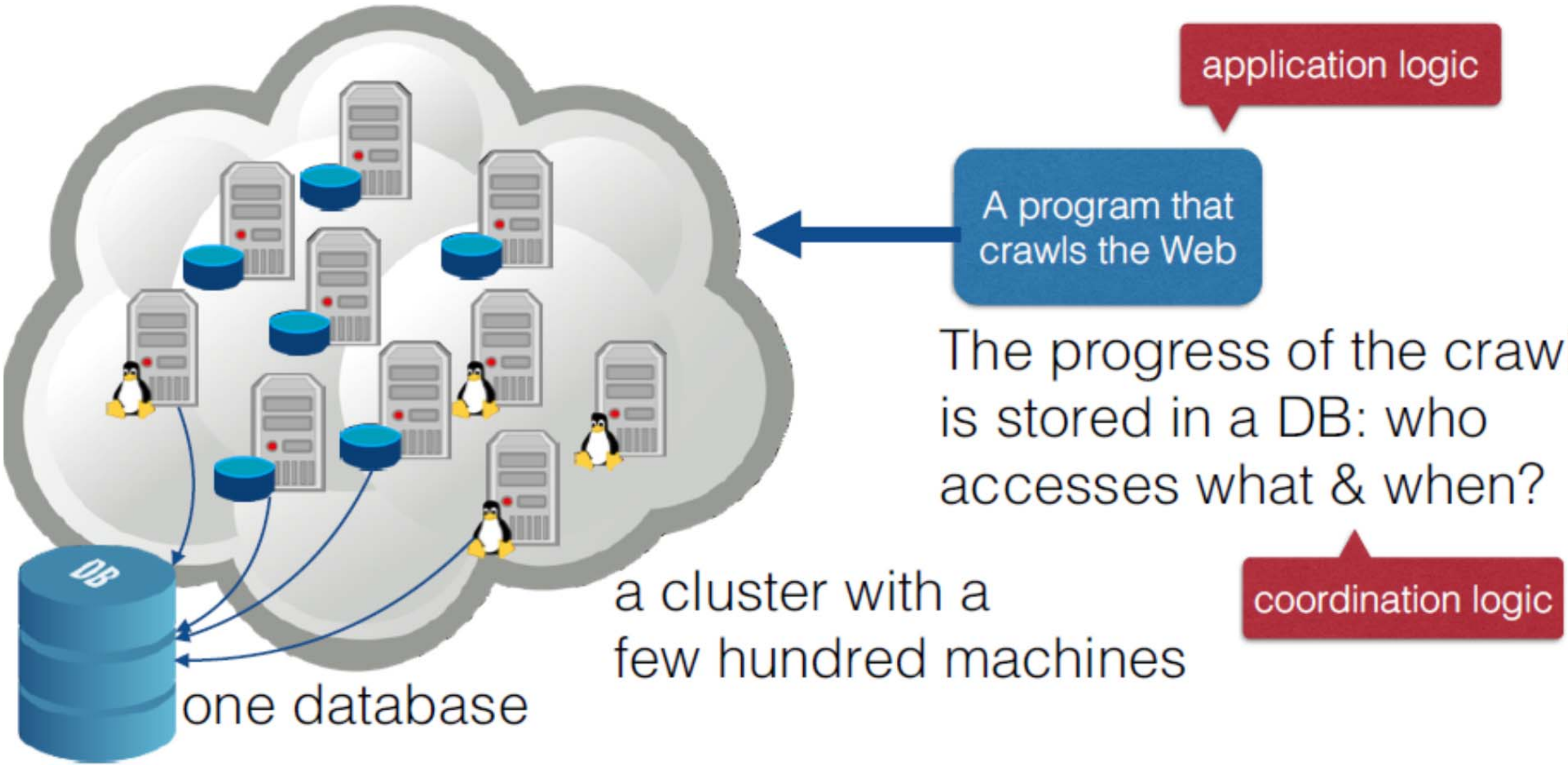
ZooKeeper in the Hadoop ecosystem



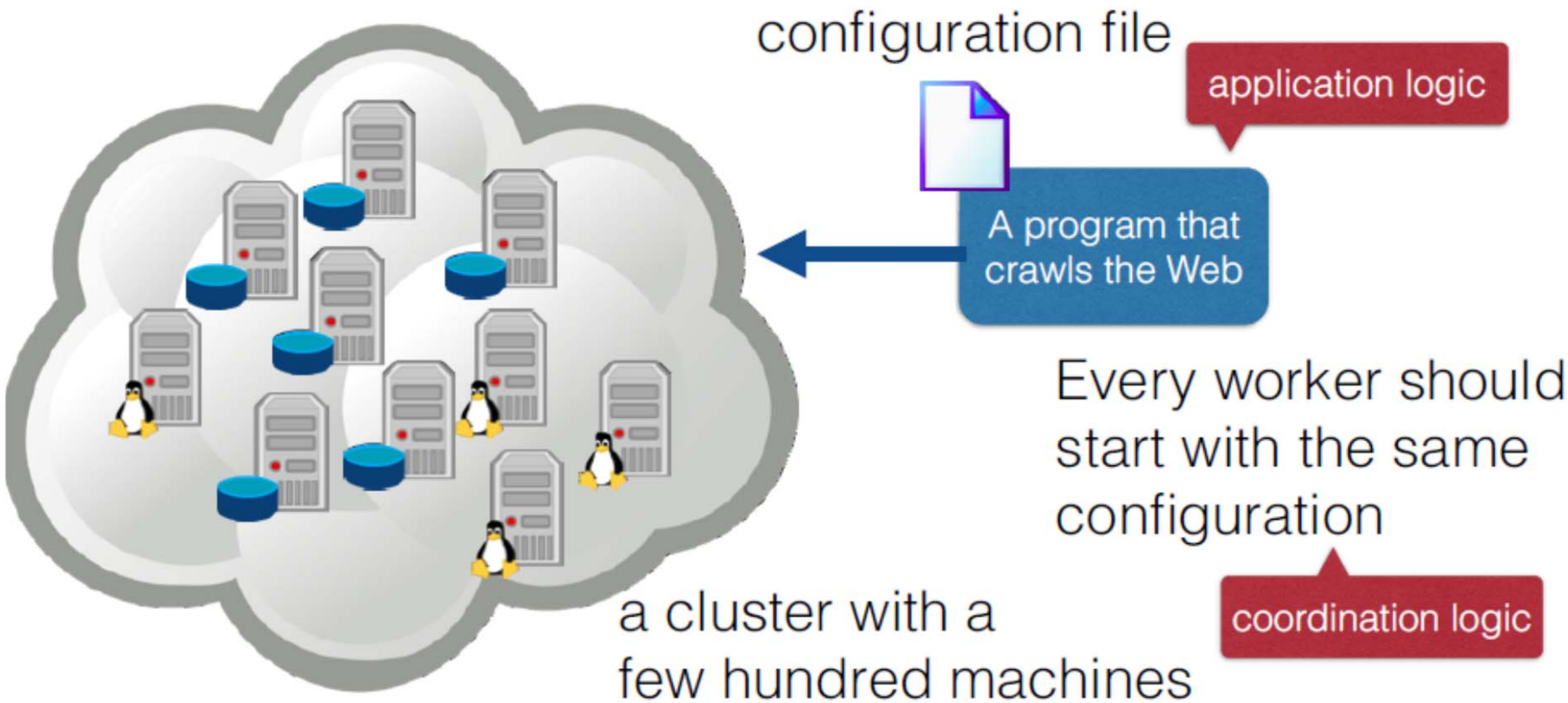
How do you elect the leader?



How do you lock a service?



How do you distribute the configuration?



ZooKeeper philosophy

- Be **specific** and develop a particular service for each coordination task
  - Locking service
  - Leader election service
  - Distribute coordination information

- Be **general** and provide an API to enable other services

- **ZooKeeper offers API for developers to build their own primitives**

Typical coordination problems

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

ZooKeeper API enables all these coordination tasks.

ZooKeeper (ZK) principles

- Design principles
  - API is **wait-free**
  - No blocking primitives in API
  - No deadlocks
  - Blocking can be implemented at client (deadlock possible)

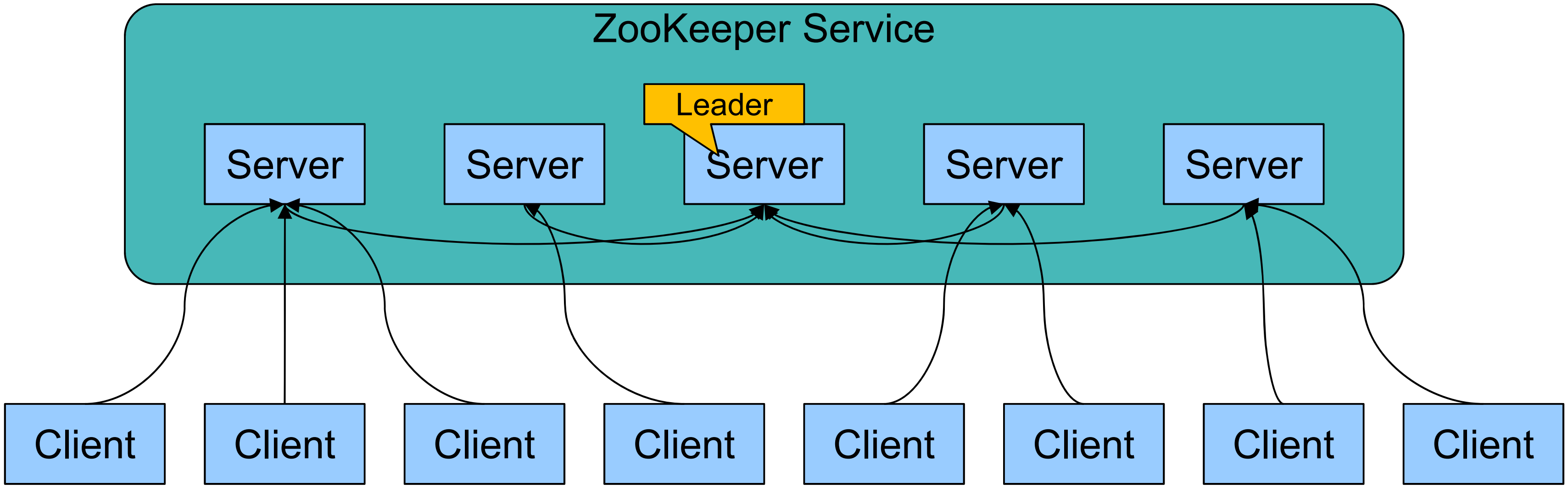
- Guarantees
  - **Writes** to ZooKeeper **are linearizable** (appear atomic)
  - Client requests are processed in **FIFO order**

- Clients receive notifications of changes before the changed data becomes visible

ZK terminology

- **Client** is a user of ZK service
- **Server** is a process providing ZK service
- **Znode** is an **in-memory** data node in ZK, organised in a **hierarchical namespace** (data tree)
- **Write** (update) is any operation which modifies the state of data tree
- Clients establish a **session** when connecting to ZK

ZooKeeper servers & service I

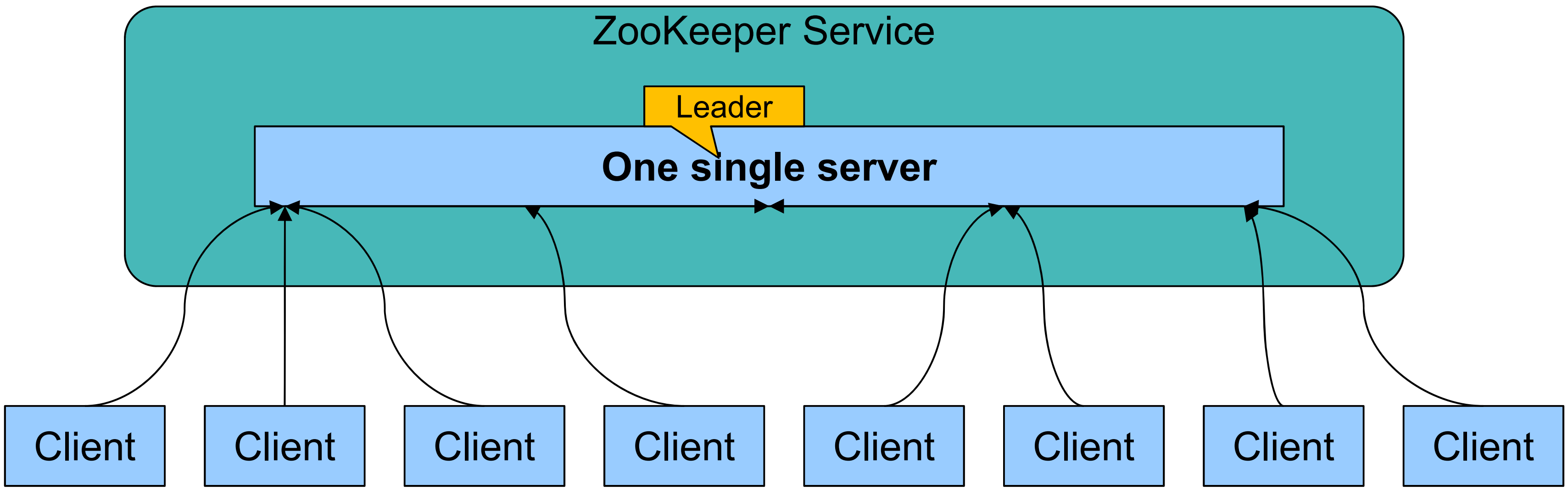


- ZooKeeper service comprised of **multiple servers**
- A designated **leader server** processing client **write requests** and a set of **follower servers** processing client **read requests**
- For development, ZooKeeper can be run in **standalone mode (a single server** which is "leader and follower" at the same time; **our modus operandi in M2-M4**)

ZooKeeper servers & service II

- All servers store a copy of data tree (in memory)
- **Leader** is elected at startup (or upon leader failure)
- **Followers** service client read requests, all updates go through leader
- **Write acknowledgements** are sent when a **majority of servers** persisted a write

ZK standalone mode



- Provides no guarantees
- Is not fault-tolerant
- Meant to ease development

- **Our expectation and modus operandi in M2-M4**

Client sessions

- Clients establish a **session** when connecting to ZK
- Session is with the ZK service
- Clients may connect to different ZK servers within the same session

ZooKeeper request processing

- ZooKeeper server services clients

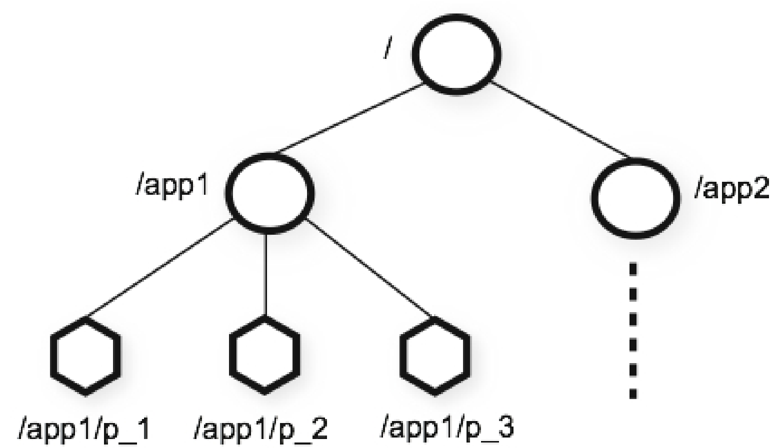
- Clients connect to exactly one server to submit requests
  - **Read requests** served **from local replica**
  - **Write requests** processed by an **agreement protocol** (a server, elected leader, initiates processing of write)

Data model
(Similar to a file system)

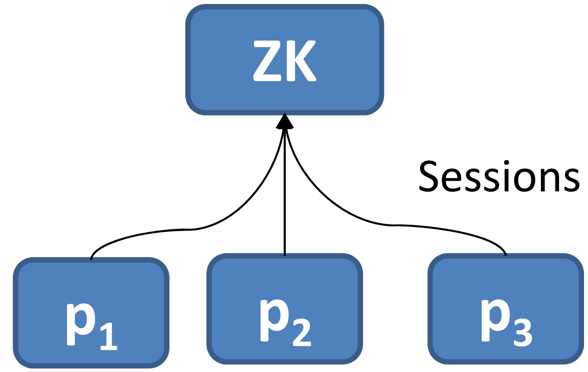
- znodes are organised in a hierarchical namespace
- znodes can be manipulated by clients through ZK API
- znodes are referred to by UNIX-style file system paths (always absolute paths)

znodes can store data (**file like**; KBs up to 1 MB) & can have children (**directory like**).

/app1/p_2

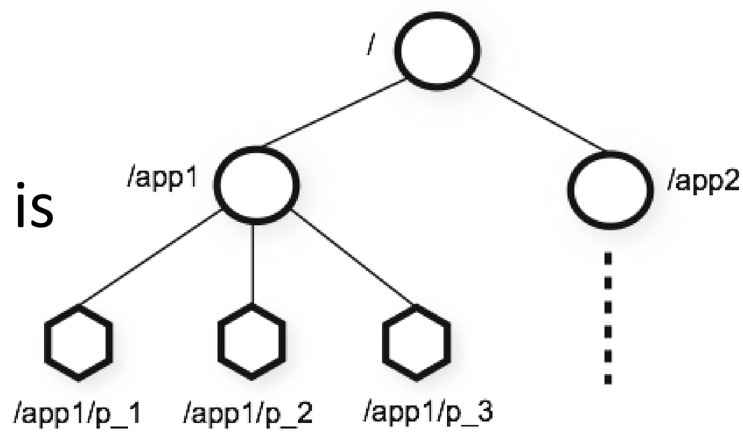


znodes



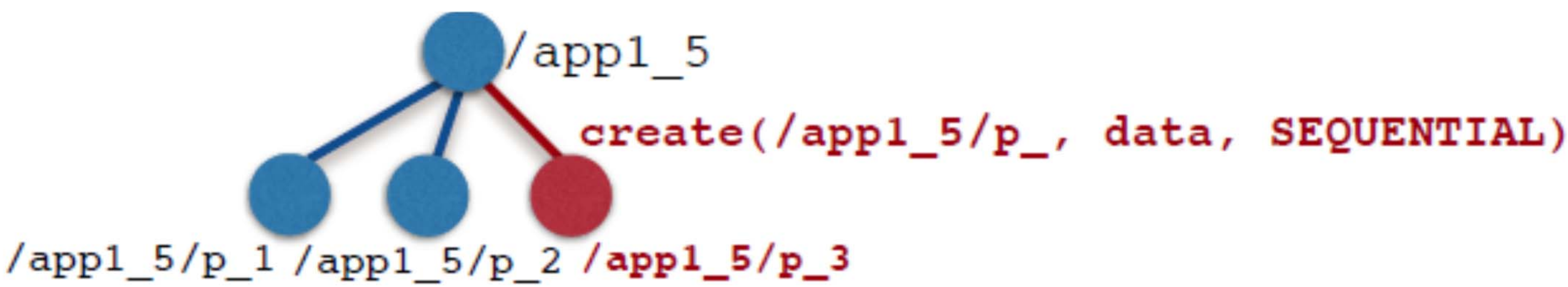
- znodes are not meant for general data storage
- znodes map to abstractions of the client (i.e., the application)

- **Group membership** protocol:
  – Client process $\mathbf{p_i}$ creates znode **p_i** under **/app1**
  – **/app1/p_i** persists as long as p_i is running (via session)



znode flags

- Clients manipulate znodes by creating and deleting them
- **EPHEMERAL** flag (*passing*, *short-lived*)
  - 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** set
- Server **notifies** client when data on znode changes
- Watches are **one-time triggers** associated with a session (unregistered once triggered or session closes)
- Watch notifications **indicate change**, not the new data (client has to retrieve data, post notification)

ZooKeeper API I
(simplified, cf. ZK API documentation)

- Create a znode with path name *path*, store *data* in it and set *flags* (ephemeral, sequential)
  - `string` **`create`**`(`*`path`*`, `*`data`*`, `*`flags`*`)`
- Delete the node *path*, if it is at expected *version*
  - `void` **`delete`**`(`*`path`*`, `*`version`*`)`
- Let client set a *watch* on znode
  - `stat` **`exists`**`(`*`path`*`, `*`watch`*`)`

ZooKeeper API II

(simplified, cf. ZK API documentation)

- Return data and meta-data of znode
  - `(`*`data, stat`*`)` **`getData`**`(path, watch)`

- Write data if version number is current version of znode
  - `stat` **`setData`**`(path, data, version)`

- Return all children of node
  - `string[]` **`getChildren`**`(path, watch)`

ZooKeeper API Notes

- No partial read/writes
- No **open, seek** or **close** methods
- No create-lock, lock, unlock

Configuration management example

1. How does a **new** node query ZK for a configuration?
2. How does a node **change** the configuration?
3. How do nodes read the **new** configuration?

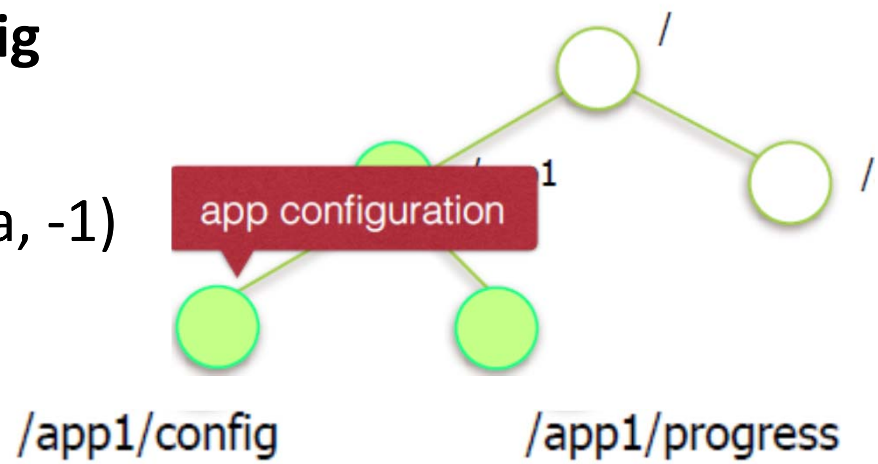
Configuration stored in /app1/config

1. getData(/app1/config, true)

2. setData(/app1/config/config_data, -1)

Watch for configuration changes

3. getData(/app1/config, true)



Group membership example

1. How can all nodes of an application **register themselves** at ZK?
2. How can a node find out about all **active** nodes of an application?

Create znode to store nodes

1. create(/app1/workers/worker_, data, SEQUENTIAL)
2. getChildren(/app1/workers, true)

Simple locking

- Client creates a lock file

  `create``(`*`/app1/z`*$_{lock}$, *`data`*`,` *`EPHEMERAL`*`)`

  If successful, client holds lock, otherwise not
- Client releases lock if it **explicitly deletes** it or if it **crashes**

  `delete``(`*`/app1/z`*$_{lock}$`)`
- Could be held by another client, watch for status changes

  `getData``(`*`/app1/z`*$_{lock}$, TRUE`)`

  Caller is notified once status of lock changes
- Simple lock may lead to "**herd effect**"

Simple lock without herd effect

Intuition: Line up clients for locks; client with lowest node name wins (obtains lock).

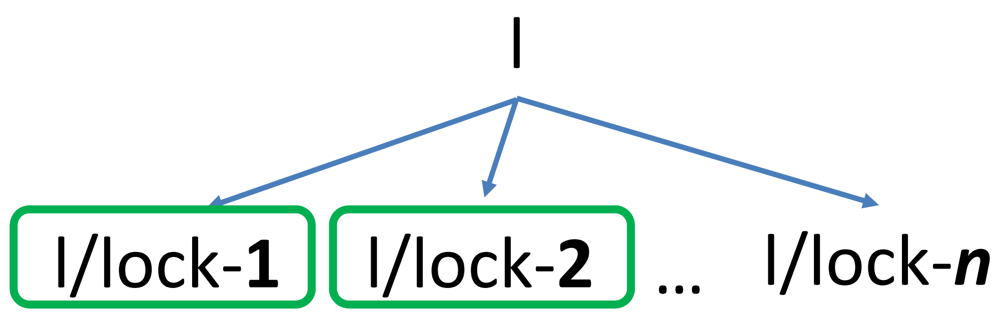
Lock

```
1 n = create(l + "/lock-", EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 if n is lowest znode in C, exit
4 p = znode in C ordered just before n
5 if exists(p, true) wait for watch event
6 goto 2
```

Holds lock

Unlock

```
1 delete(n)
```



Read/Write lock

(multiple readers, single writer)

Write Lock

```
1  n = create(l + "/write-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if n is lowest znode in C, exit
4  p = znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 2
```

Holds write lock

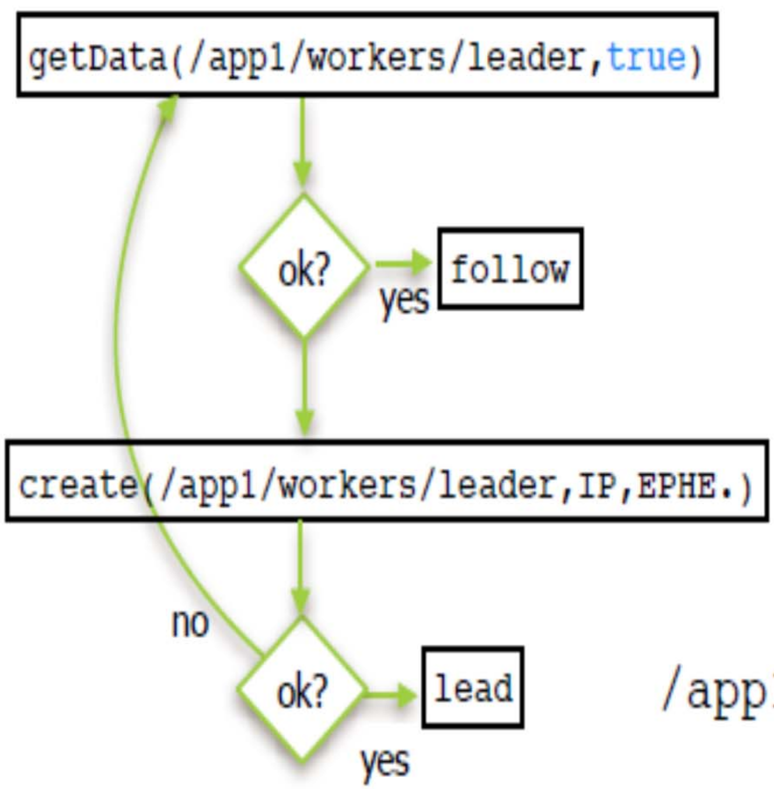
Read Lock

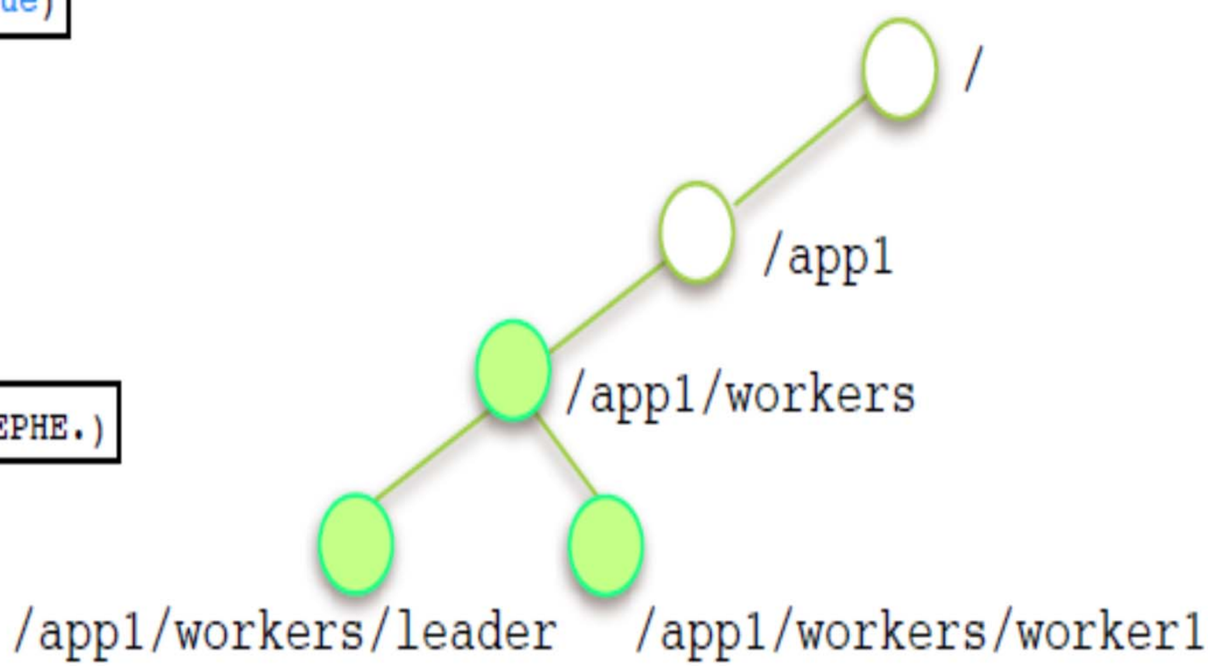
```
1  n = create(l + "/read-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if no write znodes lower than n in C, exit
4  p = write znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 2
```

Holds read lock

Leader election example

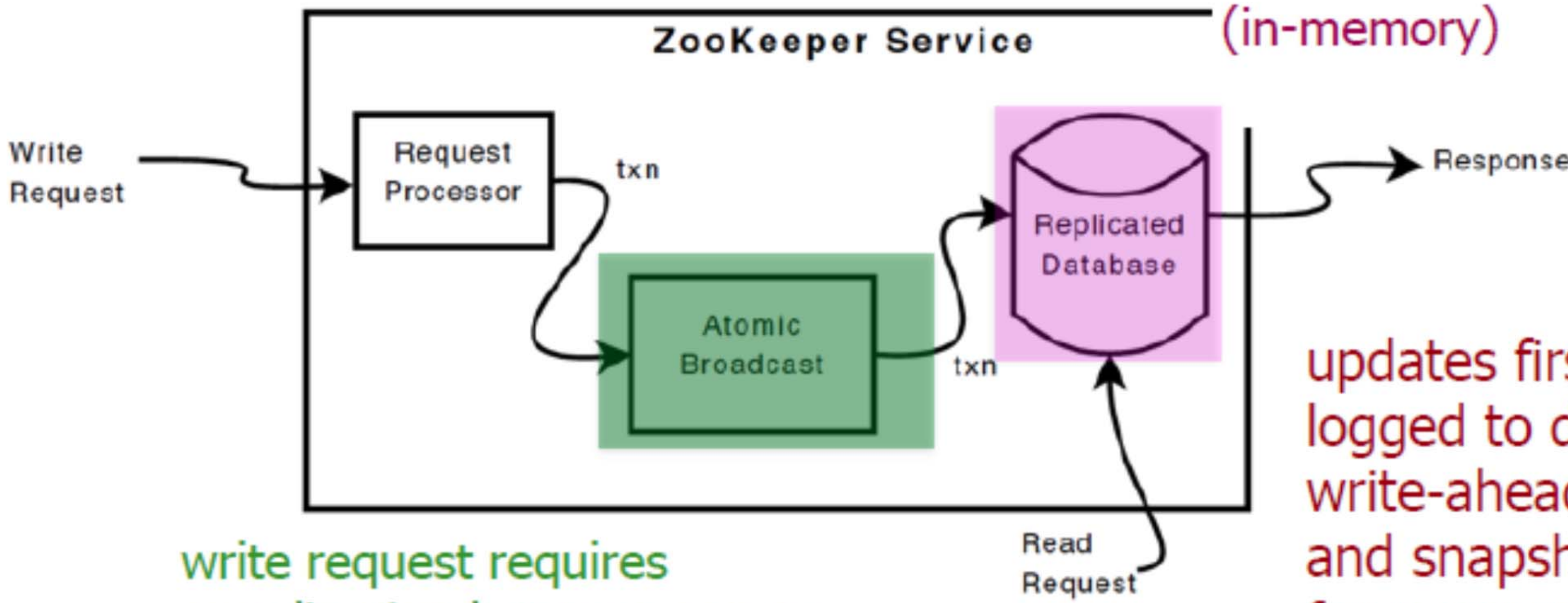
- How can all nodes elect a leader?





ZooKeeper internals

replicated across
all servers
(in-memory)



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

write request requires
coordination between servers