Distributed Systems

Spring Semester 2020

Lecture 9: Zookeeper

John Liagouris
liagos@bu.edu

Why this paper

- Widely used RSM service that has been proven useful both inside and outside of Yahoo:
  - Developed at Yahoo! for internal use
    - Inspired by Chubby (Google's global lock service)
    - Use by Yahoo! YMB and Crawler
  - Open source
  - As part of Hadoop (MapReduce)
  - It has its own Apache project now
- Interesting API for Distributed systems construction
- Generic "kernel" for building replicated "master"

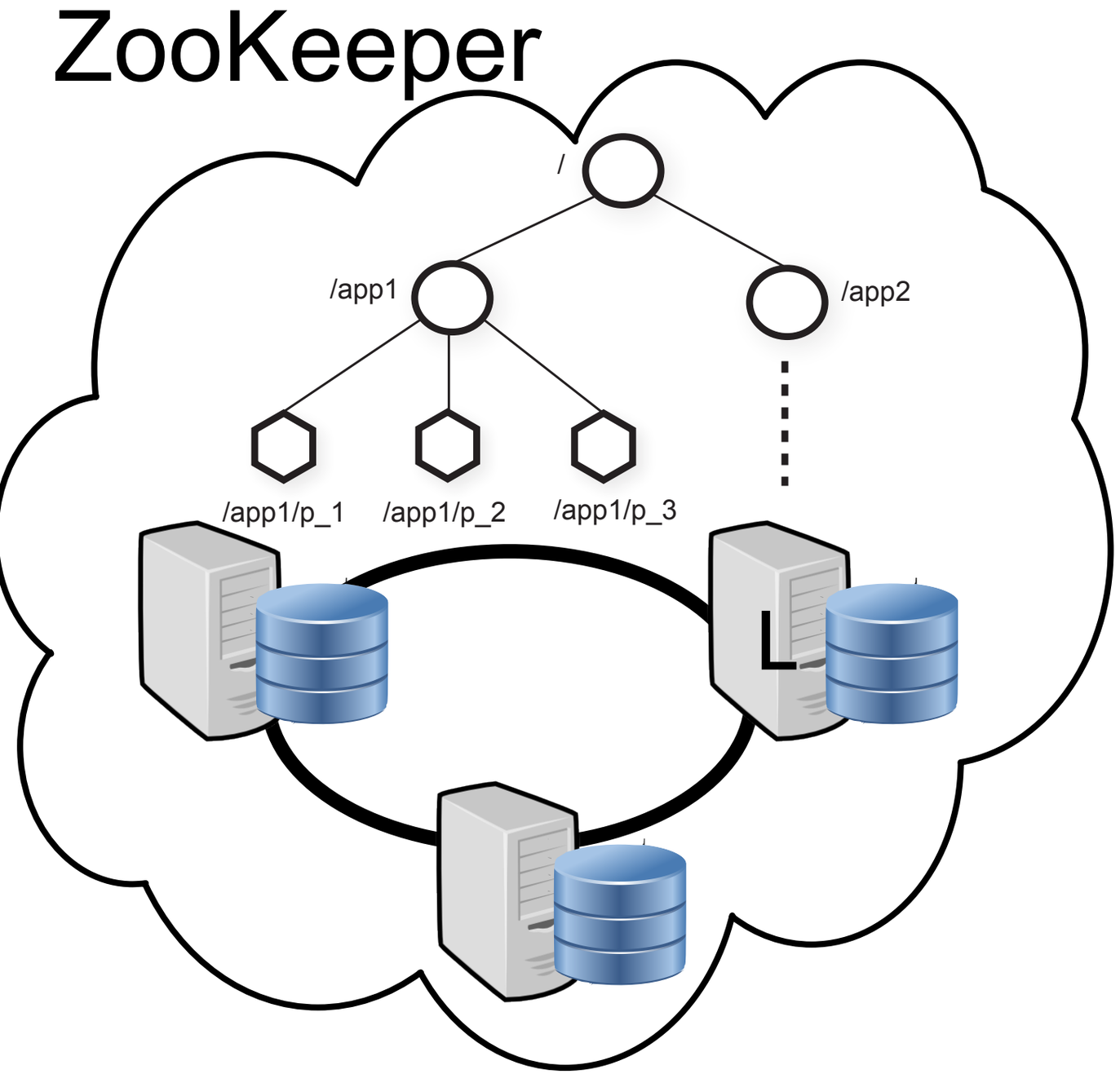
Zookeeper: a generic "master" service

Masters have the nice property of a centralized "serialization" point

- Design challenges:
  - What API?
  - How to make master fault tolerant?
  - How to get good performance?

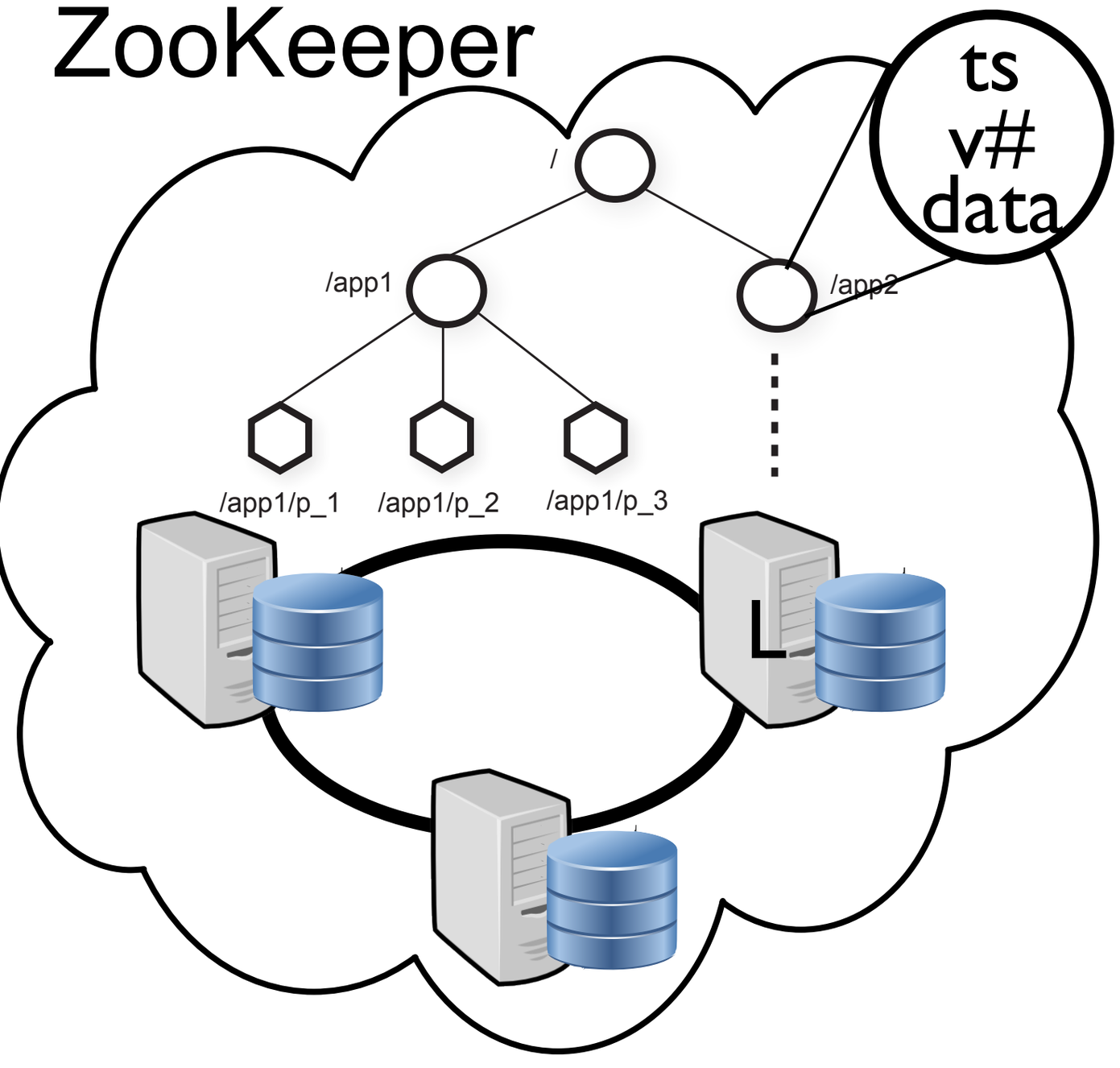
Lecture will focus on API

Zookeeper overview



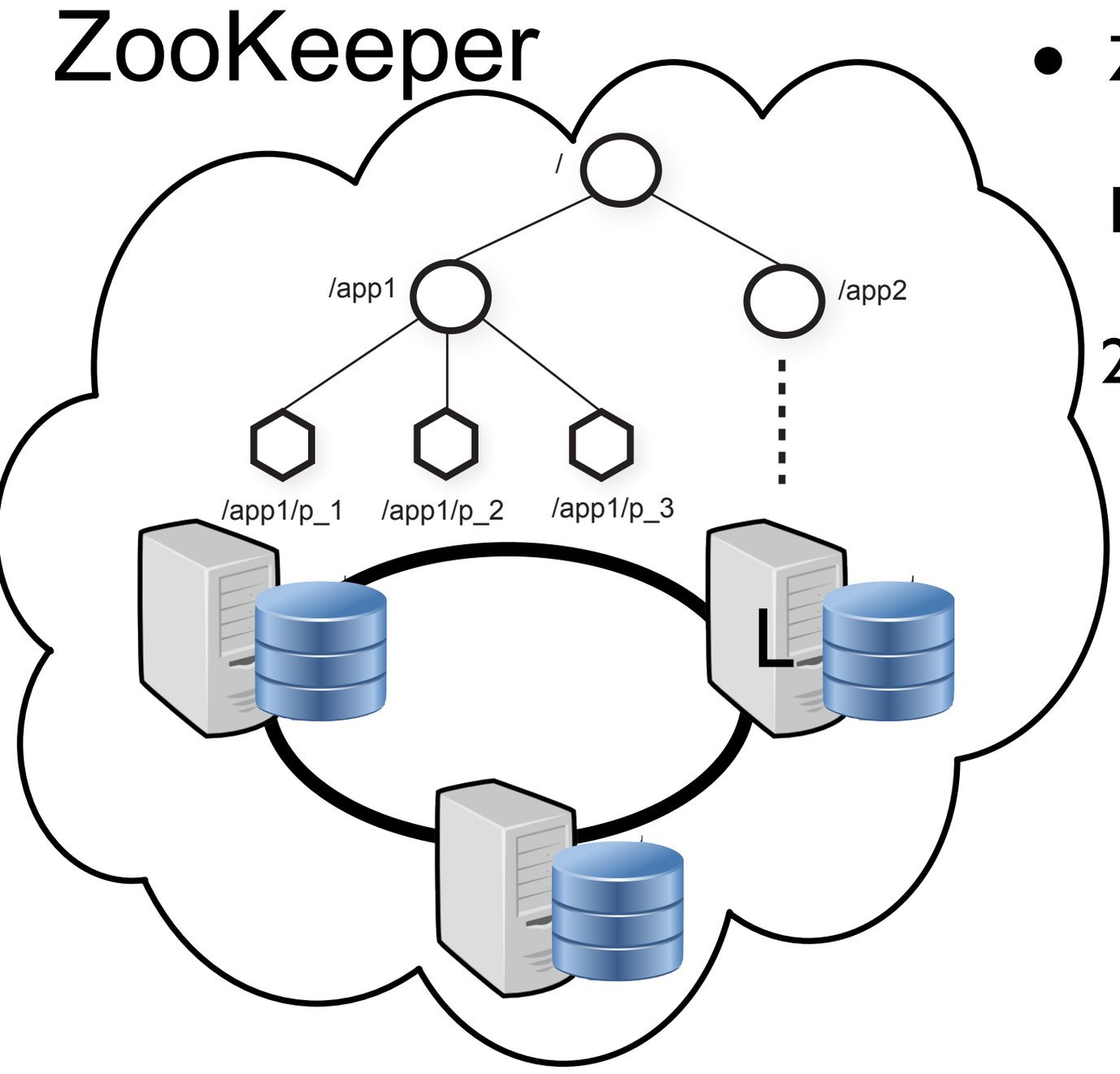
- A replicated service that exports a unified file system like tree/hierarchy of nodes (znodes)

Zookeeper overview



- A replicated service that exports a unified file system like tree/hierarchy of nodes (znodes)
  - Path uniquely names a znode
    - Nodes can have children (directory nodes)
    - Nodes have:
      - timestamp, version, and can store a single user data item (eg. 1M)

Zookeeper overview



- Znode : 3Types

1.Regular

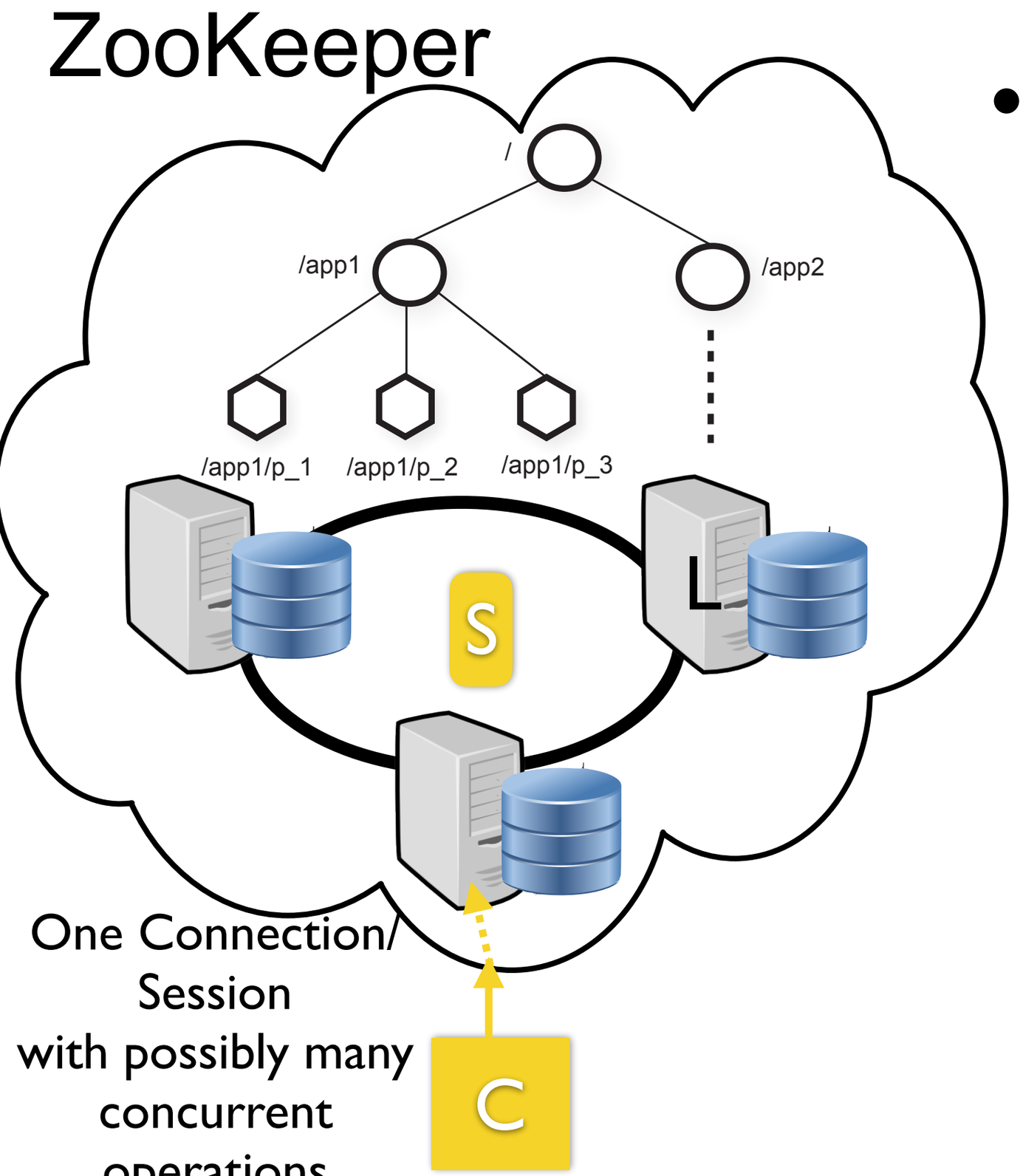
2.Sequential — name + seqno

- For a given directory sequence numbers are a monotonically increase value that is appended to names of children

- Can be used to ensure siblings are uniquely named and ordered by creation order.

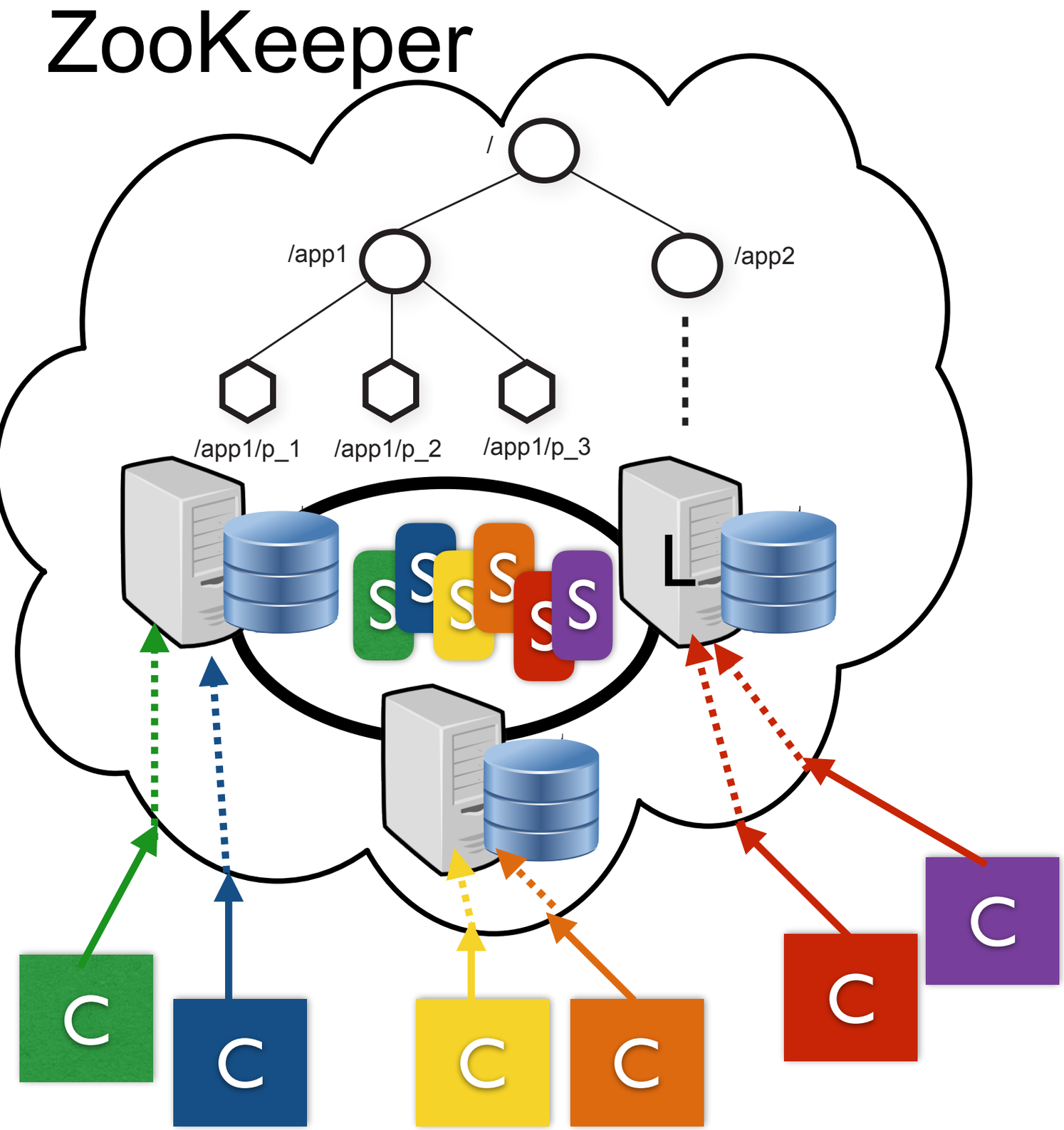
3.Ephemeral — coming up

Zookeeper overview



- Client connects to Zookeeper Service
  - Connection is to the system and represented by a session object that is maintained for the lifetime of the connection
  - At anytime there is a single server handling the client connection — all messages are to and from that server
  - Ephemeral — lifetime of znode tied to creating session

Zookeeper overview



- Normal operation many sessions — client connections distributed over Zookeeper servers
- Session allows a client to fail-over to another zookeeper service
- Sessions can timeout — used to track clients lifetime

Zookeeper API

- **create(path, data, flags);**
- **delete(path, version);**
- **exists(path, watch);**
- **getData(path, watch);**
- **setData(path, data, version);**
- **getChildren(path, watch);**
- **sync(path);**

http://zookeeper.apache.org

- Flags: regular vs ephemeral & sequential vs non-sequential
- Delete if znode.version == version
- Watches:
  - One time trigger
  - Async : but watch will always precede seeing the data
  - Data vs Child watches
- setData if znode.version = version, then update
- Sync : next couple of sides

Guarantees

Linearizable writes: all updates to the entire state are totally ordered — writes processed in order of arrival at the single leader

FIFO Client order: all requests from a single client are processed in client issue order — read on client sees prior write from the same client

Atomicity: updates are transactional — no partial results

Single System Image: same view regardless of the server **a client** connects to

Reliability:

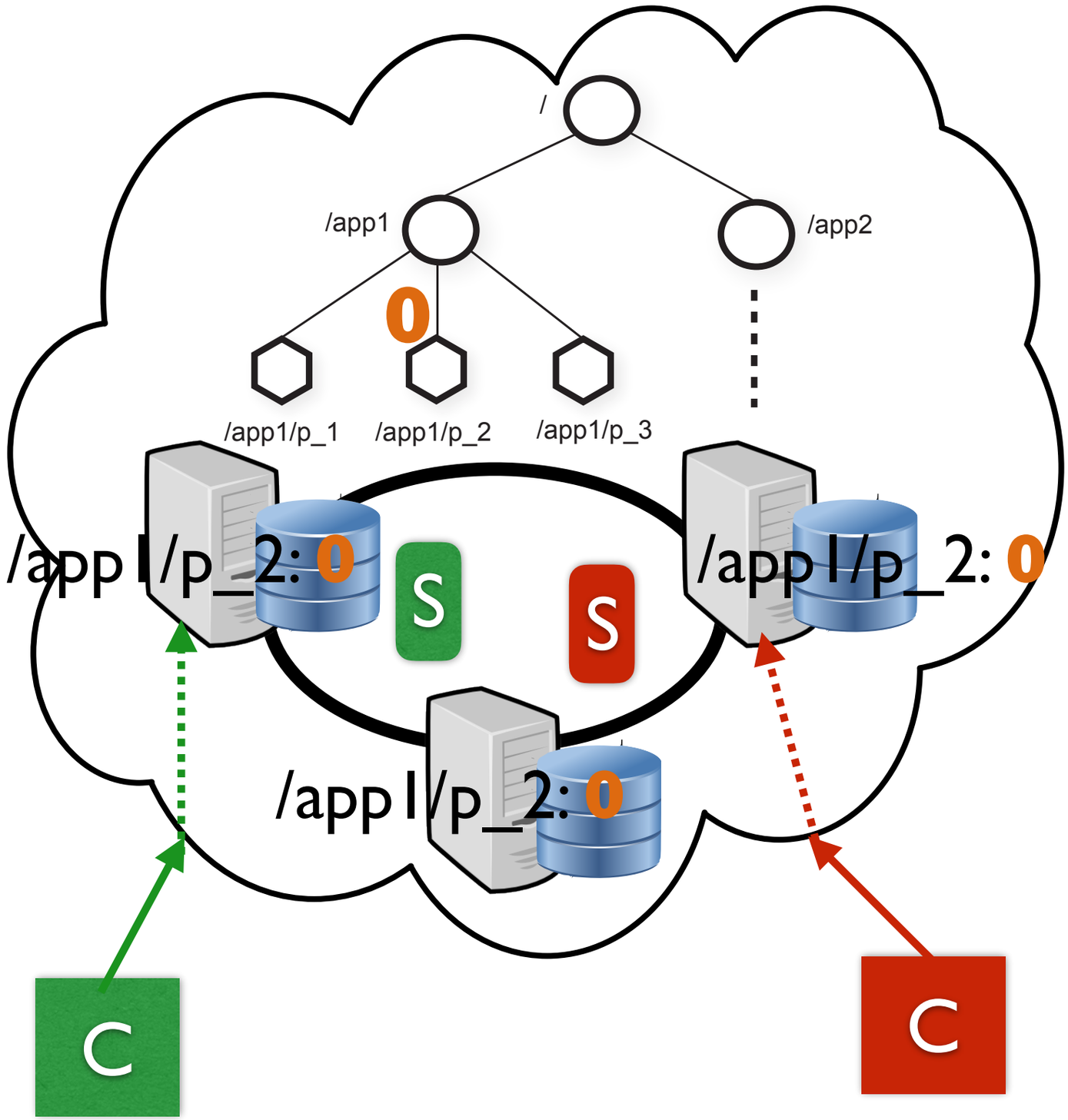
1 Successful update will persist — error unkown.

2 Will never be rolled back after returning value.

Timeliness: clients view will be up to date within a time bound or detect service outage

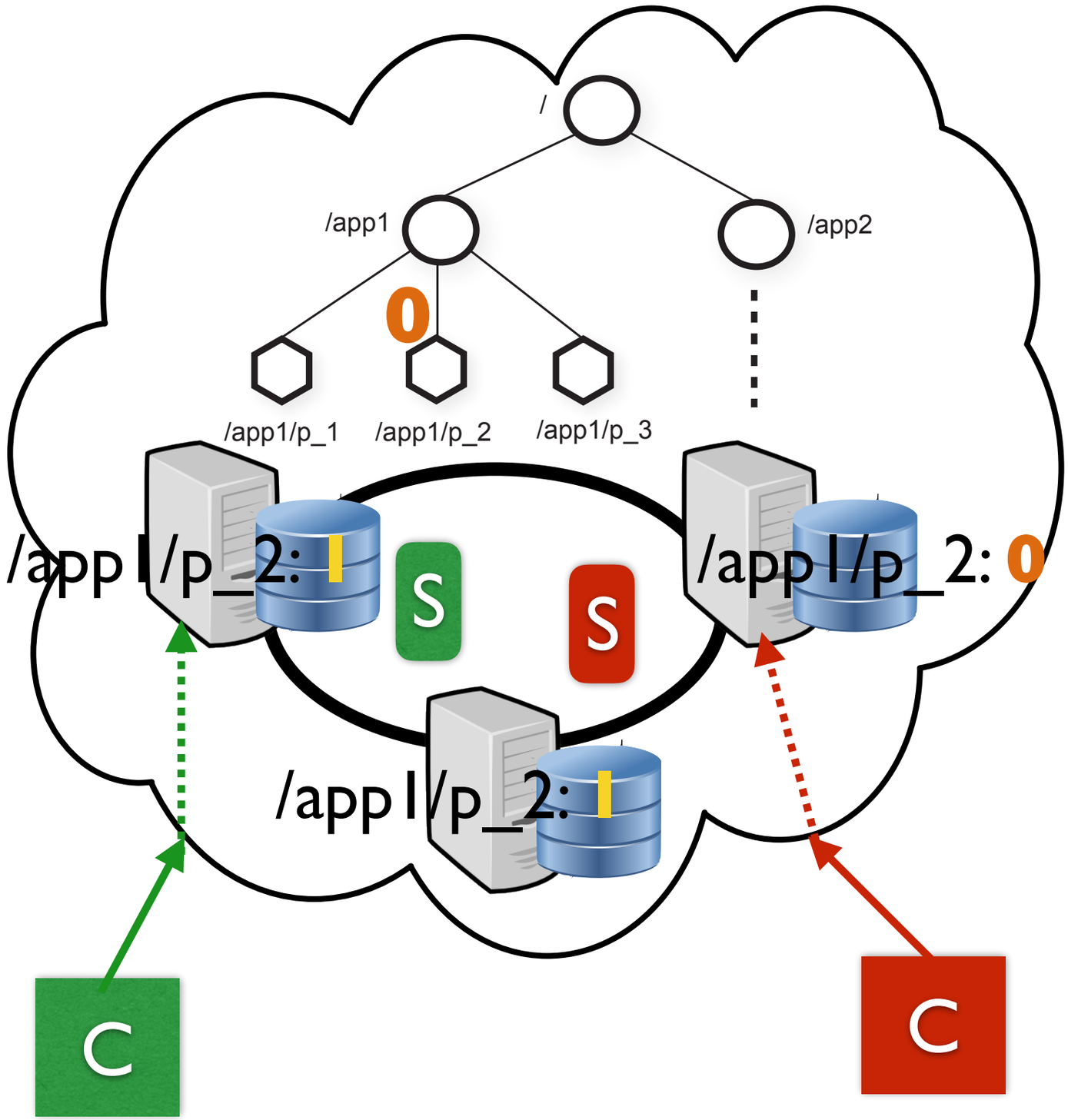
https://zookeeper.apache.org/doc/r3.1.2/zookeeperProgrammers.html

fast but possible stale reads

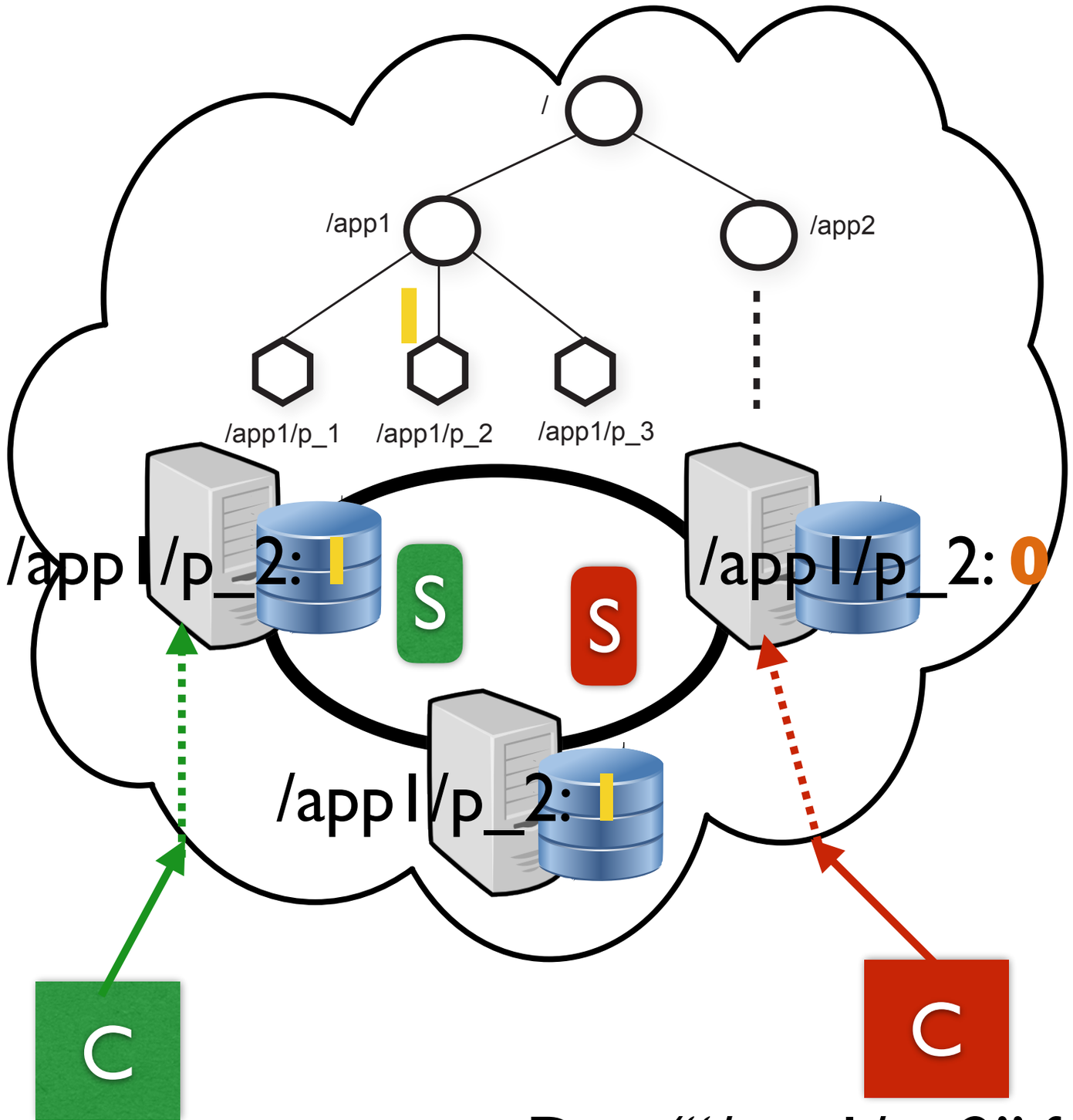


setData("/app1/p_2","1",-1)

fast but possible stale reads



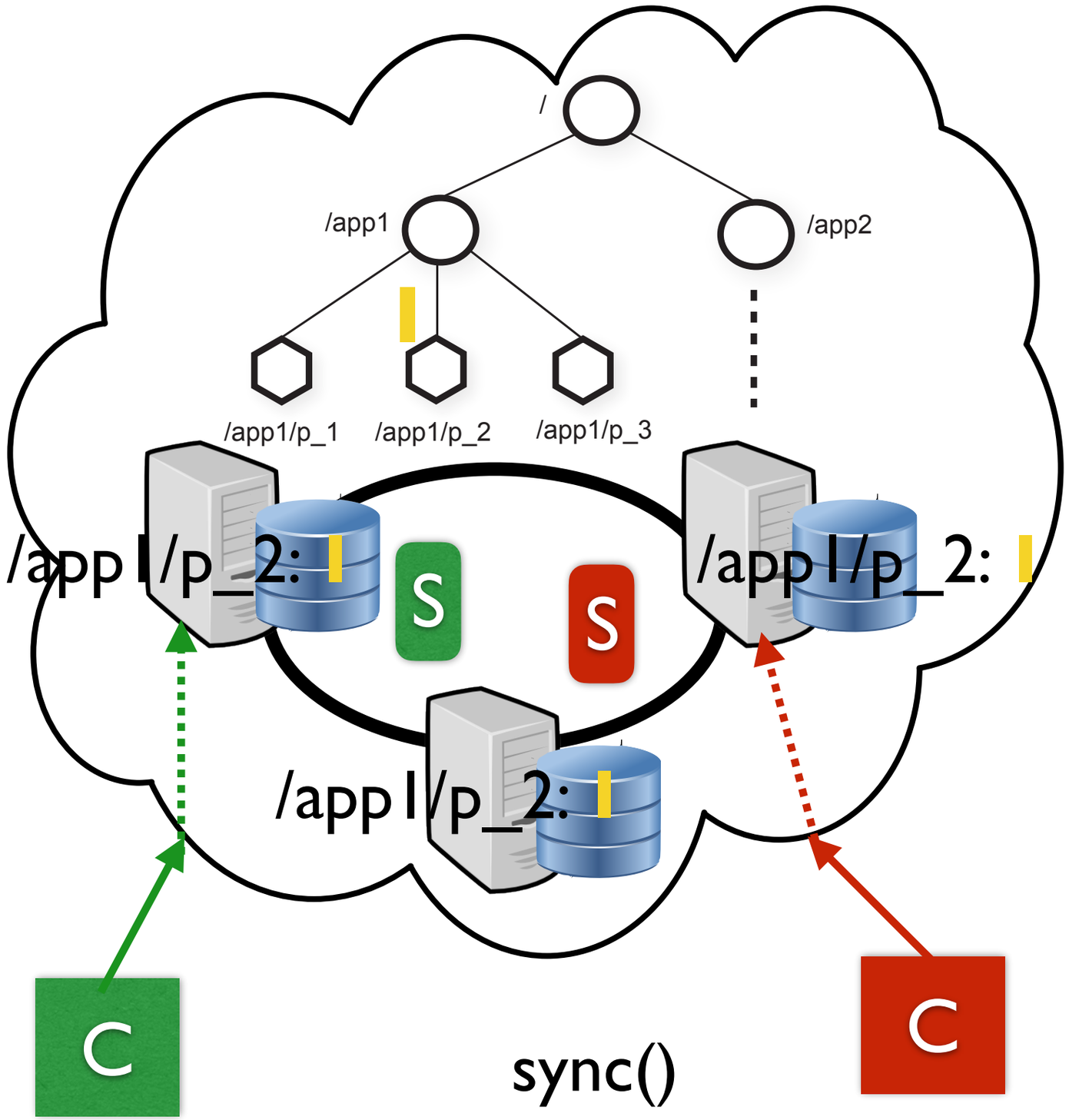
fast but possible stale reads



Fast Reads: processed locally from replica — can see stale data

getData("/app1/p_2",false) : returns 0

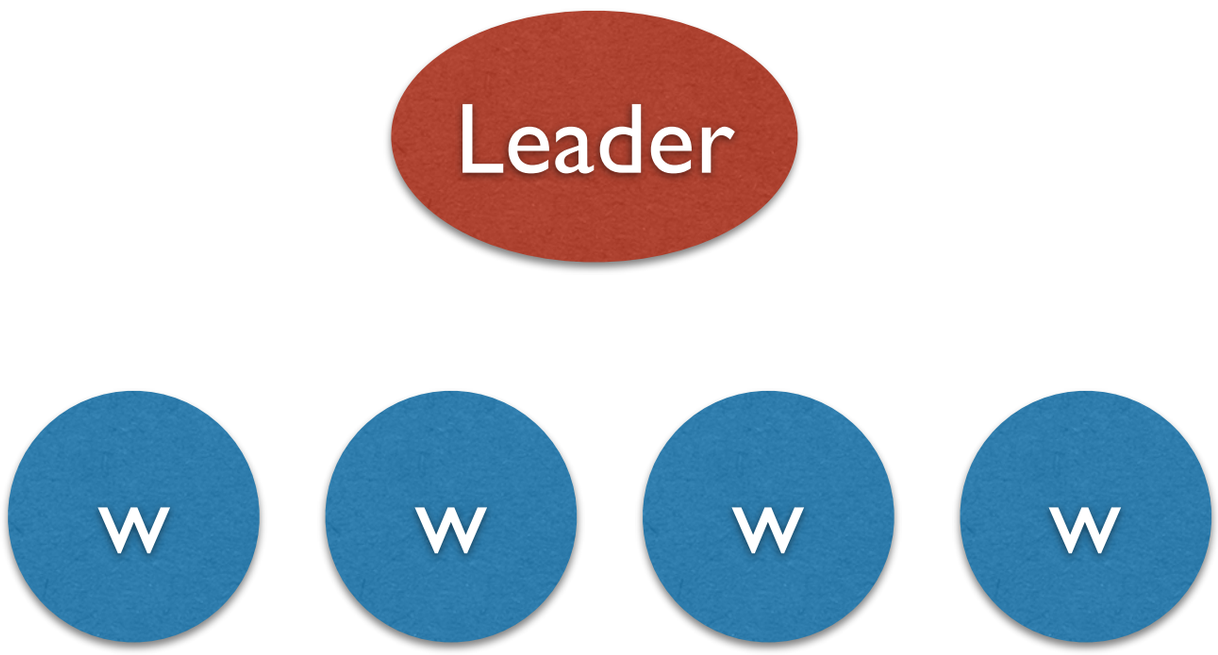
fast but possible stale reads



Can use sync before read to ensure that all prior writes have propagated

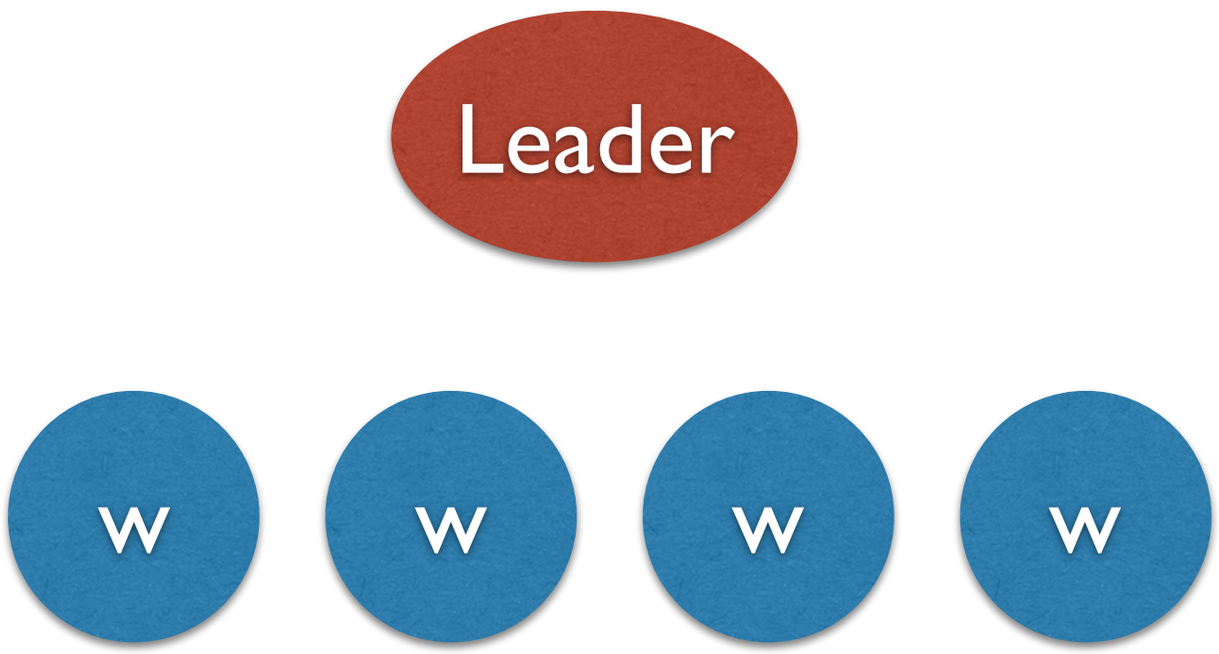
sync()
getData("/app1/p_2",false) : returns

Example from 2.3



If Leader dies : A new leader must take over and modify the 5000 config files (/myApp/config[1-5000]) before any worker can read the config

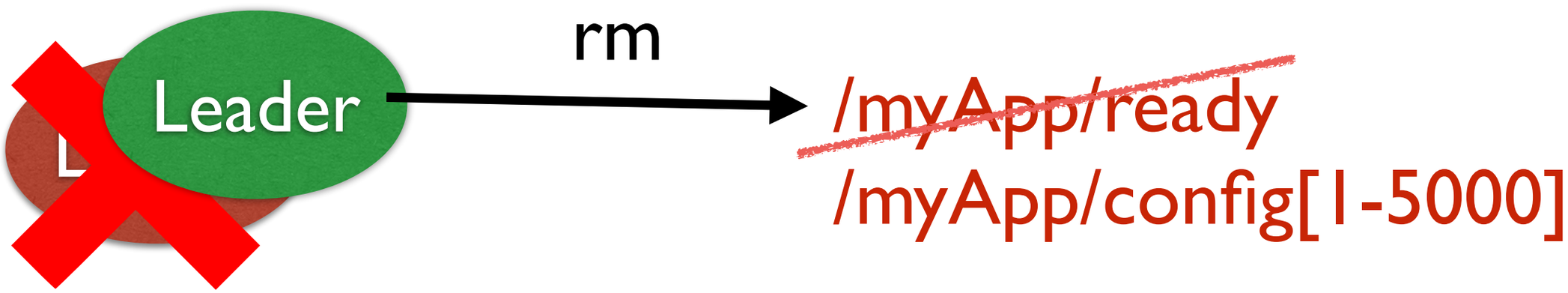
Example from 2.3

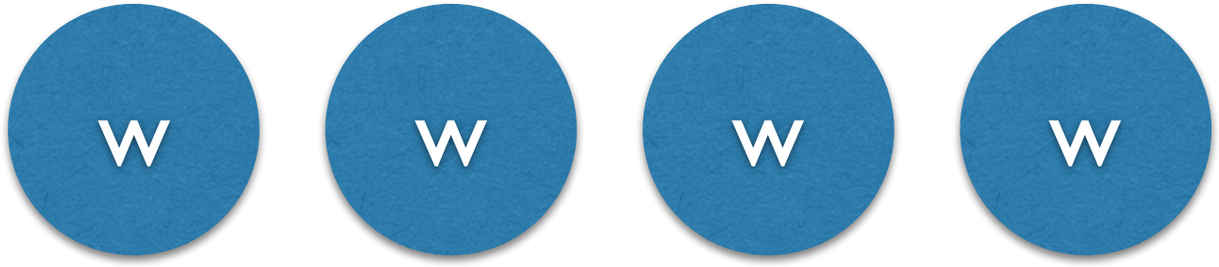


Add a /myApp/ready file

1. Processes will only use config if ready znode exists

Example from 2.3



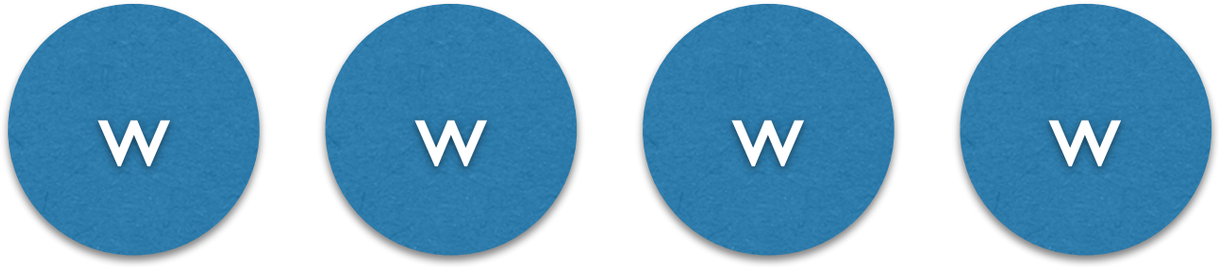


Add a /myApp/ready file

1. Processes will only use config if ready znode exists
2. New Leader deletes /myApp/ready to make config inaccessible

Example from 2.3

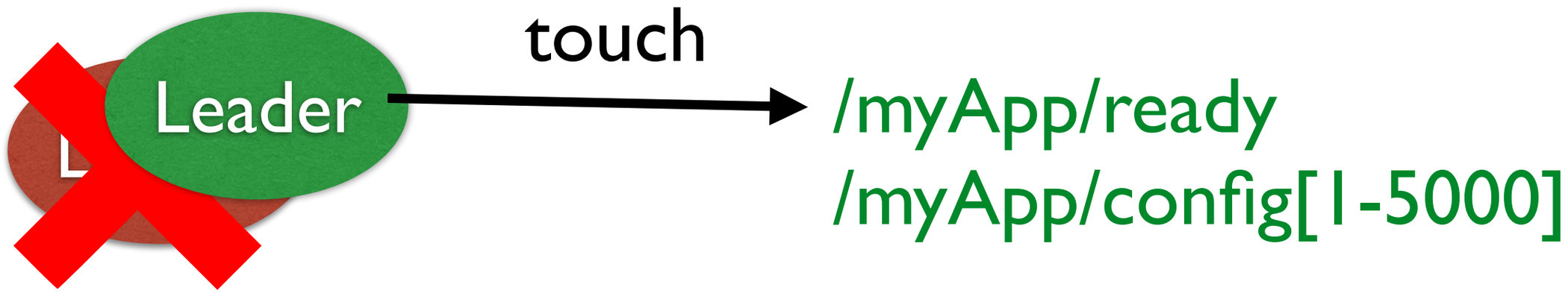
/myApp/config[1-5000]

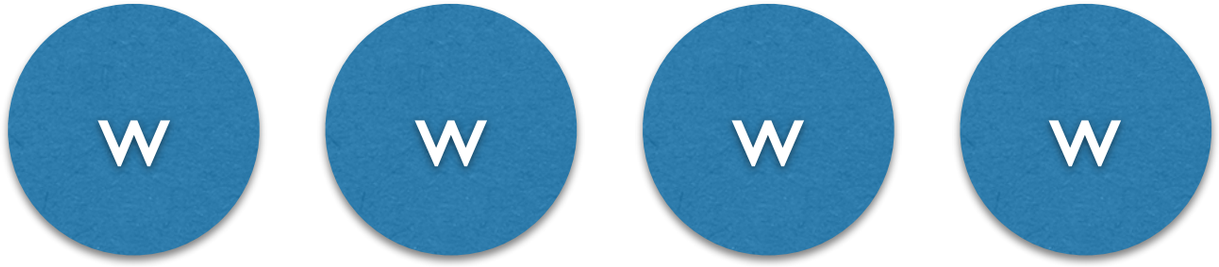


Add a /myApp/ready file

1. Processes will only use config if ready znode exists
2. New Leader deletes /myApp/ready to make config inaccessible
3. New Leader updates config metadata

Example from 2.3



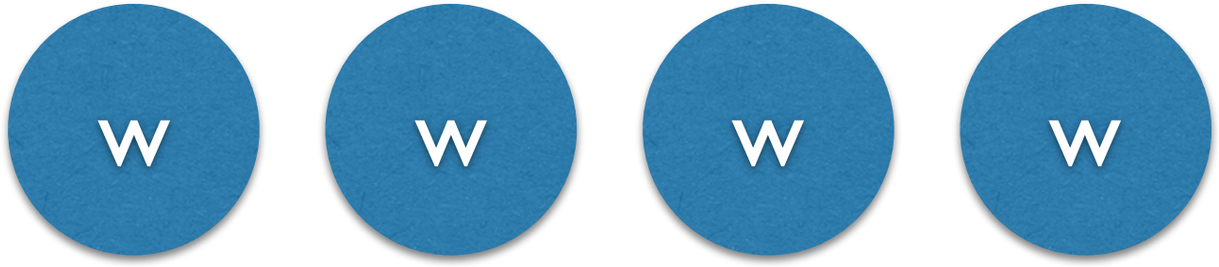


Add a /myApp/ready file

1. Processes will only use config if ready file exists
2. New Leader deletes /myApp/ready to make config inaccessible
3. New Leader updates config files
4. Creates /myApp/ready to release config

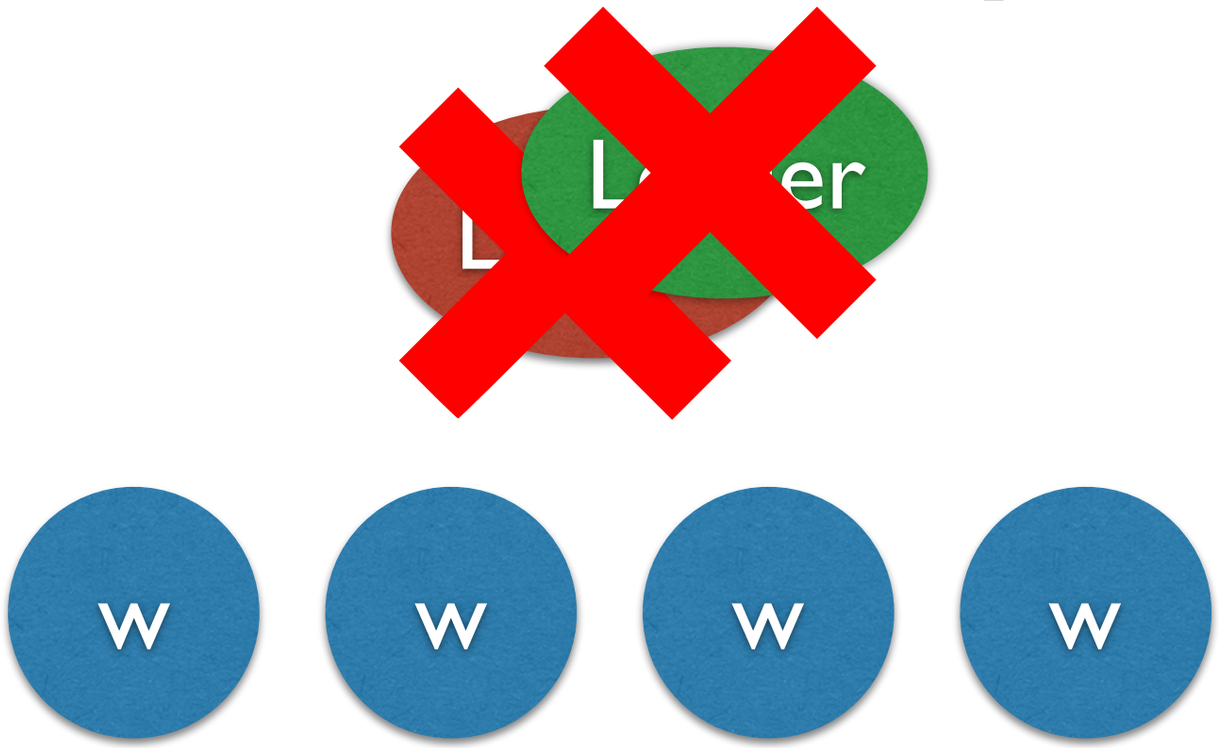
Example from 2.3

/myApp/ready
/myApp/config[1-5000]



Key point: “pipeline” updates and creation of ready … ordering guarantees ensures that ready will not be written before all config changes.
“if a process sees the ready znode, it must also see all the configuration changes…”

Example from 2.3

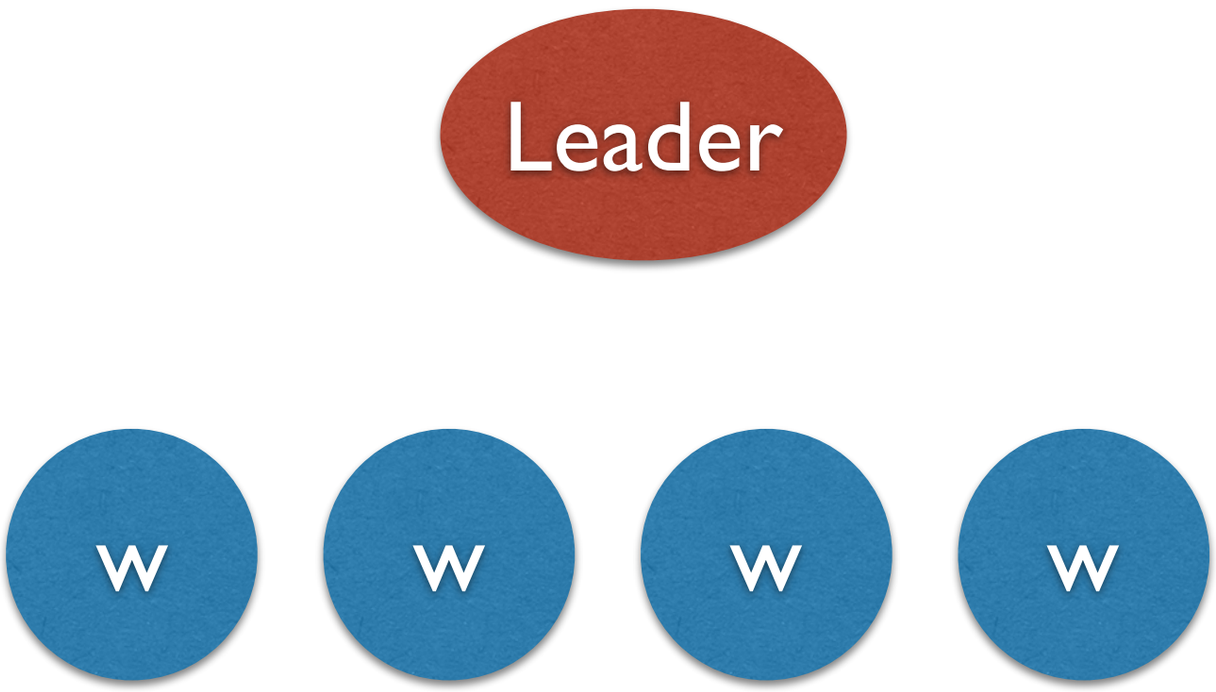


/myApp/config[1-5000]

config is mixed up

If the new leader dies before ready is created …
SAFE : no one will accidentally use partially updated config

Example from 2.3



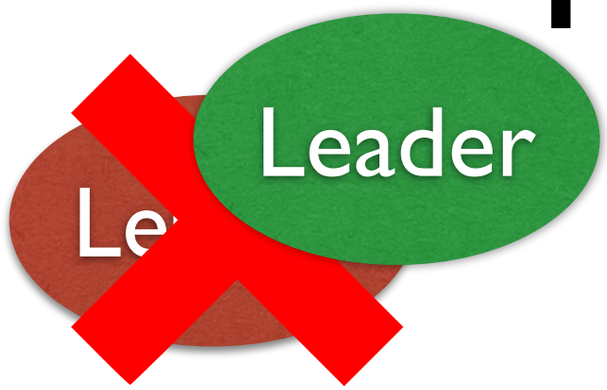
/myApp/ready
/myApp/config[1-5000]

But consider:

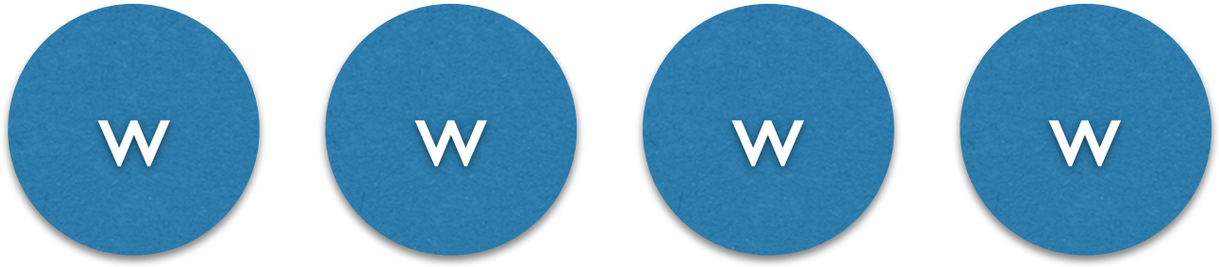
w: read ready

w: starts reading config

Example from 2.3



/myApp/config[1-5000]



But consider:

w: read ready

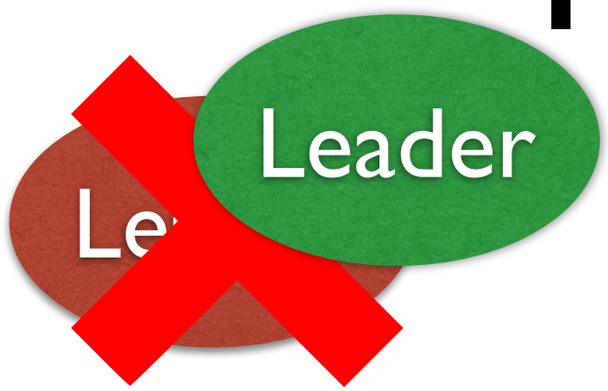
w: starts reading config

new leader

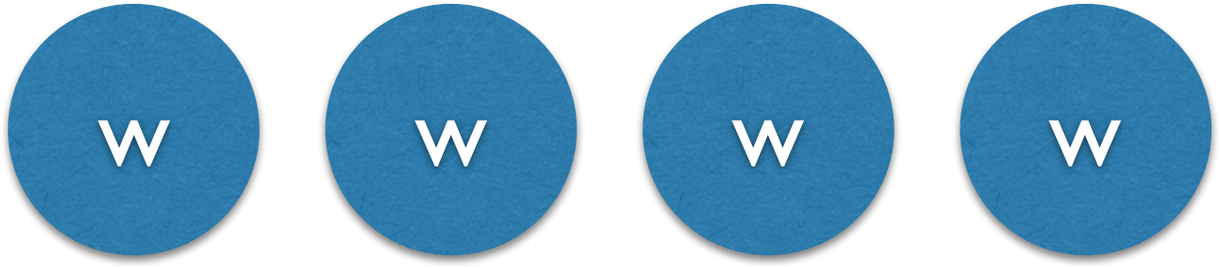
rm ready

update

Example from 2.3

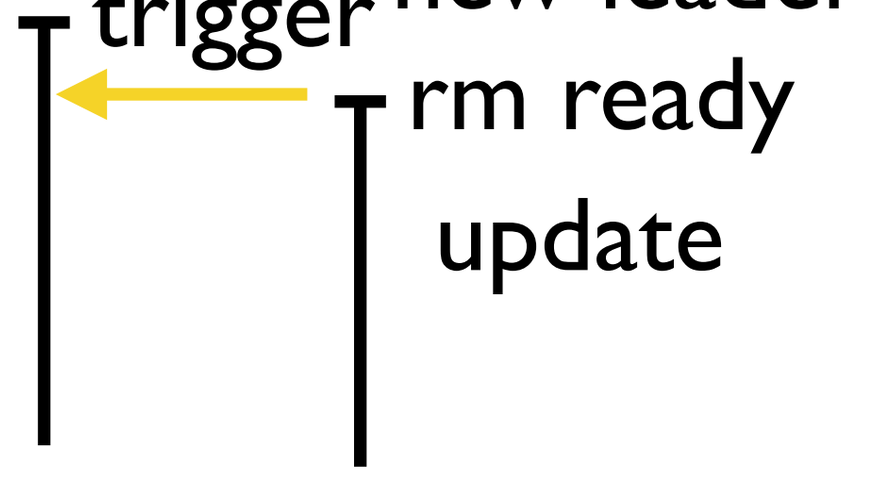


/myApp/config[1-5000]



w: read ready with watch

w: starts reading config



Lock Example 1: Slow — Thundering Herd

Lock

```
1 r = create("/app/lock", "", EPHEMERAL);
2 if (r) return
3 read("/app/lock", watch=TRUE);
4 wait for watch event
5 goto 1
```

Unlock

```
1 delete("/app/lock");
```

Or the client holding the lock dies (session timeout)

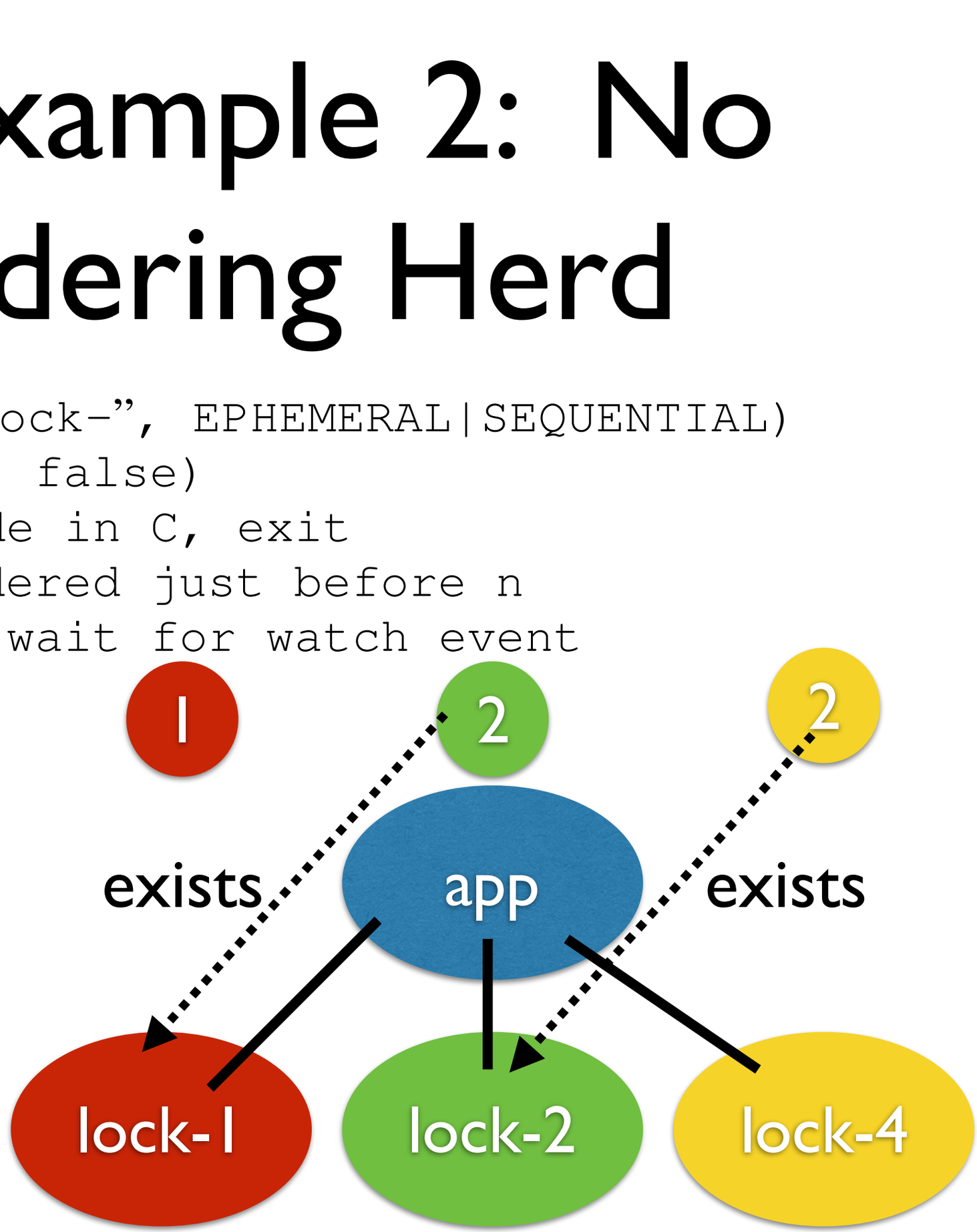
Lock Example 2: No Thundering Herd

Lock

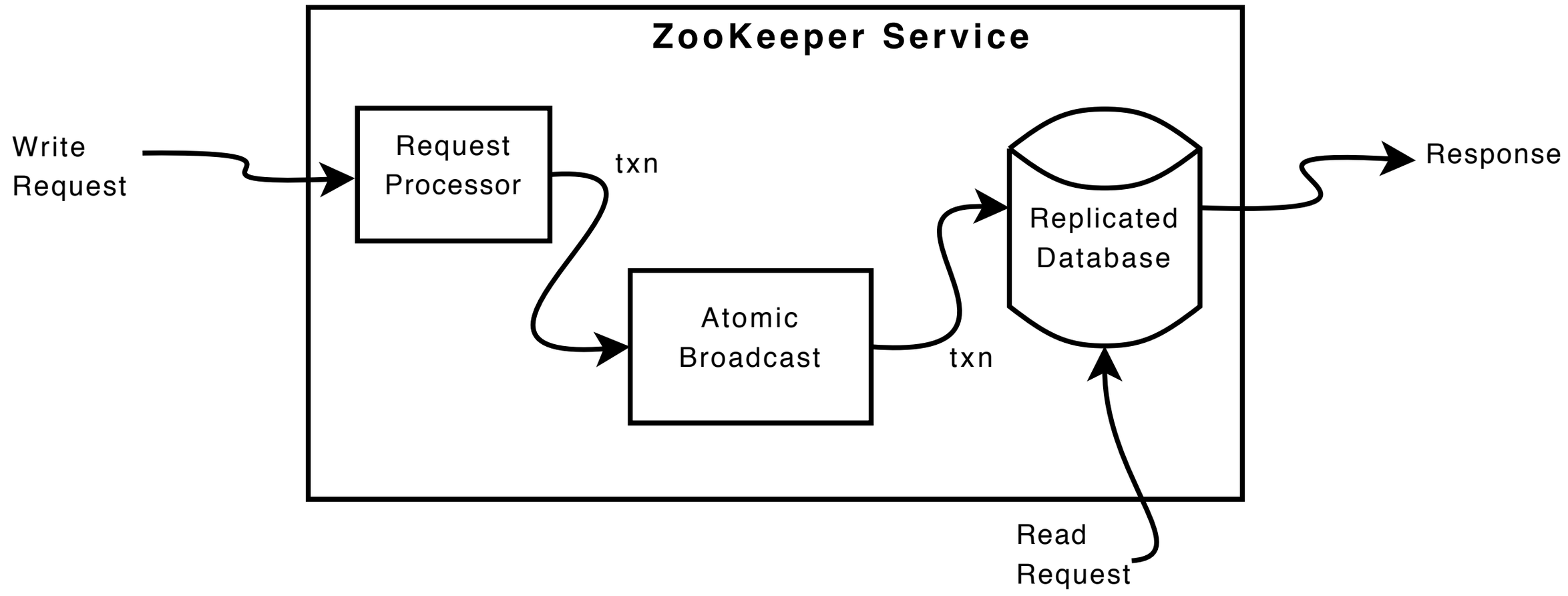
```
1 n = create(l + "/lock-", EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 if n is lowest znode in C, exit
4 p = znode in C ordered just before n
5 if exists(p, true) wait for watch event
6 goto 2
```

Unlock

```
1 delete(n)
```



Implementation



ZooKeeper as a building block

- Zookeeper simplifies building applications but is not an end-to-end solution
  - Plenty of hard problems left for apps
- With Zookeeper, at least master is fault tolerant and, won't run into split-brain problem even though it has replicated servers