Coordinating distributed systems

Marko Vukolić

Distributed Systems and Cloud Computing

Today

- **Distributed systems coordination**
- **Apache Zookeeper**
  - Simple, high performance kernel for building distributed coordination primitives
  - Zookeeper is not a specific coordination primitive per se, but a platform/API for building different coordination primitives

Zookeeper: Agenda

- Motivation and Background
- Coordination kernel
- Semantics
- Programming Zookeeper
- Internal Architecture

Why do we need coordination?

EURECOM

Coordination primitives

- Semaphores
- Locks
- Queues
- Leader election
- Group membership
- Barriers
- Configuration management
- ….

Why is coordination difficult?

- **Coordination among multiple parties involves agreement among those parties**
- **Agreement ←→ Consensus ←→ Consistency**
- **FLP impossibility result + CAP theorem**
  - Agreement is difficult in a dynamic asynchronous system in which processes may fail or join/leave

How do we go about coordination?

- **One approach**
  - For each coordination primitive build a specific service

- **Some recent examples**
  - Chubby, Google [*Burrows et al, USENIX OSDI, 2006*]
    - Lock service
  - Centrifuge, Microsoft [Adya et al, USENIX NSDI, 2010]
    - Lease service

But there is a lot of applications out there

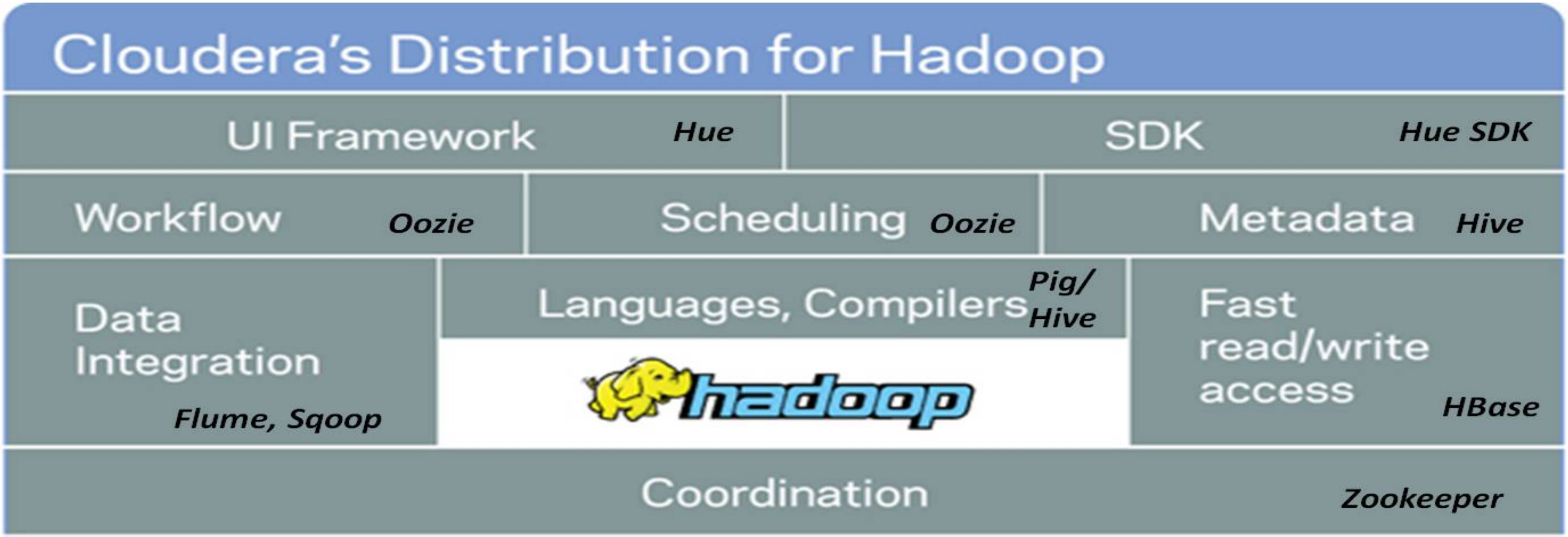
- **How many distributed services need coordination?**
  - Amazon/Google/Yahoo/Microsoft/IBM/…
- **And which coordination primitives exactly?**
  - Want to change from Leader Election to Group Membership? And from there to Distributed Locks?
  - There are also common requirements in different coordination services
    - Duplicating is bad and duplicating poorly even worse
    - Maintenance?

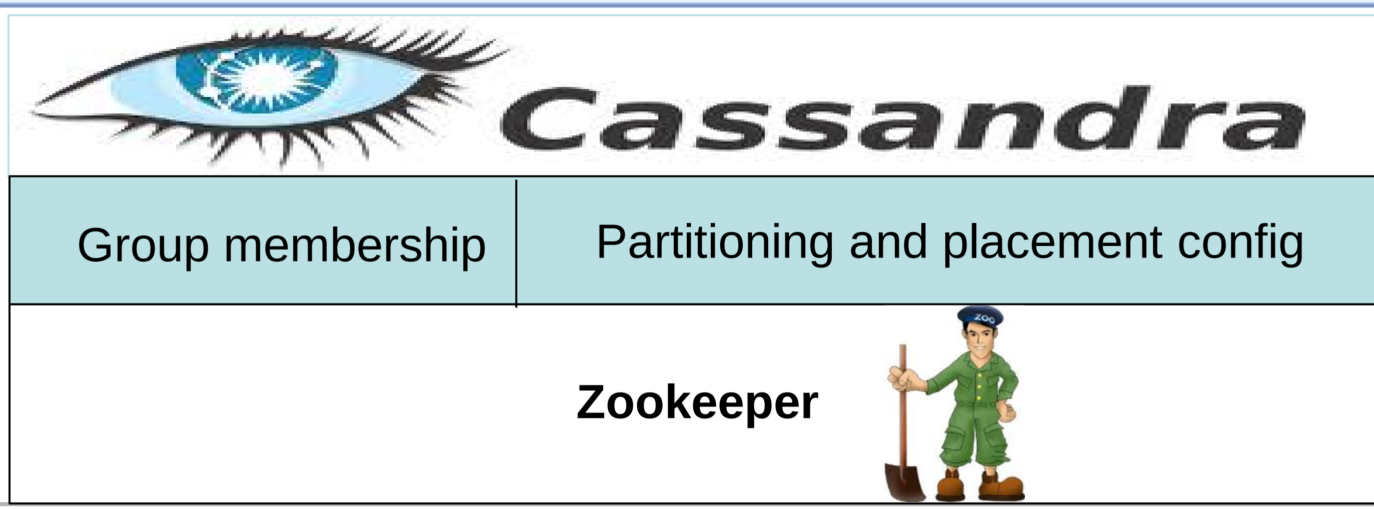
How do we go about coordination?

- **Alternative approach**
  - A coordination service
  - Develop a set of lower level primitives (i.e., an API) that can be used to implement higher-level coordination services
  - Use the coordination service API across many applications

- **Example: Apache Zookeeper**

We already mentioned Zookeeper





Origins

- **Developed initially at Yahoo!**
- **On Apache since 2008**
  - Hadoop subproject
- **Top Level project since Jan 2011**
  - ***zookeeper.apache**.org*

Zookeeper: Agenda

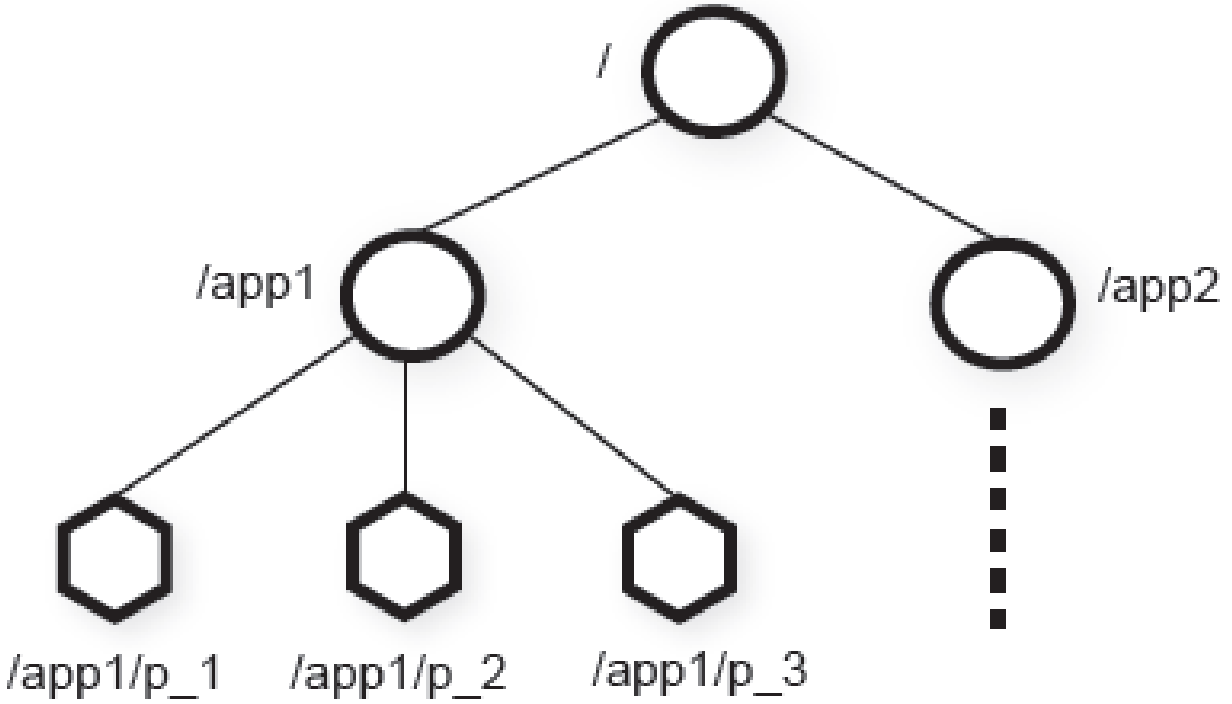
- **Motivation and Background**
- **Coordination kernel ←**
- **Semantics**
- **Programming Zookeeper**
- **Internal Architecture**

Zookeeper overview

- **Client-server architecture**
  - Clients access Zookeeper through a client API
  - Client library also manages network connections to Zookeeper servers

- **Zookeeper data model**
  - Similar to file system
  - Clients see the abstraction of a set of data nodes (***znodes)***
  - Znodes are organized in a hierarchical namespace that resembles customary file systems

Hierarchical znode namespace



Types of Znodes

- **Regular znodes**
  - Clients manipulate regular znodes by creating and deleting them explicitly
  - (We will see the API in a moment)

- **Ephemeral znodes**
  - Can manipulate them just as regular znodes
  - However, ephemeral znodes can be removed by the system when the session that creates them terminates
  - Session termination can be deliberate or due to failure

Data model

- **In brief, it is a file system with a simplified API**
- **Only full reads and writes**
  - No appends, inserts, partial reads
- **Znode hierarchical namespace**
  - Think of directories that may also contain some payload data
- **Payload not designed for application data storage but for application metadata storage**
- **Znodes also have associated version counters and some metadata (e.g., flags)**

Sessions

- **Client connects to Zookeeper and initiates a session**
  - Sessions enables clients to move transparently from one server to another
  - Any server can serve client's requests

- **Sessions have timeouts**
  - Zookeeper considers client faulty if it does not hear from client for more than a timeout
  - This has implications on ephemeral znodes

Client API

- ***create(znode, data, flags)***
  - *Flags denote the type of the znode:*
    - *REGULAR, EPHEMERAL, SEQUENTIAL*
    - *SEQUENTIAL flag: a monotonically increasing value is appended to the name of znode*
  - *znode must be addressed by giving a full path in all operations (e.g., '/app1/foo/bar')*
  - *returns znode path*
- ***delete(znode, version)***
  - *Deletes the znode if the version is equal to the actual version of the znode*
  - *set version = -1 to omit the conditional check (applies to other operations as well)*

Client API (cont'd)

- ***exists(znode, watch)***
  - *Returns true if the znode exists, false otherwise*
  - *watch flag enables a client to set a watch on the znode*
  - *watch is a subscription to receive an information from the Zookeeper when this znode is changed*
  - *NB: a watch may be set even if a znode does not exist*
    - ☞ *The client will be then informed when a znode is created*
- ***getData(znode, watch)***
  - *Returns data stored at this znode*
  - *watch is not set unless znode exists*

Client API (cont'd)

- ***setData(znode, data, version)***
  - *Rewrites znode with data, if version is the current version number of the znode*
  - *version = -1 applies here as well to omit the condition check and to force setData*
- ***getChildren(znode, watch)***
  - *Returns the set of children znodes of the znode*
- ***sync()***
  - *Waits for all updates pending at the start of the operation to be propagated to the Zookeeper server that the client is connected to*

API operation calls

- **Can be synchronous or asynchronous**

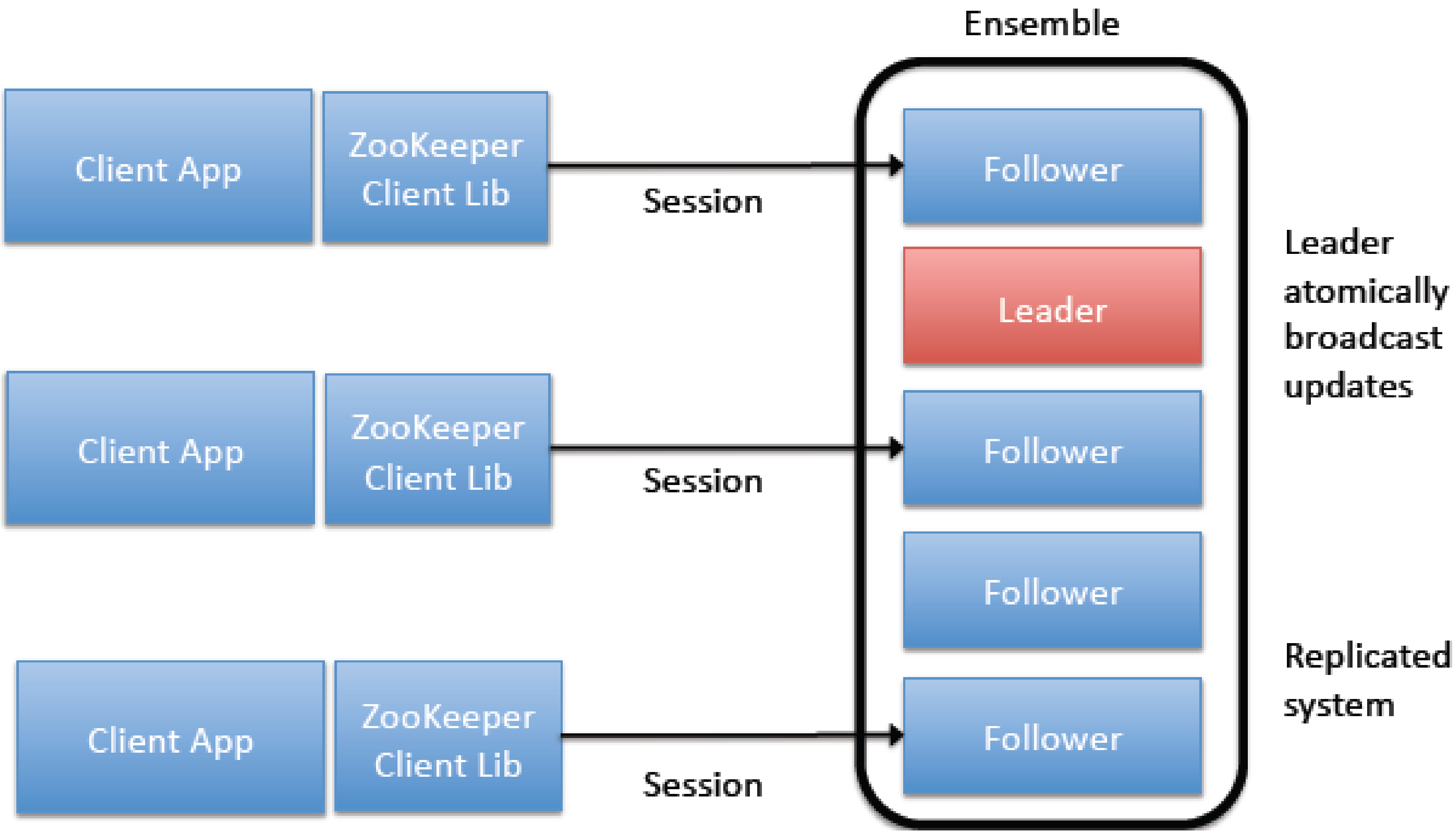
- **Synchronous calls**
  - A client blocks after invoking an operation and waits for an operation to respond
  - No concurrent calls by a single client

- **Asynchronous calls**
  - Concurrent calls allowed
  - A client can have multiple outstanding requests

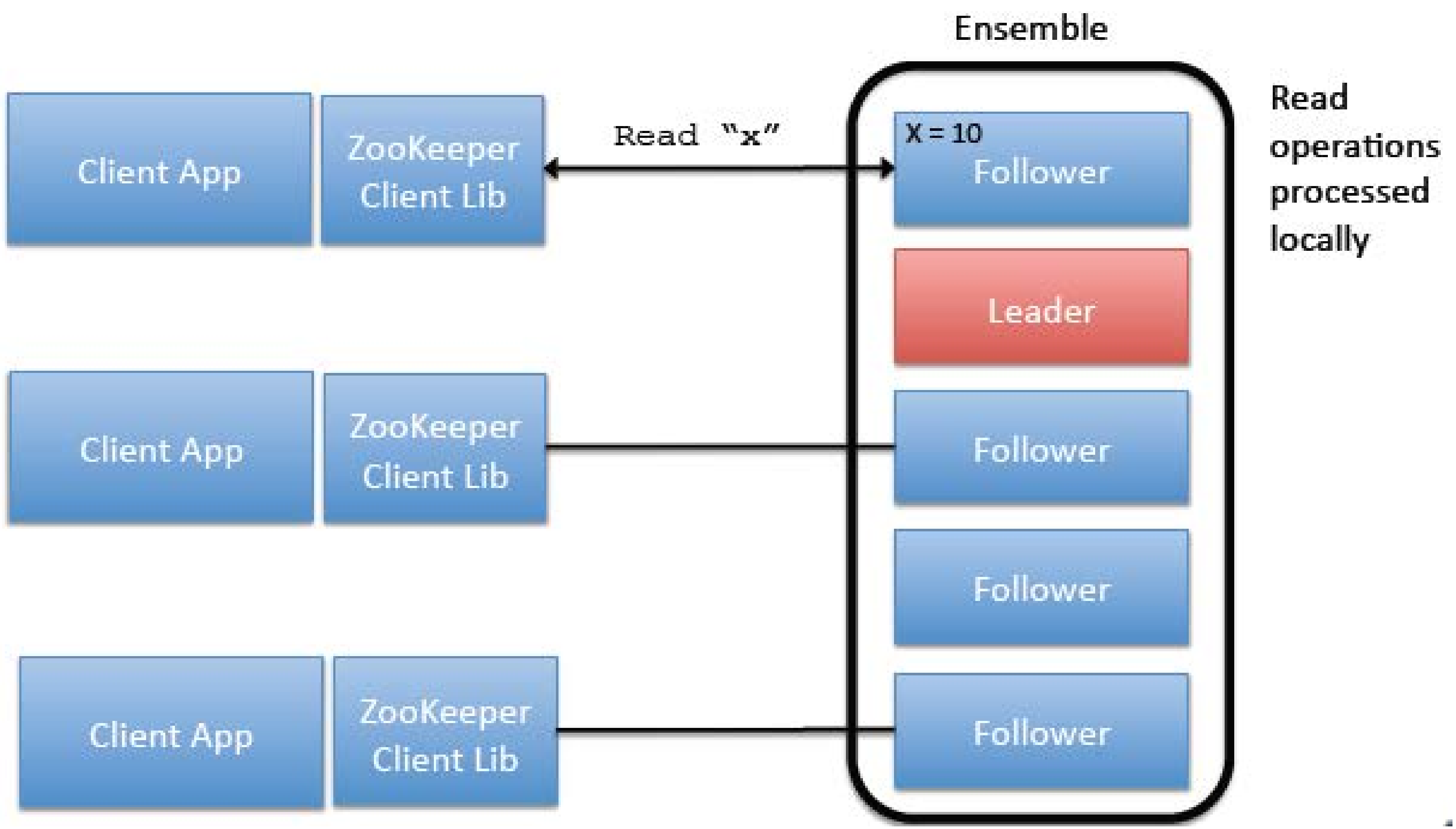
Convention

- **Update/write operations**
  - Create, setData, sync, delete
- **Reads operations**
  - exists, getData, getChildren

Session overview

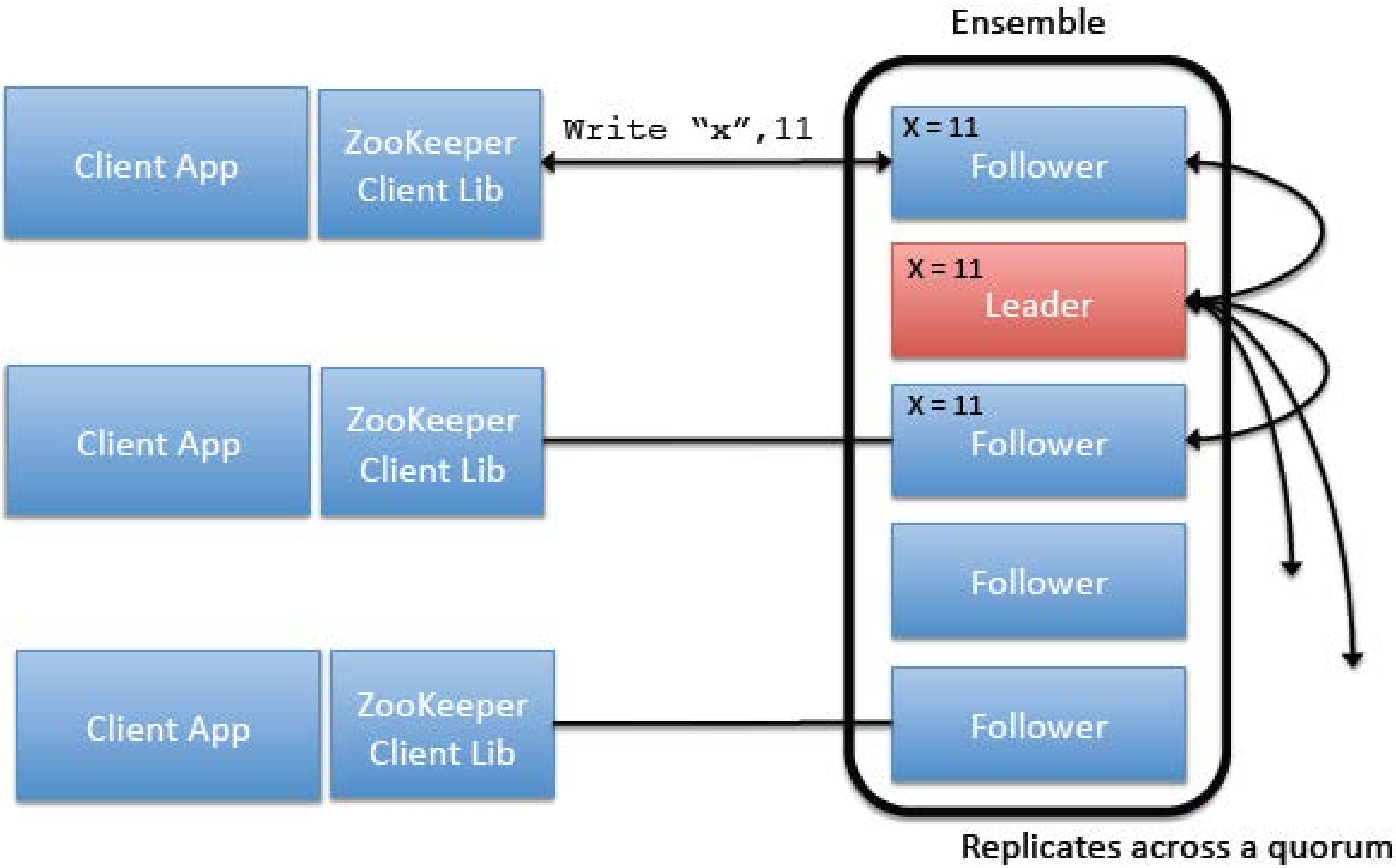


Read operations



EURECOM

Write operations



EURECOM

Atomic broadcast

- A.k.a. total order broadcast
- Critical synchronization primitive in many distributed systems
- Fundamental building block to building replicated state machines

Atomic Broadcast (safety)

- **Total Order property**
  - Let m and m’ be any two messages.
  - Let pi be any correct process that delivers m without having delivered m’
  - Then no correct process delivers m’ before m
- **Integrity (a.k.a. No creation)**
  - No message is delivered unless it was broadcast
- **No duplication**
  - No message is delivered more than once
  - (Zookeeper Atomic Broadcast – ZAB deviates from this)

State machine replication

- **Think of, e.g., a database (RDBMS)**
  - Use atomic broadcast to totally order database operations/transactions
- **All database replicas apply updates/queries in the same order**
  - Since database is deterministic, the state of the database is fully replicated
- **Extends to any (deterministic) state machine**

Consistency of total order

- **Very strong consistency**
- **“Single-replica” semantics**

Zookeeper: Agenda

- Motivation and Background
- Coordination kernel
- Semantics ←
- Programming Zookeeper
- Internal Architecture

Zookeeper semantics

- **CAP perspective: Zookeeper is in CP**
  - It guarantees consistency
  - May sacrifice availability under system partitions (strict quorum based replication for writes)

- **Consistency (safety)**
  - *Linearizable writes:* all writes are linearizable
  - *FIFO client order:* all requests from a given client are executed in the order they were sent by the client
    - ☞ Matters for asynchronous calls

Zookeeper Availability

- **Wait-freedom**
  - All operations invoked by a correct client eventually complete
  - Under condition that a quorum of servers is available

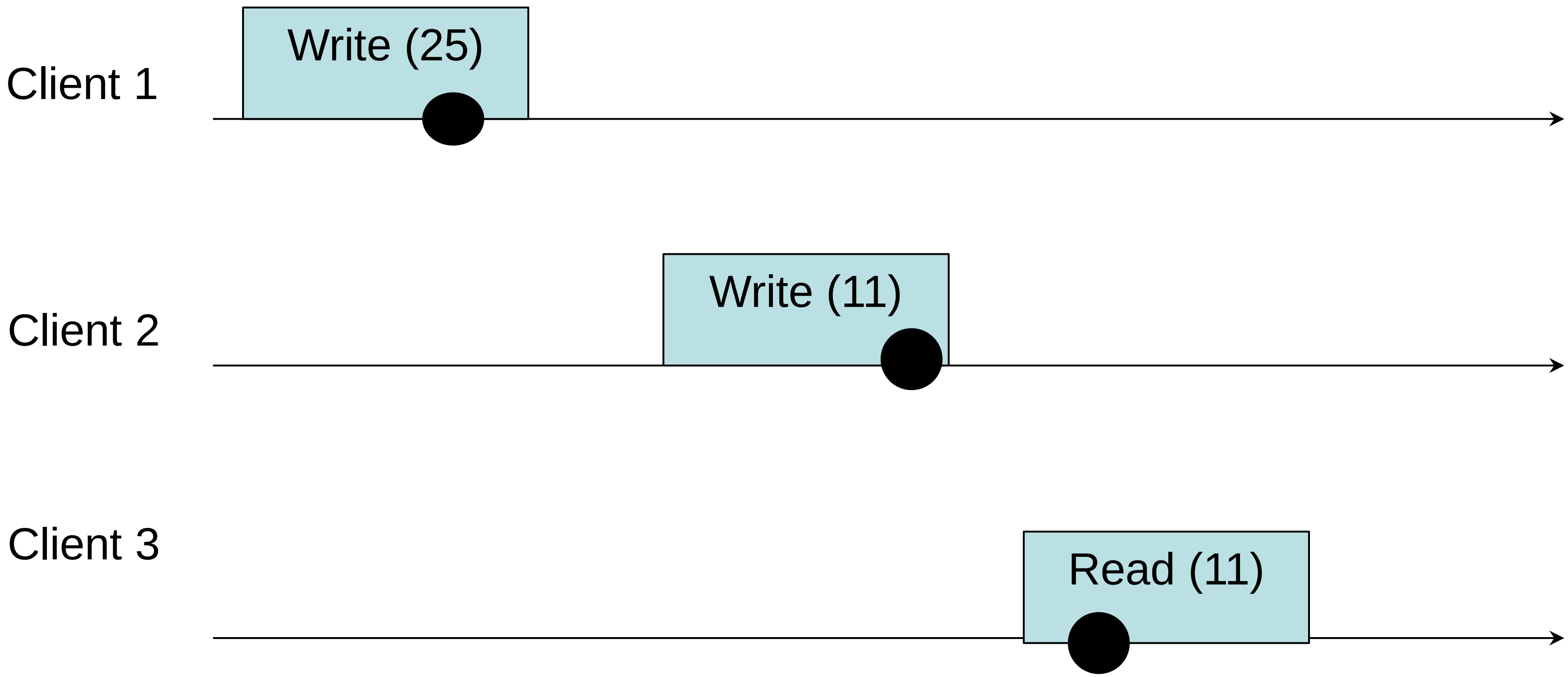
- **Zookeeper uses no locks although it can implement locks**

Zookeeper consistency vs. Linearizability

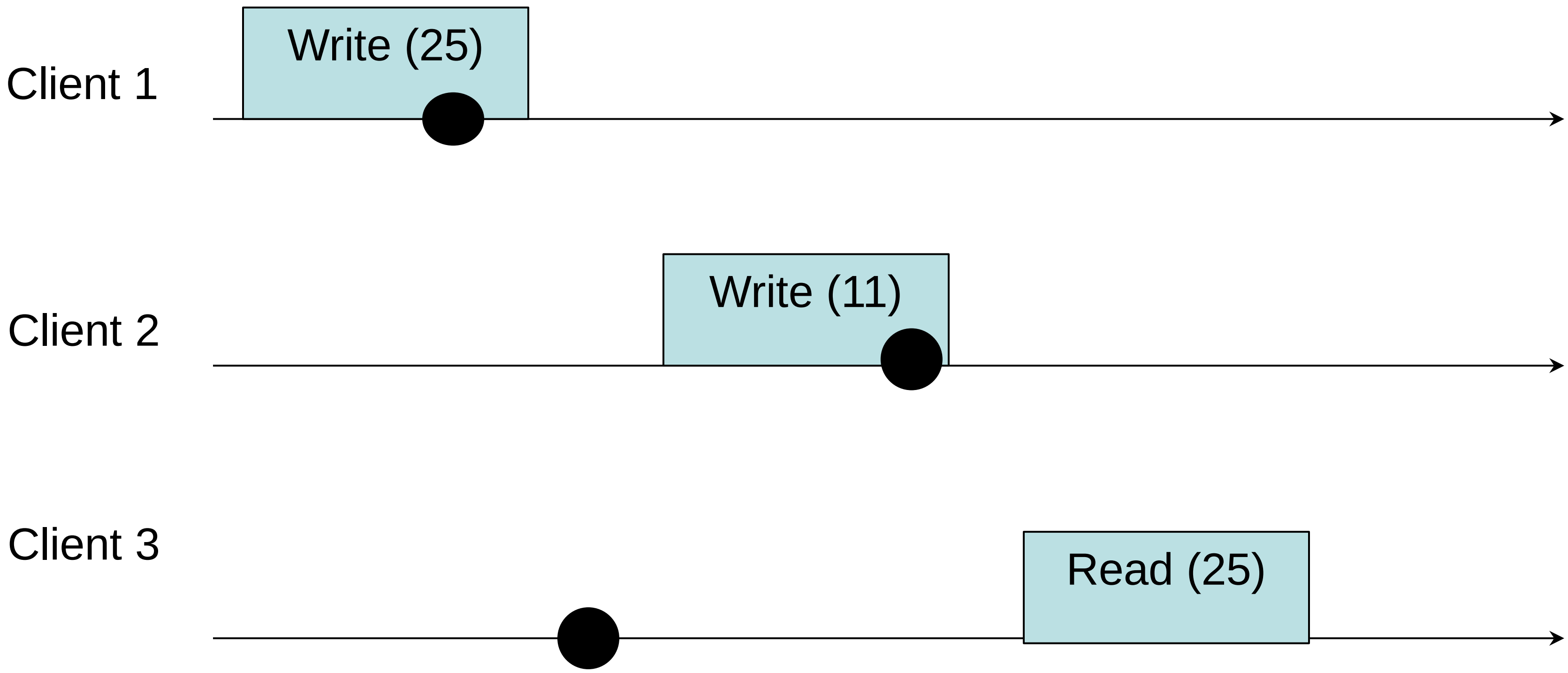
- **Linearizability**
  - All operations appear to take effect in a single, indivisible time instant between invocation and response

- **Zookeeper consistency**
  - Writes are linearizable
  - Reads might not be
    - ☞ To boost performance, Zookeeper has local reads
    - ☞ A server serving a read request might not have been a part of a write quorum of some previous operation
    - ☞ ➔ A read might return a stale value

Linearizability



Zookeeper



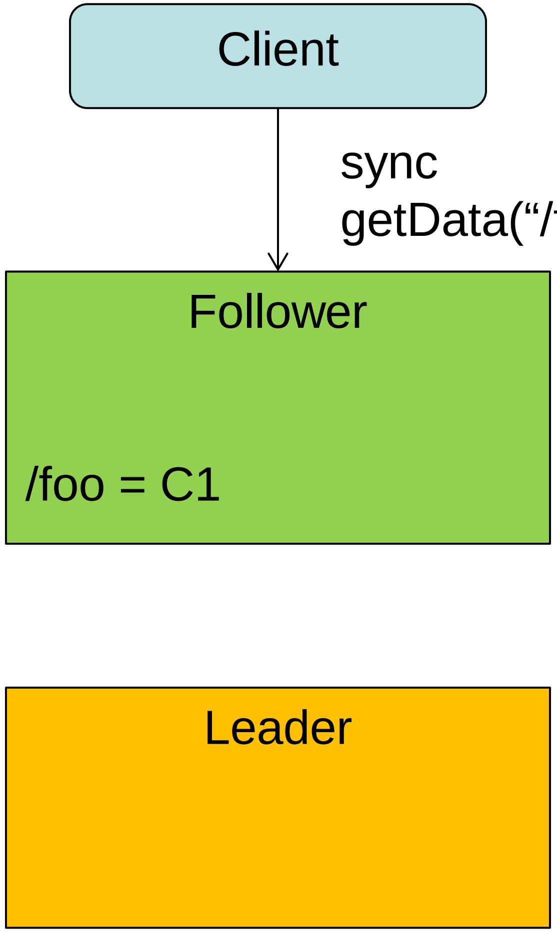
EURECOM

Is this a problem?

- **Depends what the application needs**
  - May cause inconsistencies in synchronization if not careful
- **Despite this, Zookeeper API is a universal object → its consensus number is ∞**
    - ☞ i.e., Zookeeper can solve consensus (agreement) for arbitrary number of clients
- **If an application needs linearizability**
  - There is a trick: sync operation
  - Use sync followed by a read operation within an application-level read
    - ☞ This yields a "slow read"

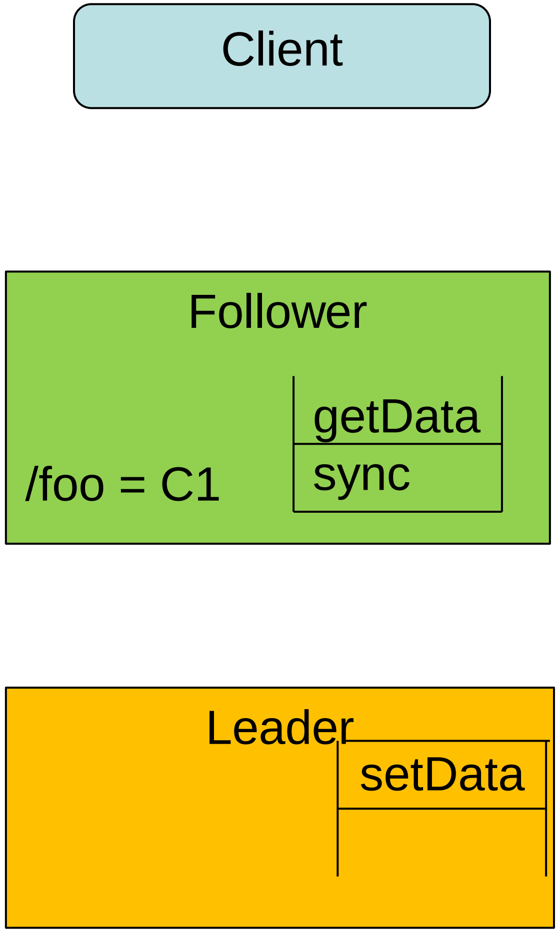
sync

- **Sync**
  - Asynchronous operation
  - Before read operations
  - Flushes the channel between follower and leader
  - Enforces linearizability



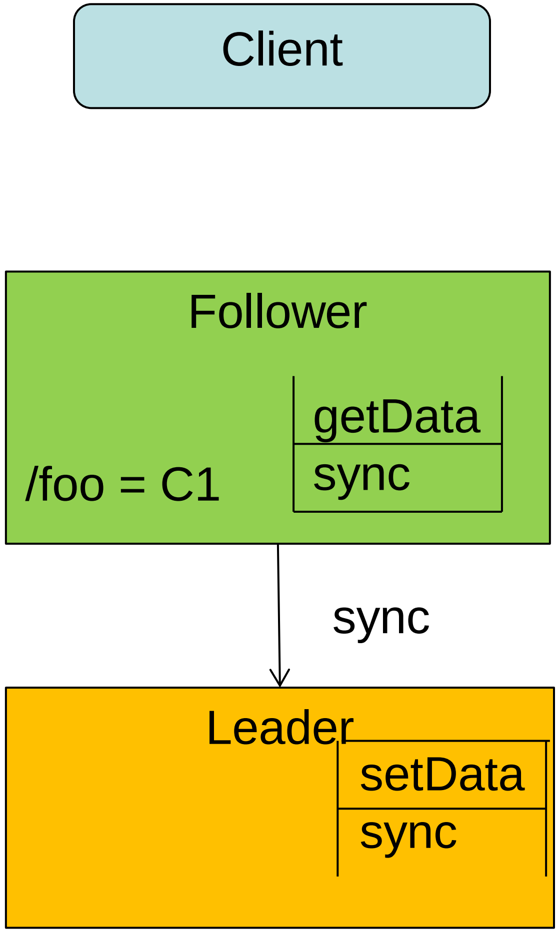
sync

- **Sync**
  - Asynchronous operation
  - Before read operations
  - Flushes the channel between follower and leader
  - Enforces linearizability



sync

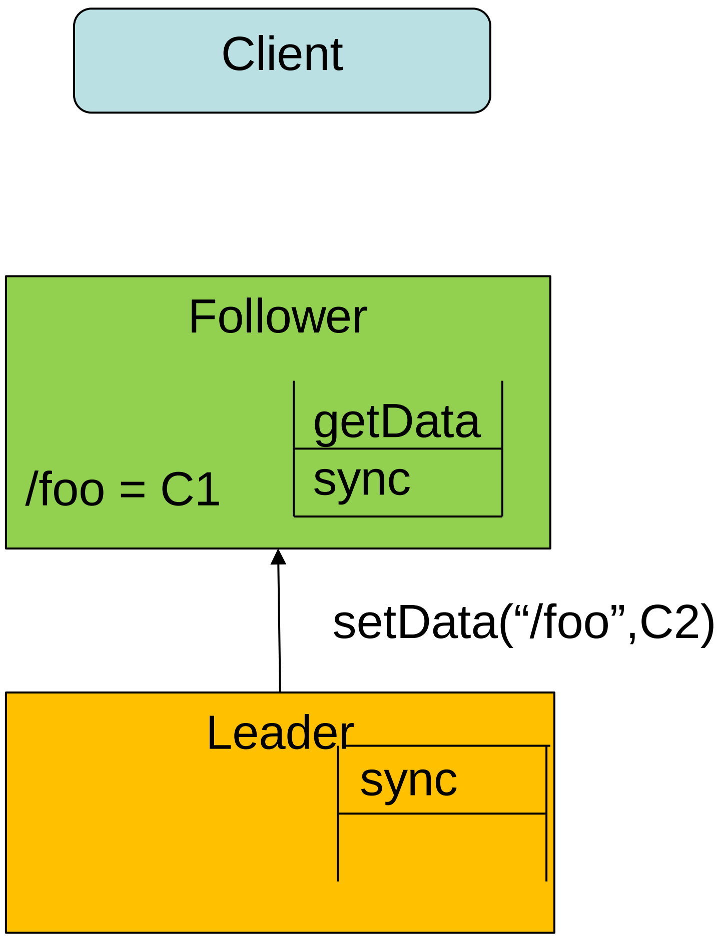
- **Sync**
  - Asynchronous operation
  - Before read operations
  - Flushes the channel between follower and leader
  - Enforces linearizability



EURECOM

sync

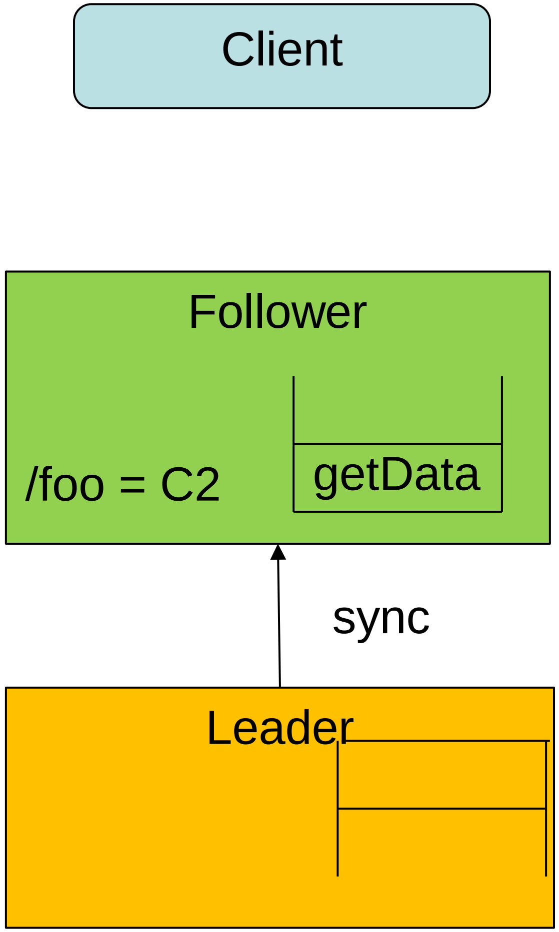
- **Sync**
  - Asynchronous operation
  - Before read operations
  - Flushes the channel between follower and leader
  - Enforces linearizability



EURECOM

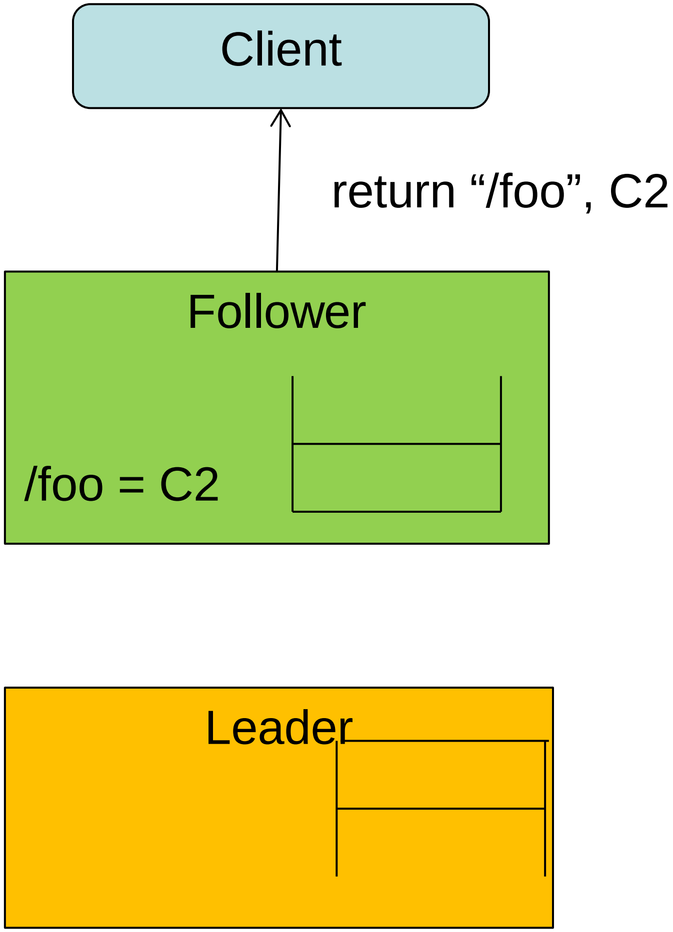
sync

- **Sync**
  - Asynchronous operation
  - Before read operations
  - Flushes the channel between follower and leader
  - Enforces linearizability



sync

- **Sync**
  - Asynchronous operation
  - Before read operations
  - Flushes the channel between follower and leader
  - Enforces linearizability



Read performance

- **Slow reads (sync + read)**
  - Linerizability
  - Slow, leader bottleneck
- **“Normal” reads**
  - Might be non-linearizable
  - 1 round-trip client/server
- **One more option: Caching reads**
  - Cache reads at a client, save on a round-trip
  - Set a watch for a notification needed for cache invalidation

Write operations (summary)

- **Always go through the slow “path”**
- **A write request is forwarded by a follower server to the leader**
- **Leader uses atomic (total-order) broadcast to disseminate messages**
  - Using ZAB protocol
- **ZAB**
  - A variant of Paxos tweaked to support FIFO/causal consistency of asynchronous calls
  - Quorum-based (2f+1 servers, tolerates f failures)

Session consistency

- **What if a follower that a client is talking to fails?**
  - Or connection is lost for any other reason
  - Some operations might have not been executed

- **Upon disconnection**
  - Client library tries to contact another server before session expires

Zookeeper: Agenda

- Motivation and Background
- Coordination kernel
- Semantics
- Programming Zookeeper ←
- Internal Architecture

Implementing consensus

- **Consensus in brief**
  - All correct processes propose a value
  - All correct processes decide a value (exactly once)
  - A decision must be proposed
  - All decisions must be the same

- ***Propose(v)***

  create(“/c/proposal-”, “v”, SEQUENTIAL)

- ***Decide()***

  C = getChildren(“/c”)

  Select znode *z in C* with smallest sequence number suffix

  v’ = getData(“/c/z”)

  Decide v’

Simple configuration management

- **Clients initialized with the name of znode**
  - E.g., “/config”

config = getData(“/config”, TRUE)

while (true)

    wait for watch notification on “/config”

    config = getData(“/config”, TRUE)

NB: A client may miss some configuration, but it will always “refresh” when it realizes the configuration is stale

Group membership

- **Idea: leverage ephemeral znodes**
- **Fix a znode “/group”**
- **Assume every process (client) is initialized with its own unique name and ID**
    - Try to adapt to the case where there are no unique names

joinGroup()

create(“/group/” + name, [address,port], EPHEMERAL)

getMembers()

getChildren(“/group”, false)

Set to true to get notified about membership changes

Locks

- **Can also use Zookeeper to implement blocking primitives**
  - Not to be confused with the fact that Zookeeper is wait-free

- **Let’s try Locks**

A simple lock

Lock(filename)

```
1:      create(filename, “”, EPHEMERAL)
        if create is successful
                return                  //have lock
        else
                getData(filename,TRUE)
                wait for filename watch
                goto 1:
```

Release(filename)

```
        delete(filename)
```

Problems?

- **Herd effect**
  - If many clients wait for the lock they will all try to get it as soon as it is released

Simple Lock without Herd Effect

Lock(filename)

```
1:    myLock=create(filename + "/lock-", "", EPHEMERAL & SEQUENTIAL)
2:    C = getChildren(filename, false)
3:    if myLock is the lowest znode in C then return
4:    else
5:        precLock = znode in C ordered just before n
6:        if exists(precLock, true)
7:            wait for precLock watch
8:            goto 2:
```

Release(filename)

```
      delete(myLock)
```

Exercise (homework)

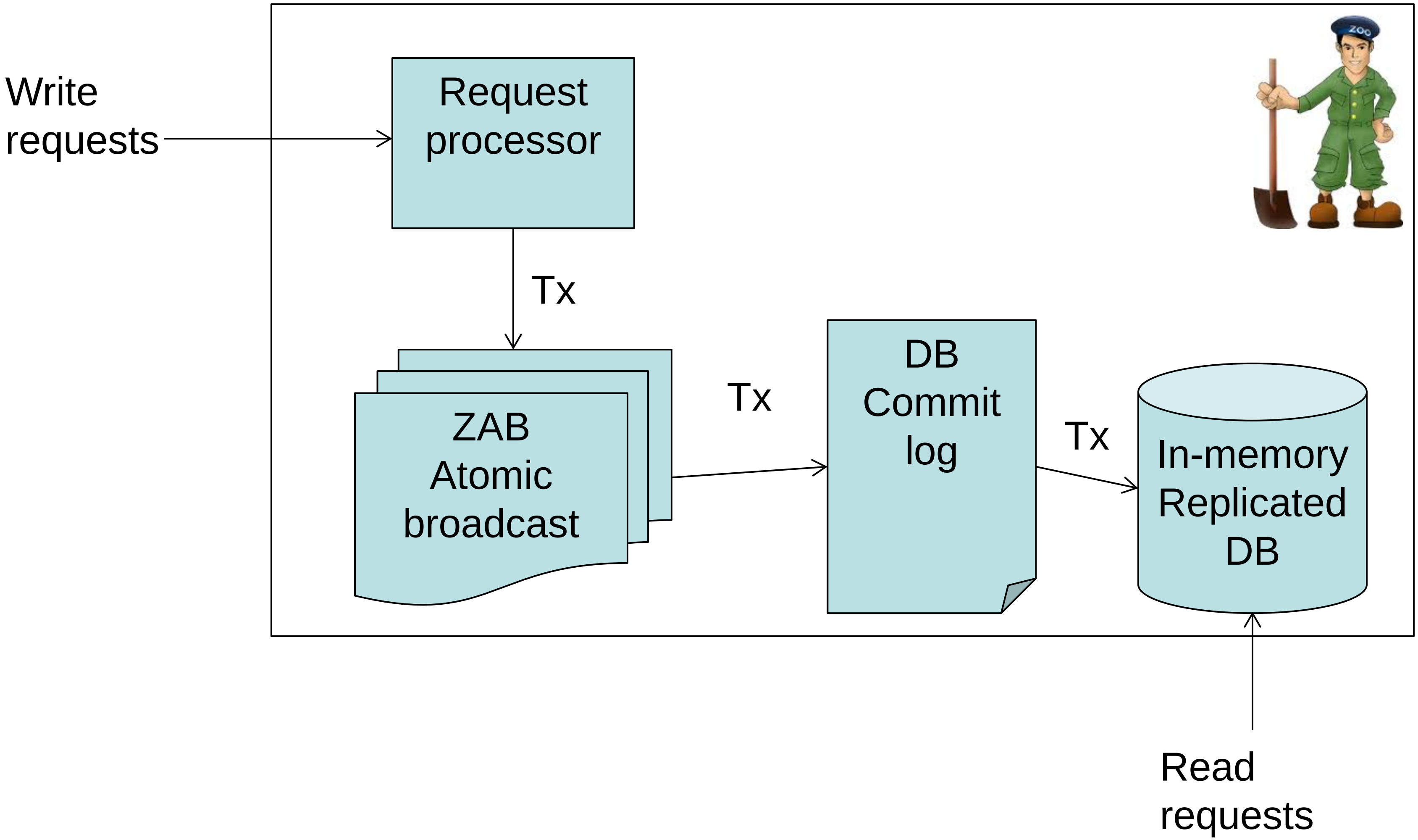
- The previous lock solves herd effect but makes reads block other reads

- Adapt the Zookeeper implementation such that reads always get the lock unless there is a concurrent write

Zookeeper: Agenda

- **Motivation and Background**
- **Coordination kernel**
- **Semantics**
- **Programming Zookeeper**
- **Internal Architecture ←**

Zookeeper components (high-level)



Zookeeper DB

- **Fully replicated**
  - To be contrasted with partitioning/placement in Cassandra/Dynamo
- **Each server has a copy of in-memory DB**
  - Store the entire znode tree
  - Default max 1 MB per znode (configurable)
- **Crash-recovery model**
  - Commit log
  - + periodic snapshots of the database

ZAB: a very brief overview

- **Used to totally order write requests**
  - Relies on a quorum of servers (f+1 out of 2f+1)
- **ZAB internally elects leader replica**
  - Not to be confused with Leader Election using Zookeeper API
- **Zookeeper adopts this notion of a leader**
  - Other servers are followers
- **All write requests are sent by followers to the leader**
  - Leader sequences the requests and invokes ZAB atomic broadcast

Request processor

- **Upon receiving a write request**
  - the leader calculates in what state system will be after the write is applied
  - Transforms the operation in the transactional update
- **Such transactional updates are then processed by ZAB, DB**
  - Guarantees idempotency of updates to the DB originating from the same operation
- **Idempotency: Important since ZAB may redeliver a message**
  - Upon recovery not during normal operation
  - Also allows more efficient DB snapshots

Further reading

(recommended)

- **Patrick Hunt, Mahadev Kumar, Flavio P. Junqueira and Benjamin Reed: Zookeeper: Wait-free coordination for Internet-scale systems. In proc. USENIX ATC (2010)**

  http://static.usenix.org/event/usenix10/tech/full_papers/Hunt.pdf

- **Zookeeper 3.4 Documentation**

  http://zookeeper.apache.org/doc/trunk/index.html

(optional)

- **Flavio Paiva Junqueira, Benjamin C. Reed, Marco Serafini**: **Zab: High-performance broadcast for primary-backup systems.** DSN 2011: 245-256
- **Michael Burrows: The Chubby Lock Service for Loosely-Coupled Distributed Systems. OSDI 2006: 335-350**
- **Atul Adya, John Dunagan, Alec Wolman: Centrifuge: Integrated Lease Management and Partitioning for Cloud Services. NSDI 2010: 1-16**