ZooKeeper & Curator

CS 475, Spring 2018
Concurrent & Distributed Systems

Review: Agreement

- In distributed systems, we have multiple nodes that need to all agree that some object has some state
- Examples:
  - Who owns a lock
  - Whether or not to commit a transaction
  - The value of a file

Review: Agreement Generally

- Most distributed systems problems can be reduced to this one:
  - Despite being separate nodes (with potentially different views of their data and the world)…
  - All nodes that store the same object O must apply all updates to that object in the same order (consistency)
  - All nodes involved in a transaction must either commit or abort their part of the transaction (atomicity)
- Easy?
  - … but nodes can restart, die or be arbitrarily slow
  - … and networks can be slow or unreliable too

Review: Properties of Agreement

- **Safety** (correctness)
  - All nodes agree on the same value (which was proposed by some node)
- **Liveness** (fault tolerance, availability)
  - If less than N nodes crash, the rest should still be OK

Review: Does 3PC guarantee agreement?

- Reminder, that means:
  - Liveness (availability)
    - **Yes!** Always terminates based on timeouts
  - Safety (correctness)
    - Hmm…

Review: Partitions

Coordinator

Soliciting Votes

Synchronized

Prepared to commit

Network Partition!!!

Yes

Participant A

Committed

Uncertain

Timeout behavior: Commit!

Yes

Participant B

Aborted

Uncertain

Yes

Participant C

Aborted

Uncertain

Yes

Participant D

Aborted

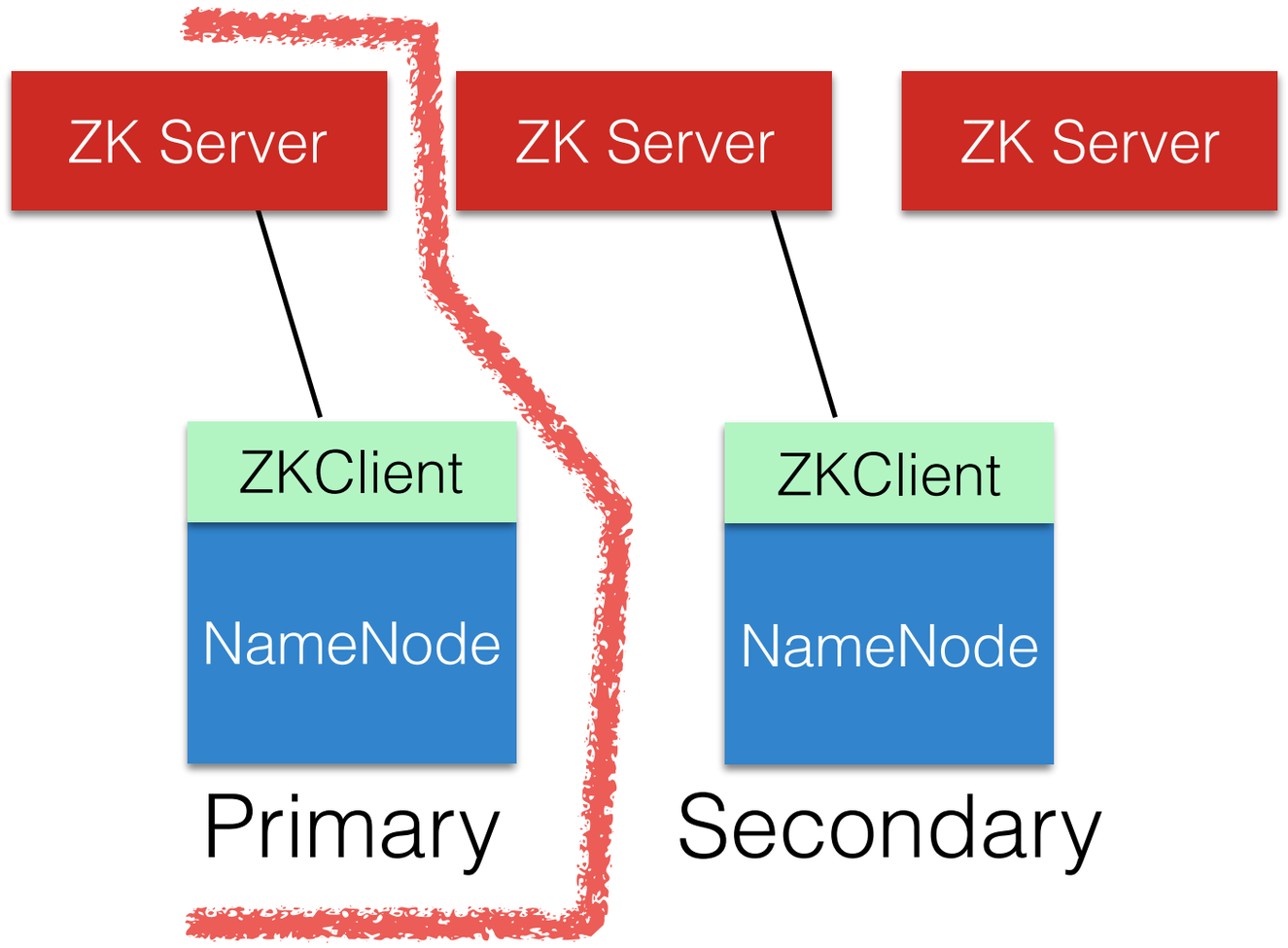
Uncertain

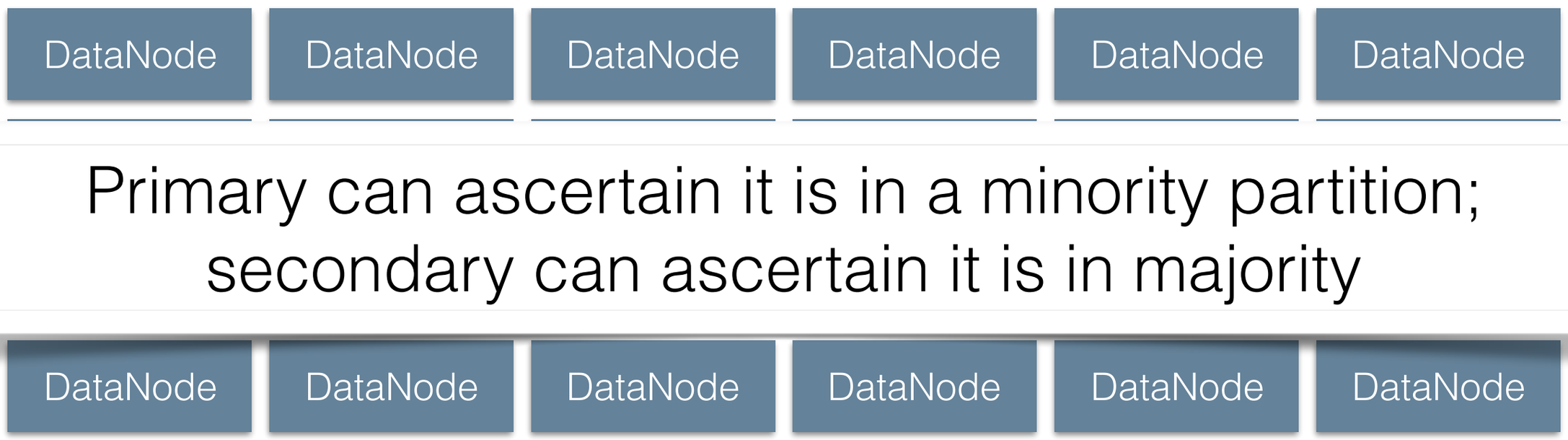
Timeout behavior: abort

Review: Partition Tolerance

- Key idea: if you always have an odd number of nodes…
- There will always be a **minority** partition and a **majority** partition
- Give up processing in the minority until partition heals and network resumes
- Majority can continue processing

Review: Partition Tolerance

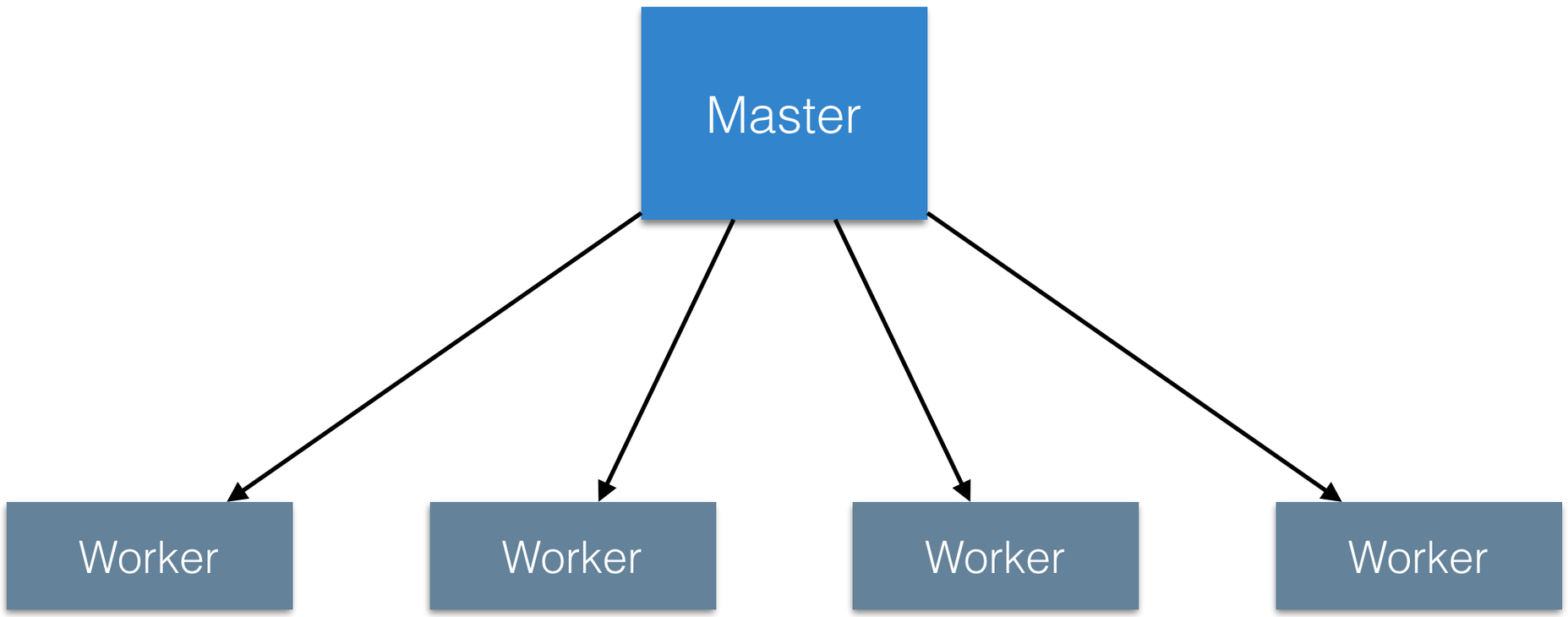




Announcements

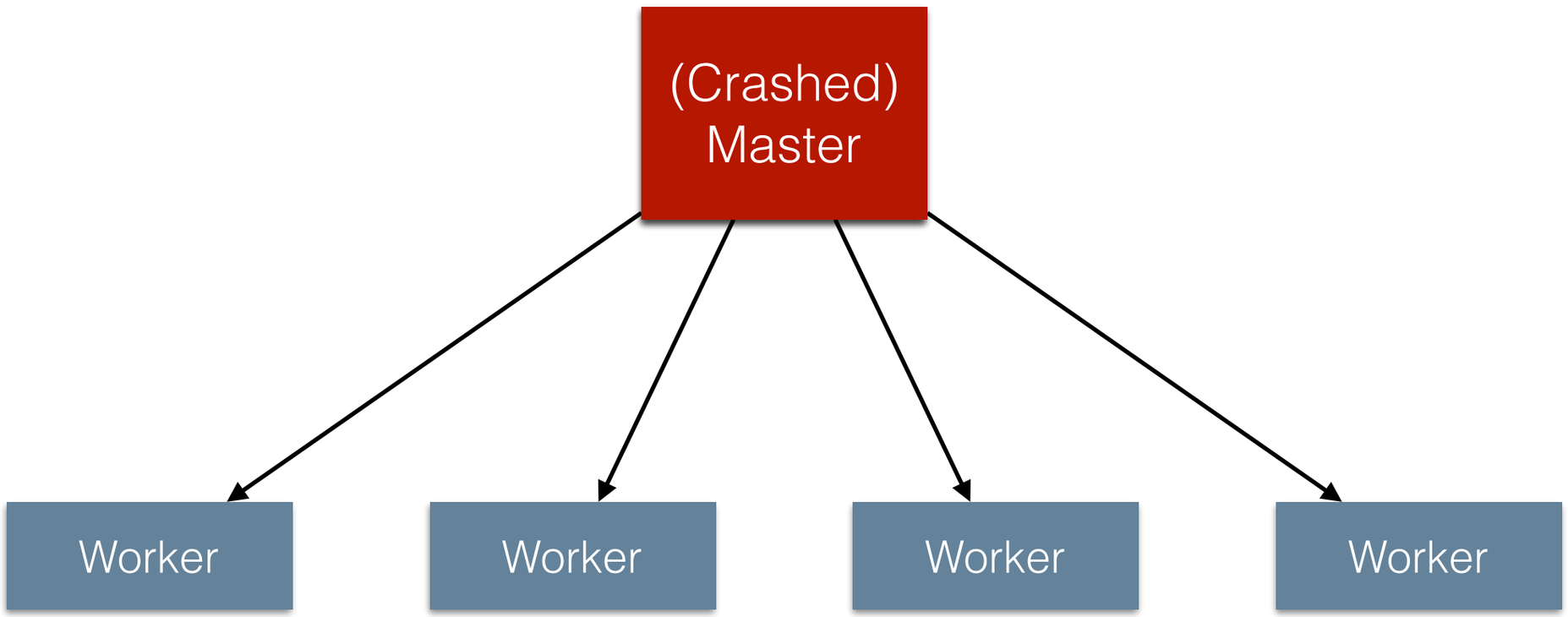
- Form a team and get started on the project!
  - http://jonbell.net/gmu-cs-475-spring-2018/final-project/
  - AutoLab available soon
- Today:
  - ZooKeeper guarantees and usage
- Additional readings:
  - Distributed Systems for Fun & Profit Ch 4
  - http://book.mixu.net/distsys/replication.html

Master/worker distributed model



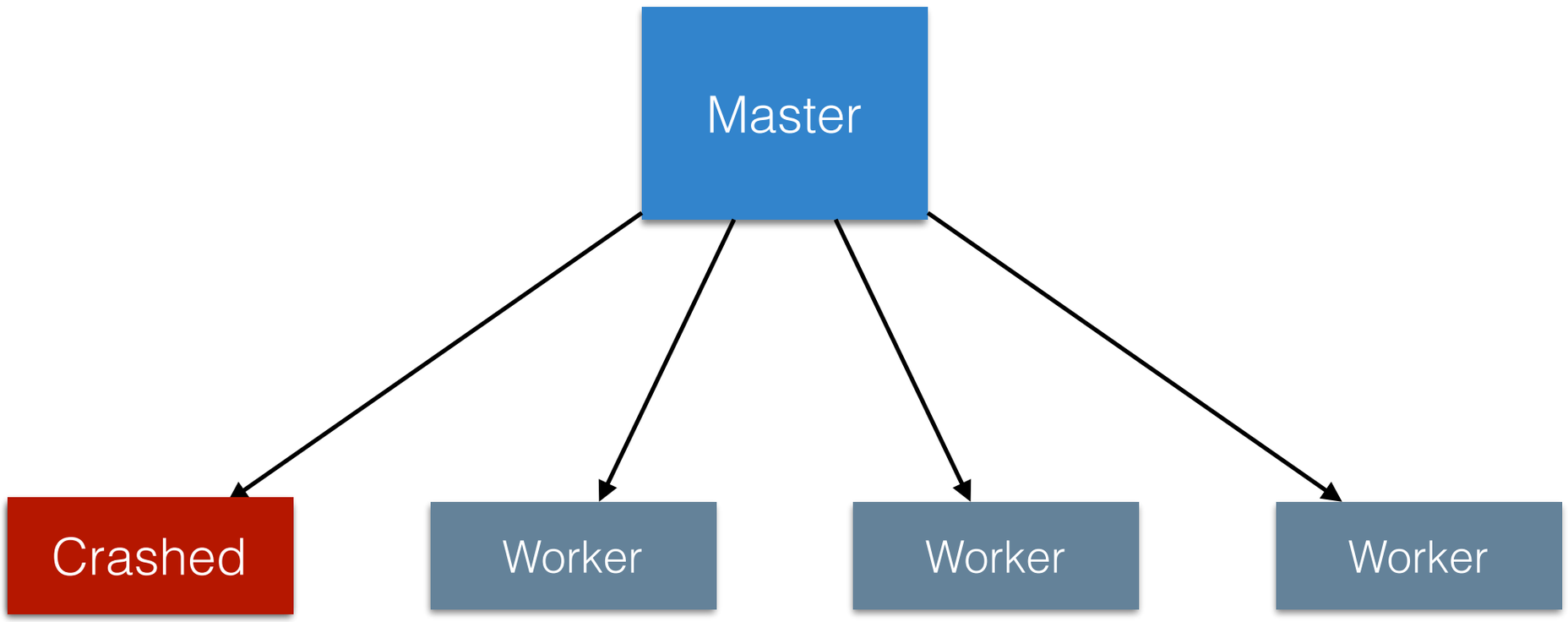
Master/worker distributed model

- Master is single point of failure!
- If master fails, no work is assigned
- Need to select a new master



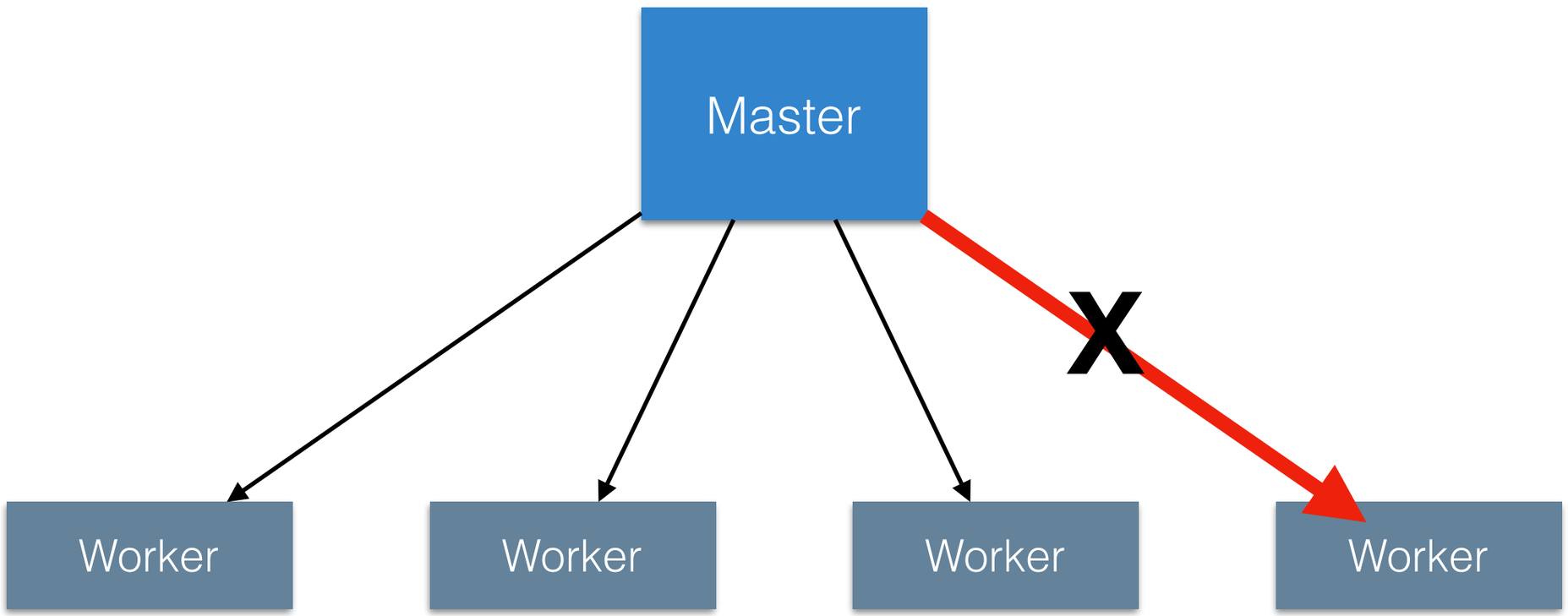
Master/worker distributed model

- If a worker fails?
- Not as bad, but need to detect its failure
- Some tasks might need to get re-assigned elsewhere



Master/worker distributed model

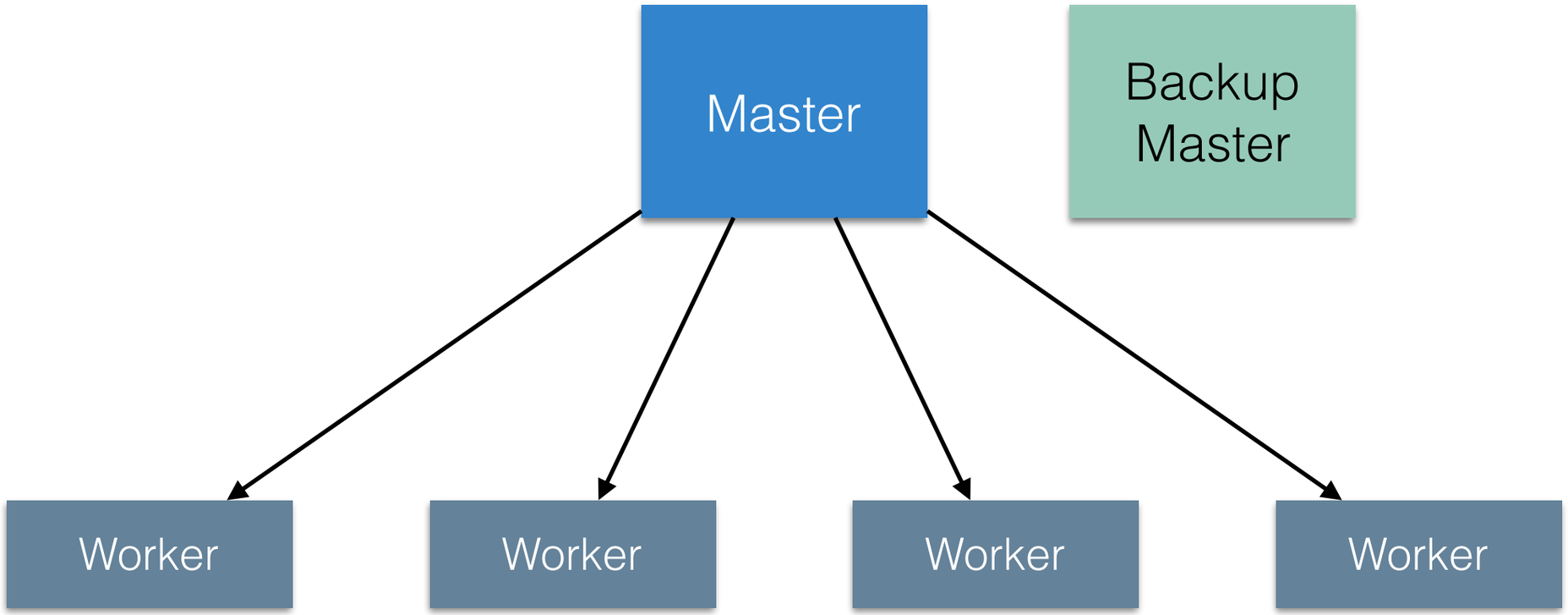
- If a worker doesn't receive a task (network link failure)?
- Again, not as bad, but need to detect
- Will need to try to re-establish link (difference between "there is no work left to do" and "I just didn't hear I needed to do something)



Coordination

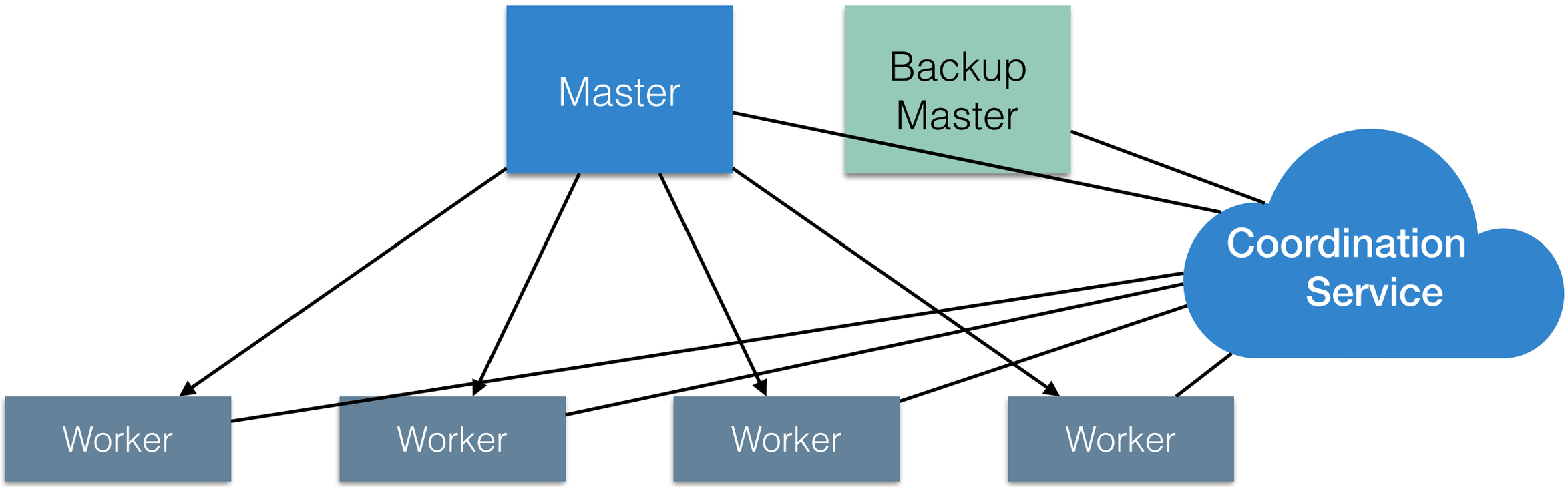
- Semaphores
- Queues
- Transactions
- Locks
- Barriers

Strawman Fault Tolerant Master/Worker System



How do we know to switch to the backup?
How do we know when workers have crashed, or network has failed?

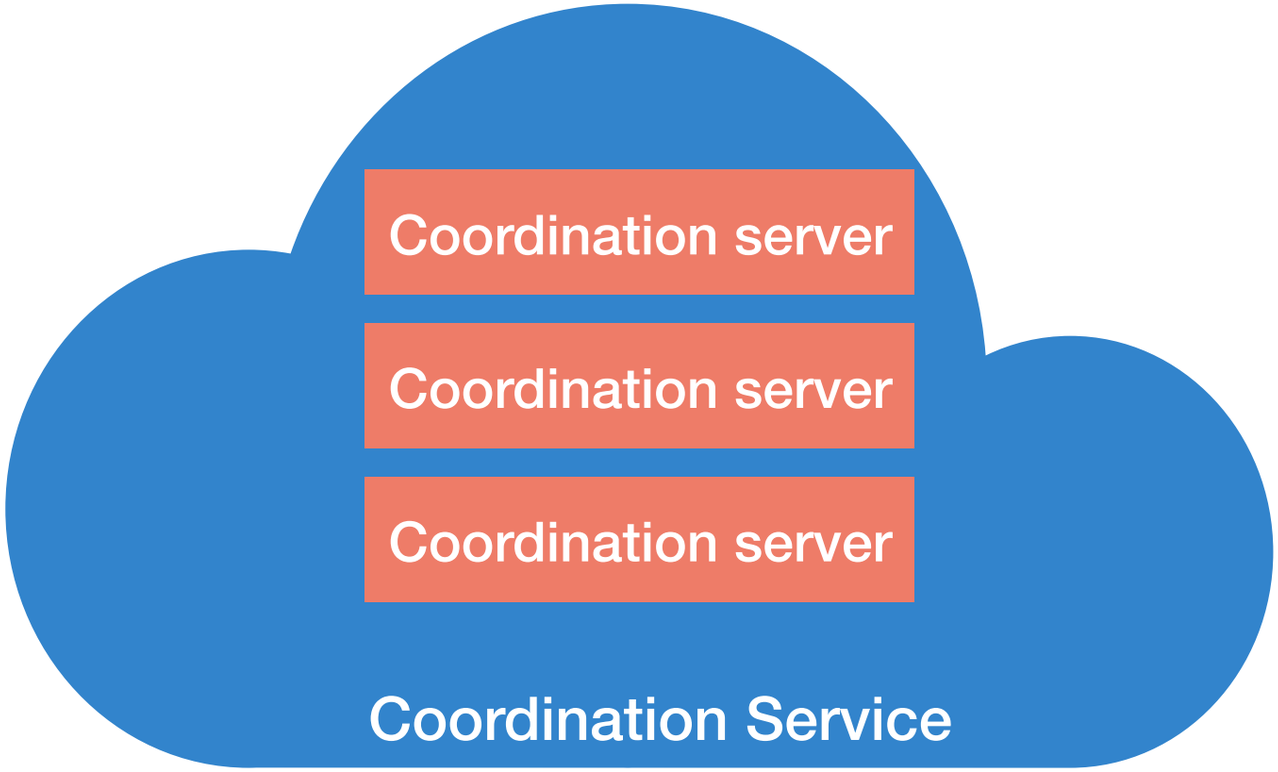
Fault Tolerant Master/Worker System



Coordination service handles all of those tricky parts. But can't the coordination service fail?

Fault-Tolerant Distributed Coordination

- Leave it to the coordination service to be fault-tolerant
- Can solve our master/worker coordination problem in 2 steps:
  - 1 - Write a fault-tolerant distributed coordination service
  - 2 - Use it
- Thankfully, (1) has been done for us!



ZooKeeper

- Distributed coordination service from Yahoo! originally, now maintained as Apache project, used widely (key component of Hadoop etc)
- Highly available, fault tolerant, performant
- Designed so that YOU don't have to implement Paxos for:
  - Maintaining group membership, distributed data structures, distributed locks, distributed protocol state, etc

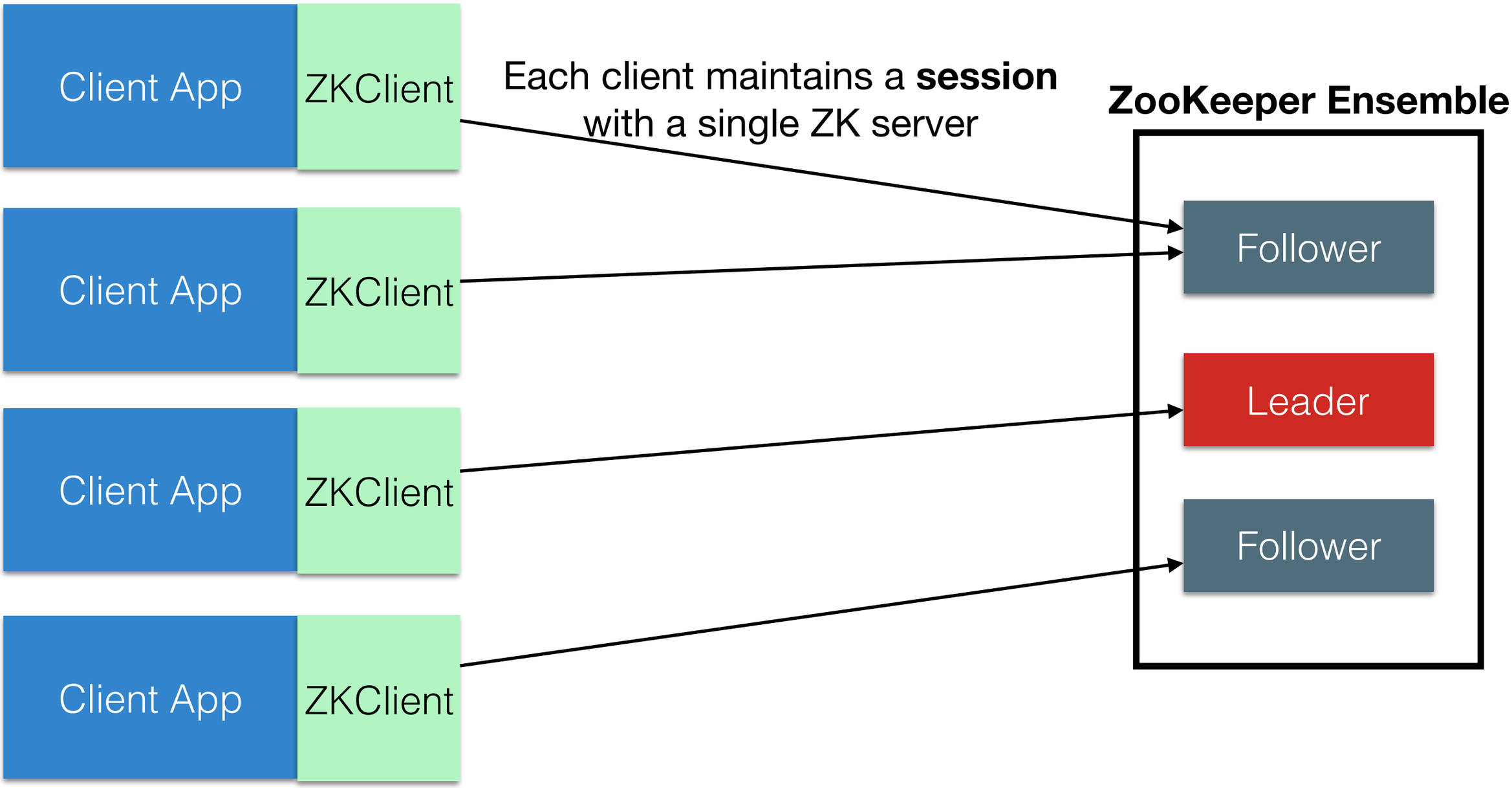
ZooKeeper - Guarantees

- **Liveness**: if a majority of ZooKeeper servers are active and communicating the service will be available
- **Atomic updates**: A write is either entirely successful, or entirely failed
- **Durability**: if the ZooKeeper service responds successfully to a change request, that change persists across any number of failures as long as a quorum of servers is eventually able to recover

Example use-cases

- Configuration management (which servers are doing what role?)
- Synchronization primitives
  - Anti-use cases:
    - Storing large amounts of data
    - Sharing messages and data that don't require liveness/durability guarantees

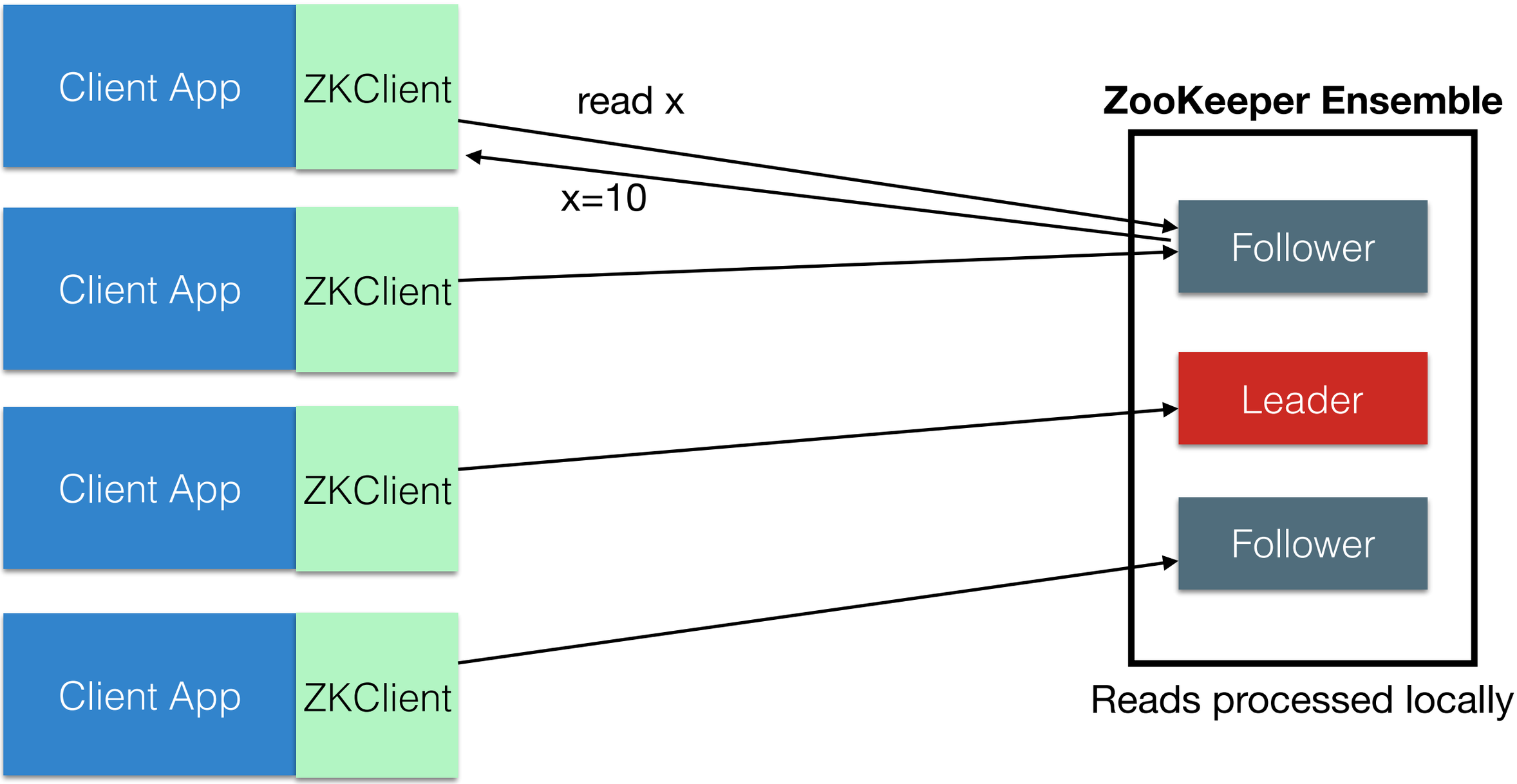
ZooKeeper - Overview



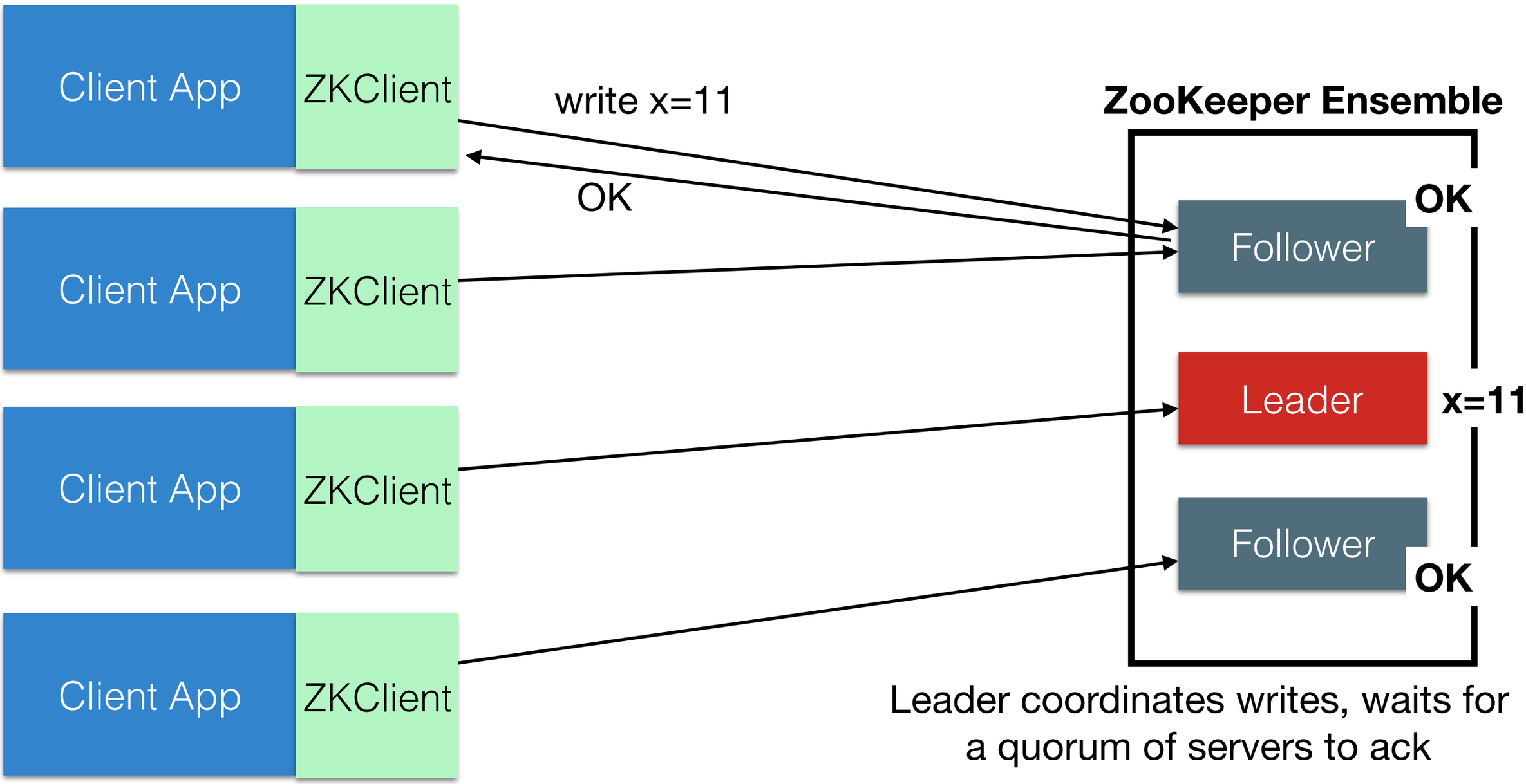
ZooKeeper - Sessions

- Each client maintains a session with a single ZK server
- Sessions are valid for some time window
- If client discovers its disconnected from ZK server, attempts to reconnect to a different server before session expires

ZooKeeper - Overview

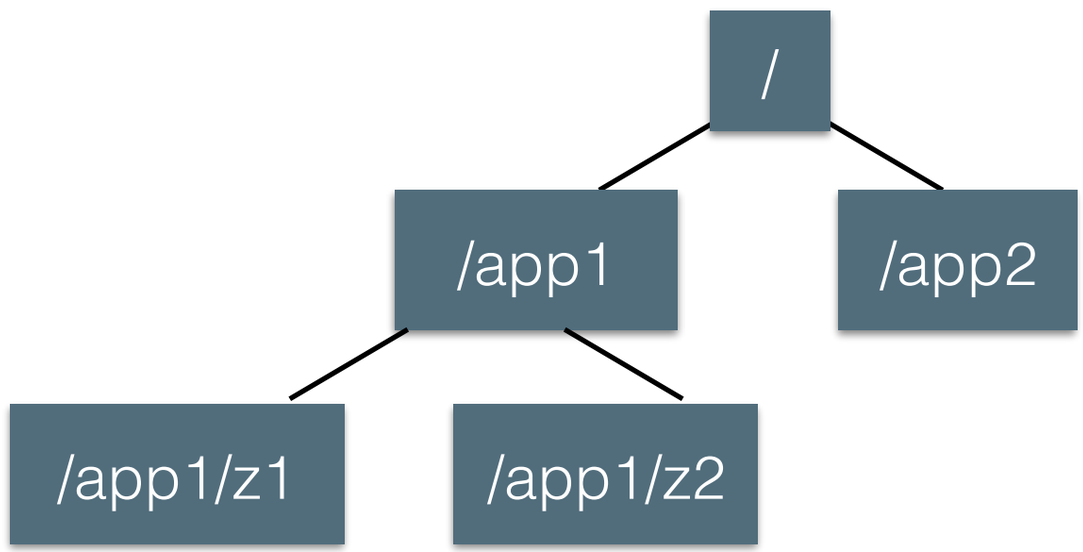


ZooKeeper - Overview



ZooKeeper - Data Model

- Provides a hierarchical namespace
- Each node is called a znode
- ZooKeeper provides an API to manipulate these nodes



ZooKeeper - ZNodes

- In-memory data
- NOT for storing general data - just metadata (they are replicated and generally stored in memory)
- Map to some client abstraction, for instance - locks
- Znodes maintain counters and timestamps as metadata

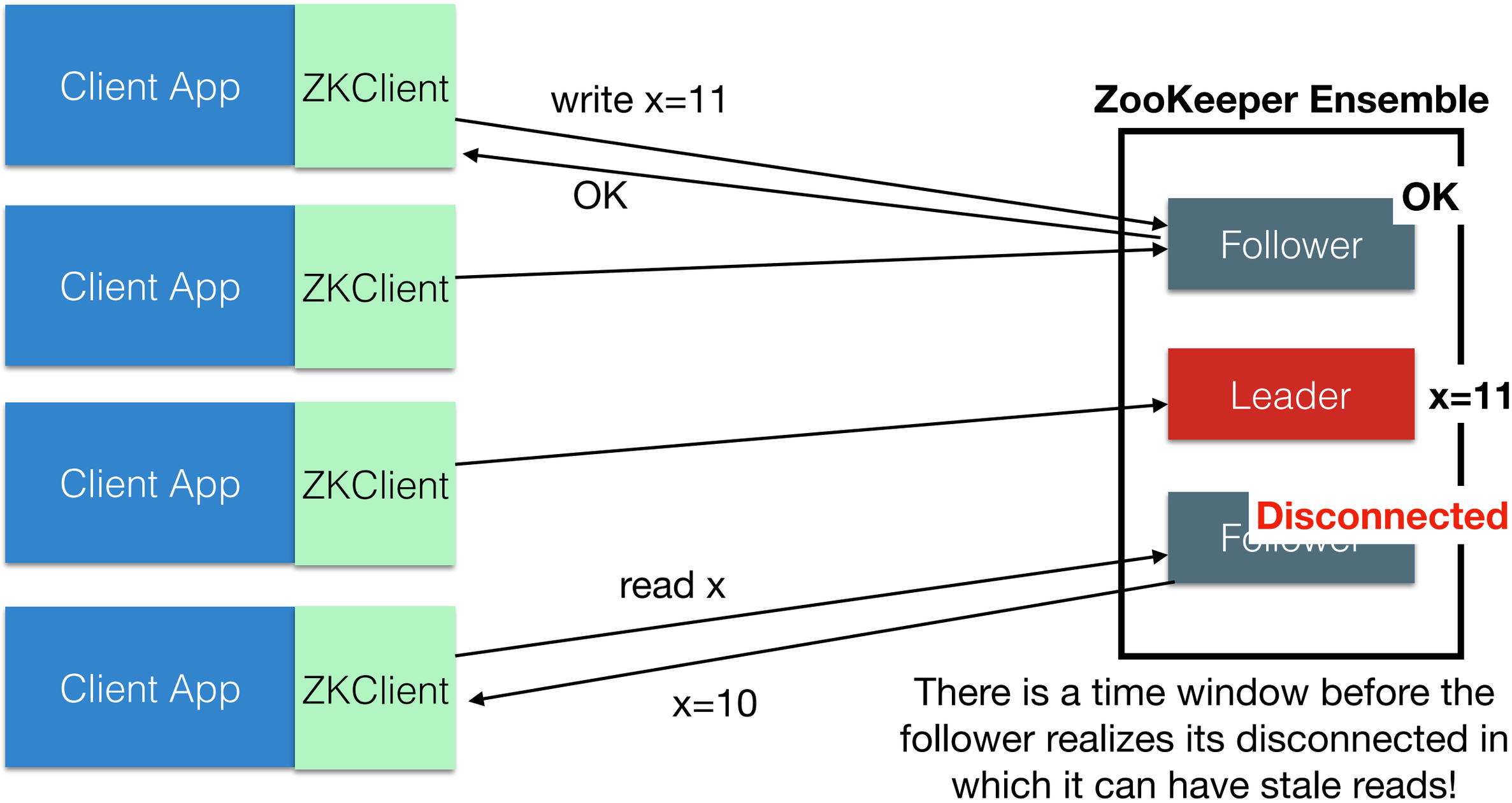
ZooKeeper - Znode Types

- Regular znodes
  - Can have children znodes
  - Created and deleted by clients explicitly through API
- Ephemeral znodes
  - Cannot have children
  - Created by clients explicitly
  - Deleted by clients OR removed automatically when client session that created them disconnects

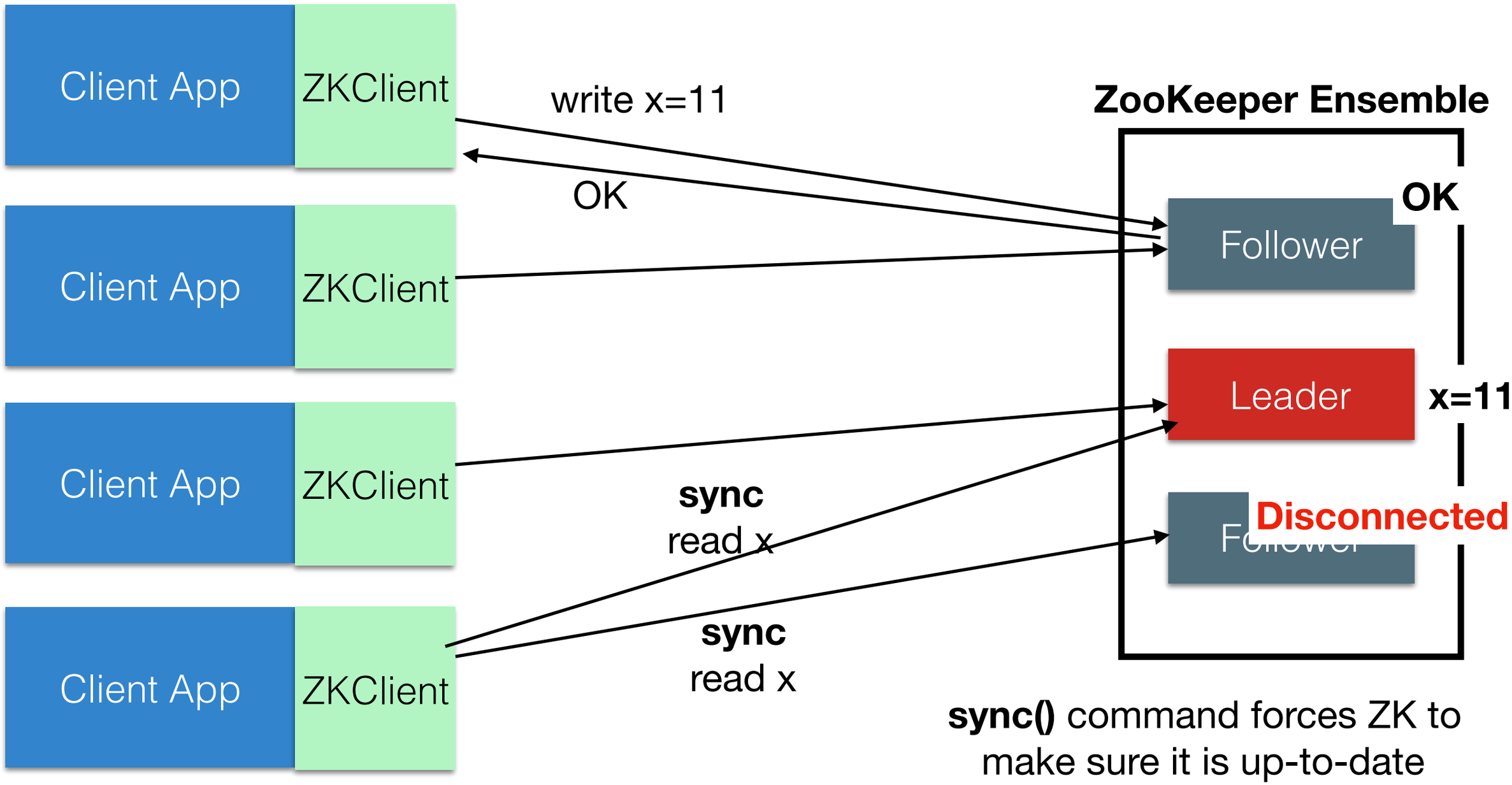
ZooKeeper - API

- Clients track changes to znodes by registering a **watch**
- Create(path, data, flags)
  Delete(path, version)
  Exists(path, watch)
  getData(path, watch)
  setData(path, data, version)
  getChildren(path, watch)
  Sync(path)

ZooKeeper - Consistency



ZooKeeper - Consistency



ZooKeeper - Consistency

- Sequential consistency of writes
  - All updates are applied in the order they are sent, linearized into a total order by the leader
- Atomicity of writes
  - Updates either succeed or fail
- Reliability
  - Once a write has been applied, it will persist until its overwritten, as long as a majority of servers don't crash
- Timeliness
  - Clients are guaranteed to be up-to-date for reads **within a time bound** - after which you either see newest data or are disconnected

ZooKeeper - Lock Example

- To acquire a lock called **foo**
- Try to create an ephemeral znode called **/locks/foo**
- If you succeeded:
  - You have the lock
- If you failed:
  - Set a watch on that node. When you are notified that the node is deleted, try to create it again.
- Note - no issue with consistency, since there is no read (just an atomic write)

ZooKeeper - Recipes

- Why figure out how to re-implement this low level stuff (like locks)?
- Recipes: https://zookeeper.apache.org/doc/r3.3.6/recipes.html
  - And in Java: http://curator.apache.org
  - Examples:
    - Locks
    - Group Membership

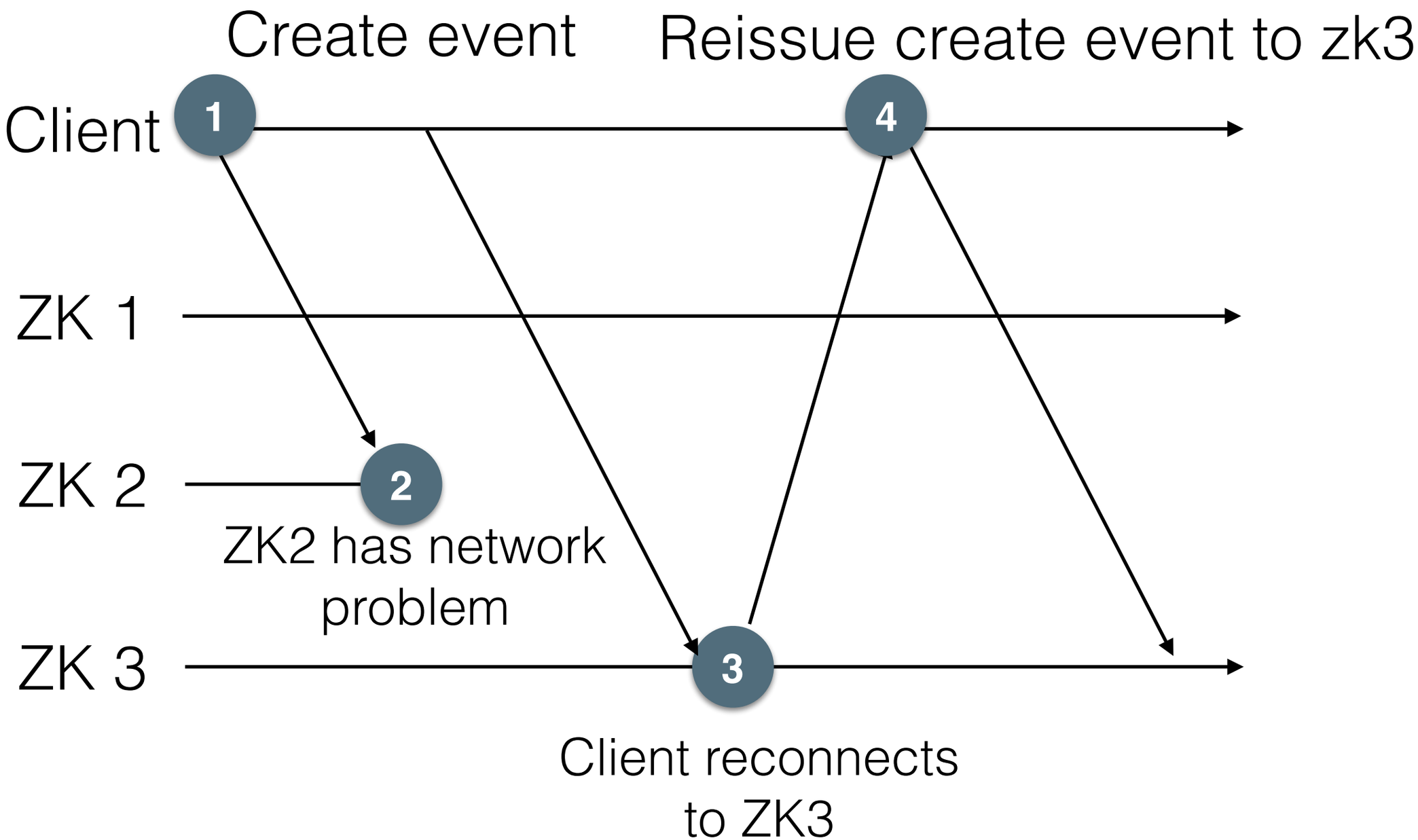
How Many ZooKeepers?

- How many ZooKeepers do you want?
  - An odd number
  - 3-7 is typical
  - Too many and you pay a LOT for coordination

Failure Handling in ZK

- Just using ZooKeeper does not solve failures
- Apps using ZooKeeper need to be aware of the potential failures that can occur, and act appropriately
- ZK client will guarantee consistency **if it is connected to the server cluster**

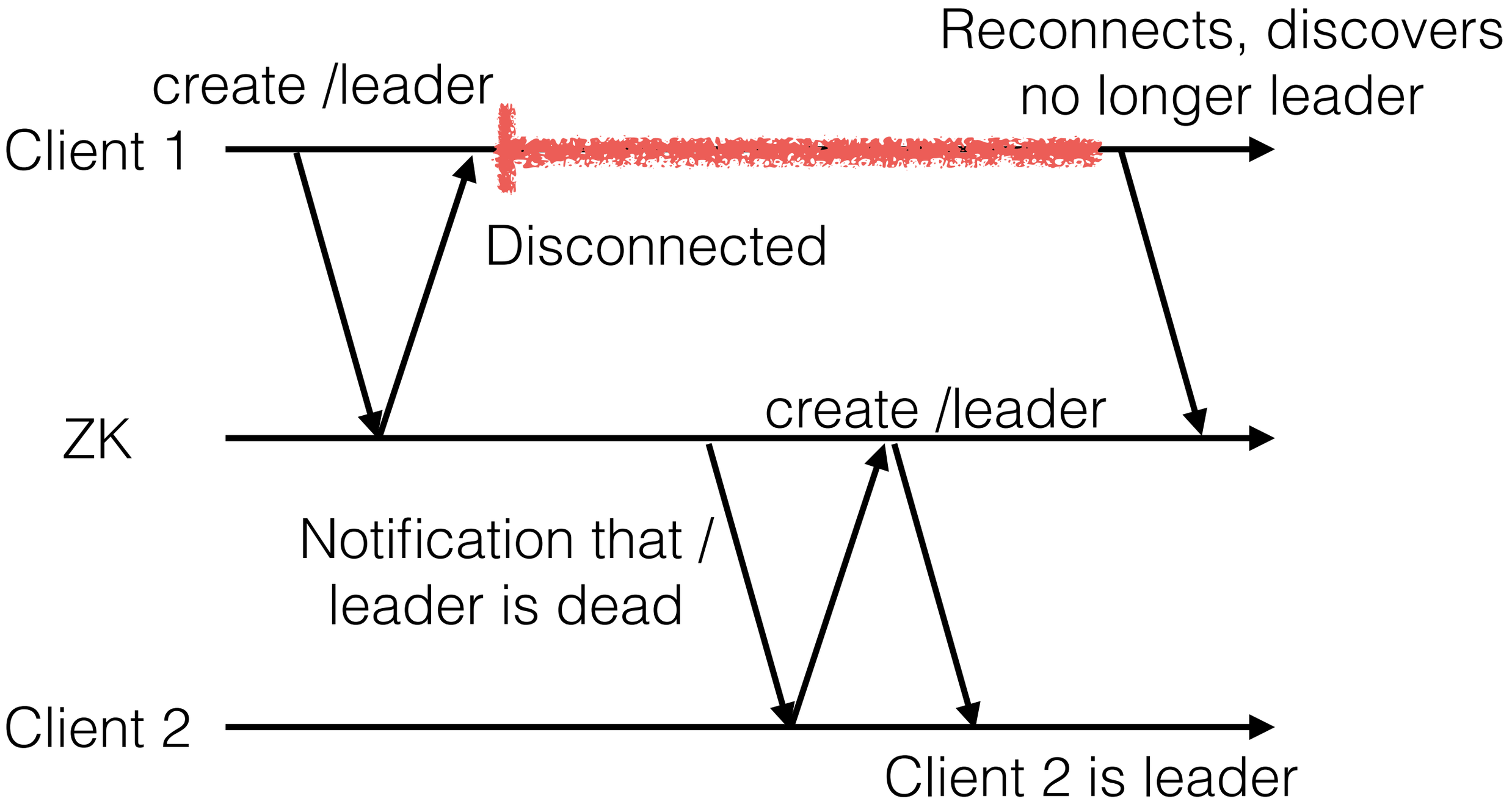
Failure Handling in ZK



Failure Handling in ZK

- If in the middle of an operation, client receives a **ConnectionLossException**
- Also, client receives a **disconnected message**
- Clients can't tell whether or not the operation was completed though - perhaps it was completed before the failure
- Clients that are disconnected can not receive any notifications from ZK

Dangers of ignorance



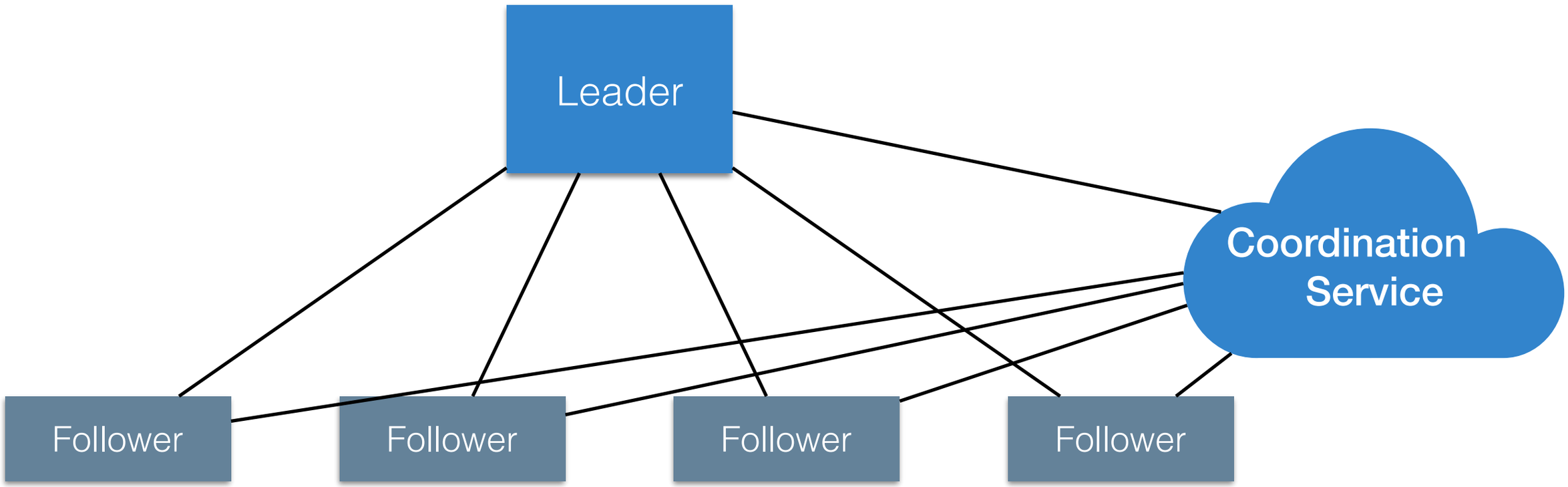
Dangers of ignorance

- Each client needs to be aware of whether or not its connected: when disconnected, can not assume that it is still included in any way in operations
- By default, ZK client will NOT close the client session because it's disconnected!
  - Assumption that eventually things will reconnect
  - Up to you to decide to close it or not

ZK: Handling Reconnection

- What should we do when we reconnect?
- Re-issue outstanding requests?
  - Can't assume that outstanding requests didn't succeed
  - Example: create /leader (succeed but disconnect), re-issue create /leader and fail to create it because you already did it!
- Need to check what has changed with the world since we were last connected

ZooKeeper in Final Project



All writes go to leader

Who is the leader? Once we hit the leader, is it sure that it still is the leader?

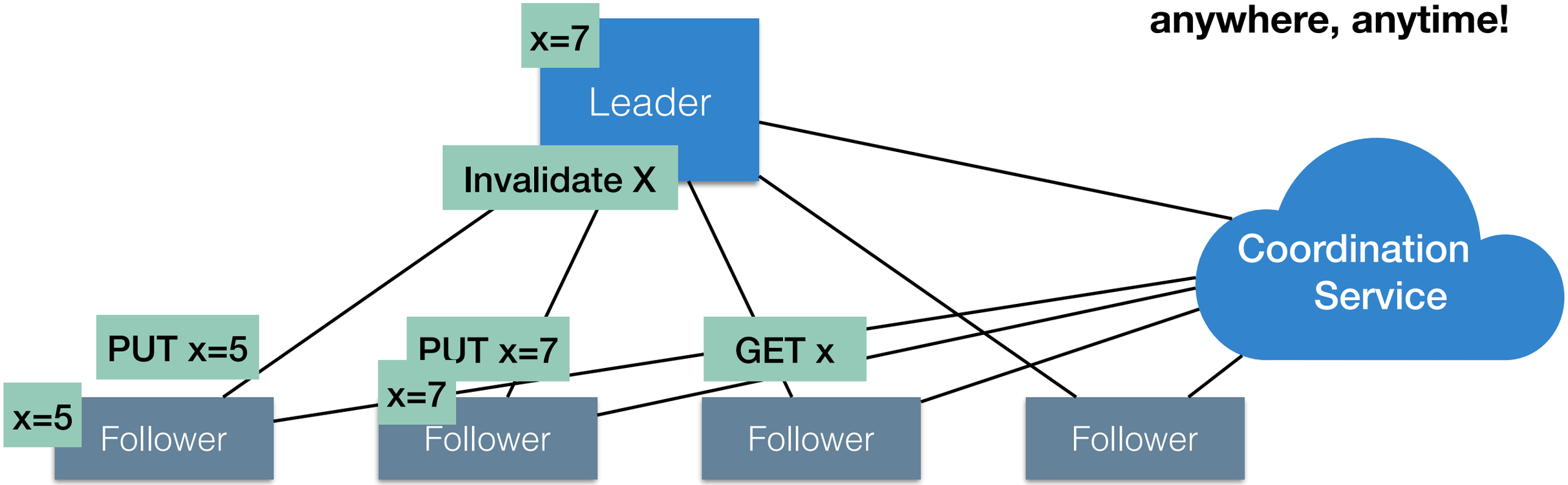
Leader broadcasts read-invalidates to clients

Who is still alive?

Reads processed on each client

If don't have data cached, contact leader - who is leader?

ZooKeeper in Final Project



All writes go to leader

Who is the leader? Once we hit the leader, is it sure that it still is the leader?

Leader broadcasts read-invalidates to clients

Who is still alive?

Reads processed on each client

If don't have data cached, contact leader - who is leader?