ZooKeeper & Curator

CS 475, Spring 2019
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

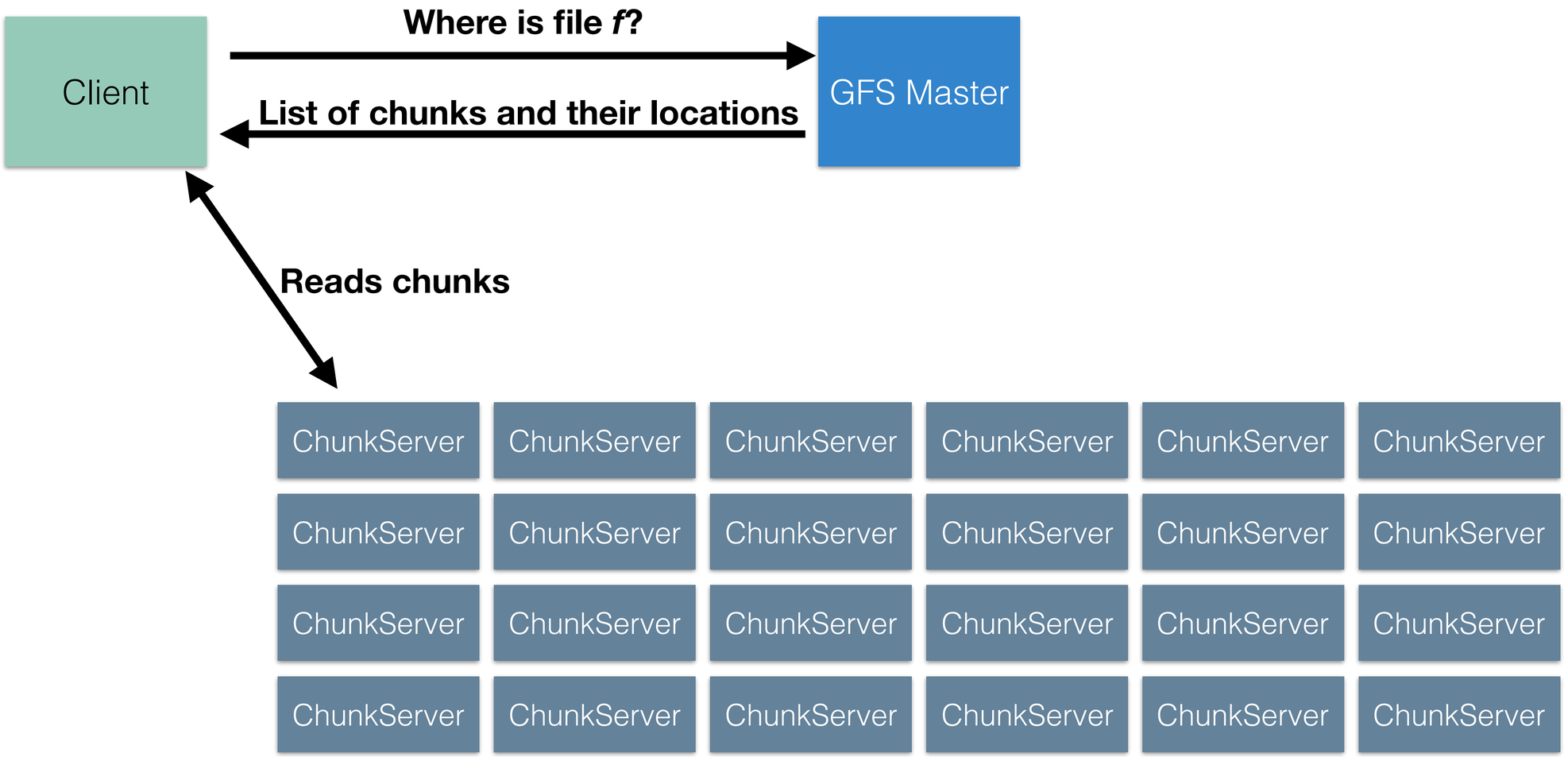
GFS Architecture

GFS Metadata Example

Chunk ID	Filename	Part of file	Master Chunk Server	Other Chunk Servers
1	/foo/bar	1 of 1	A, valid for 1 more minute	B, C
2	/another/file	1 of 2	B, valid for 1 more minute	A, C
3	/another/file	2 of 2	D, valid for 1 more minute	C, E

Note - can get very good parallelism by splitting chunks of the same file across different chunk servers

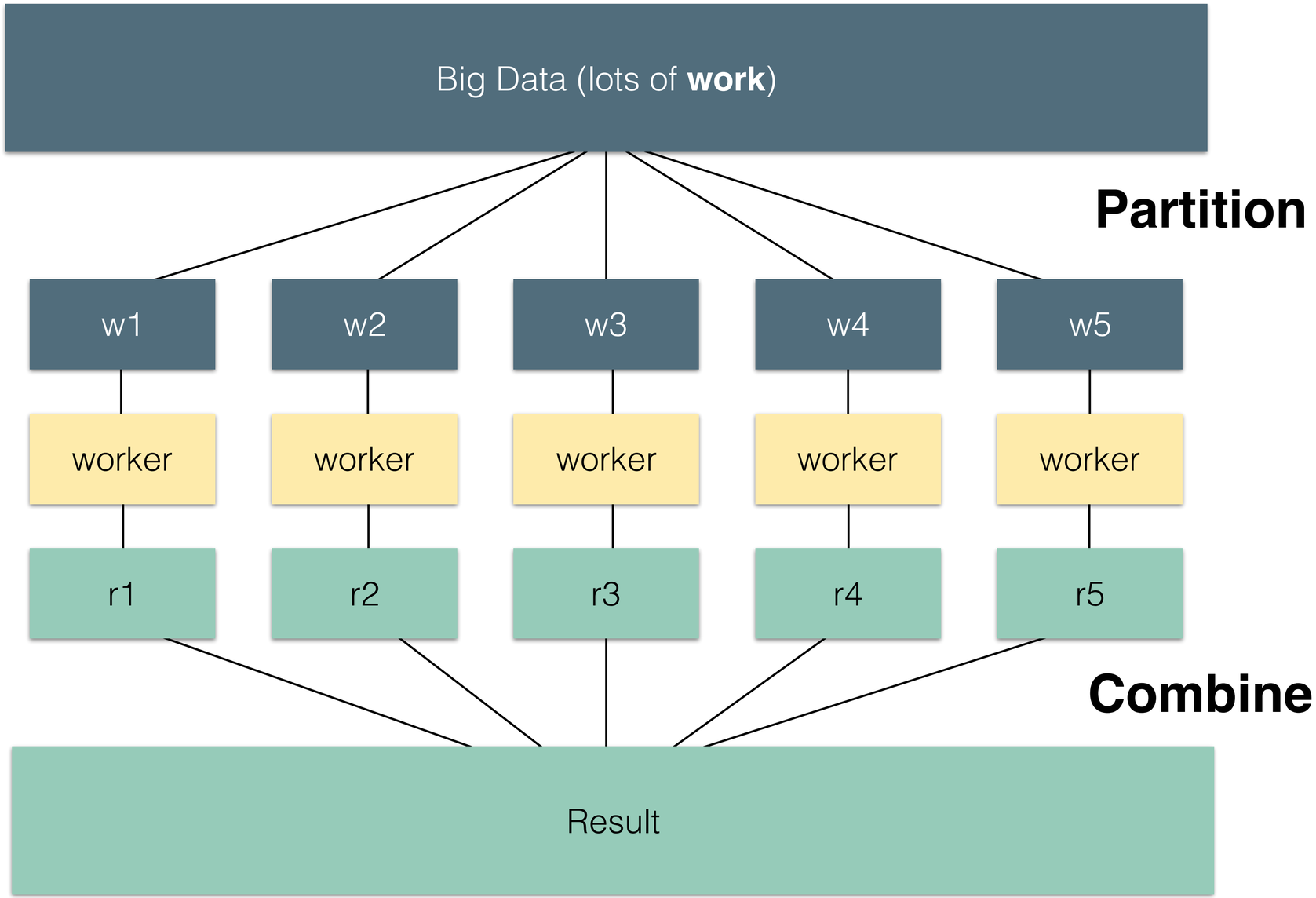
GFS - Reads


Client
Where is file *f*?
GFS Master
List of chunks and their locations
Reads chunks
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer

Today

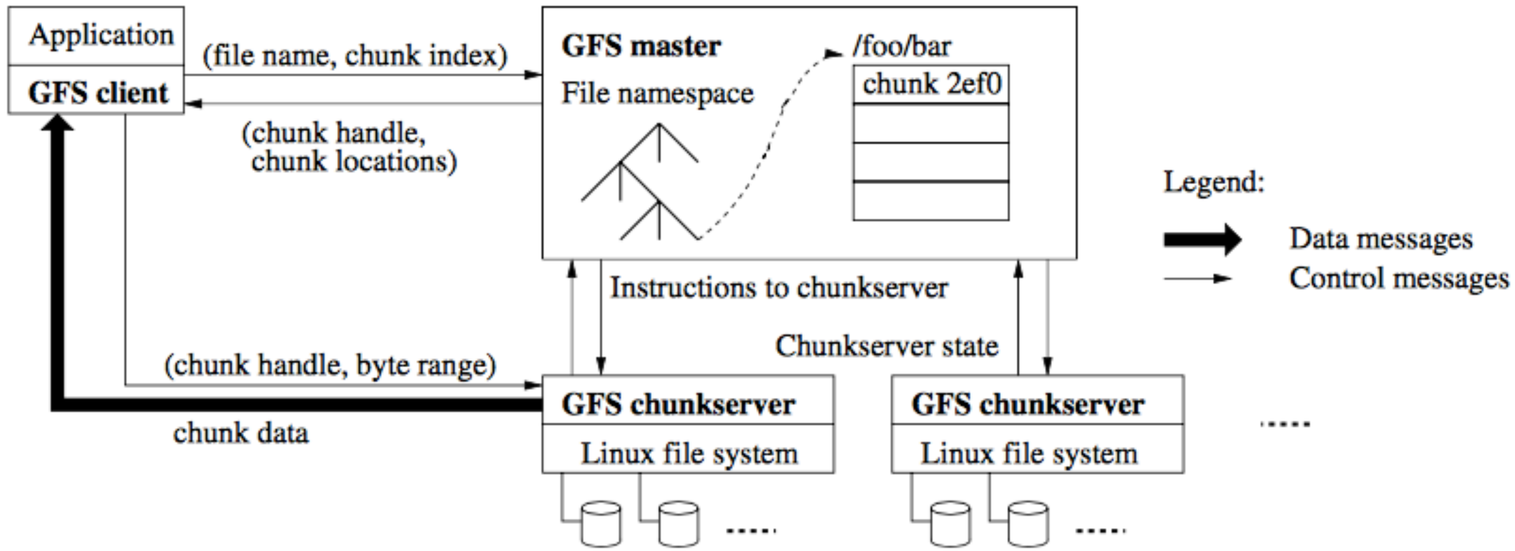
- Reminder - Project is out!
  - Fault-tolerant, sequentially consistent replicated key value store
  - Start thinking of groups (1 to 3 students per group)
- Today:
  - Wrap up MapReduce
  - ZooKeeper - what does it give us and how do we use it?

MapReduce: Divide & Conquer



MapReduce: Implementation

- Each worker node is **also** a GFS chunk server!



MapReduce: Scheduling

- One master, many workers
- Input data split into *M* map tasks (typically 64MB ea)
- *R* reduce tasks
- Tasks assigned to works dynamically; stateless and idempotent -> easy fault tolerance for workers
- Typical numbers:
  - 200,000 map tasks, 4,000 reduce tasks across 2,000 workers

MapReduce: Scheduling

- Master assigns map task to a free worker
  - Prefer "close-by" workers for each task (based on data locality)
  - Follower reads task input, produces intermediate output, stores locally (K/V pairs)
- Master assigns reduce task to a free worker
  - Reads intermediate K/V pairs from map workers
  - Reduce worker sorts and applies some *reduce* operation to get the output

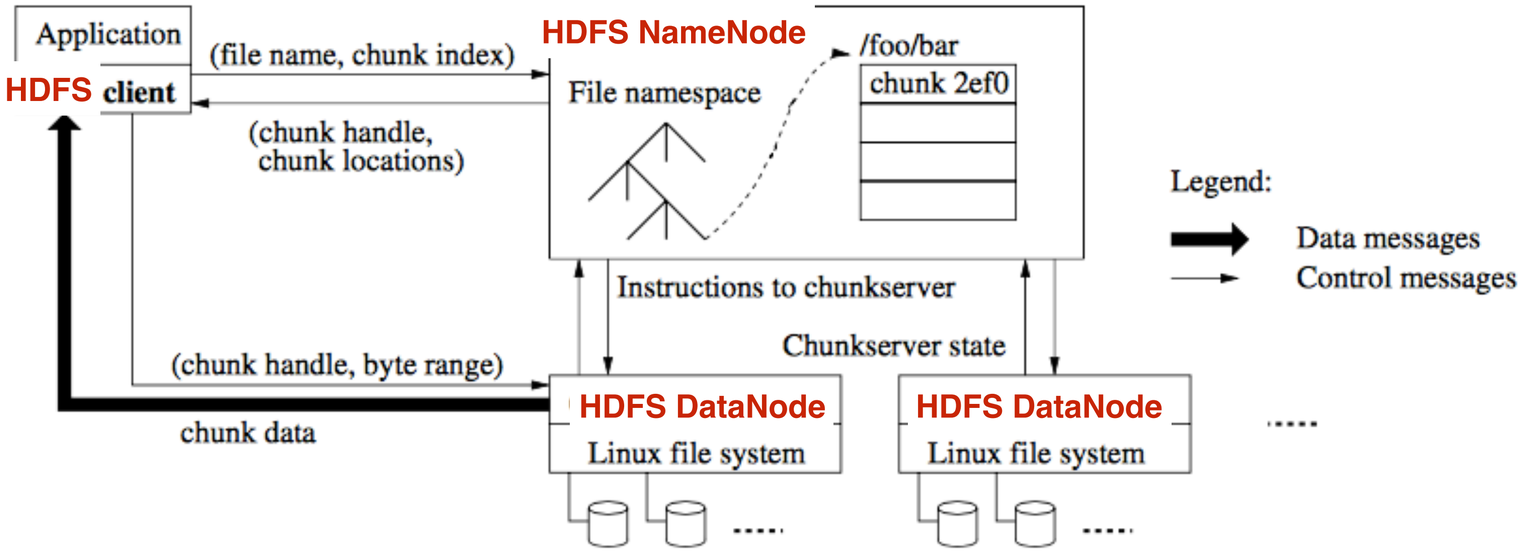
Fault tolerance via re-execution

- Ideally, fine granularity tasks (more tasks than machines)
- On worker-failure:
  - Re-execute completed and in-progress map tasks
  - Re-executes in-progress reduce tasks
  - Commit completion to master
- On master-failure:
  - Recover state (master checkpoints in a primary-backup mechanism)

MapReduce in Practice

- Originally presented by Google in 2003
- Widely used today (**Hadoop** is an open source implementation)
- Many systems designed to have easier programming models that compile into MapReduce code (Pig, Hive)

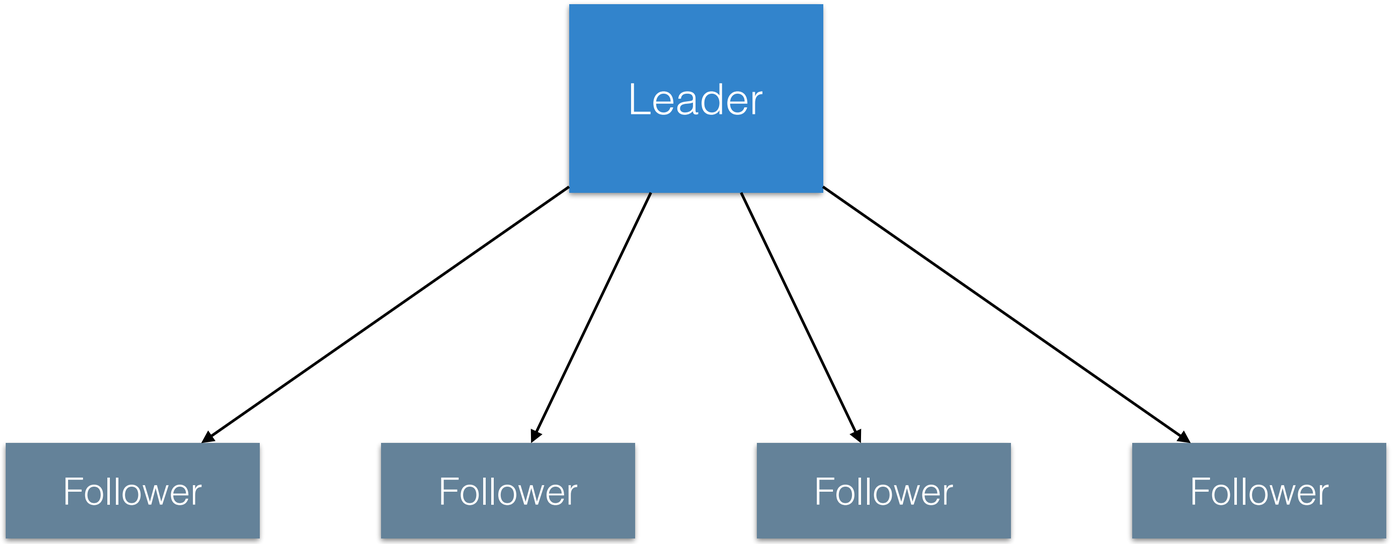
Hadoop: HDFS


Application
HDFS client
(file name, chunk index)
(chunk handle, chunk locations)
HDFS NameNode
File namespace
/foo/bar
chunk 2ef0
Instructions to chunkserver
Chunkserver state
(chunk handle, byte range)
chunk data
HDFS DataNode
Linux file system
HDFS DataNode
Linux file system
Legend:
Data messages
Control messages

HDFS (GFS Review)

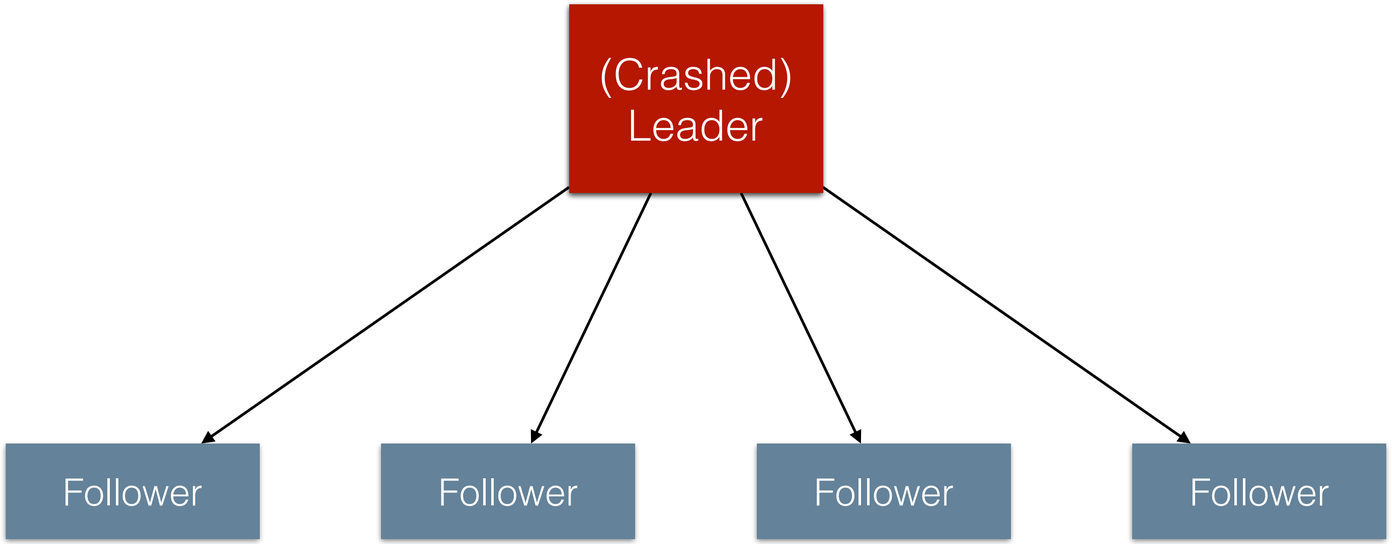
- Files are split into blocks (128MB)
- Each block is replicated (default 3 block servers)
- If a host crashes, all blocks are re-replicated somewhere else
- If a host is added, blocks are rebalanced
- Can get awesome locality by pushing the map tasks to the nodes with the blocks (just like MapReduce)

Leader/follower distributed model



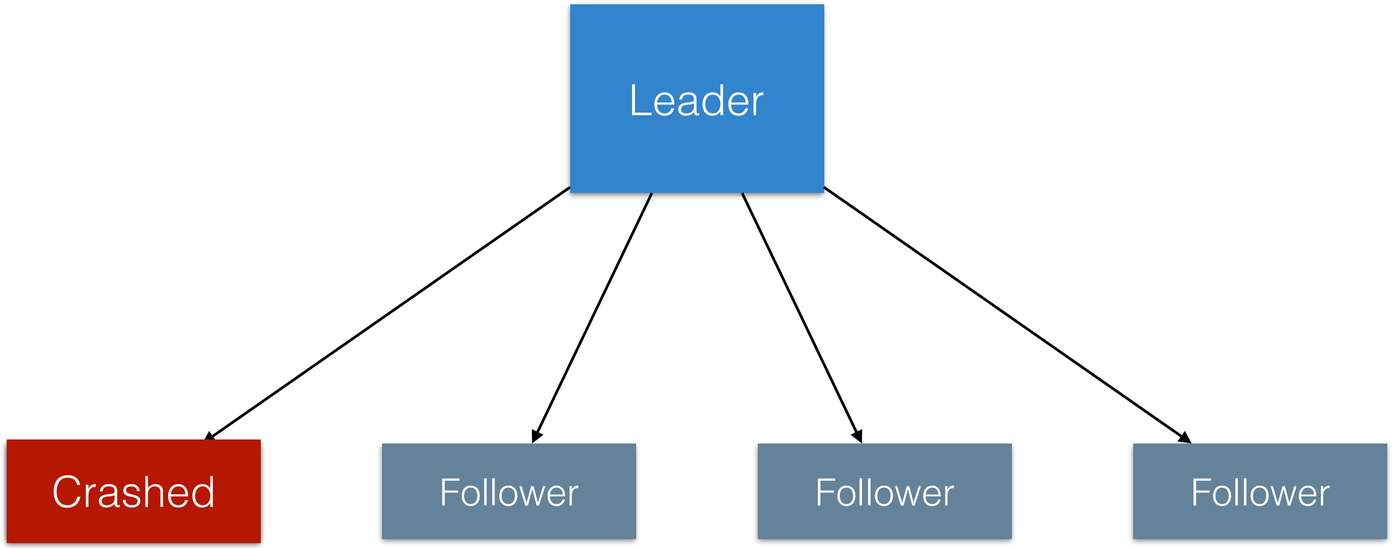
Leader/follower distributed model

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



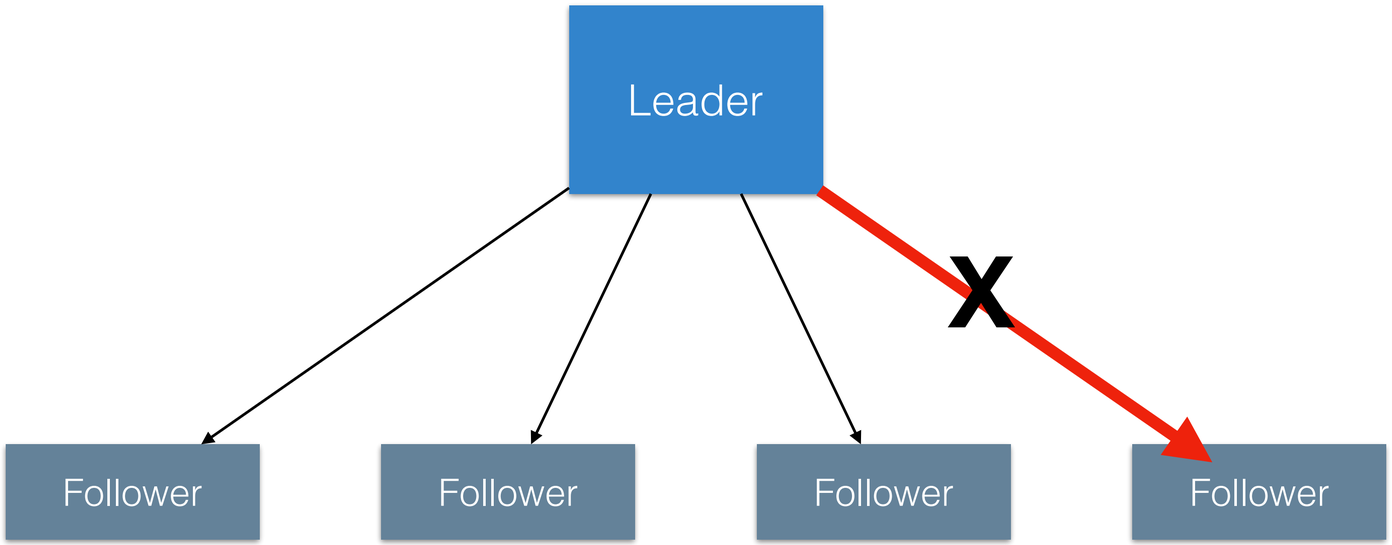
Leader/follower distributed model

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



Leader/follower distributed model

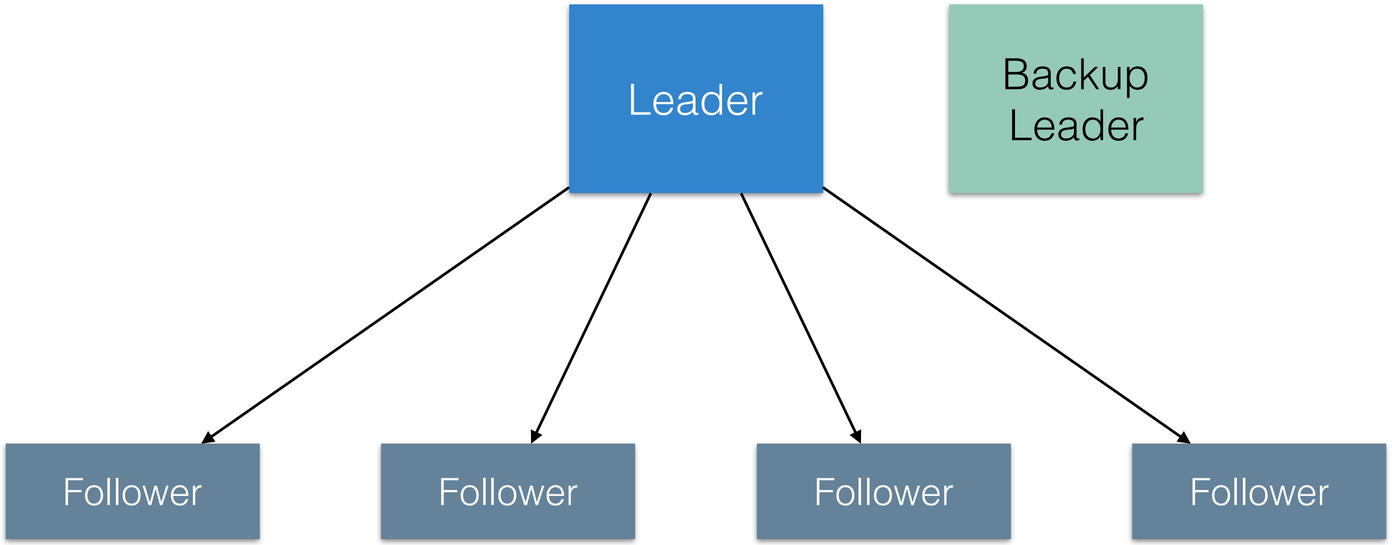
- If a follower doesn't receive a task (network link failure)?
- Again, not as bad, but need to detect
- Will need to try to re-establish link (difference between "there is no work left to do" and "I just didn't hear I needed to do something)



Coordination

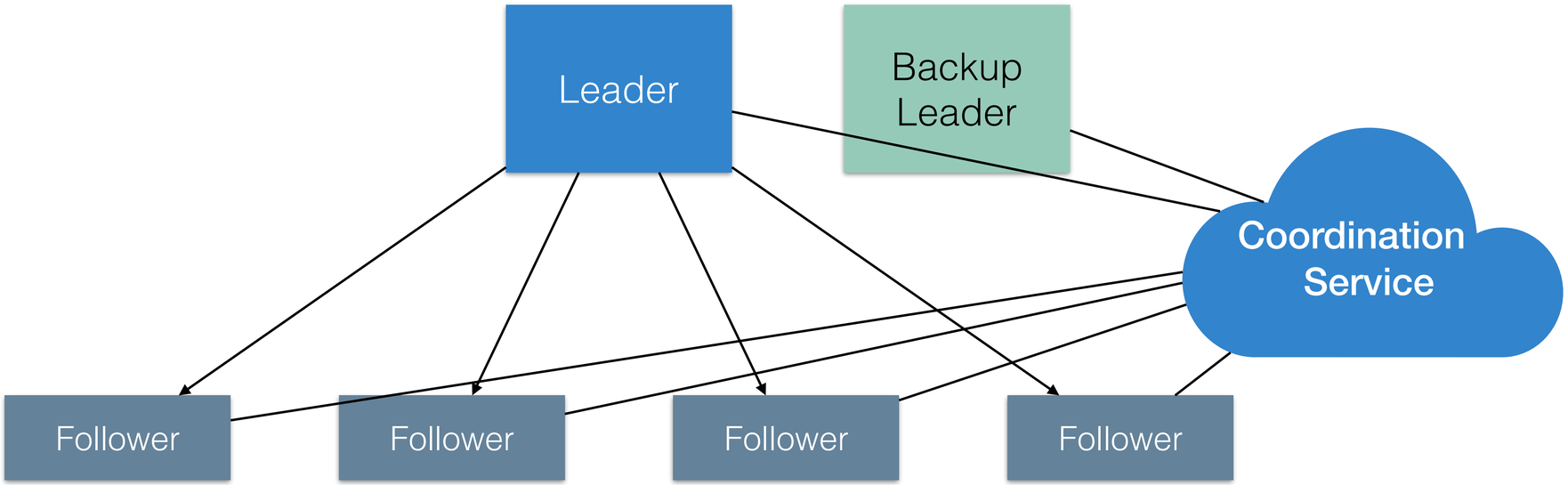
- Semaphores
- Queues
- Transactions
- Locks
- Barriers

Strawman Fault Tolerant Leader/Follower System



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

Fault Tolerant Leader/Follower System

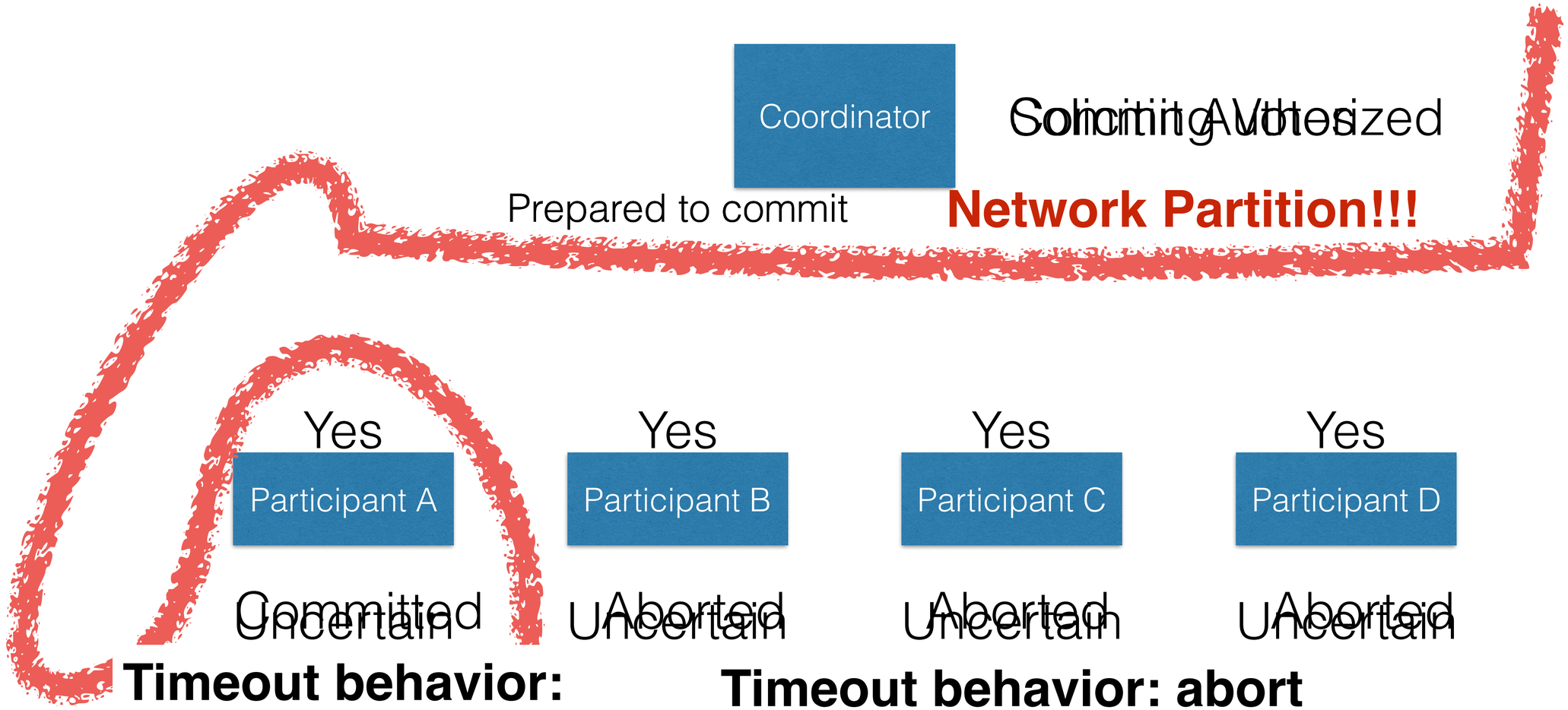


Coordination service handles all of those tricky parts.
But can't the coordination service fail?

Fault-Tolerant Distributed Coordination

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

Review: Partitions



Review: FLP - Intuition

- Why can't we make a protocol for consensus/agreement that can tolerate both partitions and node failures?
- To tolerate a partition, you need to assume that **eventually** the partition will heal, and the network will deliver the delayed packages
- But the messages might be delayed **forever**
- Hence, your protocol would not come to a result, until **forever** (it would not have the **liveness** property)

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

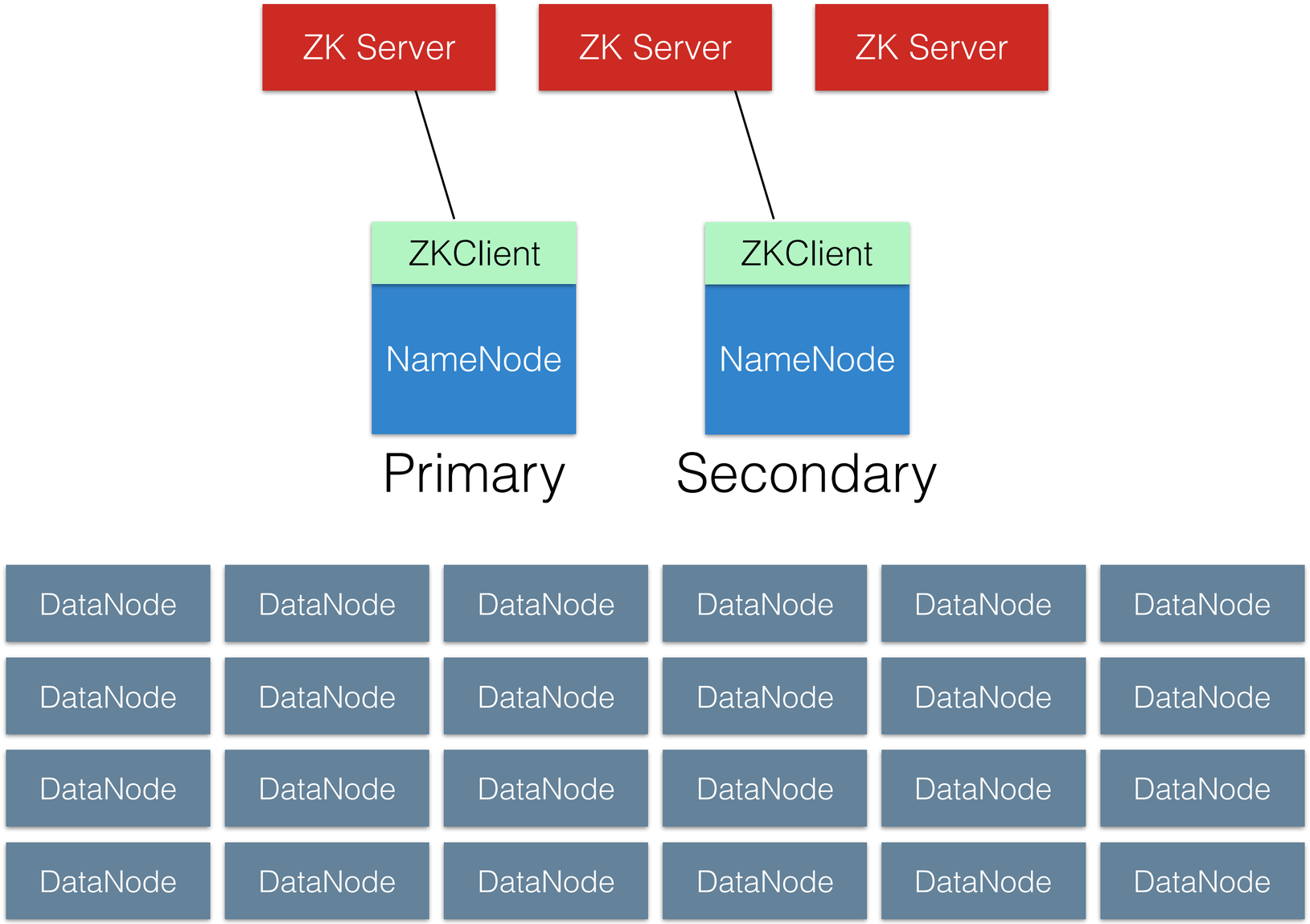
ZooKeeper - Guarantees

- **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

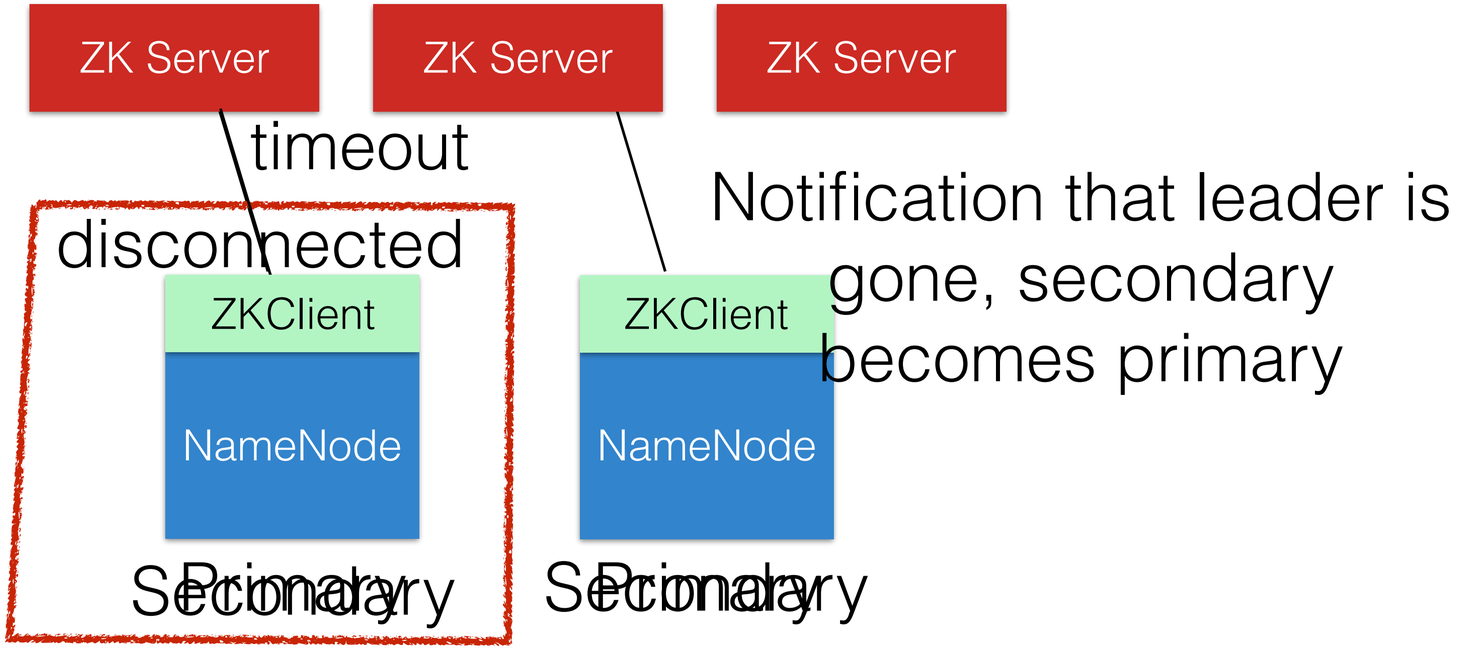
Example use-cases

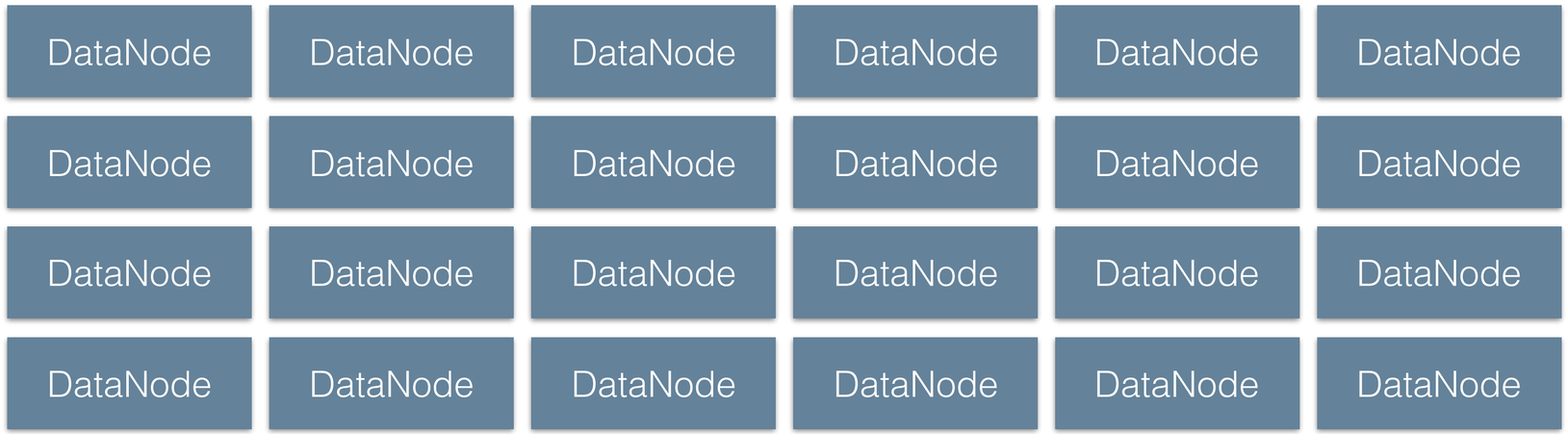
- Configuration management (which servers are doing what role?)
- Synchronization primitives
- Anti-use cases:
  - Storing large amounts of data
  - Sharing messages and data that don't require liveness/durability guarantees

Hadoop + ZooKeeper

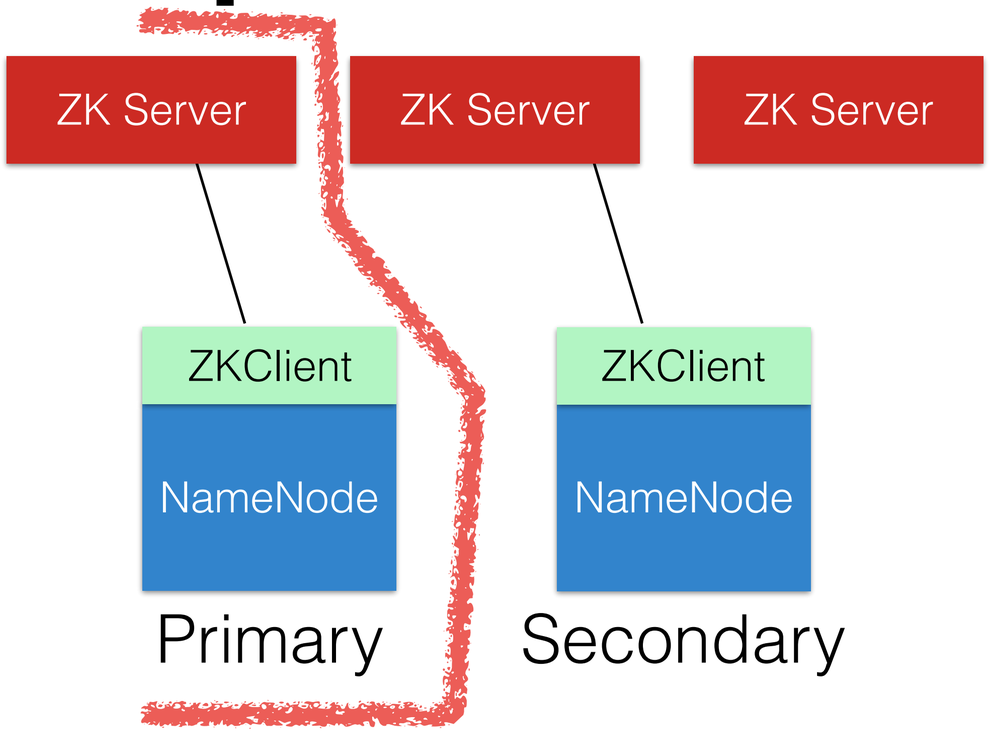


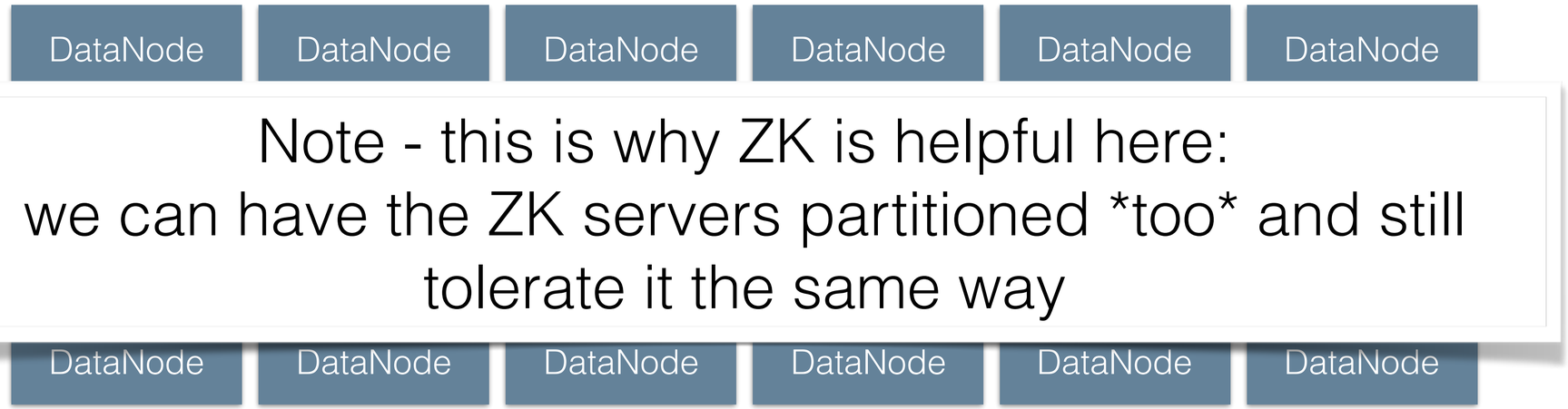
Hadoop + ZooKeeper





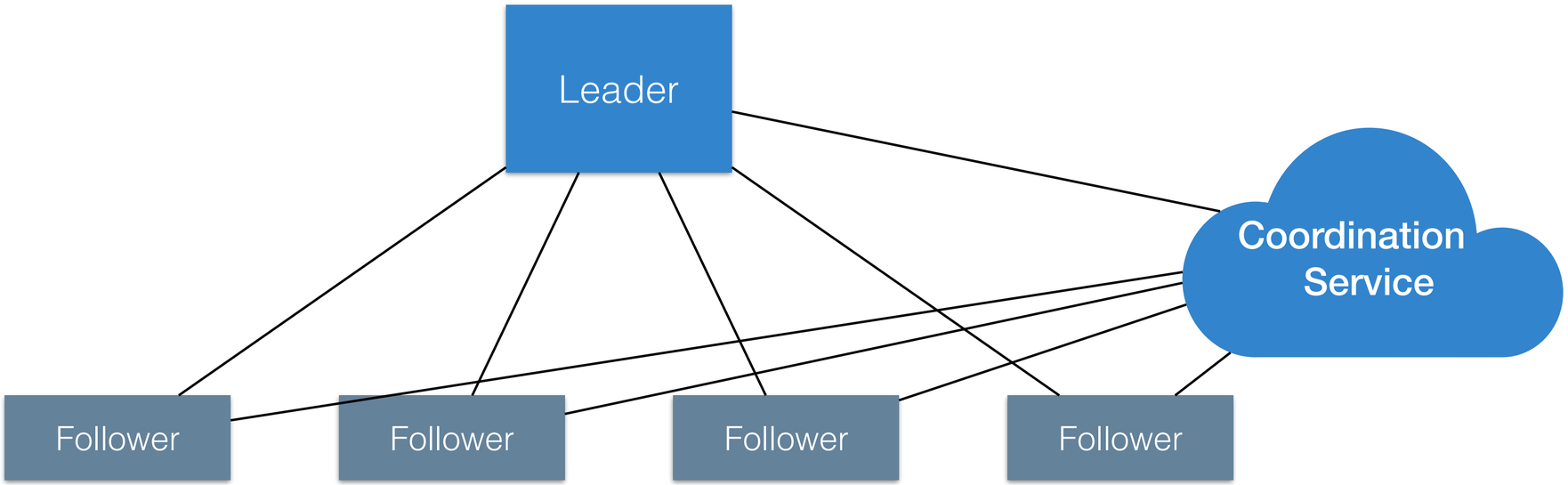
Hadoop + ZooKeeper





Note - this is why ZK is helpful here:
we can have the ZK servers partitioned *too* and still tolerate it the same way

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?

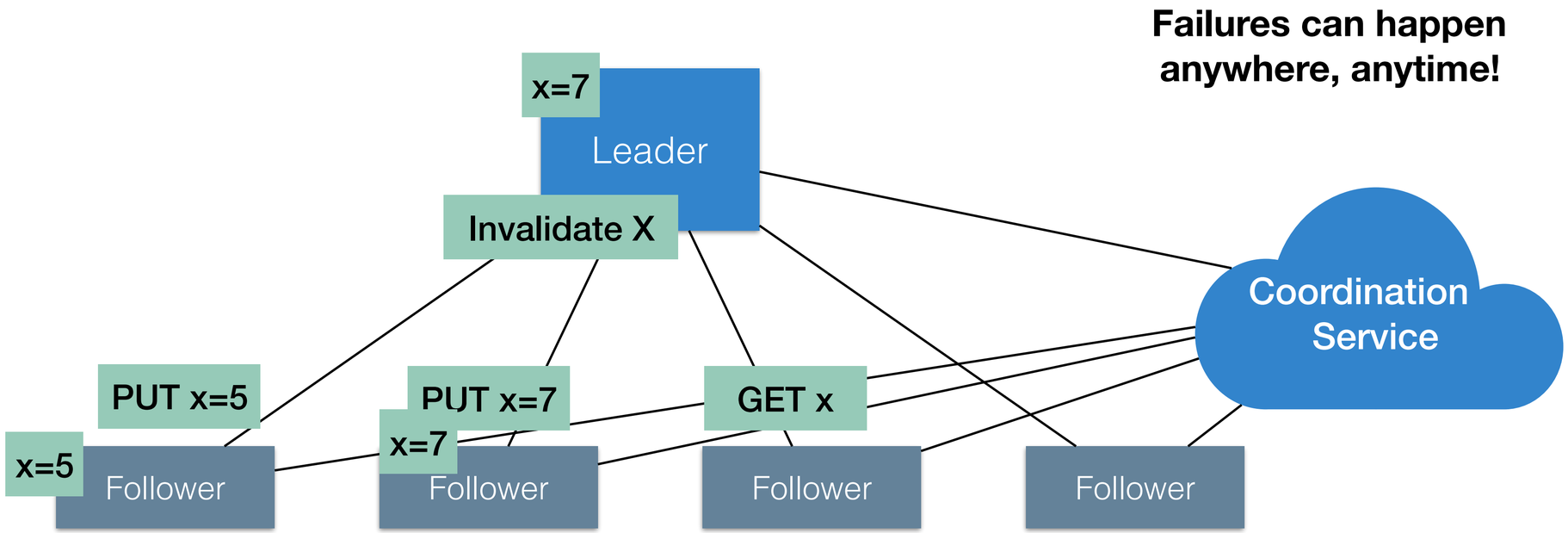
Leader broadcasts read-invalidates to clients

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?

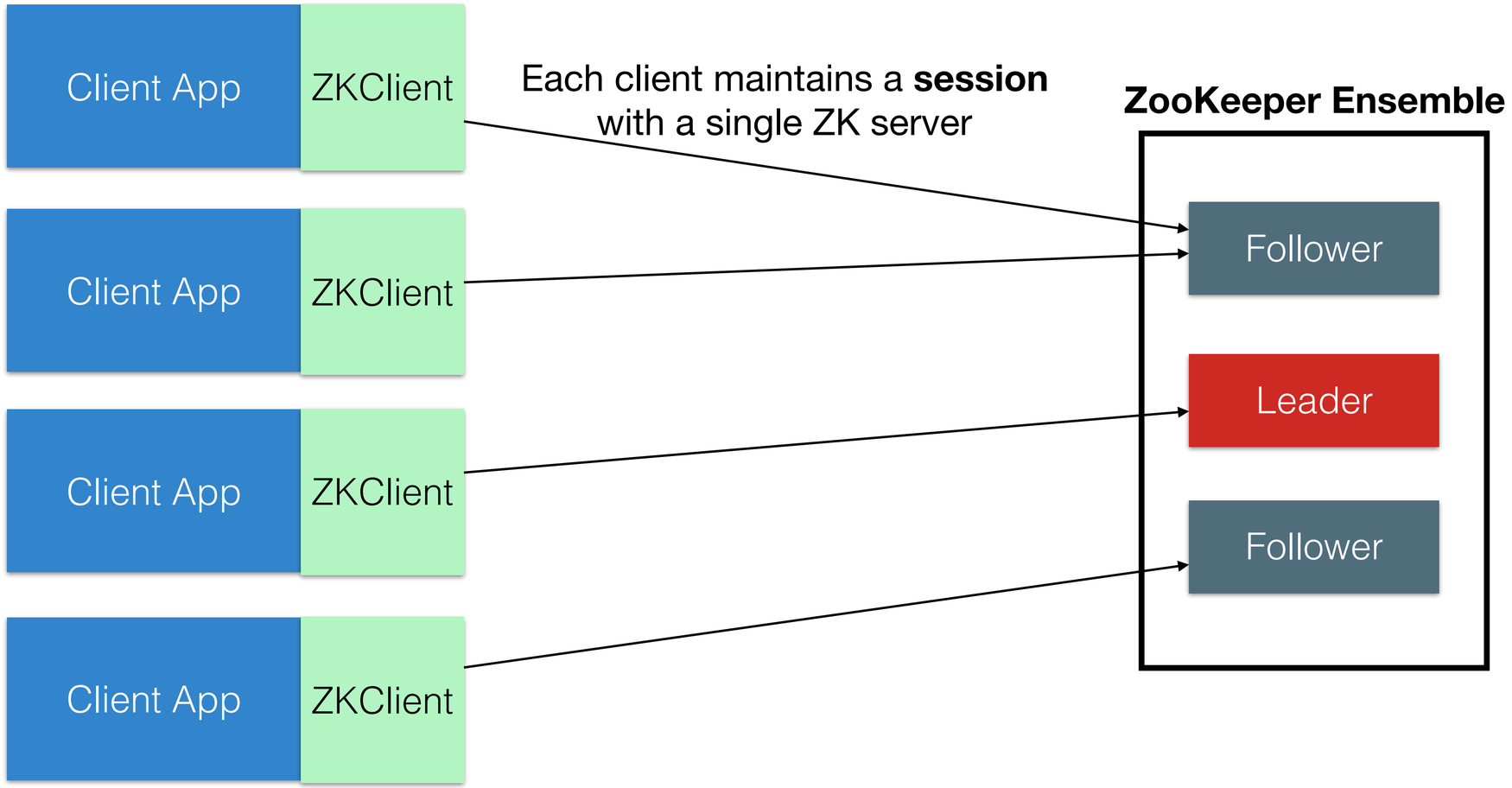
Leader broadcasts read-invalidates to clients

Who is still alive?

Reads processed on each client

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

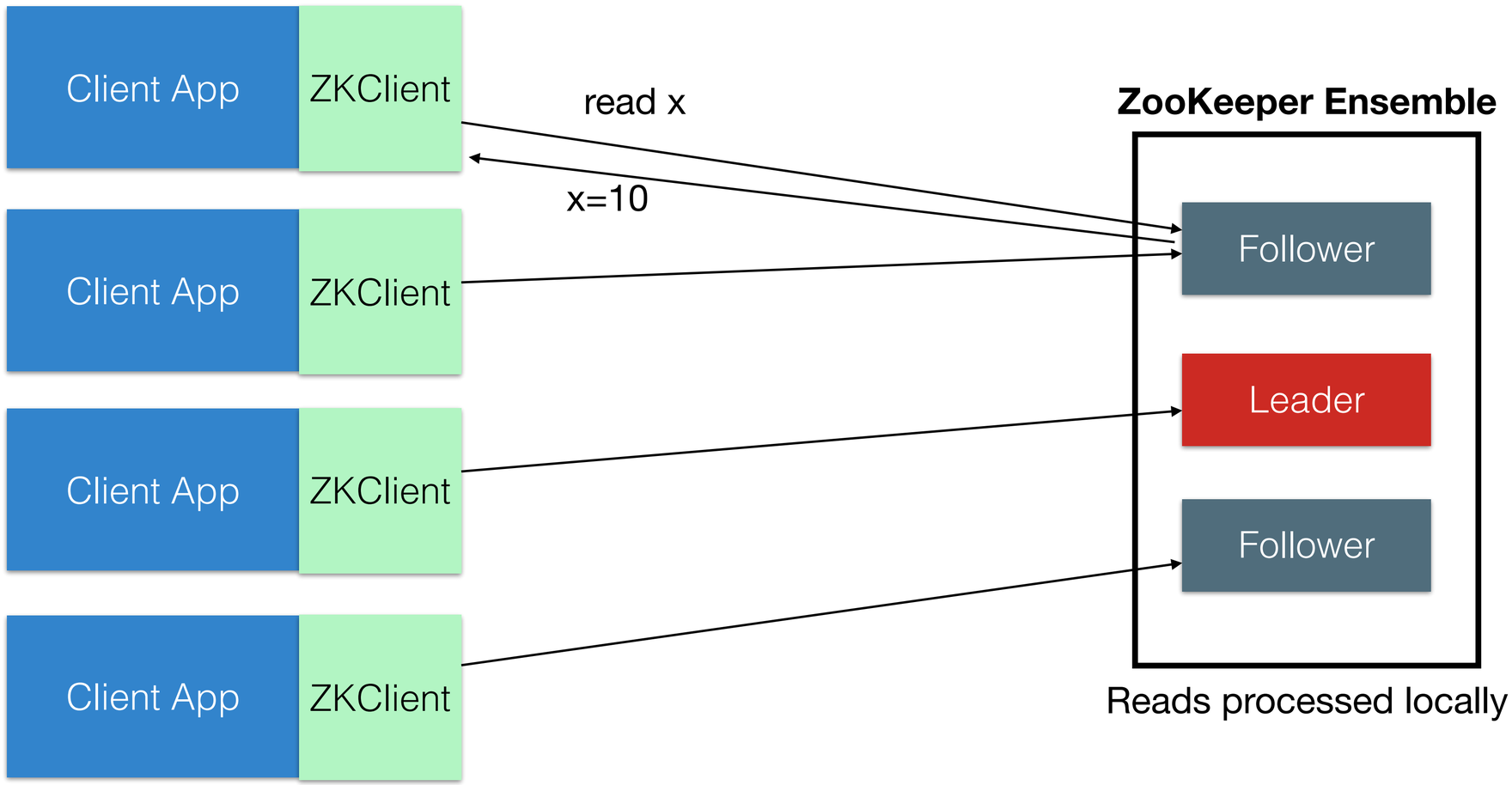
ZooKeeper - Overview



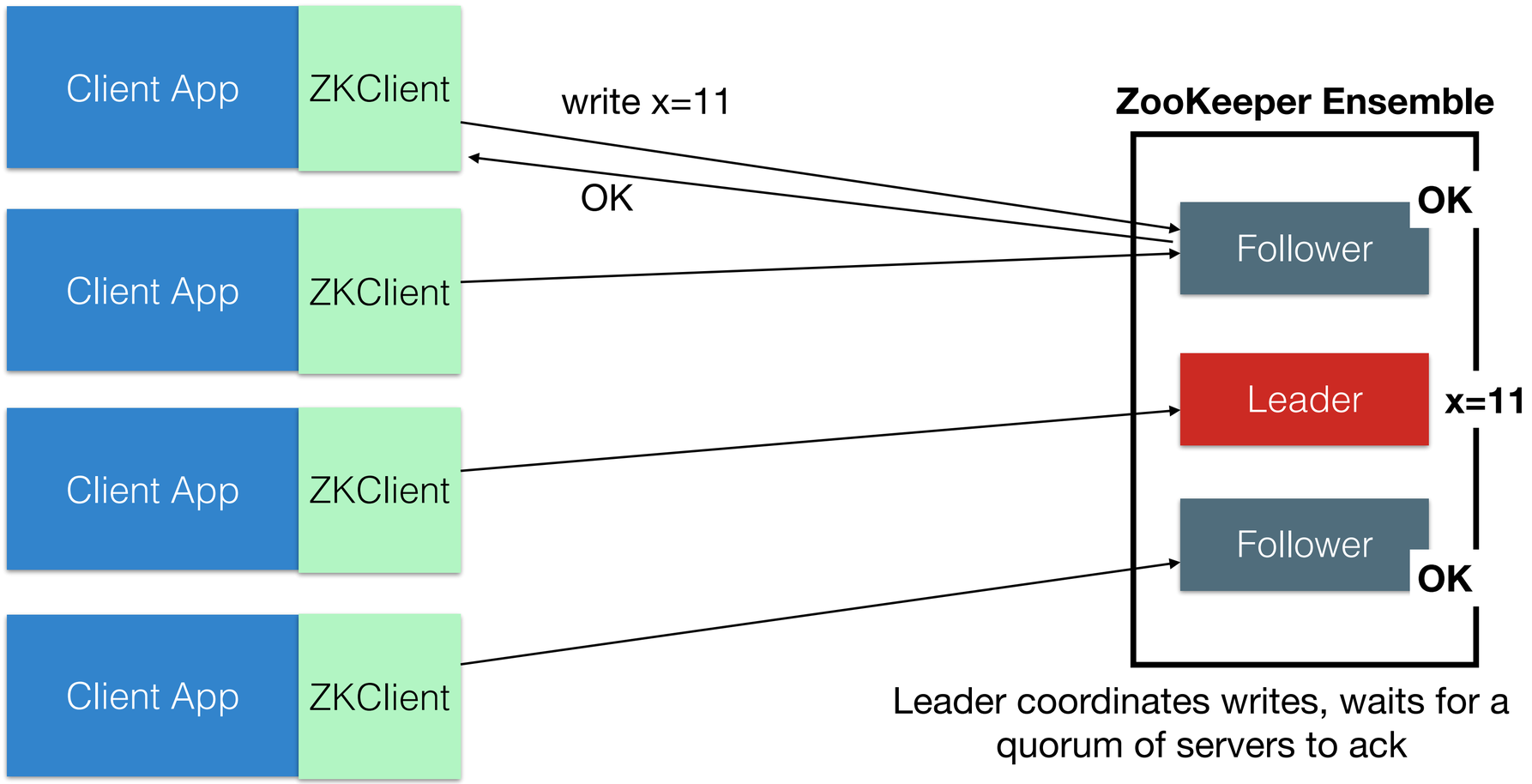
ZooKeeper - Sessions

- 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

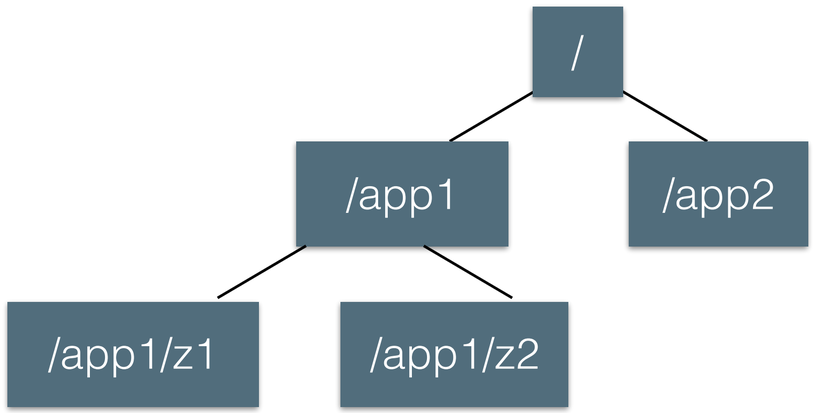


ZooKeeper - Overview



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

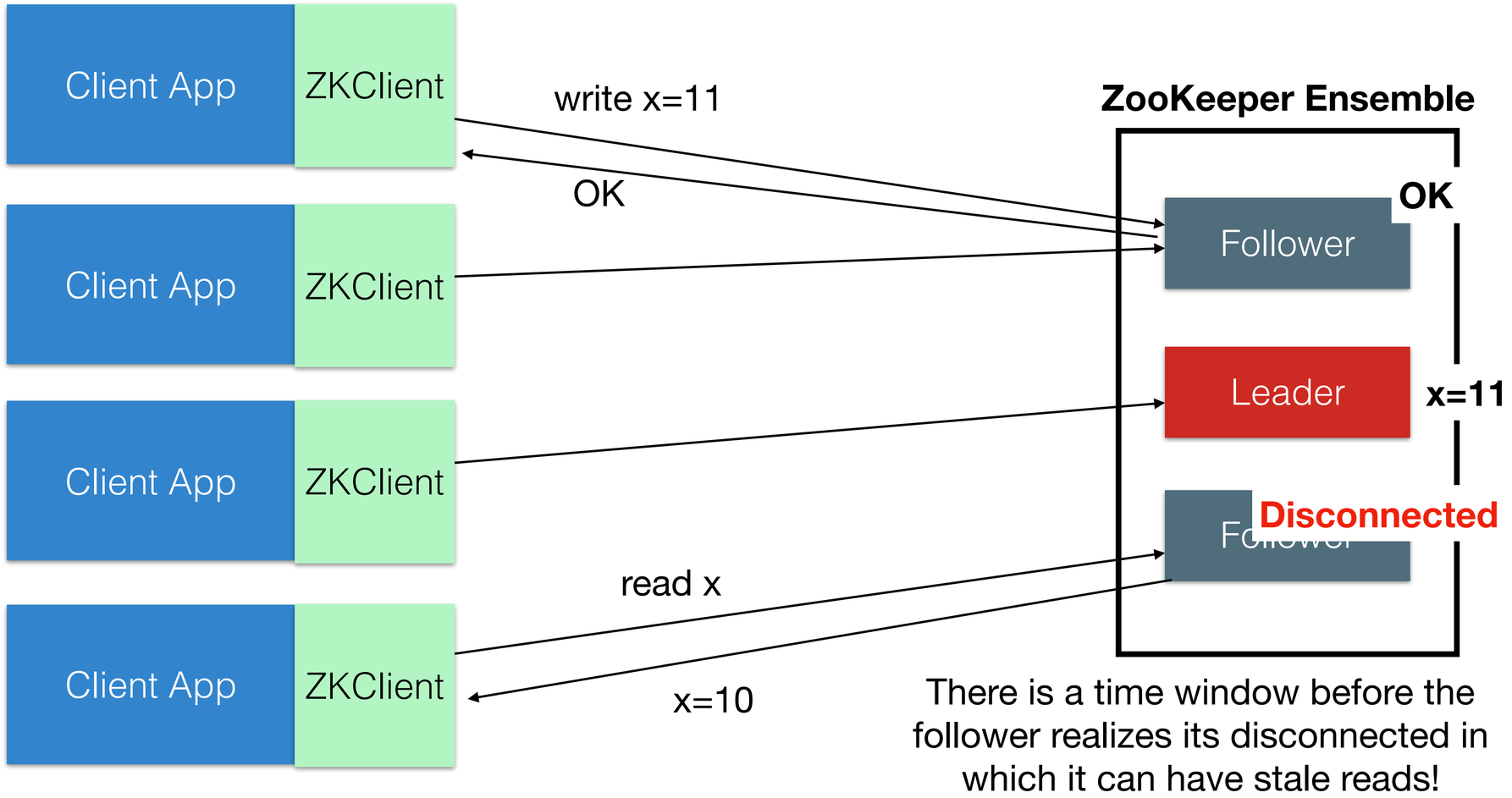
ZooKeeper - Znode Types

- 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

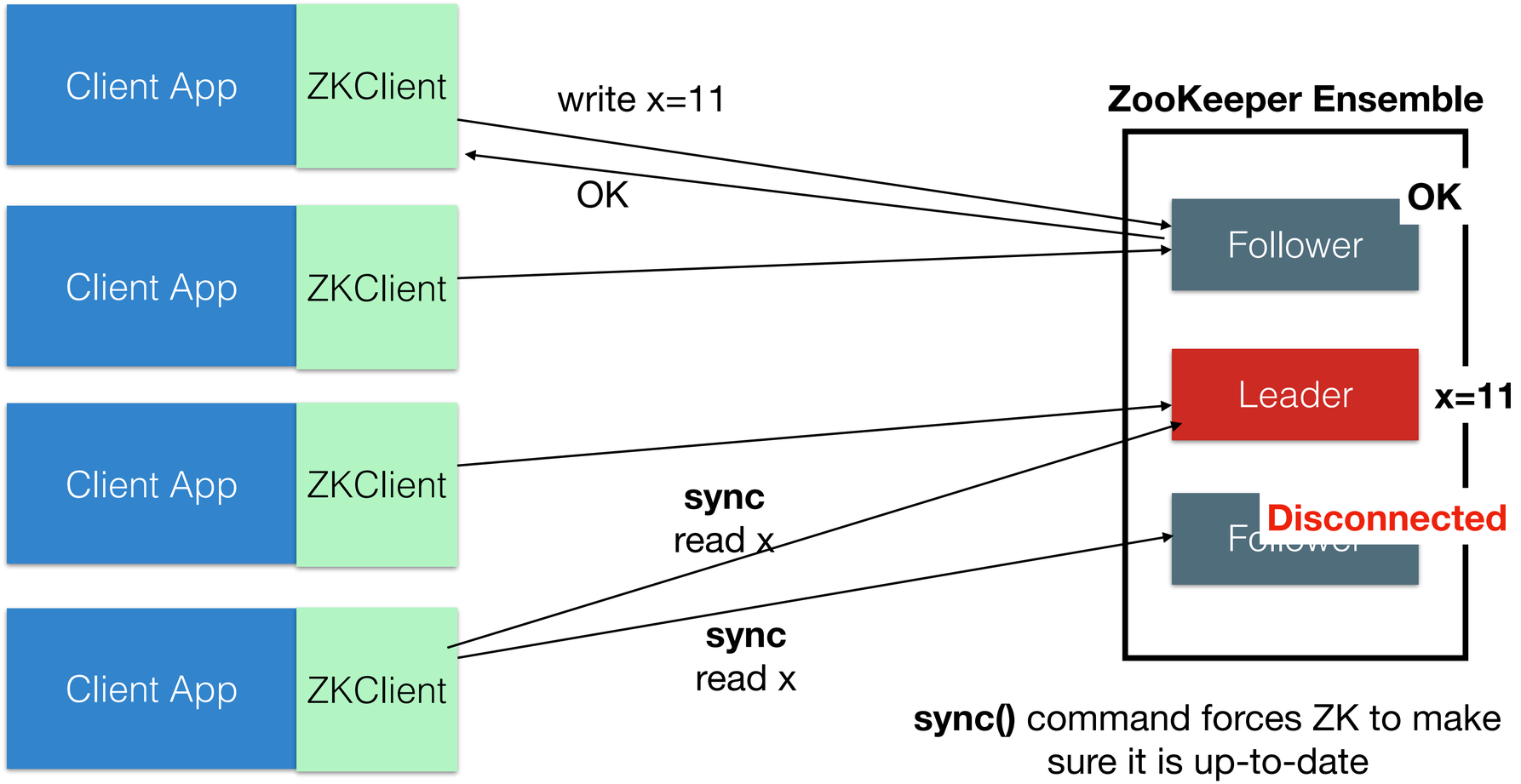
ZooKeeper - API

- 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

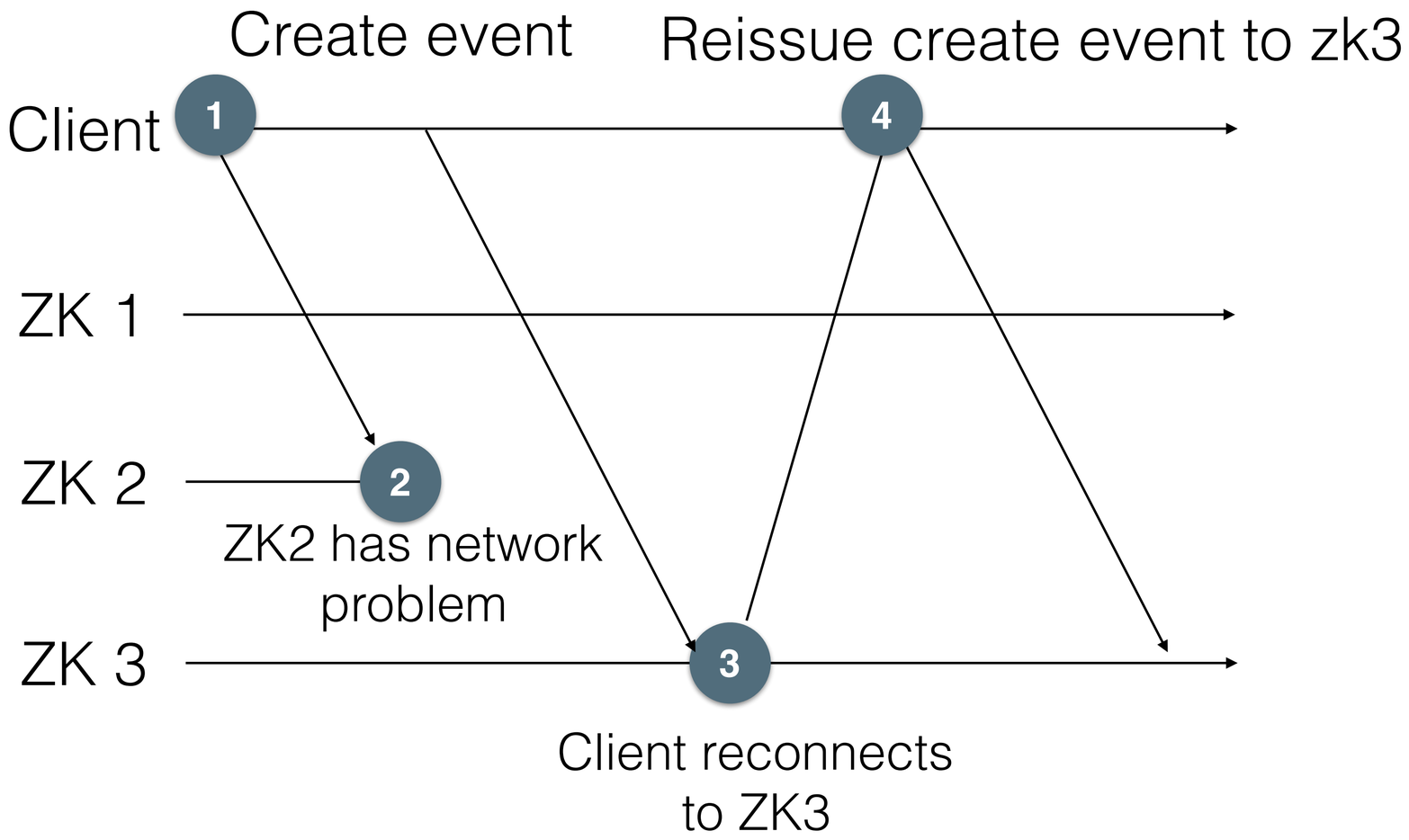
How Many ZooKeepers?

- 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**

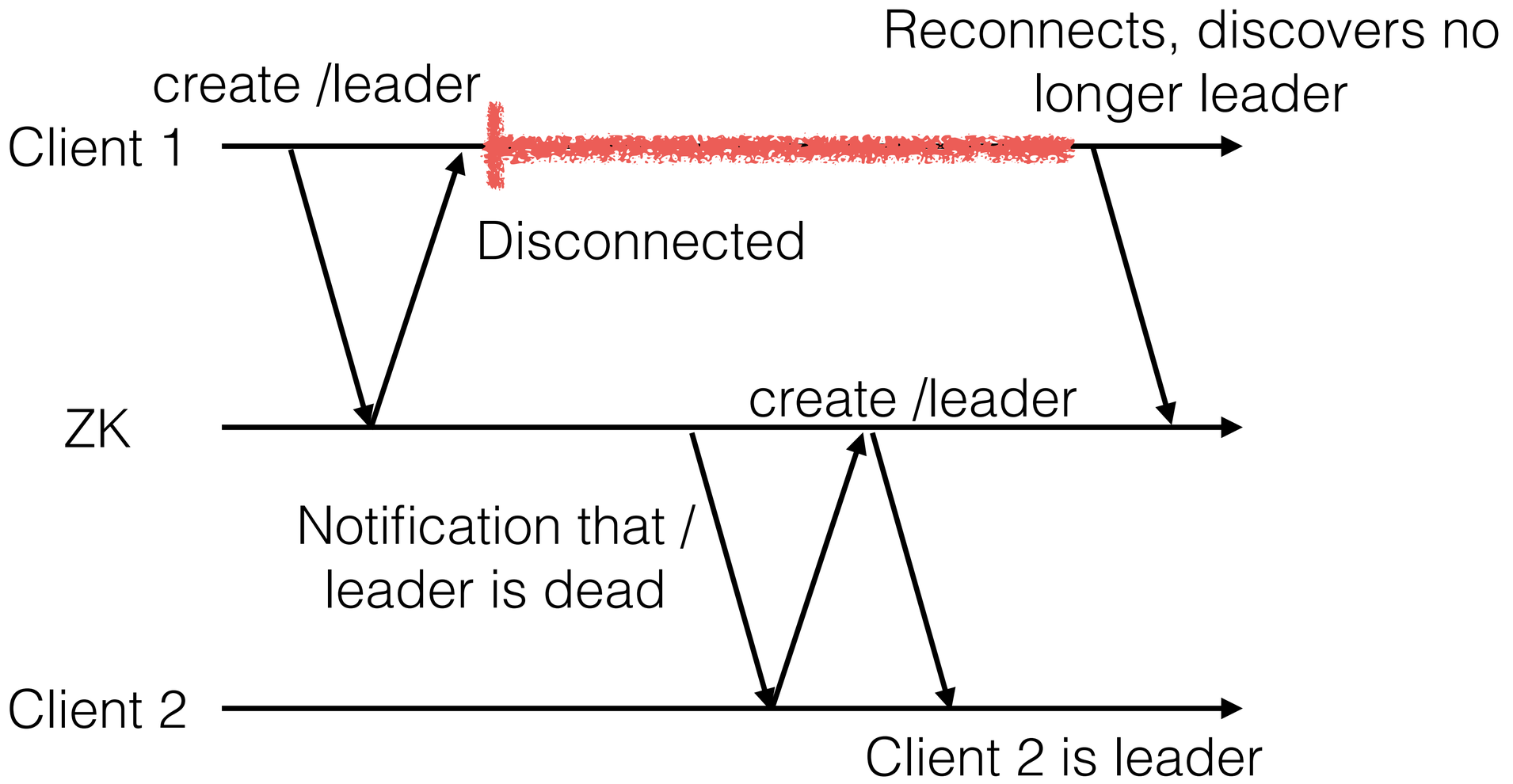
Failure Handling in ZK



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

This work is licensed under a Creative Commons Attribution-ShareAlike license

- Thanks to Luís Pina for assistance with these slides.
- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/
- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.
- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.