Distributed Coordination (Mutual Exclusion, Consensus)

Includes material adapted from Van Steen and Tanenbaum's Distributed Systems book

Survey Feedback

- Breadth vs Depth
- Example Use Cases
- Project Difficulty
- Using cloud trial version – hybrid + on premise VMs
- Programming Language - Go

SCHEDULE

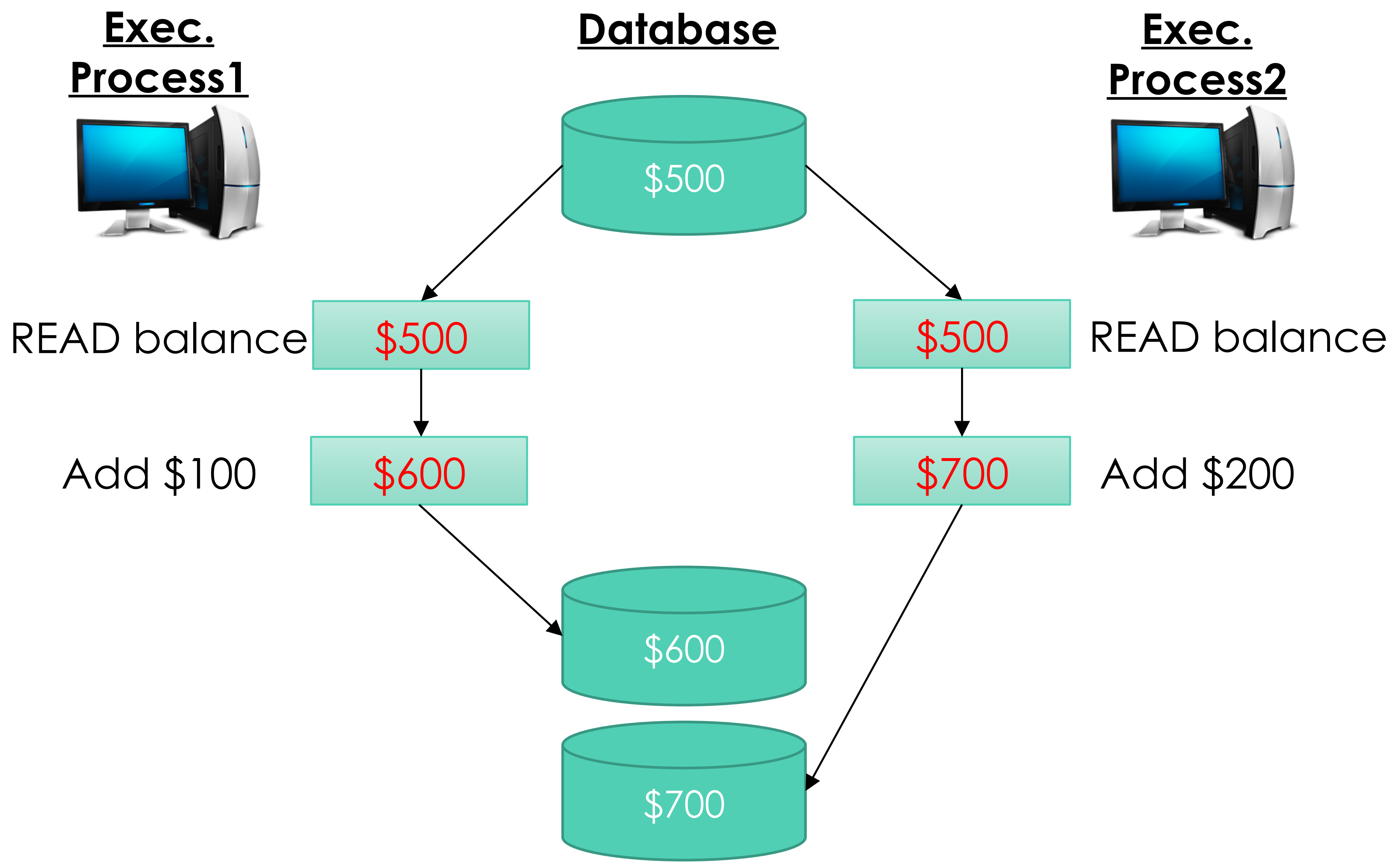
- Remaining Topics
- Midterm
- Final Project

This Week: Distributed Coordination

- Distributed Locking
- Consensus
- Elections
- State Machine Replication
- Blockchain

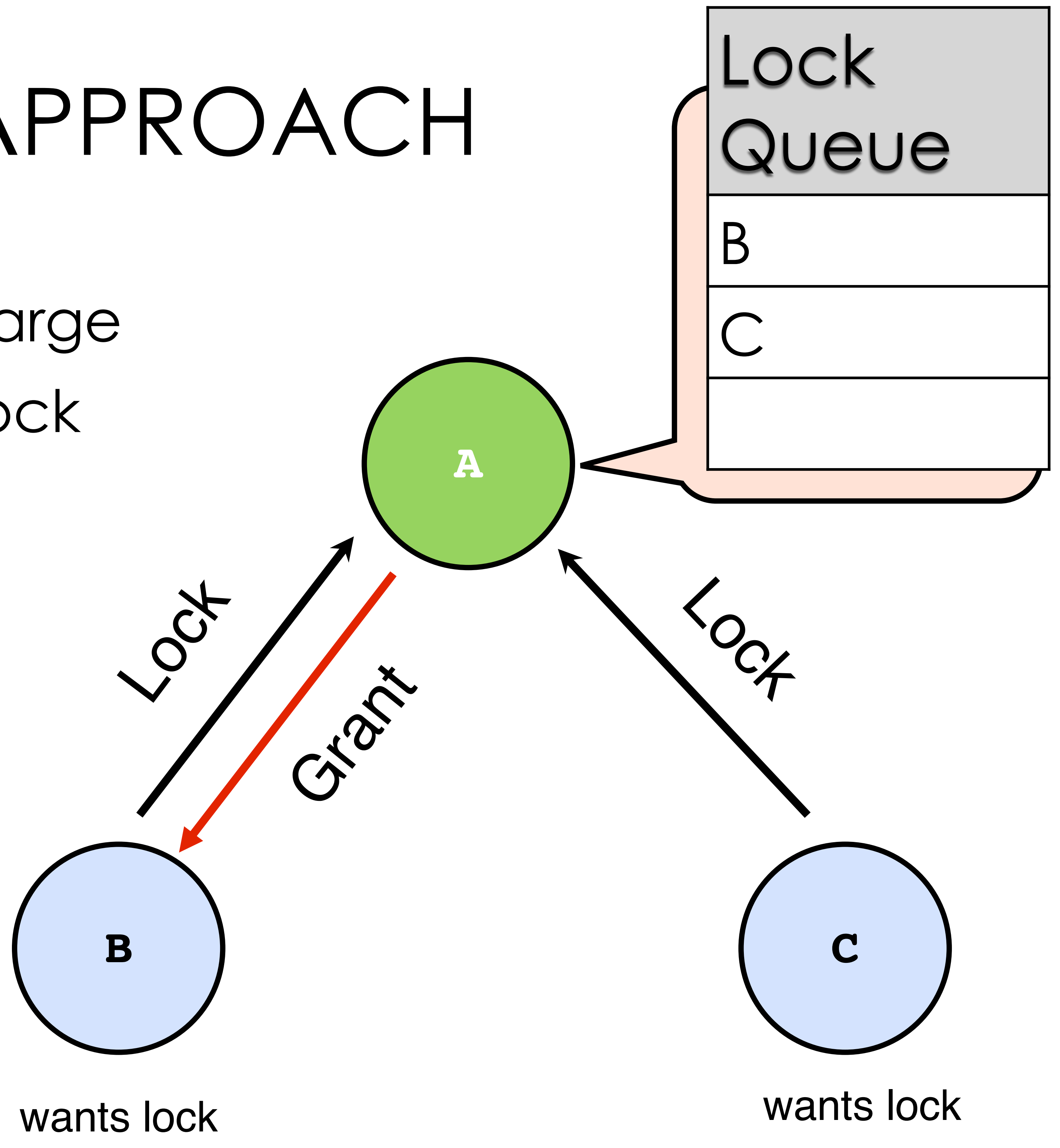
Why Lock?

- Locks let us protect a **shared** resource
  - A database, values in shared memory, files on a shared file system, throttle control on a drone, etc
- How to manage a lock in a distributed environment?
- How do locks limit scalability?



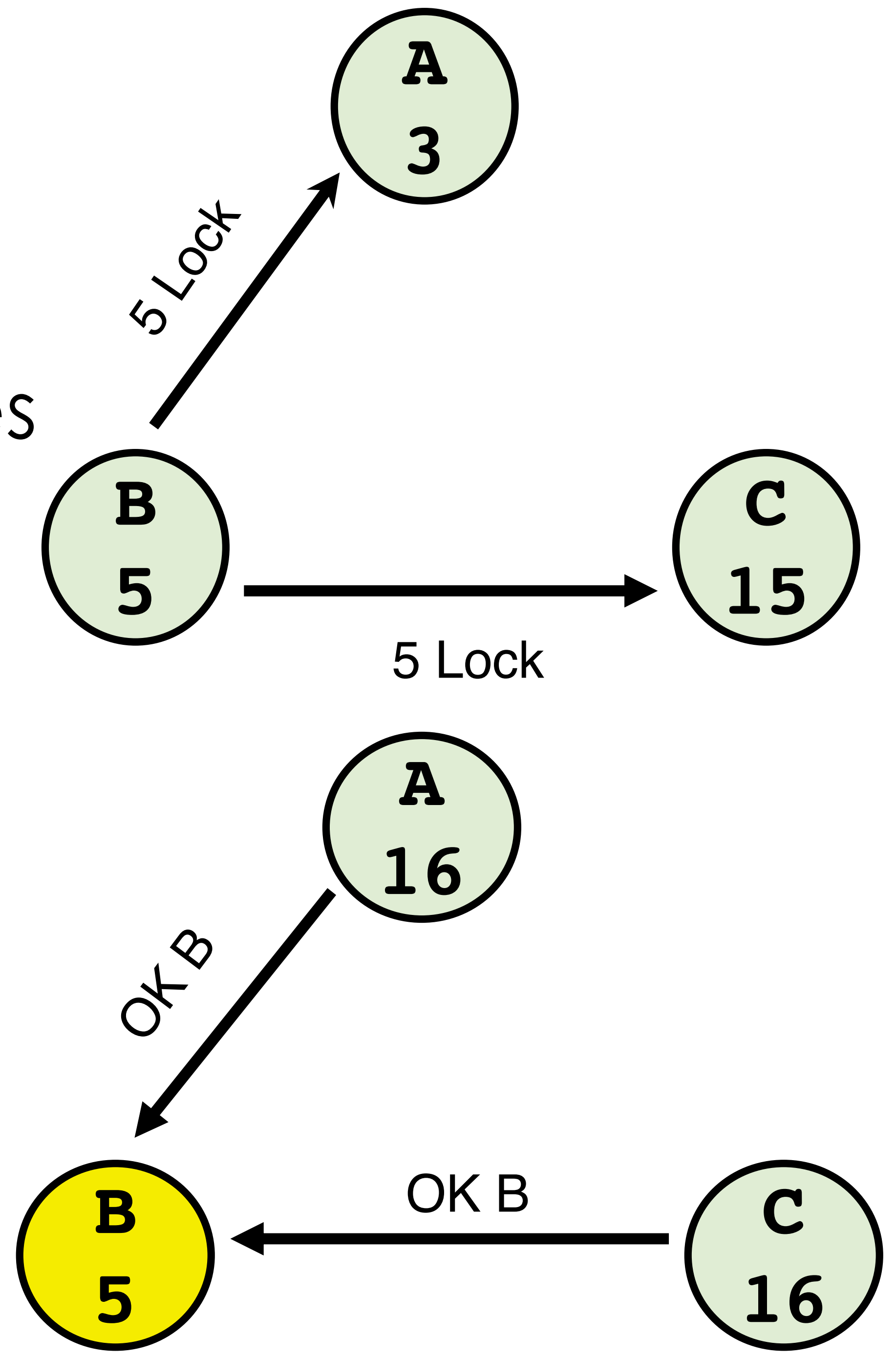
Centralized Approach

- Simplest approach: put one node in charge
- Other nodes ask coordinator for each lock
  - Block until they are granted the lock
  - Send release message when done
- Coordinator can decide what order to grant lock
- Do we get:
  - Mutual exclusion?
  - Progress?
  - Resilience to failures?
  - Balanced load?

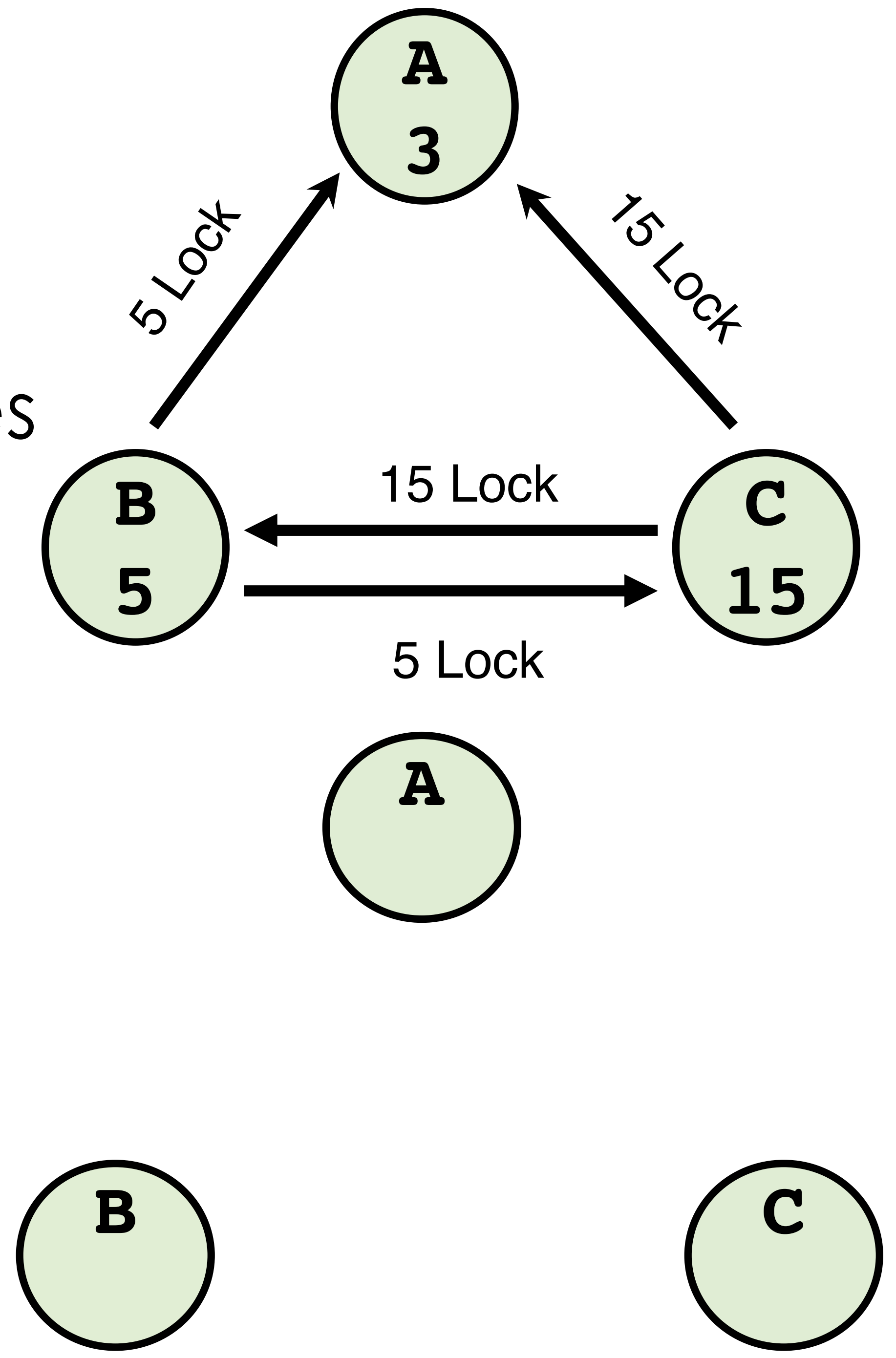


Distributed Approach

- Use Lamport Clocks to order lock requests across nodes
- Send Lock message with ++clock
  - Wait for OKs from all nodes
- When receiving Lock msg:
  - Update clock following Lamport's rules
  - Send OK if not interested
  - If I want the lock:
    - Send OK if request's clock is smaller than own
    - Else, put request in queue
- When done with a lock:
  - Send OK to anybody in queue



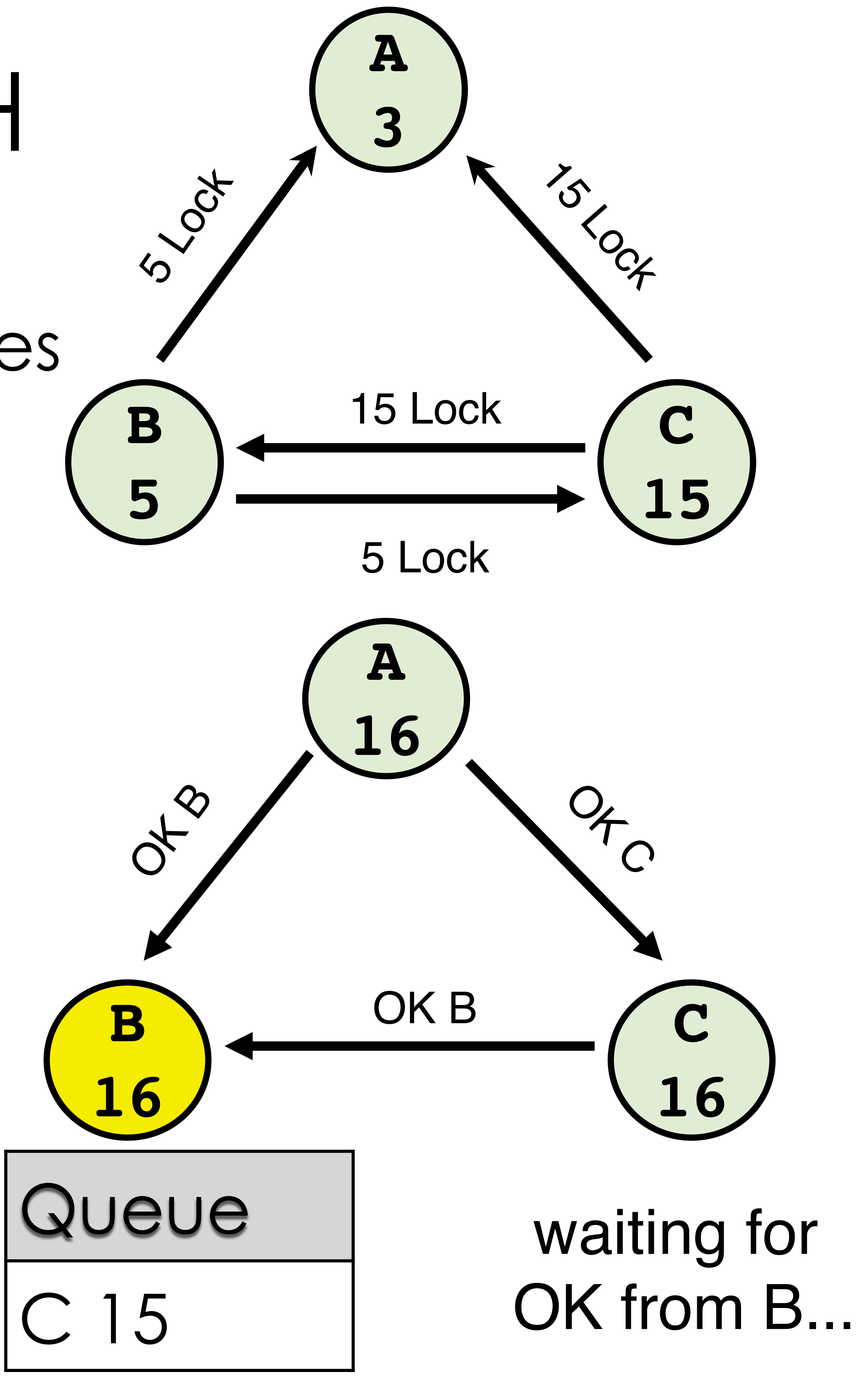
DISTRIBUTED APPROACH



- Use Lamport Clocks to order lock requests across nodes
- Send Lock message with ++clock
  - Wait for OKs from all nodes
- When receiving Lock msg:
  - Update clock following Lamport's rules
  - Send OK if not interested
  - If I want the lock:
    - Send OK if request's clock is smaller than own
    - Else, put request in queue
- When done with a lock:
  - Send OK to anybody in queue

Distributed Approach

- Use Lamport Clocks to order lock requests across nodes
- Send Lock message with ++clock
  - Wait for OKs from all nodes
- When receiving Lock msg:
  - Update clock following Lamport's rules
  - Send OK if not interested
  - If I want the lock:
    - Send OK if request's clock is smaller than own
    - Else, put request in queue
- When done with a lock:
  - Send OK to anybody in queue



Comparison

- Messages per lock acquire and release
  - Centralized:
  - Distributed:
- Delay before entry
  - Centralized:
  - Distributed:
- Problems
  - Centralized:
  - Distributed:

Comparison

- Messages per lock acquire and release
  - Centralized: 2+1=3
  - Distributed: 2(n-1)
- Delay before entry
  - Centralized: 2
  - Distributed: 2(n-1) in parallel
- Problems
  - Centralized: Coordinator crashes
  - Distributed: anybody crashes

Is the distributed approach better in any way?

Distributed Systems are Hard

- Going from centralized to distributed can be..
- Slower
  - If everyone needs to do more work
- More error prone
  - 10 nodes are 10x more likely to have a failure than one
- Much more complicated
  - If you need a complex protocol
  - If nodes need to know about all others

Often we need more than just a way to lock a resource!

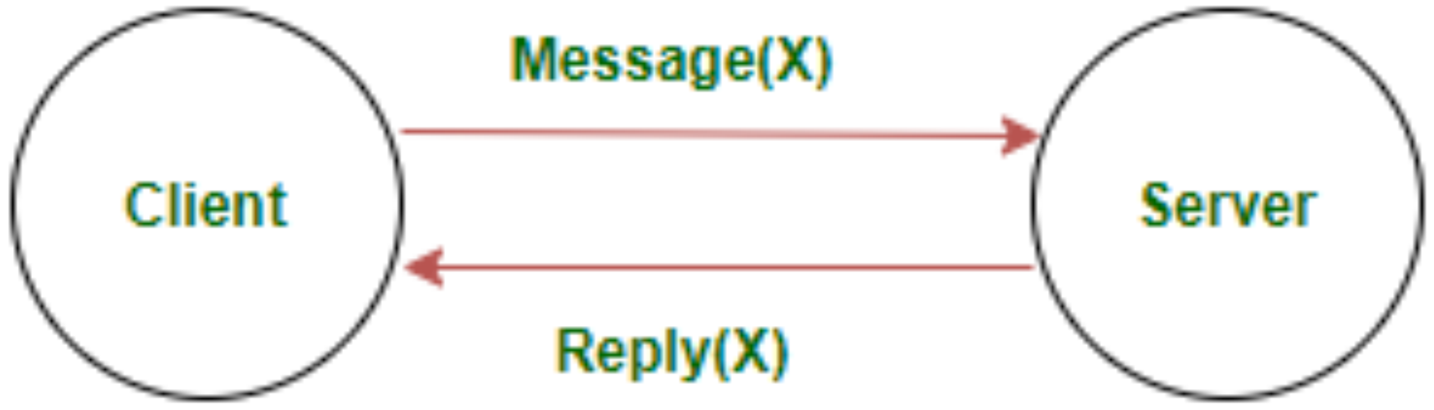
What is the meaning of Consensus

- Consensus is defined by Merriam-Webster as,
  - *general agreement,*
  - *group solidarity of belief or sentiment.*

WHY CONSENSUS?

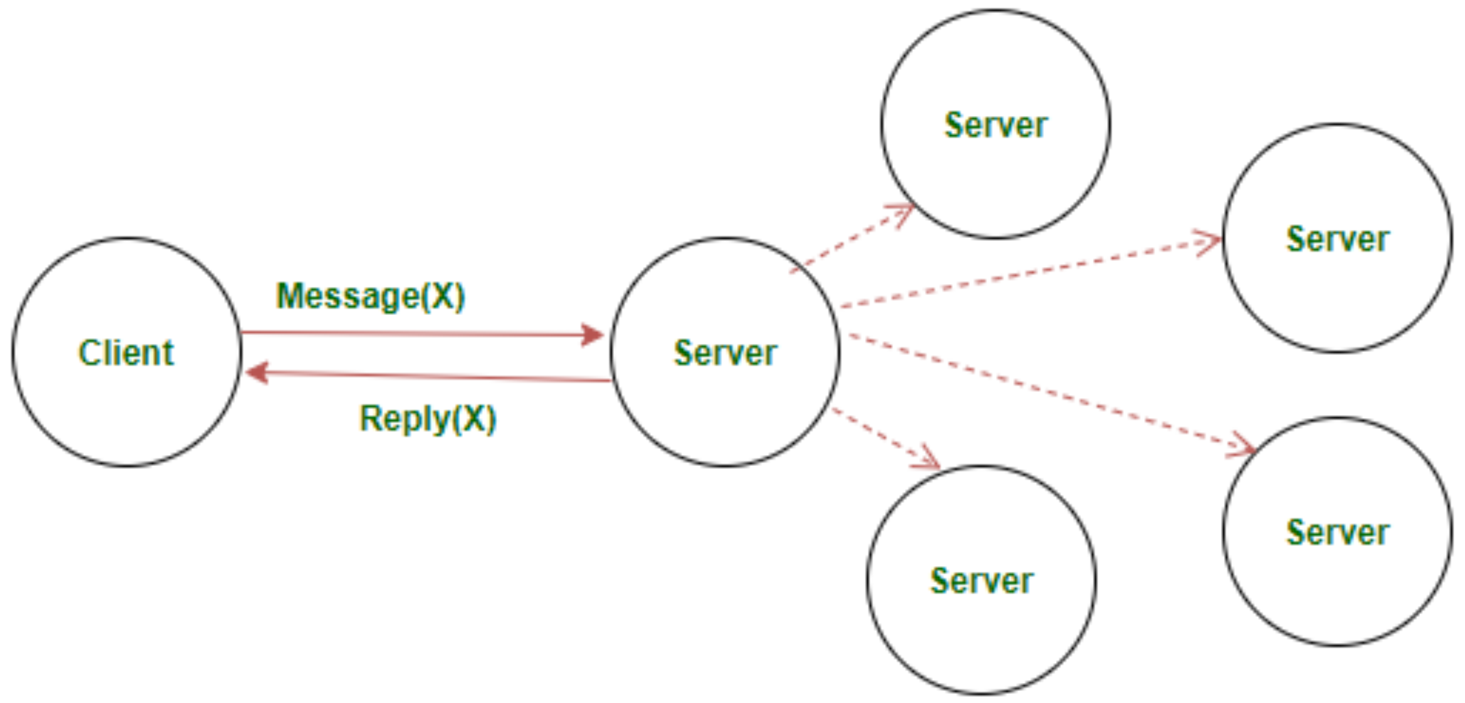
When you sent a request to a server it answers you easily

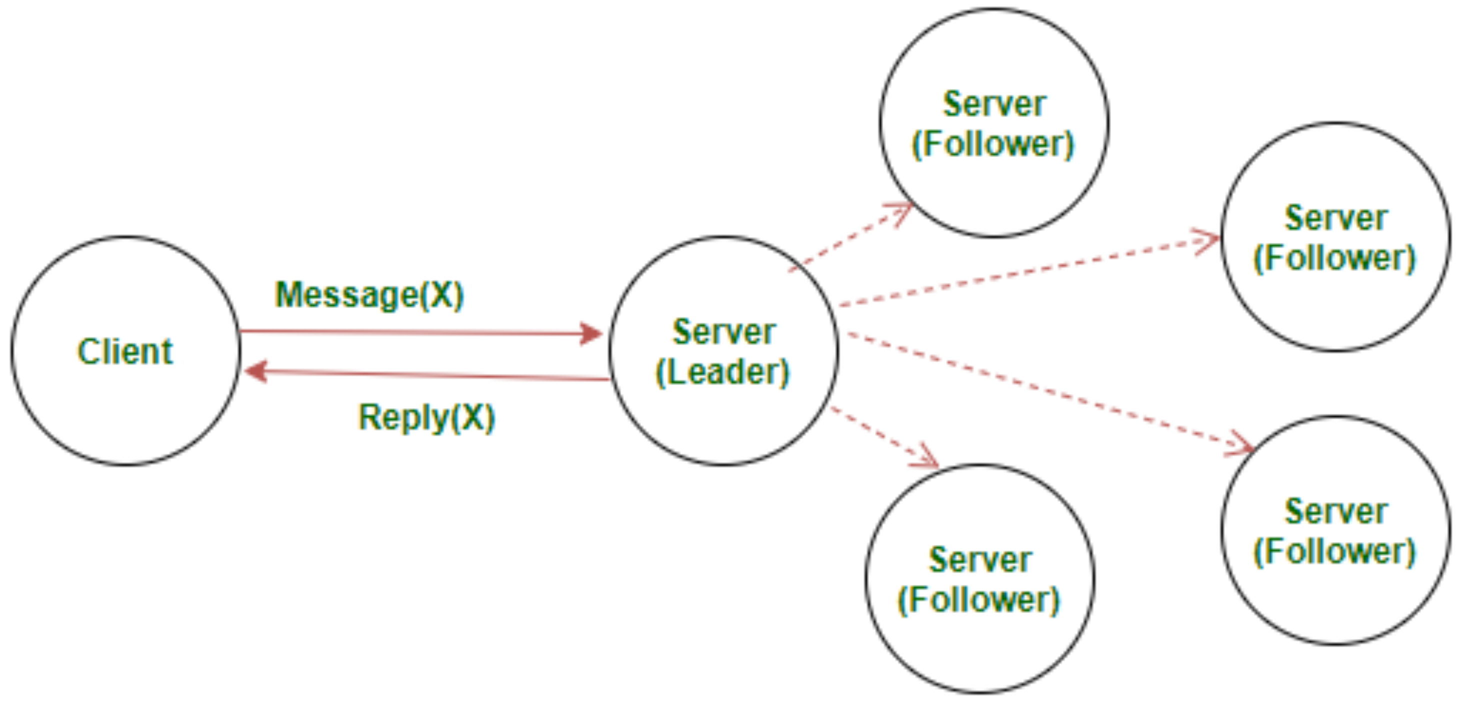
- If server fails, there is no backup
- If the number of requests increase dramatically the server won't be able to respond



WHY CONSENSUS?

- Symmetric :- Any of the multiple servers can respond to the client and all the other servers are supposed to sync up with the server that responded to the client's request, and
- Asymmetric :- Only the elected leader server can respond to the client. All other servers then sync up with the leader server.





Why Consensus?

While this creates a system that is devoid of corruption from a single source, it still creates a major problem.

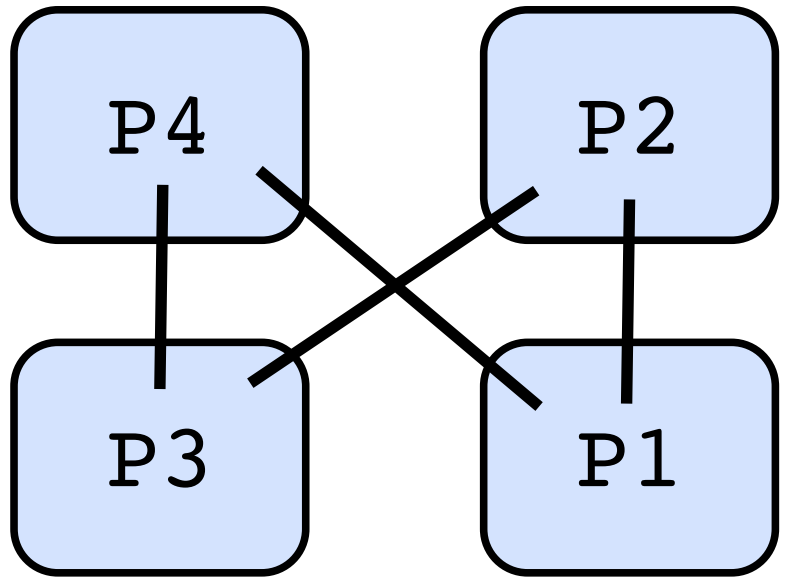
- How are any decisions made?
- How does anything get done?

Consensus Objectives

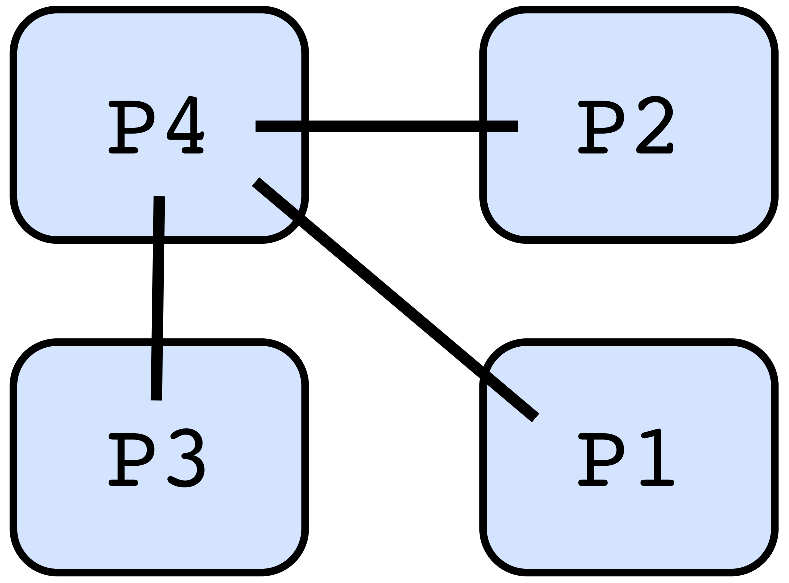
- Therefore, objectives of a consensus mechanism are:
  - **Agreement seeking**: A consensus mechanism should bring about as much agreement from the group as possible.
  - **Collaborative**: All the participants should aim to work together to achieve a result that puts the best interest of the group first.
  - **Cooperative**: All the participants shouldn’t put their own interests first and work as a team more than individuals.
  - **Egalitarian**: A group trying to achieve consensus should be as egalitarian as possible. What this basically means that each and every vote has equal weight. One person’s vote can’t be more important than another’s.
  - **Inclusive**: As many people as possible should be involved in the consensus process. It shouldn’t be like normal voting where people don’t really feel like voting because they believe that their vote won’t have any weight in the long run.
  - **Participatory**: The consensus mechanism should be such that everyone should actively participate in the the overall process.

Distributed Architectures

- Purely distributed / decentralized architectures are difficult to run correctly and efficiently (decentralized locking was pretty bad!)



Decentralized



Centralized

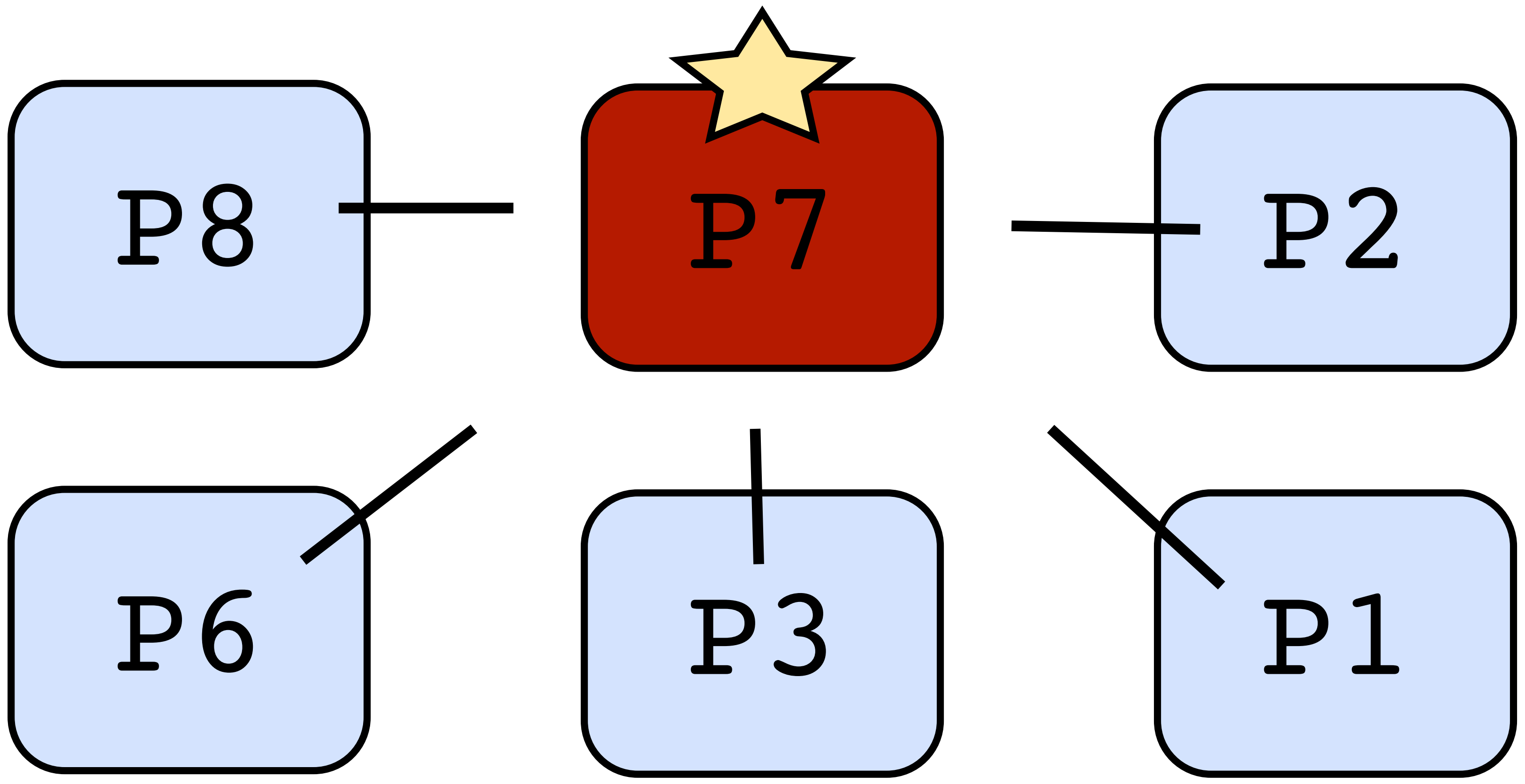
- Can we mix the two?

ELECTIONS

- Appoint a central coordinator
  - But allow them to be replaced in a safe, distributed way

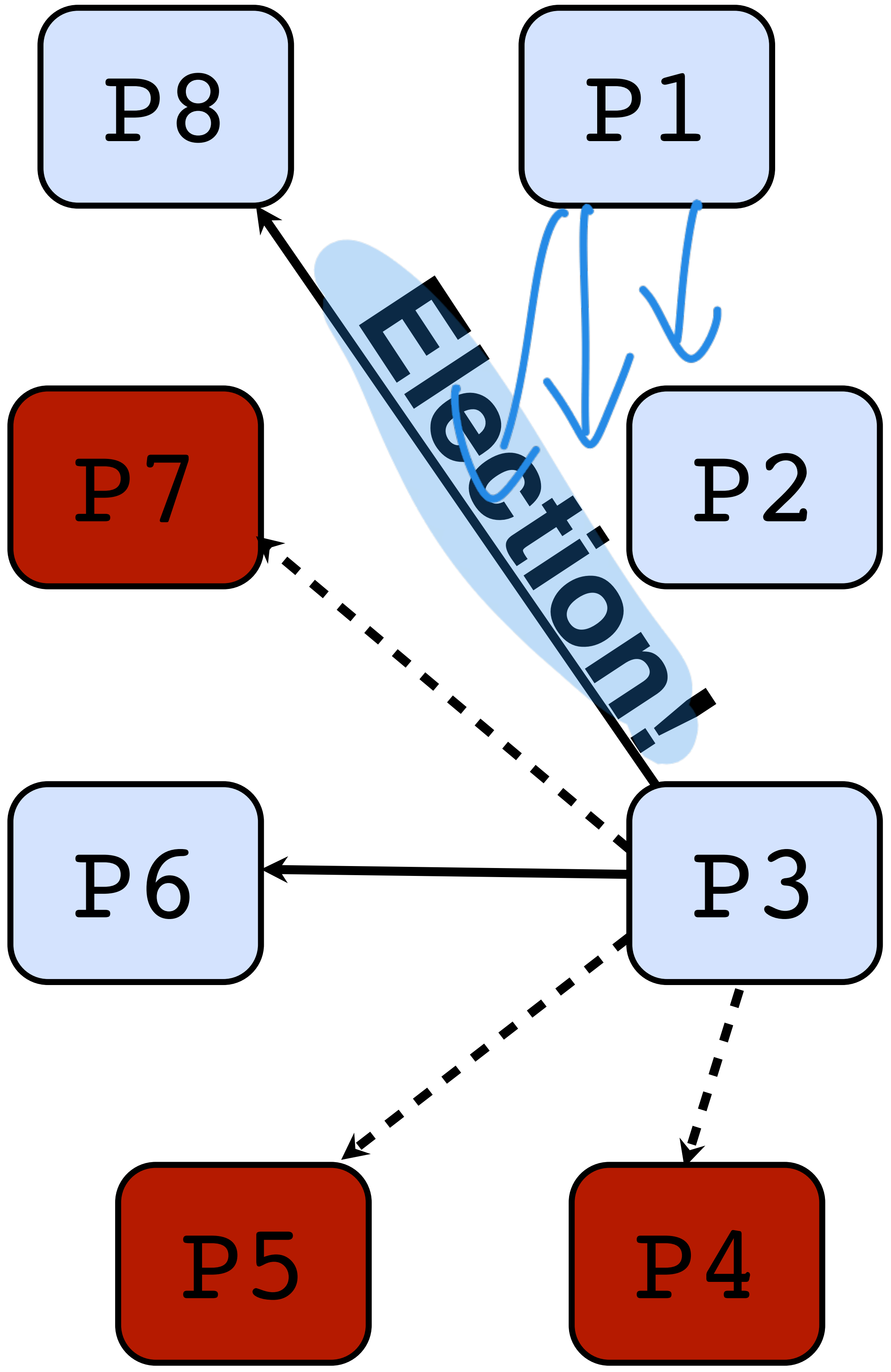
- Must be able to handle simultaneous elections
  - Reach a consistent result

- Who should win?



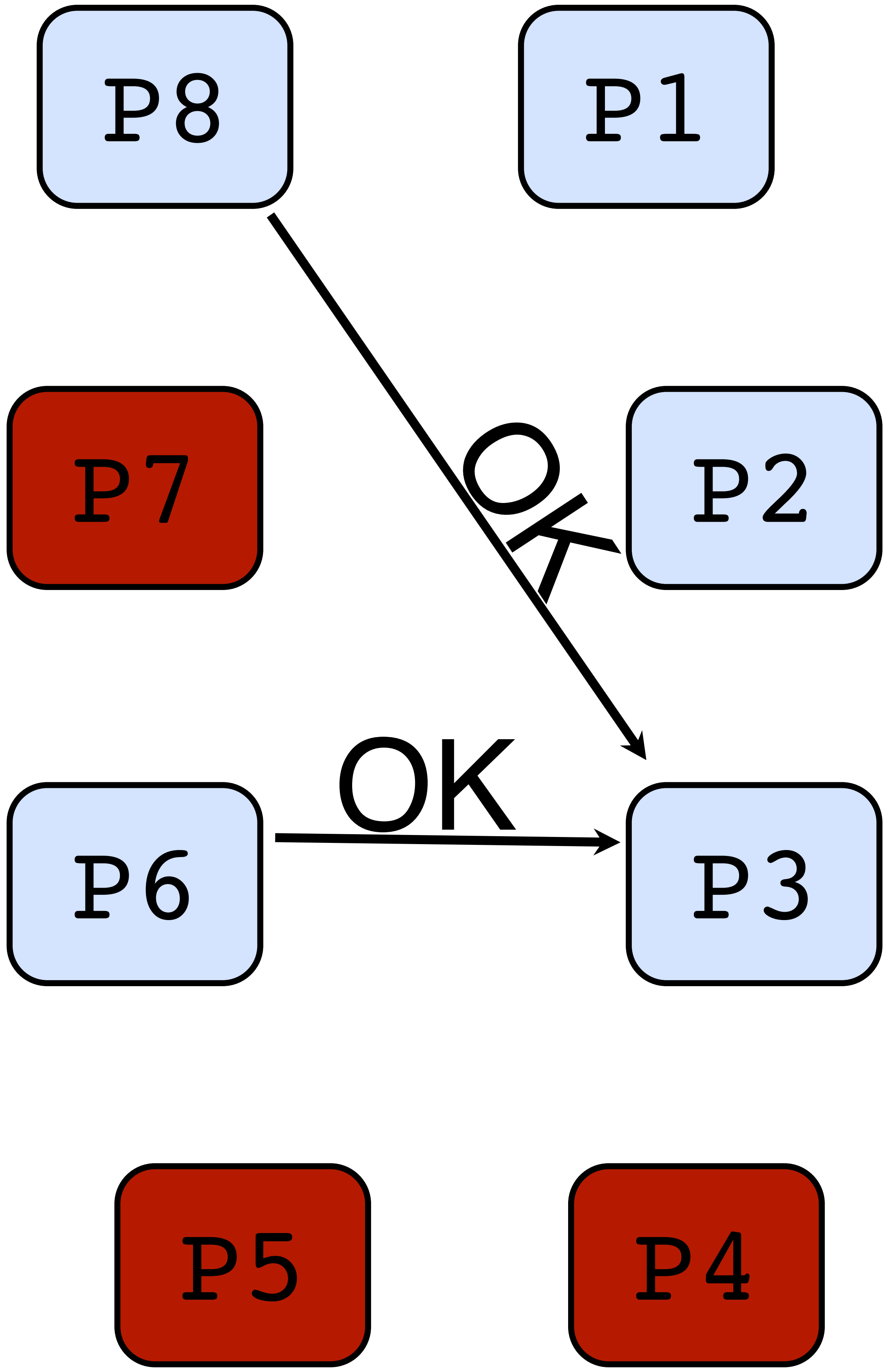
Bully Algorithm

- The biggest (ID) wins
- Any process P can initiate an election
- P sends **Election** messages to all process with higher Ids and awaits **OK** messages
- If it receives an OK, it drops out and waits for an **I won**
- If a process receives an **Election** msg, it returns an **OK**...



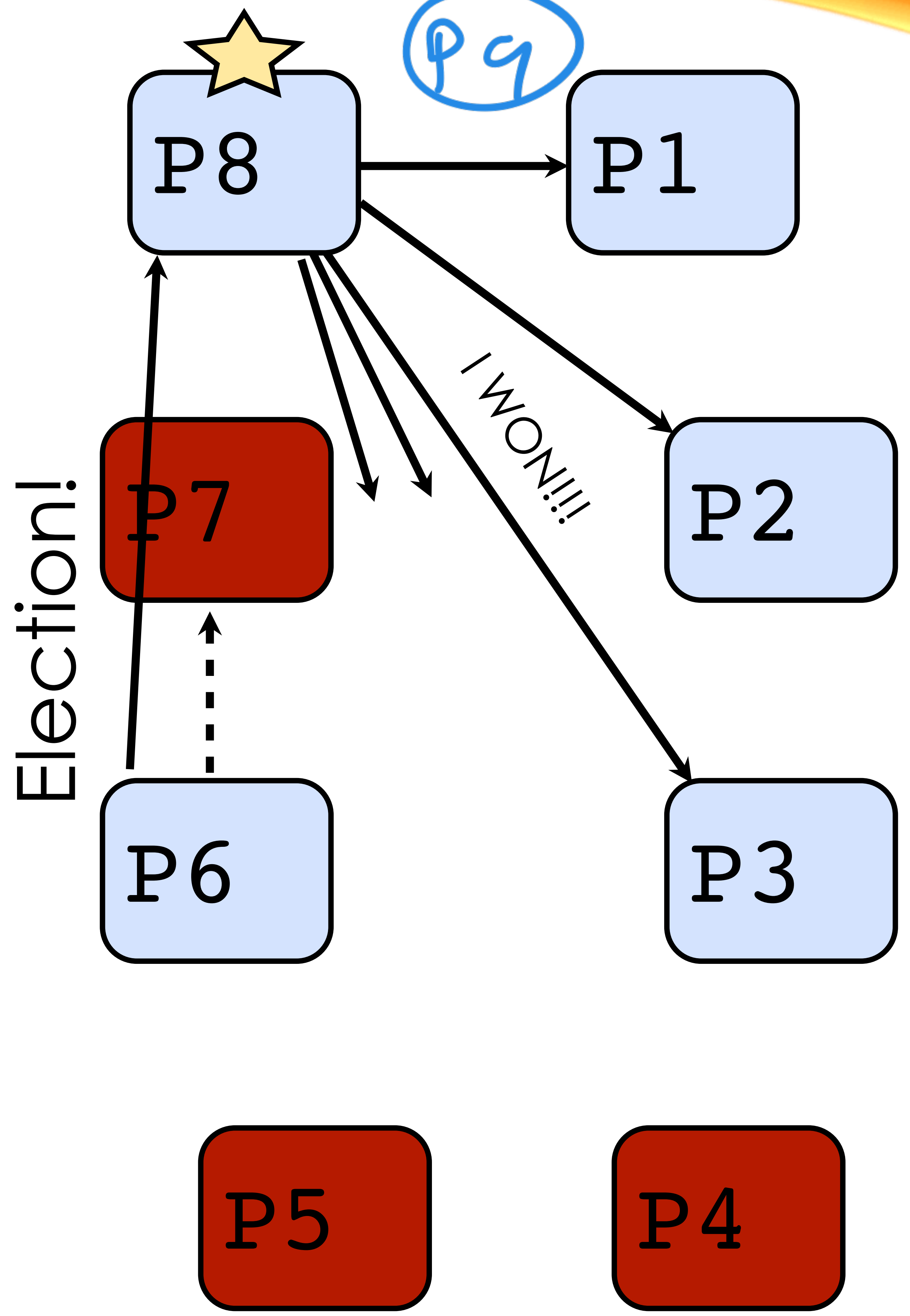
Bully Algorithm

- The biggest (ID) wins
- Any process P can initiate an election
- P sends **Election** messages to all process with higher Ids and awaits **OK** messages
- If it receives an OK, it drops out and waits for an **I won**
- If a process receives an **Election** msg, it returns an **OK**...



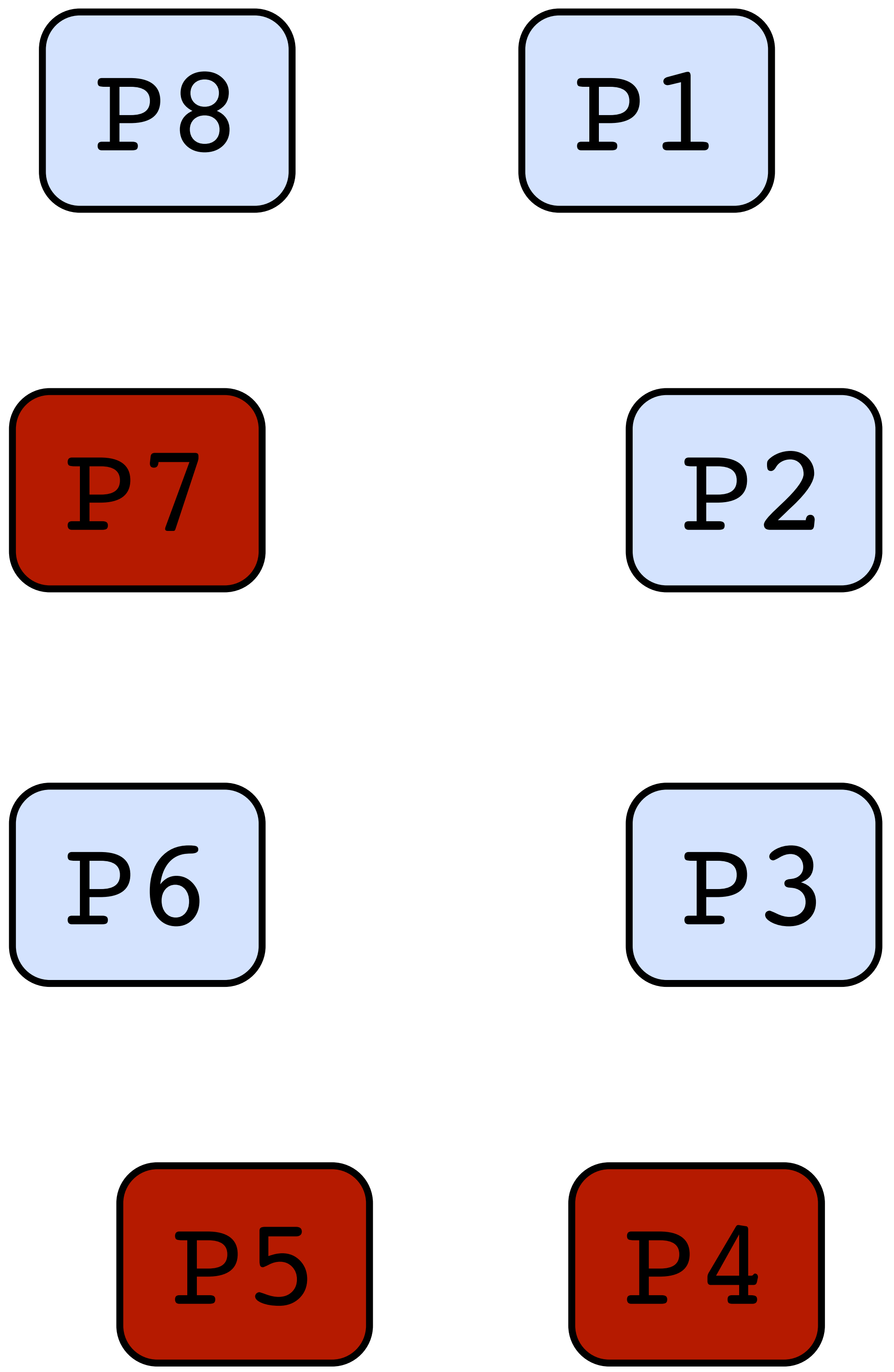
Bully Algorithm

- The biggest (ID) wins
- Any process P can initiate an election
- P sends **Election** messages to all process with higher Ids and awaits **OK** messages
- If it receives an OK, it drops out and waits for an **I won**
- If a process receives an **Election** msg, it returns an **OK** and starts **another** election
- If no **OK** messages, P becomes leader and sends **I won** to all process with lower Ids
- If a process receives a **I won**, it treats sender as the leader



Ring Algorithm

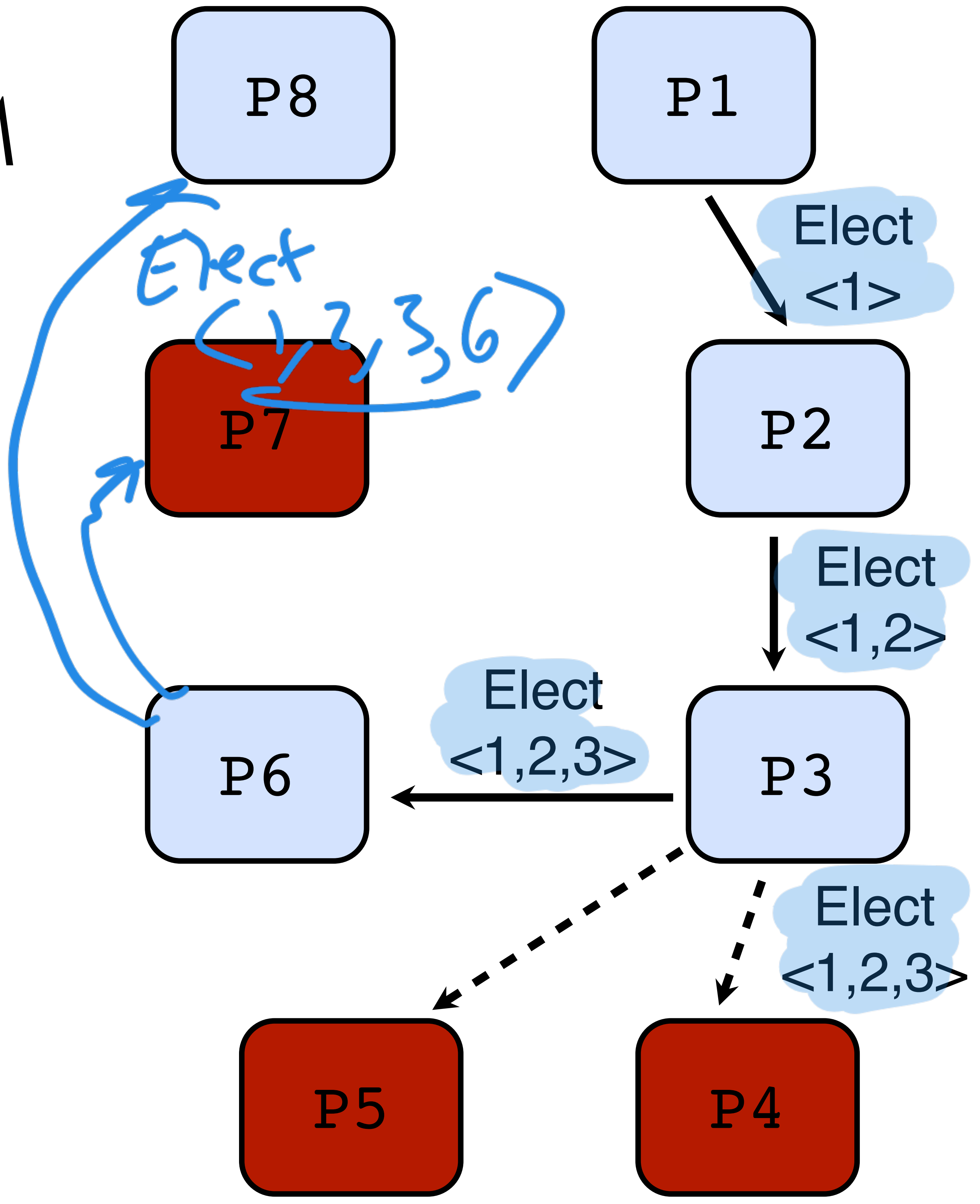
- Any other ideas?



Ring Algorithm

- **Initiator** sends an **Election** message around the ring
- Add your ID to the message
- When Initiator receives message again, it announces the winner

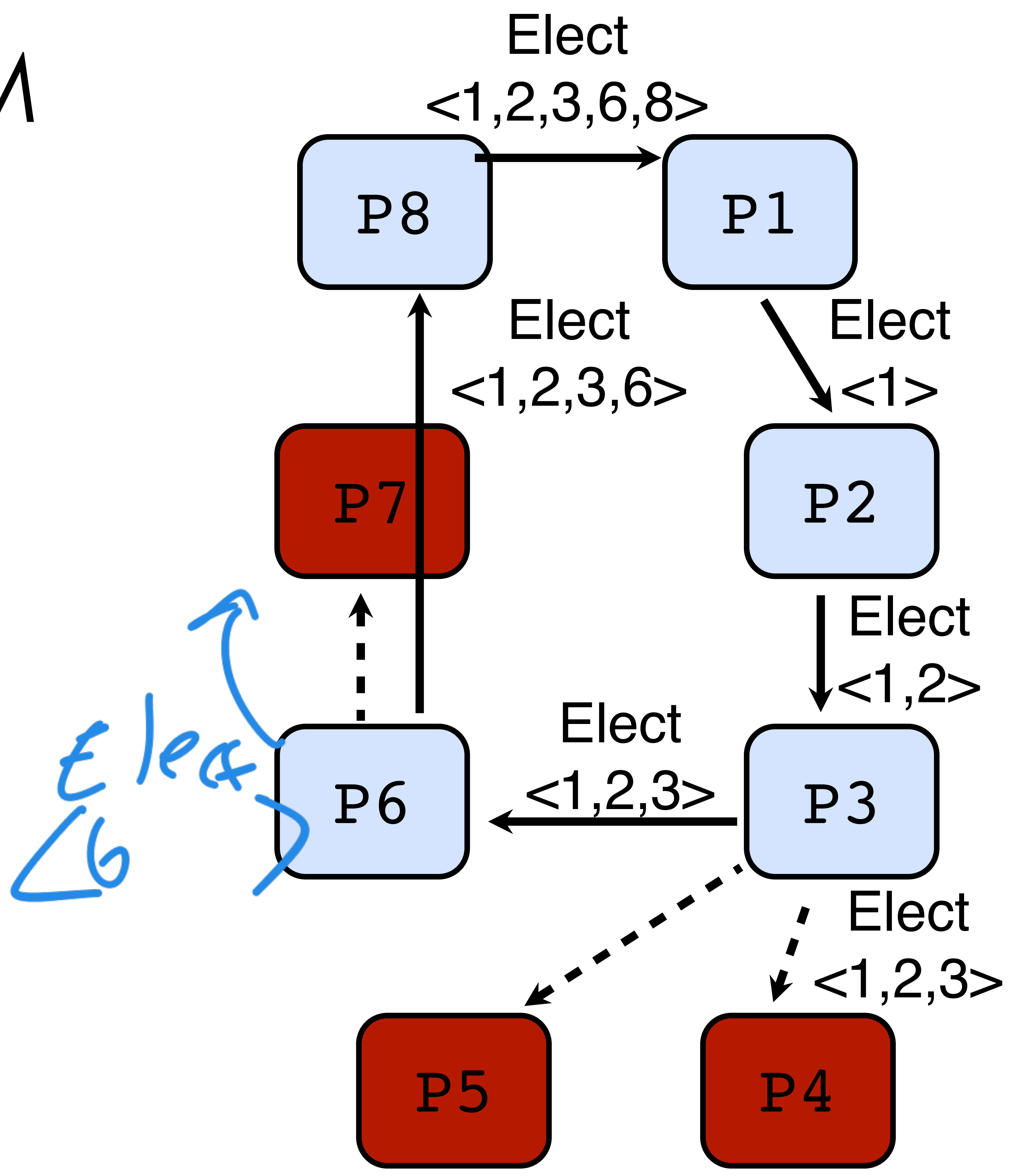
- What happens if multiple elections occur at the same time?



Ring Algorithm

- **Initiator** sends an **Election** message around the ring
- Add your ID to the message
- When Initiator receives message again, it announces the winner

- What happens if multiple elections occur at the same time?



Comparison

- Number of messages sent to elect a leader:

- Bully Algorithm
  - Worst case: lowest ID node initiates election
    - Triggers n-1 elections at every other node = O(n^2) messages
  - Best case: Immediate election after n-2 messages

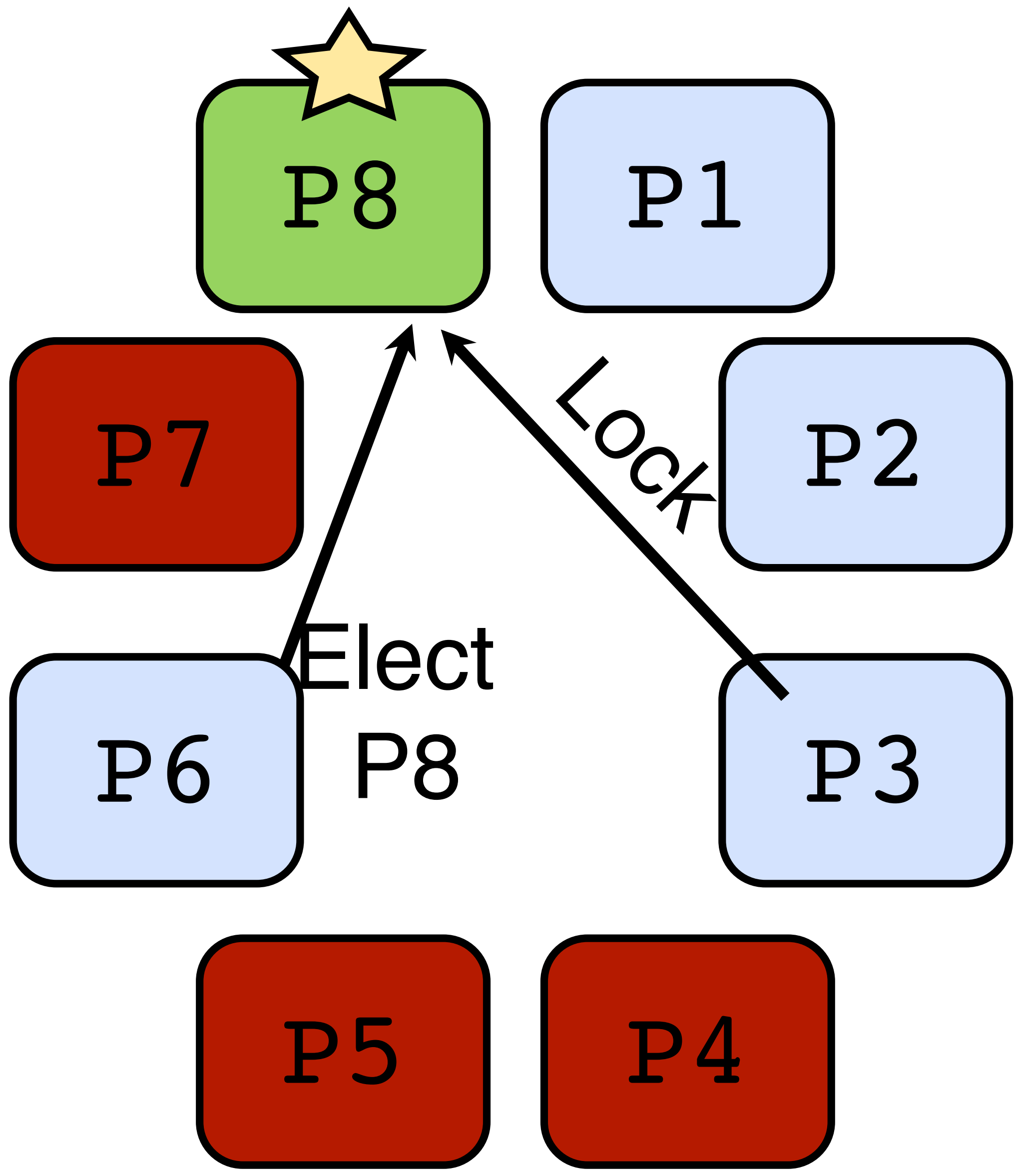
- Ring Algorithm
  - Always 2(n-1) messages
  - Around the ring, then notify all

Elections + Centralized Locking

- Elect a leader
- Let them make all the decisions about locks

- What kinds of failures can we handle?
  - Leader/non-leader?
  - Locked/unlocked?
  - During election?

This can be the basis for consensus-based distributed systems!

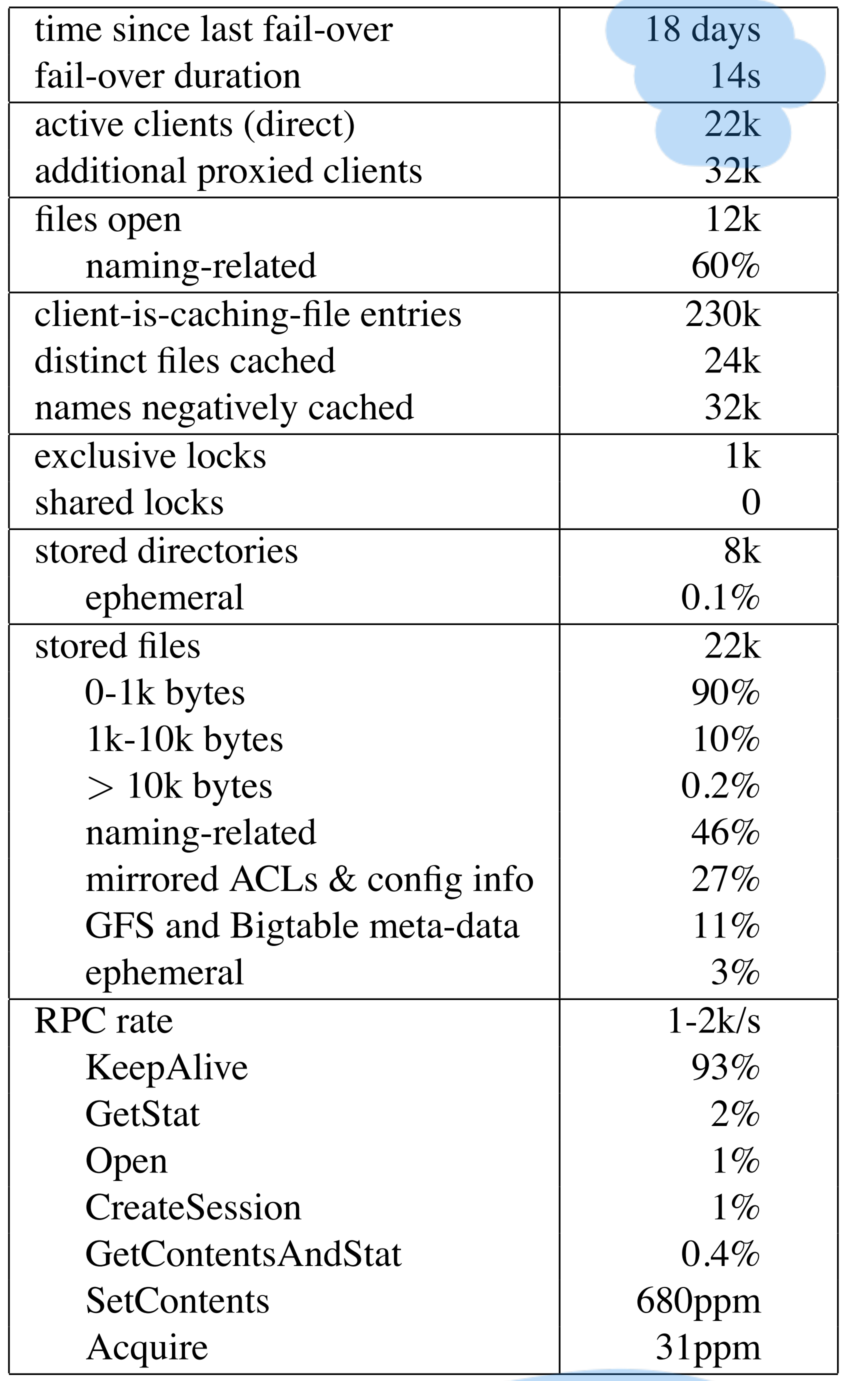


Chubby: Google's Lock Service

- Google services are composed of many thousands of nodes
- Need a way to coordinate data and access to shared resources!
  - Used by Google File System, BigTable, etc

- Chubby: lock service for loosely coupled distributed systems
  - Focuses on availability and reliability (not performance)
  - Scales to ~10,000 servers per Chubby Cell

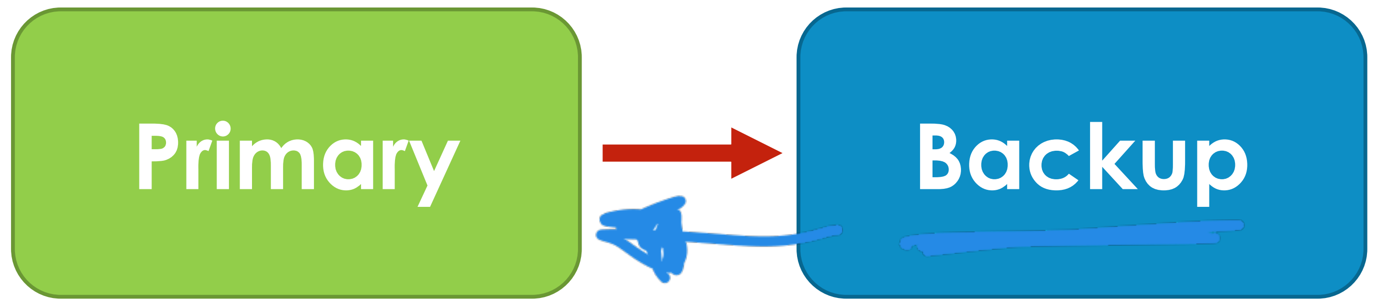
- See paper at OSDI 2006 by Mike Burrows for full details!



time since last fail-over	18 days
fail-over duration	14s
active clients (direct)	22k
additional proxied clients	32k
files open	12k
naming-related	60%
client-is-caching-file entries	230k
distinct files cached	24k
names negatively cached	32k
exclusive locks	1k
shared locks	0
stored directories	8k
ephemeral	0.1%
stored files	22k
0-1k bytes	90%
1k-10k bytes	10%
> 10k bytes	0.2%
naming-related	46%
mirrored ACLs & config info	27%
GFS and Bigtable meta-data	11%
ephemeral	3%
RPC rate	1-2k/s
KeepAlive	93%
GetStat	2%
Open	1%
CreateSession	1%
GetContentsAndStat	0.4%
SetContents	680ppm
Acquire	31ppm

State Machine Replication (SMR)

- We can think of an application as a state machine
  - A program is just **data** that is updated based on **operations** -> **state**
- Consensus means that all distributed nodes should be in the same state!
  - If a node fails, it should not disrupt the system
  - When a node recovers it should be able to "catch up"

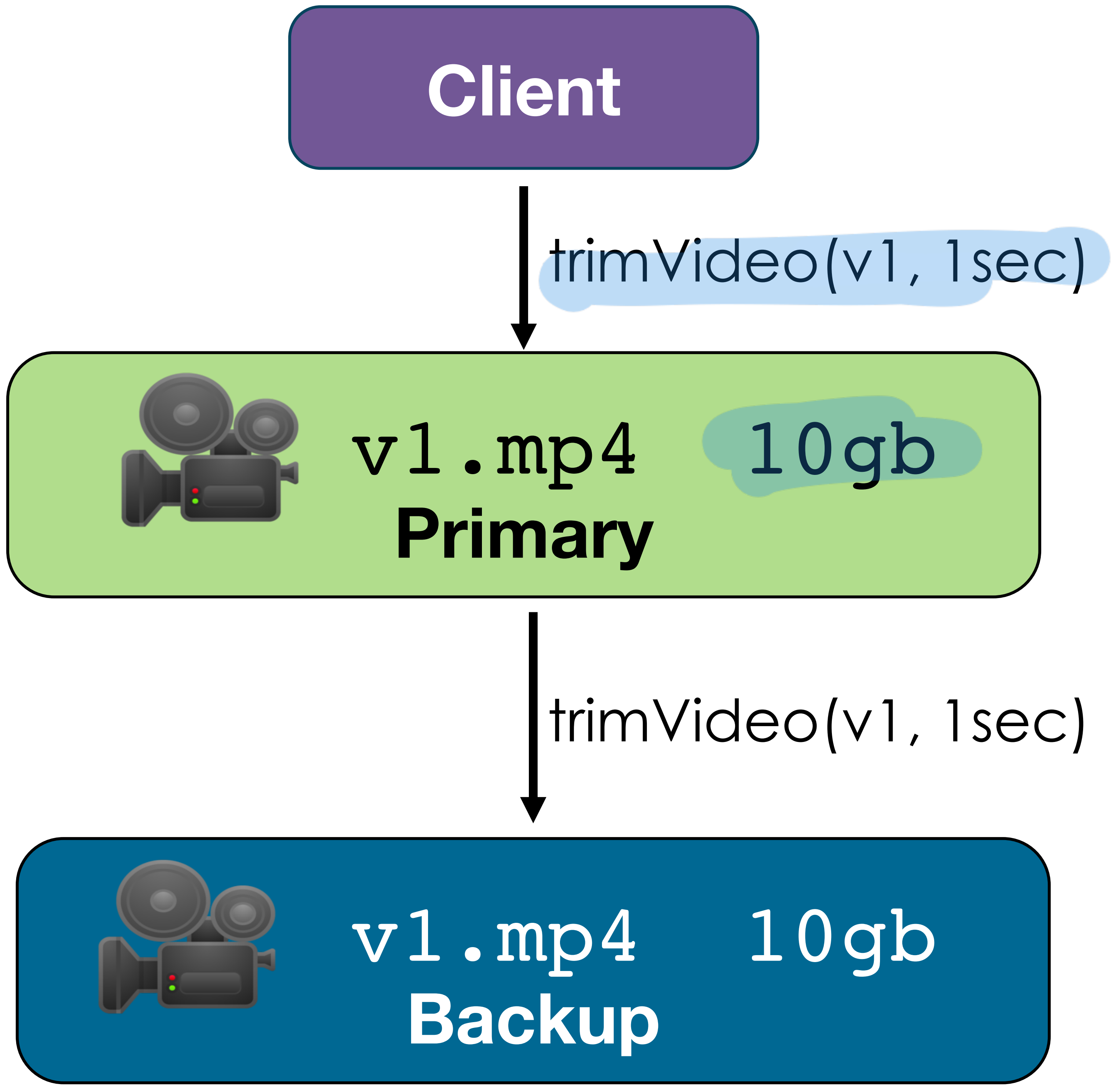


Distributed Video Editing SMR

- Sometimes **data** is big!
- Replicate the **operation** to be performed, not the data!

- Treat like a state machine
  - Incoming requests just perform some operation on that data
  - If all replicas perform same operations, they will end in the same state

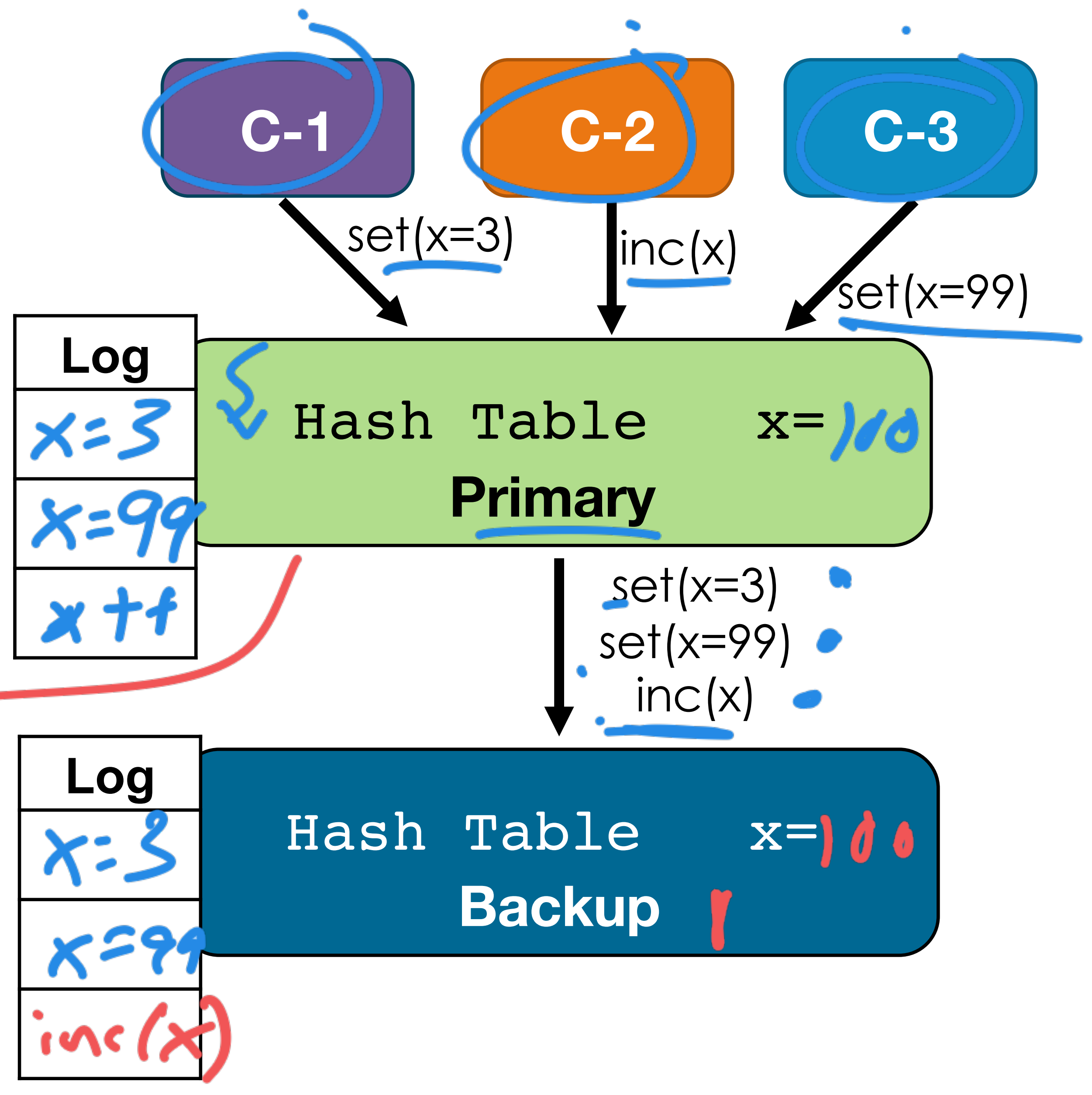
- If **Primary** fails, switch to **Backup**

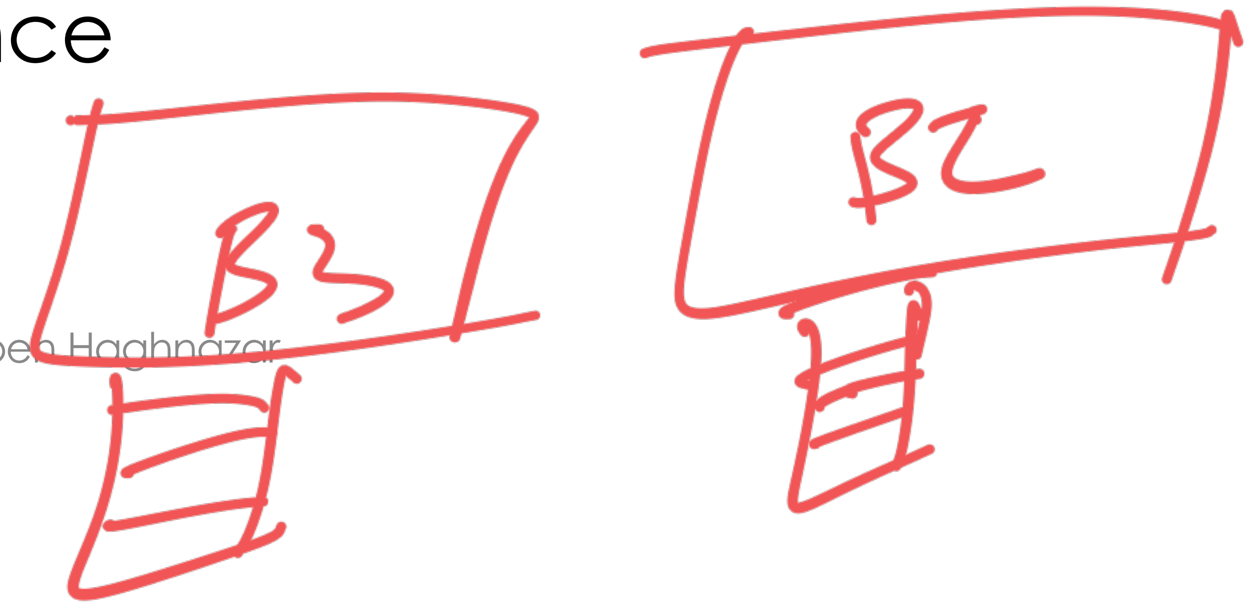


x = ?

Hash Table SMR

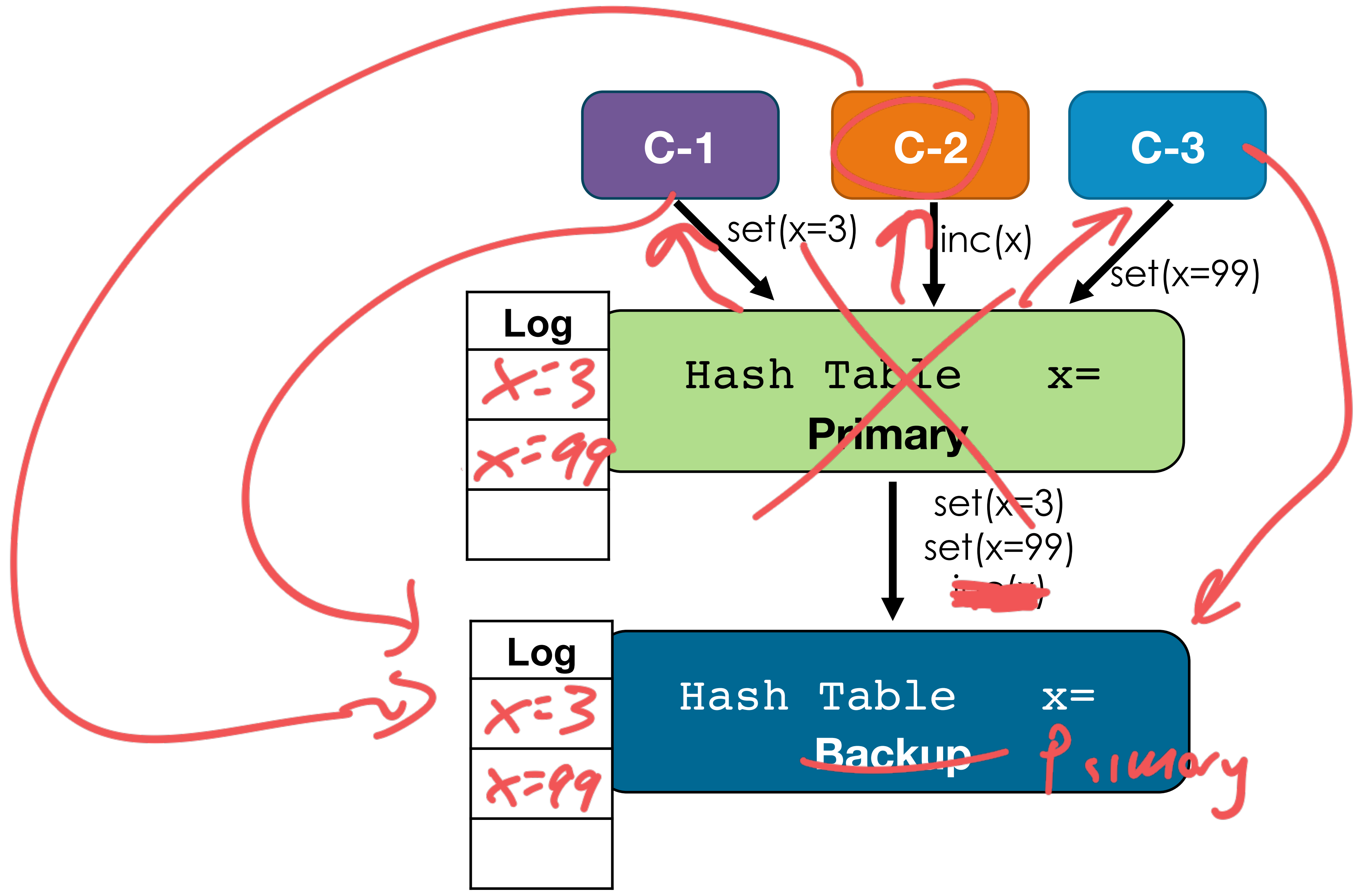
- SMR creates a **replicated log** of actions to be performed
  - E.g., updates to the value stored by a key
- Primary orders incoming requests to form the log
- Actions must be deterministic
- We can keep adding more backup replicas to improve fault tolerance





SMR Failures?

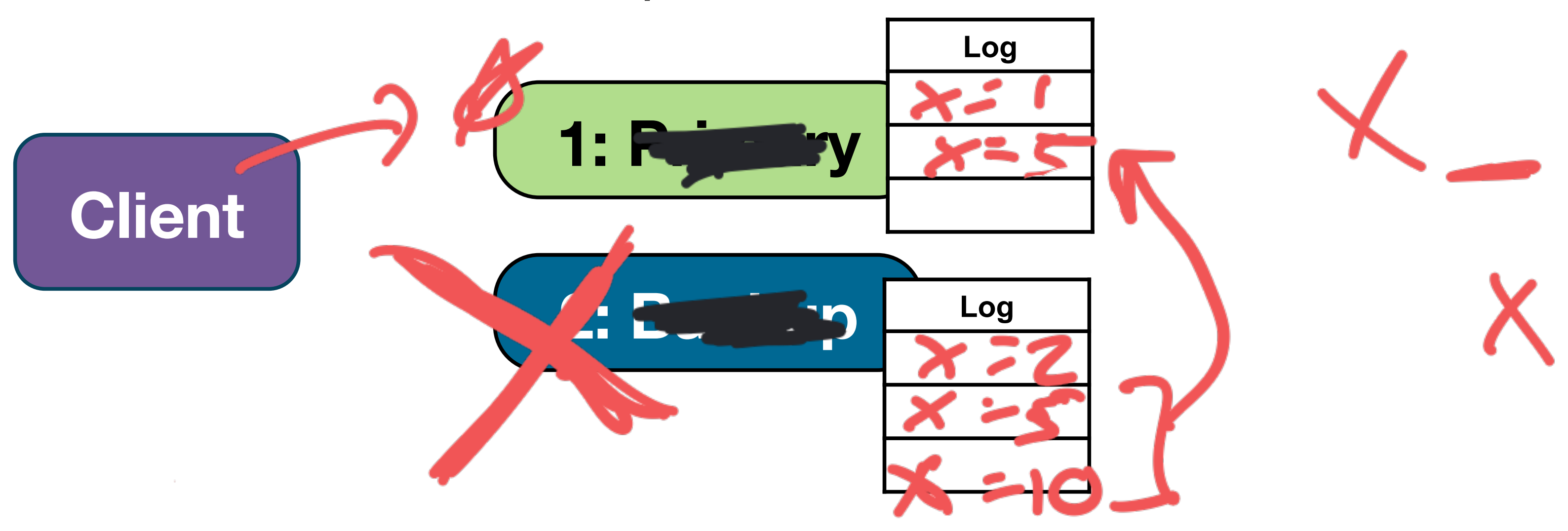
- What to do on a failure?
- How many failures can we handle?



Handling Failures

- F = number of nodes which can crash at one time
- # of nodes needed must depend on f!

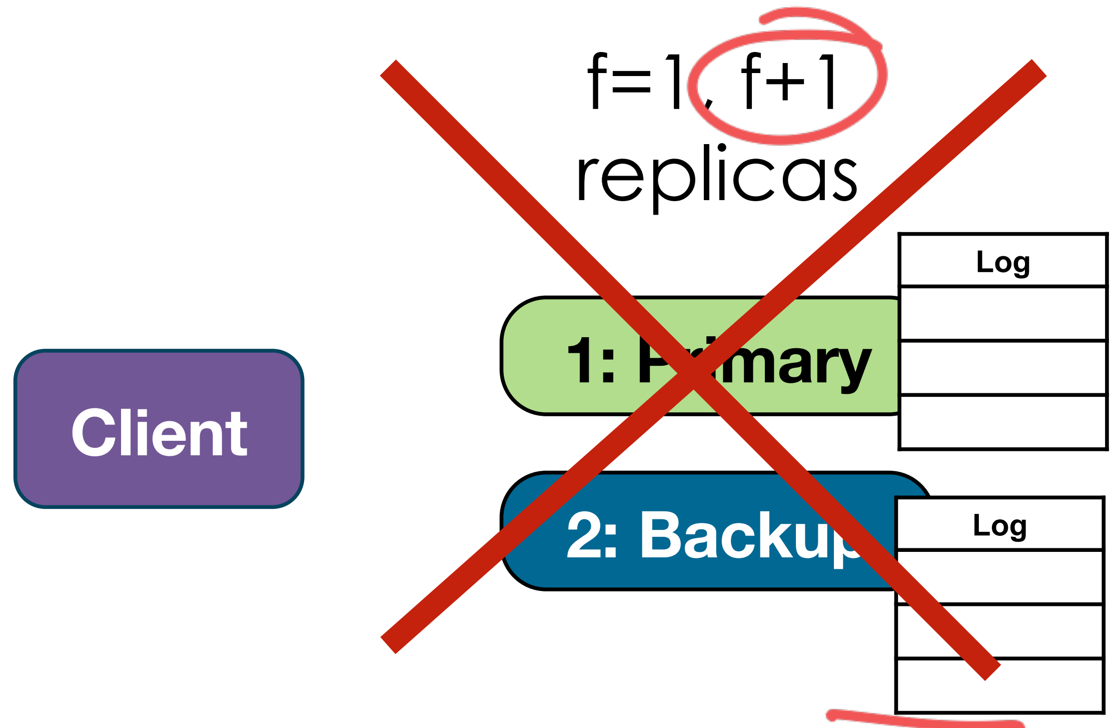
f=1, f+1=2
replicas



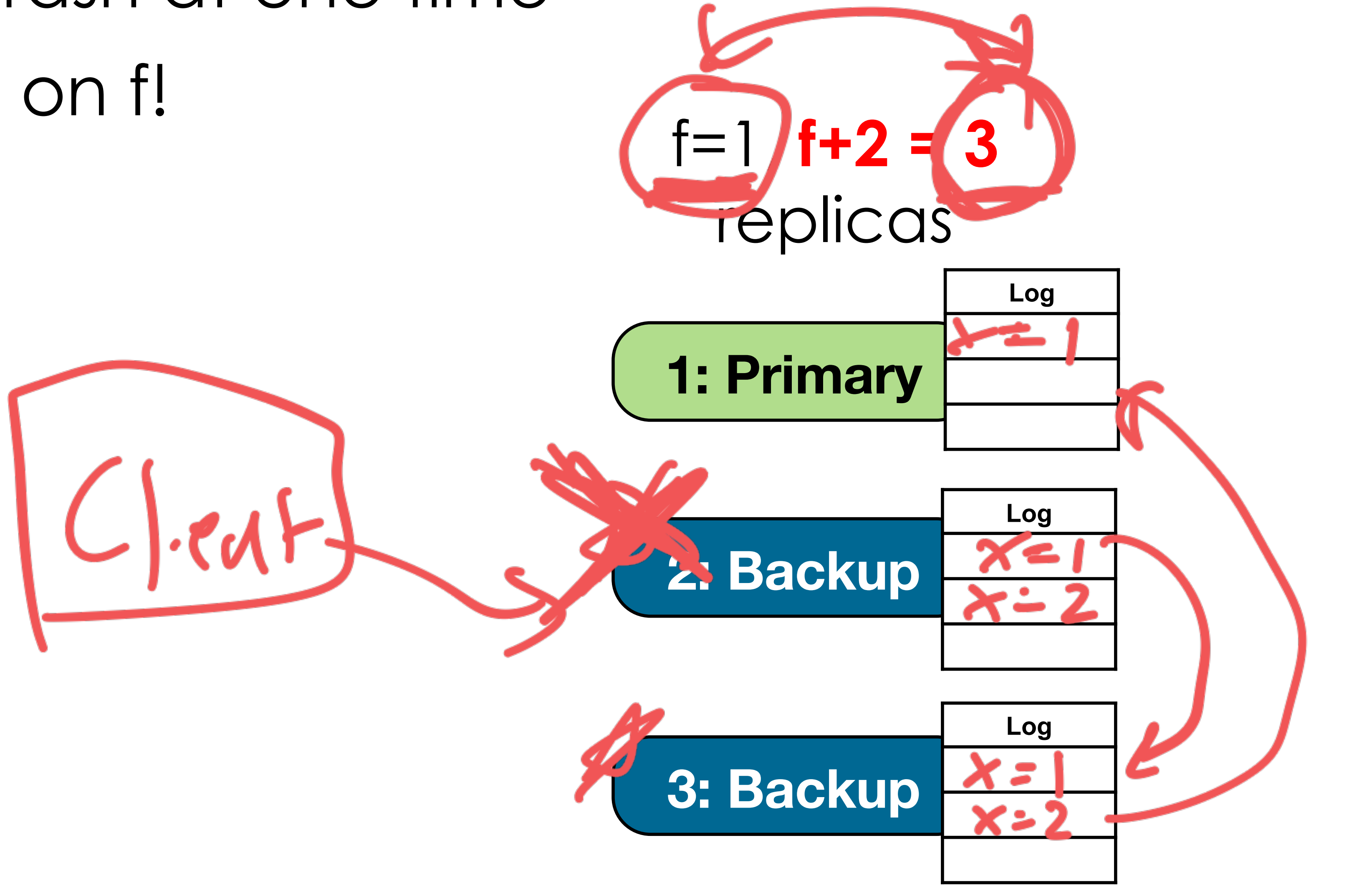
What failure scenarios can happen?

HANDLING FAILURES

- F = number of nodes which can crash at one time
- # of nodes needed must depend on f!



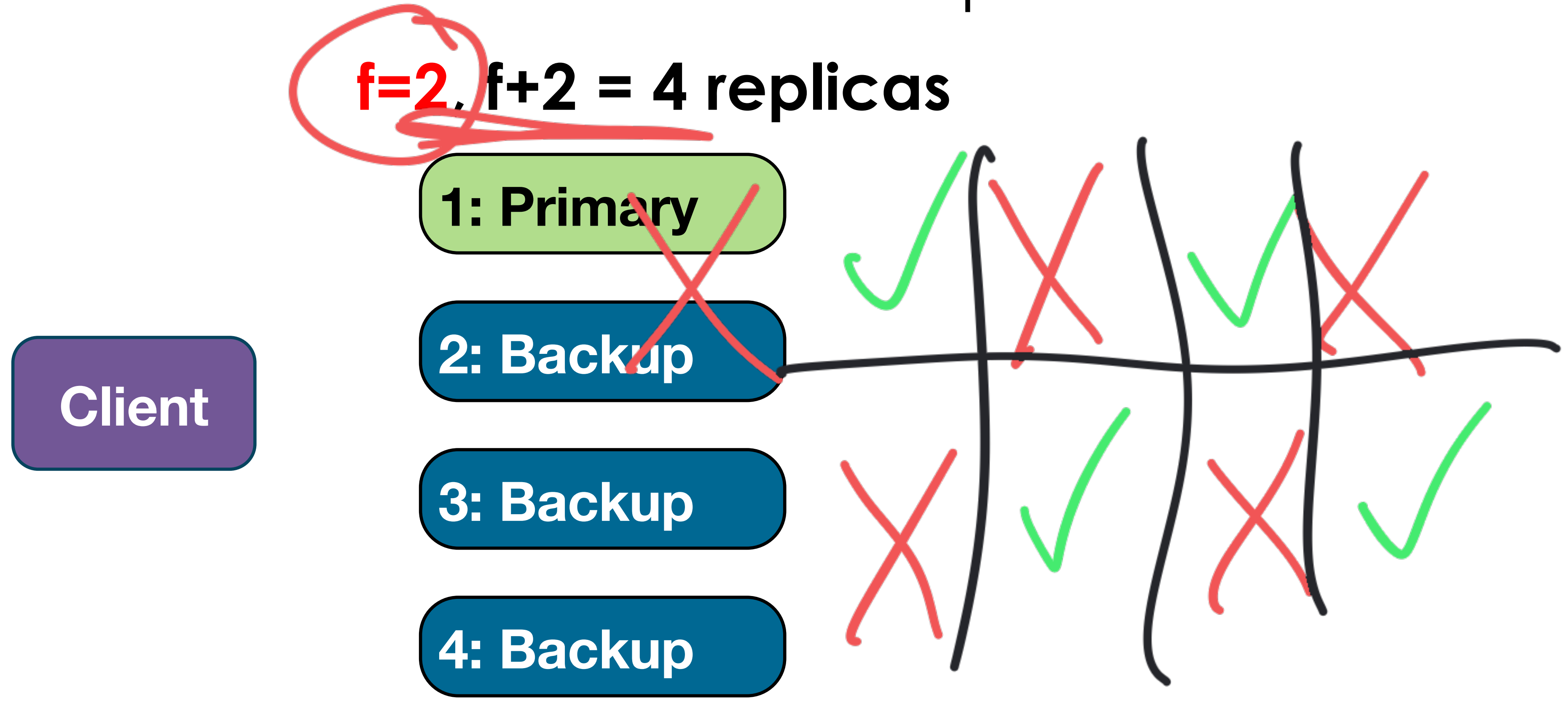
Can't resync state if failure "flip flops" between nodes!

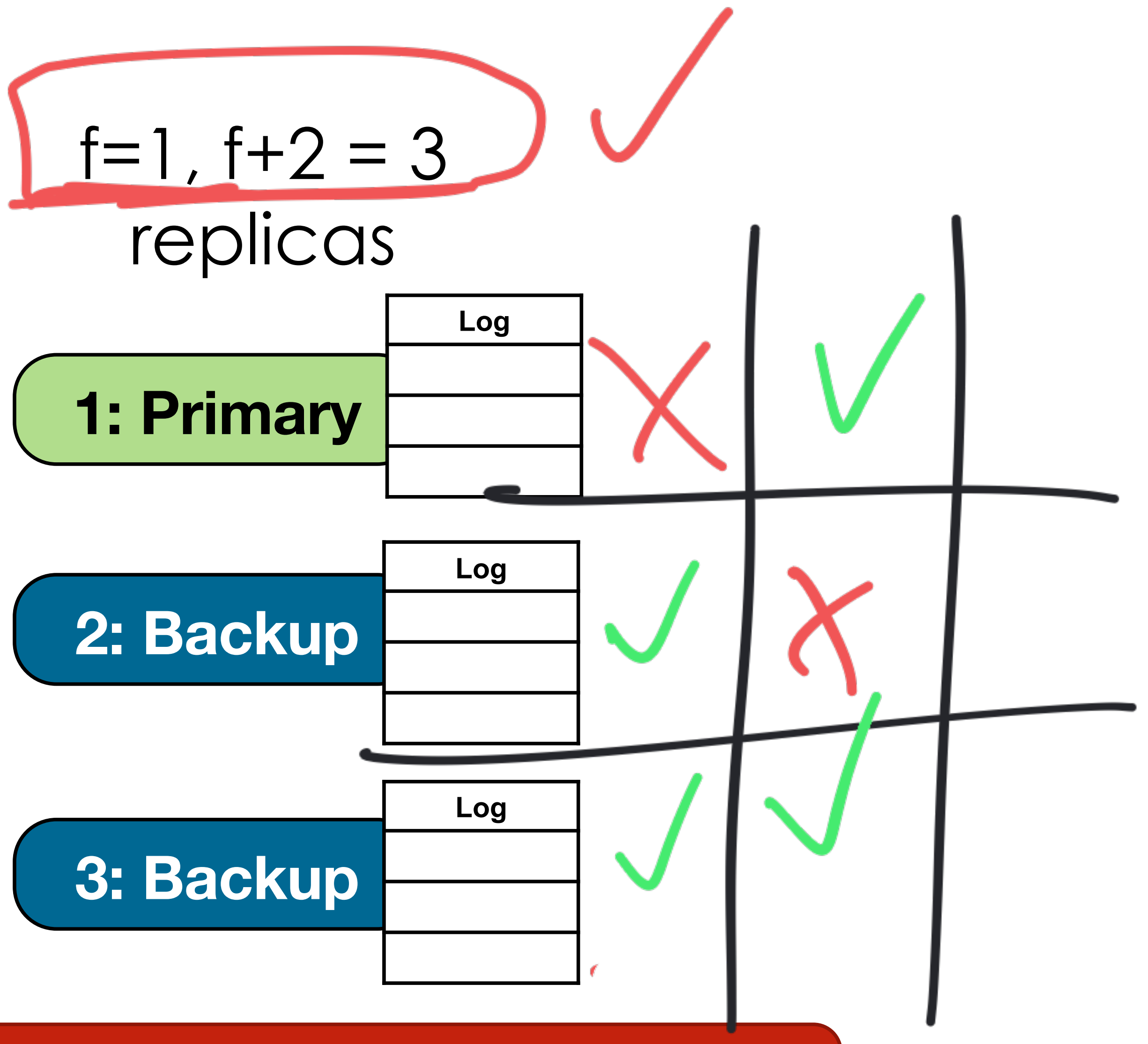


Fixed?

Handling Failures

- F = number of nodes which can crash at one time
- # of nodes needed must depend on f!

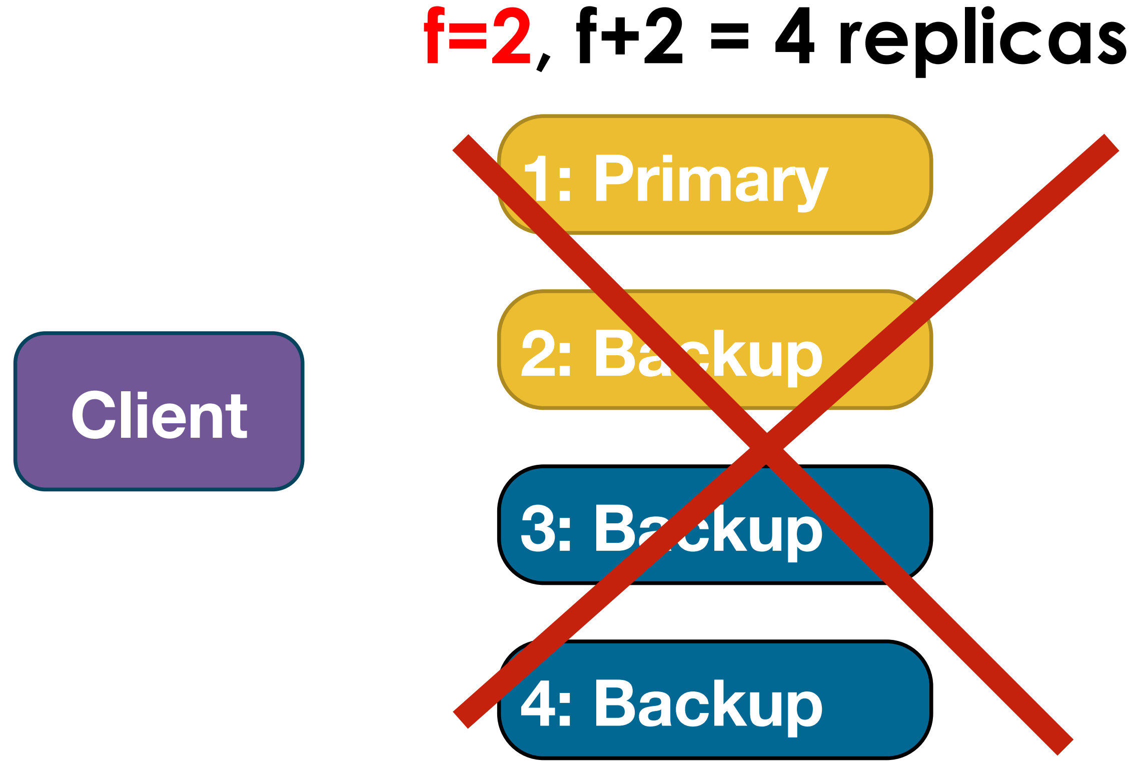




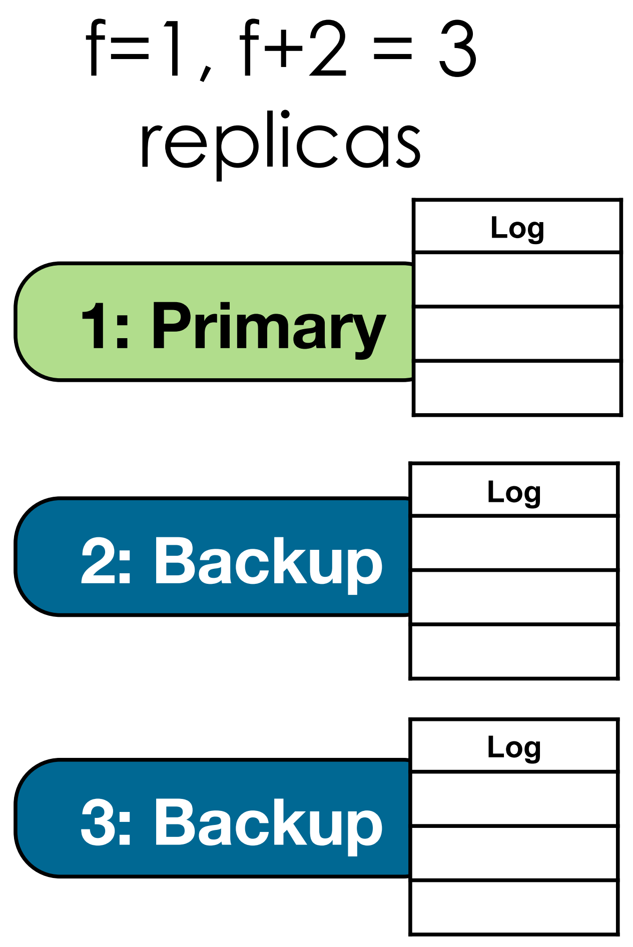
Fixed for f=2?

Handling Failures

- F = number of nodes which can crash at one time
- # of nodes needed must depend on f!



Can't resync state if failure "flip flops" between **2** nodes!

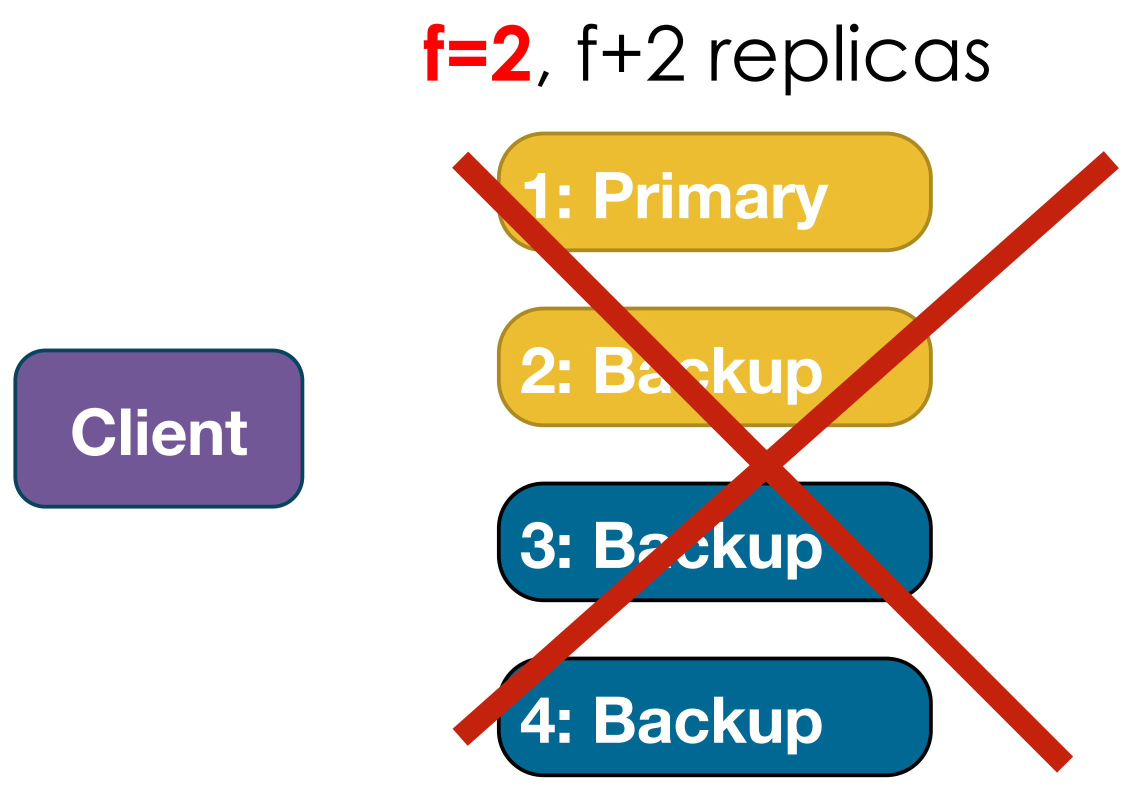


Fixed for f=2? No!

f=1 → 3

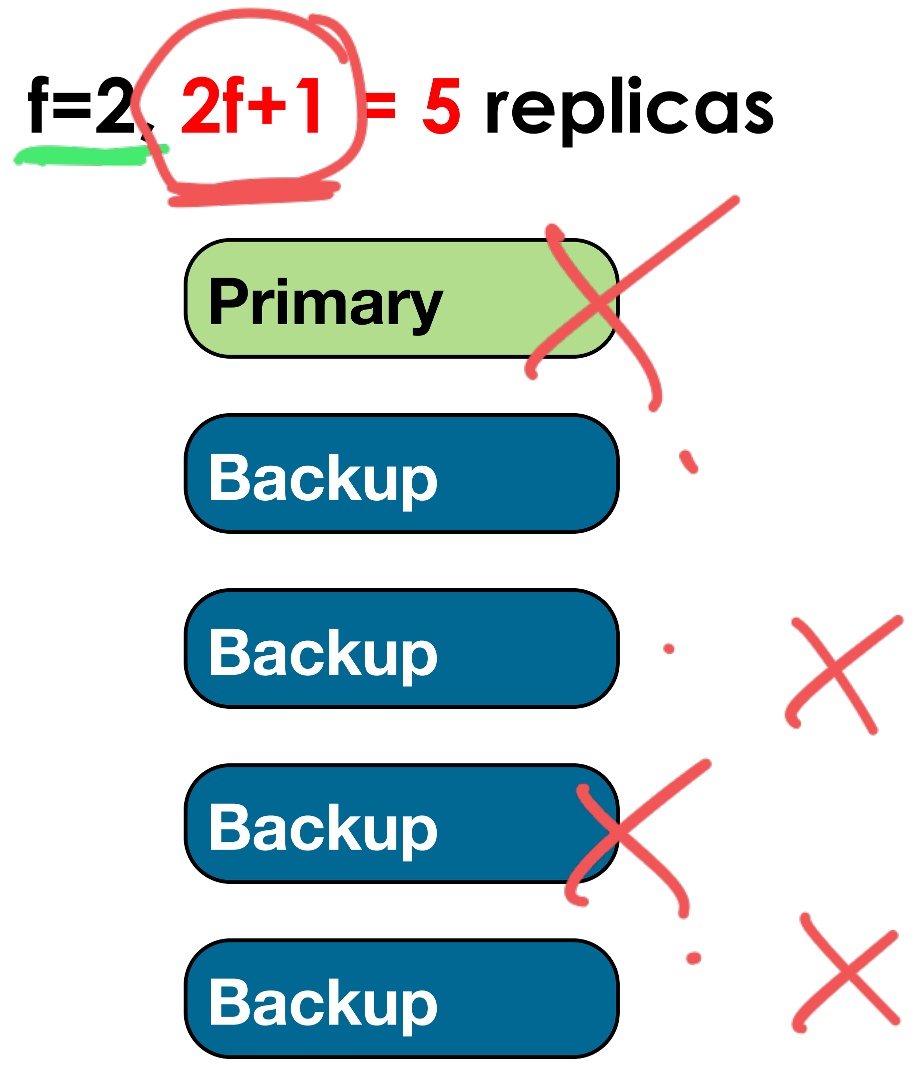
Handling Failures

- F = number of nodes which can crash at one time
- # of nodes needed must depend on f!



Can't resync state if failure "flip flops" between **2** nodes!

Use **2f+1** replicas!
Insight: Always need a **majority** of nodes to stay alive!



State Machine Replication Overview

- Provides a generic **fault tolerance** mechanism
  - Application just needs to have well defined operations and a way to avoid non-determinism
- Primary orders requests into log
- Backups execute log in order
- Log allows out of date replicas to recover
- Need **2f+1** replicas to tolerate **f** failures

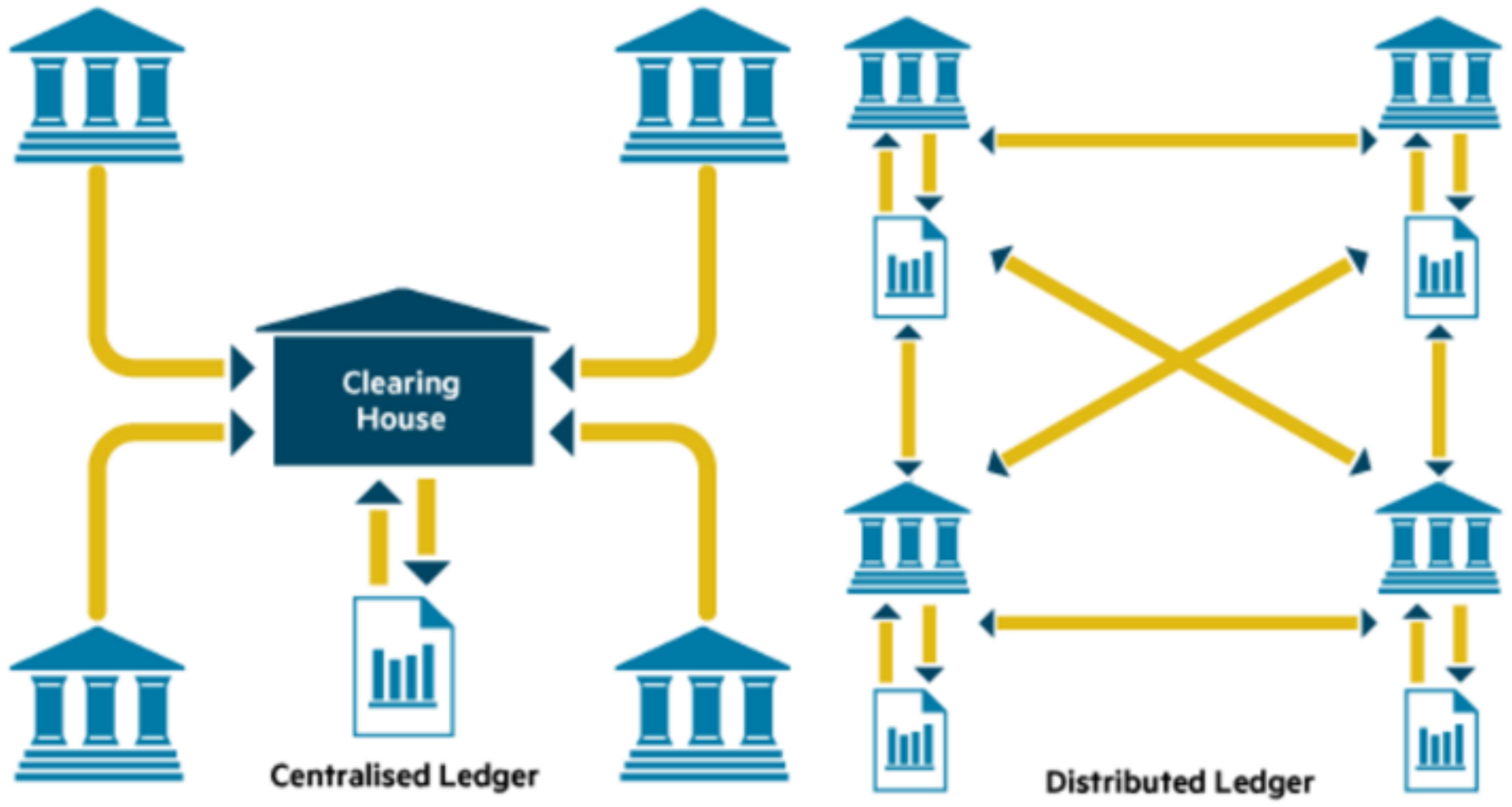
- But how do we pick who should be primary…?
  - Use an election algorithm!

Optional HW 3: Implement the ~~Election~~ algorithm used by the Raft SMR protocol

Case Study

- Two important challenges in BlockChain
  - How are any decisions made?
  - How does anything get done?

Distributed Ledger Tech

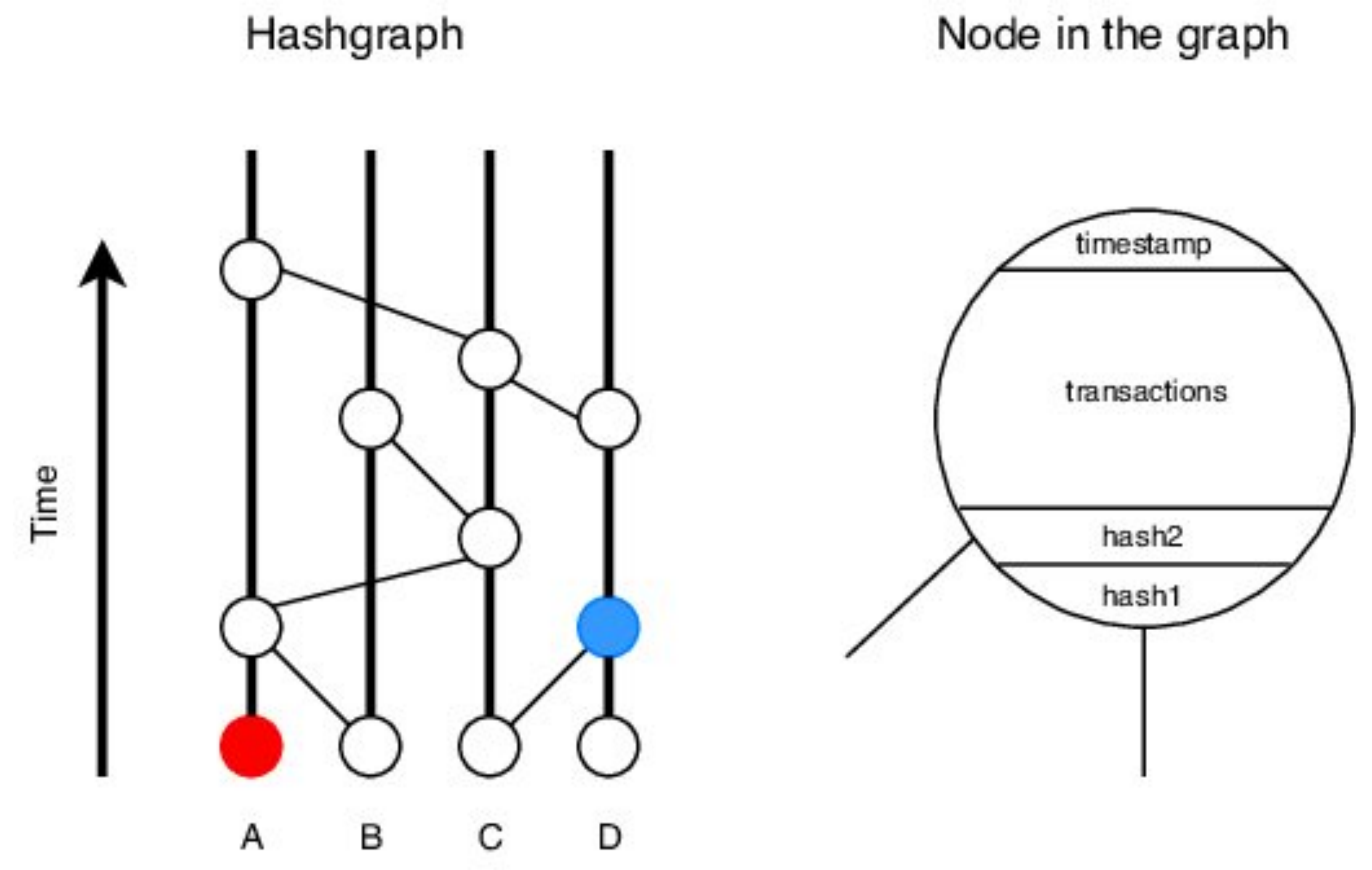


Different types of DLT

- Blockchain
- Hashgraph
- DAG
- Holochain
- Tangle
- Radix (Tempo)

HashGraph

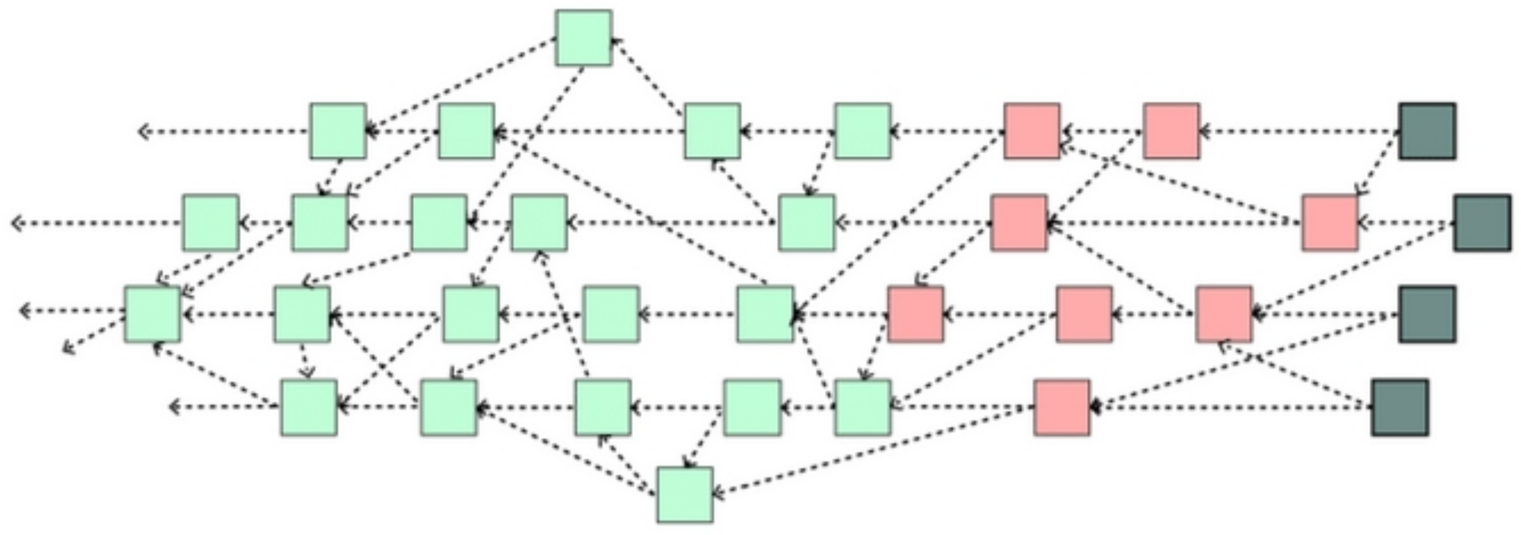
- It's so fast – 250000 transaction per second (Scalability characteristics in Distributed Systems)
- Being Time-Based and using Gossip protocol for consensus reduces the process and math complexity.
- In the level of security it is evaluating in the banking system level and it means it is a ***Byzantine Fault Tolerance*** system.
- Controlled Network (Consensus is easier)



Hashgraph Data Structure

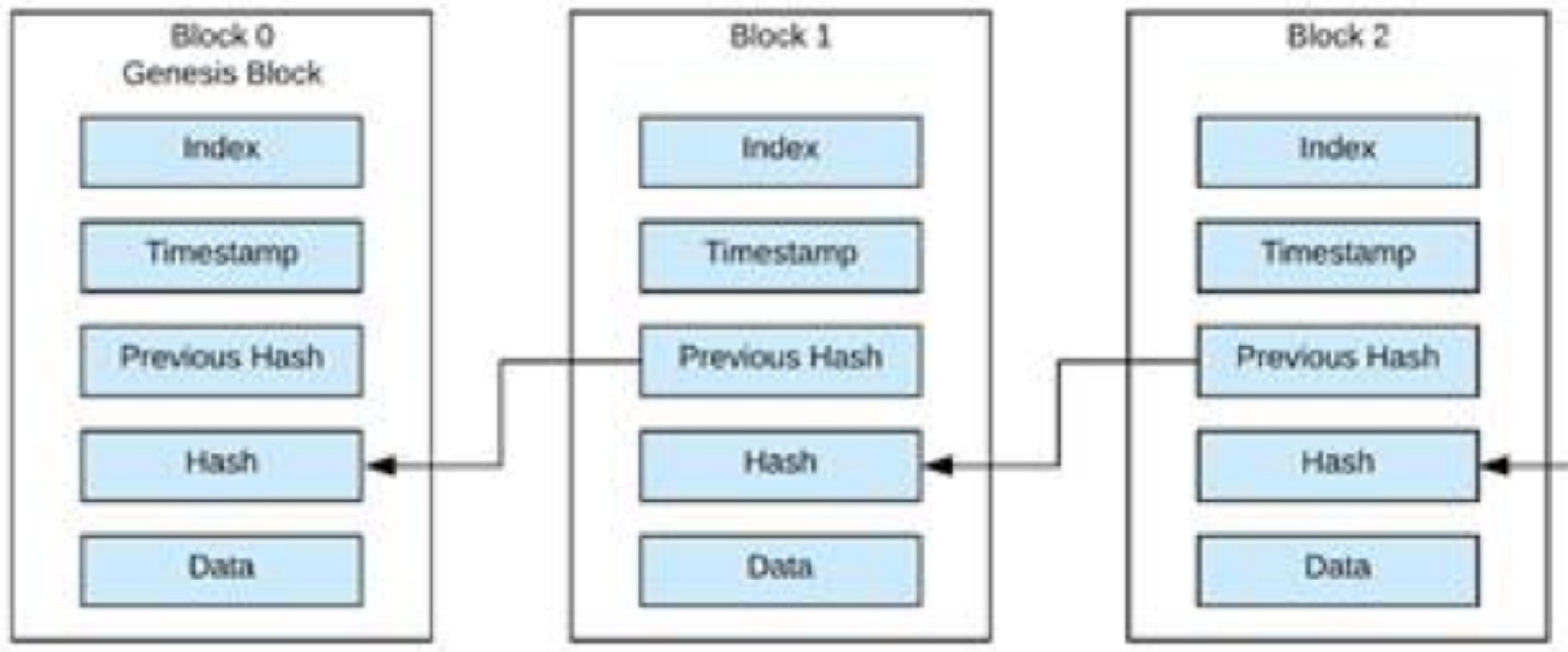
Tangle (IOTA)

- IOTA is an open-source distributed ledger and cryptocurrency designed for the Internet of things.
- Uses DAG to store transactions on its ledger, motivated by a potentially higher scalability over blockchain based distributed ledgers for nano-Transactions between IOT devices.
- There are categories of participants,
  - Transaction creators
  - Transaction verifiers

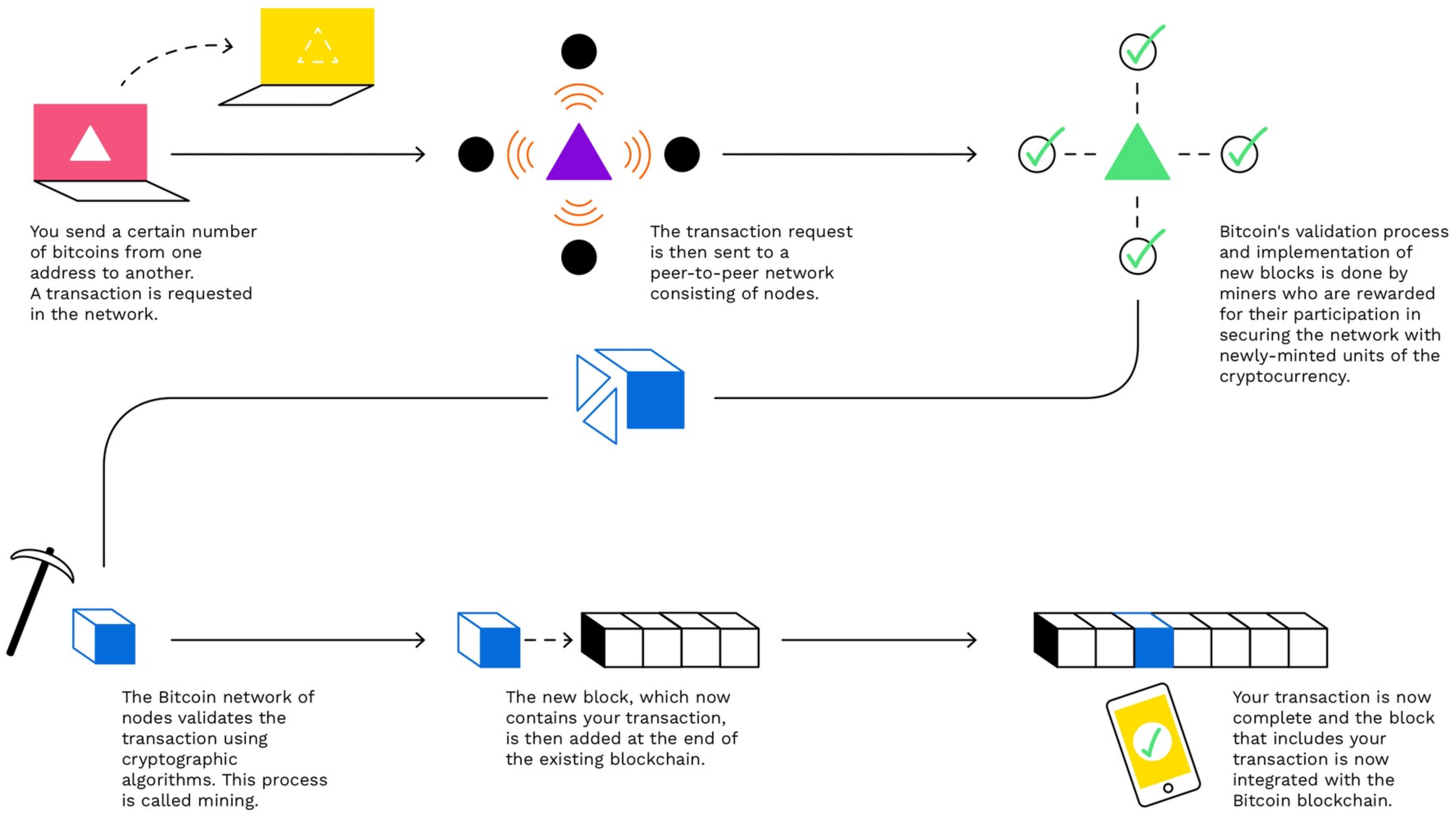


BlockChain

- Unofficial definition: A blockchain is an unchangeable and sequence of records and transactions which is called ***BLOCK***
- The blocks connects to each other with Hash Codes
- Each block contains an index, time stamp, list of transactions, evidence, and **last block hash** (which guarantees the unchangeability of the chain)



How does it work? Ex. Bitcoin



You send a certain number of bitcoins from one address to another. A transaction is requested in the network.

The transaction request is then sent to a peer-to-peer network consisting of nodes.

Bitcoin's validation process and implementation of new blocks is done by miners who are rewarded for their participation in securing the network with newly-minted units of the cryptocurrency.

The Bitcoin network of nodes validates the transaction using cryptographic algorithms. This process is called mining.

The new block, which now contains your transaction, is then added at the end of the existing blockchain.

Your transaction is now complete and the block that includes your transaction is now integrated with the Bitcoin blockchain.

Consensus in BlockChain

- A consensus mechanism enables the blockchain network to attain reliability and build a level of trust between different nodes, while ensuring security in the environment.
  - Proof of Work (PoW)
  - Proof of Stake (PoS)
  - Delegated Proof of Stake (DPoS)
  - Leased Proof of Stake (LPoS)
  - Direct Acyclic Graph (DAG)
  - Byzantine Fault Tolerance (BFT)
  - Practical Byzantine Fault Tolerance (PBFT)
  - Delegated Byzantine Fault Tolerance (DBFT)
  - Proof of Capacity (PoC)
  - Etc.