Distributed Systems
CS6421
Timing Worksheet

Prof. Roozbeh Haghnazar

Slides Credit:

Prof. Tim Wood and Prof. Roozbeh Haghnazar

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

This Week: Coordination

- Clock Synchronization
- Logical Clocks
- Vector Clocks
- Mutual Exclusion
- Election Algorithms

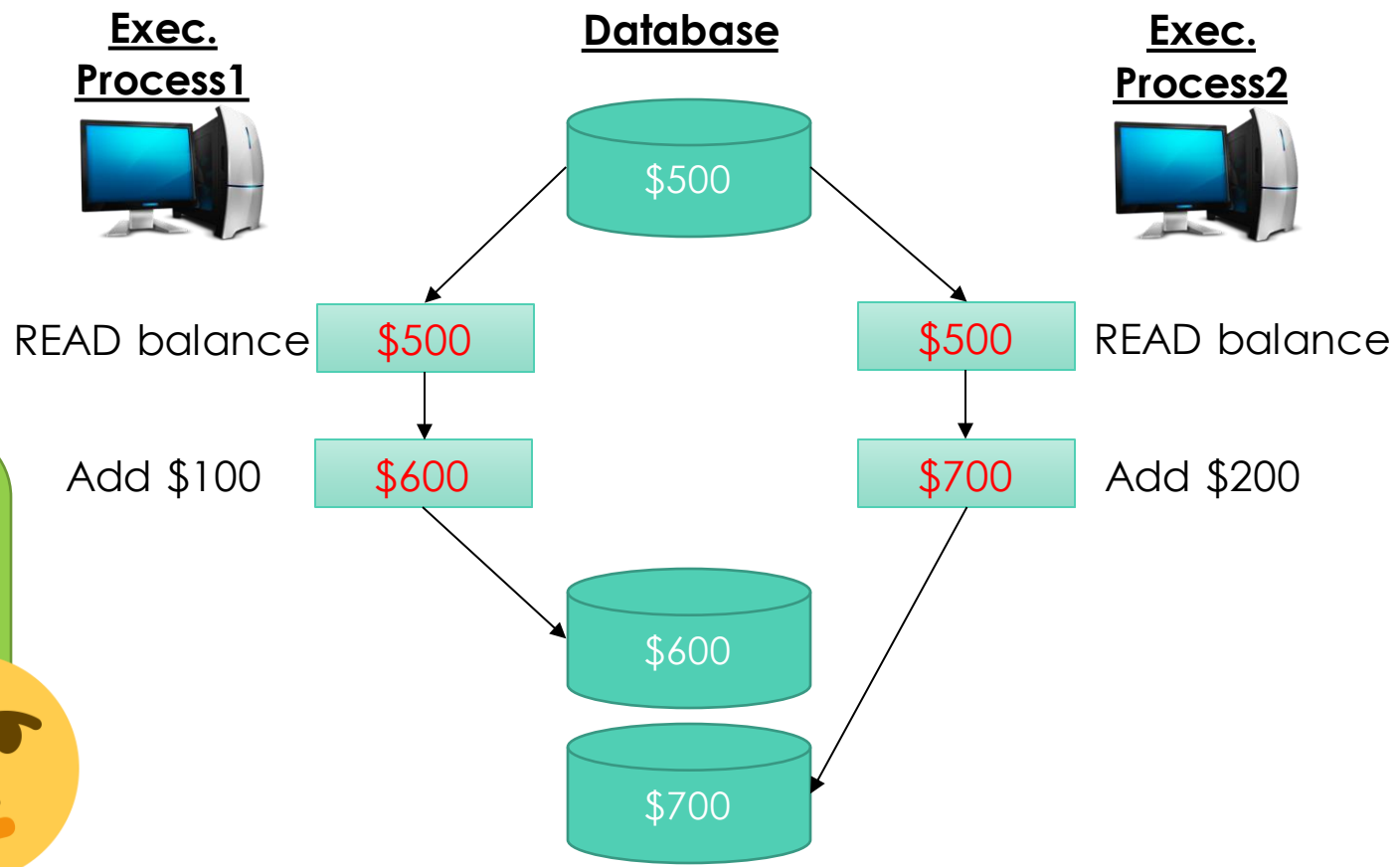
How can distributed components **coordinate** and agree on the **ordering** of events?

Challenges

- Heterogeneity
- Openness
- Security
- Failure Handling
- **Concurrency**
- Quality of Service
- Scalability
- Transparency

Problem and challenge ex.

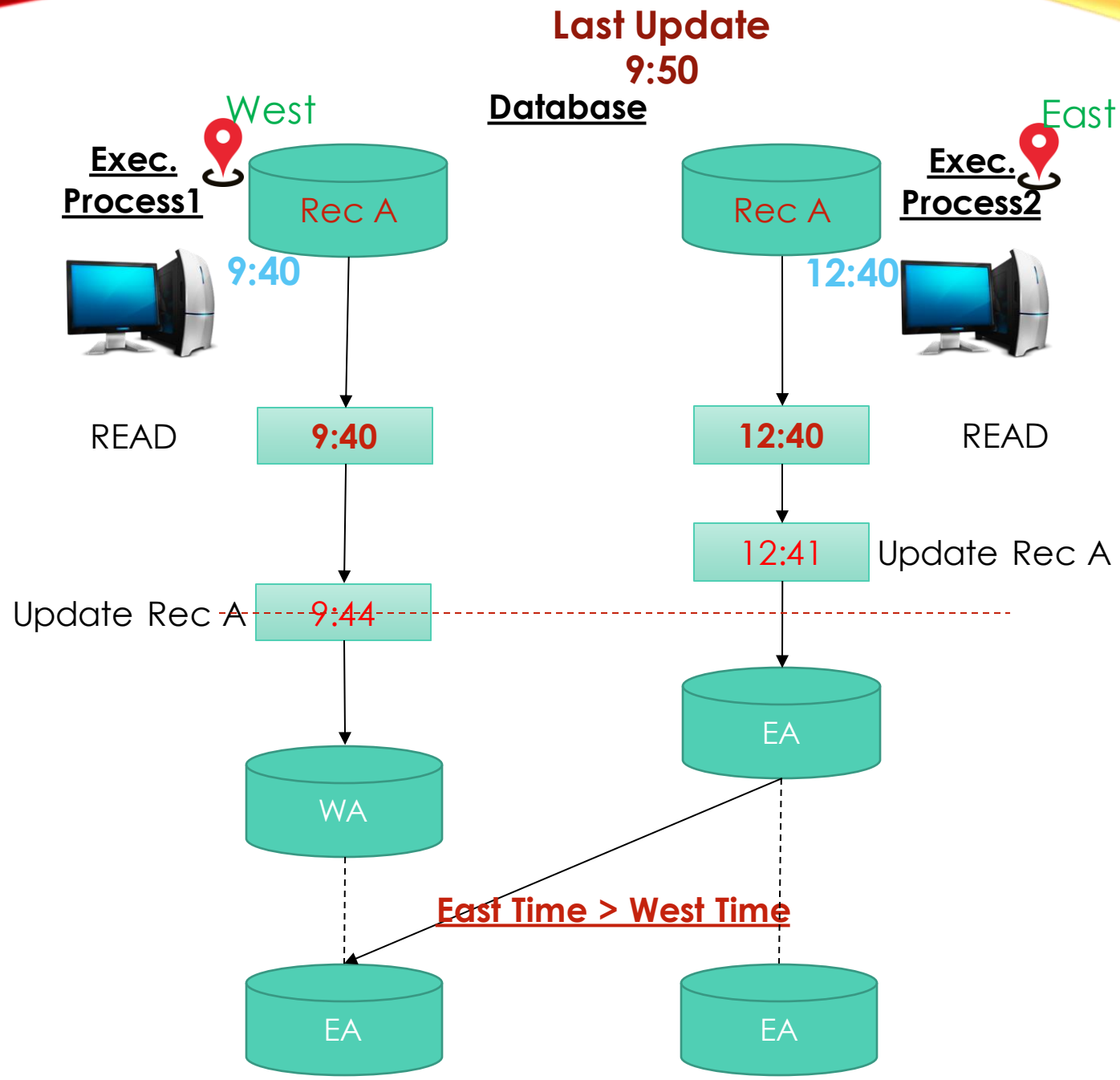
- Concurrency challenge in a database
  - Lost update anomaly



Time and clock!!!
Or
Lock

Clock and Time

- We need to replicate the last update
- Concurrent operation and replication conflict



Problem

- In centralized management, all nodes can make agreement for a shared variable value under the master node control.
- But what if that the system uses distributed management?

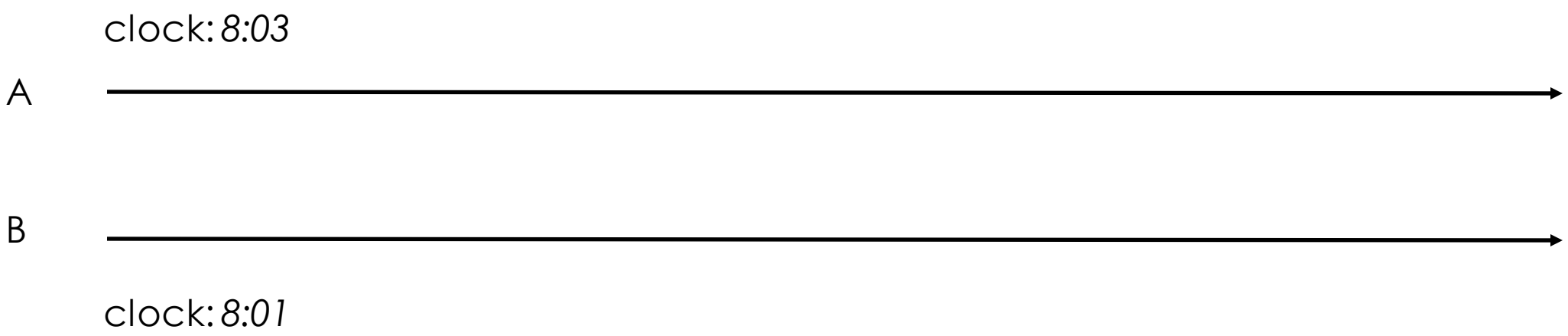
Clocks and Timing

- Distributed systems often need to order events to help with consistency and coordination

- Coordinating updates to a distributed file system
- Managing distributed locks
- Providing consistent updates in a distributed DB

Coordinating time?

- How can we synchronize the clocks on two servers?
  - Physical
  - Logical
- Clock Skew : the differences between the crystals speed



Physical clocks : Sundial

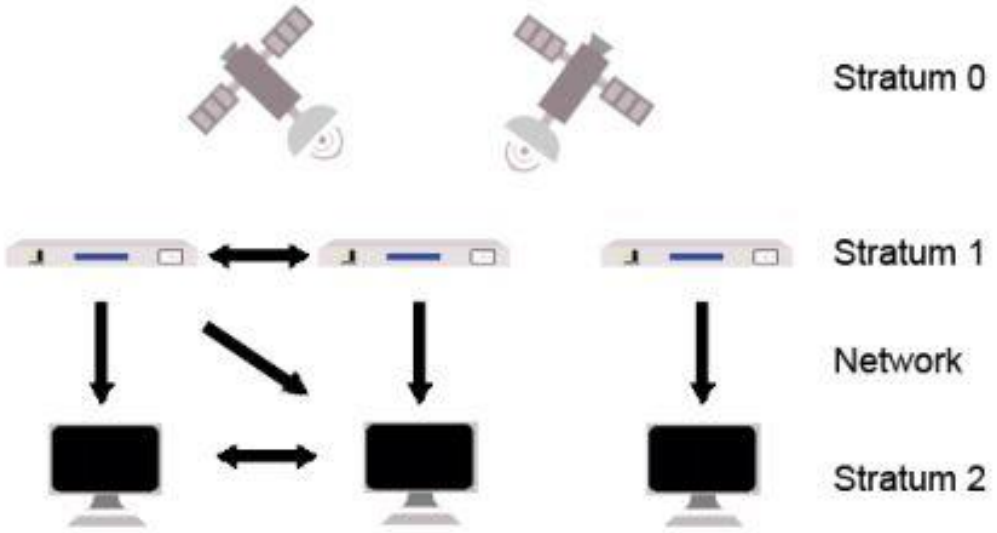
- Sundial.
- Solar Day varies due:
  - Core activities of earth
  - Rise & Tide of oceans
  - Gravity
  - Orbit around the sun, not a perfect circle

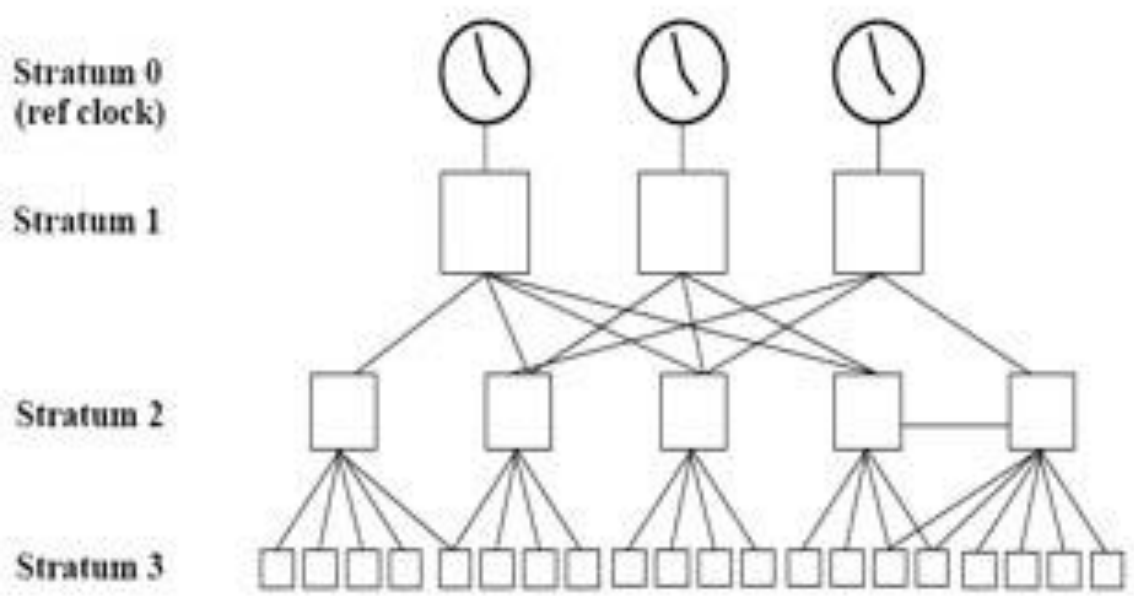
PHYSICAL CLOCKS : ATOMIC CLOCK

- Accurate time regardless of earth movement and gravity
- Since 1968, the International System of Units (SI) has defined the second as the duration of 9,192,631,770 cycles of radiation corresponding to the transition between two energy levels of the ground state of the caesium-133 atom. In 1997, the International Committee for Weights and Measures (CIPM) added that the preceding definition refers to a Caesium atom at rest at a temperature of absolute zero.
- UTC sends pulses for the ***wwv receiver***

NTP Stratum Model

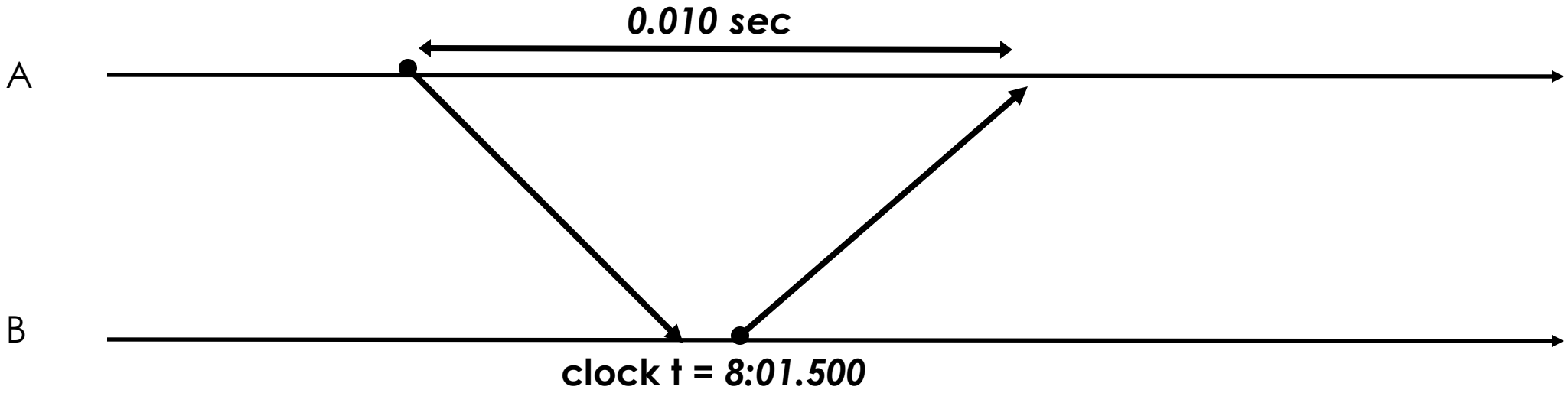
The NTP Stratum model is a representation of the hierarchy of time servers in an NTP network, where the Stratum level (0-15) indicates the device's distance to the reference clock.





CRISTIAN'S ALGORITHM

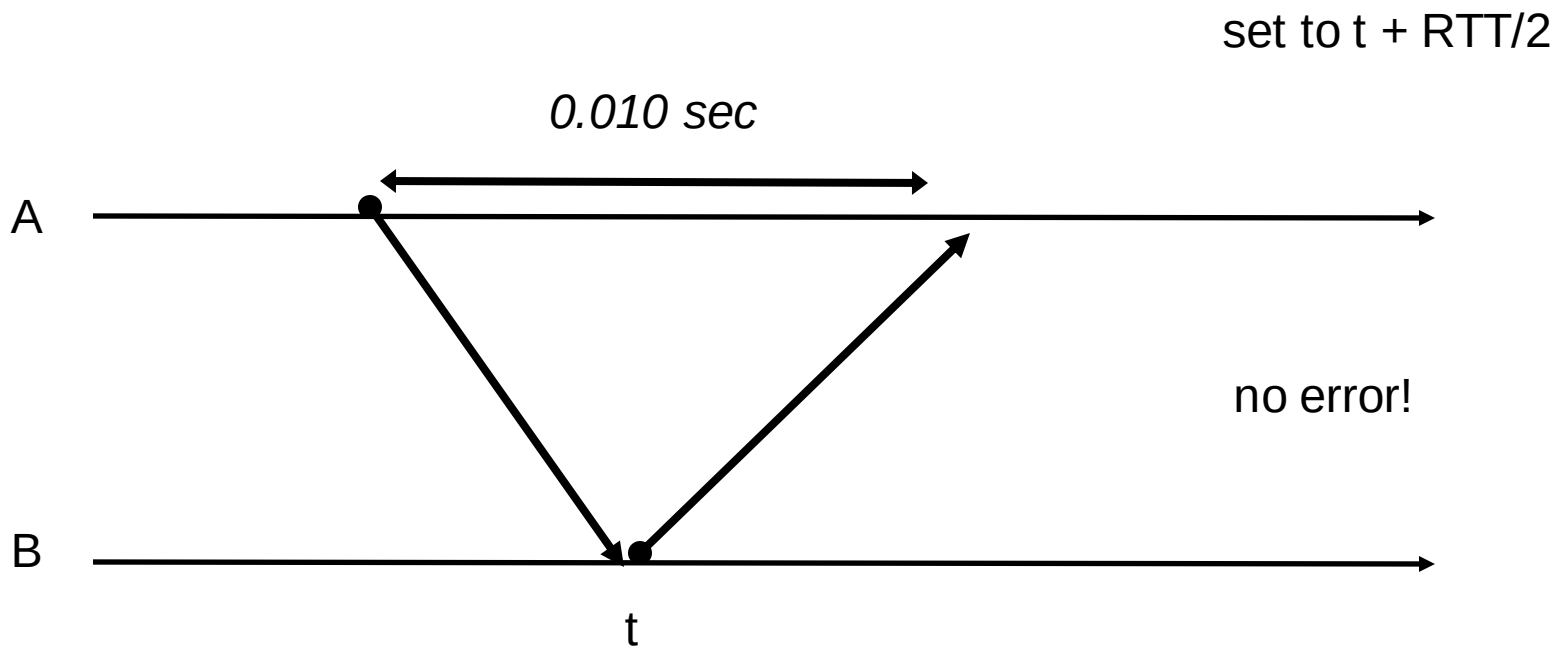
- Easy way to synchronize clock with a time server

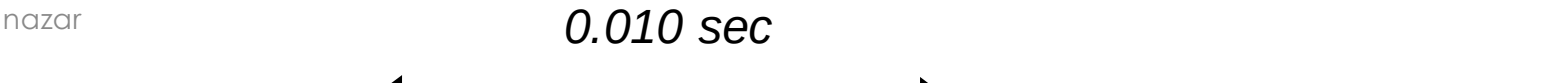


- Client sends a clock request to server
- Measures the round trip time
- Set clock to t + 1/2*RTT (8:01.505)

CRISTIAN'S ALGORITHM

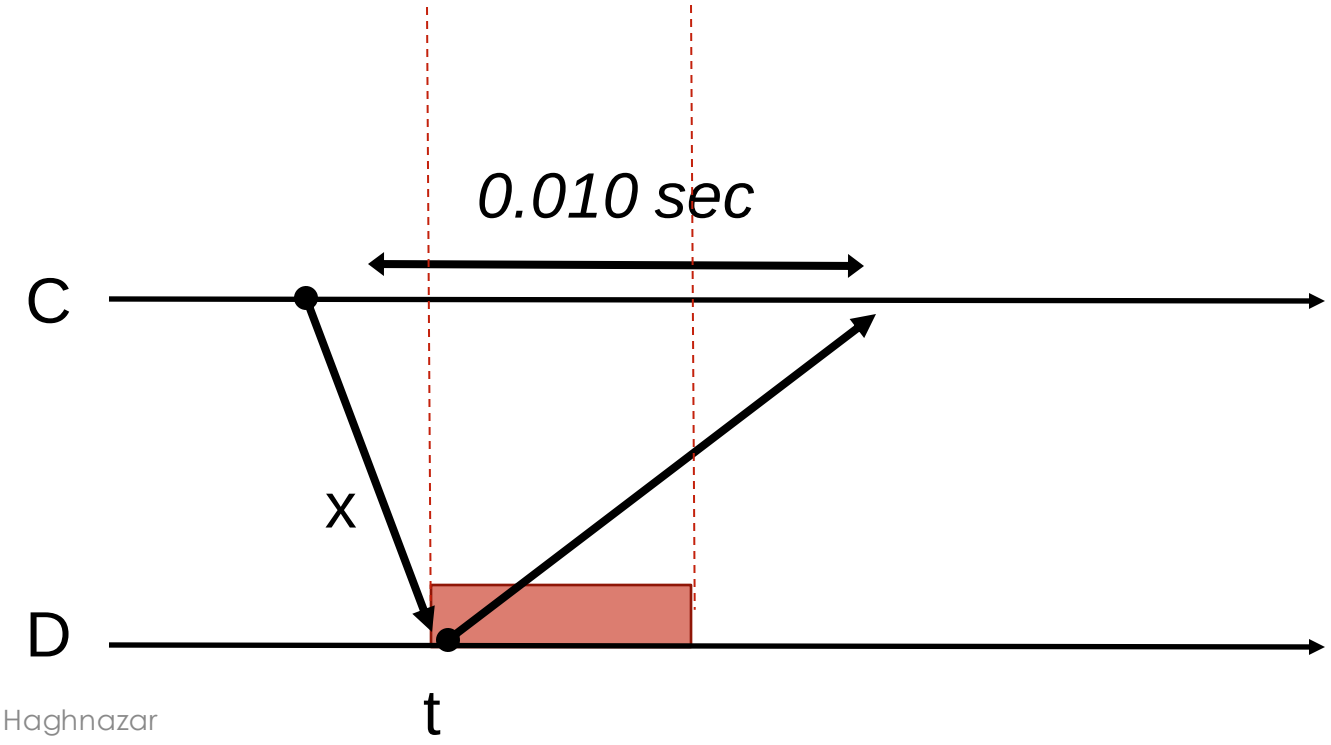
- What will affect accuracy?





Cristian's Algorithm

- Suppose the minimum delay between A and B is X

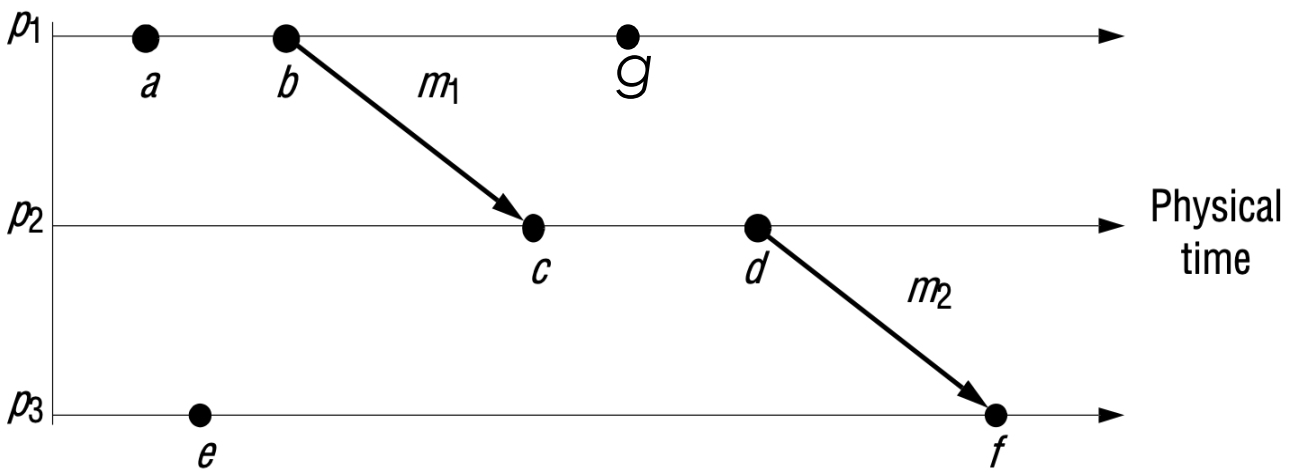


ORDERING

- Sometimes we don't actually need clock time
- We just care about the order of events!
- What event happens before another event?
- e → e' means event e happens before event e'
- Easy: we'll just use counters in each process and update them when events happen!
- Maybe not so easy…

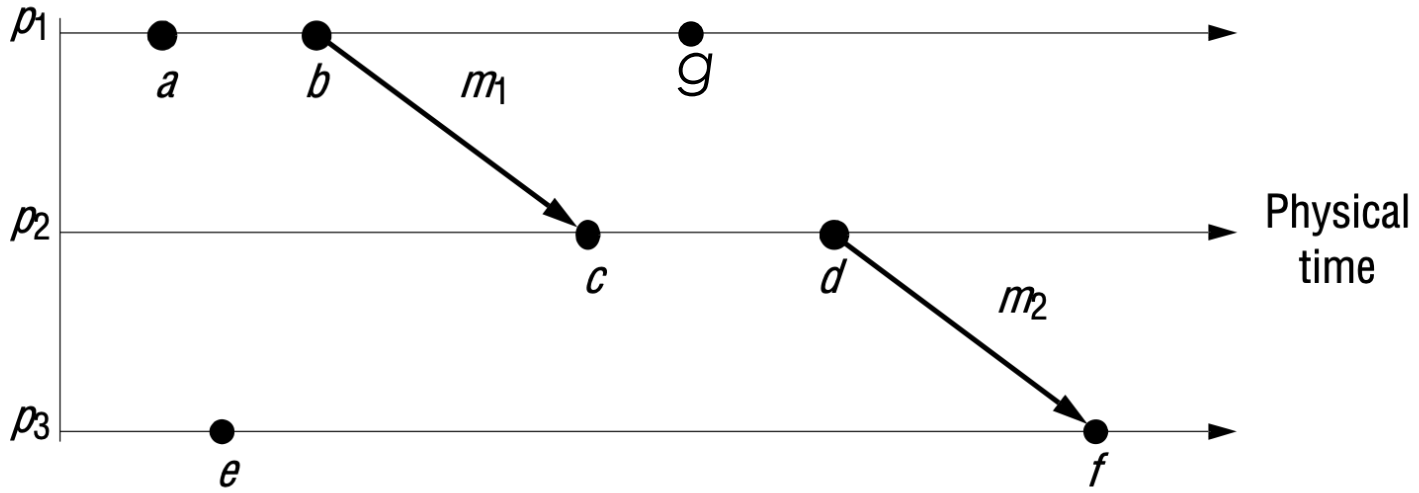
Ordering

- An event is **one** of the following:
  - Action that occurs within a process
  - Sending a message
  - Receiving a message
- What is true? What can't we know?



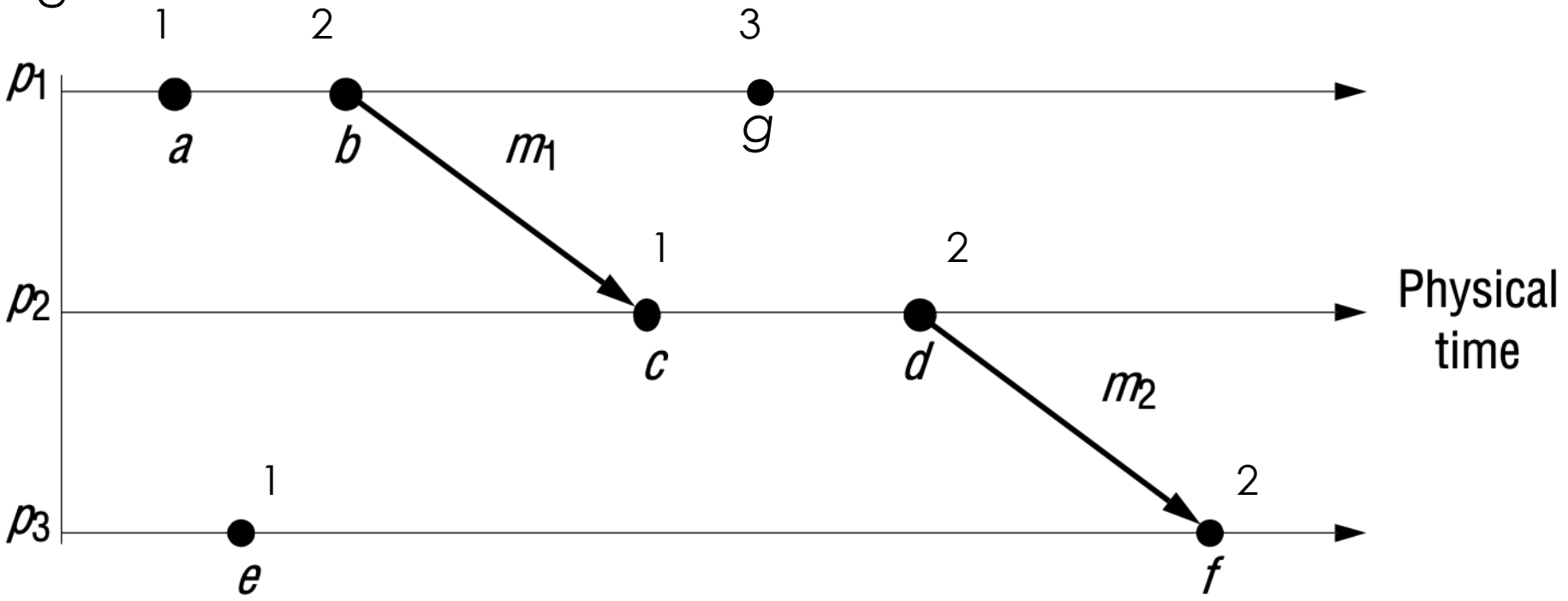
Happens Before: →

- What is true?
  - a->b, b->g, c->d, e->f (events in same process)
  - b->c, d->f (send is before receive)
- What can't we know?
  - e ?? a
  - e ?? c



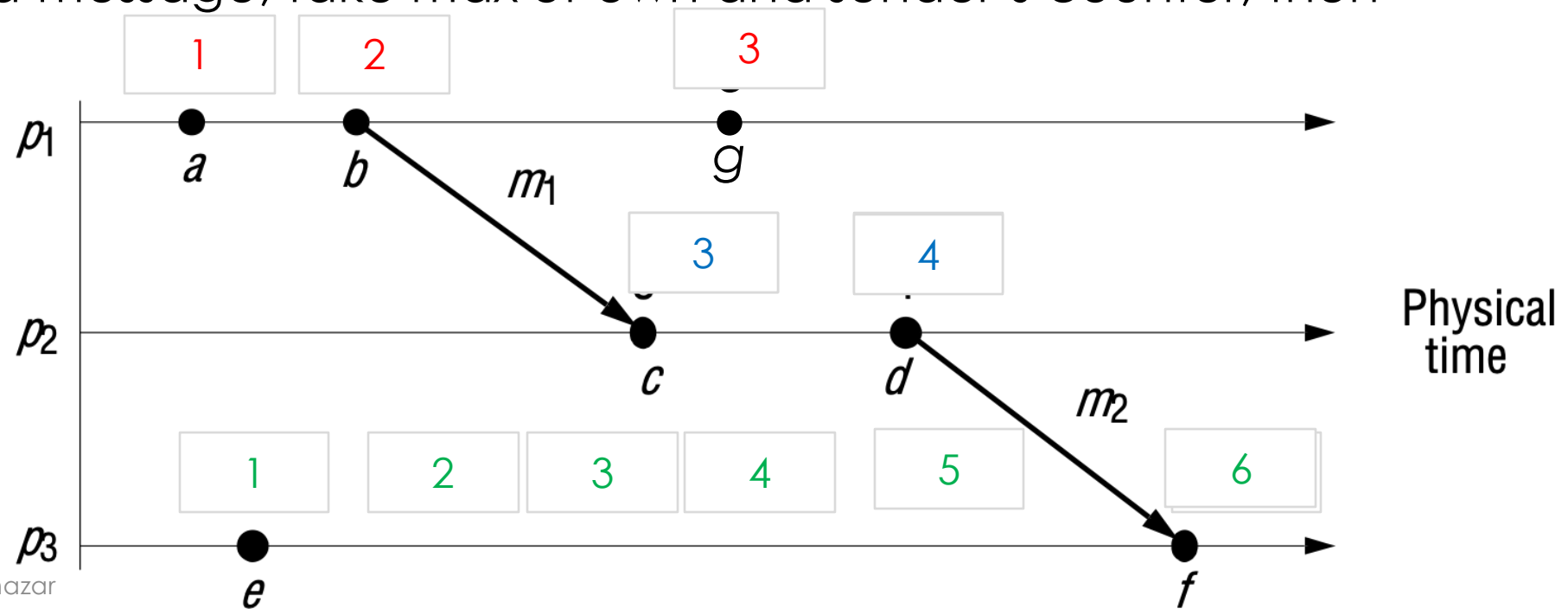
ORDERING

- If we keep count of events at each process independently, are those counters meaningful?



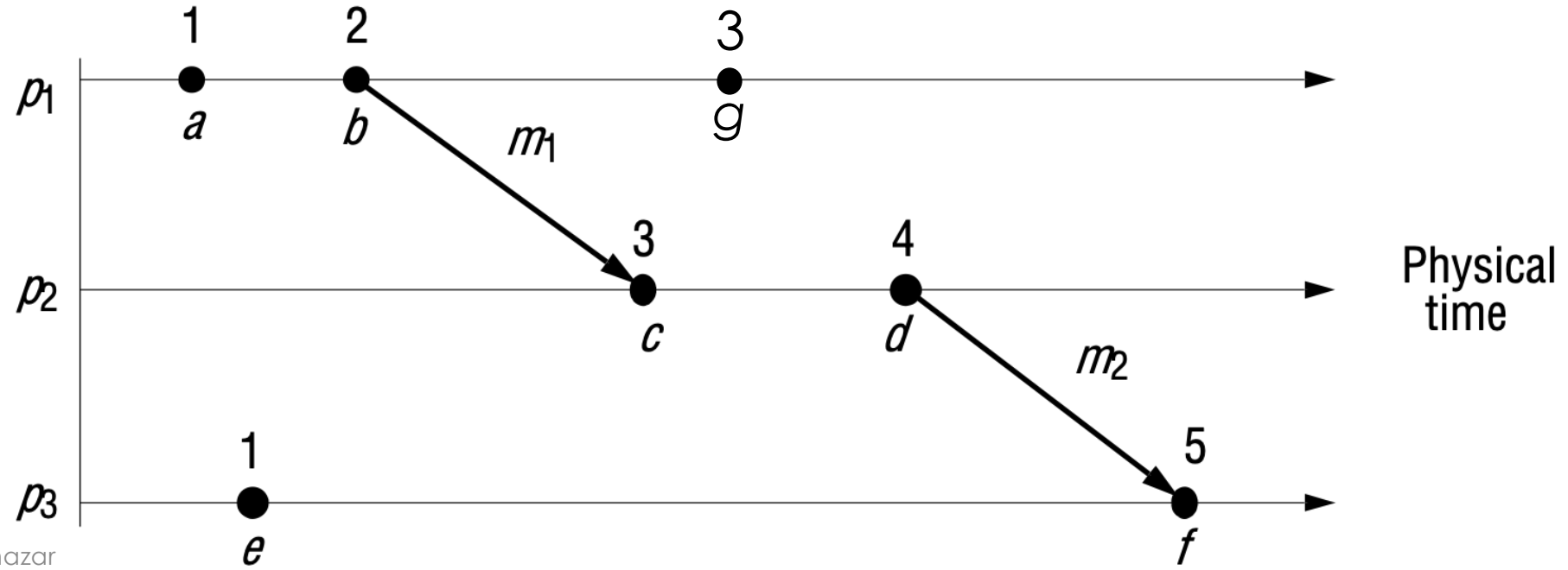
Logical Clock: Lamport Clock

- Each process maintains a counter, L
- Increment counter when an event happens
- When receiving a message, take max of own and sender's counter, then increment



Lamport Clock

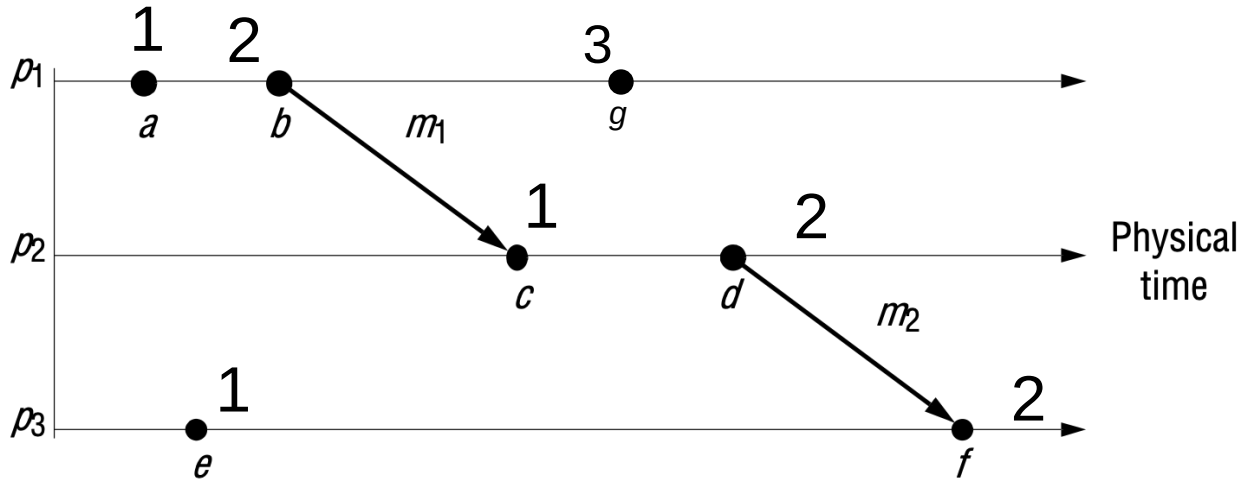
- Each process maintains a counter, L
- Increment counter when an event happens
- When receiving a message, take max of own and sender's counter, then increment



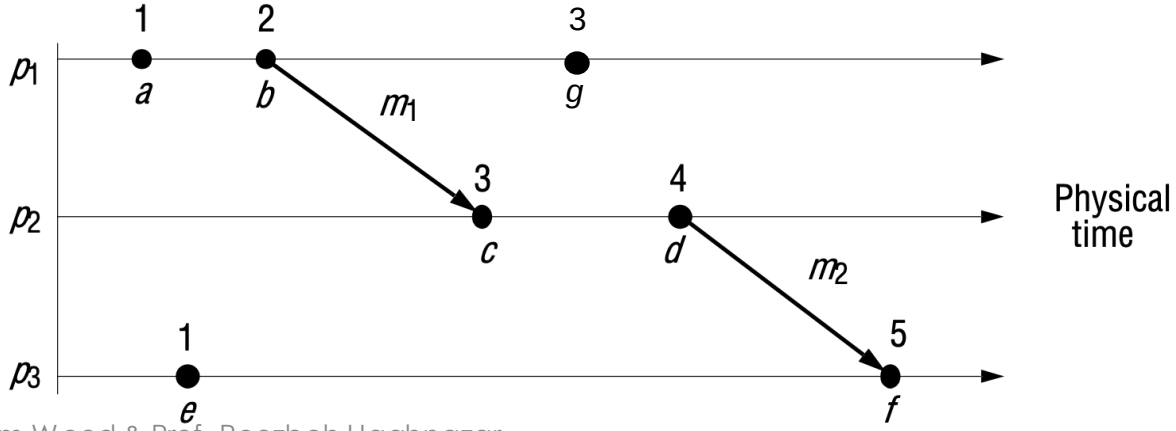
Clock Comparison

Independent clocks

- if e->e', then:
- C(e) ??? C(e')



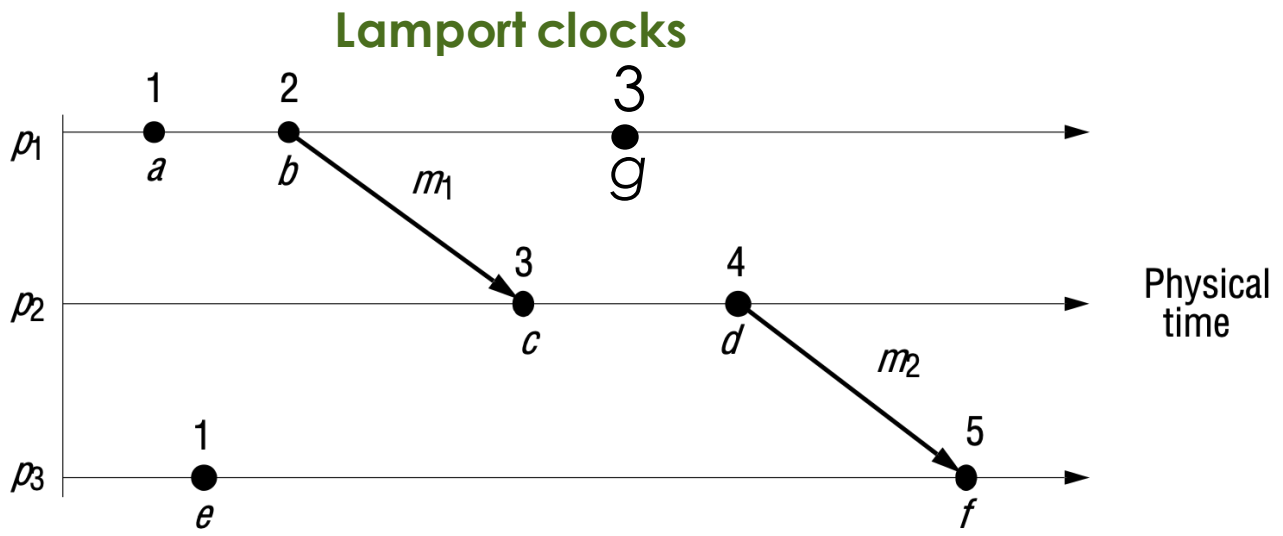
Lamport clocks



- if e->e', then:
- L(e) < L(e')

Clock Comparison

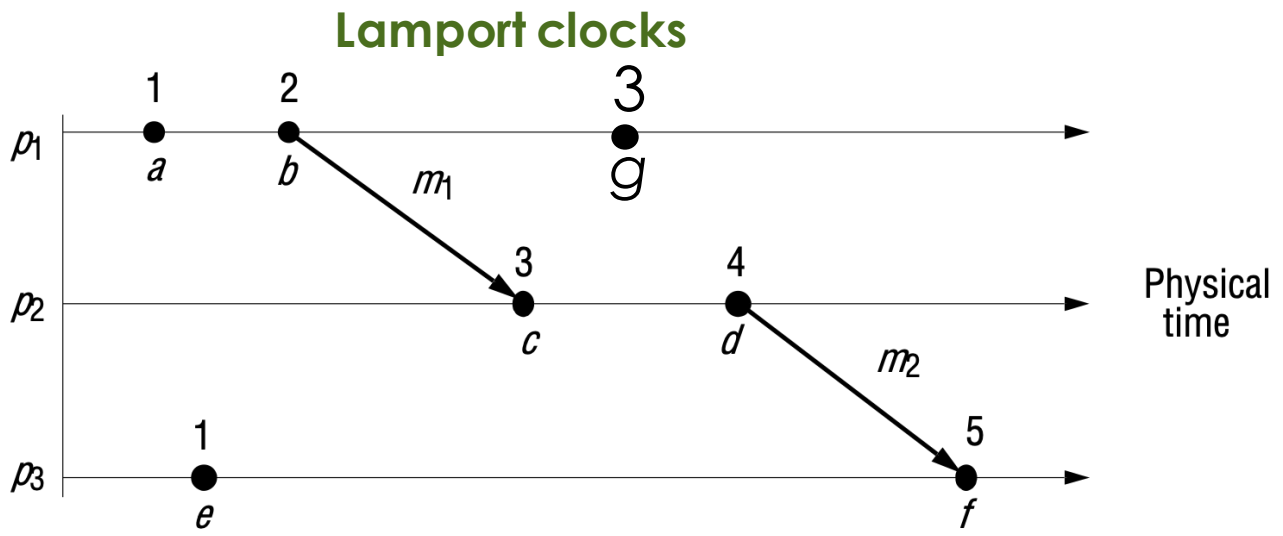
- **Is the opposite true?**
- **if L(e) < L(e') then do we know e->e'?**



- **if e->e', then:**
- **L(e) < L(e')**

Clock Comparison

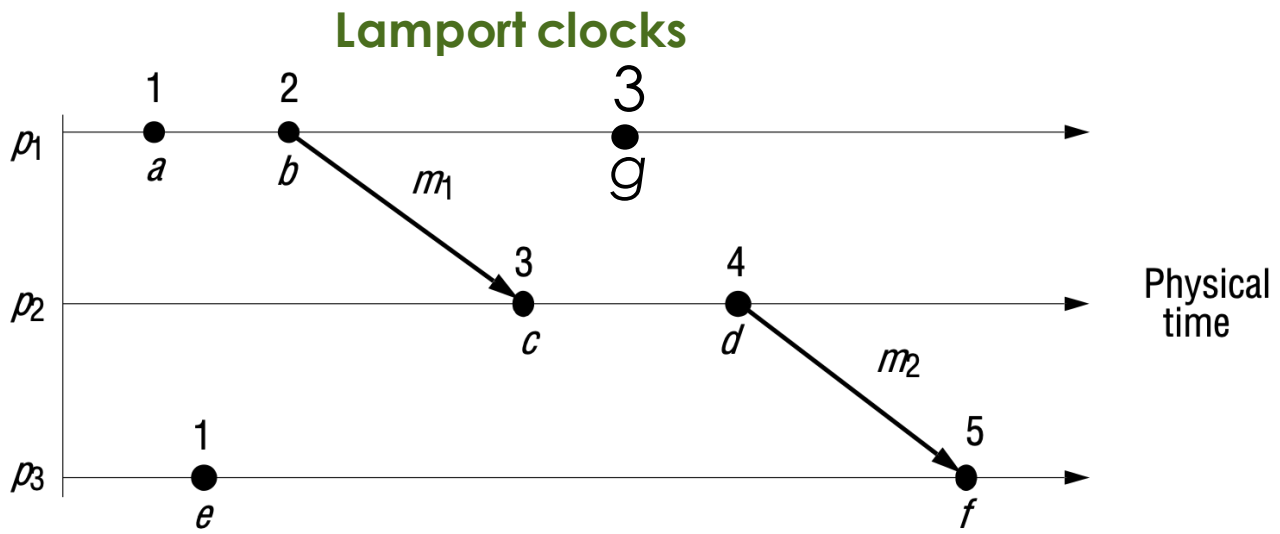
- **Is the opposite true? No!**
- **Lamport clocks don't actually let us compare two clocks to know how they are related :(**



- **if e->e', then:**
- **L(e) < L(e')**

Lamport Clocks

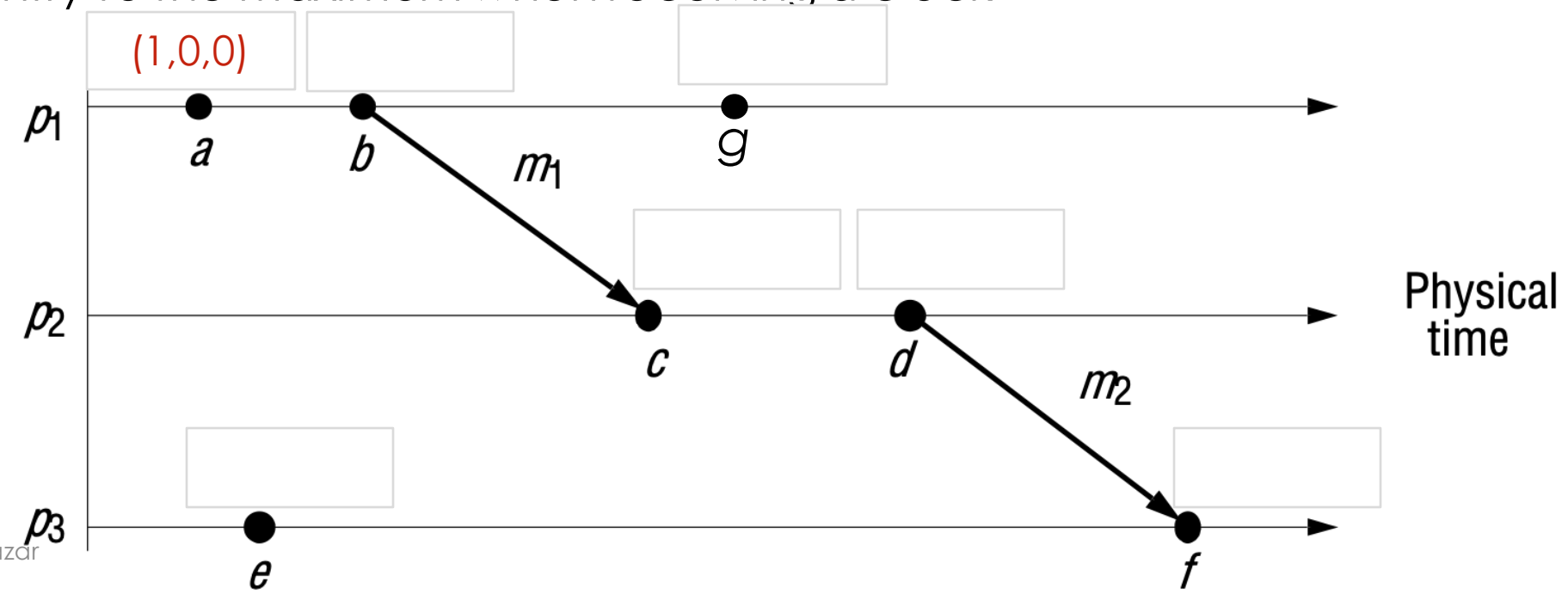
- Lamport clocks are better than nothing
  - but only let us make limited guarantees about how things are ordered
- Ideally we want a clock value that indicates:
  - If an event happened before another event
  - If two events happened *concurrently*



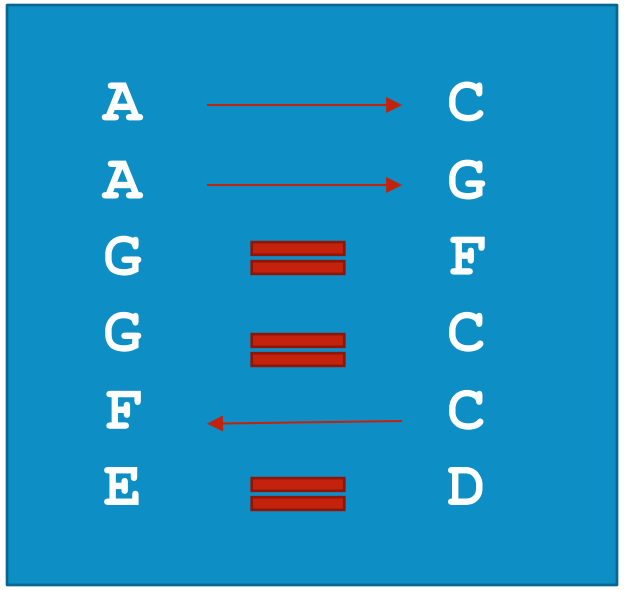
P1	P2	P3
a:1	c:3	e:1
b:2	d:4	f:5
g:3		

Vector Clocks

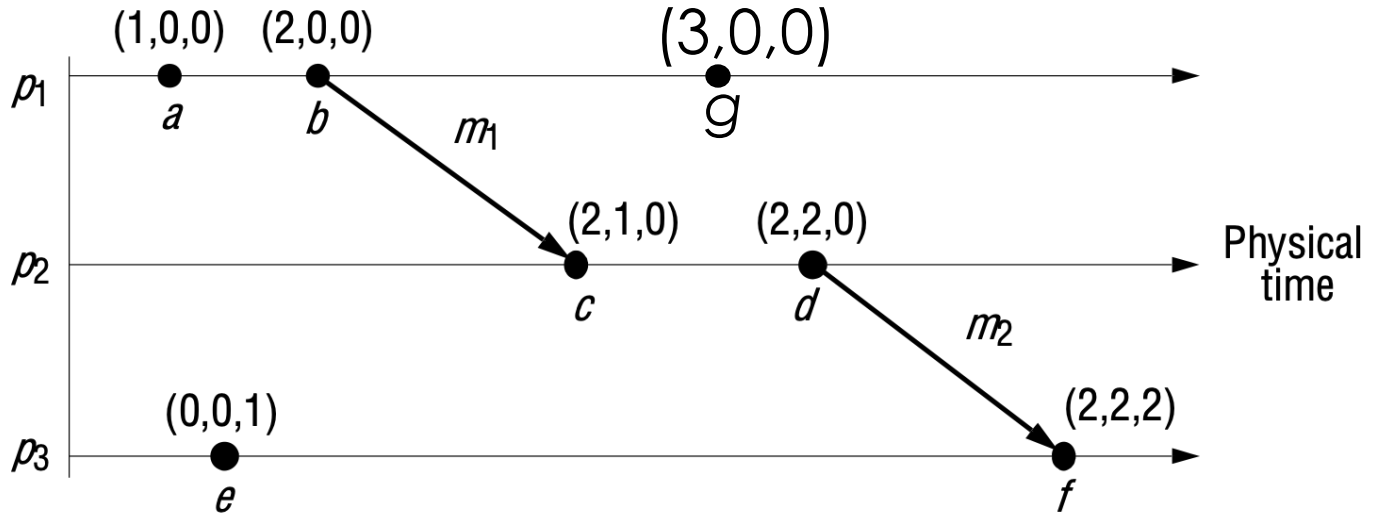
- Each process keeps an array of counters: (p1, p2, p3)
  - When p_i has an event, increment V[p_i]
  - Send full vector clock with message
  - Update each entry to the maximum when receiving a clock



Vector Clocks



- Now we can compare orderings!
- if V(e) < V(e') then e->e'
  - (a,b,c) < (d,e,f) if:
    a≤d & b≤e & c≤f
- If neither V(e) < V(e') nor V(e') < V(e) then e and e' are concurrent events



P1	P2	P3
a: 1,0,0	c: 2,1,0	e: 0,0,1
b: 2,0,0	d: 2,2,0	f: 2,2,2
g: 3,0,0		

Lamport vs Vector

- Which clock is more useful when you can't see the timing diagram?
  - Remember, your program will only see these counters!

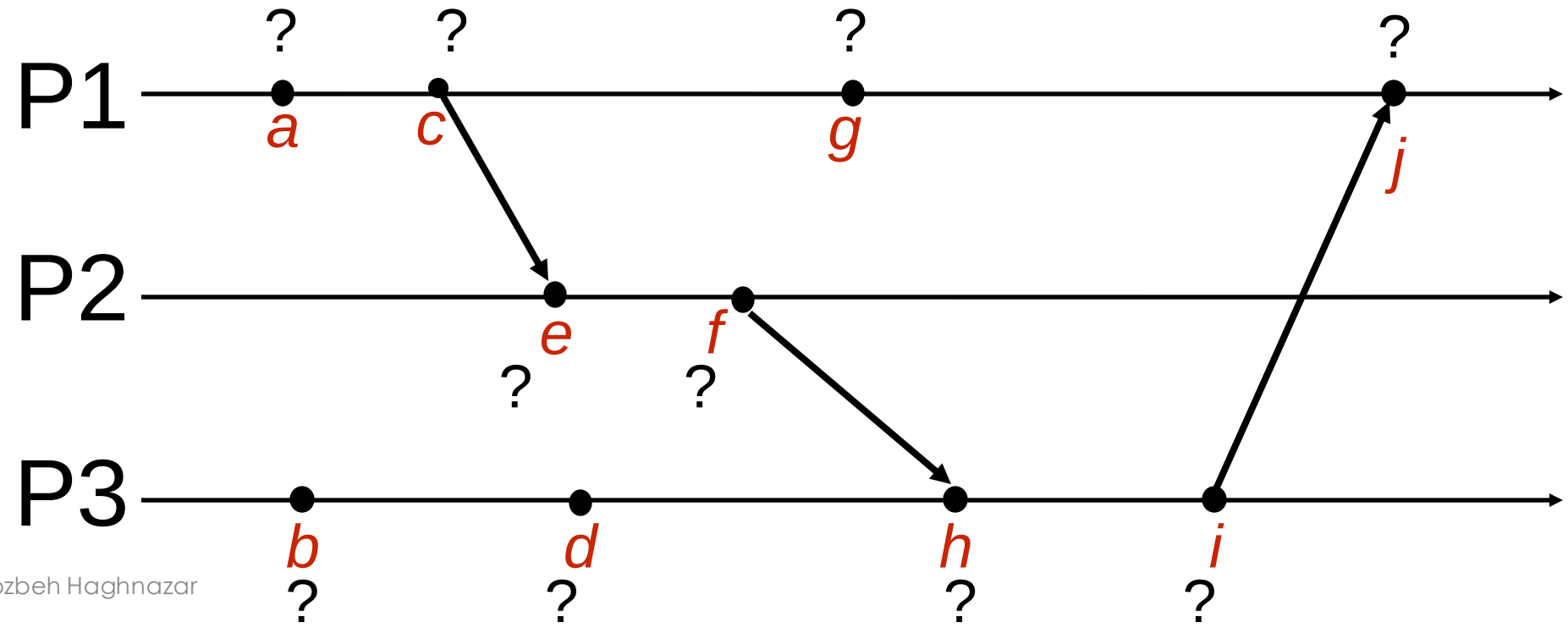
P1	P2	P3
a:1	c:3	e:1
b:2	d:4	f:5
g:3		

P1	P2	P3
a: 1,0,0	c: 2,1,0	e: 0,0,1
b: 2,0,0	d: 2,2,0	f: 2,2,2
g: 3,0,0		

VC Example

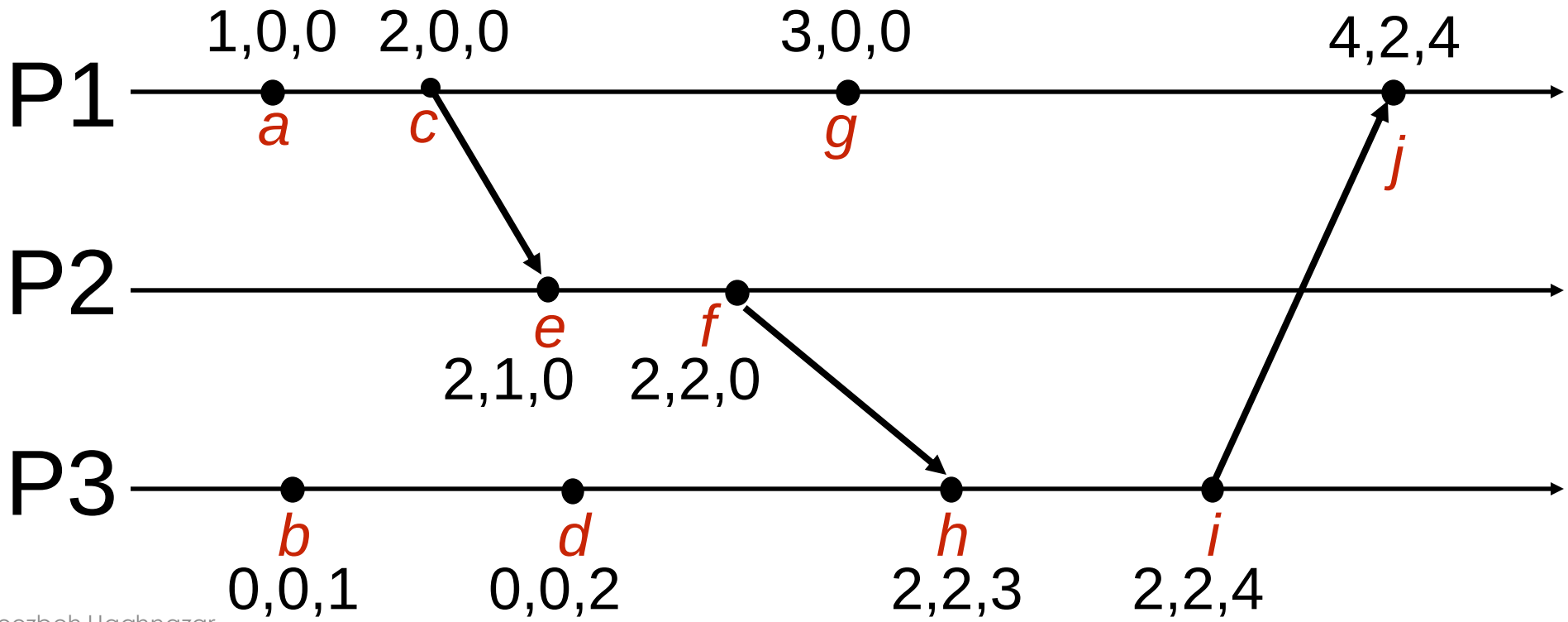
VC Rules:
- When p_i has an event, increment V[p_i]
- Update each entry to the maximum when receiving a clock
- Send full vector clock with message

- What are the vector clocks at each event?
- Assume all processes start with (0,0,0)



How to Compare VC?

- Example Answer



Vector Clocks

- Allow us to compare clocks to determine a **partial ordering** of events
- Example usage: versioning a document being edited by multiple users. How do you know the order edits were applied and who had what version when they edited?
- Is there a drawback to vector clocks compared to Lamport clocks?

Version Vectors

- Problem: If multiple servers allow the same key to be updated, its important to detect when the values are concurrently updated across a set of replicas.
- We can apply the vector clock concept to versioning a piece of data
  - This is used in many distributed data stores (DynamoDB, Riak)
- When a piece of data is updated:
  - Tag it with the **actor** who is modifying it and the version #
  - Treat the (actor: version) pairs like a vector clock
- The version vectors can be used to determine a causal ordering of updates
- Also can detect concurrent updates
- Need to have a policy for resolving conflicts
  - If two versions are concurrent, they are "siblings", return both!

EXAMPLE:

Let's assume we have a key K, with value U. We're assuming that we have an empty version vector to begin with. Client's C2 and C3 sync the same state from the system that's implementing Version Vectors.

C3 updates the value to V & sends a PUT command, with the local state of Version Vector it has(empty VV).

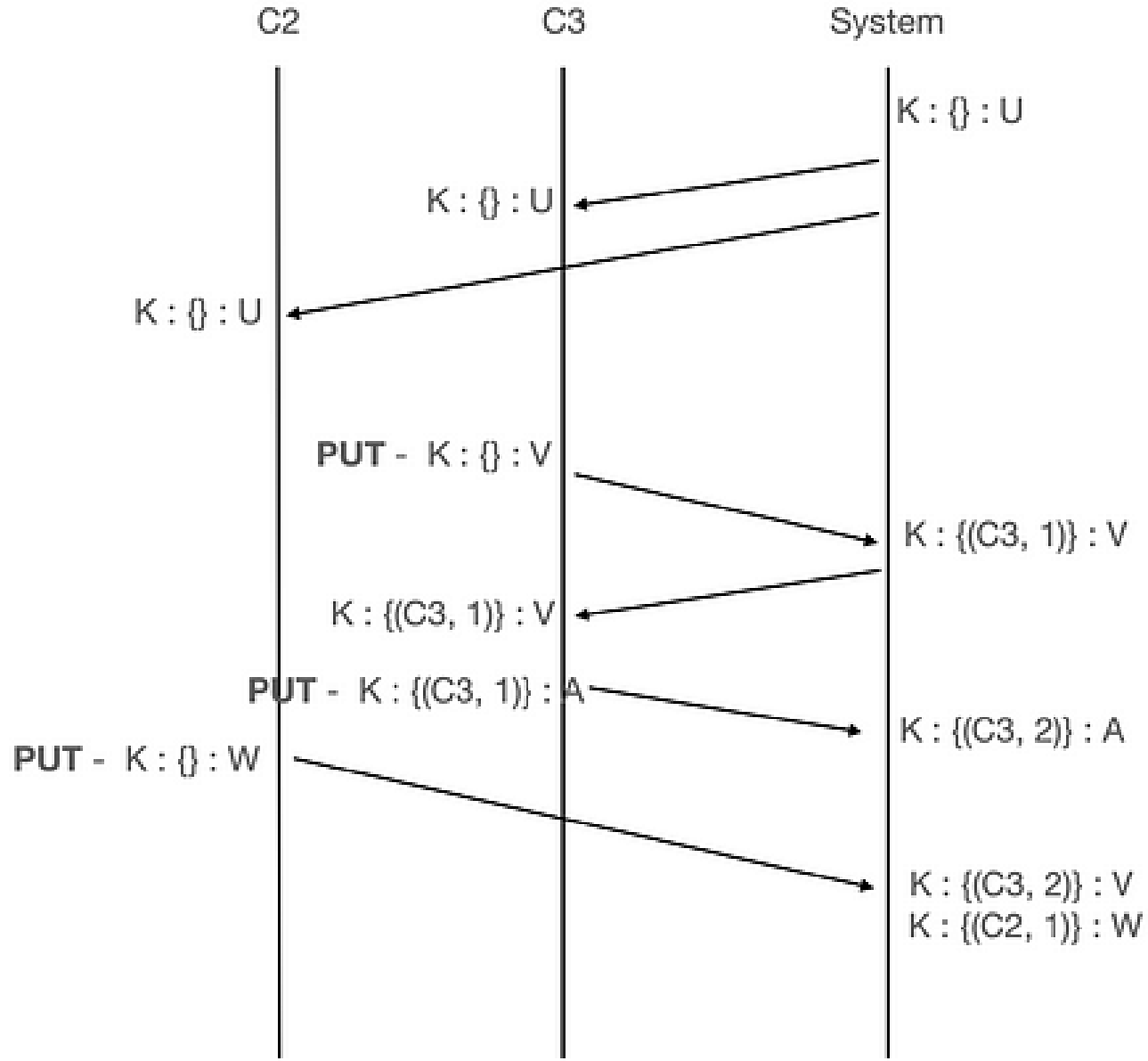
The System on receiving the Version Vector, compares it with its local state(again empty). So it creates a new Version Vector for this client(C3) and updates the value of key K to V.

C3 syncs the state from the server and again updates the value to A from V and sends a PUT command.

The System on receiving the Version Vector, compares it with its local state. This time the local state has an entry for C3, so it increments the counter to 2 & updates the value to A.

C2 updates the value to W & sends a PUT command, with the local state of Version Vector it has(empty VV).

The System on receiving the Version Vector, compares it with its local state((C3, 2)). Since there is no local state for C2, it creates a new state of (C2, 1). If we compare the 2 Version Vectors, {C3, 2} and {C2, 1}, based on what we've discussed before, they are concurrent. Since there is a conflict, the System stores both the version vectors(1 per client) as siblings.



Version Vectors

- Alice tells everyone to meet on Wednesday
- Dave and Cathy discuss and decide on Thursday
- Ben and Dave exchange emails and decide Tuesday

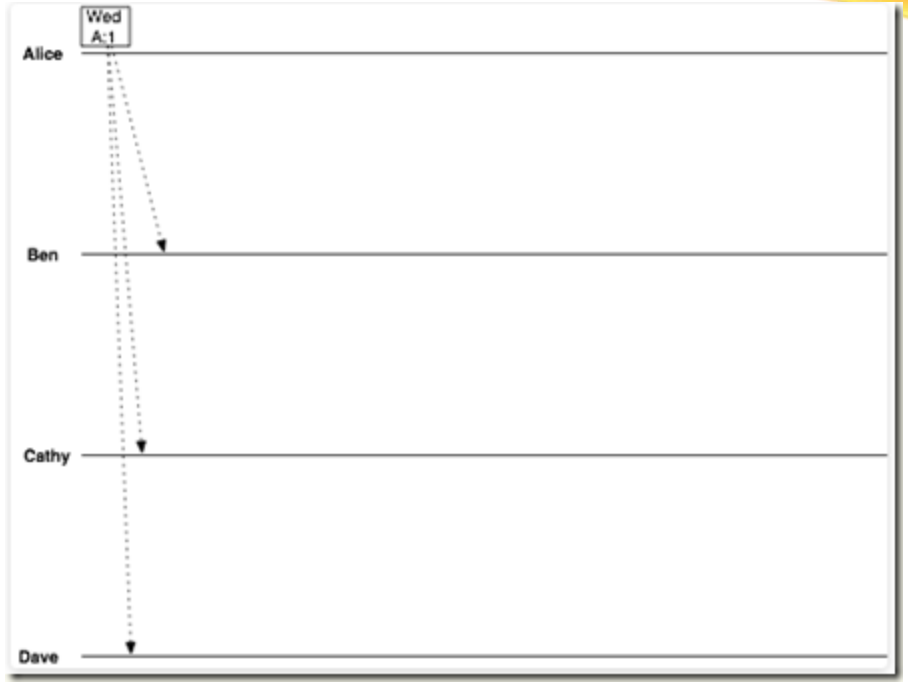
Order?

- Alice wants to know the final meeting time, but Dave is offline and Ben and Cathy disagree… what to do?

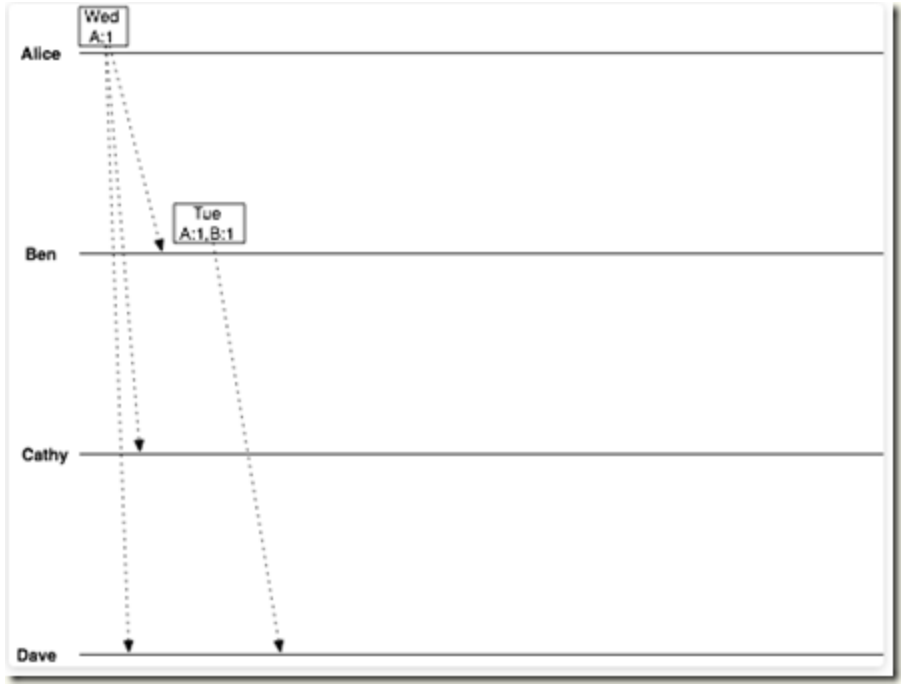
Alice	Ben	Cathy	Dave
Wednesday	Tuesday	Thursday	???

Version Vectors

- Start with Alice's initial message: "Let's meet Wednesday,"
- date = Wednesday
- vclock = Alice:1

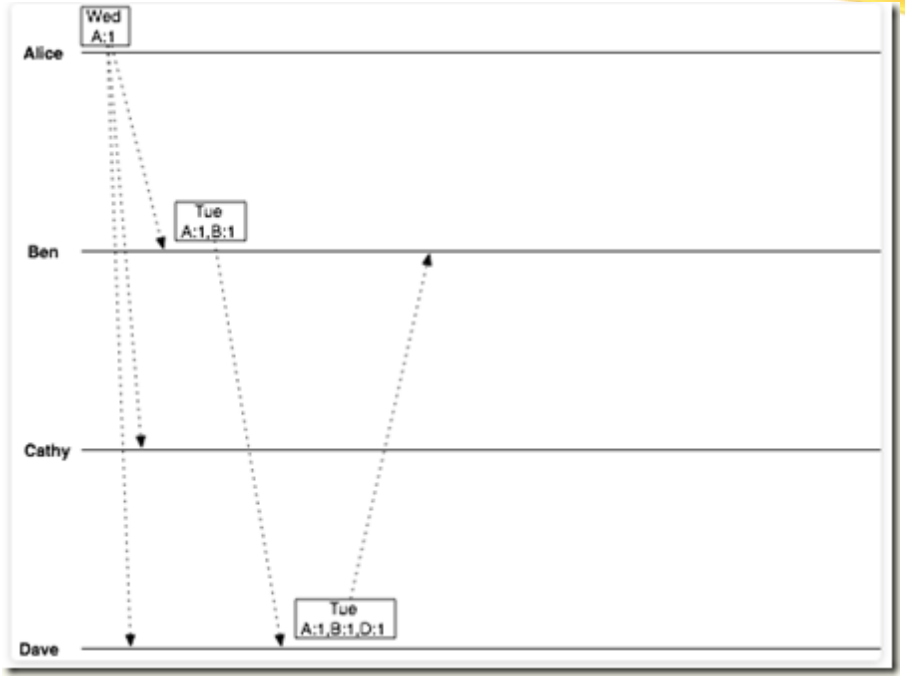


- Ben suggests Tuesday:
- date = Tuesday
- vclock = Alice:1, Ben:1

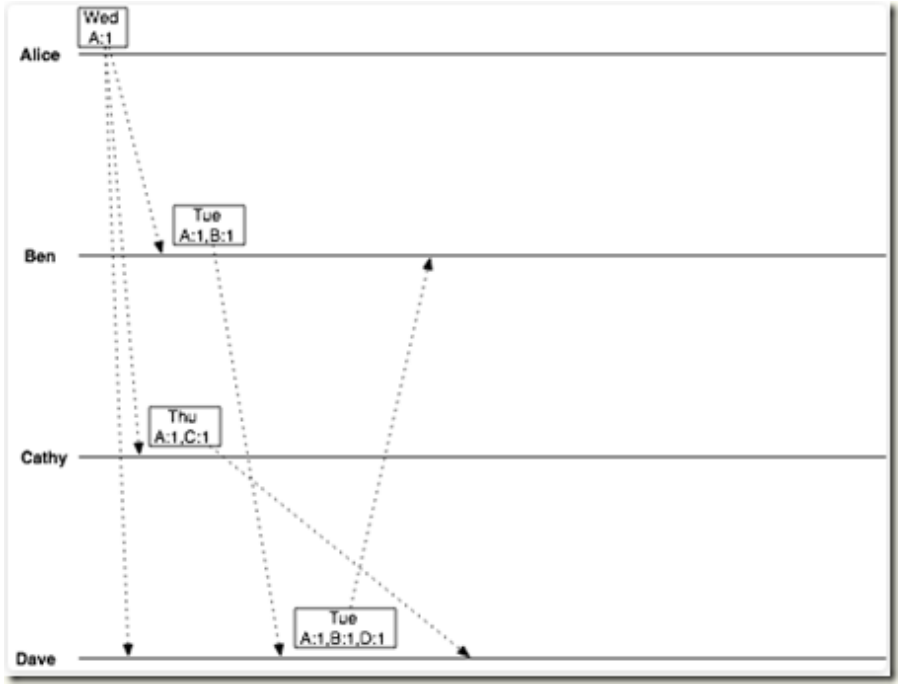


Version Vectors

- Dave replies, confirming Tuesday:
- date = Tuesday
- vclock = Alice:1, Ben:1, Dave:1



- Now Cathy gets into the act, suggesting Thursday:
- date = Thursday
- vclock = Alice:1, Cathy:1



Version Vectors

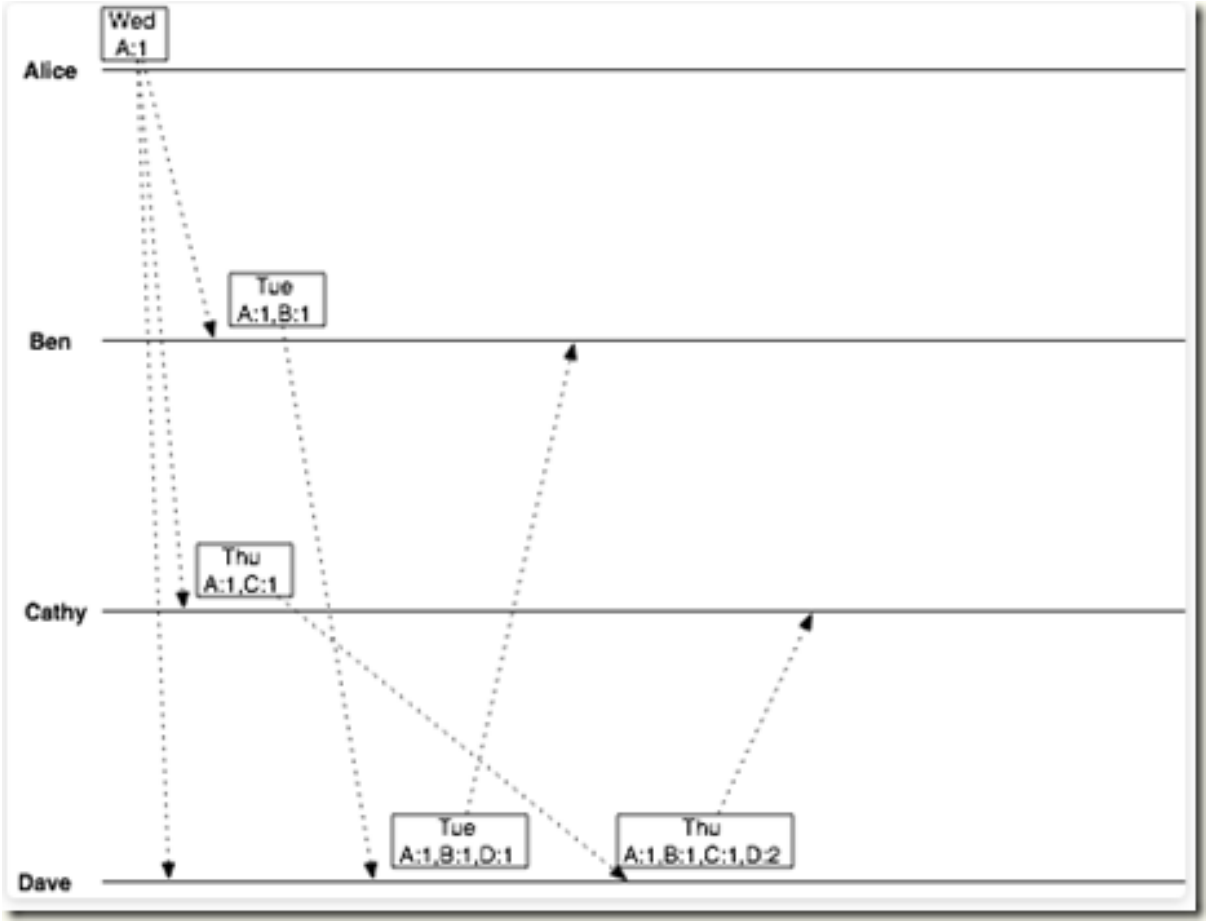
- Dave has two conflicting objects:

- date = Tuesday
- vclock = Alice:1, Ben:1, Dave:1

  and

- date = Thursday
- vclock = Alice:1, Cathy:1

- **date = Thursday**
- **vclock = Alice:1, Ben:1, Cathy:1, Dave:2**

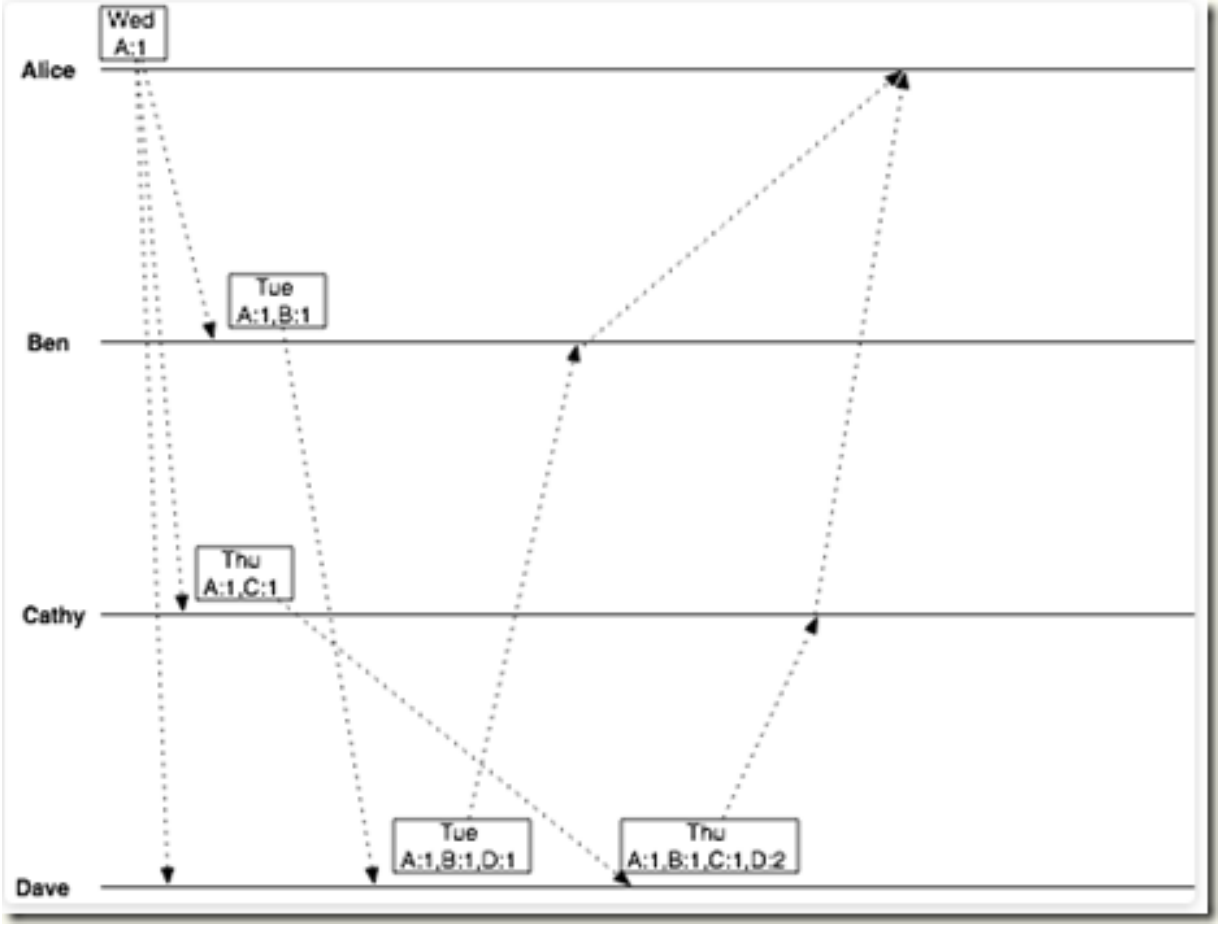


Version Vectors

- So now when Alice asks Ben and Cathy for the latest decision, the replies she receives are, from Ben:

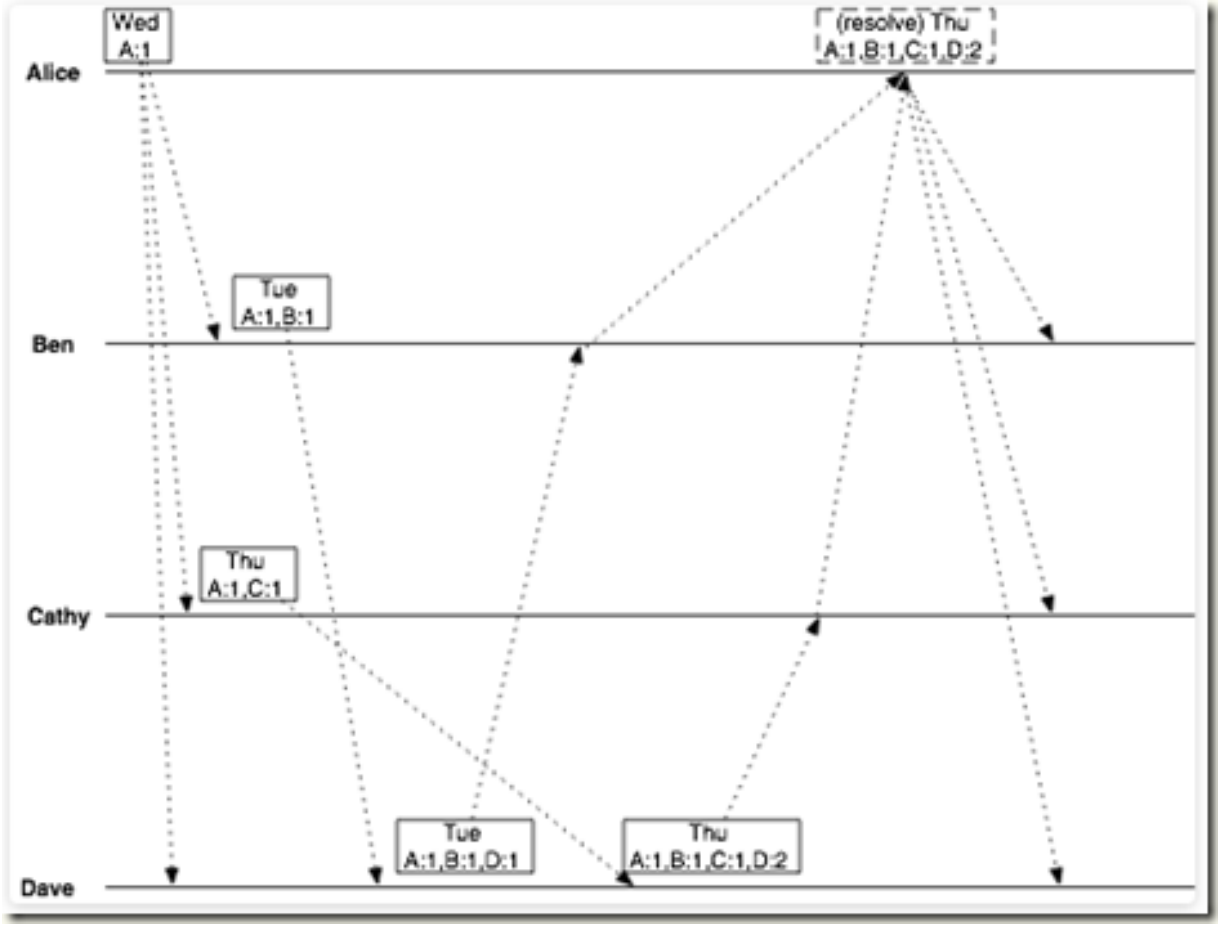
- date = Tuesday
- vclock = Alice:1, Ben:1, Dave:1
- and from Cathy:

- date = Thursday
- vclock = Alice:1, Ben:1, Cathy:1, Dave:2



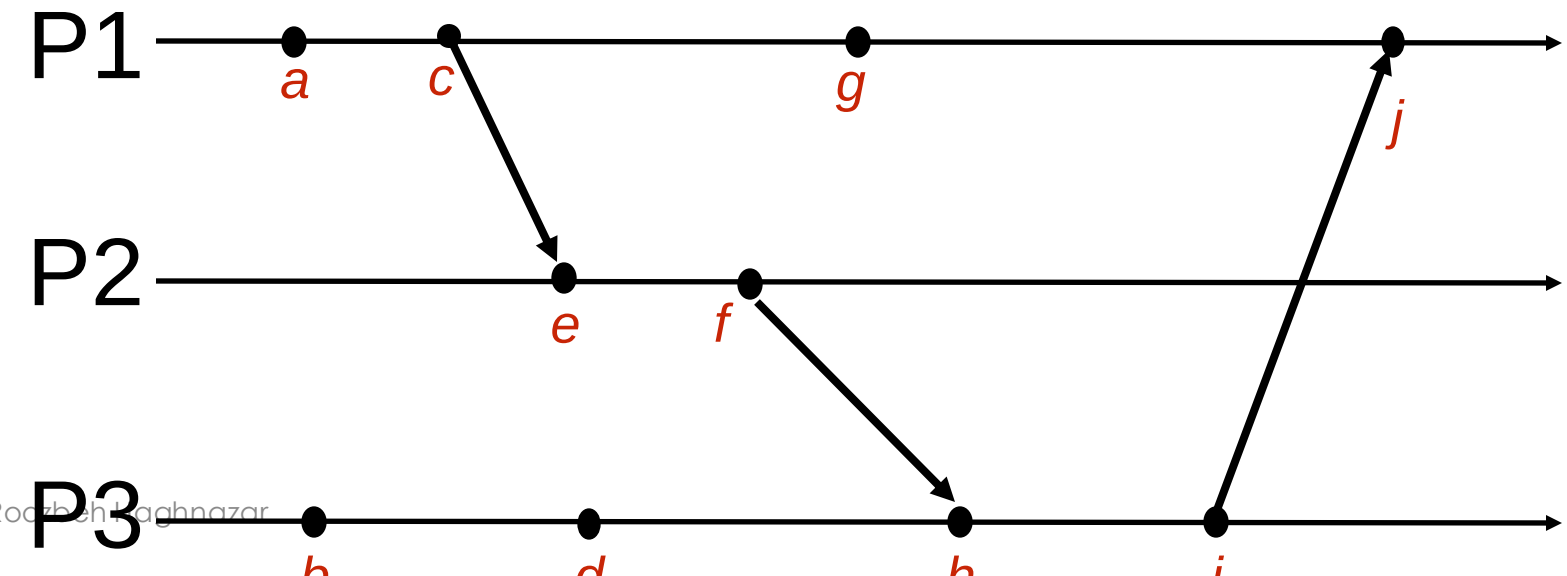
Version Vectors

- From this, she can tell that Dave intended his correspondence with Cathy to override the decision he made with Ben. All Alice has to do is show Ben the vector clock from Cathy's message, and Ben will know that he has been overruled.



DEPENDENCIES

- Vector clocks also help understand the dependency between different events and processes
- Combination of DAG, and Vector Clocks or Version clock can be very powerful to solve the conflicts also



Time and Clocks

- Synchronizing clocks is difficult
- But often, knowing an order of events is more important than knowing the "wall clock" time!
- Lamport and Vector Clocks provide ways of determining a consistent ordering of events
  - But some events might be treated as concurrent!
- The concept of vector clocks or version vectors is commonly used in real distributed systems
  - Track **ordering** of events and **dependencies** between them

Distributed **Coordination (Mutual Exclusion, Consensus)**

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

Survey Feedback

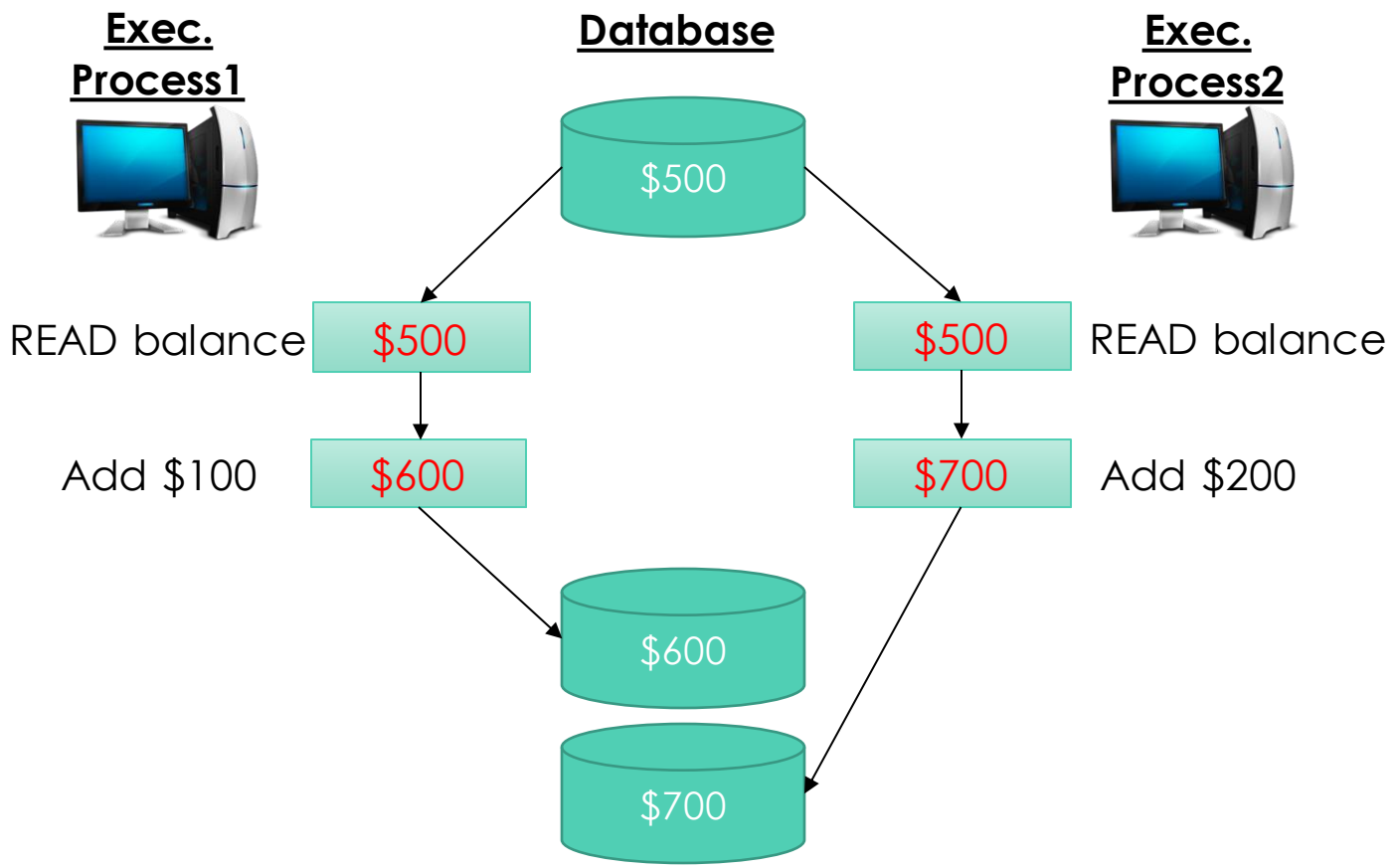
- Breadth vs Depth
- Example Use Cases
- Project Difficulty
- Using cloud trial version – hybrid + on premise VMs
- Programming Language - Go

This Week: Distributed Coordination

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

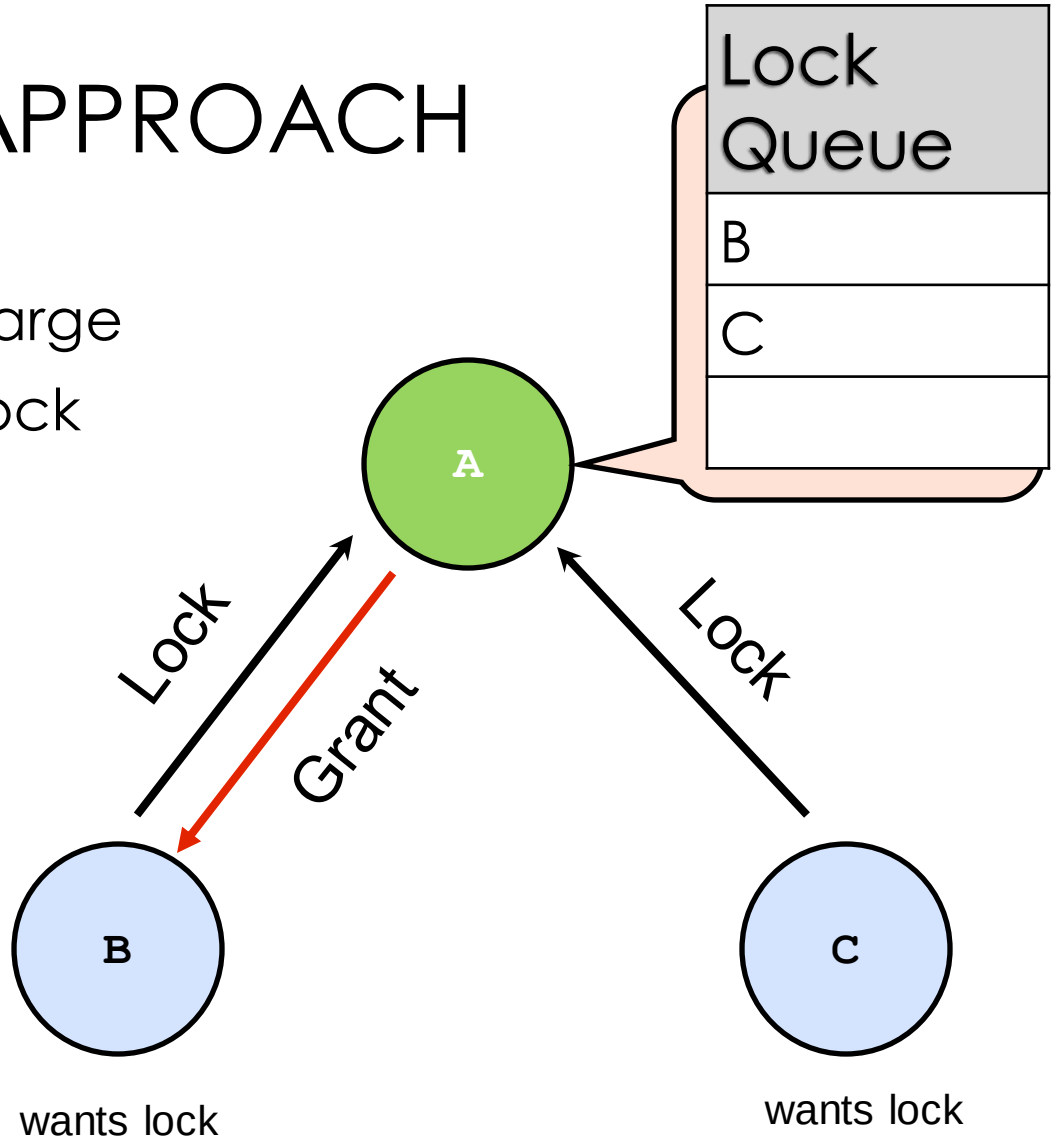
Why Lock?

- 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?



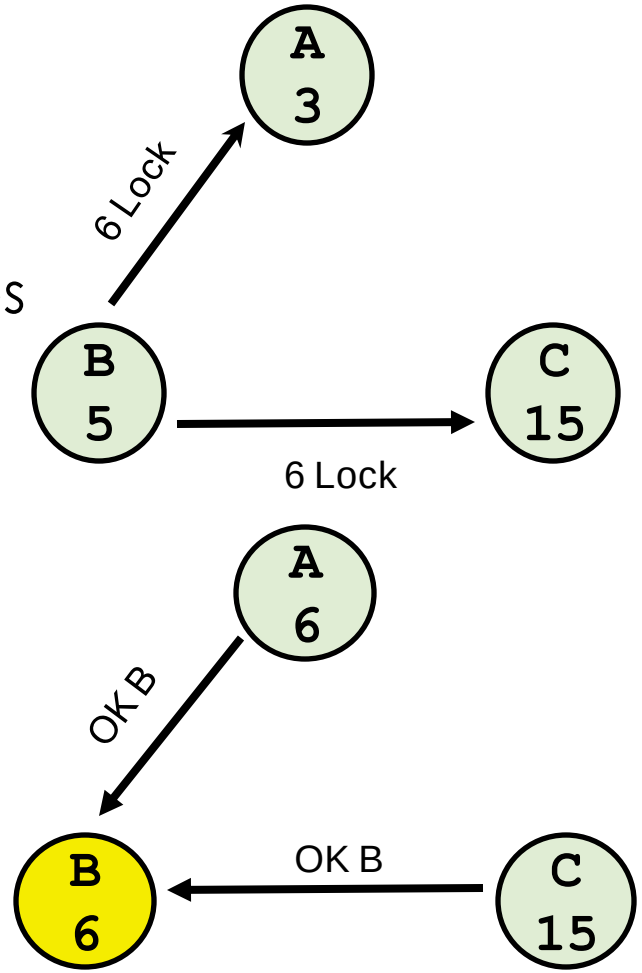
Centralized Approach

- 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?



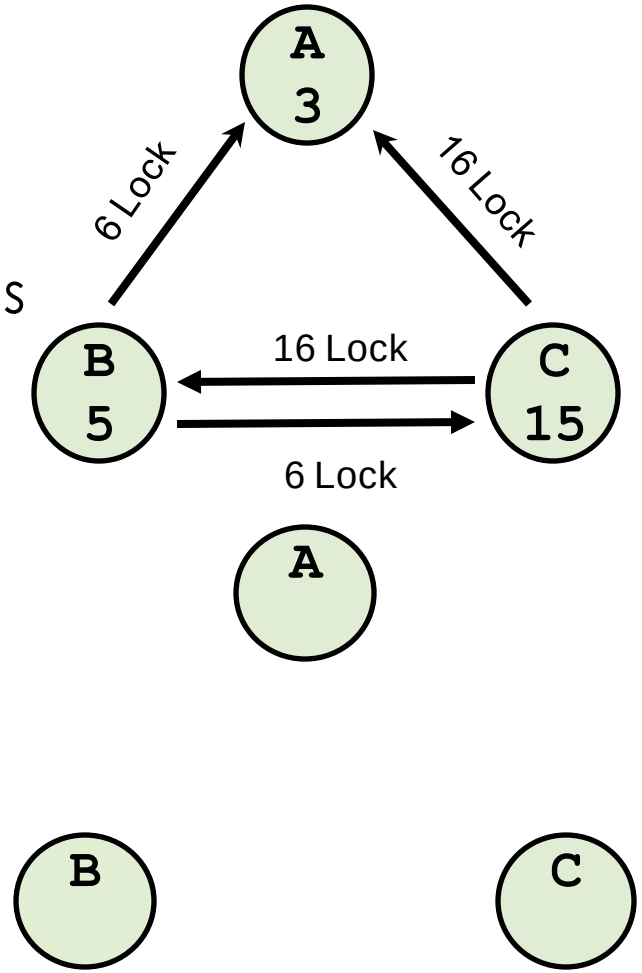
Distributed Approach

- 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



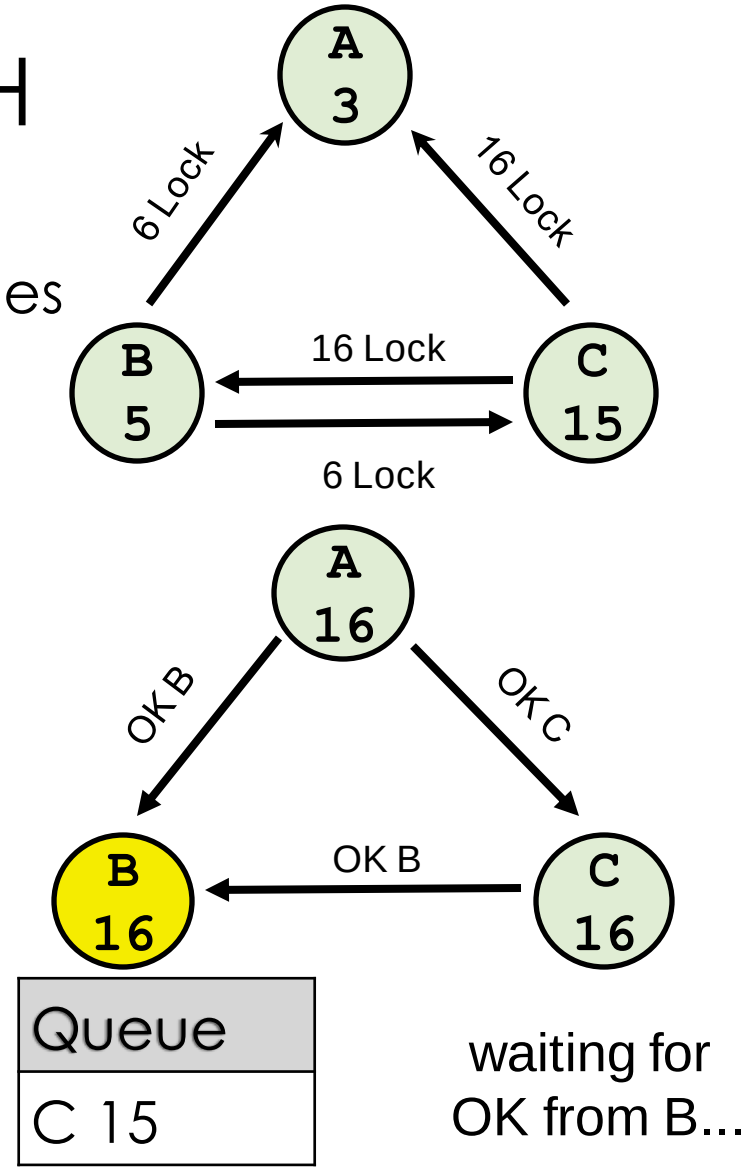
Distributed Approach

- 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!

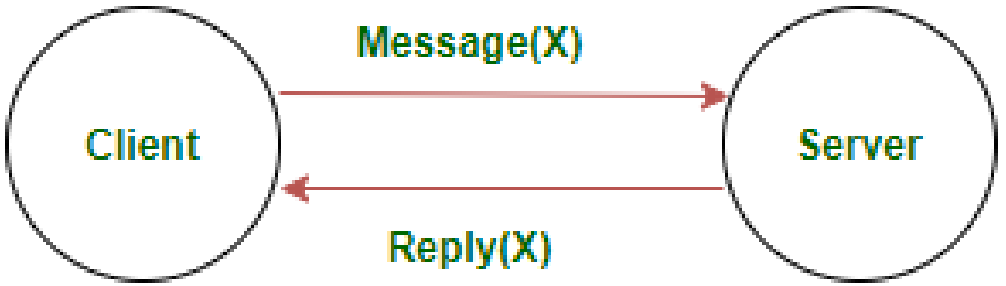
What is the meaning of Consensus

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

Why Consensus?

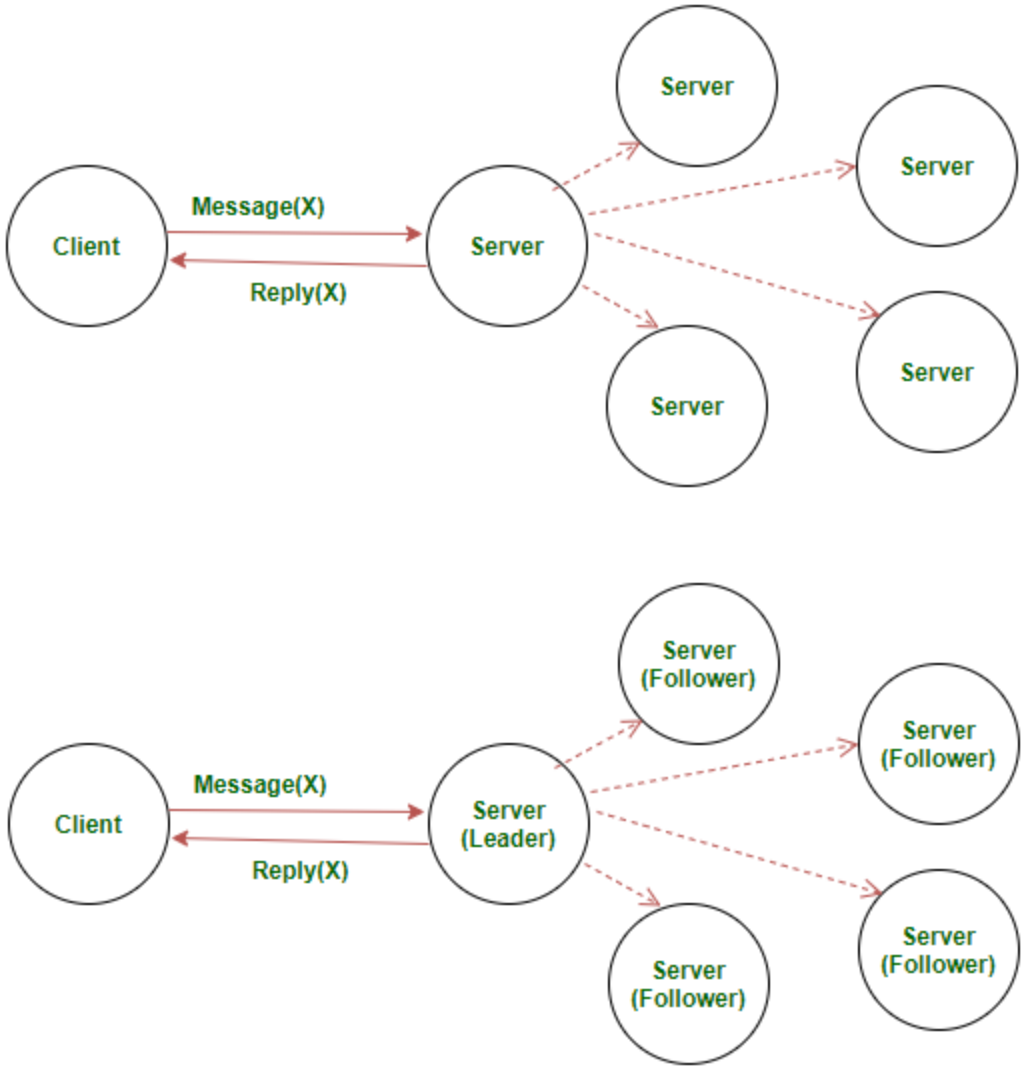
When you sent a request to a server it answers you easily

- 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
- 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.

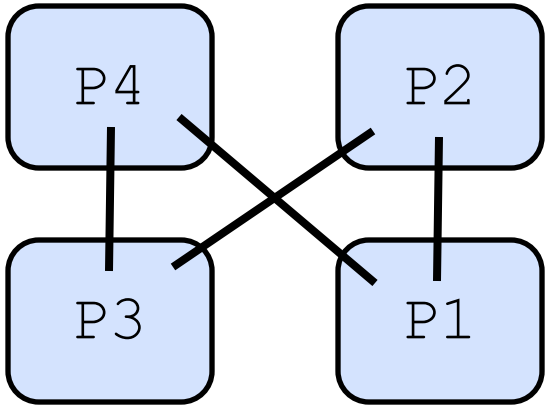
- How are any decisions made?
- How does anything get done?

Consensus Objectives

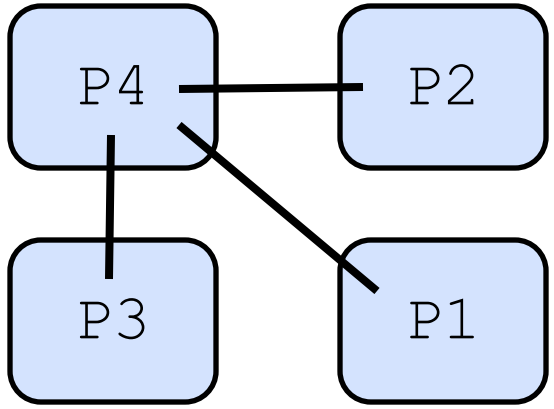
- Therefore, objectives of a consensus mechanism are:
  - **Agreement seeking**: A consensus mechanism should bring about as much agreement from the group as possible.
  - **Collaborative**: All the participants should aim to work together to achieve a result that puts the best interest of the group 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

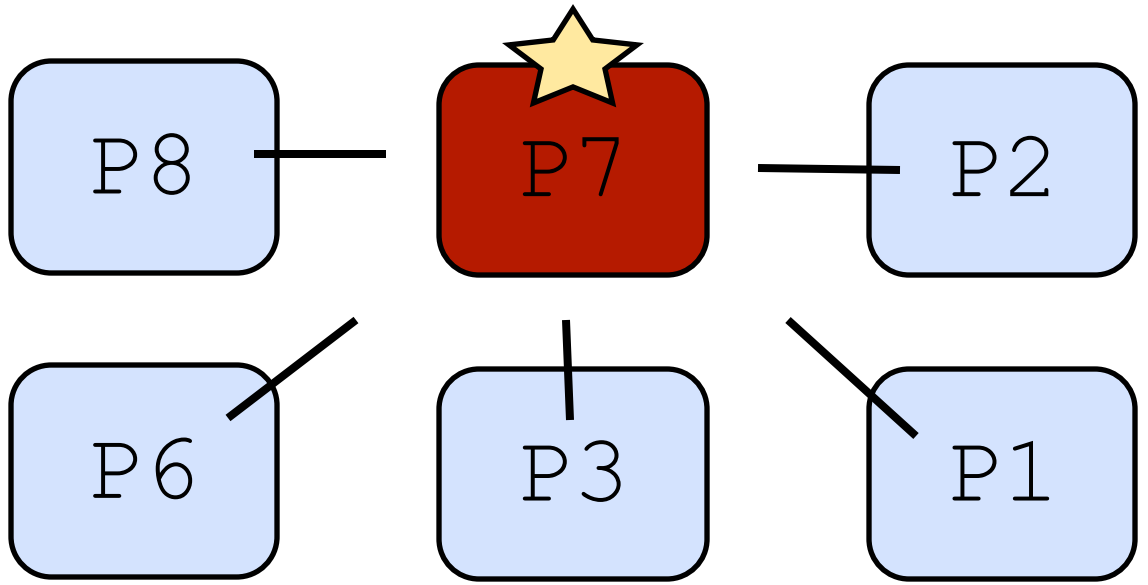


Centralized

- Can we mix the two?

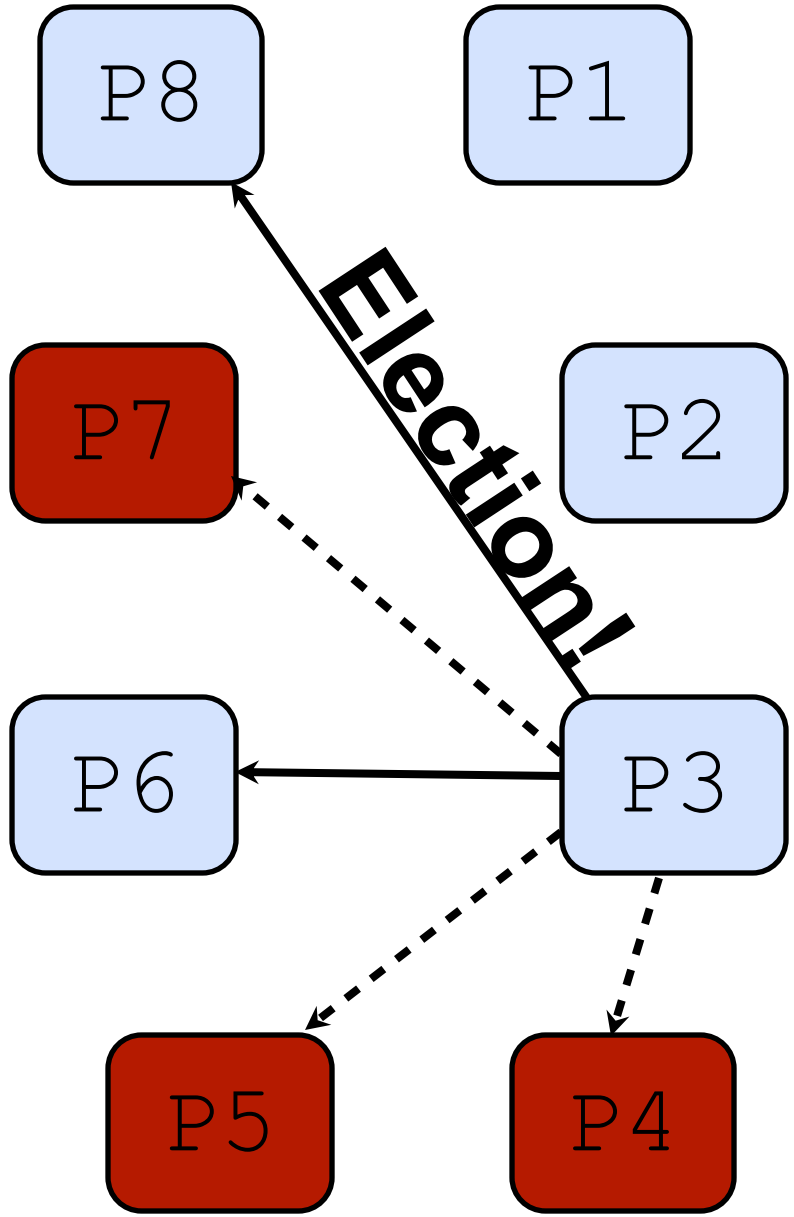
Elections

- 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?



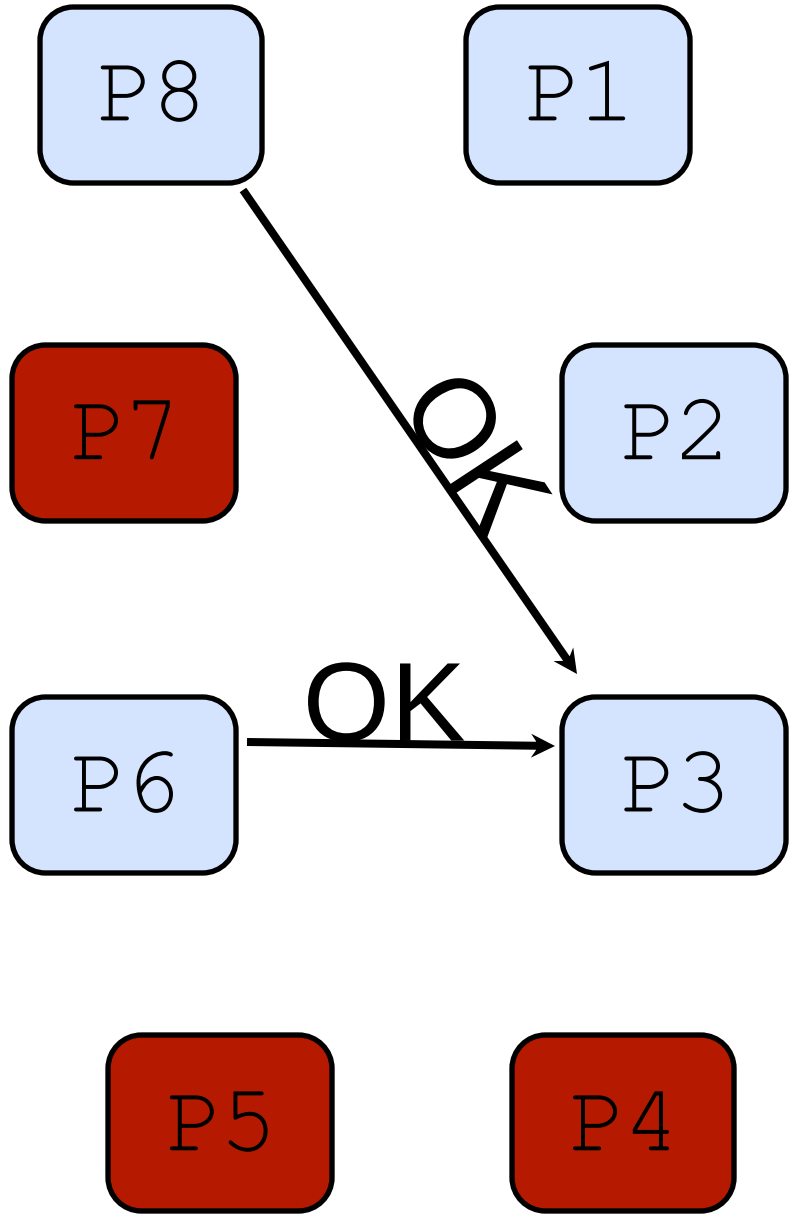
Bully Algorithm

- 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**...



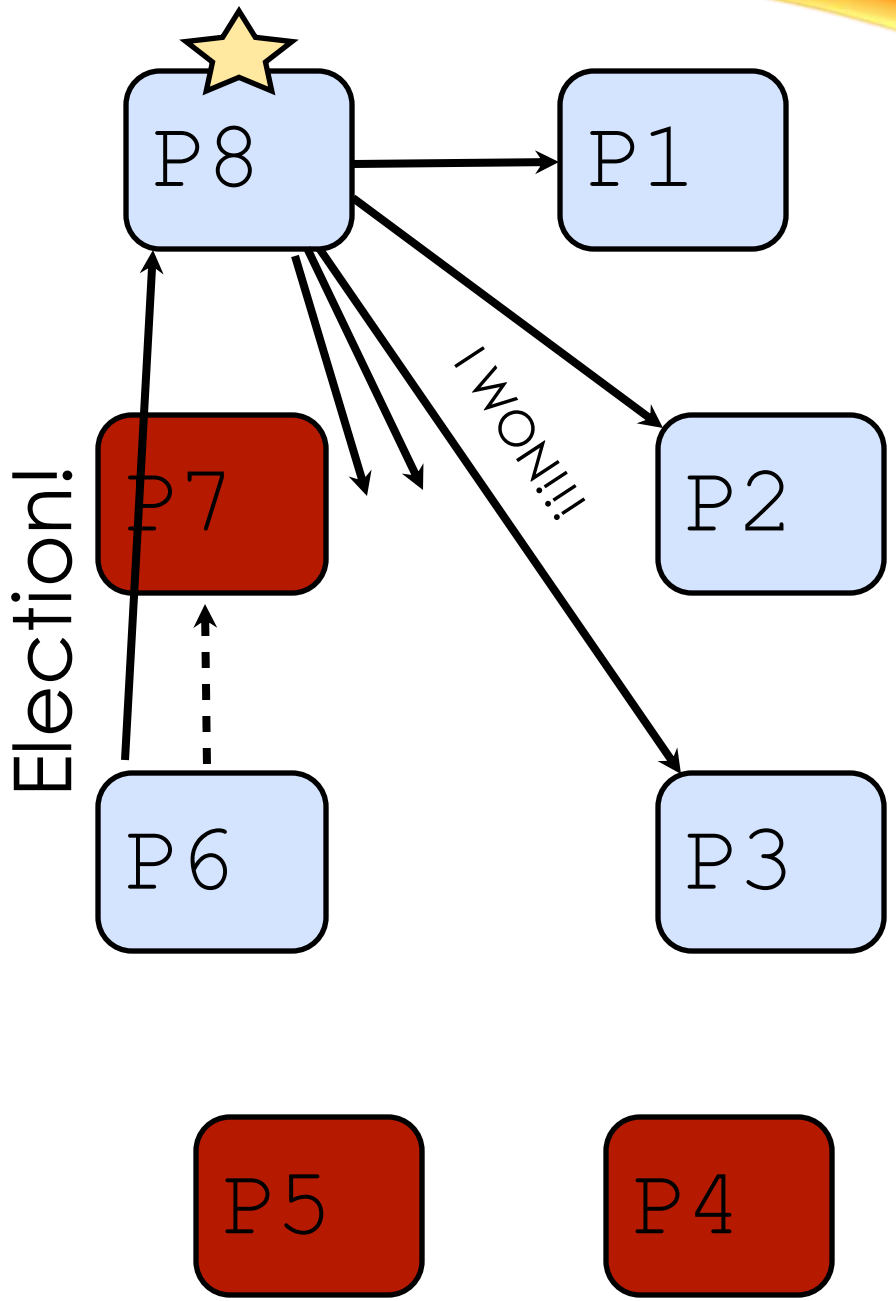
Bully Algorithm

- 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

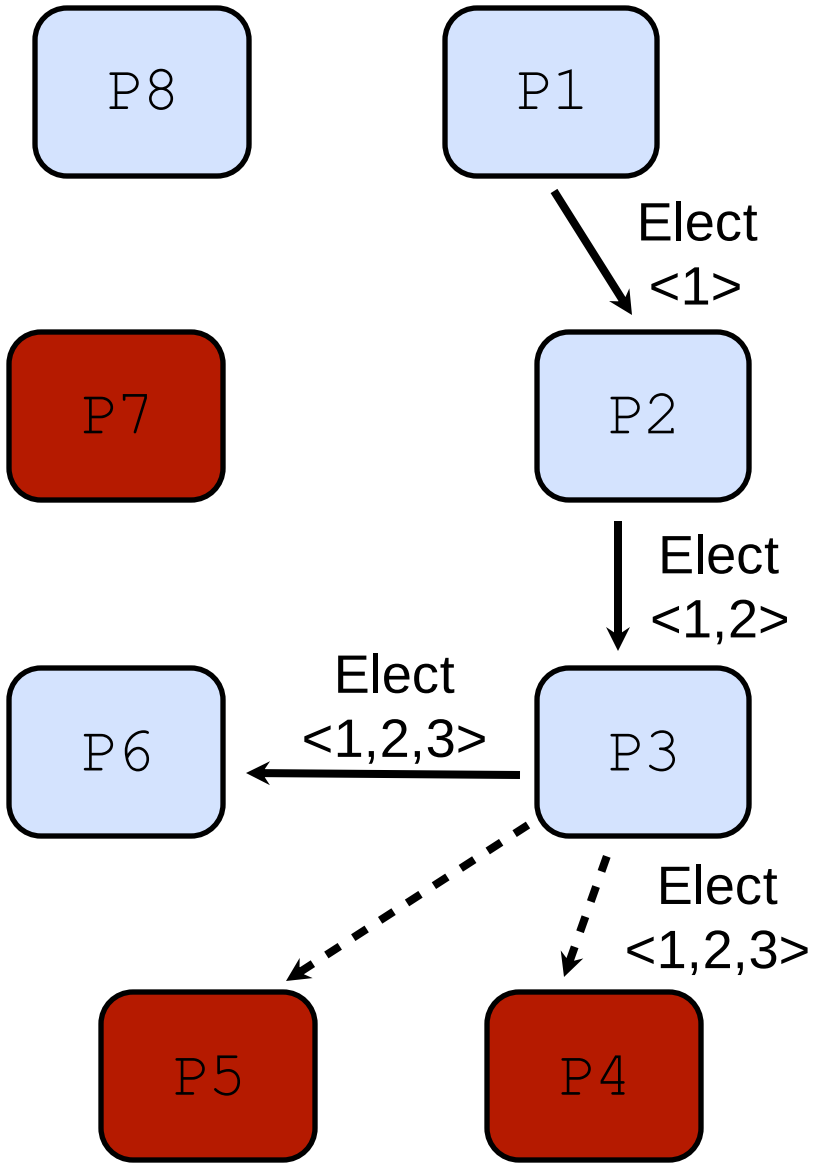
- The biggest (ID) wins
- Any process P can initiate an election
- P sends **Election** messages to all process with higher Ids and awaits **OK** messages
- If it receives an OK, it drops out and waits for an **I won**
- If a process receives an **Election** msg, it returns an **OK** and starts another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

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

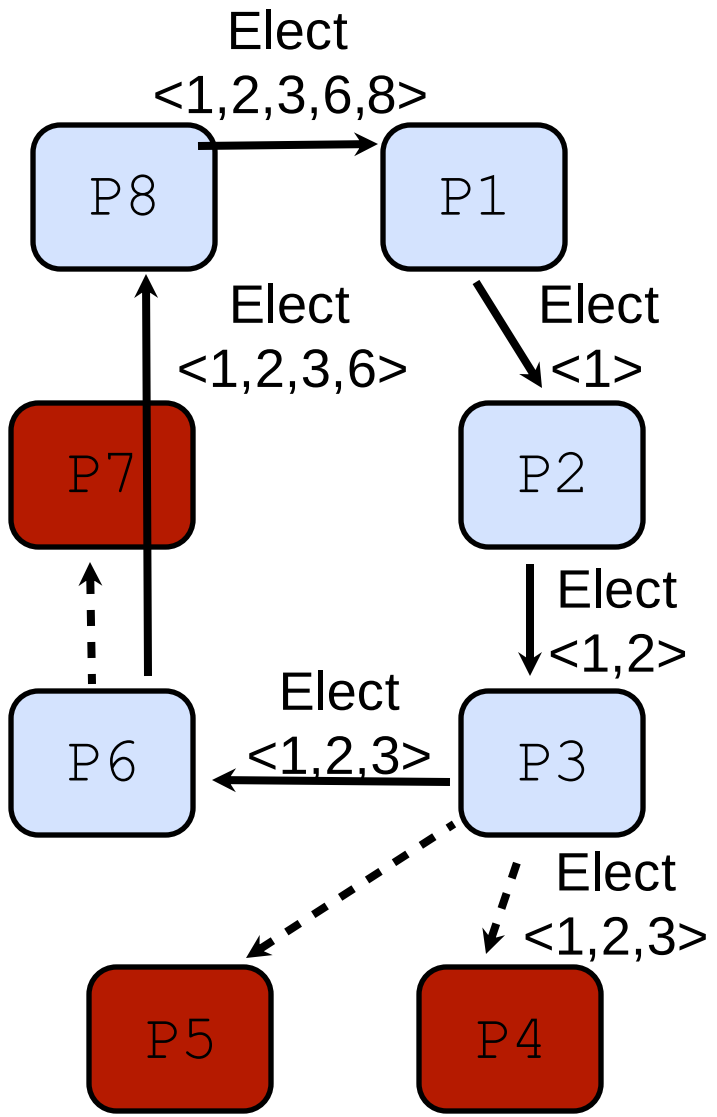
- What happens if multiple elections occur at the same time?



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

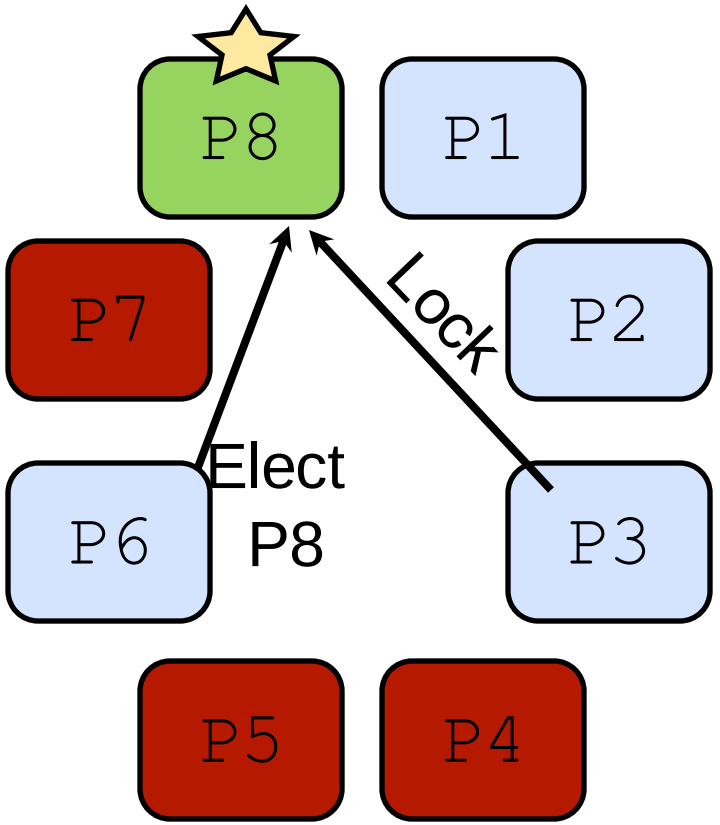
- Ring Algorithm
  - Always 2(n-1) messages
  - Around the ring, then notify all

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

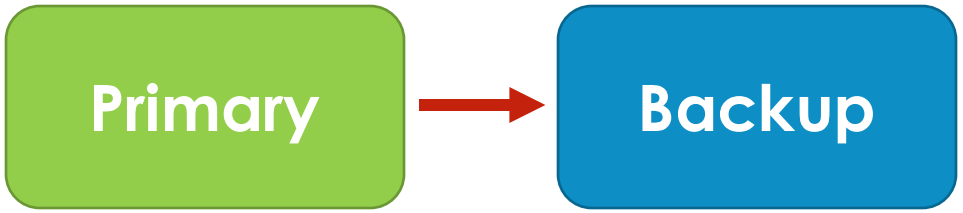
- Google services are composed of many thousands of nodes
- Need a way to coordinate data and access to shared resources!
  - Used by Google File System, BigTable, etc
- Chubby: lock service for loosely coupled distributed systems
  - Focuses on availability and reliability (not performance)
  - Scales 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"

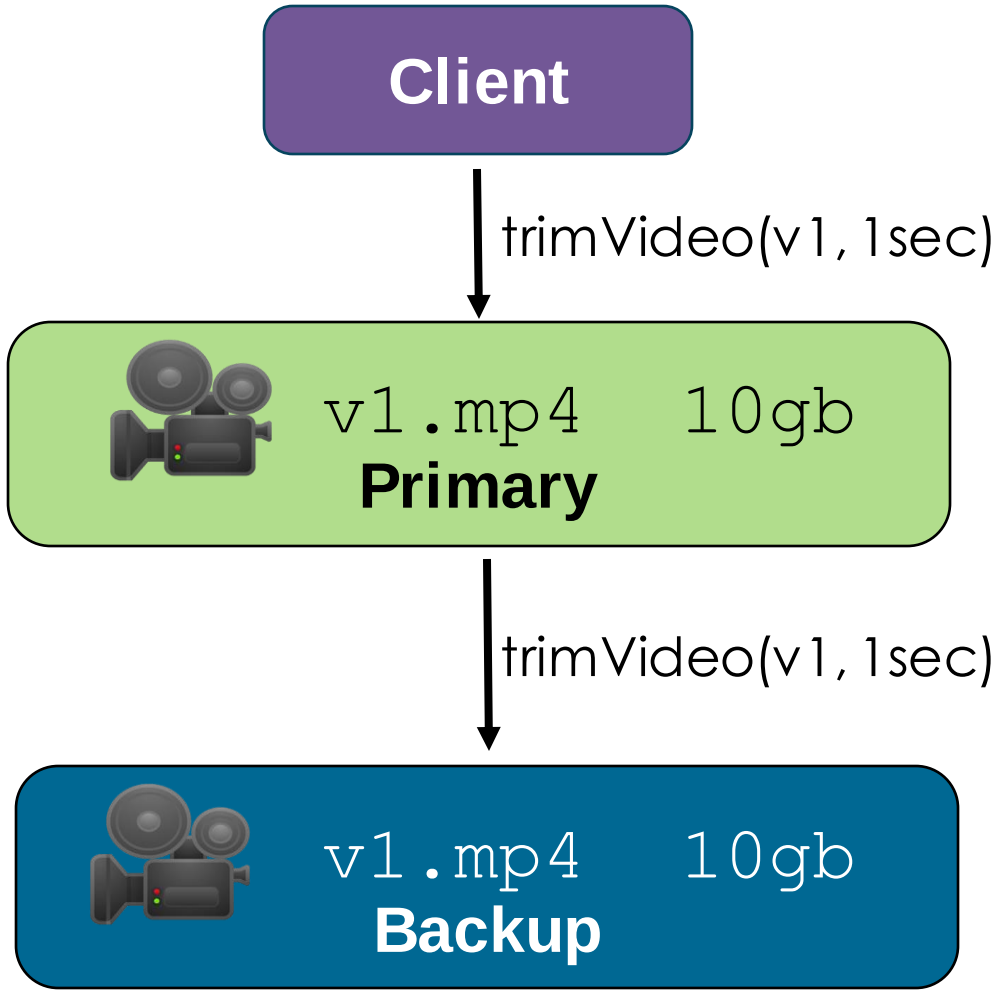


Distributed Video Editing SMR

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

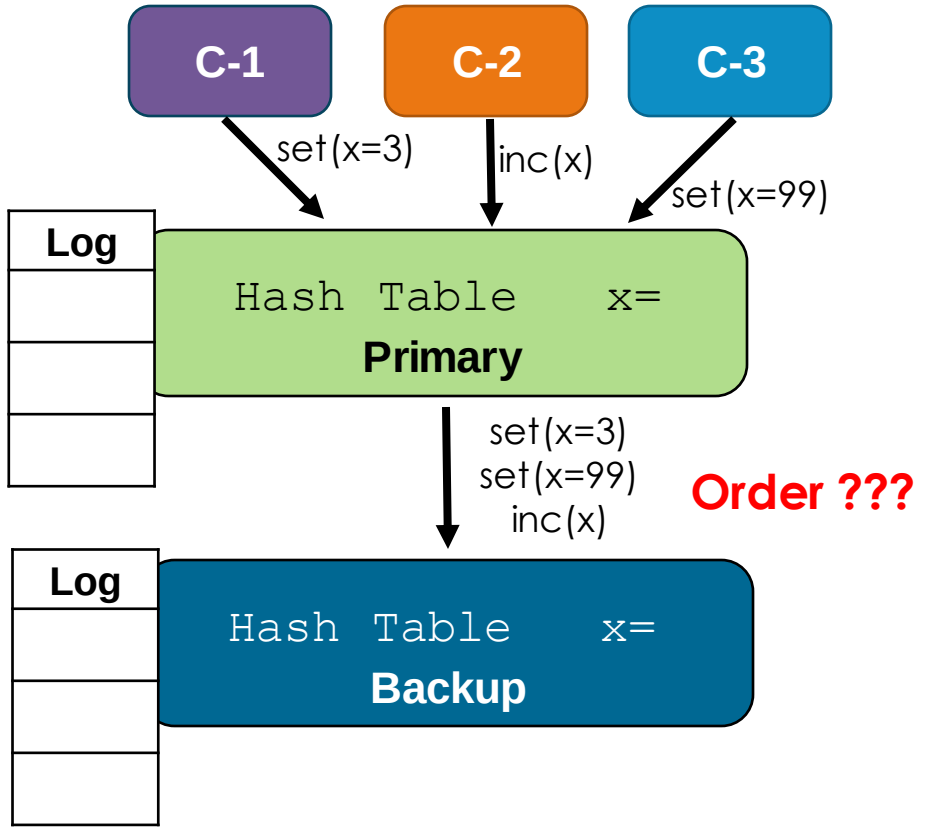
- Treat like a state machine
  - Incoming requests just perform some operation on that data
  - If all replicas perform same operations, they will end in the same state

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



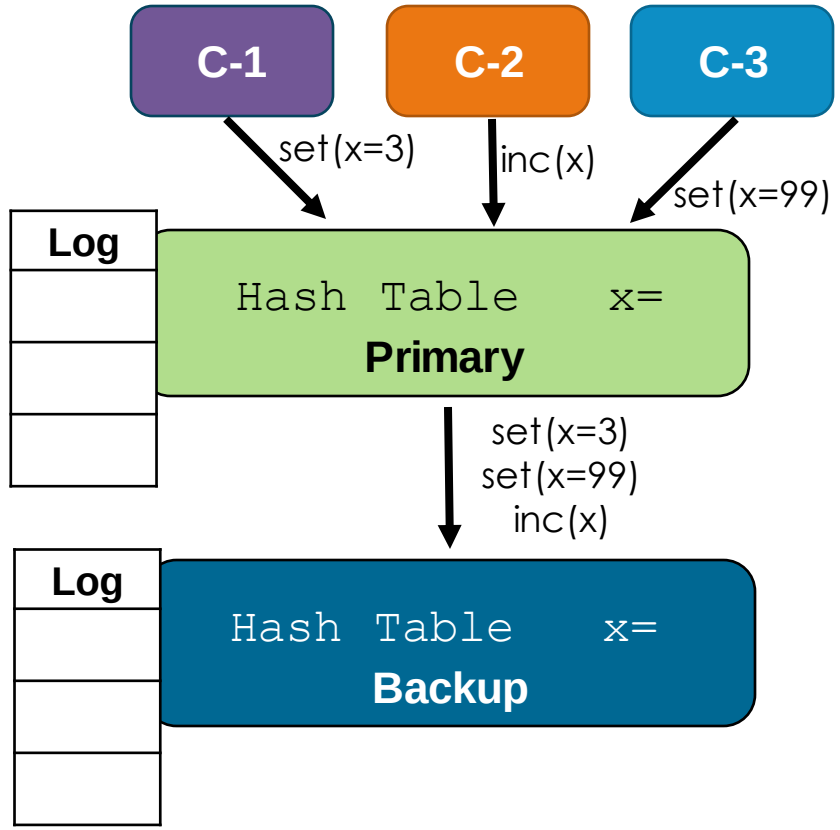
Hash Table SMR

- 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?

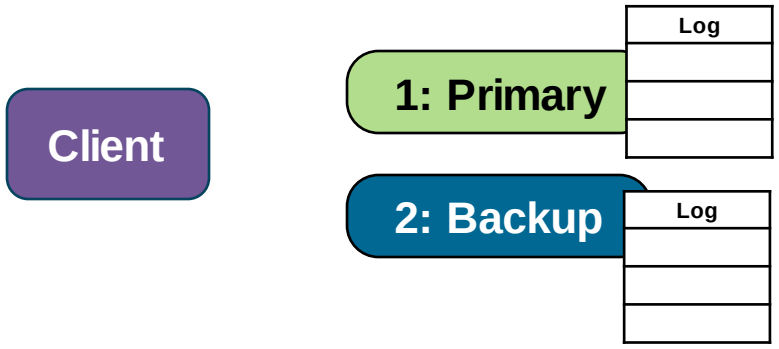
- What to do on a failure?
- How many failures can we handle?



Handling Failures

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

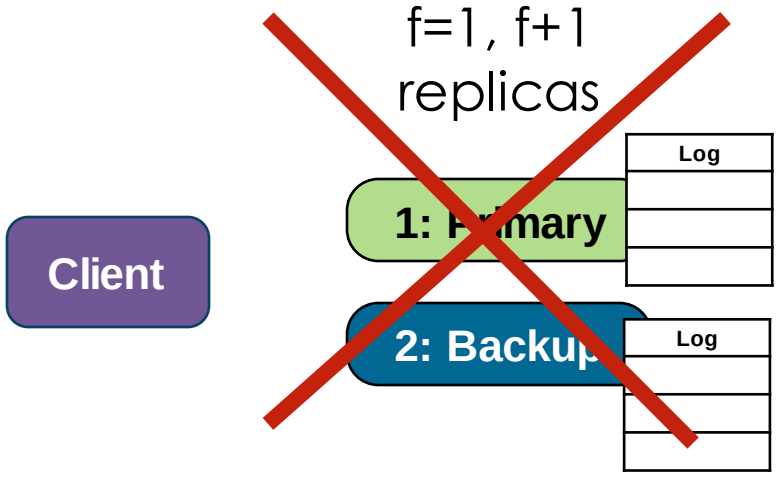
f=1, f+1=2
replicas



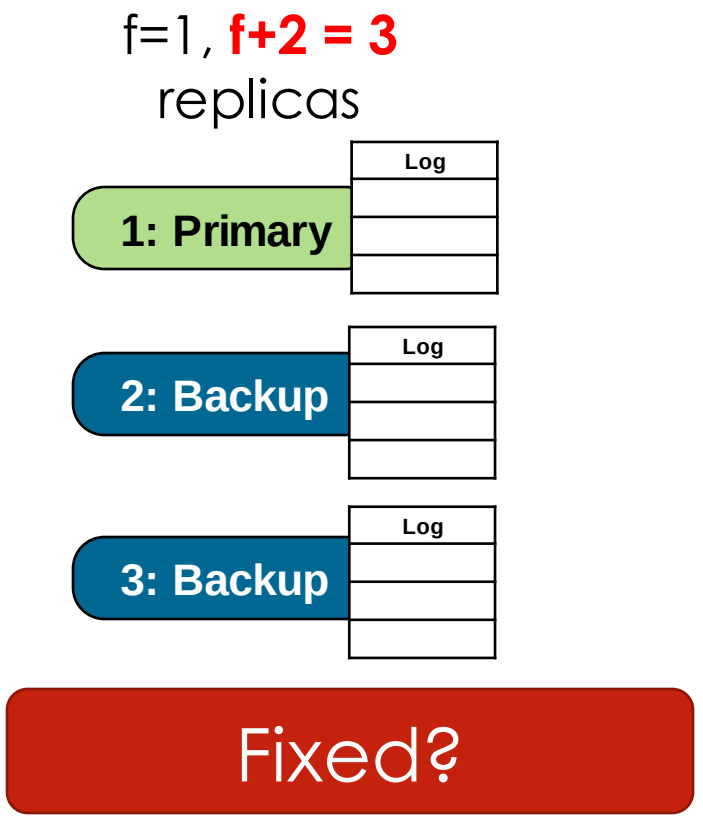
What failure scenarios can happen?

Handling Failures

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



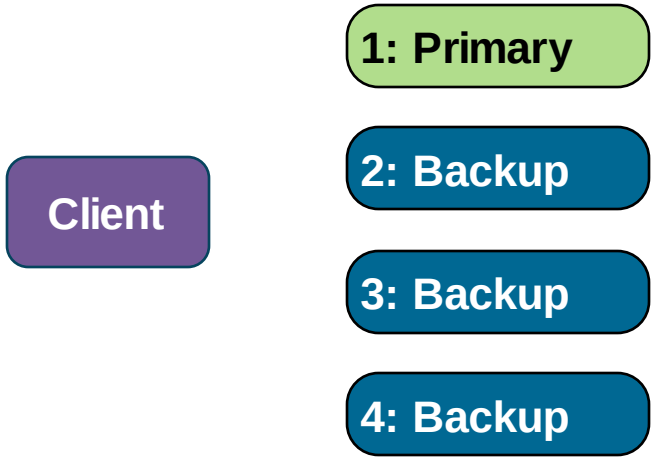
Can't resync state if failure "flip flops" between nodes!



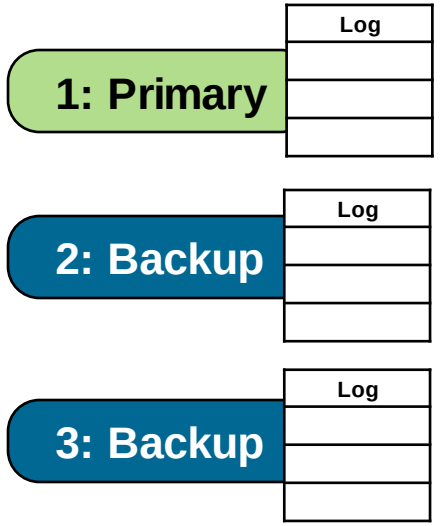
Handling Failures

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

f=2, f+2 = 4 replicas



f=1, f+2 = 3 replicas

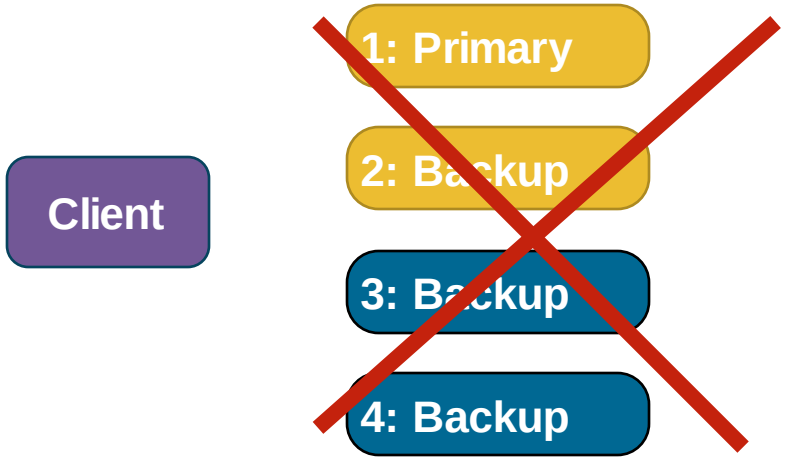


Fixed for f=2?

Handling Failures

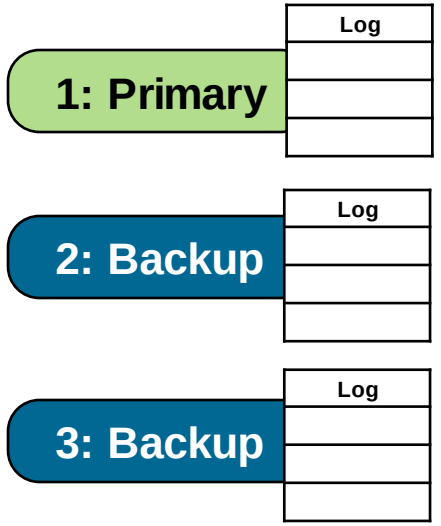
- F = number of nodes which can crash at one time
- # of nodes needed must depend on f!

f=2, f+2 = 4 replicas



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

f=1, f+2 = 3 replicas

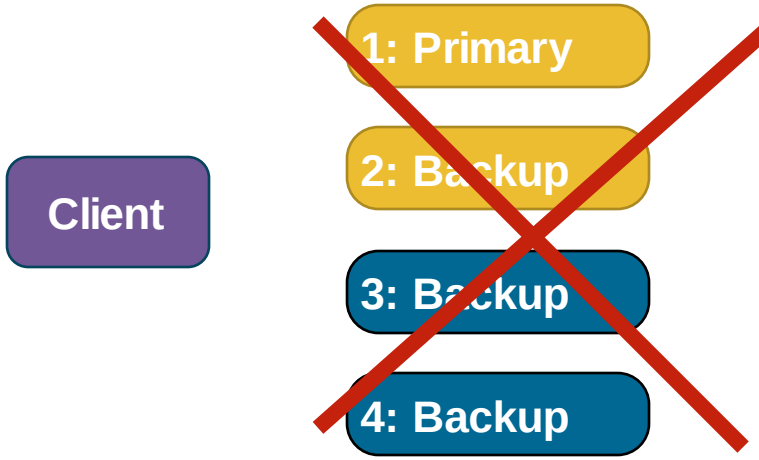


Fixed for f=2? No!

Handling Failures

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

f=2, f+2 replicas



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!

f=2, 2f+1 = 5 replicas

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

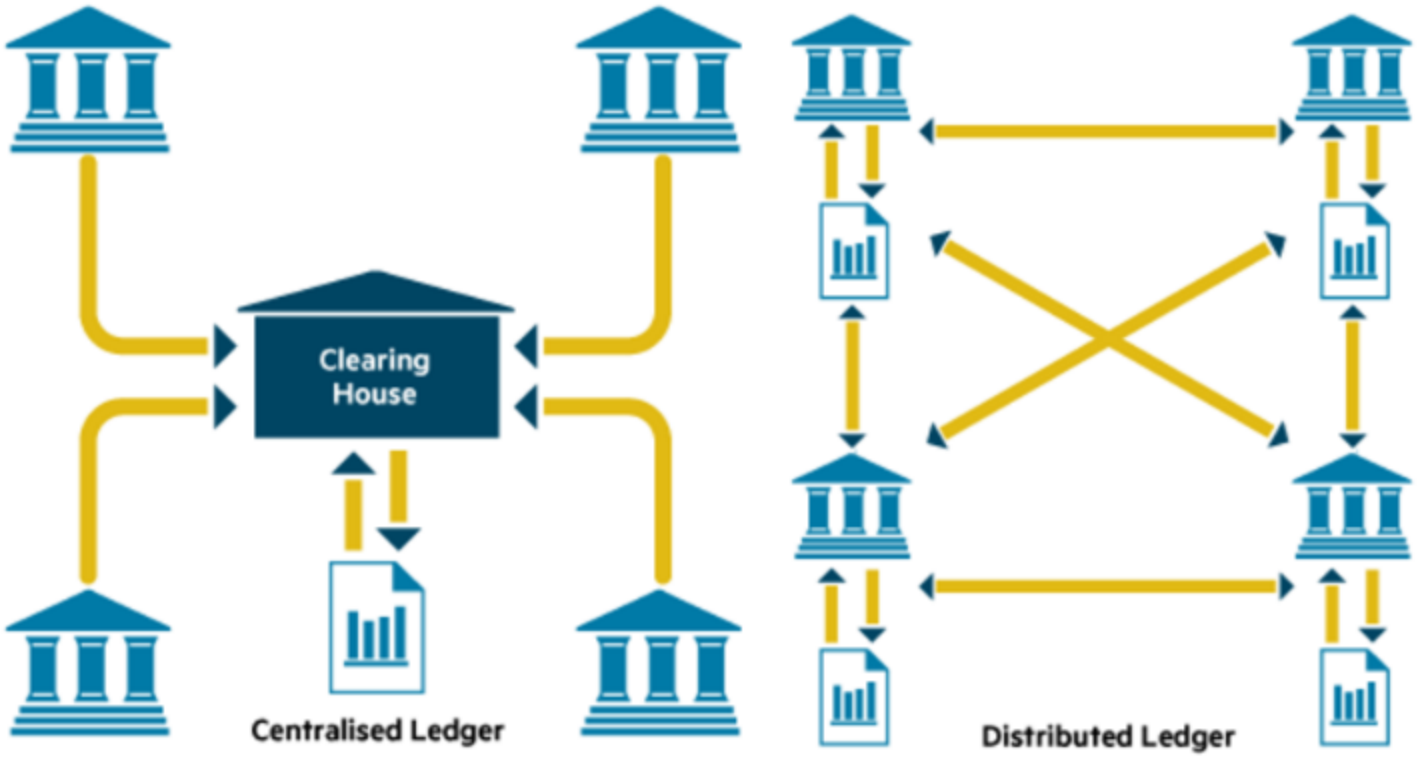
- But how do we pick who should be primary…?
  - Use an election algorithm!

Optional : Example for your project

Case Study

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

Distributed Ledger Tech



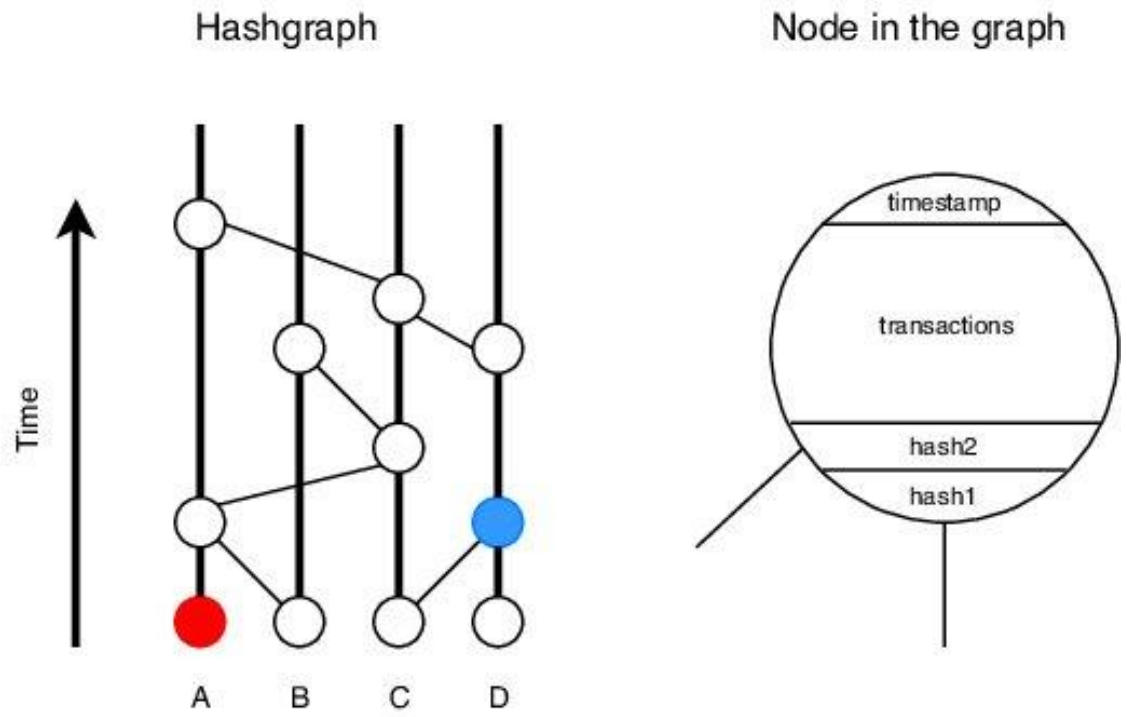
Different types of DLT

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

- https://www.researchgate.net/publication/328475892_A_Review_of_Distributed_Ledger_Technologies/figures?lo=1

HASHGRAPH

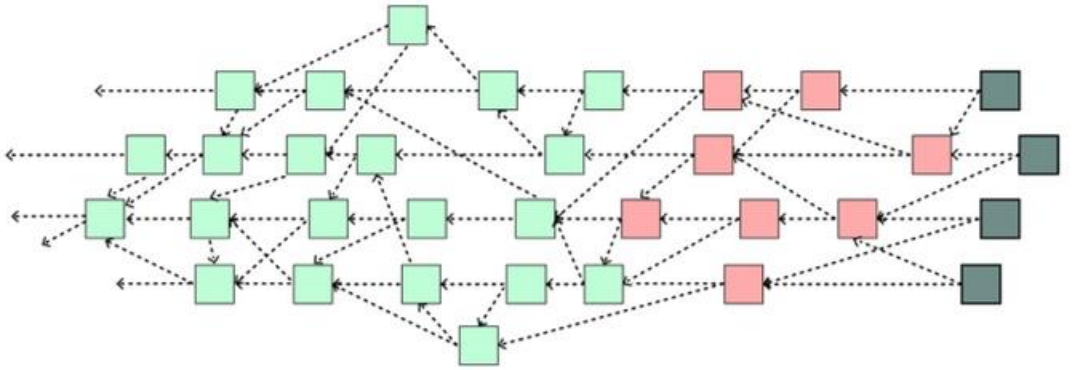
- It's so fast – 250000 transaction per second (Scalability characteristics in Distributed Systems)
- Being Time-Based and using Gossip protocol for consensus reduces the process and math complexity.
- Time-based increases the justice. Each transaction who obtained 2/3 of the votes has the most priority and it can be a weakness
- 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

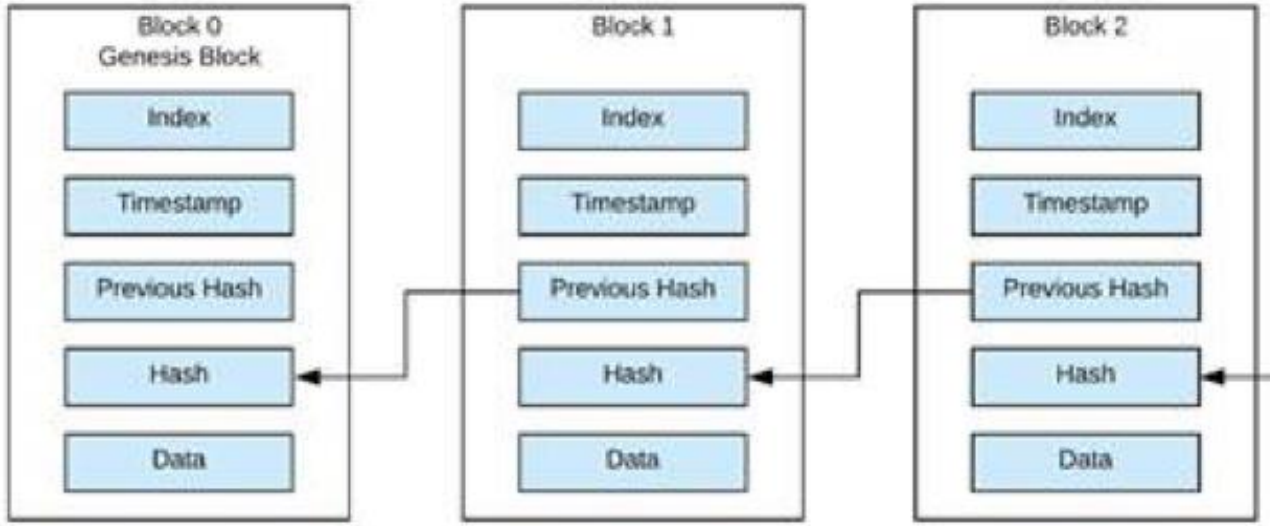
TANGLE (IOTA)

- 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

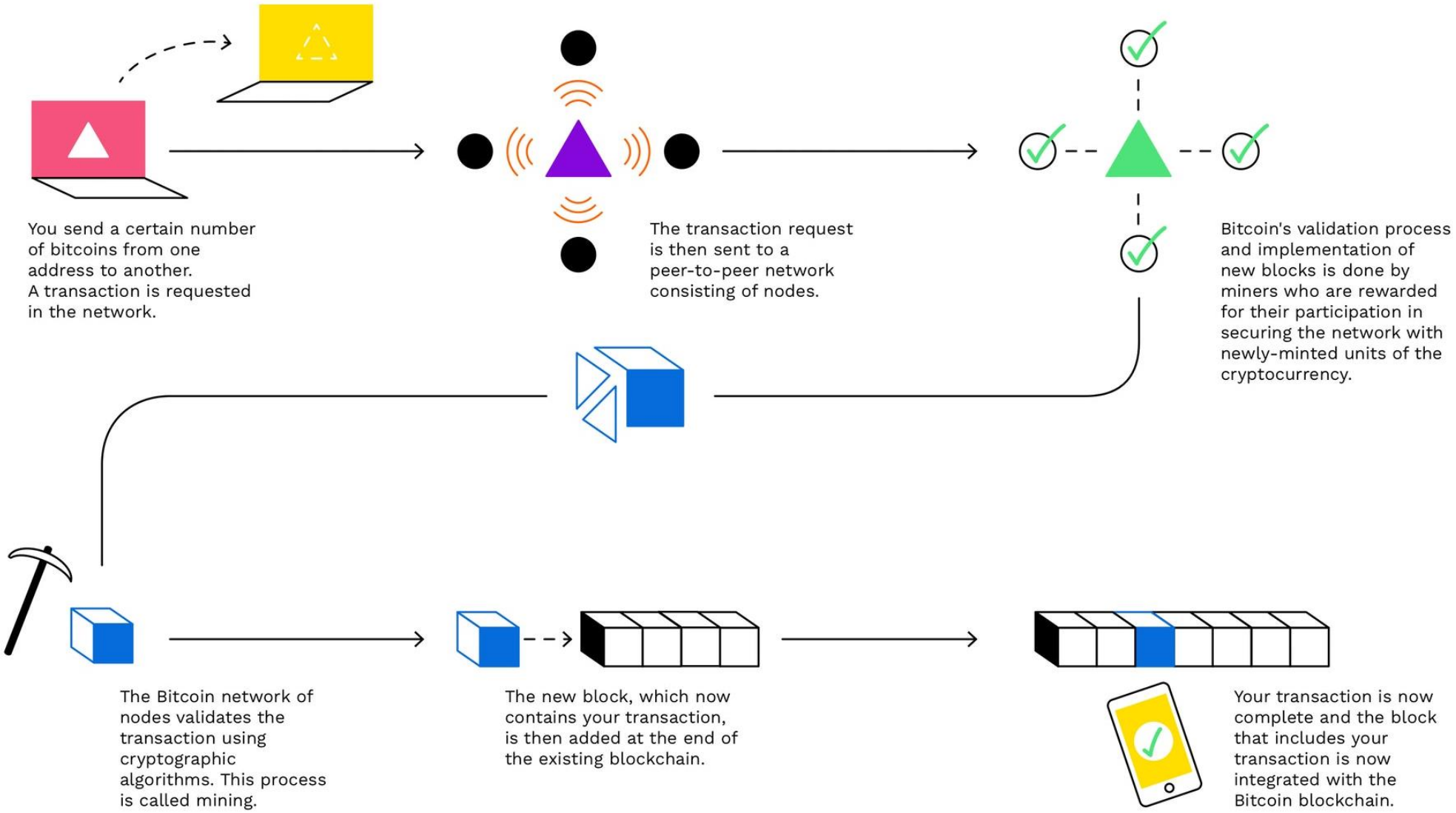


BLOCKCHAIN

- 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



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.