Distributed Systems
CS6421
Timing and Coordination

Prof. Tim Wood and Prof. Roozbeh Haghnazar

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

LAST TIME…

- Advanced Resource Management
  - MapReduce / DevOps
  - Resource Optimization problems
    - NP Hardness
    - Many-objective Optimization
  - Migration
    - Code
    - Processes
    - VMs

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?

Final Project

Questions?

- Groups of 3-4 students
- **Research-focused**: Reimplement or extend a research paper
- **Implementation-focused**: Implement a simplified version of a real distributed system

- Course website has sample ideas
  - But don't feel limited by them!
  - You don't have to use go!

- Timeline
  - Milestone 0: Form a Team - 10/12
  - Milestone 1: Select a Topic - 10/19
  - **Milestone 2: Literature Survey - 10/29**
  - Milestone 3: Design Document - 11/5
  - Milestone 4: Final Presentation - 12/14

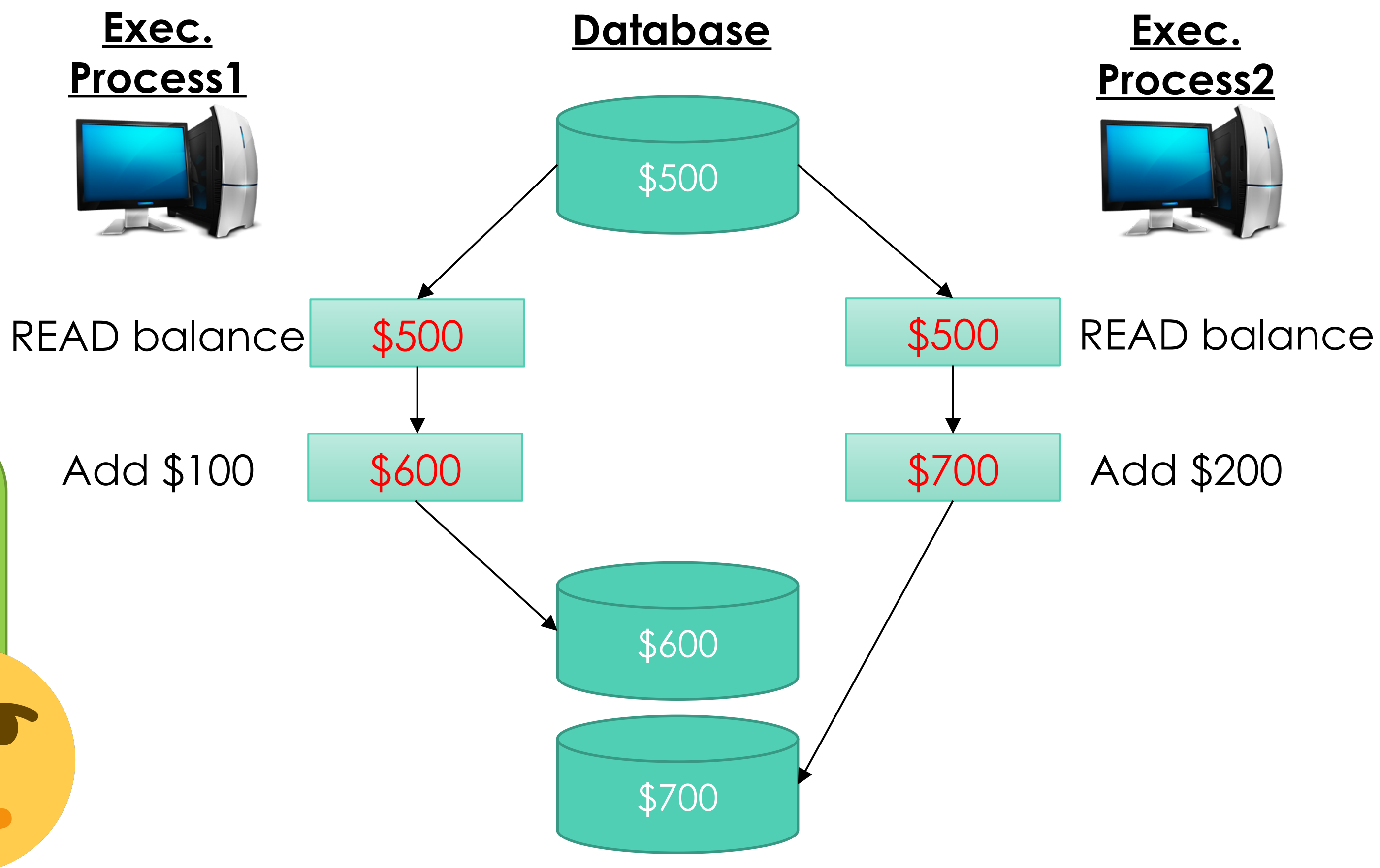
https://gwdistsys20.github.io/project/

CHALLENGES

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

Problem and Challenge Ex.

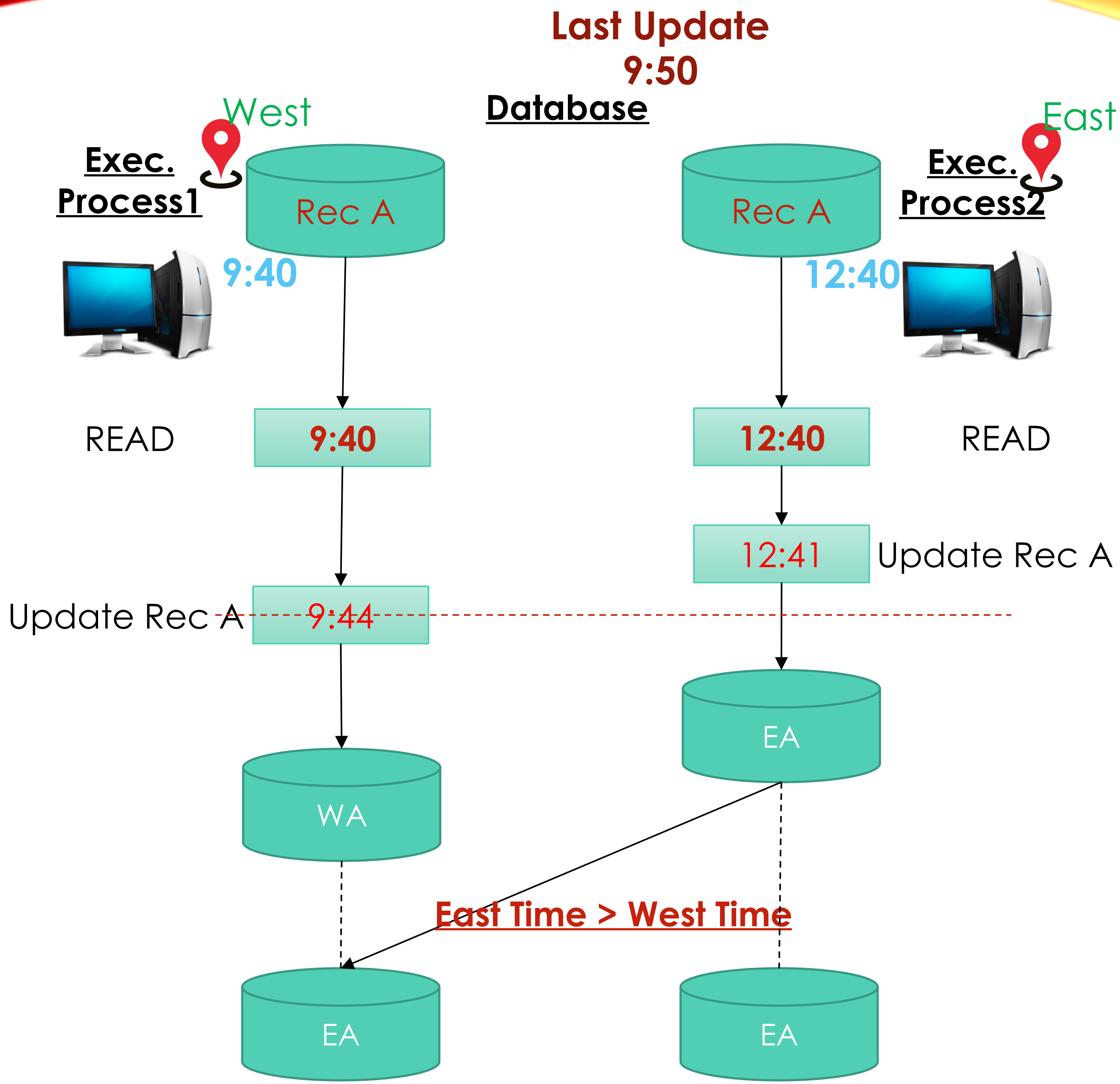
- Concurrency challenge in a database
  - Lost update anomaly



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?

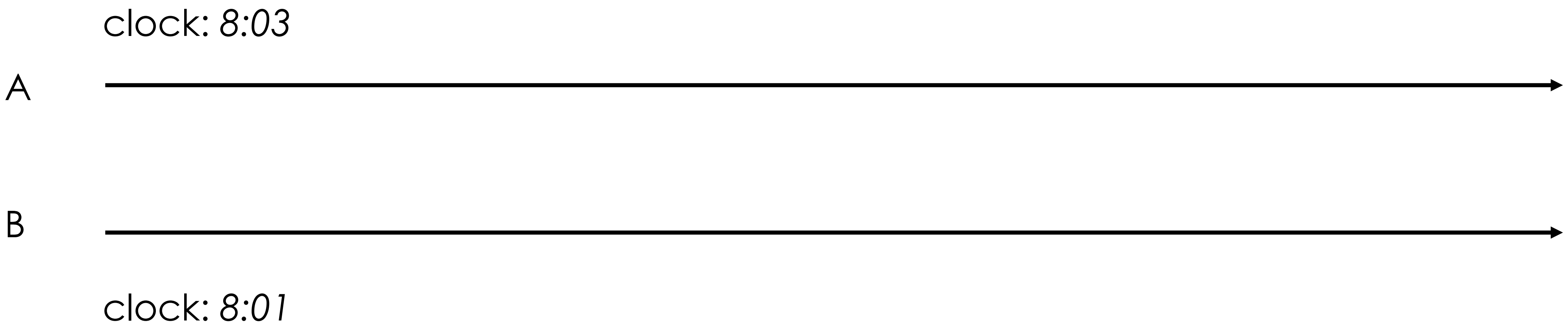
Clocks and Timing

- 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



Physical clocks : Sundial

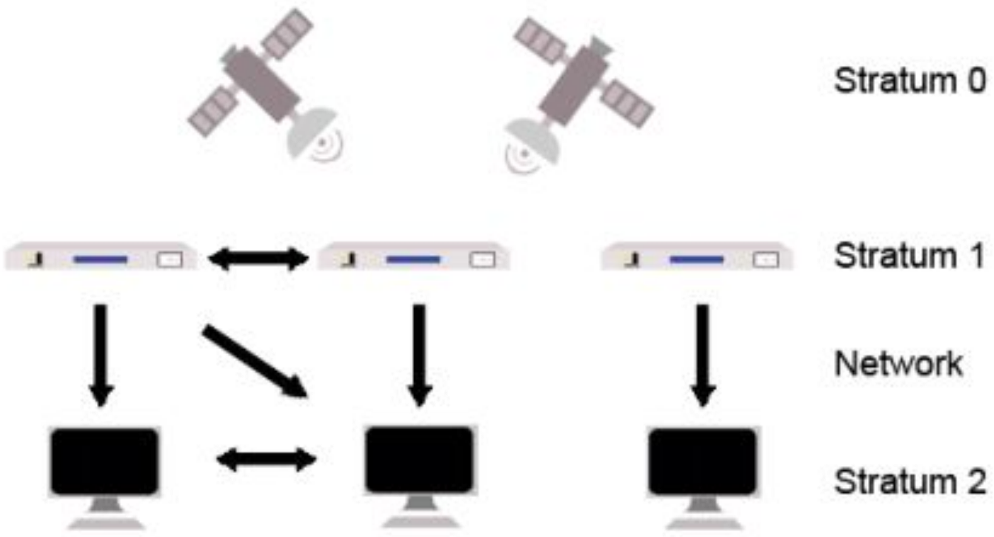
- Sundial.
- Solar Day varies due:
  - Core activities of earth
  - Rise & Tide of oceans
  - Gravity
  - Orbit around the sun, not a perfect circle

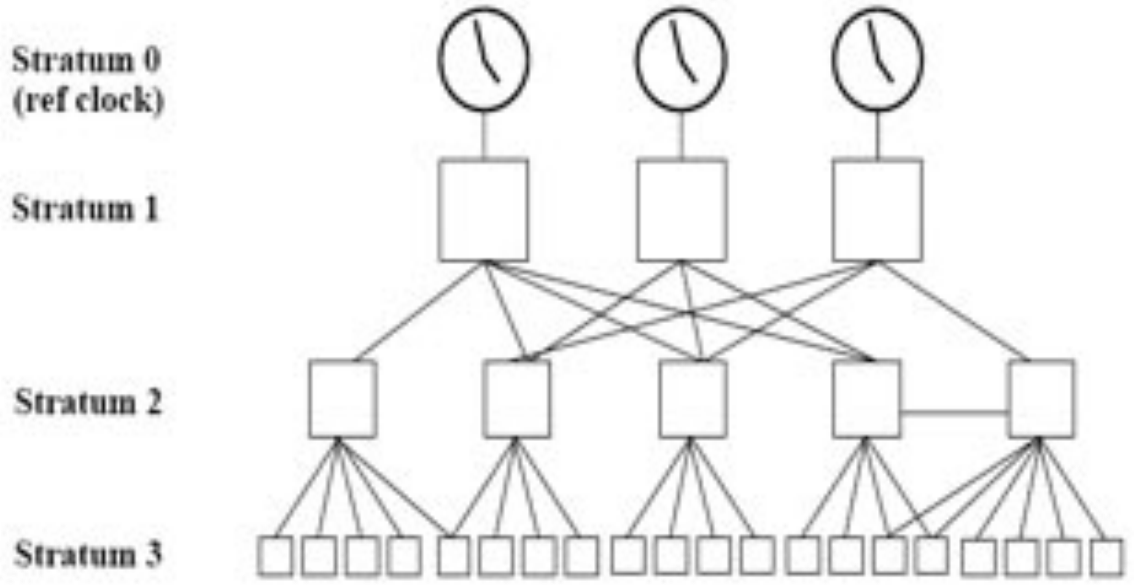
Physical clocks : Atomic Clock

- Accurate time regardless of earth movement and gravity
- Since 1968, the International System of Units (SI) has defined the second as the duration of 9192631770 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

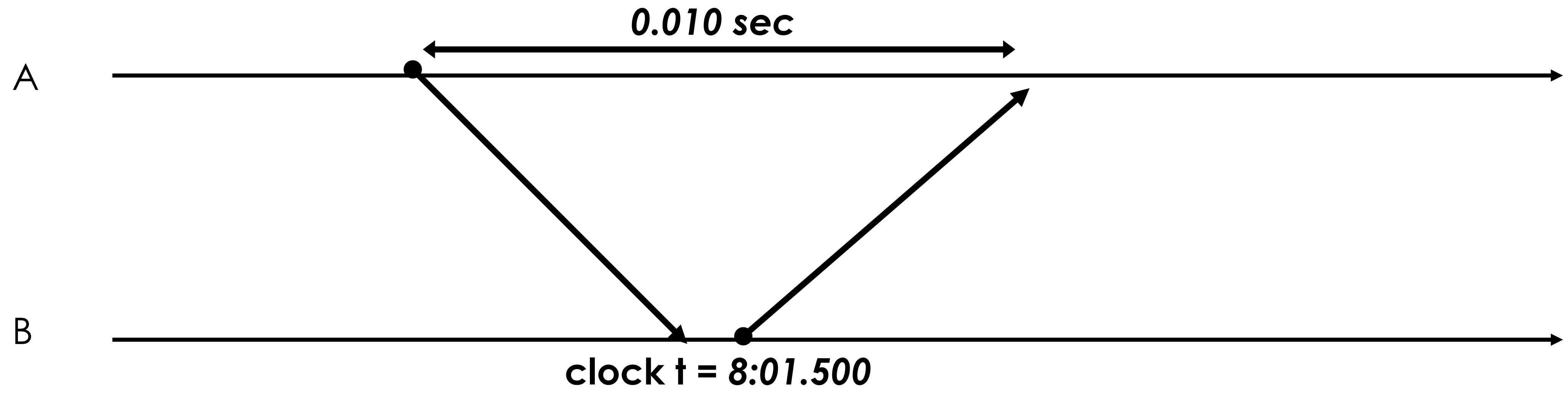
The NTP Stratum model is a representation of the hierarchy of time servers in an NTP network, where the Stratum level (0-15) indicates the device's distance to the reference clock.





CRISTIAN'S ALGORITHM

- Easy way to synchronize clock with a time server

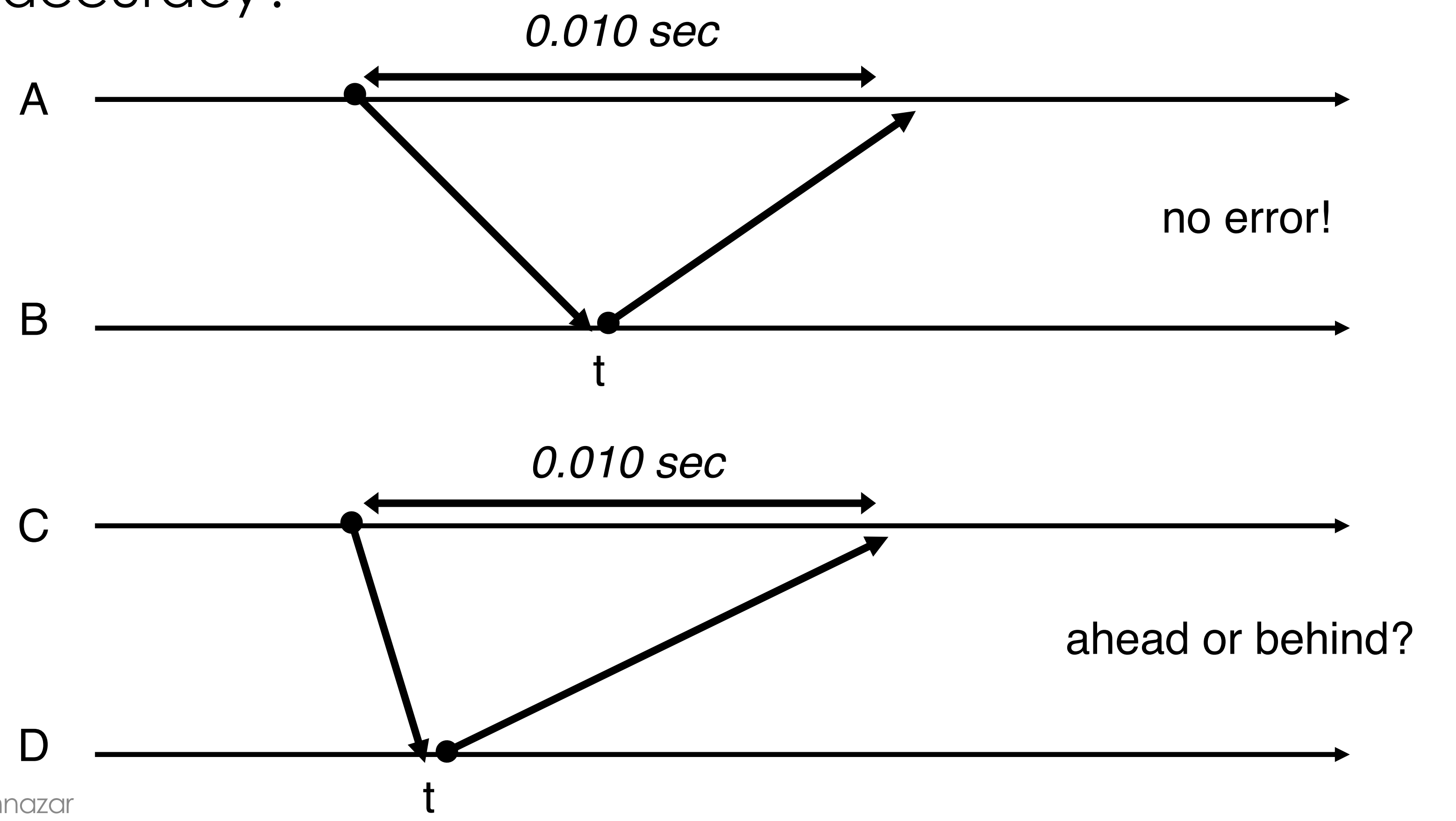


- Client sends a clock request to server
- Measures the round trip time
- Set clock to t + 1/2*RTT   (8:01.505)

CRISTIAN'S ALGORITHM

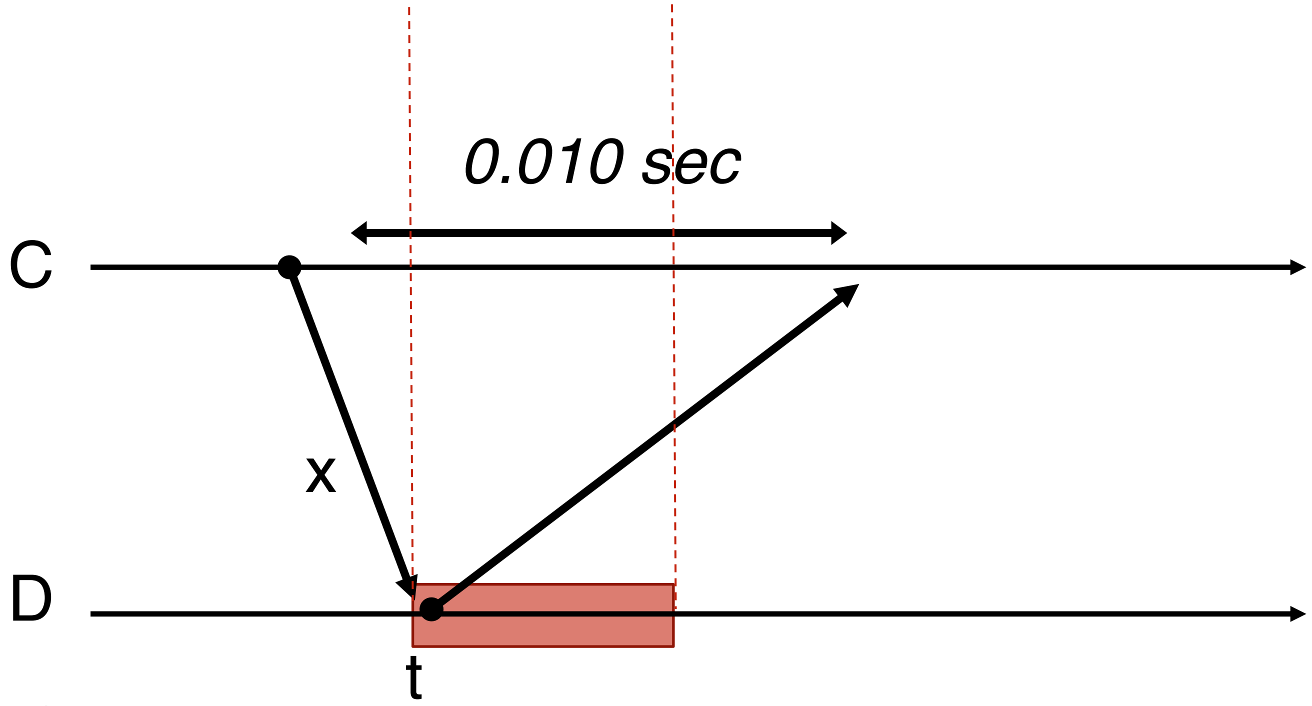
- What will affect accuracy?

set to t + RTT/2



Cristian's Algorithm

- Suppose the minimum delay between A and B is X

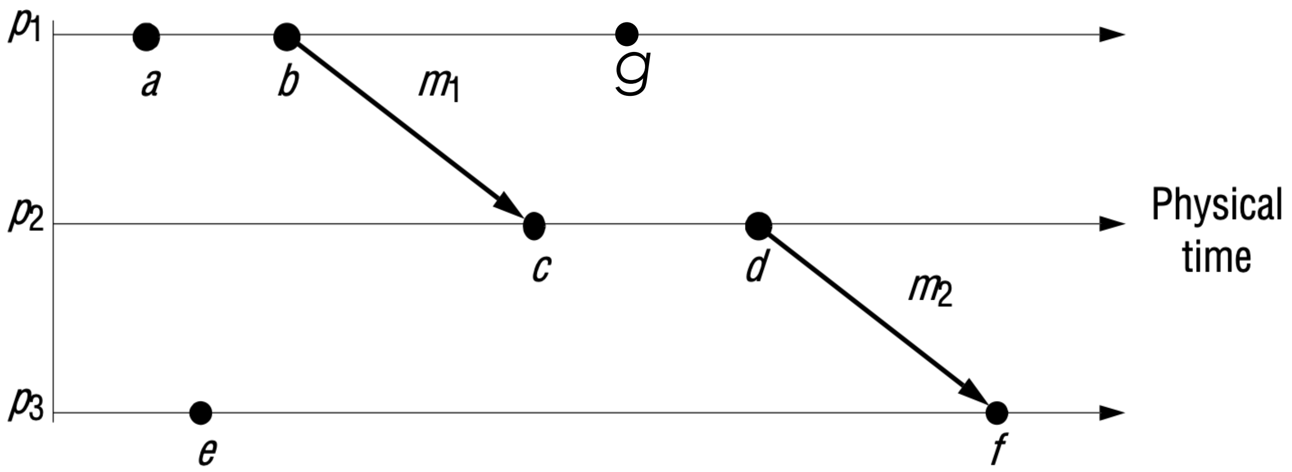


ORDERING

- 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…

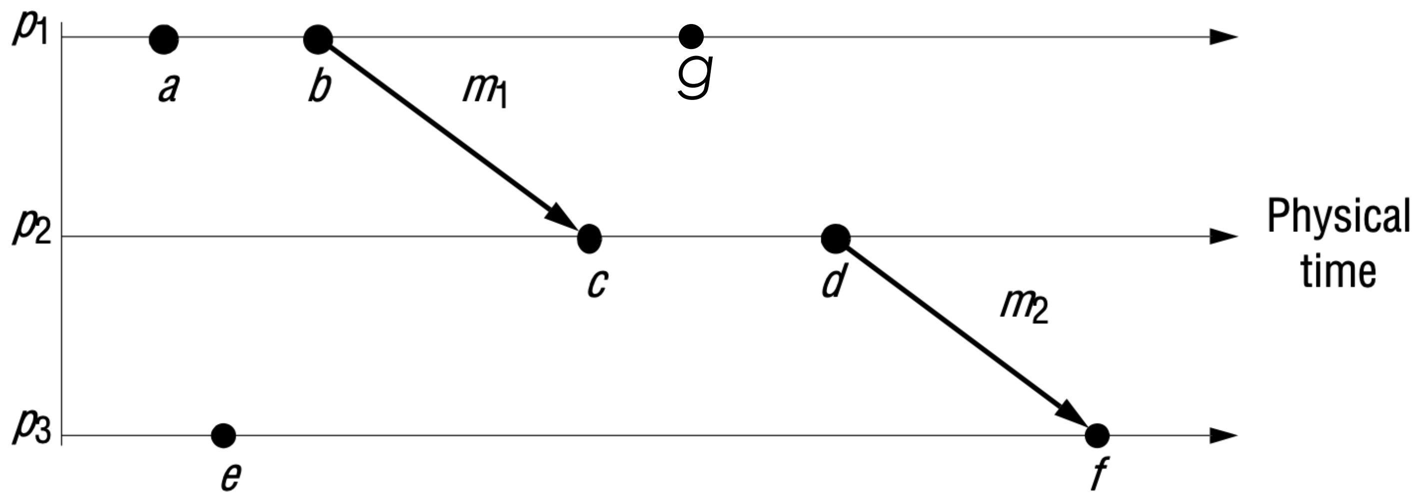
Ordering

- 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?



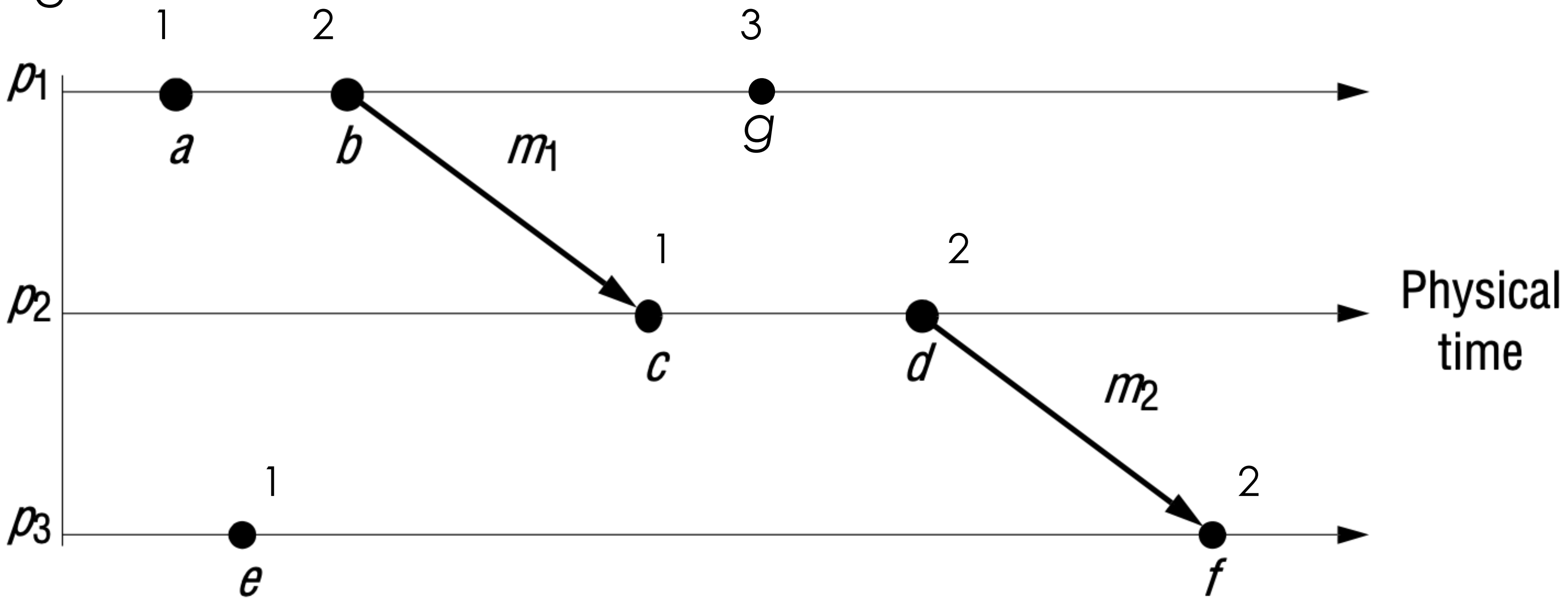
Happens Before: →

- 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



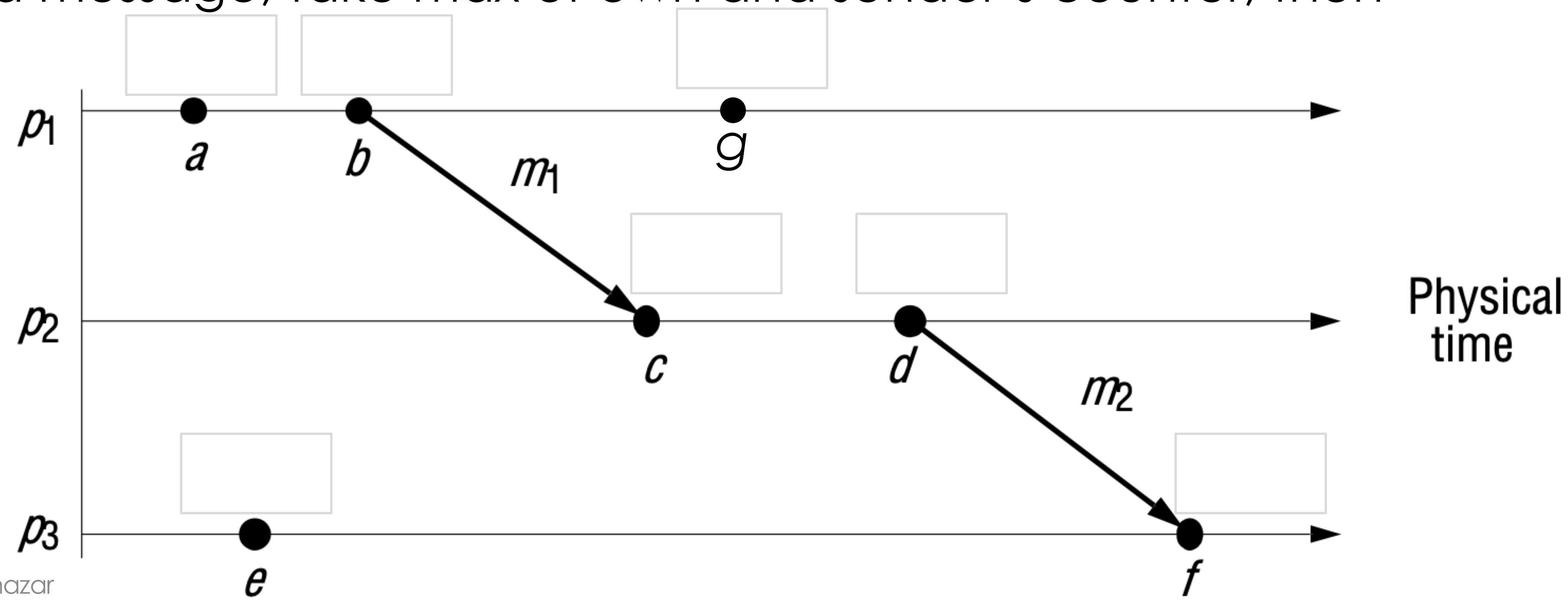
Ordering

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



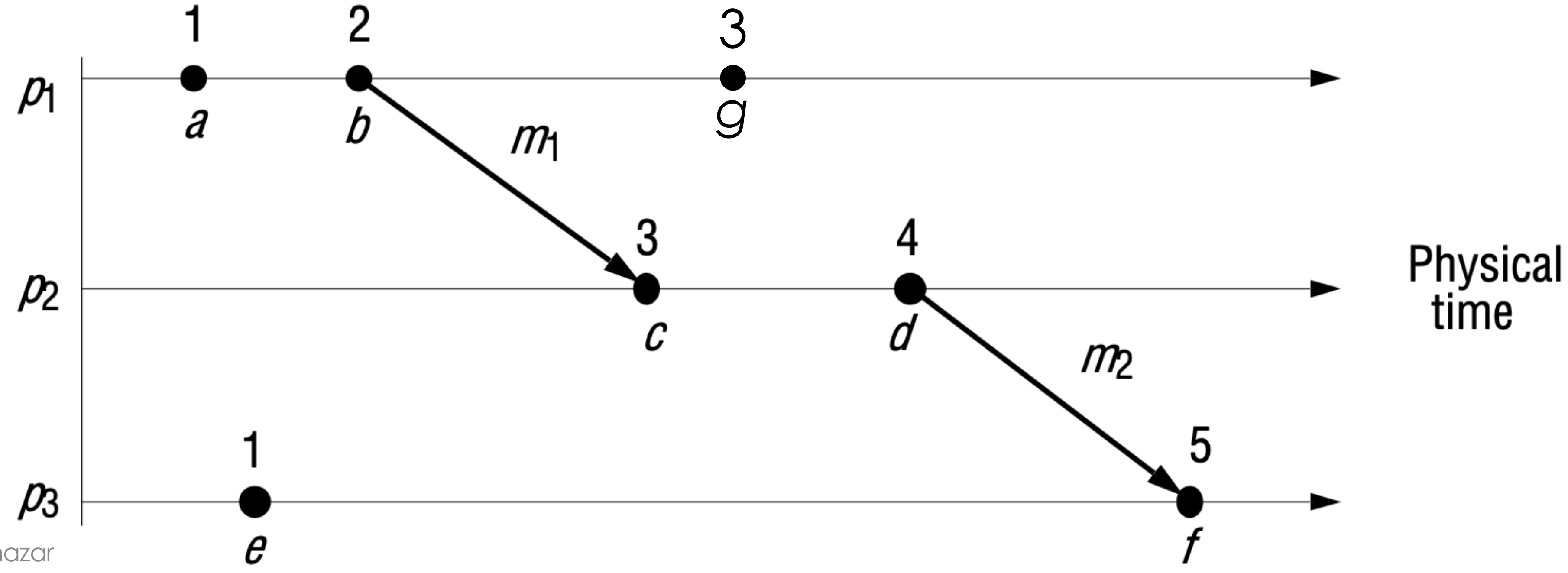
LOGICAL CLOCK: LAMPORT CLOCK

- 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



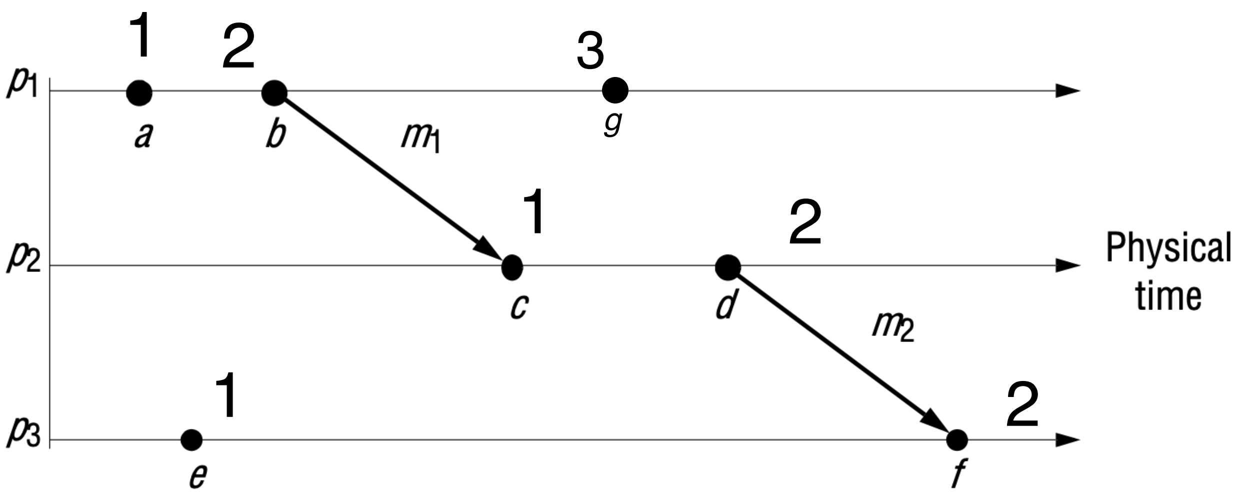
Lamport Clock

- 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



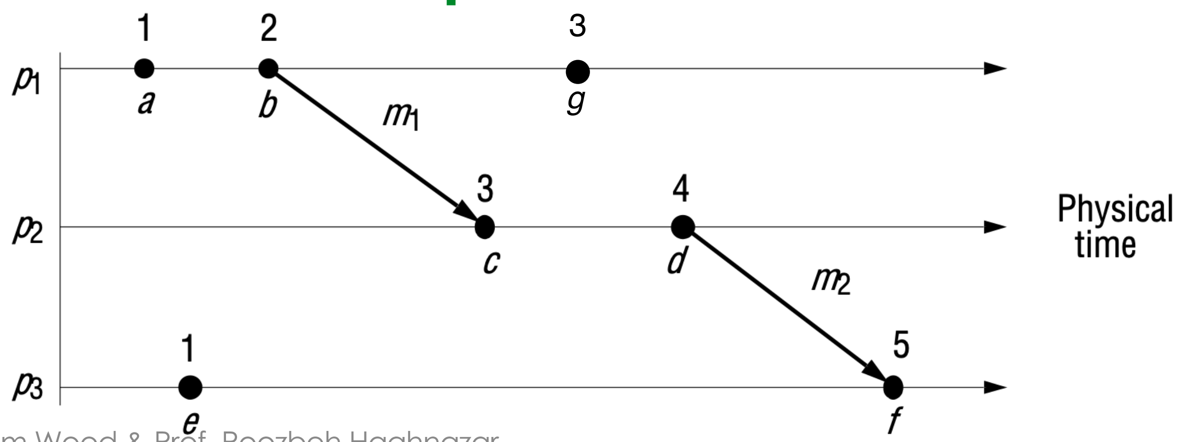
Clock Comparison

Independent clocks



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

Lamport clocks

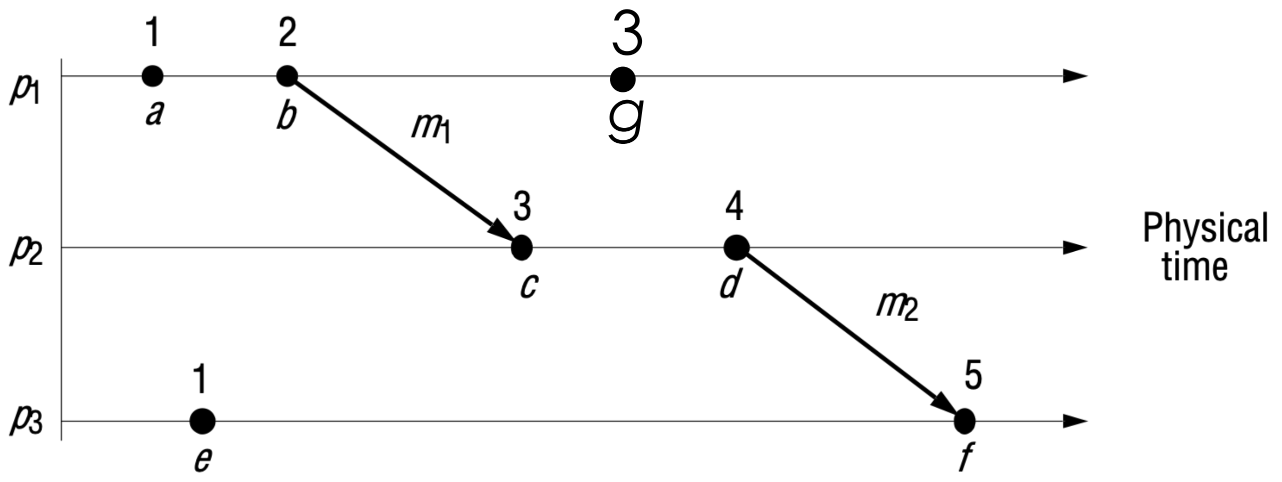


- **if e->e', then:**
- **L(e) < L(e')**

Clock Comparison

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

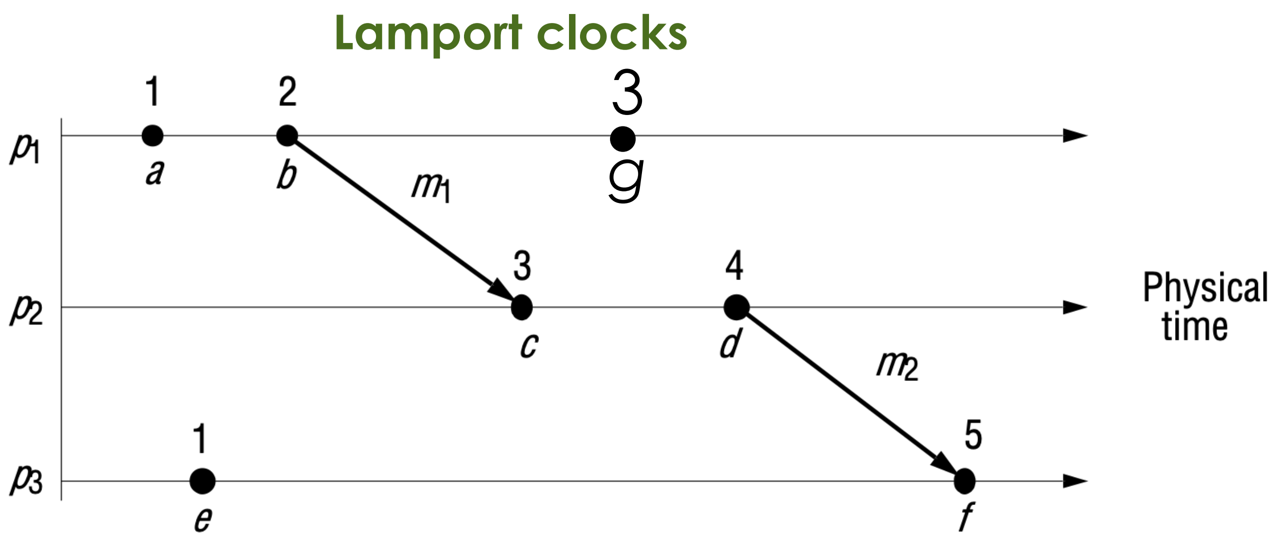
Lamport clocks



- 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 :(**

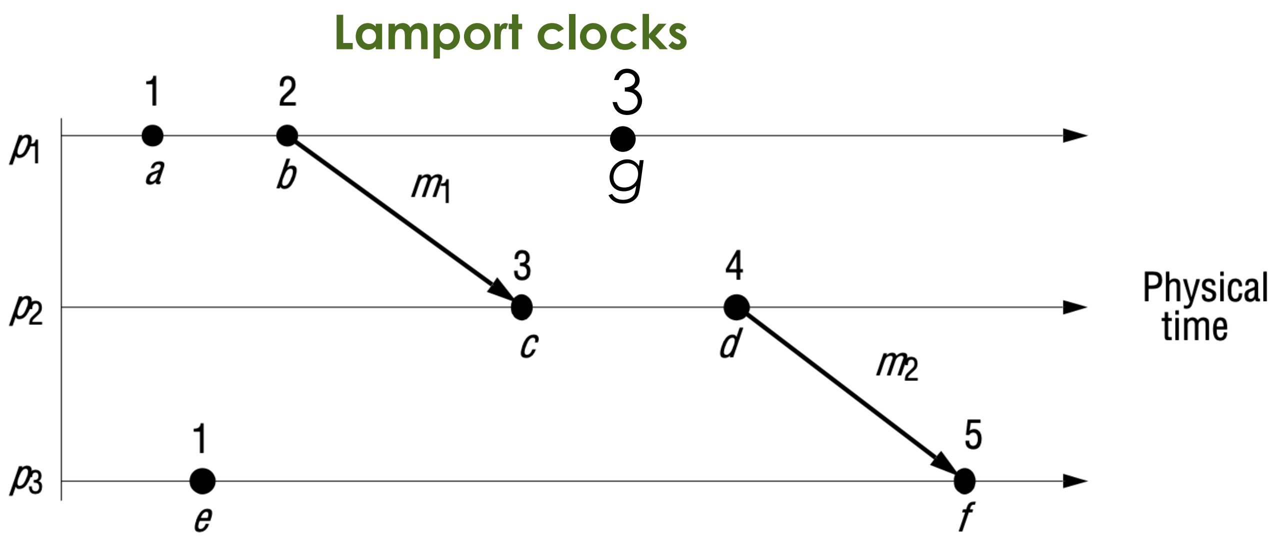


if e->e', then:

L(e) < L(e')

Lamport Clocks

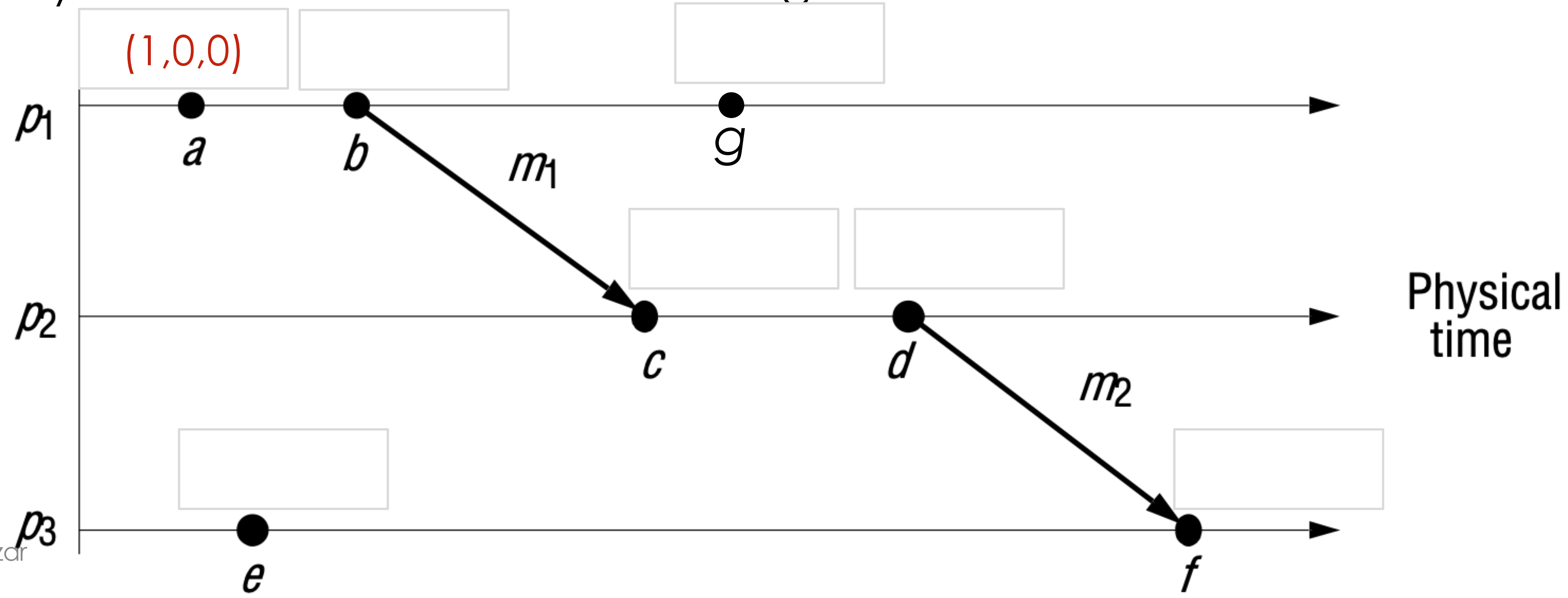
- Lamport clocks are better than nothing
  - but only let us make limited guarantees about how things are ordered
- Ideally we want a clock value that indicates:
  - If an event happened before another event
  - If two events happened *concurrently*



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

Vector Clocks

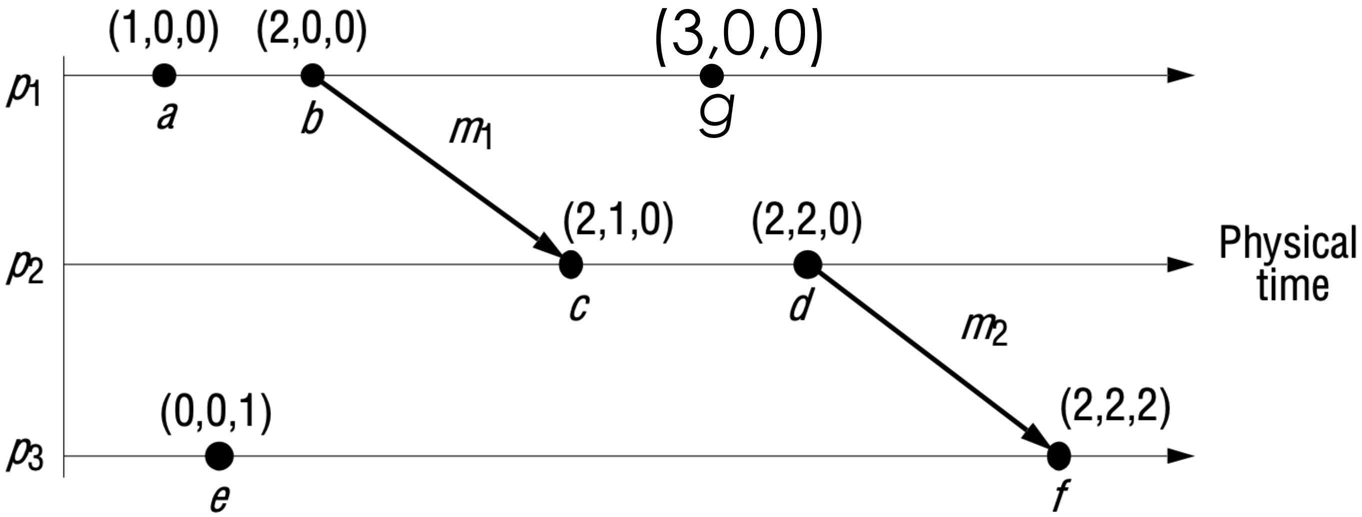
- Each process keeps an array of counters: (p1, p2, p3)
  - When p_i has an event, increment V[p_i]
  - Send full vector clock with message
  - Update each entry to the maximum when receiving a clock



Vector Clocks

A	C
A	G
G	F
G	C
F	C
E	D

- Now we can compare orderings!
- if V(e) < V(e') then e->e'
  - (a,b,c) < (d,e,f) if:
    $a \leq d$ & $b \leq e$ & $c \leq f$
- 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!

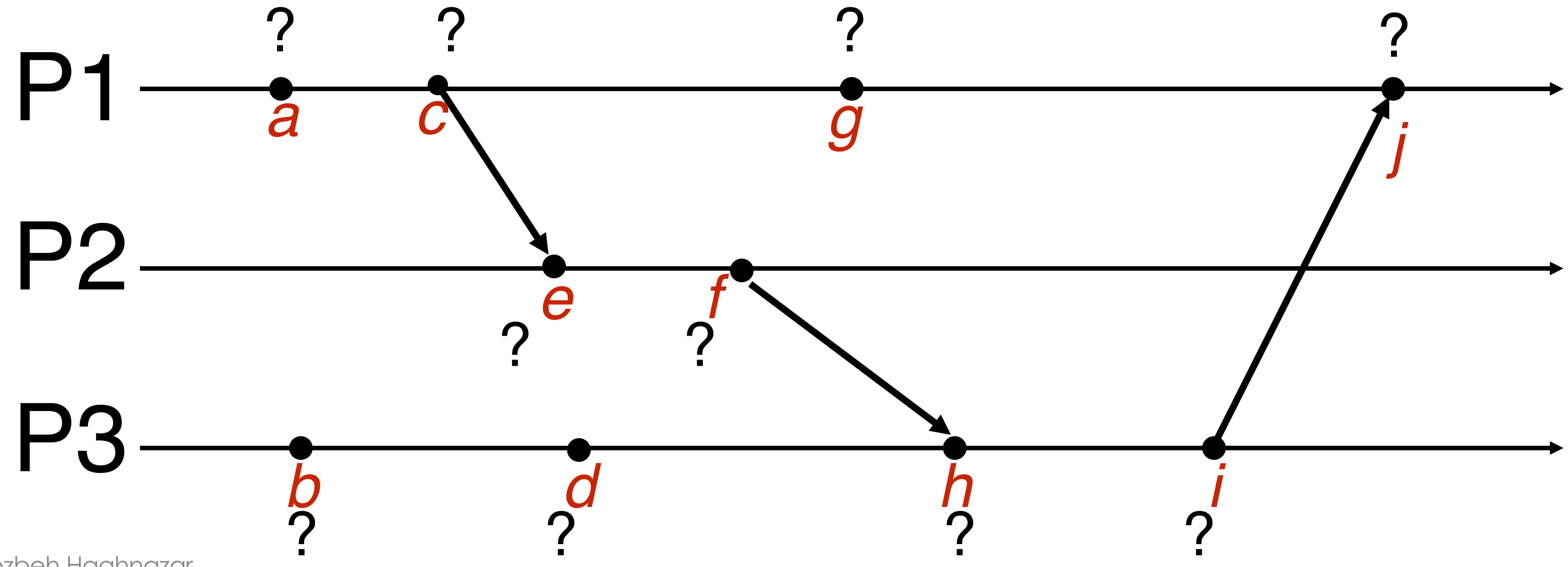
P1	P2	P3
a:1	c:3	e:1
b:2	d:4	f:5
g:3		

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

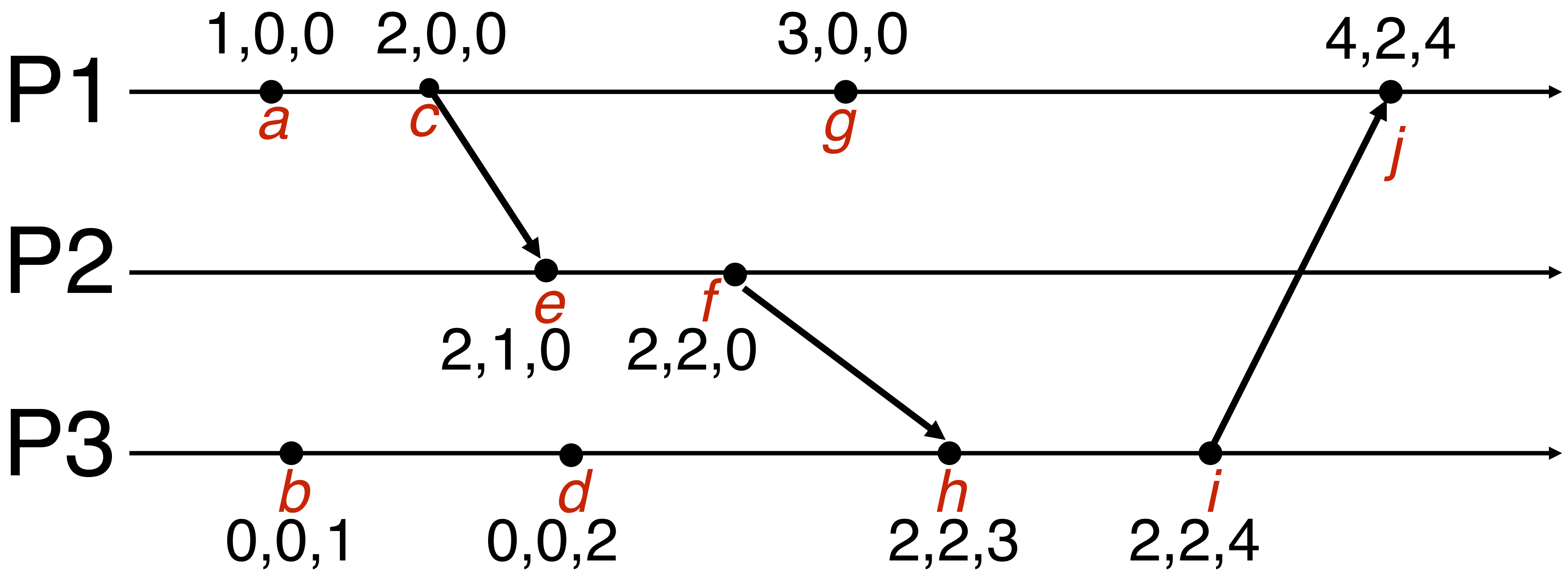
VC Rules:
- When p_i has an event, increment V[p_i]
- Update each entry to the maximum when receiving a clock
- Send full vector clock with message

- 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?

Clock Worksheet

- Do the worksheet in breakout rooms

`https://expl.ai/ENAJBDHK`

- When you finish, try this:
  - Draw the timeline for the four processes with vector clocks shown in problem 3. One of the clocks has an error – which one?

Find the bug???

P1	P2	P3	P4
a: 1,0,0,0	e: 1,1,0,0	i: 0,0,1,0	l: 0,0,0,1
b: 2,0,0,0	f: 1,2,0,1	j: 0,0,2,2	m: 0,0,0,2
c: 3,0,0,0	g: 1,3,0,1	k: 0,0,3,2	n: 0,0,0,3
d: 4,2,0,1	h: 1,4,3,2		

Version Vectors

- 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!

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

Order?

Alice	Ben	Cathy	Dave
Wednesday	Tuesday	Thursday	???

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	Ben	Cathy	Dave
Wednesday	Wednesday	Wednesday	Wednesday
	Tuesday	Thursday	Thursday
			Tuesday

Version Vectors

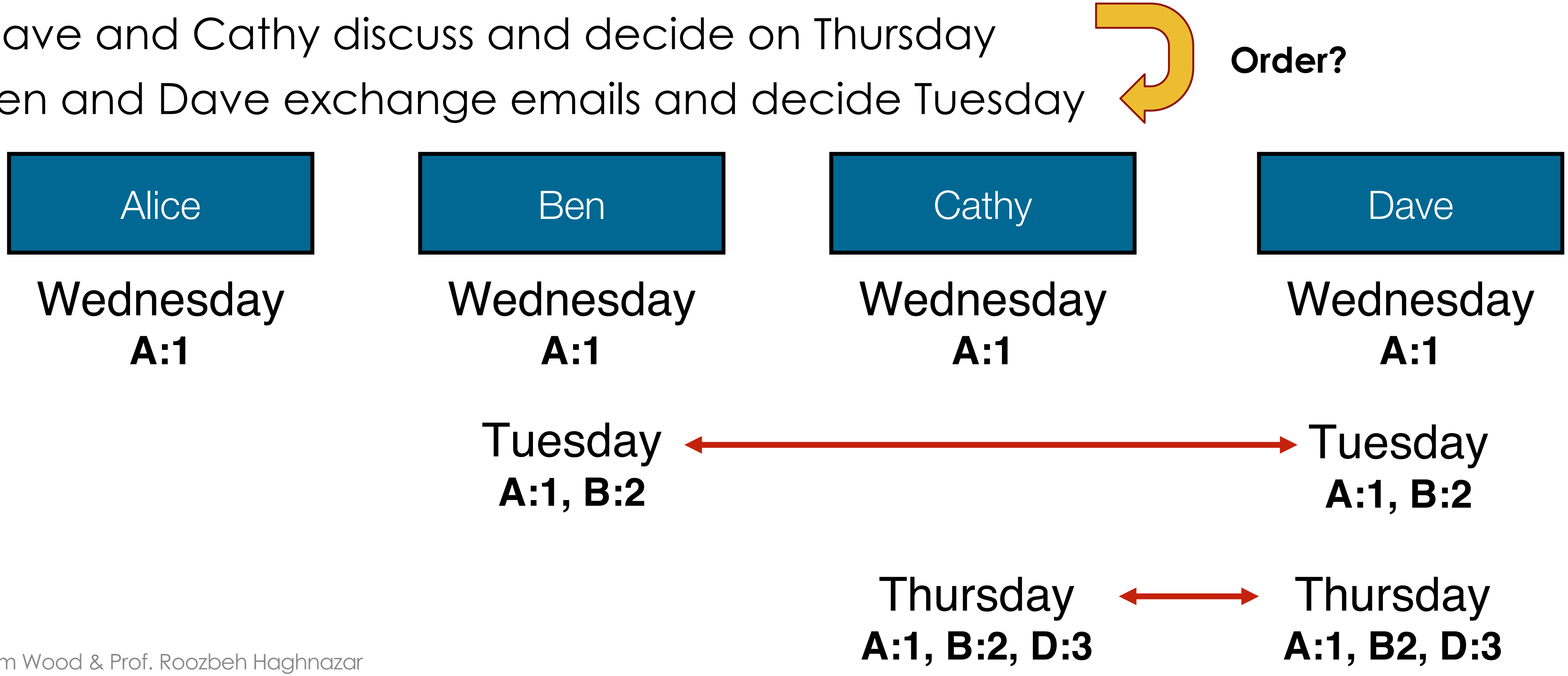
- Alice tells everyone to meet on Wednesday
- Dave and Cathy discuss and decide on Thursday
- Ben and Dave exchange emails and decide Tuesday

Order?

Alice	Ben	Cathy	Dave
Wednesday **A:1**	Wednesday **A:1**	Wednesday **A:1**	Wednesday **A:1**
		Thursday **A:1, D:2**	Thursday **A:1, D:2**
	Tuesday **A:1, B:3, D:2**		Tuesday **A:1, B:3, D:2**

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



Version Vectors

- The result ends on the order of:
  - Dave and Cathy discuss and decide on Thursday
  - Ben and Dave exchange emails and decide Tuesday

Alice	Ben	Cathy	Dave
Wednesday	Tuesday	Thursday	Tuesday
A:1	**A:1, B:3, D:2**	**A:1, D:2**	**A:1, B:3, D:2**

or

Alice	Ben	Cathy	Dave
Wednesday	Tuesday	Thursday	Thursday
A:1	**A:1, B:2**	**A:1, B:2, D:3**	**A:1, B:2, D:3**

Resolving Conflicts

- What if we have?

Alice	Ben	Cathy	Dave
Friday	Tuesday	Thursday	Thursday
A:2	**A:1, B:2**	**A:1, B:2, D:3**	**A:1, B:2, D:3**

- What are the conflicts?

Resolving Conflicts

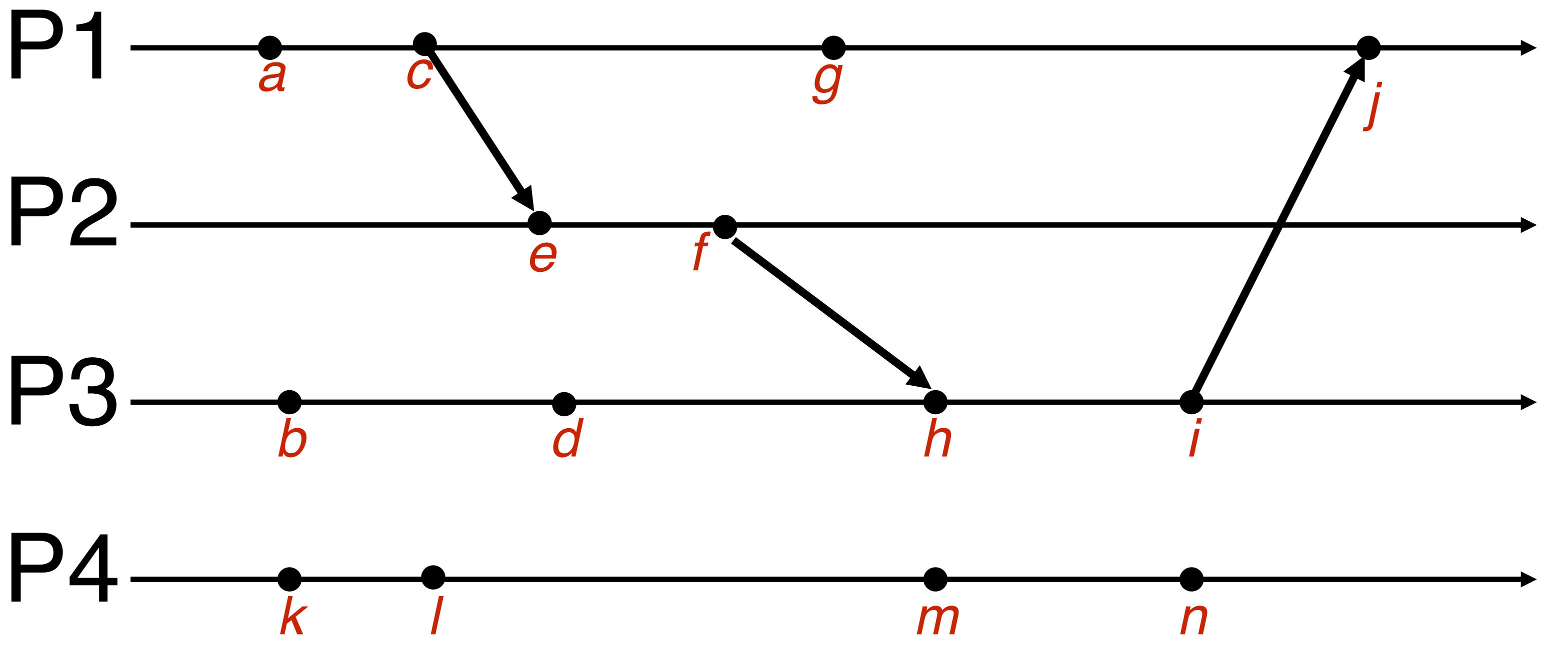
- What if we have?

Alice	Ben	Cathy	Dave
Friday	Tuesday	Thursday	Thursday
A:2	**A:1, B:2**	**A:1, B:2, D:3**	**A:1, B:2, D:3**

- How to resolve Alice vs the rest?
  - The Tuesday vs Thursday debate is not a real conflict since we can order them based on their version vectors
- We need a policy for resolving the conflicts
  - Random, Priority based, User resolved

DEPENDENCIES

- Vector clocks also help understand the dependency between different events and processes



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

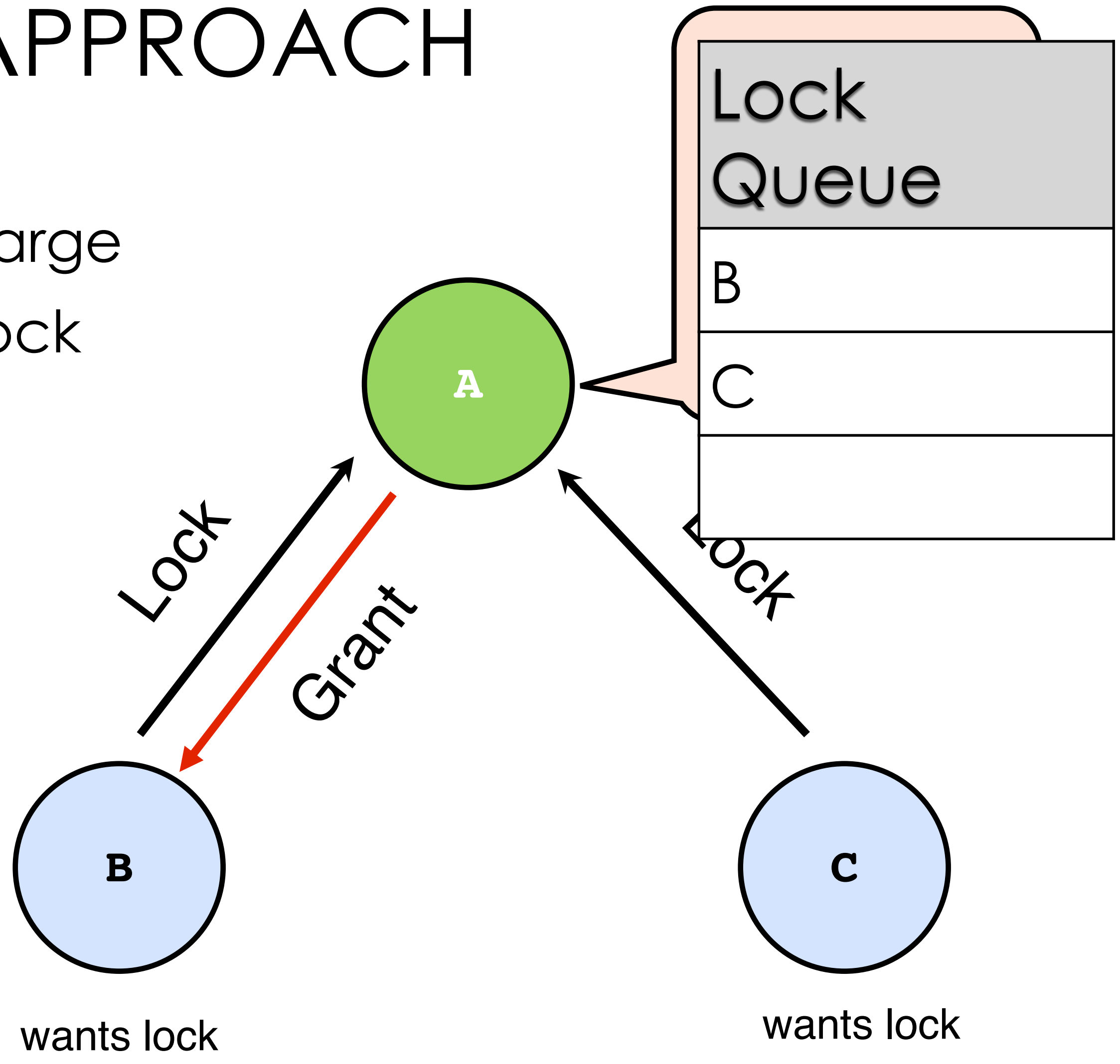
Includes material adapted from Van Steen and Tanenbaum's Distributed Systems book

(Distributed) Locking

- We need mutual exclusion to protect data
  - How does this limit scalability?
- Among processes and threads:
  - Mutexes and Semaphores
- Among distributed servers?
- Centralized or decentralized?

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?



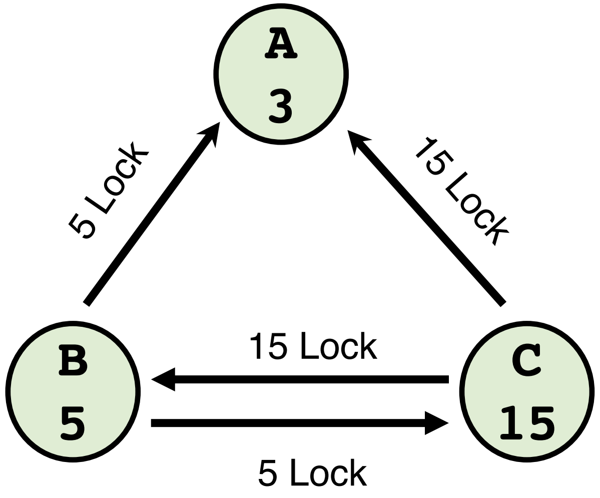
Distributed Lock

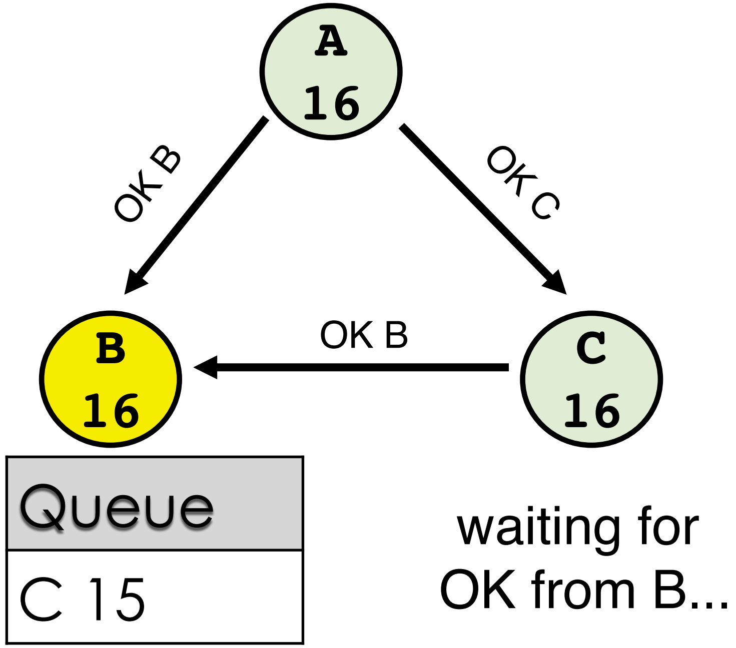
- If you want the lock: Broadcast “I want the lock!”
- On RX:
  - If we own the lock:
    - add this request to a queue
    - Reply: “wait a minute… you are # x in line”
  - Else:
    - do nothing
- When done with lock:
  - Send “You now have the lock” to first entry in queue and send the queue
  - If nobody in the queue, then hold onto the lock until somebody else needs it or randomly send it to somebody else if we need to exit.

  - if nobody else has asked me for the lock: Reply “Sure”
  - 1) else reply “no way”
  - 2) else add this request to queue
- Wait for a timeout…
  - If 50% say sure, then take the lock
  - else, “I no longer want the lock”

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:
  - Send OK if not interested
  - If I want the lock:
    - Send OK if request's clock is smaller
    - Else, put request in queue
- When done with a lock:
  - Send OK to anybody in queue





Comparison

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

Is the distributed approach better in any way?

Comparison

- Messages per lock/release
  - Centralized: 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..

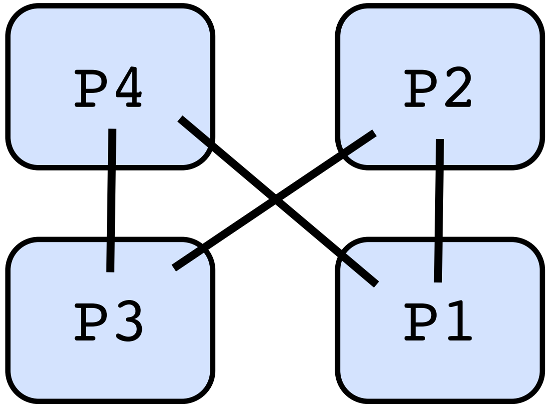
- Slower
  - If everyone needs to do more work

- More error prone
  - 10 nodes are 10x more likely to have a failure than one

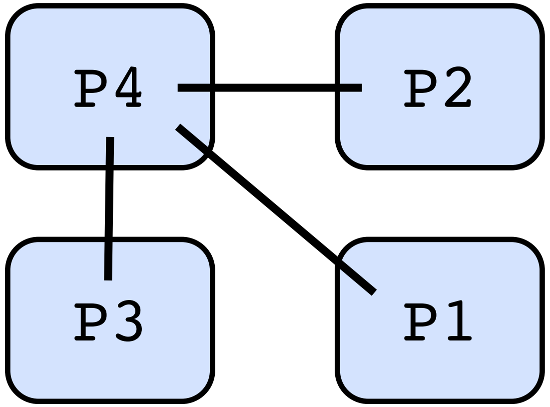
- Much more complicated
  - If you need a complex protocol
  - If nodes need to know about all others

Distributed Architectures

- Purely distributed / decentralized architectures are difficult to run correctly and efficiently



Decentralized

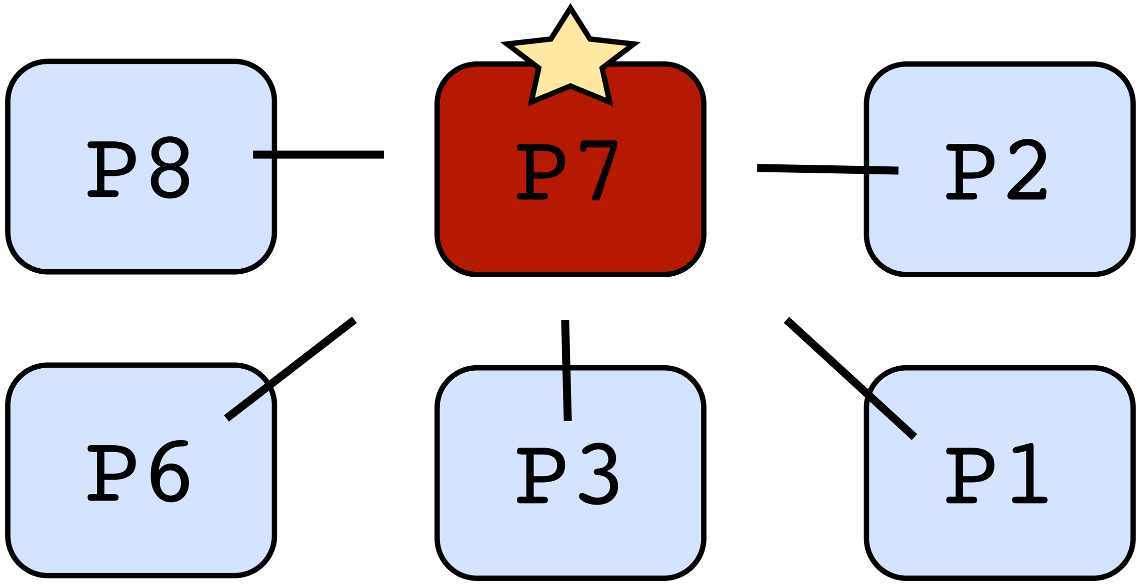


Centralized

- Can we mix the two?

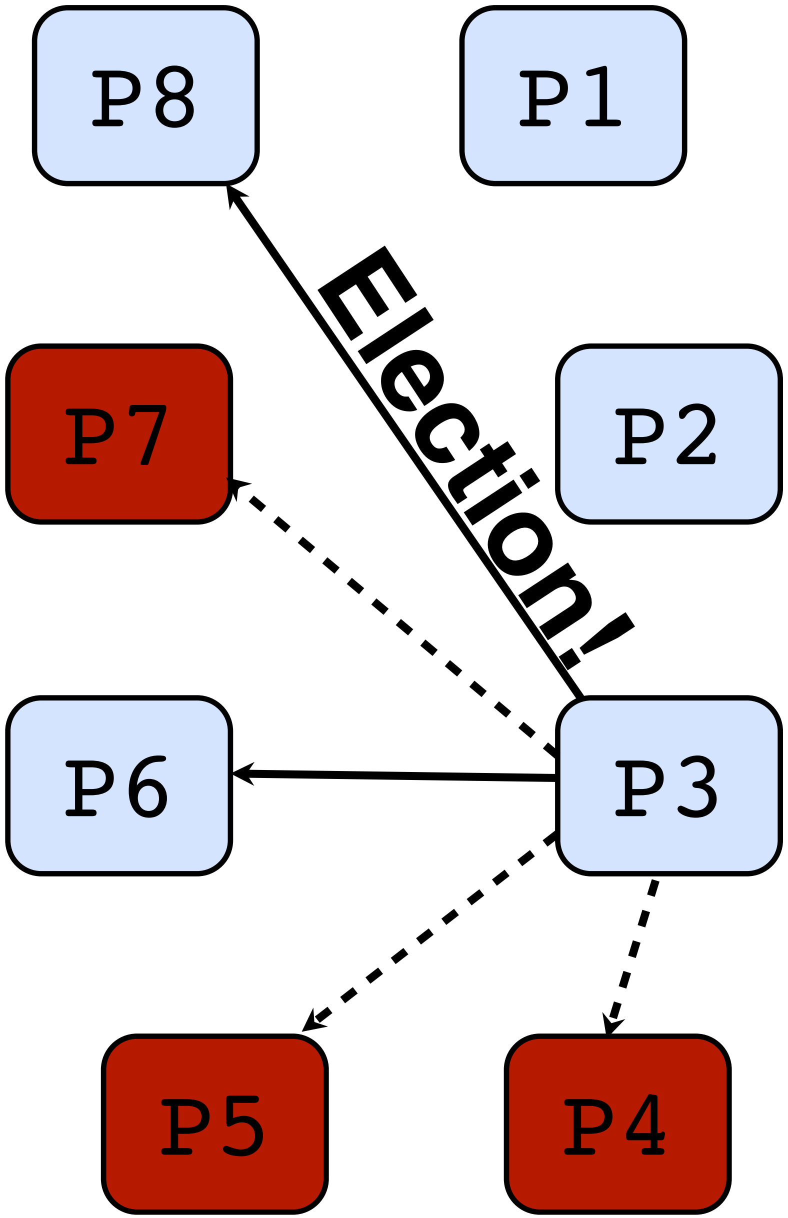
Elections

- 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?



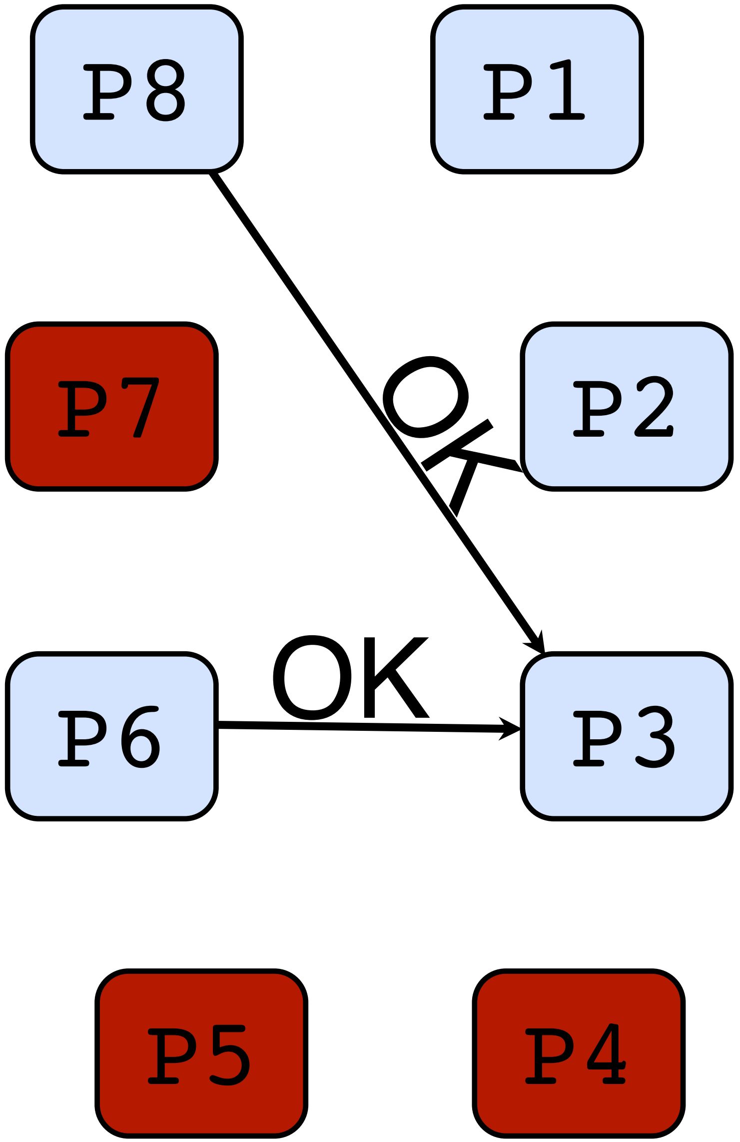
Bully Algorithm

- 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**...



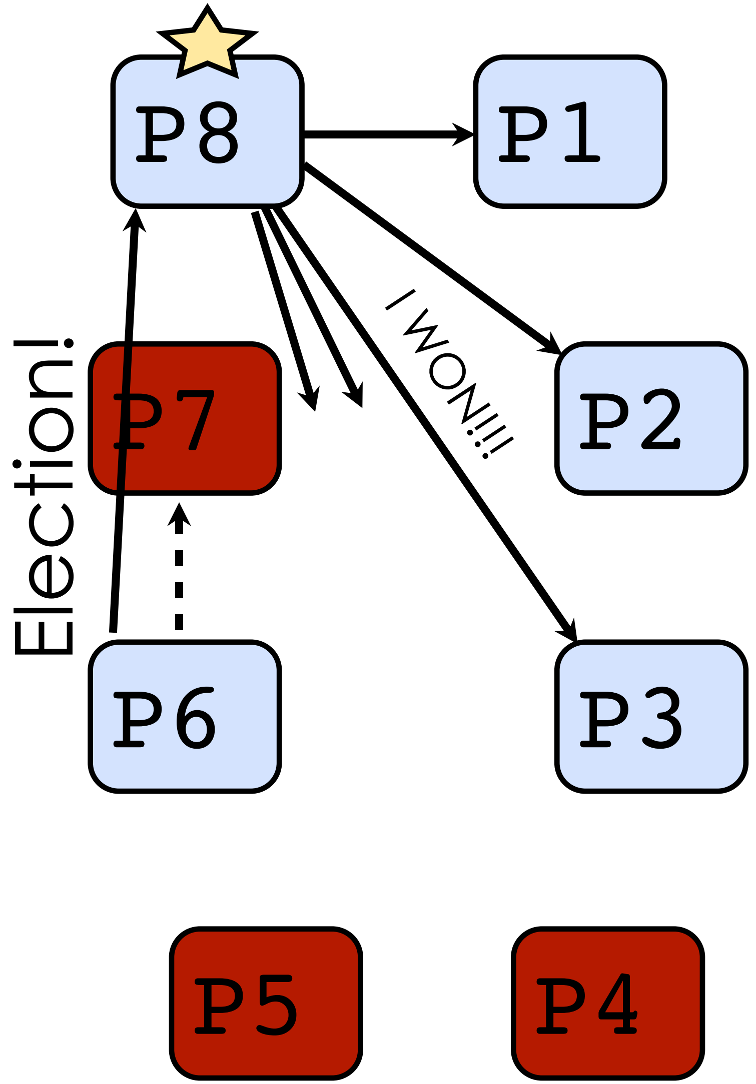
Bully Algorithm

- 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**...



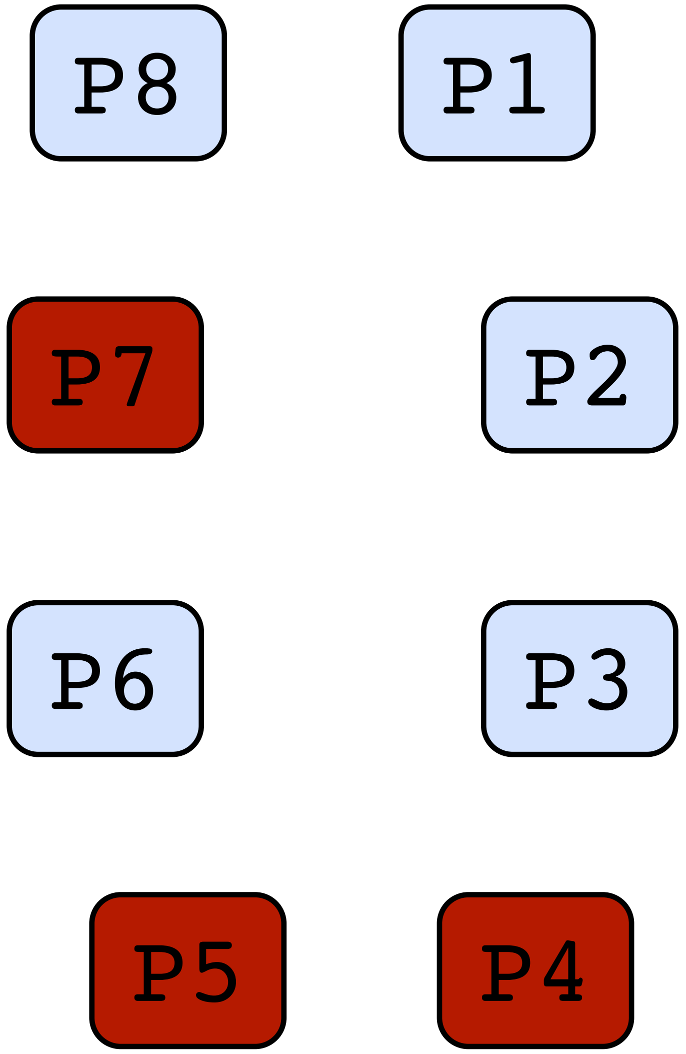
Bully Algorithm

- 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 an 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

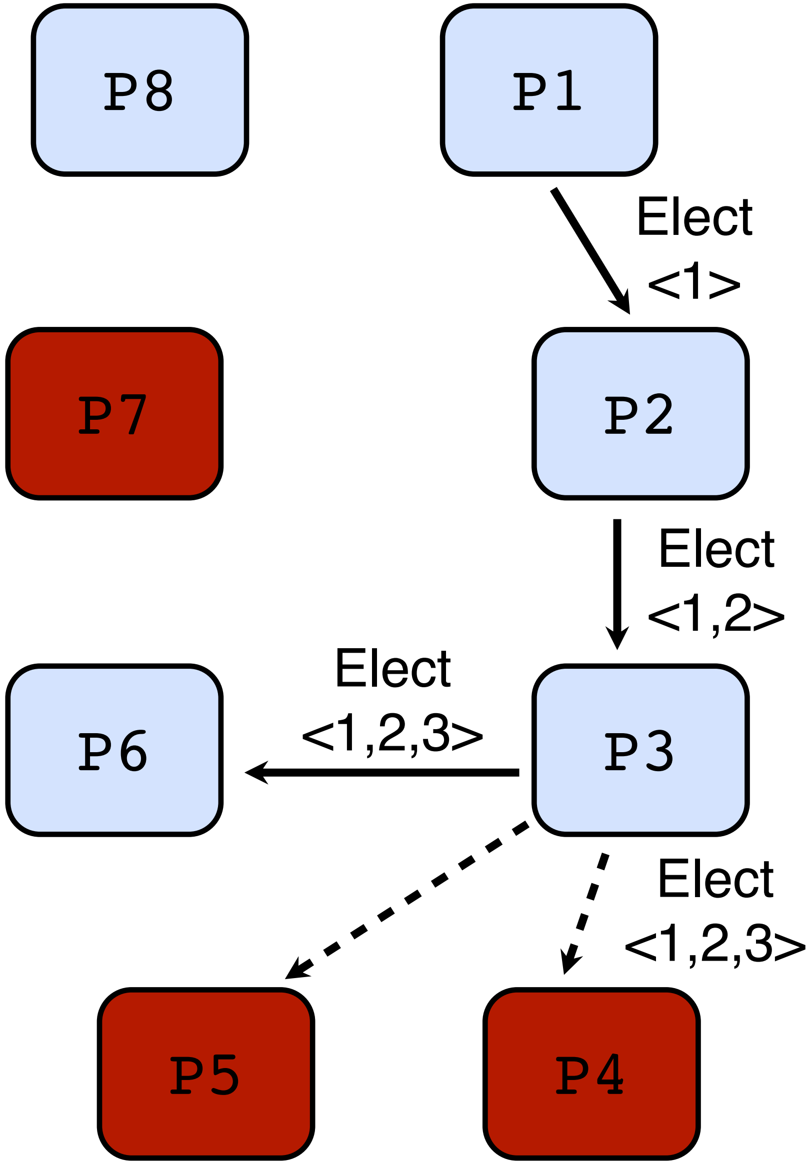
- Any other ideas?



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

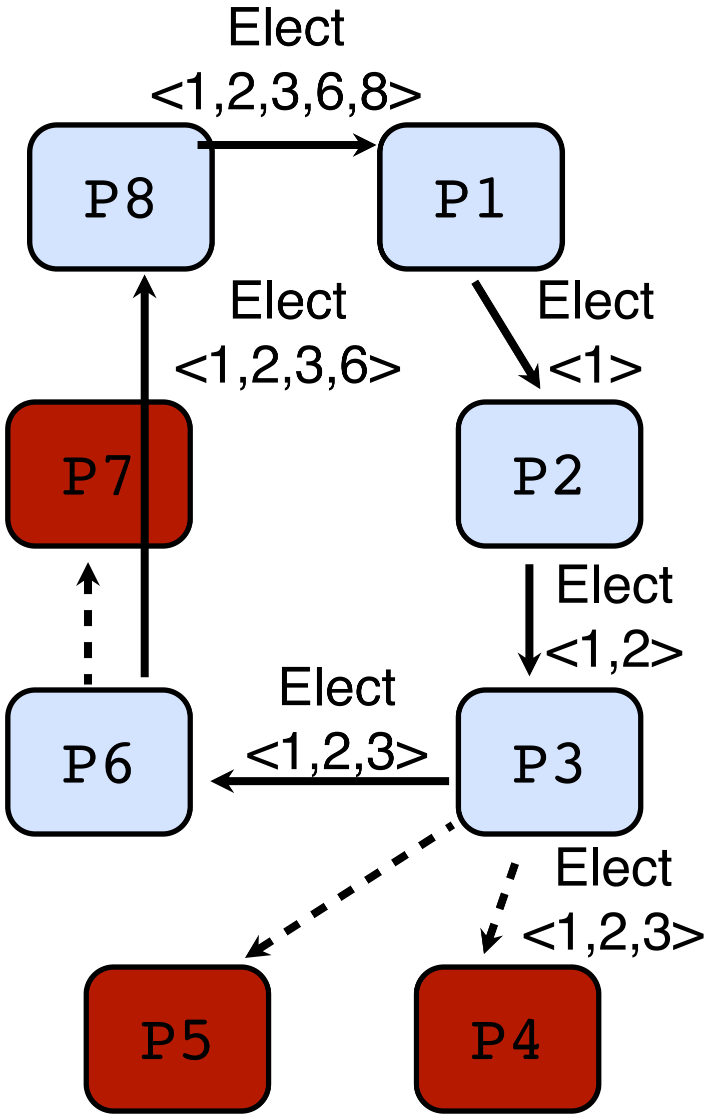
- What happens if multiple elections occur at the same time?



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?