THE UNIVERSITY OF CHICAGO

5. Coordination

MPCS: 52040 Distributed Systems

Kyle Chard

chard@uchicago.edu

Course schedule (* things might change a little)

Date	Lecture discussion topic	Class	Assessment
January 10	1. Introduction to Distributed Systems	Intro/Logistics	
January 17	2 Distributed architectures 3 Processes and virtualization	1 Docker	Homework 1 due
January 24	4 Networks and Communication	2 RPC/ZMQ/MPI	Homework 2 due
January 31	5 Naming	3 DNS/LDAP	Homework 3 due
February 7	6 Coordination and Synchronization		**Mid term exam**
February 14	7 Fault tolerance and consensus	Raft Project description	Homework 4 due (Project released)
February 21	8 Consistency and replication	4 FaaS	
February 28	9 Distributed data	5 Distributed data	
March 7	10 Data-intensive computing	6 Streaming	**Project due (March 9)** **Final exam (March 14)**

Introduction: Coordination

- (Distributed) systems rely on processes being able to cooperate to achieve a given goal
  - Problem: how do we coordinate and synchronize these processes?
- Examples of coordination
  - Manage shared access to resources so that processes are not accessing at the same time
  - Agree on the order of events (i.e., know which message was sent first)
  - Subdivide a workspace and collectively work on an analysis problem

Coordination and synchronization

- Coordination
  - *“the organization of the different elements of a complex body or activity so as to enable them to work together effectively.”*
  - Manage interactions and dependencies between activities in a distributed system

- Synchronization:
  - *“the operation or activity of two or more things at the same time or rate.”*
  - Process synchronization: ensure that one process waits for another to complete its operation

Agenda

- Part 1: Clock Synchronization
- Part 2: Logical clocks
- Part 3: Mutual exclusion
- Part 4: Election algorithms

Time

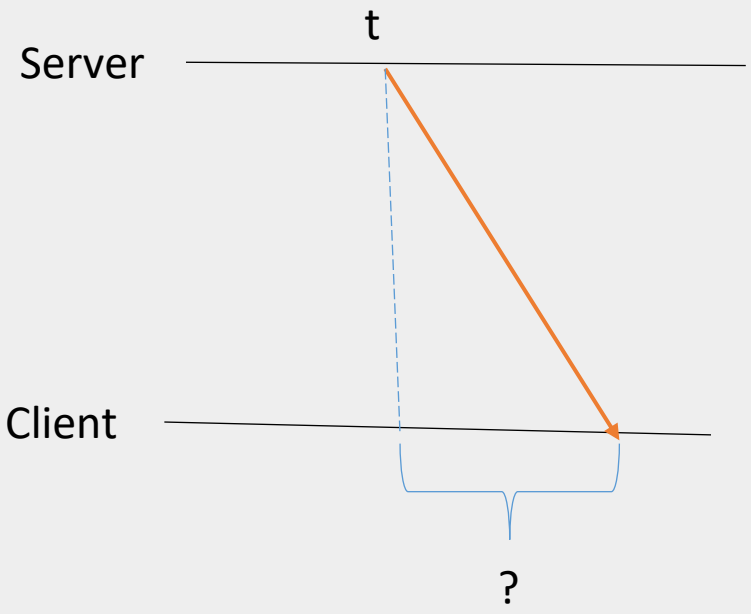
- In a centralized system *time* is easy
  - Single clock, everyone trusts that clock blindly
- Process wants to know the time… it asks the operating system
  - If process A gets the time and then process B gets the time, we know that Tb > Ta
  - Thus, we know which event happened first
- Think about all the places we implicitly rely on knowledge of time:
  - File timestamps, documentation synchronization, GitHub, compiling code, SSH connections
- Recall: distributed systems have no global clock

Clocks

- Physical clocks in computers are timers based on oscillation frequency of a crystal
- Clock tick: interrupt occurring when **counter** hits 0 (ticks normally happen many times per second)
  - Counter then reset to holding register and starts again…
- Clock skew: the difference between clocks
- Clock drift: the rate that a clock moves relative to *real* time
- Goals
  - Internal synchronization: keep deviation between two clocks on any two machines within a specified bound (precision)
  - External synchronization: keep bound to an external reference source such as UTC (accuracy)

How do we find out the time?

- Someone has more accurate idea of time than us (e.g., GPS or Atomic clock)
- Ask them to send us the time..
- Delays are inconsistent, messages are lost, etc.



Synchronization algorithms

- Christian's algorithm
- Network Time Protocol (NTP)
- Berkeley algorithm

- Approaches in wireless network
  - Reference broadcast synchronization

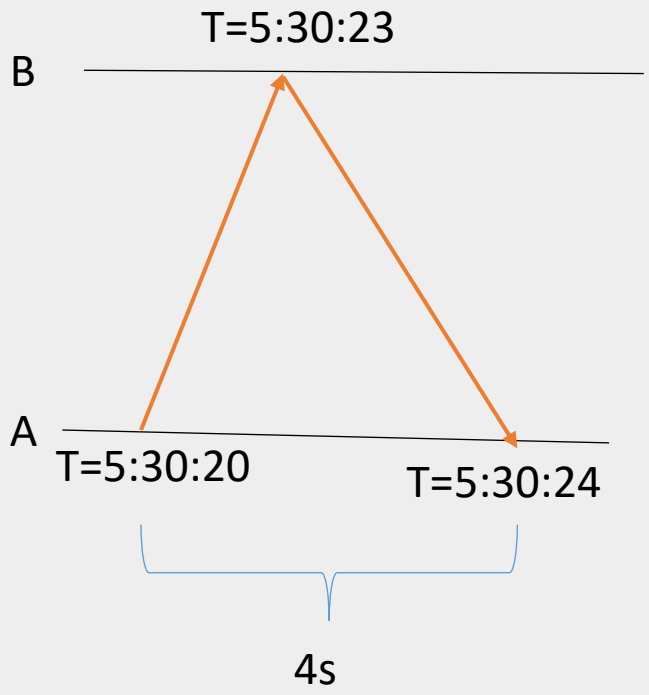
- And many others: https://en.wikipedia.org/wiki/Clock_synchronization

Christian's algorithm

- A requests the time from B
- After receiving the request B prepares a response and appends the time T from its own clock
- A then sets its time to be

  T + RTT/2

  - Assumption that the time is symmetric (ok on LAN)



Time = T+ RTT/2 = 5:30:23 + 4/2 = 5:30:25

Fixing incorrect time

- When we discover the time is out of sync what can we do?
- Can we go back in time?
  - Often not possible. What would happen if we reset our machine from 5:35 back to 5:30
- Better to slow down until correction has been made

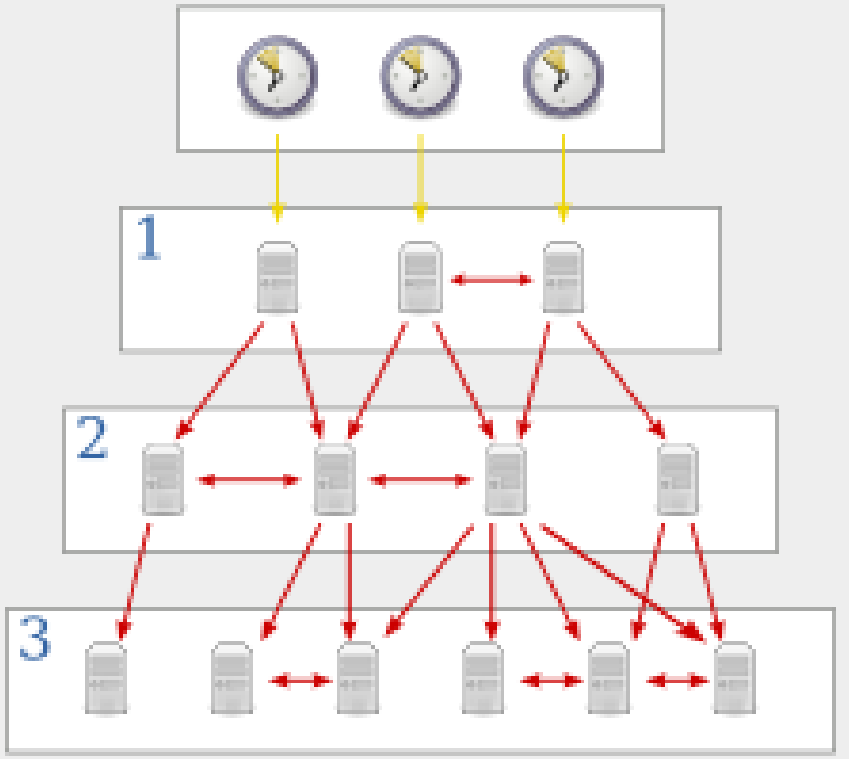
Network time protocol

- Extends Christian's algorithm for use on wide area, variable-latency networks
  - One of the oldest internet protocols in use today (1985)
- Pairwise protocol between servers
  - A will probe B and B will probe A
- Client will poll NTP servers and compute time offset and RTT delay
  - Take pairs of values, discard outliers, keep the *best* candidates (one or an average of several), then adjust frequency gradually to fix offset

NTP implementation

- Issues:
  - Symmetrical updates are not ideal if we know one server is more accurate than the other
  - What happens if we have loops?

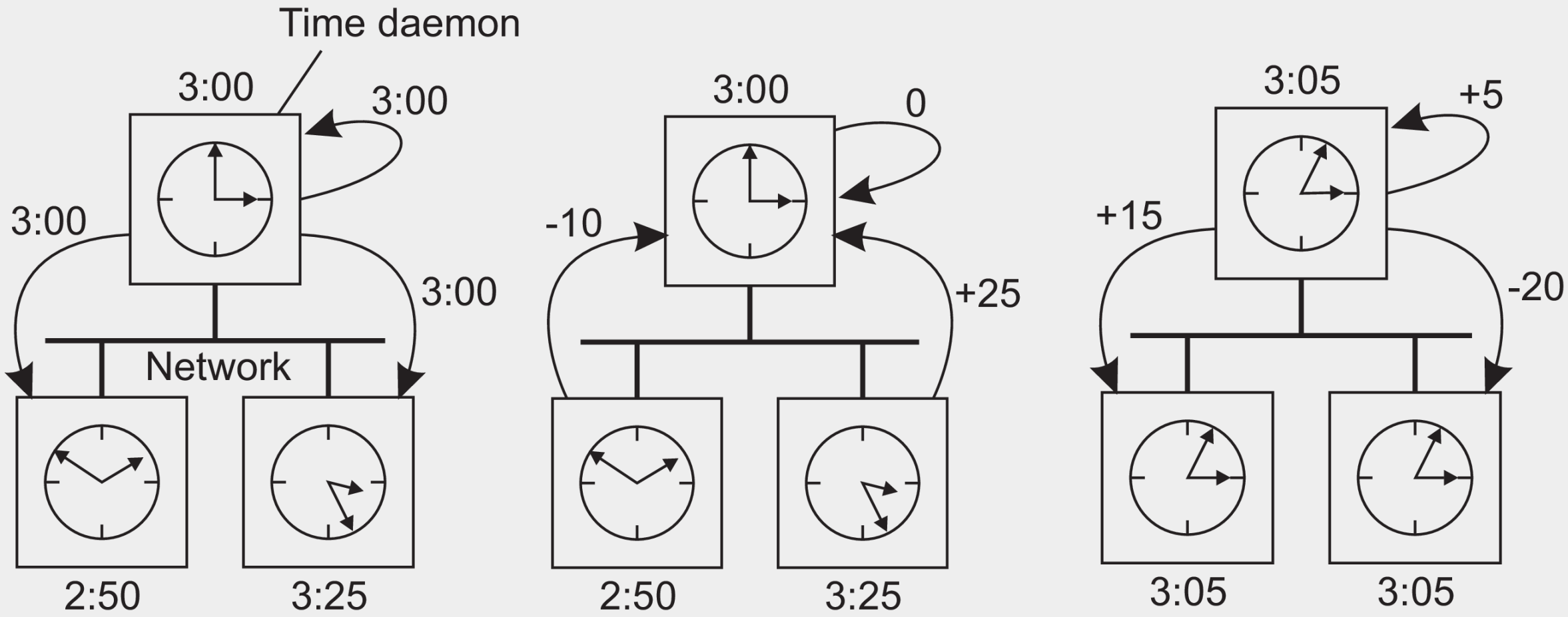
- Solution: organize into a hierarchy where the number represents distance from the reference clock (we update only from a higher stratum)
  - Stratum 0: high precision reference (GPS, Atomic clock)
  - Stratum 1 (primary time servers): computers synchronized within ms of stratum 0
  - Stratum 2: sync over network with several S1 servers
  - …
  - Stratum 16: Unsynchronized device



Precision over accuracy

- Most clock sync algorithms rely on a passive time server
  - Clients simply ask it for the time
- What do we do when machines don’t have access to a time server or UTC clock?
  - In many cases it is sufficient to just have agreement on time rather than requiring that the time is accurate with UTC
- Berkeley algorithm is decentralized, time daemons actively poll other machines for their time, compute the average and tell other machines to advance/slow their clocks.

Example: Berkeley algorithm



Summary

- Synchronization and coordination is a crucial aspect of computer systems (and distributed systems)
  - The idea of doing the right thing at the right time
- Distributed systems make it hard as there is no globally shared clock and processes on different machines have their own notion of time
  - Even with the synchronization algorithms we discussed, we cannot know the exact order of events with any certainty
- Many algorithms for synchronizing time:
  - NTP for syncing absolute time
  - Berkeley algorithm for agreeing on some time

Agenda

- Part 1: Clock Synchronization
- Part 2: Logical clocks
- Part 3: Mutual exclusion
- Part 4: Election algorithms

Distributed Systems: Chapter 6

Logical clocks

- In many cases it is not necessary to have an accurate account of real time
  - We may just need every node to agree on a common time
  - But how often do we even need to know the relative times that events occurred?
- In many cases we might only need to track the order of events
  - Message a was sent before b
  - File X is outdated by file Y
- Lamport: clock synchronization need not be absolute, if two processes do not interact it is not necessary that their clocks be synchronized

The *happens before* relationship

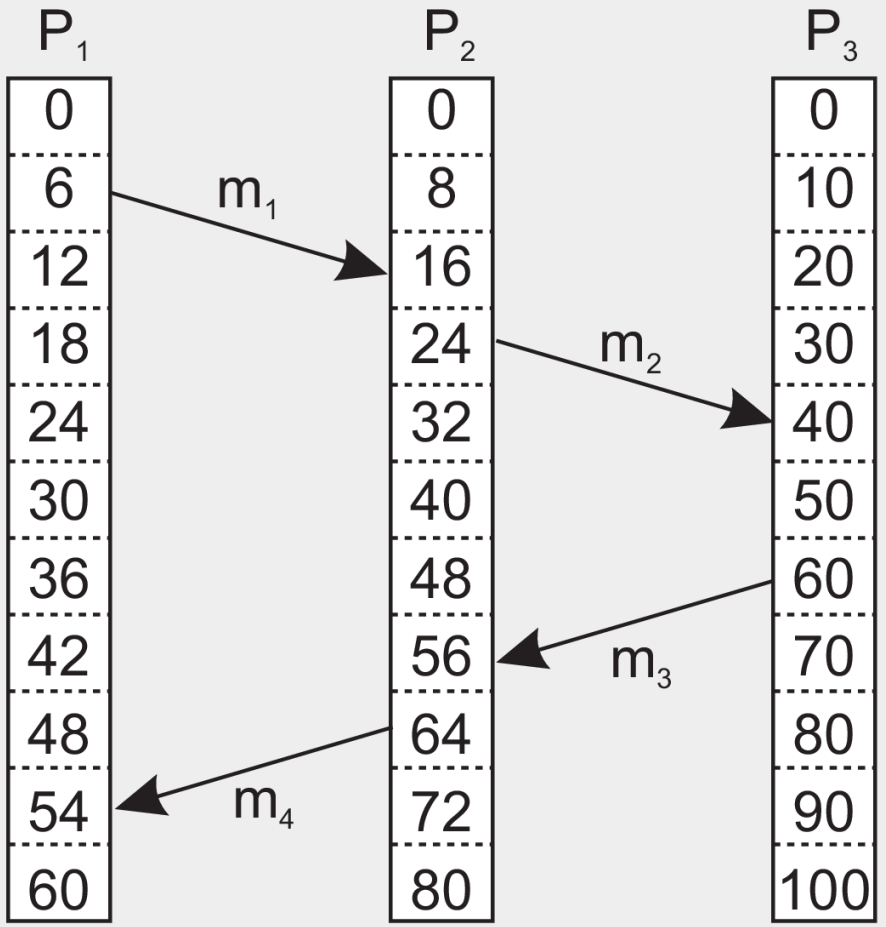
- The happens before relationship: $a \rightarrow b$
  - If a and b are two events in the same process, and a comes before b, then $a \rightarrow b$
  - If a is the sending of a message, and b is the receipt of that message, then $a \rightarrow b$
- Transitivity
  - If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$
- Concurrency
  - If events happen in different processes that do not exchange messages then we have no idea of their relationship and nothing needs to happen

Formalizing requirements

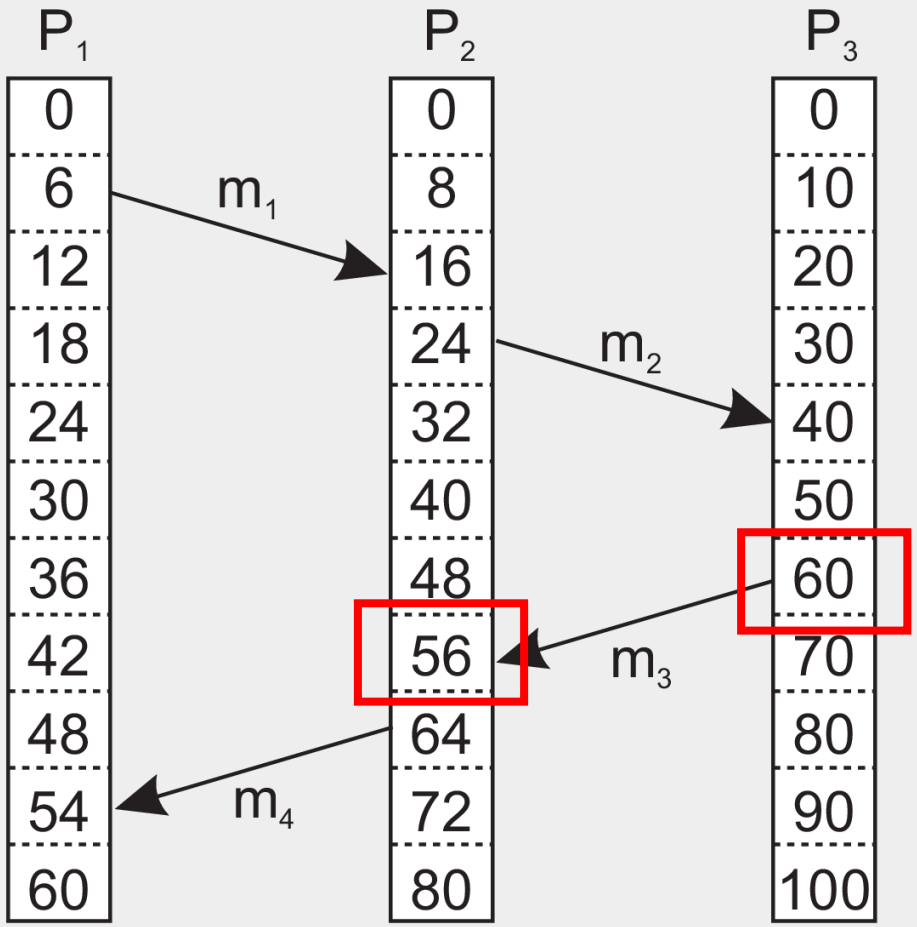
- How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

- Attach a timestamp *C(e)* to each event *e*, satisfying the following:
  - P1: If a and b are two events in the same process, and a → b, then we require that *C(a)* < *C(b)*
  - P2: If a corresponds to sending message m, and b to the receipt of that message, then also *C(a)* < *C(b)*
- Problem
  - How do we attach a timestamp to an event when there's no global clock?
  - ➔ maintain a consistent set of logical clocks, one per process

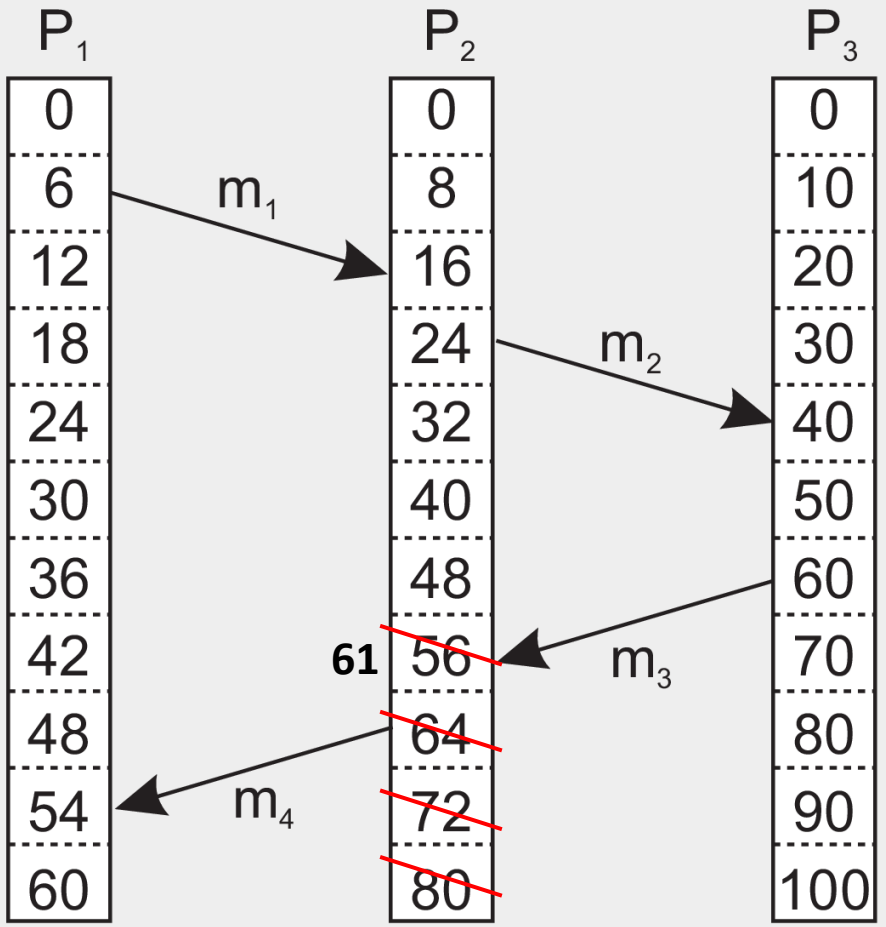
Lamport's logical clock example



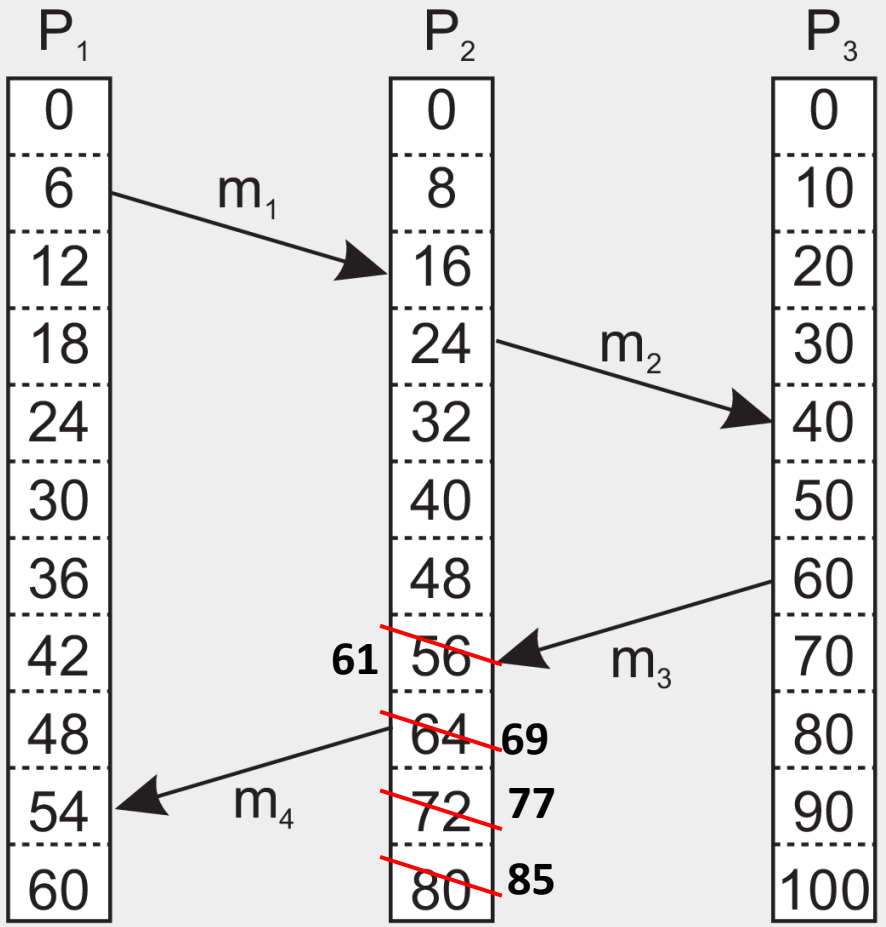
Lamport's logical clock example



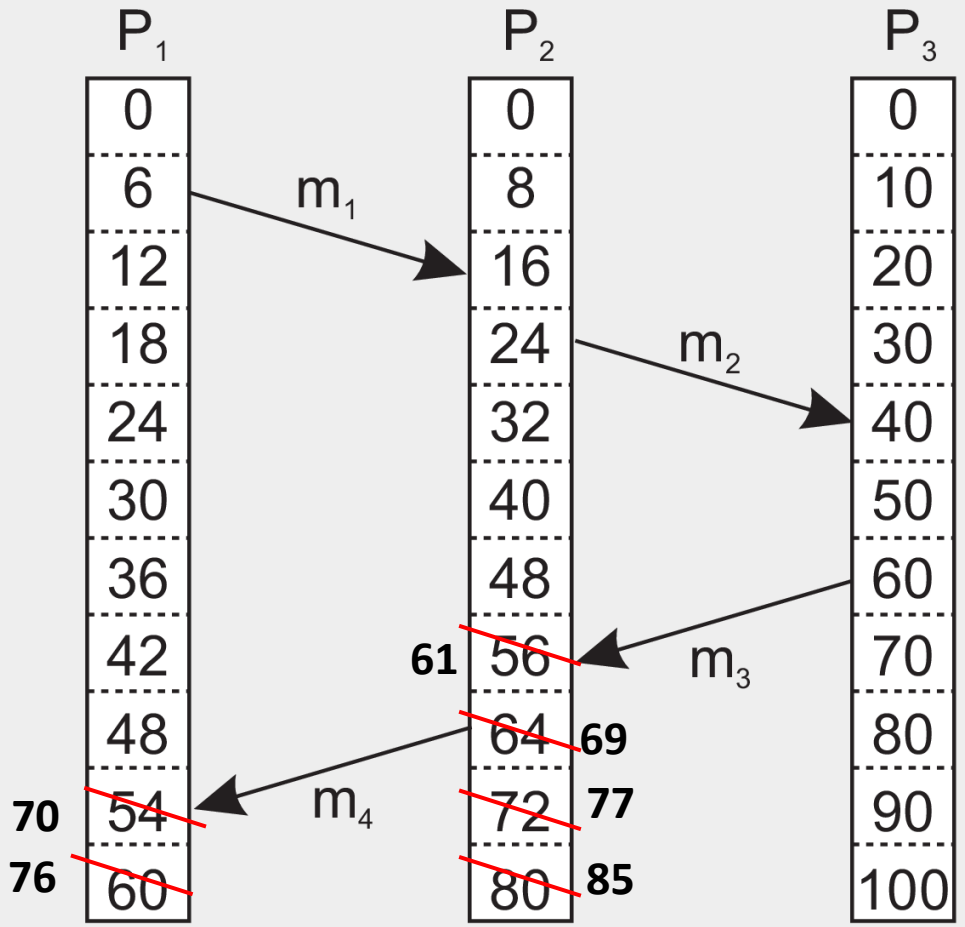
Lamport's logical clock example



Lamport's logical clock example



Lamport's logical clock example



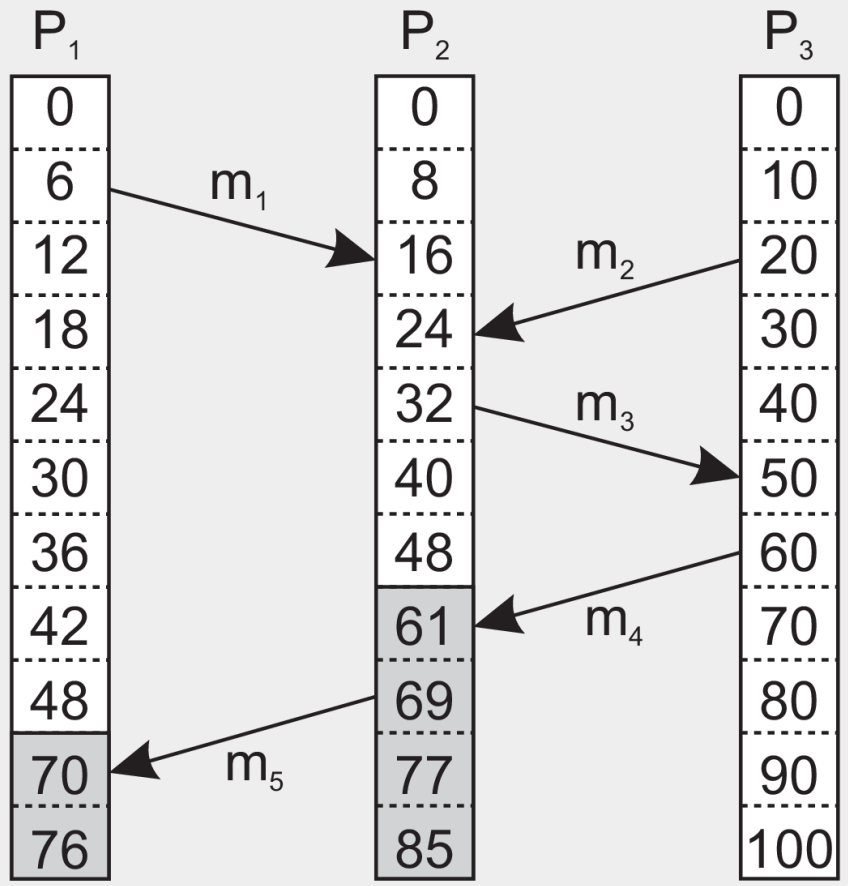
What else can we infer from logical clocks?

- Lamport's logical clocks provide guarantees on the ordering of events i.e., if a happened before b then $C(a) < C(b)$

- They do not say anything about the relationship between two events by comparing their time values $C(a)$ and $C(b)$
  - If $C(a) < C(b)$ does not imply that a happened before b

Causality: E.g., concurrent message transmission

- Consider the following example
  - We know the ordering of messages Mi < Mj
  - But just because Mi < Mj what can we conclude about their relationship?
  - E.g., we know M1 is received at 16 and M2 is sent at 20, but we don't know if M2 had anything to do with M1
- Often what we really care about is causality



Capturing causality

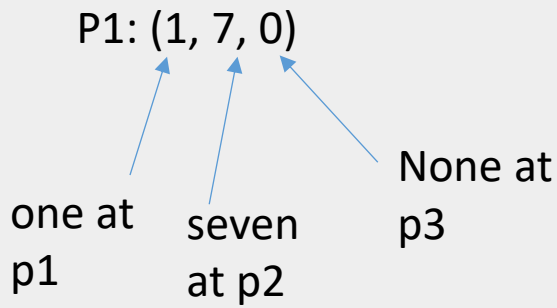
- Tracing causality is simple if we assign each event a unique name (e.g., process ID + local counter)
  - Pk is the kth event that happened at process P
- Then the problem becomes how do we track causal histories
  - E.g., if two events happened at P, then the causal history H(p2) of event p2 is {p1, p2}
- Now if we send a message from P to Q:
  - This is an event P3 on P and a new event Qk on Q (e.g., q2 if it already has {q1})
  - P also needs to send its causal history (e.g., {p1, P2}) to Q; and
  - Q needs to merge the causal histories into a new history:
    - {p1, p2, p3, q1, q2}

Problem with causal histories

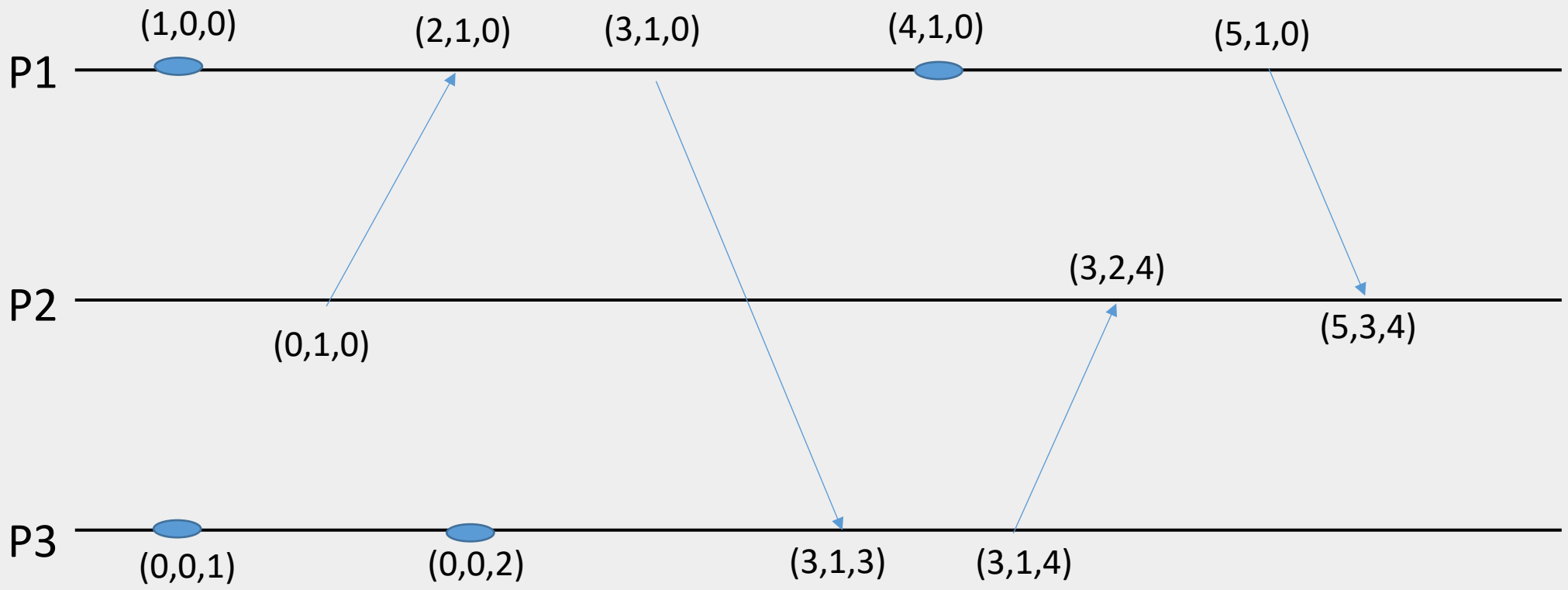
- Representation is not very efficient (store and share all these events)
- However, in practice we don't need to track all successive events from the same process… we only need the the last one
- What we can do is assign an index to each process and represent the causal history as a vector
  - Jth entry represents the number of events that happened at process Pj. E.g., (P1, P2, P3)
    - (0,5,1) == 0 events at P1, 5 events at P2, 1 event at P3
- Causality can be captured by vector clocks which are constructed by letting each process Pi maintain a vector VC

Vector Clocks

- Each Pi maintains a vector VCi
  - VCi [i] is the local logical clock at process Pi
  - If VCi [j] = k then Pi knows that k events have occurred at Pj
- Maintaining vector clocks
  1. Before executing an event Pi executes VCi [i] ← VCi [i]+1.
  2. When process Pi sends a message m to Pj, it sets m's (vector) timestamp ts(m) equal to VCi after having executed step 1
  3. Upon the receipt of a message m, process Pj sets
     VCj [k] ← max{VCj [k,; ts(m)[k]} for each k,
     after which it executes step 1 and then delivers the message to the application.



Vector clocks: Example



Agenda

- Part 1: Clock Synchronization
- Part 2: Logical clocks
- Part 3: Mutual exclusion
- Part 4: Election algorithms

Distributed Systems: Chapter 6

Mutual exclusion

- How do we control simultaneous access to the same resource?
  - Avoid corruption, inconsistency, etc.
- Fundamental problem associated with concurrency and collaboration amongst process
- Solution: grant mutual exclusive access to a process
  - Same idea as mutual exclusion on a multi-processor/thread system

Two main approaches

- Permission-based approach
  - Process wanting to access a resource first acquires permission from other processes
- Token-based approach
  - Pass a special message (a “token”) between processes
  - Only one token available and whoever has it can access the shared resource

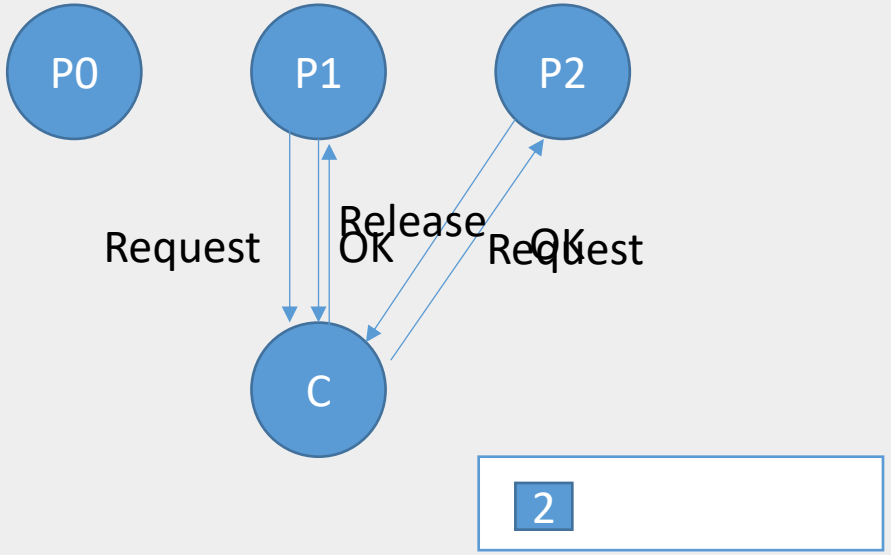
Simple approach: centralization

- The easiest approach is to simulate the same model as in a single processor system

  ➔Elect one process as the coordinator

- Must ask coordinator when you want access to a resource
  - Coordinator will only grant access if no other process is accessing the resource

Centralized coordinator-based approach



a) Process P1 asks the coordinator for permission to access a shared resource. Permission is granted

b) Process P2 then asks permission to access the same resource - the coordinator does not reply

c) When P1 releases the resource, it tells the coordinator, which then replies to P2

Centralized coordinator: pros and cons

- Pros:
  - Easy to demonstrate that it guarantees mutual exclusion
  - Fair (requests are granted in the order they arrive)
  - No starvation (no process waits for ever)
  - Easy to implement
  - Relatively efficient (requires only 3 messages per resource)

- Cons:
  - Single point of failure
  - Can’t distinguish a dead coordinator from permission denied (as no message is returned)

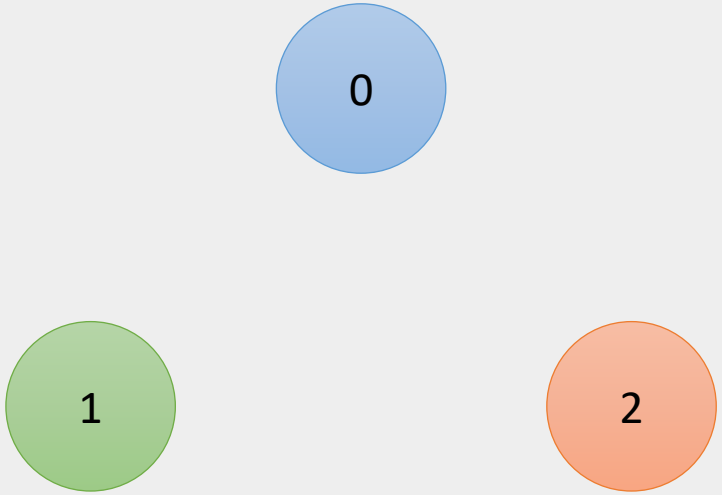
Other modes

- Distributed algorithm based on logical clocks
  - Nodes compute their timestamp and send to others requesting access
  - Receivers choose the node with the lowest time
- Token ring
  - Pass token around the ring and only process with the token can access resource
- Decentralized
  - Majority vote to access the resource

Distributed algorithm

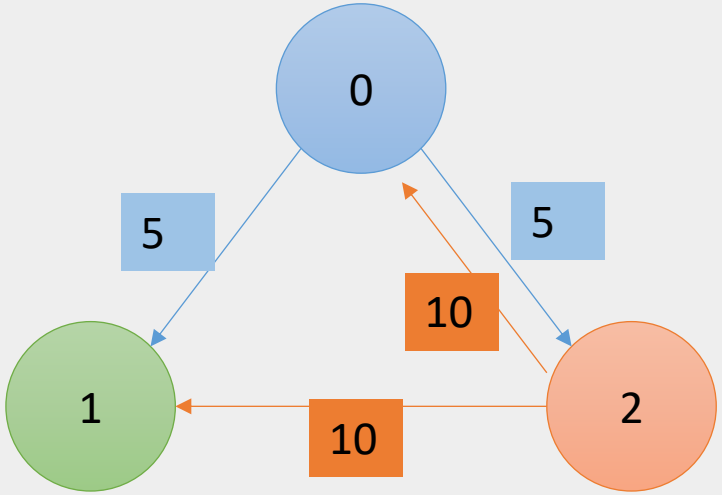
- Lamport described a simple distributed algorithm based on logical clocks, Ricart and Agrawala extended in 1981
- When a process wants to access resource it builds a message containing
  <resource name, process number, current logical time>
- When a process receives a request from another process it has 3 different options
  - If the receiver is not (and does not want to) access the resource ➔ Send OK to sender
  - If the receiver already has access to the resource, it does not reply ➔ Queue request
  - If the receiver wants to access the resource but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.
    - If the incoming message has a lower timestamp ➔ send ok
    - If its own message has a lower timestamp, queue the incoming request and send nothing

What happens if two processes try to access at the same time?



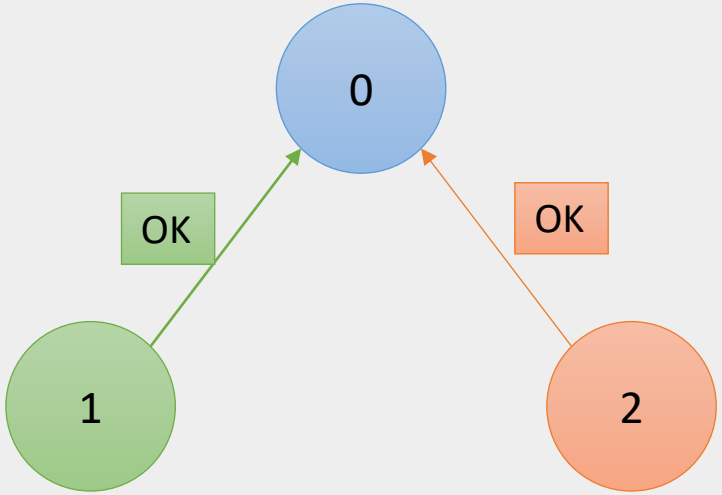
(a) Two processes (0 and 2) want to access a shared resource at same time

What happens if two processes try to access at the same time?



(a) Two processes (0 and 2) want to access a shared resource at same time

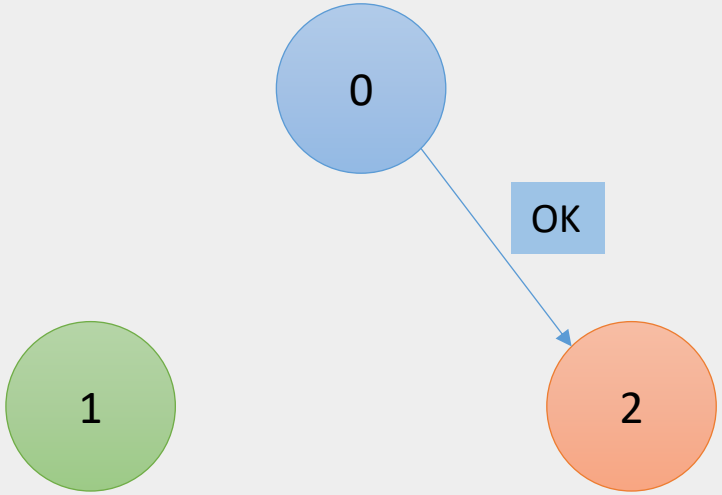
What happens if two processes try to access at the same time?



(a) Two processes (0 and 2) want to access a shared resource at same time

(b) P0 has the lowest timestamp, so it wins

What happens if two processes try to access at the same time?



(a) Two processes (0 and 2) want to access a shared resource at same time

(b) P0 has the lowest timestamp, so it wins

(c) When P0 is finished it grants access to P2

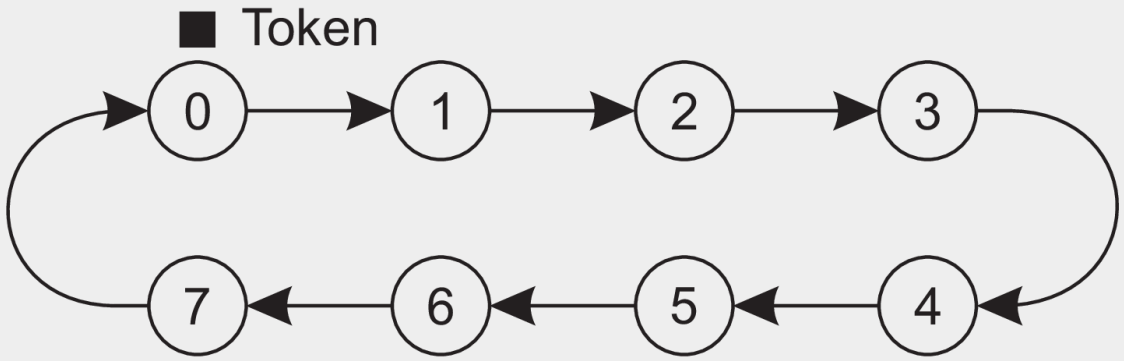
Distributed: pros and cons

- Pros
  - Mutual exclusion guaranteed
  - No starvation or deadlock

- Cons
  - Messaging overhead: (N-1) to request access, (N-1) OKs
  - N points of failure (requires a response from each other process)
    - Overcome by altering the algorithm: when request arrives receiver always replies (either grant/deny). Add a timeout to detect if request is lost and retry (or conclude it is dead)
  - Requires multicast communication (and each process must know of all other processes)

Token ring algorithm

- Organize processes in a logical ring, and let a token be passed between them
  - Token circulates around the ring from Pk to P(k+1)
- Process that currently holds the token is allowed to enter the critical region (if it wants to)



Token ring: pros and cons

- Pros
  - Guarantees mutual exclusion
  - Starvation cannot occur
  - Detecting failures is not too bad, when passing off the token a process can determine if their neighbor has failed
- Cons
  - When a process wants to access the resource it might have to wait for every other process
  - If the token is lost (e.g., because holding process fails) it must be regenerated
    - Even detecting that the token is lost might be hard as there is no known time for the token to circulate

Decentralized mutual exclusion

- Fully decentralized approaches can be constructed using a voting algorithm
- Require a majority vote from m > N/2 coordinators to access the resource
- If a coordinator does not want to give permission (e.g., as it has already given access) it will tell the requestor

Decentralized: pros and cons

- Pros
  - Without failures mutual exclusion can be guaranteed
  - No starvation or deadlock

- Cons
  - Issues if processes fail (and if they forget their votes)
    - How do we know? How do we guarantee that the quorum can be updated?
  - As a byproduct, the correctness of the voting mechanism can be violated if there is only a minority of non faulty coordinators
  - In practice the probability of violating correctness is low
  - Efficiency can be low if many processes are competing to access the same resource (e.g., non get sufficient votes many rounds in a row)

Agenda

- Part 1: Clock Synchronization
- Part 2: Logical clocks
- Part 3: Mutual exclusion
- Part 4: Election algorithms

Distributed Systems: Chapter 6

Election algorithms

- Many distributed algorithms require one process to act as a coordinator, initiator, or another role
  - E.g., mutual exclusion algorithms, time servers, consensus

- It doesn’t matter which process, but we need to guarantee one will do it

Election algorithms

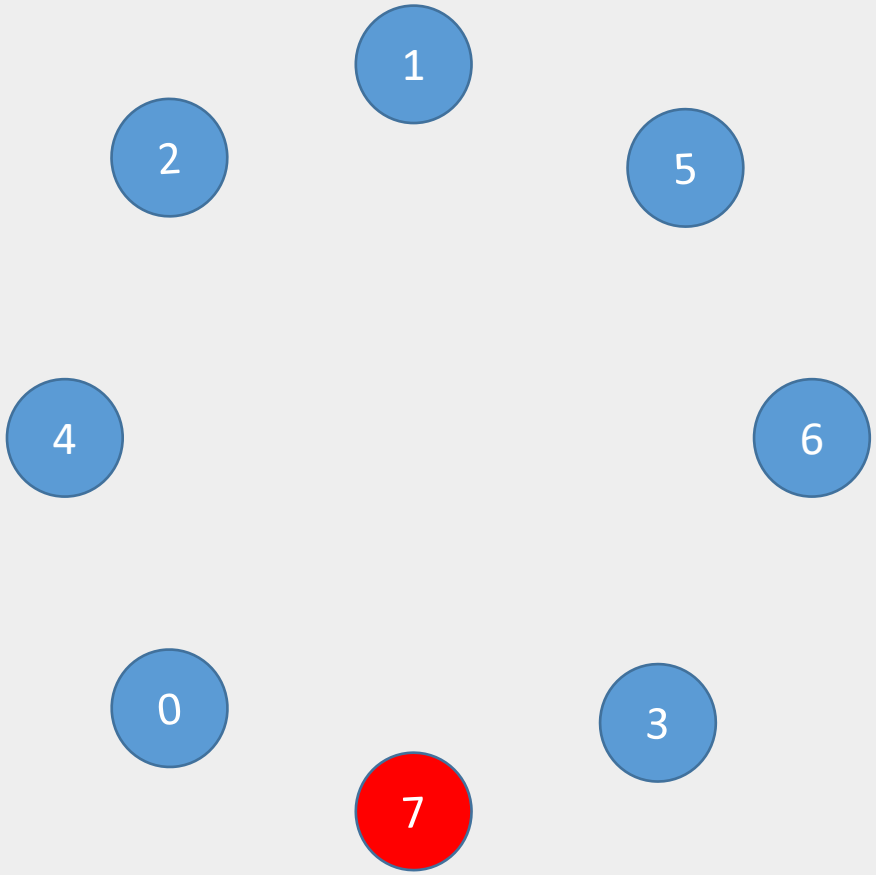
- Assumptions
  - All processes have a unique ID
  - All processes know the IDs of all other processes in the system
  - Processes do not know if another process is up or down
  - Our aim is to elect one of these processes
    - (often simplified as choosing the highest ID that is up)

- Bully: send election messages to all processes with higher identifiers, if no one replies, node wins election and becomes coordinator
- Ring: send election messages in a ring, after message goes around the entire ring, the one with the highest ID becomes coordinator

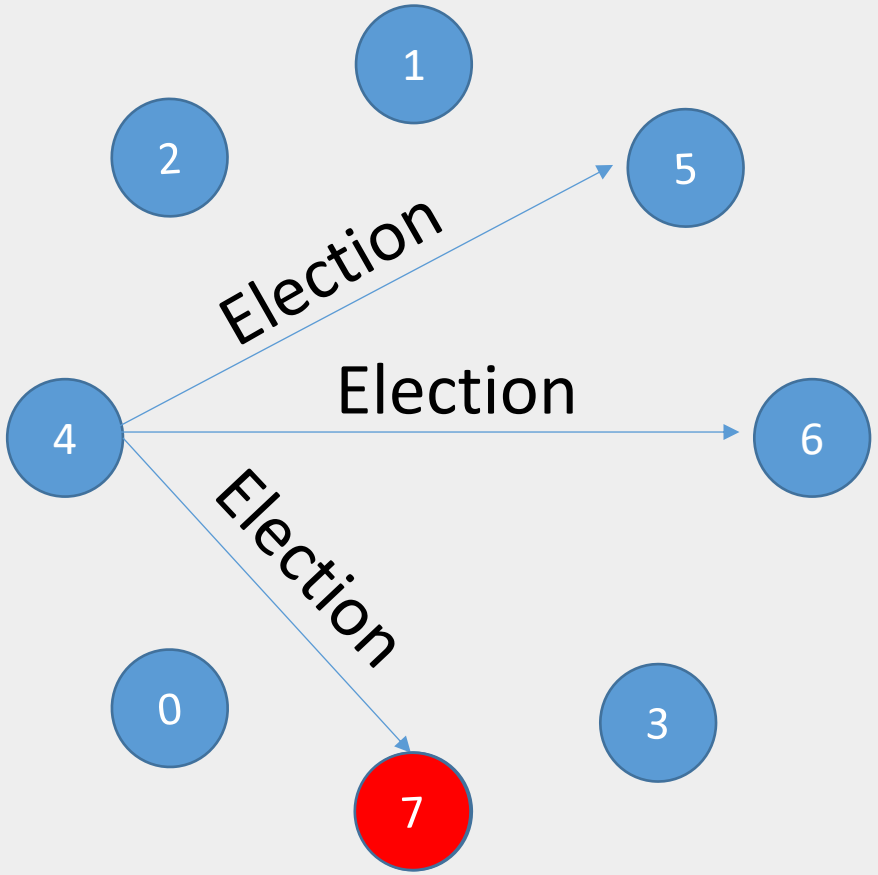
Bully algorithm

- Consider N processes {P0, … Pn-1} where id(Pk) = k
- When a process Pk notices that the coordinator is no longer responding to requests, it initiates an election:
  1. Pk sends an ELECTION message to all processes with higher identifiers:
     - Pk+1, Pk+2, …, Pn-1
  2. If no one responds, Pk wins the election and becomes coordinator
  3. If a process with higher ID answers, it takes over and Pk's job is done
- Winning process sends COORDINATOR message to all other processes
- If a process that was down comes back up, it holds an election
  - If it's the highest process it will win the election and take over as coordinator

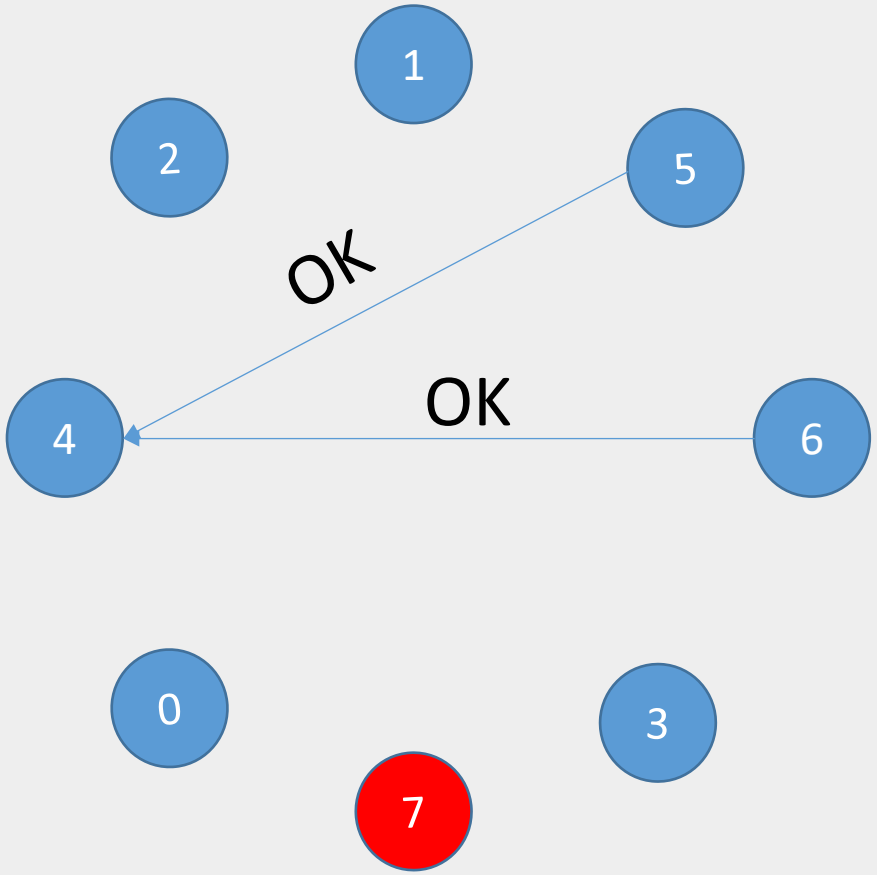
Bully algorithm



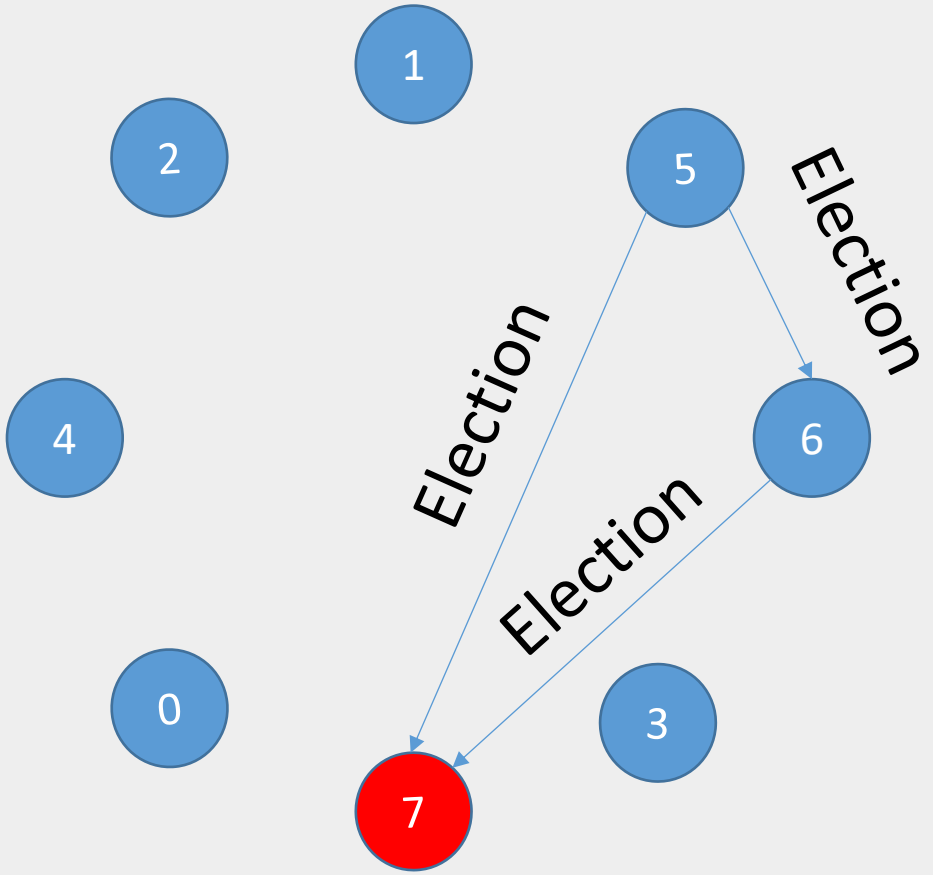
Bully algorithm



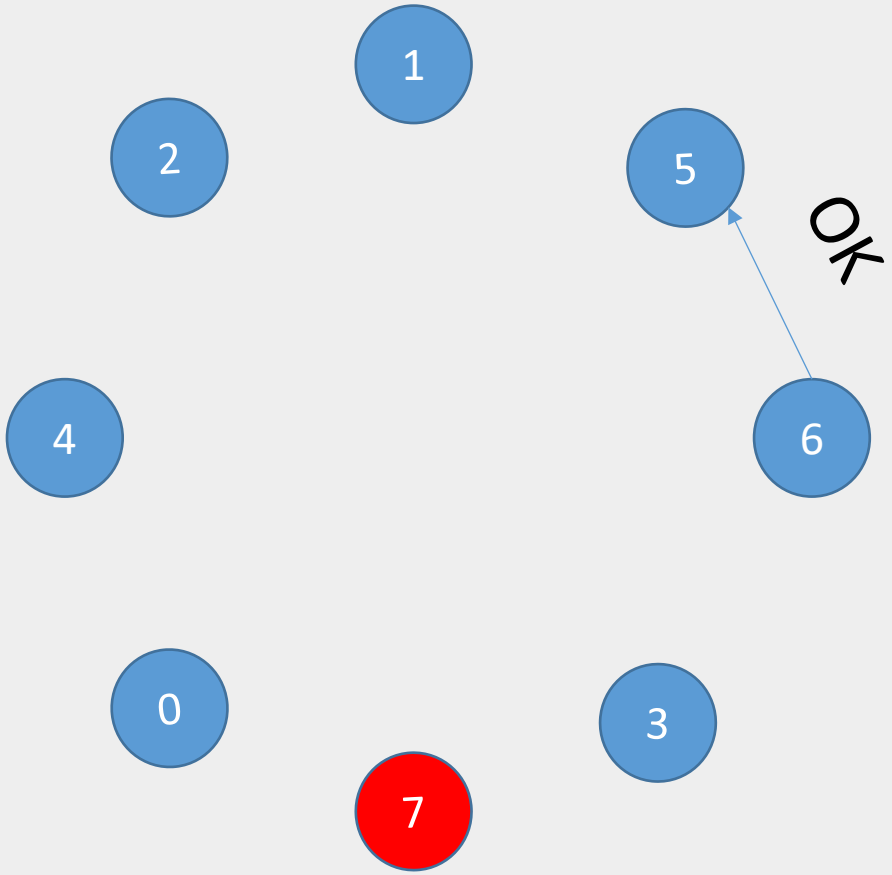
Bully algorithm



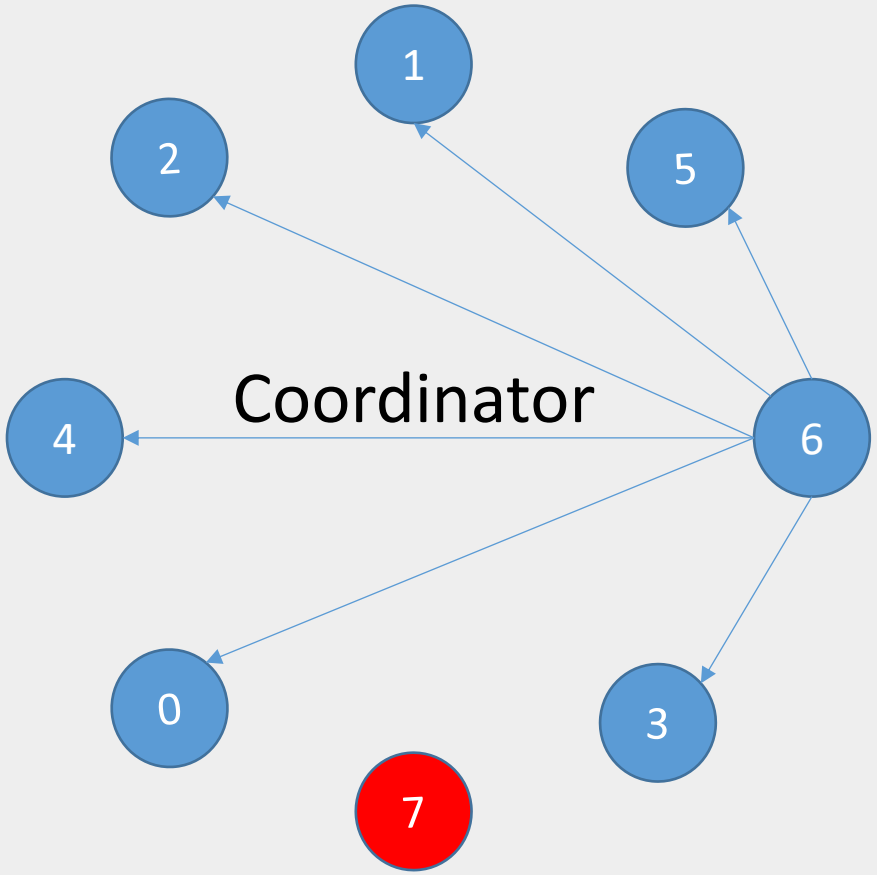
Bully algorithm



Bully algorithm



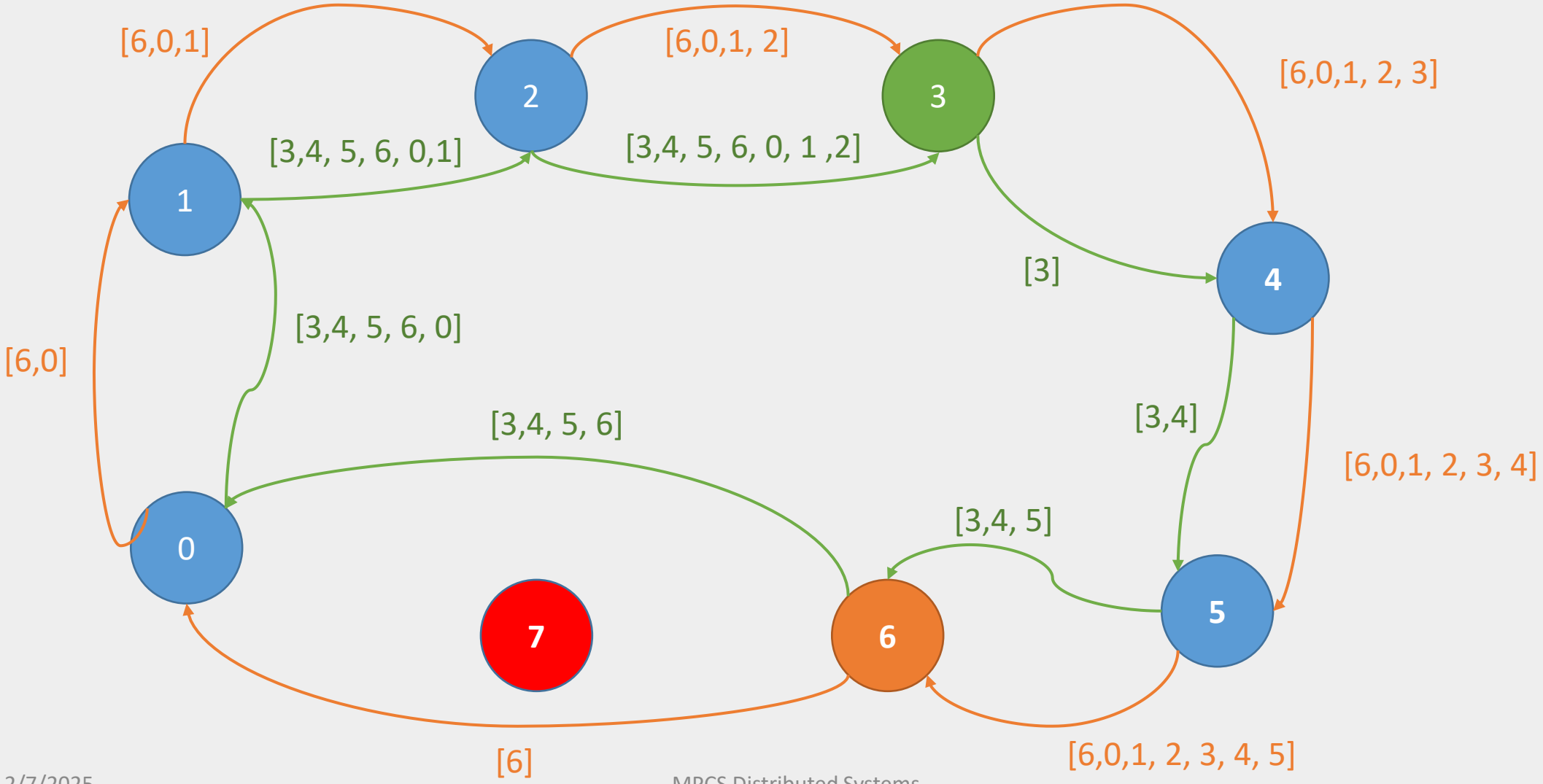
Bully algorithm



Ring algorithm

- Process priority is obtained by organizing processes into a (logical) ring and the process with the highest priority should be elected as coordinator.
  - Any process can start an election by sending an election message to its successor
    - If a successor is down, the message is passed on to the next successor
  - If a message is passed on, the sender adds itself to the list
    - When it gets back to the initiator, everyone had a chance to make its presence known
  - The initiator sends a coordinator message around the ring containing a list of all living processes
    - The one with the highest priority is elected as coordinator

Example of ring election



Homework 4

- Create a distributed password cracking service
- Use REST services to perform the cracking
- Client to manage distribution of cracking tasks

Passwords

- Cleartext passwords == bad
- One-way hashes enable comparison without revealing the password
- In practice we use salts and extending the length to reduce feasibility of brute force attacks

Hash functions

- Not the same as encryption
- Cryptographic hash function maps input data to a fixed size output
  - Hash(input) => output
  - Hash(‘mpcs’) => cce6b3fb87d8237167f1c5dec15c3133
- We cannot go from output to input
  - Unhash(cce6b3fb87d8237167f1c5dec15c3133) is not possible

Password cracking

- If we use a simple hash (MD5) then given a hashed password we can try each combination of strings to see if it matches
  - A, aa, ab, ac, … zz

- Implement password cracking in a REST service
- Create a client to distribute workload to many running REST services
- Add fault tolerance (if a REST service were to be removed)
- Analyze scaling performance
- Optimize by caching passwords seen before