Synchronization

Distributed Systems IT332

Outline

- Clock synchronization
- Logical clocks
- Election algorithms
- Mutual exclusion
- Transactions

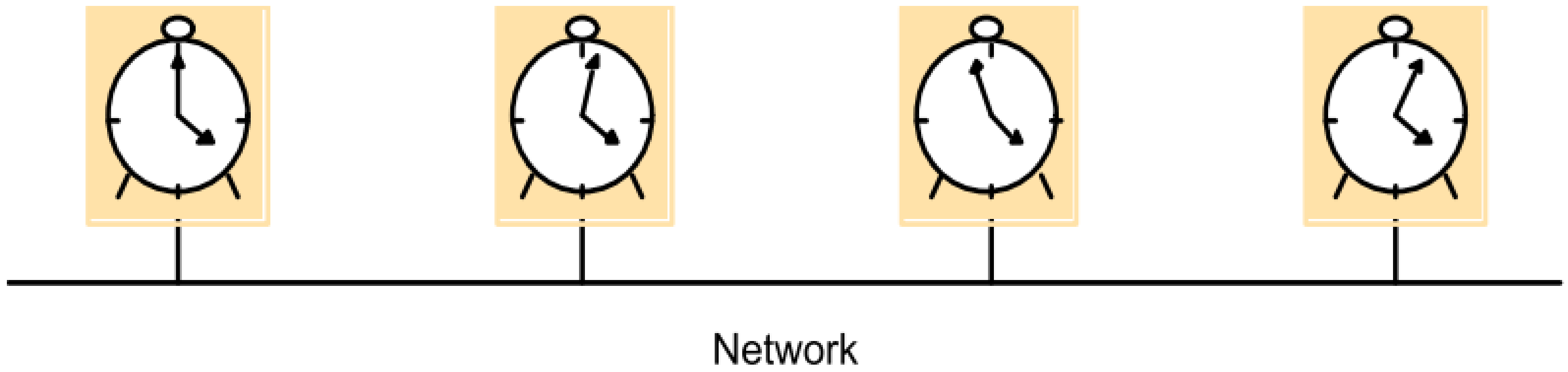
Hardware/Software Clocks

- Physical clocks in computers are realized as crystal oscillation counters at the hardware level.
  - Usually scaled to approximate physical time t, yielding software clock C(t), $C(t) = \alpha H(t) + \beta$
  - C(t) measures time relative to some reference event .
    - Example: 64 bit counter for # of nanoseconds since last boot
  - C(t) carries an approximation of real time, never $C(t) = t$.

Hardware/Software Clocks: problems

- Skew: difference between two clocks at one point in time.
- Drift: two clocks tick at different rates.
  - Create ever-widening gap in perceived time.
  - due to physical differences in crystals(quartz oscillators oscillate at slightly different frequencies, plus heat, humidity, voltage, etc.
  - Accumulated drift can lead to significant skew.
  - Clock drift rate: Difference in precision between a prefect reference clock and a physical clock.
    - Usually, 10^{-6} sec/sec, 10^{-7} to 10^{-8} for high precision clocks.

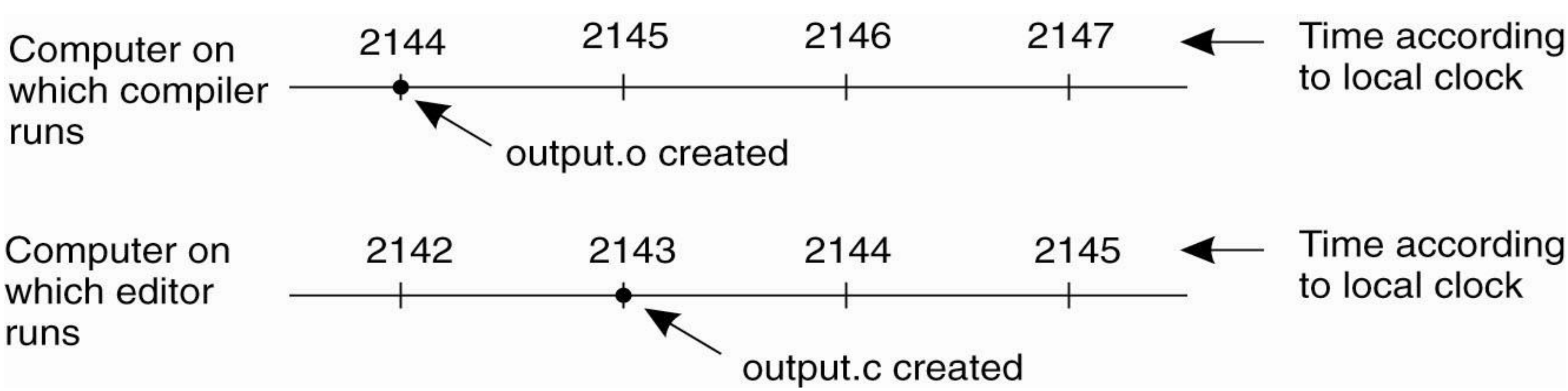
Hardware/Software Clocks



Skew between computer clocks in a distributed system

Clock Synchronization

- Time is unambiguous in centralized systems
  - System clock keeps time, all entities use this for time
- Distributed systems: each node has own system clock
  - Problem: an event that occurred after another event may nevertheless be assigned an earlier time.



Clock Synchronization

- Is it possible to synchronize all systems clocks together?

Clock Synchronization

- Clock Synchronization is a mechanism to synchronize the time of all the computers in a DS

- We will study
  - Coordinated Universal Time
  - Clock Synchronization Algorithms
    - Cristian's Algorithm
    - Berkeley Algorithm

Coordinated Universal Time (UTC)

- All the computers are generally synchronized to a standard time called Coordinated Universal Time (UTC)
  - UTC is the primary time standard by which the world regulates clocks and time. It is available via radio signal, telephone line, satellite (GPS)
- UTC is broadcasted via the satellites
  - UTC broadcasting service provides an accuracy of 0.5 msec
- Computer servers and online services with UTC receivers can be synchronized by satellite broadcasts
  - Many popular synchronization protocols in distributed systems use UTC as a reference time to synchronize clocks of computers

Clock Synchronization

- Need to synchronize machines with a UTC source or with one another
- External synchronization: Synchronize process's clock with an authoritative external reference clock $S(t)$ by limiting skew to a delay bound $\rho > 0$:

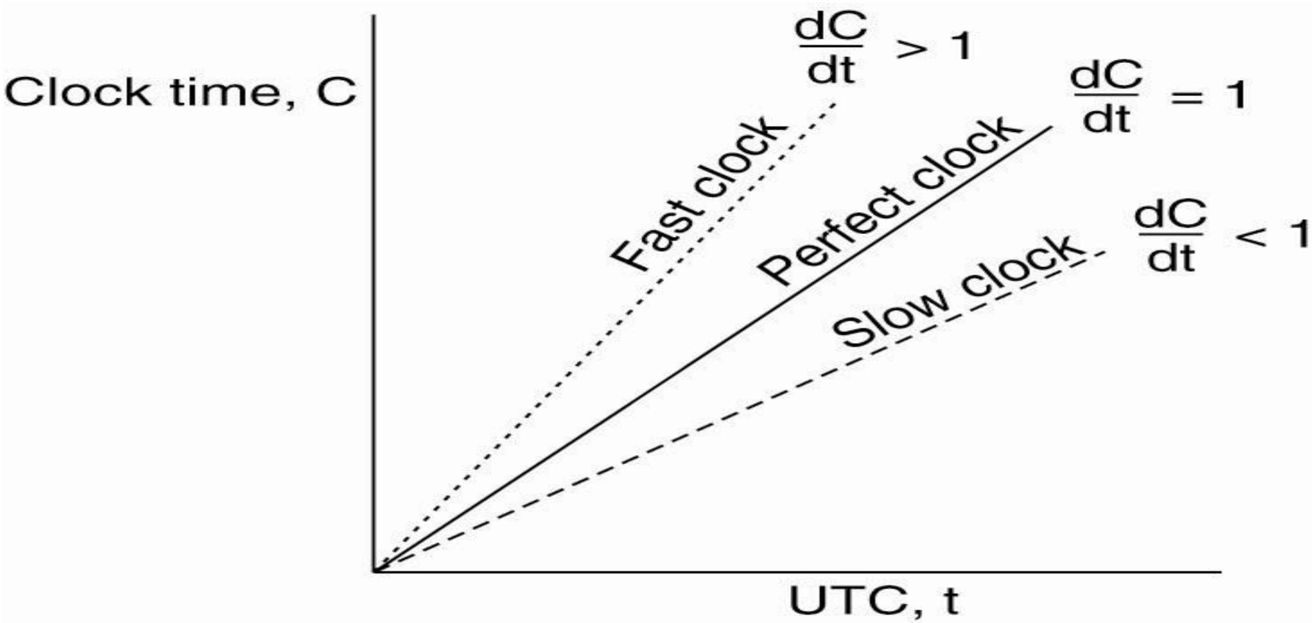
  → $|S(t) - C_i(t)| < \rho$ for all t

- Each clock has a maximum drift rate ρ : $1-\rho \le dC/dt \le 1+\rho$
- Internal Synchronization of the local clocks within a distributed system to disagree by not more than a delay bound $\rho > 0$:

  → $|C_i(t) - C_j(t)| < \rho$ for all i, j, t

- For a system with external synchronization bound of ρ , the internal synchronization is bounded by 2ρ
  - Two clocks may drift by 2ρ in time Δt
  - To limit drift to δ => resynchronize every $\delta/2\rho$ seconds

Clock Synchronization



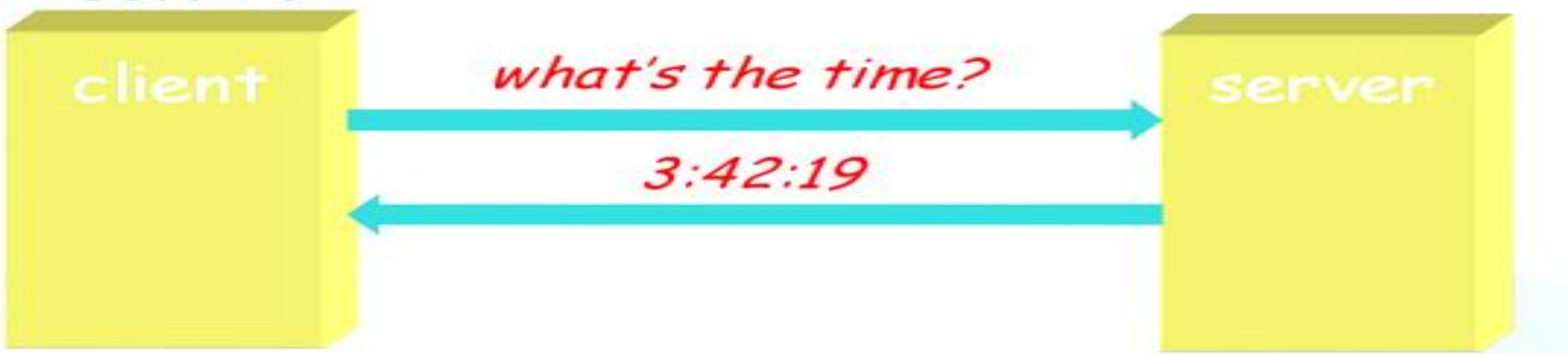
The relation between clock time (C) and UTC (t) when clocks tick at different rates.

Getting accurate time

- Attach GPS receiver to each computer
  - ± 1 msec of UTC
- Attach WWV radio receiver
  - Obtain time broadcasts from Boulder or DC
  - ± 3 msec of UTC (depending on distance)
- Attach GOES receiver
  - ± 0.1 msec of UTC
- Not practical solution for every machine
  - Cost, size, convenience, environment

RPC

- Synchronize from another machine
  - One with a more accurate clock
- Machine/service that provides time information:
- Simplest synchronization technique
  - Issue RPC to obtain time
  - Set time

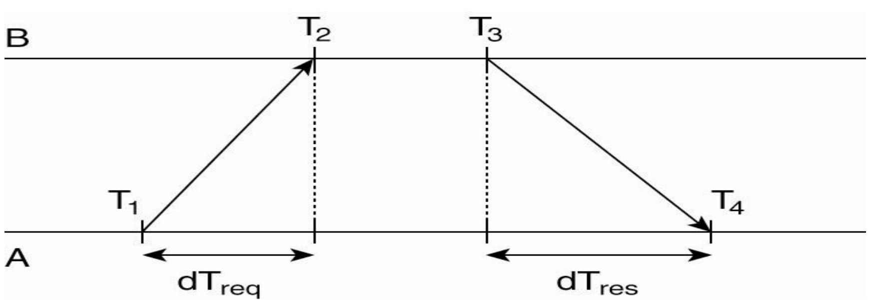


Does not account for network or processing latency

Cristian's Algorithm

- Use UTC-synchronized time server S
- The time server is passive
- Widely used in LAN.
- Assume that networks delays are symmetric
- Machine A periodically requests time from server B: T1: request sent and T4: reply received.
  - A receives time T_2 and T_3 from server, sets clock to T_3+T_{res} where T_{res} is the time to send reply from B to A.
  - Use (T_{req}+T_{res})/2 as an estimate of T_{res}:

$$T_{res} = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

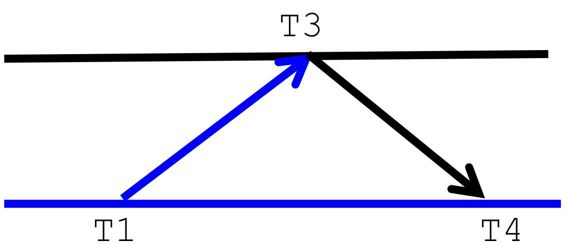


Network Time Protocol (NTP)

- Widely used standard based on Cristian's algorithm
- Improve accuracy by making 8 measurements, take the minimum value for T_{res} as the best estimate for the delay between the two machines.
- A hierarchy of time servers: a time server in level k synchronizes with a server in level ≤ k-1
  - Level 0 is the atomic clock
- Can we synchronize clocks backward?
  - Clock cannot go backwards: If time needs to be adjusted backward, slow down the clock until time catches up.

Example

- At 5:08:15.100, server B requests time from the time-server A. At 5:08:15.900, server B receives a reply from timeserver A with the timestamp of 5:09:25.300 . (assume there is no processing time at the time-server)
  - Send request at 5:08:15.100 (T1)
  - Receive response at 5:08:15.900 (T4)
  - Response contains 5:09:25.300 (T3)



- What would be the time at server B after synchronization?

Answer

- Send request at 5:08:15.100 (T_0)
- Receive response at 5:08:15.900 (T_1)
  - Response contains 5:09:25.300 (T_{server})

- Elapsed time is $T_1 - T_0$

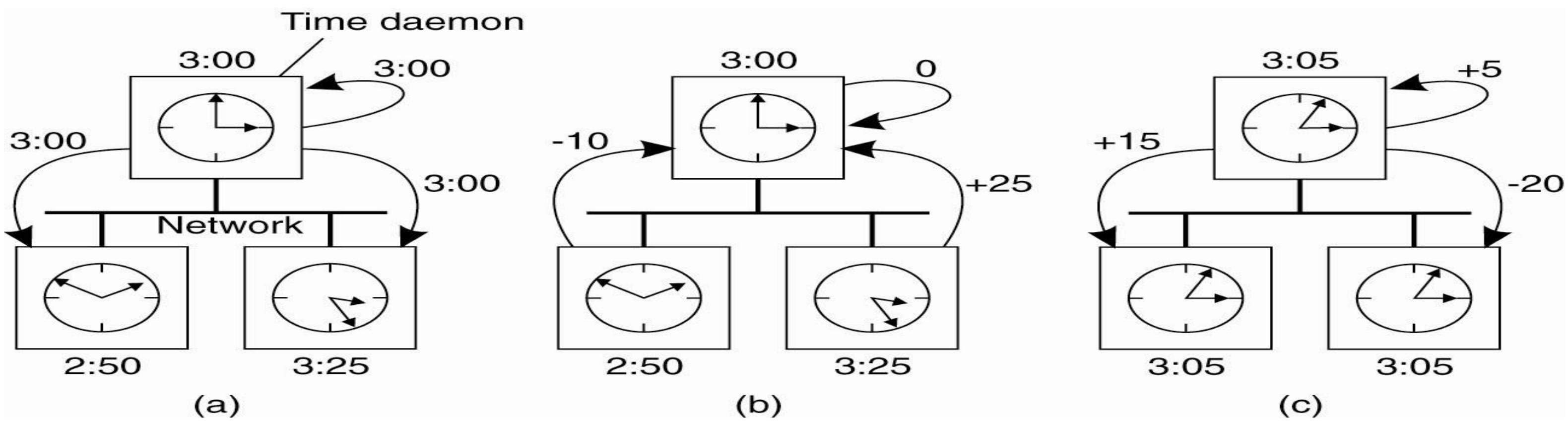
  5:08:15.900 - 5:08:15.100 = 800 msec
- Best guess: timestamp was generated

  400 msec ago
- Set time to T_{server}+ *elapsed time*

  5:09:25.300 + 400 = 5:09.25.700

Berkeley Algorithm

- Keep clocks synchronized with one another
- Assumes that no computer has an accurate time source.
- Machines run Time Daemon.
- One computer is elected as master, others are slaves.
- Master periodically polls slaves for their times and calculate average (including its time)
- Return differences to slaves to synchronize all clocks to average.
- Failure of master => election of a new master



a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock

- If two machines do not interact ever. Do we need to synchronize them?

Logical Clocks

- If two machines do not interact, there is no need to synchronize them
- What usually matters is that processes agree on the order in which events occur rather than the time at which they occurred
  - Absolute time is not important
  - Use logical clocks
  - No concept of happened- when

Event Ordering

- Problem: define a global ordering of all events that occur in a system
- Events in a single process or machine are locally ordered
- In a distributed system:
  - No global clock, local clocks may be unsynchronized
  - Can not order events on different machines using local times
- Key idea [Lamport]
  - Processes exchange messages
  - Message must be sent before received
  - Send/receive used to order events and synchronize logical clocks

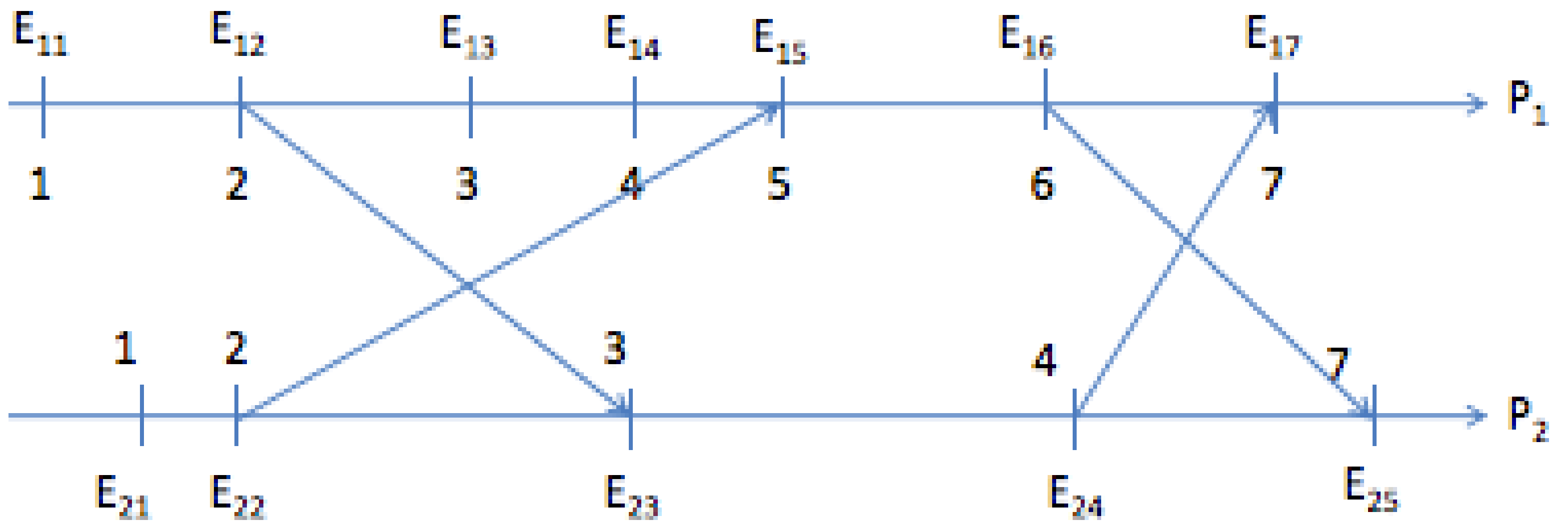
Happened-Before (HB) Relation

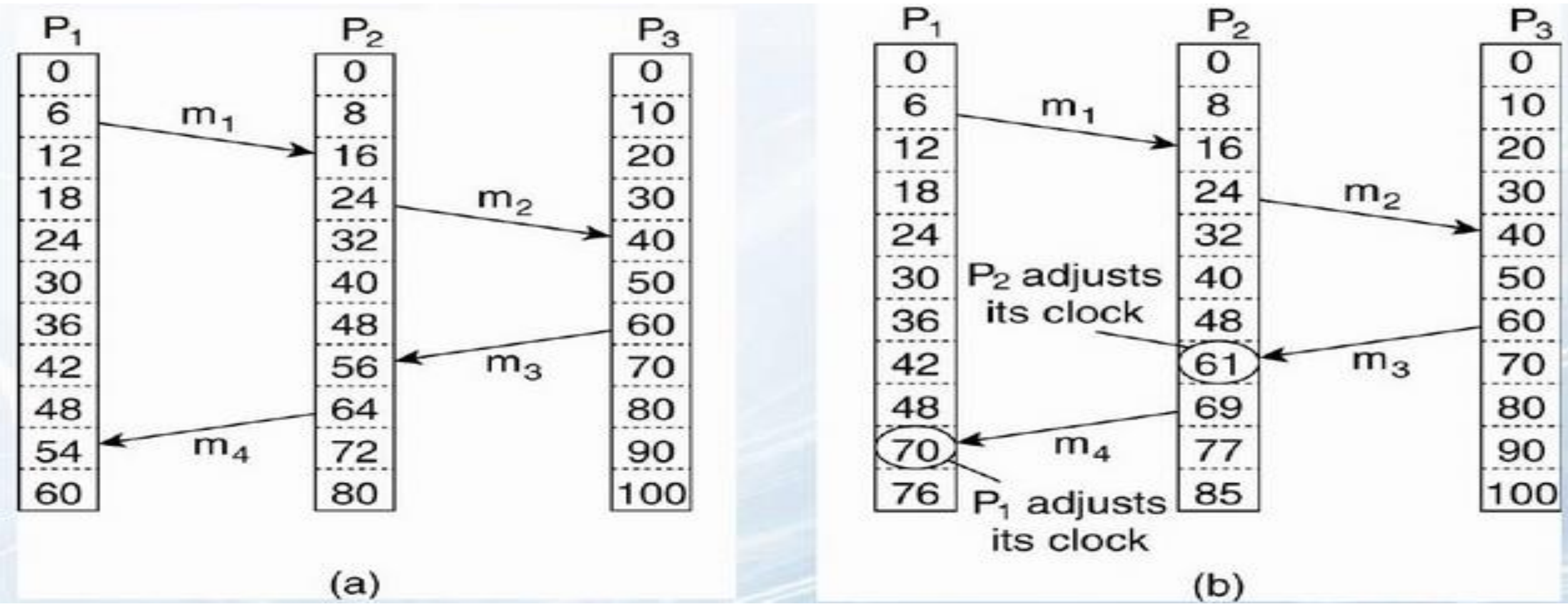
- If A and B are events in the same process and A occurs before B, then A→B
- If A represents sending of a message and B is the receipt of this message, then A→B (clock(A)< clock (B)
- Relation is transitive:
  - A→B and B→C implies A→C
- Unordered events are concurrent
  - A !→B and B !→A implies A || B

Lamport's Logical Clocks

- Goal: assign timestamps to events such that
  If A→ B then timestamp(A) < timestamp(B)
- Lamport's Algorithm
  - Each process i maintains a logical clock Li
  - Whenever an event occurs locally at i, Li = Li+1
  - When i sends message to j, piggyback Li
  - When j receives message from i, Lj =max (Li, Lj) + 1
- In this algorithm, A→B implies L(A) < L(B), but L(A) < L(B) does not necessarily imply A→B

Lamport's Logical Clocks: An Example

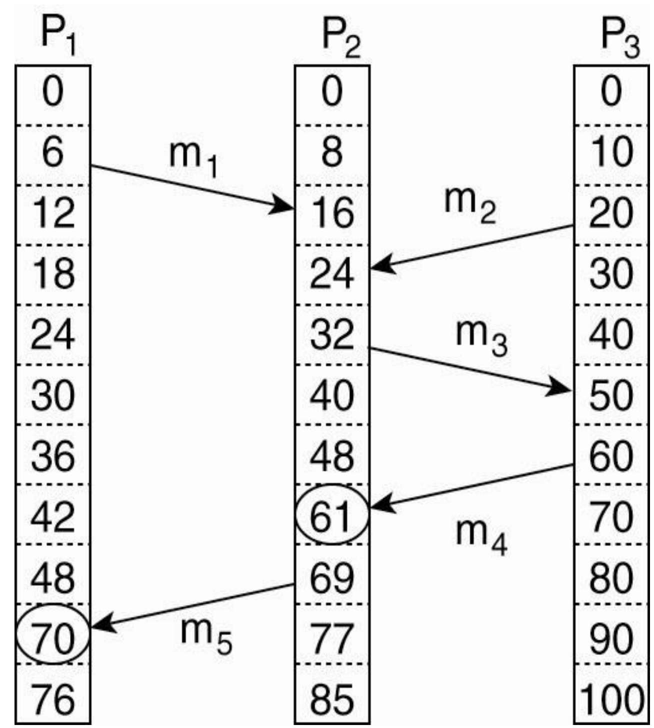




(a) Three processes, each with its own clock. The clocks run at different rates.
(b) Lamport's algorithm corrects the clocks.

Causality

- Lamport's logical clocks:
  - If A→B then L(A) < L(B)
  - Problem: Reverse is not true!!
    - Nothing can be said about events by comparing timestamps!
- Need to capture causality
  - If A→B then A causally precedes B
  - Need a timestamping mechanism such that:
    - T(A) < T(B) iff A causally precedes B



Event A:m1 is received at t=16
Event B:m2 is sent at t=20
L(A)<L(B), but A does not causally precede B.

Solution: Vector Clocks

- Each process i maintains a vector clock Vi of size N, where N is the number of processes
  - Vi[i] = number of events that have occurred at process i
  - Vi[j] = number of events i knows have occurred at process j
  - Initially, Vi[j]= 0 for all i, j = 1,..,N

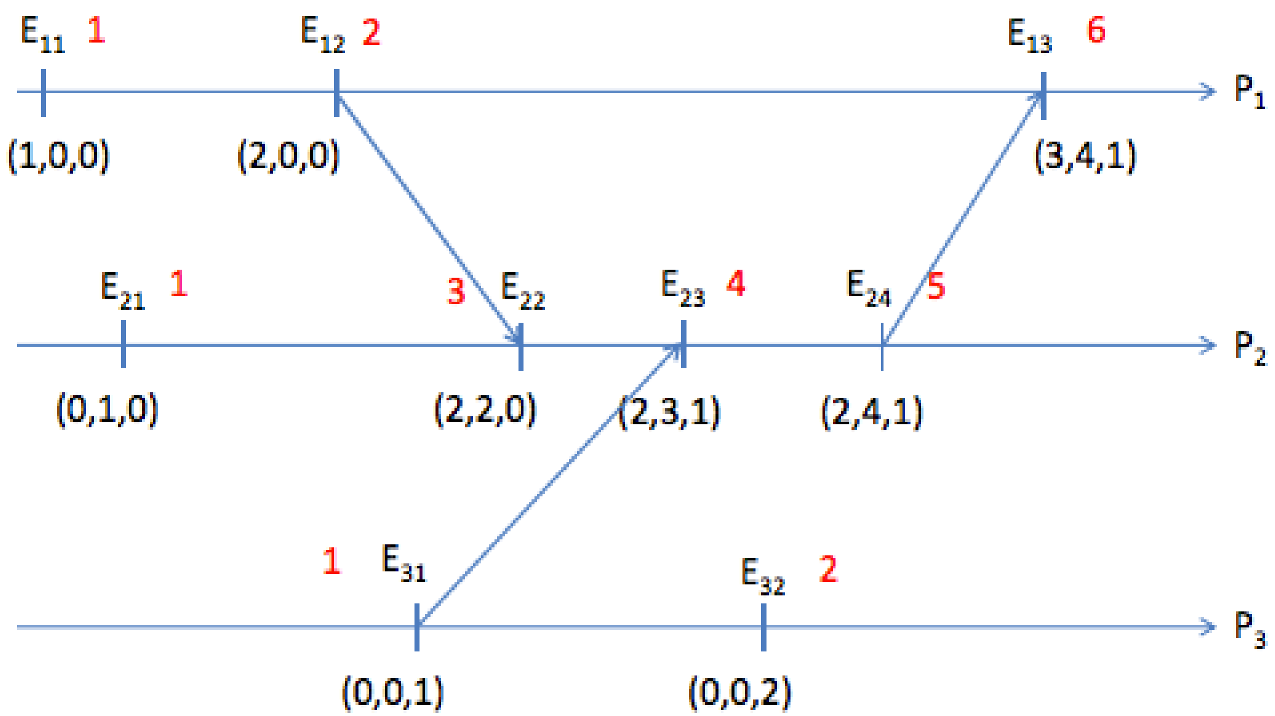
- Update vector clocks as follows:
  - Local event at pi: increment Vi [i] by 1 (before time stamping event)
  - When a message is sent from pi to pj : piggyback vector Vi
  - Receipt of a message from pi by pj: *Vj [k] =max(Vj[k],Vi [k]), j≠k ; Vj [j] = Vj [j]+1*

  Receiver is told about how many events the sender knows occurred at another process k

- We have V(A)<V(B) iff A causally precedes B!
  - V(A)<V(B) iff for all i, V(A)[i] ≤ V(B)[i] and there exists k such that V(A)[k] < V(B)[k]

- A and B are concurrent iff V(A)!< V(B) and V(B)!< V(A)

Vector Clock: An Example



Election Algorithms

- Many distributed algorithms need one process to act as a leader or coordinator
  - How to select this process dynamically
  - Doesn’t matter which process does the job, just need to pick one
  - Example: pick a master in Berkeley clock synchronization algorithm
- Election algorithms: technique to pick a unique coordinator
  - Assumption: each process has a unique ID
  - Goal: find the non-crashed process with the highest ID

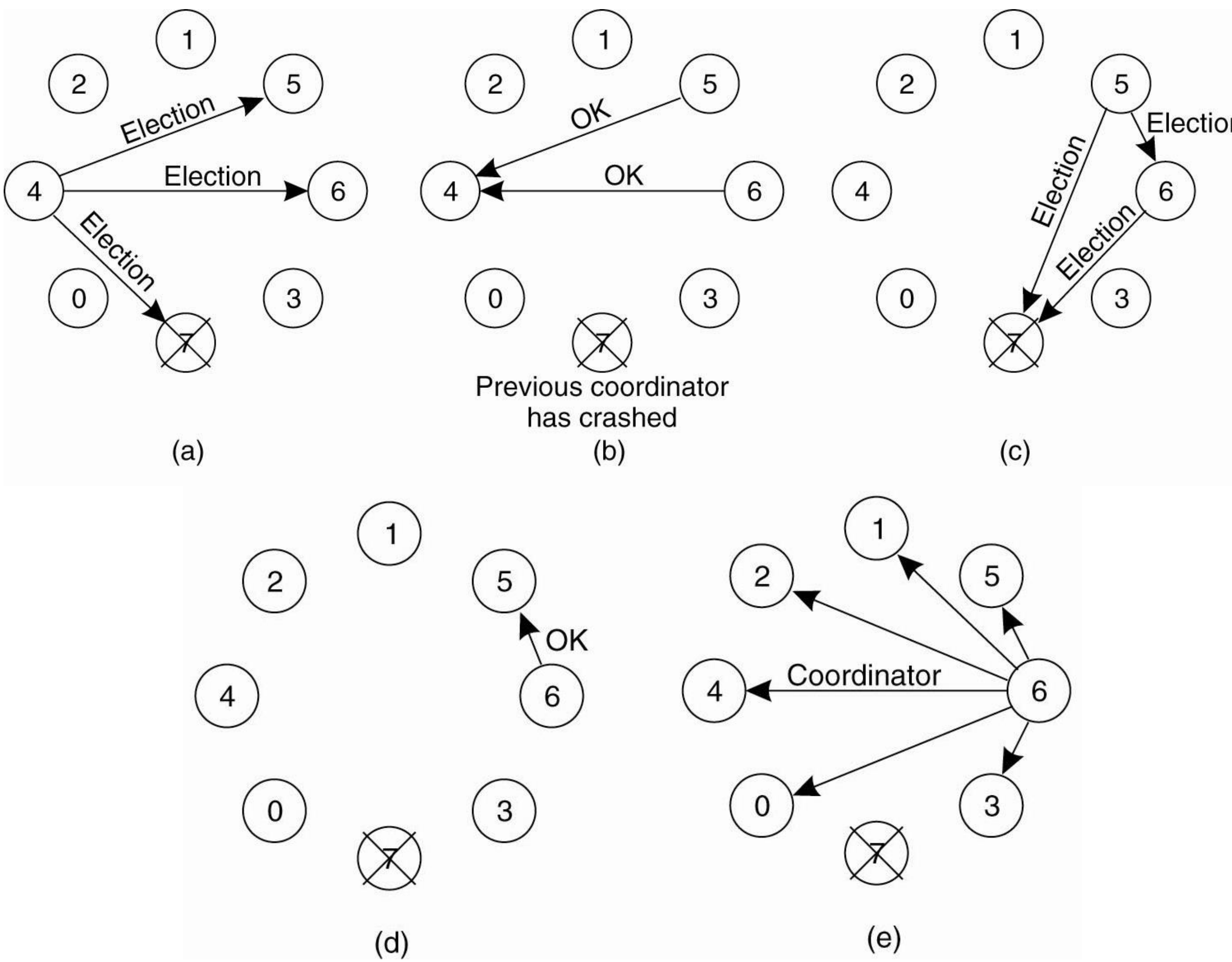
Bully Algorithm

- Assumptions
  - Each process knows the ID and address of every other process
  - Communication is reliable
- A process initiates an election if it just recovered from failure or it notices that the coordinator has failed
- Three types of messages: Election, OK, Coordinator
- Several processes can initiate an election simultaneously
  - Need consistent result

Bully Algorithm Details

- Any process P can initiate an election
- P sends Election messages to all process with higher IDs and awaits OK messages
  - If no OK messages, P becomes coordinator and sends Coordinator messages to all processes with lower IDs
  - If it receives an OK, it drops out and waits for an Coordinator message
- If a process receives an Election message
  - Immediately sends Coordinator message if it is the process with highest ID
  - Otherwise, returns an OK and starts an election
- If a process receives a Coordinator message, it treats sender as the coordinator

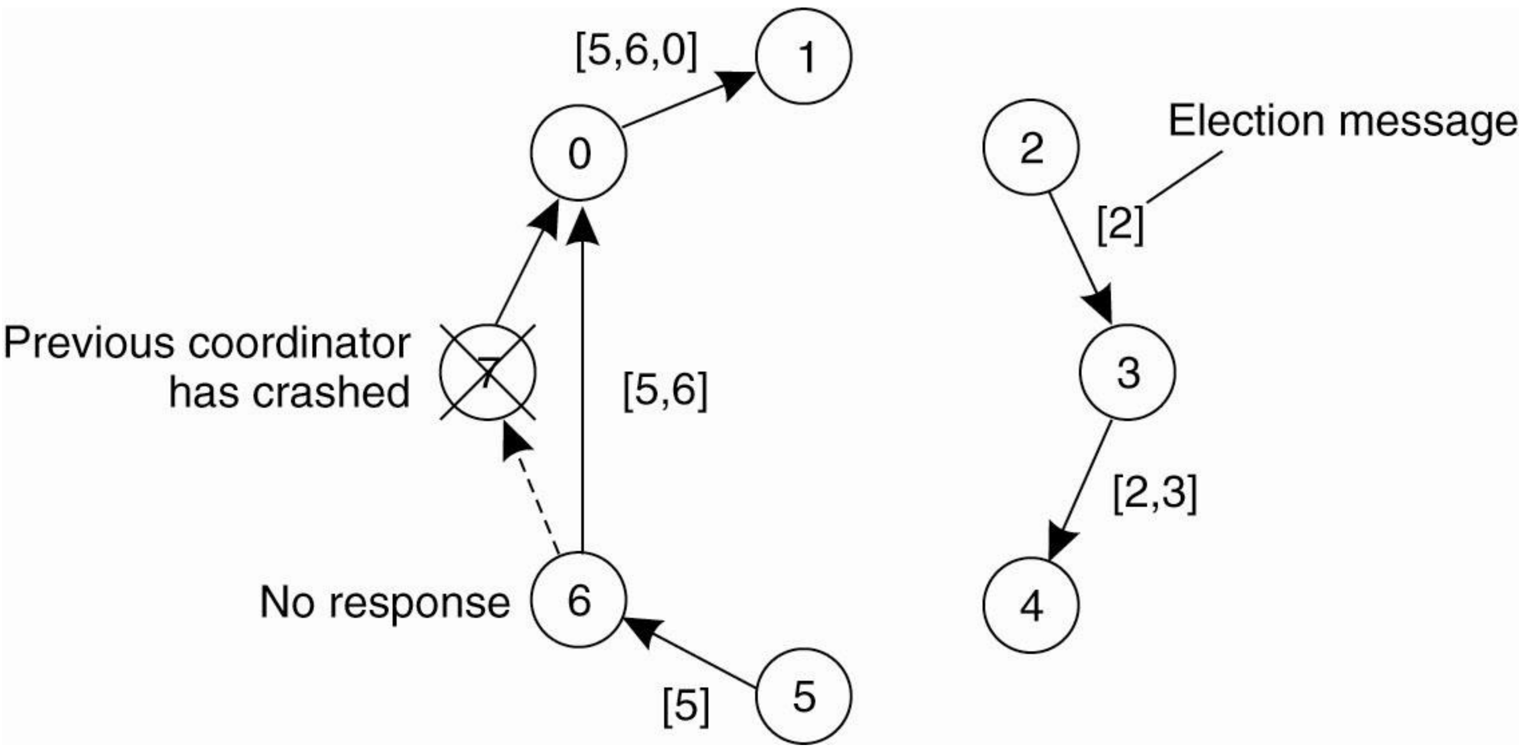
Bully Algorithm Example



Ring Algorithm

- Processes are arranged in a logical ring, each process knows the structure of the ring
- A process initiates an election if it just recovered from failure or it notices that the coordinator has failed
- Initiator sends Election message to closest downstream node that is alive
  - Election message is forwarded around the ring
  - Each process adds its own ID to the Election message
- When Election message comes back, initiator picks node with highest ID and sends a Coordinator message specifying the winner of the election
  - Coordinator message is removed when it has circulated once.
- Multiple elections can be in progress

Ring Algorithm Example

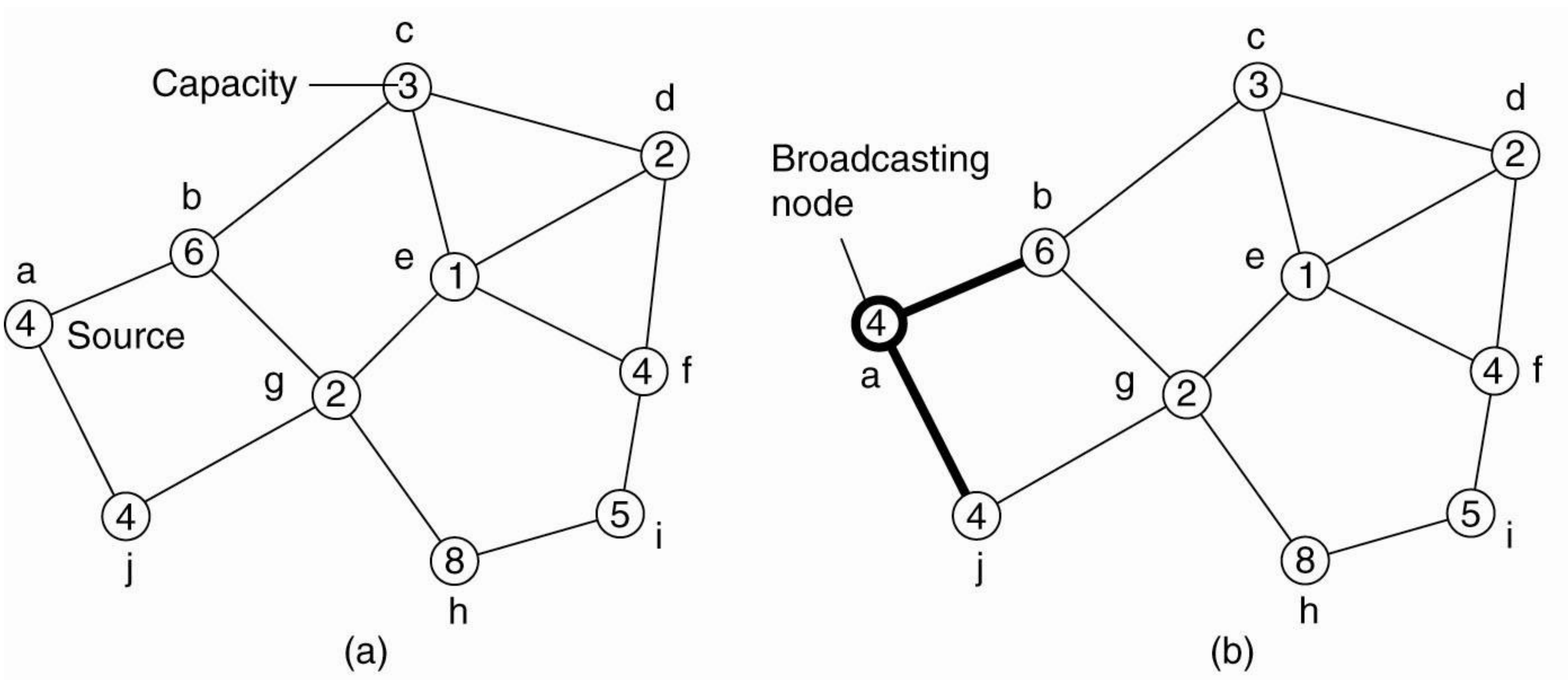


Comparison of Bully and Ring Algorithms

- Assume n processes and one election in progress
- Bully algorithm
  - Worst case: initiator is node with lowest ID
    - Triggers n-2 elections at higher ranked nodes: $O(n^2)$ messages
  - Best case: initiator is node with highest ID
    - Immediate election: n-1 messages
- Ring algorithm
  - 2n messages always

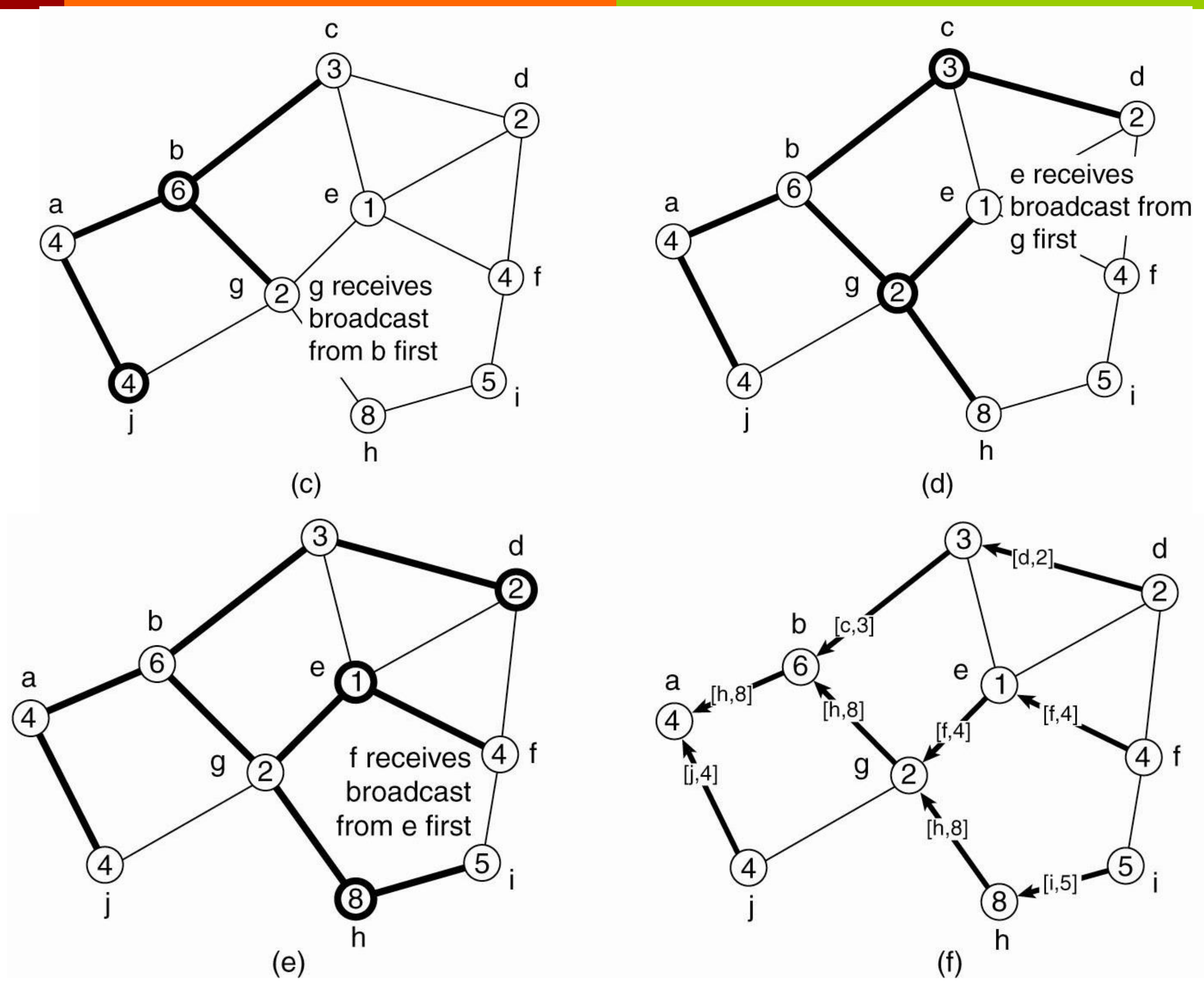
Election in Wireless Environments

- Goal: elect the best leader (e.g., node with longest battery lifetime)



Node a initiates an election.

Election in Wireless Networks



In the end, source a notes that h is the best leader and broadcasts this info to all nodes.

Mutual Exclusion

- Processes in a distributed system may need to simultaneously access the same resource
- Need to grant mutual exclusive access to shared resources by processes
- Solutions:
  - Via a centralized server (Centralized algorithm)
  - Decentralized, using a peer-to-peer system (Decentralized algorithm)
  - Distributed, with no topology imposed (Distributed algorithm)
  - Distributed, along a logical ring (A token ring algorithm)

Terminology

- In concurrent programming a critical section is a piece of code that accesses a shared resource that must not be concurrently accessed by more than one thread of execution.
- Mutual exclusion (ME, often abbreviated to mutex) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.

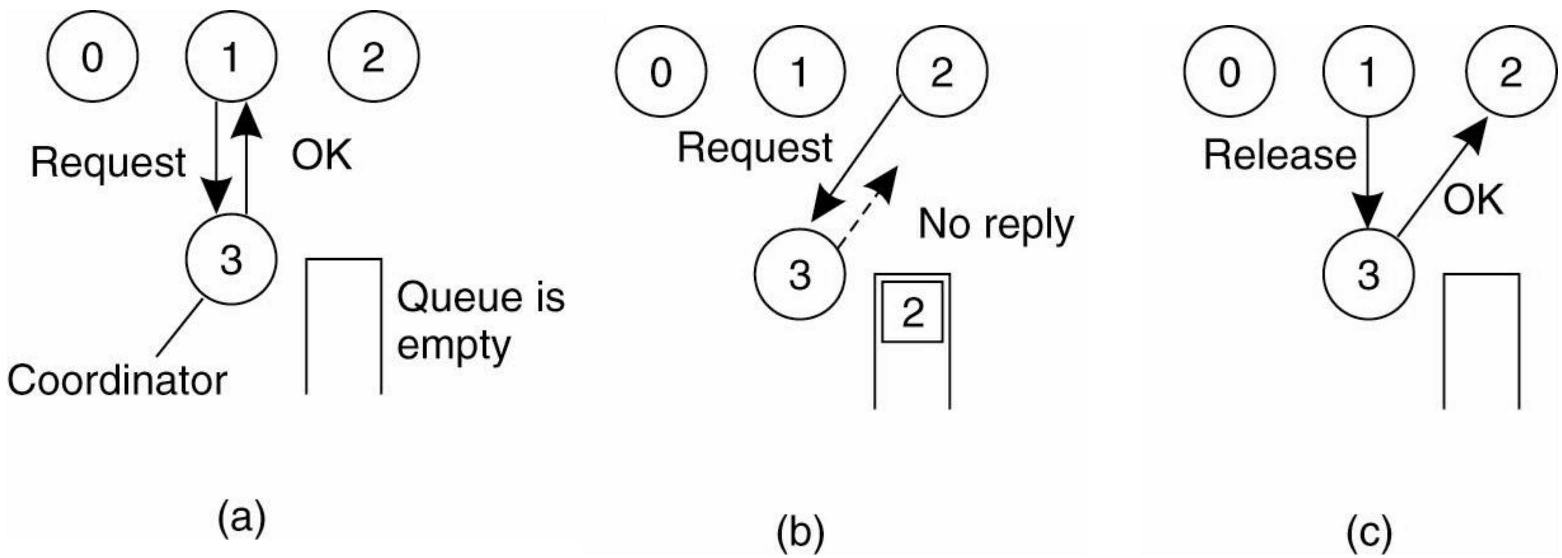
Mutual Exclusion

- Prevent simultaneous access to a resource
- Two basic kinds:
  - Permission based
    - A Centralized Algorithm
    - A Decentralized Algorithm
    - A Distributed Algorithm
  - Token based
    - A Token Ring Algorithm

Centralized Mutual Exclusion

- Assume processes are numbered
- One process is elected coordinator
- Every process needs to check with coordinator before entering the critical section
- To obtain exclusive access: send request, await reply
- To release: send release message
- Coordinator:
  - Receive request: if resource is available and queue empty, sendOK; if not, queue request
  - Receive release: remove next request from queue and sendOK

Centralized Mutual Exclusion



a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.

b) Process 2 then asks permission to access the same resource. The coordinator does not reply.

c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

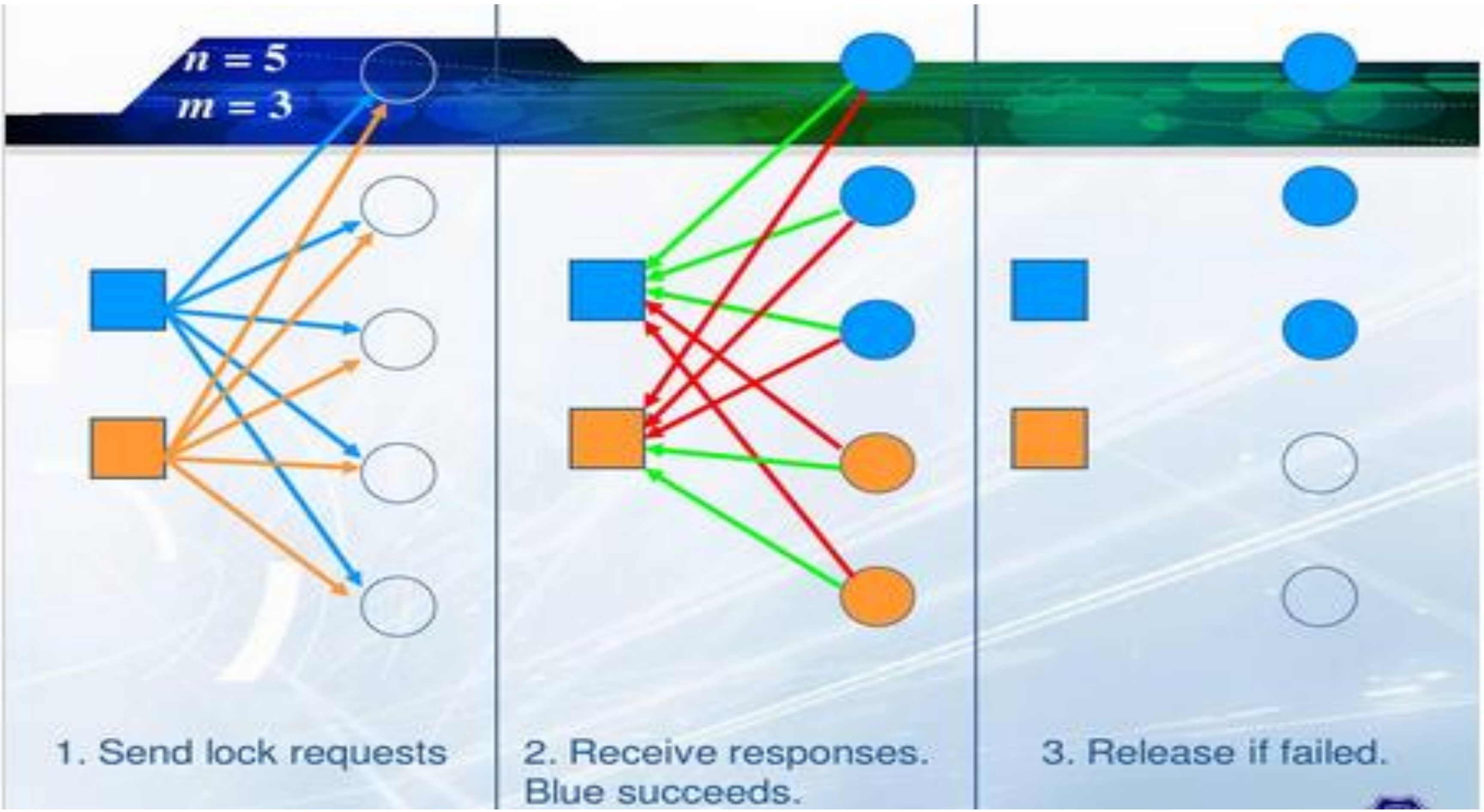
Properties of Centralized Mutual Exclusion

- Simulate centralized locking with blocking calls
- Advantages
  - Fair: requests are granted in the order they were received
  - Simple: three messages per use of a resource (request, OK, release)
  - No starvation
- Drawbacks:
  - Single point of failure
  - Performance bottleneck in large distributed systems
  - How do you detect a dead coordinator?
    - A process can not distinguish between "permission denied" from a dead coordinator – No response from coordinator in either case

A Decentralized Algorithm

- Each resource is replicated n times. Each replica has its own coordinator
- Access requires majority vote from m> n/2 coordinators.
  - Nonblocking: coordinators return OK or “no”
- Coordinator crashes => forgets previous votes (i.e., resets itself)
- If request is denied, process will back off for a randomly-chosen time, and try again
- Drawbacks
  - Low resource utilization when many nodes want to access the same resource
  - Starvation can occur

Example

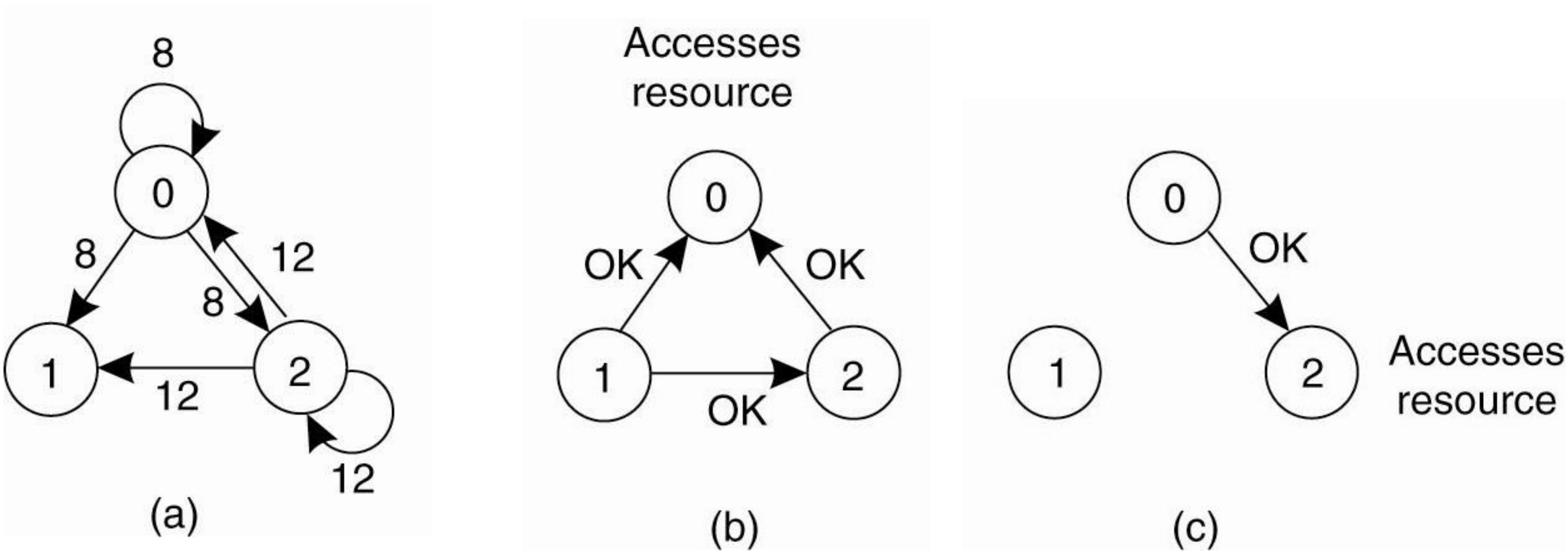


$n = 5$
$m = 3$
1. Send lock requests
2. Receive responses.
Blue succeeds.
3. Release if failed.

A Distributed Algorithm

- Based on event ordering and time stamps
  - Each process maintains a logical clock
- Process k enters critical section as follows
  - Increment logical clock: $L_k = L_k + 1$
  - Multicast a message (L_k ,k)to all other processes
  - Wait until a reply is received from every other process
  - Enter critical section
- Upon receiving a request message, process j
  - Sends an OK message if outside of critical section
  - If already in critical section, does not reply, queue the request
  - If wants to enter critical section, sends an OK message if(L_k ,k) < (L_j ,j), else queue the request
- When a process is finished with the critical section
  - Send OK messages to all processes on its queue and delete them from the queue

A Distributed Algorithm



a) Two processes want to access a shared resource at the same moment.

b) Process 0 has the lowest timestamp, so it wins.

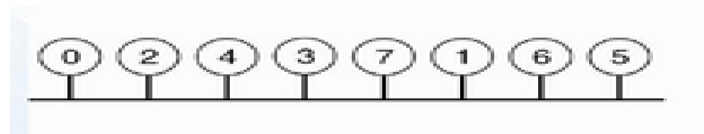
c) When process 0 is done, it sends OK to 2,so 2 can now go ahead

Properties of Distributed Algorithm

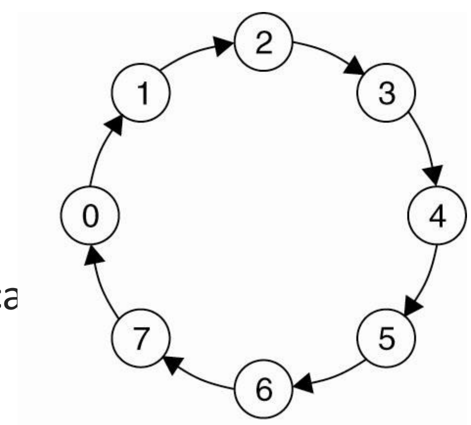
- Fully distributed
- N points of failure!
- All processes are involved in all decisions
  - Any overloaded process can become a bottleneck
- Improvements
  - Shows that a fully distributed system is possible.
  - When a request comes in, always sends a reply granting or denying permission.
    - This helps detect dead processes
  - Enter critical section when the process has got permission from a simple majority of the other processes

A Token Ring Algorithm

- Assume known group of processes
  - Some order can be imposed
  - Construct logical ring in software
  - Process communicates with neighbour



- Use a token to arbitrate access to critical section
- P_0 gets token to Resource R
- Token circulates around the ring: P_i passes to P_{i+1} mod N
- Must wait for token before entering CS
- Process which acquires a token cheks if it needs the critica
  - If No: Pass the token to neighbor
  - If yes: access resource, holds token until done



Features

- Only one process at a time has a token
  - ME is guaranteed
- Order well defined
  - No starvation!
- If token is lost (e.g:Process died)
  - It will be have regenerated
- Detecting token loss is non-trivial
- Doesn’t guarantee FIFO order (sometimes it is undesirable)
- Failing nodes can break the ring

Comparison

Algorithm	Delay before entry	Messages per entry / exit	Problems
Centralized	2	3	Coordinator crashes
Decentralized	2mk	3mk	Starvation, Low efficiency
Distributed	2(n-1)	2(n-1)	Crash of any process
Token Ring	0 to n-1	1 to infinity	Token may be lost, Ring can be broken if processes crash