Distributed Algorithms

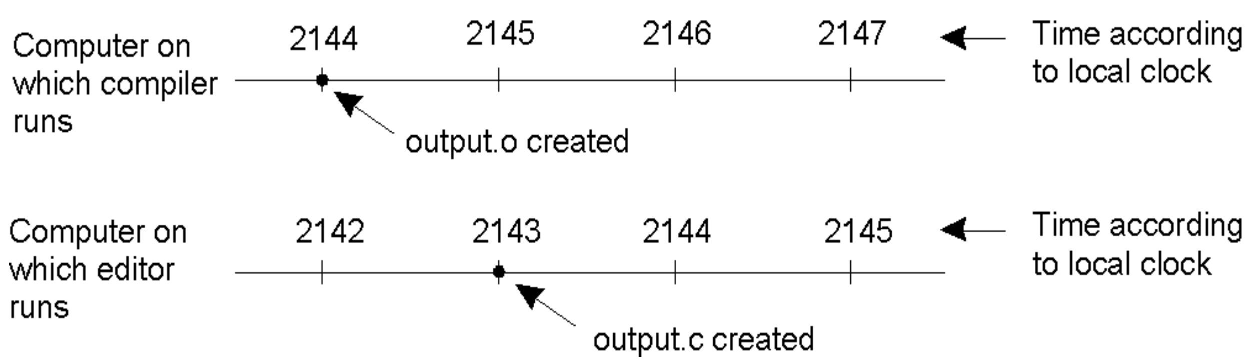
Distributed Algorithms

- Clock Synchronization
- Leader Election
- Mutual Exclusion

Clocks Synchronization

Clock Synchronization

- Time in unambiguous in centralized systems
  - System clock keeps time, all entities use this for time
- Distributed systems: each node has own system clock
  - Crystal-based clocks are less accurate (1 part in million)
  - *Problem:* An event that occurred after another may be assigned an earlier time

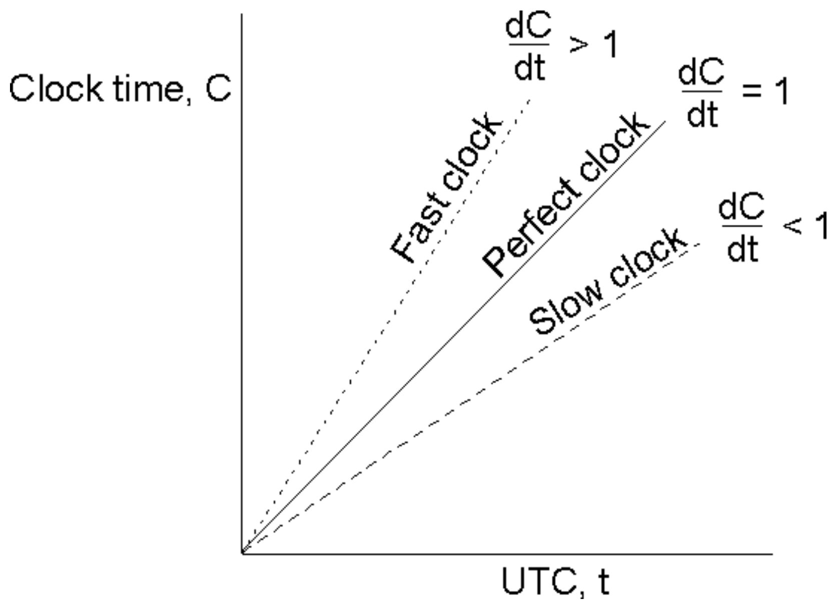


Physical Clocks: A Primer

- How do you tell time?
  - Use astronomical metrics (solar day)
- Accurate clocks are atomic oscillators (one part in 10^{13})
- Coordinated universal time *(UTC)* – international standard based on atomic time
  - Add leap seconds to be consistent with astronomical time
  - UTC broadcast on radio (satellite and earth)
  - Receivers accurate to 0.1 – 10 ms
- Most clocks are less accurate (e.g., mechanical watches)
  - Computers use crystal-based blocks (one part in million)
  - Results in *clock drift*
- Need to synchronize machines with a master or with one another

Clock Synchronization

- Each clock has a maximum drift rate ρ
  - $1-\rho <= \delta X/\delta\tau <= 1+\rho$
  - Two clocks may drift by $2\rho\ \Delta\tau$ in time Δt
  - To limit drift to δ => resynchronize every $\delta/2\rho$ seconds

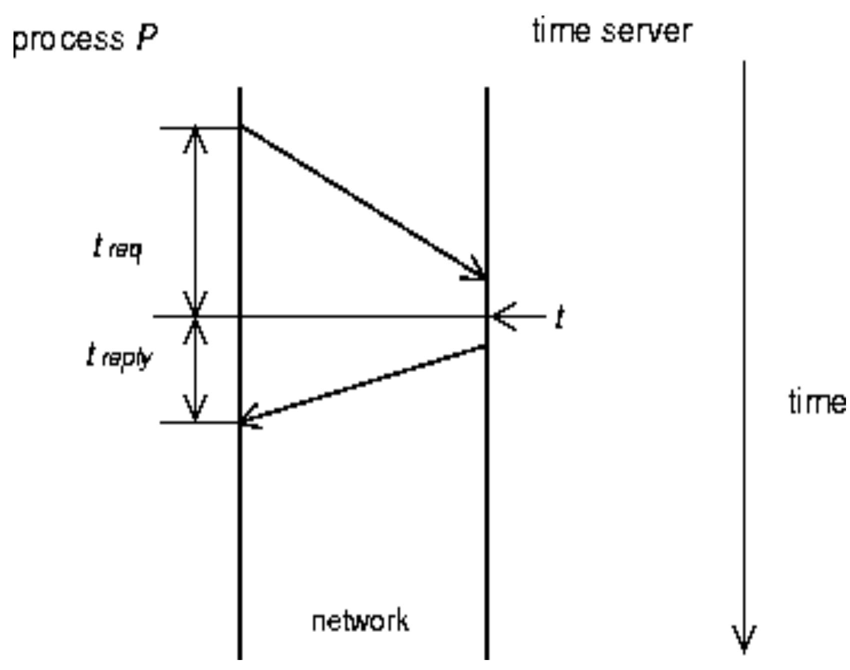


Cristian's Algorithm

- Synchronize machines to a *time server* with a UTC receiver
- Machine P requests time from server every $\delta/2\rho$ seconds
  - Receives time t from server, P sets clock to $t+t_{reply}$ where t_{reply} is the time to send reply to P

Use $(t_{req}+t_{reply})/2$ as an estimate of *the reply*

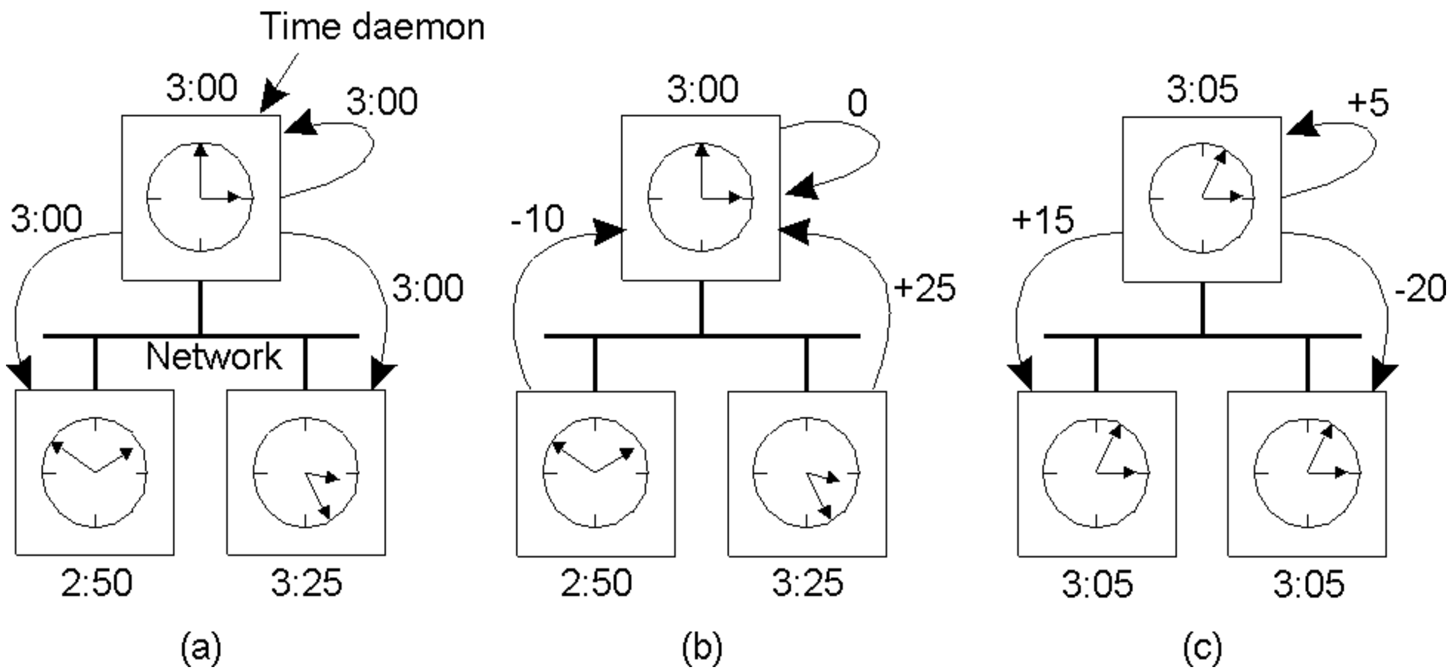
- Improve accuracy by making a series of measurements



Berkeley Algorithm

- Used in systems without UTC receiver
  - Keep clocks synchronized with one another
  - One computer is *master*, other are *slaves*
  - Master periodically polls slaves for their times
    - Average times and return differences to slaves
    - Communication delays compensated as in Cristian’s algo
  - Failure of master => election of a new master

Berkeley Algorithm

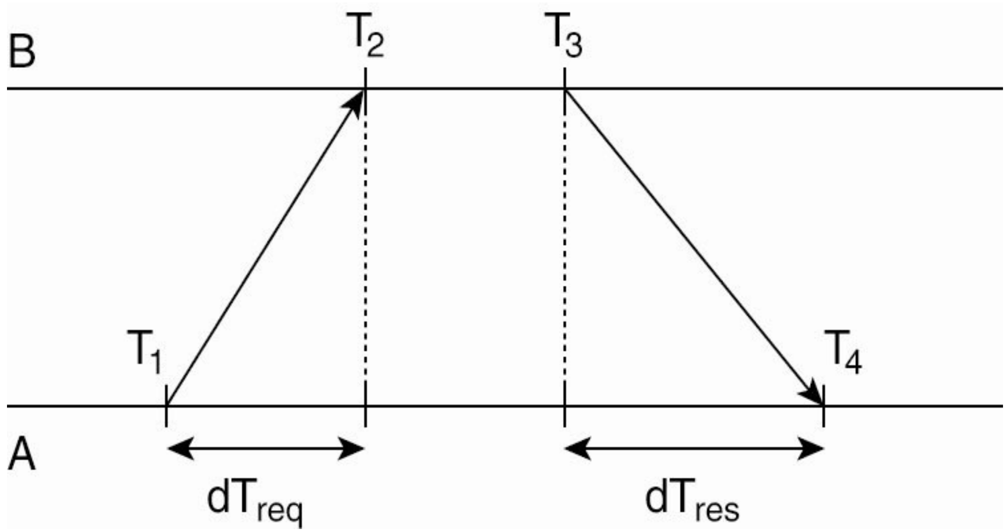


a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock

Distributed Approaches

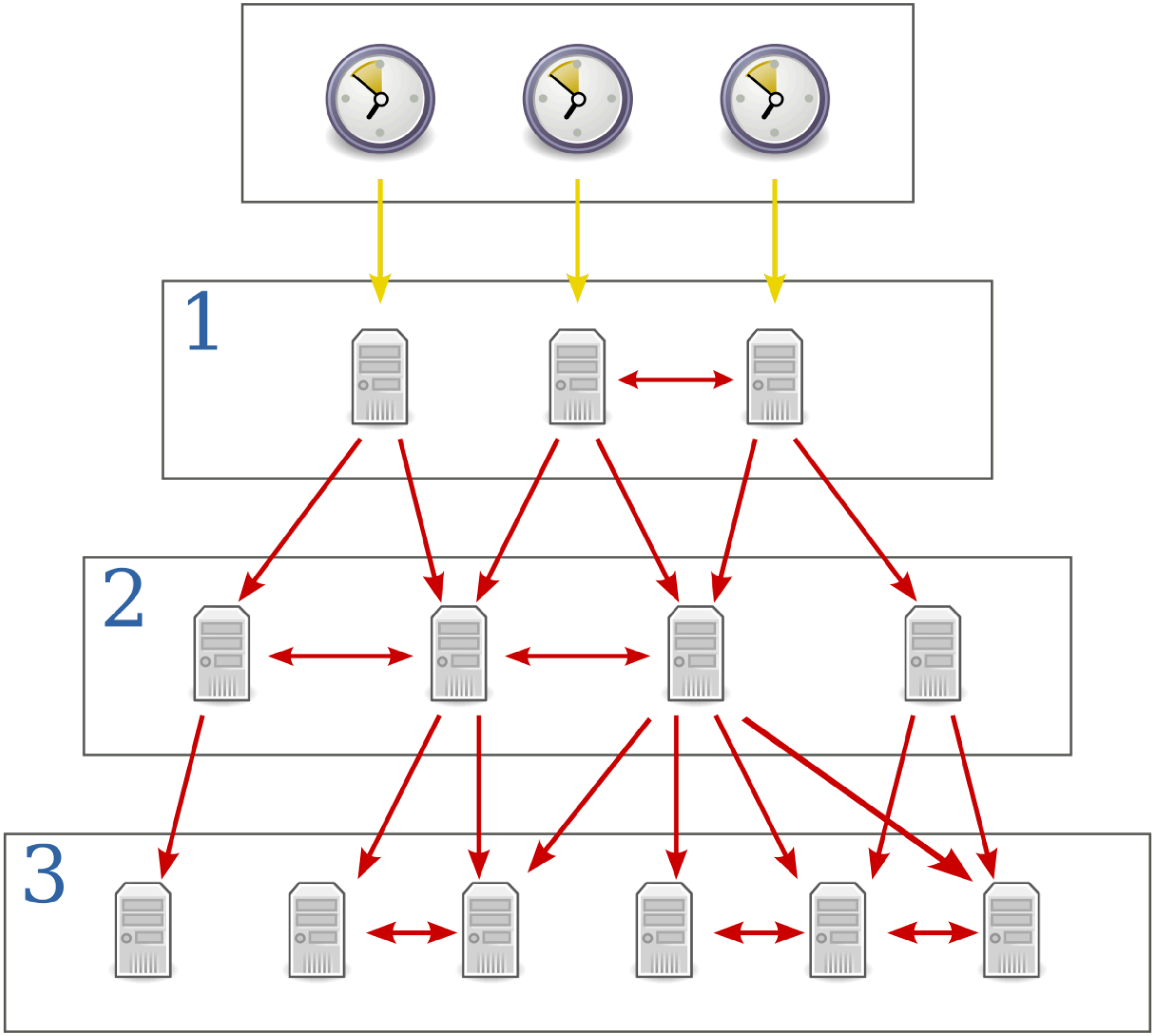
- Both approaches studied thus far are centralized
- Decentralized algorithms: use resync intervals
  - Broadcast time at the start of the interval
  - Collect all other broadcast that arrive in a period *S*
  - Use average value of all reported times
  - Can throw away few highest and lowest values
- Approaches in use today
  - *rdate*: synchronizes a machine with a specified machine
  - Network Time Protocol (NTP) - discussed in next slide
    - Uses advanced techniques for accuracies of 1-50 ms

Network Time Protocol



- Widely used standard - based on Cristian's algorithm.
  - Hierarchical – uses notion of stratum
- **Clock can not go backward**

Network Time Protocol



Logical Clocks

- For many problems, internal consistency of clocks is important
  - Absolute time is less important
  - Use *logical* clocks
- Key idea:
  - Clock synchronization need not be absolute
  - If two machines do not interact, no need to synchronize them
  - More importantly, processes need to agree on the *order* in which events occur rather than the *time* at which they occurred

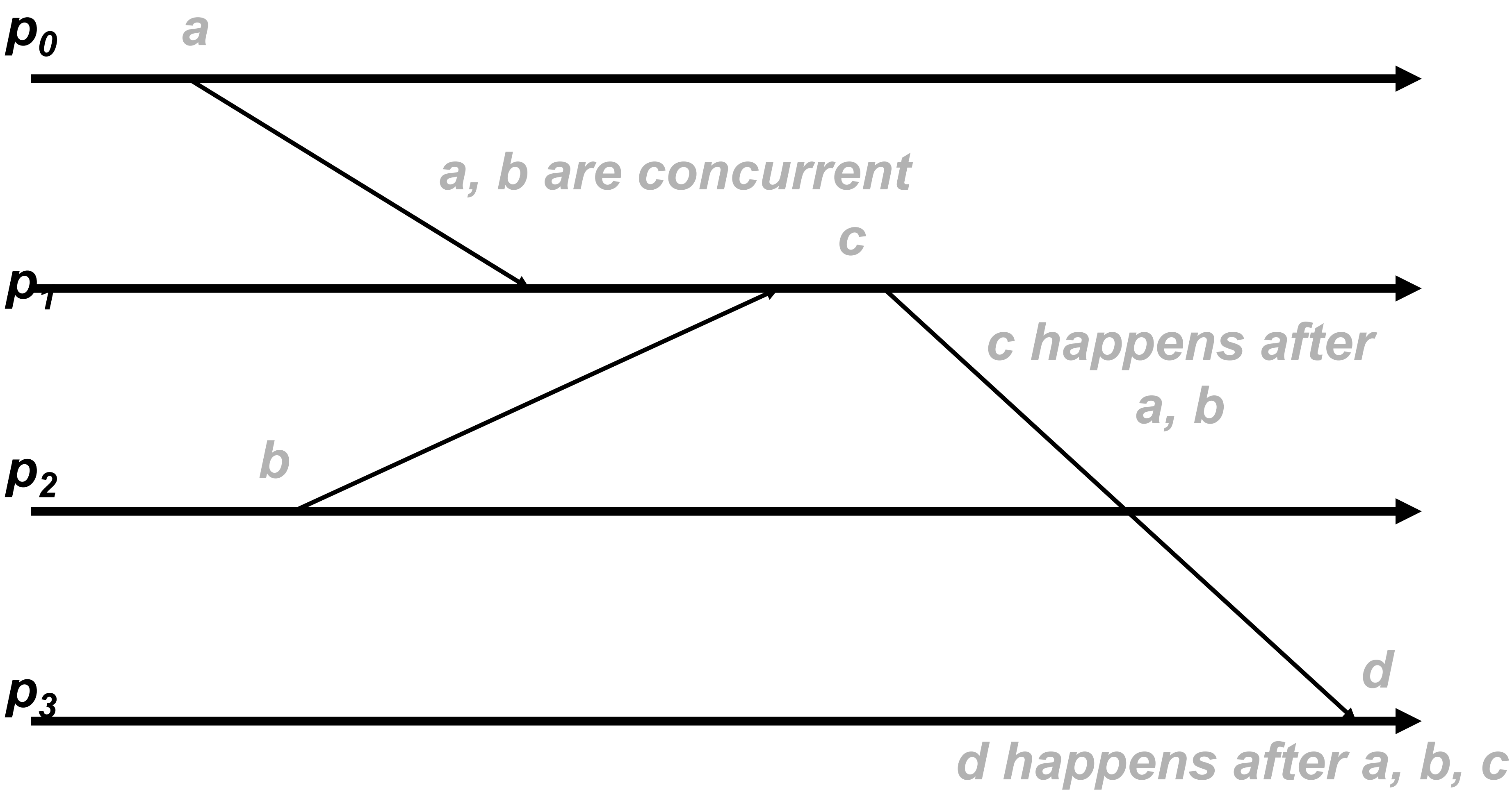
Event Ordering

- *Problem:* define a total ordering of all events that occur in a system
- Events in a single processor machine are totally 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 clocks)

Happened Before Relation

- If *A* and *B* are events in the same process and *A* executed before *B*, then *A -> B*
- If A represents sending of a message and B is the receipt of this message, then A -> B
- Relation is transitive:
  - A -> B and B -> C => A -> C
- Relation is undefined across processes that do not exchange messages
  - Partial ordering on events

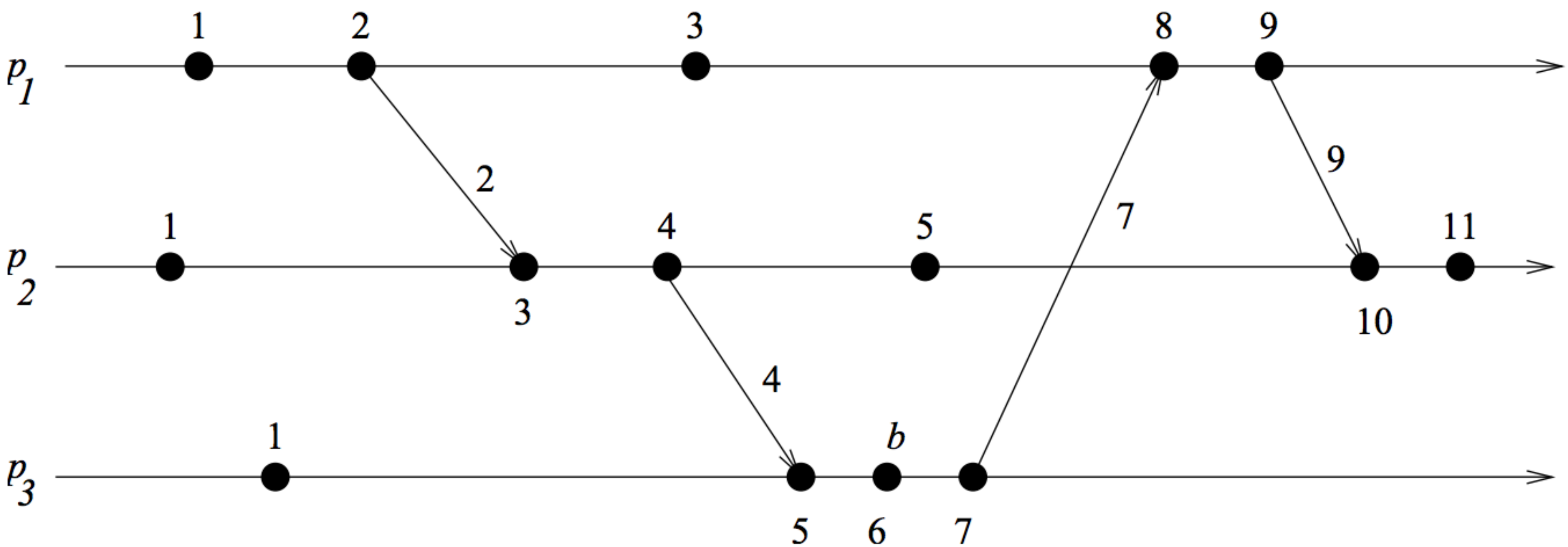
Logical time as a time-space picture



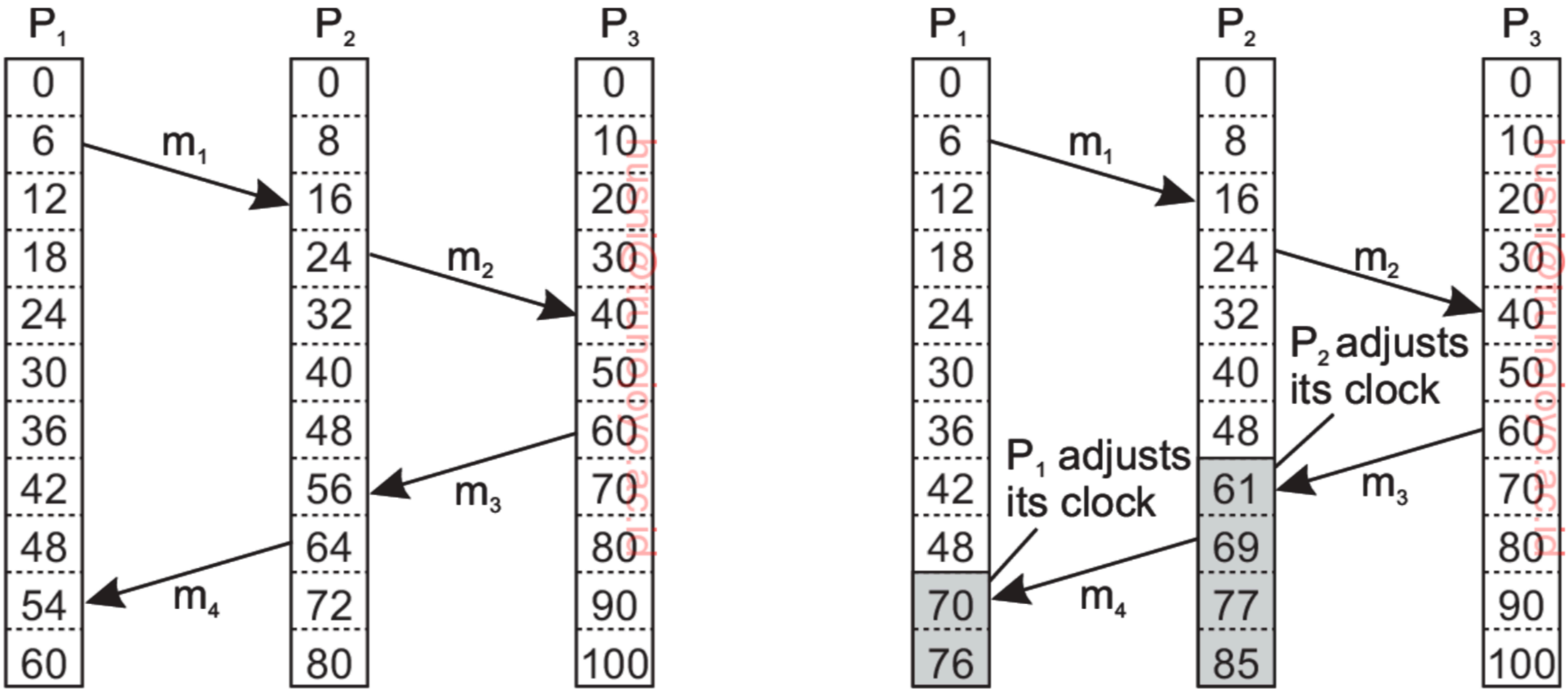
Event Ordering Using *HB*

- Goal: define the notion of time of an event such that
  - If A-> B then C(A) < C(B)
  - If A and B are concurrent, then C(A) <, = or > C(B)
- Solution:
  - Each processor maintains a logical clock LC_i
  - Whenever an event occurs locally at I, $LC_i = LC_i + 1$
  - When *i* sends message to *j*, piggyback Lc_i
  - When *j* receives message from *i*
    - If $LC_j < LC_i$ then $LC_j = LC_i + 1$ else do nothing
  - Claim: this algorithm meets the above goals

Lamport's Logical Clocks

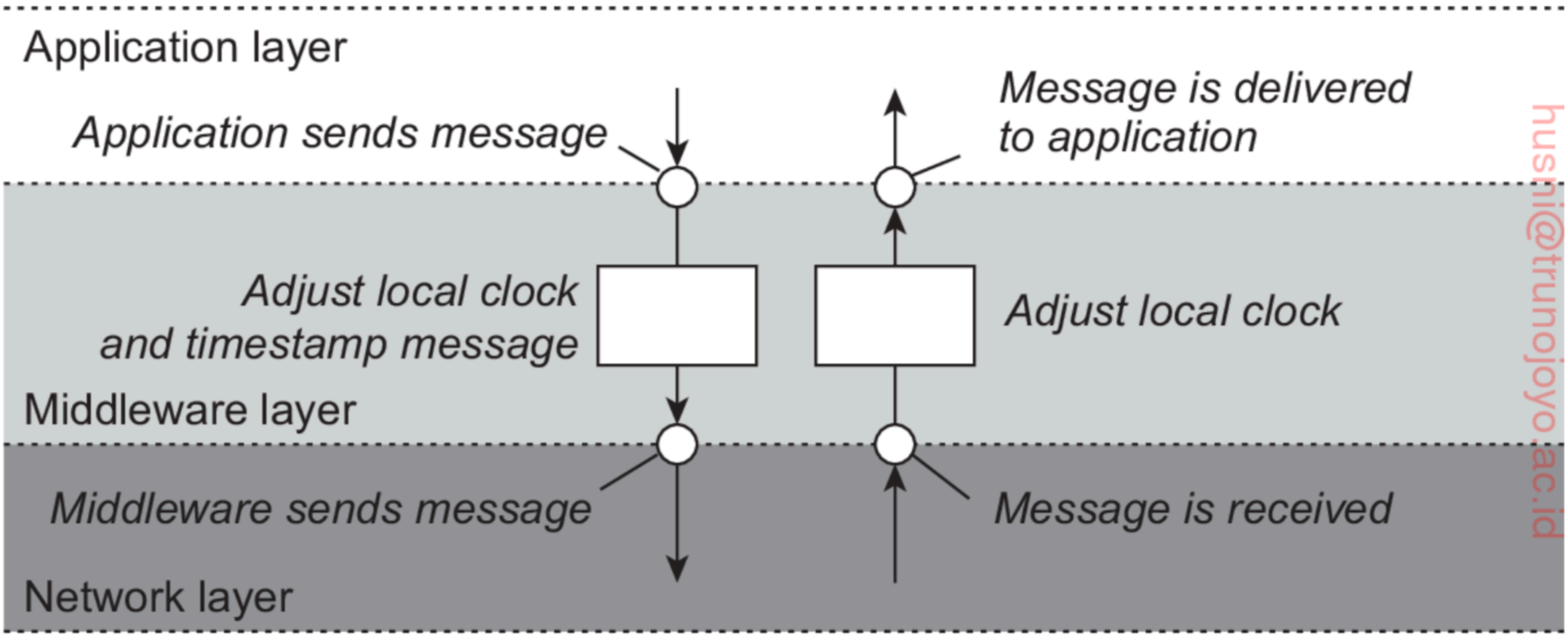


Lamport's Logical Clocks



(a) (b)

Position of L's C is Dist. Systems

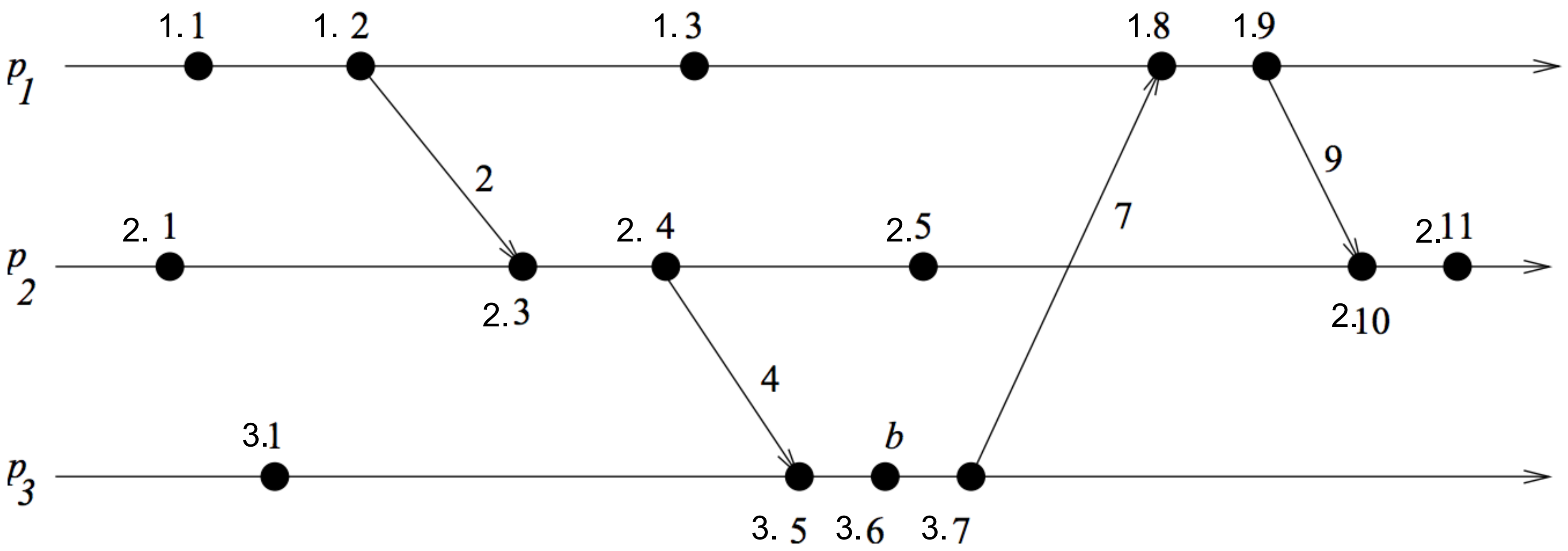


husni@trunojoyo.ac.id

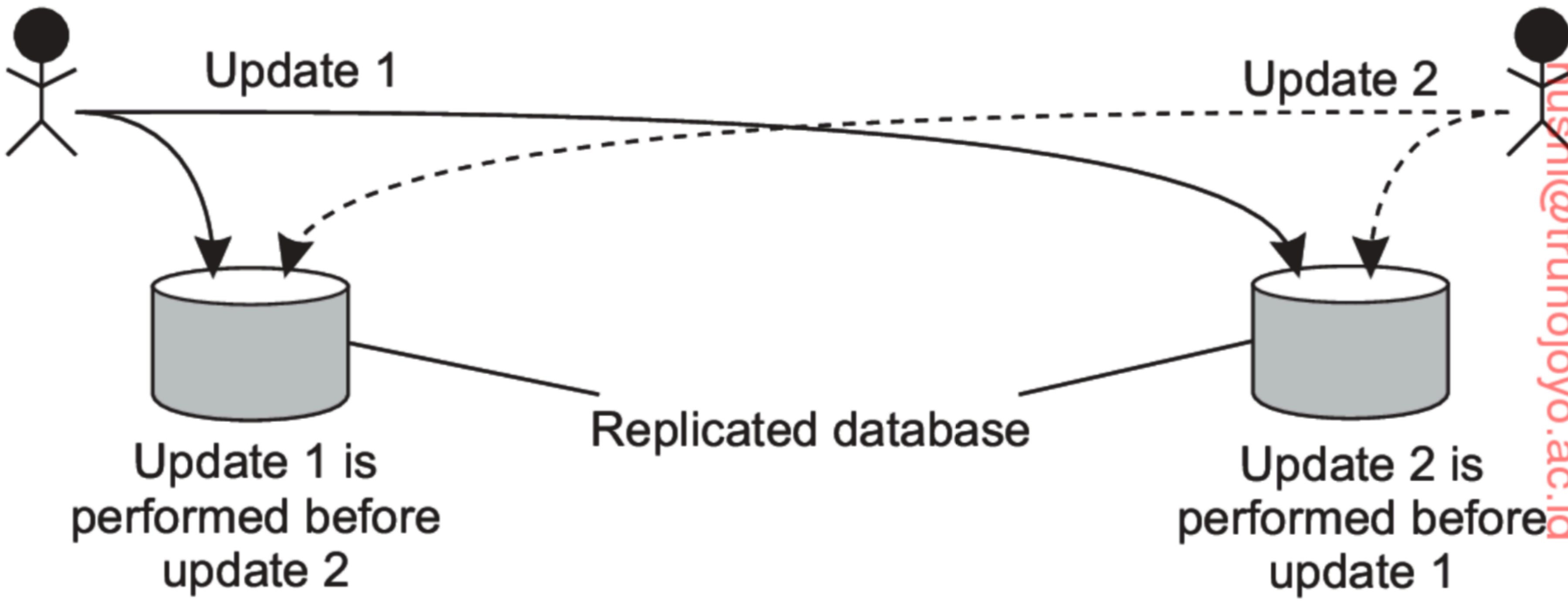
Partial Order to Total Order

- Lamport Logical clocks are only a partial ordering technique
- To convert a partial order into a total order:
  - impose an arbitrary order by appending '.' and a process' id to a logical time value in each process.
  - The process id can be used to break ties.
- The total order is just a tie-breaking rule to assign an order for the events, so it **does not actually tell us the real order**.
- Many system designers use this to convert Lampert's logical clocks partial order to a total order.

Total Order



Total Order Use Case



Totally Ordered Mutli-casting

husni@trunojoyo.ac.id

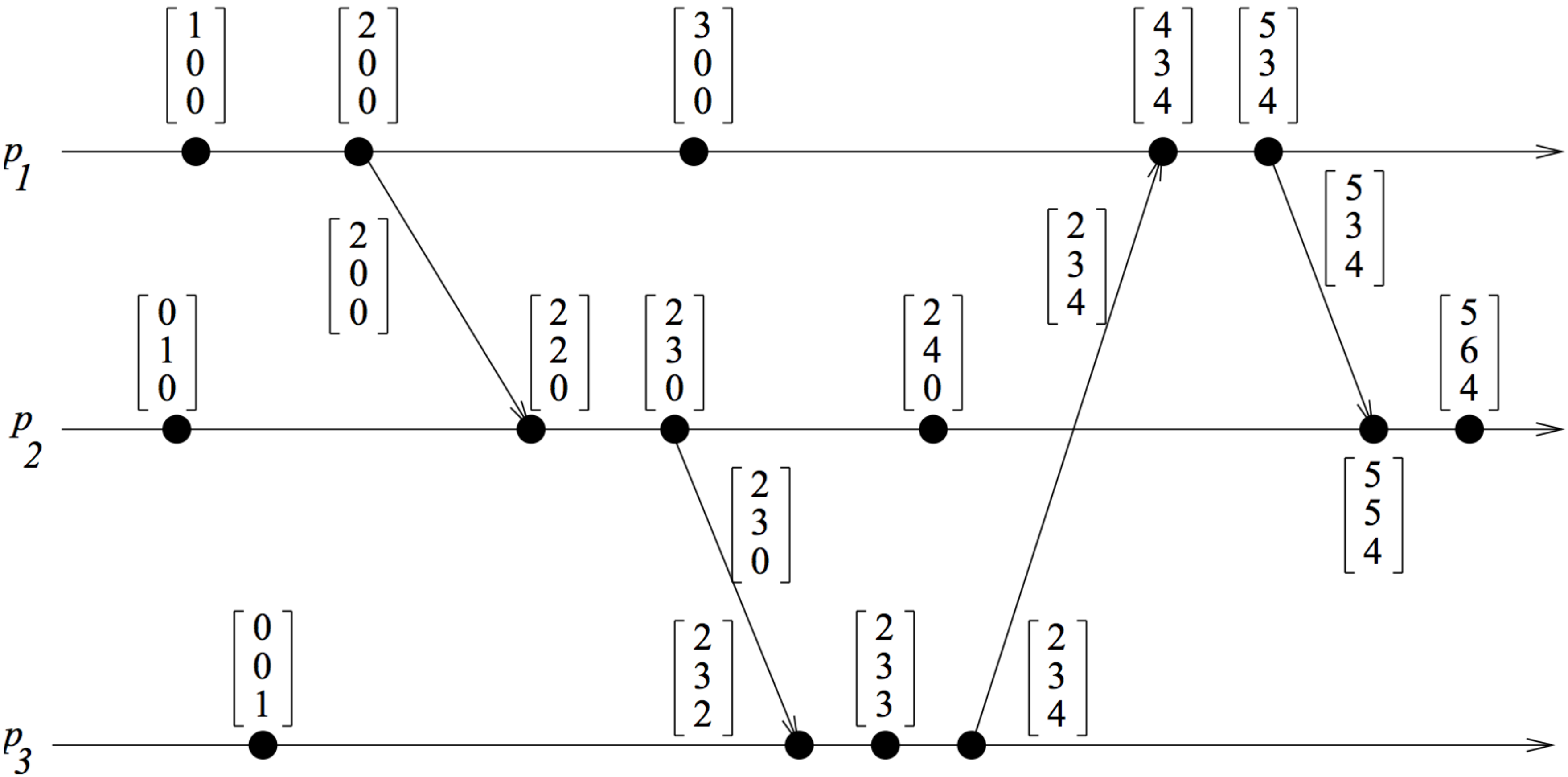
Causality

- Lamport's logical clocks
  - If *A -> B* then *C(A) < C(B)*
  - Reverse is not true!!
    - Nothing can be said about events by comparing time-stamps!
    - If *C(A) < C(B)*, then ??
- Need to maintain *causality*
  - Need a time-stamping mechanism such that:
    - If *T(A) < T(B)* then *A* should have causally preceded *B*

Vector Clocks

- Each process *i* maintains a vector V_i
  - $V_i[i]$: number of events that have occurred at i
  - $V_i[j]$: number of events I knows have occurred at process j
- Update vector clocks as follows
  - Local event: increment $V_i[i]$
  - Send a message :piggyback entire vector V
  - Receipt of a message: $V_j[k] = \max(V_j[k], V_i[k])$
    - Receiver is told about how many events the sender knows occurred at another process *k*

Example



p_1
p_2
p_3
[1 0 0]
[2 0 0]
[3 0 0]
[4 3 4]
[5 3 4]
[2 0 0]
[2 3 4]
[5 3 4]
[0 1 0]
[2 2 0]
[2 3 0]
[2 4 0]
[5 6 4]
[2 3 0]
[5 5 4]
[0 0 1]
[2 3 2]
[2 3 3]
[2 3 4]

Vector Clocks

- Define VT(e)<VT(e') if,
  - for all i, VT(e)[i]≤VT(e')[i], and
  - for some j, VT(e)[j]<VT(e')[j]
- Example: if VT(e)=[2,1,1,0] and VT(e')=[2,3,1,0] then VT(e)<VT(e')
- Notice that not all VT's are "comparable" under this rule: consider [4,0,0,0] and [0,0,0,4]
- If VT's are not comparable, the corresponding events are concurrent or casually-independent.

Leader Election

Election Algorithms

- Many distributed algorithms need one process to act as coordinator
  - Doesn’t matter which process does the job, just need to pick one
- Election algorithms: technique to pick a unique coordinator (aka *leader election*)
- Examples: take over the role of a failed process, pick a master in Berkeley clock synchronization algorithm
- Types of election algorithms: Bully and Ring algorithms

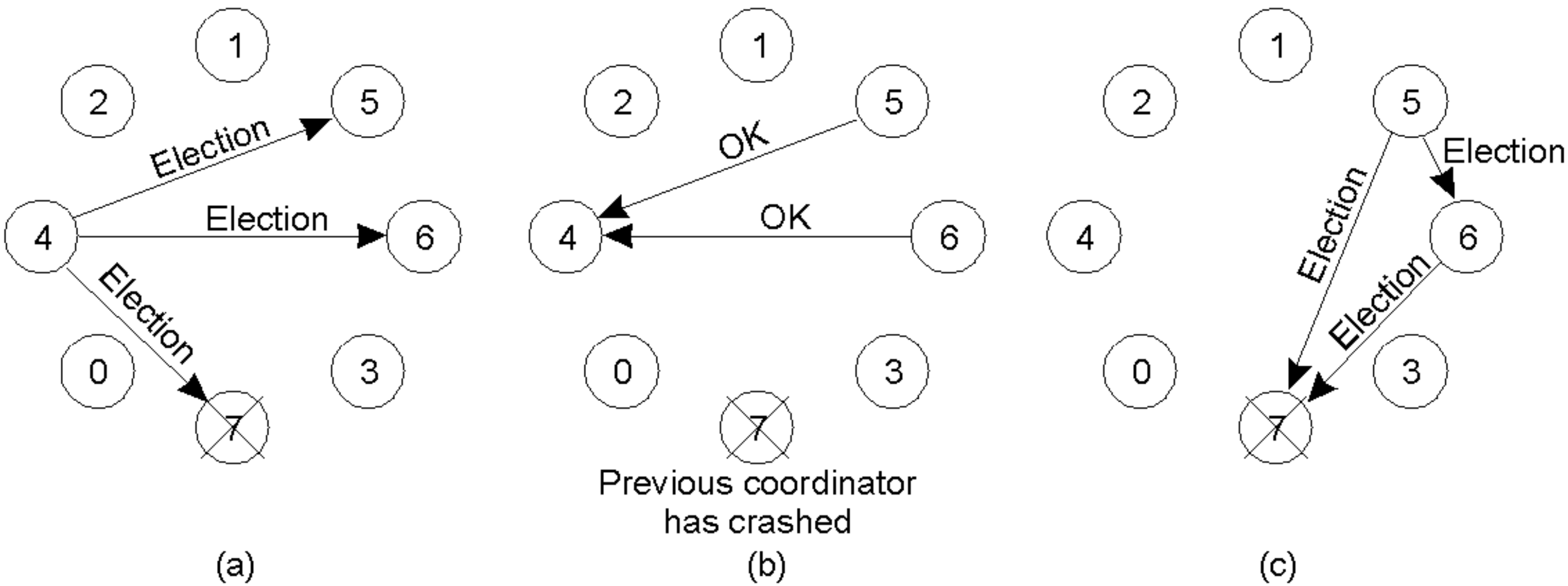
Bully Algorithm

- Each process has a unique numerical ID
- Processes know the Ids and address of every other process
- Communication is assumed reliable
- *Key Idea*: select process with highest ID
- Process initiates election if it just recovered from failure or if coordinator failed
- 3 message types: *election, OK, I won*
- Several processes can initiate an election simultaneously
  - Need consistent result
- $O(n^2)$ messages required with n processes

Bully Algorithm Details

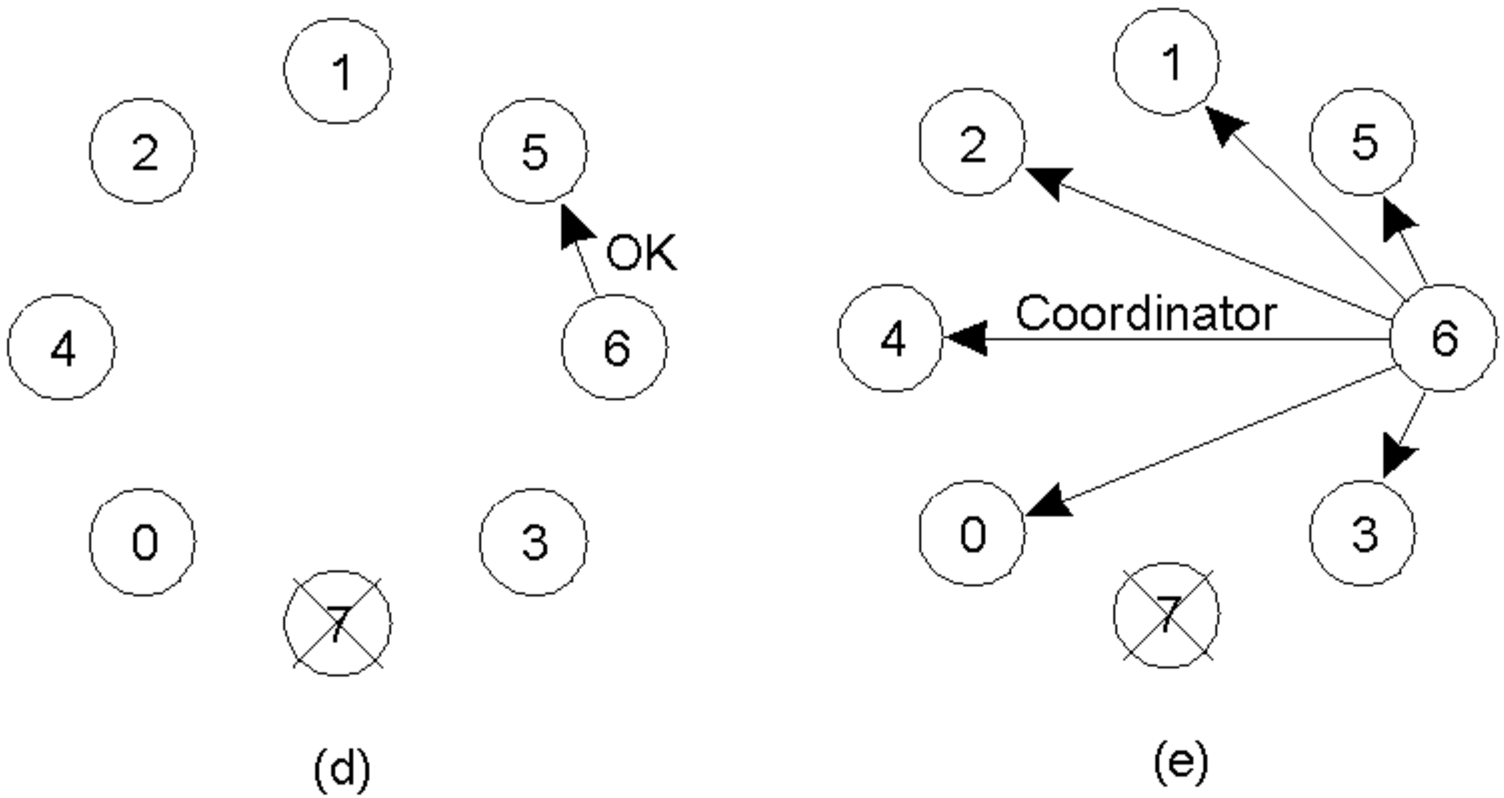
- Any process *P* can initiate an election
- *P* sends *Election* messages to all processes with higher Ids and awaits *OK* messages
- If no *OK* messages, *P* becomes coordinator and sends *I won* messages to all process with lower Ids
- If it receives an *OK*, it drops out and waits for an *I won*
- If a process receives an *Election* msg, it returns an *OK* and starts an election
- If a process receives a *I won*, it treats sender an coordinator

Bully Algorithm Example



- The bully election algorithm
- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

Bully Algorithm Example



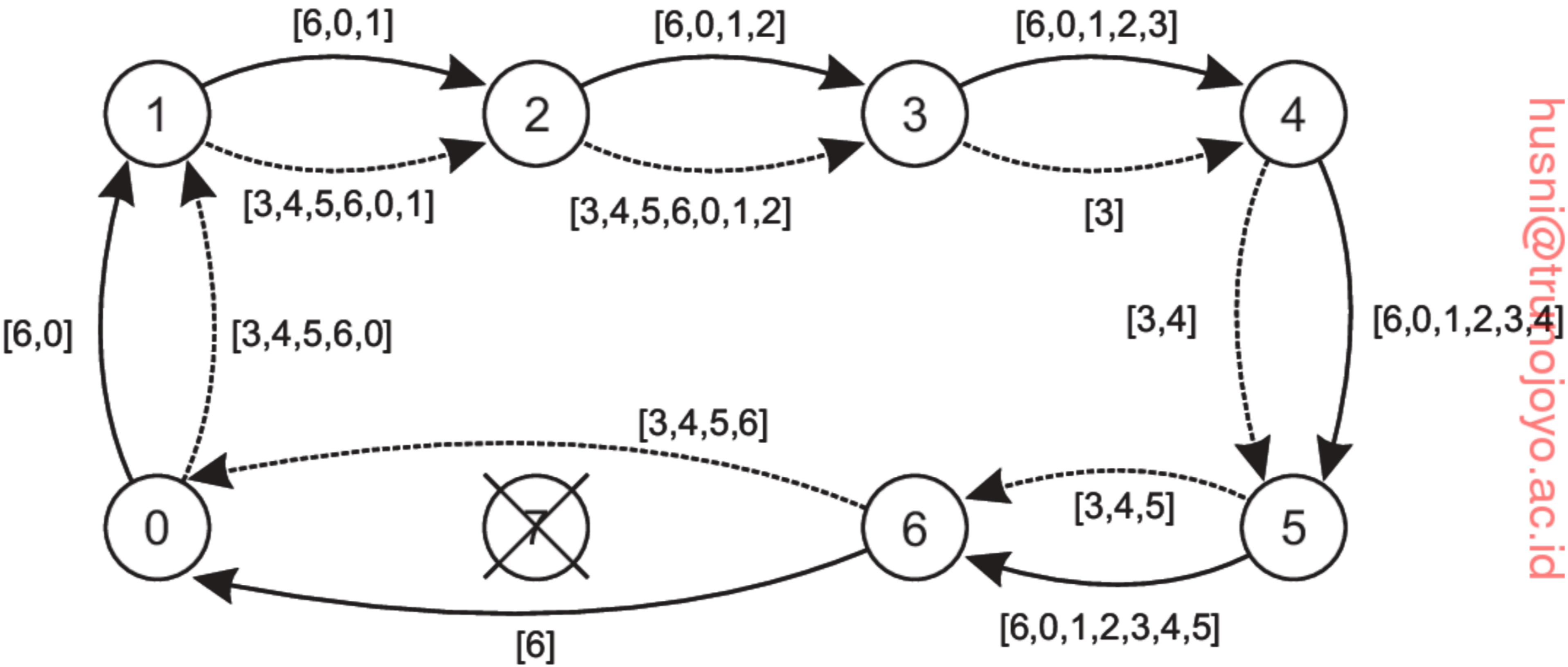
d) Process 6 tells 5 to stop

e) Process 6 wins and tells everyone

Ring-based Election

- Processes have unique Ids and arranged in a logical ring
- Each process knows its neighbors
  - Select process with highest ID
- Begin election if just recovered or coordinator has failed
- Send *Election* to closest downstream node that is alive
  - Sequentially poll each successor until a live node is found
- Each process tags its ID on the message
- Initiator picks node with highest ID and sends a coordinator message
- Multiple elections can be in progress
  - Wastes network bandwidth but does no harm

A Ring Algorithm



husni@trunojoyo.ac.id

Comparison

- 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: $O(n^2)$ msgs
  - Best case: immediate election: n-2 messages
- Ring
  - 2(n-1) messages always

Mutual Exclusion

Distributed Synchronization

- Distributed system with multiple processes may need to access share data or resources
  - For a single process with multiple threads
    - Semaphores, locks, monitors
  - How do you do this for multiple processes in a distributed system?
    - Processes may be running on different machines

- **Solution:** mutual exclusion (critical sections)

Mutual Exclusion

- Can be classified into two main categories:
  - **Permission-based approaches**
    - The process that wants to access resources requests a permission from other process(es).
  - **Token-based approaches**
    - Passing a special message between processes. This message is called Token.
    - Whoever has the token can access the shared resource
- Can also be classified as:
  - Centralized or Distributed.

Centralized Mutual Exclusion

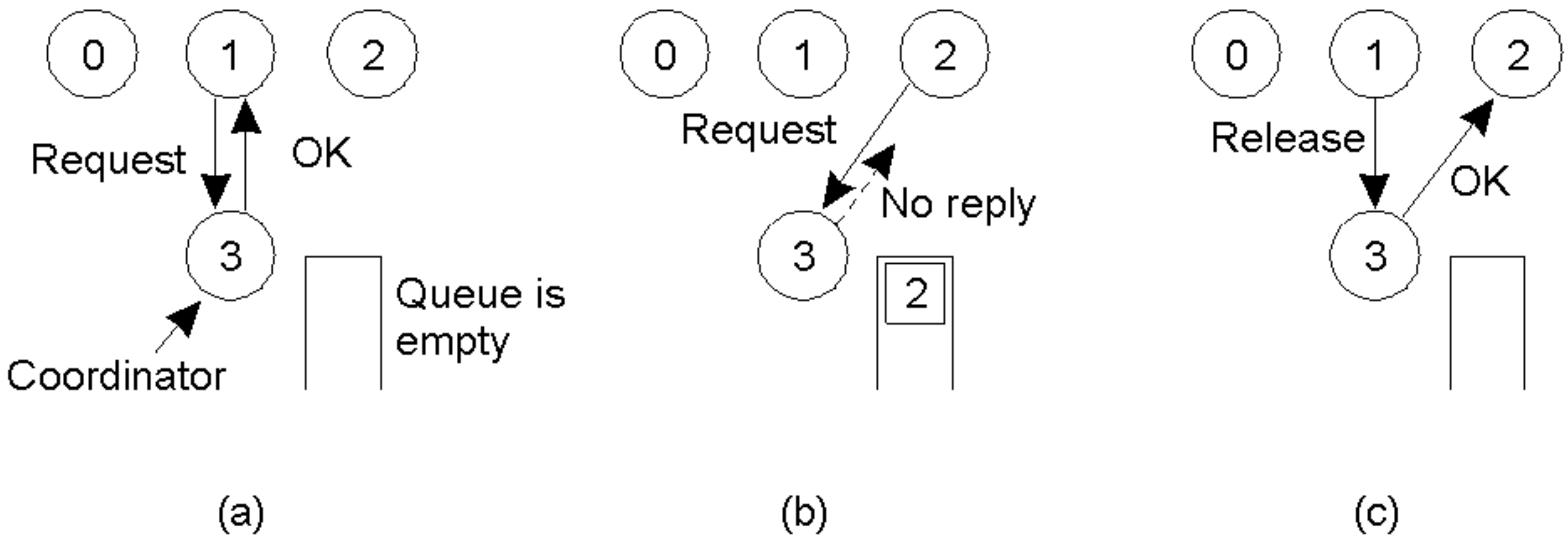
- Assume processes are numbered
- One process is elected coordinator (highest ID).
- Every process checks with coordinator before entering the critical section

- Request Lock:
  - send request
  - await reply
- Release Lock:
  - send release message

Centralized Mutual Exclusion

- Coordinator:
  - Receive *request*: if available and queue empty, send grant; if not, queue request
  - Receive *release*: remove next request from queue and send grant

Mutual Exclusion: A Centralized Algorithm



a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted
b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

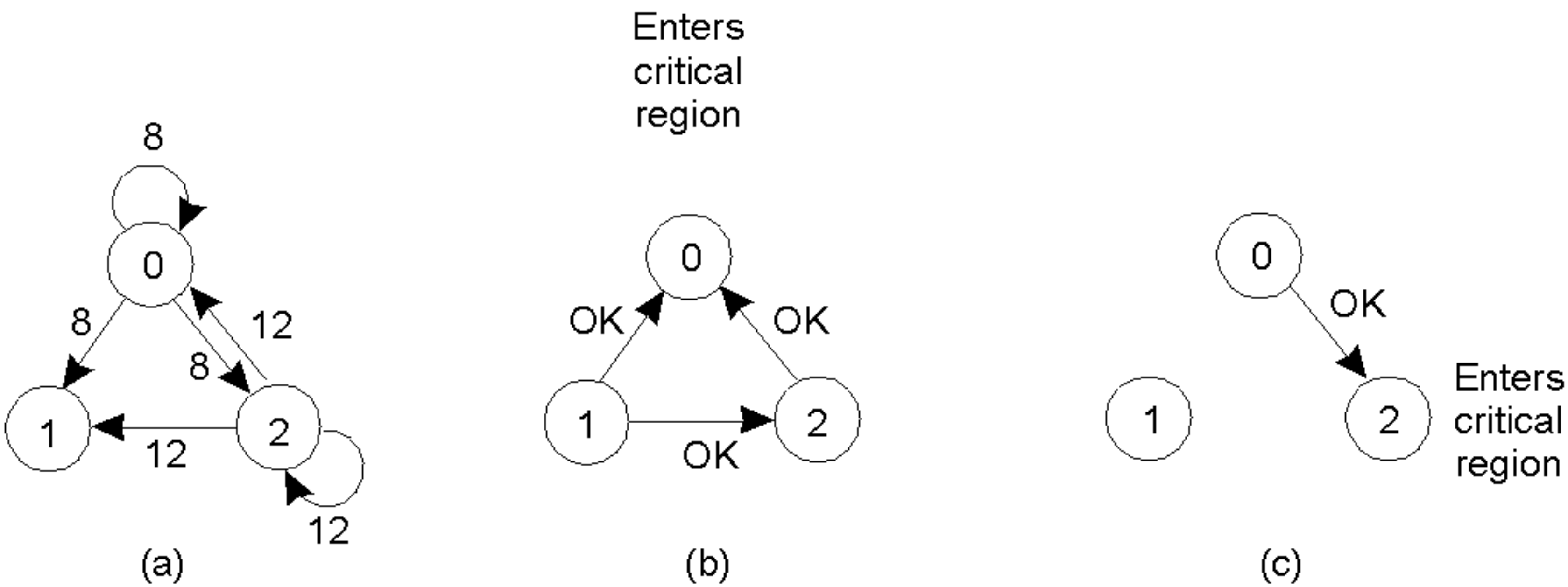
Properties

- Simulates centralized lock using blocking calls
- **Fair**: requests are granted the lock in the order they were received
- **Simple**: three messages per use of a critical section (request, grant, release)
- **Shortcomings**:
  - Single point of failure
  - How do you detect a dead coordinator?
    - A process can not distinguish between "lock in use" from a dead coordinator
      - No response from coordinator in either case
  - Performance bottleneck in large distributed systems

Distributed Algorithm

- [Ricart and Agrawala]: needs 2(n-1) messages
- Based on event ordering and time stamps
- Process *k* enters critical section as follows
  - Generate new time stamp $TS_k = TS_k+1$
  - Send *request*(k, TS_k) all other *n-1* processes
  - Wait until *reply(j)* received from all other processes
  - Enter critical section
- Upon receiving a *request* message, process *j*
  - Sends *reply* if no contention
  - If already in critical section, does not reply, queue request
  - If wants to enter, compare TS_j with TS_k and send reply if $TS_k<TS_j$, else queue

A Distributed Algorithm

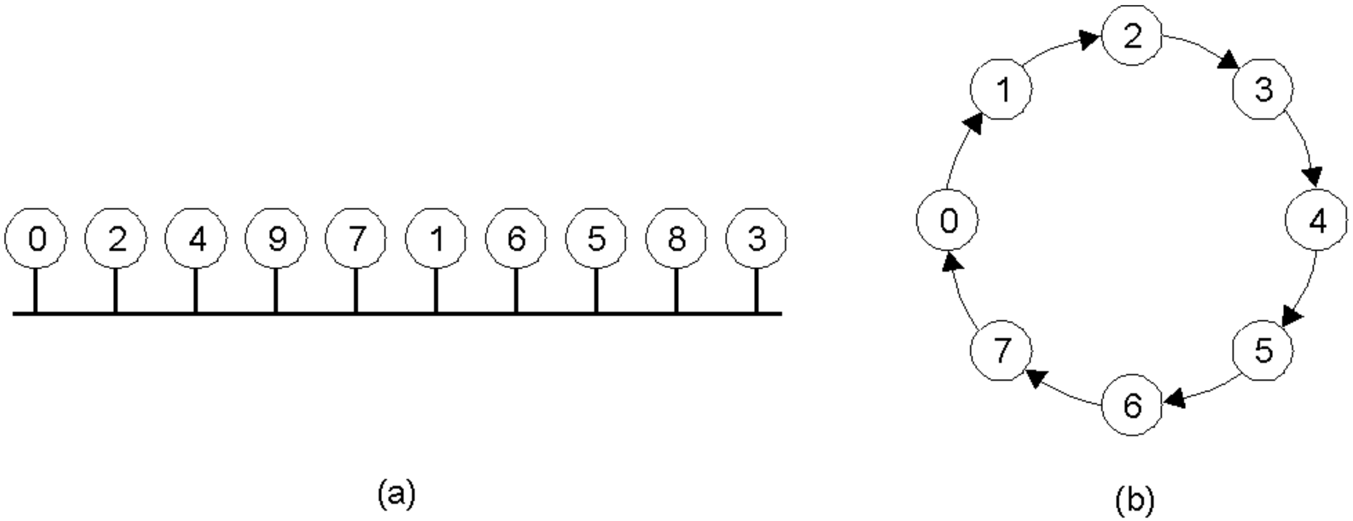


a) Two processes want to enter the same critical region at the same moment.
b) Process 0 has the lowest timestamp, so it wins.
c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

Properties

- Fully decentralized
- *N* points of failure!
  - Can be fixed by using time-outs
- All processes are involved in all decisions
  - Any overloaded process can become a bottleneck
    - Can be fixed by asking for majority permission.

A Token Ring Algorithm



a) An unordered group of processes on a network.

b) A logical ring constructed in software.

- Use a token to arbitrate access to critical section
- Must wait for token before entering CS
- Pass token to neighbor once done or if not interested
- Detecting token loss in not-trivial

Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

- A comparison of three mutual exclusion algorithms.