Synchronization

Distributed Systems [6]

殷亚凤

Email: yafeng@nju.edu.cn

Homepage: http://cs.nju.edu.cn/yafeng/

Room 301, Building of Computer Science and Technology

Review

- Protocols
- Remote Procedure Call
- Message-oriented Communication
- Message-oriented middleware
- Stream-oriented communication
- Multicast communication

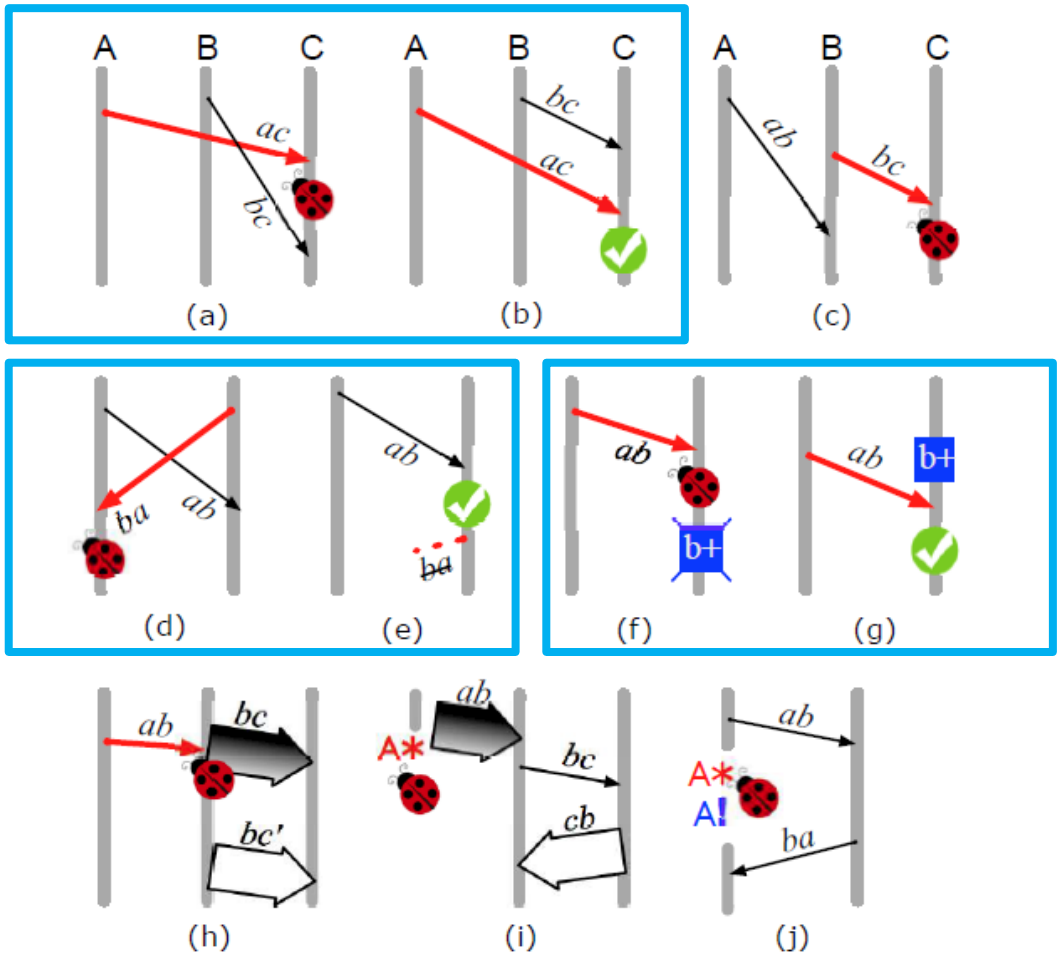
This Lesson

- Clock synchronization
- Physical clocks
- Logical clocks
- Mutual Exclusion
- Election algorithms

Synchronization Problem

- How processes cooperate and synchronize with one another in a distributed system
  - In single CPU systems, critical regions, mutual exclusion, and other synchronization problems are solved using methods such as semaphores.
  - These methods will not work in distributed systems because they implicitly rely on the existence of shared memory.
    - If two events occur in a distributed system, it is difficult to determine **which event occurred first**.
- How to decide on relative ordering of events
  - Does one event precede another event?
  - Difficult to determine if events occur on different machines.

Triggering Patterns



Tanakorn Leesatapornwongsa et al. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems, Asplos 2016.

Synchronization

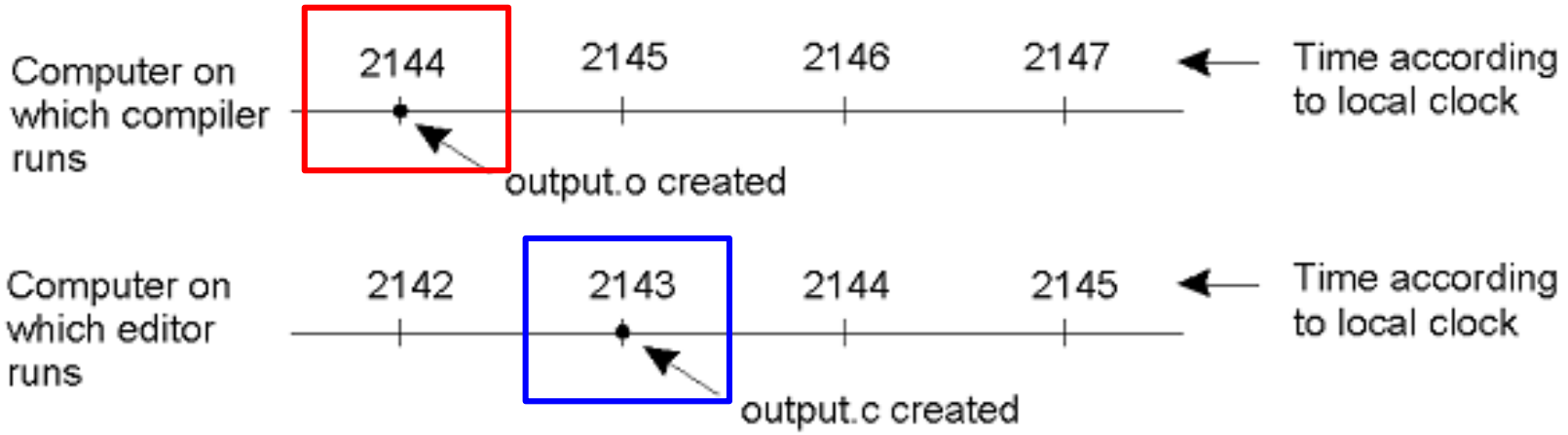
- Clocks
  - How to synchronize events based on actual time?
    - Clock synchronization
  - How to determine relative ordering?
    - Logical clocks
- How do we synchronize for sharing
  - Mutual exclusion
- Election
  - How do we determine the "coordinator" of a distributed system?

Clock synchronization

- In a centralized system:
  - Time is unambiguous: A process gets the time by issuing a system call to the kernel. If process *A* gets the time and latter process *B* gets the time. The value *B* gets is higher than (or possibly equal to) the value *A* got
  - Example: UNIX-make examines the time at which all the source and object files were last modified:
    - If time (input.c) > time(input.o) then recompile input.c
    - If time (input.c) < time(input.o) then no compilation is needed

Clock synchronization is not Trivial

- In a distributed system:
  - Achieving agreement on time is not trivial!



Physical vs Logical Clocks

- For algorithms where clocks must not only be the same, but also must not deviate from real-time, we speak of physical clocks.
- For algorithms where only internal consistency of clocks matters (not whether clocks are close to real time), we speak of logical clocks.
- Clock synchronization need not be absolute! (due to Lamport, 1978):
  - If two processes do not interact, their clocks need not be synchronized.
  - What matters is not that all processes agree on exactly what time is it, but rather, that they agree on the order in which events occur.

Clocks

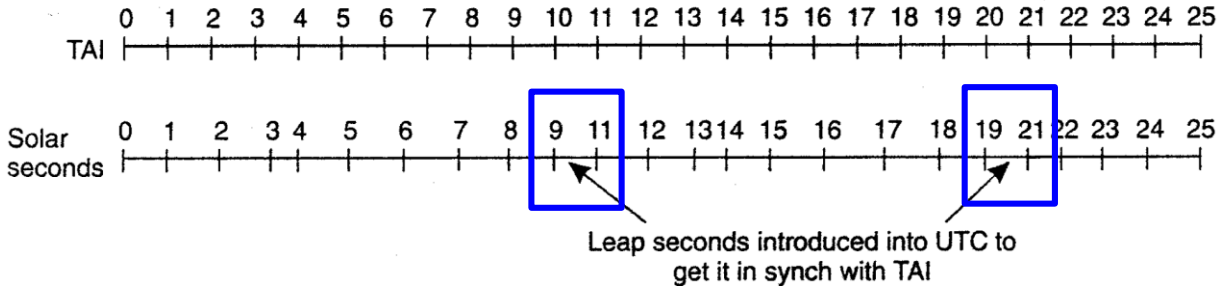
- A computer has a timer, not really a clock.
- Timer is a precisely machined quartz crystal oscillating at a frequency that depends on how the crystal was cut and the amount of tension.
- Two registers are associated with the crystal: counter & holding register
  - Each **oscillation** of the crystal decrements the counter by one. When counter gets to zero, an interrupt is generated and the counter reloaded from holding register.
  - In this way, it is possible to program a timer to generate an interrupt 60 **times a second**, or any other desired frequency. Each interrupt is called a clock tick
  - At each clock tick, **the interrupt procedure adds 1** to the time stored in memory (this keeps the software clock up to date)
- On each of *n* computers, the crystals will run at slightly different frequencies, causing the software clocks gradually to get out of sync (clock skew).

Temps Atomique International

- Labs around the world have atomic clocks and each of them periodically tells the BIH (Bureau International de l'Heure) in Paris how many times its clock ticked.
- The BIH averages these to produce the TAI (Temps Atomique International).
- Originally the atomic time is computed to make the atomic second equal to the mean solar second.

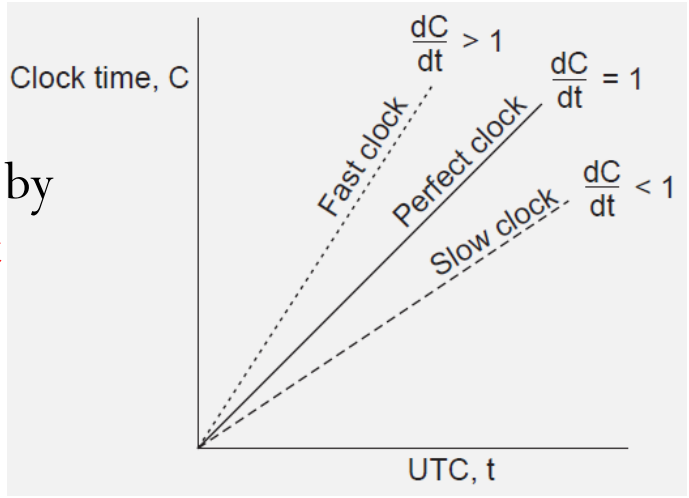
Physical clocks

- Universal Coordinated Time (UTC):
  - Based on the number of transitions per second of the cesium 133 atom (pretty accurate).
  - At present, the real time is taken as the average of some 50 cesium-clocks around the world.
  - Introduces a leap second from time to time to compensate that days are getting longer.
- UTC is broadcast through short wave radio and satellite. Satellites can give an accuracy of about ± 0.5 ms.



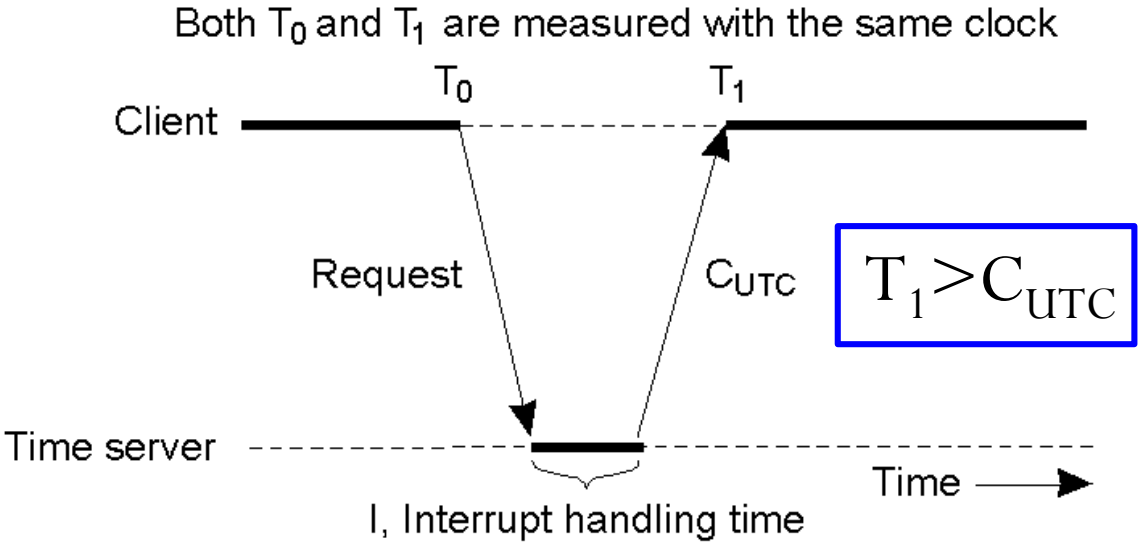
Physical clocks Synchronization

- Suppose we have a distributed system, we have to distribute its time to each machine.
- Basic principle
  - Every machine has a timer that generates an interrupt H times per second.
  - There is a clock in machine p that ticks on each timer interrupt. Denote the value of that clock by $C_p(t)$, where t is UTC time.
  - Ideally, we have that for each machine p, $C_p(t) = t$, or, in other words, $dC/dt = 1$.
- In practice: $1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$
- Never let two clocks in any system differ by more than δ time units $\Rightarrow$ synchronize at least every $\delta/(2\rho)$ seconds.

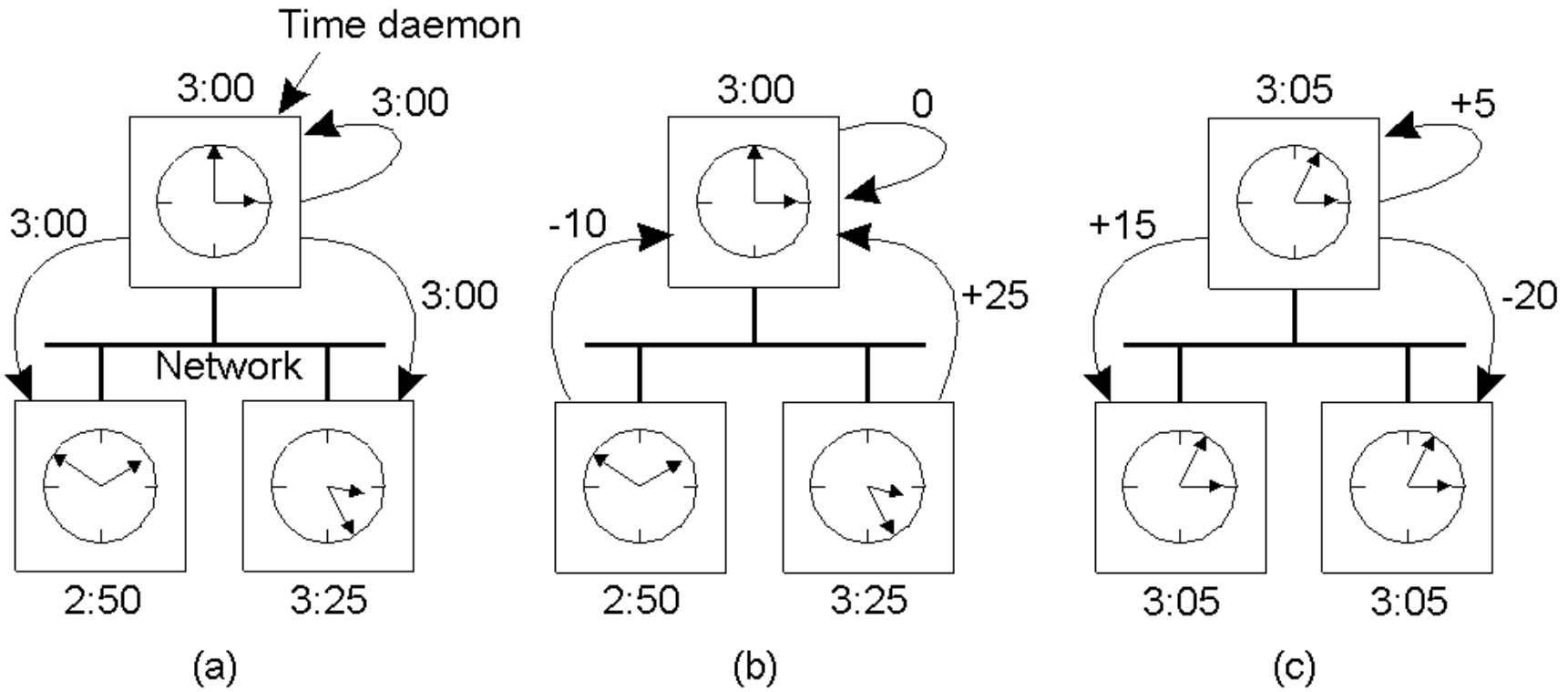


Cristian's Algorithm

- One time server (WWV receiver); all other machines stay synchronized with the time server.
- Cannot set T_1 to C_{UTC} because time must never run backwards. Changes are introduced gradually by adding more or less seconds for each interrupt.
- The propagation time is included in the change.
  - Estimated as: $(T_1 - T_0 - I)/2$ (can be improved by continuous probing & averaging)



The 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

Averaging Algorithm

- Divide the time into fixed length resynchronization intervals
- The i-th interval starts at T_0+iR and runs until $T_0+(i+1)R$, where T_0 is an agreed upon moment in the past, and R is a system parameter
- At the beginning of each interval, each machine broadcasts the current time according to its own clock (broadcasts are not likely to happen simultaneously because the clocks on different machine do not run at exactly the same speed)
- After a machine broadcasts its time, it starts a timer to collect all other broadcasts that arrive during some interval S
- When all broadcasts arrive, an algorithm is run to compute the new time from them
  - Simplest algorithm: Average the values from all other machines
- Variations: discard m highest and m lowest values and average the rest; Add to each message an estimate of the propagation time from source

The Internet Network Time Protocols

- Layered client-server architecture, based on UDP message passing
- Synchronization at clients with higher strata number less accurate due to increased latency to strata 1 time server
- Failure robustness: if a strata 1 server fails, it may become a strata 2 server that is being synchronized through another strata 1 server
- Modes:
  - **Procedure call**
    - Similar to Cristian's algorithm
  - **Multicast**
    - One computer periodically transmits time
  - **Symmetric**
    - To be used where high accuracy is needed

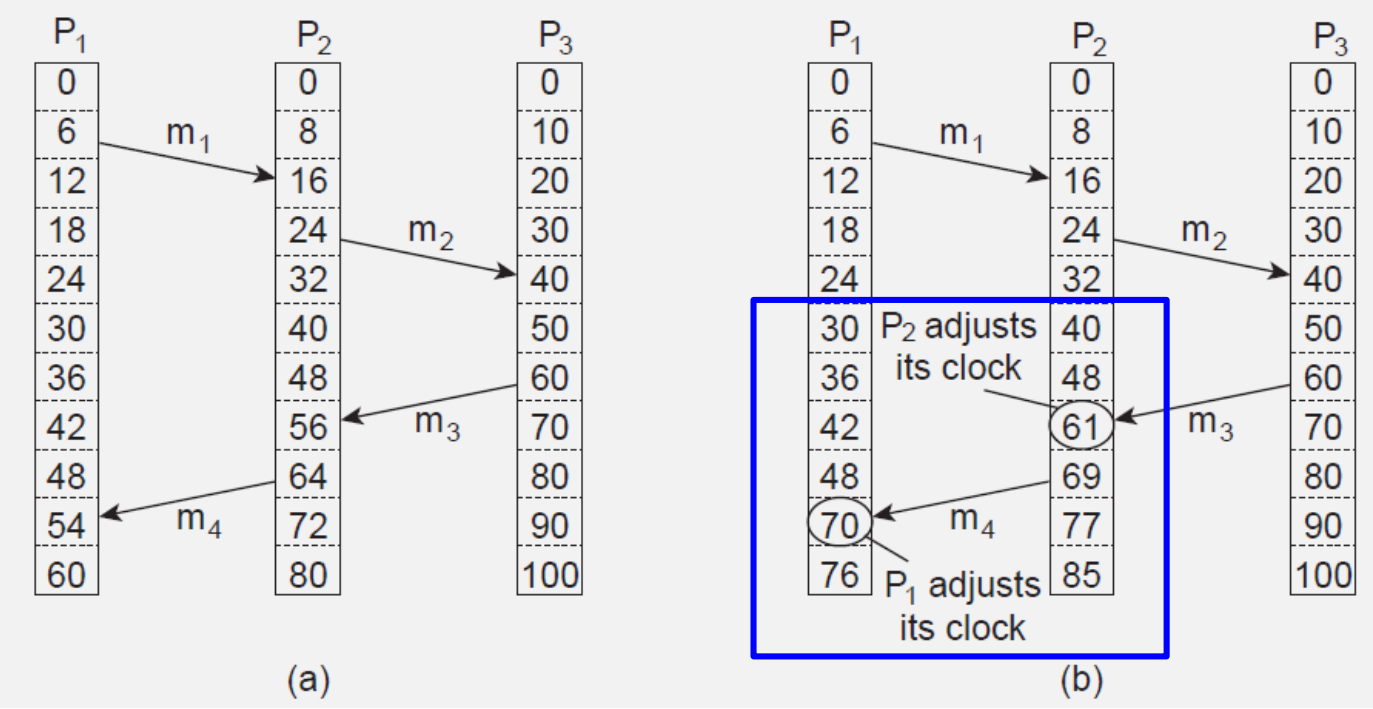
Logical clocks: The Happened-before relationship

- We first need to introduce a notion of ordering before we can order anything.
- The happened-before relation
  - 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 $a \rightarrow b$.
  - If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.
- This introduces a partial ordering of events in a system with concurrently operating processes.

Logical clocks

- 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 properties:
  - P1: If a and b are two events in the same process, and a→b, then we demand that $C(a) < C(b)$.
  - P2: If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.
- How to attach a timestamp to an event when there's no global clock ⇒ maintain a consistent set of logical clocks, one per process.

Lamport's Algorithm



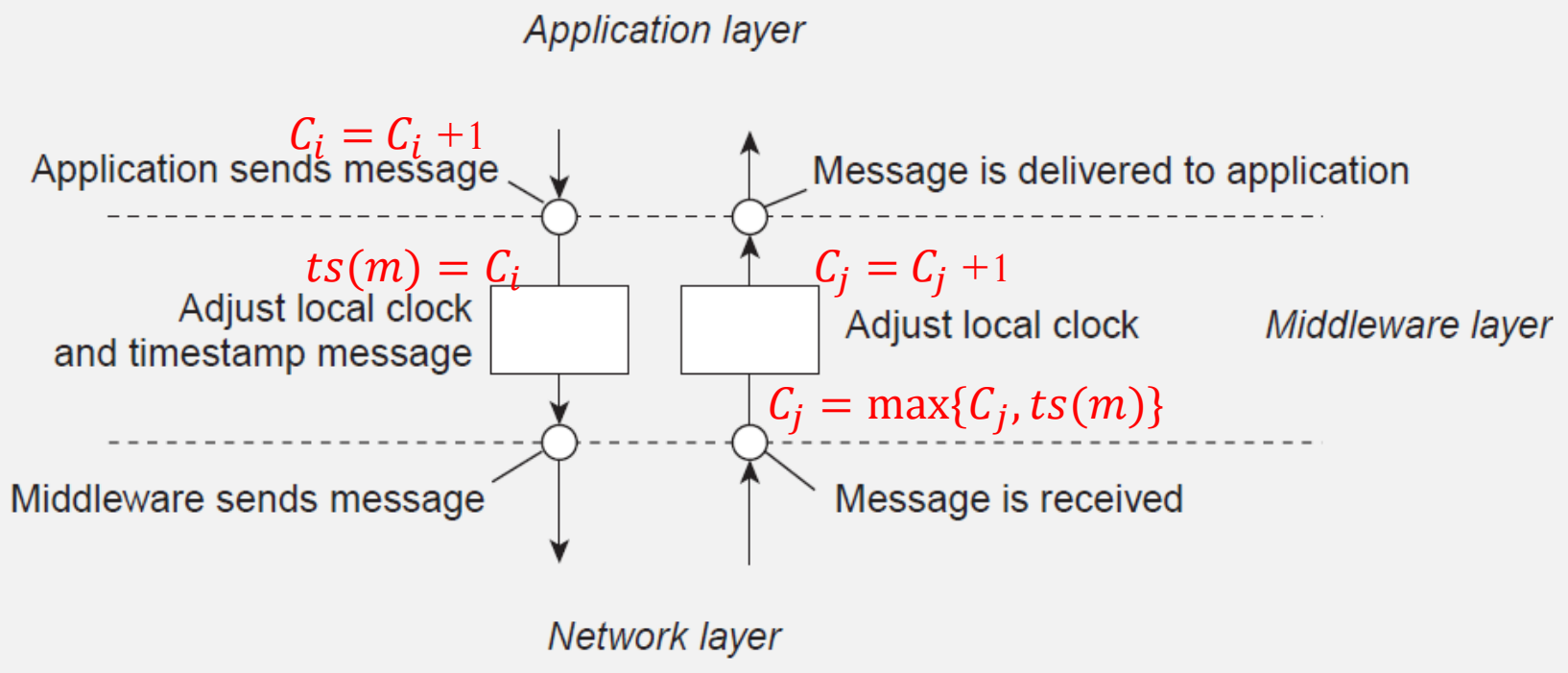
- Three processes, each with its own clock. The clocks run at different rates.
- Lamport's algorithm corrects the clocks.
- Lamport solution:
  - Between every two events, the clock must tick at least once

Lamport's Algorithm

- Each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:
  - 1: For any two successive events that take place within P_i , C_i is incremented by 1.
  - 2: Each time a message m is sent by process P_i , the message receives a timestamp $ts(m) = C_i$.
  - 3: Whenever a message m is received by a process P_j , P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.
- Property P1 is satisfied by (1); Property P2 by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.

Lamport's Algorithm– example

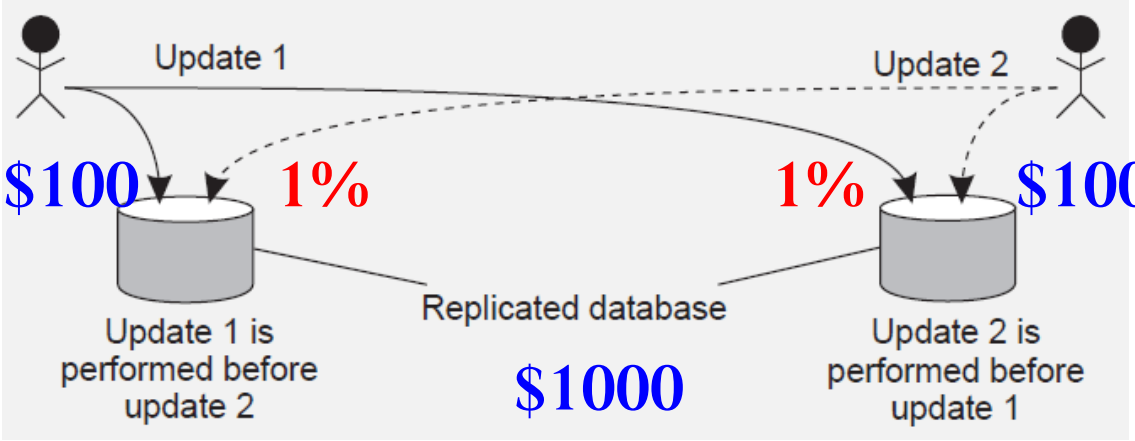
- Adjustments take place in the middleware layer



- If two events happen in processes 1 and 2, both with time 40, the former becomes 40.1 and the latter becomes 40.2

Example: Totally ordered multicast

- We sometimes need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere:
  - P1 adds $100 to an account (initial value: $1000)
  - P2 increments account by 1%
  - There are two replicas



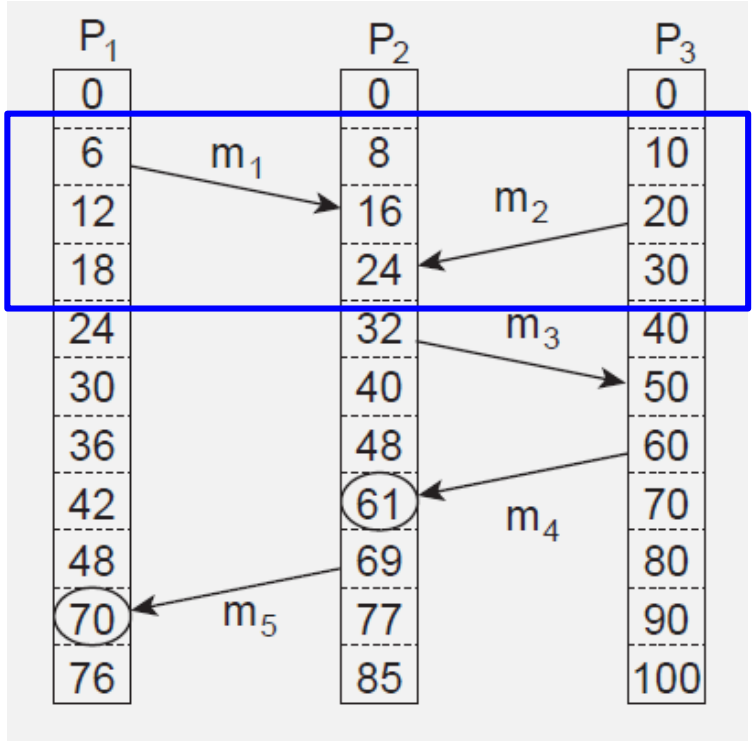
- In absence of proper synchronization: replica #1 ← $1111, while replica #2 ← $1110.

Example: Totally ordered multicast

- Solution
  - Process P_i sends timestamped message msg_i to all others. The message itself is put in a local queue $queue_i$.
  - Any incoming message at P_j is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.
- P_j passes a message msg_i to its application if:
  - msg_i is at the head of $queue_j$
  - for each process P_k , there is a message msg_k in $queue_j$ with a larger timestamp.
- We are assuming that communication is reliable and FIFO ordered.

Vector clocks

- Lamport's clocks:
  - If a causally preceded b then timestamp(a) < timestamp(b)
  - Do not guarantee that if C(a) < C(b) that a causally preceded b
- We cannot conclude that a causally precedes b.



Event a: m_1 is received at T = 16

Event b: m_2 is received at T = 24

Vector clocks

- Solution
  - Each process P_i has an array $VC_i[1..n]$, where $VC_i[j]$denotes the number of events that process P_i knows have taken place at process P_j.
  - When P_i sends a message m, it adds 1 to $VC_i[i]$, and sends VC_i along with m as vector timestamp $vt(m)$. Result: upon arrival, recipient knows P_i's timestamp.
  - When a process P_j delivers a message m that it received from P_i with vector timestamp $ts(m)$, it

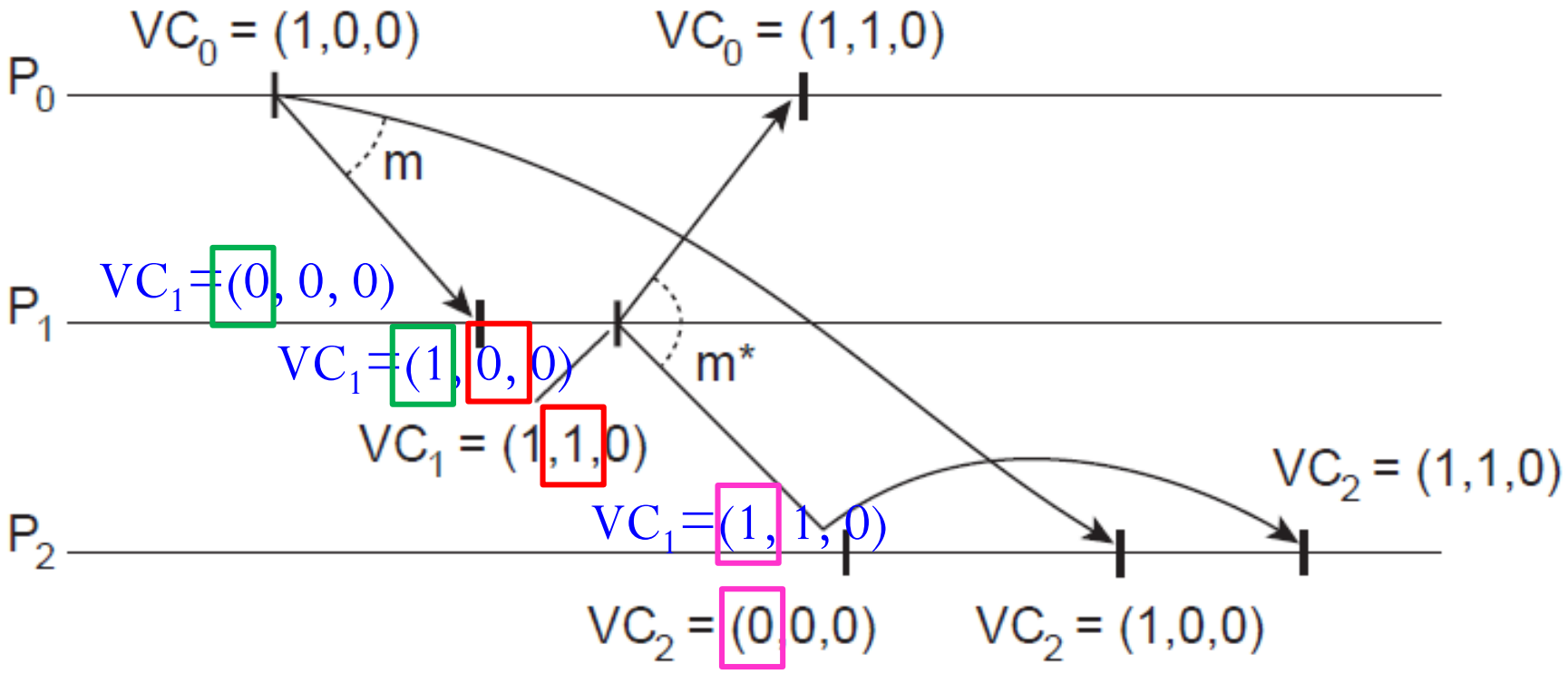
    (1) updates each $VC_j[k]$ to $\max\{VC_j[k], ts(m)[k]\}$

    (2) increments $VC_j[j]$ by 1.

Causally ordered multicasting

- We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.
- Adjustment
  - P_i increments $VC_i[i]$ only when sending a message, and P_j "adjusts" VC_j when receiving a message (i.e., effectively does not change $VC_j[j]$).
- P_j postpones delivery of m until:
  - $ts(m)[i] = VC_j[i] + 1$.
  - $ts(m)[k] \leq VC_j[k]\ for\ k \neq i$.

Causally ordered multicasting

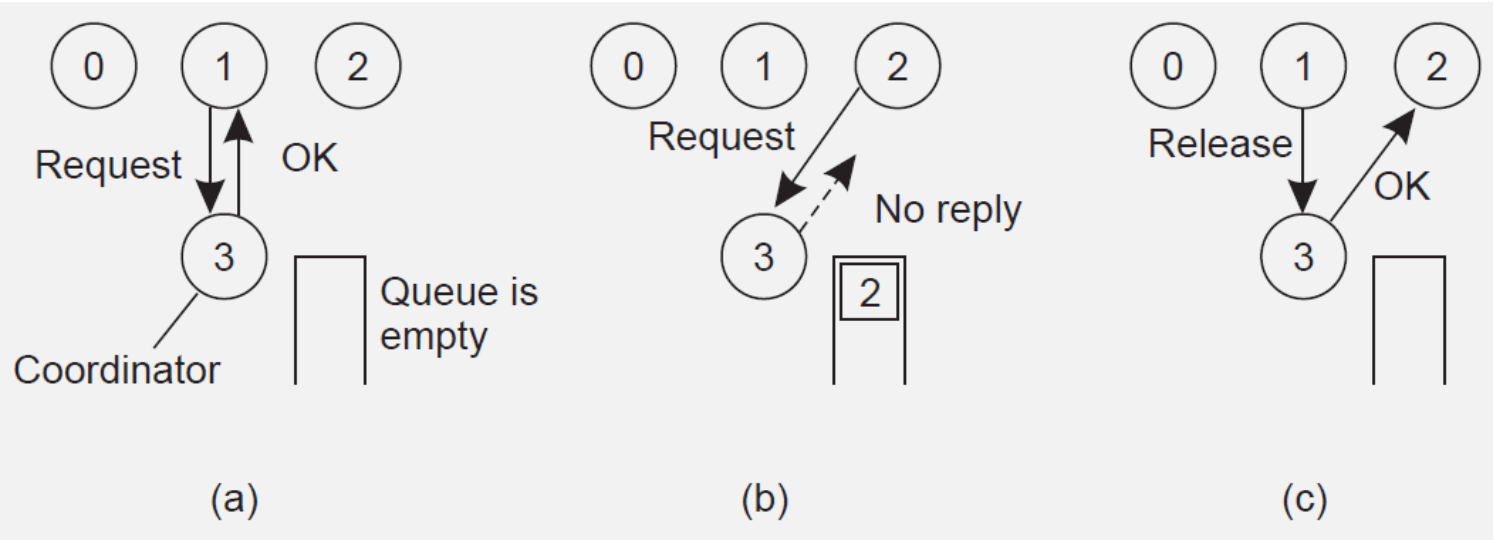


$$ts(m)[i] = VC_j[i] + 1.$$
$$ts(m)[k] \leq VC_j[k] \; for \; k \neq i.$$

Mutual exclusion

- A number of processes in a distributed system want exclusive access to some resource.
- Basic solutions
  - Via a centralized server.
  - Completely decentralized, using a peer-to-peer system.
  - Completely distributed, with no topology imposed.
  - Completely distributed along a (logical) ring.

Centralized Mutual Exclusion



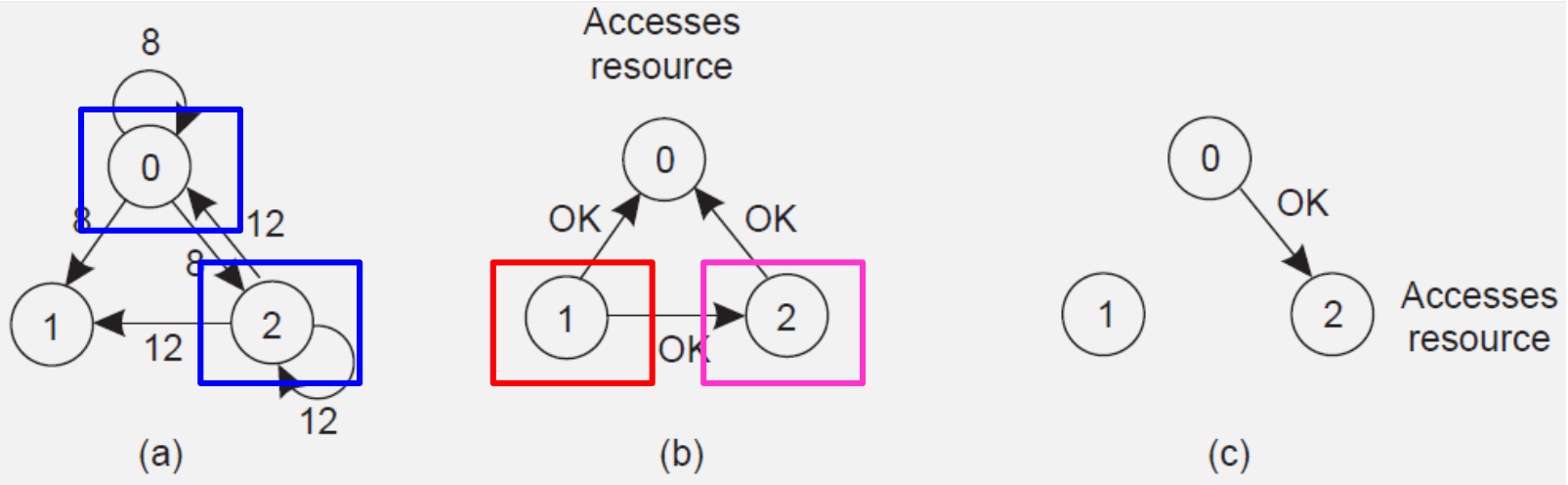
- Advantages
  - Obviously it guarantees mutual exclusion
  - It is fair (requests are granted in the order in which they are received)
  - No starvation (no process ever waits forever)
  - Easy to implement (only 3 messages: request, grant and release)
- Shortcomings
  - Coordinator: A single point of failure; A performance bottleneck
  - If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied"

A Distributed Algorithm

- Based on a total ordering of events in a system (happens-before relation)
- Algorithm:
  - When a process wants to enter a critical region:
    - **Builds a message**: {name of critical region; process number; current time}
    - **Sends the message** to all other processes (assuming reliable transfer)
  - When a process receives a request message from another process:
    - If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender
    - If the receiver is already in the critical region, it does not reply. Instead it queues the request
    - If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp with the one contained in the message it has sent everyone: If the incoming message is lower, the receiver sends back an OK message; otherwise the receiver queues the request and sends nothing

Example

- 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.

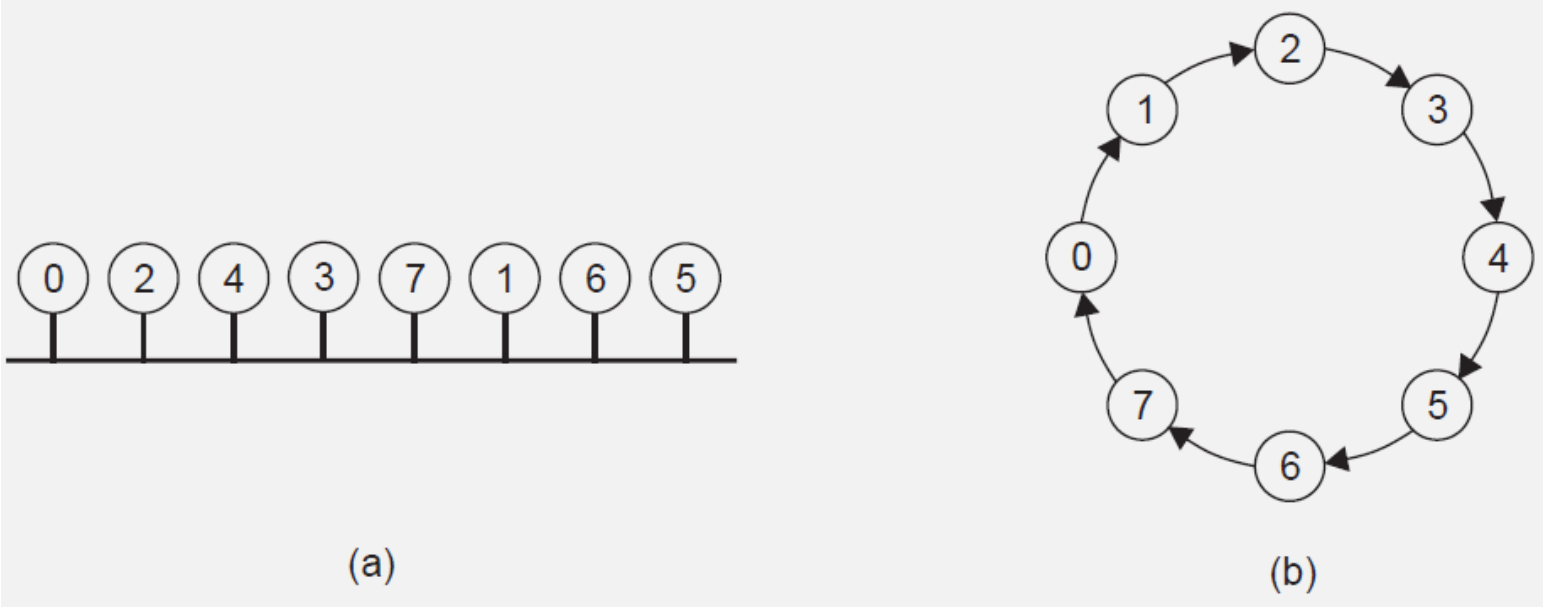


Problems

- The single point of failure has been replaced by *n* points of failure: if any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions
- If reliable multicasting is not available, each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing
- Slower, more complicated, more expensive, and less robust than centralized algorithm!

Mutual exclusion: Token ring algorithm

- Organize processes in a logical ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to).



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.

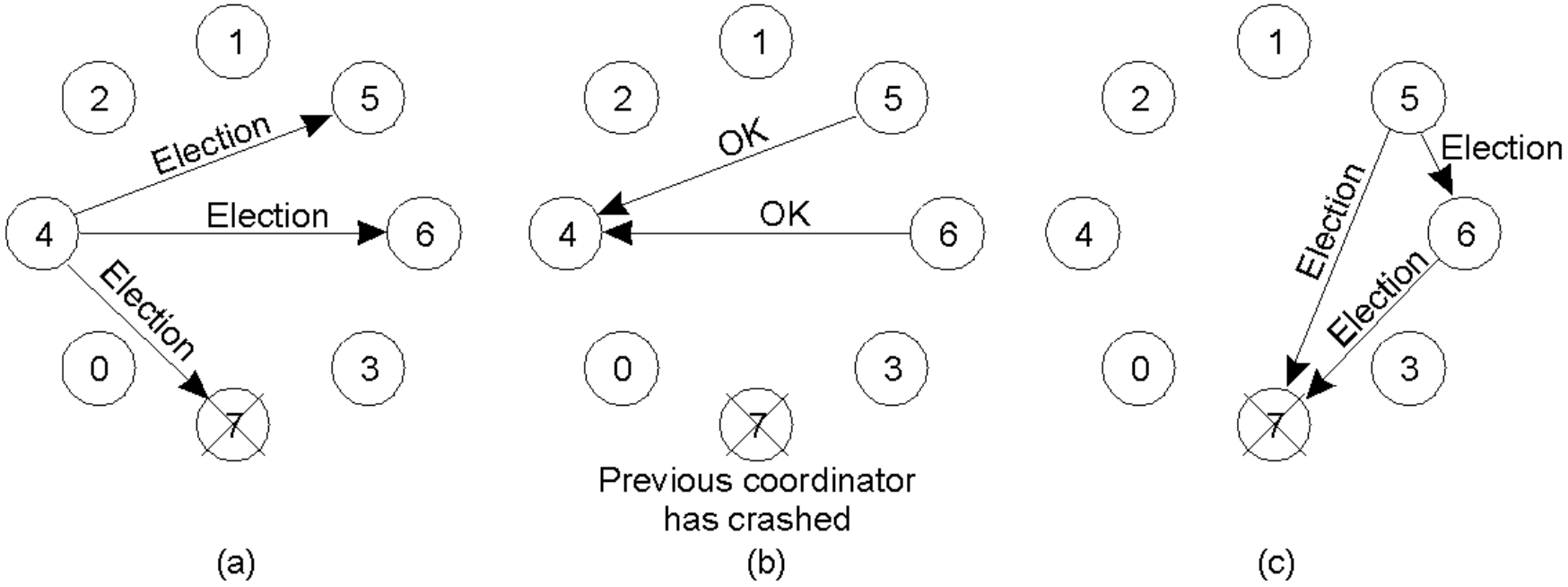
Election algorithms

- An algorithm requires that some process acts as a coordinator. The question is how to select this special process dynamically.
- In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions ⇒ single point of failure.
- If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?

Election by bullying

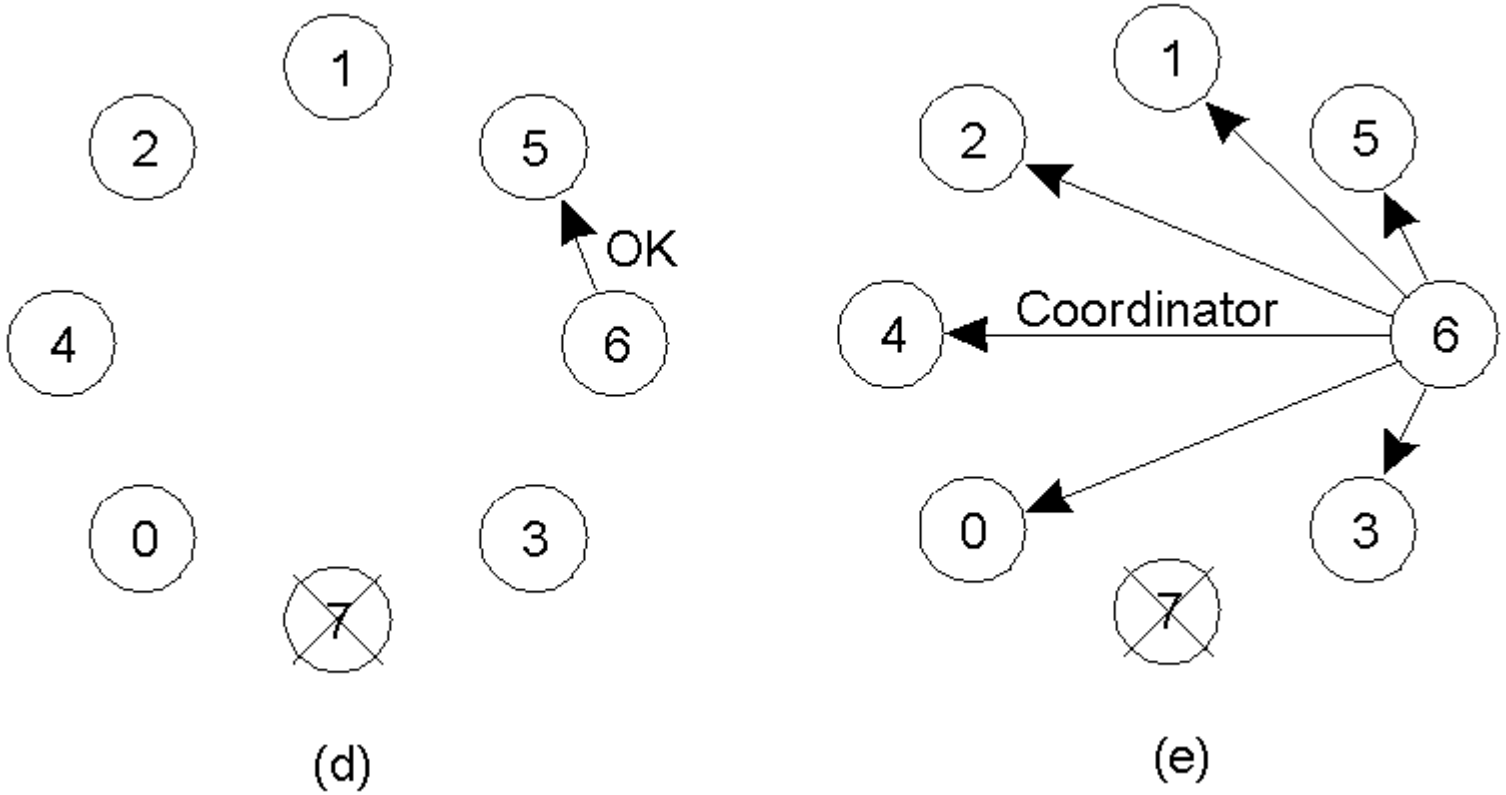
- Each process has an associated priority (weight). The process with the highest priority should always be elected as the coordinator. Issue How do we find the heaviest process?
  - Any process can just start an election by sending an election message to all other processes (assuming you don't know the weights of the others).
  - If a process P_{heavy} receives an election message from a lighter process P_{light}, it sends a take-over message to P_{light}. P_{light} is out of the race.
  - If a process doesn't get a take-over message back, it wins, and sends a victory message to all other processes.

The Bully Algorithm (1)



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

The Bully Algorithm (2)



d) Process 6 tells 5 to stop

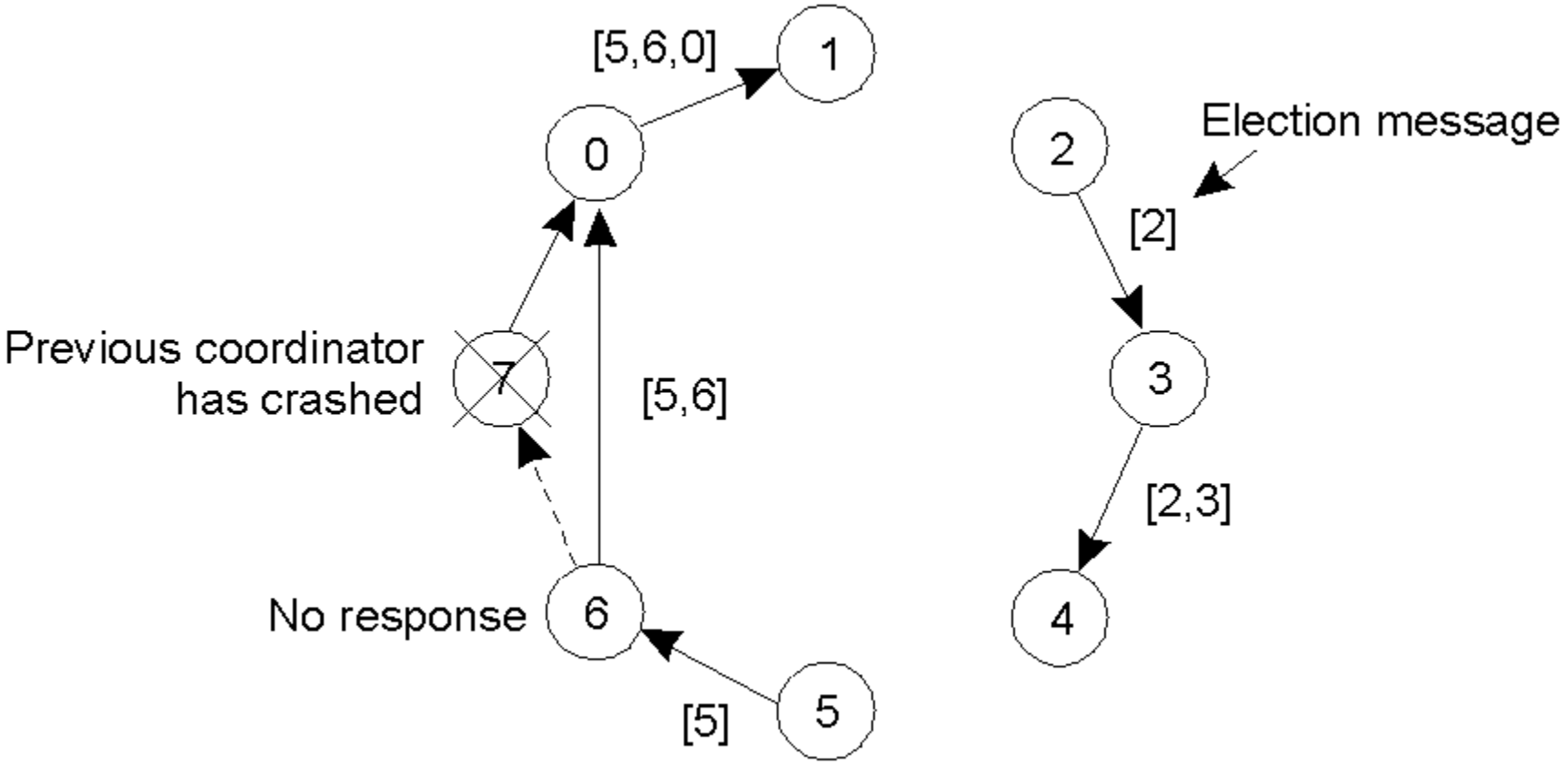
e) Process 6 wins and tells everyone

Election in a ring

- Process priority is obtained by organizing processes into a (logical) ring. 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.

A Ring Algorithm

- Election algorithm using a ring.



Assignment 1

- Reading the research papers in Edge Computing, Distributed Machine Learning Systems, Network Functions Virtualization, then writing a survey / research report in one of the three research areas.

- Please write the report with the provided template and the length of the report is expected to be 6~8 pages.

- Submission: please email the report to distrisys AT 126.com; Deadline: 23:59, April 26, 2019.