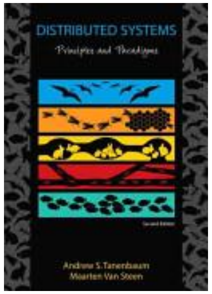
Chapter 6: SYNCHRONIZATION

How to agree on the order of events when there is no global clock?



Thanks to the authors of the textbook [**TS**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.

Turgay Korkmaz
korkmaz@cs.utsa.edu

Chapter 6: SYNCHRONIZATION

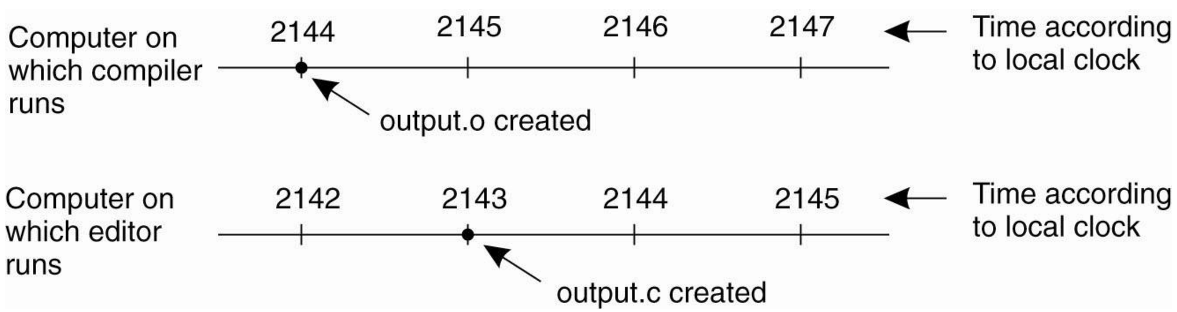
- CLOCK SYNCHRONIZATION
  - Physical Clocks
  - Global Positioning System
  - Clock Synchronization Algorithms
- LOGICAL CLOCKS
  - Lamport’s Logical Clocks
  - Vector Clocks
- MUTUAL EXCLUSION
  - A Centralized Algorithm
  - Decentralized Algorithm
  - A Distributed Algorithm
  - A Token Ring Algorithm
- GLOBAL POSITIONING OF NODES
- ELECTION ALGORITHMS

Objectives

- To understand synchronization and related issues in DS
- To learn about clocks and how to sync them

Introduction

- Synchronization is much harder in DS than single systems because there is no global clock in DS
- What are the implications of not having a global clock?
  - An event that occurred after another event may nevertheless be assigned an earlier time.



  - Many applications (finance, security, collaborative sensing) depend on accurate time…
- So, clocks must be synchronized.

Physical Clocks

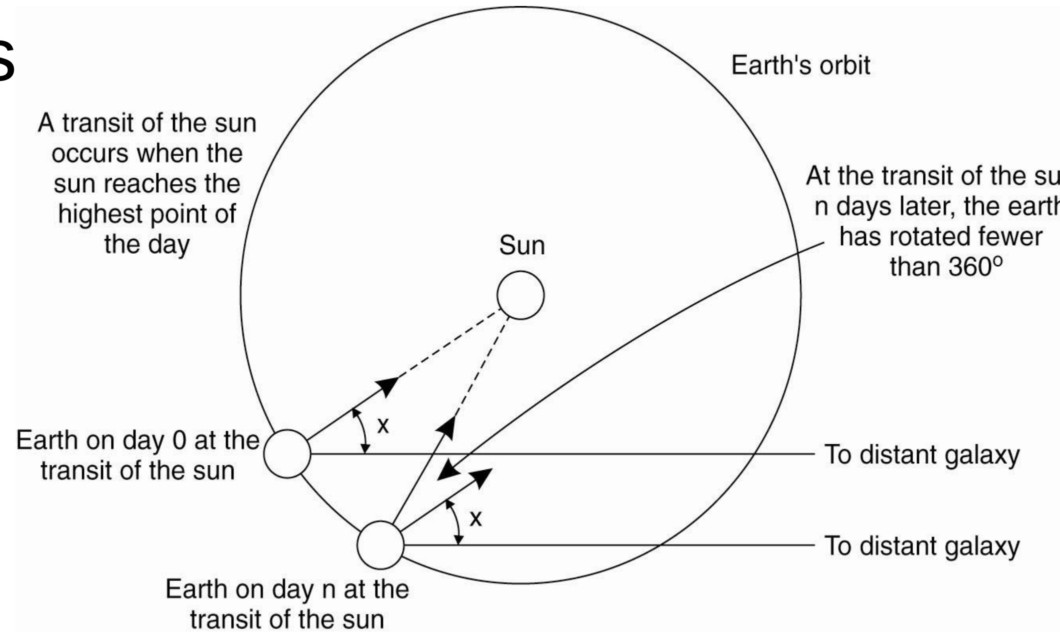
- Clock vs. Timer
- A quartz crystal oscillates at a well-defined frequency
  - Associate two registers counter and holding register
  - Set holding register to a value x

  1. Counter ← holding register
  2. For each oscillation counter--;
  3. When counter reaches 0, interrupt (clock tick) to update software clock
  4. Go to 1.
- Different quartz crystals may oscillate at different rates.
- So, this may cause two clocks to differ from each other (called clock skew)
- How to sync N clocks with a **global clock** or with each other?

Global Clock

How time in real world is actually measured?

- Astronomical time is based on the computation of the mean solar day
- Earth's rotation is variable



- Atomic clock
  - The interval that it takes the cesium 133 atom to make exactly 9,192,631,770 transitions.
  - International Atomic Time (TAI) is based on very accurate physical clocks (drift rate 10^{-13})
  - 3msec less than mean solar day... leap seconds

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.
- How can we provide UTC time to people?
  - NIST broadcast a pulse at the start of each second with accuracy of $\pm$10ms. Satellites can give an accuracy of about $\pm$0.5 ms.
  - That is how your atomic clock works!

How to sync N clocks with a global clock?

Let each computer have a UTC receiver.

- ±10ms might be too much for some applications (e.g., GPS)
- It might be costly (e.g., in case of sensor nodes)
- Indoor equipments may not get the UTC signals

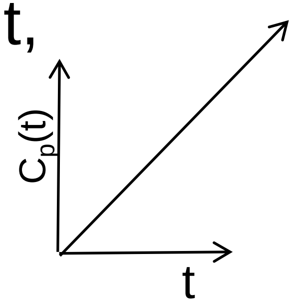
We may have some nodes with a UTC receiver, then can we sync others with those nodes?

What if none have UTC receiver, can we sync them with each other?

Clock Synchronization Algorithms

system model

- All algorithms have the same system model:
  - Each machine has a timer causing H interrupts/sec.
  - The interrupt handler adds 1 to software clock C
  - C keeps track of the number of ticks since some agreed-upon time in the past
  - Let $C_p(t)$ be the clock at p when the UTC time is t,
    - In a perfect world, $C_p(t) = t$ (i.e., $C'_p(t)=dC/dt=1$)
  - The **skew** of a clock is $C'_p(t) - 1$
  - The **offset** relative to a specific time is $C_p(t) - t$



Clock Synchronization Algorithms

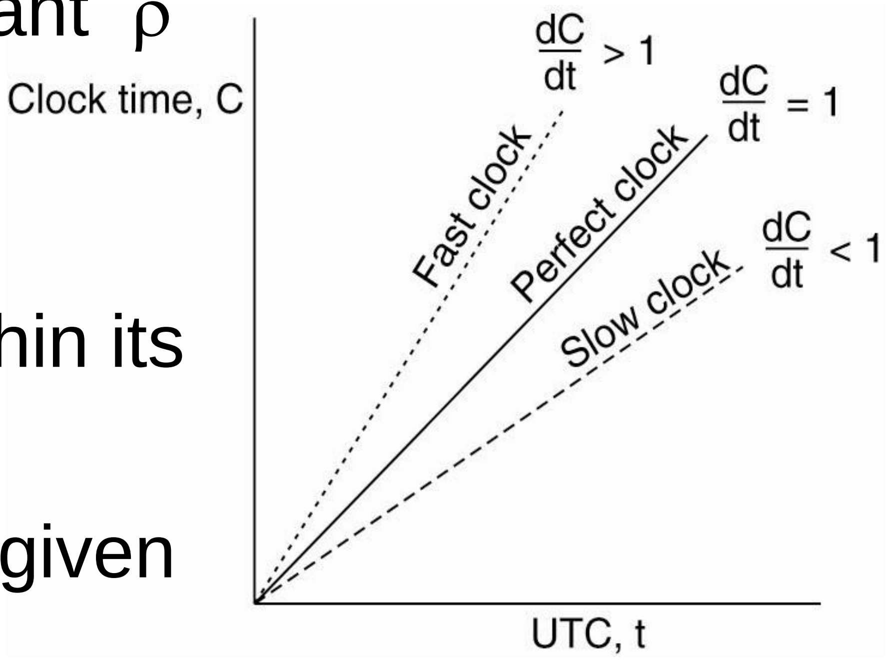
system model

- Real timers do not tick exactly H times per second.
  For example, H=60 should generate 216,000 thick per hour but it may range 215,998 to 216,002 per hour
- So if there exists a constant ρ such that

  $$1-\rho \le dC/dt \le 1+\rho$$

  then, timer is working within its specifications

  ρ (maximum drift rate) is given by the manufacturer



- How often two clocks should be synchronized?

Clock Synchronization Algorithms

-----------------x--------------------------|-------------->

both

sync Δt how much apart?

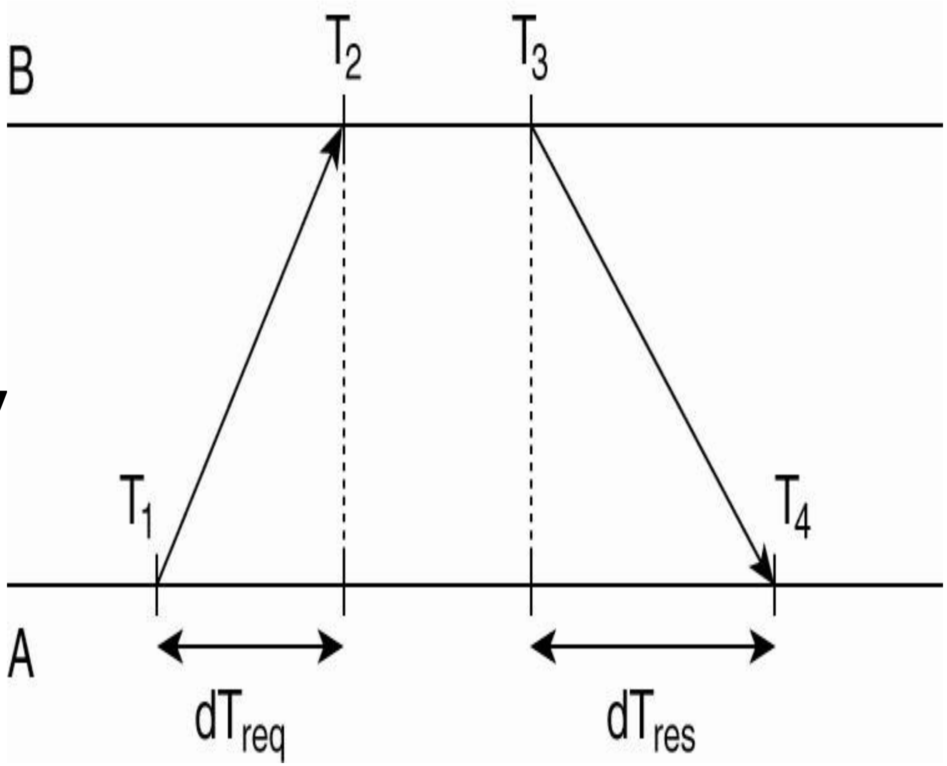
- If two clocks are drifting from UTC in the opposite directions, they would be apart as much as $2\rho\ \Delta t$
- So if we want to guarantee that no two clocks ever differ by more than δ (i.e., $2\rho\ \Delta t < \delta$)

  then we should sync them $\Delta t < \delta/2\rho$ seconds
- Various algorithms differ in precisely how to do this re-sync!
  - NTP (Network Time Protocol)
  - The Berkeley algorithm
  - Clock sync in wireless networks

NTP: basic idea

At least one machine has a UTC receiver

- Suppose we have a server with UTC receiver.
- The server has an accurate clock
- So clients can simply contact it and get the accurate time (every $\delta/2\rho$ sec)



- A gets T1, T2, T3, T4.
- How should A adjust its clock?
- The problem is the delay which causes inaccuracy

NTP: basic idea

Suppose propagation delay is the same in both ways?

- If A's clock is slow

$T2 - \theta - T1 \cong T4 - (T3 - \theta)$

$\theta = ((T2-T1) + (T3-T4))/2$

Add θ to A's clock

- If A's clock is fast

$T2 + \theta - T1 \cong T4 - (T3 + \theta)$

$\theta = ((T4-T3) + (T1-T2))/2$

Subtract θ from A's clock

But, time cannot run backward

Introduce the difference gradually (e.g., instead of 10ms add 9ms for each interrupt for 1 sec)

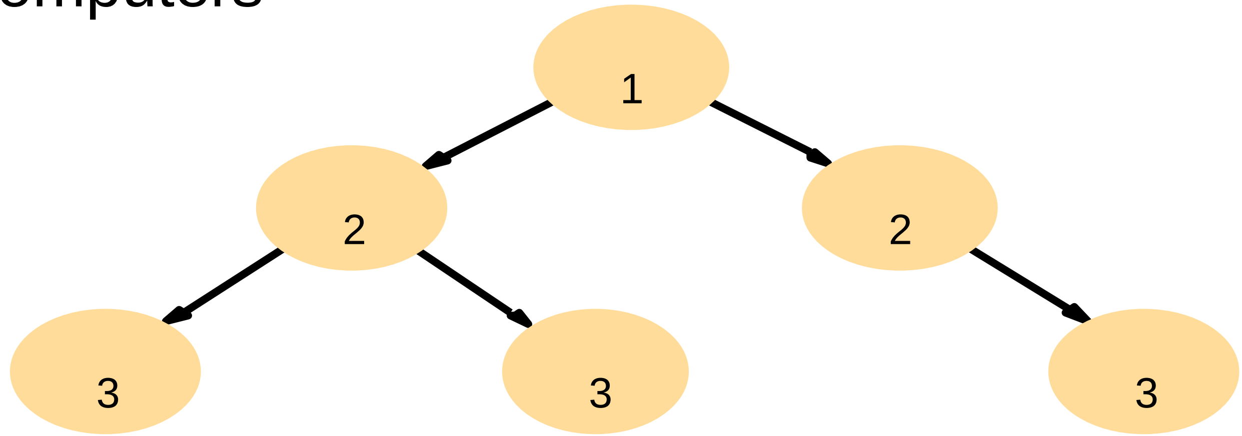
NTP

At least one machine has a UTC receiver

- Use this basic idea in a pairwise manner to distribute time information over the Internet.
- Objectives
  - Enable clients on Internet to be synchronized to UCT
  - Reliable service through redundant servers/paths
  - Provide protection against interference with the time service, whether malicious or accidental
- **Need**: accurate measure of round trip delay, interrupt handling & processing messages

NTP (cont.)

- Provided by a network of **servers** located across the Internet
- Primary servers are connected to UCT sources
- Secondary servers are synchronized to primary servers
- Synchronization subnet - lowest level servers in users' computers



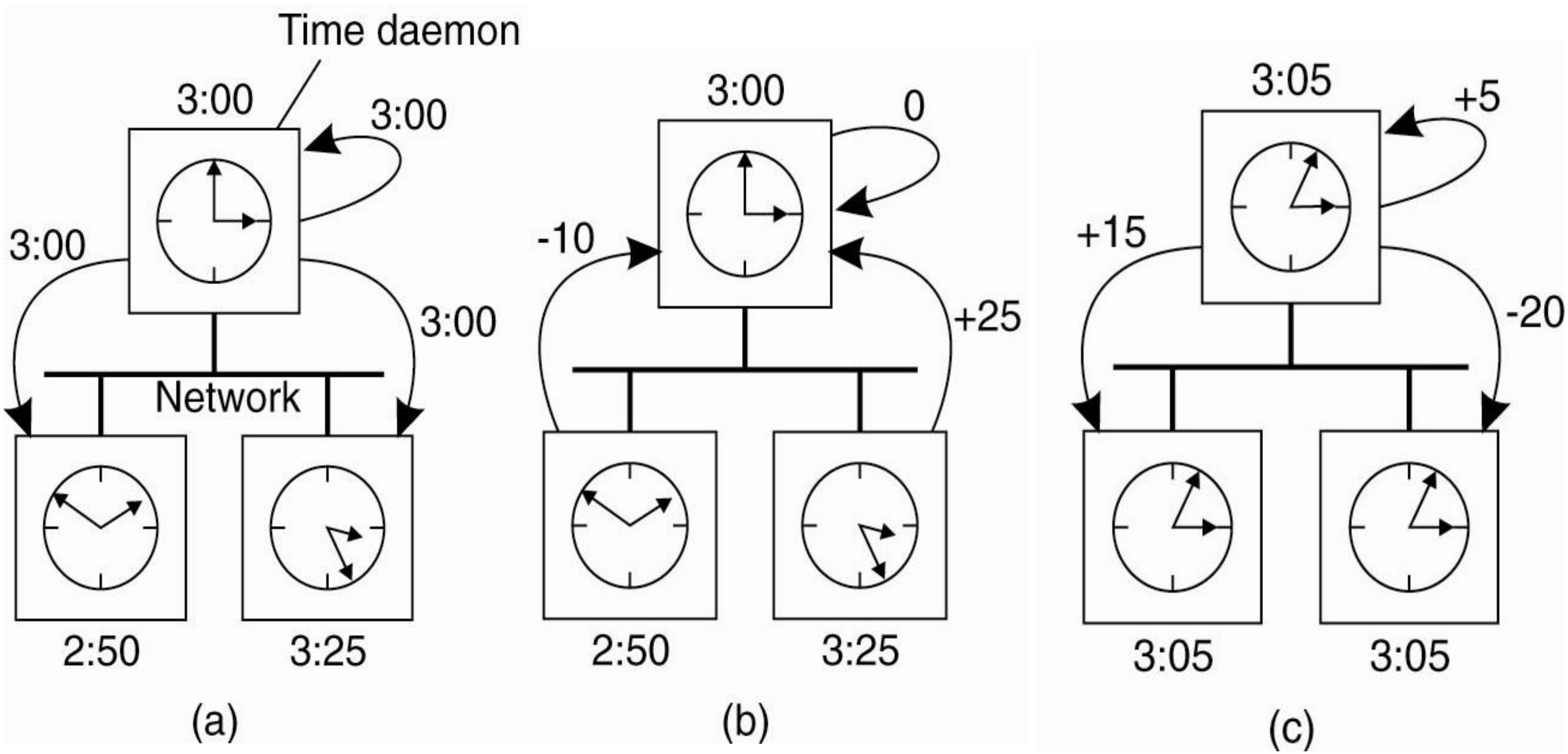
Berkeley Algorithm

No machine has UTC receiver

- Operator manually sets the time at the time server (daemon)
- Time server is active and does the followings:
  - periodically poll all machines
  - compute the average and
  - tell other machines to adjust their times
    - gradually slow down or advance the clock

Berkeley Algorithm

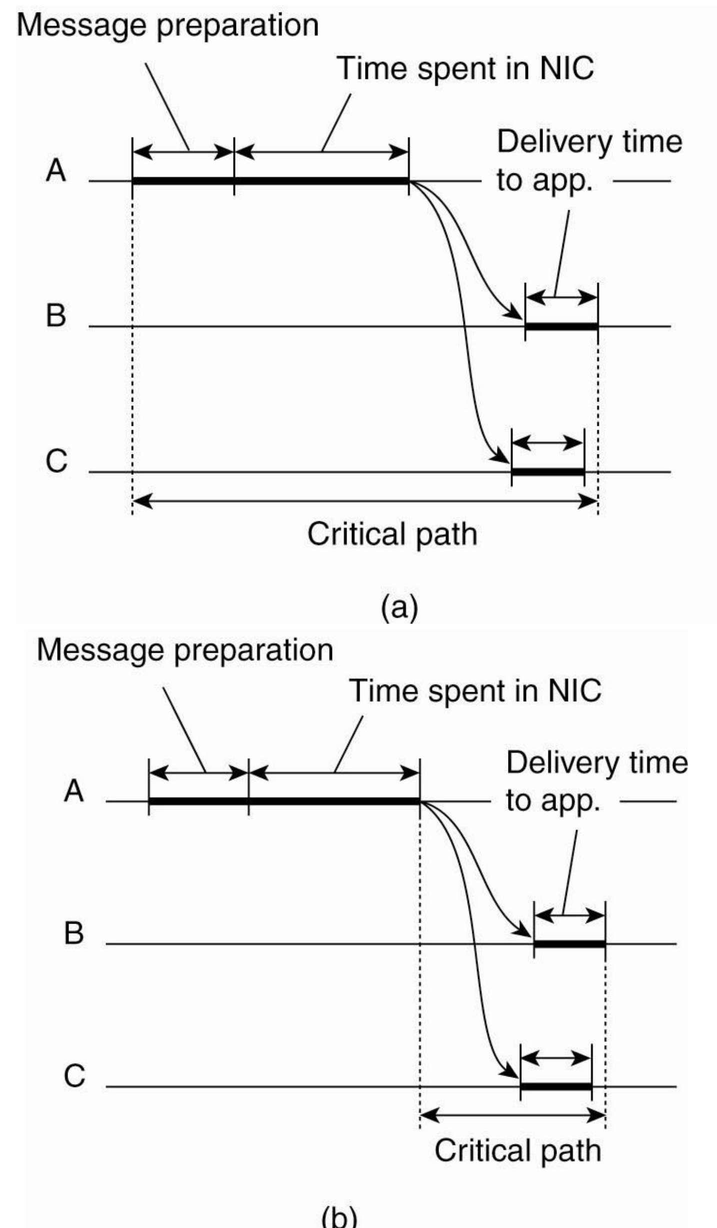
No machine has UTC receiver



- Time does not need to be the actual time…
- As long as all machines agree, then that is OK for many applications
- Gradually advance or slow down the clock…

Clock Sync in Wireless networks

- No time server
- Nodes may not contact each other
- Resource constrained
- Multi-hop routing is expensive and has variable delay
- New algorithms are needed
  - Simply taking average may not work
  - New methods using linear regression is used



Knowing exact time

Knowing an agreed time

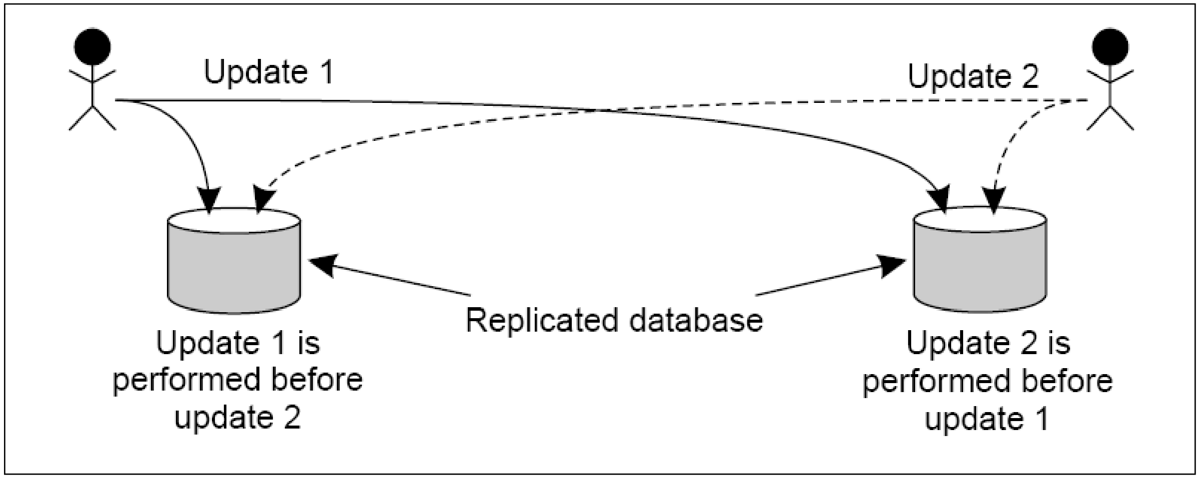
One step further: agree on the ordering of events

LOGICAL CLOCKS

Time in Distributed Systems

- Example: update replicated databases ($1000)

Add $100



Add 1% interest

Result $1111

Result $1110

- Different orders: lead to inconsistency
- We must execute these updates in the same order.
- If we can, then there may be no need for a global clock in a distributed system

How to order events?

- The order of two events occurring at two different computers **cannot** be determined based on their "**local**" time unless they are sync with a global clock.
- Let us first introduce a notion of ordering, namely **happens-before** relation (→) to capture the causal dependencies between events
  - If A and B are events in the **same process** and A occurred before B, then A → B
  - If A is the sending of a message and B is the receipt of that message in a **different process**, then A → B
  - If A→B, and B → C, then A → C (**transitive**).
- This introduces a *partial ordering of events* in a system with concurrently operating processes.

How to order events?

- **Problem**: We need a way of measuring time to assign a time value *C(a)* to every event *a* such that

  if $a \rightarrow b$ then $C(a) < C(b)$

- **Solution**: 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 $a \rightarrow b$, then we demand that $C(a) < C(b)$.

  P2: For different processes, if event *a* corresponds to sending a message *m*, and *b* to the receipt of that message, then also $C(a) < C(b)$.

- Another problem: How to attach a timestamp to an event when there's no global clock?
- Maintain a consistent set of **logical clocks**, one per process.

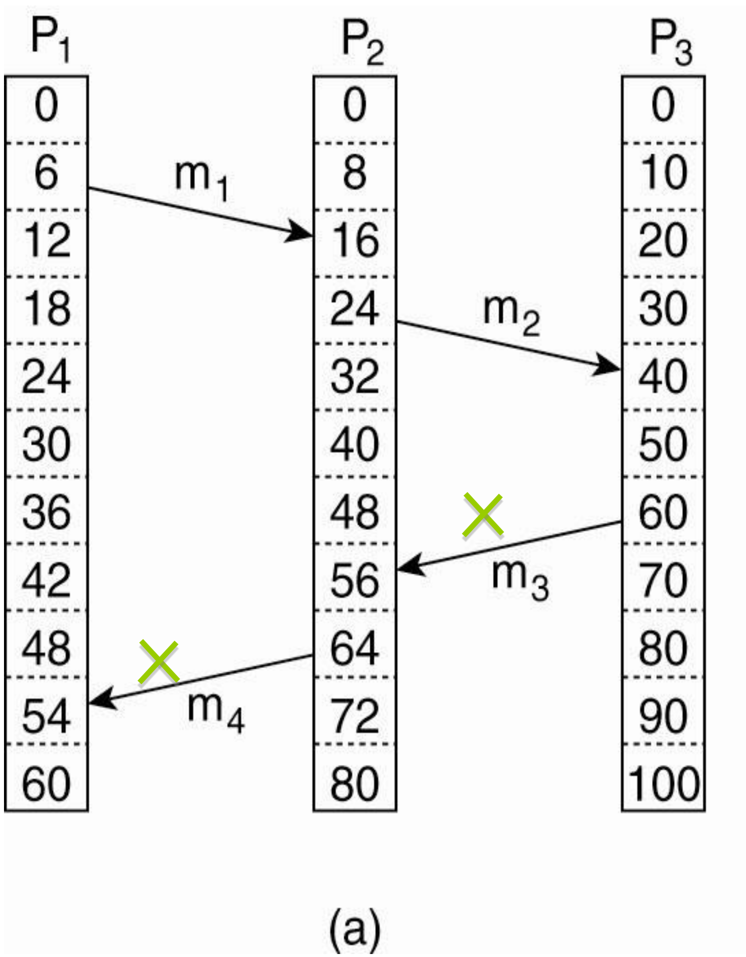
Logical Clocks (Lamport, 1978)

- Each process *Pi* maintains a logical clock *Ci*, which is a monotonically increasing **software counter** (if event *a* happens at *Pi*, then *C(a)* ← *Ci)*
- Update the logical clock/counter as follows:
  1. For any two successive events (e.g., send/receive message) that take place within *Pi*, *Ci* is incremented by 1
  2. Each time a message *m* is sent by process *Pi*, the message receives a timestamp *ts*(*m*) = *Ci;*
  3. Whenever a message *m* is received by a process *Pj*, *Pj* adjusts its local counter *Cj* to max{*Cj*, *ts*(*m*)}; then Cj++ before passing *m* to the application;

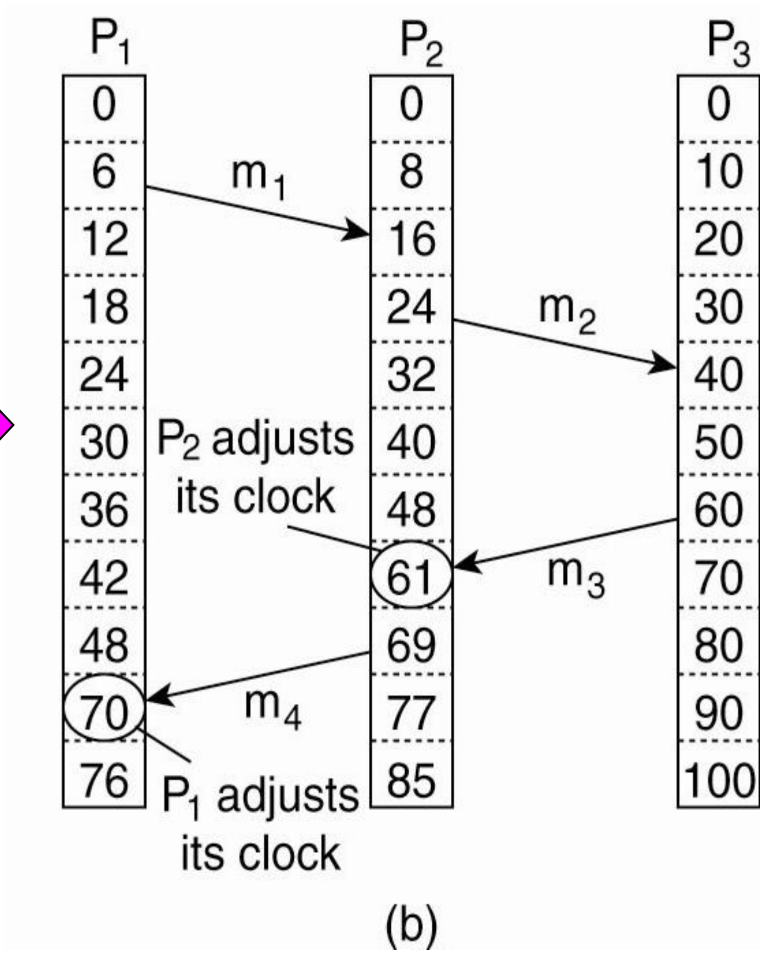
Notes

- 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. Time.id

Logical Clock: Example



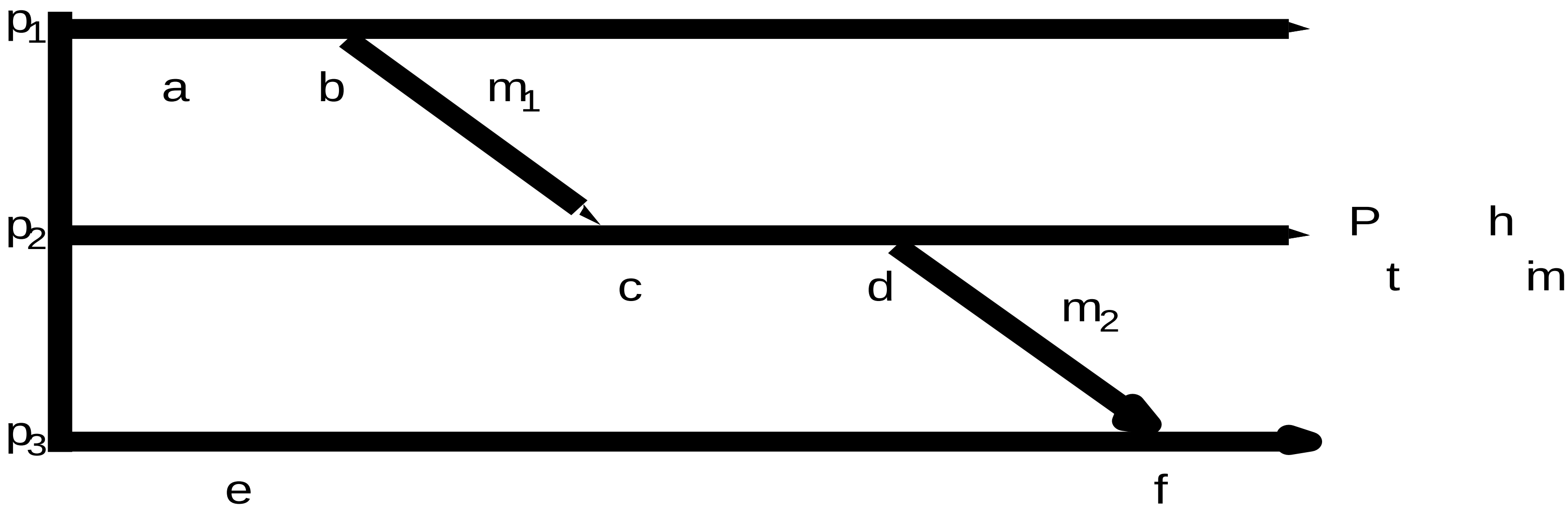
(a)



(b)

Logical Clock: Properties

"Happened Before": Partial Order



- $a \rightarrow b$ (at $p1$) $c \rightarrow d$ (at $p2$); $b \rightarrow c$; also $d \rightarrow f$
- Not all events are related by the "$\rightarrow$" relation
  - *a* and *e* (different processes and no message chain)
  - they are not related by "$\rightarrow$"
  - they are said to be <u>concurrent</u> (written as $a \,||\, e$)

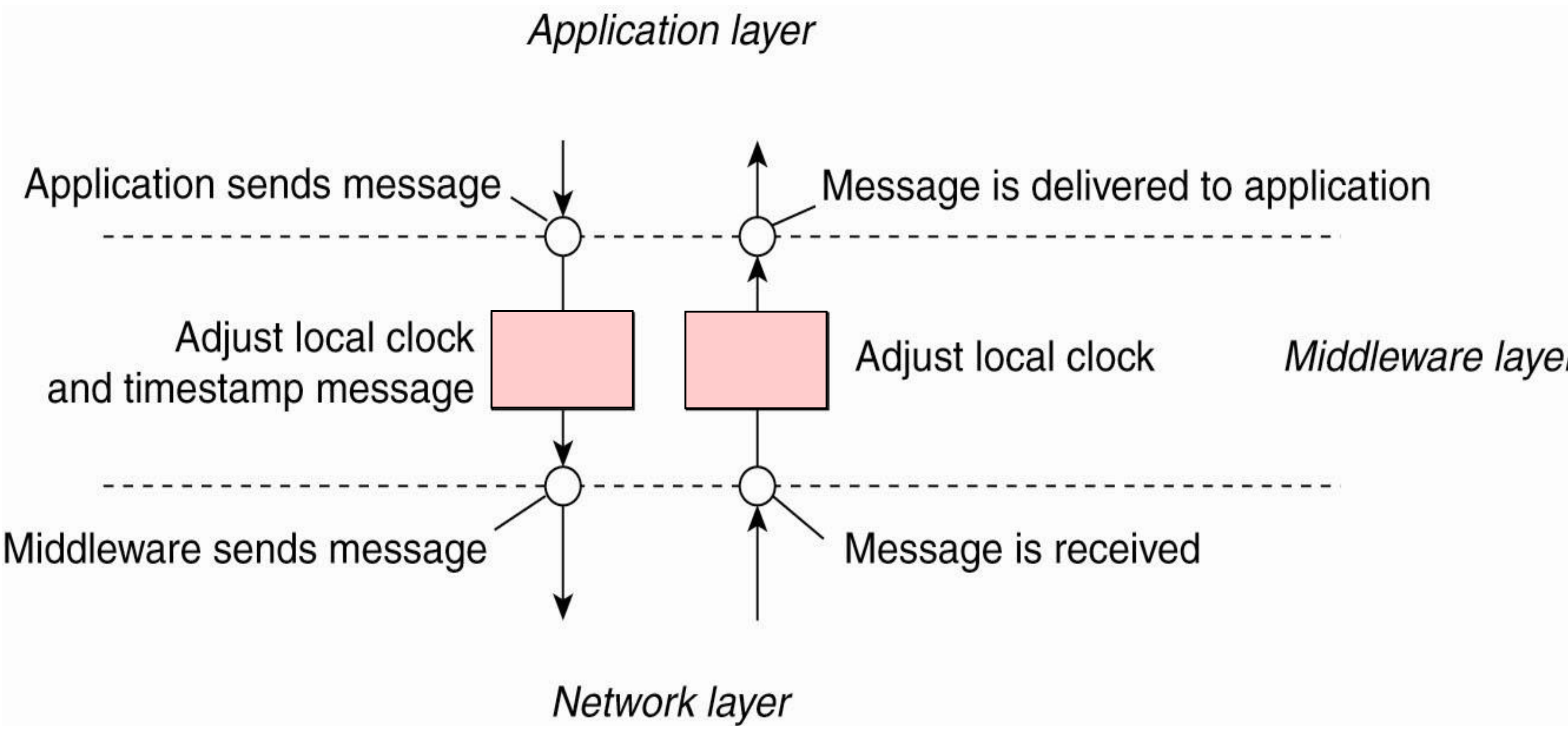
Logical Clock: Properties

irreflexive partial order

- *e* → *e*' implies *L*(*e*) < *L*(*e*')
- The converse is not true, that is *L*(*e*) < *L*(*e*') does not imply *e* → *e*'. (e.g. *L*(*b*) > *L*(*e*) but *b* || *e*)
- Lamport's "happened before" relation defines an **irreflexive *partial order*** among the events in the distributed system

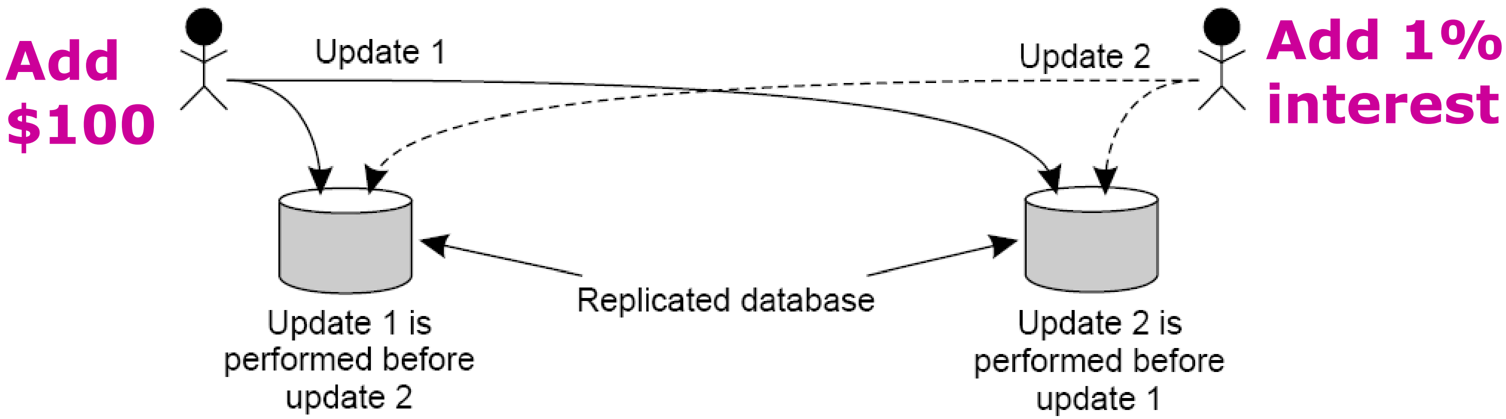
Logical Clock: Where to Put It?

- The positioning of Lamport's logical clocks in distributed systems



Example: Logical clocks in Totally-Ordered Multicast

- Consider the bank example we discussed before



- For consistency, both server should execute u1, u2 or u2, u1 at both sides…
- This requires totally-order multicast, where all messages are delivered in the same order to each receiver

Totally-Ordered Multicast

- Consider a group of n distributed processes,
- m ($\leq n$) processes multicasts "update" messages
  - How to guarantee that all the updates are performed ***in the same order*** by all the processes?
- Assumptions
  - No messages are lost (**Reliable delivery**)
  - Messages from the <u>same</u> sender are received in the order they were sent (**FIFO**)
  - A copy of each message is also sent to the sender

Totally-Ordered Multicast (cont.)

- Process *Pi* sends timestamped message *msgi* to all others.

  timestamp (ts) is the logical clock
- The message itself is put in a local queue *queuei*.
- Any incoming message at *Pj* is queued in *queuej*, according to its timestamp, and acknowledged to every other process.
- *Pj* passes a message *msgi* to its application if:
  - (1) *msgi* is at the head of *queuej*
  - (2) for each process *Pk*, there is an **acknowledgement** message *msgk* in *queuej* with a larger timestamp. $ts(msg_k) < ts(ack_k)$
- In essence, messages are ordered according to their timestamps following Lamport's algorithm
- This is very important for replicated servers!

Totally-Ordered Multicast (cont.)

For Example: Replicated Databases

S1 sends request R(u1, 20) at time 20

S2 sends request R(u2, 15) at time 15

S1 receives R(u1, 20) at time 21, and R(u2, 15) at time 22; send ack. for u2 request at time 23;

S2 receives R(u2, 15) at time 16, and R(u1, 20) at time 21; send ack. for u1 request at time 22;

S1's message queue (events re-ordered w. ts)

- *R(u2,**15**):22*, R(u1,**20**):21, A(s2,u1,**22**):24

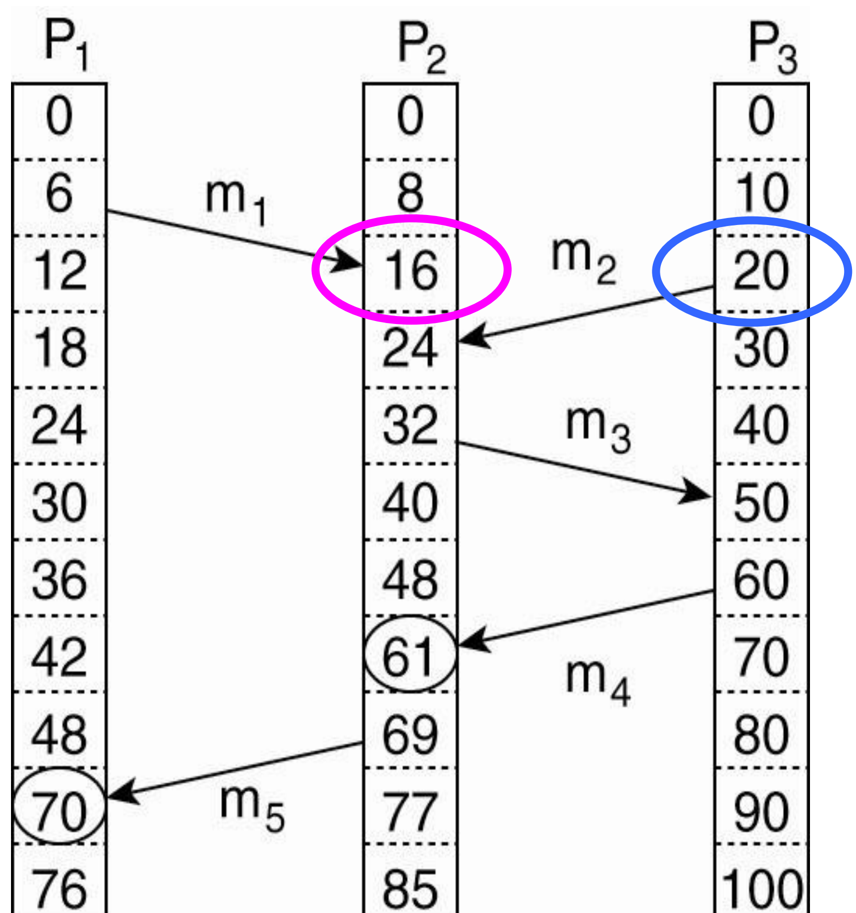
S2's message queue

- R(u2,15):16, R(u1,20):21, A(s1,u2,23):24

So update order: R(u2) → R(u1) on both servers

Problem with Lamport's Clocks

- **Observation:** Lamport's clocks do not guarantee that if $C(a) < C(b)$ THEN *a* **causally preceded** *b*:
  - Event *a*: *m*1 is received at $T = 16$.
  - Event *b*: *m*2 is sent out at $T = 20$.
- We **cannot** conclude that *a* **causally** precedes *b*.
- **Solution**: Vector clocks may capture causality



Vector Clocks

- Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties

  P1: VC_i [i] is the number of events that have occurred so far at P_i. In other words, VC_i [i] is the local logical clock at process P_i .

  P2: If VC_i [j] = k then P_i knows that k events have occurred at P_j. It is thus P_i's knowledge of the local time at P_j .

- To maintain P1 and P2, respectively
  - Increase VC_i [i] when a new event happens at P_i
  - Piggyback vectors along with messages that are sent

Vector Clock: Update

Specifically, perform the following steps:

1: Before executing an event, P_i executes $VC_i [\, i \,] \leftarrow VC_i [i\,] + 1$.

2: When process P_i sends *m* to P_j, it sets *m*'s (vector) timestamp $ts(m) = VC_i$ after Step 1

3: Upon the receipt of *m*, process P_j adjust $VC_j [k\,] \leftarrow \max\{VC_j [k\,], ts(m)[k\,]\}$, for k=1, 2, .. N. Then P_j executes Step 1 and delivers *m* to the application

Pj knows how many events occurred at Pi before Pi sends m

Pj also knows how many events occurred at other processes before Pi sends m (on which m may causally depend on)

Causally-Ordered Multicasting

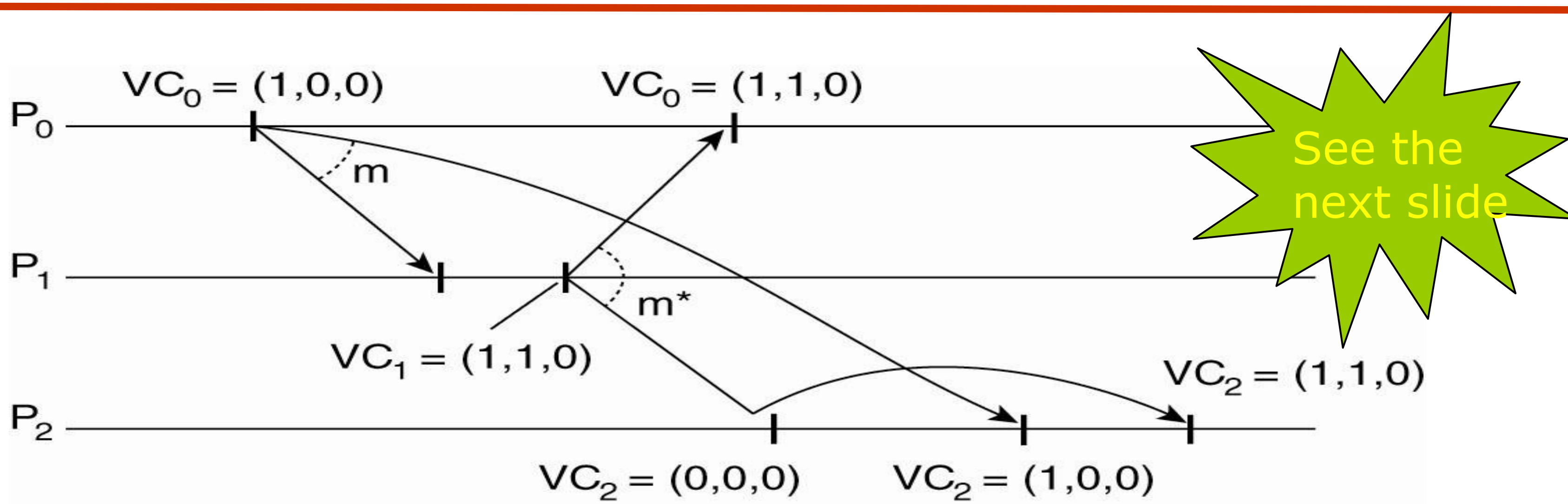
- Ensure to deliver a message only if all causally preceding messages have already been delivered
- Weaker than totally-ordered multicasting (if two messages are not related, they can be ordered in any order or even delivered in different orders at different locations)
- *Assume clocks are **only adjusted** when*
  - *P_i sending and receiving delivering*

    $VC_i[i]$++ $VC_i[k] = \max\{VC_i[k], ts(m)[k]\}$ for $k=1, 2, \ldots N$
- *Upon receiving FIRST CHECK if P_j has to* postpone delivery of *m from P_i* : (if the following cond is false, postpone)
  - *R1:* $ts(m)[i] == VCj[i] + 1$;

    → m is the next message expected from Pi
  - *R2:* $ts(m)[k] \leq VCj[k]$ for $k \neq i$

    → Pj has seen all messages that have been seen by Pi when Pi sent m

Causally-Ordered Multicasting (cont.)



P_j postpones delivery of *m from* P_i until:

R1: ts(*m*)[*i*] == *VCj*[*i*] + 1;

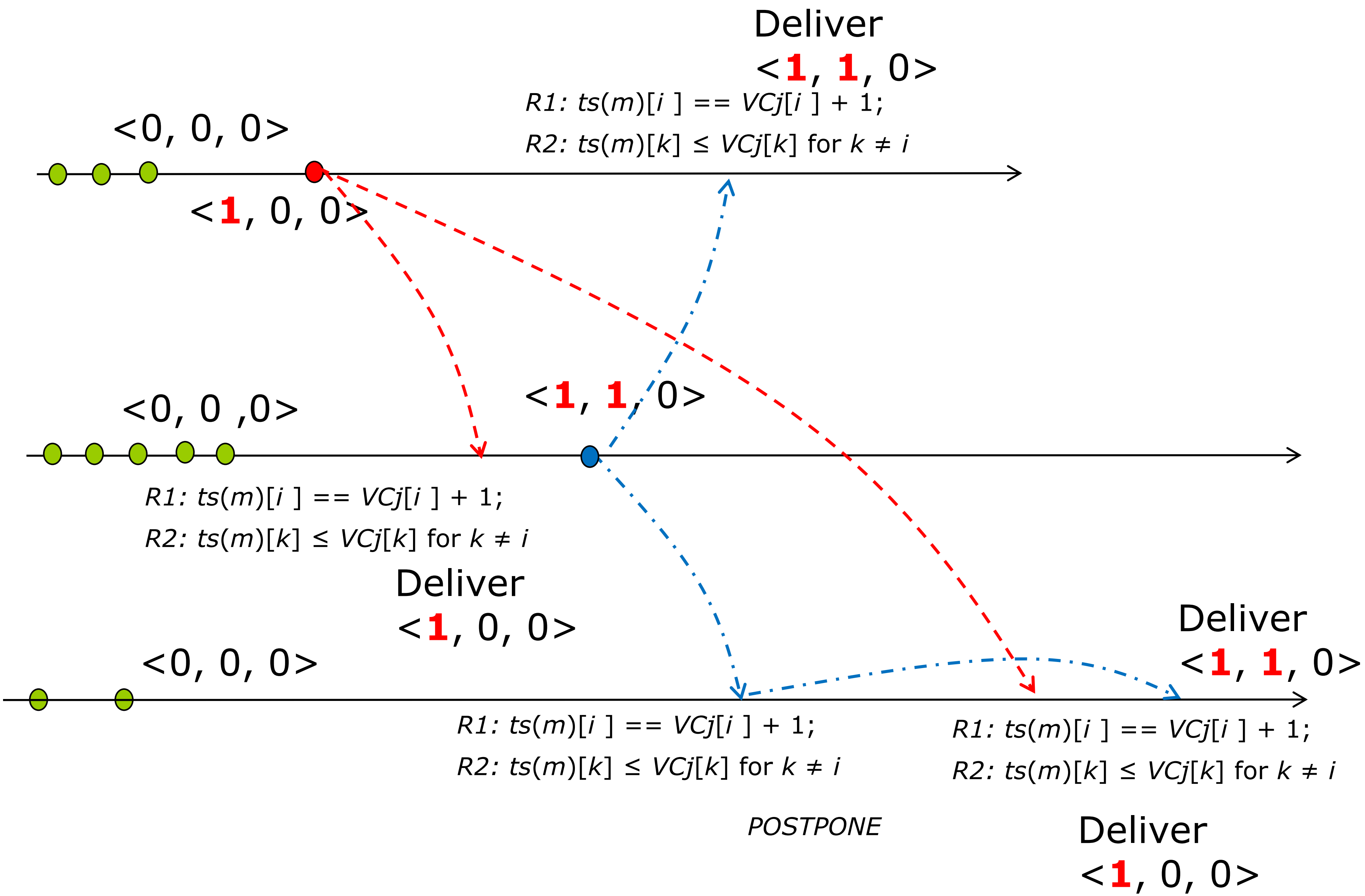
→ m is the next message expected from Pi

R2: ts(*m*)[*k*] ≤ *VCj*[*k*] for *k* ≠ *i*

→ Pj has seen all messages that have been seen by Pi when Pi sent m

- For P2, when receive m* from P1, ts(m*) = (1,1,0), but VC2 = (0,0,0); m* is delayed as R2 is not satisfied;
- Whe P2 receive m from P0 ts(m)=(1,0,0), with VC2=(0,0,0) → both R1 and R2 is ok, and m is delivered → VC2 = (1,0,0) → m* is delivered

Causally-Ordered Multicasting (cont.)



A note on Ordered Message Delivery

This support should be in middleware or application?

- Middleware don’t see the content, so it can only detect potential causalities
- Not all causalities can be detected (3rd channel is used)
- Applications can better deal with these problems, but this is a distraction for the program developer…

Semaphore, monitor, … cannot be used in DS. why?

New solutions are needed to access shared resources…

MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS

Mutual Exclusion in Distributed Systems

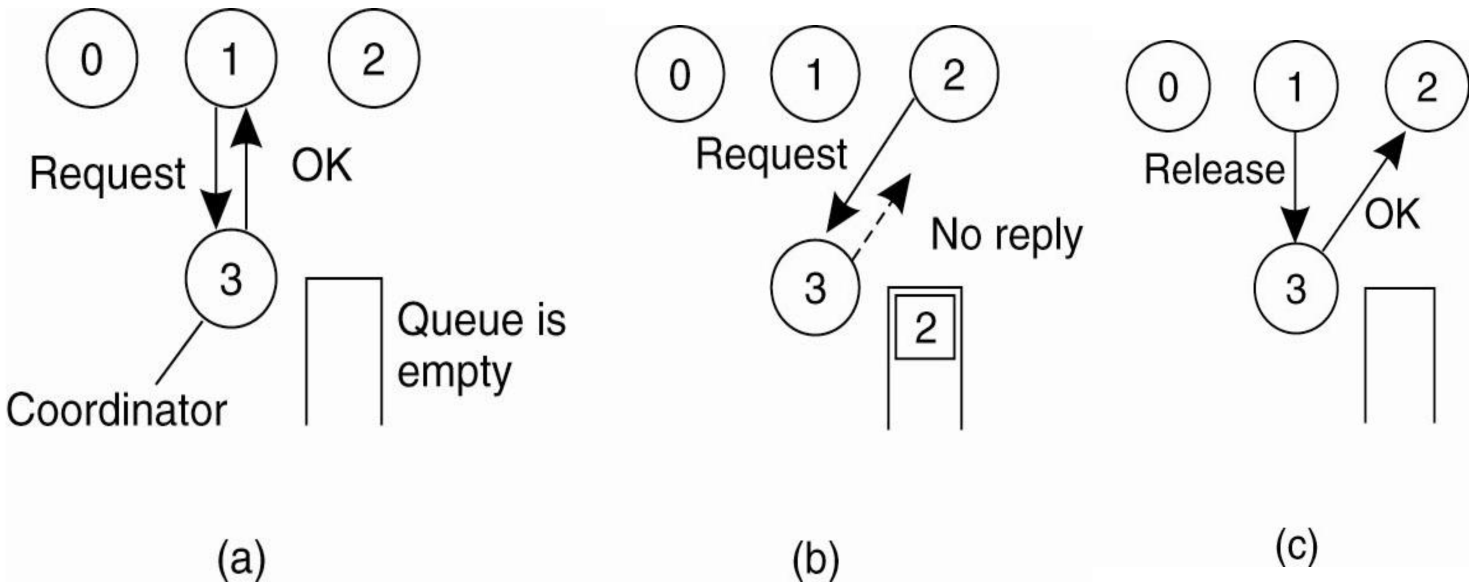
- To provide exclusive access to some resources in DS, we need new approaches
- There are two main approaches:
  - **Token**-based
    - Pass a token (a special msg) between processes
    - There is only one token
    - Whoever has the token uses recourse and passes it to next
    - + Starvation and deadlock can be avoided easily
    - - token might get lost!
  - **Permission**-based
    - The process that wants to access the resource should get permission of other processes
    - There are many ways to do this and we will see a few

Basic Solutions

Solutions

- **Centralized server** (deterministic), using client-server
- **Decentralized** (probabilistic), using a peer-to-peer system
- **Completely distributed** (deterministic)
  - with no topology imposed
  - along a **(logical) ring**

Centralized Mutual Exclusion



- + guarantees mutual exclusion
- + fair, no starvation, no deadlock on one resource
- + easy to implement
- - coordinator is a single point of failure
- - how to distinguish dead coordinator from permission denied (send explicit msg)
- - performance bottleneck

Decentralized Mutual Exclusion

- Vote by extending the central solution as follows:
  - Assume every resource is replicated *n* times, and each replica has its own coordinator
  - A coordinator always responds immediately to a request.
  - Access requires a **majority vote** from $m > n/2$ coordinators;
  - When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted

    (this may cause problem as it may grant another process a permission after recovery)

    - Let p be the prob that a coordinator resets and

      P[k] be prob that k out of m coordinator rests ($P[k] = \binom{m}{k} p^k (1-p)^{m-k}$)
    - At least 2m-n coordinator needs to reset in order to violate the correctness of voting mechanism $P[violation] = \sum_{k=2m-n}^{n} P[k]$
    - With p=0.001, n=32, m=0.75n → incorrect permission grant: 10^{-40}

Decentralized Mutual Exclusion (cont'd)

- To implement the voting mechanisms, a DHT-based system is used (Lin 2004)
- Each resource has a unique name *rname* and i^{th} replica is named *rname-i*
- Every process can generate a n keys given the resource name and look up each node responsible for a replica (and controlling access to that replica)
- If process gets less than m votes, it will back off random amount time before next attempt
- - As request for a resource increases, no one gets majority vote (utilization drops)

Completely Distributed Solution

with no topology imposed

- A probabilistic mutual exclusion algorithm may not be good enough for some
- Can we design a deterministic distributed mutual exclusion algorithm?
- Yes, but this would requires total ordering of all events (which one happened first?)
  - For this, we can use Lamport (1978)
  - Ricart & Agrawala’s alg made it more efficient

    (we will describe this solution)

Ricart & Agrawala's Alg (1)

Distributed solution with no topology imposed

- Process P wants to access R
  - Builds a msg (R, P, C_P (current time at P))
  - Multicast it to all other processes including itself

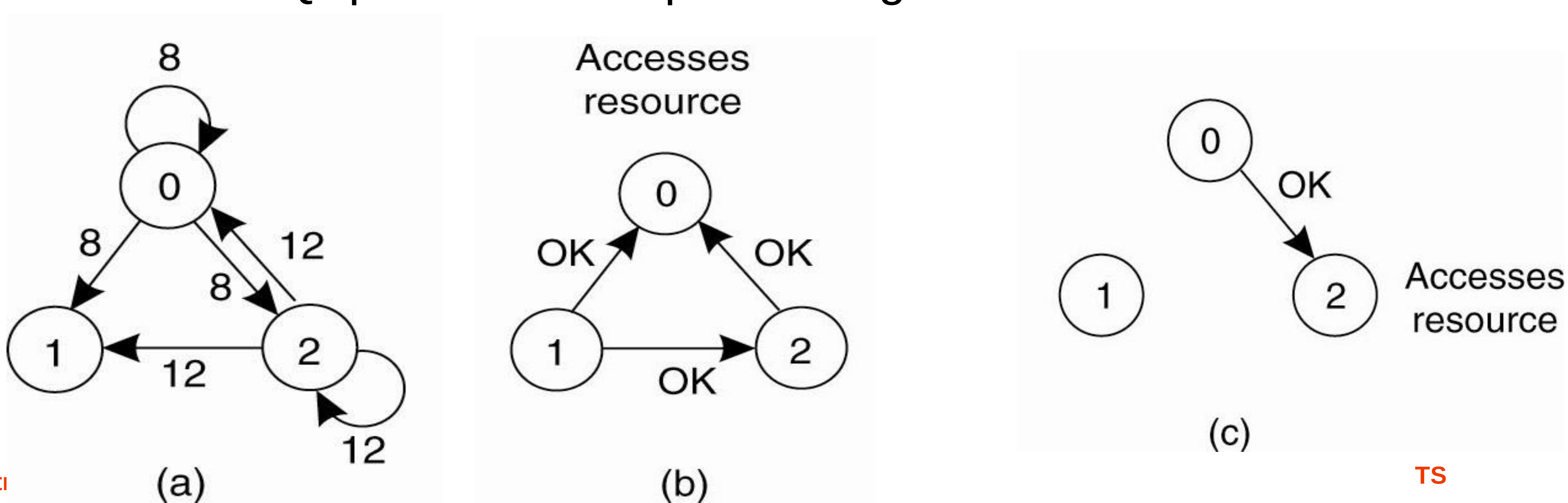
    [assume that comm is reliable, no msg is lost]
  - Waits n OK from others

    → what do other processes do? (see next slide) →
  - Upon receiving n OK, P uses the resource R
  - When P finishes, it sends OK to all in the queue and remove them

Ricart & Agrawala's Alg (2)

Distributed solution with no topology imposed

- Upon receipt of a msg (R, P, C_P), every process Q takes one of the following actions:
  - If Q has no interest in R, send OK to P.
  - If Q already has access to R, queue this request.
  - If Q is waiting for the resource [i.e., (R, Q, C_Q) is in the queue], then compare time stamps, the lowest one wins:
    - If $C_P < C_Q$, then Q sends OK
    - Else Q queues that request and gets OK from P



Ricart & Agrawala's Alg (3)

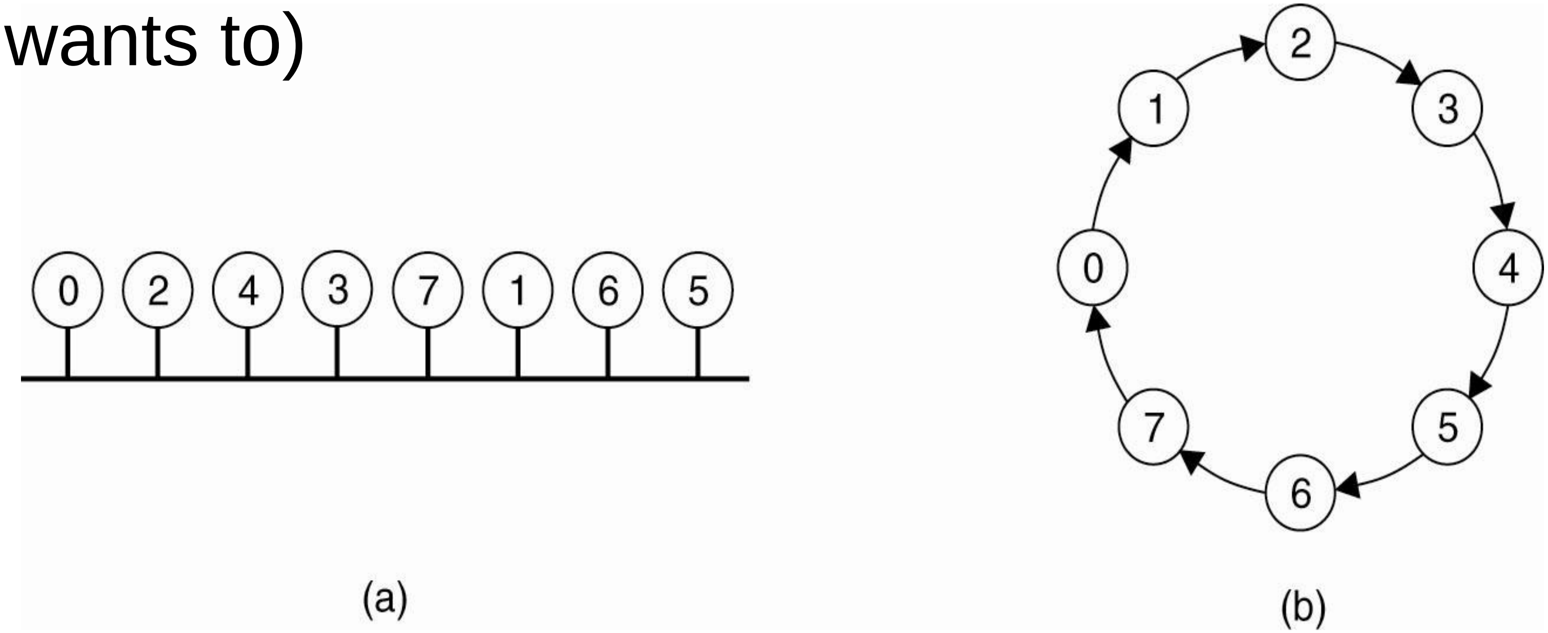
Distributed solution with no topology imposed

- \+ Like centralized algorithm
  - mutual exclusion is guaranteed without starvation or deadlock (on a single resource)
- \- Number of messages per entry is 2(n-1)
- ? No single point of failure
  - Actually, this has n points of failure !!!
  - P[any one of n fails] >> P[one fails]
- ? Who will keep membership list (App or middleware)
- ??? All nodes are involved in all decision (all can be bottleneck)
  - Some improvements can be made (e.g., majority voting)
  - But overall this is slower, more complicated than original centralized solution…
  - Then why bother !!!! ….

Completely Distributed Solution

Logical Token Ring

- 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)



What if the token is lost?
What if a process is dead? (get ACK)

A Comparison of the Four Algorithms

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	3mk, k = 1,2,...	2 m	Starvation, low efficiency
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

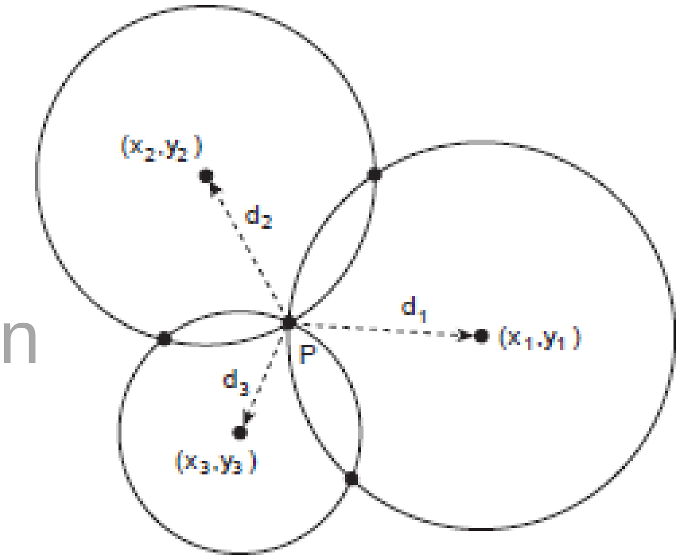
Every one enters critical section

No one is interested in critical section

Global positioning of nodes

Problem

- How can a single node efficiently estimate the latency between any two other nodes in a distributed system?



Solution

- Construct a geometric overlay network, in which the distance d(P,Q) reflects the actual latency between P and Q.

Solution

P needs to solve three equations in two unknowns (x_P, y_P):

$$d_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2}$$

Select the leader…

ELECTION ALGORITHMS

Election algorithms

- Many distributed algorithms require that some process acts as a coordinator.
- In many systems, the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions ➔ single point of failure.
- (To avoid single point of failure) we need to select this special process **dynamically** (how?)

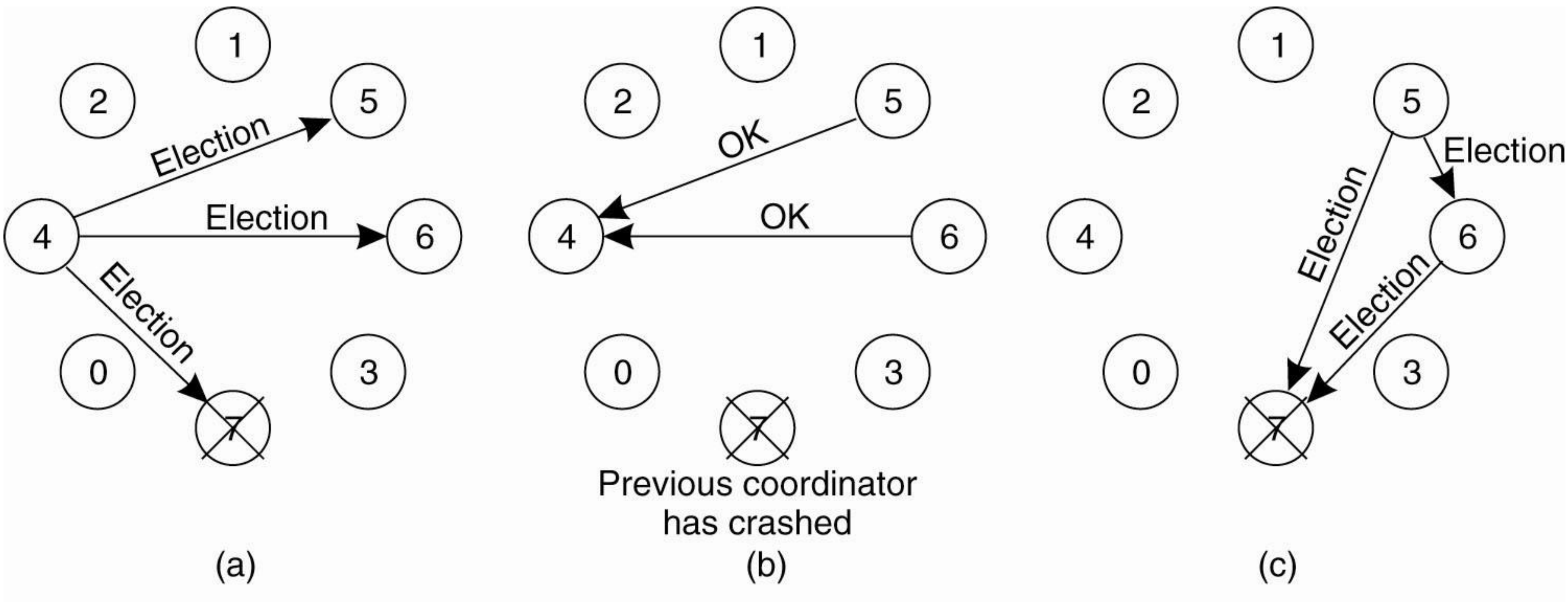
Then

- Where is the line between centralized or distributed solution?
- Is a fully distributed solution, i.e., one without a coordinator, always more robust than any centralized/coordinated solution?

Election by bullying

- Each process has an associated priority (weight). The process with the highest priority should always be elected as the coordinator.
- 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)

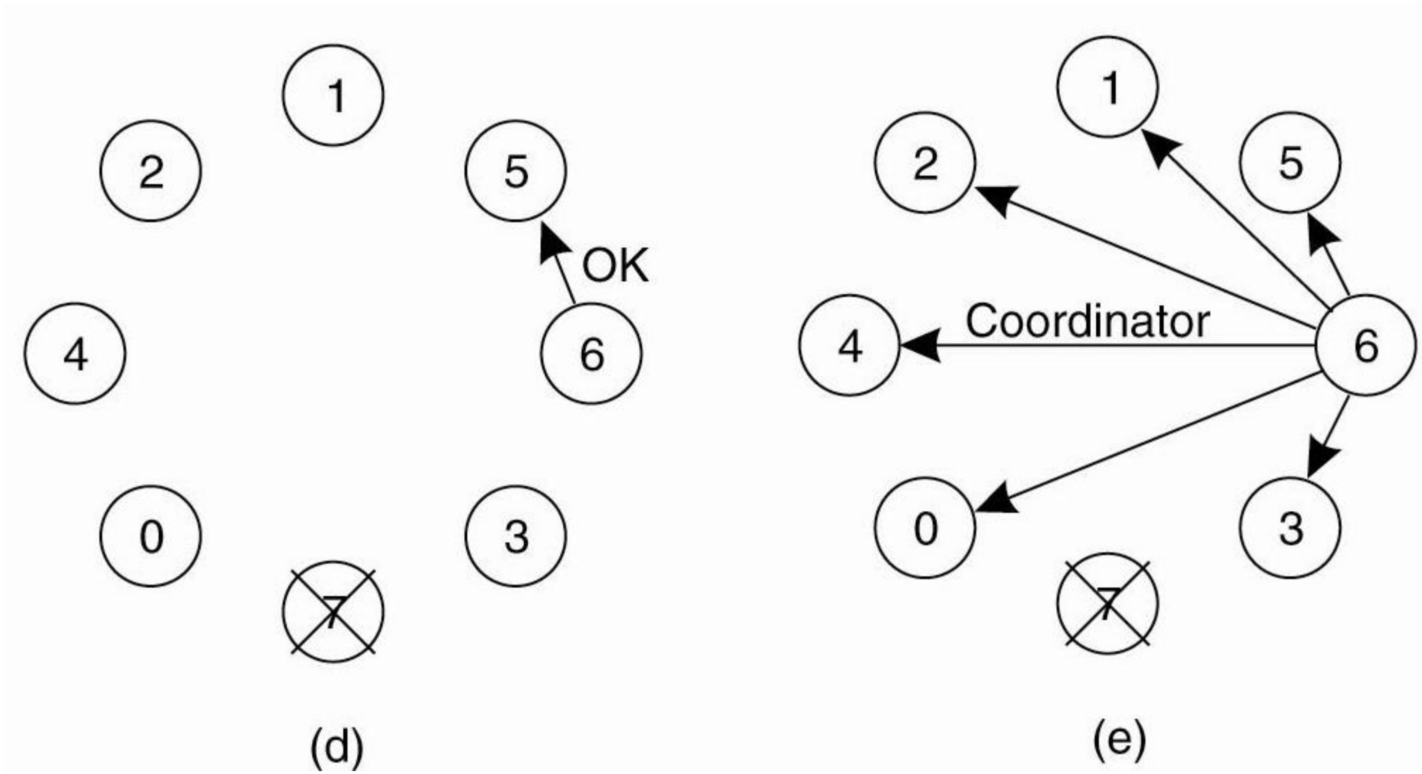


(a) Process 4 holds an election.

(b) Processes 5 and 6 respond, telling 4 to stop.

(c) Now 5 and 6 each hold an election.

The Bully Algorithm (2)



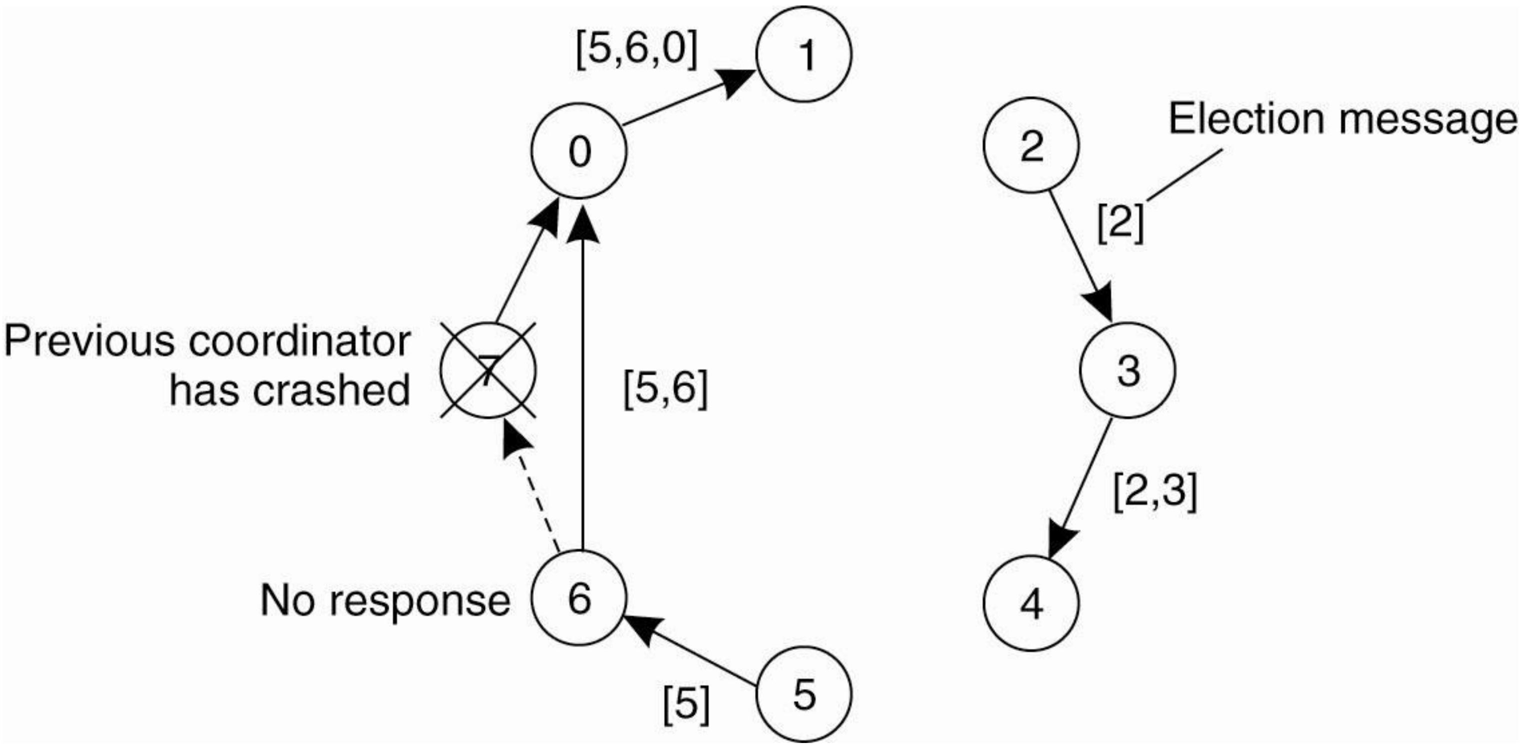
(d) Process 6 tells 5 to stop.

(e) Process 6 wins and tells everyone.

A Ring Algorithm

- 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 Example



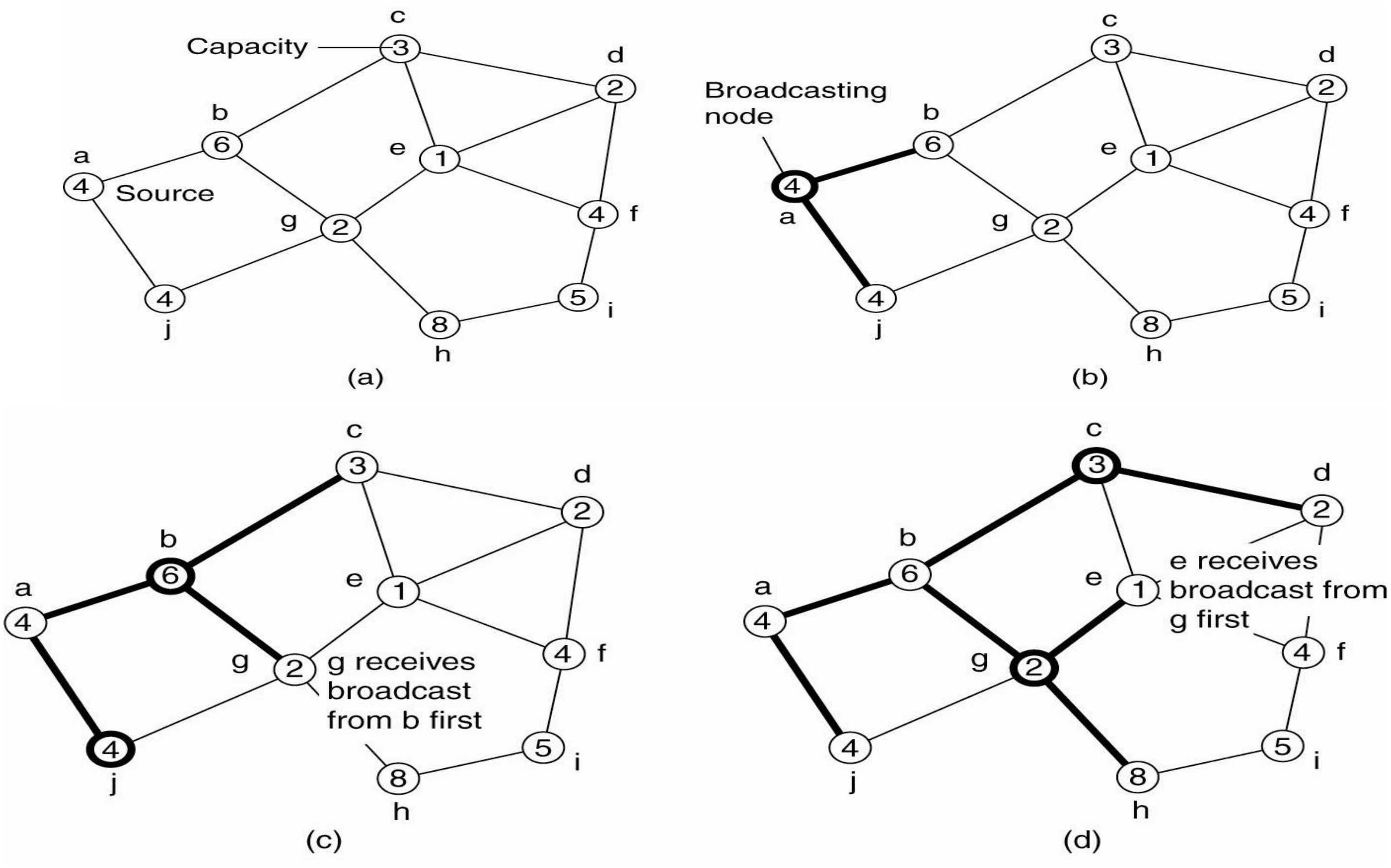
- Does it matter if two processes initiate an election?
- What happens if a process crashes during the election?

EXTRAS

Elections in Wireless Environments

- Traditional Election algorithms assumed that
  - Msg passing is reliable
  - Topology does not change often
- But these are not realistic in wireless environments
  - Mobile and ad hoc
  - Handle failures and partition/join

Elections in Wireless Environments Ex



(a) Initial network. (b)–(e) The build-tree phase..

Elections in Large-Scale Systems (1)

Requirements for superpeer selection:

1. Normal nodes should have low-latency access to superpeers.
2. Superpeers should be evenly distributed across the overlay network.
3. There should be a predefined portion of superpeers relative to the total number of nodes in the overlay network.
4. Each superpeer should not need to serve more than a fixed number of normal nodes.

Elections in Large-Scale Systems (2)

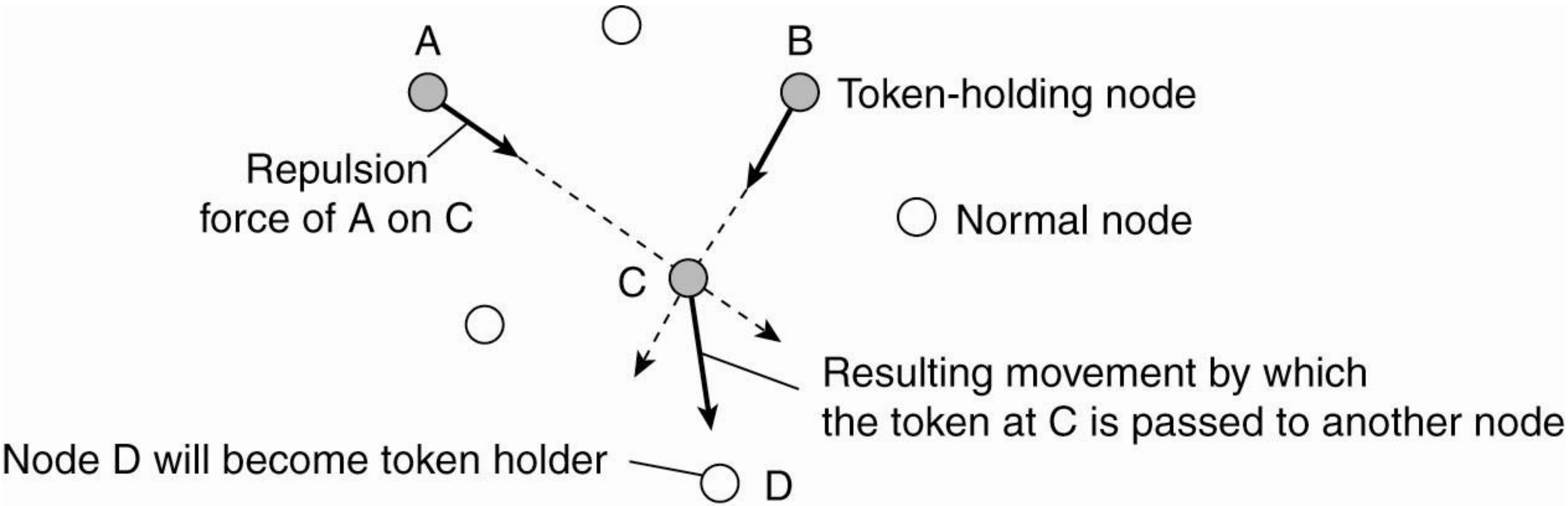
DHTs

- Reserve a fixed part of the ID space for superpeers.
- Example: if S superpeers are needed for a system that uses m-bit identifiers, simply reserve the $k = \lceil \log_2 S \rceil$ leftmost bits for superpeers. With N nodes, we'll have, on average, $2^{k-m}N$ superpeers.

Routing to superpeer

- Send message for key p to node responsible for
- p AND $11\cdots 1100\cdots 00$

Elections in Large-Scale Systems (3)



Moving tokens in a two-dimensional space using repulsion forces.

Summary

- Physical clock/time in distributed systems
- Logical clock/time and ‘Happen Before’ Relation
- Vector clocks
- Distributed synchronizations
- Election algorithms