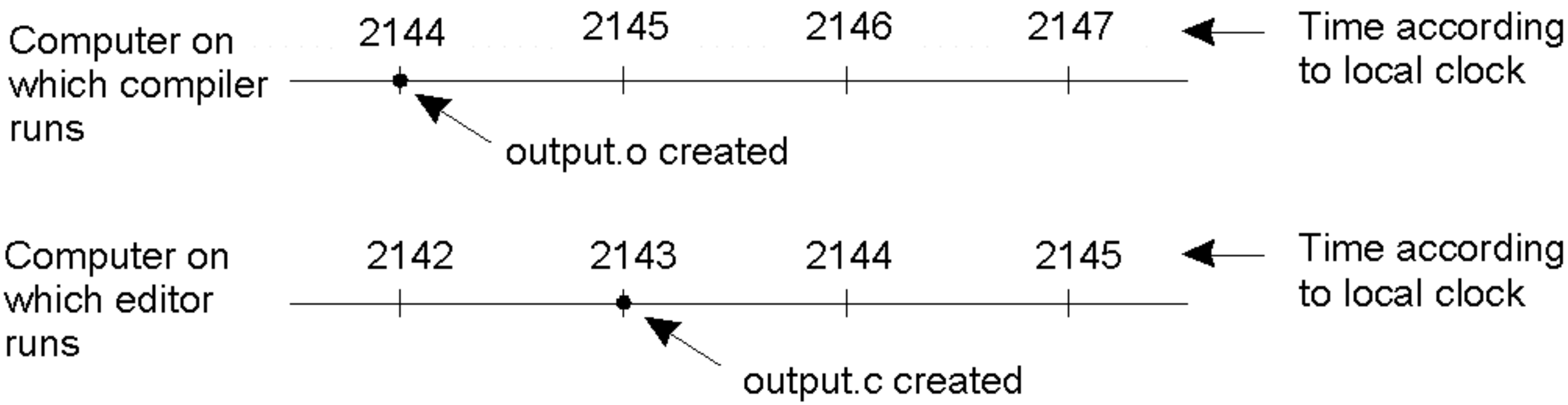
ECE151 – Lecture 7

Chapter 5

Synchronization

Clock Synchronization

Physical clocks
Logical clocks
Vector clocks



When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Physical Clocks

Problem: Sometimes we simply need the exact time, not just an ordering.

Solution: 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.

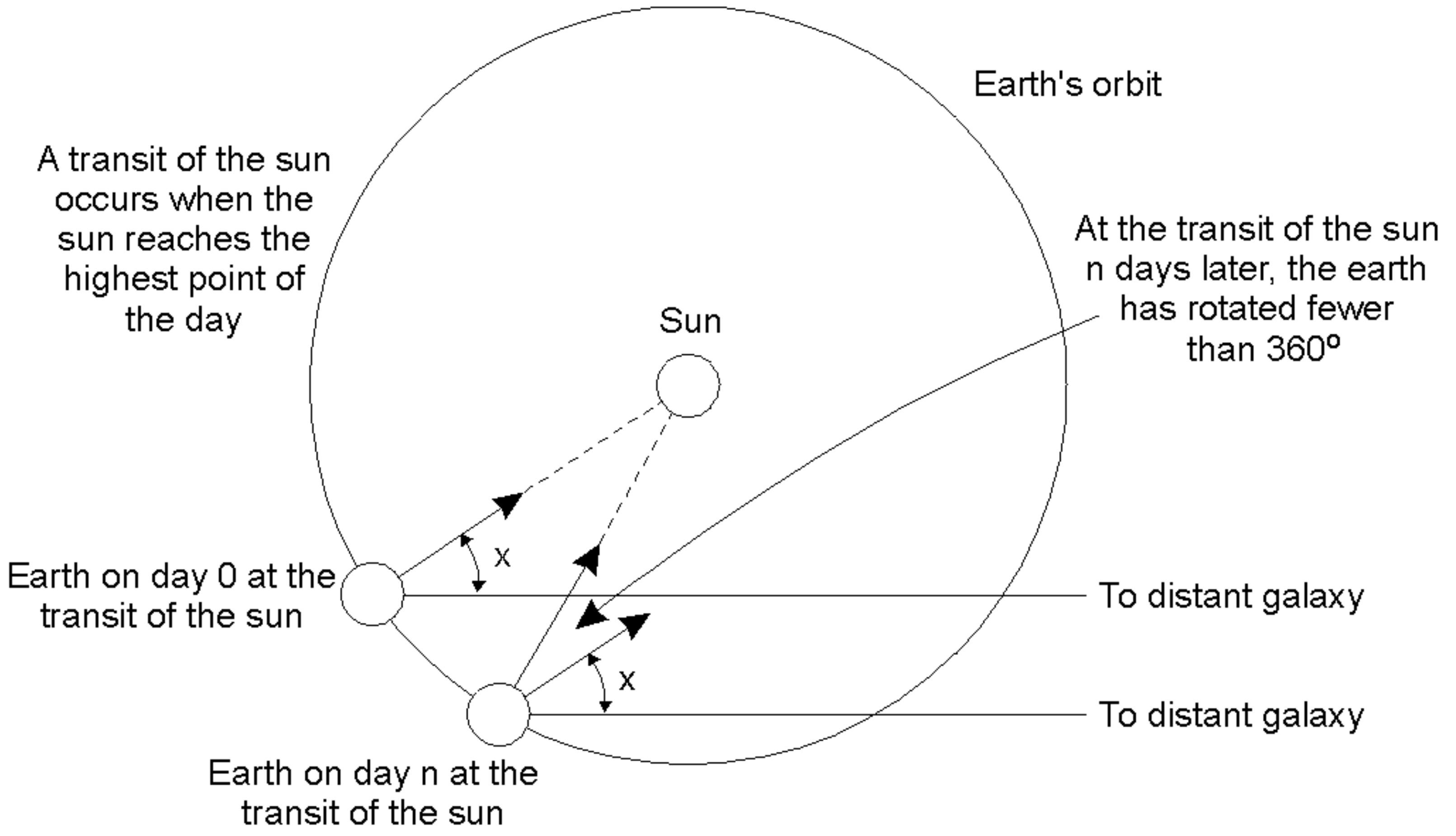
Introduces a leap second from time to time to compensate that days are getting longer.

UTC is **broadcast** through short wave radio and satellite.

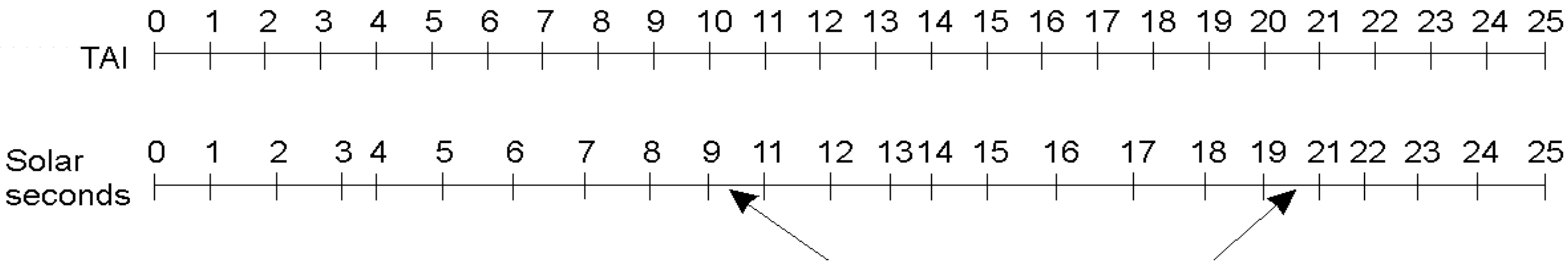
Satellites can give an accuracy of about 0 5 ms.

Question: Does this solve all our problems? Don’t we now have some global timing mechanism?

Physical Clocks (1)



Physical Clocks (2)



Leap seconds introduced into UTC to
get it in synch with TAI

TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

Physical Clocks

Problem: Suppose we have a distributed system with a UTC-receiver somewhere in it we still 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 $Cp(t)$, where t is UTC time.

Ideally, we have that for each machine p, $Cp(t) = t$, or, in other words, $dC/dt = 1$

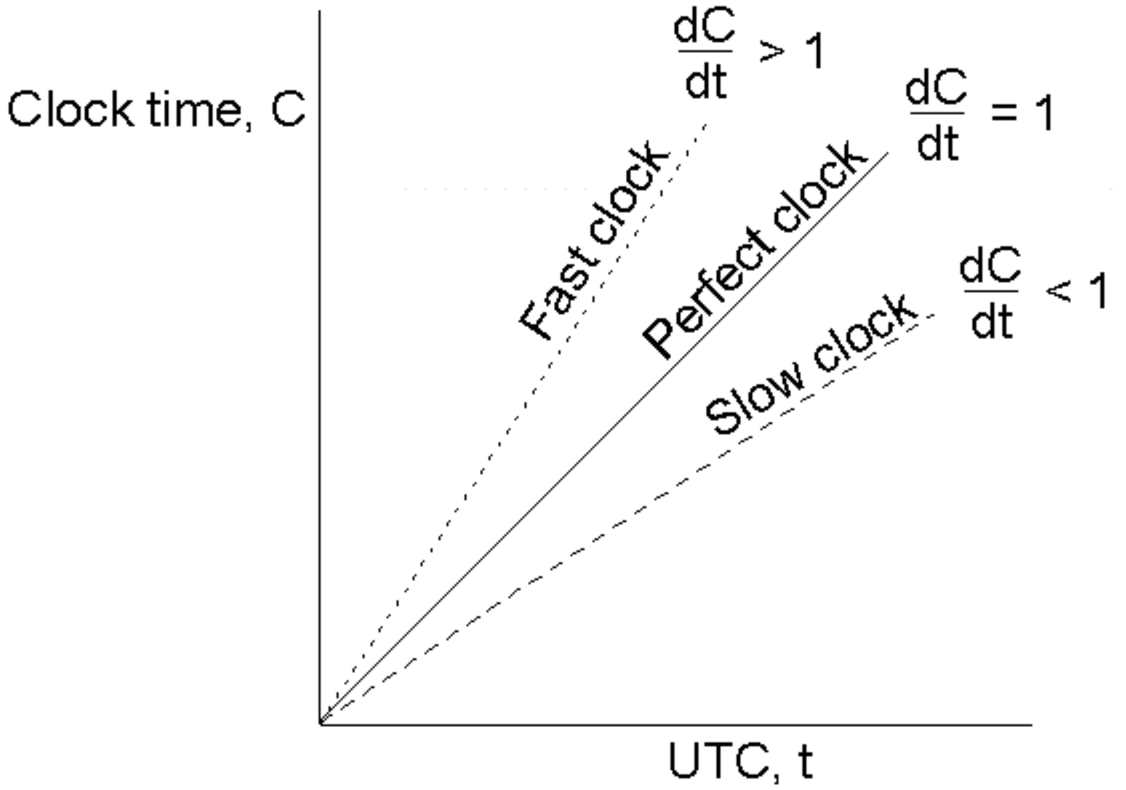
Clock Synchronization Algorithms

The relation between clock time and UTC when clocks tick at different rates.

In practice:

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

Goal: Never let two c
any system differ b
than δ time units
synchronize at leas
/(2 ρ) seconds.



Clock Synchronization Principles

Principle I: Every machine asks a **time server** for the accurate time at least once every $\delta/(2\rho)$ seconds.

Okay, but you need an accurate measure of round trip delay, including interrupt handling and processing incoming messages.

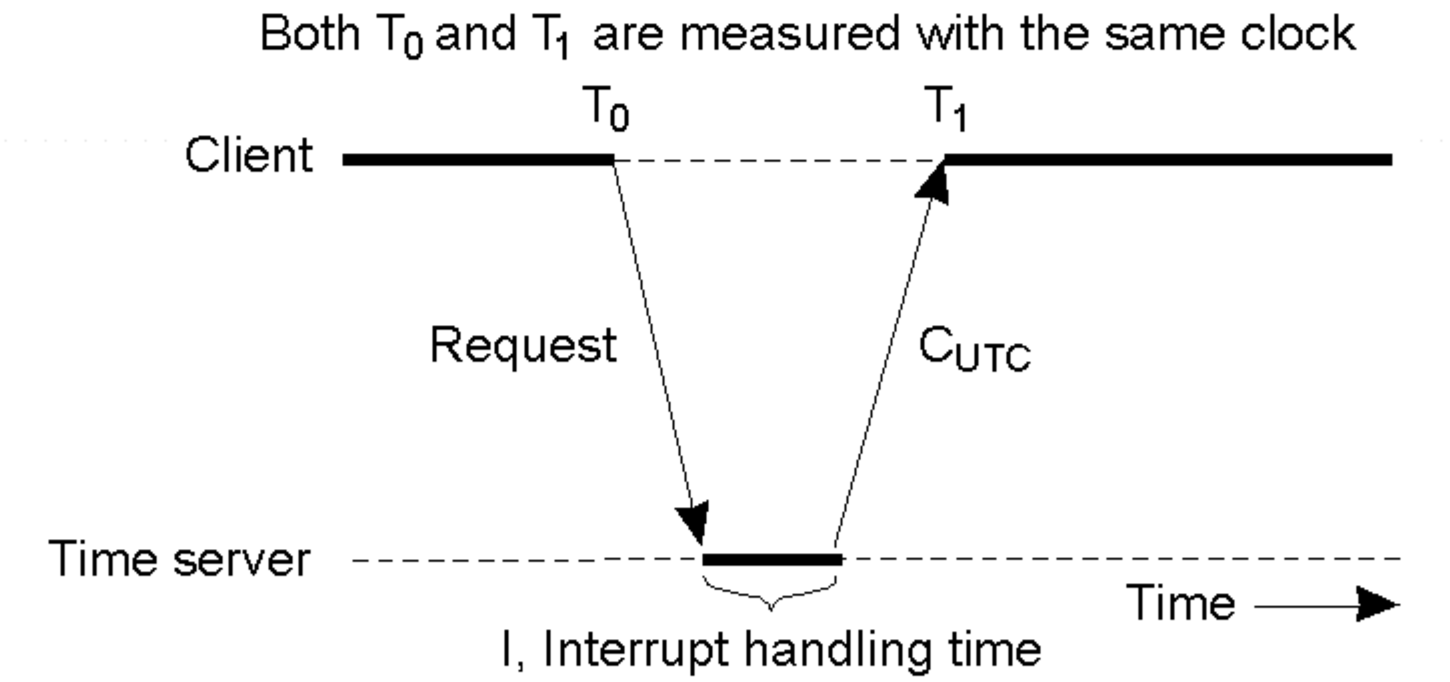
Principle II: Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

Okay, you'll probably get every machine in sync. Yo don't even need to propagate UTC time (why not?)

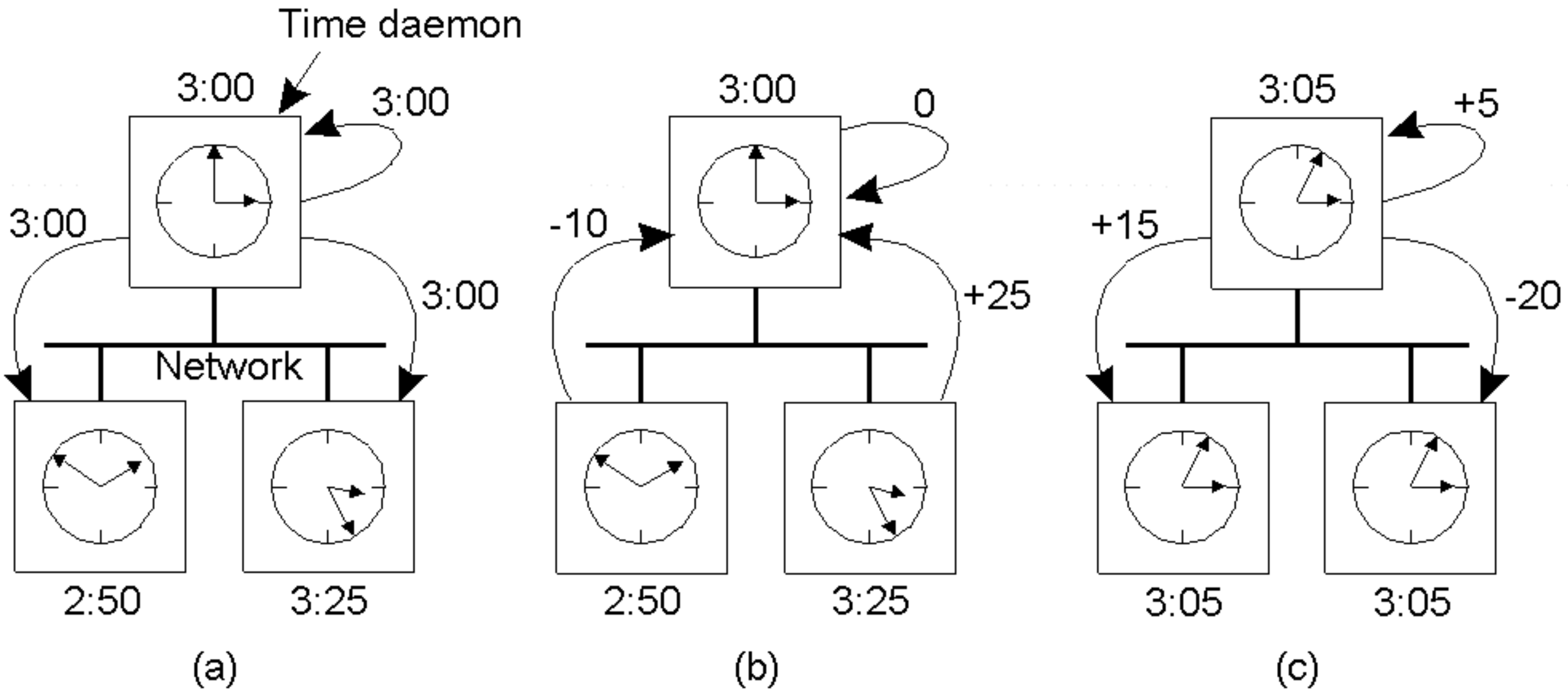
Fundamental problem: You'll have to ensure that setting time back is **never** allowed (smooth adjustments)

Cristian's Algorithm

Getting the current time from a time server.



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

The Happened-Before Relationship

Problem: We first need to introduce a notion of ordering before we can order anything.

The **happened-before** relation on the set of events in a distributed system is the smallest relation satisfying:

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 $a \rightarrow c$.

Note: this introduces a partial ordering of events in a system with concurrently operating processes.

Logical Clocks

Problem: How do we maintain a global view on the system’s behavior that is consistent with the happenedbefore relation?

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: If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.

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

Each process Pi maintains a **local** counter Ci and adjusts this counter according to the following rules:

1: For any two successive events that take place within Pi, Ci is incremented by 1.

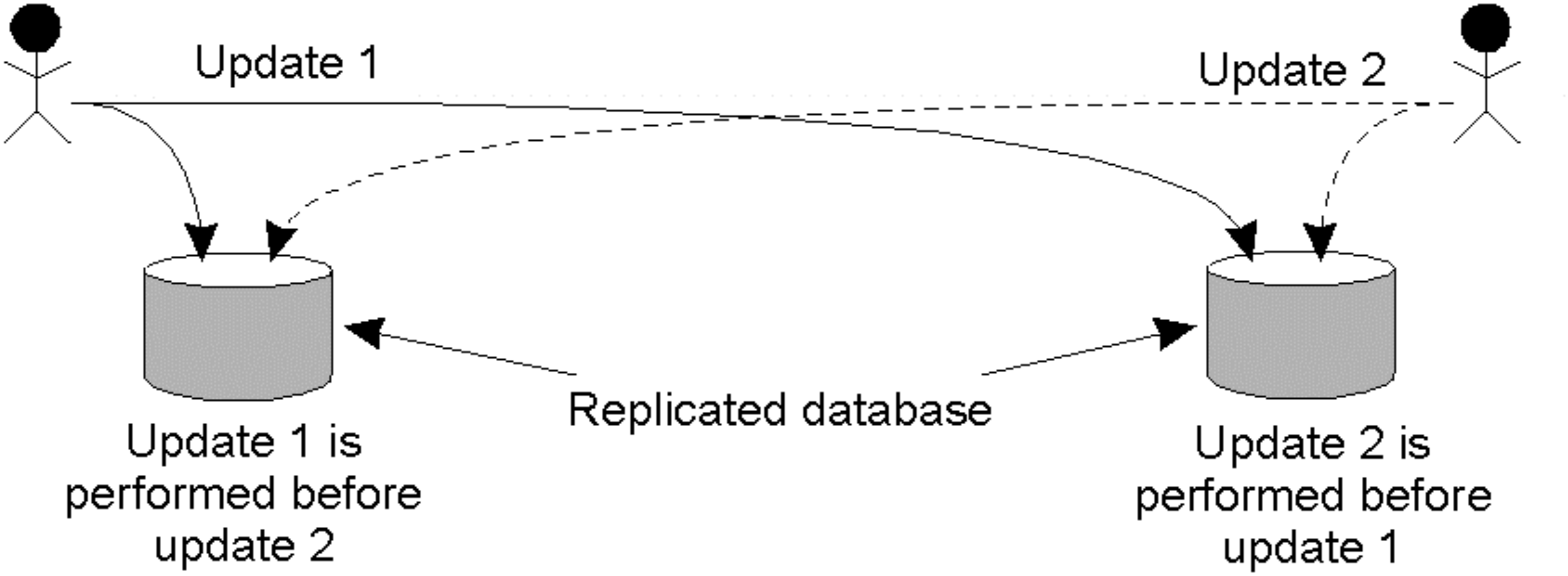
2: Each time a message m is sent by process Pi, the message receives a timestamp $Tm = Ci$.

3: Whenever a message m is received by a process Pj, Pj adjusts its local counter Cj:

$$Cj = \max(Cj + 1, Tm + 1)$$

Property **P1** is satisfied by (1); Property **P2** by (2) and (3).

Lamport Timestamps



a) Three processes, each with its own clock. The clocks run at different rates.

b) Lamport's algorithm corrects the clocks.

Total Ordering with Logical Clocks

Problem: it can still occur that two events happen at the same time. Avoid this by attaching a process number to an event:

Pi timestamps event *e* with *Ci(e)*

Then: *Ci(a)* before *Cj(b)* if and only if:

1: $Ci(a) < Cj(a)$ or

2: $Ci(a) < Cj(b)$ and $i = j$

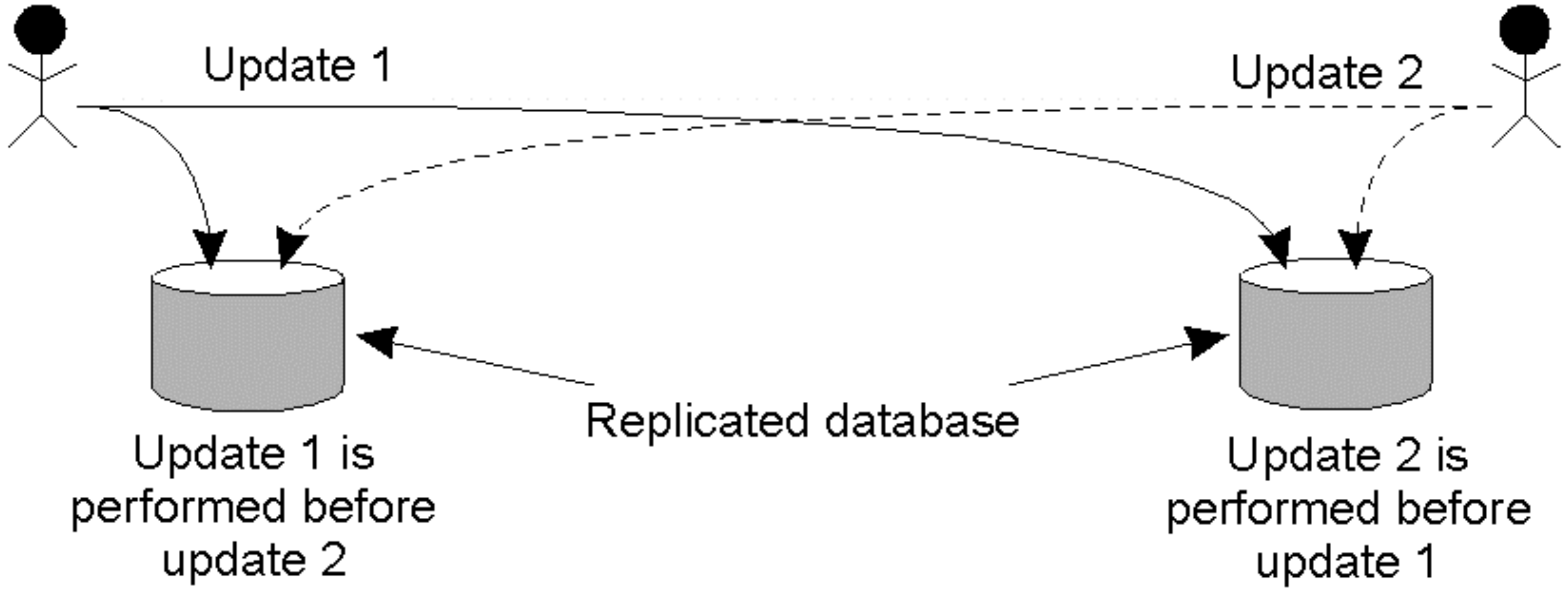
Example: Totally-Ordered Multicasting

Problem: We sometimes need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere:

Process *P*1 adds $100 to an account (initial value: $1000)

Process *P*2 increments account by 1%

There are two replicas



Outcome: in absence of proper synchronization, replica #1 will end up with $1111, while replica #2 ends up with $1110.

Example: Totally-Ordered Multicast

Process *Pi* sends timestamped message *msgi* to all others. The message itself is put in a local queue *queuei*.

Any incoming message at *Pj* is queued in *queuej*, according to its timestamp.

Pj passes a message *msgi* to its application if:

(1) *msgi* is at the head of *queuej*

(2) for each process *Pk*, there is a message *msgk* in *queuej* with a larger timestamp.

Note: We are assuming that communication is reliable and FIFO ordered.

Extension to Multicasting: Vector Timestamps

Observation: Lamport timestamps do not guarantee that if $C(a) < C(b)$ that a indeed happened before b.

We need **vector timestamps** for that.

Each process Pi has an array $Vi[1..n]$, where $Vi[j]$ denotes the number of events that process Pi knows have taken place at process Pj.

When Pi sends a message m, it adds 1 to $Vi[I]$, and sends Vi along with m as **vector timestamp** $vt(m)$. Result: upon arrival, each other process knows Pi's timestamp.

Question: What does $Vi[j] = k$ mean in terms of messages sent and received?

Extension to Multicasting: Vector Timestamps

When a process Pj receives a message m from Pi with vector timestamp $vt(m)$, it
(1) updates each $Vj[k]$ to $\max(Vj[k], V(m)[k])$, and
(2) increments Vj j by 1. **NOTE:** Book is wrong!

To support causal delivery of messages, assume you increment your own component only when sending a message. Then, Pj postpones delivery of m until:

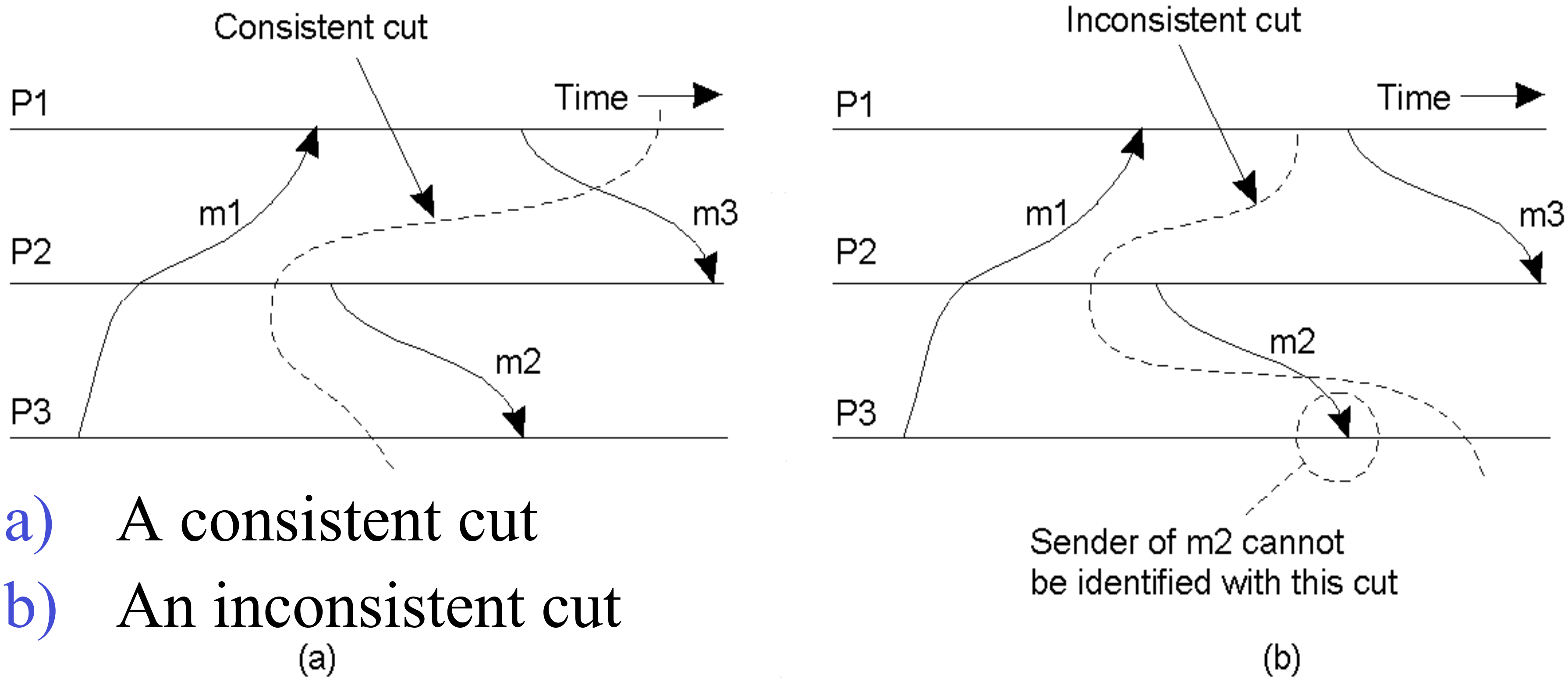
- $vt(m)[i] = Vj[i] + 1$.
- $vt(m)[k]$ =< $Vj[k]$ for k not= i.

Example: Take $V3 = [0, 2, 2]$, $vt(m) = [1, 3, 0]$.
What information does $P3$ have, and what will it do when receiving m (from $P1$)?

Global State

Basic Idea: Sometimes you want to collect the current state of a distributed computation, called a **distributed snapshot**.
It consists of all local states and messages in transit.

Important: A distributed snapshot should reflect a **consistent** state:



Sender of m2 cannot be identified with this cut

(a)

(b)

a) A consistent cut
b) An inconsistent cut

Global State

Any process *P* can initiate taking a distributed snapshot

P starts by recording its own local state
P sends a marker along each of its outgoing channels

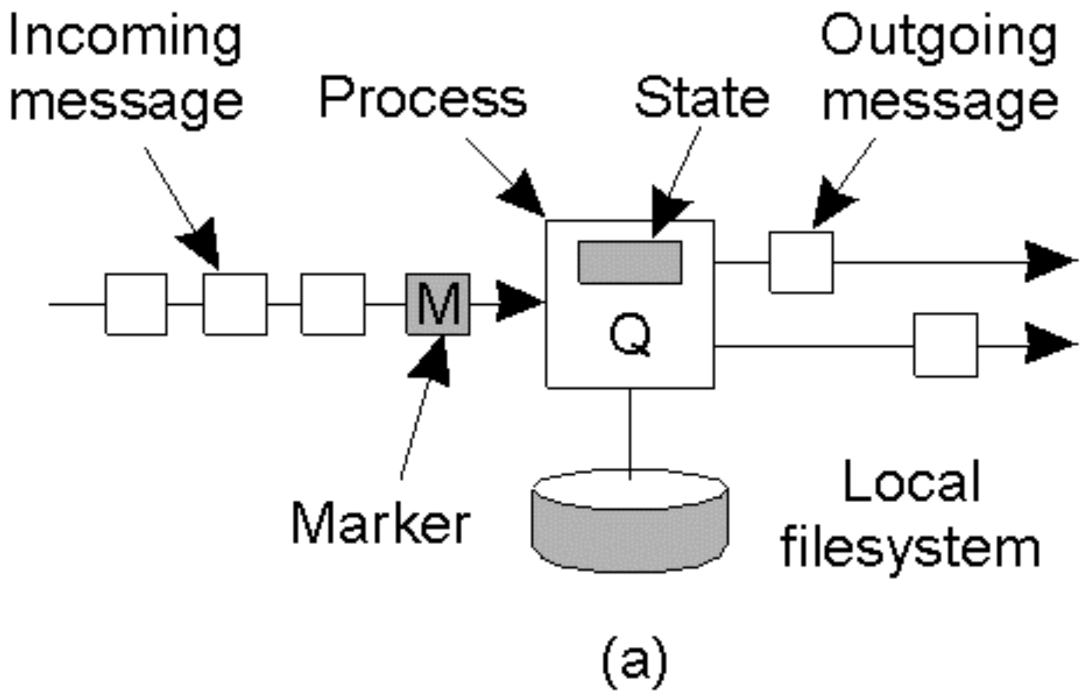
When *Q* receives a marker through channel *C*, its action depends on whether it had already recorded its local state:

- Not yet recorded: it records its local state, and sends the marker along each of its outgoing channels
- Already recorded: the marker on *C* indicates that the channel's state should be recorded:

  All messages received before this marker and
  after *Q* recorded its own state.

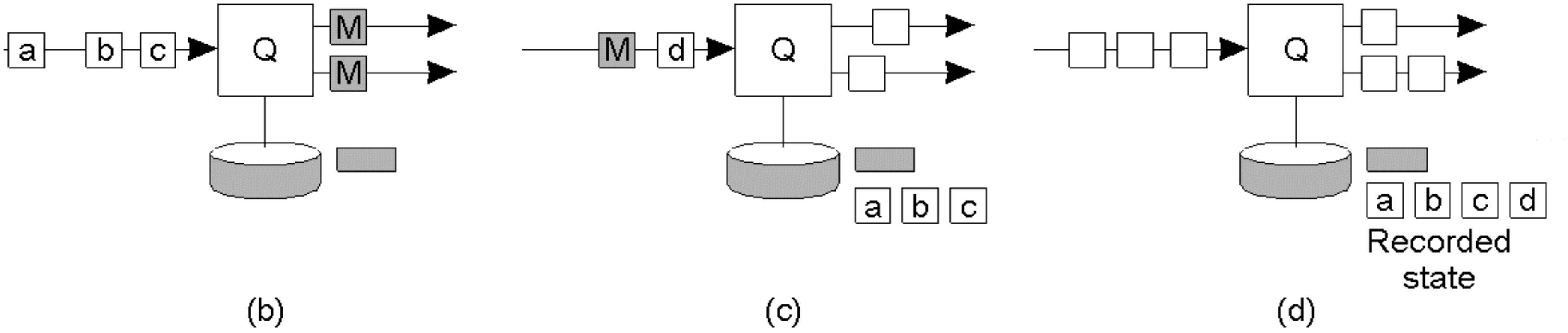
Q is finished when it has received a marker along each of its incoming channels

Global State



a) Organization of a process and channels for a distributed snapshot

Global State



b) Process Q receives a marker for the first time and records its local state

c) Q records all incoming message

d) *Q* receives a marker for its incoming channel and finishes recording the state of the incoming channel

Election Algorithms

Principle: An algorithm requires that some process acts as a coordinator. The question is how to select this special process **dynamically**.

Note: In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions with a single point of failure.

Question: If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?

Question: Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/ coordinated solution?

Election by Bullying

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

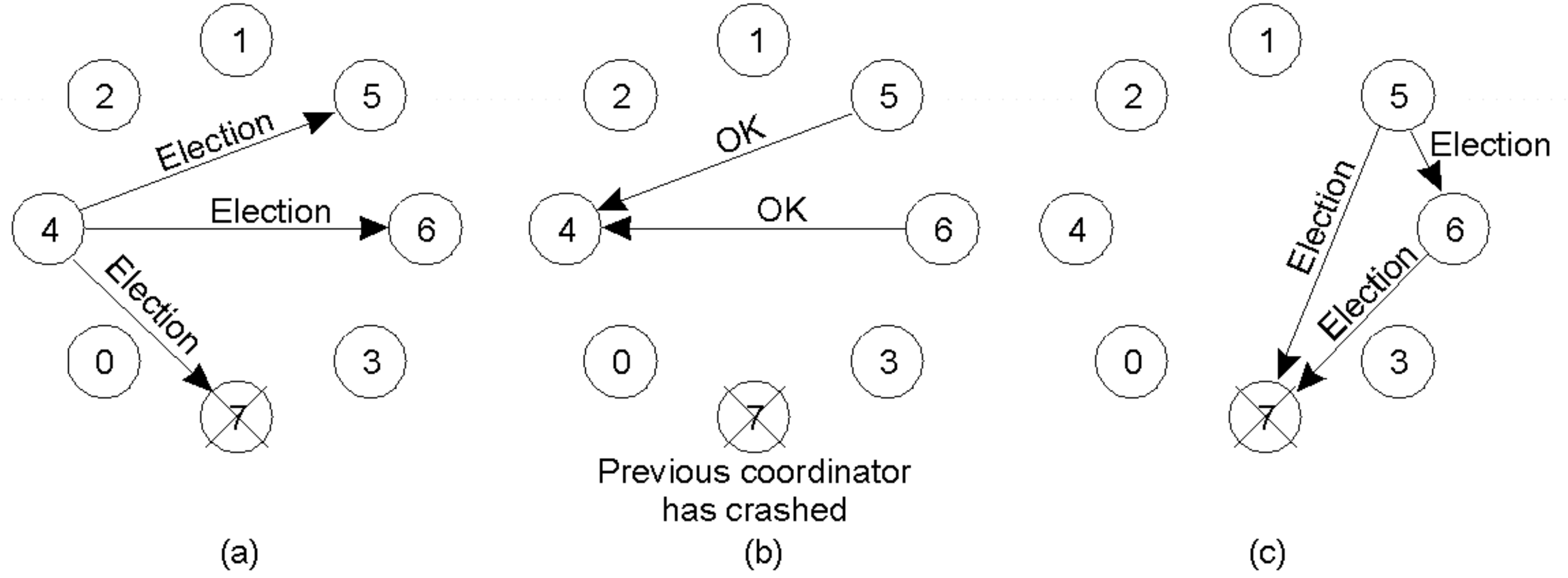
Issue: How do we find the heaviest process?

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



The bully election algorithm

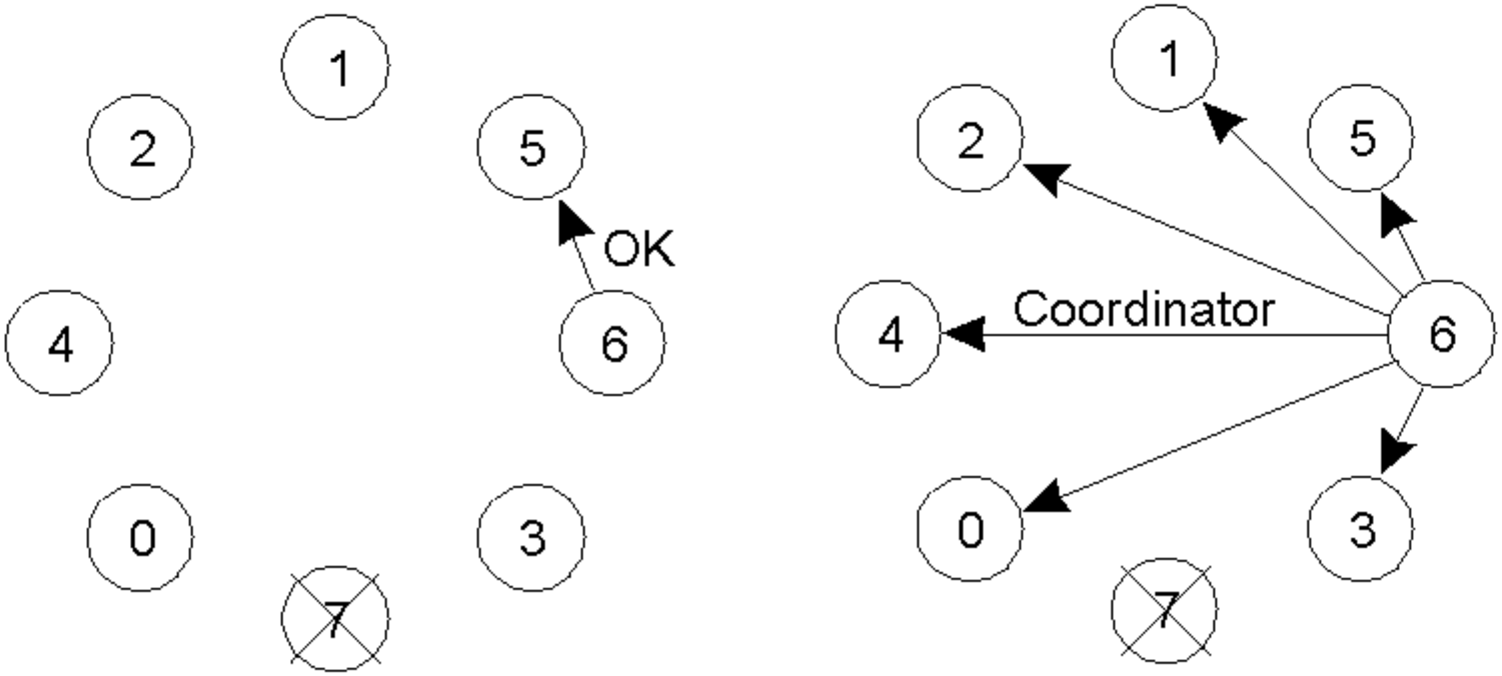
a) Process 4 holds an election

b) Process 5 and 6 respond, telling 4 to stop

c) Now 5 and 6 each hold an election

Global State

d) Process 6 tells 5 to stop
e) Process 6 wins and tells everyone



(d) (e)

Election in a Ring

Principle: 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.

Does it matter if two processes initiate an election?

What happens if a process crashes *during* the election?

A Ring Algorithm