Distributed Systems

Synchronization

Chapter 6

Synchronization

- Synchronization and coordination are two closely related phenomena. In **process synchronization** we make sure that one process waits for another to complete its operation. When dealing with **data synchronization**, the problem is to ensure that two sets of data are the same. When it comes to **coordination**, the goal is to manage the interactions and dependencies between activities in a distributed system.
- coordination in distributed systems is often much more difficult compared to that in uniprocessor or multiprocessor systems

Clock Synchronization

- Synchronization based on “Actual Time”.
- Note: time is really easy on a uniprocessor system.
- Achieving agreement on time in a DS is not trivial(importance).
- **Question**: is it even possible to synchronize all the clocks in a Distributed System?
- With multiple computers, “clock skew” ensures that no two machines have the same value for the “current time”. But, how do we measure time?

Logical Clocks

- Synchronization based on “relative time”.
- Note that (with this mechanism) there is no requirement for “relative time” to have any relation to the “real time”.
- What’s important is that the processes in the Distributed System *agree on the ordering in which certain events occur*.
- Such “clocks” are referred to as *Logical Clocks*.

Lamport's Logical Clocks

- **First point**: if two processes do not interact, then their clocks do not need to be synchronized – they can operate *concurrently* without fear of interfering with each other.
- **Second (critical) point**: it does not matter that two processes share a common notion of what the "real" current time is. What does matter is that the processes have some agreement on the order in which certain events occur.
- Lamport used these two observations to define the "happens-before" relation (also often referred to within the context of *Lamport's Timestamps*).

The “Happens-Before” Relation (1)

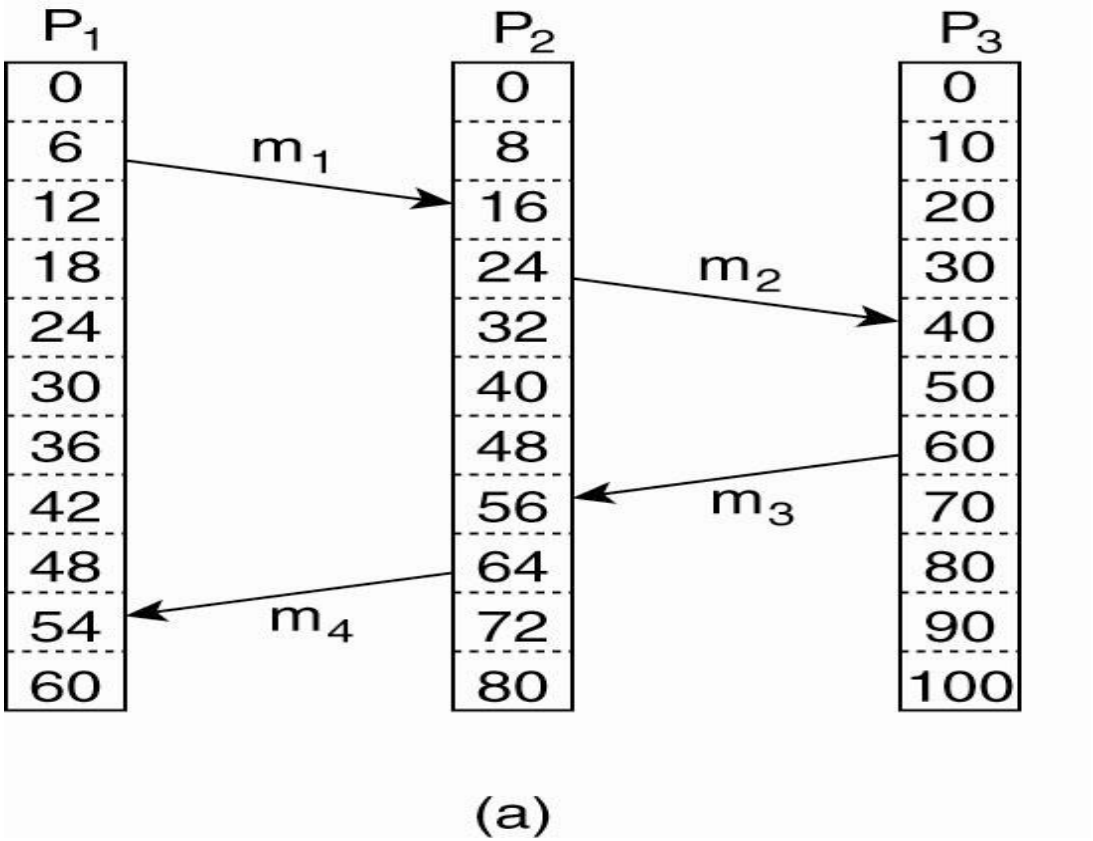
- If A and B are events in the same process, and A occurs before B, then we can state that:
- *A “happens-before” B is true*.
- Equally, if A is the event of a *message being sent by one process*, and B is the event of the same *message being received by another process*, then A “happens-before” B is also true.
- (Note that a message cannot be received before it is sent, since it takes a finite, nonzero amount of time to arrive … and, of course, time is not allowed to run backwards).

Lamport's Logical Clocks (1)

- The "happens-before" relation $\rightarrow$ can be observed directly in two situations:

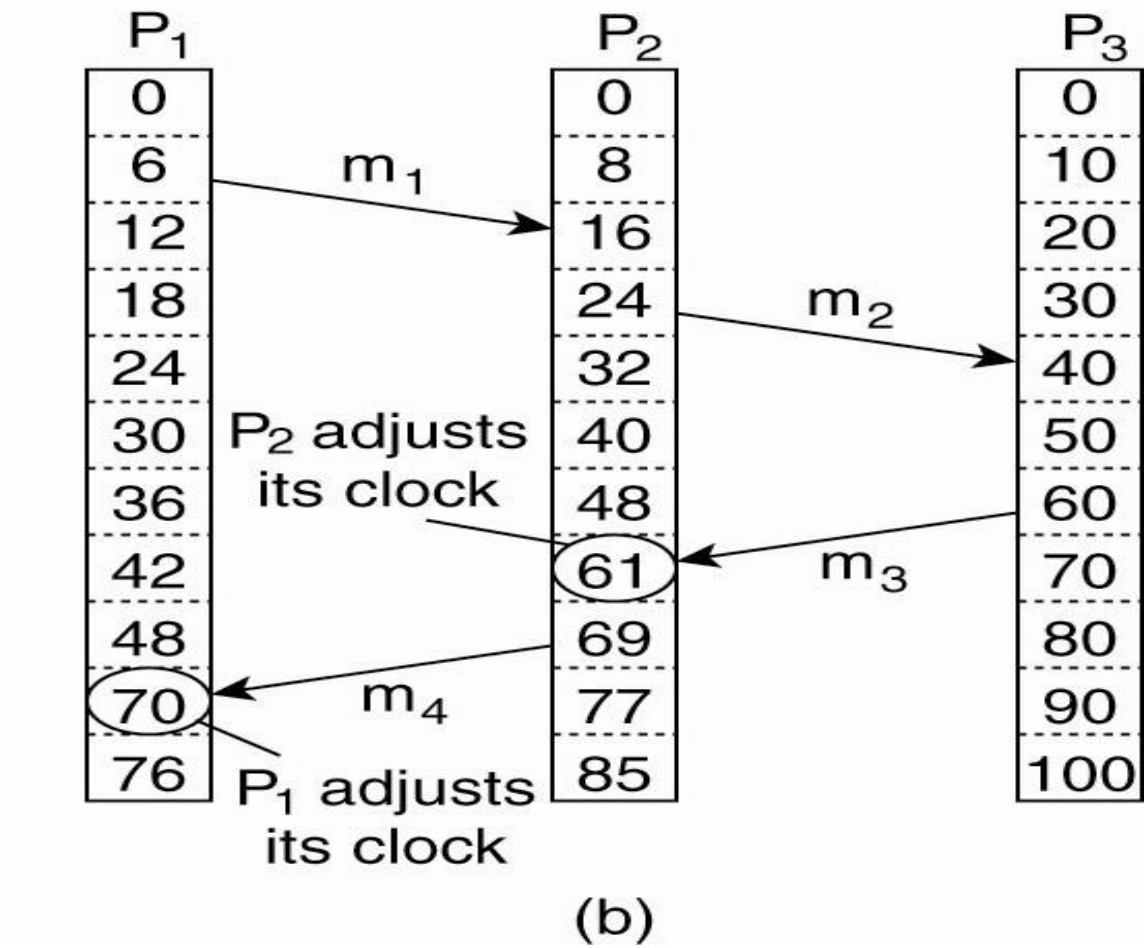
1. If *a* and *b* are events in the same process, and *a* occurs before *b*, then $a \rightarrow b$ is true.
2. If a is the event of a message being sent by one process, and *b* is the event of the message being received by another process, then $a \rightarrow b$.

Lamport's Logical Clocks (2)



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

Lamport's Logical Clocks (3)



(b) Lamport's algorithm corrects the clocks

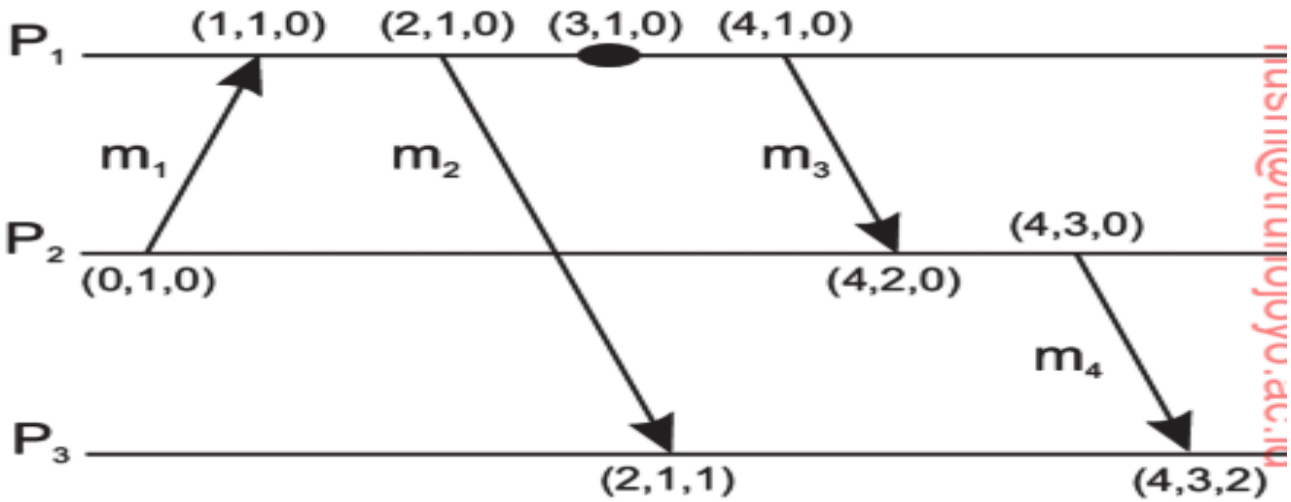
Vector clock

Vector Clock is an algorithm that generates partial ordering of events and detects causality violations in a distributed system. These clocks expand on Scalar time to facilitate a causally consistent view of the distributed system, they detect whether a contributed event has caused another event in the distributed system. It essentially captures all the causal relationships.

or N given processes, there will be vector/ array of size N.

- **How does the vector clock algorithm work :**
- Initially, all the clocks are set to zero.
- Every time, an Internal event occurs in a process, the value of the processes's logical clock in the vector is incremented by 1
- Also, every time a process sends a message, the value of the processes's logical clock in the vector is incremented by 1.

Vector clock



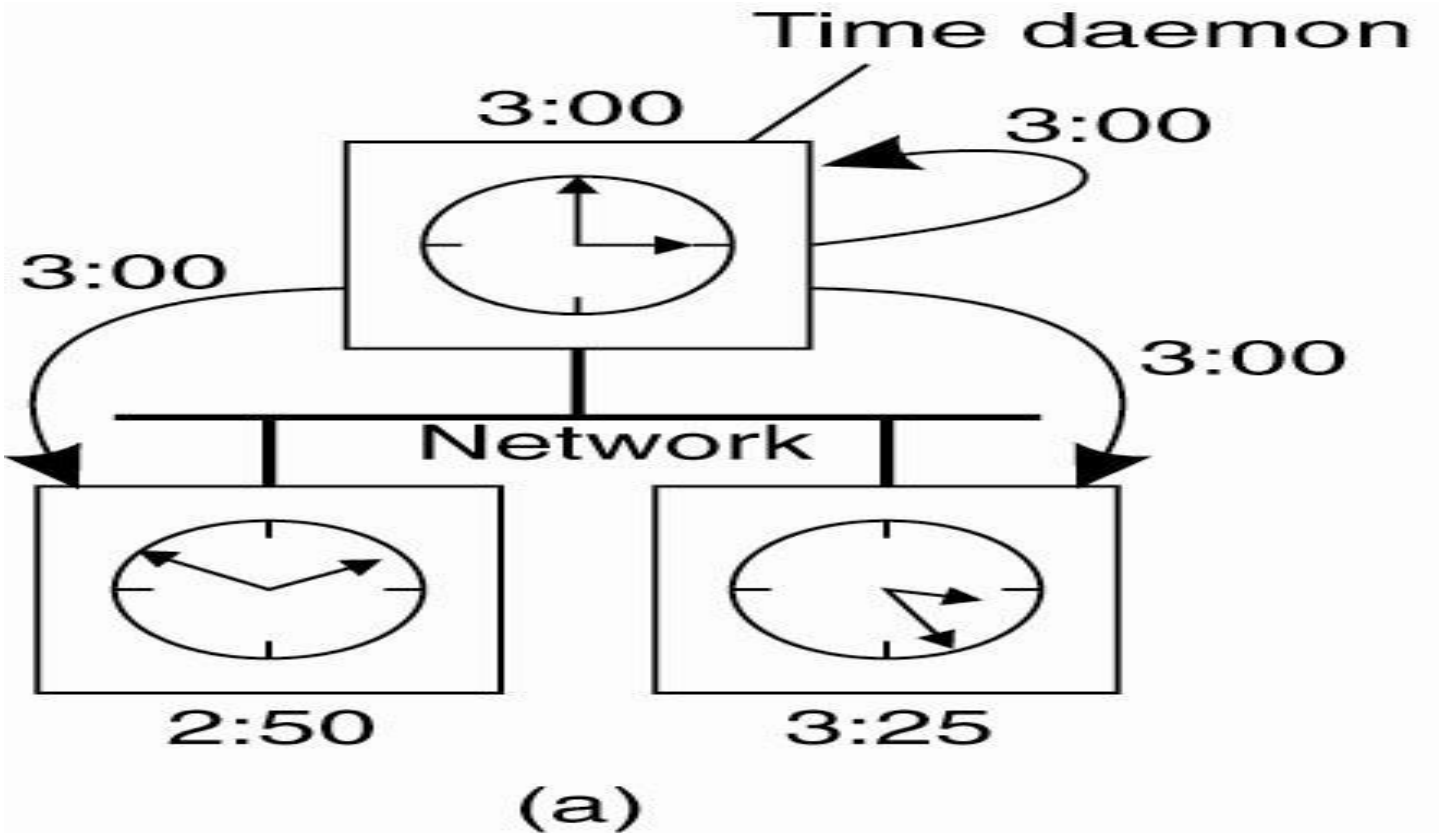
Physical Clock

- The physical clocks are used to adjust the time of nodes. Each node in the system can share its local time with other nodes in the system. The time is set based on UTC (Universal Time Coordination). UTC is used as a reference time clock for the nodes in the system.
- The basis for keeping global time is a called **Universal Coordinated Time**, but is abbreviated as **UTC**.
- UTC is the basis of all modern civil timekeeping and is a worldwide standard.
- The accuracy of these stations is about $\pm$ 1 msec, but due to random atmospheric fluctuations that can affect the length of the signal path, in practice the accuracy is no better than $\pm$ 10 msec.

Physical Clocks synchronization algorithm

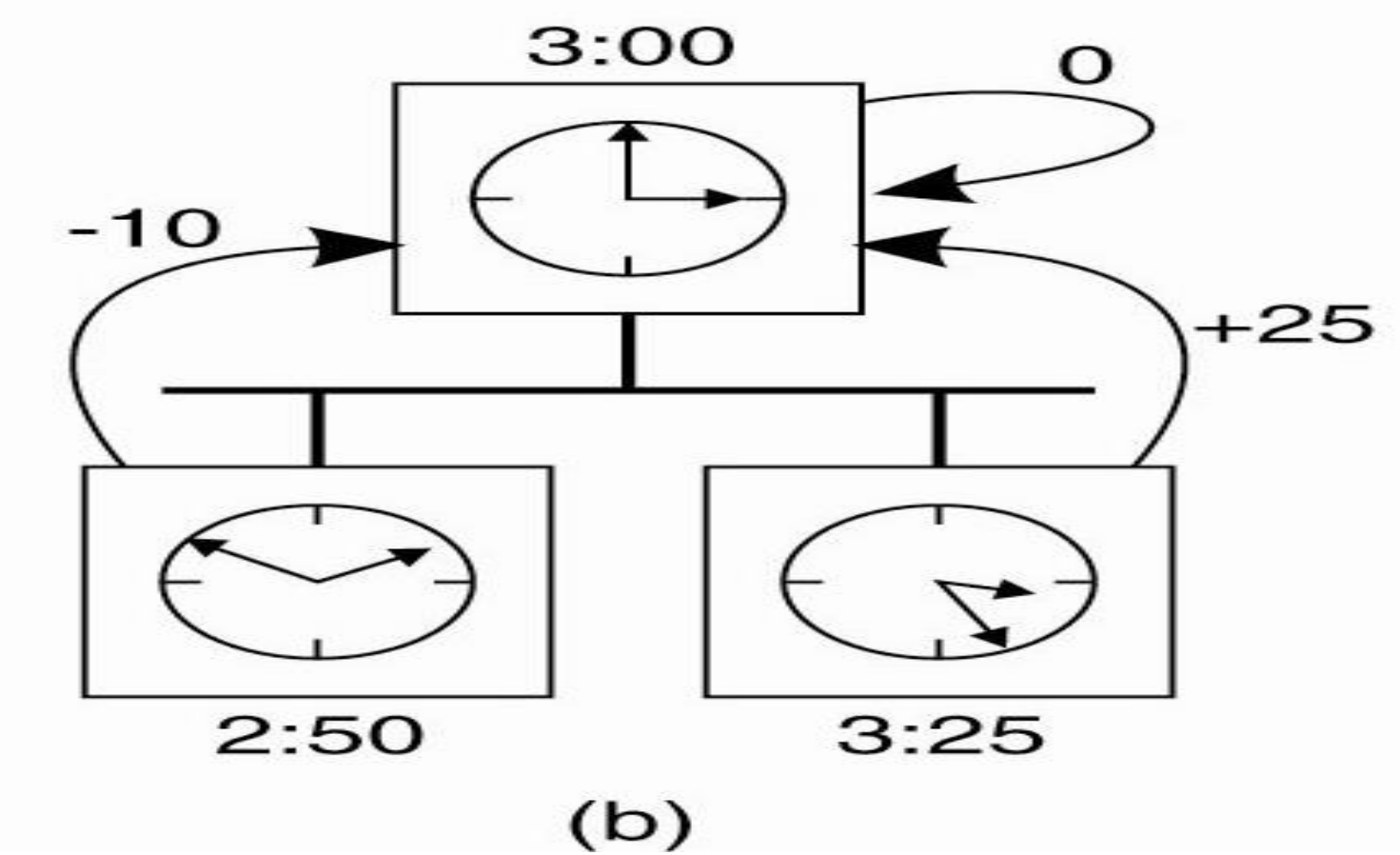
If one machine has a UTC receiver, the goal becomes keeping all the other machines synchronized to it. If no machines have UTC receivers, each machine keeps track of its own time, and the goal is to keep all the machines together as well as possible.

The Berkeley Algorithm (1)



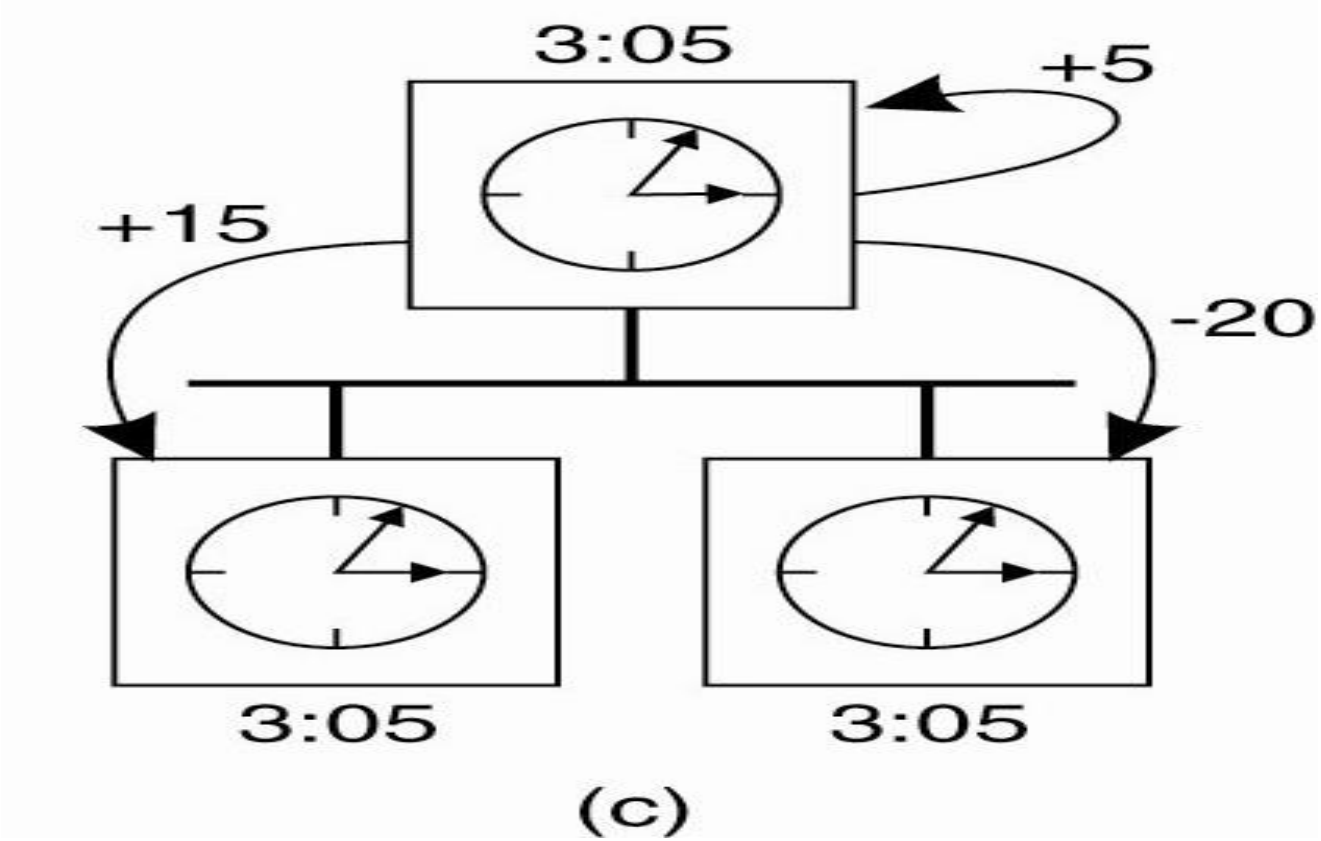
(a) The time daemon asks all the other machines for their clock values

The Berkeley Algorithm (2)



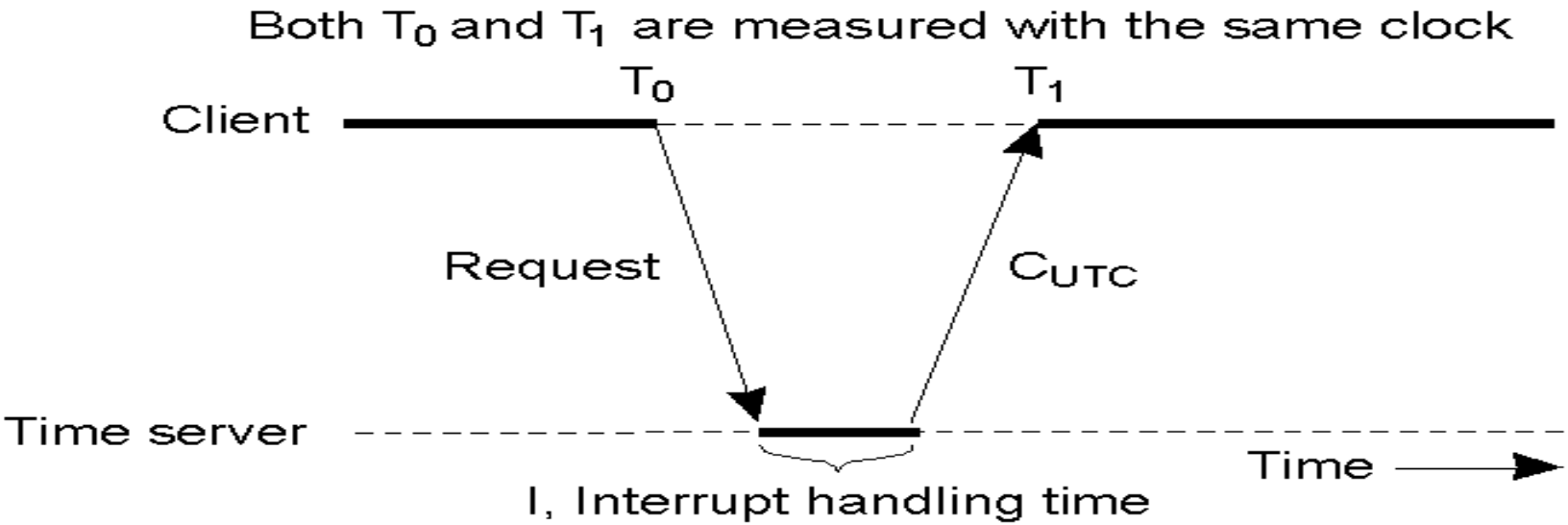
(b) The machines answer

The Berkeley Algorithm (3)

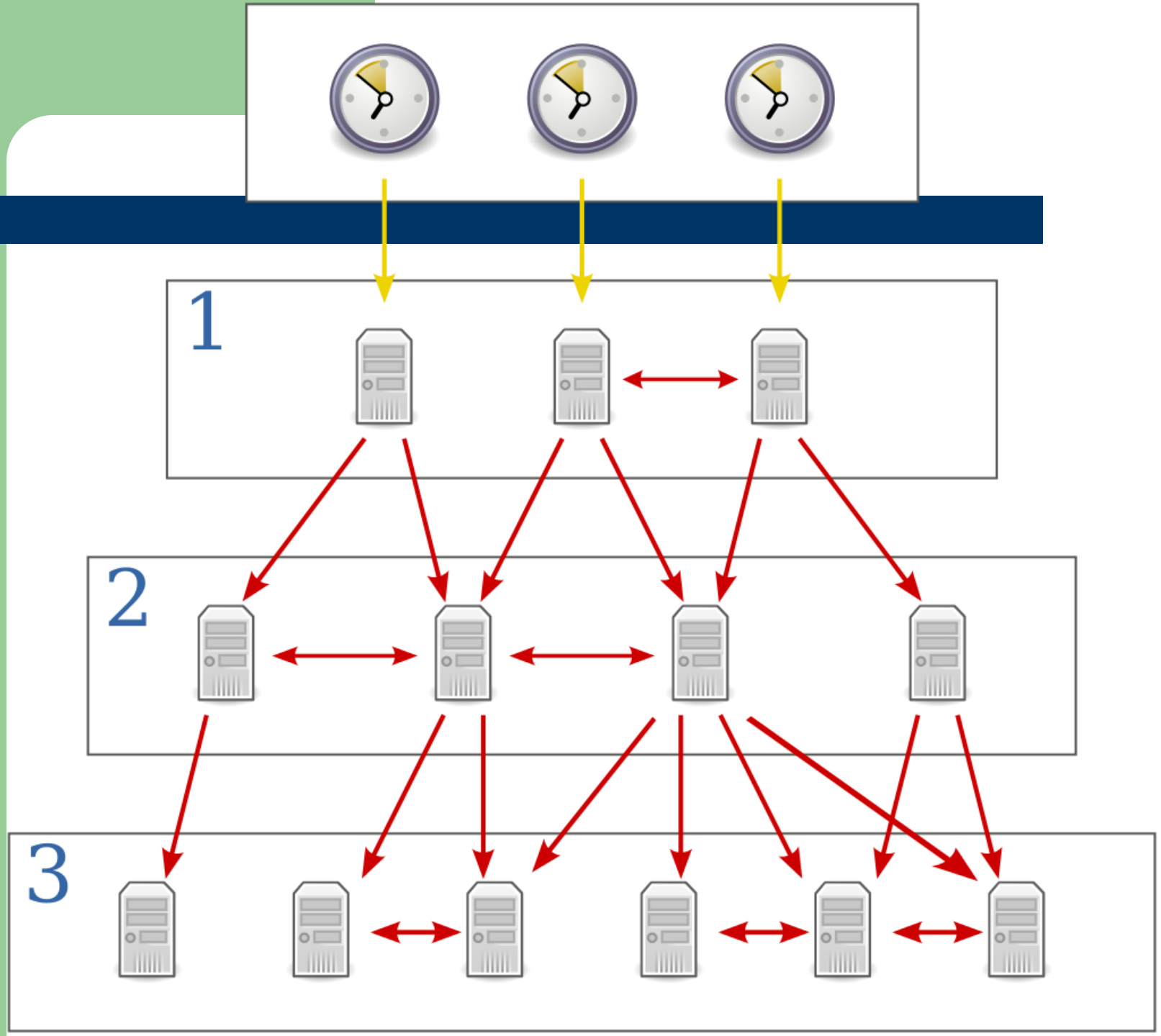


(c) The time daemon tells everyone how to adjust their clock

Clock Synchronization (Network Time Protocol) Cristian's Algorithm



- Getting the current time from a “time server”, using periodic client requests.
- Problem results from the delay introduced by the network request/response: latency.


1
2
3

- Degrees of separation from the UTC source are defined as strata.
- A reference clock -- which receives true time from a dedicated transmitter or satellite navigation system -- is categorized as stratum-0;
- A computer that is directly linked to the reference clock is stratum-1;
- A computer that receives its time from a stratum-1 computer is stratum-2, and so on.
- Accuracy is reduced with each additional degree of separation.

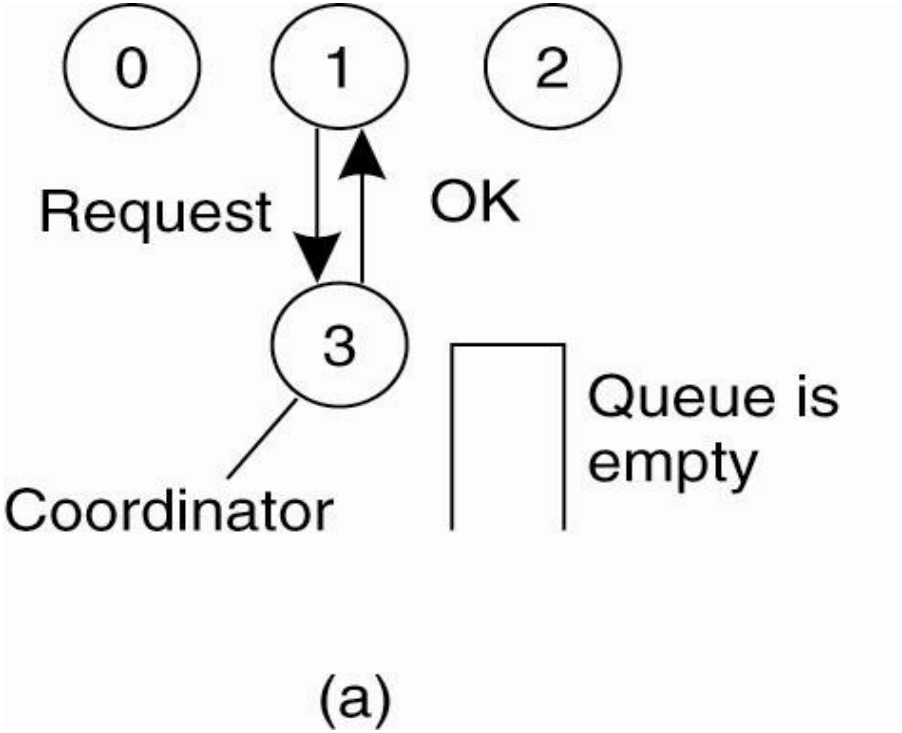
Mutual Exclusion within Distributed Systems

- It is often necessary to protect a *shared resource* within a Distributed System using "mutual exclusion" – for example, it might be necessary to ensure that no other process changes a shared resource while another process is working with it.
- In non-distributed, uniprocessor systems, we can implement "critical regions" using techniques such as semaphores, monitors and similar constructs – thus achieving *mutual exclusion*.
- These techniques have been adapted to Distributed Systems …

DS Mutual Exclusion: Techniques

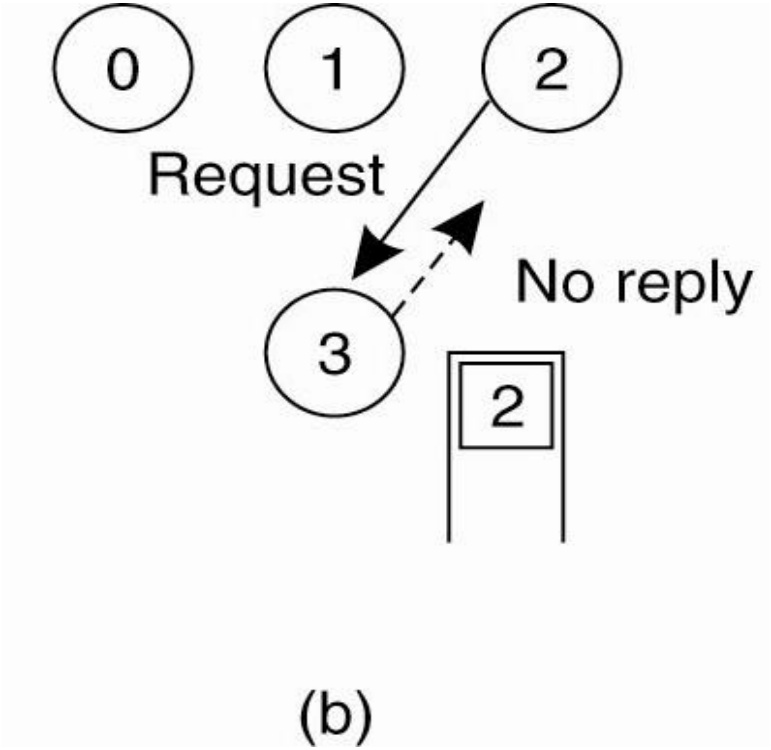
- **Centralized**: a single coordinator controls whether a process can enter a critical region.
- **Distributed**: the group *confers* to determine whether or not it is safe for a process to enter a critical region.

Mutual Exclusion
A Centralized Algorithm (1)



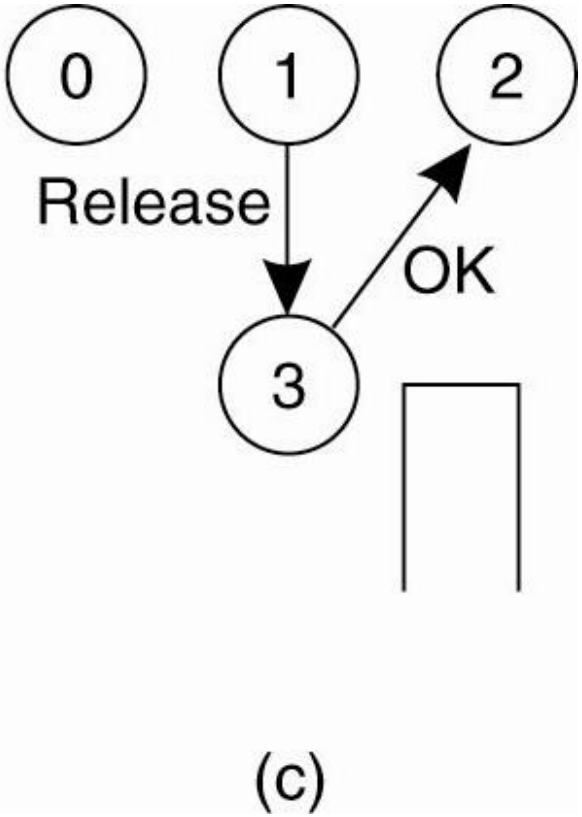
(a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.

Mutual Exclusion
A Centralized Algorithm (2)



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

Mutual Exclusion
A Centralized Algorithm (3)



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

Comments: The Centralized Algorithm

- ***Advantages***:
  - It works.
  - It is fair.
  - There's no process starvation.
  - Easy to implement.
- ***Disadvantages***:
  - There's a single point of failure!
  - The coordinator is a bottleneck on busy systems.

Distributed Mutual Exclusion

- Based on work by Ricart and Agrawala (1981).
- Requirement of their solution: *total ordering* of all events in the distributed system (which is achievable with Lamport's timestamps).
- Note that messages in their system contain three pieces of information:
  1. The critical region ID.
  2. The requesting process ID.
  3. The current time.

Mutual Exclusion: Distributed Algorithm

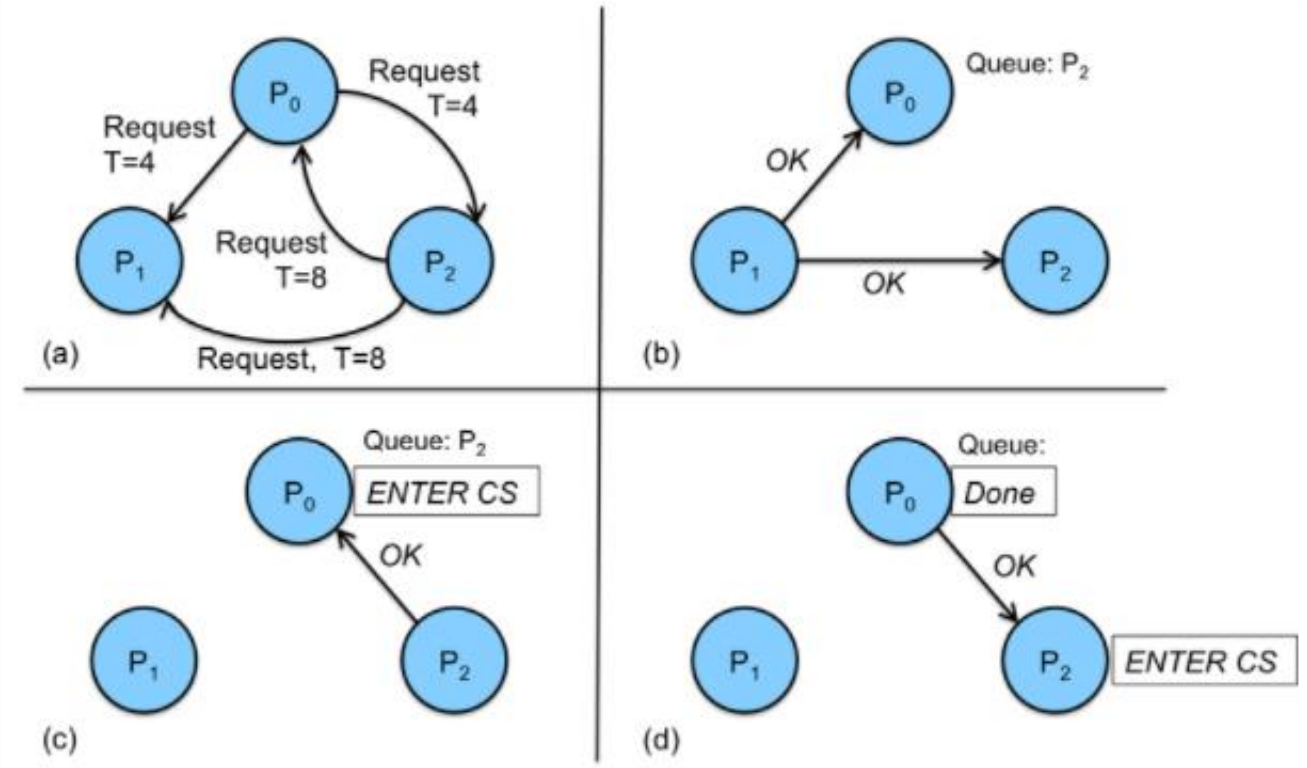
1. When a process (the "requesting process") decides to enter a critical region, a message is sent to all processes in the Distributed System (including itself).
2. What happens at each process depends on the "state" of the critical region.
3. If not in the critical region (and not waiting to enter it), a process sends back an OK to the requesting process.
4. If in the critical region, a process will queue the request and will not send a reply to the requesting process.
5. If **waiting** to enter the critical region, a process will:
    a) Compare the timestamp of the new message with that in its queue (note that the lowest timestamp wins).
    b) If the received timestamp wins, an OK is sent back, otherwise the request is queued (and no reply is sent back).
6. When all the processes send OK, the requesting process can safely enter the critical region.
7. When the requesting process leaves the critical region, it sends an OK to all the processes in its queue, then empties its queue.

Distributed Algorithm (1)

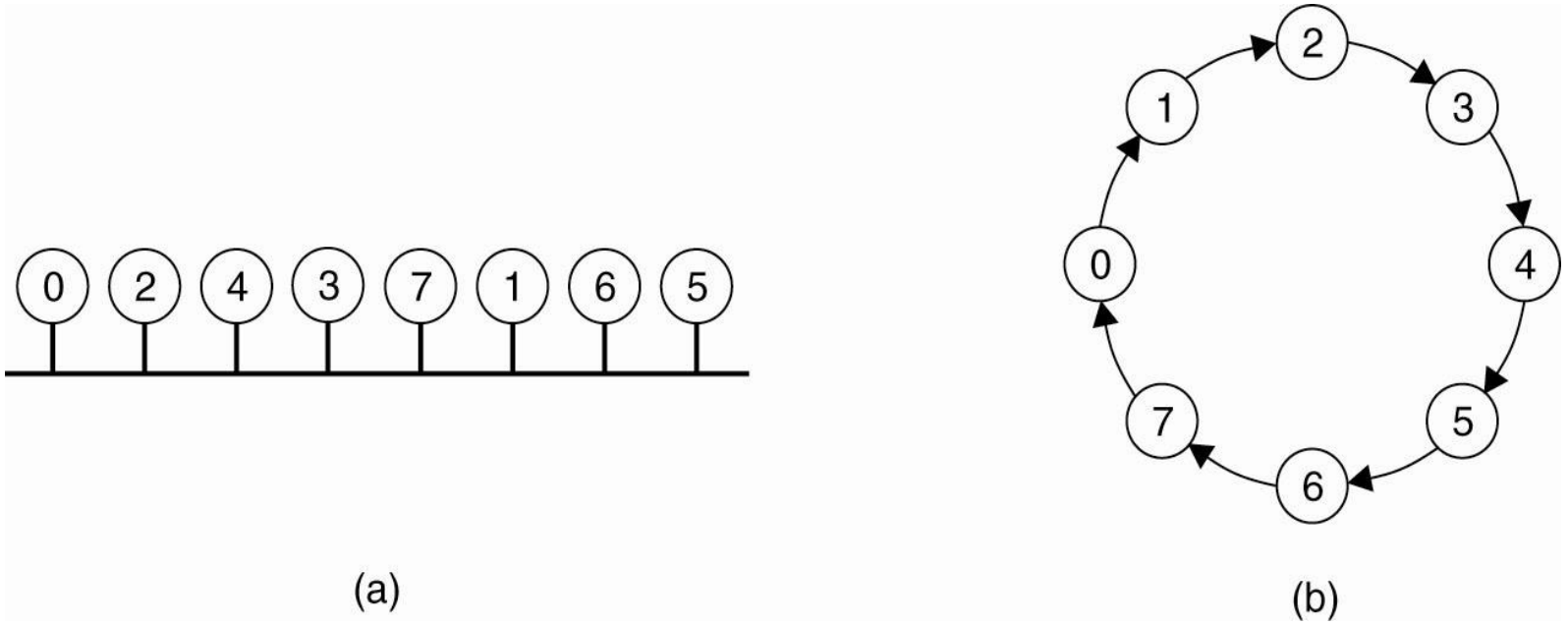
- Three different cases:

1. If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.

Distributed Algorithm (2)



A Token Ring Algorithm



(a) An unordered group of processes on a network.

(b) A logical ring constructed in software.

Comments: Token-Ring Algorithm

- **Advantages**:
  - It works (as there's only one token, so mutual exclusion is guaranteed).
  - It's fair – everyone gets a shot at grabbing the token at some stage.
- **Disadvantages**:
  - Lost token! How is the loss detected (it is in use or is it lost)? How is the token regenerated?
  - Process failure can cause problems – a broken ring!
  - Every process is required to maintain the current logical ring in memory – not easy.

Election Algorithms

- Many Distributed Systems require a process to act as *coordinator* (for various reasons). The selection of this process can be performed automatically by an "election algorithm".
- For simplicity, we assume the following:
  - Processes each have a unique, positive identifier.
  - All processes know all other process identifiers.
  - The process with the highest valued identifier is duly elected coordinator.
- When an election "concludes", a coordinator has been chosen and is known to all processes.

Goal of Election Algorithms

- The overriding goal of all election algorithms is to have all the processes in a group *agree* on a coordinator.
- There are two types of election algorithm:

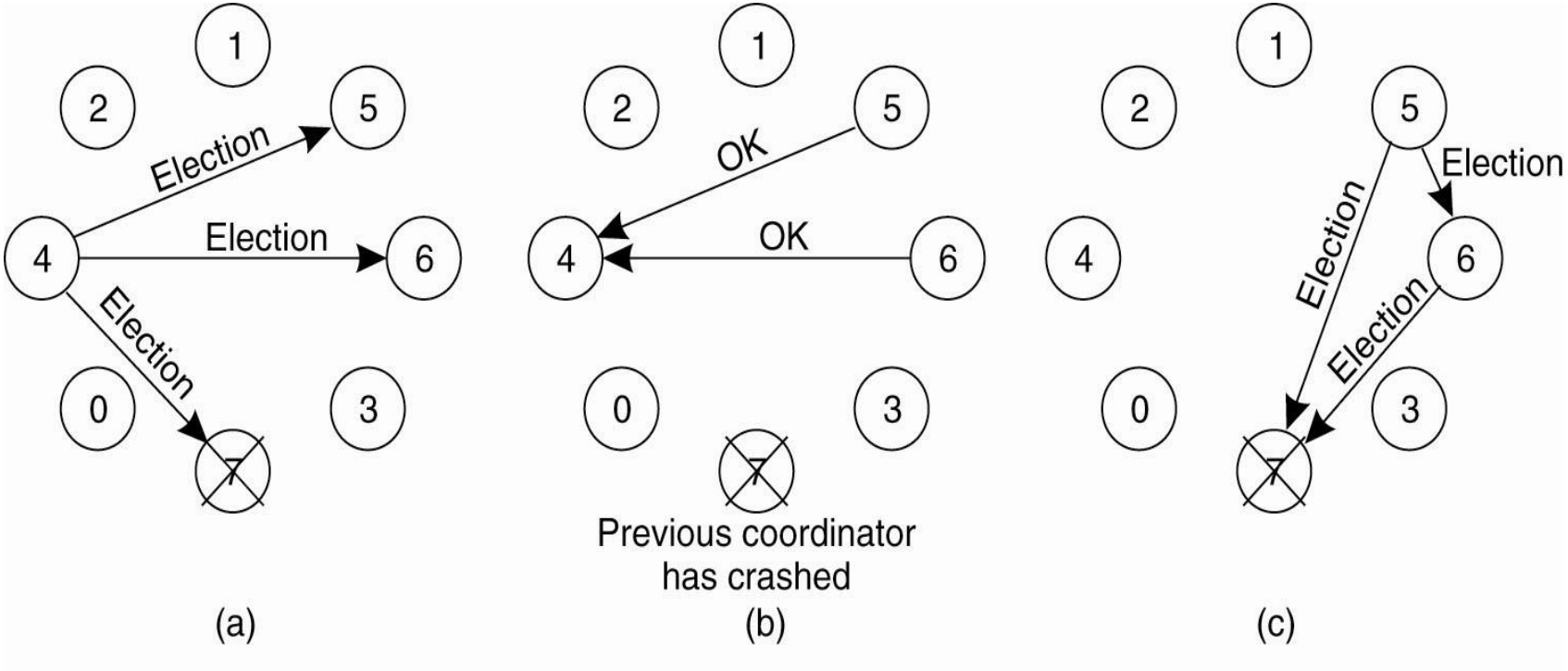
1. **Bully**: “the biggest guy in town wins”.
2. **Ring**: a logical, cyclic grouping.

Election Algorithms

- The Bully Algorithm:

1. *P* sends an *ELECTION* message to all processes with higher numbers.
2. If no one responds, *P* wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. *P*'s job is done.

The Bully Algorithm (1)



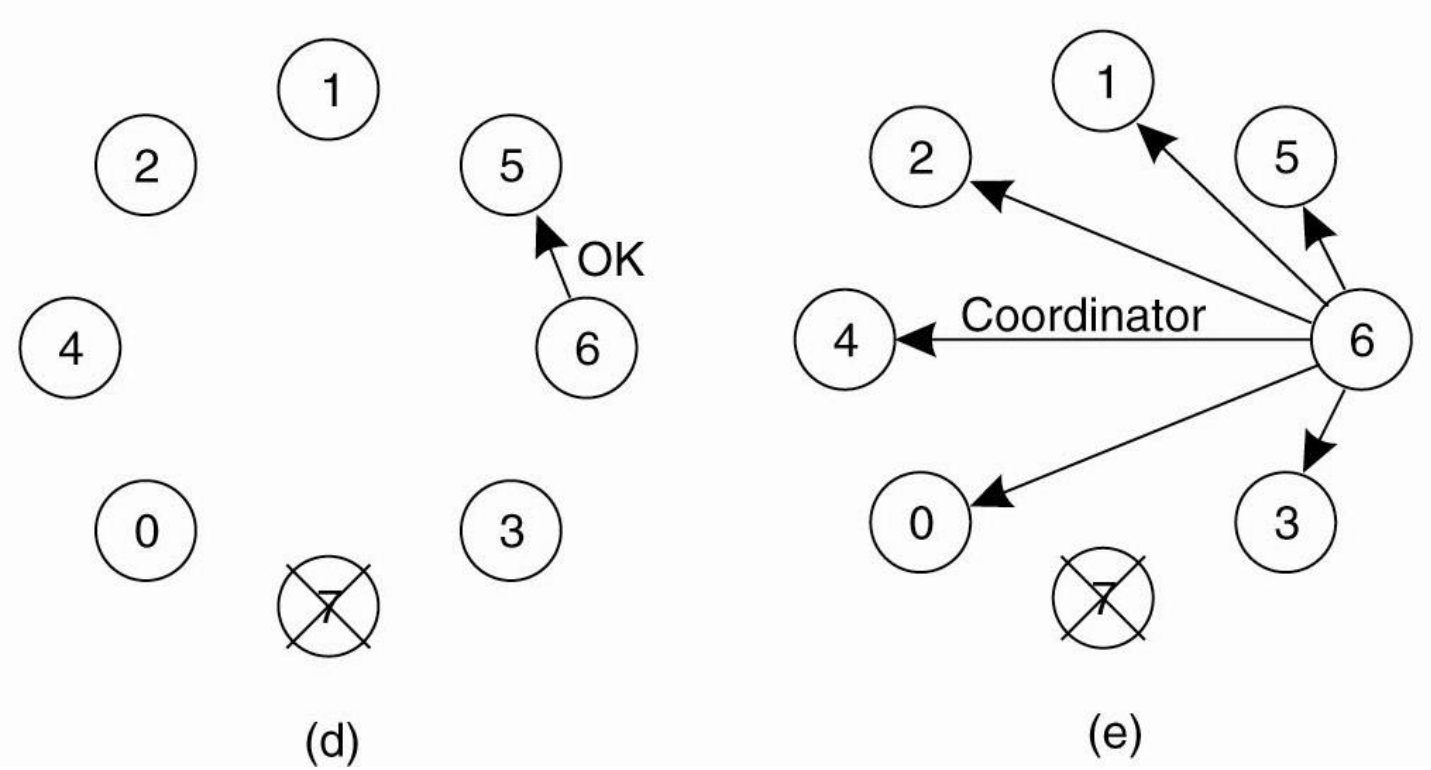
The bully election algorithm:

(a) Process 4 holds an election.

(b) 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.

The “Ring” Election Algorithm

- The processes are ordered in a “logical ring”, with each process knowing the identifier of its successor (and the identifiers of all the other processes in the ring).
- When a process “notices” that a coordinator is down, it creates an ELECTION message (which contains its own number) and starts to circulate the message around the ring.
- Each process puts itself forward as a candidate for election by adding its number to this message (assuming it has a higher numbered identifier).
- Eventually, the original process receives its original message back (having circled the ring), determines who the new coordinator is, then circulates a COORDINATOR message with the result to every process in the ring.
- With the election over, all processes can get back to work.

A Ring Algorithm

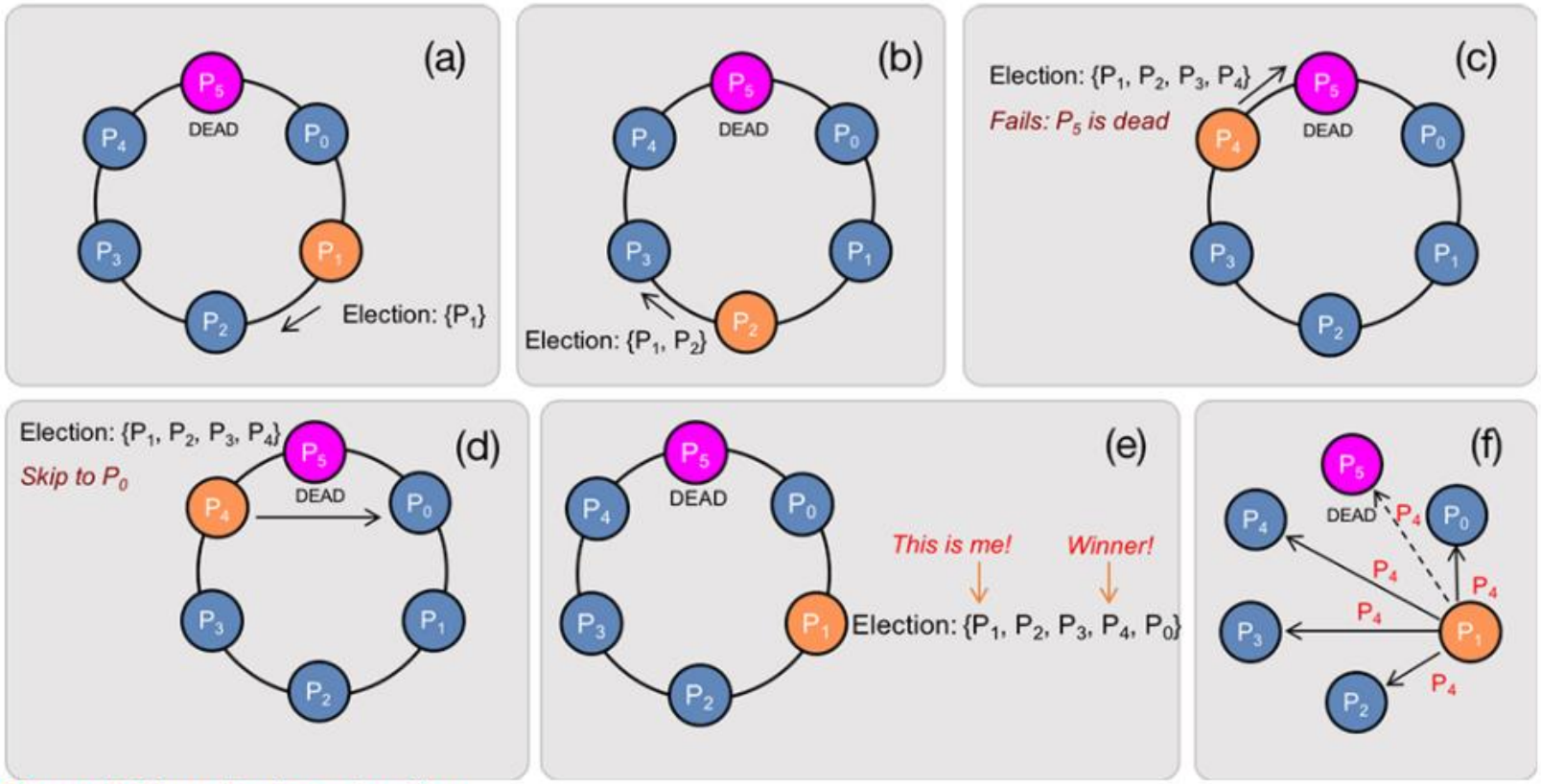


Figure 4. Ring election algorithm

Location systems

When looking at very large distributed systems that are dispersed across a wide-area network, it is often necessary to take proximity into account.

Just imagine a distributed system organized as an overlay network in which two processes are neighbors in the overlay network, but are actually placed far apart in the underlying network.

If these two processes communicate a lot, it may have been better to ensure that they are also physically placed in each other proximity.

GPS: Global Positioning System

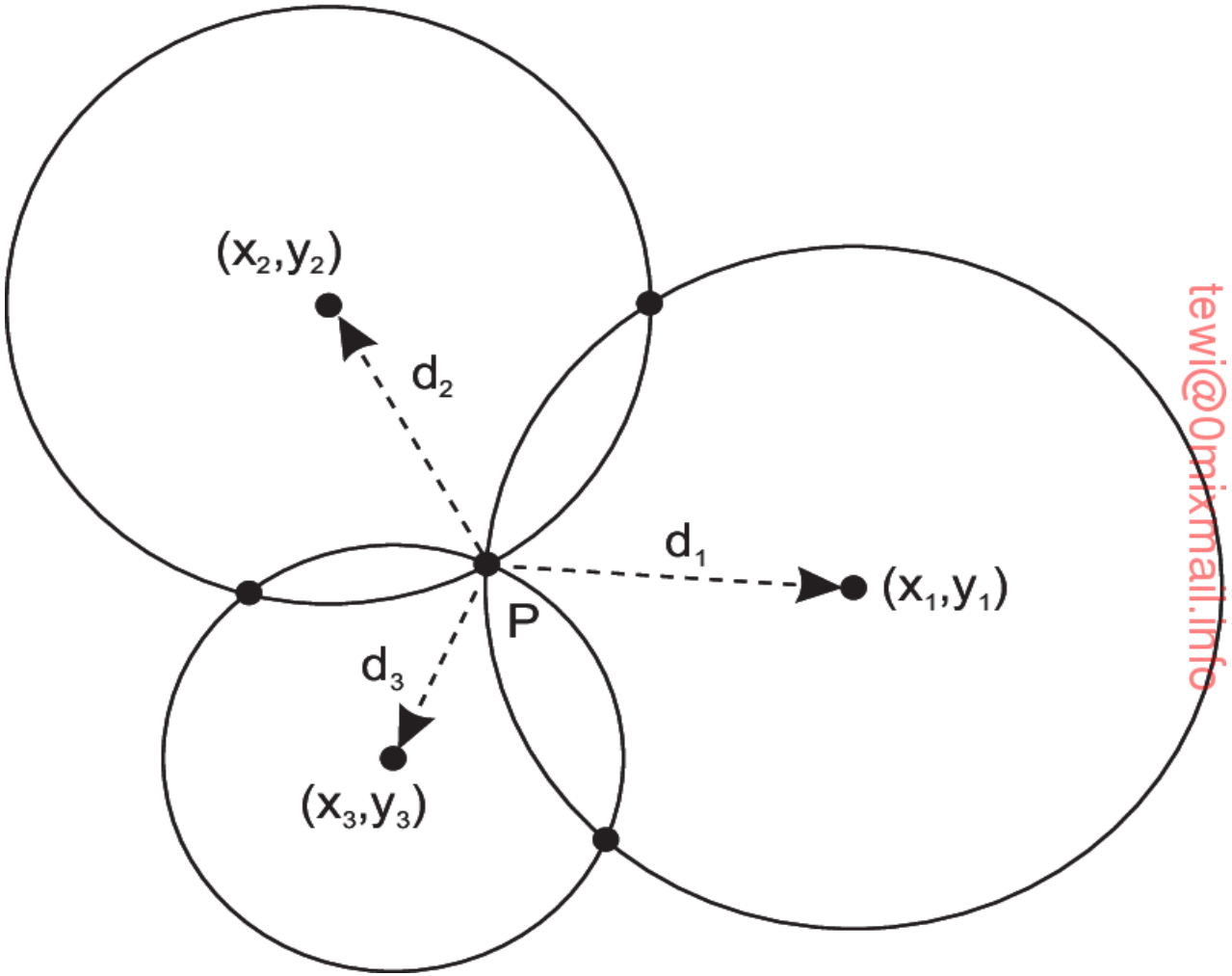
Let us start by considering how to determine your geographical position anywhere on Earth. This positioning problem is by itself solved through a highly specific, dedicated distributed system, namely **GPS**, which is an acronym for **Global Positioning System**.

it initially was used mainly for military applications, it by now has found its way to many civilian applications, notably for traffic navigation. However, many more application domains exist. For example, modern smartphones now allow owners to track each other's position.

GPS uses up to 24 satellites each circulating in an orbit at a height of approximately 20,000 km.


A
B
C

GPS: Global Positioning System




Latitude
Longitude

GPS: Global Positioning System

This principle of intersecting circles can be expanded to three dimensions, meaning that we need to know the distance to four satellites to determine the longitude, latitude, and altitude of a receiver on Earth.

Distributed event matching

•As a final subject concerning the coordination among processes, we consider distributed event matching. Event matching, or more precisely, **notification filtering**, is at the heart of publish-subscribe systems. The problem boils down to the following:

- A process specifies through a subscription S in which events it is interested.
- When a process publishes a notification N on the occurrence of an event, the system needs to see if S *matches* N.
- In the case of a match, the system should send the notification N, possibly including the data associated with the event that took place, to the subscriber.

Centralized implementations

- A simple, naive implementation of event matching is to have a fully centralized server that handles all subscriptions and notifications. In such a scheme, a subscriber simply submits a subscription, which is subsequently stored. When a publisher submits a notification, that notification is checked against each and every subscription, and when a match is found, the notification is copied and forwarded to the associated subscriber.

Gossip-based coordination

As a final topic in coordination, we take a look at a few important examples in which gossiping is deployed. In the following, we look at aggregation, large-scale peer sampling, and overlay construction, respectively.

Aggregation

•Gossiping can be used to discover nodes that have a few outgoing wide-area links, to subsequently apply directional gossiping.

•Consider the following information exchange. Every node P_i initially chooses an arbitrary number, say v_i. When node P_i contacts node P_j, they each update their value.

A peer-sampling service

.

A solution is to construct a fully decentralized **peer-sampling service**, or **PSS** for short.

Each node maintains a list of *c* neighbors, where, ideally, each of these neighbors represents a randomly chosen *live* node from the current set of nodes. This list of neighbors is also referred to as a **partial view**

A peer-sampling service

- The different selection operations are specified as follows:
- `selectPeer`: Randomly select a neighbor from the local partial view
- `selectToSend`: Select some other entries from the partial view, and add to the list intended for the selected neighbor.
- `selectToKeep` : Add received entries to partial view, remove repeated items, and shrink view to *c* items.

Gossip-based overlay construction

The lowest layer constitutes an unstructured peer-to-peer system in which nodes periodically exchange entries of their partial views with the aim to provide a peer-sampling service.

The lowest layer passes its partial view to the higher layer, where an additional selection of entries takes place. This then leads to a second list of neighbors corresponding to the desired topology.