Synchronization in Distributed Systems

Issue

- Synchronization within one system is hard enough
  - Semaphores
  - Messages
  - Monitors
  - ...
- Synchronization among processes in a distributed system is much harder
  - Synchronization based on **time**.
  - Synchronization based on **token ring**.
  - Synchronization based on **diffusing calculus**.

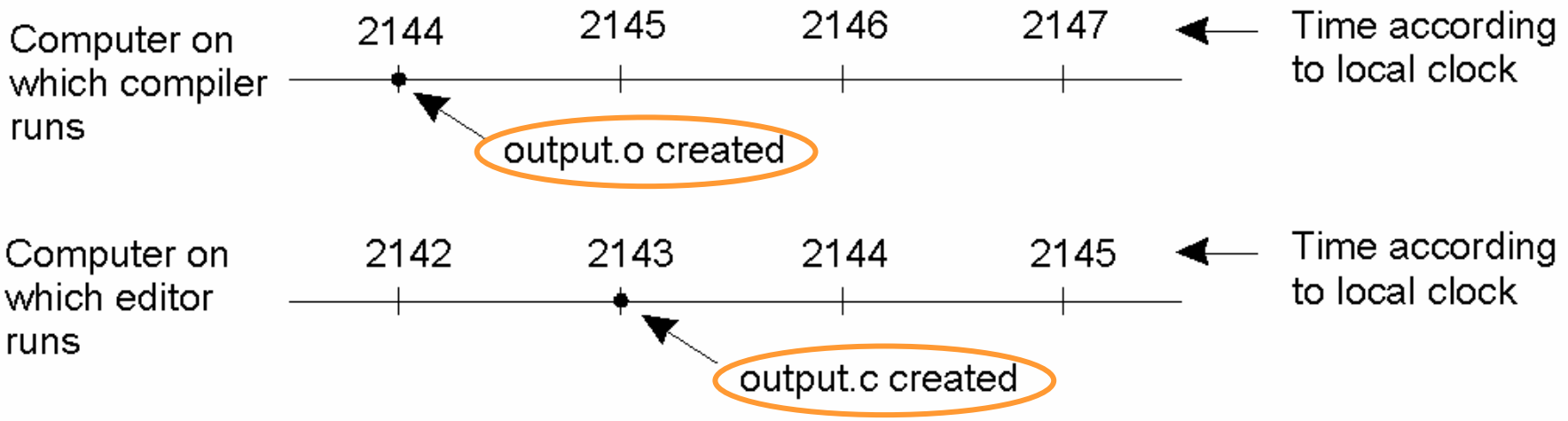
Synchronization in Distributed Systems

Time Based Synchronization

What about using *Time*?

- **`make`** recompiles if *foo.c* is newer than *foo.o*
- Scenario
  - **`make`** on machine *A* to build *foo.o*
  - Test on machine *B*; find and fix a bug in *foo.c*
  - Re-run **`make`** on machine *B*
  - *Nothing happens!*
- Why?

Clock Synchronization



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

System Clocks

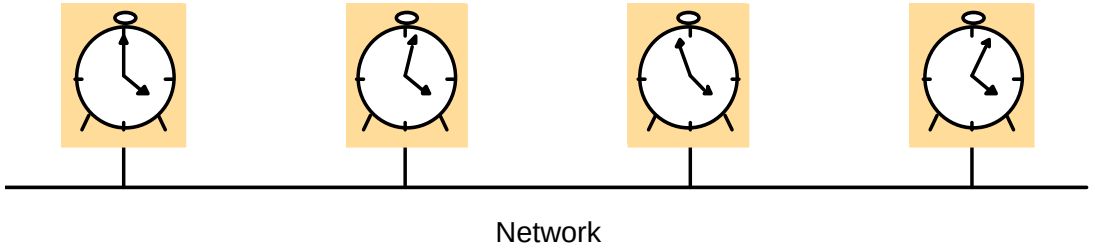
- Almost all computers have clocks (timers)
- A precisely cut quartz crystal kept under tension
  - Oscillates at a well-defined frequency
- Two registers – Counter & holding register
  - Each oscillation of crystal decrements counter by 1
  - Interrupt when counter is zero & reload from holding reg.
  - Each interrupt is a **tick**
- Clocks at multiple CPUs cannot be guaranteed to oscillate at exact same frequency

Terms

- Clock Skew
  - The difference in time values of two clocks is called clock skew
- Clock synchronization
  - Two clocks are said to be synchronized at a particular instance of time if the clock skew of the two clocks is less than some specified constant δ
  - A set of clocks are said to be synchronized if the clock skew of any two clocks in this set is less than δ

Clock Synchronization Issues

- A distributed system requires:
  - External Synchronization
    - Each process i, synchronizes the clock C_i with an authoritative, external source of time
  - Internal Synchronization
    - Each process i,j, synchronizes the clock C_i and C_j with each other
- Each computer runs its own physical clock



How is Time Actually Measured?

- Solar time – Based on earth's rotation
  - Transit of sun: Sun reaching the highest apparent in sky
  - Solar day: Time b/w two consecutive sun transits
  - Solar second: $1/86400^{th}$ of a solar day
- Atomic time
  - Second: Time for cesium-133 atom to make 9,192,631,770 transitions
  - International atomic time (TAI)
- Leap seconds to resolve difference b/w TAI & solar time (UTC)
- NIST broadcasts UTC on radio station (WWV)

Clock Synchronization Algorithms

- Two related problems
  - If one machine has WWV receiver: synchronize all machines to machine with the WWV receiver
  - No WWV receiver: Keep all machines relatively synchronized
- Many algorithms with some key assumptions
  - Each machine has timer that causes interrupt H times/sec
  - Increments software clock on each interrupt
  - $C_p(t)$ indicates clock value when UTC time is t
  - In ideal world dC/dt = 1

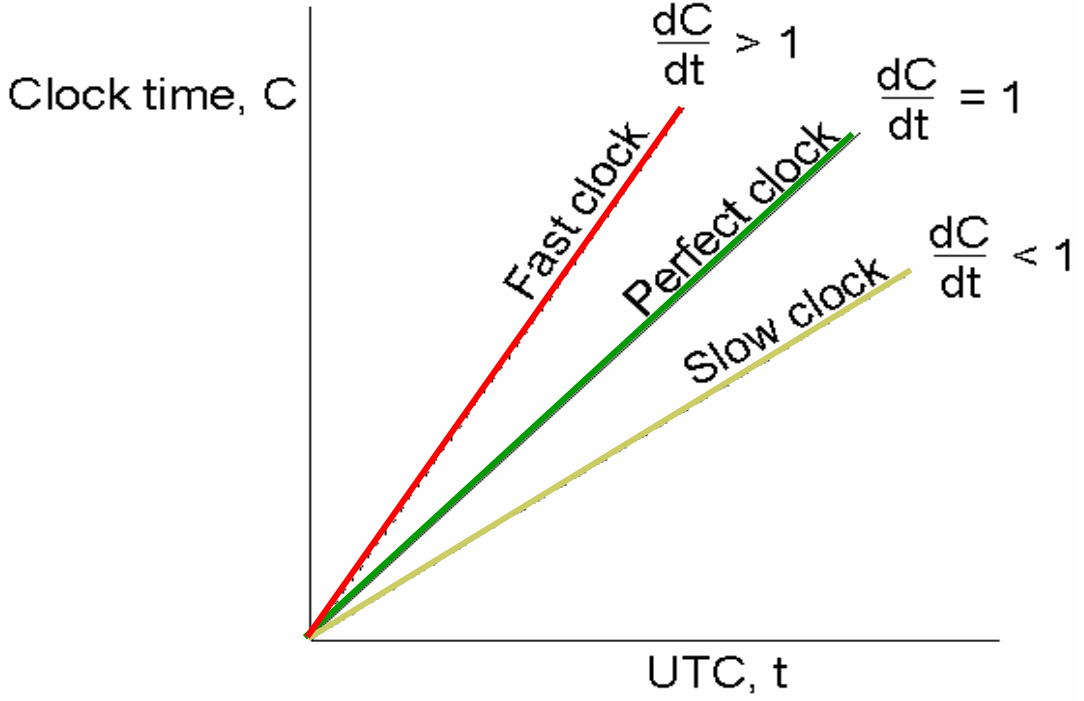
Clock Synchronization Algorithms

If $1 - \rho < dC/dt < 1 + \rho$ ρ maximum drift rate

If 2 clock drift from UTC in the opposite directions, after Δt,

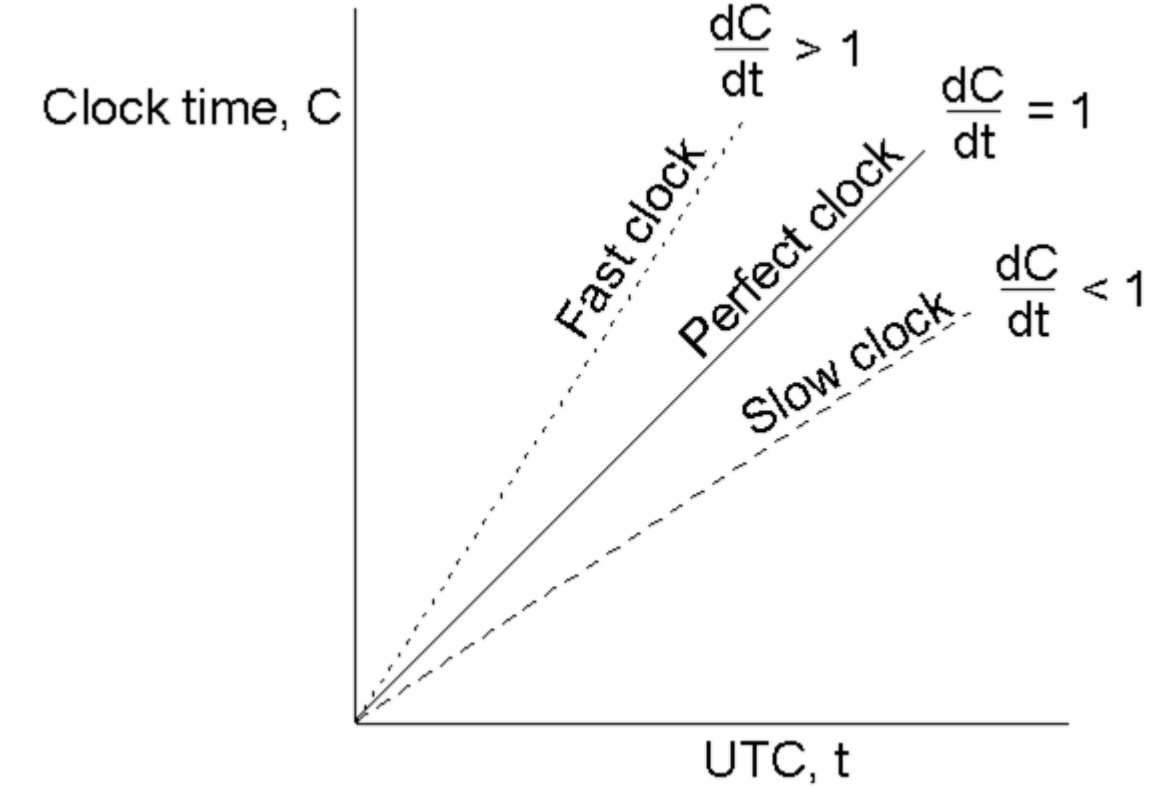
$\varepsilon = 2\rho\,\Delta t$

Resynchronization interval → $\delta/2\rho$ → δ maximum time difference



Drift & Max. Drift Rate

- In real world dC/dt is not one
- Maximum drift rate: ρ such that $1-\rho \leq dC/dt \leq 1+\rho$
  - Specified by manufactures



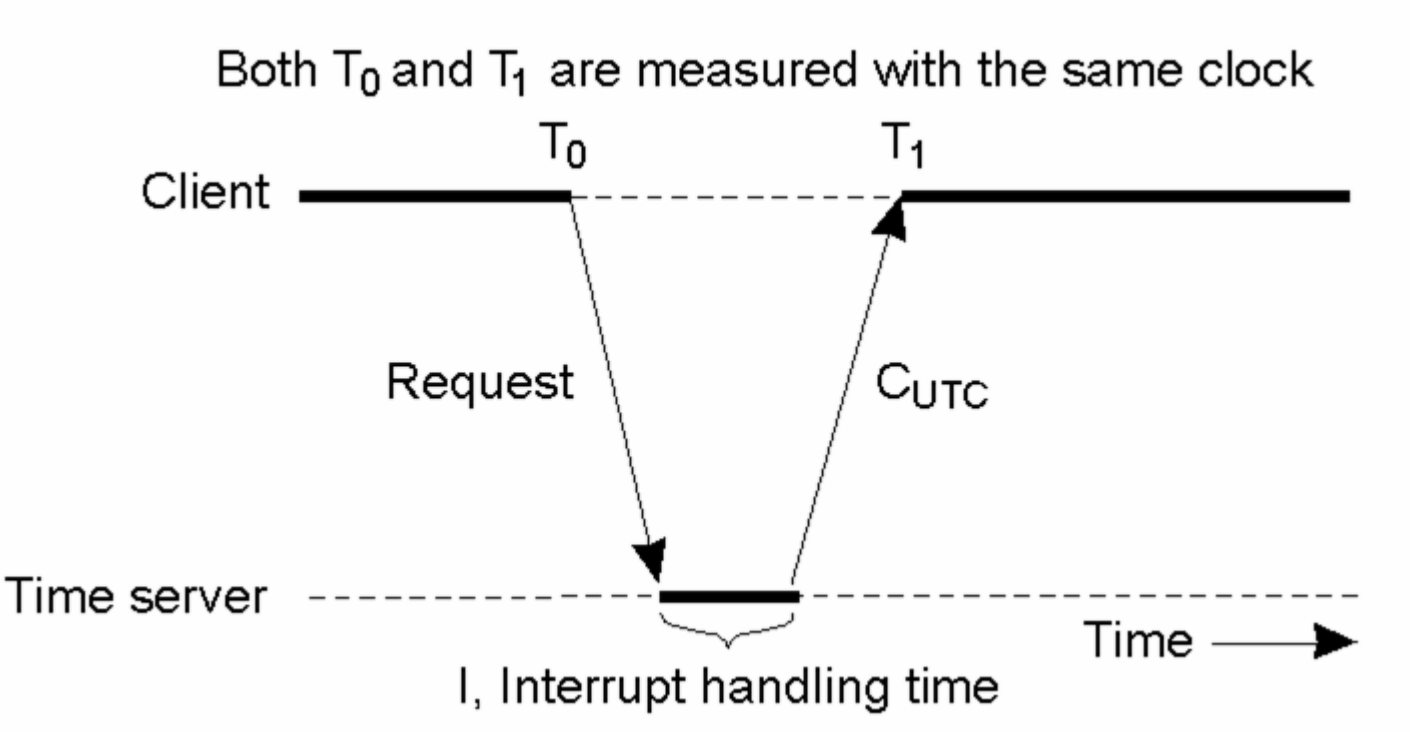
- The relation between clock time and UTC when clocks tick at different rates.

Clock Synchronization Algorithms

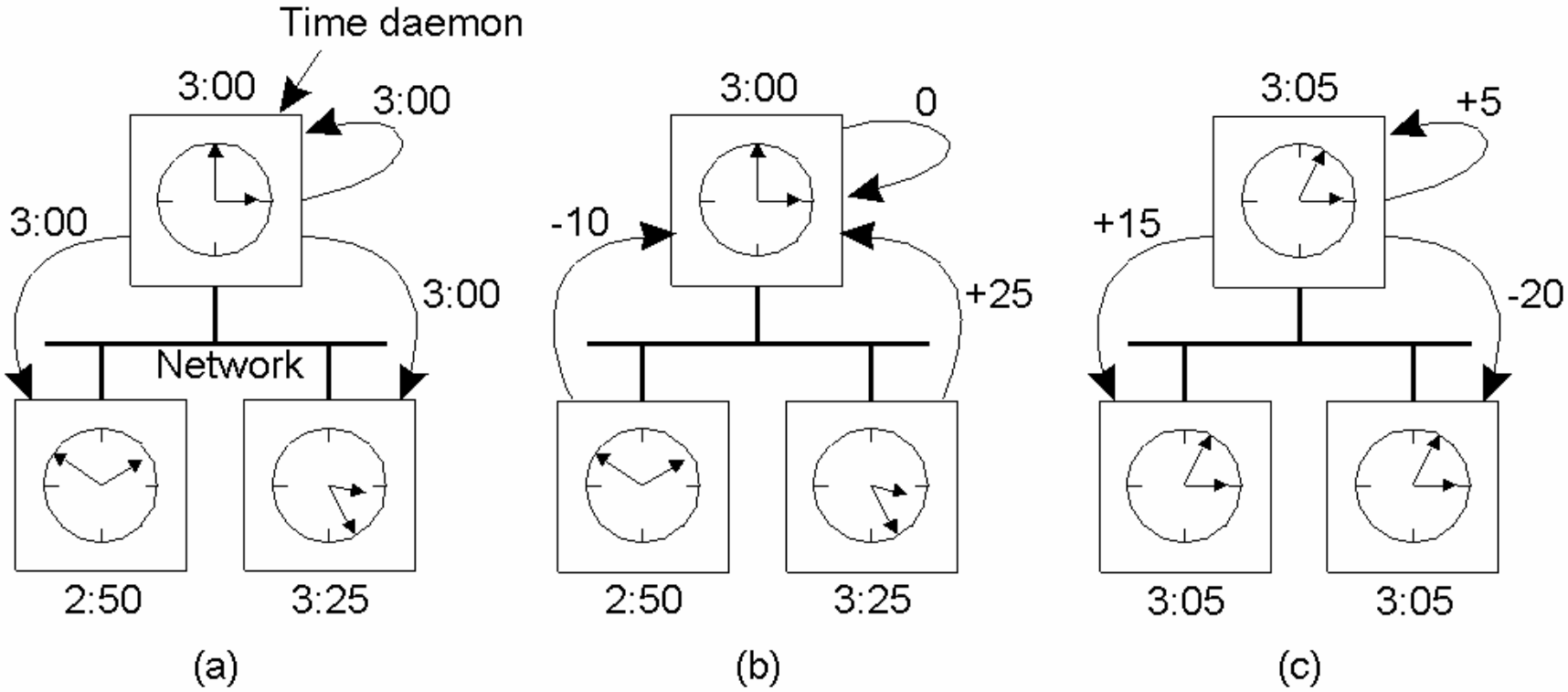
- Two clocks drifting from UTC in opposite directions at rate of ρ need to be synchronized every $\delta/2\rho$ secs.
- Christian's Algorithm:
  - Suited when one machine has WWV receiver
  - Each machine sends a request to time server periodically (period < $\delta/2\rho$) seconds
  - Time server responds with its current time (C_{UTC})
- Simple scheme
  - Set receivers time to C_{UTC}
  - Two problems
    - Clock might run backward !!!
    - Doesn't consider processing time

Cristian's Algorithm

- Introduce change gradually – Reduce time by a small amount



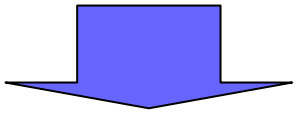
The Berkeley Algorithm



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

- Centralized algorithms have disadvantages.
- Decentralized algorithms can use averaging methods.
- NTP (Network Time Protocol) provides an accuracy of 1-50 msec using advanced algorithms.

- For many purposes it is sufficient that all machines agree on the same time.



Logical clocks

Often processes need to agree on the order in which events occur.

Logical Clocks

- For many applications it is sufficient if all machines agree upon some time
  - Synchronization with UTC not needed
- Logical clocks
- Lamport showed that in many cases clock synchronization is not needed
  - What actually is needed is agreement with regards to ordering of events
  - Example – Compilation occurred before file editing

Concepts in Logical Clocks

- **Happens-before** relation
- a->b if one of the following is true
  - a and b are events in same process and a occurs before b
  - If a is an event of sending a message and b is the event of receiving the same message is another process a->b.
    - Implies that message cannot be received before it is sent
- Happens-before is transitive
  - If a->b and b->c then a->c
- If two events in x and y are in two processes that never exchange messages then x & y are concurrent

Lamport's Timestamps Algorithm

- We need a way of measuring time such that for every event **a** we can assign a time **C(a)** on which all processes agree.
- If **a->b** than **C(a) < C(b)**
- Alternatively
  - If a and b are in same process and a precedes b C(a) < C(b)
  - If a representing sending a message and b represents receiving the same message C(a) < C(b)
- Clock C should always move forward

Lamport's Algorithm

- Each process runs with its own clock
  - Clocks need not be synchronized
- Processes sends messages that are time-stamped with the local clock time
- When a process receives a message
  - If its local clock is more than time-stamp of message no need of any adjustment
  - If local clock less than message's time-stamp, local clock incremented to one more than message's time-stamp
- Between every two events the clock is incremented
- Use decimal point followed by process number for global uniqueness

Lamport Timestamps

If $a \xrightarrow{\text{happens before}} b$ ⇒ $C(a) < C(b)$

If a snd., b rcv. ⇒ $C(a) < C(b)$

$C(a) \neq C(b)$

⇓

If $C(b) < C(a)$ ⇒ $C(b) = C(a)+1$

We obtain a total ordering of all events in the system

⇓

Vector timestamps

each process P_i has a vector V_i so that:

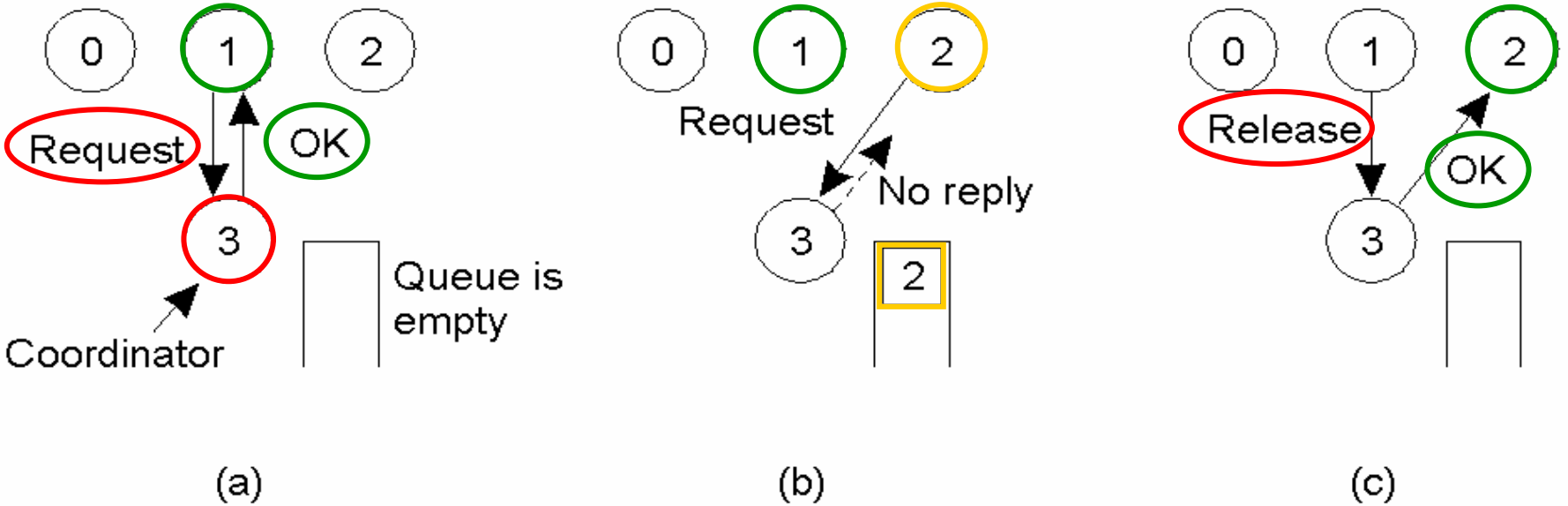
$V_i[i]$ is the number of events occurred so far at P_i

If $V_i[j]=k$ ⟶ P_i knows k events occurred at P_j

Mutual Exclusion:

A Centralized Algorithm

(to simulate a single processor system, needs a coordinator)



a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted

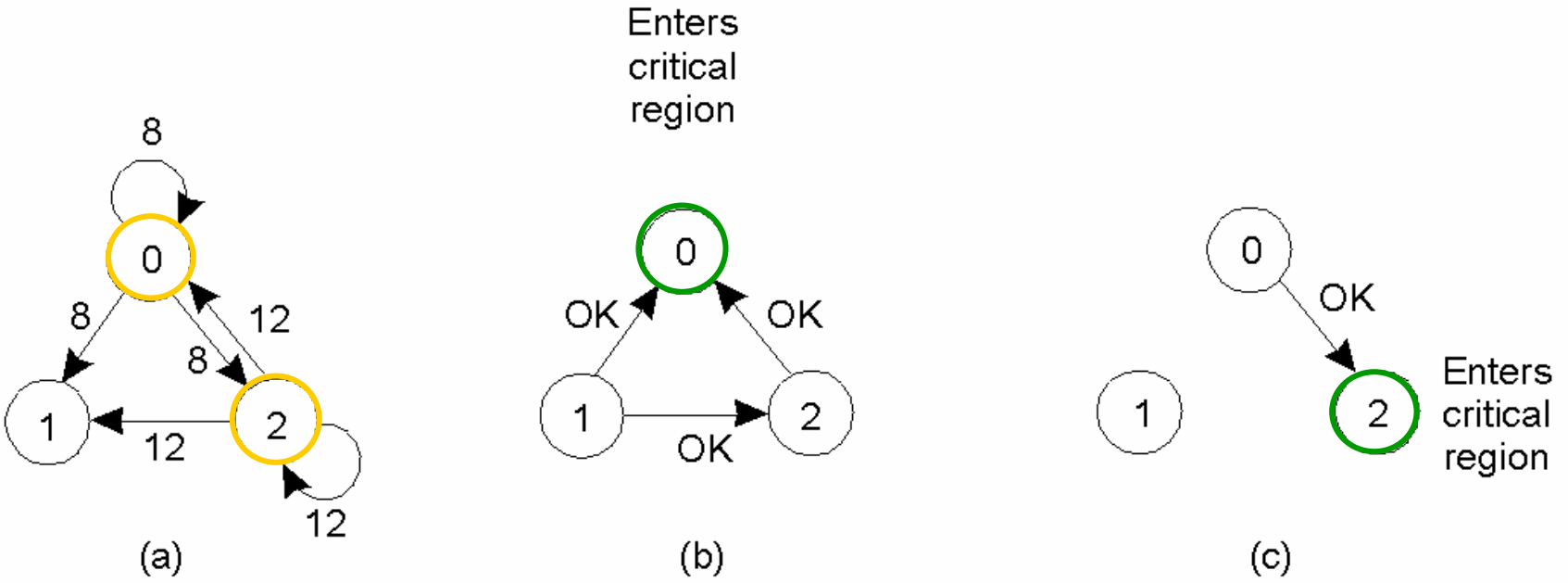
b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.

c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2

A Distributed Algorithm

requires a total ordering of all events in the system

A message contains the critical region name, the process number and the current time



a) Two processes (0,2) want to enter the same critical region at the *same* moment.

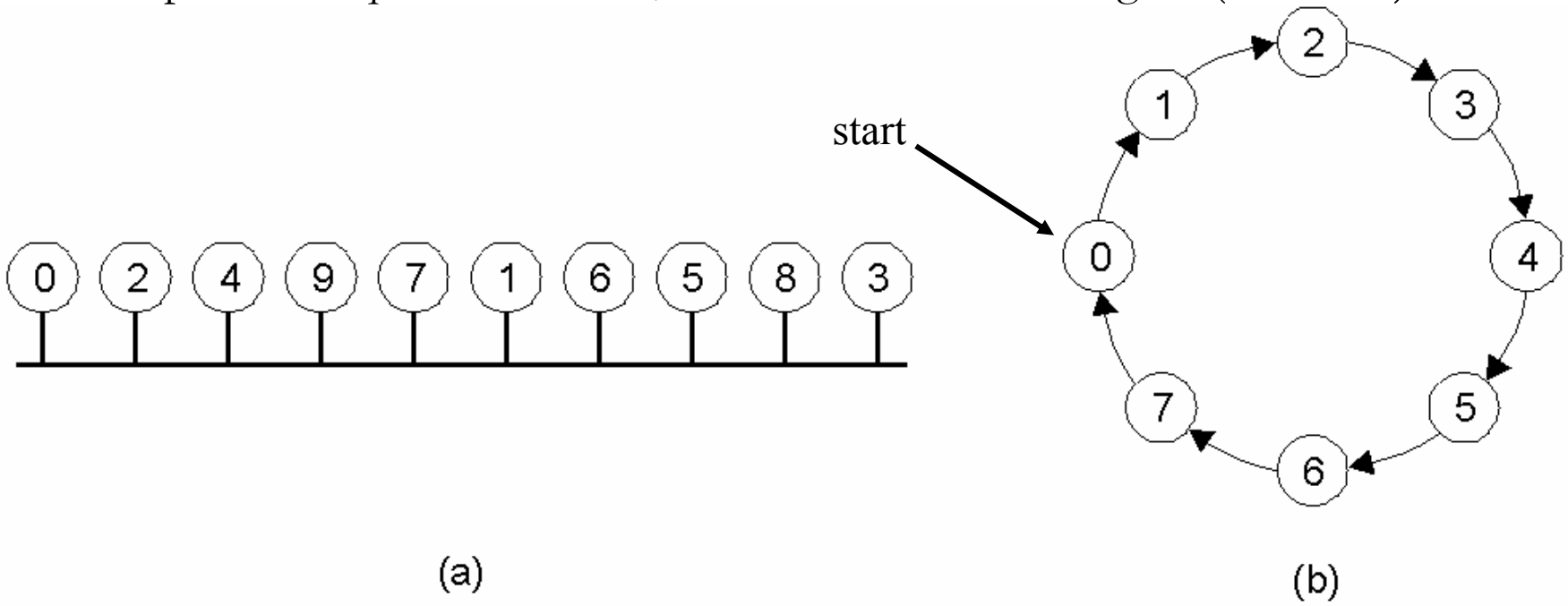
b) Process 0 has the lowest timestamp, so it wins and enters the critical region.

c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

d) This algorithm is worse than the centralized one (n points of failure, scaling, multiple messages…)

A Token Ring Algorithm

when the process acquires the token, it accesses the critical region (if needed)



a) An unordered group of processes on a network.
b) A logical, ordered, ring constructed in software. Each process knows who is the next in line