

Introduction to Distributed Systems

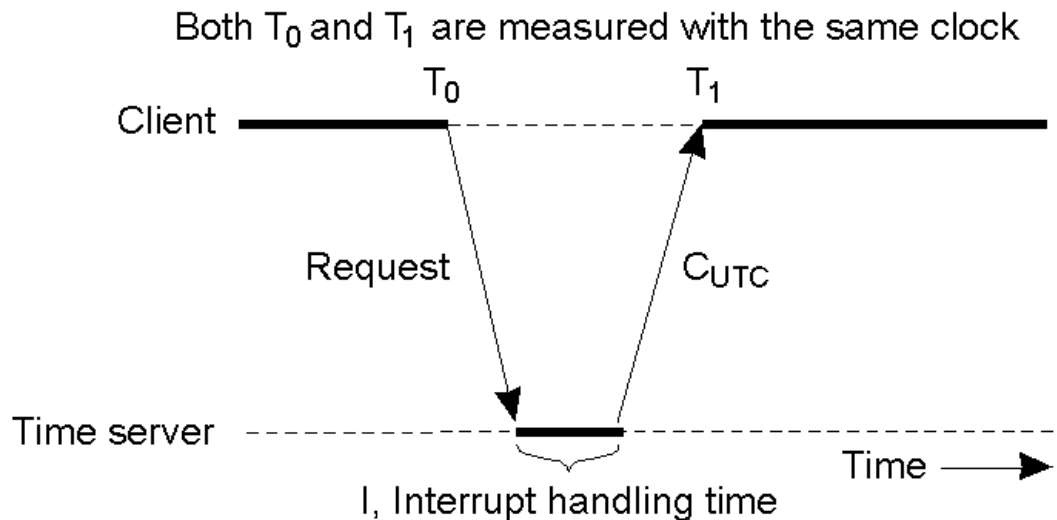
- Practice Issue: Process Communication
- Conceptual Issue: Local and Global Clocks

Absence of Global Clock

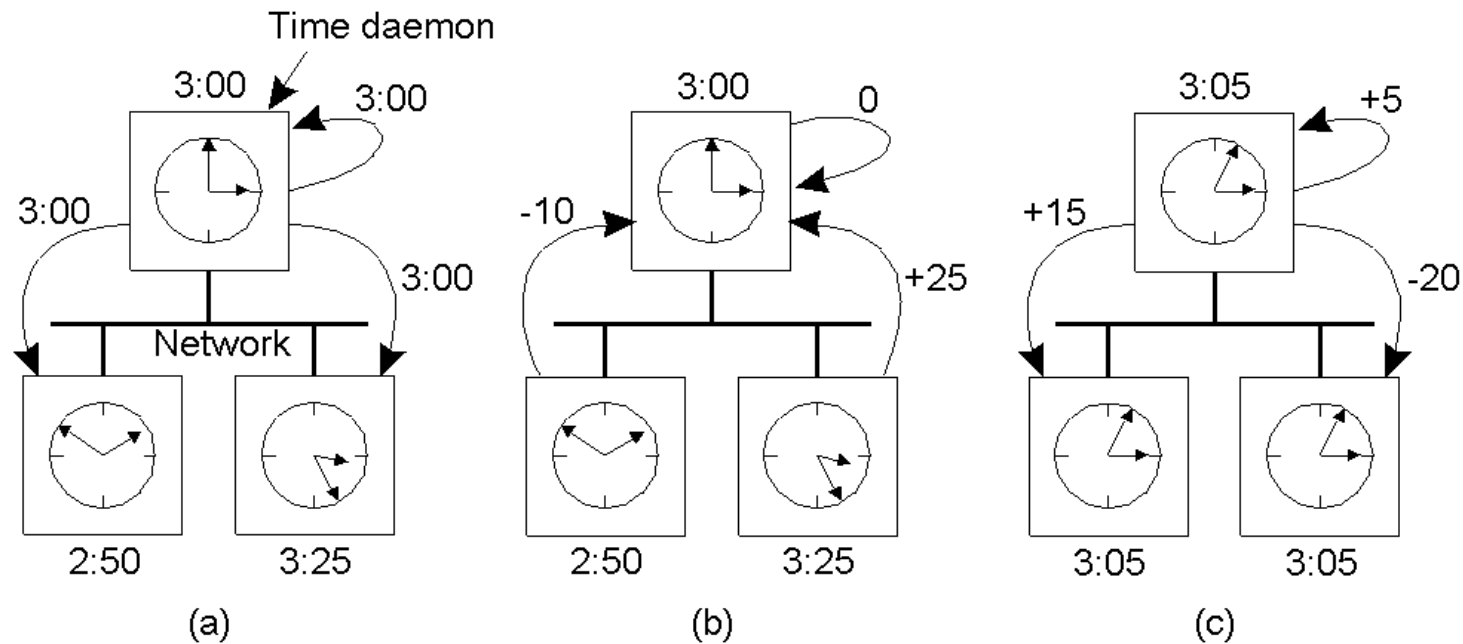
- Problem: synchronizing the activities of different part of the system
- What about using a single shared clock?
 - two different processes can see the clock at different times due to unpredictable transmission delays
- What about using radio synchronized clocks?
 - Propagation delays are unpredictable
- Software approaches
 - Clock synchronization algorithms
 - Logical clocks

Cristian's Algorithm

- Basic idea: get the current time from a time server.
- Issues:
 - Error due to communication delay - can be estimated as $(T_1 - T_0 - I)/2$
 - Time correction on client must be gradual



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

Logical clocks

- The need to order events in a distributed system has motivated schemes for “logical clocks”
- These artificial clocks provide some but not all of the functionality of a real global clock
- They build a clock abstraction based on underlying physical events of the system

“Happens-before” relation: definitions

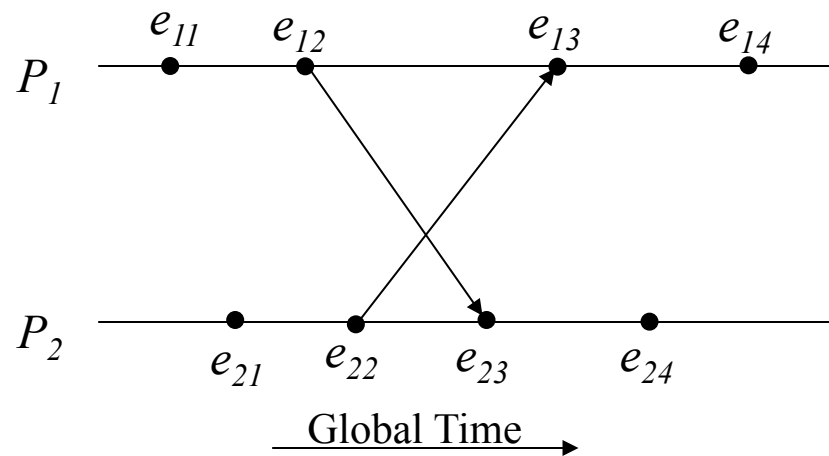
- “Happens-before” relation (\rightarrow):
 - $a \rightarrow b$ if a and b are in the same process and a occurred before b
 - $a \rightarrow b$ if a is the event of sending a message and b is the event of receiving the same message by another process
 - if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$, i.e. the relation “ \rightarrow ” is transitive
- The *happens-before* relation is a way of ordering events based on the behavior of the underlying computation

“Happens-before” relation: definitions (2)

- Two distinct events a and b are said to be *concurrent* ($a \parallel b$) if and

$$a \not\rightarrow b \quad b \not\rightarrow a$$
- For any two events in the system, either $a \rightarrow b$, $b \rightarrow a$ or $a \parallel b$
- Example:

$e_{11} \parallel e_{21}$
 $e_{22} \rightarrow e_{13}, e_{13} \rightarrow e_{14}$
 thus $e_{22} \rightarrow e_{14}$



Lamport's Logical Clocks: definitions

- A logical clock C_i at each process P_i is a function that assigns a number $C_i(a)$ to any event a , called *timestamp*
 - timestamps are monotonically increasing values
 - example: $C_i(a)$ could be implemented as a counter
- We want to build a logical clock $C(a)$ such that:
 - if $a \rightarrow b$ then $C(a) < C(b)$

Lamport's Logical Clocks: implementation

- If we want a logical clock $C(a)$ to satisfy:
if $a \rightarrow b$ then $C(a) < C(b)$

the following conditions must be met:

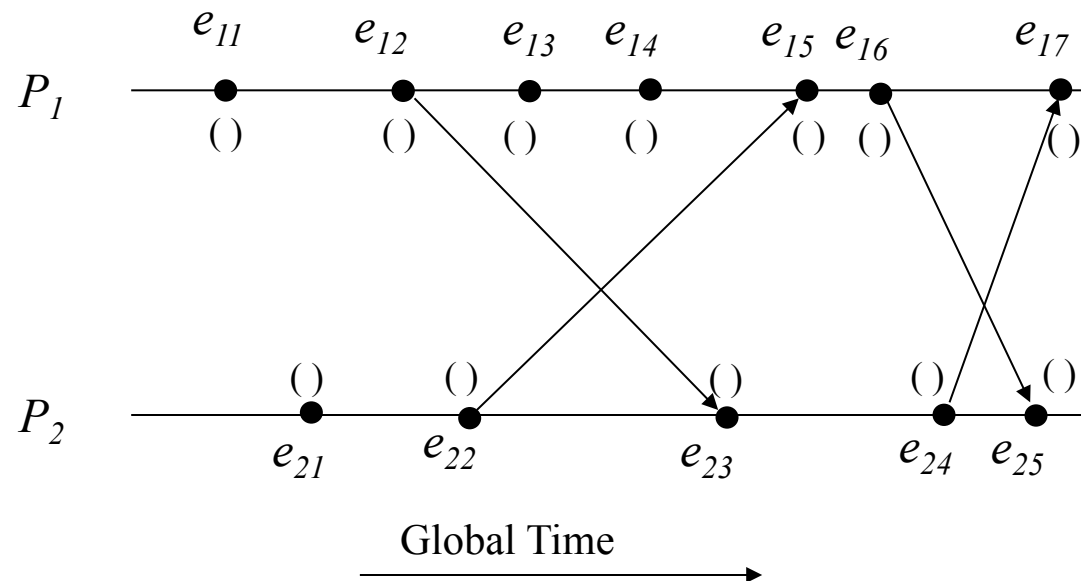
- if a and b are in the same process and a occurred before b , then $C_i(a) < C_i(b)$
- if a is the event of sending a message in process P_i and b is the event of receiving the same message by process P_j then $C_i(a) < C_j(b)$

Lamport's Logical Clocks: implementation (2)

- Two implementation rules that satisfy the previous correctness conditions are:
 - clock C_i is incremented by d at each event in process P_i :
$$C_i := C_i + d \quad (d > 0)$$
 - if event a is the sending of a message m by process P_i , then
 - message m is assigned the timestamp $t_m = C_i(a)$ ($C_i(a)$ is obtained after applying previous rule).
 - upon receiving message m , process P_j sets its clock to:
$$C_j := \max(C_j, t_m) + d \quad (d > 0)$$

Lamport's Logical Clocks: example

- Fill the blanks ...



Lamport's Logical Clocks: example

