Absence of Global Clock

• Problem: synchronizing the activities of different part of the system (e.g. process scheduling)
• 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 \((T_1-T_2-I)/2\)
  – Time correction on client must be gradual
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
“Happened before” relation: definitions

- “Happened before” relation (→):
  - a → b if a and b are in the same process and a occurred before b
  - a → 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 → b and b → c then a → c, i.e. the relation “→” is transitive

- The *happened before* relation is a way of ordering events based on the behavior of the underlying computation
“Happened before” relation: definitions (2)

• Two distinct events $a$ and $b$ are said to be concurrent ($a \parallel b$) if and
  
  \[ a \rightarrow b \quad b \rightarrow a \]

• 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:
  \[
  \text{if } a \rightarrow b \text{ then } C(a) < C(b)
  \]
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 implementations 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

Global Time

$P_1$

(1) $e_{11}$ (2) $e_{12}$ (3) $e_{13}$ (4) $e_{14}$ (5) $e_{15}$ (6) $e_{16}$ (7) $e_{17}$

$P_2$

(1) $e_{21}$ (2) $e_{22}$ (3) $e_{23}$ (4) $e_{24}$ (7) $e_{25}$
Lamport’s Clock limitations

- In Lamport’s system of logical clocks if \( a \rightarrow b \) then \( C(a) < C(b) \)
- However the opposite is not true
  - if \( C(a) < C(b) \) is not necessarily true that \( a \rightarrow b \) (see example)
  - the Vector Clocks version of Lamports’ clock idea addresses this limitation

\[
C(e_{11}) < C(e_{22}) \text{ and } e_{11} \rightarrow e_{22} \\
\text{but} \\
C(e_{11}) < C(e_{32}) \text{ and } e_{11} \nRightarrow e_{32}
\]
Vector clocks

• The timestamp $C_i$ of an event $a$ is a vector of length $n$
  – $C_i[i]$ is $P_i$’s own logical clock
  – $C_i[j]$ is $P_i$’s best guess of logical time at $P_j$’s

• Implementation rules:
  – events $a$ and $b$ are on same process: $C_i[i] = C_i[i] + d$
  – $a$ is the sending and $b$ the receiving of a message $m$: $\forall k, C_j[k] = \max(C_j[k], t_m[k])$
Example
Example
Vector clock: timestamp comparison

• Vector timestamps can be compared in the obvious way:
  - $t^a = t^b$ iff $\forall i, \ t^a[i] = t^b[i]$
  - $t^a \neq t^b$ iff $\exists i, \ t^a[i] \neq t^b[i]$
  - $t^a \leq t^b$ iff $\forall i, \ t^a[i] \leq t^b[i]$
  - $t^a < t^b$ iff $(t^a \leq t^b \land t^a \neq t^b)$

• Important observation:
  - $\forall i, \forall j : C_i[i] \geq C_j[i]$
Causally related events

• In a system with vector clocks:
  – $a \rightarrow b$ iff $t^a < t^b$

• Practical consequence: by comparing vector timestamps we can tell if two events are causally related:
  – $t^a < t^b \Rightarrow a \rightarrow b$
Proof Outline

• Proof Obligation
  (1) $a \rightarrow b$ implies $t^a < t^b$
  (2) $t^a < t^b$ implies $a \rightarrow b$

Cases to Consider
(a) a and b on the same process
(b) a and b on different processes