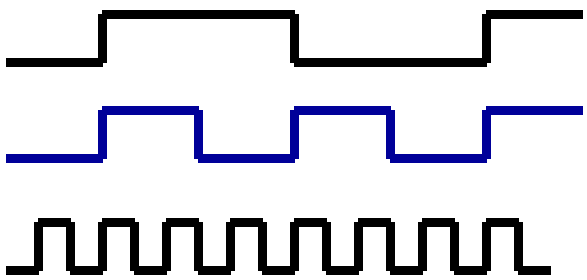
Logical Clocks and Causal Ordering

Why do we need global clocks?

- For causally ordering events in a distributed system
  - Example:
    - Transaction T transfers Rs 10,000 from S1 to S2
    - Consider the situation when:
      - State of S1 is recorded after the deduction and state of S2 is recorded before the addition
      - State of S1 is recorded before the deduction and state of S2 is recorded after the addition
- Should not be confused with the clock-synchronization problem



What data is being transmitted? 0101?

Yes, if this is the clock

If this is the clock, then 01110001

The receiver needs to know the clock of the sender

Ordering of Events

Lamport's *Happened Before* relationship:

For two events a and b, a → b if

- *a and b are events in the same process and a occurred before b, or*
- *a is a send event of a message m and b is the corresponding receive event at the destination process, or*
- *a → c and c → b for some event c*

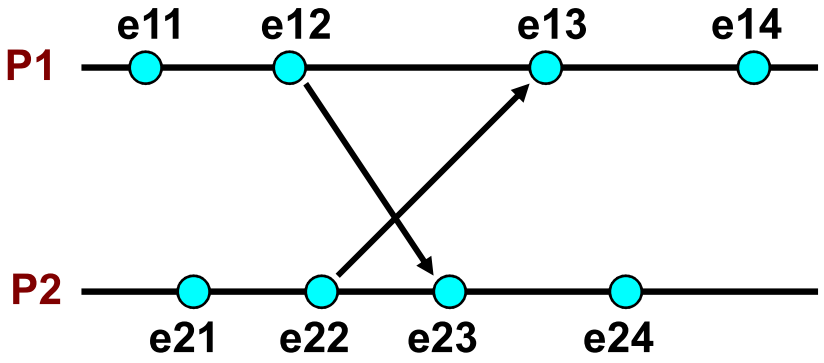
Causally Related versus Concurrent

Causally related events:

Event a causally affects event b if $a \rightarrow b$

Concurrent events:

- Two distinct events a and b are said to be concurrent (denoted by a||b) if $a \not\rightarrow b$ and $b \not\rightarrow a$



A space-time diagram

e11 and e21 are concurrent

e14 and e23 are concurrent

e22 causally affects e14

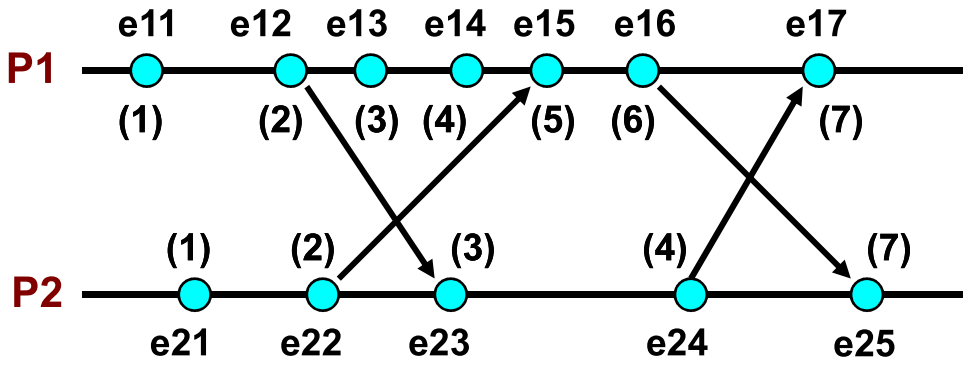
Lamport's Logical Clock

Each process i keeps a clock C_i

- Each event a in i is time-stamped $C_i(a)$, the value of C_i when a occurred
- C_i is incremented by 1 for each event in i
- In addition, if a is a send of message m from process i to j, then on receive of m,

$$Cj = \max (Cj, C_i(a)+1)$$

How Lamport's clocks advance



Points to note

- if a → b, then C(a) < C(b)
- → is a partial order
- Total ordering possible by arbitrarily ordering concurrent events by process numbers:
  - If a is any event in process P_i and b is any event in process P_j then a => b if and only if

    $C_i(a) < C_j(b)$ or

    $C_i(a) = C_j(b)$ and $P_i < P_j$ where < is an arbitrary relation that totally orders the processes to break ties. A simple way to implement < is to assign unique identification numbers to each process and then

    $P_i < P_j$ if i < j

Limitation of Lamport's Clock

$a \rightarrow b$ implies $C(a) < C(b)$

BUT

$C(a) < C(b)$ doesn't imply $a \rightarrow b$!!

So not a true clock !!

Solution: *Vector Clocks*

Each process P_i has a clock C_i, which is a vector of size n

The clock C_i assigns a vector $C_i(a)$ to any event *a* at P_i

Update rules:

- $C_i[i]$++ for every event at process i
- If a is send of message m from i to j with vector timestamp t_m, then on receipt of m:

  $Cj[k] = \max(Cj[k], t_m[k])$ for all k

Partial Order between Timestamps

For events a and b with vector timestamps t^a and t^b,

- Equal: $t^a = t^b$ iff $\forall i, t^a[i] = t^b[i]$
- Not Equal: $t^a \neq t^b$ iff $\exists i, t^a[i] \neq t^b[i]$
- Less or equal: $t^a \leq t^b$ iff $\forall i, t^a[i] \leq t^b[i]$
- Not less or equal: $t^a \leq t^b \mid$ iff $\exists i, t^a[i] > t^b[i]$
- Less than: $t^a < t^b$ iff $(t^a \leq t^b$ and $t^a \neq t^b)$
- Not less than: $t^a < t^b \mid$ iff $\neg(t^a \leq t^b$ and $t^a \neq t^b)$
- Concurrent: $t^a \mid\mid t^b$ iff $(t^a \not< t^b$ and $t^b \not< t^a)$

Causal Ordering

- $a \rightarrow b$ iff $t_a < t_b$
- Events a and b are causally related iff $t_a < t_b$ or $t_b < t_a$, else they are concurrent
- Note that this is still not a total order

Use of Vector Clocks in Causal Ordering of Messages

- If send(m1) → send(m2), then every recipient of both message m1 and m2 must “deliver” m1 before m2.
  - “deliver” – when the message is actually given to the application for processing

Problem of Vector Clock

- Message size increases since each message needs to be tagged with the vector
- Size can be reduced in some cases by only sending values that have changed