From Physical Clocks to Logical Clocks

- Physical clocks (last time)
  - With a receiver, a clock can be synchronized to within 0.1–10 ms of UTC
  - On a network, computer clocks can be synchronized to within 30 ms of each other (using NTP)
  - Quartz crystal clocks drift 1 µs per second (1 ms per 16.6 minutes)
  - In 30 ms, a 100 MIPS machine can execute 3 million instructions
  - We will refer to these clocks as *physical clocks*, and say they measure *global* time
- Idea — abandon idea of physical time
  - For many purposes, it is sufficient to know the **order** in which events occurred
  - Lamport (1978) — introduce logical (*virtual*) *time*, synchronize *logical clocks*

Events and Event Ordering

- For many purposes, it is sufficient to know the **order** in which two events occurred
  - An event may be an instruction execution, may be a function execution, etc.
  - Events include message send / receive
- Within a single process, or between two processes on the same computer,
  - the order in which two events occur **can** be determined using the physical clock
- Between two different computers in a distributed system,
  - the order in which two events occur **cannot** be determined using local physical clocks, since those clocks cannot be synchronized perfectly

The "Happened Before" Relation

- Lamport defined the *happened before* relation (denoted as "→"), which describes a **causal ordering** of events:

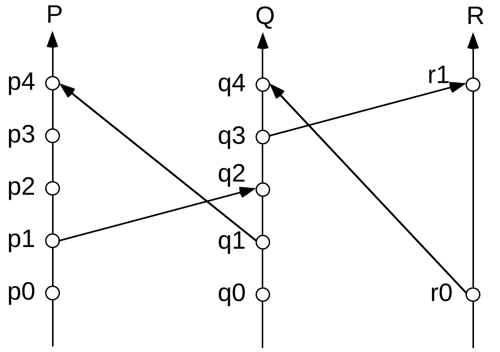
(1) if *a* and *b* are events in the same process, and *a* occurred before *b*, then $a \rightarrow b$

(2) if *a* is the event of sending a message *m* in one process, and *b* is the event of receiving that message *m* in another process, then $a \rightarrow b$

(3) if $a \rightarrow b$, and $b \rightarrow c$, then $a \rightarrow c$ (i.e., the relation "→" is transitive

- Causality:
  - Past events influence future events
  - This influence among causally related events (those that can be ordered by "→") is referred to as *causal affects*
  - If $a \rightarrow b$, event *a* causally affects event *b*

The "Happened Before" Relation (cont.)



- Concurrent events;
  - Two distinct events *a* and *b* are said to be *concurrent* (denoted "$a \parallel b$"), if neither $a \rightarrow b$ nor $b \rightarrow a$
  - In other words, concurrent events do not causally affect each other
- For any two events *a* and *b* in a system, either: $a \rightarrow b$ or $b \rightarrow a$ or $a \parallel b$

Lamport's Logical Clocks

- To implement "$\rightarrow$" in a distributed system, Lamport (1978) introduced the concept of logical clocks, which captures "$\rightarrow$" numerically
- Each process P_i has a *logical clock* C_i
- Clock C_i can assign a value $C_i(a)$ to any event a in process P_i
  - The value $C_i(a)$ is called the *timestamp* of event a in process P_i
  - The value $C(a)$ is called the *timestamp* of event a in whatever process it occurred
- The timestamps have no relation to physical time, which leads to the term *logical clock*
  - The logical clocks assign monotonically increasing timestamps, and can be implemented by simple counters

Conditions Satisfied by the Logical Clocks

- **Clock condition**: if $a \rightarrow b$, then $C(a) < C(b)$
  - If event a happens before event b, then the clock value (timestamp) of a should be less than the clock value of b
  - Note that we can **not** say: if $C(a) < C(b)$, then $a \rightarrow b$
- **Correctness conditions** (must be satisfied by the logical clocks to meet the clock condition above):

  [C1] For any two events a and b in the same process P_i, if a happens before b, then $C_i(a) < C_i(b)$

  [C2] If event a is the event of sending a message m in process P_i, and event b is the event of receiving that same message m in a different process P_k, then $C_i(a) < C_k(b)$

Implementation of Logical Clocks

- **Implementation Rules** (guarantee that the logical clocks satisfy the correctness conditions):

  [IR1] Clock C_i must be incremented between any two successive events in process P_i:

  $C_i := C_i + d$ ($d>0$) (usually $d=1$)

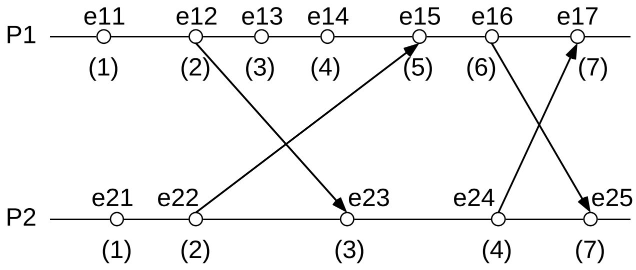
  [IR2] If event a is the event of sending a message m in process P_i, then message m is assigned a timestamp $t_m = C_i(a)$

  When that same message m is received by a different process P_k, C_k is set to a value greater than or equal to its present value, and greater than t_m:

  $C_k := \max(C_k, t_m + d)$ ($d>0$) (usually $d=1$)

Example of Logical Clocks

- Updating logical clocks using Lamport's method:

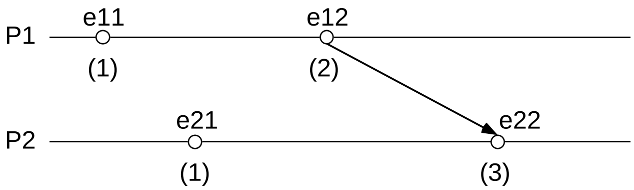


"e*nn*" is event; "(*n*)" is clock value

- Notes:
  - Clocks initially 0, $d=1$
  - Most clocks incremented due to IR1
  - Sends e12, e22, e16, and e24 use IR1
  - Receives e23, e15, and e17 set to C_k
  - Receive e25 sets to $t_m + d = 6+1 = 7$

Obtaining a Total Ordering Using Logical Clocks

- The happened before relationship "$\rightarrow$" defines an irreflexive **partial order** among events



- A **total order** of events ("$\Rightarrow$") can be obtained as follows:
  - If a is any event in process P_i, and b is any event in process P_k, then $a \Rightarrow b$ if and only if either:

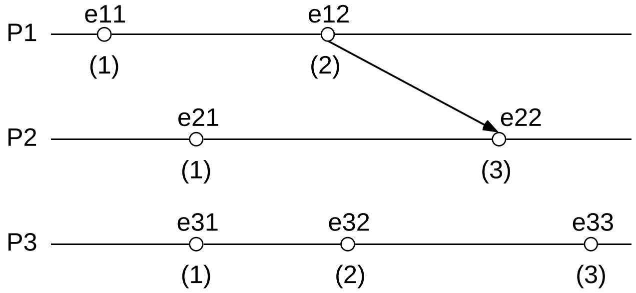
    $C_i(a) < C_k(b)$ **or**

    $C_i(a) = C_k(b)$ and $P_i \ll P_k$

    where "<<" denotes a relation that totally orders the processes to break ties

Limitation of Logical Clocks

- With Lamport's logical clocks, if $a \rightarrow b$, then $C(a) < C(b)$
  - The following is **not** necessarily true if events a and b occur in different processes: if $C(a) < C(b)$, then $a \rightarrow b$
- Example illustrating this limitation:



  - $C(e11) < C(e22)$, and $e11 \rightarrow e22$ is true
  - $C(e11) < C(e32)$, but $e11 \rightarrow e32$ is false

↪ Cannot determine whether two events are causally related from timestamps

Vector Clocks

- Independently proposed by Fidge and by Mattern in 1988
- Vector clocks:
  - Assume system contains n processes
  - Each process P_i has a clock C_i, which is an integer vector of length n

    $C_i = (C_i[1], C_i[2], \ldots C_i[n])$
  - $C_i(a)$ is the timestamp (clock value) of event a at process P_i
  - $C_i[i](a)$, entry i of of C_i, is P_i's logical time
  - $C_i[k](a)$, entry k of of C_i (where $k \neq i$), is P_i's best guess of the logical time at P_k
    - More specifically, the time of the occurrence of the last event in P_k which "happened before" the current event in P_i (based on messages received)

Implementation of Vector Clocks

- **Implementation Rules**:

  [IR1] Clock C_i must be incremented between any two successive events in process P_i:

  $C_i[i] := C_i[i] + d$ (d>0, usually d=1)

  [IR2] If event a is the event of sending a message m in process P_i, then message m is assigned a vector timestamp $t_m = C_i(a)$

  When that same message m is received by a different process P_k, C_k is updated as follows:

  $\forall p,\ C_k[p] := \max(C_k[p], t_m[p] + d)$ (usually d=0 unless needed to model network delay)

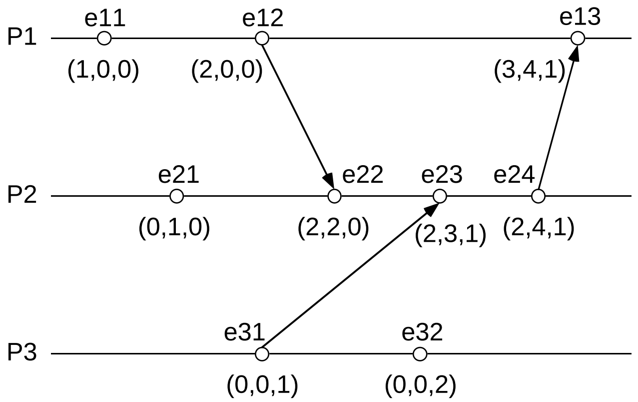
- It can be shown that $\forall i, \forall k: C_i[i] \geq C_k[i]$

Implementation of Vector Clocks (cont.)

- Rules for comparing timestamps can also be established so that if $t_a < t_b$, then $a \rightarrow b$
  - $t_a = t_b$ iff for all i, $t_a[i] = t_b[i]$
  - $t_a <> t_b$ iff for any i, $t_a[i] <> t_b[i]$
  - $t_a <= t_b$ iff for all i, $t_a[i] <= t_b[i]$ (each one equal or less)
  - $t_a < t_b$ iff $t_a <= t_b$ and $t_a <> t_b$ (some (but not all) equal, some less)
  - Overcomes the limitation of Lamport’s logical clocks
- Examples:
  - 1 1 2 3 = 1 1 2 3
  - 1 1 2 3 <> 1 1 2 4
  - 1 1 2 3 <= 1 1 2 4   1 1 2 3 <= 1 1 2 3
  - 1 1 2 3 < 1 1 2 4

Example of Vector Clocks

- Updating vector clocks:



“e*nn*” is event; “(*n*,*n*,*n*)” is clock value

- Notes:
  - Events e11, e21, and e12 updated by IR1
  - Receive e22 updated by IR1 and IR2
  - Receive e13 tells P1 about P2 and P3 (P3 clock is old, but better than nothing!)