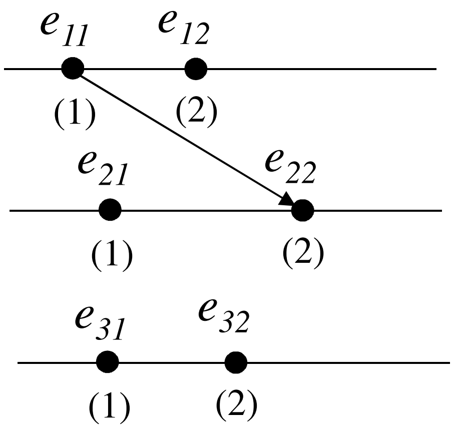
Lamport’s Clock limitations

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

$C(e_{11}) < C(e_{22})$ and $e_{11} \rightarrow e_{22}$
but
$C(e_{11}) < C(e_{32})$ and $e_{11} \not\rightarrow e_{32}$



Example: Totally Ordered Multicasting

- There are multiple replicas of bank account database.

- We need to guarantee the update operations to all the replicas following the same order.

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:

    $\forall k \neq j$, $C_j[k] = \max(C_j[k], t_m[k])$, and

    $C_j[j] = C_j[j] + d$

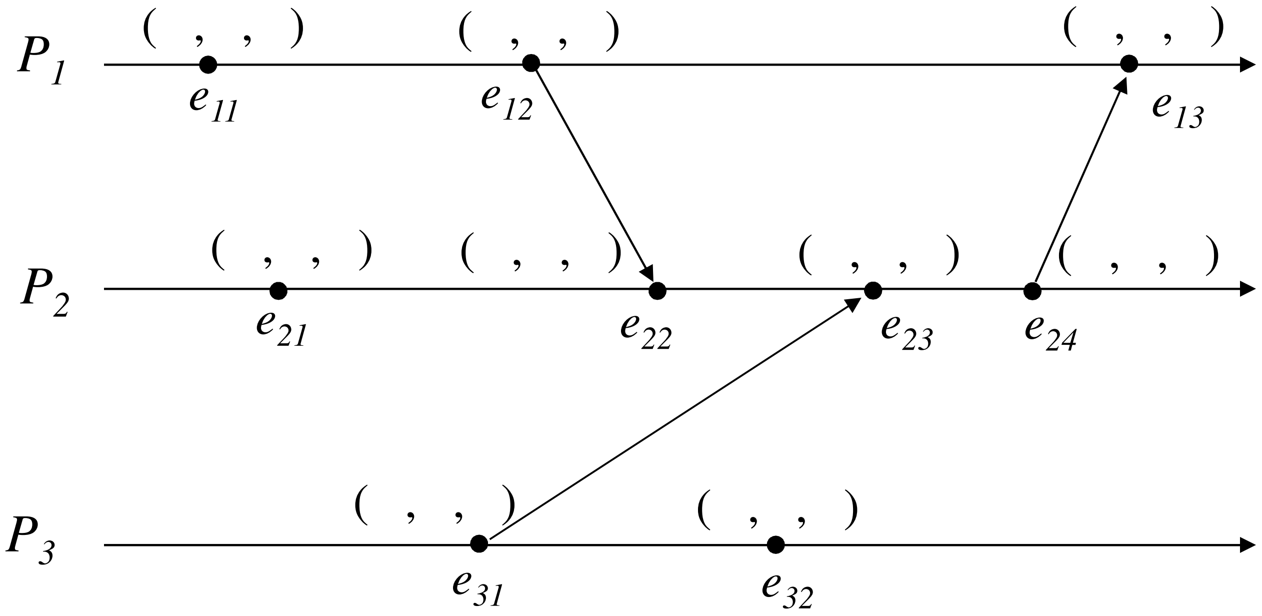
Vector clock: timestamp comparison

- 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 $(\ t^a \leq t^b \wedge t^a \neq t^b\ )$
- Important observation:
  - $\forall i, \forall j : C_i[i] \geq C_j[i]$

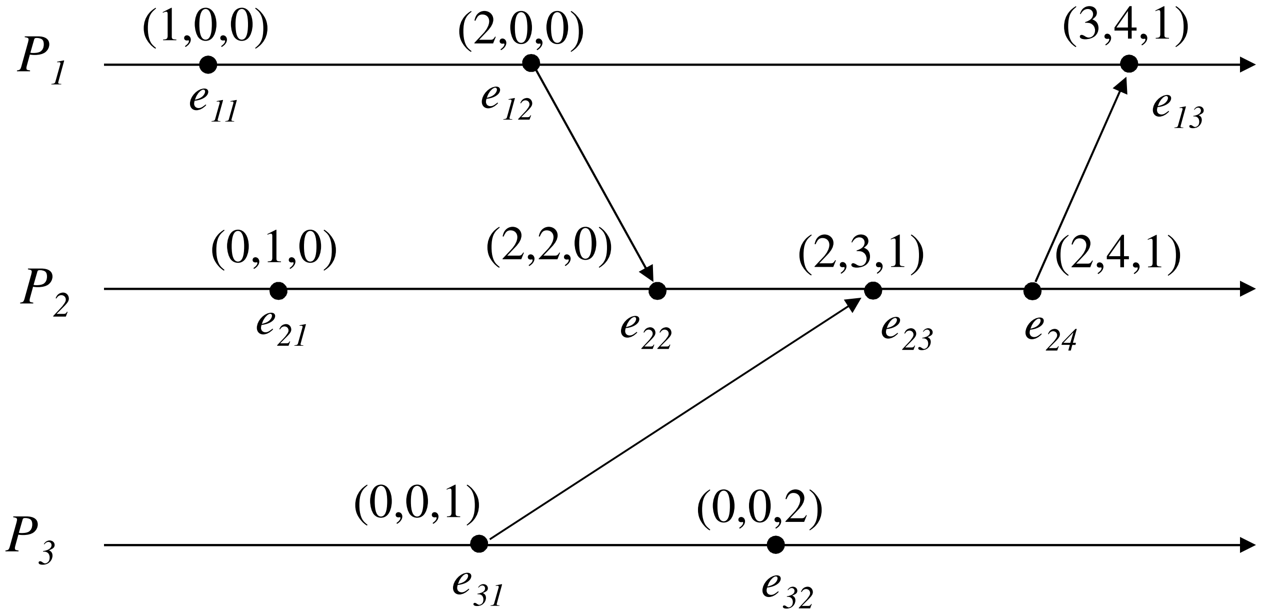
Causally related events

- 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:
  - $t^a < t^b \Rightarrow a \rightarrow b$

Example



Example


P_1
(1,0,0)
e_{11}
(2,0,0)
e_{12}
(3,4,1)
e_{13}
P_2
(0,1,0)
e_{21}
(2,2,0)
e_{22}
(2,3,1)
e_{23}
(2,4,1)
e_{24}
P_3
(0,0,1)
e_{31}
(0,0,2)
e_{32}

Using Happens-Before for Data Race Detection

- Data race:
  - Simultaneous shared memory accesses
  - At least one of them is write
- Detection algorithm:
  - Record memory accesses and happens-before relation at runtime based on the synchronization events
  - If no order between two shared memory accesses in two processes/threads, they may incur potential data races.

Mutual exclusion in distributed systems

- All the solutions to the mutual exclusion problem studied assume presence of shared memory
  - Ex. Semaphores, monitors, etc. all rely on shared variables
- The mutual exclusion problem is complicated in distributed system by
  - lack of shared memory
  - lack of a common physical clock
  - unpredictable communication delays
- Several algorithms have been proposed to solve this problem with different performance trade-offs
  - Token-based solutions
  - Permission-based solutions

A centralized algorithm

- A simple solution to the distributed mutual exclusion problem:
  - a single *control site* in charge of granting permissions to access the resource
  - require 3 messages
  - time to grant a new permission is $2T$ (T = average message delay)
- This solution has drawbacks:
  - existence of a single point of failure
  - control site is a bottleneck

Lamport’s Algorithm

- Assumption: messages delivered in FIFO order (no loss messages)
- Requesting the CS
  - P_i sends message **REQUEST(t_i, i)** to other processes, then enqueues the request in its own *request_queue*$_i$
  - when P_j receives a request from P_i, it returns a timestamped **REPLY** to P_i and places the request in *request_queue*$_j$
  - *request_queue* is ordered according to **(t_i, i)**
- A process P_i executes the CS only when:
  - P_i has received a message with timestamp larger than t_i from all other processes
  - its own request in the first of the *request_queue*$_i$

Lamport's Algorithm (2)

- Releasing the critical section:
  - when done, a process remove its request from the queue and sends a timestamped **RELEASE** message to all
  - upon receiving a RELEASE message from P_i, a process removes P_i' s request from the request queue

Lamport's Algorithm Example