# coordination

## time, clocks, event ordering

## physical / logical clocks

### physical clocks

- clock drift: clocks tick at different rates
- clock skew: difference between two clocks at one point of time
- **NTP (network time protocol)**: synchronise local clocks (no global time)
- limitation:

  no absolute synchronisation, just estimation;

  clock skew can't be too large

### logical clocks

- assign **sequence numbers** to events
- **happened-before, concurrent**

  $a \rightarrow b$ event $a$ happened before event $b$

  e.g. $a$ is message being sent; $b$ is receipt of the message

  Transitive relationship:

  If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

  - If $a$ and $b$ occur on different processes that do not exchange messages, then neither $a \rightarrow b$ nor $b \rightarrow a$ are true
    - These events are said to be **concurrent**
    - Written as $a \parallel b$

  partial order 偏序关系，有些元素之间是不能比的，即concurrent (do not affect each other)

- lamport clocks

**on** initialisation **do**
    $t := 0$    ▷ each node has its own local variable $t$
**end on**

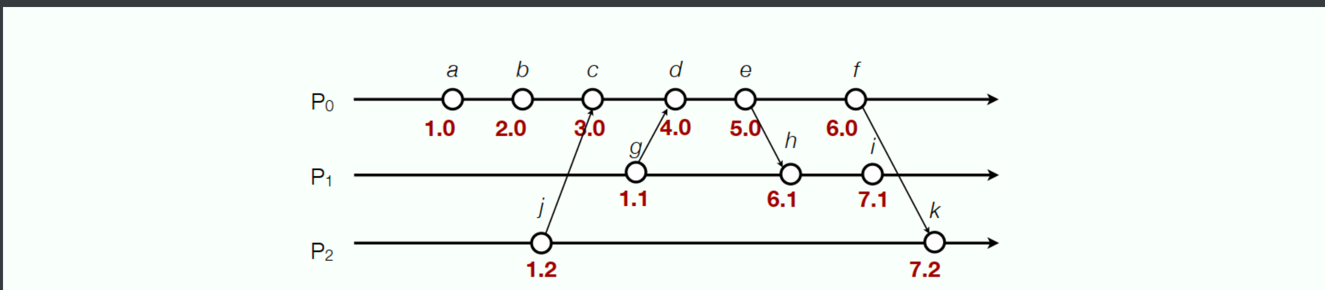
**on** any event occurring at the local node **do**
    $t := t + 1$
**end on**

**on** request to send message $m$ **do**
    $t := t + 1$; send $(t, m)$ via the underlying network link
**end on**

**on** receiving $(t', m)$ via the underlying network link **do**
    $t := \max(t, t') + 1$
    deliver $m$ to the application
**end on**

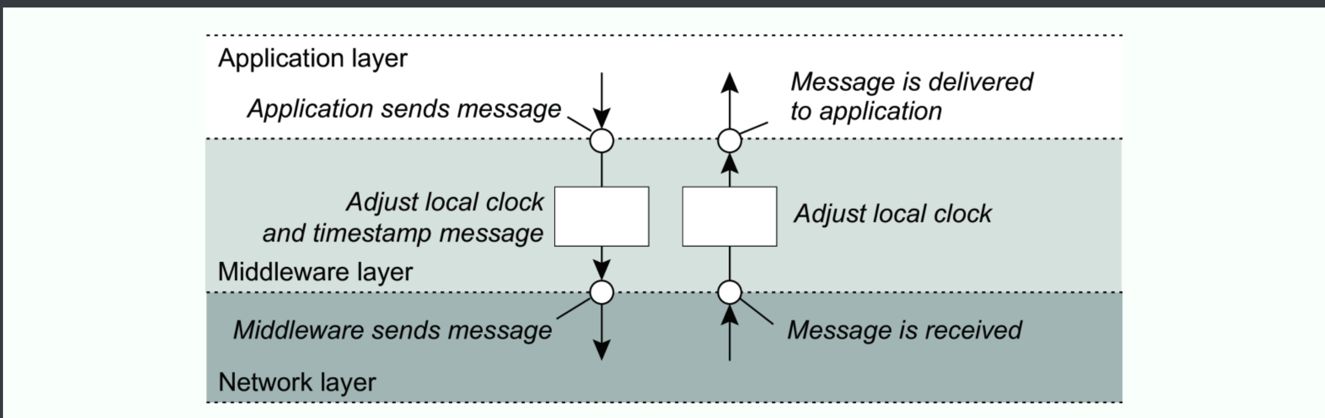
每个process的local counter都单调递增，发送和接受的消息都带有时间戳，收到信息要将当前 local clock 更新到和这个信息相同的时间戳；

- unique logical timestamps (total ordering)




1.0 means $T_i = 1$, $i = 0$

- logical clocks in middleware




## vector clocks

- <u>motivation</u>: **lamport clocks do not capture causality**

L(e) < L(e') does not mean e -> e'

- algorithm in pseudo code

**on** initialisation at process $P_i$ **do**
  $V := \langle 0, 0, \ldots, 0 \rangle$ // *V is a local variable at process* $P_i$
**end on**

**on** any event occurring at process $P_i$ **do**
  $V[i] := V[i] + 1$
**end on**

**on** request to send message $m$ at process $P_i$ **do**
  $V[i] := V[i] + 1$;
  send $(V, m)$ via network
**end on**

**on** receiving $(V', m)$ at process $P_i$ via the network **do**
  $V[j] := \max(V[j], V'[j])$ for every $j \in \{1, \ldots, n\}$
  $V[i] := V[i] + 1$;
  deliver $m$ to the application
**end on**

V[i] = # events that process Pi has timestamped

V[j] (j ≠ i) = # events that have occurred at Pj that have potentially affected Pi

- comparing vector timestamps

• Define:
$V = V' \Leftrightarrow V[i] = V'[i]$ for i = 1 ... N
$V \le V' \Leftrightarrow V[i] \le V'[i]$ for i = 1 ... N
$V < V' \Leftrightarrow V \le V'$ and $V \ne V'$

= 完全相同

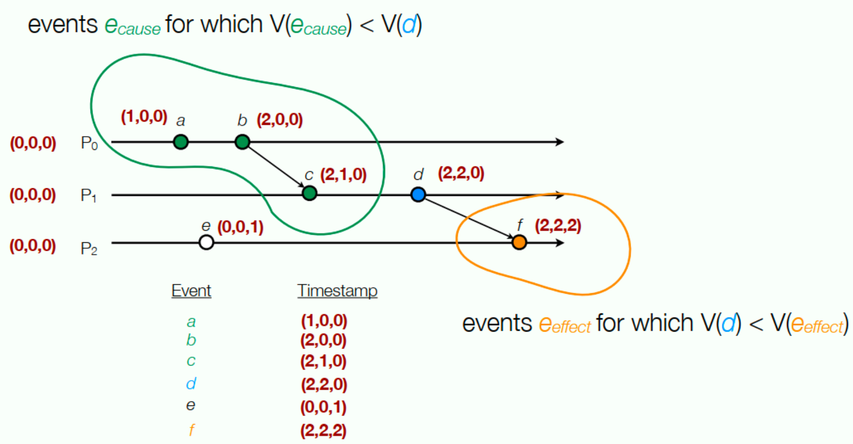
<= 每个元素都小于等于

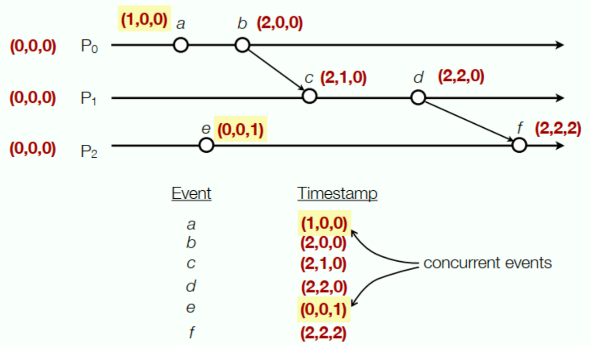
< 有的元素小于，有的元素等于

**if V(e) < V(e') then e -> e'**

**if neither V(e) <= V(e') nor V(e') <= V(e) then e || e' (concurrent)**

- example






Two events e and e’ are **concurrent** if neither V(e) ≤ V(e’) nor V(e’) ≤ V(e)

- applications

  useful for **causally ordered multicast** (a message is delivered only if all messages that may have causally proceded it have been received and delivered as well)

## distributed mutual exclusion

### what is distributed mutual exclusion

- distributed clients, exclusive access to a resource
- different from multi-thread

  no shared memory, no shared clock
- properties

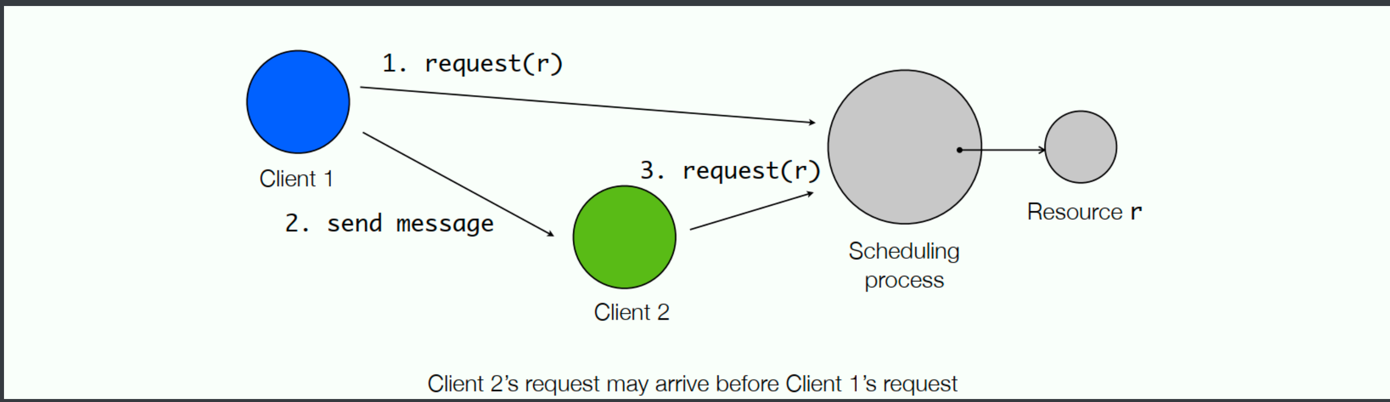
  safety: critical section

  liveness: no livelocks / deadlocks

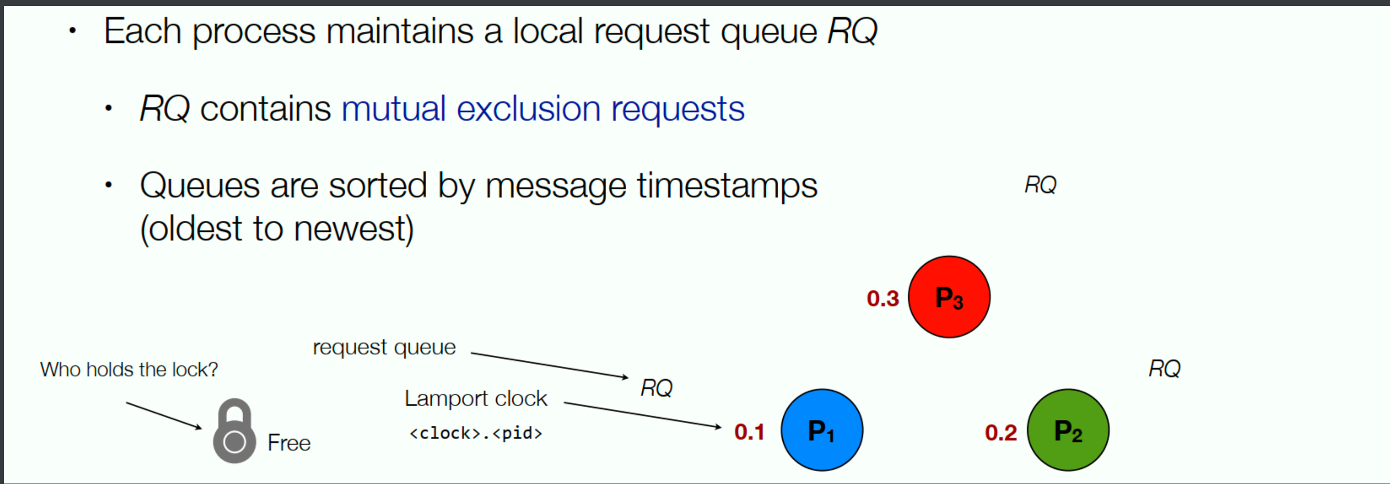
  fairness: requests are granted in happend-before order

### central coordinator: a scheduling process

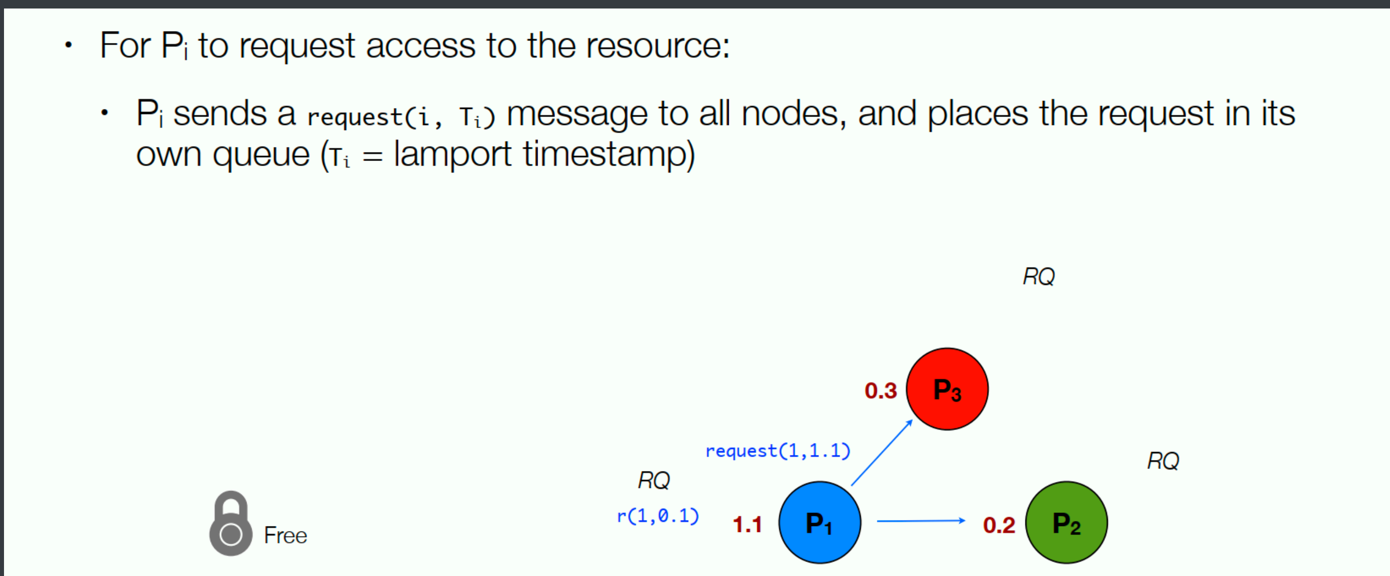
problem: wouldn't reserve the order of the requests with FIFO




## lamport's mutual exclusion alg.: no scheduling process

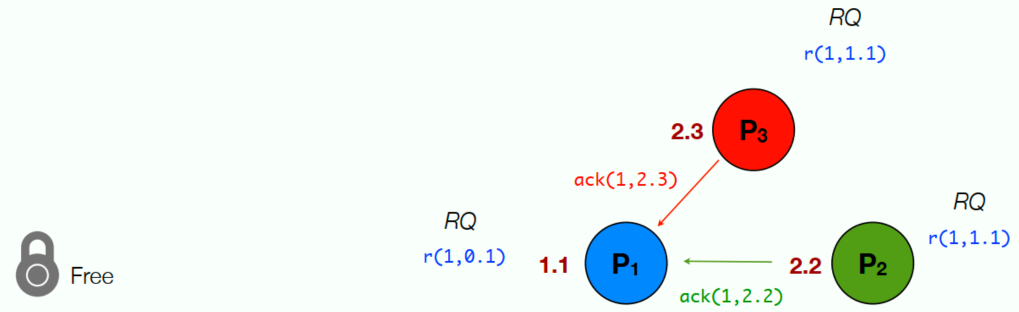



`request` :

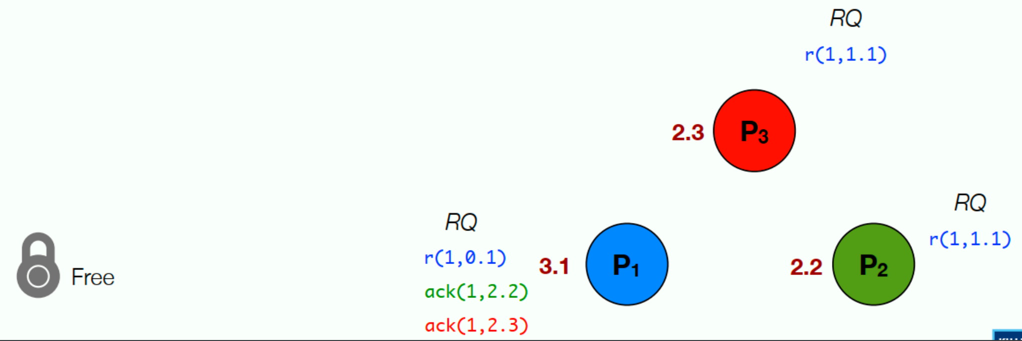



`ack` :

- When a process $P_j$ receives such a request:
  - It replies immediately with a timestamped ack message and places the request in its queue

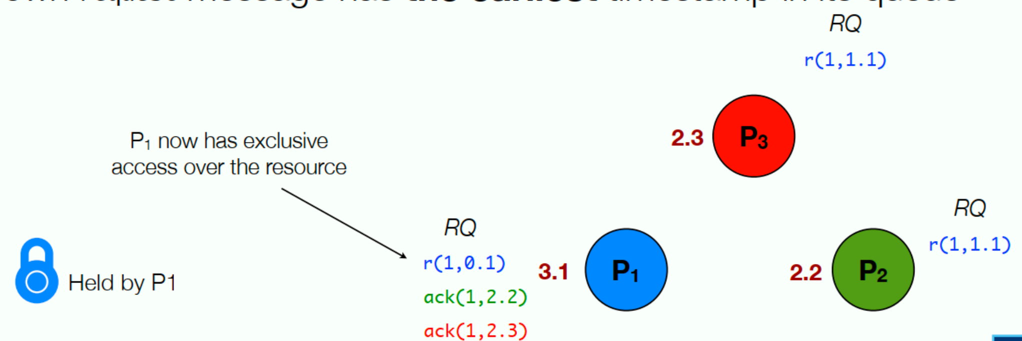



- When a process $P_j$ receives such a request:
  - It replies immediately with a timestamped ack message and places the request in its queue



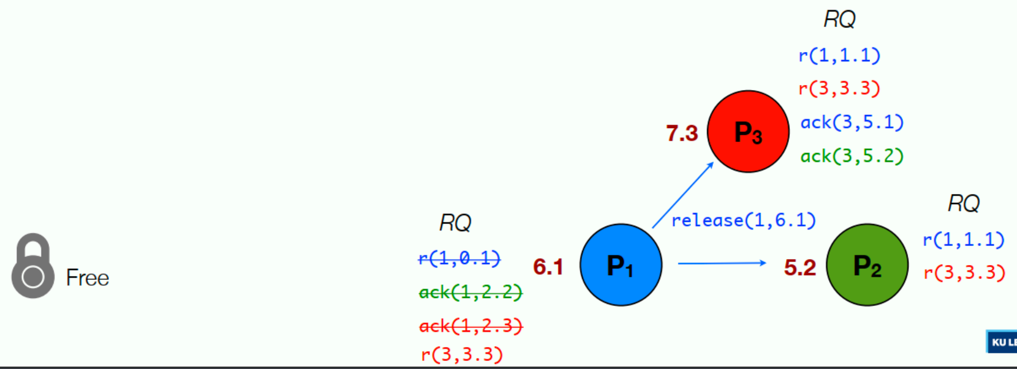

`acquire`:

- For $P_i$ to acquire the resource (enter the critical section), two conditions must hold:
  - $P_i$ has received **all** replies to its `request` message
  - $P_i$'s own `request` message has **the earliest** timestamp in its queue

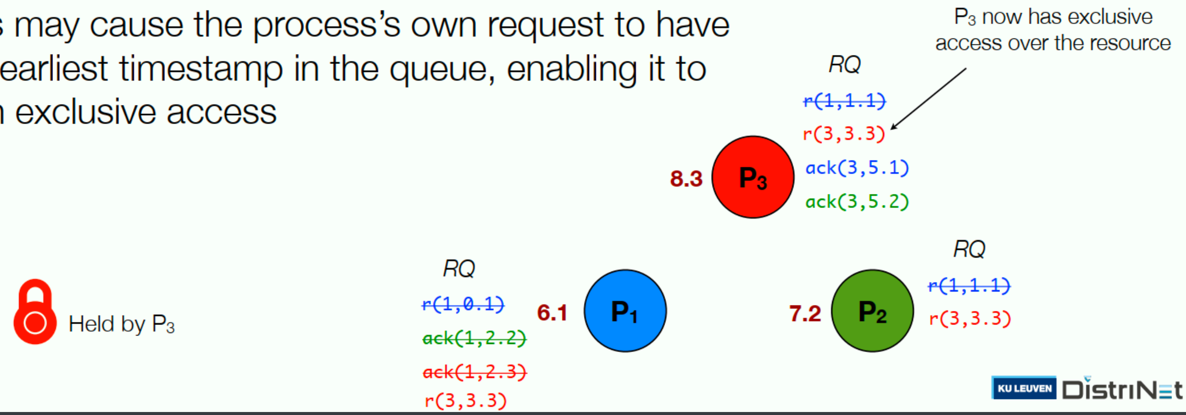


`release` :

- To release the resource (exit the critical section):
  - Remove own request from own request queue
  - Send a timestamped `release(i,Tᵢ)` message




- When a process receives a `release(i,Tᵢ)` message:
  - Remove the previous `request(i,_)` message for that process from local request queue
  - This may cause the process's own request to have the earliest timestamp in the queue, enabling it to gain exclusive access




- problems
  1. failure (if one process crashes, no other process can acquire access to the resource anymore, no ack)

     this is a disadvantage compared to centralized coordinator
  2. a lot of messaging traffic

     - Requests: ($N$-1) requests + ($N$-1) ack replies = 2($N$-1) messages
     - Releases: ($N$-1) release messages

     to reduce this: **ricart & agrawala's alg.**

lock process does not reply to a request until it releases

no explicit release then

$N$-$1$ requests + $N$-$1$ replies + $0$ releases = 2($N$-1) messages

## mutual exclusion in practice

Apache Zookeeper