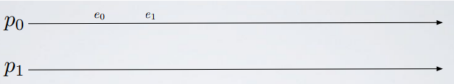
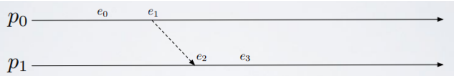
4. LOGICAL CLOCK

In a Distributed System, each system has its own **logical clock**.
If clocks are not aligned it is not possible to order events generated by different processes

GOAL: find a way to timestamp event that follows out intuitive notion of causality

CAUSAL RELATIONSHIP

1. two events occurred at sine oricess p_i happened in the same order as p_i observes them


2. when p_i sensa a message to p_j, the send event happens before the recieve event:


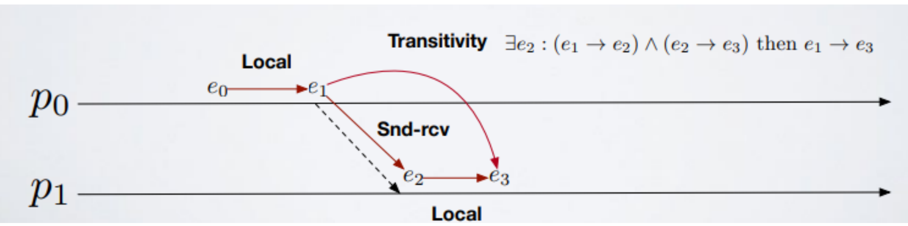
   Lamport introduced the happened-before relation that capture the causal deoendencies between events (**causal order relation**):

- we denote with $\rightarrow_i$ the ordering relation between events in a process p_i.
- we denote with → the happened-before relation between any pair of events.

Happened-Beore RELATION:

Two event e and e' related by happened-before relation ($e \rightarrow e'$) if:

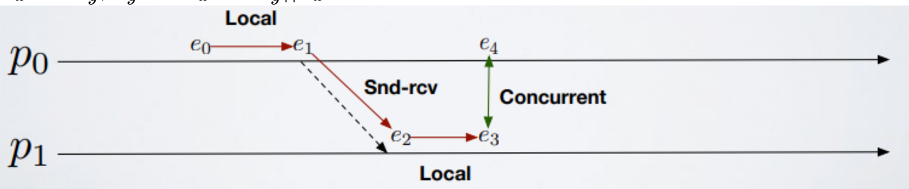
- **Local ordering**: $\exists p_i | e \rightarrow_i e'$
- **snd-rcv ordering**: $\forall m, send(m) \rightarrow recive(m)$
  - $e = send(m)$ is the event of sending a message m.
  - $e' = recive(m)$ is the event of recepit of the same message m
- **Transativity**: $\exists e'' : (e \rightarrow e'') \wedge (e'' \rightarrow e') \Rightarrow e \rightarrow e'$
  - the happened-before relation is transitive

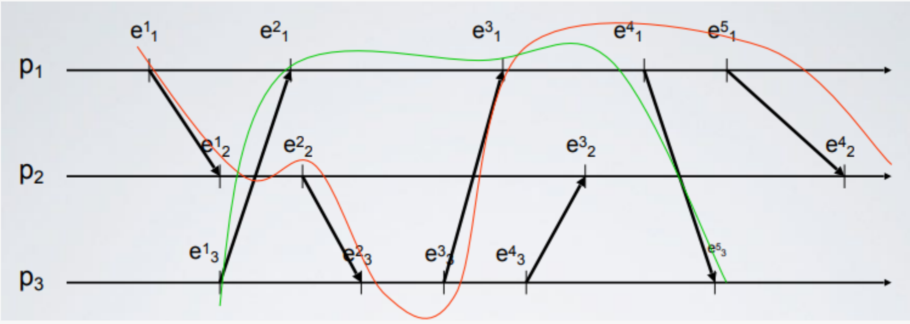


Applying these three rules is possible to define a causal ordered sequence of events $e_1, e_2, \ldots, e_n$.
Notes:

- the sequence $e_1, e_2, \ldots, e_n$ **may not be unique**.
- it **may exists a couple of events such that** e_1 **and** e_2 **are not in happenedbefore relation.**
- **if** e_4 **and** e_3 **are not in happened-before relation then they are concurrent** $(e_4 || e_3)$.

- for any two events e_x and e_y in the execution history of a distributed system, either $e_x \rightarrow e_y, e_y \rightarrow e_x$ or $e_y || e_x$.

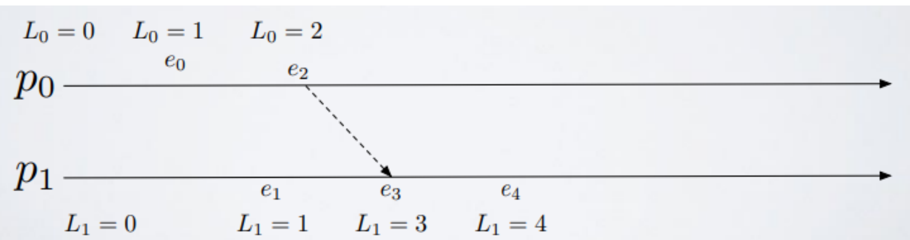


> **EXAMPLE**:
>
> 
>
>
> $S_1 =< e^1_1, e^1_2, e^2_2, e^2_3, e^3_3, e^3_1, e^4_1, e^5_1, e^4_2 >$
> $S_2 =< e^1_3, e^2_1, e^3_1, e^4_1, e^5_3 >$
> Concurrent Elements: (e^1_3, e^1_2)

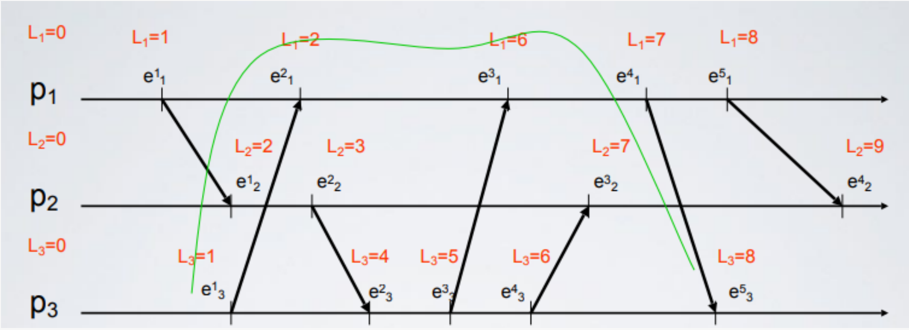
Logical/Lamport/Scalar Clock: monotonically increasing software counting register (not related to physical clock).
Each process p_i emplys its logical clock L_i to apply a timestamp to events.
$L_i(e)$ is the **logical timestamp** assigned, using the logical clock, by a process p_i to event e.
Property:
if $e \rightarrow e'$ then $L(e) < L(e')$

Each process p_i initializes its logical clock $L_i = 0$

- when p_i sends a message m:
  - creates an event $send(m)$
  - increases L_i
  - timestamps m with $t = L_i$
- when p_i recives a message m with timestamp t
  - updates its logical clock $L_i = max(t, L_i)$
  - produces an event $recive(m)$
  - increases L_i



because of the property (if $e \to e'$ then $L(e) < L(e')$)

> **Example**:
>
> 
>

Limits of Scalar Logical Clock:
Scalar Logical clocks can guarantee the property

- if $e \to e'$ then $L(e) < L(e')$

But it is **not possible to guarantee**:

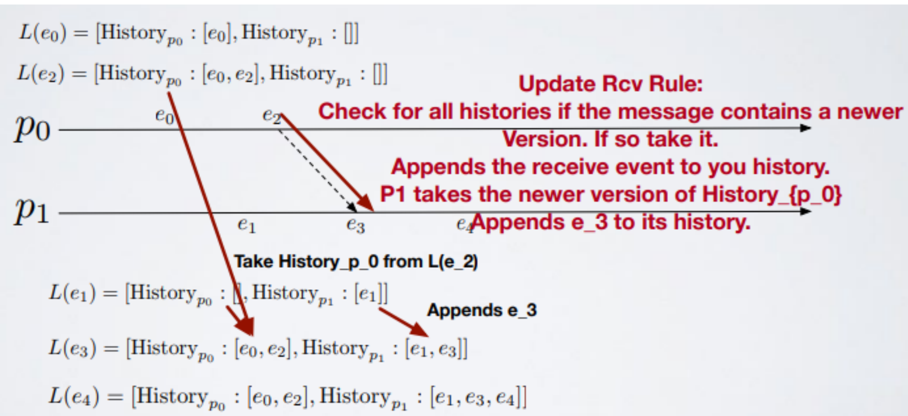
- if $L(e) < L(e')$ then $e \to e'$ IS not true everytime

So it is **not possible to determine**, analyzing only scalar clocks, **if two events are concurrent or correlated by the happened-before relation**.

VECTOR CLOCK

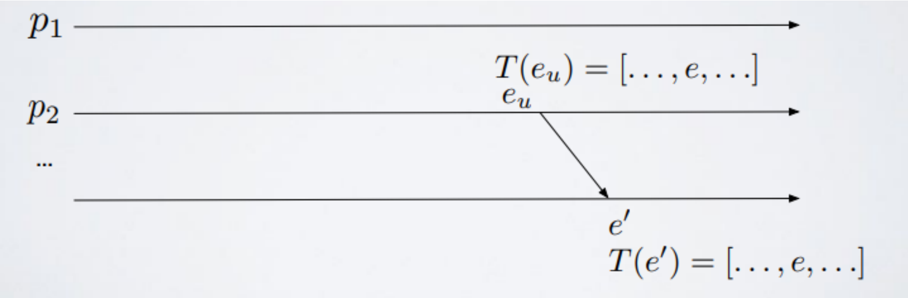
GOAL: capture causality (if $L(e) < L(e')$ then $e \to e'$)

$L(e)$ has not to be a single number. what if $L(e)$ is a history of events that happened before e (including e)?



$$L(e_i) > L(e_j) \Leftrightarrow \forall k : L(e_j)_{History_k} \subseteq L(e_i)_{History_k} \wedge \exists x : L(e_j)_{History_x} \subset L(e_i)_{History_x}$$

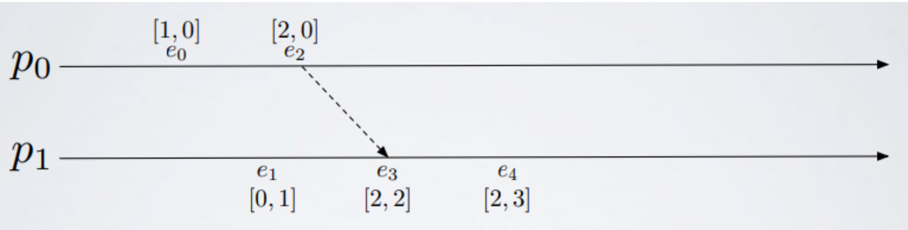
(there is a causal path form $e \to e'$ and the structure that contains e is less than the structure that contains e')



$History_x \subset History'_x \rightarrow History_x$ is a proper prefix of $History'_x$

We can say that:

$$History_x \subset History'_x \rightarrow len(History_x) < len(History'_x)$$



> An event e is in happened-before relation with an event e' if in his $History$ there is a tuple of elements that $\subseteq$ and a tuple that is strictly $\subset$.

> A vector clock for a set of N processes is an array of N integer counters:

- Each process p_i maintains a vector clock V_i and timestamps events by mean of it.
- Similarly to scalar clock, a vector clock is attached to message m (in this case we attach an array of integer).

Implementation:

- each process p_i **initializes its clock** $V_i = 0$
- p_i **increases** $V_i[i] + 1$ **when it generates a new event** e.
- when p_i **sends a message** m then:
  - creates an event $send(m)$.
  - $V_i[i] + 1$.
  - timestamps m with $t = V_i$.
- when p_i **recives a message** m containing timestamp $V - t$ then:
  - updates its logical clock: $V_i[j] = max(V_t[j], V_i[j] \;\; \forall j \in \{1, \dots, N\})$.
  - generates an event $recive(m)$.
  - increases V_i.

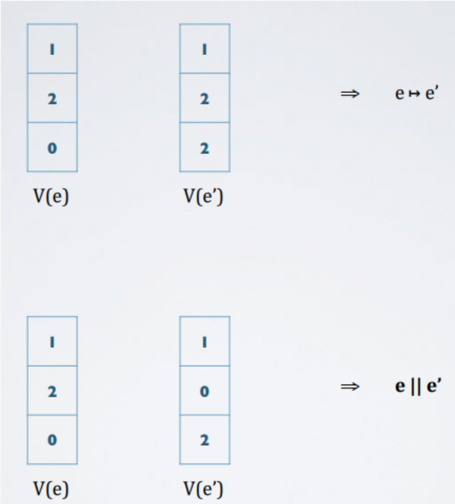
where:

- $V_i[i]$ represents the number of events produced by p_i.
- $V_i[j]$ with $i \neq j$ represents the number of events genereted by p_j that p_i knows.

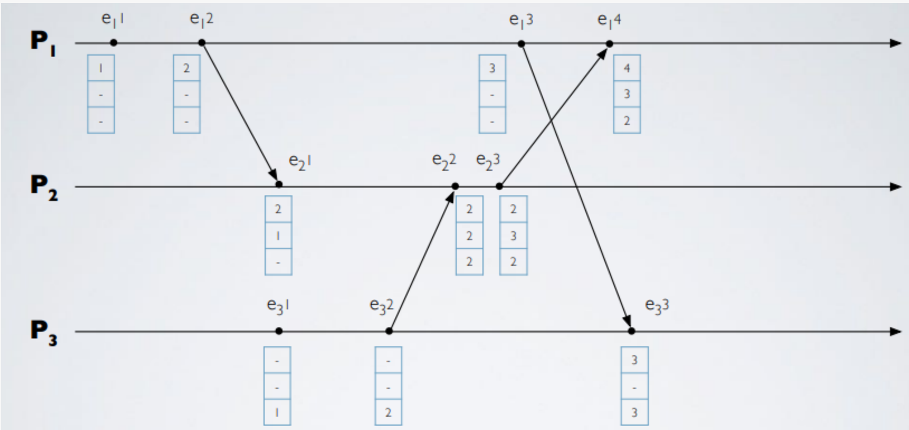
Properties:

- $V = V' \Leftrightarrow V[j] = V'[j] \;\; \forall j \in \{1, .., N\}$
- $V \leq V' \Leftrightarrow V[j] \leq V'[j] \;\; \forall j \in \{1, .., N\}$
- $V < V' \Leftrightarrow V \leq V' \wedge \; \exists \, j \in \{1, ..., N\} | V[j] < V'[j]$

Example 1:



Example 2:



$[-, -, 1] < [4, 3, 2] \Rightarrow e_3^1 \rightarrow e_1^4$
$[4, 3, 2] ? [3, -, 3] \Rightarrow e_1^4 || e_3^3$

Each mechanism can be used to solve different problems:

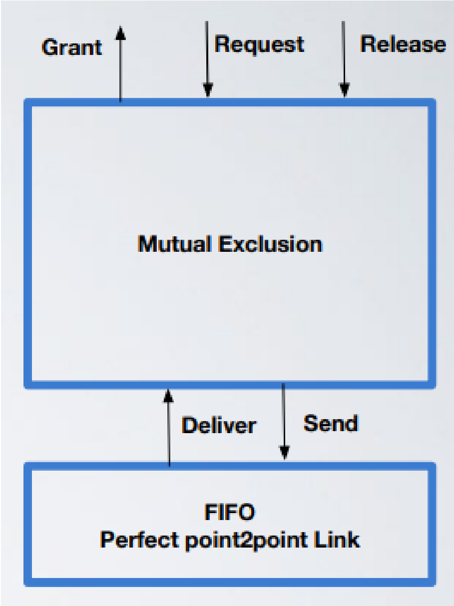
- Scalar Timestamp → Lamport's Mutual Exclusion
- Vector Timestamp → Causal Broadcast

MUTUAL EXCLUSION ABSTRACTION

Events:

- **Request**: from upper layer - **requests access to Critical Section** (CS).
- **Grant**: to upper layer - **grant the access to CS**.
- **Release**: from upper layer - **release the CS**.
Properties:
- (**Mutual Exclusions**) at any time t, only one process is inside the CS.

- (**Liveness**) if a process p requests access, then it eventually enters the CS.
- (**Fairness**) if the request of process p happens before the request of process q, then q cannot access the CS before p.

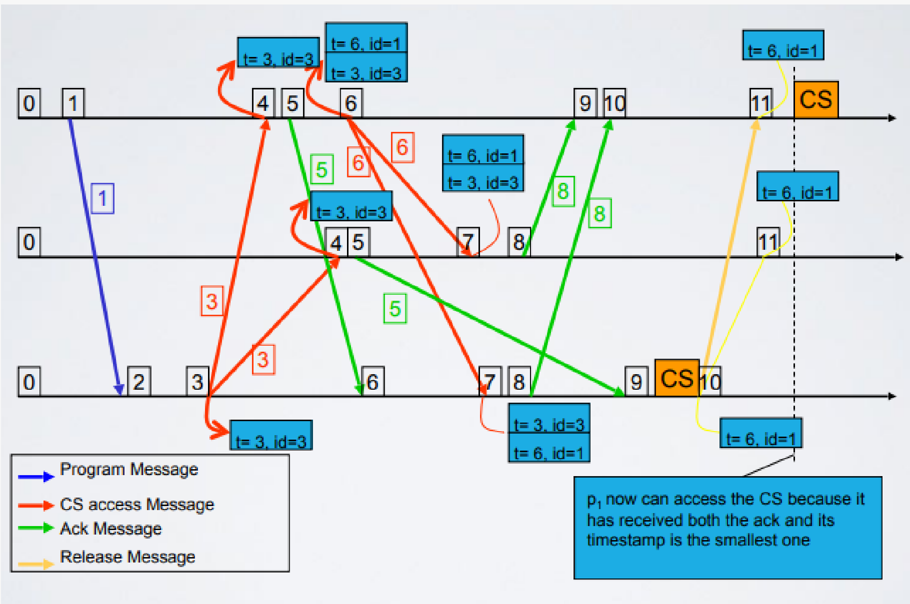


The algprithm assumes **no crashes** ($F = 0$):
when a process wants to enter the CS (critical section) it sends a request message to all the oteher (using **scalar clocks**). The algorithm assume a FIFO link.

> **Lamport's Algorithm**:
>
> **Algorithm 1** Lamport's ME Algorithm on process p_i - MSGS are REQ, ACK, RLS
>
> **upon event** INIT
> $Requests = Acks = \emptyset$
> $scalar\_clock = 0$
> $my\_req = \bot$
> $\Pi = \{p_0, p_1, \ldots, p_{n-1}\}$ ▷ Set of all processes
>
> ▷ Request access to CS from upper layer
> **upon event** REQUEST
> $scalar\_clock = scalar\_clock + 1$
> $my\_req = (REQ, ts =< i, scalar\_clock >)$
>   **for all** $p_j \in \Pi$ **do**
>   SEND FIFOPERFECTLINK(p_j, req_msg) ▷ Send a REQ containing my ID (i) and ts (scalar_clock) to all $p \in \Pi$
>
> ▷ Release CS from upper layer
> **upon event** RELEASE
> $my\_req = \bot$
> $scalar\_clock = scalar\_clock + 1$
>   **for all** $p_j \in \Pi$ **do**
>   SEND FIFOPERFECTLINK($p_j, (RLS, ts =< i, scalar\_clock >)$)
>
> ▷ $ts(x) < ts(y)$ when $scalar\_clock$ of x is less than the one of y, or they are equal and the id that sent x is less than the id that sent y
> **upon event** $\nexists req \in Requests : ts(req) < ts(my\_req) \wedge \forall p \in \Pi : \exists m \in Acks | ts(m) > ts(my\_req) \wedge sender(m) = p$
>   **trigger event** GRANTED
>
> **upon event** DELIVER MESSAGE(m)
> $scalar\_clock = max(clock(m), scalar\_clock) + 1$
>   **if** m is a REQ **then**
> $Request\_set = Request\_set \cup \{m\}$
> $scalar\_clock = scalar\_clock + 1$
>   SEND FIFOPERFECTLINK($sender(m), (ACK, ts =< i, scalar\_clock >)$)
>   **else if** m is a ACK **then**
> $Acks = Acks \cup \{m\}$
>   **else if** m is a RLS $\wedge \exists req \in Request\_set$: sender(req)=sender(m) **then**
> $Requests = Requests \setminus \{req\}$

**Local data structures to each process p_i:

- ck is the counter for process p_i.
- $Request$: a set mainteines by p_i where CS access requests are stored.
  **Algorithm rules for a process p_i:
- **access** the CS:
  - p_i sends a **request message** (attaching ck) to all the other processes.
  - p_i adds its request to $Requests$ structure.
- **request reception** from process p_i:
  - p_i puts p_j request (including the timestamp) in its $Requests$.
  - p_i sends back an ACK message to p_j including its local timestamp ck.
- p_i **enters the CSS iff**:
  - the request of p_i is the one with smallest timestamp in its $Requests$.
  - p_i has already recived an ACK with timestamp t' from any other processes and $t' > t$.
- **release** of the CS:
  - p_i send a $Release$ message to all the other processes.
  - p_i deletes its request from $Requests$.
- **reception of release message** from a process p_i:
  - p_i deletes p_j request from $Requests$.

> **Example**:
>
> 
>

DEMOSTRATION (by **contraddiction**): **why only one can enter the CS**:

1. **Mutual Exclusion**: assume that p_1 and p_2 enter CS at the same time.
   - you cannot enter CS if you have not received acks from everyone, and such acks happened after your request (when you create a new request its ts is greater then the tss of all old acks).
   - $\Rightarrow$ both the process have received an ACK from any other process and each my_req is the smallest in the respective queue:
     - p_i received the ack from p_j. When p_j sends the ack to p_i it inserts in its set the req of p_i.
     - p_j received the ack from p_i. When p_i sends the ack to p_j it inserts in its set the req of p_j.

    - if p_j $Requests$ after acking p_i, then $ts(req, p_j) > ts(req, p_i)$ and that's a **contradiction**. (The same for p_i)
    - So p_j $Requests$ before acking p_i, then by FIFO the (req, p_j) reaches p_i before the ack of p_j. Thus p_i has (req, p_j) in its set and p_j has (req, p_i) in its set. **Since request are total ordered then we have a contradiction**.
2. **Fairness**: different requests are satisfied in the same order as they are generated (such order comes from the happened-before relation).
    - **Proof**: suppose p_i enters before p_j, even if (req, p_i) happened after (req, p_j). Since p_i enters only after the ack of p_j, by FIFO it sees (req, p_j) before receiving the ack that allows him to enter. Since (req, p_j) happens before (req, p_i) we have $ts((req, p_j)) < ts((req, p_i))$, thus (req, p_i) is not the request with minimal timestamp in the set of p_i.

> **Algorithm Cost**:

Number of operation for a CS execution: $3(N-1)$:

- $N-1$ requests.
- $N-1$ acks.
- $N-1$ releases
  Deelay to enter the CS: $2 \leq deelay \leq N+2$:
- $\Omega(2)$
- $O(N+2)$