Causal memory: definitions, implementation, and programming

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, Phillip W. Hutto

Presented By: Natasha Mittal

Motivation

- To provide abstraction of shared memory in distributed computing systems
- Enforcing traditional consistency guarantees leads to latencies that prevent scaling to large systems.

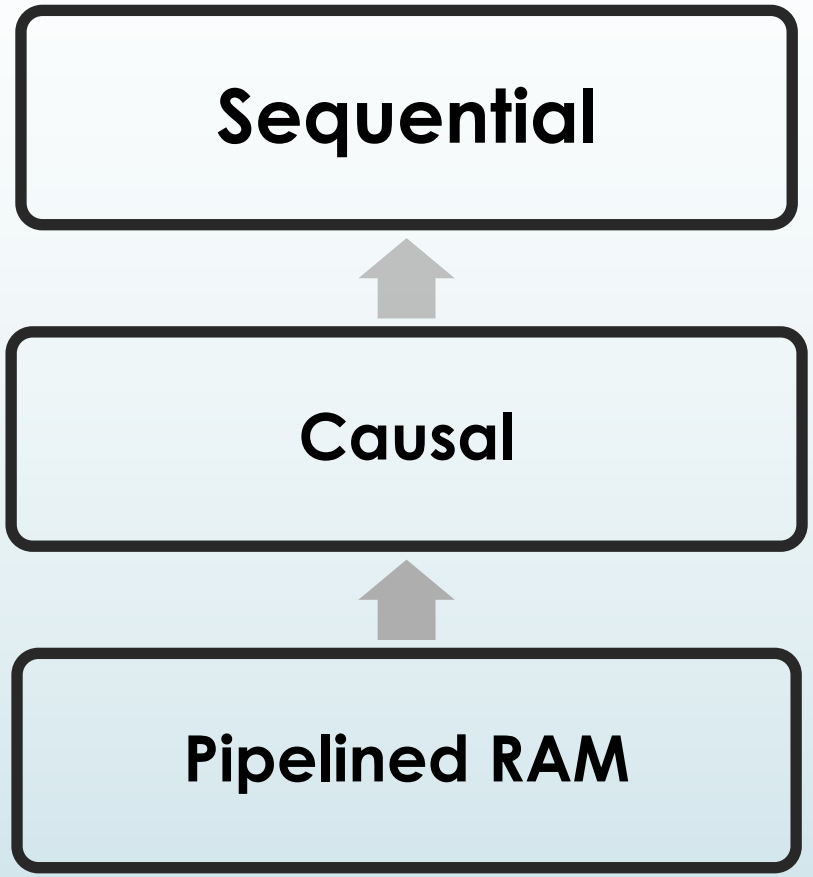
Benefits of Causal Memory

- Provides an abstraction which ensures that processes in system agree on relative ordering of operations that are causally related.
- It is weakly consistent, so it admits more executions than either atomic or sequentially consistent memories.

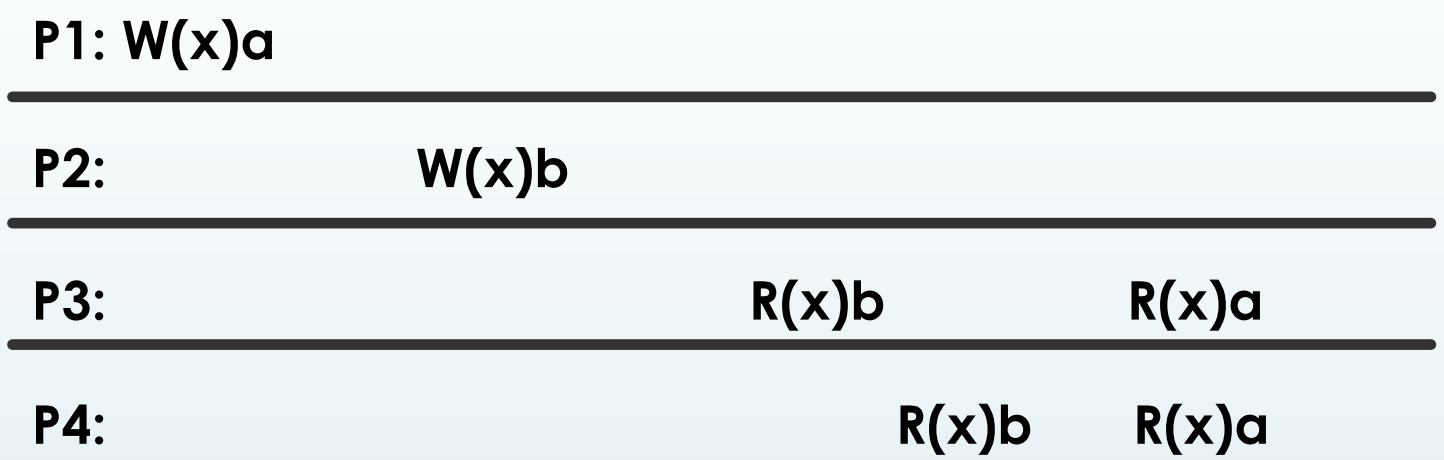
Notations

- $\mathcal{P} = \{p_1, p_2, \ldots\ldots, p_n\}$ be set of processes.
- $w_i(x)v$ denotes a write operation by process p_i which stores value v at location x.
- $r_i(x)v$ denotes a read operation which reports process p_i that value v is stored at location x.
- L_i known as $\boldsymbol{local\ execution\ history}$ denotes sequence of read and write operations of process p_i.
- $H = < L_1, L_2, \ldots\ldots, L_n >$ is known as $\boldsymbol{history}$ which is a collection of histories, one for each process.
- $o_1 \rightarrow o_2$ means that o_1 proceeds o_2

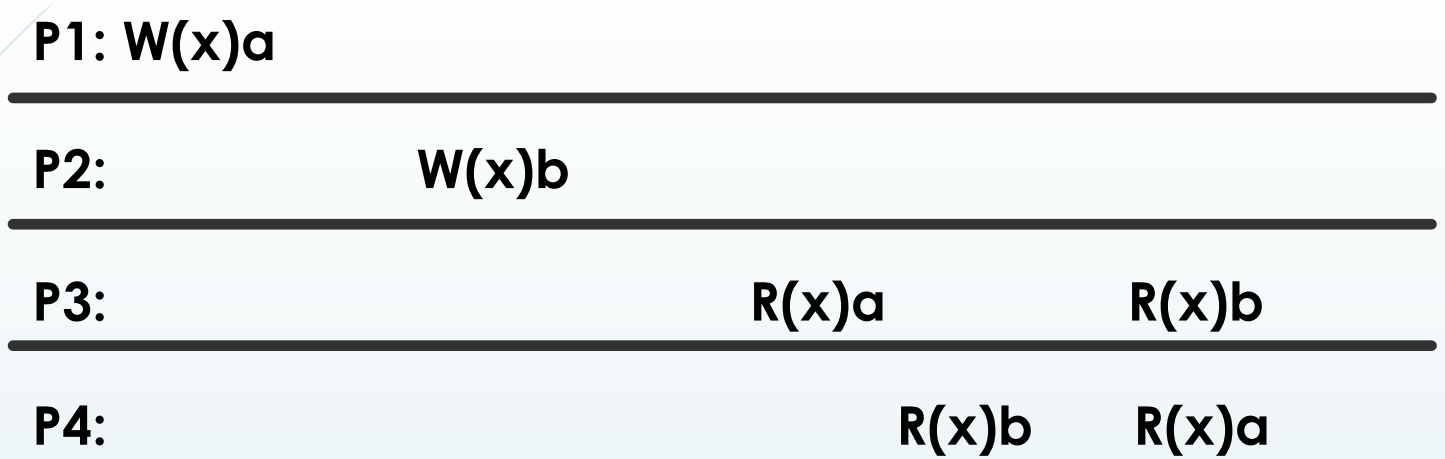
Consistency Models



Sequential Consistency



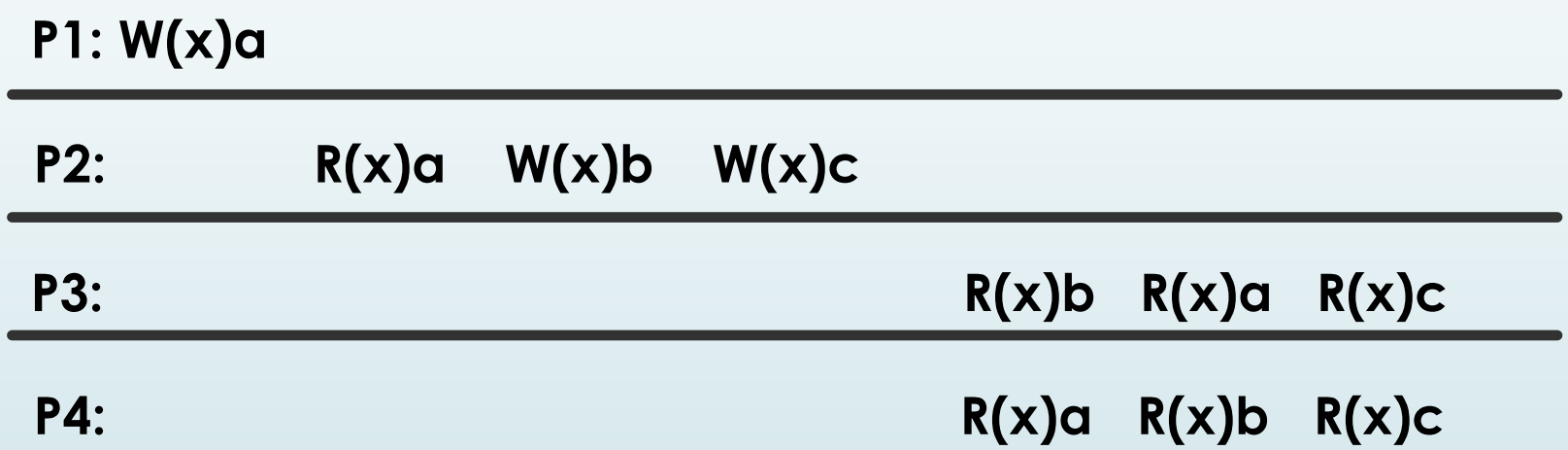
Is it sequentially consistent?



Is it sequentially consistent?

Pipelined RAM Consistency

- Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.



Causal Memory

- Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.
- A causality order of operations in history is determined by program order and a writes into order that associates a write operation with each read operation.
- A writes into order $\mapsto$ on H is any relation with the following properties:
  - If $o_1 \mapsto o_2$, then there are x and v such that $o_1 = w(x)v$ and $o_2 = r(x)v$
  - For any operation o_2, there is at most one o_1 such that $o_1 \mapsto o_2$
  - if $o_2 = r(x)v$ for some x and there is no o_1 such that $o_1 \mapsto o_2$, then $v = \perp$
- $o_1 \rightsquigarrow o_2$ if and only if one of the following cases holds:
  - $o_1 \mapsto o_2$ for some p_i (o_1 proceeds o_2 in L_i)
  - $o_1 \overset{i}{\mapsto} o_2$ (o_2 reads the value written by o_1)

P1: W(x)0	**W(x)1**		
P2:	**R(x)1**	**W(y)2**	
P3:		**R(y)2**	**R(x)0**

$S_1 = w_1(x)0, w_1(x)1, w_2(y)2$

$S_2 = w_1(x)0, w_1(x)1, r_2(x)1, w_2(y)2$

$S_3 = w_2(y)2, r_3(y)2, w_1(x)0, r_3(x)0, w_1(x)1$

A history that is PRAM but not causal

P1: W(x)1				
P2:	**R(x)1**	**W(x)2**		
P3:			**R(x)2**	**R(x)1**
P4:			**R(x)1**	**R(x)2**

Is it Causal?

Is it PRAM?

P1: W(x)1				
P2:	**R(x)1**	**W(x)2**		
P3:			**R(x)2**	**R(x)1**
P4:			**R(x)1**	**R(x)2**

Is it Causal? **NO**

Is it PRAM? **YES**

Logical (or Lamport) Ordering

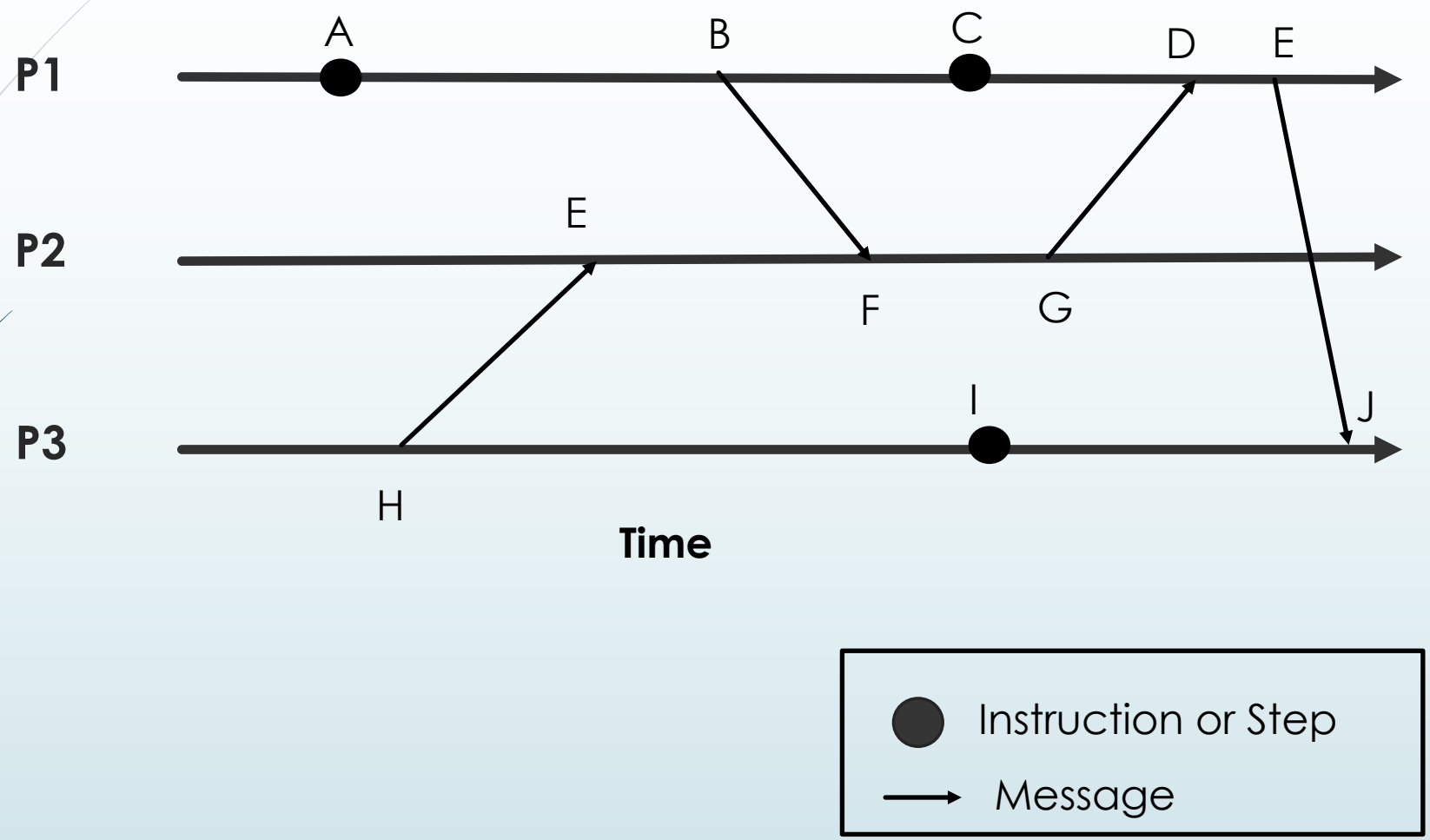
- Define a logical relation $Happens - Before$ among pairs of events
- $Happens - Before$ denoted as $\rightarrow$
- Three rules:
  - On the same process: $a \rightarrow b$, if $time(a) < time(b)$ (using the local clock)
  - If p_1 sends m to p_2: $send(m) \rightarrow receive(m)$
  - (Transitivity) If $a \rightarrow b$ *and* $b \rightarrow c$ then $a \rightarrow c$
  - Creates a *partial order* among events
    - Not all events related to each other via $\rightarrow$

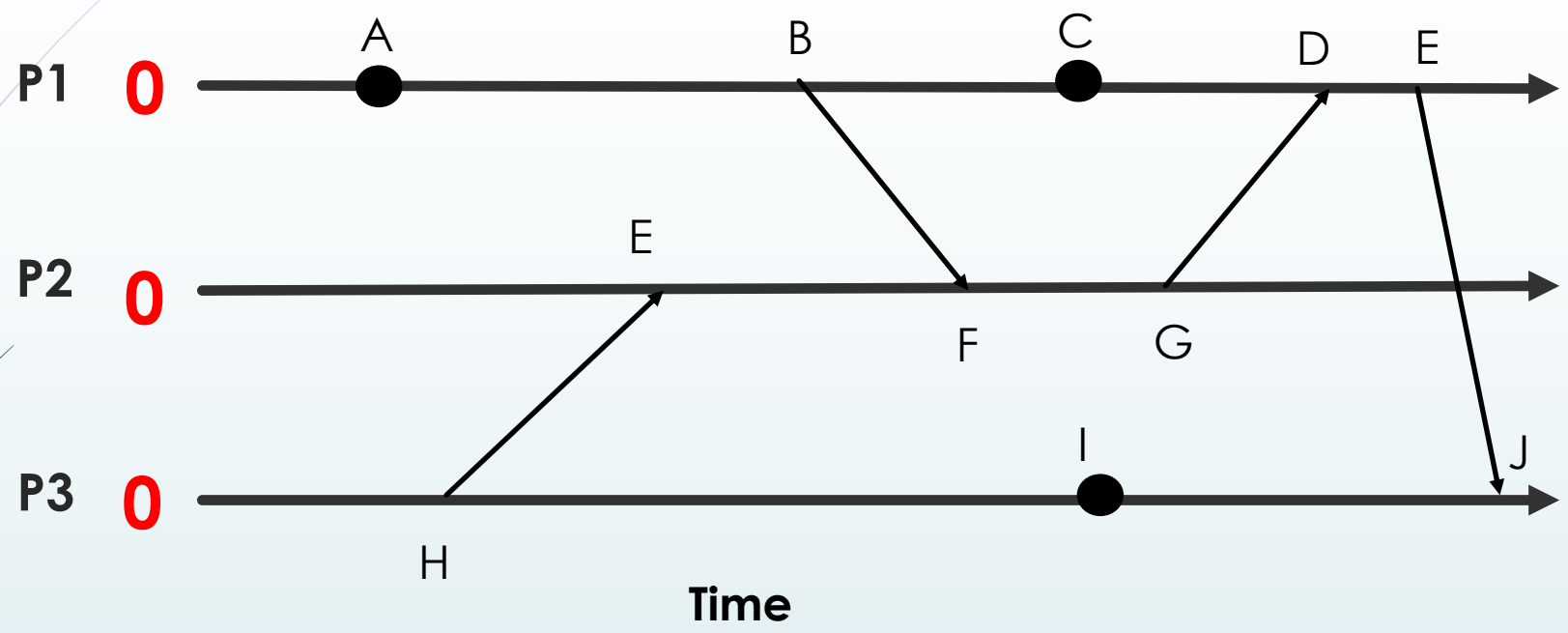
Source: https://www.coursera.org/lecture/cloud-computing/2-4-lamport-timestamps-4nPtE

Lamport Timestamps

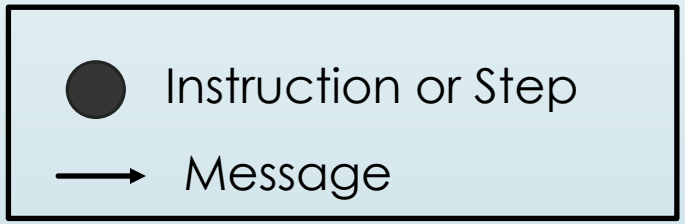
- **Goal: Assign logical (Lamport) timestamp to each event**
- **Timestamps obey causality**
- **Rules**
  - Each process uses a local counter (clock) which is an integer
  - Initial value of counter is zero
  - A process increments its counter when a send or an instruction happens at it
  - The counter is assigned to the event as its timestamp
  - A send (message) event carries its timestamp
  - For a receive (message) event the counter is updated by
    - $\max(local\ clock, message\ timestamp) + 1$

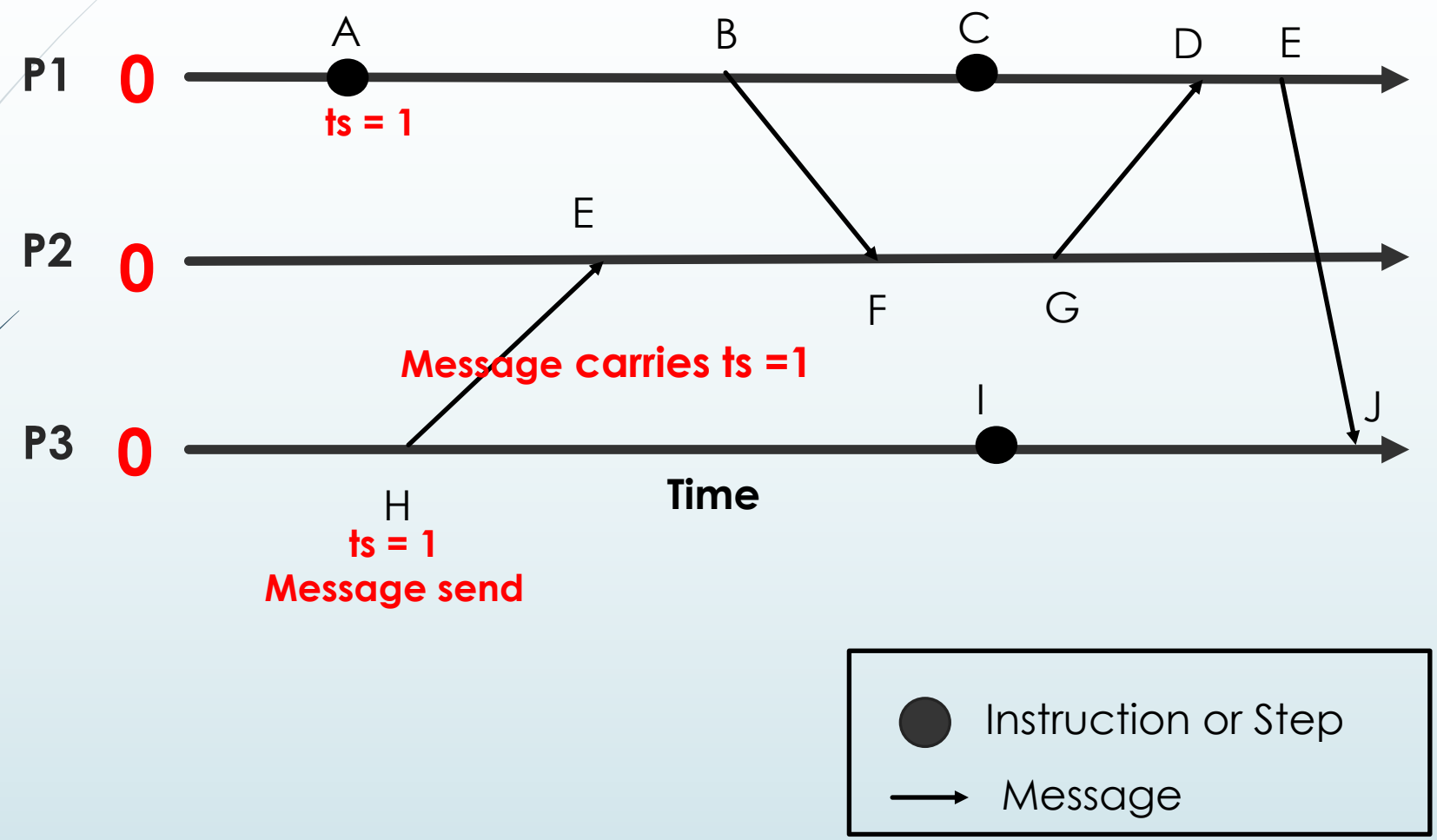
Source: https://www.coursera.org/lecture/cloud-computing/2-4-lamport-timestamps-4nPtE

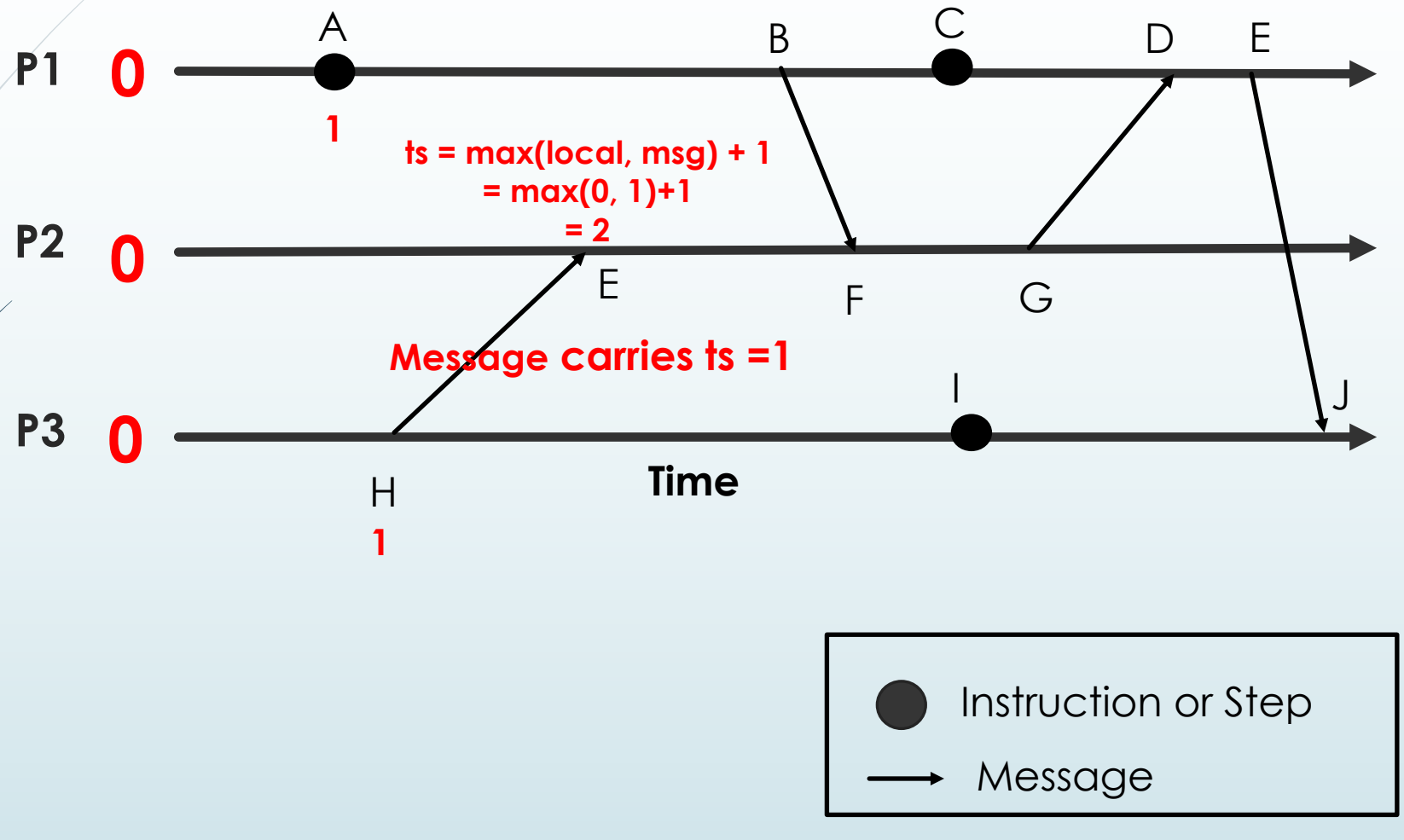

P1
P2
P3
A
B
C
D
E
E
F
G
I
J
H
Time
Instruction or Step
Message

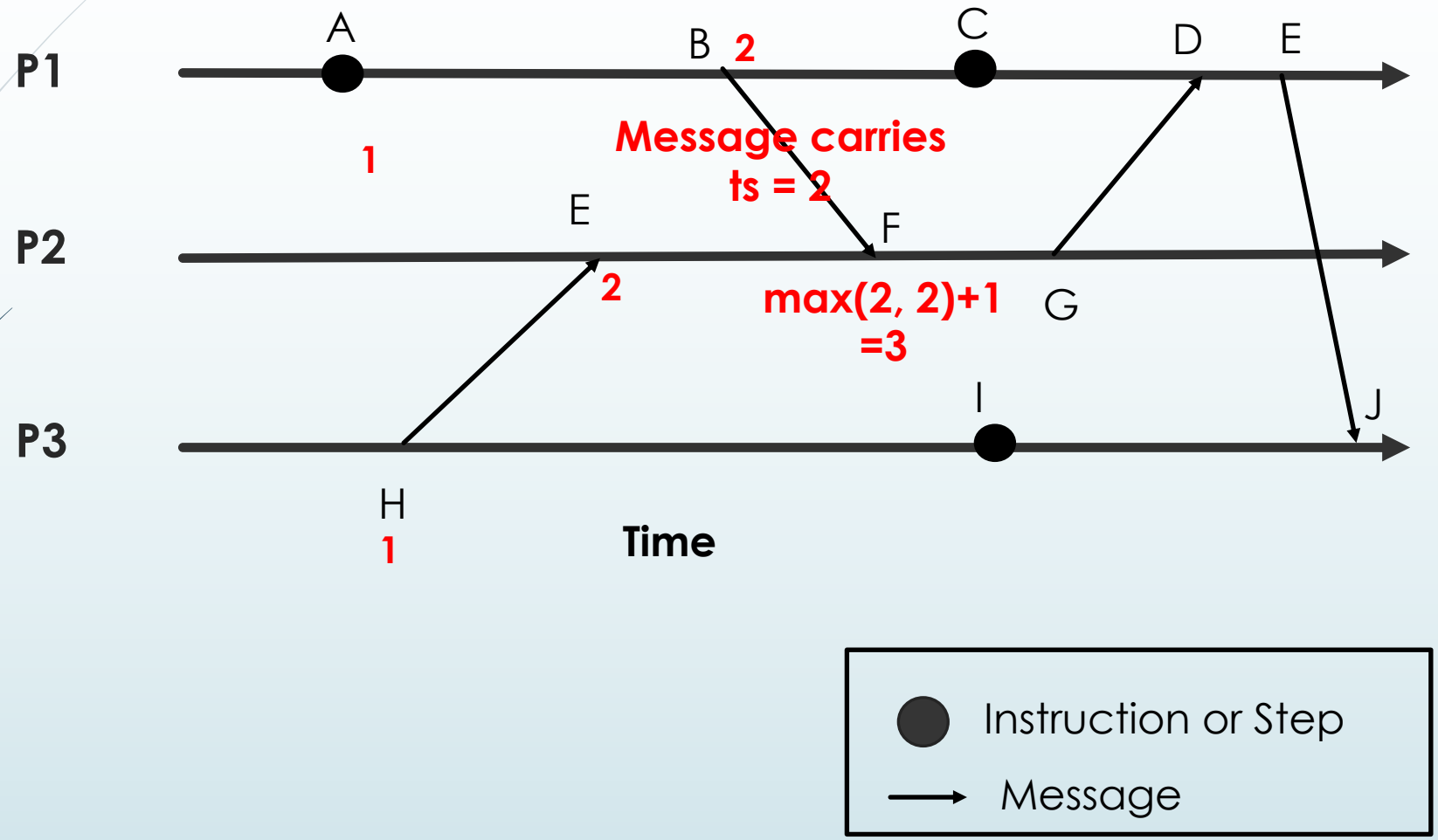


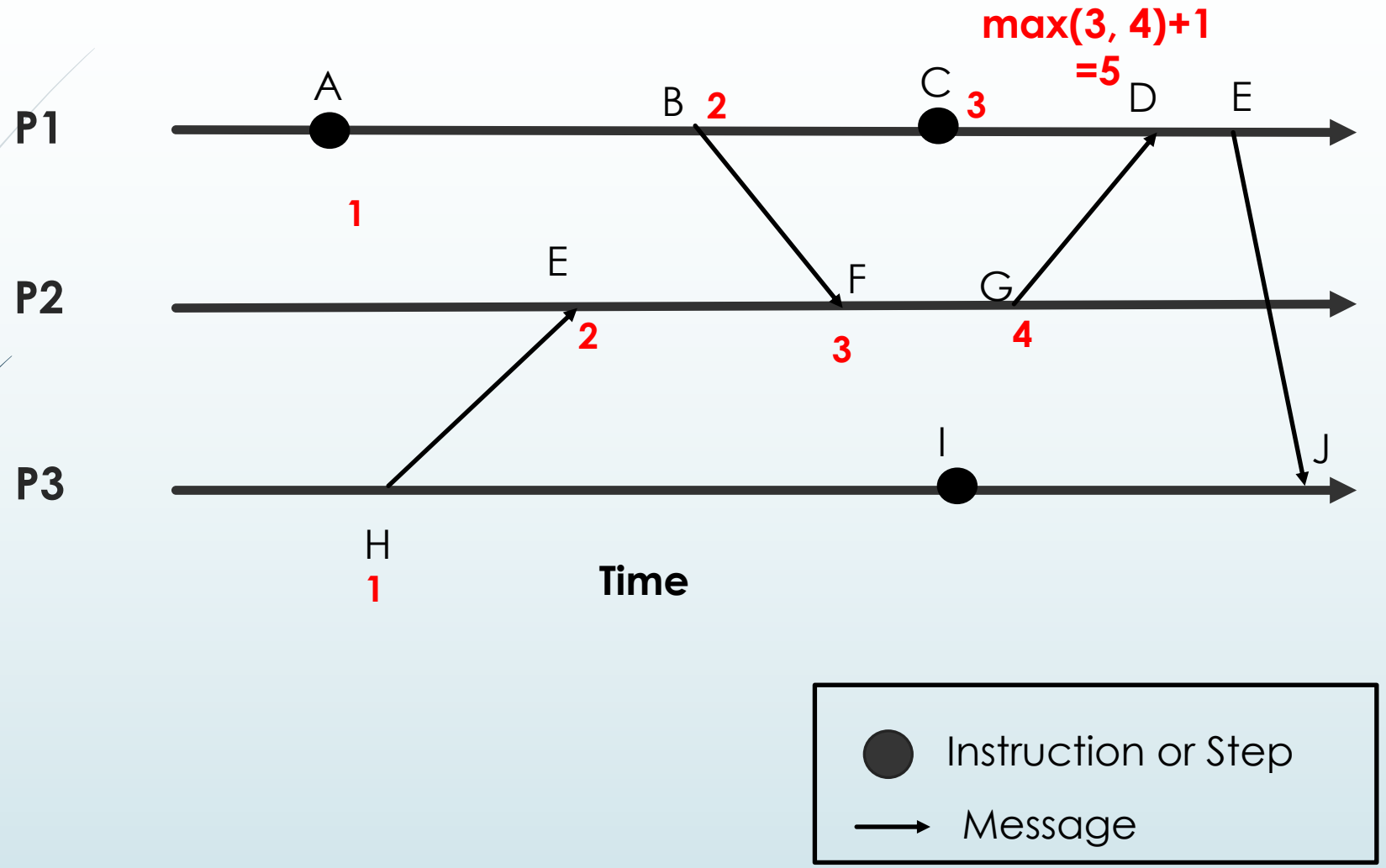
Initial counters

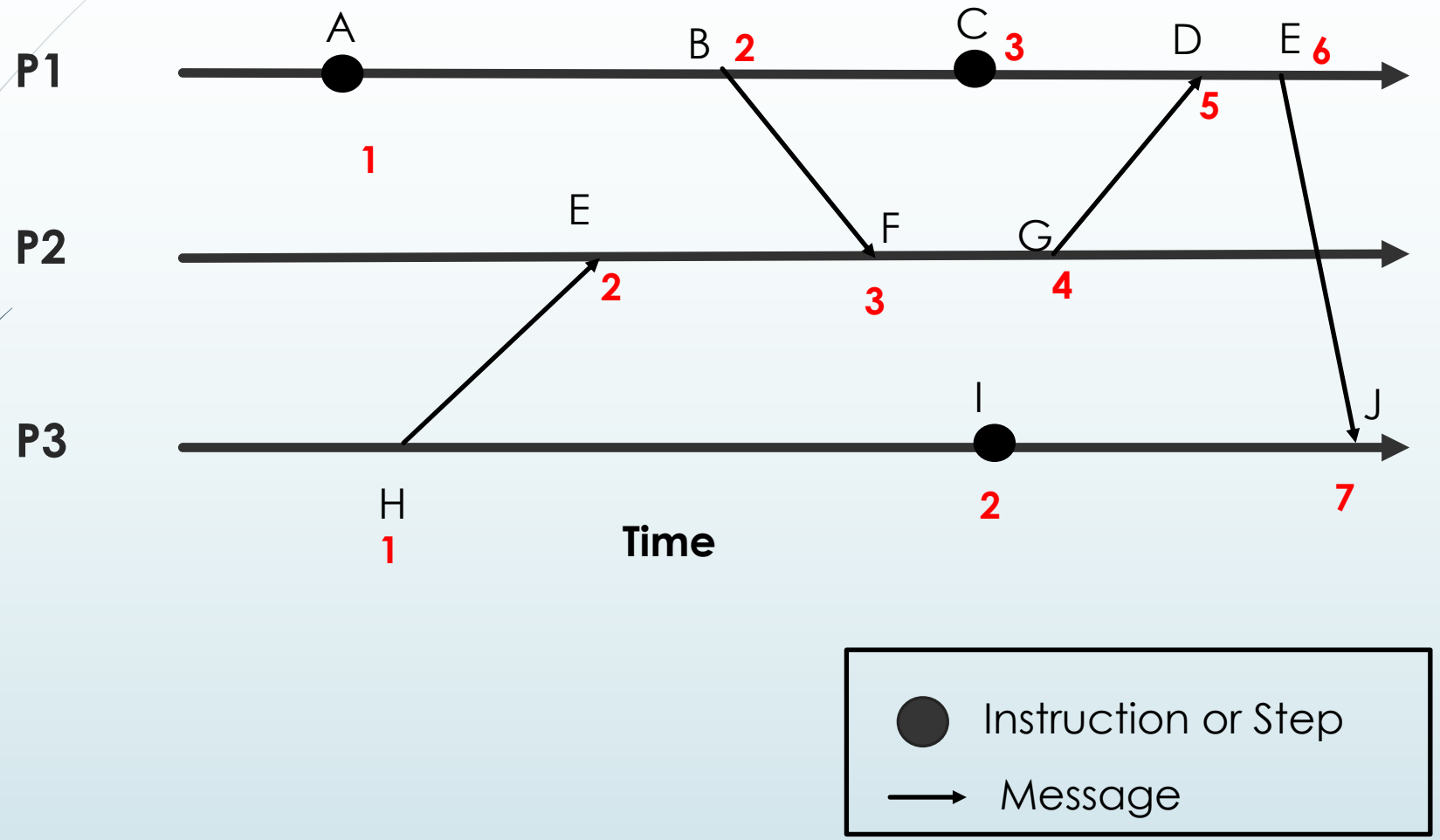


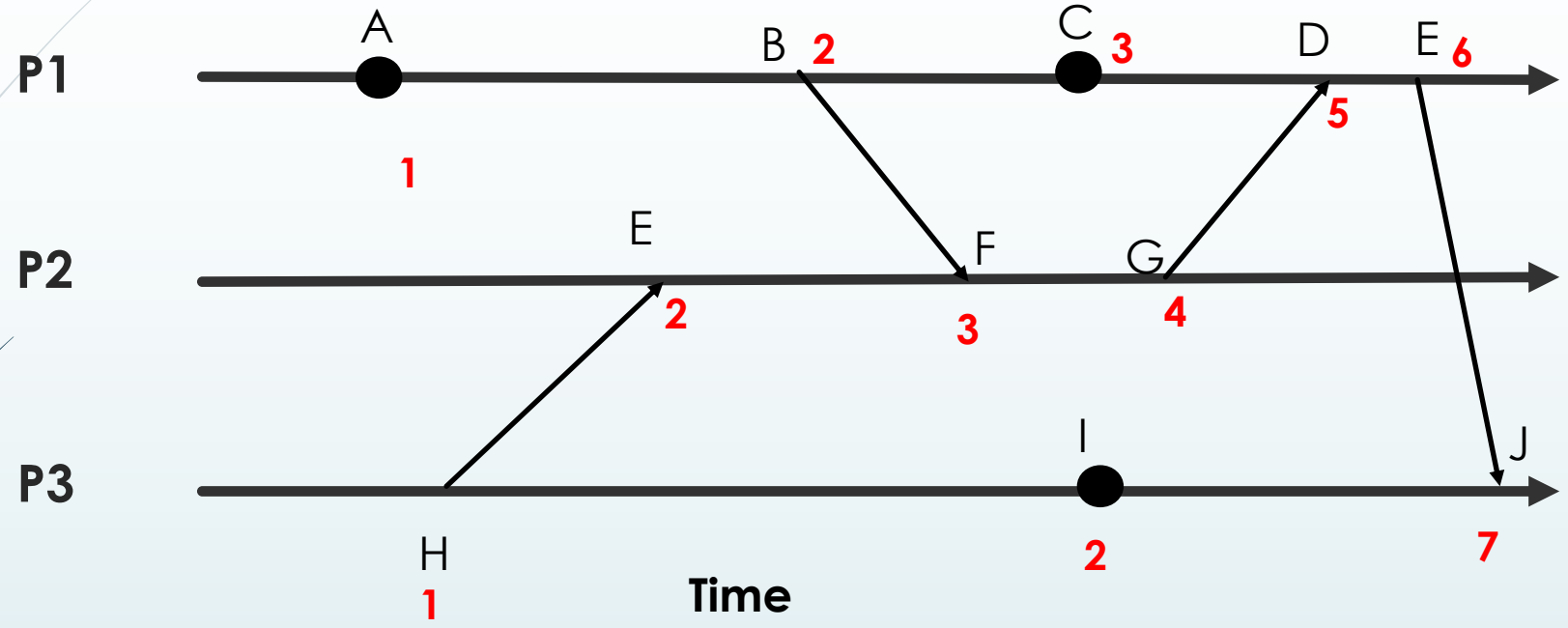

P1
0
A
B
C
D
E
ts = 1
P2
0
E
F
G
Message carries ts =1
P3
0
I
J
H
ts = 1
Message send
Time
Instruction or Step
Message


P1
0
A
1
B
C
D
E
ts = max(local, msg) + 1
= max(0, 1)+1
= 2
P2
0
E
F
G
Message carries ts =1
P3
0
H
1
I
J
Time
Instruction or Step
Message


P1
P2
P3
A
B
2
C
D
E
1
Message carries
ts = 2
E
F
2
max(2, 2)+1
=3
G
I
J
H
1
Time
Instruction or Step
Message


max(3, 4)+1
=5
P1
P2
P3
A
B
C
D
E
E
F
G
H
I
J
1
2
3
1
2
3
4
Time
Instruction or Step
Message

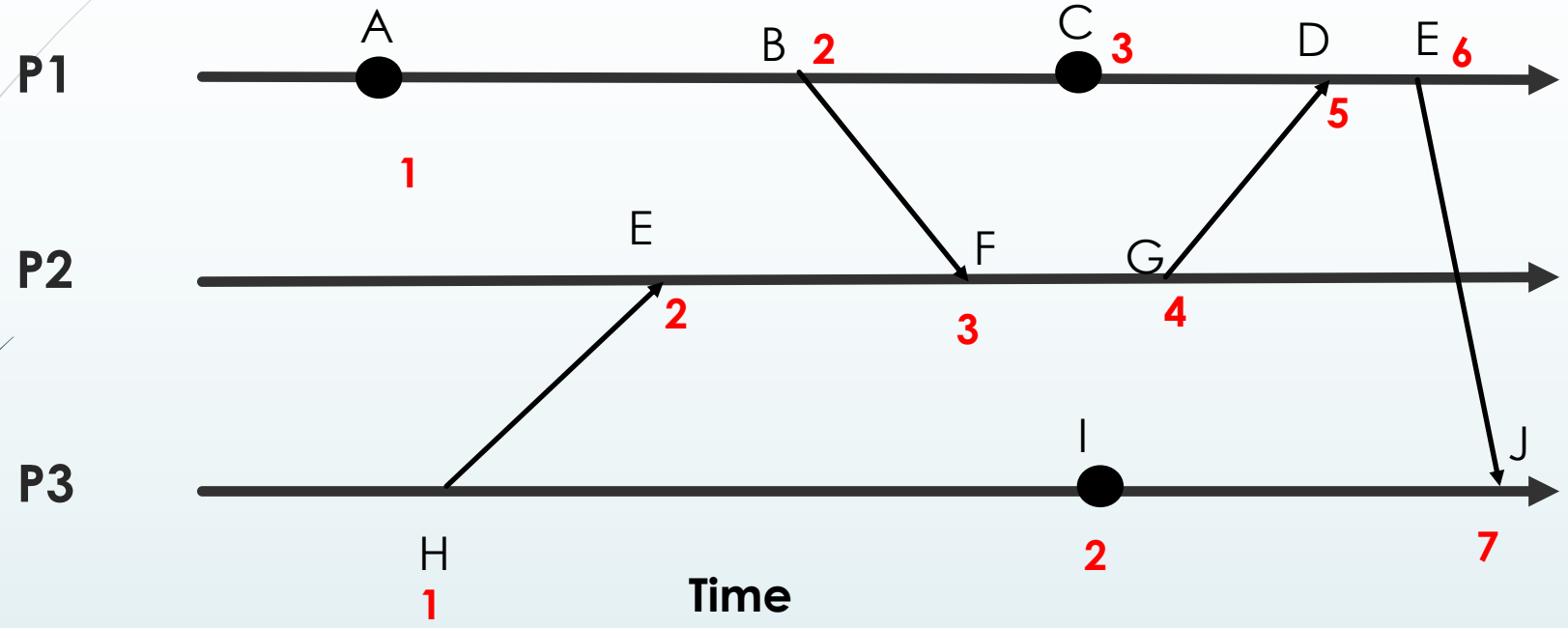

P1
P2
P3
A
B
C
D
E
F
G
H
I
J
1
2
3
4
5
6
7
Time
Instruction or Step
Message



A →B :: 1 < 2
B → F :: 2 < 3
A → F :: 1 < 3

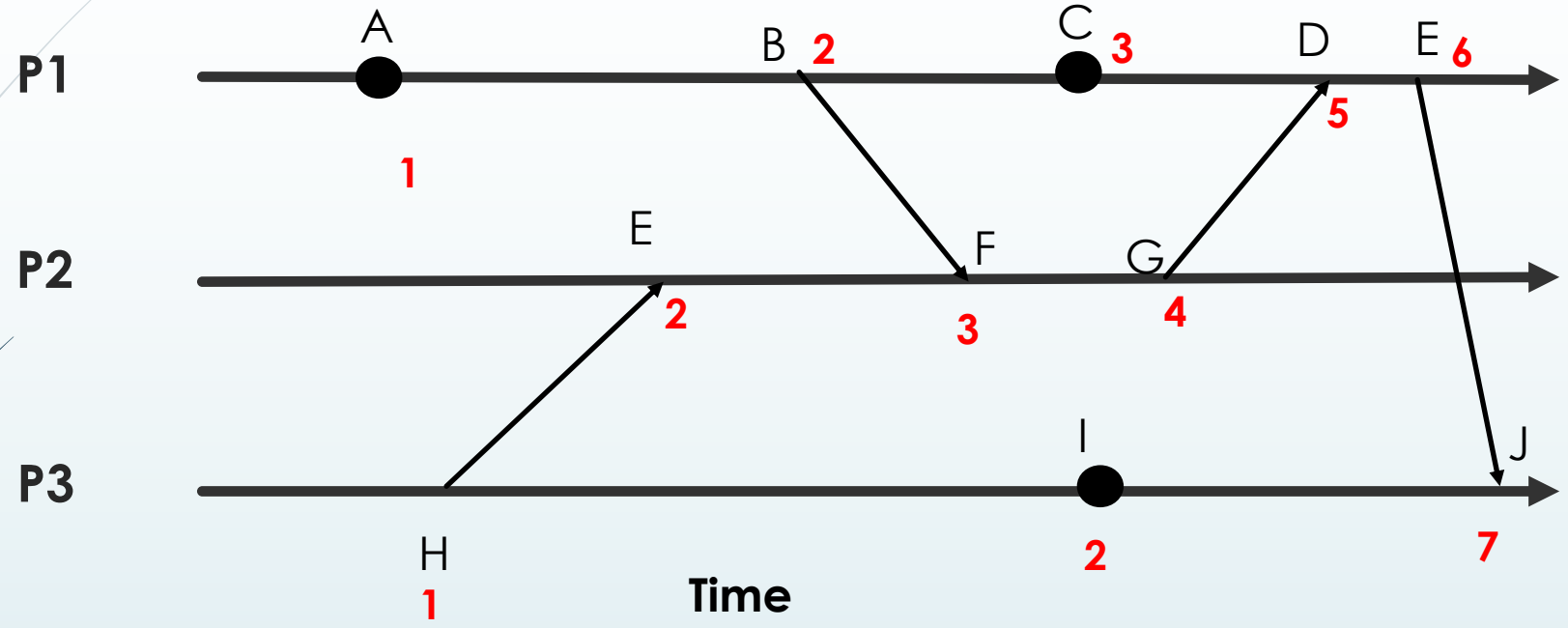
Instruction or Step

Message



H → G :: 1 < 4
F → J :: 3 < 7
H → J :: 1 < 7
C → J :: 3 < 7

Instruction or Step
Message



C → F ? :: 3 = 3

H → C ? :: 1 < 3

(C, F) and (H, C) are pairs of concurrent events

Instruction or Step

Message

Drawback

- Lamport timestamps not guaranteed to be ordered or unequal for concurrent events

- $E1 \rightarrow E2 \Rightarrow timestamp(E1) < timestamp(E2)$, BUT

  $timestamp(E1) < timestamp(E2) \Rightarrow \{E1 \rightarrow E2\}$ OR {E1 and E2 concurrent}

Source: https://www.coursera.org/lecture/cloud-computing/2-4-lamport-timestamps-4nPtE

Vector Timestamps

- Each process uses a vector of integer clocks
- Suppose there are N processes in the group $1 \dots N$
- Each vector has N elements
- Process i *maintains vector* $V_i[1 \dots N]$
- j^{th} element of vector clock at p_i, $V_i[j]$, is $i's$ knowledge of latest events at p_j
- Incrementing vector clocks:
  - On an instruction or send event at p_i, it increments only its i^{th} element of its vector clock
  - Each message carries the send-event's vector timestamp $V_{message}[1 \dots N]$
  - On receiving a message at p_i :
    - $\boldsymbol{V_i[i] = V_i[i] + 1}$
    - $\boldsymbol{V_i[j] = max(V_{message}[j], V_i[j])\ for\ j \neq i}$

Source: https://www.coursera.org/lecture/cloud-computing/2-5-vector-clocks-dy8wf

- $VT_1 = VT_2$,

  if and only if

  $VT_1[i] = VT_2[i], for\ all\ i = 1, \ldots, N$

- $VT_1 \leq VT_2$,

  if and only if

  $VT_1[i] \leq VT_2[i], for\ all\ i = 1, \ldots, N$

- Two events are causally related

  if and only if

  $VT_1 < VT_2$, that is,

  if and only if $VT_1 \leq VT_2$ *and*

  there exists j such that

  $1 \leq j \leq N$ and $VT_1[j] < VT_{2[j]}$

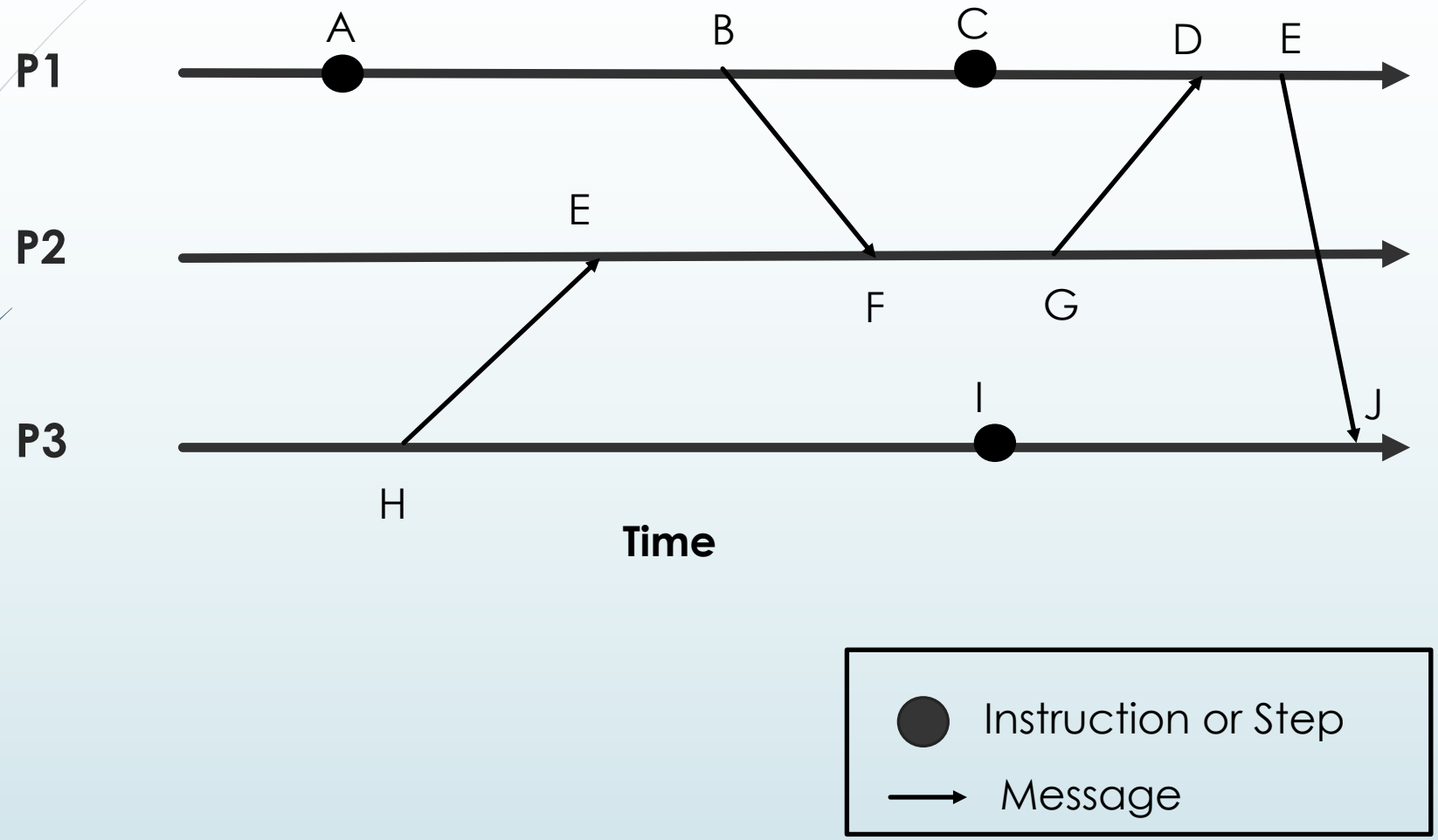
- Two events VT_1 and VT_2 are concurrent

  if and only if

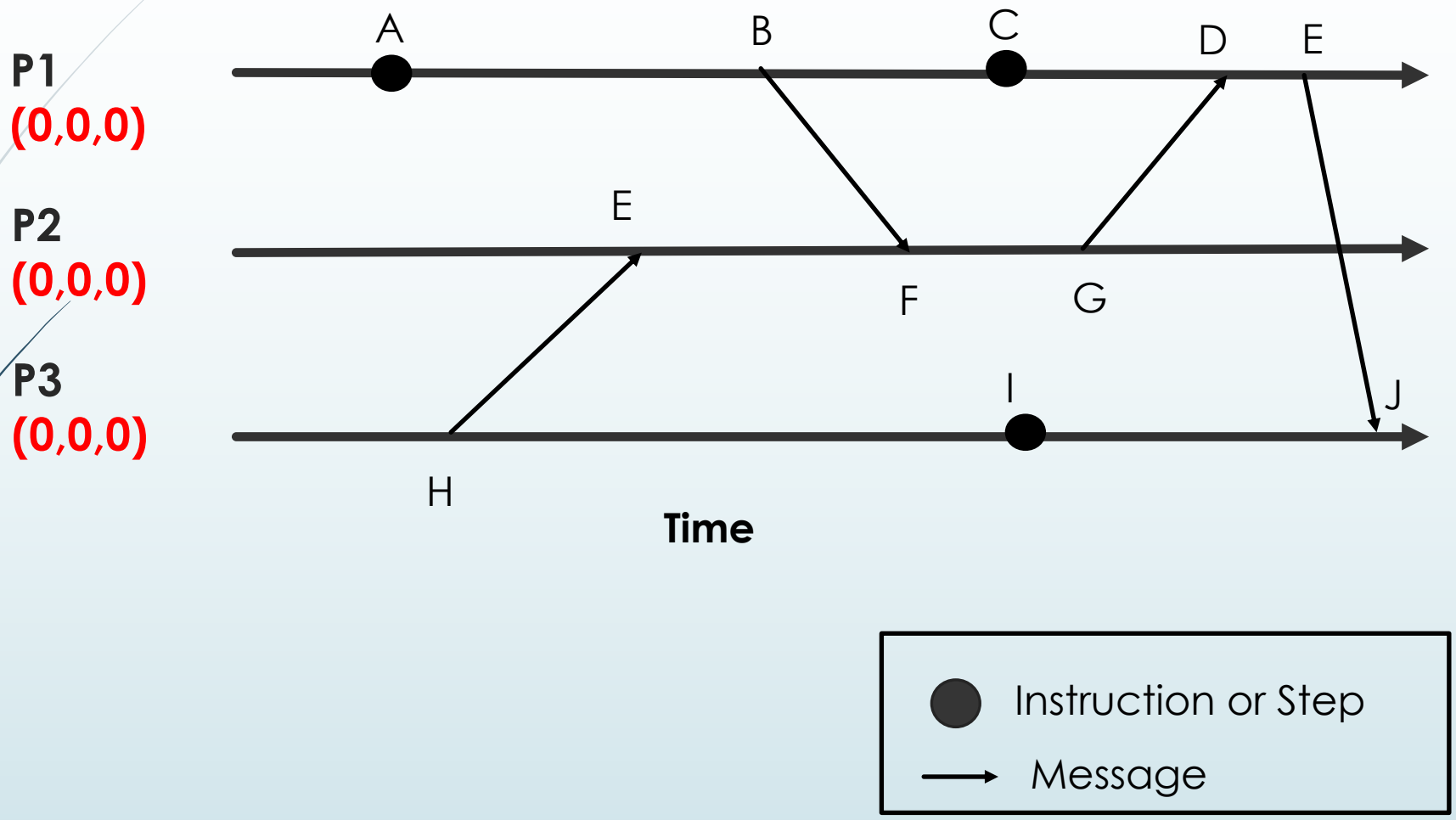
  NOT ($VT_1 \leq VT_2$) AND NOT ($VT_2 \leq VT_1$)

Source: https://www.coursera.org/lecture/cloud-computing/2-5-vector-clocks-dy8wf

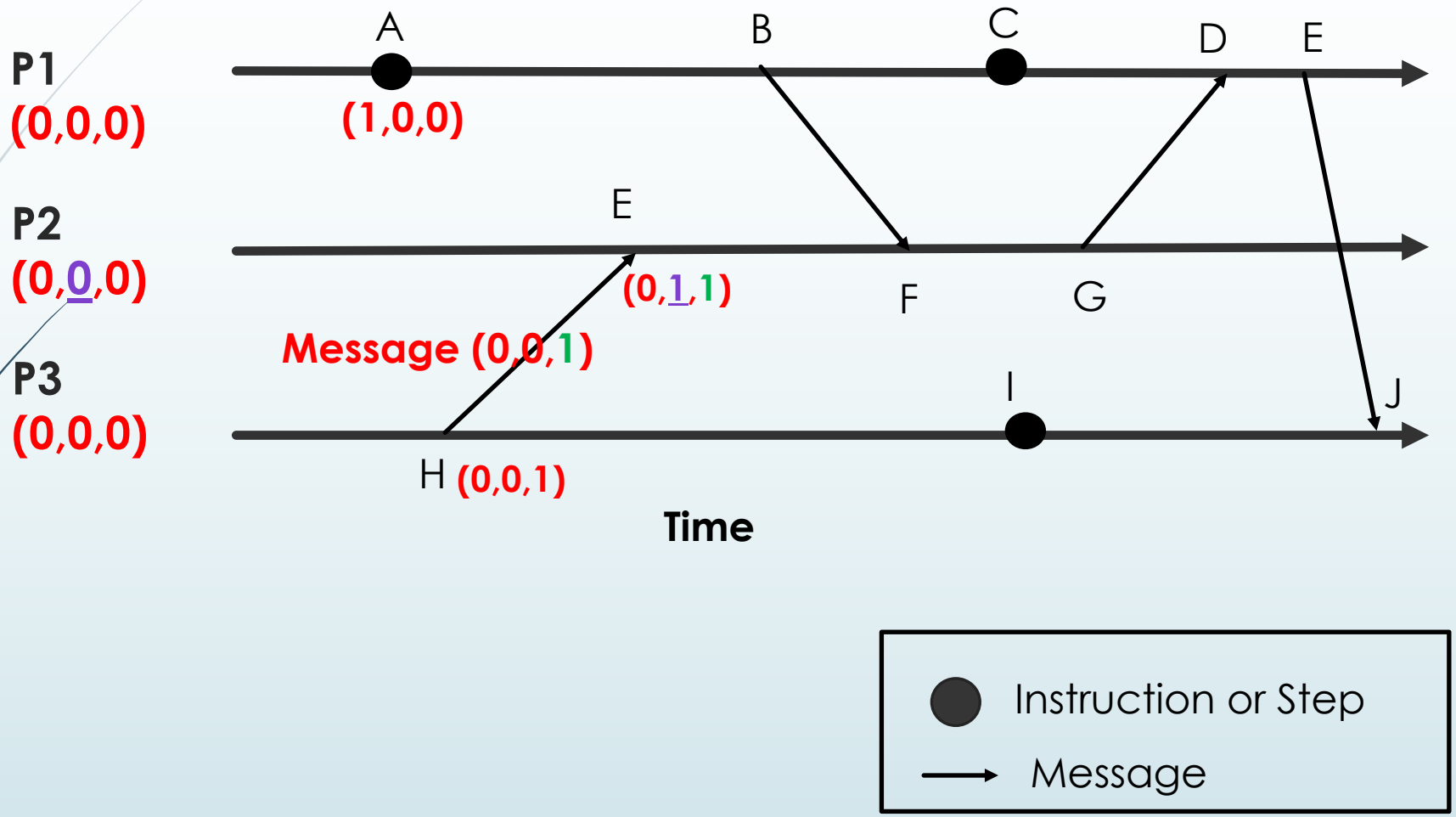
Vector Timestamps



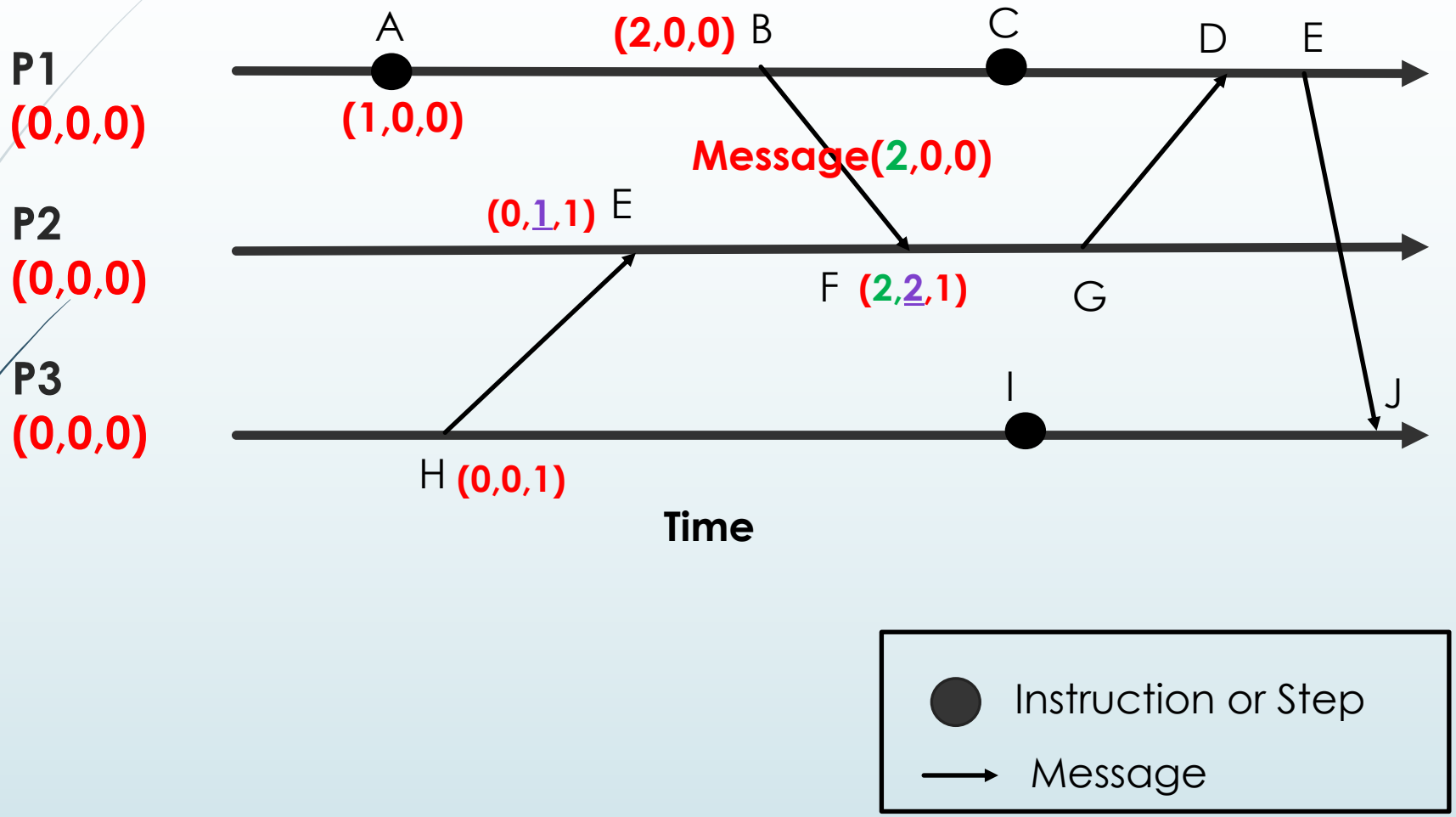
Vector Timestamps



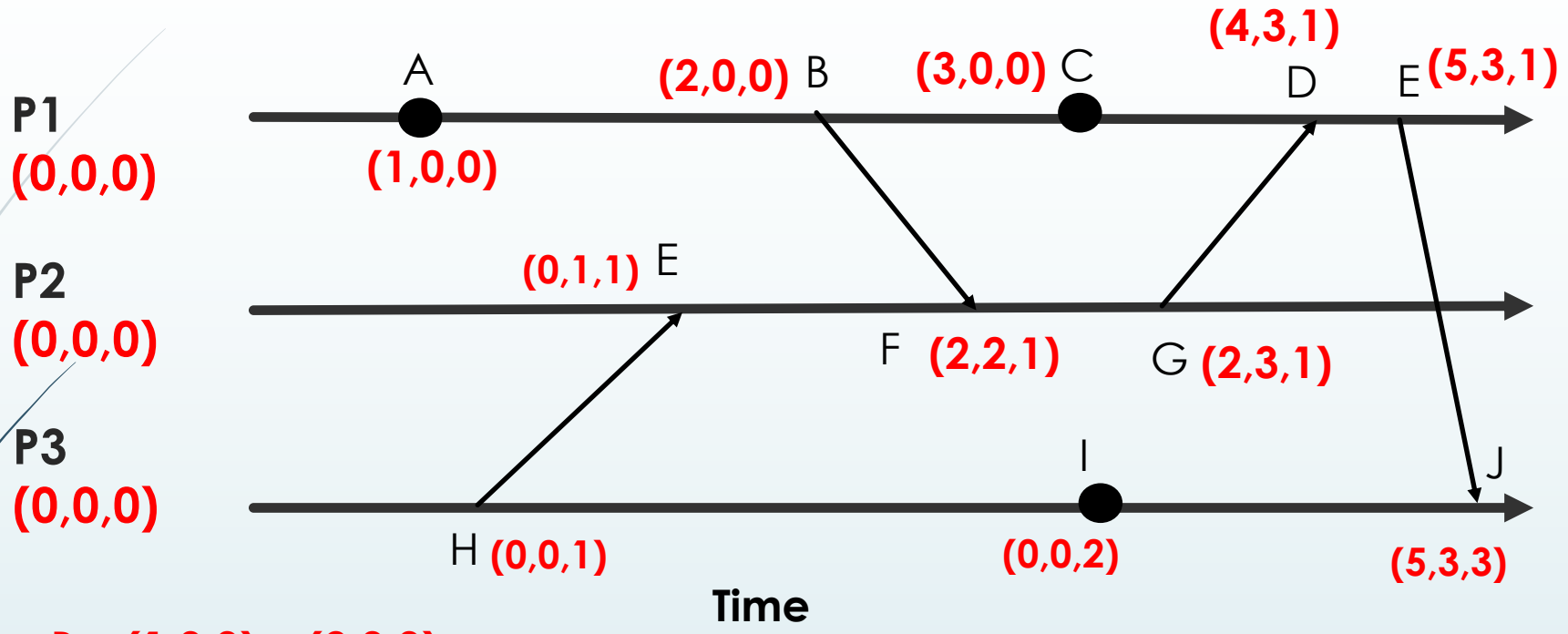
Vector Timestamps



Vector Timestamps



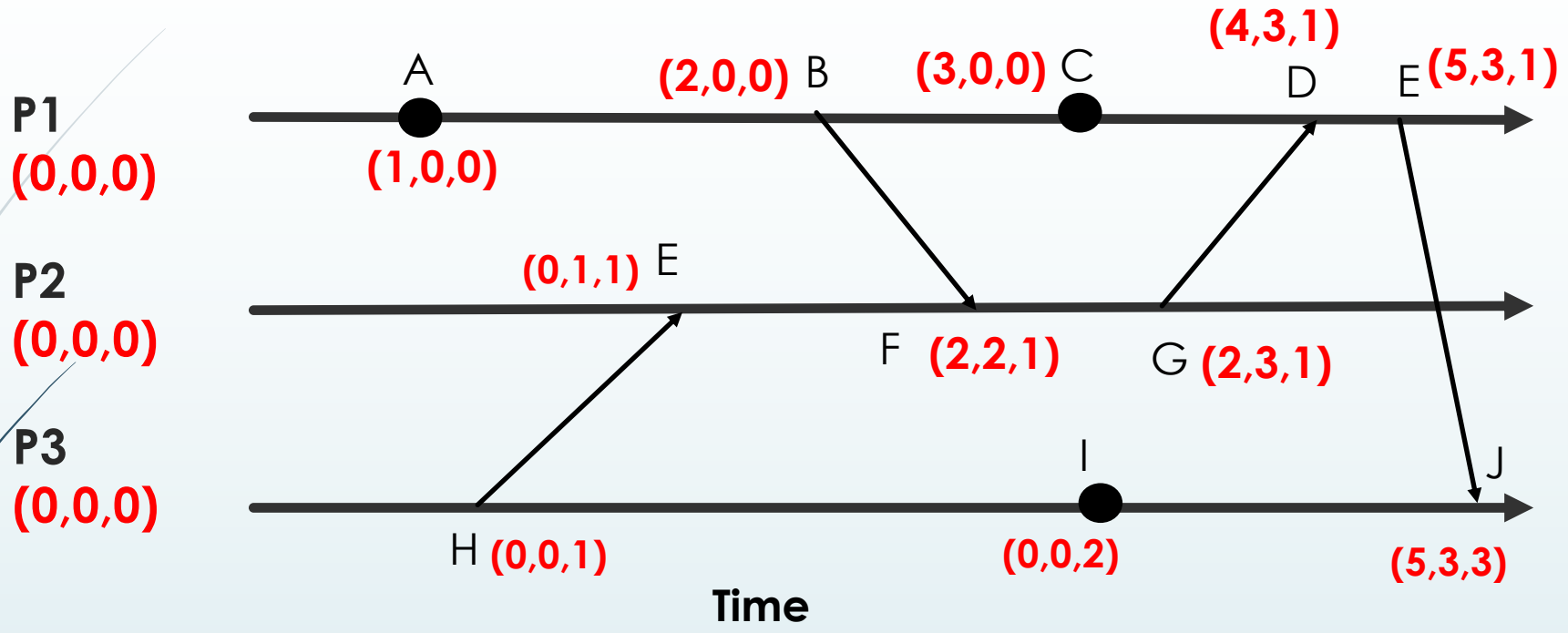
Vector Timestamps



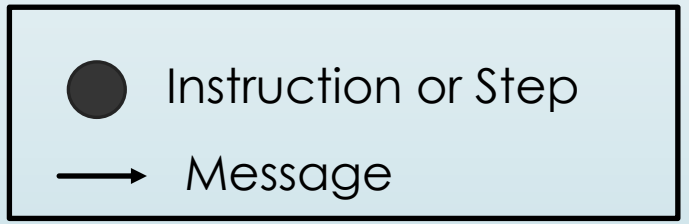
A → B :: (1,0,0) < (2,0,0)
B → F :: (2,0,0) < (2,2,1)
A → F :: (1,0,0) < (2,2,1)

H → G :: (0,0,1) < (2,3,1)
F → J :: (2,2,1) < (5,3,3)
H → J :: (0,0,1) < (5,3,3)
C → J :: (3,0,0) < (5,3,3))

Vector Timestamps



C & F :: (3,0,0) ||| (2,2,1)
H & C :: (0,0,1) ||| (3,0,0)
(C, F) and (H, C) are pairs of concurrent events



Programming

- Two classes of programs:
  - Concurrent-write free
  - Data Race-free
- Any program in these classes, if written to run correctly on sequentially consistent memory, also runs correctly in a system with causal memory.

Programming

- Two classes of programs:
  - Concurrent-write free
  - Data Race-free
- Any program in these classes, if written to run correctly on sequentially consistent memory, also runs correctly in a system with causal memory.

WHY ?

Concurrent-write free programs

- Advantage of using causal memory is that normal memory accesses can be implemented without blocking
- Processes need not synchronize with each other in performing these accesses.
- So, programs running on causal memory must do their own synchronization.
- One way to achieve this is to ensure that no two writes can be concurrent.

- **Definition:**
  - H is concurrent-write free with respect to $\rightsquigarrow$ if there are no two write operations w_1 and w_2 in H that are concurrent with respect to $\rightsquigarrow$.

Data-race free programs

- Two operations o_1 and o_2 in H compete with respect to $\rightsquigarrow$:
  - if both access the same location
  - at least one is a write
  - they are concurrent with respect to $\rightsquigarrow$.

- H is data-race free with respect to $\rightsquigarrow$, if it contains no pair of operations that compete with respect to $\rightsquigarrow$.

- A history that is data-race free with respect to $\rightsquigarrow$, has the property that all writes to a given location are linearly ordered with respect to $\rightsquigarrow$.

Discussion

- Has anyone implemented causal memory in a working system?