LOGICAL TIME

Dr. A. Baskar

"CAUSALITY BETWEEN EVENTS" — WHAT DOES IT MEAN?

- Two events are **causally related** if one event can affect or influence the other.
- Examples:
- A process sends a message → another process receives it (send happens **before** receive)
- A variable is updated → later another process reads it
- If event A can affect event B, we say:
- **A** → **B (A happens before B)**
- This relationship is called the **happened-before relation**.

WHY PHYSICAL TIME USUALLY TRACKS CAUSALITY

- In a single computer, we can assume:
  - One clock
  - One CPU
  - Events are ordered by physical time
- So we can say:
  - If time(A) < time(B), A probably happened before B.
  - Easy.

BUT IN DISTRIBUTED SYSTEMS… CLOCKS ARE NOT RELIABLE GLOBALLY

- A distributed system has:
  - multiple computers
  - each with its own clock
  - clocks drift (not perfectly synchronized)
  - messages take unpredictable time
    (network delay, congestion, failures, etc.)
- So we **cannot trust physical time** to say which event happened first.
- Example:
  - Machine A clock says 10:00:05
    Machine B clock says 09:59:59
  - But event on B might actually have occurred earlier in real life.
- Therefore:
- **There is no meaningful single "global clock".**

COMPUTATIONS PROGRESS "IN SPURTS"

- Distributed programs do not move smoothly.
  - Each node:
  - runs for a while,
  - pauses,
  - waits for messages,
  - resumes again.
- So instead of smooth synchronized time, progress is irregular.

WHY DO WE NEED "LOGICAL TIME"?

- **No global clock → physical time cannot order events correctly**
- Each machine has its own hardware clock.
  - clocks drift
  - network delays vary
  - clocks cannot be perfectly synchronized
- So two events may have timestamps like:
  - Event A on machine 1 → 10:00:05
  - Event B on machine 2 → 09:59:58
- Real world: A actually happened **before** B.
- But timestamps wrongly say B happened first.
- So physical time can give **incorrect ordering**.
- Therefore we cannot rely on physical time to decide causality.

WHAT WE ACTUALLY CARE ABOUT IS NOT SECONDS… BUT CAUSALITY

- **Which events can influence which other events?**
- Example:
  - P1 sends message M
  - P2 receives M
  - P2 updates data
- send(M) → receive(M) → update
- We must maintain this order.
- Logical time guarantees:
- If event A causally affects event B

then timestamp(A) < timestamp(B)

- That is the **monotonicity property**. (consistently moves in one direction)
- Physical time cannot guarantee this.

SUMMARY

- Logical time is used in distributed systems because there is no reliable global physical clock. Physical time cannot correctly capture the order of events due to clock drift and unpredictable message delays.
- Logical time ignores real time and instead preserves only the causal “happened-before” relationships between events, which is exactly what we need to reason about distributed computations.

LOGICAL CLOCKS = TOOLS TO TRACK CAUSALITY

- Different logical clocks give different accuracy:

Clock Type	Tracks what
Scalar Clock	Partial order (causal direction, but weak)
Vector Clock	Exact causality + detects concurrency
Matrix Clock	Tracks who knows that others know (meta-causality)

A FRAMEWORK FOR A SYSTEM OF LOGICAL CLOCKS

- **Clock Consistency Condition**.
- We want timestamps such that:
- If event **A happens before B** (A → B),
  then logical time must satisfy:

  $C(A) < C(B)$

- where **C(e)** is the logical timestamp of event *e*.
- This is called the **Clock Consistency Condition.**
- Logical clocks are not about real time — they enforce **causal ordering**.

- **Time domain T**
- A set of "time values" (integers, vectors, matrices).
- **Clock function C**
- Maps each event e to a timestamp:
  - $C : H \rightarrow T$
- Where:
  - **H** = set of all events
  - **T** = time domain
  - Timestamp = C(e).

A FRAMEWORK FOR A SYSTEM OF LOGICAL CLOCKS

- **Causal precedence (<)**
- Logical clocks represent the **happened-before relation (→)**
- **Clock Consistency (Monotonicity) Condition**
  - If: ei → ej
  - then timestamps must satisfy:
  - C(ei) < C(ej)
- **Strong consistency**
- **ei → ej ⇔ C(ei) < C(ej)**
- This means:
- → = "happened-before / causality"
- **C(e)** = timestamp assigned by the logical clock
- The symbol ⇔ means BOTH ways:
  - If ei → ej, then C(ei) < C(ej)
  - If C(ei) < C(ej), then ei → ej
- That is **stronger than Lamport clocks.**

IMPLEMENTING LOGICAL CLOCKS

Clock Type	Data Structure
Scalar	Single integer
Vector	Array of integers
Matrix	2D array

A FRAMEWORK FOR A SYSTEM OF LOGICAL CLOCKS

- **Scalar (Lamport) clocks guarantee only ONE direction:**
  - $ei \rightarrow ej \Rightarrow C(ei) < C(ej)$
  - but
  - $C(ei) < C(ej)$ **does NOT imply** $ei \rightarrow ej$
    (events may just be concurrent)
- **Vector clocks guarantee BOTH directions:**
  - $ei \rightarrow ej \Leftrightarrow VC(ei) < VC(ej)$
  - So vector clocks perfectly model causality.

HOW LOGICAL CLOCKS ARE ACTUALLY IMPLEMENTED

- To implement logical clocks, every process must maintain:
- Some **data structure** to store time
- A **protocol (rules)** to update time correctly
- Because our goal is:
- If event ei → ej, then the logical clock must satisfy
  C(ei) < C(ej)

<u>Two data structures kept by each process</u>

- Each process **pi** keeps:

1. Local logical clock — lci

  - **How far this process itself has progressed.**

2. Global logical clock — gci

  - **pi's best knowledge of global logical time (what others may have done).**

- So gci = local view of what's happening in the whole system.

WHY PROTOCOL IS NEEDED

- Just storing clocks is not enough.
- The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently.
- We need rules to ensure:
- causality is preserved
- timestamps are comparable
- future events never go backward

R1: THIS RULE GOVERNS HOW THE LOCAL LOGICAL CLOCK IS UPDATED BY A PROCESS WHEN IT EXECUTES AN EVENT.

- Rule R1 — Local progress rule
- When process pi performs any local event, it must update its **local logical clock (lci)**.
- **R1**: update local clock on every event

R2: THIS RULE GOVERNS HOW A PROCESS UPDATES ITS GLOBAL LOGICAL CLOCK TO UPDATE ITS VIEW OF THE GLOBAL TIME AND GLOBAL PROGRESS.

- **Rule R2 — Global update rule (communication rule)**
- This governs **sending and receiving messages**.
- Update the process's view of global time (gci).
- When we receive someone else's timestamp:
- merge it with our own
- advance our clock so causality holds
- **R2**: update global clock when exchanging messages (merge timestamps)

TYPES OF LOGICAL CLOCKS

- 1. Scalar Logical Clocks (Lamport)
- 2. Vector Clocks
- 3. Matrix Clocks

SCALAR LOGICAL CLOCKS (LAMPORT)

- A **Lamport scalar clock** assigns a single integer to every event in a distributed system so that:
- If event A causally happened before event B,

  then timestamp(A) < timestamp(B).
- Formal definition
- Let:
  - C = logical clock function
  - e = event
- Then:
  - $C : H \rightarrow \mathbb{N}$
- Logical clocks must satisfy the **clock consistency condition**:
  - If $e_i \rightarrow e_j$ then $C(e_i) < C(e_j)$
- This means timestamps must **preserve causality**.

SCALAR LOGICAL CLOCKS (LAMPORT)

- "Time should never go backward, and messages should always look like they happened after they were sent."
- **Lamport Clock Rules (R1 & R2)**
  - Every process Pi keeps an integer:
  - LCi (initially 0)
- **Rule R1: Before every local event**
  - (internal computation, send, etc.)
  - LCi = LCi + 1
- **Rule R2: On receiving a message (m, Tm)**
  - Message carries timestamp Tm.
  - When Pi receives:
  - LCi = max(LCi, Tm) + 1
- **Why max + 1?**
  - Because the receiving event must be:
  - later than local events
  - later than the send event

ALGORITHM

- **RULE R1 — Local Clock Update (Before Any Event)**
- Before executing an event (send, receive, or internal), process pi executes:
  - Ci := Ci + d (d > 0)
- Where:
  - Ci = logical clock at process pi
  - d = positive increment (usually 1)
- **Simple meaning**
- Whenever something happens at a process, **logical time must advance**.
- Even if nobody sends messages.
- **Why?**
- Because events inside the same process must remain ordered:
  - e1 happens before e2
  - ⇒ timestamp(e1) < timestamp(e2)
- If we didn't increment, two different events could have the **same time**, which breaks ordering.

ALGORITHM

- **RULE R2 — Message Handling (Send + Receive)**
- R2 has **two parts**:

(A) Sending a message

- Each message piggybacks the clock value of its sender at sending time.
  - Steps:
  - Apply R1
  - Attach the current clock value to the message
  - So:
  - Ci = Ci + 1
  - send(message, Ci)
  - This timestamp represents:
  - "This message happened at this logical time."

(B) Receiving a message

  - When pi receives a message with timestamp Cmsg, it executes:
  - Ci := max(Ci, Cmsg)   -----Move my clock forward so it is **at least as large as the sender's clock**.
  - Execute R1 ------------------- receiving event is strictly after sending
  - Deliver the message --------the system must store the event **with correct timestamp**

EXAMPLES

- **Example 1 — Send then Receive**
- Initial:
  - P1: C1 = 0, P2: C2 = 0
- **Event — P1 sends message**
  - Apply R1:
  - C1 = 1
  - Send with timestamp 1.
  - Message: (m, 1)
- **Event — P2 receives message (m,1)**
- Step 1:
  - C2 = max(0,1) = 1
- Step 2 (R1):
  - C2 = 2
- Final:
  - send = 1
  - receive = 2
- Correct ordering:
- send < receive

EXAMPLE 2 — RECEIVER ALREADY AHEAD

- Initial:
- P1: C1 = 2 , P2: C2 = 10
- P1 sends:
  - C1 = 3
  - (m,3)
- P2 receives (m,3):
  - C2 = max(10,3) = 10
  - C2 = 11   (after R1)
- Meaning:
- receiver was already in the "future"
- message still comes later than send

EXAMPLE 3 — TWO MESSAGES CROSSING (IMPORTANT)

- P1: C1 = 0   P2: C2 = 0
- P1 sends to P2:
  - C1 = 1
  - (m1, 1)
- P2 sends to P1:
  - C2 = 1
  - (m2, 1)
- P2 receives m1:
  - C2 = max(1,1)=1
  - C2 = 2
- P1 receives m2:
  - C1 = max(1,1)=1
  - C1 = 2
- Result:
  - Event timestamps:
  - both receive events = 2
  - We cannot know which one truly happened first.
- This proves:
- Lamport clocks do NOT detect concurrency.

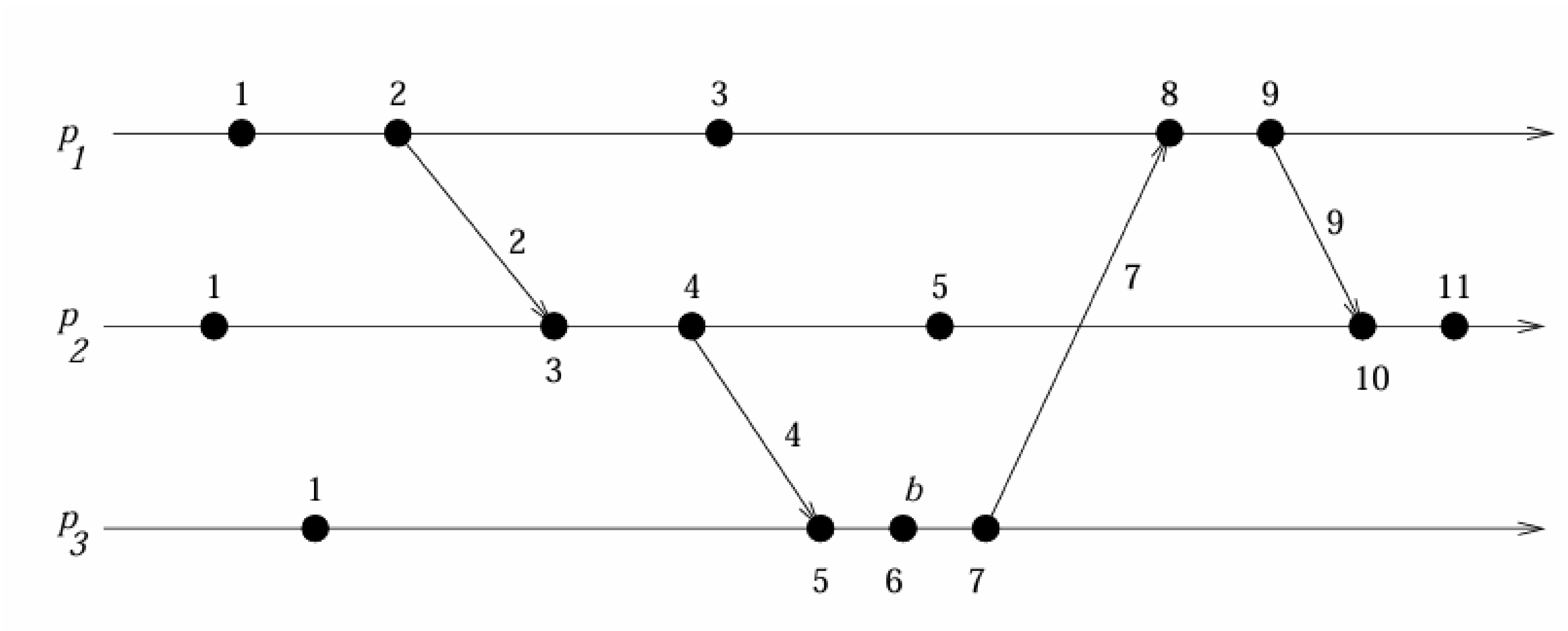
WHAT LAMPORT CLOCKS DO NOT GUARANTEE

- If two events are **concurrent**, Lamport clocks may still assign:
- $C(x) < C(y)$ *even though they are unrelated*
- So:

$$C(x) < C(y) \not\Rightarrow x \to y$$

- That's why we sometimes need **vector clocks** — but scalar clocks are enough when we only need ordering that respects causality, not detect concurrency.

EXAMPLE



- **INITIAL STATE**
- $C_1 = 0, C_2 = 0, C_3 = 0$
- **PROCESS P1 — LEFT SIDE**
- **Event (1) — internal event**
- Applies R1:

$$C_1 = 0 + 1 = 1$$

- timestamp written as **1** above the dot.
- **Event (2) — send message to P2**
- Still R1 (because sending is also an event):

$$C_1 = 1 + 1 = 2$$

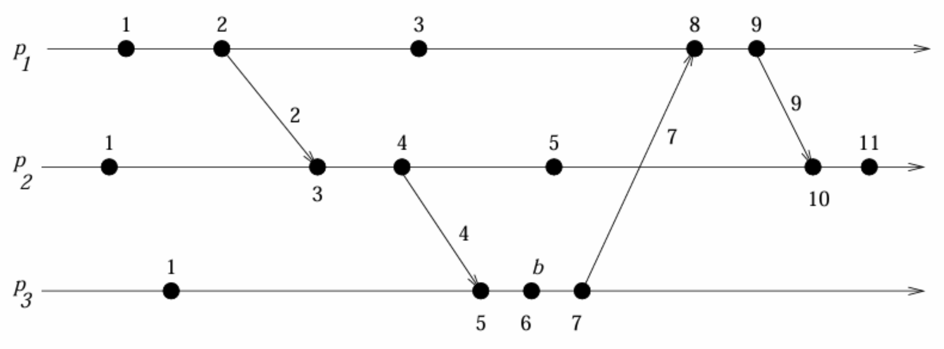
- Message leaves P1 carrying timestamp:

$$C_{msg} = 2$$

- label at send event is **2.**
- **Event (3) — internal**
- Again R1:

$$C_1 = 2 + 1 = 3$$

- timestamp at event = **3.**



- **PROCESS P2 — LEFT SIDE**
- **Event (1) — internal**
- R1:

$$C_2 = 0 + 1 = 1$$

- timestamp is **1**

- **Event (3) — RECEIVE message from P1 (timestamp 2)**
- Message arrives carrying:

$$C_{msg} = 2$$

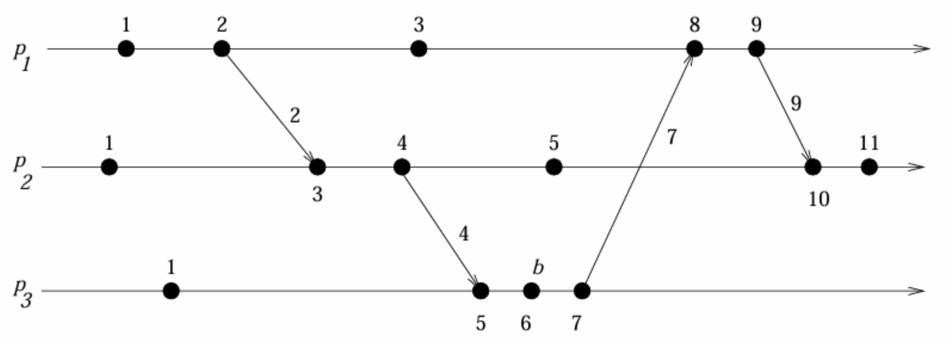
- Apply R2:
- **Step 1 — synchronize**

$$C2=\max(1, 2)=2$$

- **Step 2 — receive event**

$$C_2 = 2 + 1 = 3$$

- timestamp at receive = **3**
- Note:
- The receive event MUST be after the send (2 < 3). Good.



- **Event (4) — send to P3**
- R1:

$$C_2 = 3 + 1 = 4$$

- Message carries timestamp:

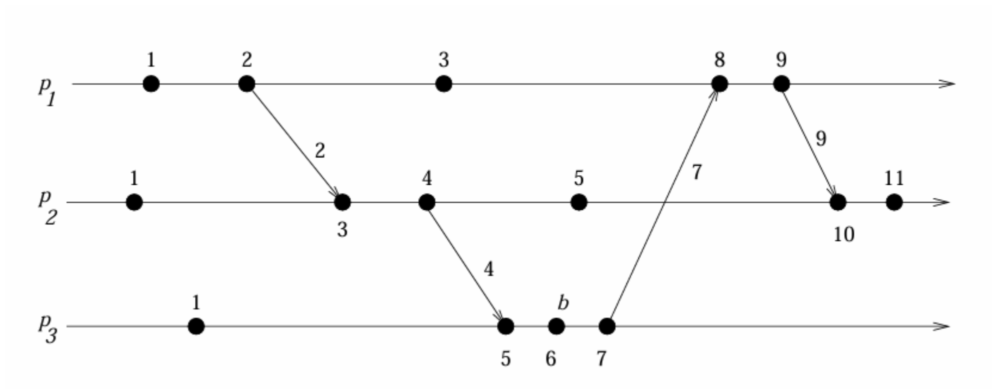
$$C_{msg} = 4$$

- timestamp = **4**

- **Event (5) — internal**
- R1:

$$C_2 = 4 + 1 = 5$$

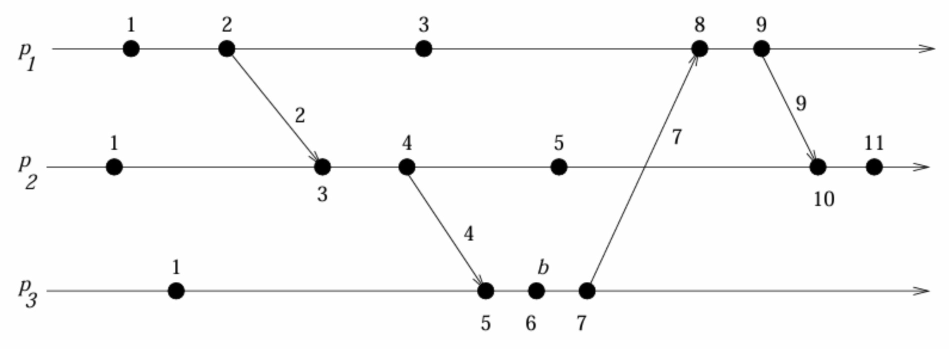
- timestamp = **5**



- **PROCESS P3 — LEFT SIDE**
- **Event (1) — internal**
- R1:

$$C_3 = 0 + 1 = 1$$

- timestamp = **1**



- **Event (5) — RECEIVE message from P2 (timestamp 4)**
- Message carries:

$$C_{msg} = 4$$

- Apply R2:
- **Step 1**

C3=max(1, 4)=4

- **Step 2 (receive)**

$$C_3 = 4 + 1 = 5$$

- timestamp = **5**

- **Event (6) — internal**
- R1:

$$C_3 = 5 + 1 = 6$$

- timestamp = **6**
-

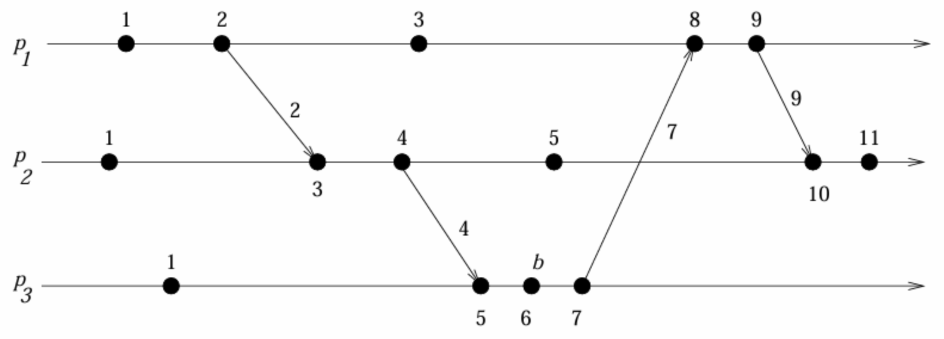
- **Event (7) — send to P1**
- R1:

$$C_3 = 6 + 1 = 7$$

- Message carries:

$$C_{msg} = 7$$

- timestamp = **7**



- **PROCESS P1 — RIGHT SIDE**
- **Event (8) — RECEIVE message from P3 (timestamp 7)**
- Current:

$$C_1 = 3$$

- Message:

$$C_{msg} = 7$$

- Apply R2:
- **Step 1**

C1=max(3, 7)=7

- **Step 2**

$$C_1 = 7 + 1 = 8$$

- timestamp = **8**

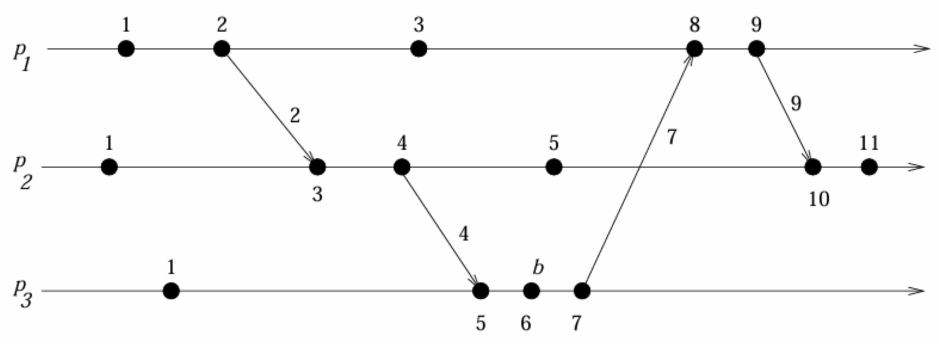
- **Event (9) — send to P2**
- R1:

$$C_1 = 8 + 1 = 9$$

- Message carries:

$$C_{msg} = 9$$

- timestamp = **9**



- **PROCESS P2 — RIGHT SIDE**
- **Event (10) — RECEIVE message from P1 (timestamp**
- Current:

$$C_2 = 5$$

- Message:

$$C_{msg} = 9$$

- Apply R2:
- **Step 1**

$$C2=\max(5, 9)=9$$

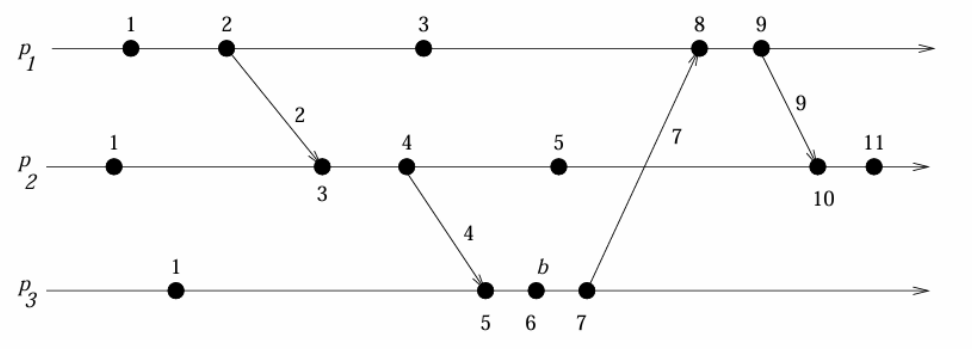
- **Step 2**

$$C_2 = 9 + 1 = 10$$

- timestamp = **10**
- **Event (11) — internal**
- R1:

$$C_2 = 10 + 1 = 11$$

- timestamp = **11**



VECTOR TIME (VECTOR CLOCKS)

VECTOR TIME (VECTOR CLOCKS)

- **What is a Vector Clock?**
- A **vector clock** is an array of integers that helps us:
  - detect causality
  - tell whether two events are related or concurrent
- In a system with **N processes**, each process keeps a vector:

$$VC[i] = [v_1,\ v_2,\ \dots,\ v_N]$$

- Interpretation:
- v_k =process i's knowledge of **how many events have happened at process k**, that causally affect i.
- So each process stores **its view of everyone's time**, not only its own.

What vector clocks can do (big motivation)

- Vector clocks can tell:
- **happens-before**

$$e_i \to e_j$$

- **concurrency**

$$e_i \parallel e_j$$

- neither happened before the other

Key properties of vector clocks

- **Strong consistency**

$$e_i \to e_j \Leftrightarrow VC(e_i) < VC(e_j)$$

- Unlike scalar clocks — this is **both directions.**

- **Detect concurrency**

$$VC(e) \not< VC(f) \text{ and } VC(f) \not< VC(e)$$

- $\Rightarrow$events are concurrent.

RULE 1

- R1: Before executing any event, increment your own entry.

RULE R1 — Local Event Update

- At process pi:

vti[i] := vti[i] + d   (usually d = 1)

RULE2

- When process p_i receives a message (m, vt) that carries the sender's vector clock:
- Process p_ip erforms three actions **in order**.
- **Step 1 — Update global logical time (merge vectors)**
- For every entry k, where $1 \le k \le n$:

$$vt_i[k] := \max(vt_i[k],\ vt[k])$$

- Take the element-wise maximum of the local vector and the received vector.
- **Step 2 — Execute R1 (local advance)**
- R1 says:

$$vt_i[i] := vt_i[i] + 1$$

- Receiving a message is also an event, so the receiver must advance its own logical time.
- This guarantees: receive happens **after** send.
- **Step 3 — Deliver the message**

WORKED EXAMPLE (3 PROCESSES)

- **Initial state:**
  - P1: [0,0,0]
  - P2: [0,0,0]
  - P3: [0,0,0]
- **Step 1 — P1 does event**
  - P1: R1 → [1,0,0]
- **Step 2 — P1 sends message to P2**
  - Increment (send is an event):
  - P1: [2,0,0]
  - Send (msg, [2,0,0])

Step 3 — P2 receives message

Before receive:

P2: [0,0,0]

Message carries: [2,0,0]

max([0,0,0], [2,0,0]) = [2,0,0]

Apply R1 (receive event):

P2: [2,1,0]

Step 4 — P2 sends message to P3

Increment:

P2: [2,2,0]

Send ([2,2,0])

Step 5 — P3 receives

Before:

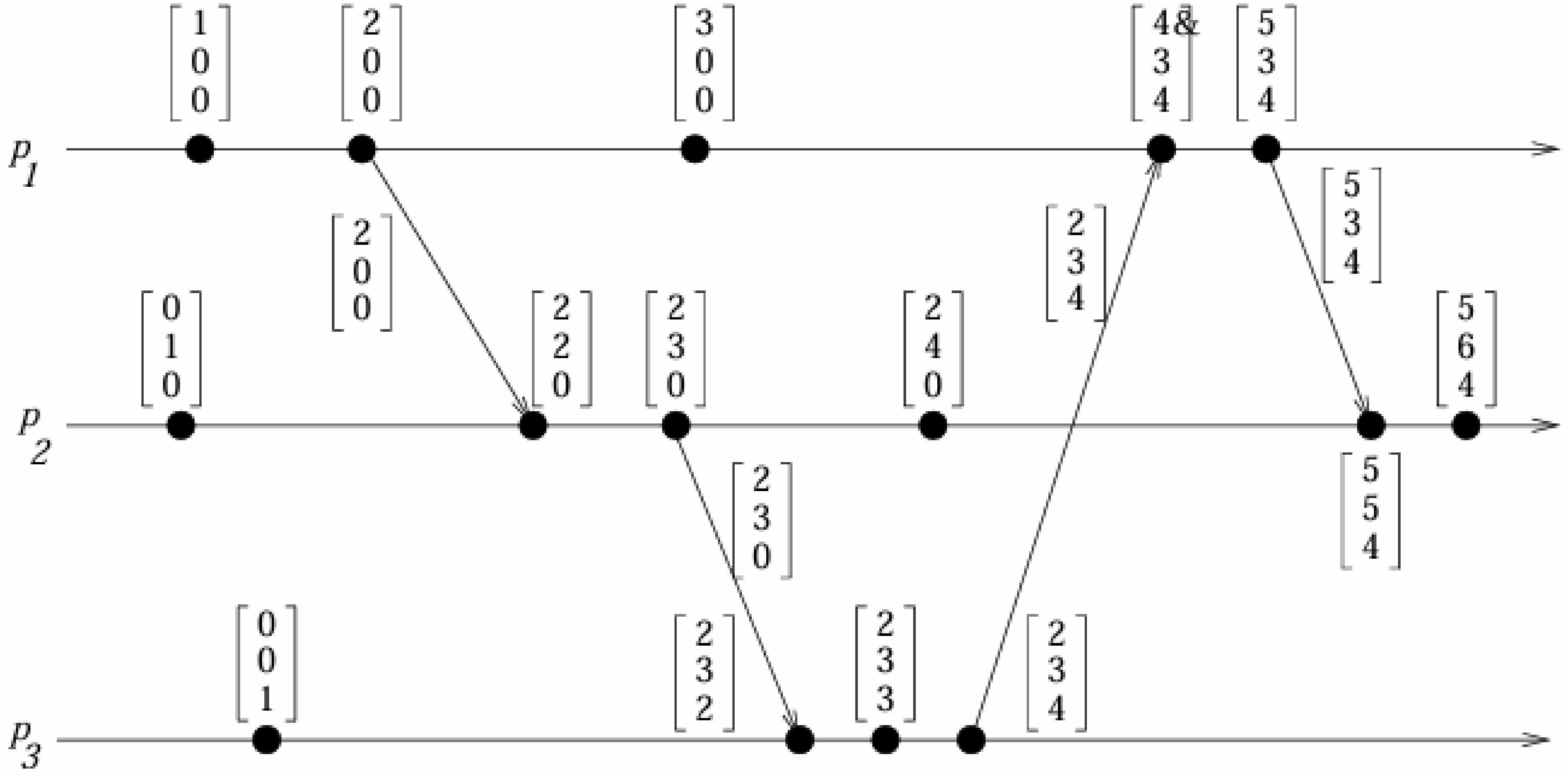
P3: [0,0,0]

Merge:

max([0,0,0],[2,2,0]) = [2,2,0]

Increment:

P3: [2,2,1]



p_1
p_2
p_3
[1 0 0]
[2 0 0]
[3 0 0]
[4& 3 4]
[5 3 4]
[2 0 0]
[5 3 4]
[2 3 4]
[0 1 0]
[2 2 0]
[2 3 0]
[2 4 0]
[5 6 4]
[5 5 4]
[2 3 0]
[0 0 1]
[2 3 2]
[2 3 3]
[2 3 4]

RULE-1 (Local Event)
Before executing any local event
VC[i] = VC[i] + 1

Only the entry for that process increases.
Example at the start:
Process p_1:

$$[0, 0, 0] \rightarrow [1, 0, 0]$$

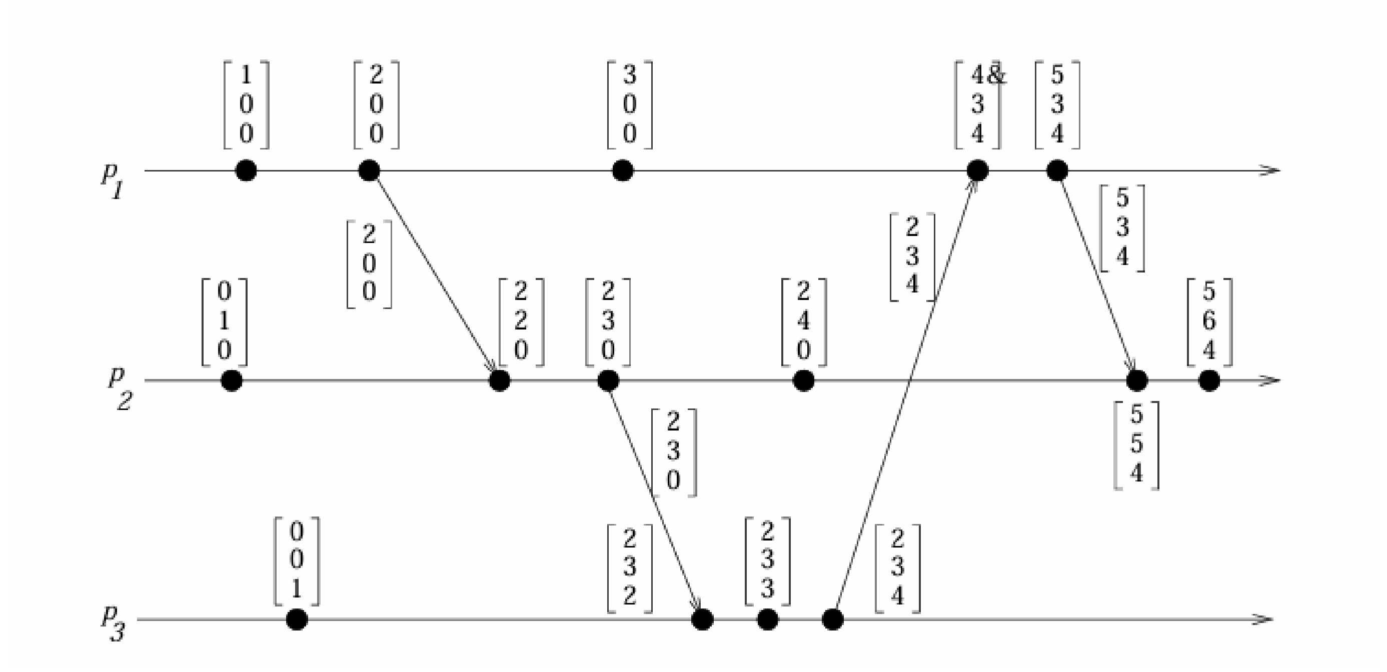
Process p_2:

$$[0, 0, 0] \rightarrow [0, 1, 0]$$

Process p_3:

$$[0, 0, 0] \rightarrow [0, 0, 1]$$

Nothing is communicated — each process only advances **its own clock**.



Message from $p_1 \rightarrow p_2$

Sender at p_1:

$$[1,0,0] \rightarrow [2,0,0]$$

Message carries:

$$[2,0,0]$$

Receiver p_2 current clock:

$$[0,1,0]$$

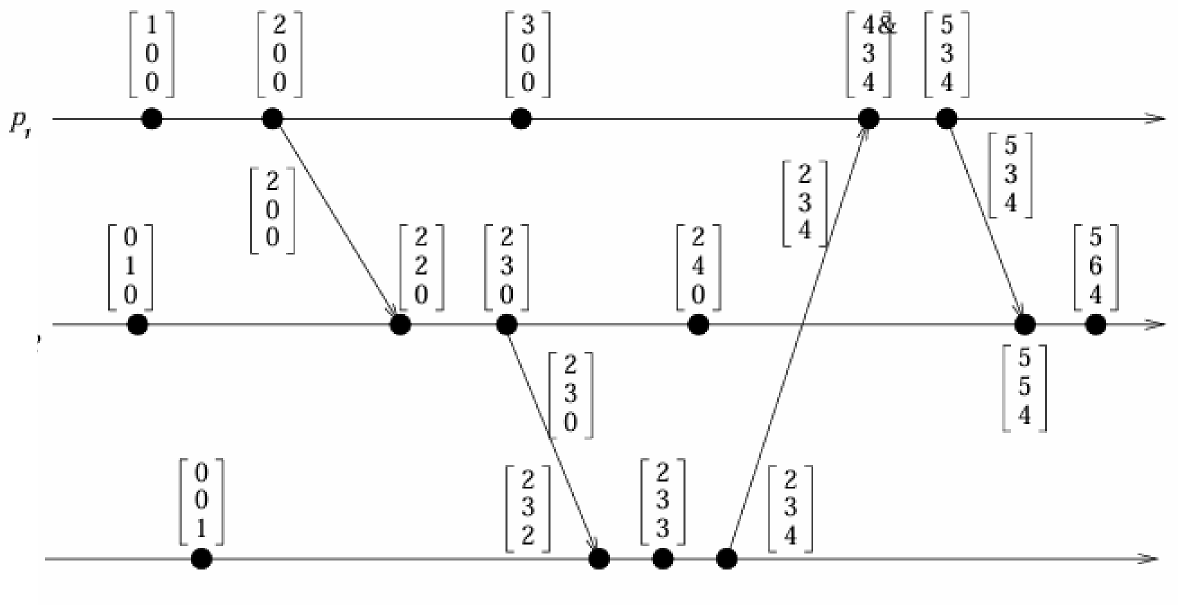
Step-1: element-wise max:

$$\max([0,1,0],[2,0,0]) = [2,1,0]$$

Step-2: increment own entry:

$$[2,1,0] \rightarrow [2,2,0]$$

That's the value shown in your diagram at p_2.



Message from $p_2 \to p_3$

Sender increments first:

$$[2, 2, 0] \to [2, 3, 0]$$

Message carries:

$$[2, 3, 0]$$

Receiver p_3 had:

$$[0, 0, 1]$$

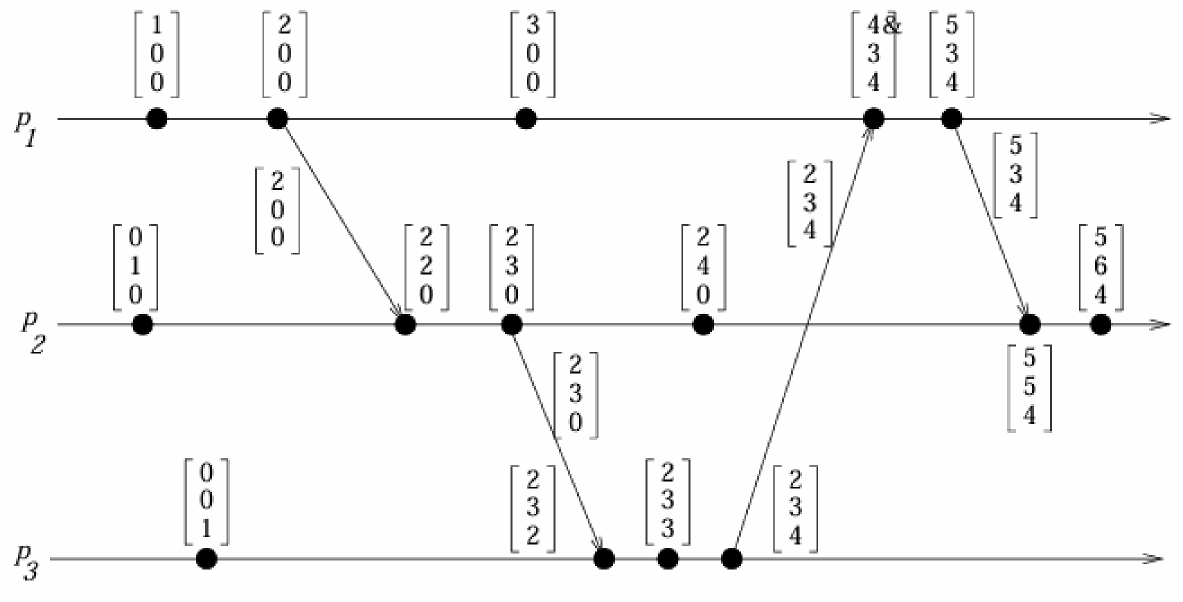
Take max:

$$\max([0, 0, 1], [2, 3, 0]) = [2, 3, 1]$$

Increment own entry:

$$[2, 3, 1] \to [2, 3, 2]$$

Matches diagram.



Message from $p_3 \to p_1$

Sender first:

$$[2,3,2] \to [2,3,3]$$

Message carries:

$$[2,3,3]$$

Receiver p_1 had:

$$[3,0,0]$$

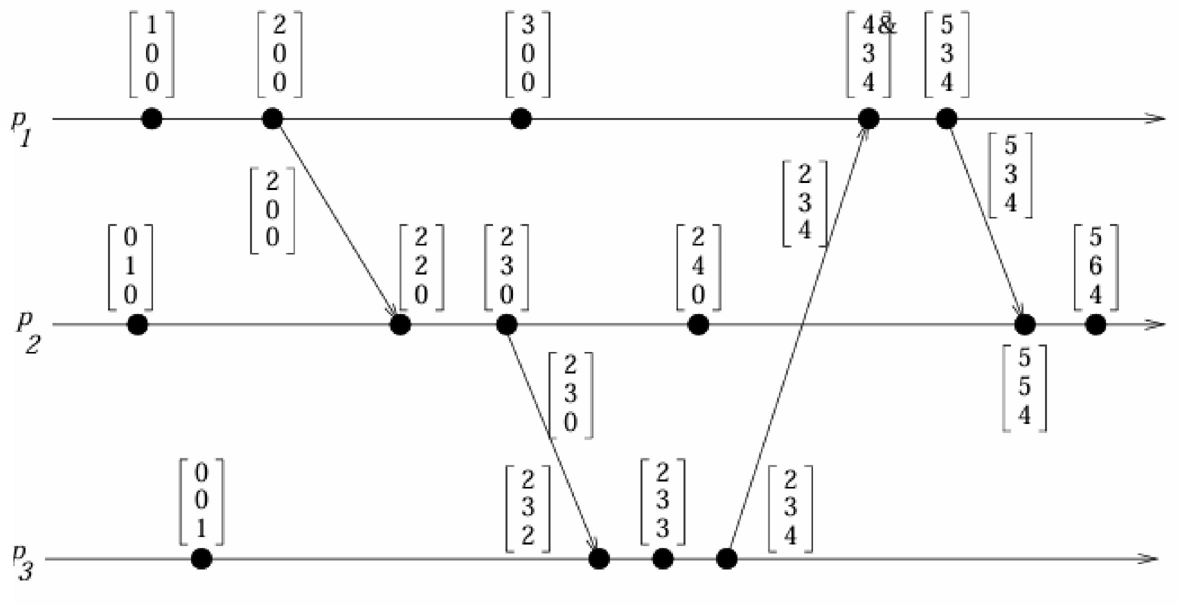
Take max:

$$\max([3,0,0],[2,3,3]) = [3,3,3]$$

Increment own entry:

$$[3,3,3] \to [4,3,3]$$

Diagram shows slightly different final form $[4,3,4]$ later because of **another event affecting p_3** before the next interaction — consistent with causality.



SINGHAL–KSHEMKALYANI DIFFERENTIAL TECHNIQUE

- Between two messages sent to the **same process**, usually only a few entries change.
- So instead of sending the full vector, the sender sends **only the entries that changed**.
- **At sender** p_i
  - It remembers:
  - Last timestamp it sent to p_j
  - Before sending next message to p_j:
  - Check which entries changed
- Send only those entries — **as pairs**

$$(i_k, v_k)$$

  - where:
  - i_k =index (which process)
  - v_k =new clock value
- So compressed timestamp:

$$\{(i_1, v_1), (i_2, v_2), \ldots, (i_{n_1}, v_{n_1})\}$$

SINGHAL–KSHEMKALYANI DIFFERENTIAL TECHNIQUE

- **At receiver** p_j
- When $p_j$receives the message:
- For each pair (i_k, v_k):
- Then applies **Rule R1** (increment its own entry).

Assume 4 processes:

$$VT_1 = [4, 1, 0, 2]$$

Previously, p_1 sent to p_3:

$$[3, 1, 0, 2]$$

Since then, only entry for itself changed:

$$[4, 1, 0, 2]$$

So instead of sending whole vector:

❌ Normal vector clock:

$$[4, 1, 0, 2]$$

✓ Differential technique sends only:

$$\{(1, 4)\}$$

Receiver updates:

$$VT_3[1] = \max(VT_3[1], 4)$$

Much smaller message.