Sistemas Distribuídos

Synchronization - Logical Clocks

Eurico Pedrosa

António Rui Borges

Universidade de Aveiro - DETI

2025-04-09

Logical Clocks

From Physical Time to Logical Clocks

Physical Synchronization Limitations

- **Time adjustment algorithms** introduce **variability** into the readings of local clocks in a distributed system.
- This makes it **impractical** to use physical time to **precisely synchronize activities** of different processes across nodes.
- Even with high-quality synchronization, the **best achievable precision** is usually within the **millisecond range**.
    - Within this interval, **millions of instructions** can be executed by each processor, making precise coordination difficult.

From Physical Time to Logical Clocks

The Insight from Lamport (1978)

- **Exact time agreement** between non-interacting processes **is not necessary**.
- If two processes **do not communicate**, **differences in their clocks** are **unobservable** and **irrelevant**.
- What **really matters** is that all processes **agree on the order** of **interacting events**.

From Physical Time to Logical Clocks

Logical Clocks

- Based on Lamport's insight, **logical clocks** were introduced.
- Logical clocks do **not track actual time**, but instead:
  - **Capture causality** and the **flow of information** between processes.
  - Enable consistent **event ordering**, which is essential for **correctness** in distributed applications.

Events and Their Ordering

What is an Event?

An **event** is any **relevant activity** that occurs during the **execution of a process**.

Among all types of events, **communication events**—particularly **message sending** and **message receiving**—are **especially important** for synchronization.

Events and Their Ordering

Fundamental Observations for Ordering Events

The ordering of events in a distributed system relies on two basic, but powerful, principles:

1. **Intra-process ordering**: If two events occur in the **same process**, they happen in the **order perceived by that process** (i.e., program order).
2. **Inter-process message causality**: If a **message is sent** from one process and **received** by another, the **send event must precede the receive event**.

Events and Their Ordering

Classification of Event Pairs

For ordering purposes, **event pairs** are classified as:

- **Sequential**: If it is **possible to determine** that one event **occurred before** the other.
- **Concurrent**: If it is **not possible to determine** any causal or temporal relationship—neither event can be said to have happened before the other.

Lamport's Happened-Before Relation

In his foundational work, **Leslie Lamport (1978)** introduced the concept of **event ordering** in distributed systems using a **partial ordering** relation called **"happened before"**, denoted as:

$$e \prec e'$$

This formalism extends the basic observations about event ordering and communication to a general model for distributed systems.

Lamport's Happened-Before Relation

Formal Definition of the Happened-Before Relation

Let e, e', e'' be events, and p_i be one of the processes $\{p_0, p_1, \ldots, p_{N-1}\}$, each executing on a distinct node of the distributed system. Then:

1. **Intra-process order** (local program order):

   If e and e' occur in the same process p_i, and e precedes e', then $e \prec e'$

2. **Message causality**:

   $$\text{If } e = \text{send}(m) \text{ and } e' = \text{receive}(m), \text{then } e \prec e'$$

3. **Transitivity**:

   $$\text{If } e \prec e' \text{ and } e' \prec e'', \text{then } e \prec e''$$

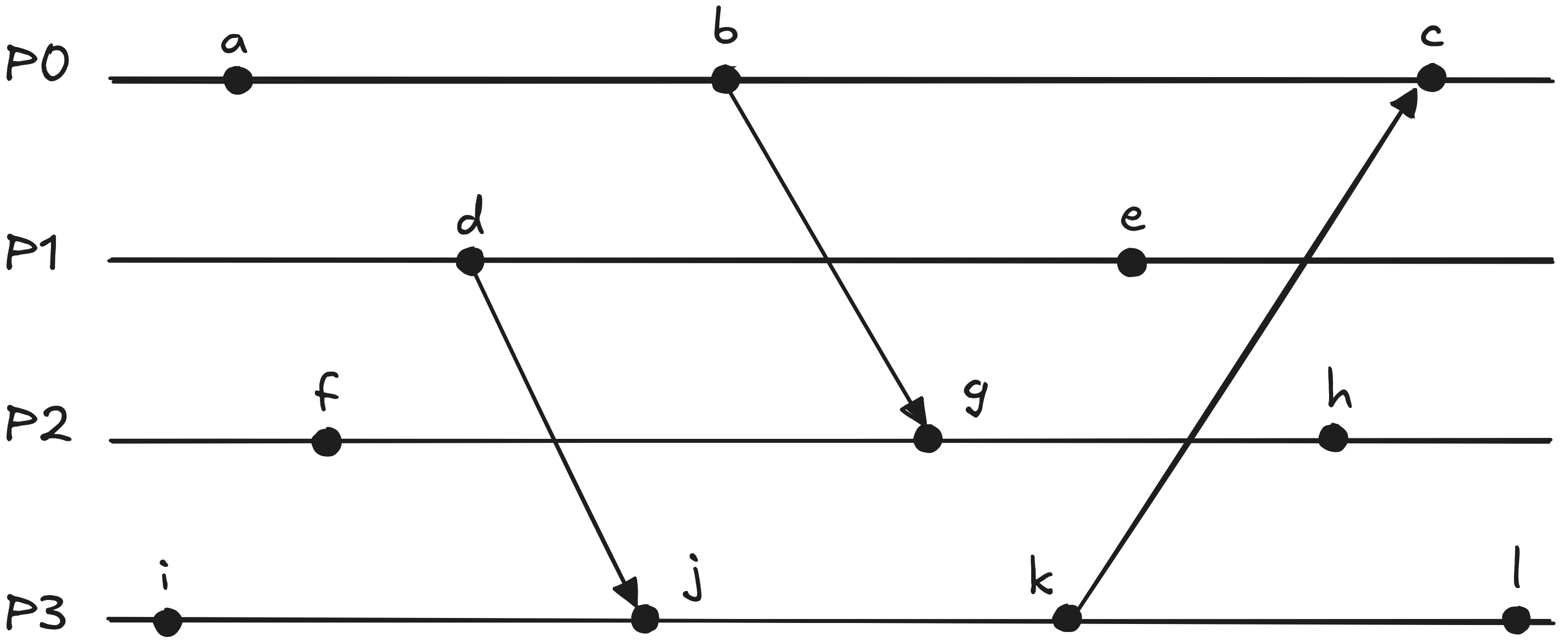
Lamport’s Happened-Before Relation

Interpretation

This relation $\prec$ defines a **partial order** over the set of all events in a distributed system:

- Some events are **comparable** (ordered causally or by process sequence).
- Others are **incomparable** (concurrent), meaning there is no causal or temporal relationship between them.

Lamport's Happened-Before Relation



- **Sequencial Events**
  - $f \prec g \wedge g \prec h \Rightarrow f \prec h$
  - $d \prec j \wedge j \prec k \wedge k \prec c \Rightarrow j \prec c$
- **Concurrent Events**
  - $\neg(f \prec c) \wedge \neg(c \prec f) \Rightarrow f \parallel c$
  - $\neg(i \prec e) \wedge \neg(e \prec i) \Rightarrow i \parallel e$

Scalar Logic Clock

To make the **"happened-before" $\prec$ relation** numerically explicit, **Lamport (1978)** proposed a mechanism called the **logical clock**.

A **scalar logical clock** is:

- A **local counter of events** maintained independently by each process.
- It is **monotonically increasing**.
- It does **not correspond to real-world (physical) time**.

Scalar Logic Clock

Logical Clock Rules (Per Process p_i)

Each process p_i, with $i = 0, 1, \ldots, N - 1$, maintains its own **logical clock** Ck_i, and updates it based on the following rules:

1. **Initialization**:

$$Ck_i = 0$$

2. **Local Event Occurrence**: When a process experiences a local (non-communication) event:

$$Ck_i := Ck_i + \alpha_i$$

where α_i is a constant (typically $\alpha_i = 1$).

Scalar Logic Clock

3. **Message Sending**:
   - The process first updates its clock: $Ck_i := Ck_i + \alpha_i$.
   - Then, it **attaches a timestamp** $\text{ts} = Ck_i$ to the message.
4. **Message Reception**:
   - Let ts be the timestamp received in the message.
   - The process sets:

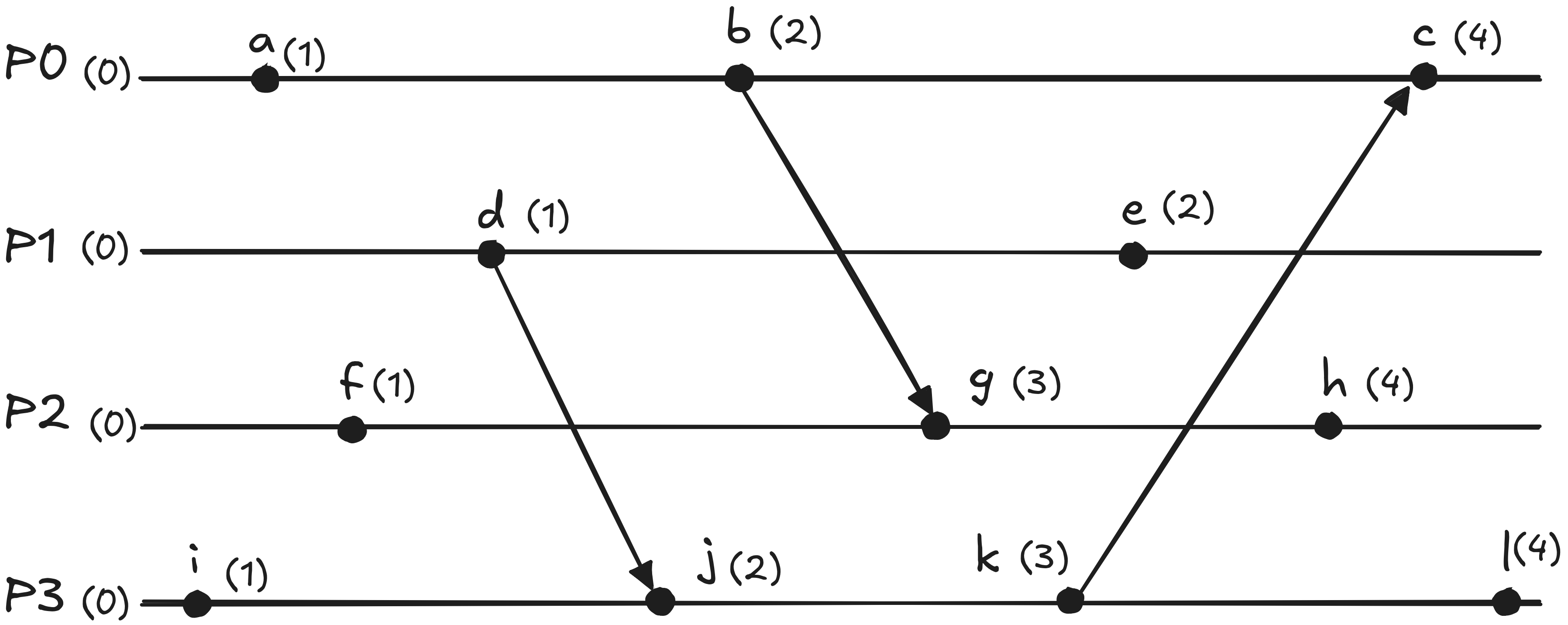
$$Ck_i := \max(Ck_i, \text{ts})$$

   - Then, it updates the clock for the receive event:

$$Ck_i := Ck_i + \alpha_i$$

This procedure ensures that **logical clocks respect the "happened-before" relation**, i.e., if $e \prec e'$, then $Ck(e) < Ck(e')$.

Scalar Logic Clock



Scalar Logic Clock

While **Lamport's logical clocks** guarantee a **partial order** of events (i.e., they respect the causal "happened-before" relation), **total ordering** of events is also possible under certain conditions.

Key Insight

Lamport showed that:

- Groups of **related events**, such as **message exchanges** between processes, can be assigned a **total order**—i.e., they can be perceived **in the same sequence by all processes** in a distributed system.
- This is possible **if each message includes a timestamp** generated using **logical clocks**, as he prescribed.

Scalar Logic Clock

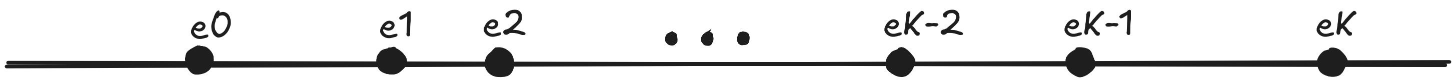
Total Ordering Condition

Let:

- e_j, with $j = 0, 1, \ldots, K - 1$, be events associated with the sending or receiving of messages m_j
- p_i, with $i = 0, 1, \ldots, N - 1$, be the participating processes

Then:

- The events e_j can be **totally ordered** if and only if there exists a **one-to-one mapping** between each event e_j and a **unique point on the numerical line**, based on a property such as the message's **logical timestamp**.



This enables a consistent **global ordering of events** across distributed nodes, even when physical clocks cannot be synchronized.

Scalar Logic Clock

When assigning **logical timestamps** to events in a distributed system, there may be cases where **two different messages** are given the **same timestamp**:

$$\mathrm{ts}(m_p) = \mathrm{ts}(m_q), \quad \text{with } p \neq q$$

To ensure a **deterministic total order** of events, Lamport proposed the use of an **extended timestamp**.

Scalar Logic Clock

Extended Timestamp

An **extended timestamp** is defined as the ordered pair:

$$(\mathrm{ts}(m), \mathrm{id}(m))$$

Where:

- $\mathrm{ts}(m)$: Logical timestamp of the message.
- $\mathrm{id}(m)$: Identifier of the process that sent the message.

This structure guarantees **unique ordering** even when timestamps are equal.

Scalar Logic Clock

Total Ordering Rule

Given two messages m_p and m_q, their extended timestamps are ordered as follows:

$$(\mathrm{ts}(m_p), \mathrm{id}(m_p)) < (\mathrm{ts}(m_q), \mathrm{id}(m_q)) \quad \Leftrightarrow$$

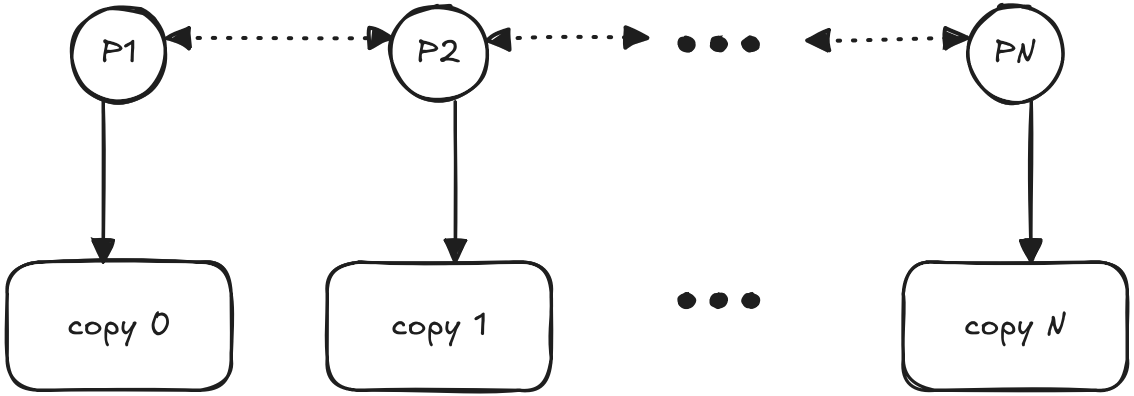
$$\mathrm{ts}(m_p) < \mathrm{ts}(m_q) \quad \text{or} \quad (\mathrm{ts}(m_p) = \mathrm{ts}(m_q) \wedge \mathrm{id}(m_p) < \mathrm{id}(m_q))$$

This **lexicographic ordering** ensures a **consistent and conflict-free total ordering** of events across the distributed system.

Scalar Logic Clock - Maintaining Synchronized Replicas

Scenario: A distributed application maintains N **replicas** of a **shared data region**, each located in a **different geographical site**, and accessed by a corresponding process p_i, for $i = 0, 1, \ldots, N - 1$.

Each process performs **read and write operations** on its local replica.



Scalar Logic Clock - Maintaining Synchronized Replicas

How can operations be organized so that all replicas remain permanently synchronized—**that is, they always reflect the same content across all registers?**

Fundamental Challenge:

Write operations **diverge** replicas unless a **synchronization protocol** is used to enforce **consistency** across all copies. This requires:

- **Consistent ordering** of updates
- **Reliable propagation** of write operations
- **Conflict resolution** (if updates occur concurrently)

Scalar Logic Clock - Maintaining Synchronized Replicas

Conditions for Permanent Synchronization of Replicated Data

To ensure that all replicated copies of a shared data region remain consistently synchronized across geographically distributed nodes, the following conditions must be met:

Synchronization Requirements

1. **Propagation Before Execution**:
   - Whenever a process p_i wants to **modify a register** in its local copy, it must **first propagate the operation** to **all other processes** managing the other replicas.
2. **Uniform Execution Order**:
   - All processes must **execute operations in the same order**, regardless of the order in which they are received.

Scalar Logic Clock - Maintaining Synchronized Replicas

System Assumptions:

To meet these requirements successfully, the system must assume that:

1. **No Process Failures**:
   - All processes p_i operate correctly throughout (i.e., no **crash** or **Byzantine failures**).

2. **No Message Loss**:
   - The **communication network is reliable**, ensuring that **all messages are delivered** without loss.

These assumptions are foundational for implementing **strong consistency** or **linearizability** in replicated systems.

Scalar Logic Clock - Maintaining Synchronized Replicas

Lamport's Algorithm for Replicated Data Synchronization

To ensure **permanent synchronization** and a **uniform execution order** of operations on replicated data, **Lamport proposed an algorithm** based on **logical clocks** and **message exchange**.

Key Idea:

Operations are **not executed immediately**; they are **broadcast**, **timestamped**, and **queued** until they can be executed in **globally consistent order**.

Scalar Logic Clock - Maintaining Synchronized Replicas

Algorithm Steps

1. **Intent to Modify** When process p_i determines that the next operation will **modify** a local register:
   - It **creates a message** describing the operation.
   - It **attaches a logical timestamp** from its local clock marking the event.

2. **Broadcast the Operation**
   - The message is **sent to all group members**, including itself.

Scalar Logic Clock - Maintaining Synchronized Replicas

3. **Message Handling**
   - Upon receiving the message, each process:
     - **Adjusts its logical clock** according to Lamport's rules.
     - **Inserts the message** into a **local priority queue**, ordered by **extended timestamps**.
4. **Acknowledgment**
   - Each process sends an **acknowledgment** to all members (including itself) for every received operation.

Scalar Logic Clock - Maintaining Synchronized Replicas

5. **Execution Condition**
   - A message (i.e., an operation) is **executed** by each process **only when**:
     - It is at the **head of the local queue**, **and**
     - **All acknowledgments** from group members have been received.

This guarantees that **every process executes operations in the exact same order**, achieving **strong consistency**.

Vector Logic Clock

Vector Logic Clock

Lamport's scalar logical clocks respect causality in one direction:

$$e \prec e' \Rightarrow Ck(e) < Ck(e')$$

But the **converse is not guaranteed**:

$$Ck(e) < Ck(e') \nRightarrow e \prec e'$$

That is:
If $Ck_{i(e_i)} < Ck_{j(e'_j)}$, it does **not necessarily mean** that event e_i happened before e'_j when $i \neq j$.

Vector Logic Clock

Solution by Mattern (1989) and Fidge (1991)

To overcome this limitation, they introduced a new type of logical clock known as the **vector clock**, which:

- Stores not only the **local history** of events,
- But also incorporates **partial knowledge** about the **other processes' clocks**.

This allows vector clocks to:

Accurately capture causal relationships between events across processes. Determine with certainty whether **two events are causally related or concurrent**.

Vector Logic Clock

Core Idea

- Each process maintains a **vector of counters** (one entry per process):
  - It updates its own component on local events.
  - It merges vector information from incoming messages.
  - Thus, each timestamp reflects a **causality-aware view of the system's state**.

Vector clocks let us say:

$$VC(e) < VC(e') \iff e \prec e'$$

and

$$VC(e) \nleq VC(e') \wedge VC(e') \nleq VC(e) \Rightarrow e \parallel e'$$

Vector Logic Clocks: Structure and Interpretation

In a system with N processes, a **vector clock** is a data structure used to capture **causal relationships** among events. It consists of a **collection of monotonically increasing counters**, with **no connection to real time**.

Data Structure

Each process p_i maintains its own **vector clock** V_i, implemented as an array of size N:

$$V_i = \left[V_{i[0]}, V_{i[1]}, \ldots, V_{i[N-1]}\right]$$

Vector Logic Clocks: Structure and Interpretation

Each element represents the **logical time** of one of the processes:

- $V_{i[i]} = Ck_i \longrightarrow$ The current **local logical clock** of process p_i.
- $V_{i[j]} = Ck_j$ (with $j \neq i$) $\longrightarrow$ The **most recent known value** of process p_j's clock, as **perceived by** process p_i, based on the **timestamps of messages received** from p_j.

Note: p_j may have executed more events and updated its clock further, but p_i won't know until it **receives a message** carrying updated information.

Vector Logic Clocks: Update Rules

Each process p_i maintains a **vector clock** V_i, which it updates in response to different types of events.

Update Rules

1. **Initialization**
   - At the start of execution, the vector clock is initialized as

$$V_{i[j]} = 0, \quad \text{for all } j = 0, 1, ..., N - 1$$

2. **Local Event**
   - When process p_i performs a local (non-communication) event:

$$V_{i[i]} := V_{i[i]} + \alpha_i$$

where α_i is a constant (typically $\alpha_i = 1$).

Vector Logic Clocks: Update Rules

3. **Message Sending**:
   - Before sending a message, process p_i updates its clock:

$$V_{i[i]} := V_{i[i]} + \alpha_i$$

   - It then attaches the entire vector clock V_i as the timestamp to the message.

4. **Message Reception**: Let ts be the vector timestamp received in the message. Then, process p_i updates its vector clock as follows:

$$V_{i[j]} := \max\left(V_{i[j]}, \mathrm{ts}[j]\right) \quad \text{for all } j = 0, 1, \ldots, N-1 \text{ and } i \neq j$$

   Then, it performs its **local update** for the reception event:

$$V_{i[i]} := V_{i[i]} + \alpha_i$$

Vector Logic Clocks: Comparison

In a system with N processes, vector timestamps are used to capture the **causal relationships** between events. Let V and V' be two vector timestamps. Their comparison follows these formal rules:

1. **Equality**:

$$V = V' \iff \forall 0 \leq j < N, V[j] = V'[j]$$

2. **Less than or equal (component-wise)**:

$$V \leq V' \iff \forall 0 \leq j < N, V[j] \leq V'[j]$$

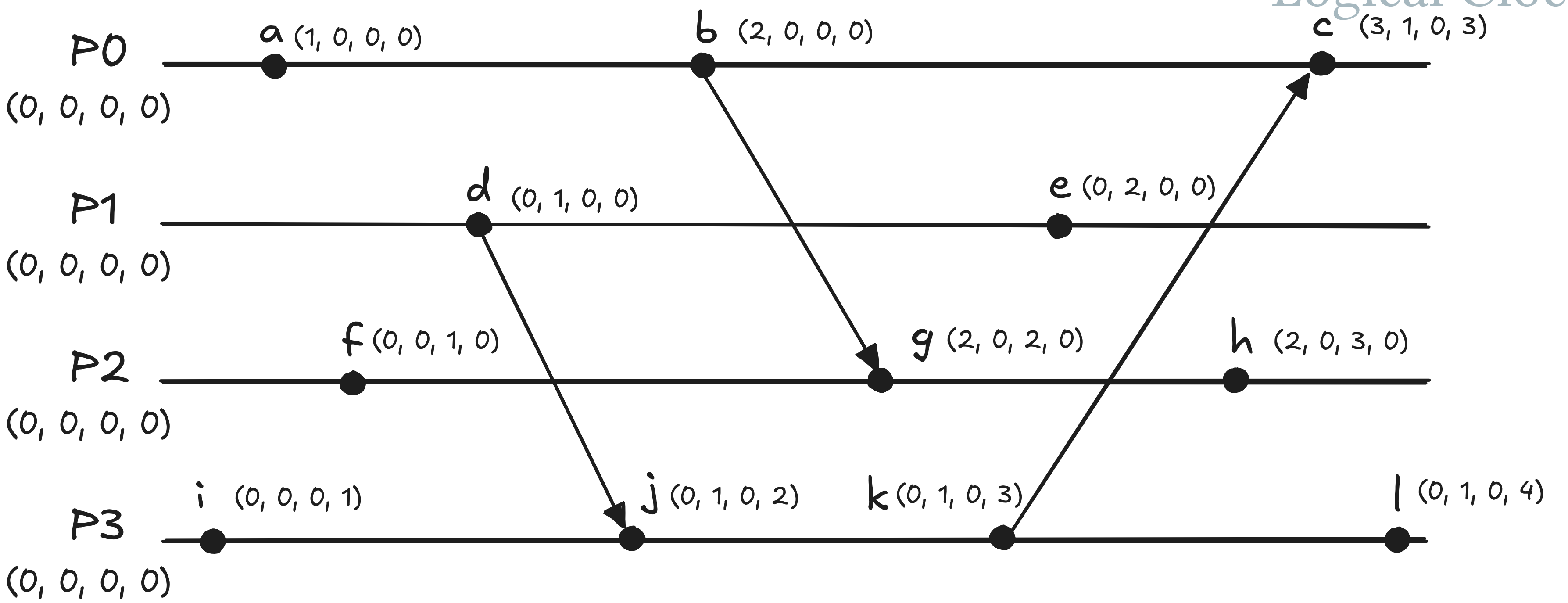
3. **Strictly less than**:

$$V < V' \iff V \leq V' \wedge V \neq V'$$

Vector Logic Clocks: Insights

- $V < V'$: The event with timestamp V **causally happened before** the event with timestamp V'.
- $V \parallel V'$ (i.e., **incomparable**): If neither $V < V'$ nor $V' < V$, the two events are **concurrent**—no causal relation exists between them.

Vector Logic Clocks: Example



- **Sequencial Events**
  - $f \prec h \Rightarrow V_2(f) < V_2(h)$
  - $d \prec c \Rightarrow V_1(d) < V_0(c)$
- **Concurrent Events**
  - $\neg[V_2(f) < V_0(c)] \wedge \neg[V_0(c) < V_2(f)] \Rightarrow f \parallel c$
  - $\neg[V_3(i) < V_1(e)] \wedge \neg[V_1(e) \prec V_3(i)] \Rightarrow i \parallel e$

Vector Clocks: Key Results

Let e and e' be two events occurring in processes p_i and p_j, respectively. Let $V_{i(e)}$ and $V_{j(e')}$ be their associated **vector timestamps**.

Key Property of Vector Clocks

$$e \prec e' \quad \Leftrightarrow \quad V_{i(e)} < V_{j(e')}$$

Where:

- $\prec$ denotes the **“happened-before” relation** (as defined by Lamport).
- $<$ denotes the **strict vector clock comparison**.

This means that:

- If e causally **happened before** e', then $V_{i(e)} < V_{j(e')}$.
- Conversely, if $V_{i(e)} < V_{j(e')}$, then $e \prec e'$.

Vector Clocks: Key Results

Unlike Lamport's scalar clocks (which only preserve one direction of implication), **vector clocks fully characterize causality**.

Suggested Reading

- M. van Steen and A.S. Tanenbaum, Distributed Systems, 4th ed., distributed-systems.net, 2023.