Distributed Computing

Introduction
Types
Logical Clocks
Lamport's Timestamp
Vector Clock
Singhal–Kshemkalyani differential technique
Fowler–Zwaenepoel’s Direct-Dependency Technique
Physical Clock Synchronisation
Causal Ordering
NTP

Introduction to Distributed Computing

Understanding the Idea

Distributed computing is the science (and art) of making **multiple independent computers** work together so well that, from the user's perspective, they behave like a **single coherent system**.

The difficulty is not in connecting machines — we have networks for that — but in **coordinating** them so that:

- They give correct results.
- They hide internal complexity.
- They tolerate failures.

Real-world analogy: An international airline operates flights from multiple hubs. Passengers don't need to know which airport is handling baggage or which city the flight plan was generated in — the airline appears as one unified service.

Definition

A distributed system is a collection of autonomous computers that appear to its users as a single coherent system.

Key points:

- **Autonomous** → Each computer (node) has its own CPU, memory, and operating system.
- **Independent** → No shared physical memory or single system clock.
- **Coherence** → The system's behaviour should be indistinguishable from that of a single machine.

Characteristics

1. **No Shared Global Clock**
    - Each node's internal clock may drift differently.
    - Makes it hard to know *exactly* when something happened relative to events on other nodes.
2. **Independent Failures**
    - One node can crash without bringing down others.
    - The system must detect and mask such failures.
3. **Concurrency**
    - Multiple nodes execute processes simultaneously.
    - Race conditions are possible if interactions aren't managed.
4. **Geographical Distribution**
    - Nodes may be thousands of kilometres apart.
    - Network latency and bandwidth become important factors.

Example: Google search results are compiled by many machines worldwide, yet presented instantly and consistently.

Advantages

- **Scalability**: Can grow capacity by adding nodes.
- **Fault Tolerance**: Redundancy keeps service available during failures.
- **Resource Sharing**: Expensive resources can be shared across sites.
- **Performance**: Parallel processing speeds up large computations.

Example: Pixar renders animated films using render farms — each frame may be processed by a different machine.

Challenges

- **Synchronization**: No global clock; must use logical ordering.
- **Communication**: Messages can be delayed, lost, or arrive out of order.
- **Consistency**: Maintaining identical copies of data across nodes under concurrent updates is difficult.
- **Fault Recovery**: Restoring state after failure without data loss.

Types of Distributed Systems

1. **Client–Server**: Clear separation between requesters and providers.
2. **Peer-to-Peer**: All nodes can be both client and server.
3. **Cluster Computing**: Tightly connected computers in one location.
4. **Grid Computing**: Loosely connected, heterogeneous resources.
5. **Cloud Computing**: On-demand resources delivered over the Internet.

Logical Clocks

Why Clocks Matter

In distributed systems, we care about *when* events happen, but:

- No single shared clock exists.
- Hardware clocks drift.
- Network delays distort message timing.

We need mechanisms to **order events** without relying on perfectly synced real time.

Four Types of Clocks

1. **Physical Clocks**
   - Reflect real-world time (UTC).
   - Synchronised using protocols like NTP or GPS signals.
   - Used for timestamps visible to users, legal logs.
2. **Logical Clocks**
   - Abstract counters used to order events consistently.
   - Ignore actual wall-clock time.
3. **Hybrid Clocks**
   - Combine physical time with logical counters.
   - Example: Google Spanner's TrueTime API.
4. **Vector Clocks**
   - Track causality explicitly by keeping separate counters for each process.

Scalar Time (Lamport Timestamps)

Goal:

To assign a **scalar time** (just a single number) to each event in such a way that:

- If event **A** happens before **B** in real life, then the timestamp of **A** is less than that of **B**.
- Events are totally ordered (no ties), even if they are unrelated.

Rules:

1. **Before any local event** (something happening inside the same process, such as computation or sending a message), **increment the local clock**.
2. **When sending a message**, attach your current clock value to the message.
3. **When receiving a message** with timestamp `C_received`, update your clock to:

$$C = \max(C_{\text{local}}, C_{\text{received}}) + 1$$

Lamport Timestamps Example

Given:

- Two processes: P1 and P2
- Both start with a local clock value of **C = 1**.

Step-by-step:

1. P1 local event A:
   P1 increments clock from 1 to 2 before sending a message.
2. P1 sends message:
   Sends message with timestamp (2) to P2.
3. P2 receives message:
   - P2's local clock is still 1 before receiving.
   - Applies formula:

$$C_{\mathrm{P2}} = \max(1, 2) + 1 = 3$$

Now P2's clock is 3.

Elaborate Example Setup

Processes: P1, P2, P3

All start with Lamport clock `C = 1`.

Rule recap: increment before every local event (including **send**), attach the clock on send, and on **receive** set `C := max(C_local, C_received) + 1`.

```
P1:  C1=1
e1: local compute              (increment)          C1=2
e2: send m1 -> P2              (increment+attach) C1=3,  m1.ts=3
e3: local compute                                   C1=4

P3:  C3=1
e4: local compute              (increment)          C3=2
e5: send m2 -> P2              (increment+attach) C3=3,  m2.ts=3
                                                    (e2 and e5 are concurrent)

P2:  C2=1
e6: local compute              (increment)          C2=2
e7: receive m1(ts=3)           (max(2,3)+1)         C2=4
e8: send m3 -> P3              (increment+attach) C2=5,  m3.ts=5
e9: receive m2(ts=3)           (max(5,3)+1)         C2=6

P3:
e10: receive m3(ts=5)          (max(3,5)+1)         C3=6
```

Limitation

In the scenario:

1. P1 sends m1 at time 3
2. P3 sends m2 at time 3

These two “send” actions are concurrent —

- P1 doesn’t know P3 is sending something.
- P3 doesn’t know P1 is sending something.
  There is no cause-and-effect link between them.

But when P2 receives these messages:

- The first one to arrive gets a smaller Lamport time.
- The second one to arrive gets a larger Lamport time.

From the timestamps alone, you can’t tell that the sends were unrelated — it just looks like one happened “before” the other because the clock numbers are different.

Why this is a problem

- Lamport timestamps **force a total order**: they arrange *all* events into a single sequence, even if real-world time says they were independent.
- That means you **lose information about concurrency**.
- If you need to know “Did these events happen independently?”, Lamport timestamps won’t help — you’d need **vector clocks** or a similar mechanism.

Vector clocks

Goal

Vector clocks keep track of **what each process knows about everyone else's history**.
This lets you say:

- **A happened before B** (causality)
- **A and B are concurrent** (unrelated — they didn't know about each other)

The rules

Imagine each process keeps a **scoreboard** with one slot for each process in the system:

- **My own score** = how many events I've personally seen or done.
- **Other scores** = the last known count of events from those processes.

1. **Local event** → increase *your own* score by 1.
2. **Send a message** → include your entire scoreboard.
3. **Receive a message** →
    - Compare each scoreboard slot with yours and take the bigger number.
    - Then increase *your own* score by 1.

Structure

For a system of **n processes**:

- Each process p_i maintains a **vector clock**:
  $vt_i = [vt_i(1), vt_i(2), \ldots, vt_i(n)]$
  - $vt_i(i)$: The **local logical clock** of p_i.
  - $vt_i(j)$: The latest knowledge p_i has of p_j's logical time.
- The entire vector represents p_i's **view of the global logical time.**

Rules for Updating Vector Clocks

Initial State

- All clocks are initialised to zero:
  $vt_i = [0, 0, \ldots, 0]$.

R1 – Internal Event

- Before executing any event (internal computation, sending a message, etc.):
  $vt_i(i) = vt_i(i) + d$, where $d > 0$ (usually $d = 1$).

R2 – Receiving a Message

When p_i receives a message m with vector clock vt_m from sender p_s:

1. **Merge Clocks:**
   For all k from 1 to n:
   $vt_i(k) = \max(vt_i(k), vt_m(k))$
2. **Increment Local Clock:**
   $vt_i(i) = vt_i(i) + d$ (same as R1).
3. **Deliver the message.**

Sending a Message

- A message is **piggybacked** with the sender's vector clock at send time.

Interpretation

- If $vt_i(j) = x$, it means **process p_i knows that process p_j's logical time has advanced to x.**
- **Event Ordering:**
  - Event **a** happens before **b** if:
    $vt_a < vt_b$ (vector comparison: all components ≤ and at least one <).
  - If neither $vt_a < vt_b$ nor $vt_b < vt_a$, events are **concurrent**.

Scenario with P1, P2, P3

Start:

P1: [0,0,0]

P2: [0,0,0]

P3: [0,0,0]

P1 does something (local event)
`P1: [1,0,0]`

P1 sends to P2
P2 merges: max of each slot → `[1,0,0]`
P2 increments its own slot → `[1,1,0]`

P2 sends to P3
P3 merges: `[1,1,0]`
P3 increments its own slot → `[1,1,1]`

Singhal–Kshemkalyani differential technique

The **Singhal–Kshemkalyani differential technique** is an optimisation of vector clocks aimed at reducing the communication overhead.

Observation

- In a distributed system, when one process repeatedly sends messages to the same other process, **most entries in the vector clock remain unchanged** between two consecutive sends.
- Only a **few vector clock entries** (corresponding to processes that had relevant events) will have updated values.

Why This Happens More with Large Systems

Between two consecutive messages from pip_ipi to pjp_jpj, **only some entries in pip_ipi's vector clock change**.
This happens because:

- In large distributed systems, not all processes interact frequently.
- The logical time of unrelated processes stays the same between sends.

Goal

Reduce:

- **Message size** (fewer timestamp entries sent)
- **Communication bandwidth**
- **Buffer requirements** (less storage needed for in-transit messages)

How the Differential Technique Works

1. **Track last sent clock**:

   Each process p_i keeps a record of the vector clock it last sent to p_j.
2. **When sending a message**:
   - Compare the **current** vector clock to the **last sent** vector clock for p_j.
   - Send **only the changed entries** (entry index + new value), instead of the entire n-length vector.
3. **On receiving**:
   - p_j updates only those vector entries that were received.
   - Missing entries are assumed unchanged since last update from p_i.

Worst Case

- If **all entries** have changed since last send → must send **full vector clock** of size n.
- This is rare in practice.

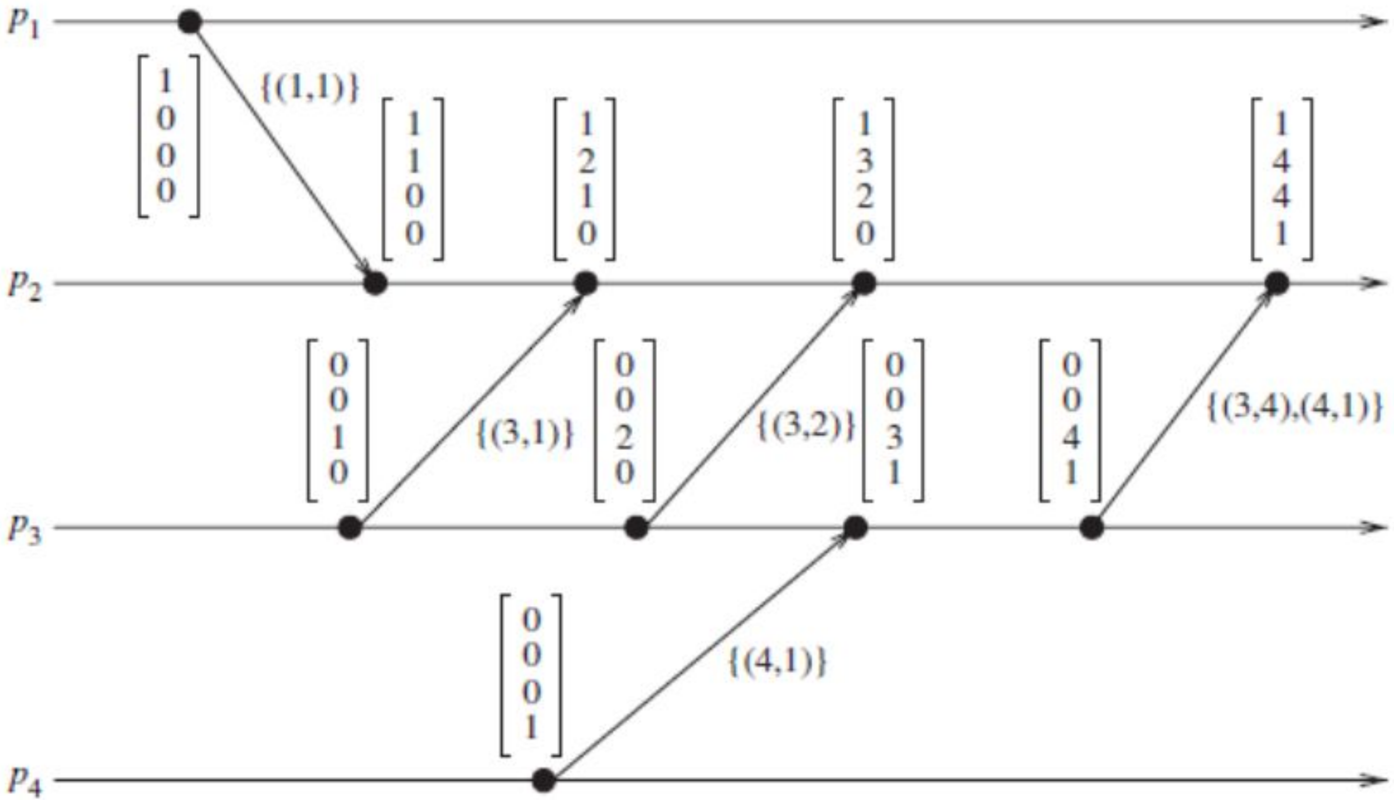
Average Case

- Usually, **only a few entries change**, so the size of the timestamp on a message is **less than n**.

Benefit

- **Saves bandwidth**: Instead of sending an n-element vector clock, only a small set of changed entries is transmitted.
- This becomes significant when n is large and the frequency of change is low.

Singhal–Kshemkalyani's differential technique Example



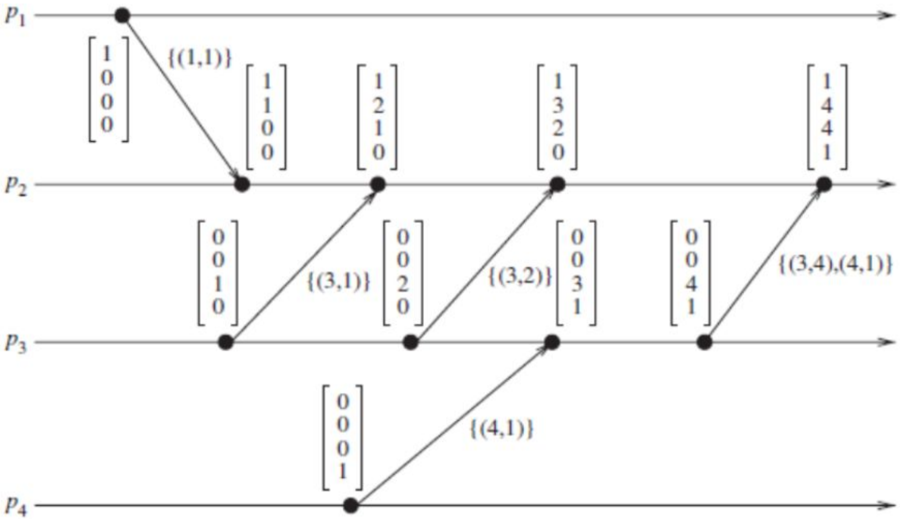
Setup

- **4 processes:** p_1, p_2, p_3, p_4
- Each keeps a **vector clock** of length 4.
- Initially, all vector clocks are `[0,0,0,0]`.
- Notation `{(x,y)}` means:
  - "Send only index x with value y" (instead of full vector clock).

Step-by-Step Execution

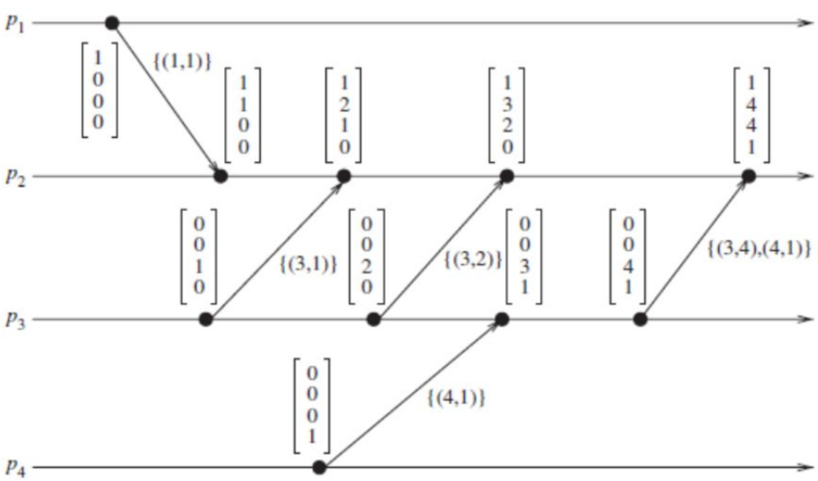
Step 1 – p_1 internal event

- p_1 increments its own clock:
  `[1,0,0,0]`
- Sends a message to p_2.
  Last sent vector to p_2 was `[0,0,0,0]`.
  Changed entry: index 1 changed from 0 → 1.
  Send `{(1,1)}`.



At p_2:

- Start from its current clock `[0,0,0,0]`.
- Update entry 1 to 1 → `[1,0,0,0]`.
- Increment local entry 2 (own process index) → `[1,1,0,0]`.



Rule R1 – Internal Event or Before Executing an Event

Before a process p_i executes **any event** (internal computation, sending a message, or after merging a received vector), it must:

$$vt_i(i) = vt_i(i) + d$$

where typically $d = 1$.

Why we increment the "own process index"

- Each vector clock entry corresponds to a process.
- The i-th entry in vt_i is **that process's own logical time**.
- Incrementing it signals **progress in local time** — meaning an event has happened at that process.

Example

If p_2 receives a message and merges vector clocks,
before finishing that event, it **increments its own entry (index 2)**:

From:

$$[1, 0, 0, 0] \quad \text{(after merging)}$$

Increment entry 2:

$$[1, 1, 0, 0]$$

This ensures:

- Causality is preserved
- The timestamp reflects that p_2 performed an event (the message receipt)

p3 internal event → `[0,0,1,0]` → sends to p2 with `{(3,1)}`.

p2 merges: max of `[1,1,0,0]` and `{(3,1)}` → `[1,1,1,0]` → increments own entry → `[1,2,1,0]`.

p2 internal event → `[1,3,1,0]` → sends to p3 with `{(3,2)}` (entry 3 from 1 → 2).

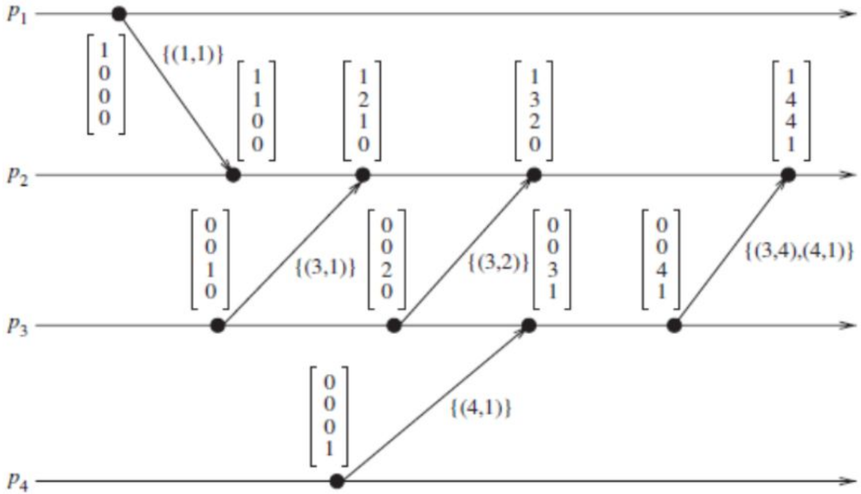
p3 merges with its own `[0,0,1,0]` → `[1,3,2,0]` → increments own entry → `[1,3,3,0]`.

p4 internal event → `[0,0,0,1]` → sends to p3 with `{(4,1)}`.

p3 merges `[1,3,3,0]` with `{(4,1)}` → `[1,3,3,1]` → increments own → `[1,3,4,1]`.

p3 sends to p1 with `{(3,4),(4,1)}` (two entries changed since last send to p1).

p1 merges `[1,0,0,0]` with `{(3,4),(4,1)}` → `[1,0,4,1]` → increments own entry → `[2,0,4,1]`.



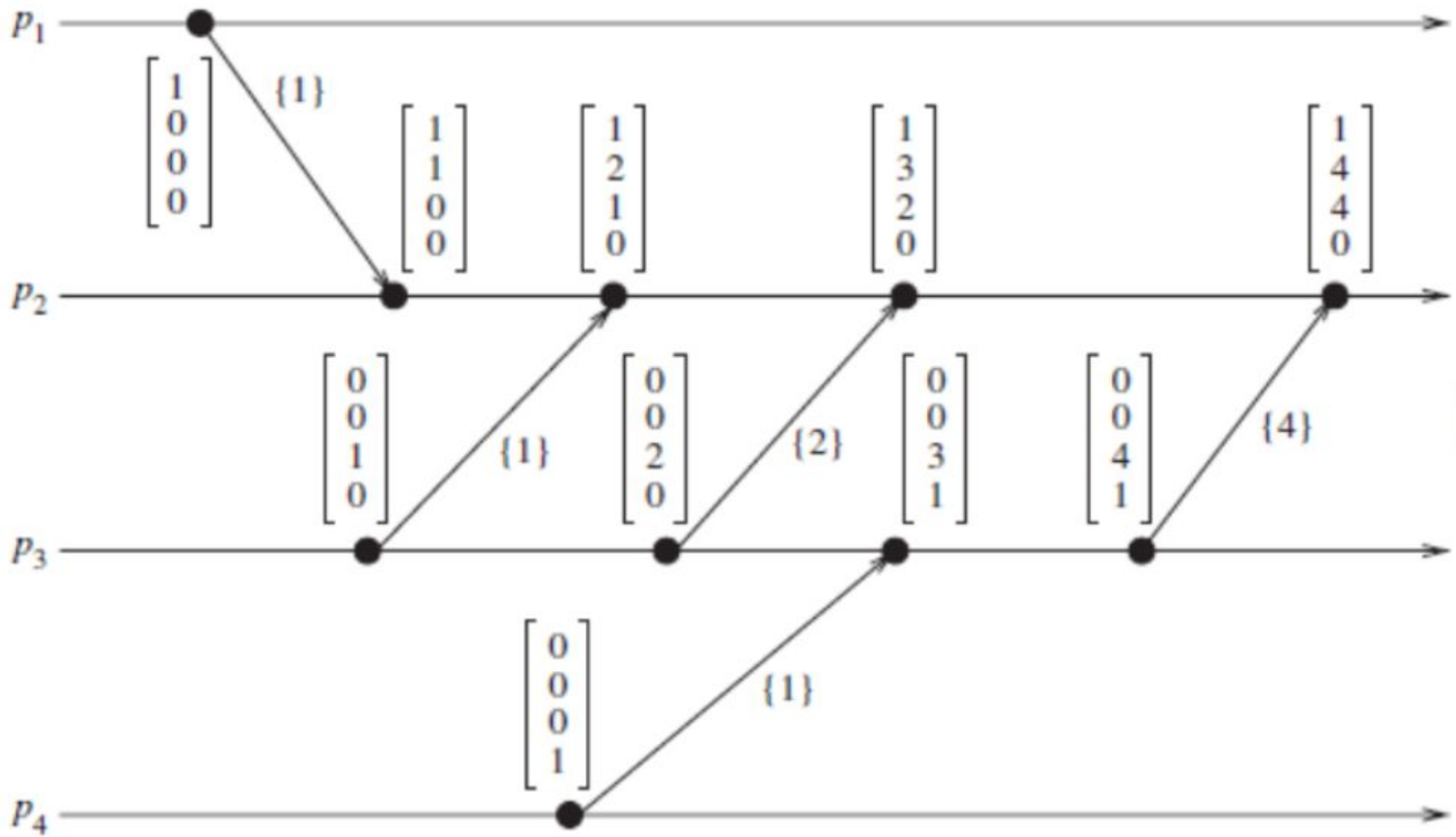
Fowler–Zwaenepoel's Direct-Dependency Technique

Purpose

To **further reduce the runtime overhead** of tracking causality in distributed systems compared to both:

- **Basic vector clocks** (which send full nnn-entry vectors), and
- **Singhal–Kshemkalyani's differential technique** (which sends only changed entries).

Fowler–Zwaenepoel (FZ) goes further by **eliminating the need to send any vector clock entries at runtime** — instead, only a **single scalar value** is transmitted, and the full vector is reconstructed later.



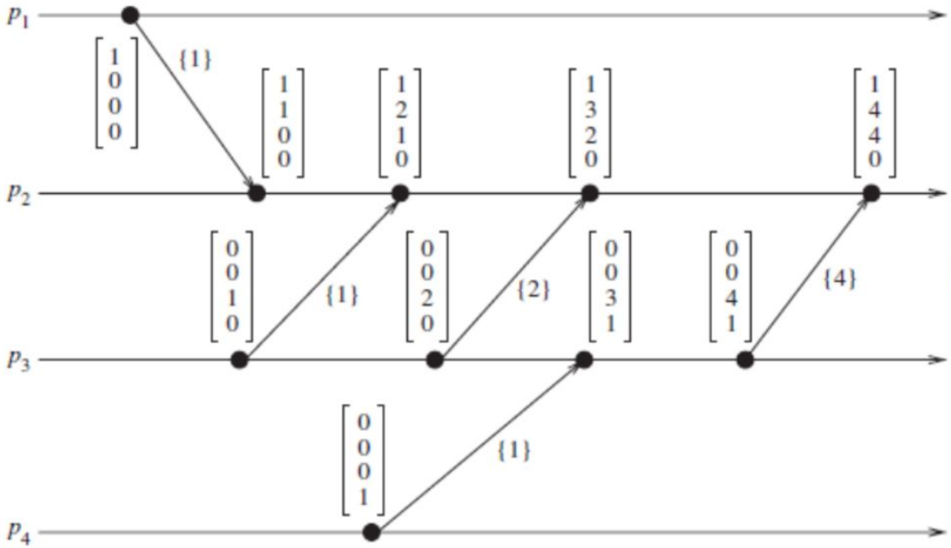
P_1
P_2
P_3
P_4
{1}
{1}
{2}
{4}
{1}
[1 0 0 0]
[1 1 0 0]
[1 2 1 0]
[1 3 2 0]
[1 4 4 0]
[0 0 1 0]
[0 0 2 0]
[0 0 3 1]
[0 0 4 1]
[0 0 0 1]

Setup

Four processes p_1, p_2, p_3, p_4. In **Fowler–Zwaenepoel (FZ)** we **don't ship vector clocks**. Each message carries only a **scalar** (writer's local counter), shown in braces: `{1}`, `{2}`, `{4}`.

Each receiver simply records a **direct dependency** "this event at me depends on that sender@scalar".

The column vectors drawn above events in the figure are **not sent**; they show what the **full vector time would be if you reconstructed it offline** from those dependencies.



Breaking it down

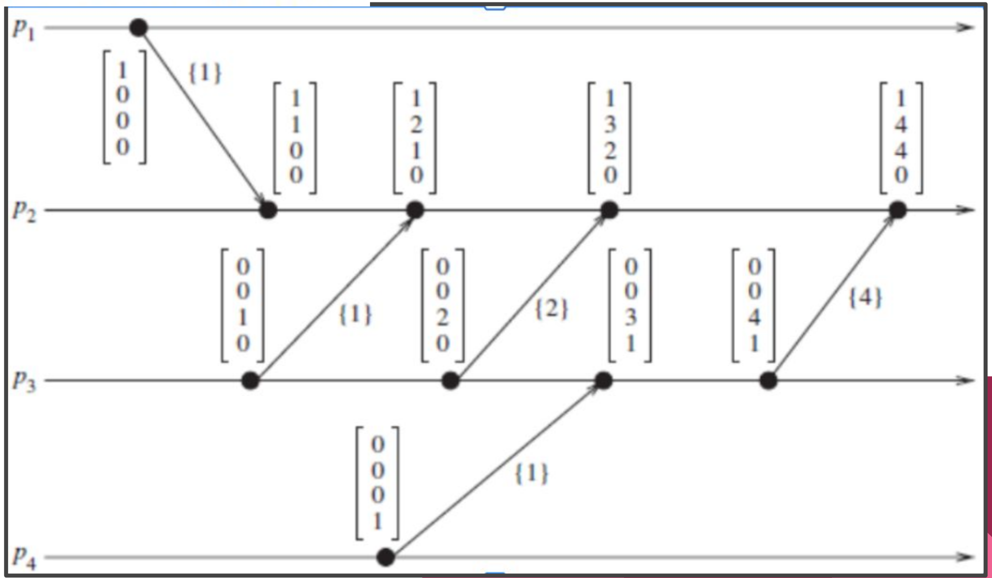
1) $p_1 \rightarrow p_2$ with `{1}`

p_1 performs one local event (counter becomes 1) and sends to p_2 tagged `{1}`.

p_2 records a direct dependency on $p_1@1$.

Offline view of that receive at p_2: $[1, 1, 0, 0]$.

(Intuition: p_2 has seen one thing from p_1, and this receive is a local event at p_2.)



2) $p_3 \to p_2$ with `{1}`

p_3 performs one local event (counter 1) and sends `{1}` to p_2.

p_2 now also depends directly on $p_3@1$.

Offline view of this later point at p_2 becomes $[1, 2, 1, 0]$: its knowledge of p_1 is still 1, p_3 is 1, and p_2 has advanced again.

3) $p_2 \to p_3$ with `{2}`

By now p_2's local scalar is 2, so it sends `{2}` to p_3.

p_3 records a direct dependency on $p_2@2$.

Offline view of that receive at p_3: it now knows "up to $p_2 = 2$ and p_3 performs a receive event", so its reconstructed vector jumps accordingly (shown in the boxes along p_3 in your figure).

4) $p_4 \to p_3$ with `{1}`

p_4 does its first local event and sends `{1}` to p_3.

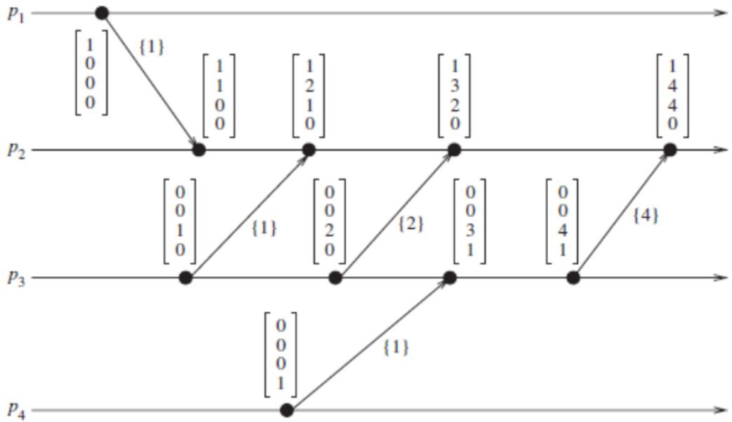
p_3 records a direct dependency on $p_4@1$.

Offline view at p_3 now includes a non-zero 4th component (it has evidence of one event at p_4).

5) $p_3 \to p_2$ with `{4}`

After its two receives and a send, p_3's scalar has reached 4, so it sends `{4}` to p_2.

p_2 records a direct dependency on $p_3@4$.



Physical Clock Synchronization

Centralized Systems

In a **centralized system**, clock synchronization is not a problem because there is usually only one clock for the entire system.

- Processes simply query the kernel for the current time, ensuring a **single, consistent notion of time**.
- If one process retrieves the time and another does so immediately after, the second will always get a **later time value**.
- This natural ordering means there is **no ambiguity** in event timestamps—ordering is guaranteed.

Distributed Systems

In a **distributed system**, the situation is very different:

- There is **no global clock** and no shared memory.
- Each processor has **its own internal clock** and its own idea of time.
- These clocks can **drift** apart over time due to minor differences in their oscillators, sometimes by **seconds per day**.
- Different clocks tick at slightly **different rates**, meaning even if they start synchronized, they will **gradually diverge**.
- This clock drift can cause serious issues for applications that rely on consistent timestamps—for example, **distributed databases, logging systems, and authentication protocols**.

What is Clock Synchronization?

Clock synchronization ensures that all physically distributed processors share a **common notion of time**.

- It is critical for:
  - **Security systems** (e.g., time-based authentication, certificate validity checks).
  - **Fault diagnosis & recovery** (accurate ordering of failure events).
  - **Scheduled operations** (e.g., batch jobs, backups).
  - **Database consistency** (ordering transactions).
  - **Timeout-based protocols** (accurate timeouts depend on correct clock sync).
- Good synchronization **simplifies application design** because developers can trust that timestamps across machines are consistent.

Physical Clocks

- In distributed systems, clocks need to be synchronized both with each other and with an external real-world reference like UTC (Coordinated Universal Time).
- These clocks are called physical clocks. Due to drift, synchronization must be performed periodically to correct for clock skew (the divergence between clocks).

Why Physical Clocks When Logical Clocks Already Exist?

Logical clocks (e.g., Lamport clocks, vector clocks) provide a way to **order events** in a distributed system without relying on physical time. They solve the **"happens-before"** problem and guarantee a consistent **causal ordering** of events.

However, **logical clocks alone are not enough** for many real-world applications:

1. **No real-world meaning**: Logical clocks produce numbers that indicate order, but they do not correspond to actual wall-clock time or real-world schedules.
2. **External interaction**: If a system interacts with the outside world (e.g., logging, scheduling with humans, coordinating with other organisations), we must map events to **real physical time**.
3. **Timeouts and deadlines**: Applications like leases, authentication tokens, or retry timers require a **real duration** in seconds/minutes, not just an event ordering.
4. **Legal and compliance needs**: Financial transactions, medical records, or audit logs require timestamps in **absolute time** for accountability.
5. **Mixed systems**: Many systems use both physical and logical clocks—physical clocks for **real-time measurements** and logical clocks for **causal consistency**.

Logical clocks are great for reasoning about causality, but physical clocks are essential when you need your timestamps to have **real-world meaning**.

Causal Ordering in Distributed Systems

Definition

Causal ordering ensures that if one event **causes** another, all processes in the system agree on that order.
Formally:
If **event A** → **event B** (A happens-before B), then every process must observe A **before** B.

Why It Matters

- Maintains **logical consistency** across distributed processes.
- Preserves **cause-and-effect relationships** in message passing.
- Prevents anomalies like **reading a reply before seeing the request**.

Happens-Before Relation (→)

1. **Within the same process**: If `A` occurs before `B`, then `A → B`.
2. **Message passing**: If a message is sent in event `A` and received in event `B`, then `A → B`.
3. **Transitivity**: If `A → B` and `B → C`, then `A → C`.

Example

1. **P1** sends message `m1` to **P2**.
2. **P2** processes `m1` and sends `m2` to **P3**.
3. All processes must see `m1` before `m2` to preserve causality.

How to Achieve Causal Ordering

- **Logical Clocks** (Lamport clocks) → Ensure consistent event ordering.
- **Vector Clocks** → Track causality precisely by maintaining per-process counters.

Network Time Protocol (NTP)

NTP is the most widely used protocol for **synchronizing clocks across the Internet**.

- **Method**: Uses **offset delay estimation** to calculate the time difference and network delay between a client and server.
- **Architecture**:
  - **Root level**: Primary servers that synchronize directly with **UTC** (via atomic clocks, GPS clocks, or radio signals).
  - **Secondary level**: Secondary servers that get time from the primary servers and act as backups.
  - **Lowest level**: Client systems in the synchronization subnet that query servers for the current time.
- The design is **hierarchical**, which improves scalability and avoids overloading the primary servers.

Benefits of the Hierarchical Design

- **Scalability**: Reduces load on primary servers by distributing queries.
- **Fault Tolerance**: Secondary servers act as backups.
- **Accuracy**: Minimizes network delay impact via offset-delay estimation.