Logical Time

Distributed Systems

Ali Kamandi, PH.D.

School of Engineering Science

College of Engineering

University of Tehran

kamandi@ut.ac.ir

2024

Applications

Distributed algorithms design:

- deadlock detection
- replicated databases

Tracking of dependent events:

- distributed debugging
- build a checkpoint in replicated databases

Knowledge about the progress:

- garbage collection
- termination detection

Concurrency measure:

- all events that are not causally related can be executed concurrently

Need of Logical Clock

In **real life**, the global time is obtained from loosely synchronized clocks.

In distributed systems, the **rate of occurrence** of events is several magnitudes higher and the **event execution time** is several magnitudes smaller.

If the physical clocks are not **precisely synchronized**, the causality relation between events may not be accurately captured.

Network time Protocol, can maintain time accurate to a few tens of milliseconds on Internet, are not adequate to capture the causality relation in distributed systems.

Logical Clock

Each process has a **logical clock** that is advanced using a set of rules.

Every event is assigned a timestamp and the causality relation between events can be generally inferred from their timestamps.

Monotonicity property:

If an event **a** causally affects an event **b**, then the timestamp of **a** is smaller than timestamp of **b**.

General Definition of Logical Clock

Logical clock C is a function that maps an event *e* to an element in the time domain T.   C(e) called the timestamp of e.

$$C: H \longrightarrow T$$

Such that the following property is satisfied:

For two events e_i and e_j : $e_i \longrightarrow e_j \implies C(e_i) < C(e_j)$

This monotonicity property is called the **clock consistency condition**. When T and C satisfy the following condition,

For two events e_i and e_j : $e_i \longrightarrow e_j \iff C(e_i) < C(e_j)$

The system of clocks is said to be **strongly consistent**.

Implementing logical clock

Data structure for each process and protocol to update it.

- Process: p_i
- Local logical clock: lc_i
- Global logical clock: gc_i

Rule 1:

How the local logical clock is updated by a process when it executes an event.

Rule 2:

How a process updates its global logical clock to update its view of the global time.

Scalar time: Lamport Clock (1978)

Time domain: set of non-negative integers.

Rule 1:

Before executing an event (send, receive or internal) process P_i executes the following:

$$C_i \coloneqq C_i + d \qquad (d > 0).$$

In general, every time **Rule 1** is executed, d can have a different value.

Scalar time: Lamport Clock (1978)

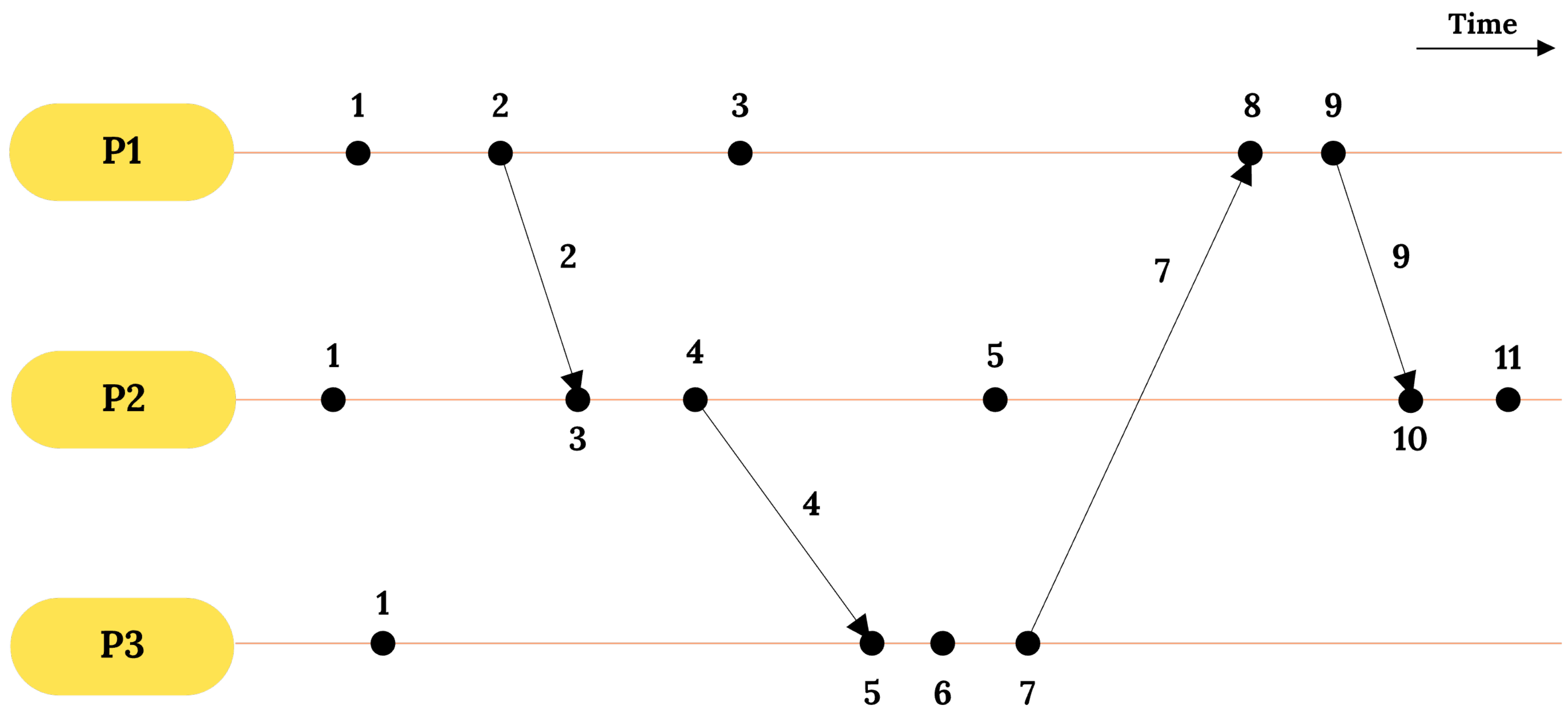
Rule 2:

Each message piggybacks the clock value of its sender at sending time. When a process P_i receives a message with timestamp C_{msg}, it executes the following actions:

- $C_i := \max(C_i, C_{msg})$;
- Execute **Rule 1**;
- Deliver the message;

Example

Evolution of scalar time with **d=1** :



Consistency Property

Clearly, scalar clocks satisfy the monotonicity and hence the consistency property:

For two events e_i and e_j : $e_i \longrightarrow e_j \implies C(e_i) < C(e_j)$

Total ordering

Scalar clocks can be used to totally order events.

Two or more events at different processes may have an identical timestamp.

$$C(e_1) = C(e_2) \implies e_1 \parallel e_2$$

Timestamp of an event is denoted by a tuple (t,i), t is its time of occurrence and i is the identity of the process.

x : (h,i)

y : (k,j)

$$\boldsymbol{x \prec y \iff \left(h < k \text{ } or \text{ } (h = k \text{ } and \text{ } i < j)\right)}$$

$$\boldsymbol{x \prec y \implies x \rightarrow y \vee x \parallel y}$$

Event counting

If we set d to 1:

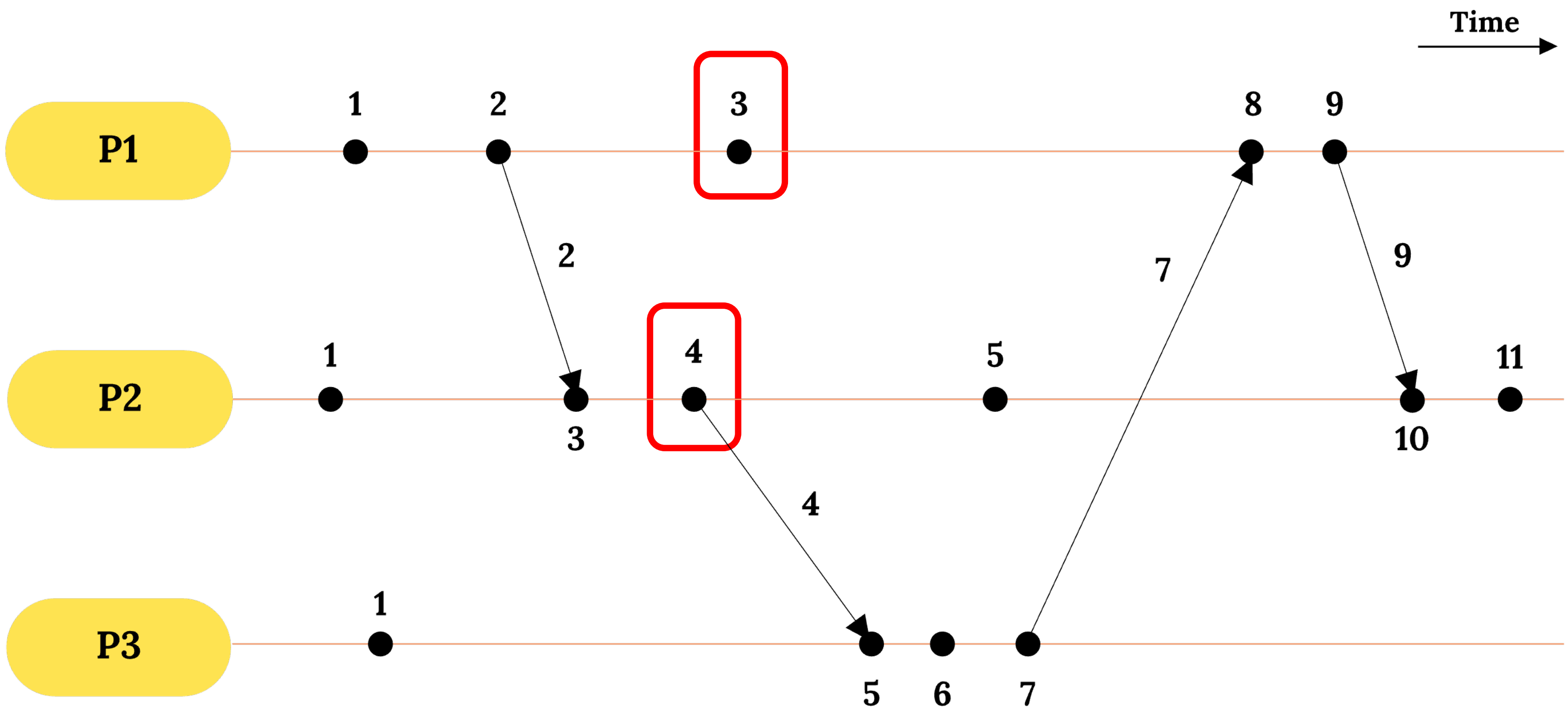
Timestamp h: h -1 is the minimum logical duration, height of the event h.

h – 1 events precede event e on the longest causal path ending at e.

h -1 events have been produced before e.

No Strong Consistency

$$C(e_i) < C(e_j) \implies e_i \nRightarrow e_j$$



Neiger-Toueg-Welch clock

Disadvantage of Lamport clock: **huge jump**.

Scheduled tasks: every 100 clock ticks.

Each process has its own clock and increments the clock whenever it feels like it.

Clock can be the physical clock.

(clock, id, eventCount)

- eventCount: count of send/receive events.

This protocol can't change the clock value.

When a message is received with a timestamp later than the current extended clock value, its delivery is delayed until clock exceeds the message timestamp, at which point the receive event is assigned the extended clock value of the time of delivery.

Neiger-Toueg-Welch clock

If some process’s clock is too far off, it will have trouble getting its messages delivered quickly (if its clock is ahead) or receiving messages (if its clock is behind)—the net effect is to add a round-trip delay to that process equal to the difference between its clock and the clock of its correspondent. But the protocol works well when the processes’ clocks are closely synchronized, which has become a plausible assumption in the last 10-15 years thanks to the Network Time Protocol, cheap GPS receivers, and clock synchronization mechanisms built into most cellular phone networks.

Vector Time

The system of vector clocks was developed independently by **Fidge**, **Mattern** and **Schmuck**.

Time domain: set of n-dimensional non-negative integer vectors.

Each process p_i maintains a vector $vt_i[1..n]$, where $vt_i[i]$ is the local logical clock of p_i and describe the logical time progress at process p_i.

Vector Time

Process $\boldsymbol{p_i}$ uses the following two rules **R1** and **R2** to update its clock:

R1 Before executing an event, process p_i updates its local logical time as follows:

$$\boldsymbol{vt_i[i]} := \boldsymbol{vt_i[i] + d} \qquad (d > 0)$$

R2 Each message ***m*** is piggybacked with the vector clock ***vt*** of the sender process at sending time. On the receipt of such a message ***(m,vt)***, process p_i executes the following sequence of actions:

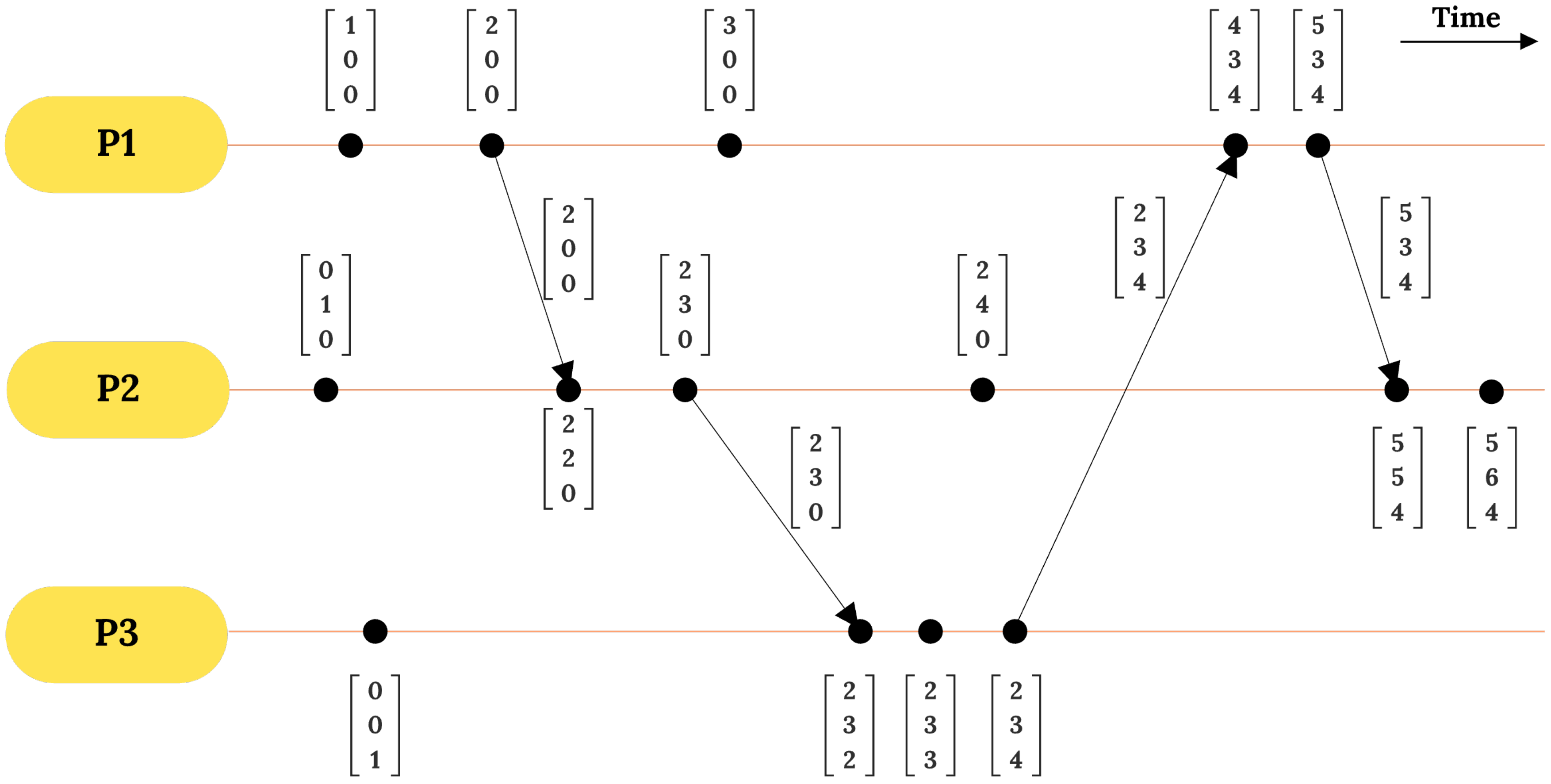
1. Update its global logical time as follows:

$$1 \le k \le n \quad : \boldsymbol{vt_i[k]} := \boldsymbol{max(vt_i[k], vt[k])};$$

2. Execute **R1**;
3. Deliver the message ***m***;

Example

Vector clocks progress with the increment value d = 1. Initially, a vector clock is [0,0,0,…,0].



Vector Time

The following relations are defined to compare two vector timestamps, $\boldsymbol{vh}$ and $\boldsymbol{vk}$:

$$vh = vk \;\Leftrightarrow\; \forall x : vh[x] = vk[x]$$

$$vh \le vk \;\Leftrightarrow\; \forall x : vh[x] \le vk[x]$$

$$vh < vk \;\Leftrightarrow\; vh \le vk \text{ and } \exists x : vh[x] < vk[x]$$

$$vh \parallel vk \;\Leftrightarrow\; \neg(vh < vk) \wedge \neg(vk < vh).$$

Isomorphism

If two events x and y have timestamps $\boldsymbol{vh}$ and $\boldsymbol{vk}$, respectively, then

$$x \longrightarrow y \quad \Leftrightarrow \quad vh < vk$$

$$x \parallel y \quad \Leftrightarrow \quad vh \parallel vk$$

Thus, there is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps. This is a very powerful, useful, and interesting property of **vector clocks**.

Isomorphism

If the process at which an event occurred is known, the test to compare two timestamps can be simplified as follows:

if events x and y respectively occurred at processes $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$ and are assigned timestamps $\boldsymbol{vh}$ and $\boldsymbol{vk}$, respectively, then

$$x \longrightarrow y \quad \Leftrightarrow \quad vh[i] \leq vk[i]$$

$$x \parallel y \quad \Leftrightarrow \quad vh[i] > vk[i] \ \wedge \ vh[j] < vk[j]$$

Strong consistency

The system of vector clocks is **strongly consistent**; thus, by examining the vector timestamp of two events, we can determine if the events are causally related. However, Charron–Bost showed that the dimension of vector clocks cannot be less than n, the total number of processes in the distributed computation, for this property to hold.

Event Counting

If d is always 1 in rule **R1**, then the *ith* component of vector clock at process p_i, $vt_i[i]$, denotes the number of events that have occurred at p_i until that instant.

So, if an event e has timestamp $vh, vh[j]$ denotes the number of events executed by process p_j that causally precede e. Clearly, $\sum vh[j] - 1$ represents the **total number of events** that causally precede e in the distributed computation.

References

- Singhal, Chapter 3
- Aspnes, Chapter 7