Module -3

Theoretical Foundations

Inherent Limitations of a Distributed System

Two **inherent limitations of distributed systems** are:

- lack of global clock
- lack of shared memory.

Absence of Global clock

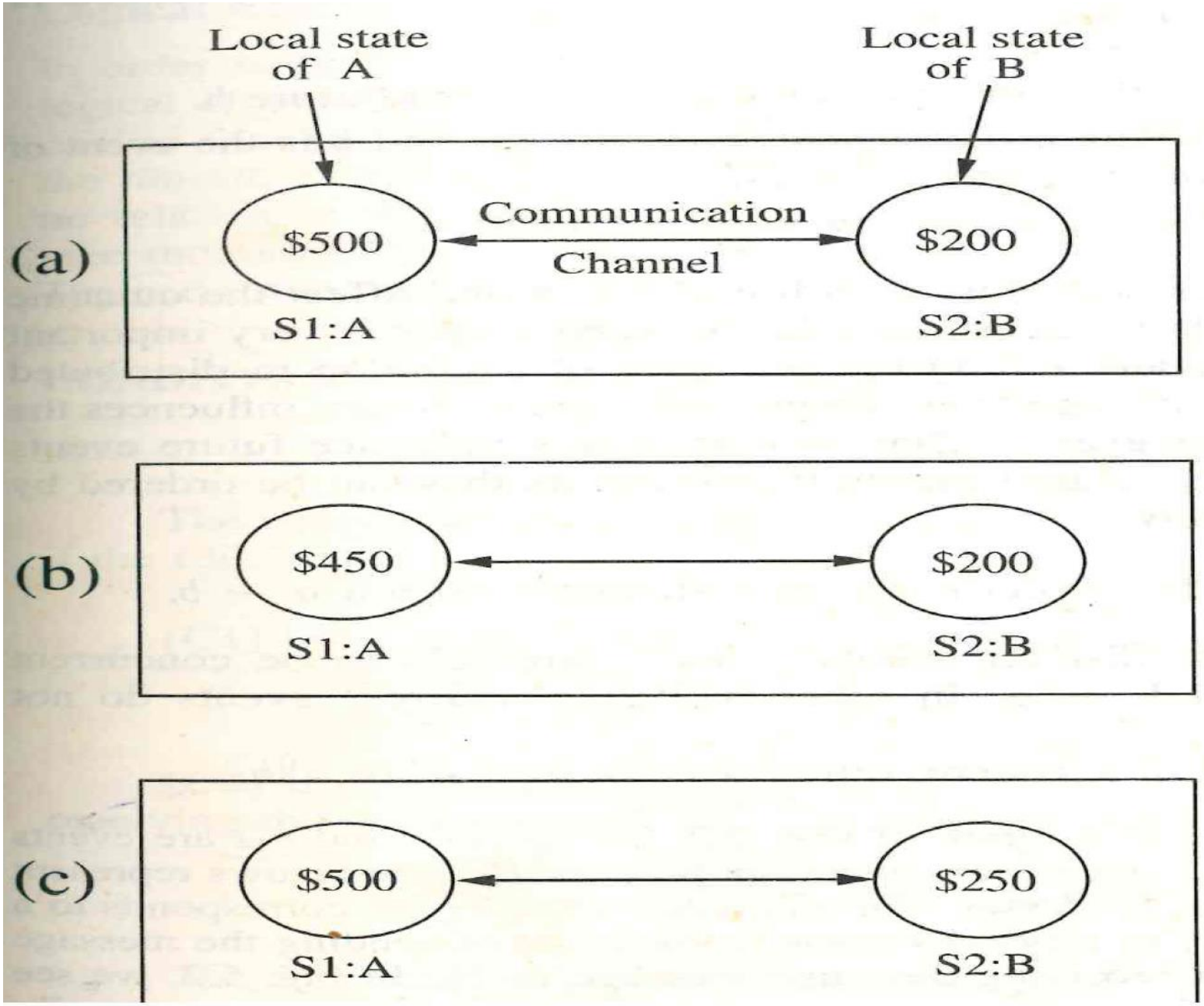
- difficult to make temporal order of events
- difficult to collect up-to-date information on the state of the entire system

Absence of Shared Memory

- no up-to-date state of the entire system to any individual process as there's no shared memory
- coherent view -- all observations of different processes (computers) are made at the same physical time

  we can obtain a coherent but partial view of the system

  or incoherent view of the system
- complete view (global state) -- local views (local states) + messages in transit difficult to obtain a coherent global state

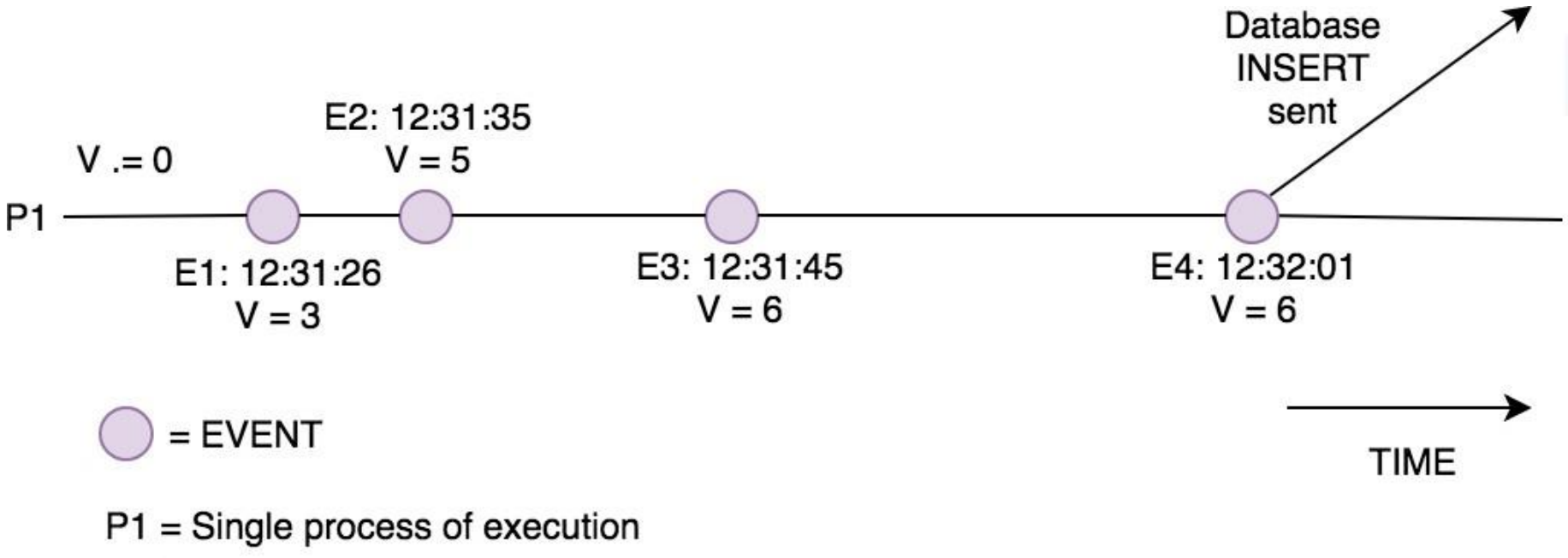

Local state of A
Local state of B
(a)
$500
Communication
Channel
$200
S1:A
S2:B
(b)
$450
$200
S1:A
S2:B
(c)
$500
$250
S1:A
S2:B

Distributed Systems
Physical, Logical, and Vector Clocks

Physical Clock

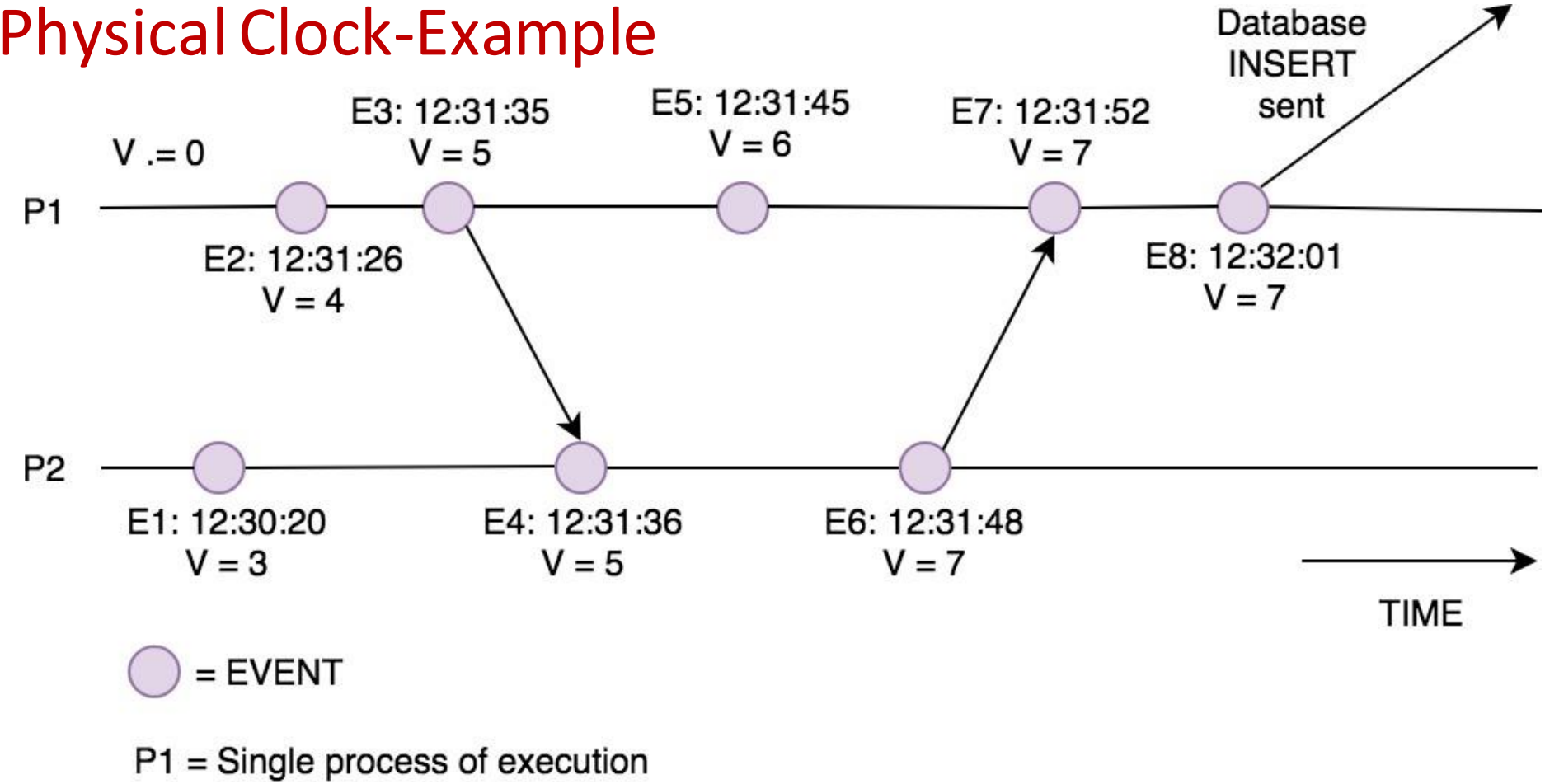
- When we have a single machine, a timeline of events is relatively easy to create, even if we have multiple processes (threads) creating events. This is due to all the events, across processes, sharing the same P**hysical Clock**.

Physical Clock-Example



this shows a series of events by a single process [P1], leading up to a database INSERT request being made. You can see that using timestamps allow the events to be placed in chronological order, showing how the variable V ended up assuming the value 6 when sent to the database.

Physical Clock-Example



this shows 2 processes sending messages between each other affecting the variable V. You can see that the timestamps still work as they are both on the same machine with the same single view of time. Thus, we can order the events and determine what caused V to be equal to 7 when it was sent to the database.

Global Clock

- Let's take our previous example to prove in a distributed system.

- If we have events leading up to an INSERT now happening **across multiple machines, each with their own local clock, and we use timestamps to place events in chronological order, we now have to guarantee that every machines clock has the exact same time. This is known as having a Global Clock, and is not easily achieved in a distributed system**

Logical Clocks

- In distributed systems, physical clocks are not always precise, so we can't rely on physical time to order events. Instead, we can use **logical clocks** to create a partial or total ordering of events.

Lamport's Logical Clock

The happened before → relation

- a → b , if a and b are events in the same process and a occurred before b
- a → b , if a is the event of sending a message m in a process and b is the event of receipt of the same message m by another process
- if a → b and b → c, then a → c (transitive)
- **event a causally affects b** if a → b
- **concurrent**: a || b if !(a → b) and !(b → a)
- **for any two events in a system, either a → b or b → a or a || b**

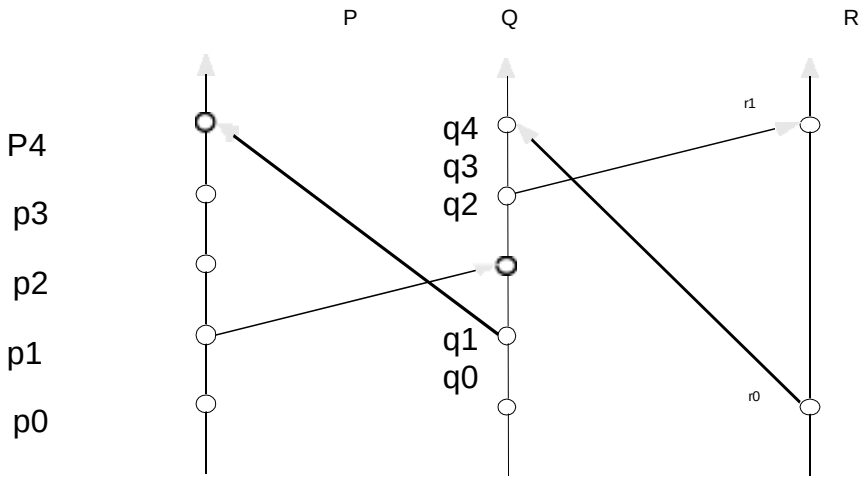
The "happened before" relation

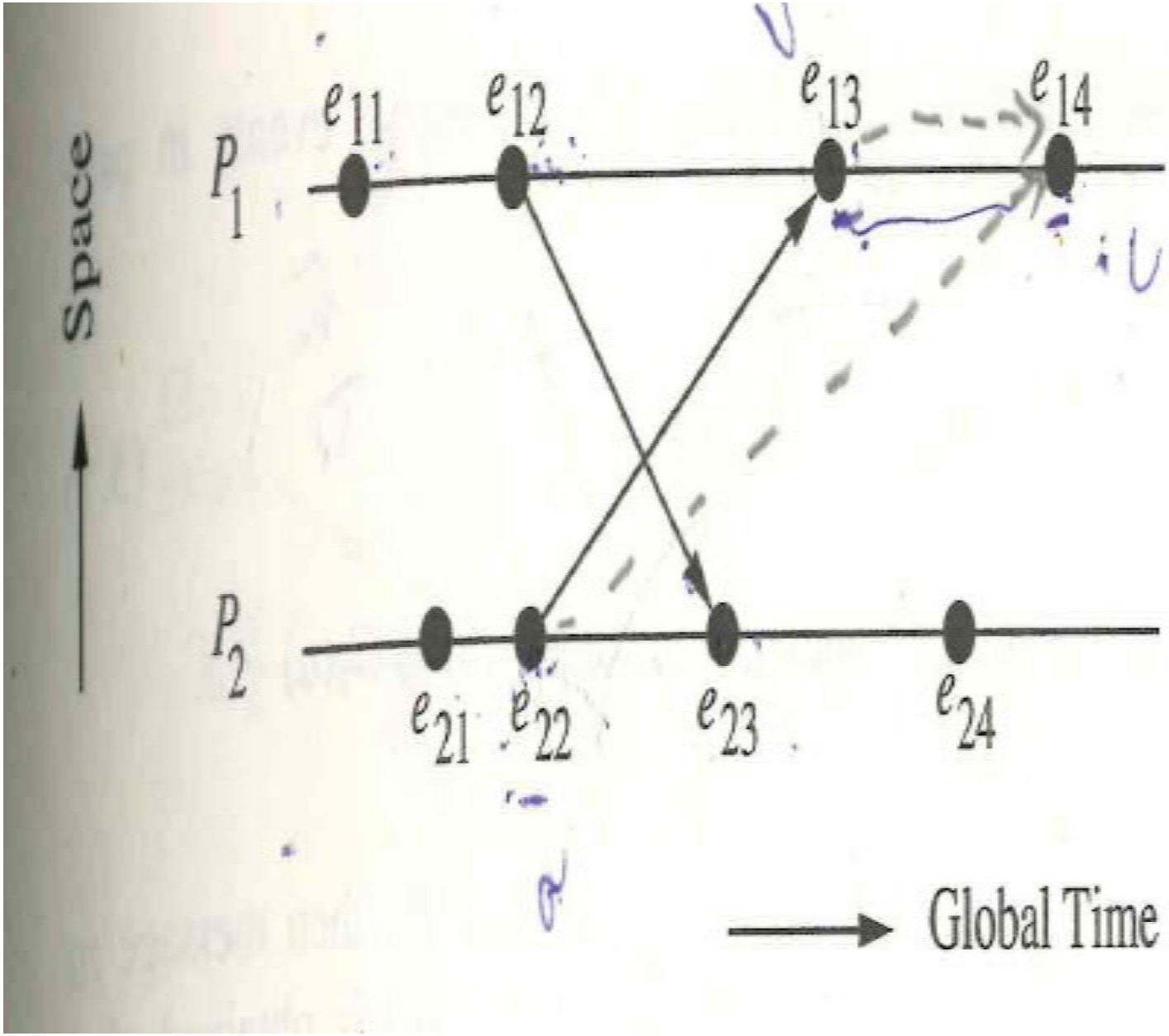
•Concurrent events;

■Two distinct events *a* and *b* are said to be *concurrent* (denoted "*a* ||*b*"), if neither *a*|| *b* nor *b*|| *a*

■In other words, concurrent events do not causally affect each other

•For any two events *a* and *b* in a system, either: *a*||*b* or *b*|| *a* or *a*||*b*




Space
P_1
P_2
e_{11}
e_{12}
e_{13}
e_{14}
e_{21}
e_{22}
e_{23}
e_{24}
Global Time

Logical Clocks

- This is more formally specified as a way of placing events in some timespan so the following property will always be true:

Given 2 events (e1, e2) where one is caused by the other (e1 contributes to e2 occurring). Then the timestamp of the 'caused by' event (e1) is less than the other event (e2). To provide this functionality any **Logical Clock** must provide 2 rules:

- *Rule 1*: this determines how a local process updates its own clock when an event occurs.
- *Rule 2*: determines how a local process updates its own clock when it receives a message from another process. This can be described as how the process brings its local clock inline with information about the global time.

Realization

To realize the relation $\rightarrow$ we need a clock C_i at each process P_i in the system, and adjust the clock according to the following rules.

$C_i(a)$ -- timestamp of event a at P_i

if $a \rightarrow b$, then $C(a) < C(b)$

Condition requirements:

1. for any two events **a** and **b** in a process P_i,
   if a occurs before b, then $C_i(a) < C_i(b)$
2. if **a** is the event of sending a message **m** in P_i
   and **b** is the event of receiving the same message **m**
   at process P_j, then
   $C_i(a) < C_j(b)$

Implementation rules:

1. two successive events in P_i

$$C_i = C_i + d \ (\ d > 0 \ )$$

if a and b are two successive events in P_i and

$a \rightarrow b$ then

$$C_i(b) = C_i(a) + d \ (\ d > 0 \ )$$

2. event a: sending of message m by process P_i,
timestamp of message m : $t_m = C_i(a)$
then

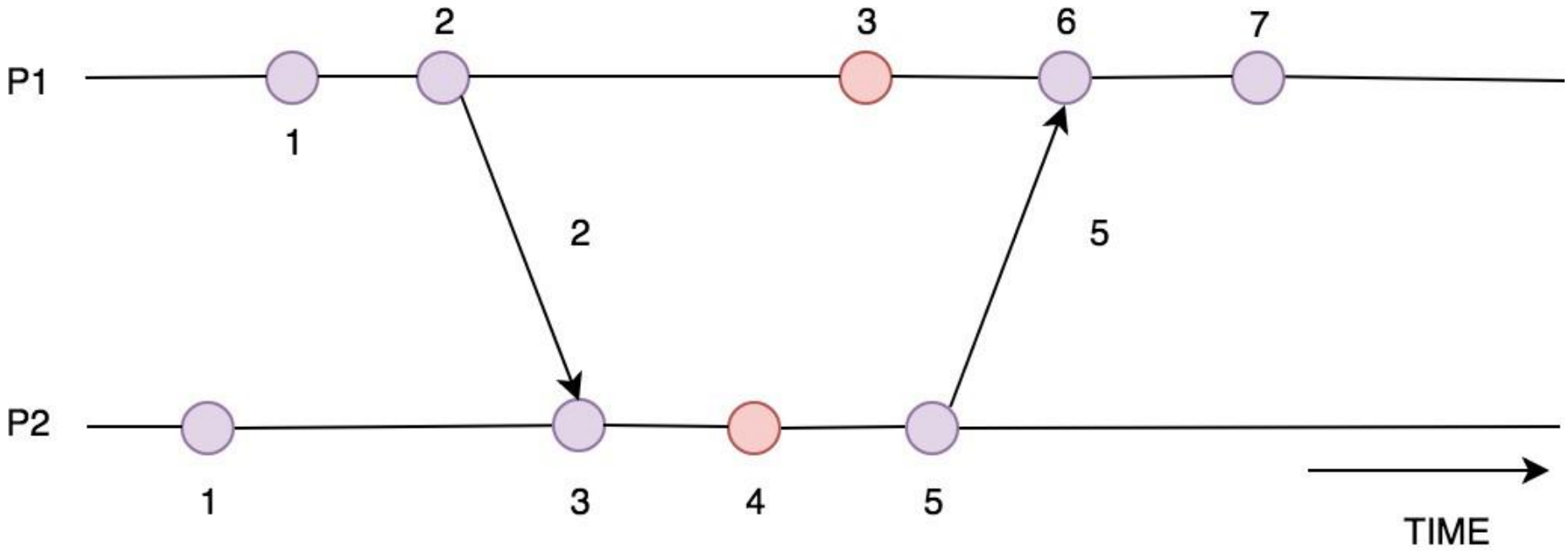
$$C_j = \max (C_j, t_m + d) \quad d > 0$$

- $\rightarrow$ is irreflexive, defines partial order among events
- Totally ordering relation (=>) can be defined by

a is any event in process P_i
b is any event in process P_j **a** => **b** iff

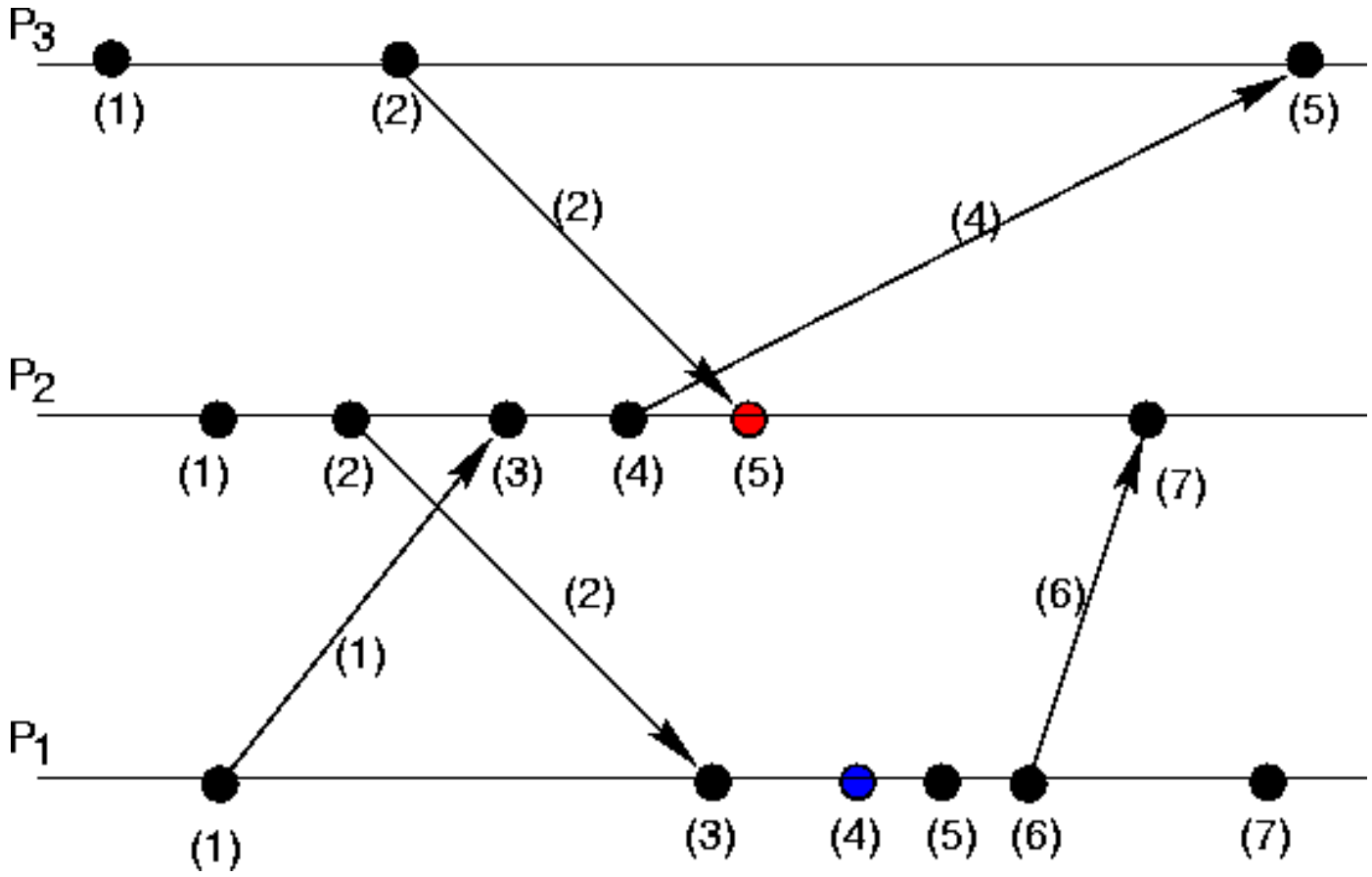
either $C_i(a) < C_j(b)$
or $C_i(a) = C_j(b)$ and $P_i \prec P_j$ (e.g. $P_i \prec P_j$ if $i \leq j$, to break ties)

Example of Logical Clocks



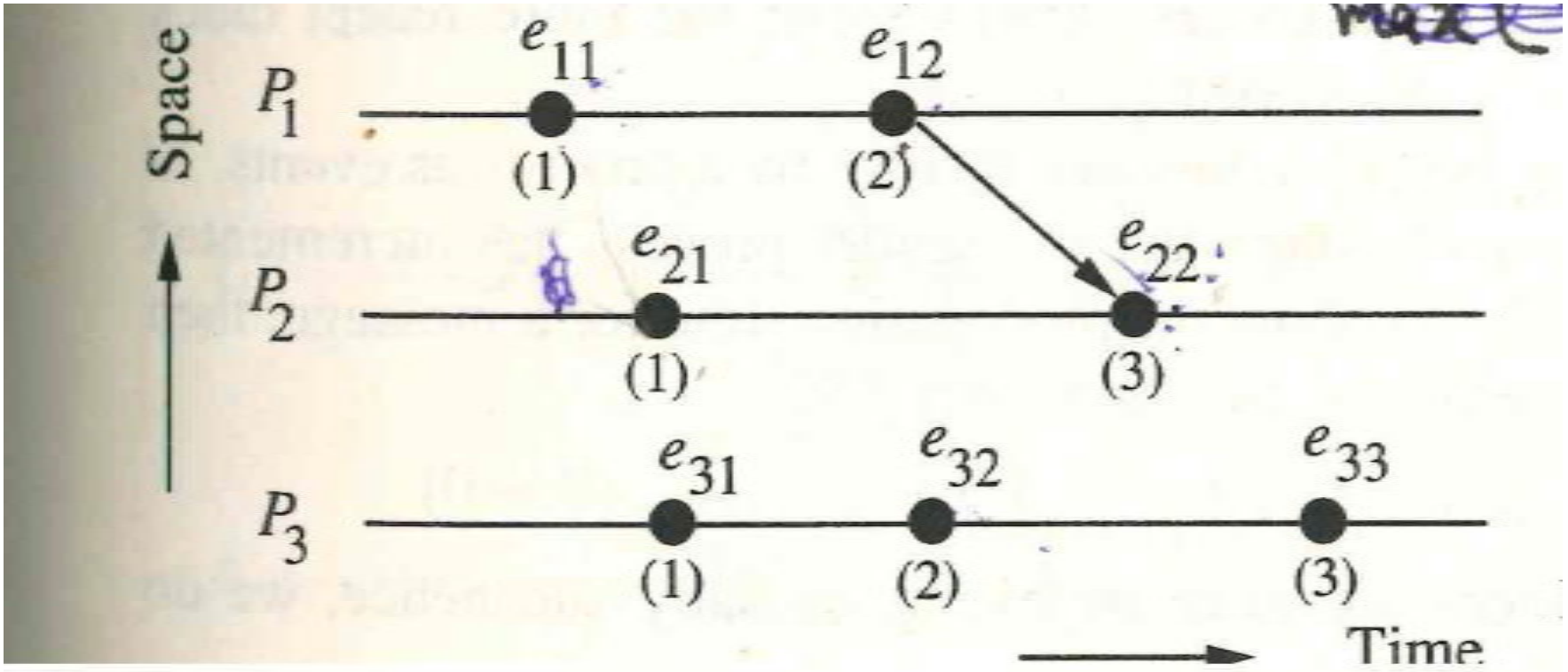
- the above diagram shows how using Scalar Time, the 2 processes (P1 and P2) update their local clocks based on the messages they receive.

Example of Logical Clocks



Limitation of Lamport's Clocks

- if a → b then C(a) < C(b)
  but C(a) < C(b) does **not** necessarily imply a → b



Limitation of Logical Clocks

- **Scalar Time** provides an eventually consistent state of time. This means there may be places where the recorded time differs between processes but, given a finite amount of time, the processes will converge on a single view of the correct time.
- Causing this is the fact that internal events in a process (that apply *Rule 1)* loose strong consistency with regards to concurrent events on another process.
- This is due to the use of a single number for both our global and local clocks across processes. **In order to become strongly consistent we need a way of representing local time and global time separately. This is where Vector Clocks come in.**