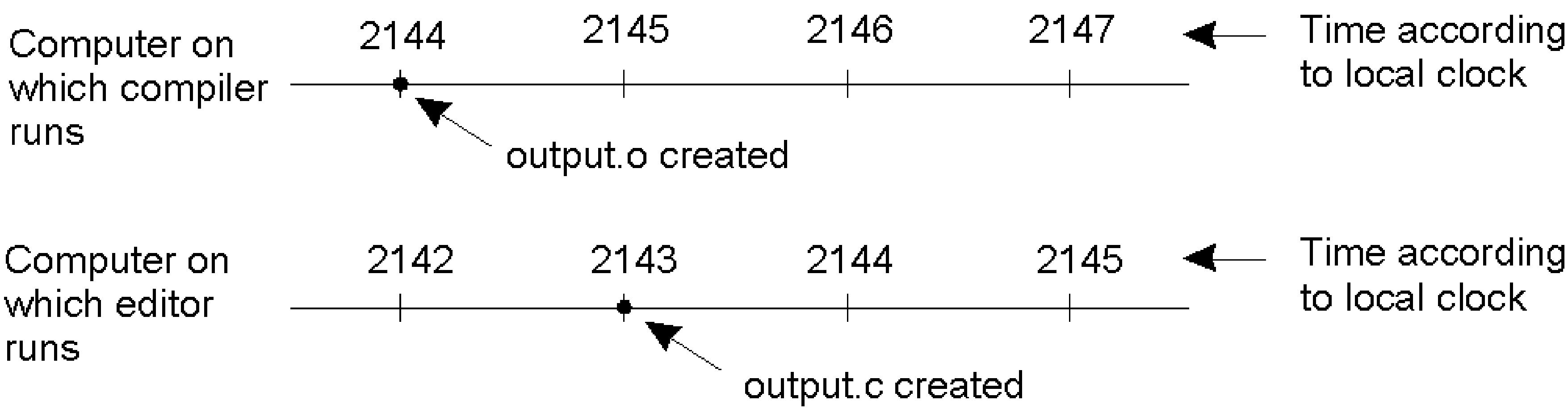
Distributed Systems

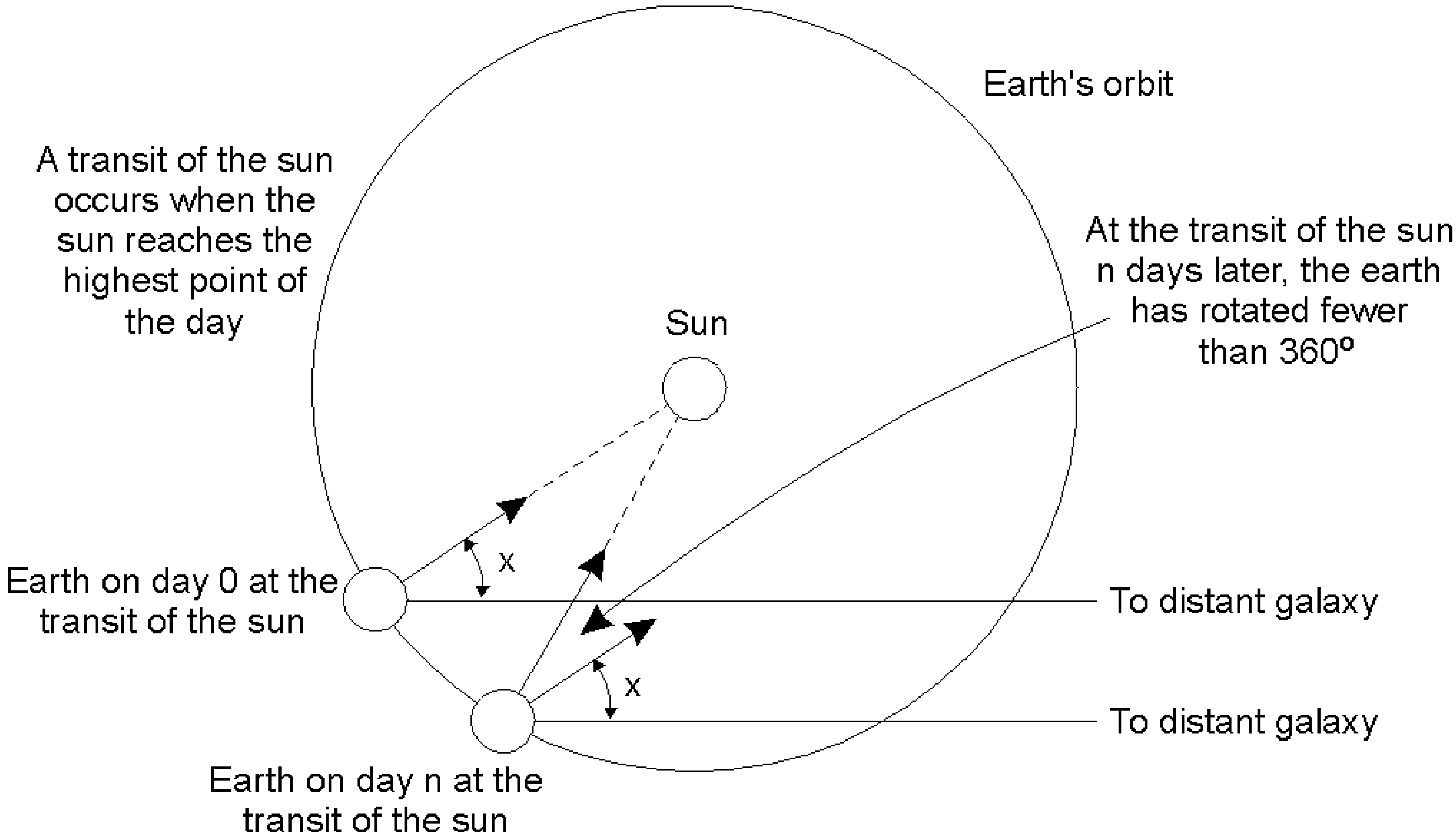
Logical Clocks

Clock Synchronization



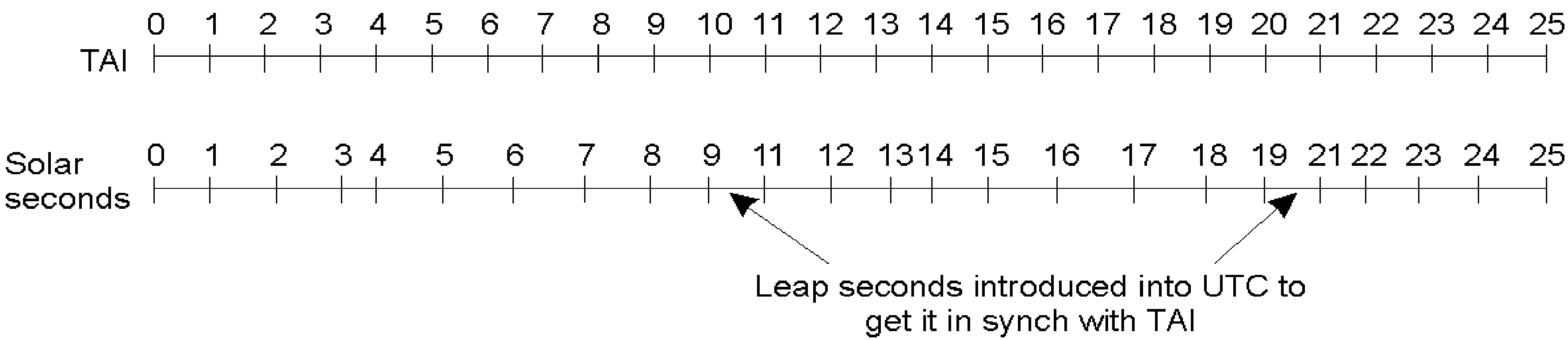
- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Physical Clocks (1)



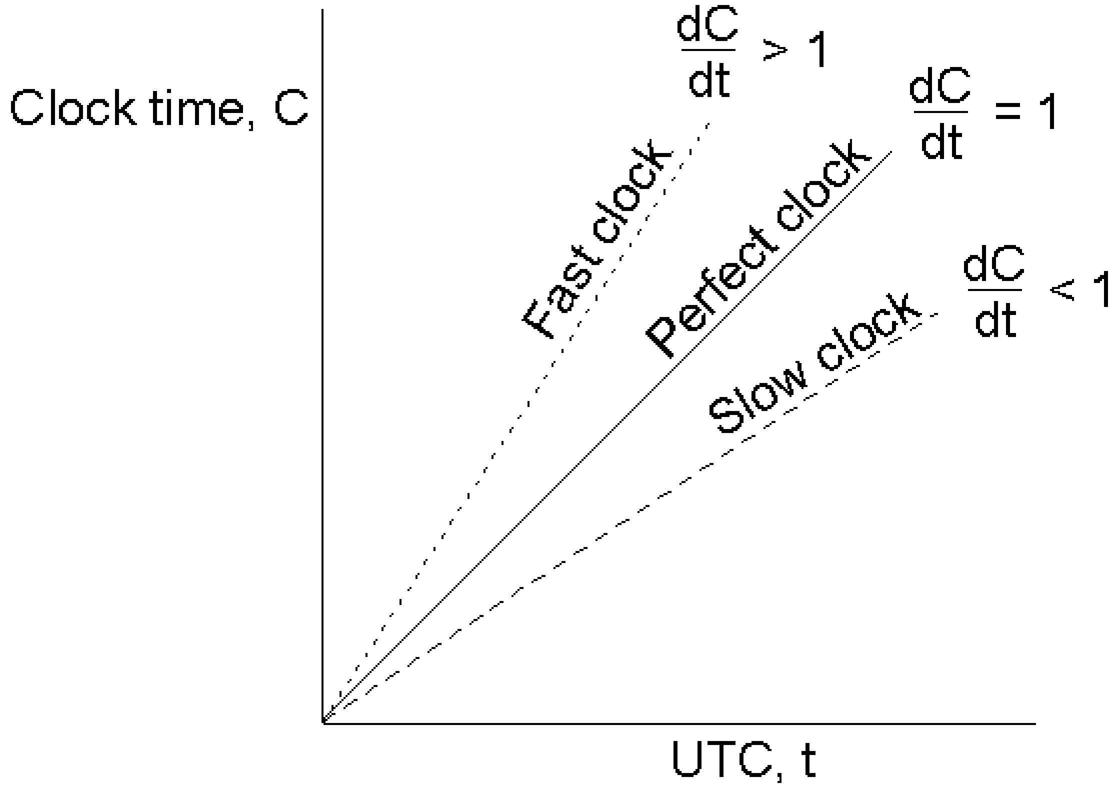
- Computation of the mean solar day.

Physical Clocks (2)



- TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

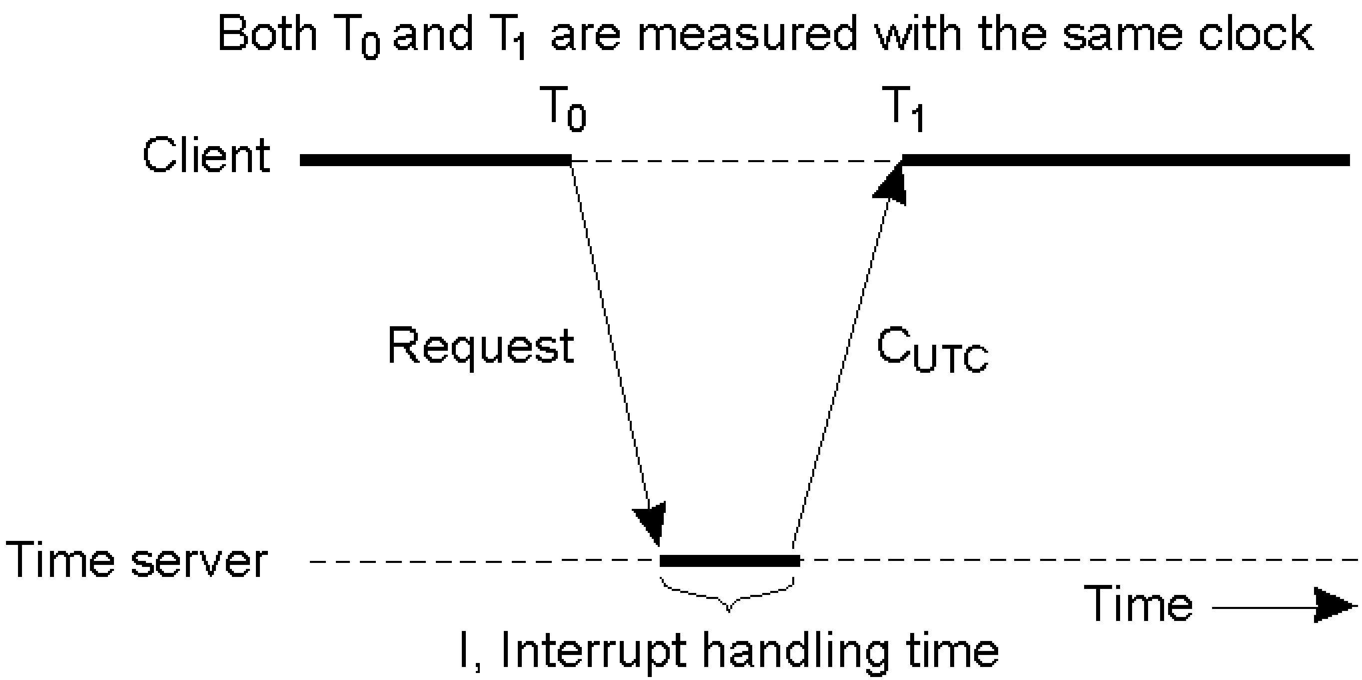
Clock Synchronization Algorithms



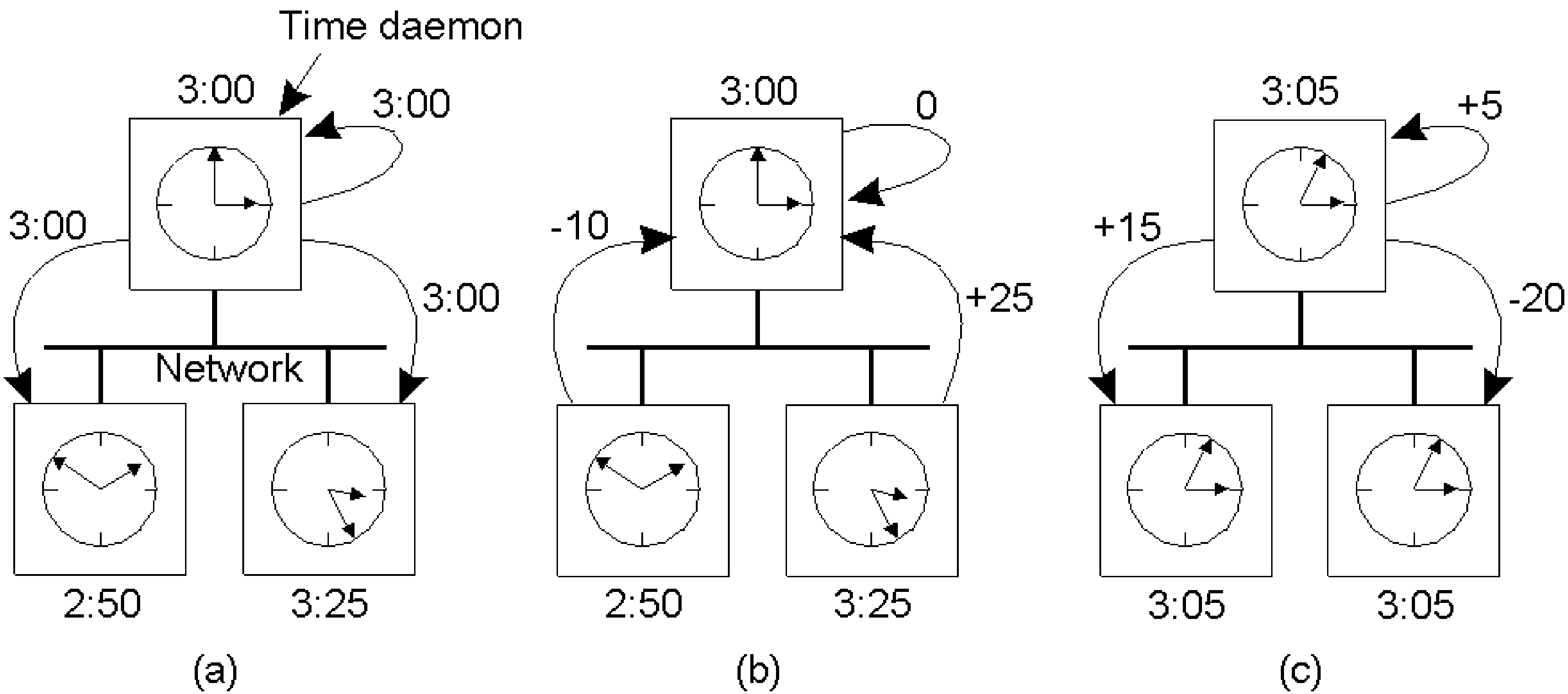
- The relation between clock time and UTC when clocks tick at different rates.

Cristian's Algorithm (Passive Server)

- Getting the current time from a time server.



The Berkeley Algorithm (Active Server)



a) The time daemon asks all the other machines for their clock values

b) The machines answer

c) The time daemon tells everyone how to adjust their clock

Decentralized Averaging Algorithm

- A machine broadcasts its time
- It starts a local timer to collect all other broadcasts that arrive during some interval S
- When all broadcasts arrive, a new time is computed
  - An average is a good guess
  - A slight variation is to discard the m highest and m lowest values and average the rest

Logical clocks

Assign sequence numbers to messages

- All cooperating processes can agree on order of events
- vs. **physical clocks**: time of day

Assume no central time source

- Each system maintains its own local clock
- No total ordering of events
  - No concept of *happened-when*

Happened-before

Lamport's "ha   ned-before" notation

$a \rightarrow b$ event a happened before event b

e.g.: a*:* message being sent, b: message receipt

Transitive:

if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

Logical clocks & concurrency

Assign "clock" value to each event

- if $a \rightarrow b$ then **clock(a) < clock(b)**
- since time cannot run backwards

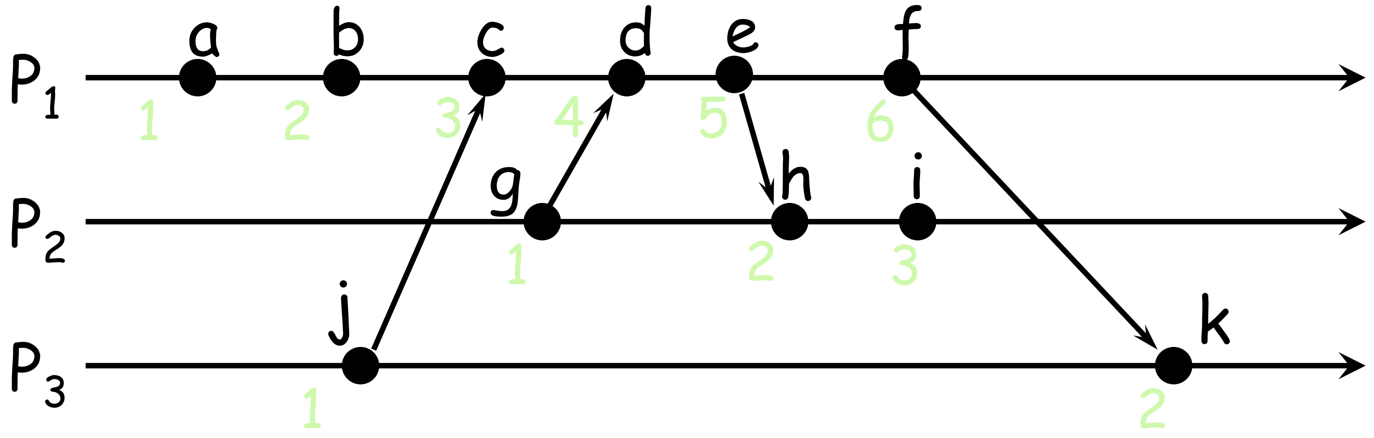
If a and b occur on different processes that do not exchange messages, then neither $a \rightarrow b$ nor $b \rightarrow a$ are true

- These events are **concurrent**

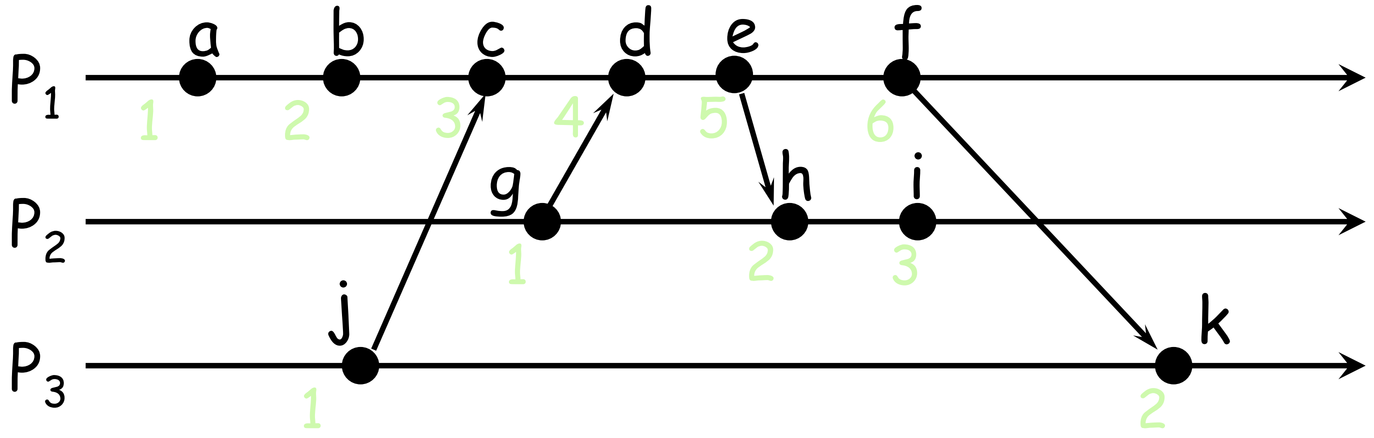
Event counting example

- Three systems: P_0, P_1, P_2
- Events *a*, *b*, *c*, …
- Local event counter on each system
- Systems occasionally communicate

Event counting example



Event counting example



Bad ordering:

e → h

f → k

Lamport's algorithm

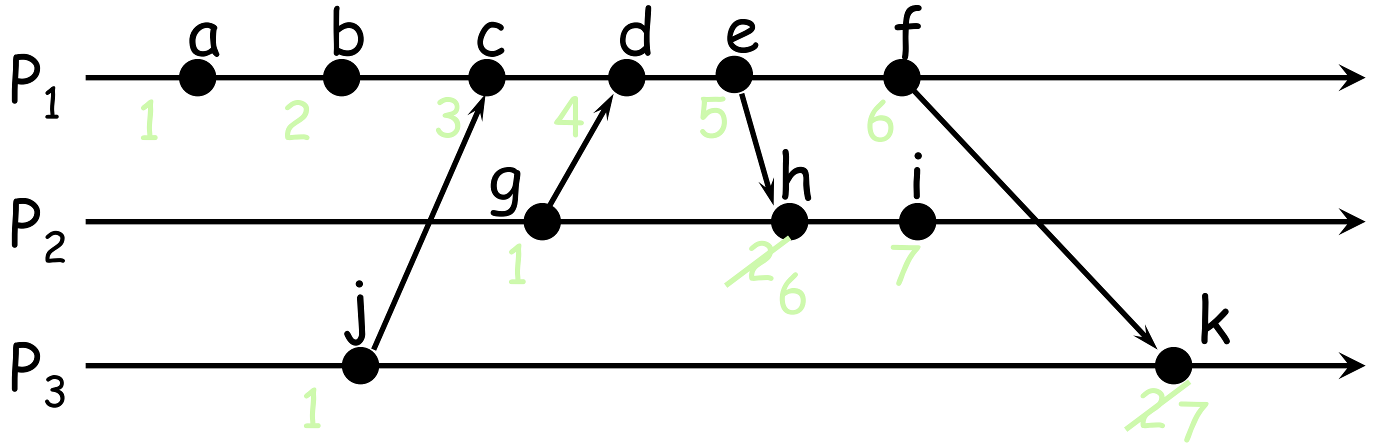
- Each message carries a timestamp of the sender's clock
- When a message arrives:
  - if receiver's clock < message timestamp
    set system clock to (message timestamp + 1)
  - else do nothing
- Clock must be advanced between any two events in the same process

Lamport’s algorithm

Algorithm allows us to maintain time ordering among related events

- ***Partial ordering***

Event counting example

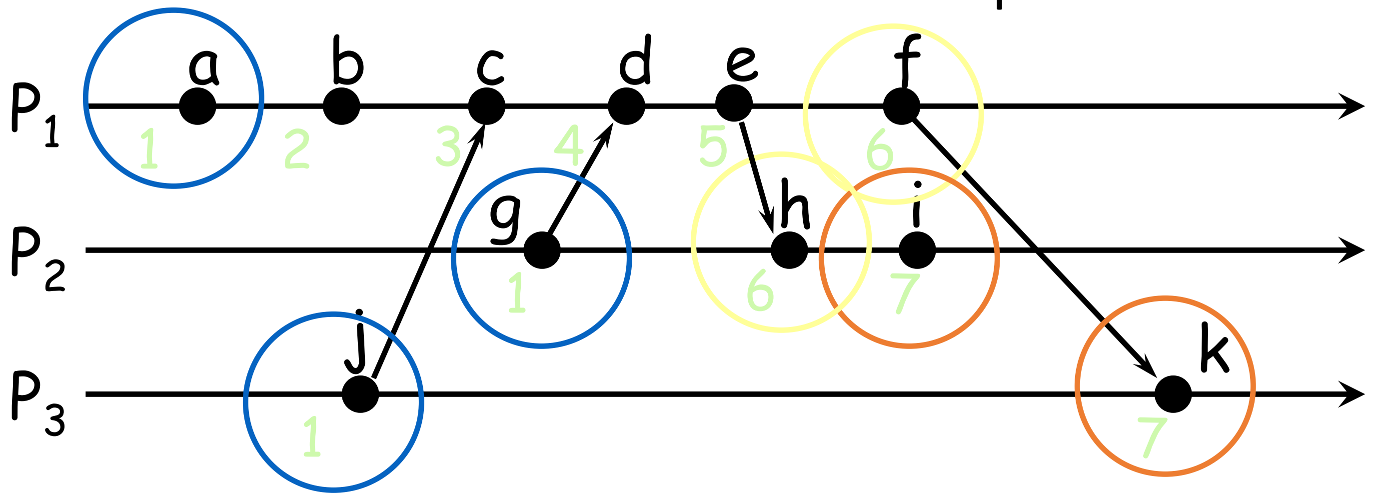


Summary

- Algorithm needs monotonically increasing software counter
- Incremented at least when events that need to be timestamped occur
- Each event has a **Lamport timestamp** attached to it
- For any two events, where a $\rightarrow$ b:

  $$L(a) < L(b)$$

Problem: Identical timestamps



$a \rightarrow b$, $b \rightarrow c$, …: local events sequenced

$i \rightarrow c$, $f \rightarrow d$, $d \rightarrow g$, … : Lamport imposes a *send*→*receive* relationship

Concurrent events (e.g., a & i) *may* have the same timestamp … or not

Unique timestamps (total ordering)

We can force each timestamp to be unique

- Define <u>global logical timestamp</u> (T_i, i)
  - T_i represents local Lamport timestamp
  - i represents process number (globally unique)
    - E.g. (host address, process ID)
- Compare timestamps:

  $(T_i, i) < (T_j, j)$

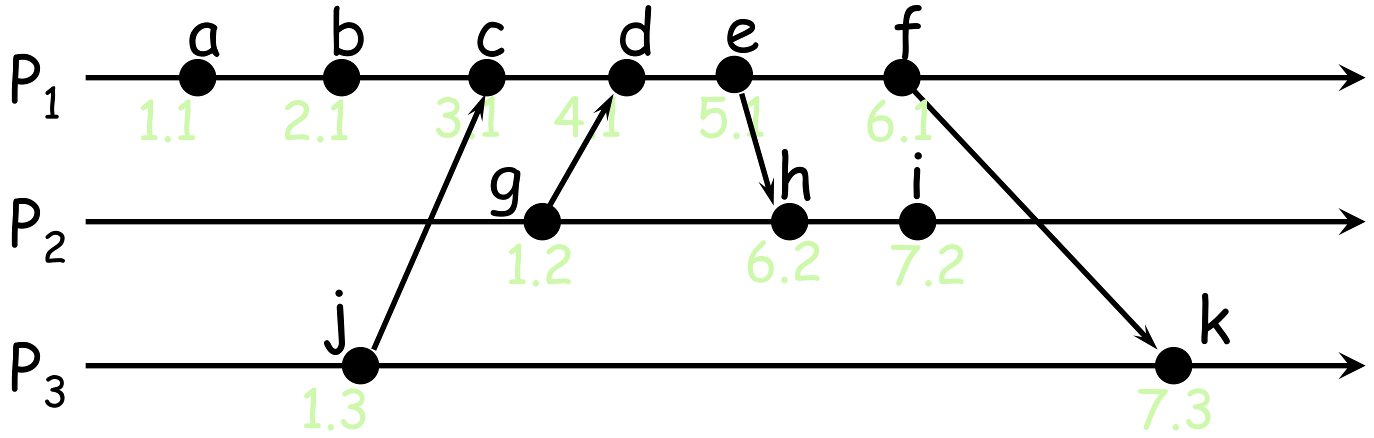
  if and only if

  $T_i < T_j$ or

  $T_i = T_j$ and $i < j$

Does not relate to event ordering

Unique (totally ordered) timestamps



Problem: Detecting causal relations

If L(e) < L(e')

- Cannot conclude that e→e'

Looking at Lamport timestamps

- Cannot conclude which events are causally related

Solution: use a **vector clock**

Vector clocks

Rules:

1. Vector initialized to 0 at each process

   $V_i[j] = 0$ for $i, j = 1, \ldots, N$
2. Process increments its element of the vector in local vector before timestamping event:

   $V_i[i] = V_i[i] + 1$
3. Message is sent from process P_i with V_i attached to it
4. When P_j receives message, compares vectors element by element and sets local vector to higher of two values

   $V_j[i] = \max(V_i[i], V_j[i])$ for $i=1, \ldots, N$

Comparing vector timestamps

<u>Define</u>

$V = V'$ iff $V[i] = V'[i]$ for $i = 1 \ldots N$
$V \leq V'$ iff $V[i] \leq V'[i]$ for $i = 1 \ldots N$

For any two events e, e'

if $e \rightarrow e'$ then $V(e) < V(e')$

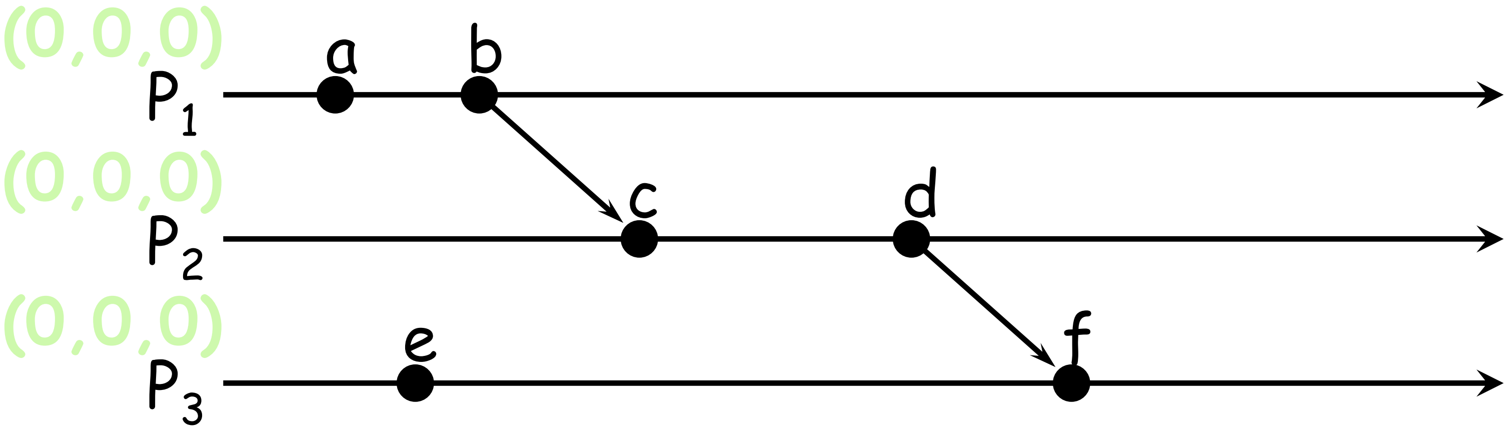
- Just like Lamport's algorithm

if **$V(e) < V(e')$ then $e \rightarrow e'$**

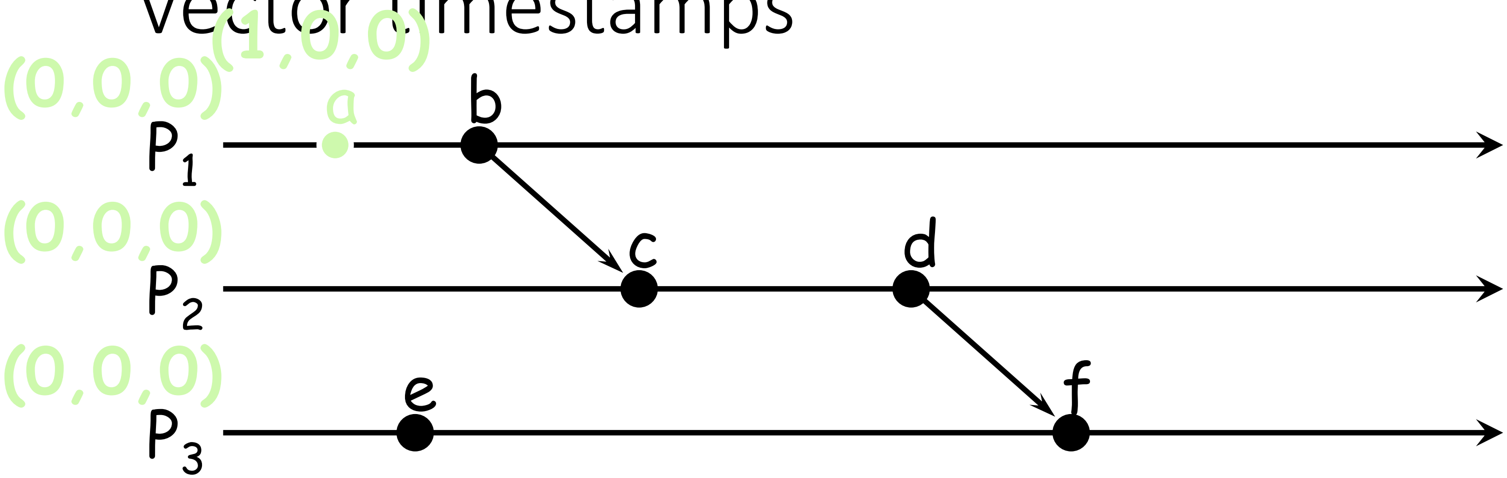
Two events are concurrent if **neither**

$V(e) \leq V(e')$ nor $V(e') \leq V(e)$

Vector timestamps

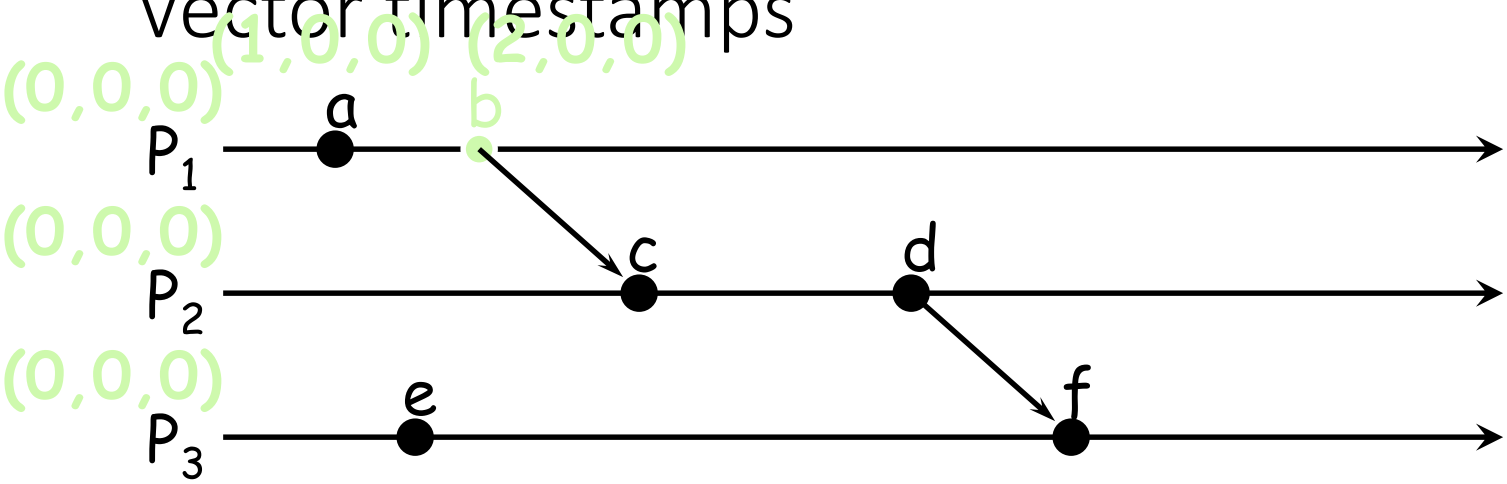


Vector timestamps



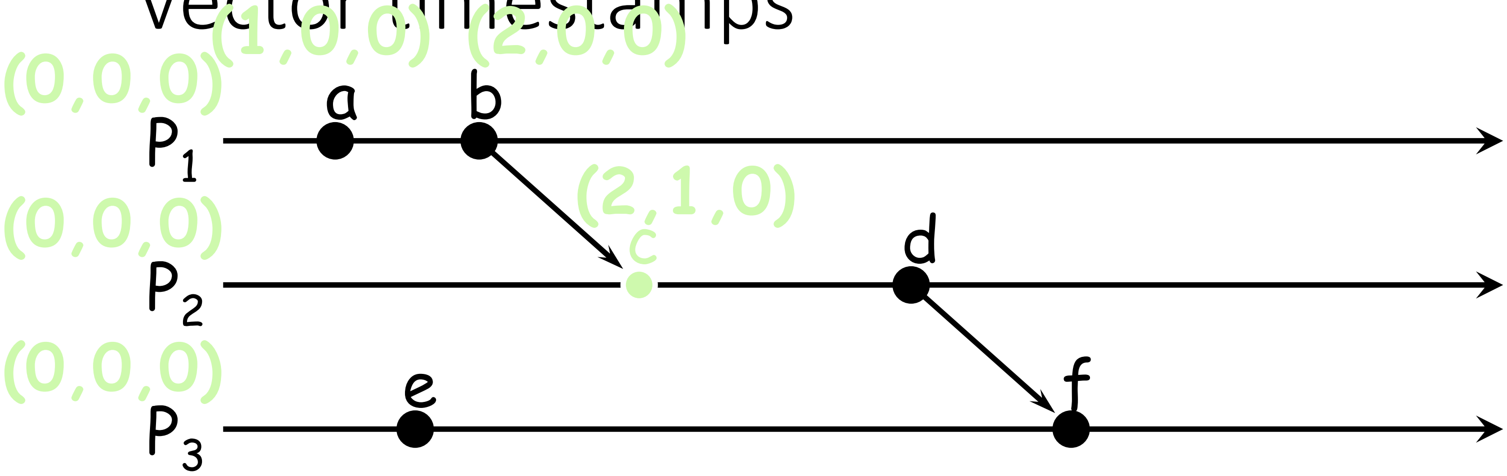
Event	timestamp
a	(1,0,0)

Vector timestamps



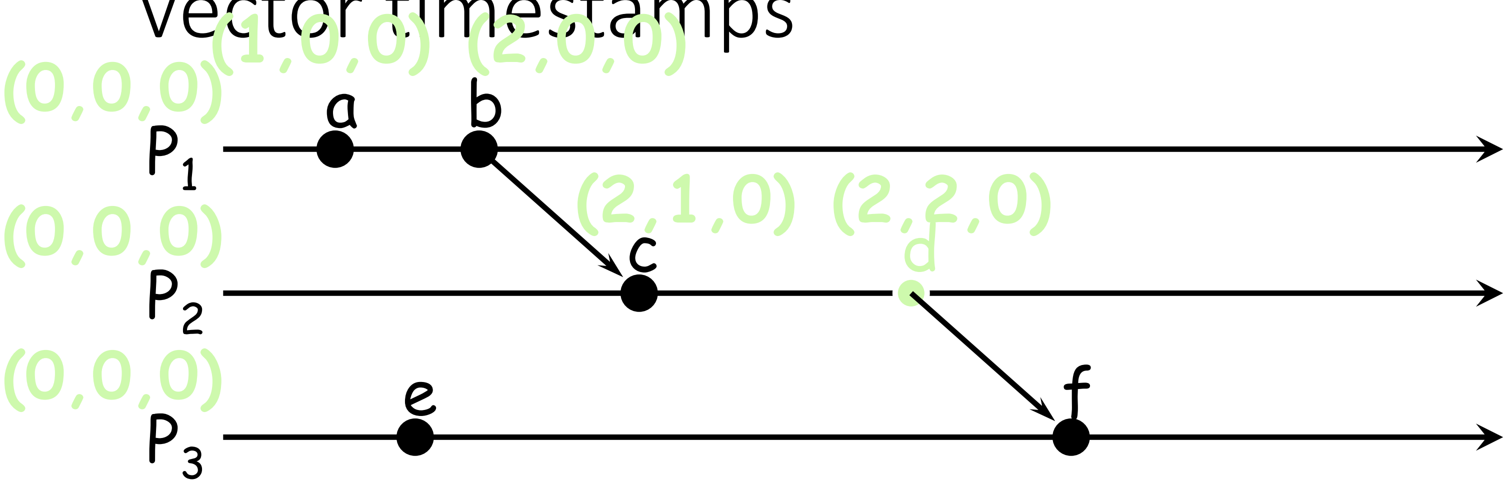
Event	timestamp
a	(1,0,0)
b	(2,0,0)

Vector timestamps



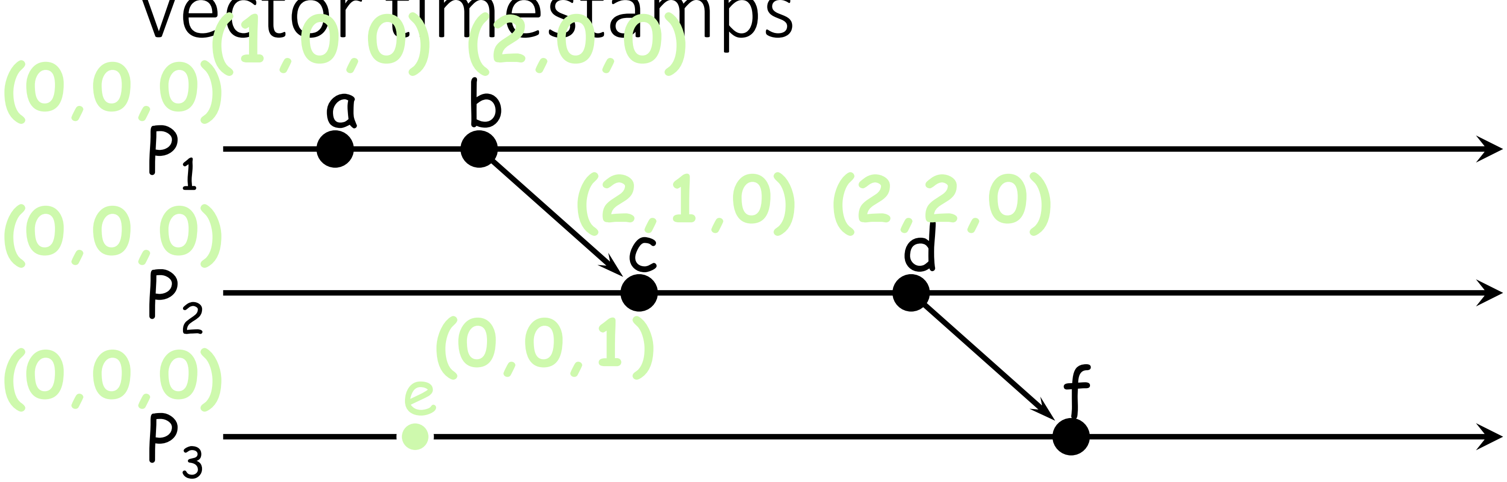
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)

Vector timestamps



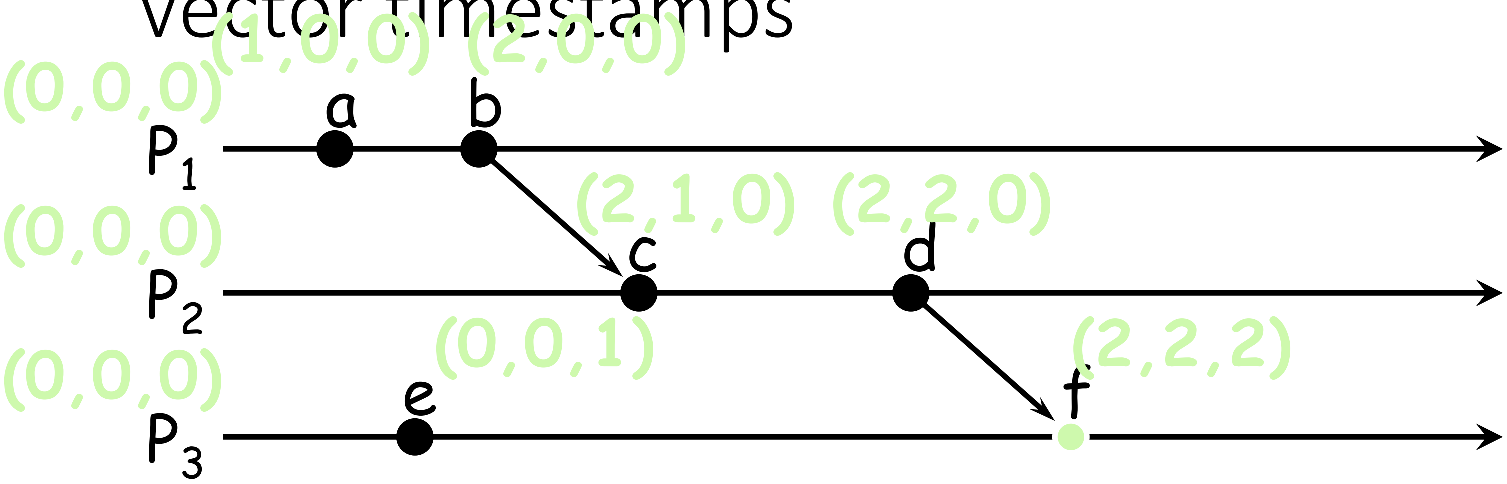
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)

Vector timestamps



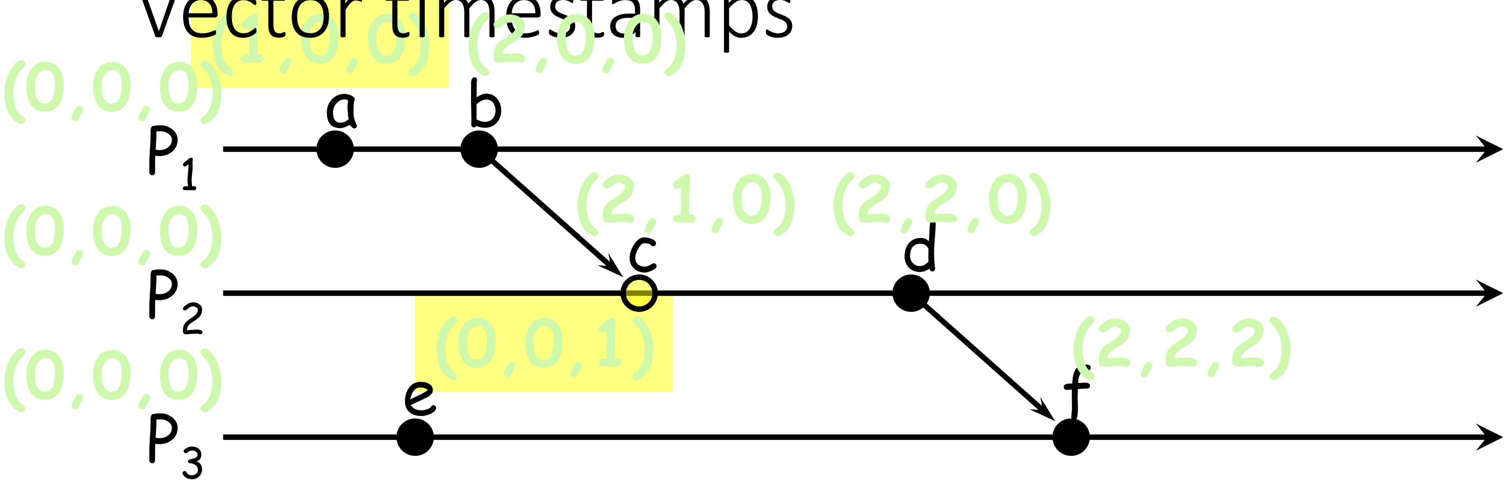
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)

Vector timestamps



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

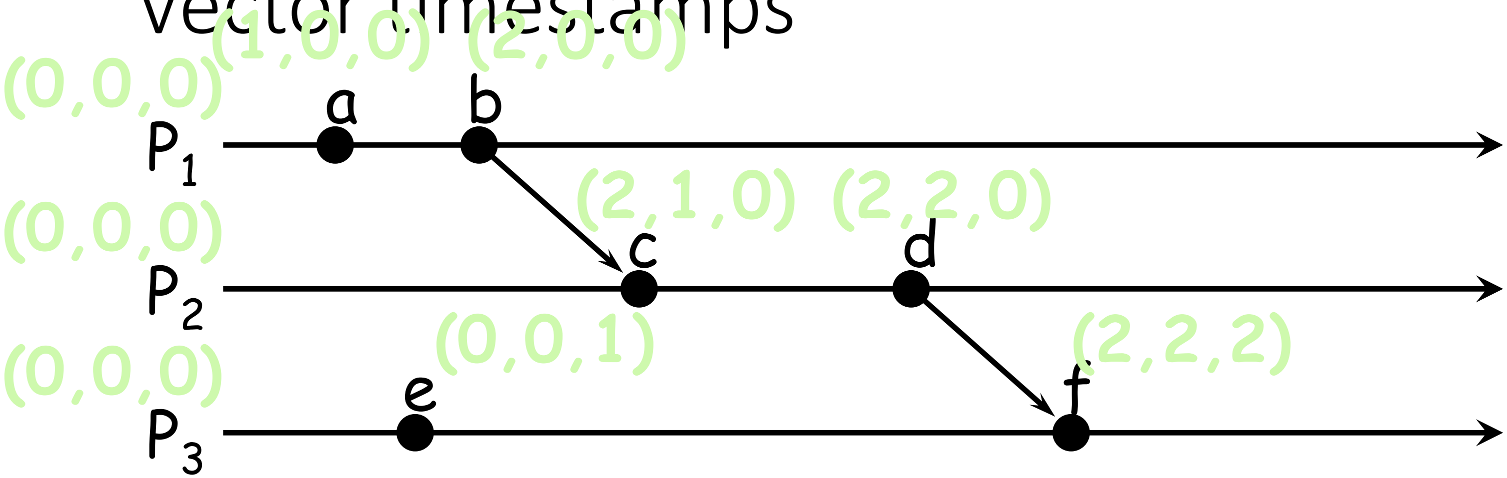
Vector timestamps



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

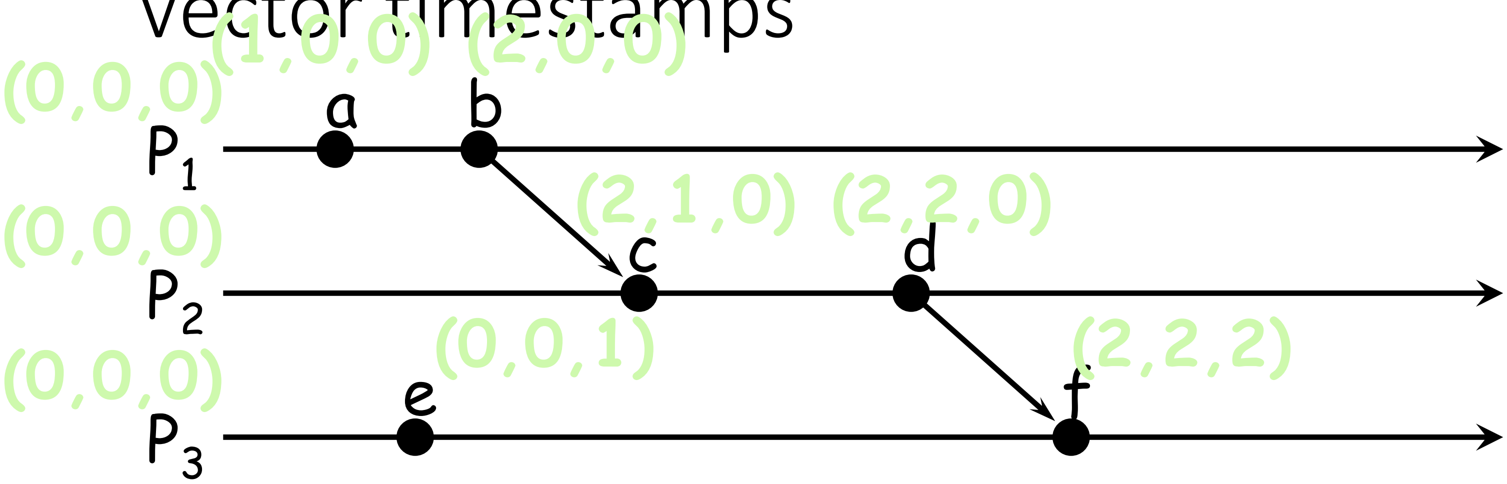
Vector timestamps



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

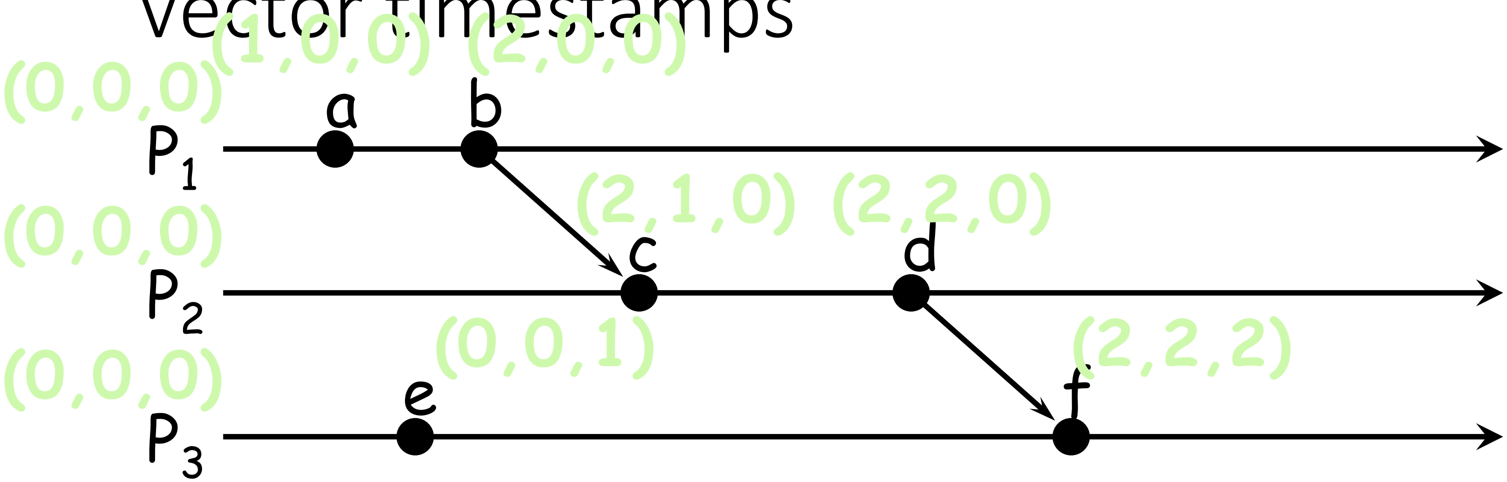
Vector timestamps



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

Vector timestamps



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

Summary: Logical Clocks & Partial Ordering

- Causality
  - If *a*->*b* then event *a* can affect event *b*
- Concurrency
  - If neither *a*->*b* nor *b*->*a* then one event cannot affect the other
- Partial Ordering
  - Causal events are sequenced
- Total Ordering
  - All events are sequenced