

---

# Distributed Systems

Winter Term 2025/26

Roland Wismüller  
Universität Siegen  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: November 27, 2025



---

# Distributed Systems

Winter Term 2025/26

## 6 Time and Global State



- ➔ Synchronization of physical clocks
- ➔ Lamport's happened before relation
- ➔ Logical clocks
- ➔ Global state

### Literature

- ➔ Tanenbaum, van Steen: Kap. 5.1-5.3
- ➔ Colouris, Dollimore, Kindberg: Kap. 10
- ➔ Stallings: Kap 14.2



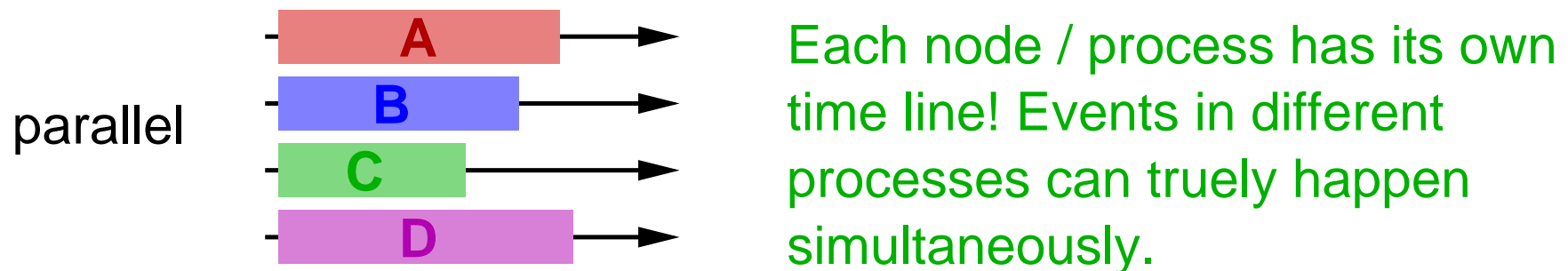
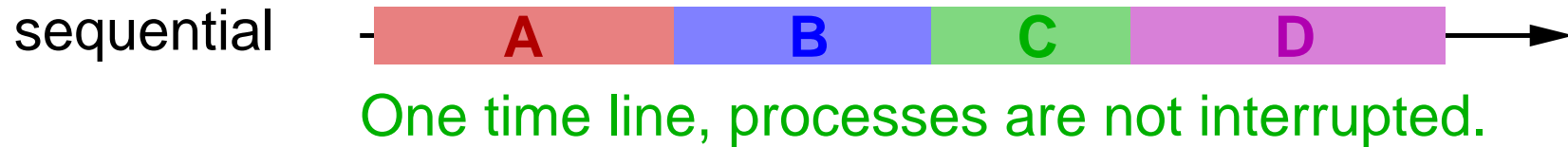
### What is the difference between a distributed system and a single/multiprocessor system?

- ➔ Single or multiprocessor system:
  - ➔ concurrent processes: pseudo-parallel by time sharing or truly parallel
  - ➔ global time: all events in the processes can be ordered unambiguously in terms of time
  - ➔ global state: at any time a unique state of the system can be determined
- ➔ Distributed system
  - ➔ true parallelism
  - ➔ no global time
  - ➔ no unique global state



## Concurrency vs. (true) parallelism

### Example: 4 processes





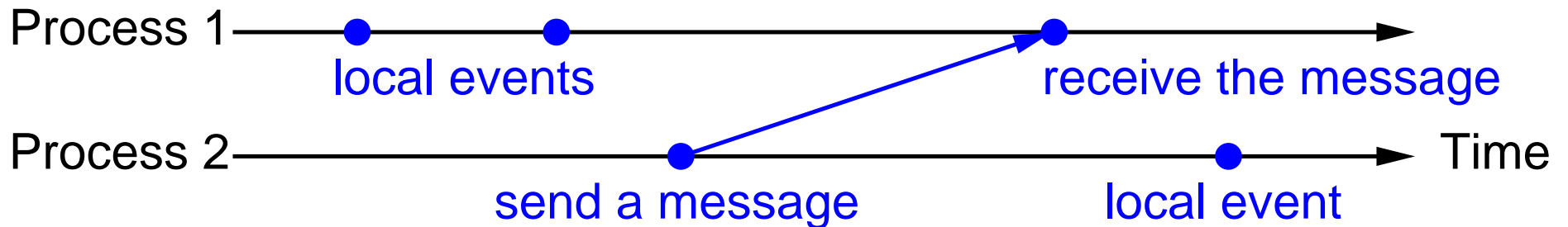
## Global Time

- ➔ In a single/multiprocessor system
  - ➔ each event can (at least theoretically) be assigned a unique time stamp of the same local clock
  - ➔ for multiprocessor systems: synchronization at the shared memory
  
- ➔ In distributed systems:
  - ➔ many local clocks (one per node)
  - ➔ exact synchronization of clocks is (on principle!) not possible
  - ➔  $\Rightarrow$  the sequence of events on different nodes can not (always) be determined uniquely
    - ➔ (cf. special theory of relativity)

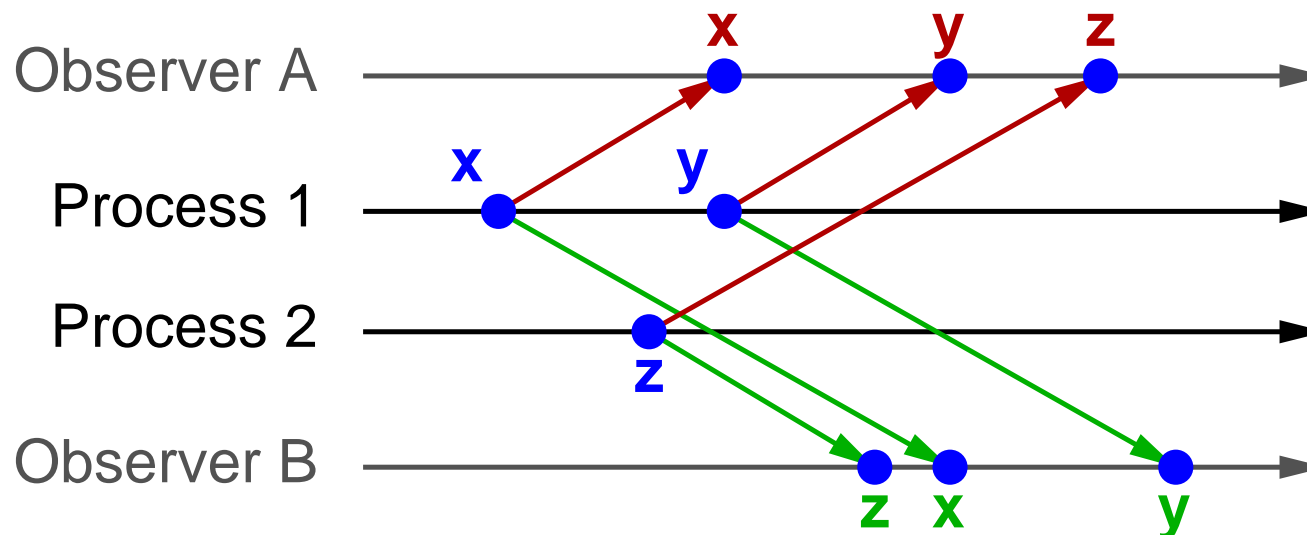


## An effect of distribution

➔ Visualization of events in distributed systems: **Lamport diagram**



➔ Scenario: two processes observe two other processes





### An effect of distribution ...

- ➔ The observers may see the events in different order!
- ➔ Problem e.g., if the observers are replicated databases and the events are database updates
  - ➔ replicas are no longer consistent!
- ➔ Even from time stamps of (local) clocks it is not possible to determine the order of events in a meaningful way
- ➔ Hence, in such cases:
  - ➔ events with timestamps of **logical clocks** (👉 6.3)
  - ➔ logical clocks allow conclusions to be made about causal order



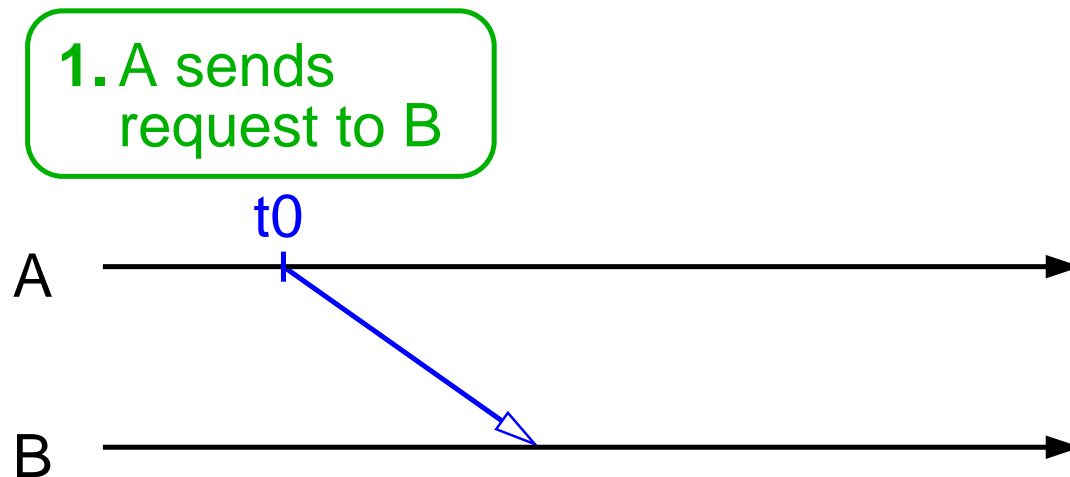
### 6.1 Synchronizing Physical Clocks

- ➔ Physical clock shows 'real' time
  - ➔ based on UTC (Universal Time Coordinated)
- ➔ Each computer has its own (physical) clock
  - ➔ quartz oscillator with counter in HW and if necessary in SW
- ➔ Clocks usually differ from each other (**offset**)
  - ➔ Offset changes over time: **clock drift**
    - ➔ typ.  $10^{-6}$  for quartz crystals,  $10^{-13}$  for atomic clocks
- ➔ Goal of clock synchronization:
  - ➔ keep the offset of the clocks under a given limit
  - ➔ **clock skew**: maximum allowed deviation



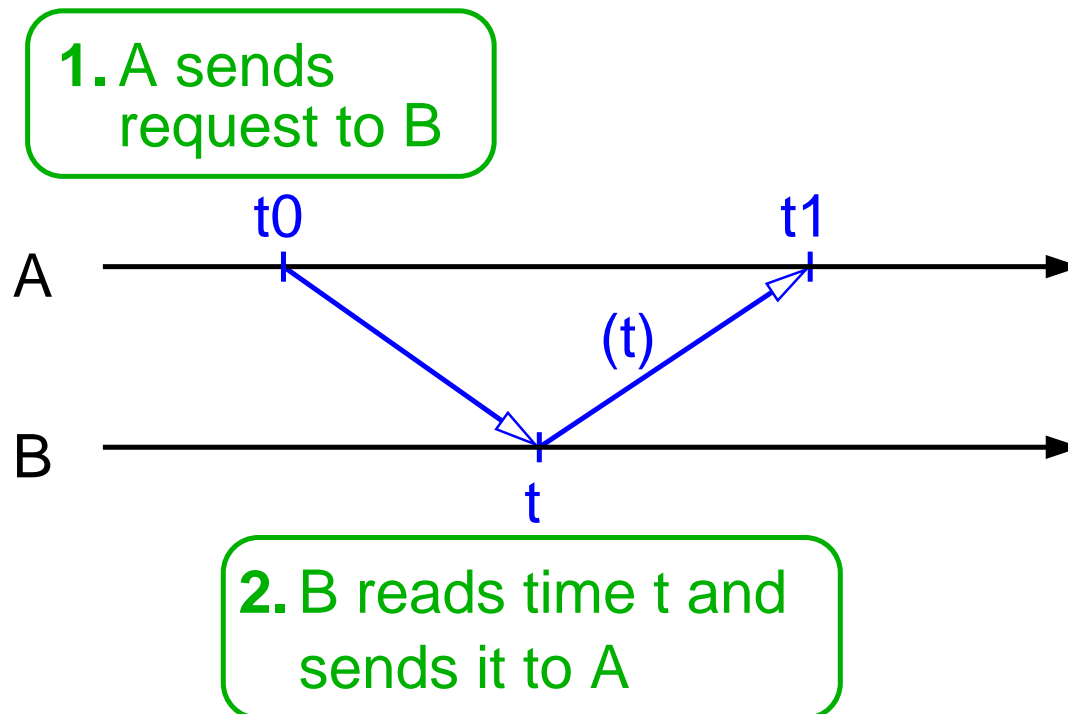
## Cristian's Method

- ➔ Assumption:  $A$  and  $B$  want to synchronize their clocks with each other
  - ➔  $B$  can also be a time server (e.g. with GPS clock)
- ➔ Protocol:



## Cristian's Method

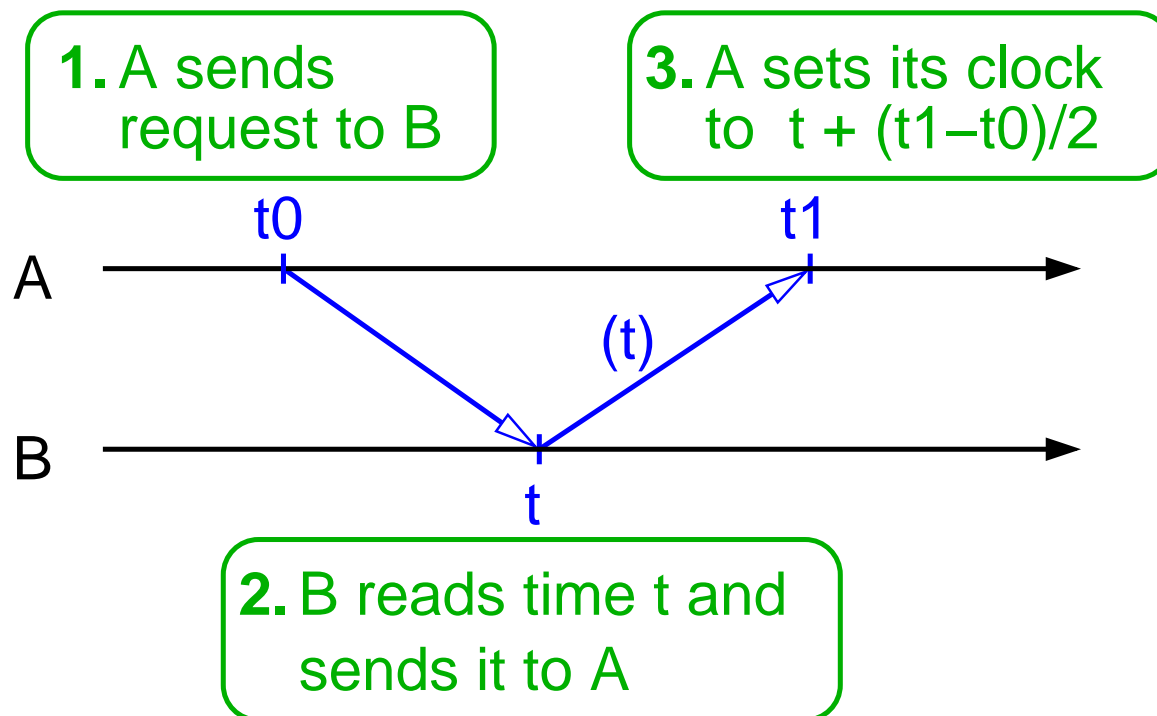
- ➔ Assumption:  $A$  and  $B$  want to synchronize their clocks with each other
  - ➔  $B$  can also be a time server (e.g. with GPS clock)
- ➔ Protocol:





## Cristian's Method

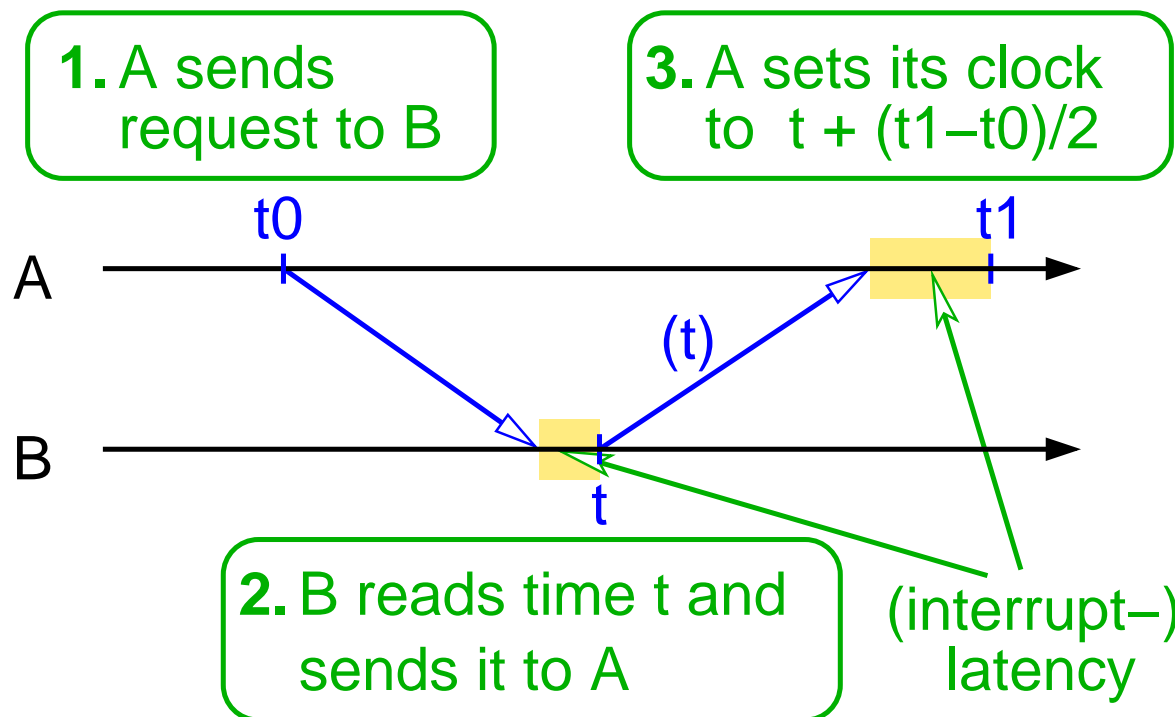
- ➔ Assumption: *A* and *B* want to synchronize their clocks with each other
  - ➔ *B* can also be a time server (e.g. with GPS clock)
- ➔ Protocol:



- ➔ *A* must take the runtime of the reply message into account
- ➔ estimate: runtime = half the round trip time =  $(t_1 - t_0)/2$

## Cristian's Method

- ➔ Assumption: *A* and *B* want to synchronize their clocks with each other
  - ➔ *B* can also be a time server (e.g. with GPS clock)
- ➔ Protocol:

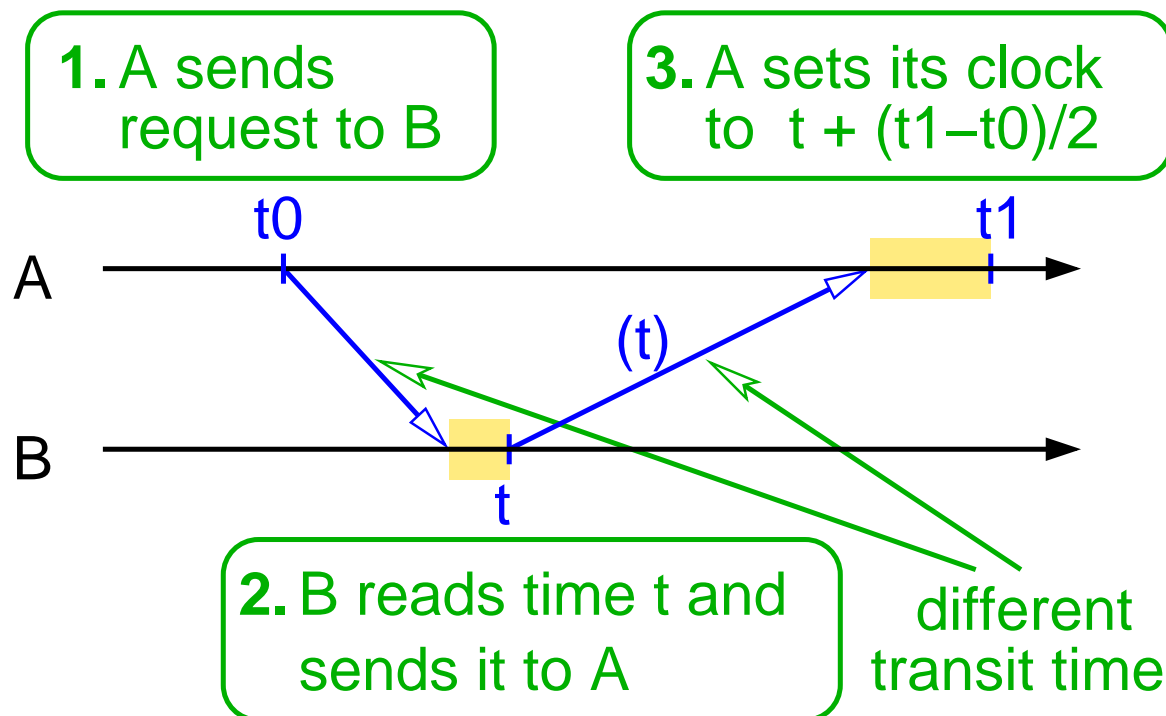


- ➔ *A* must take the runtime of the reply message into account
- ➔ estimate: runtime = half the round trip time =  $(t_1 - t_0)/2$



## Cristian's Method

- ➔ Assumption: *A* and *B* want to synchronize their clocks with each other
  - ➔ *B* can also be a time server (e.g. with GPS clock)
- ➔ Protocol:



- ➔ *A* must take the runtime of the reply message into account
- ➔ estimate: runtime = half the round trip time =  $(t_1 - t_0)/2$



### Cristian's Method: Discussion

- ➔ Problem: runtimes of both messages may be different
  - ➔ systematic differences (different paths / latencies)
  - ➔ statistical fluctuations of the transit time
- ➔ Accuracy estimate, if minimum transit time ( $min$ ) is known:
  - ➔  $B$  can have determined  $t$  at the earliest at time  $t_0 + min$ , at the latest at time  $t_1 - min$  (measured with  $A$ 's clock)
  - ➔ thus accuracy  $\pm ((t_1 - t_0)/2 - min)$
- ➔ To improve accuracy:
  - ➔ execute the message exchange multiple times
  - ➔ use the one with minimum round trip time



## Adjusting the clock

- ➔ Turning back is problematic
  - ➔ order / uniqueness of time stamps
- ➔ Non-monotonous “jumping” of the time also problematic
- ➔ Therefore: clock is generally adjusted slowly
  - ➔ runs faster / slower, until clock skew has been compensated

## Further protocols

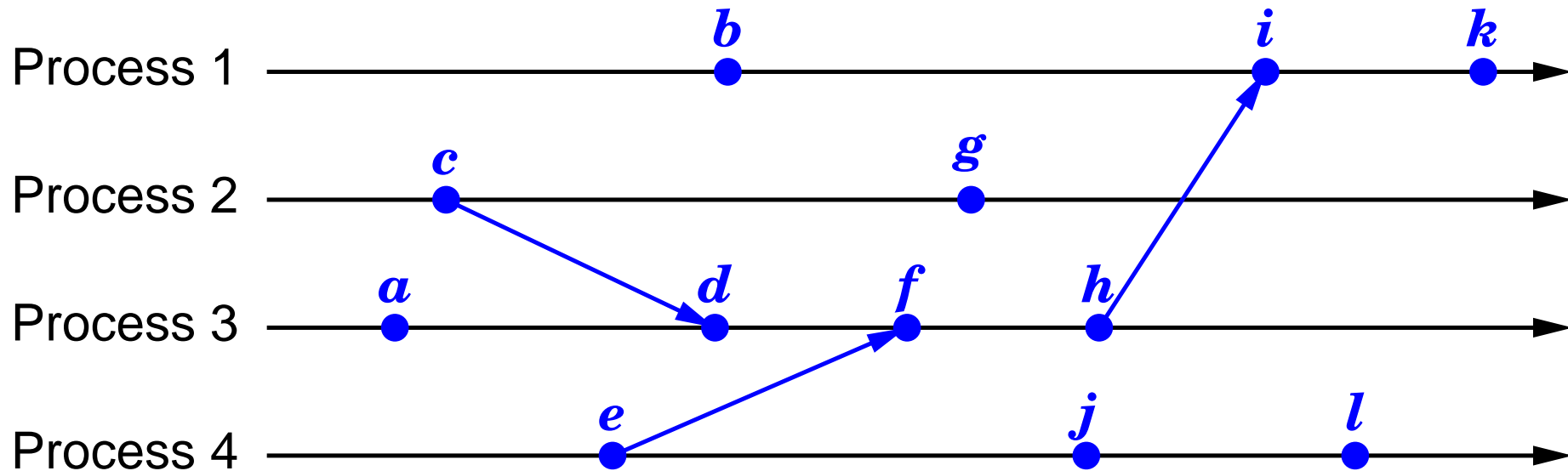
- ➔ Berkeley algorithm: server calculates mean value of all clocks
- ➔ NTP (Network Time Protocol): hierarchy of time servers in the Internet with periodic synchronization
- ➔ IEEE 1588: clock synchronization for automation systems



### 6.2 Lamport's Happened-Before Relation

- ➔ In two cases, the order of events can also be determined without a global clock:
  - ➔ if the events are in the same process, local clock is sufficient
  - ➔ the sending of a message is always before its reception
- ➔ Definition of the happened-before causality relation  $\rightarrow$  (*causality relation*)
  - ➔ if events  $a, b$  are in the same process  $i$  and  $t_i(a) < t_i(b)$  ( $t_i$ : time stamp with  $i$ 's clock), then  $a \rightarrow b$
  - ➔ if  $a$  is the sending of a message and  $b$  its receipt, then  $a \rightarrow b$
  - ➔ if  $a \rightarrow b$  and  $b \rightarrow c$ , then also  $a \rightarrow c$  (transitivity)
- ➔  $a \rightarrow b$  means, that  $b$  **may** causally depend on  $a$

### Examples



➡ Among others, we have here:

➡  $b \rightarrow i$  and  $a \rightarrow h$  (events in the same process)

➡  $c \rightarrow d$  and  $e \rightarrow f$  (sending / receiving a message)

➡  $c \rightarrow k$  and  $a \rightarrow i$  (transitivity)

➡  $g \not\rightarrow l$  and  $l \not\rightarrow g$ :  $l$  and  $g$  are **concurrent** (*nebenläufig*)

---

# Distributed Systems

Winter Term 2025/26

27.11.2025

Roland Wismüller  
Universität Siegen  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: November 27, 2025



### 6.3 Logical Clocks

- ➔ Physical clocks cannot be synchronized exactly
  - ➔ therefore: unsuitable for determining the **order** in which events occurred
- ➔ Logical clocks
  - ➔ refer to the causal order of events (happened-before relation)
  - ➔ no fixed relationship to real time
- ➔ In the following:
  - ➔ Lamport timestamps
    - ➔ are consistent with the happened-before relation
  - ➔ vector timestamps
    - ➔ allow sorting of events according to causality (i.e. happened-before relation)

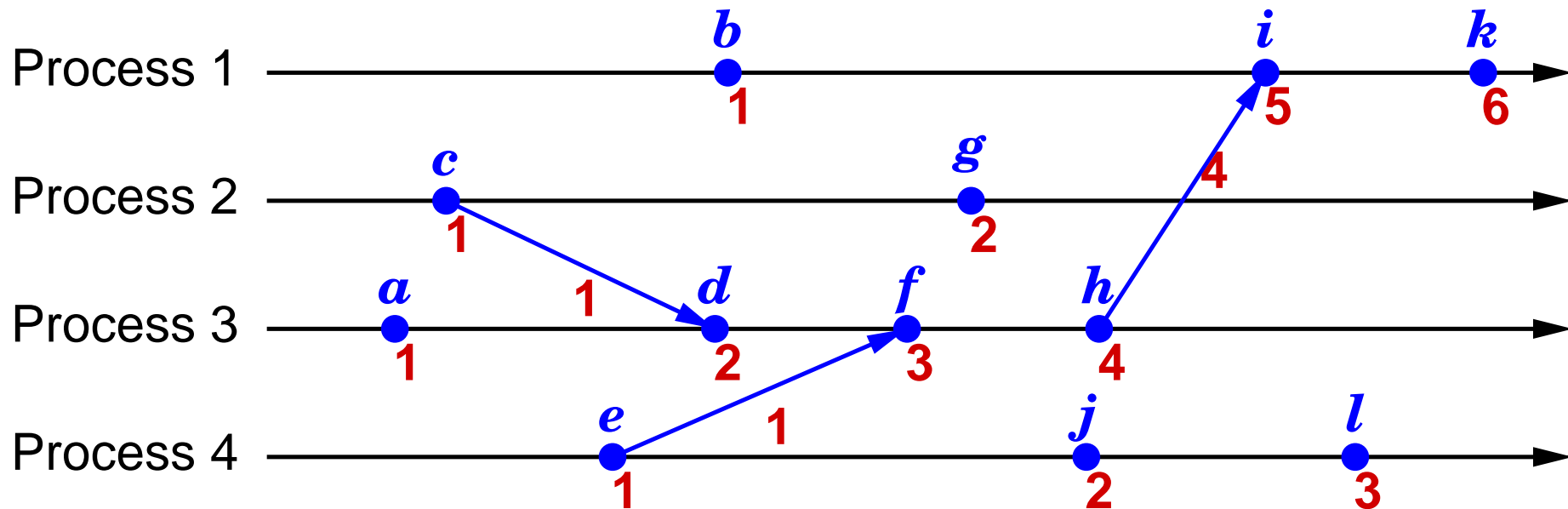


### Lamport Timestamps

- ➔ Lamport timestamps are natural numbers
- ➔ Each process  $i$  has a local counter  $L_i \in \mathbb{N}_0$
- ➔  $L_i$  is updated as follows:
  - ➔ at (more precisely: before) each local event:  $L_i := L_i + 1$
  - ➔ in each message, the time stamp  $L_i$  of the send event is also sent
  - ➔ at receipt of a message with time stamp  $t$ :  
 $L_i := \max(L_i, t + 1)$
- ➔ Lamport time stamps are consistent with the causality:
  - ➔  $a \rightarrow b \Rightarrow L(a) < L(b)$ , where  $L$  is the Lamport timestamp in the respective process
  - ➔ but the reversal does not apply!

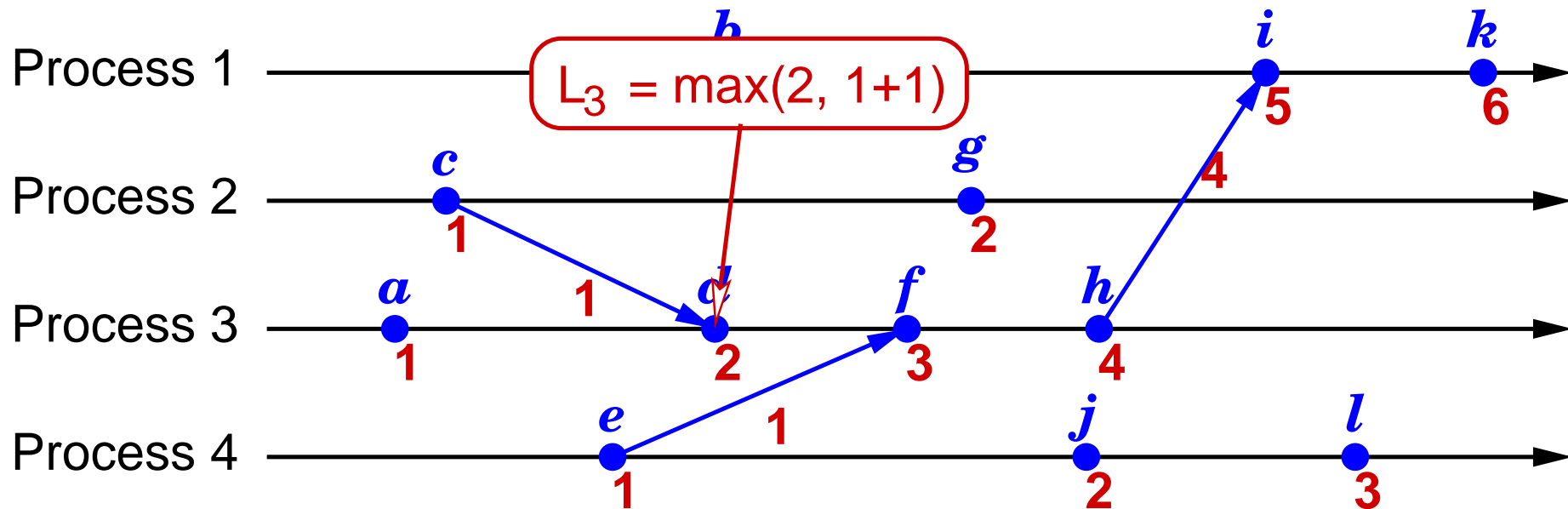


## Lamport Timestamps: Example



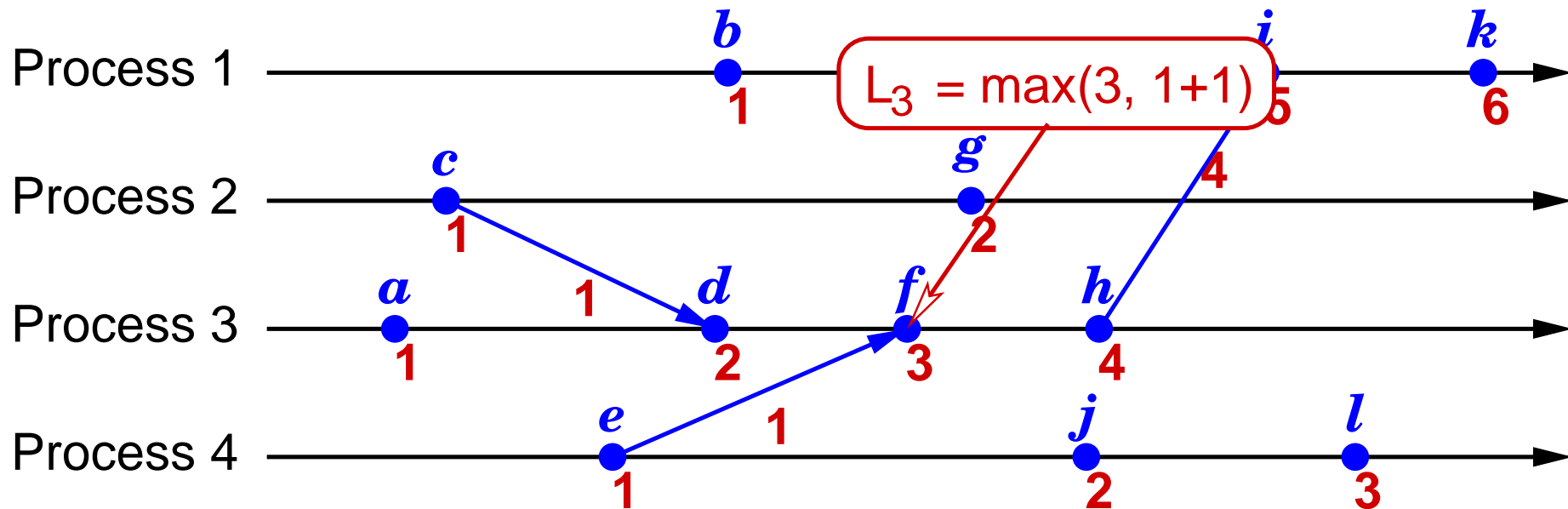


## Lamport Timestamps: Example



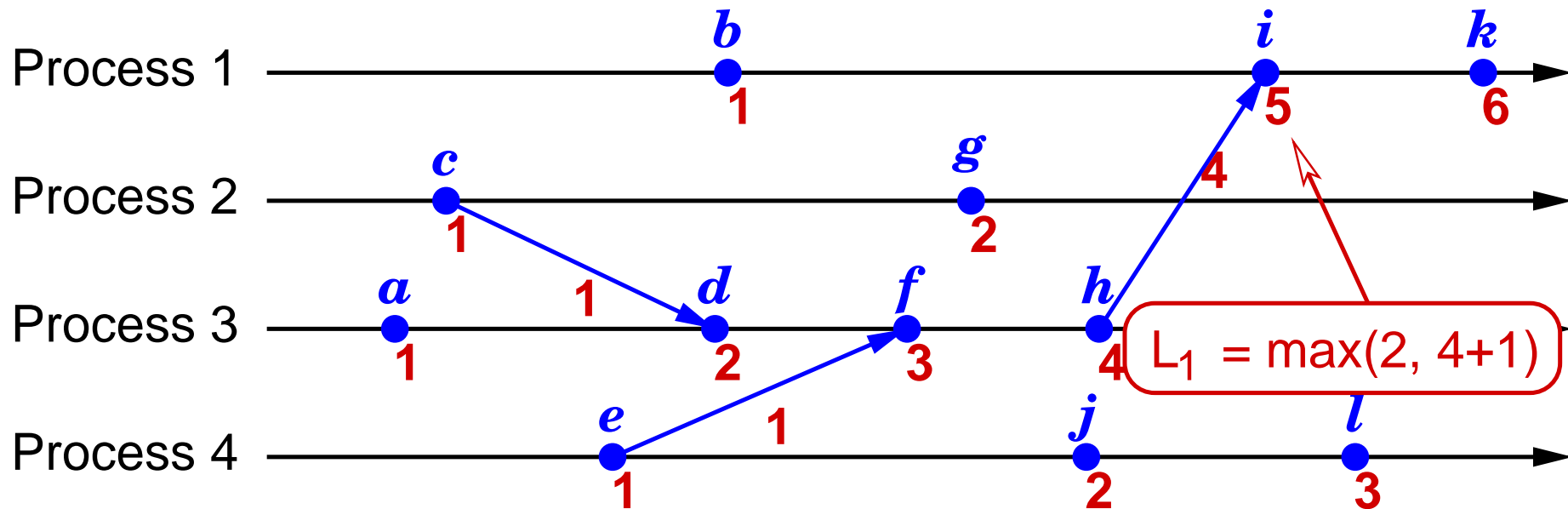


## Lamport Timestamps: Example

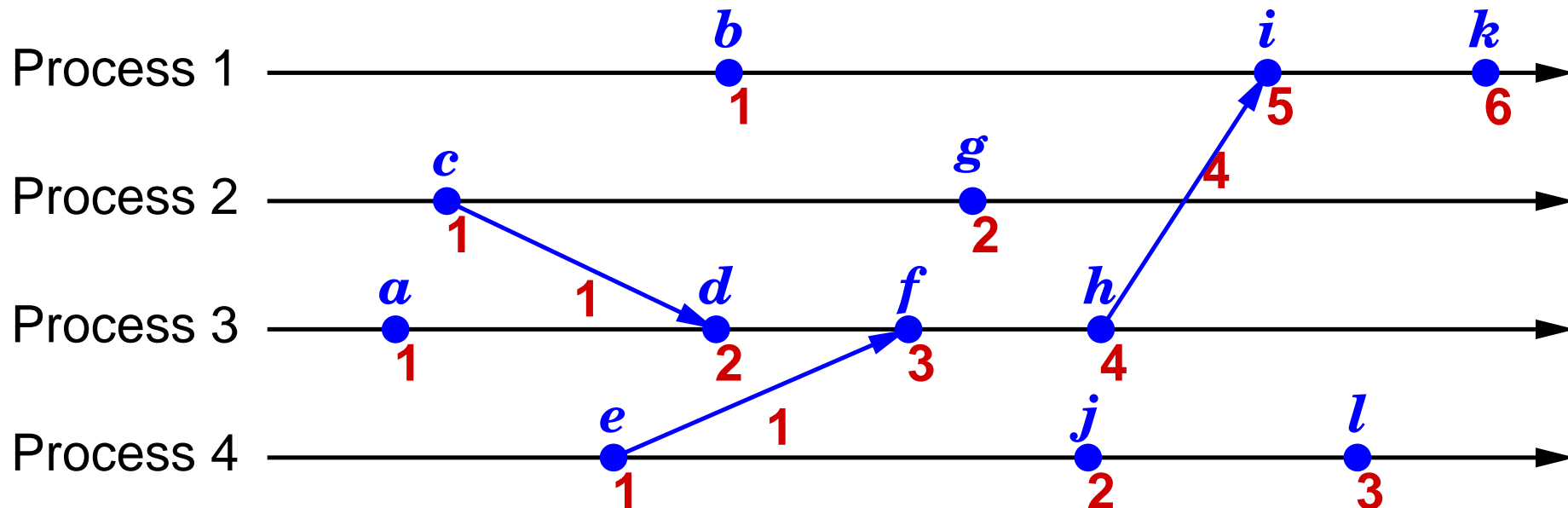




## Lamport Timestamps: Example



### Lamport Timestamps: Example



➔ Among others, we have here:

➔  $c \rightarrow k$  and  $L(c) < L(k)$

➔  $g \not\rightarrow j$  and  $L(g) \not< L(j)$

➔  $g \not\rightarrow l$ , but still  $L(g) < L(l)$



### Vector Timestamps

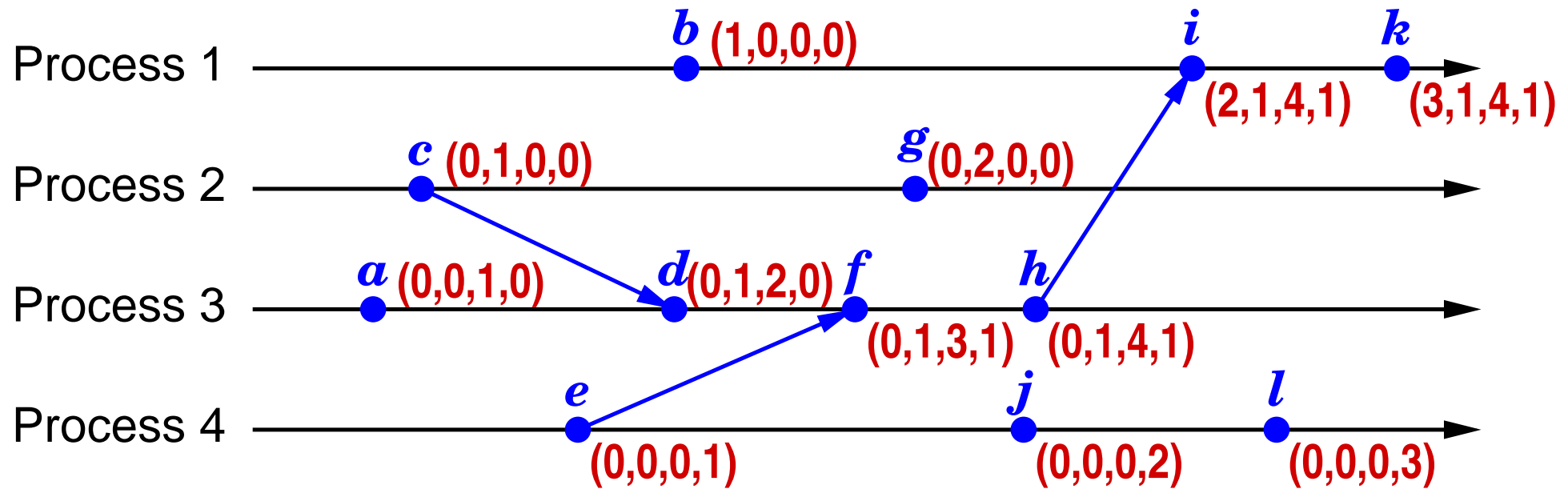
- ➔ Objective: timestamps that characterize causality
  - ➔  $a \rightarrow b \Leftrightarrow V(a) < V(b)$ , where  $V$  is the vector timestamp in the respective process
- ➔ Vector timestamps
  - ➔ in a system with  $N$  processes, each process has its own vector  $V_i \in (\mathbb{N}_0)^N$  of  $N$  integers
  - ➔  $V_i[i]$ : number of events that have occurred so far in process  $i$
  - ➔  $V_i[j]$ ,  $j \neq i$ : number of events in process  $j$ , of which  $i$  knows
    - ➔ i.e. by which it could have been causally influenced



### Vector Timestamps ...

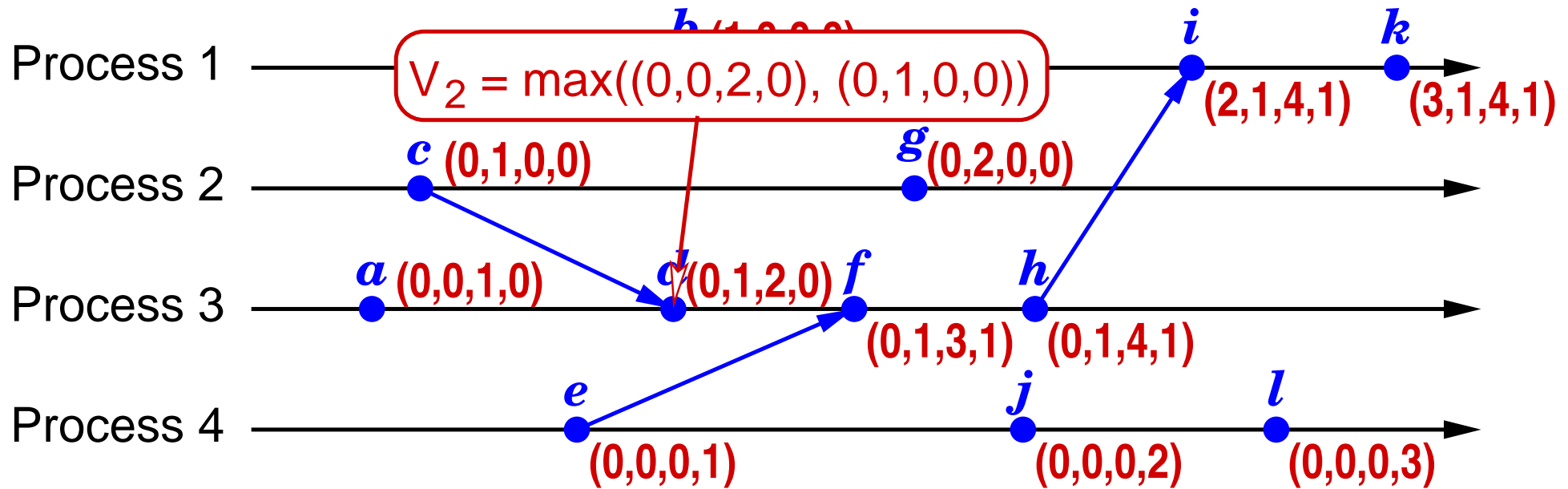
- ➔ Update of  $V_i$  in process  $i$ :
  - ➔ before any local event:  $V_i[i] := V_i[i] + 1$
  - ➔  $V_i$  is included in every message sent
  - ➔ when receiving a message with timestamp  $t$ :  
 $V_i[j] := \max(V_i[j], t[j])$  for all  $j = 1, 2, \dots, N$
- ➔ Comparison of vector timestamps:
  - ➔  $V = V' \Leftrightarrow V[j] = V'[j]$  for all  $j = 1, 2, \dots, N$
  - ➔  $V \leq V' \Leftrightarrow V[j] \leq V'[j]$  for all  $j = 1, 2, \dots, N$
  - ➔  $V < V' \Leftrightarrow V \leq V' \wedge V \neq V'$
  - ➔ the relation  $<$  defines a **partial** order

### Vector Timestamps: Example



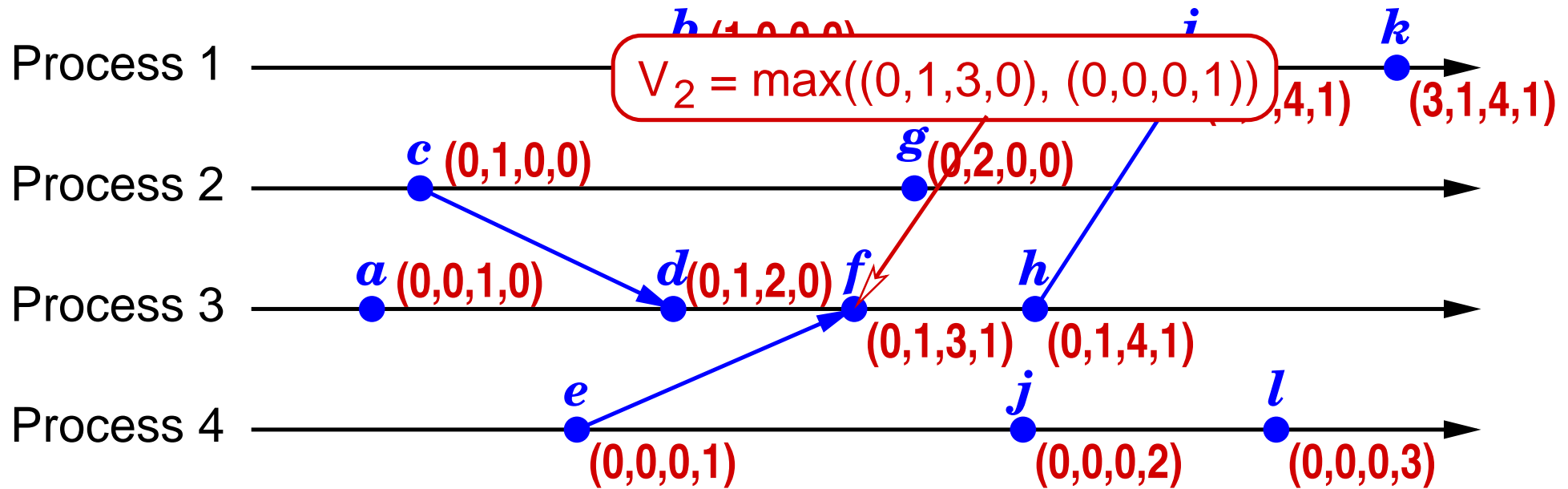


## Vector Timestamps: Example



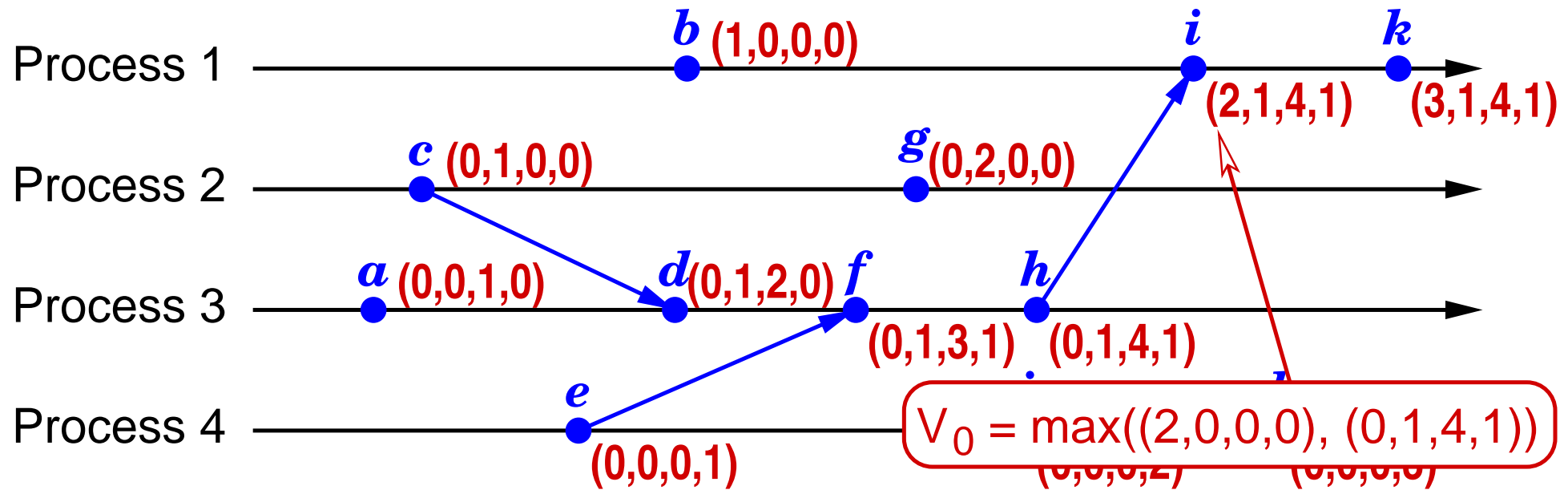


## Vector Timestamps: Example

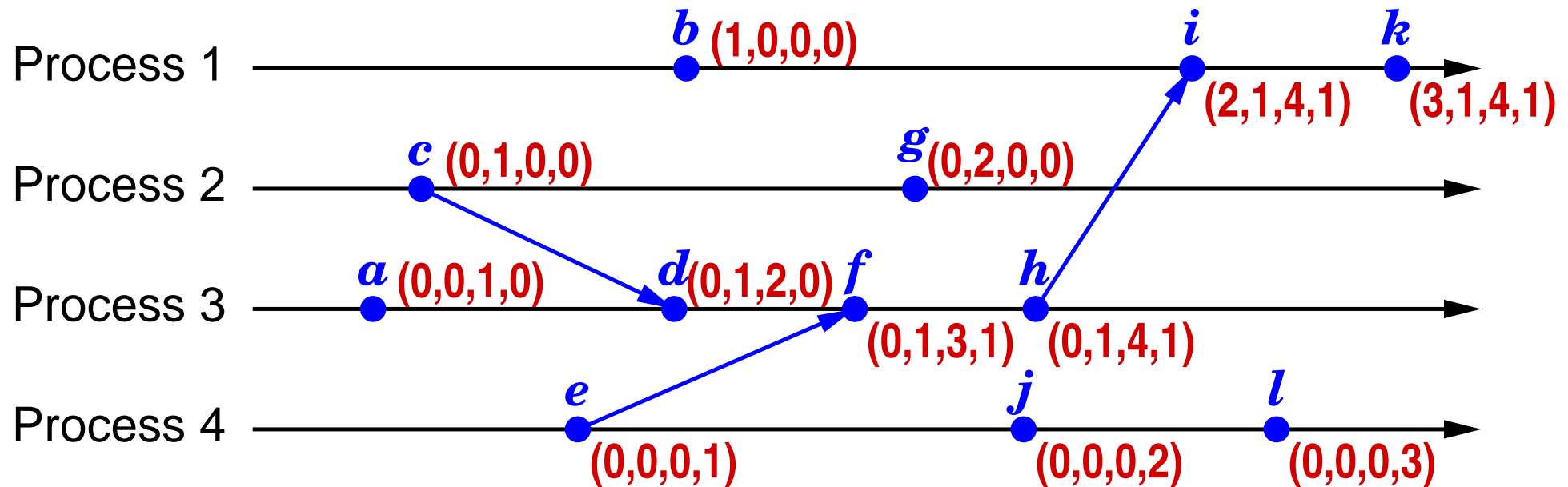




## Vector Timestamps: Example



### Vector Timestamps: Example



➔ Among others, we have here:

➔  $c \rightarrow k$  and  $V(c) < V(k)$

➔  $g \not\rightarrow l$  and  $V(g) \not\leq V(l)$ , as well as  $l \not\rightarrow g$  and  $V(l) \not\leq V(g)$

➔  $V(l)$  and  $V(g)$  not comparable  $\Leftrightarrow l$  and  $g$  concurrent



### A Motivating Example

- ➔ Scenario: peer-to-peer application, processes send requests to each other
- ➔ Question: when can the application terminate?
- ➔ Answer: when no process is processing a request

### A Motivating Example

- ➔ Scenario: peer-to-peer application, processes send requests to each other
- ➔ Question: when can the application terminate?
- ➔ **Wrong** answer: when no process is processing a request
  - ➔ reason: requests can still be on the way in messages!



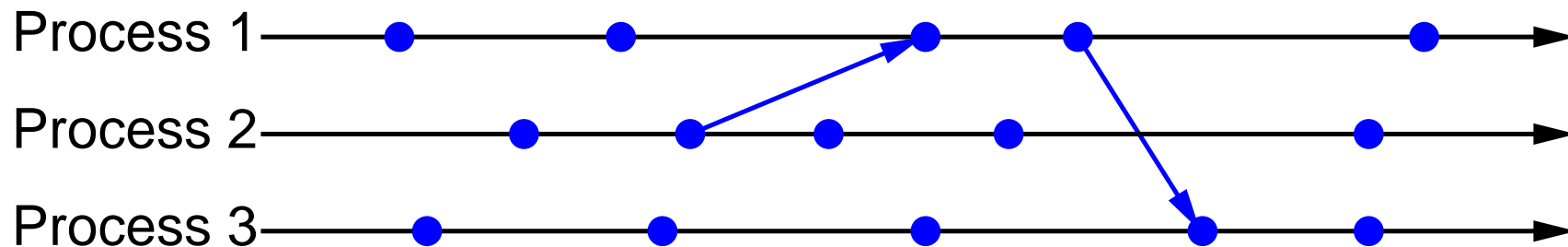
- ➔ Other applications: distributed garbage collection, distributed deadlock detection, ...



- ➔ How can we determine the overall state of a distributed process system?
  - ➔ naïvely: union of the states of all processes (**wrong!**)
- ➔ Two aspects have to be considered:
  - ➔ messages that are still in transit
    - ➔ must be included in the state
  - ➔ lack of global time
    - ➔ a global state at time  $t$  cannot be defined!
    - ➔ process states always refer to local (and thus different) times
    - ➔ question: condition on local times?  $\Rightarrow$  **consistent cuts**

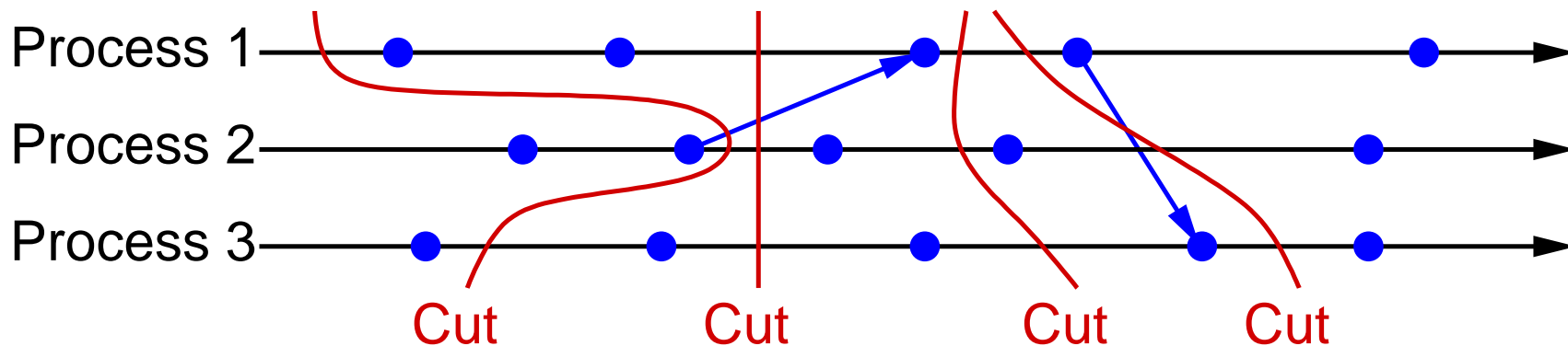
### Consistent Cuts

- ➔ Objective: build a meaningful global state from local states (which are not determined simultaneously)
- ➔ Processes are modeled by sequences of events:



### Consistent Cuts

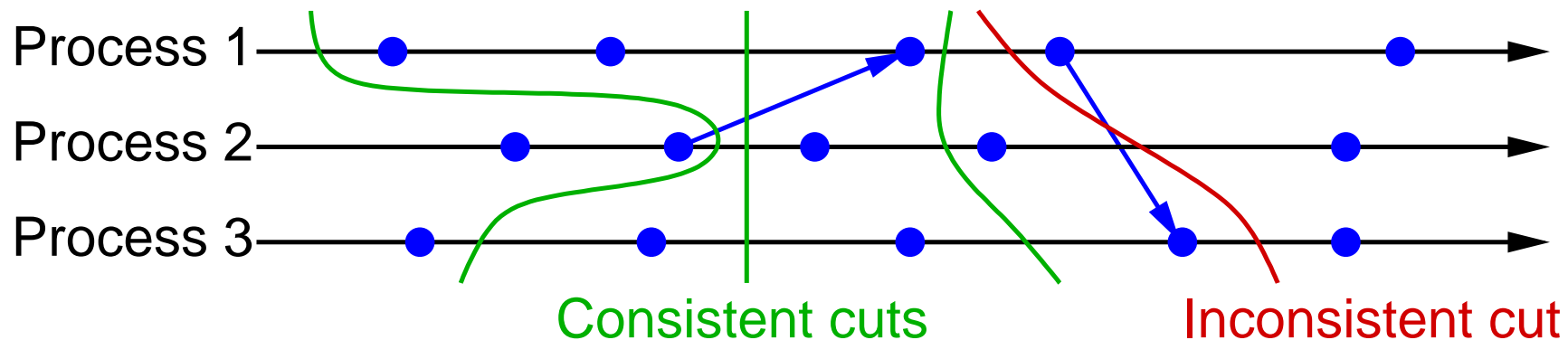
- ➔ Objective: build a meaningful global state from local states (which are not determined simultaneously)
- ➔ Processes are modeled by sequences of events:



- ➔ **Cut**: consider a **prefix** of the event sequence in each process

### Consistent Cuts

- ➔ Objective: build a meaningful global state from local states (which are not determined simultaneously)
- ➔ Processes are modeled by sequences of events:



- ➔ **Cut**: consider a **prefix** of the event sequence in each process
- ➔ **Consistent cut**:
  - ➔ if the cut contains the reception of a message, it also contains the sending of this message



### The Snapshot Algorithm of Chandy and Lamport

- ➔ Determines online a “snapshot” of the global state
  - ➔ i.e.: a consistent cut
- ➔ The global state consists of:
  - ➔ the local states of all processes
  - ➔ the status of all communication connections
    - ➔ i.e. the messages in transmission
- ➔ Assumptions / properties:
  - ➔ reliable message channels with sequence retention
  - ➔ process graph is strongly connected
  - ➔ each process can trigger a snapshot at any time
  - ➔ the processes are not blocked during the algorithm



### The Snapshot Algorithm of Chandy and Lamport ...

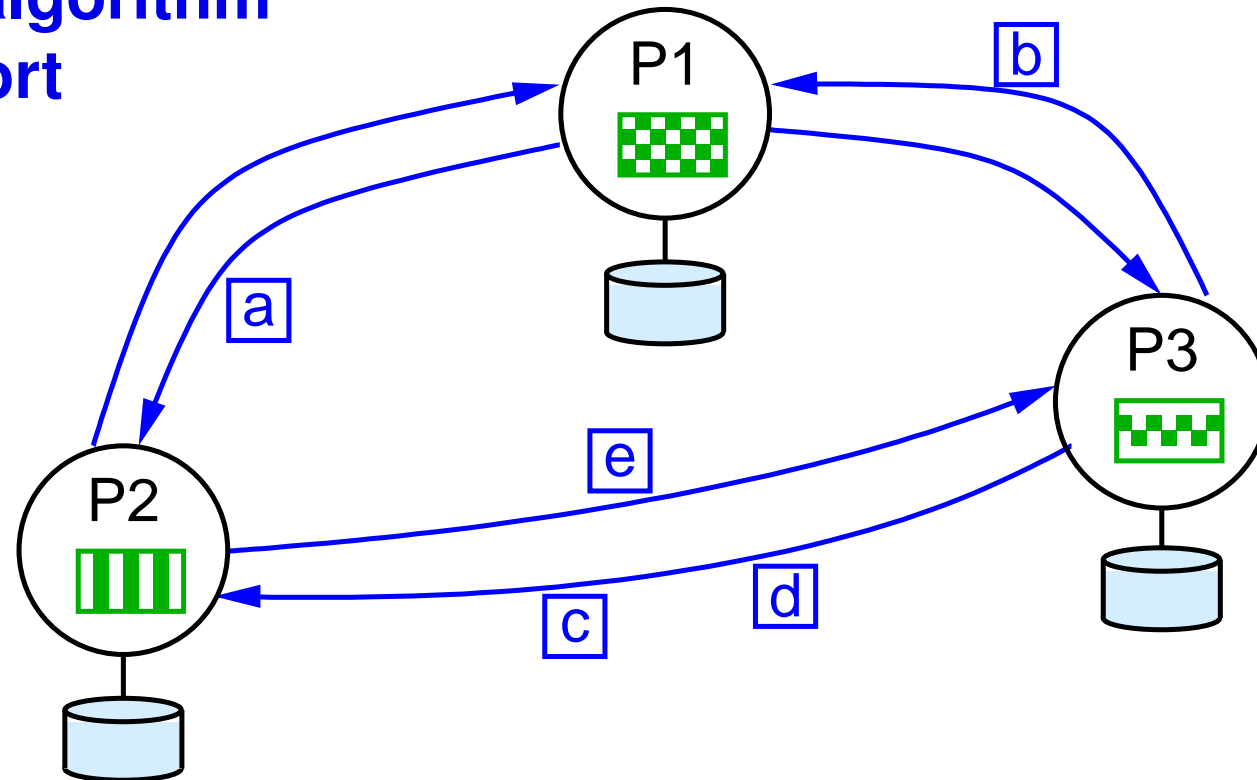
- ➔ When a process wants to initiate a snapshot:
  - ➔ process first saves its local state
  - ➔ then it sends a marker message over each outgoing channel
- ➔ When a process receives a marker message:
  - ➔ if it has not yet saved its local state:
    - ➔ it saves its local state
    - ➔ and sends a marker over each outgoing channel
  - ➔ else:
    - ➔ for the channel where the marker was received, it saves all messages that have been received since the local state was saved
    - ➔ i.e., it records the status of the channel



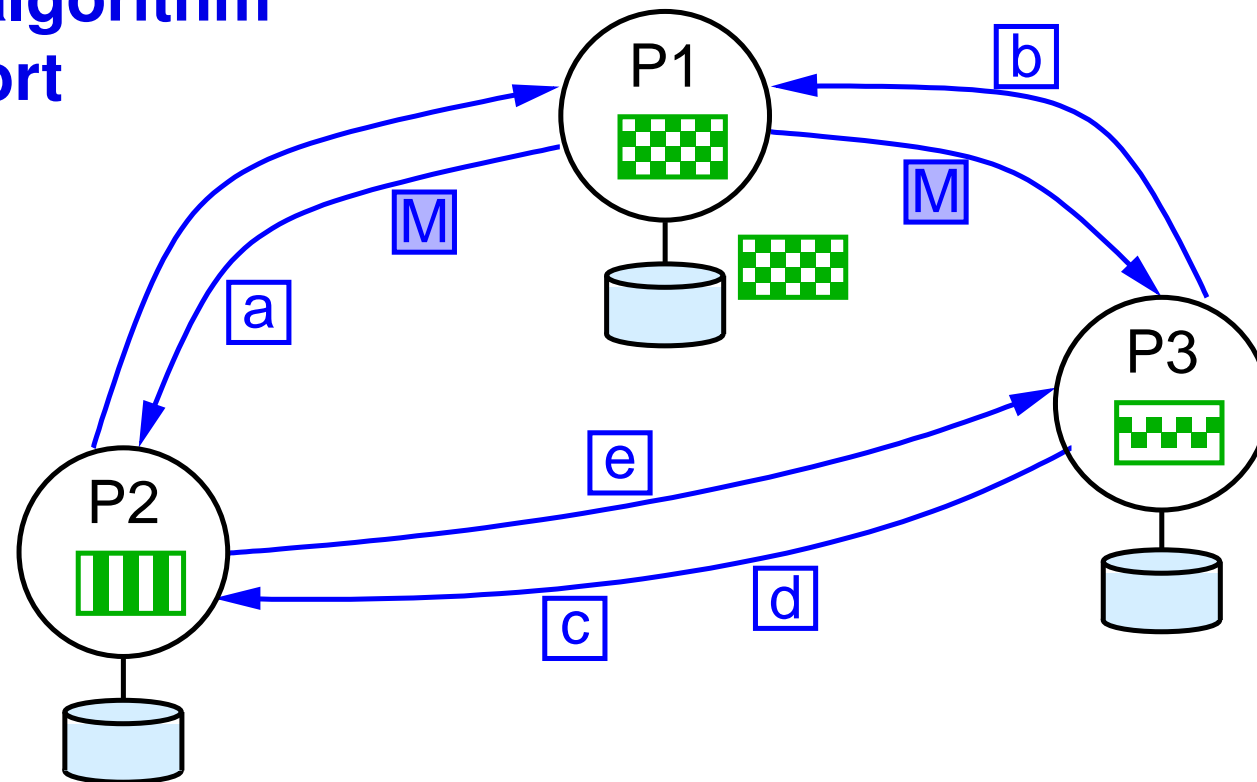
### The Snapshot Algorithm of Chandy and Lamport ...

- ➔ The algorithm terminates when each process has received a marker message on each channel
  - ➔ the determined consistent section is then (initially) stored in a distributed way

### Example for the algorithm of Chandy/Lamport

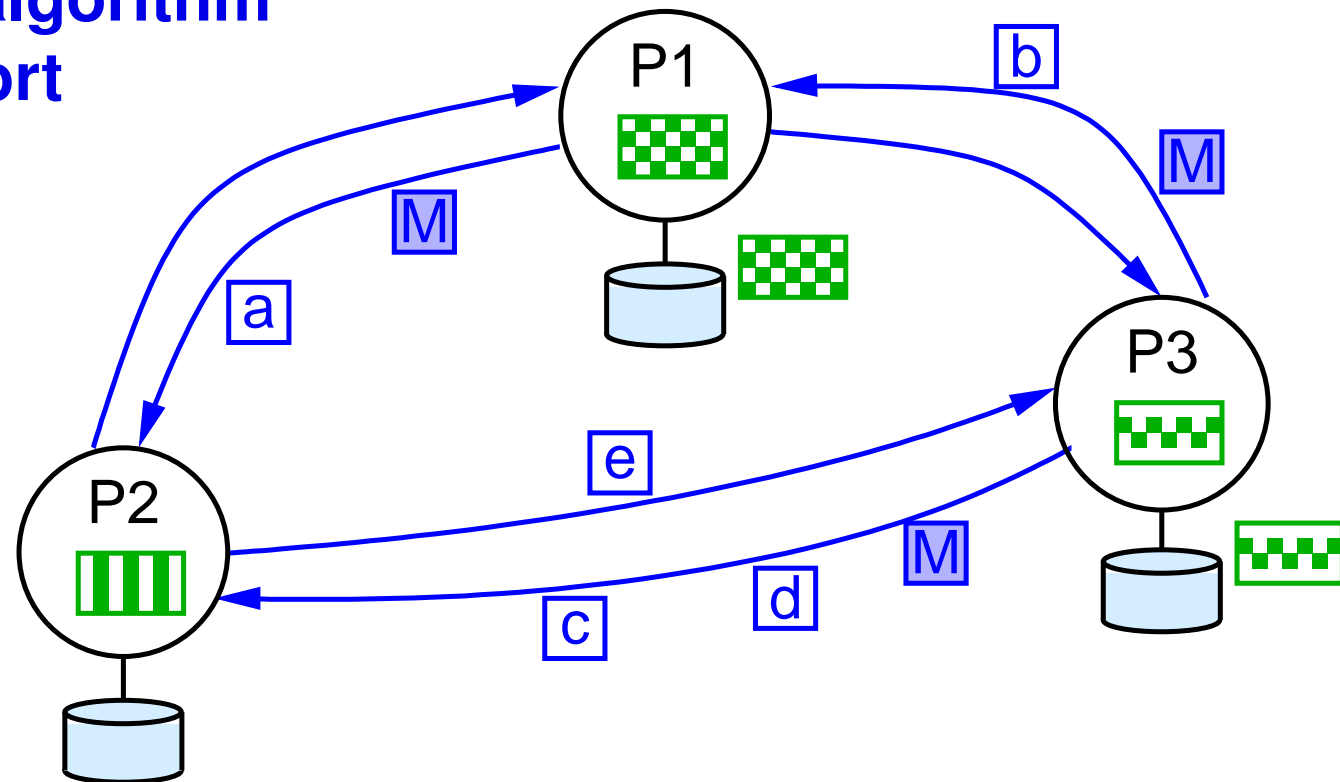


### Example for the algorithm of Chandy/Lampert



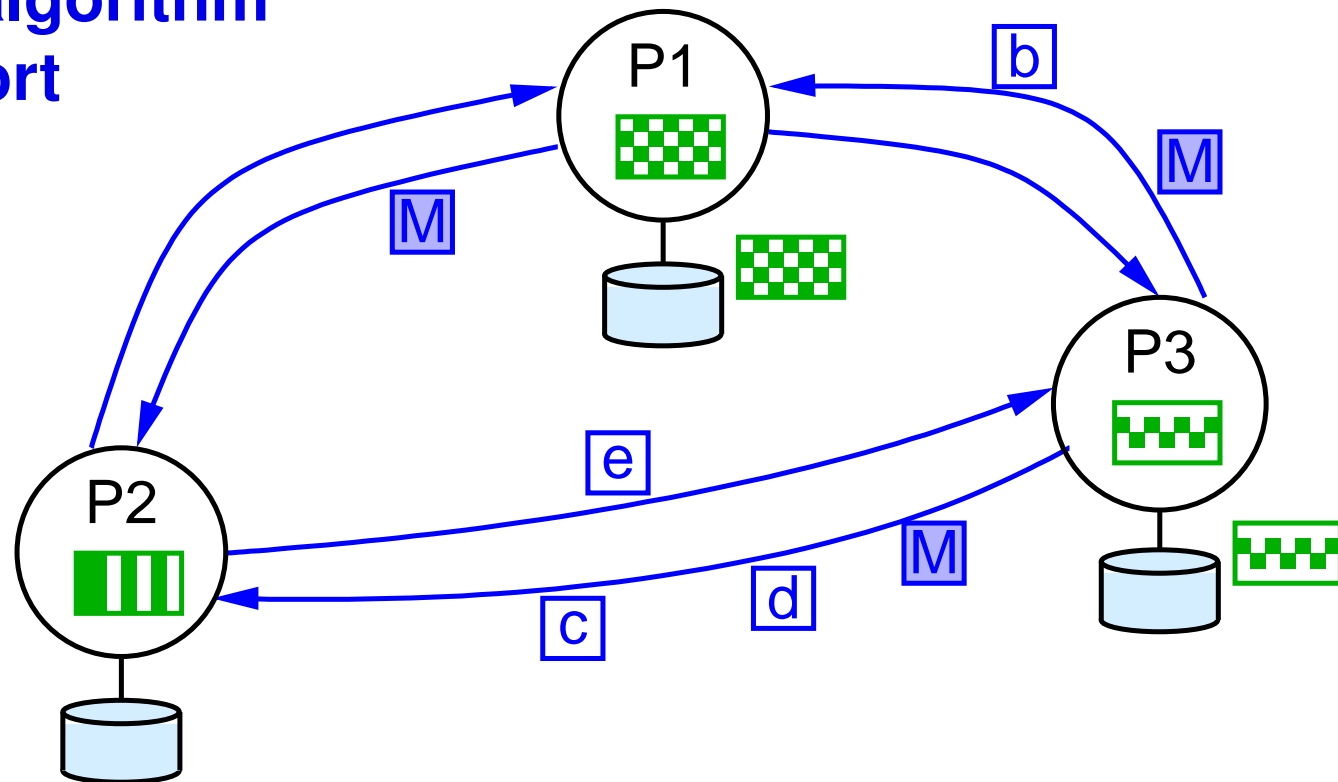
1. P1 initiates a snapshot, saves its state, and sends markers

### Example for the algorithm of Chandy/Lampert



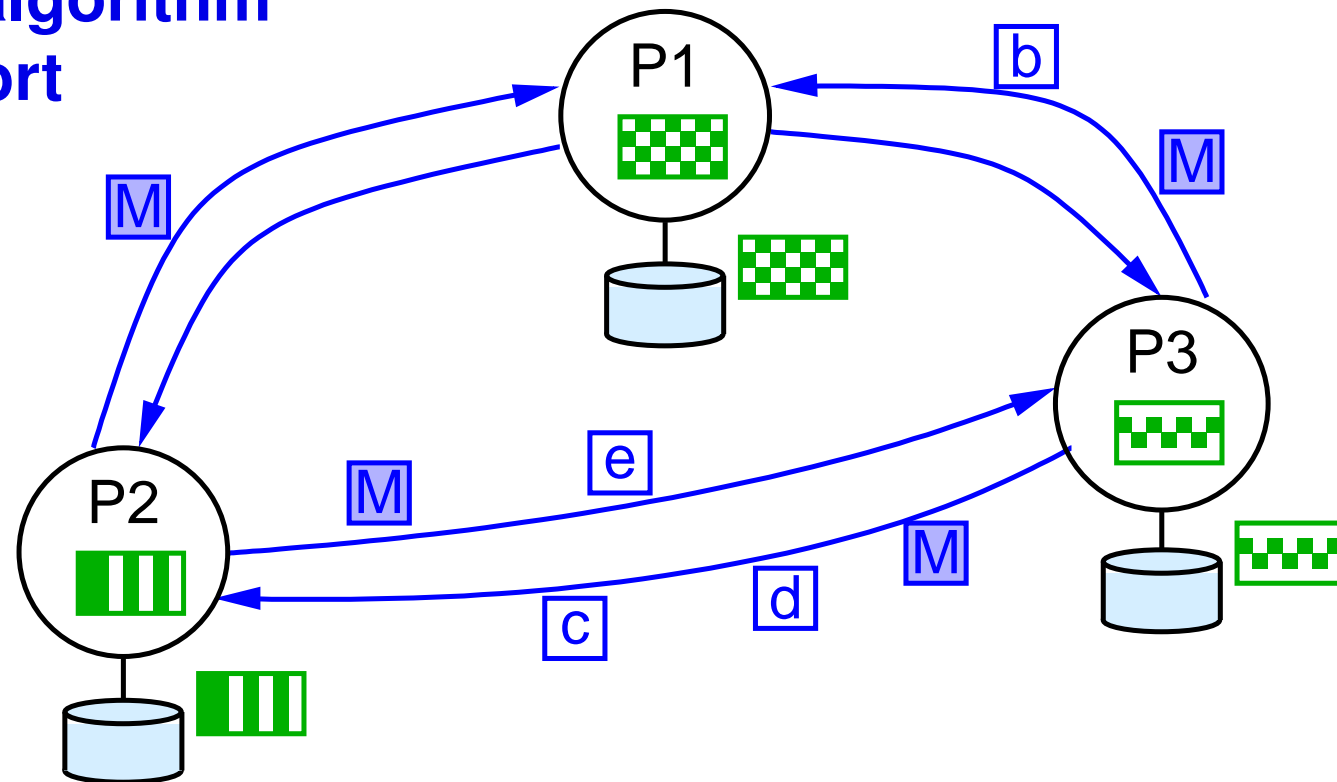
1. P1 initiates a snapshot, saves its state, and sends markers
2. P3 receives a marker from P1, saves its state, and sends markers

### Example for the algorithm of Chandy/Lamport



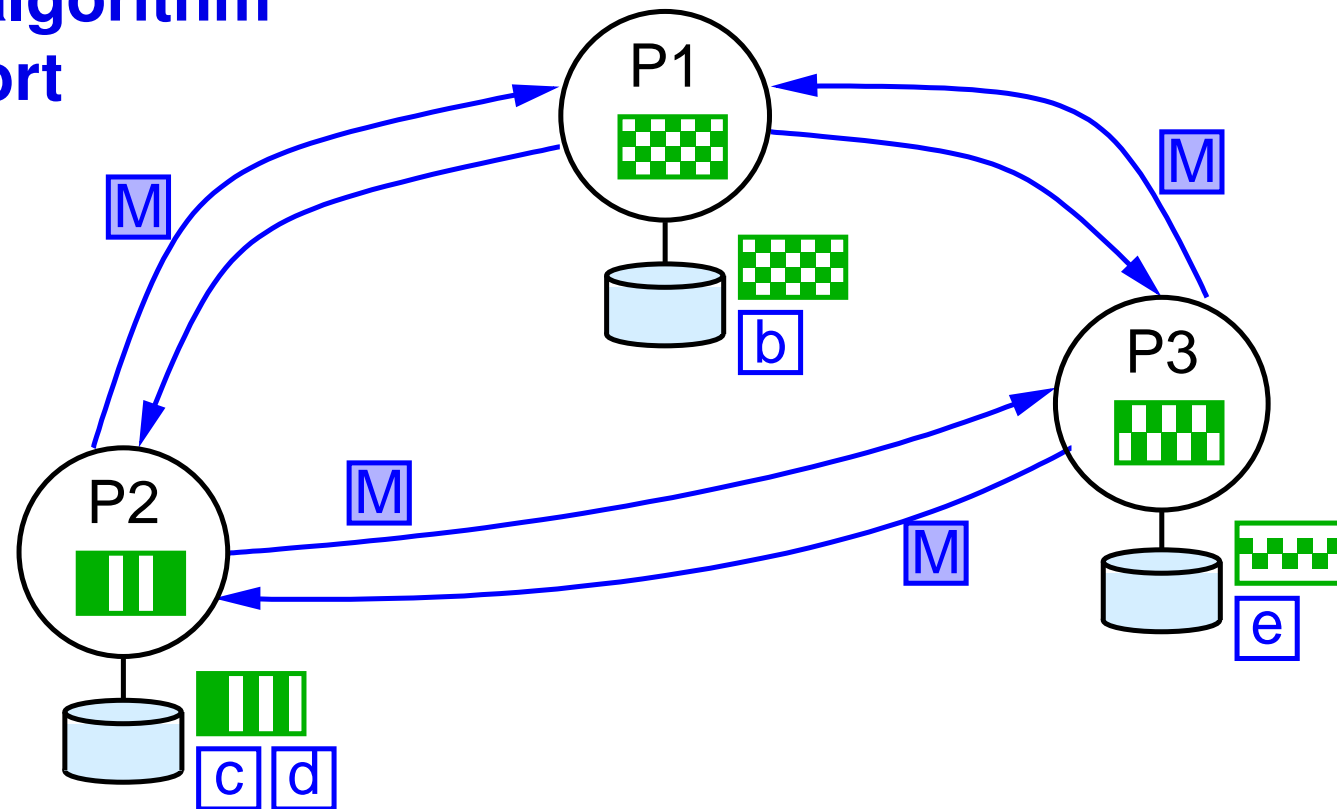
1. P1 initiates a snapshot, saves its state, and sends markers
2. P3 receives a marker from P1, saves its state, and sends markers
3. P2 receives and processes a

### Example for the algorithm of Chandy/Lamport



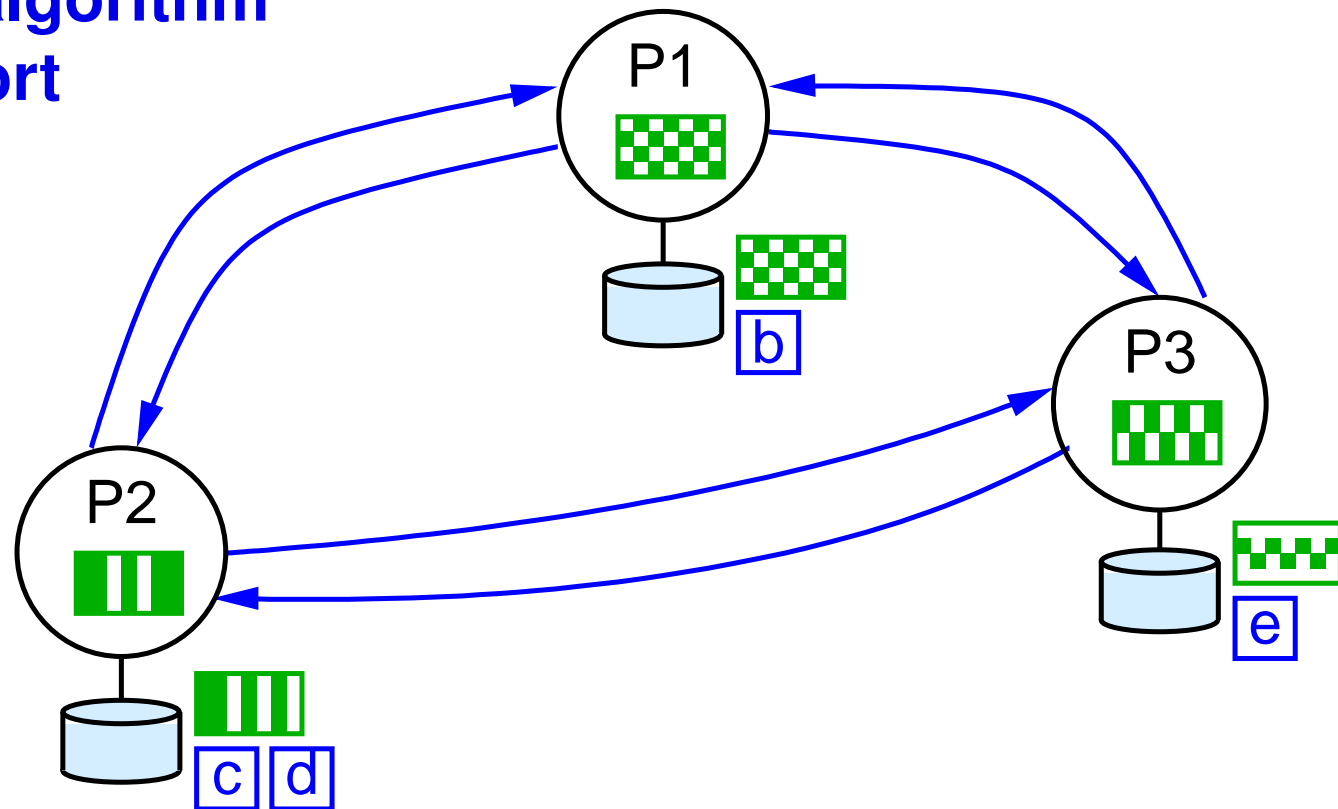
1. P1 initiates a snapshot, saves its state, and sends markers
2. P3 receives a marker from P1, saves its state, and sends markers
3. P2 receives and processes a  
P2 receives the marker from P1, saves its state, and sends markers

### Example for the algorithm of Chandy/Lampert



1. P1 initiates a snapshot, saves its state, and sends markers
2. P3 receives a marker from P1, saves its state, and sends markers
3. P2 receives and processes a  
P2 receives the marker from P1, saves its state, and sends markers
4. P1, P2, P3 save the incoming messages, until all markers are received

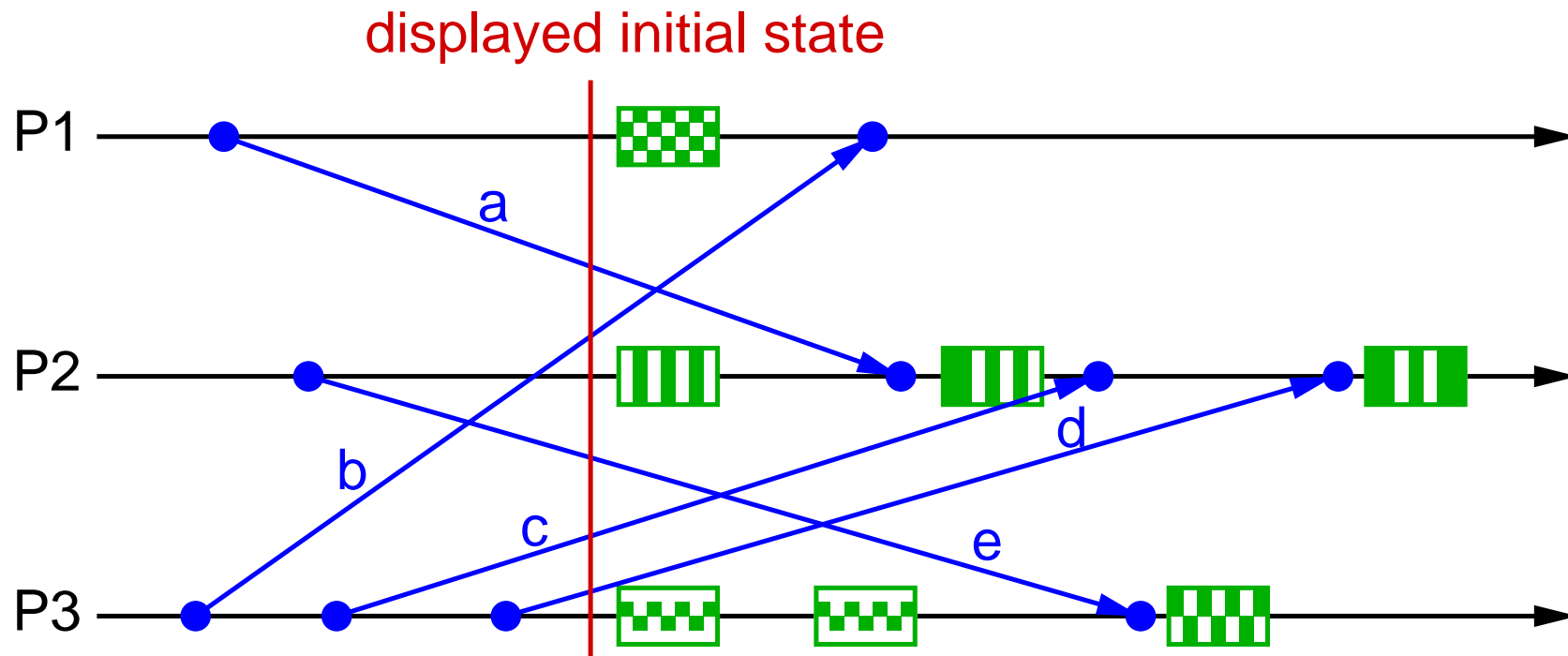
### Example for the algorithm of Chandy/Lampert



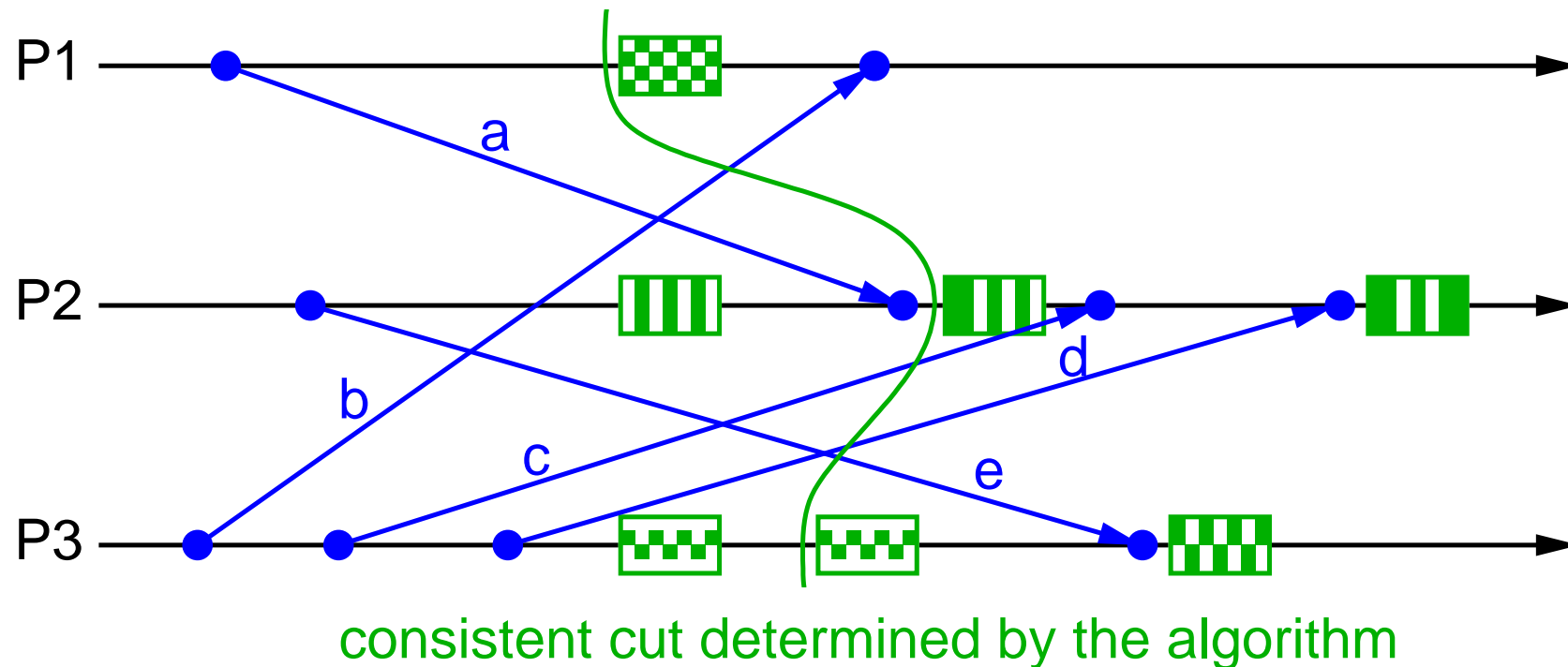
1. P1 initiates a snapshot, saves its state, and sends markers
2. P3 receives a marker from P1, saves its state, and sends markers
3. P2 receives and processes a  
P2 receives the marker from P1, saves its state, and sends markers
4. P1, P2, P3 save the incoming messages, until all markers are received



## Sequence in the Example and Selected Cut



### Sequence in the Example and Selected Cut



- ➔ The cut consists of the local states of P1, P2, P3 and the messages b, c, d, e