

# **TDDD25**

## **Distributed Systems**

# **Time and State in Distributed Systems**

**Christoph Kessler**

IDA  
Linköping University  
Sweden

2024

# Agenda

## TIME AND STATE IN DISTRIBUTED SYSTEMS

1. Time in Distributed Systems
2. Lamport's Logical Clocks
3. Vector Clocks
4. Causal Ordering of Messages
5. Global States and their Consistency
6. Cuts of a Distributed Computation
7. Recording of a Global State ("Snapshot")

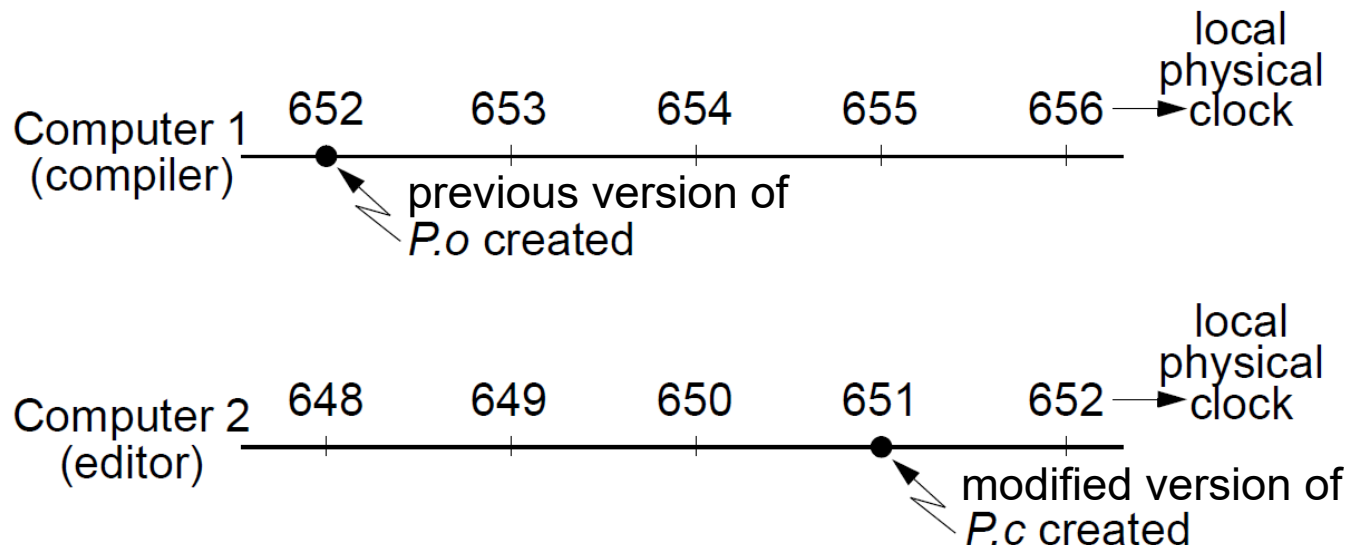
# Time in Distributed Systems


- Because each machine in a distributed system has its own clock, **there is no notion of global physical time.**
  - The  $n$  crystals on the  $n$  computers will run at slightly different rates, causing the clocks gradually to get out of synchronization and give different values.
- **Problems:**
  - **Time triggered systems:** systems in which activities are scheduled to occur at predefined moments in time.  
If activities are to be coordinated over a distributed system, we need a coherent notion of time.
    - ▶ Example: time-triggered real-time systems
  - Maintaining the **consistency** of distributed data is often based on the **time** when a certain **modification** has been performed.
    - ▶ Example: a *make* program →

# Time in Distributed Systems

## The *make*-program example

- When the programmer has finished changing some source files, she starts *make*
  - make*** examines the times at which object and source files were last modified, and decides which sources have to be (re)compiled



- Although *P.c* is modified *after* *P.o* has been generated, because of the clock drift the time assigned to *P.c* is smaller.  
→ *P.c* will **not** be recompiled for the new version! 

# Time in Distributed Systems

## Solutions:

- **Synchronization of physical clocks**
  - Computer clocks are synchronized with one another to an achievable, known, degree of accuracy
    - within the bounds of this accuracy, we can coordinate activities on different computers using each computer's local clock.
  - Clock synchronization is needed for distributed real-time systems.
- **Logical clocks**
  - In many applications we are not interested in the physical time at which events occur; what is important is the **relative order** of events.
    - ▶ The *make-program* is such an example.
  - In such situations we do not need synchronized physical clocks.
  - Relative ordering is based on a *virtual* notion of time - **logical time**.
  - Logical time is implemented using **logical clocks**.

# Lamport's Logical Clocks

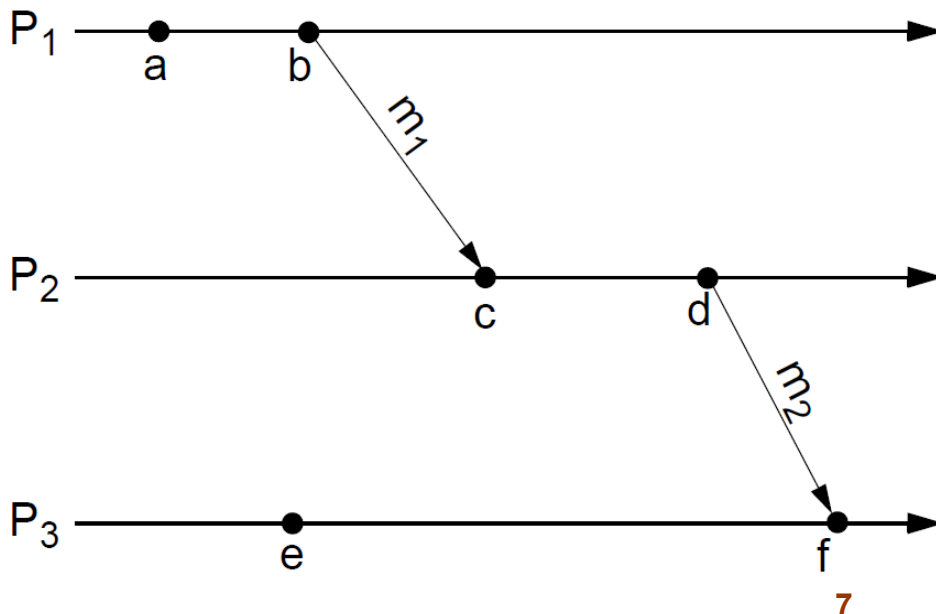
- The order of events occurring at different processes is critical for many distributed applications.
  - Example: `P.o_created` and `P.c_created` in *make-program* example.
- **Ordering** can be based on two simple situations:
  1. If two events occurred in the *same* process, then they occurred in the order observed following the respective process;
  2. Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving it.
- Ordering by Lamport is based on **happened-before relation** (denoted " $\rightarrow$ "):
  - $a \rightarrow b$ , if  $a$  and  $b$  are events in the *same* process and  $a$  occurred before  $b$ ;
  - $a \rightarrow b$ , if  $a$  is the event of sending a message  $m$  in a process, and  $b$  is the event of the same message  $m$  being received by another process.

If  $a \rightarrow b$  and  $b \rightarrow c$ , then  $a \rightarrow c$  (the relation  $\rightarrow$  is **transitive**).

# Lamport's Logical Clocks

- If  $a \rightarrow b$ , we say that event  $a$  **causally affects** event  $b$ .
  - The two events are **causally related**.
- There are events which are not related by the happened-before relation.
  - If both  $a \rightarrow e$  and  $e \rightarrow a$  are false, then  $a$  and  $e$  are **concurrent** events:
    - ▶ we write  $a \parallel e$ .

## Example:



$a \rightarrow b$ ,  $c \rightarrow d$ ,  $e \rightarrow f$ ,  
 $b \rightarrow c$ ,  $d \rightarrow f$

$a \rightarrow c$ ,  $a \rightarrow d$ ,  $a \rightarrow f$ ,  
 $b \rightarrow d$ ,  $b \rightarrow f$ , ...

$a \parallel e$ ,  $c \parallel e$ , ...

# Lamport's Logical Clocks

Using physical clocks, the happened-before relation cannot be captured.

- It is possible that  $b \rightarrow c$  and at the same time  $T_b > T_c$  (where  $T_b$  is the physical time of event  $b$ ).

Logical clocks can be used to capture the happened-before relation.

- **A logical clock is a monotonically increasing software counter.**
- There is a logical clock  $C_{P_i}$  at each process  $P_i$  in the system.
- The value of the logical clock is used to assign **timestamps** to events.
  - $C_{P_i}(a)$  is the timestamp of event  $a$  in process  $P_i$ .
- There is no relationship between a logical clock and any physical clock.

To capture the happened-before relation, logical clocks have to be implemented so that:

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



# Lamport's Logical Clocks

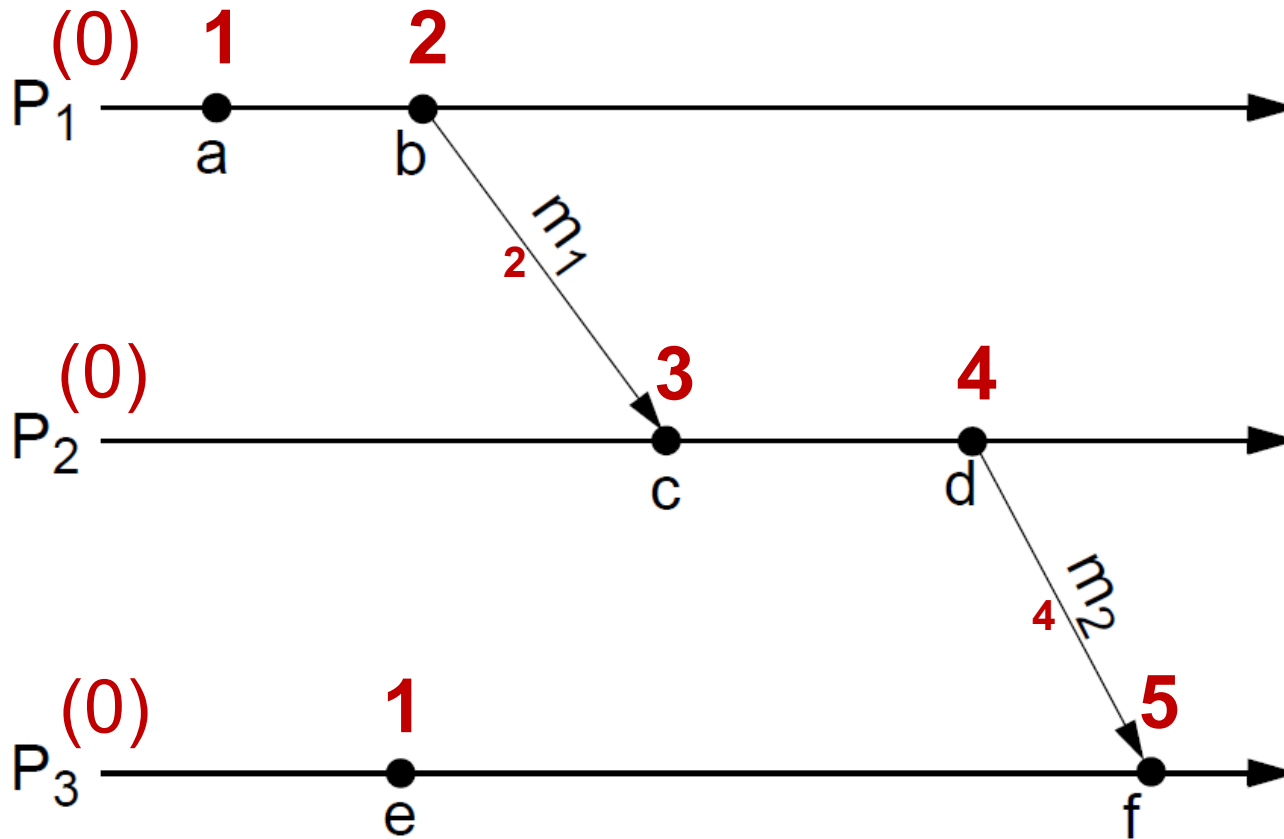
Implementation of logical clocks is performed using the following **rules** for updating the clocks and transmitting their values in messages:

- [R1]: Each event issued at process  $P_i$  is timestamped with the value obtained after incrementing the local clock  $C_{P_i}$ :  $C_{P_i} := C_{P_i} + 1$ .
- [R2]: a) **If**  $a$  is the event of sending a message  $m$  from process  $P_i$ ,  
**then** the timestamp  $t_m = C_{P_i}(a)$  is included in  $m$   
 ( $C_{P_i}(a)$  is the logical clock value obtained after applying rule R1).
- b) On **receiving** message  $m$  by process  $P_j$ ,  
 its logical clock  $C_{P_j}$  is updated as follows:  
 $C_{P_j} := \max ( C_{P_j}, t_m )$ .
- c) The new value of  $C_{P_j}$  is used to timestamp the event of receiving message  $m$  by  $P_j$  (applying rule R1).

- If  $a$  and  $b$  are events in the *same* process and  $a$  occurred before  $b$ , then  $a \rightarrow b$ , and (by R1)  $C(a) < C(b)$ .
- If  $a$  is the event of sending a message  $m$  in a process, and  $b$  is the event of the same message  $m$  being received by another process, then  $a \rightarrow b$ , and (by R2)  $C(a) < C(b)$ .
- If  $a \rightarrow b$  and  $b \rightarrow c$ , then  $a \rightarrow c$ , and (by induction)  $C(a) < C(c)$ .

# Lamport's Logical Clocks

## Example



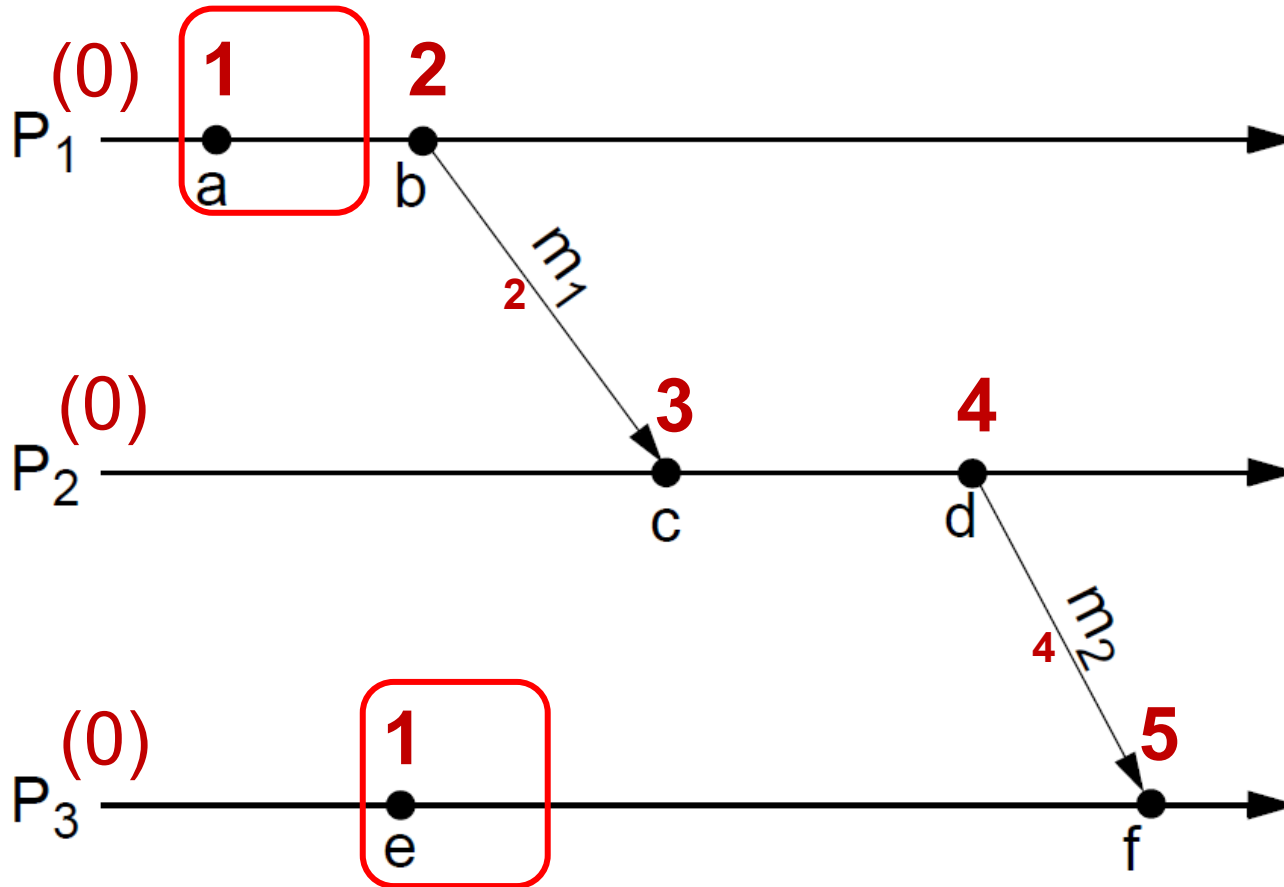
# Problems with Lamport's Logical Clocks (1)

- Lamport's logical clocks impose only a **partial order** on the set of events
  - pairs of distinct events generated by different processes can have identical timestamp.
- For certain applications a **total ordering** is needed; they consider that no two events can occur at the same time.
- In order to enforce total ordering, a **global logical timestamp** is introduced:
  - The global logical timestamp of an event  $a$  occurring at process  $P_i$ , with logical timestamp  $C_{P_i}(a)$ , is a **pair**  $(C_{P_i}(a), i)$ , where  $i$  is an identifier of process  $P_i$
  - We define
$$(C_{P_i}(a), i) \prec (C_{P_j}(b), j) \text{ if and only if } C_{P_i}(a) < C_{P_j}(b), \\ \text{or } C_{P_i}(a) = C_{P_j}(b) \text{ and } i < j.$$

= lexical order on pairs

# Lamport's Logical Clocks

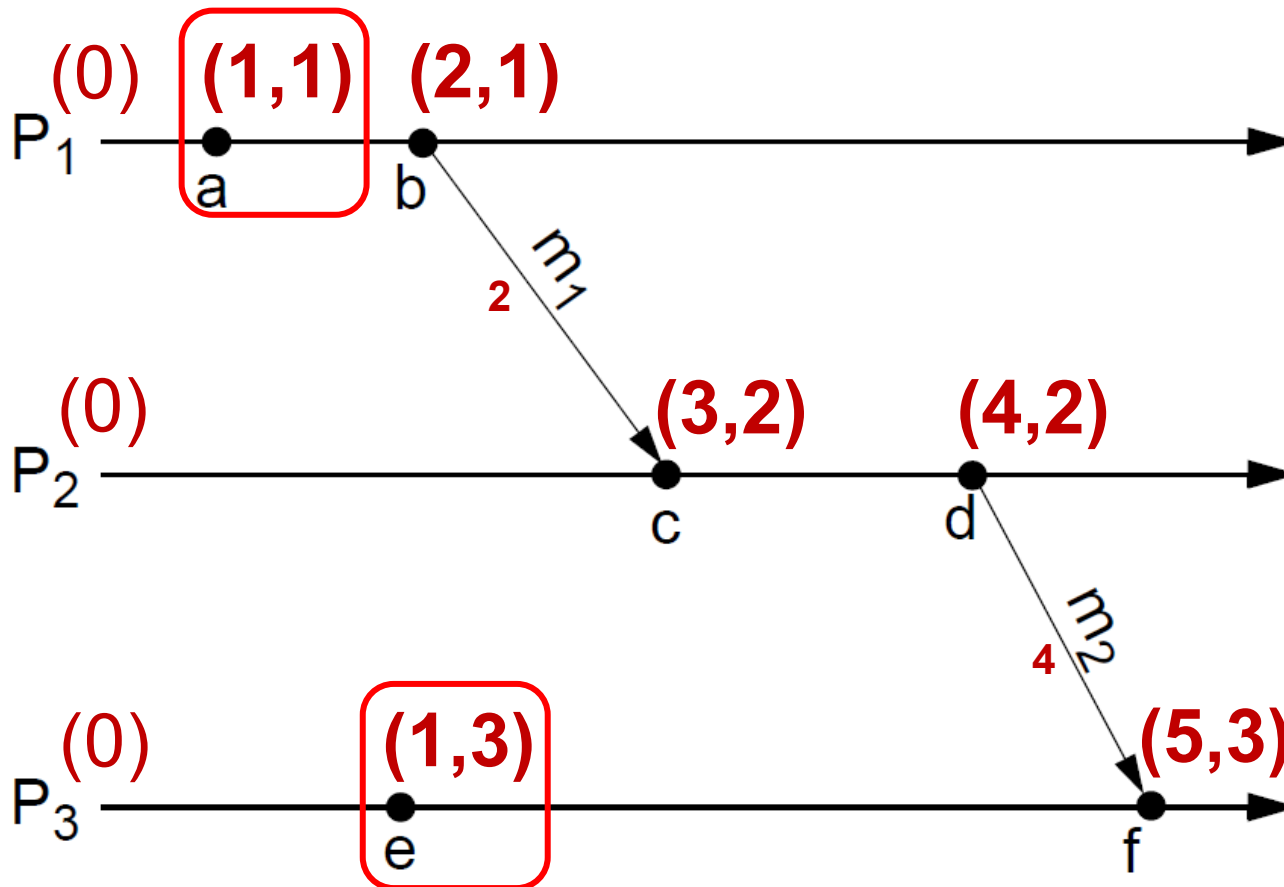
## Example



# Lamport's Logical Clocks

## with Global Logical Timestamps

### Example



# Problems with Lamport's Logical Clocks (2)

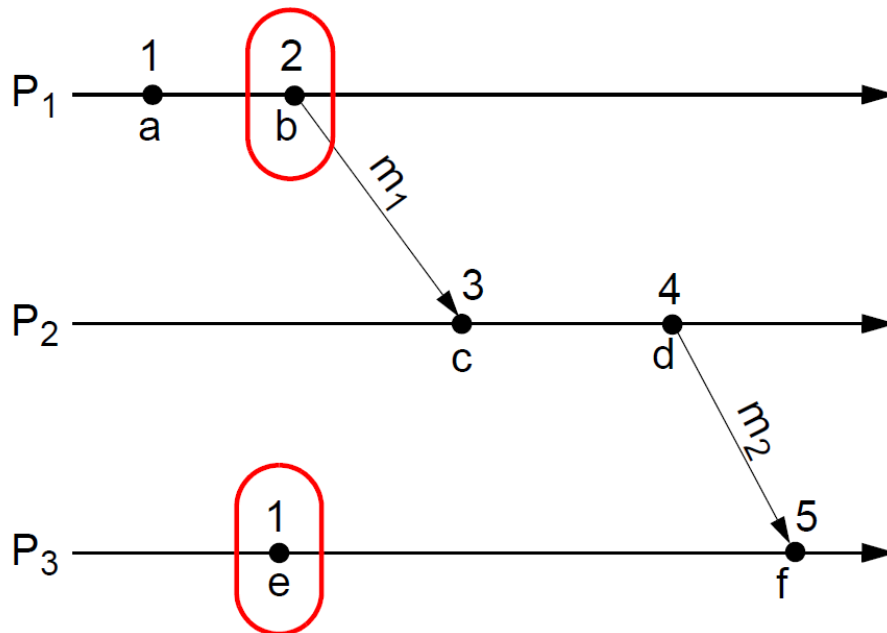
- Lamport's logical clocks are **not** powerful enough to perform a **causal ordering of events**.

We have seen earlier:

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

However, the reverse is not always true:

- if  $C(a) < C(b)$ , then  $a \rightarrow b$  is **not necessarily true**.  
(it is only guaranteed that  $b \rightarrow a$  is not true).

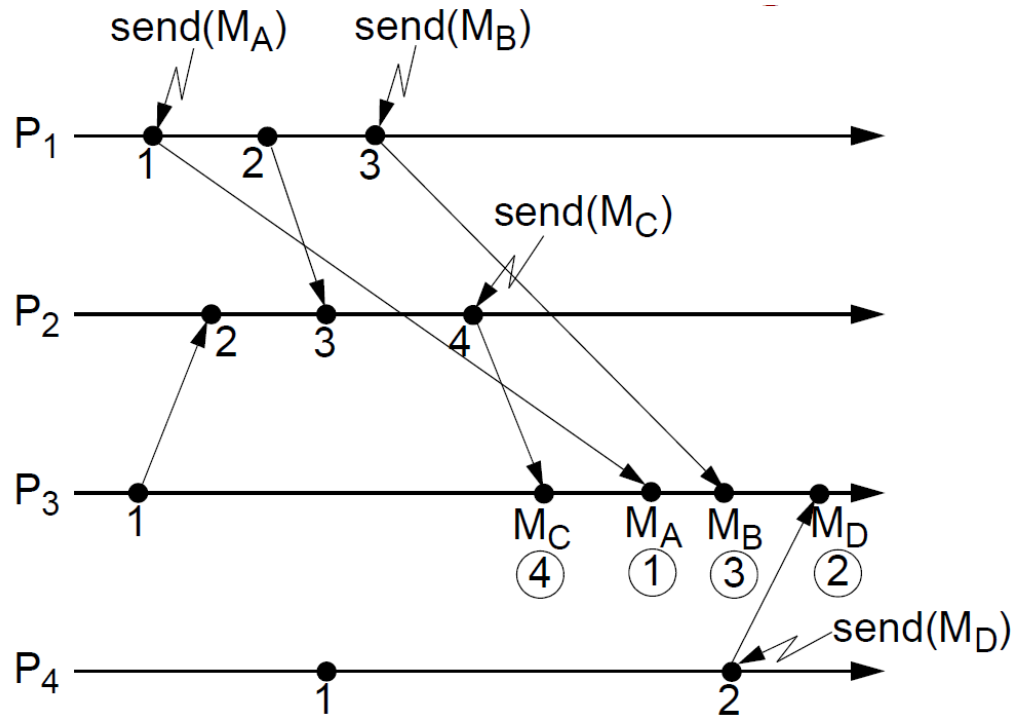


$C(e) < C(b)$ ,  
however there is no causal  
relation from event  $e$  to event  $b$ .

By just looking at the timestamps  
of the events, we cannot say  
whether two events are causally  
related or not.

If  $C(x) < C(y)$ ,  
it might be that  $x \rightarrow y$  or  $x \parallel y$

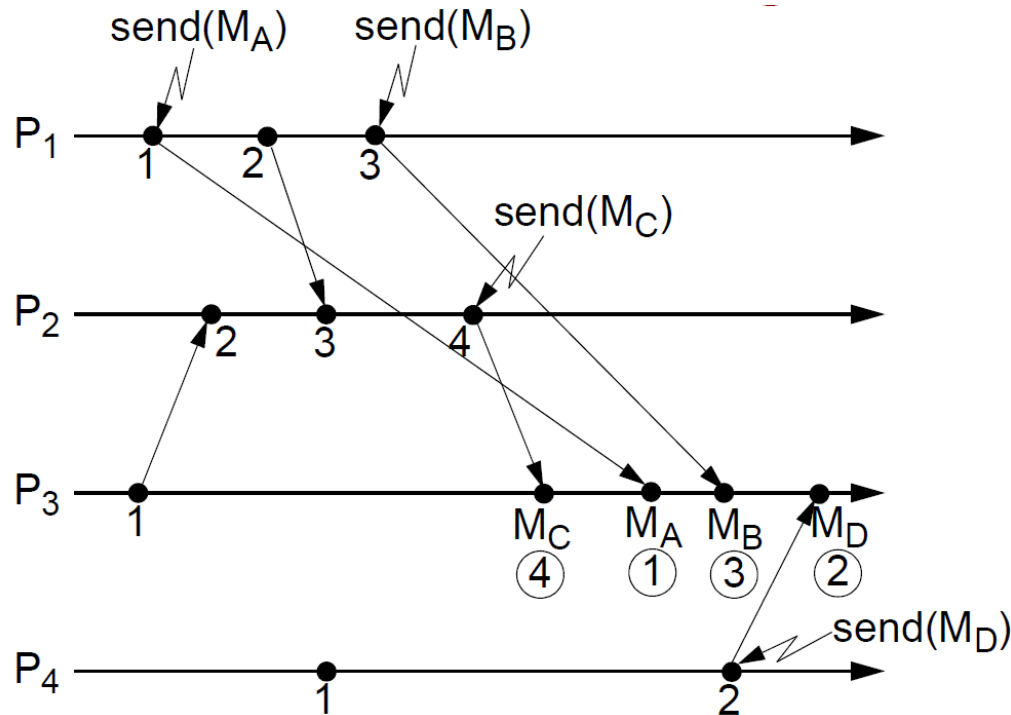
# Problems with Lamport's Logical Clocks



We want messages received by  $P_3$  to be processed in their *causal* order.  
Can we use the associated timestamp for this purpose?

- Process  $P_3$  receives messages  $M_A$ ,  $M_B$ ,  $M_C$ , and  $M_D$ .
  - $\text{send}(M_A) \rightarrow \text{send}(M_C)$ ,  $\text{send}(M_A) \rightarrow \text{send}(M_B)$ ,
  - $\text{send}(M_B) \parallel \text{send}(M_C)$ ,  $\text{send}(M_A) \parallel \text{send}(M_D)$ ,
  - $\text{send}(M_B) \parallel \text{send}(M_D)$ ,  $\text{send}(M_C) \parallel \text{send}(M_D)$ .

# Problems with Lamport's Logical Clocks



$\text{send}(M_A) \rightarrow \text{send}(M_C)$ ,  $\text{send}(M_A) \rightarrow \text{send}(M_B)$   
 $\rightarrow$  process  $M_A$  *before*  $M_C$  and  $M_B$

But,  $P_3$  needs not wait for  $M_B$  and  $M_D$  in order to process them before  $M_C$ ; similarly, the delivery of  $M_B$  is not needed to be delayed after that of  $M_D$ .



By processing the messages in order of their *timestamp*, all happened-before relations will be correctly enforced, **but** additional, unneeded, delays will be introduced (due to enforcement of ordering where, in fact, not needed).

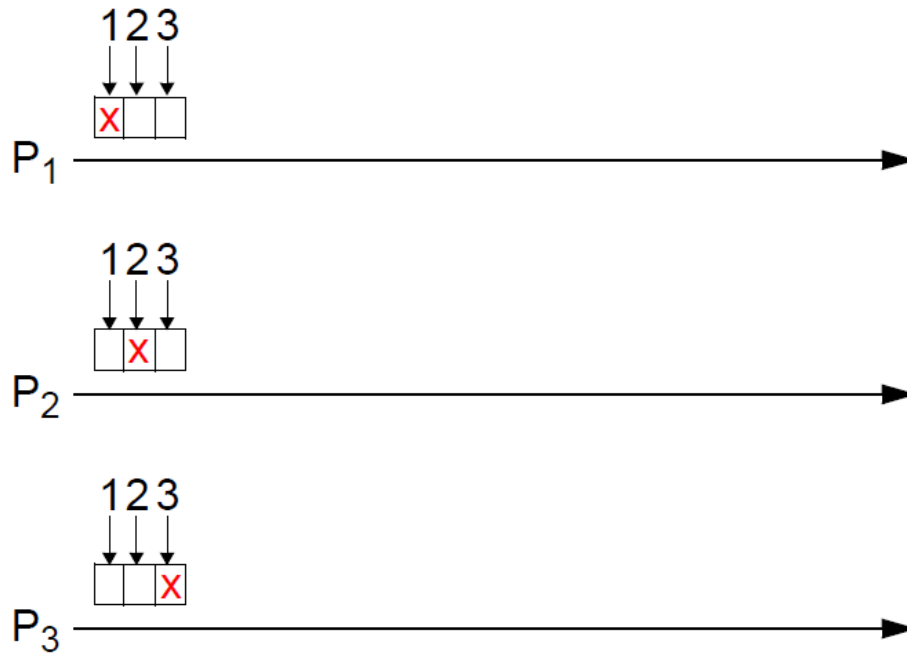


# Vector Clocks

- Vector clocks give the ability to decide whether two events are **causally related** or not by simply looking at their timestamp.
- Each **process**  $P_i$  has a **clock**  $C_{P_i}^v$ 
  - $C_{P_i}^v$  is an **integer vector** of length  $n$ 
    - ▶  $n$  is the number of processes
  - The value of  $C_{P_i}^v$  is used to assign timestamps to events in process  $P_i$ .
    - ▶  $C_{P_i}^v(a)$  is the **timestamp** of event  $a$  in process  $P_i$ .
  - $C_{P_i}^v[i]$ , the  $i$ th entry of  $C_{P_i}^v$ , corresponds to an event counter in  $P_i$ 
    - ▶ simply, counts the events in  $P_i$
- $C_{P_i}^v[j]$ , for  $j \neq i$ , is  $P_i$ 's "best guess" of the local event counter at  $P_j$ 
  - $C_{P_i}^v[j]$  indicates the value of the local event counter of  $P_j$  at the occurrence of the **last** event at  $P_j$  which is in a **happened-before** relation to the current event at  $P_i$ .

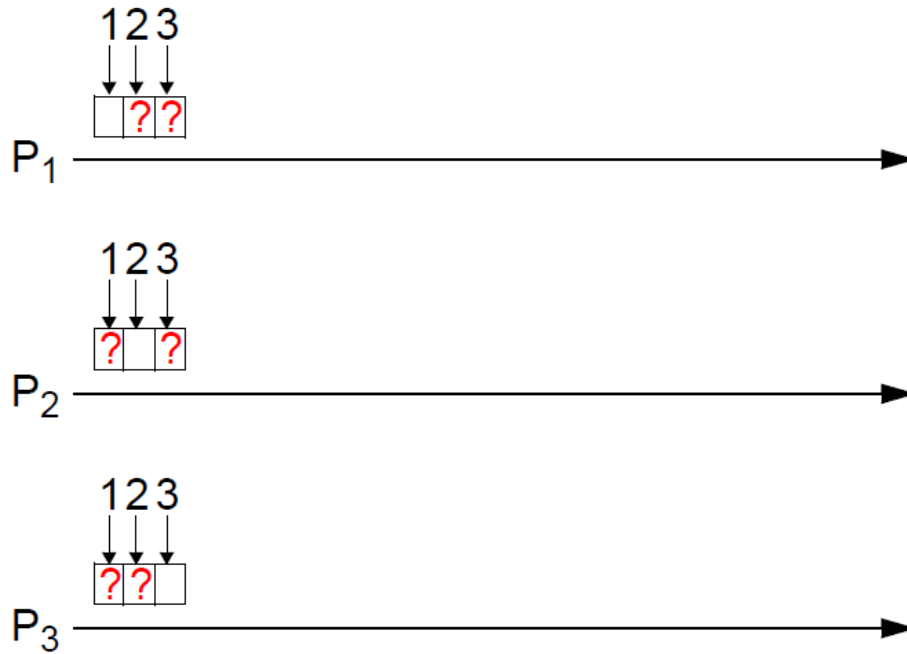
# Vector Clocks

## Example ( $n=3$ )



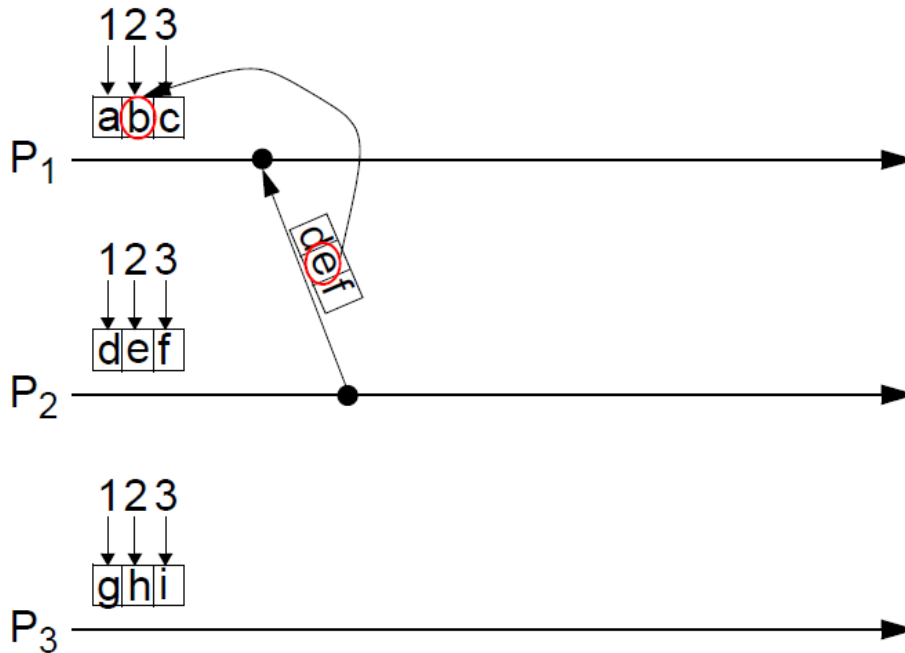
- $C_{P_i}^V[l]$ , the  $i$ th entry of  $C_{P_i}^V$ , corresponds to an event counter in  $P_i$  (simply, counts the events in  $P_i$ ).

# Vector Clocks



- $C_{P_i}^V[l]$ , the  $i$ th entry of  $C_{P_i}^V$ , corresponds to an event counter in  $P_i$  (simply, counts the events in  $P_i$ ).
- What about all the other entries?

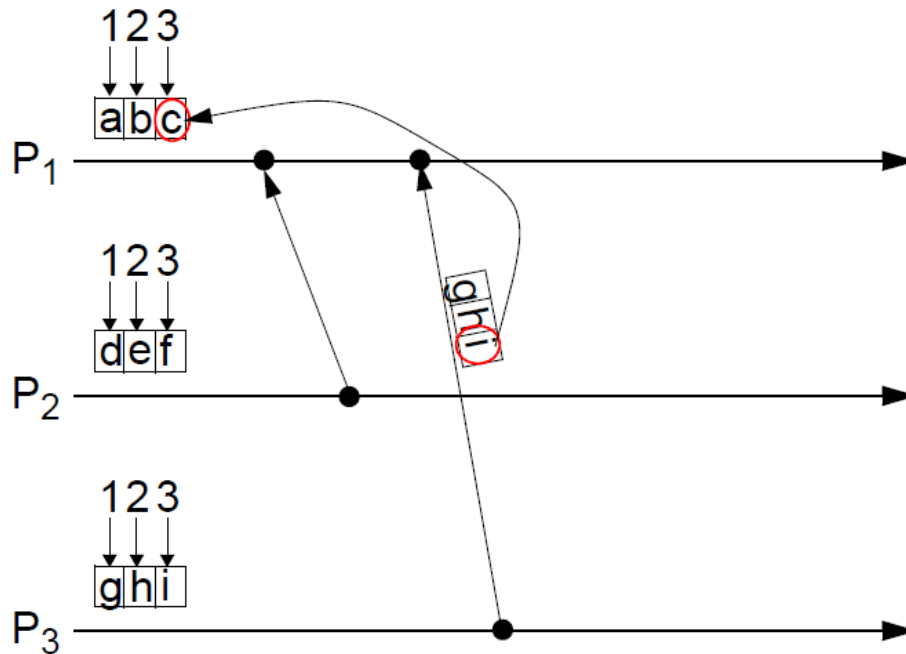
# Vector Clocks



- $C_{P_i}^V[j]$ , the  $i$ th entry of  $C_{P_i}^V$ , corresponds to an event counter in  $P_j$  (simply, counts the events in  $P_j$ ).
- $C_{P_i}^V[j]$ ,  $j \neq i$ , is  $P_i$ 's best guess of the local event counter at  $P_j$ .

*Best guess of  $P_i$ : the value of the local counter in  $P_j$ , at the moment of the most recent interaction from  $P_j$  to  $P_i$ .*

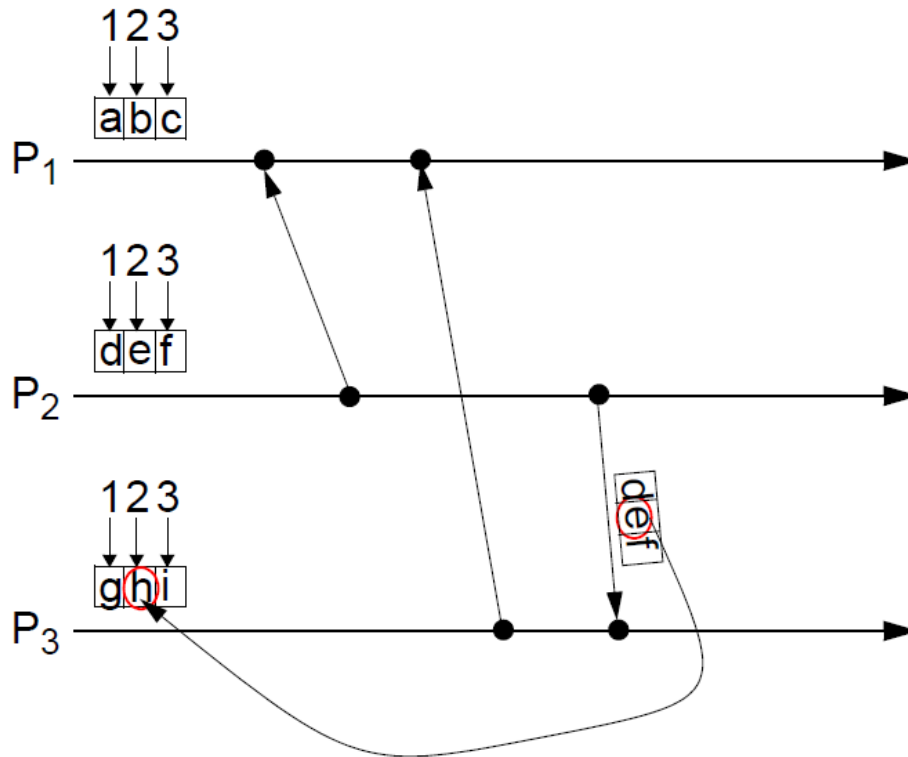
# Vector Clocks



- $C_{P_i}^V[l]$ , the  $i$ th entry of  $C_{P_i}^V$ , corresponds to an event counter in  $P_i$  (simply, counts the events in  $P_i$ ).
- $C_{P_i}^V[j]$ ,  $j \neq i$ , is  $P_i$ 's best guess of the local event counter at  $P_j$ .

*Best guess of  $P_i$* : the value of the local counter in  $P_j$ , at the moment of the most recent interaction from  $P_j$  to  $P_i$ .

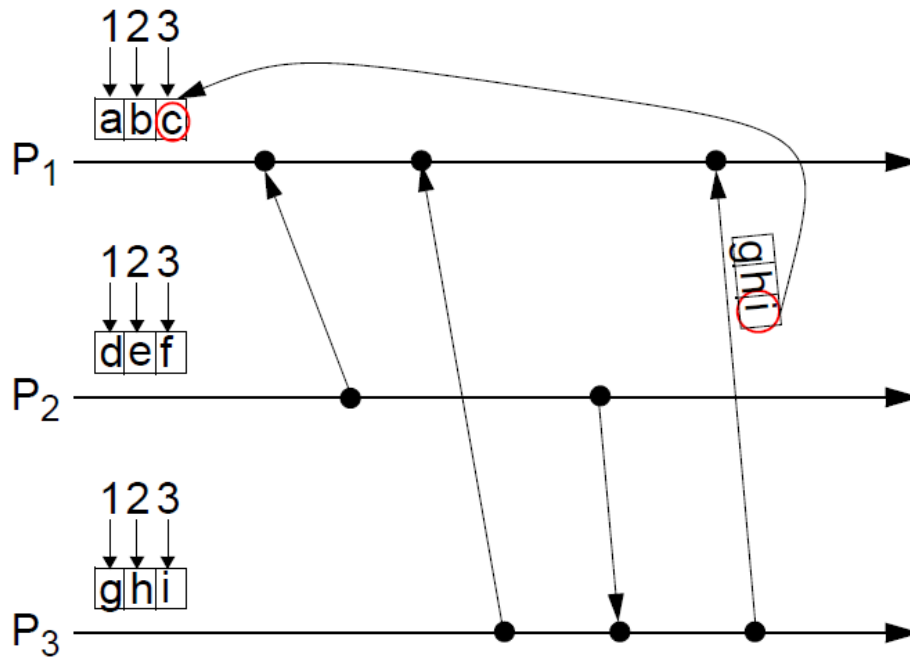
# Vector Clocks



- $C_{P_i}^V[j]$ , the  $i$ th entry of  $C_{P_i}^V$ , corresponds to an event counter in  $P_i$  (simply, counts the events in  $P_i$ ).
- $C_{P_i}^V[j]$ ,  $j \neq i$ , is  $P_i$ 's best guess of the local event counter at  $P_j$ .

*Best guess of  $P_i$* : the value of the local counter in  $P_j$ , at the moment of the most recent interaction from  $P_j$  to  $P_i$ .

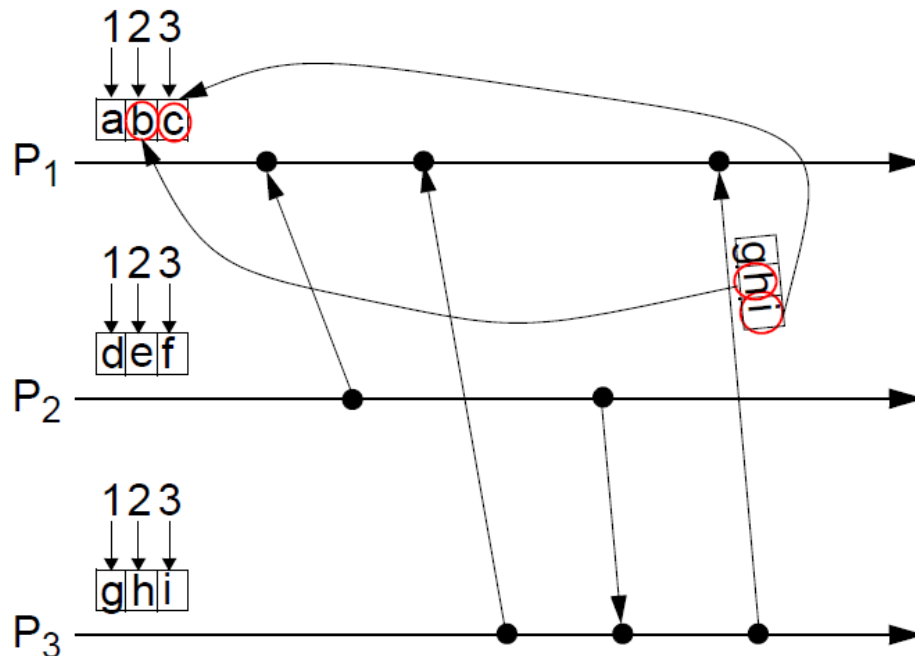
# Vector Clocks



- $C_{P_i}^V[l]$ , the  $i$ th entry of  $C_{P_i}^V$ , corresponds to an event counter in  $P_i$  (simply, counts the events in  $P_i$ ).
- $C_{P_i}^V[j]$ ,  $j \neq i$ , is  $P_i$ 's best guess of the local event counter at  $P_j$ .

*Best guess of  $P_i$ :* the value of the local counter in  $P_j$ , at the moment of the most recent interaction from  $P_j$  to  $P_i$ .

# Vector Clocks



- $C_{P_i}^V[l]$ , the  $i$ th entry of  $C_{P_i}^V$ , corresponds to an event counter in  $P_i$  (simply, counts the events in  $P_i$ ).
- $C_{P_i}^V[j]$ ,  $j \neq i$ , is  $P_i$ 's best guess of the local event counter at  $P_j$ .

*Best guess of  $P_i$ :* the value of the local counter in  $P_j$ , at the moment of the most recent **direct or indirect interaction** from  $P_j$  to  $P_i$ .  $\Rightarrow C_{P_i}^V[j]$  indicates the value of the local event counter of  $P_j$  at the occurrence of the last event at  $P_j$  which is in a *happened-before* relation to the current event at  $P_i$ .

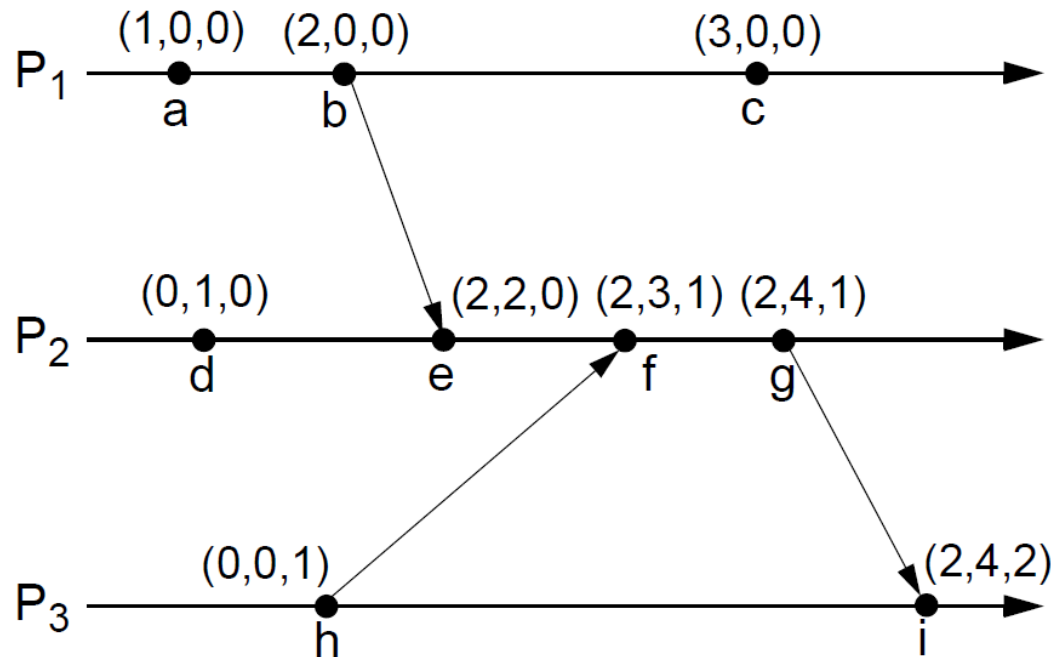


# Vector Clocks

Implementation of vector clocks is performed using the following **rules** for updating the clocks and transmitting their values in messages:

- [R1]: Each event issued at process  $P_i$  is timestamped with the value of the vector clock  $C^v_{P_i}$  obtained after incrementing the corresponding element  $C^v_{P_i}[i]$ :  $C^v_{P_i}[i] := C^v_{P_i}[i] + 1$ .
- [R2]: (a) If  $a$  is the event of **sending** a message  $m$  from process  $P_i$ , then the timestamp  $t_m = C^v_{P_i}(a)$  is included in  $m$  ( $C^v_{P_i}(a)$  is the vector clock value obtained after applying rule R1).
- (b) On **receiving** message  $m$  by process  $P_j$ , its vector clock  $C^v_{P_j}$  is updated as follows:
- $$\forall k \text{ in } \{ 1, 2, \dots, n \}, \quad C^v_{P_j}[k] := \mathbf{max} ( C^v_{P_j}[k], t_m[k] )$$
- (c) The new value of  $C^v_{P_j}$  is used to timestamp the event of receiving message  $m$  by  $P_j$  (applying rule R1).

# Vector Clocks



For any two vector timestamps  $u$  and  $v$ , we have:

- $u = v$  if and only if  $\forall i, u[i] = v[i]$
- $u \leq v$  if and only if  $\forall i, u[i] \leq v[i]$
- $u < v$  if and only if  $u \leq v \wedge u \neq v$
- $u \parallel v$  if and only if  $\neg(u < v) \wedge \neg(v < u)$

Two events  $a$  and  $b$  are **causally related** if and only if  $C^v(a) < C^v(b) \vee C^v(b) < C^v(a)$ .

Otherwise, the events are **concurrent**.

# Vector Clocks

Vector clocks have the property which we missed for Lamport's clocks:

- $a \rightarrow b$  if and only if  $C^v(a) < C^v(b)$ .

Thus, by just looking at the timestamps of the events, we can tell whether two events are causally related or not.

→ **Vector clocks can be used for causal ordering of events/messages.**

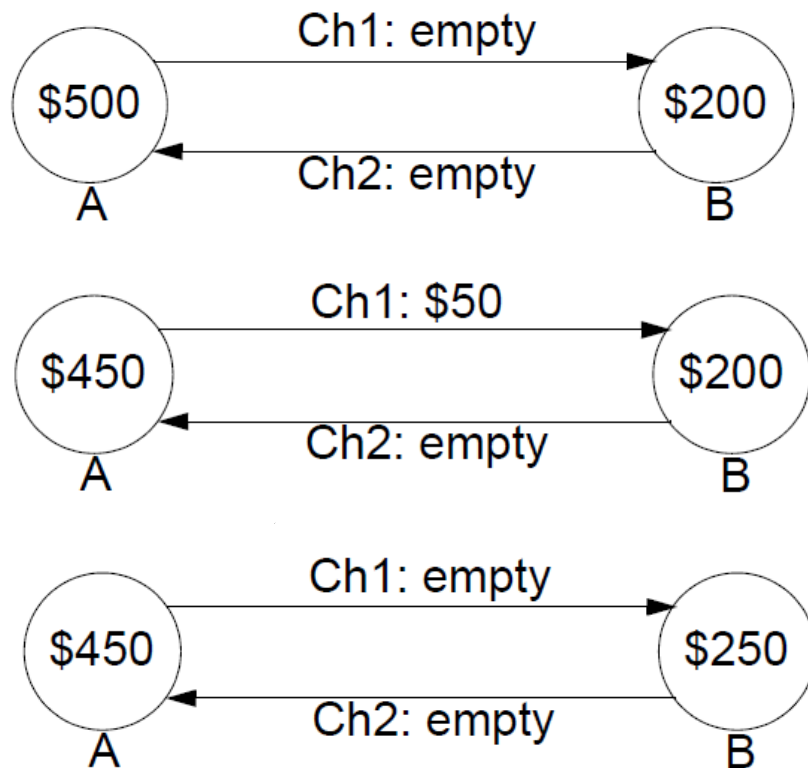
# Global States

The problem is how to **collect and record a consistent global state in a distributed system.**

- “State” is application-specific
- Example use cases:
  - Monitoring of a distributed shared data structure
    - ▶ Bank account example →
  - Distributed garbage collection
  - Progress monitoring for dynamic load balancing
  - Distributed deadlock detection
- Why a problem?
  - Because there is no global clock (no coherent notion of time) and no shared memory!

# Global States

Consider a bank system with two accounts A and B at two different sites; we transfer \$50 between A and B.



A	Ch1	B	C : consistent NC: not consistent
500	empty	200	?
500	50	200	NC
450	50	200	C
450	empty	200	NC
500	50	250	NC
450	50	250	NC
450	empty	250	C
500	empty	250	NC

# Global States

- In general, a **global state** consists of
  - a set of local states and
  - a set of states of the communication channels.
- The state of a communication channel in a **consistent global state** should be the sequence of messages sent along the channel before the sender's state was recorded, excluding the sequence of messages received along the channel before the receiver's state was recorded.
- It is difficult to record channel states to ensure the above rule
  - global states are very often recorded *without* using channel states.
  - ▶ This is the case in the definition below.

# Formal Definition (1)

- $LS_i$  is the **local state** of process  $P_i$ .
  - Beside other information, the local state also includes a record of all messages sent and received by the process.
- We consider the **global state**  $GS$  of a system as the collection of the local states of its processes:
$$GS = ( LS_1, LS_2, \dots, LS_n ).$$
  - A certain global state can be *consistent* or not!

## Formal Definition (2)

**send**( $m_{ij}^k$ ) denotes the event of sending message  $m_{ij}^k$  from  $P_i$  to  $P_j$

**rec**( $m_{ij}^k$ ) denotes the event of receiving message  $m_{ij}^k$  by  $P_j$ .

- $send(m_{ij}^k) \in LS_i$  if and only if the sending event occurred *before* the local state was recorded;
- $rec(m_{ij}^k) \in LS_j$  if and only if the receiving event occurred *before* the local state was recorded.

$$\mathbf{transit}(LS_i, LS_j) = \{ m_{ij}^k \mid send(m_{ij}^k) \in LS_i \wedge rec(m_{ij}^k) \notin LS_j \}$$

$$\mathbf{inconsistent}(LS_i, LS_j) = \{ m_{ij}^k \mid send(m_{ij}^k) \notin LS_i \wedge rec(m_{ij}^k) \in LS_j \}$$

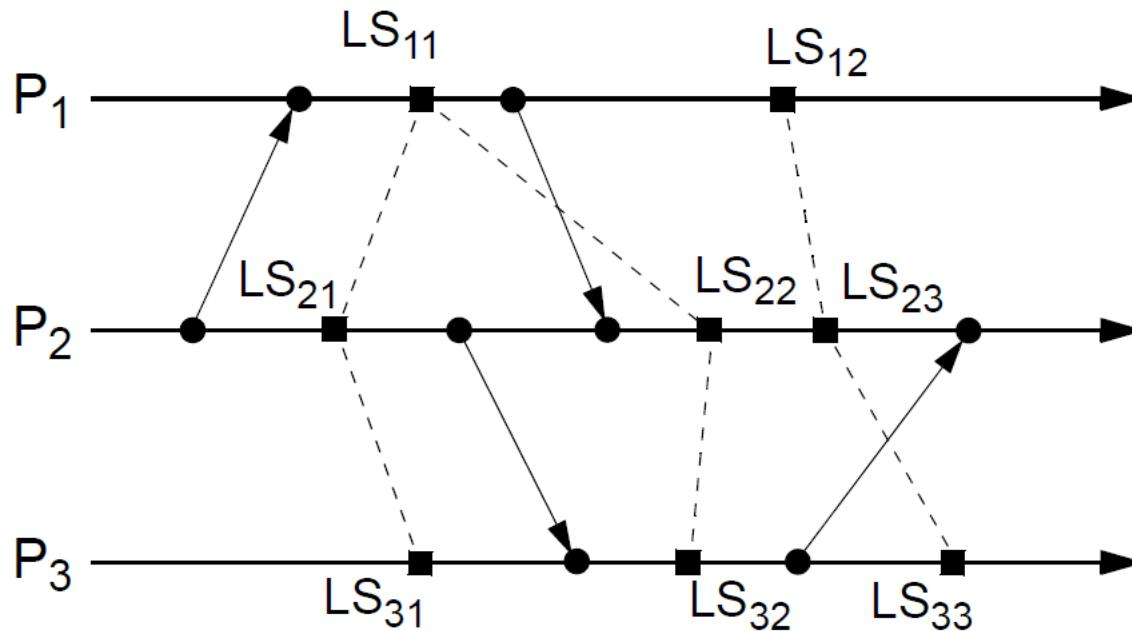


## Formal Definition (3)

- A global state  $GS = (LS_1, LS_2, \dots, LS_n)$  is **consistent** if and only if:  
 $\forall i, \forall j: 1 \leq i, j \leq n :: inconsistent(LS_i, LS_j) = \emptyset$ 
  - In a consistent global state, for every received message a corresponding send event is recorded in the global state.
  - In an inconsistent global state, there is at least one message whose receive event is recorded but its send event is not recorded.
- A global state  $GS = (LS_1, LS_2, \dots, LS_n)$  is **transitless** if and only if:  
 $\forall i, \forall j: 1 \leq i, j \leq n :: transit(LS_i, LS_j) = \emptyset$ 
  - All messages recorded to be sent are also recorded to be received.
- A global state is **strongly consistent** if it is consistent and transitless.
  - A strongly consistent state corresponds to a consistent state in which all messages recorded as sent are also recorded as received.

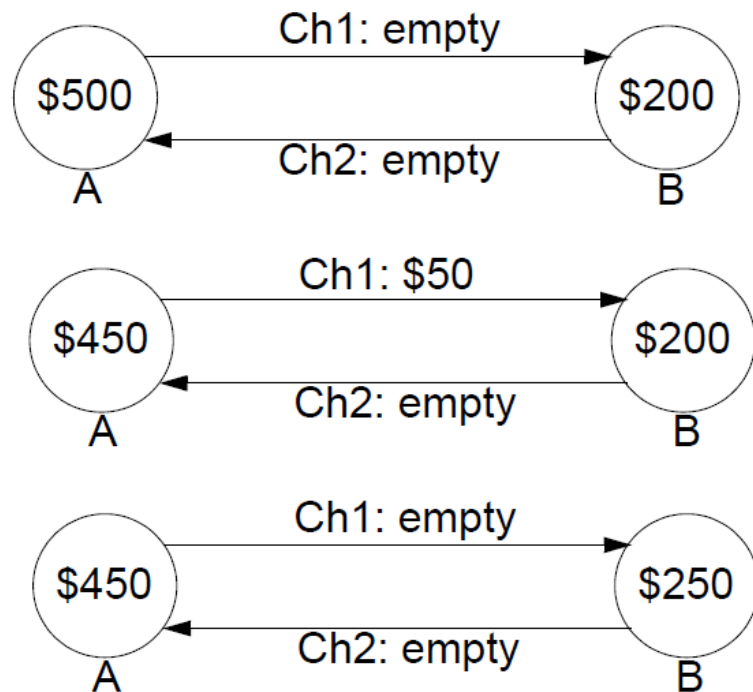
Note: the global state, as defined here, is seen as a collection of the local states, without explicitly capturing the state of the channel.

# Example



- $(LS_{11}, LS_{22}, LS_{32})$  is inconsistent
- $(LS_{12}, LS_{23}, LS_{33})$  is consistent
- $(LS_{11}, LS_{21}, LS_{31})$  is strongly consistent

## Example (2)



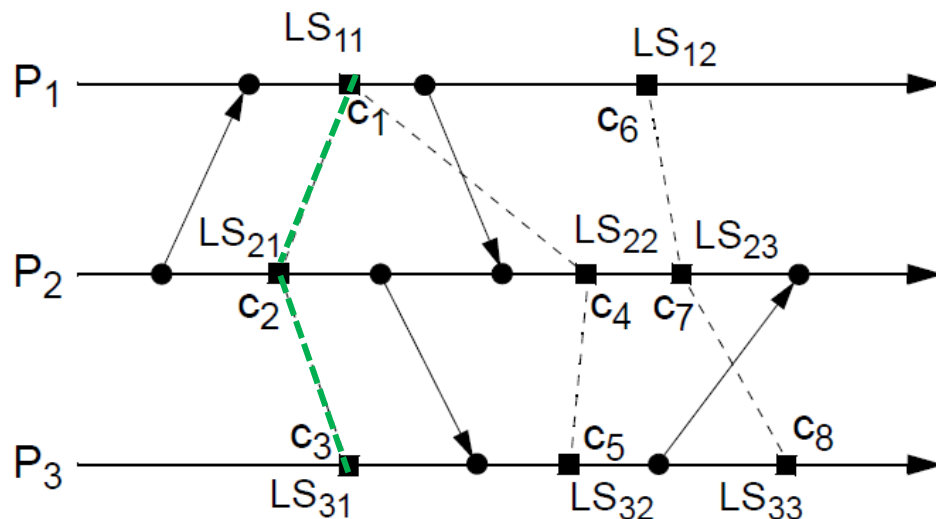
A	B	C : consistent NC: not consistent
500	200	{A,B}: strongly C
450 (mess <sub>1</sub> sent)	200	{A,B}: C
500	250 (mess <sub>1</sub> received)	{A,B}: NC
450 (mess <sub>1</sub> sent)	250 (mess <sub>1</sub> received)	{A,B}: strongly C

After registering of the receive event(s),  
a consistent state becomes strongly consistent.

It is considered to be a normal (transient) situation.

# Cuts of a Distributed Computation

- A **cut** is a graphical representation of a **global state**.
  - A consistent cut is a graphical representation of a consistent global state.
- A **cut** of a distributed computation is a set
 
$$Ct = \{ c_1, c_2, \dots, c_n \}, \text{ where } c_i \text{ is the cut event at process } P_i.$$
- A **cut event** is the event of recording a local state of the process.

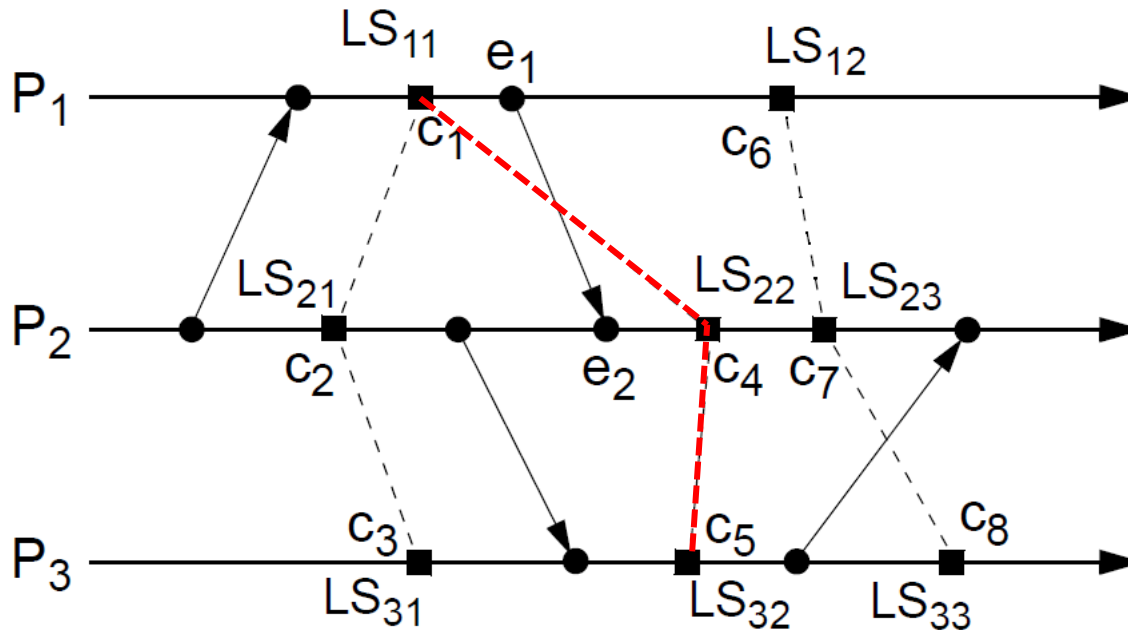


Example:

$\{ c_1, c_2, c_3 \}$  is a cut

# Cuts of a Distributed Computation

- Let  $e_k$  denote an event at process  $P_k$ .
- A cut  $Ct = \{c_1, c_2, \dots, c_n\}$  is a **consistent cut** if and only if
  - $\forall P_i, \forall P_j$ , if  $\exists e_i, \exists e_j$  such that  $(e_i \rightarrow e_j) \wedge (e_j \rightarrow c_j)$  then  $\neg(c_i \rightarrow e_i)$
- A cut is *consistent* if every message that was *received before* a cut event was *sent before* the cut event at the sender process.



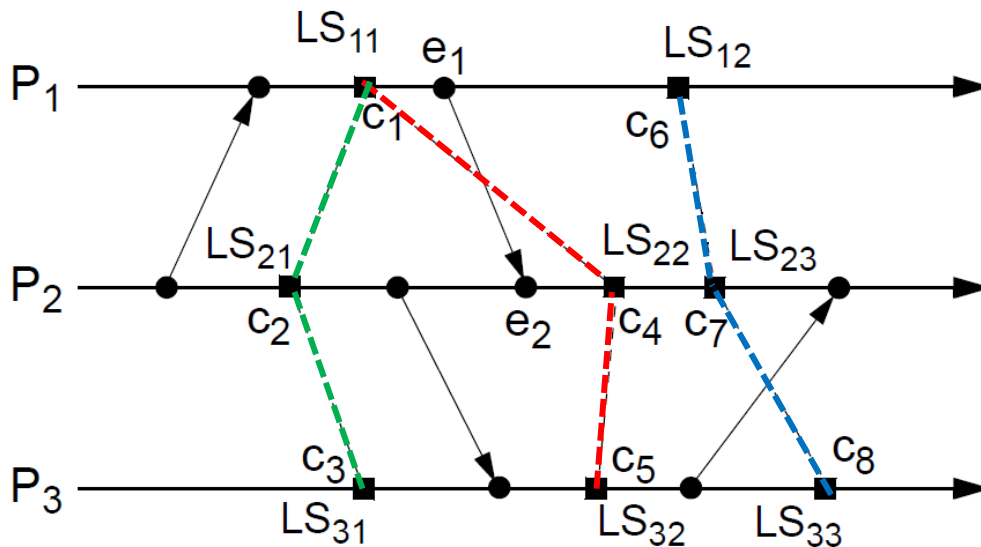
$\{c_1, c_4, c_5\}$  is not consistent, as  $(e_1 \rightarrow e_2) \wedge (e_2 \rightarrow c_4) \wedge (c_1 \rightarrow e_1)$

# Cuts of a Distributed Computation

## Theorem

**A cut  $Ct = \{ c_1, c_2, \dots, c_n \}$  is a consistent cut if and only if no two cut events are causally related, that is,  $\forall c_i, \forall c_j \in Ct: \neg(c_i \rightarrow c_j) \wedge \neg(c_j \rightarrow c_i)$**

- **A set of concurrent cut events forms a consistent cut.**



- $\{ c_1, c_2, c_3 \}$  strongly consistent (no communication line is crossed)
- $\{ c_6, c_7, c_8 \}$  consistent (comm. line crossed, but no causal relation)
- $\{ c_1, c_4, c_5 \}$  not consistent ( $c_1 \rightarrow c_4$ )

# Global State Recording

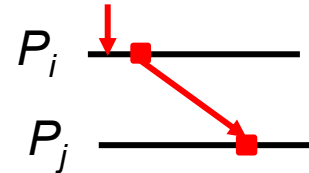
## (Chandy-Lamport Algorithm)

- The algorithm records:
  - a collection of local states, which give a consistent global state of the system, and
  - the state of the channels, which is consistent with the collected global state.
- Such a recorded "view" of the system is called a **snapshot**.
- We assume here that
  - processes are connected through one-directional channels and message delivery is FIFO.
  - the graph of processes and channels is strongly connected (there exists a path between any two processes).
- The algorithm is based on the use of a special message, the **snapshot token**, in order to control the state collection process.

# Global State Recording

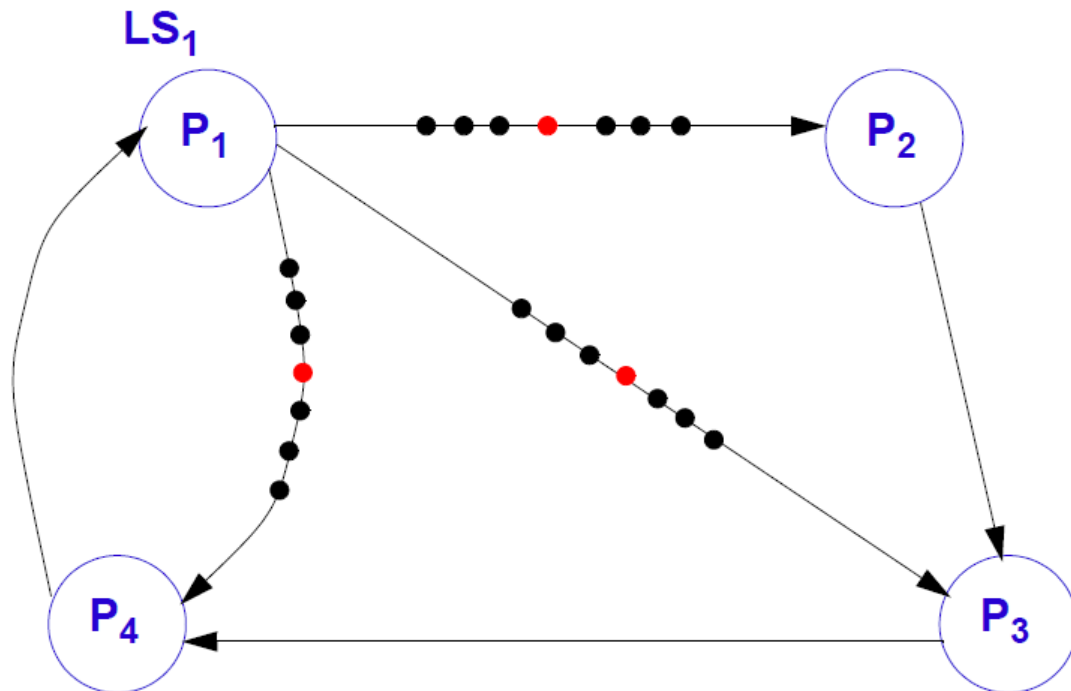
How to collect a global state?

- A process  $P_i$  records its **local state**  $LS_i$  and later sends a message  $m$  to  $P_j$ 
  - $LS_j$  at  $P_j$  has to be recorded *before*  $P_j$  has received  $m$ .
- The **channel state**  $Sch_{ij}$  of the channel  $Ch_{ij}$  consists of all messages that process  $P_i$  sent *before* recording  $LS_i$  and which have not been *received* by  $P_j$  when recording  $LS_j$ .
- A **snapshot** is started at the request of a particular process  $P_i$ , for example, when  $P_i$  suspects a deadlock because of long delay in accessing a resource.  
 $P_i$  then records its state  $LS_i$  and, before sending any other message, it **sends a token** to every  $P_j$  that  $P_i$  communicates with.
- When  $P_j$  **receives a token** from  $P_i$ , and this is the first time it received a token, it must record its state before it receives the next message from  $P_i$ .  
 After recording its state,  $P_j$  sends a token to every process it communicates with, before sending them any other message.



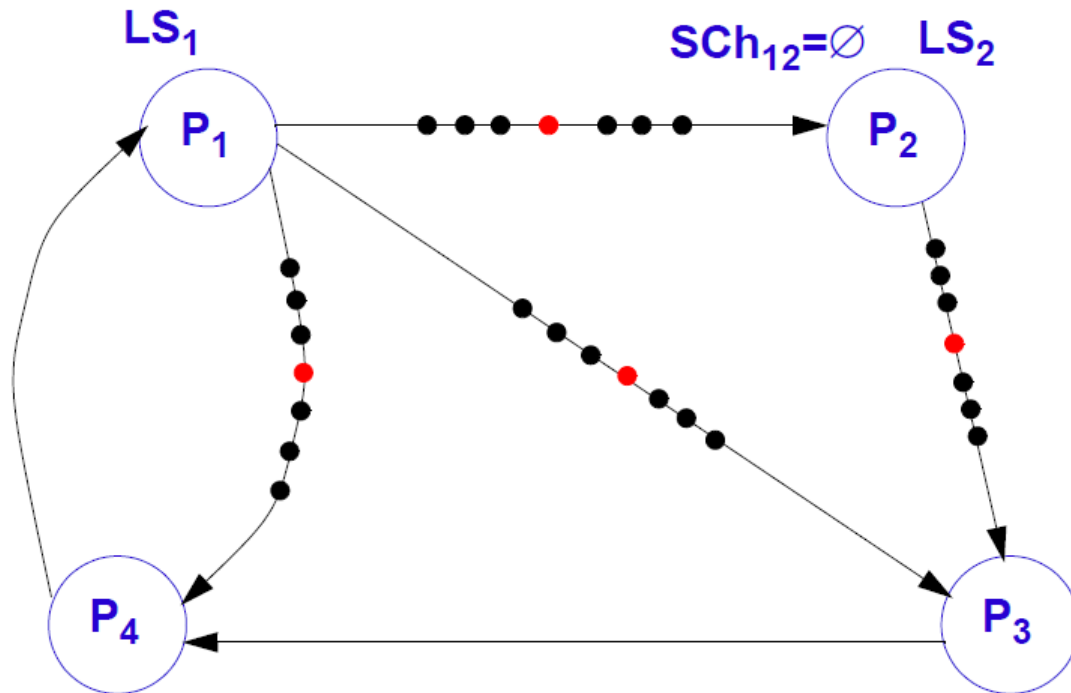


# Global State Recording



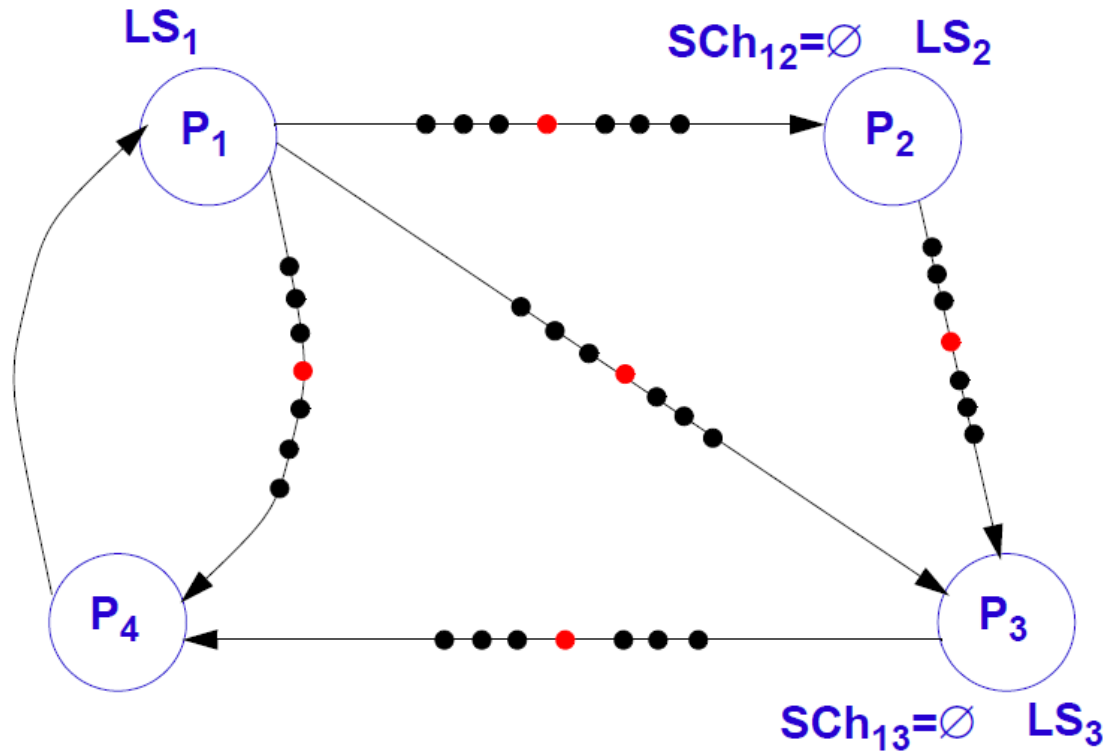
- $P_1$  initiates the global state recording: it saves its local state and send out the snapshot token on all its output channels.

# Global State Recording



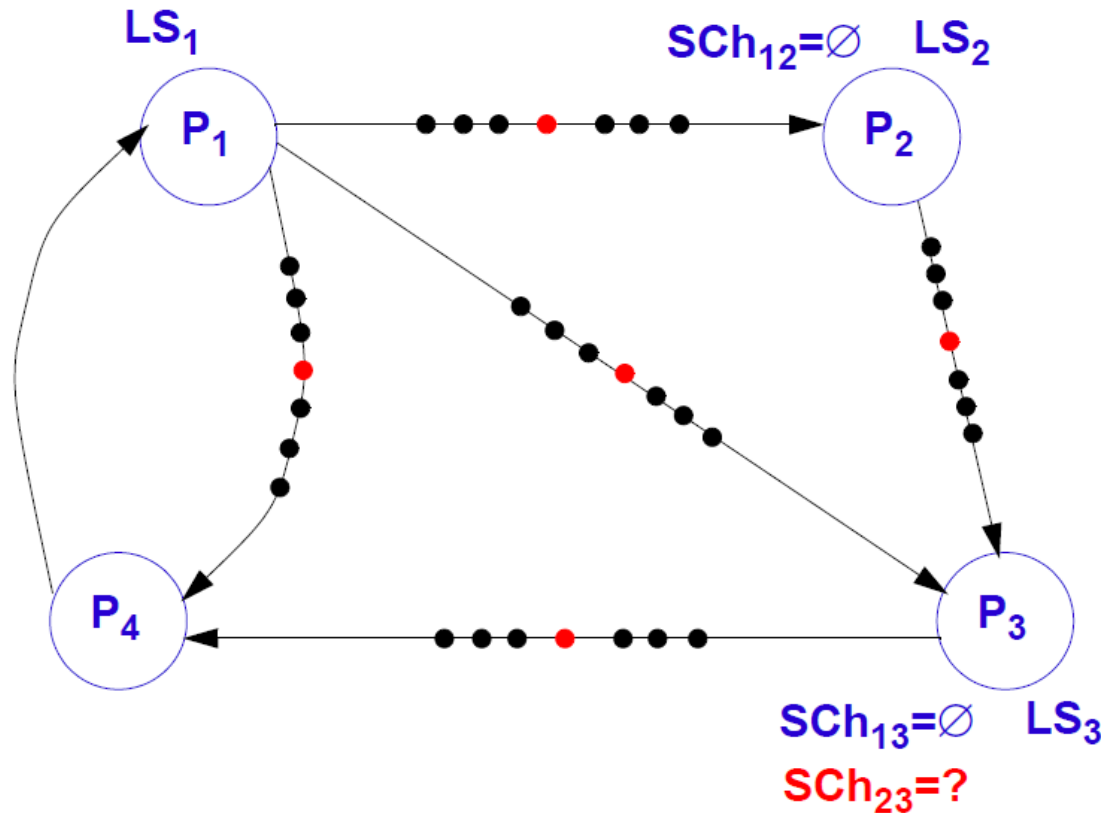
- $P_2$  receives the snapshot token from  $P_1$ : it saves its local state and the state of the channel  $P_1 \rightarrow P_2$ ; It sends the snapshot token on all its output channels.

# Global State Recording



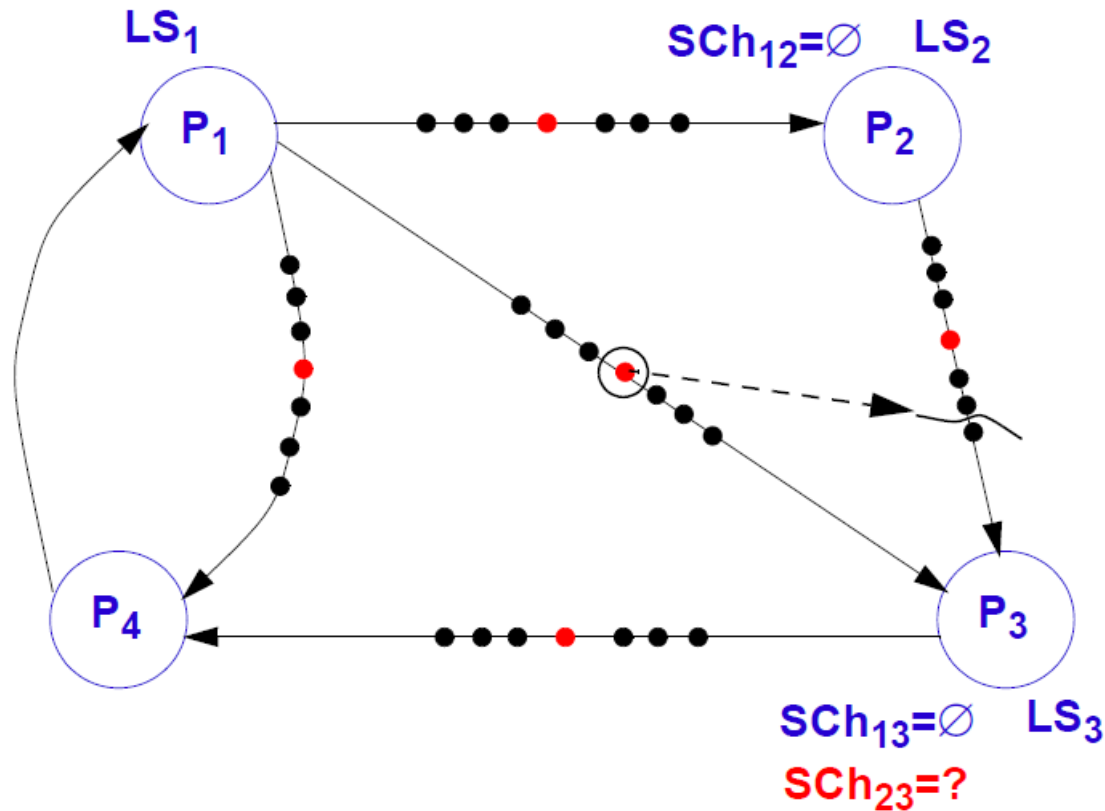
- $P_3$  receives the snapshot token from  $P_1$ : it saves its local state and the state of the channel  $P_1 \rightarrow P_3$ ; It sends the snapshot token on all its output channels.

# Global State Recording



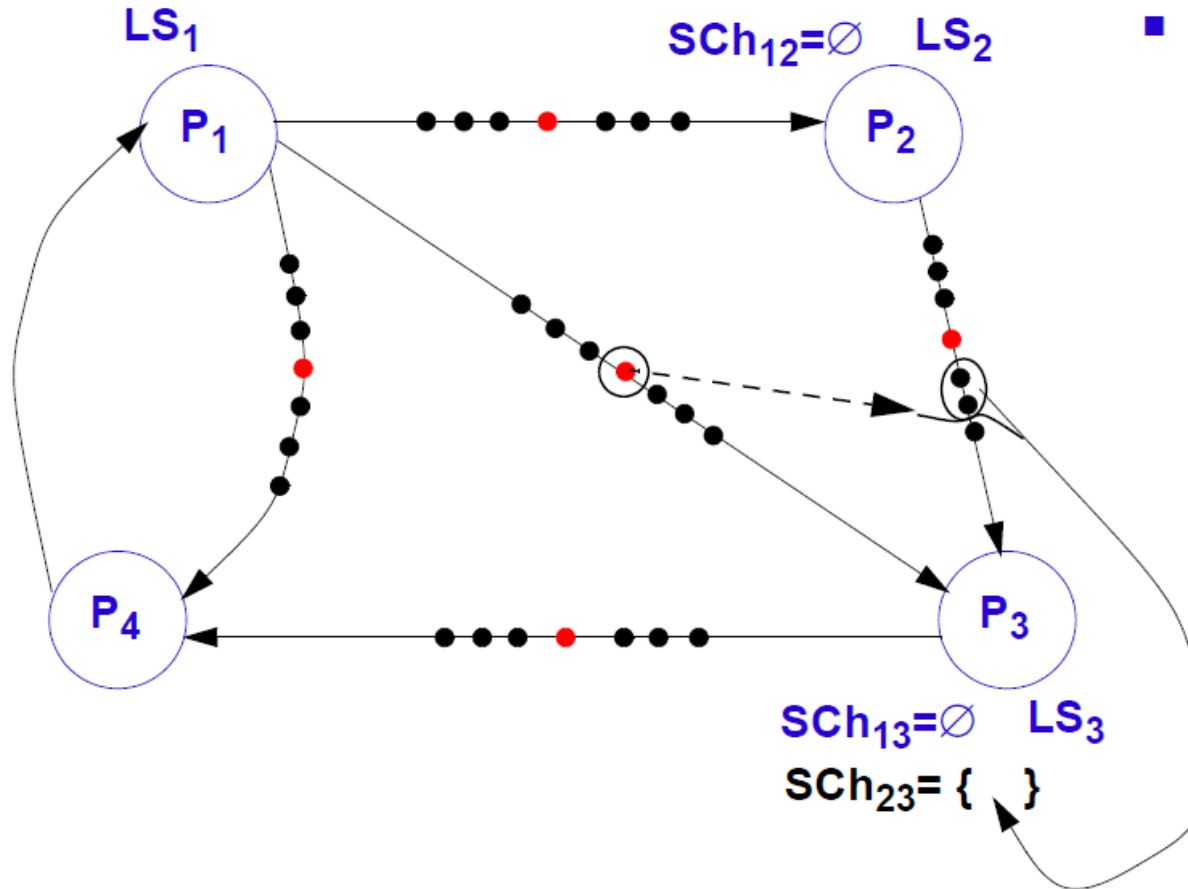
- $P_3$  receives the snapshot token from  $P_2$ : the local state  $LS_3$  is already saved! It saves the state of channel  $P_2 \rightarrow P_3$  so that it is consistent with states  $LS_2$  and  $LS_3$  (that already are saved)! (Snapshot token on its output channel is already sent!)

# Global State Recording



- $P_3$  receives the snapshot token from  $P_2$ : the local state  $LS_3$  is already saved! It saves the state of channel  $P_2 \rightarrow P_3$  so that it is consistent with states  $LS_2$  and  $LS_3$  (that already are saved)! (Snapshot token on its output channel is already sent!)

# Global State Recording



- $P_3$  receives the snapshot token from  $P_2$ : the local state  $LS_3$  is already saved! It saves the state of channel  $P_2 \rightarrow P_3$  so that it is consistent with states  $LS_2$  and  $LS_3$  (that already are saved)! (Snapshot token on its output channel is already sent!)

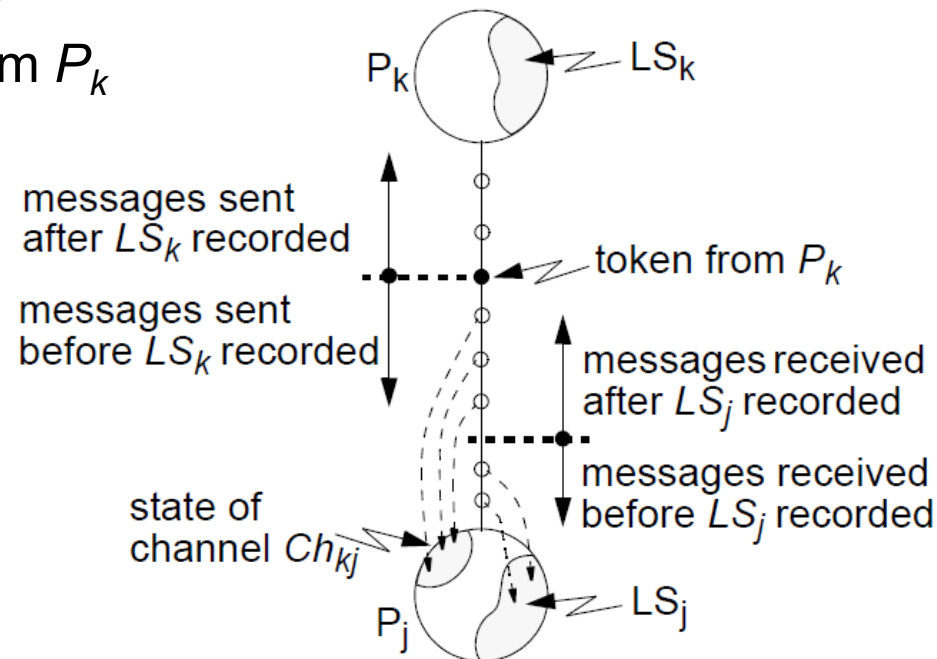
# Global State Recording

## Channel states?

- $P_i$  sends a token to  $P_j$  and this is the first time  $P_j$  received a token  
 →  $P_j$  immediately records its state.  
 All messages sent by  $P_i$  before sending the token  
 have been received at  $P_j$   
 →  $SCh_{ij} := \emptyset$ .

- $P_j$  receives a token from  $P_k$ , but  $P_j$  already recorded its state.
  - $M$  is the set of messages that  $P_j$  received from  $P_k$   
 after  $P_j$  recorded its state and  
 before  $P_j$  received the token from  $P_k$   
 →  $SCh_{kj} := M$ .

- The algorithm terminates when all processes have received tokens on all their input channels.
- The process that initiated the snapshot should be informed; it can collect the global snapshot.



# Global State Recording

- **Rule for sender  $P_i$ :**

*/\* performed by the initiating process  
and by any other process at the reception of the first token \*/*

[SR1]:  $P_i$  records its state.

[SR2]:  $P_i$  sends a token on each of its outgoing channels.

- **Rule for receiver  $P_j$ :**

*/\* executed whenever  $P_j$  receives a token from another process  $P_i$   
on channel  $Ch_{ij}$  \*/*

[RR1]: **if**  $P_j$  has not yet recorded its state then

Record the state of the channel:  $SCh_{ij} := \emptyset$ .

Follow the "Rule for sender".

**else**

Record the state of the channel:  $SCh_{ij} := M$ ,  
where  $M$  is the set of messages that  $P_j$  received from  $P_i$   
after  $P_j$  recorded its state and  
before  $P_j$  received the token on  $Ch_{ij}$ .

**end if.**



# Acknowledgments

- Most of the slide contents is based on a previous version by Petru Eles, IDA, Linköping University.