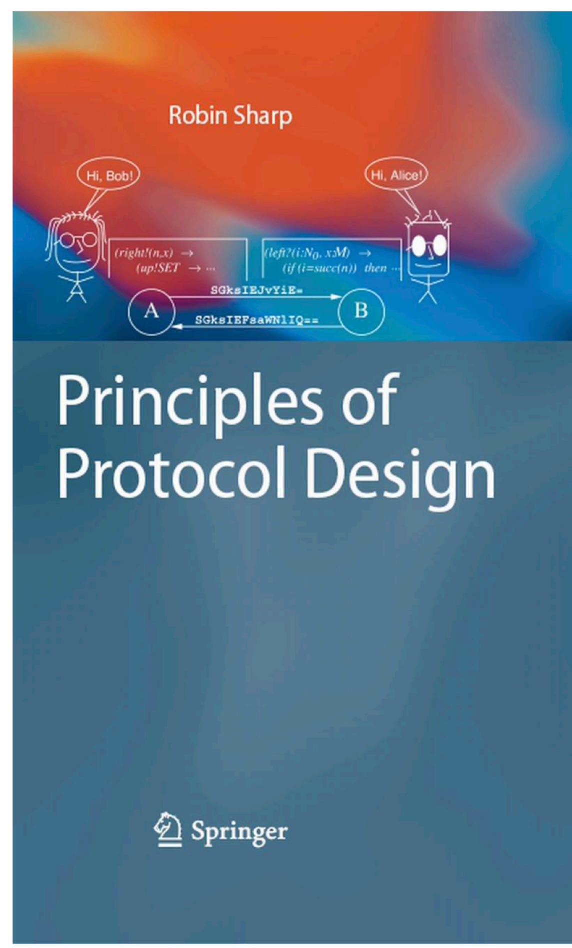
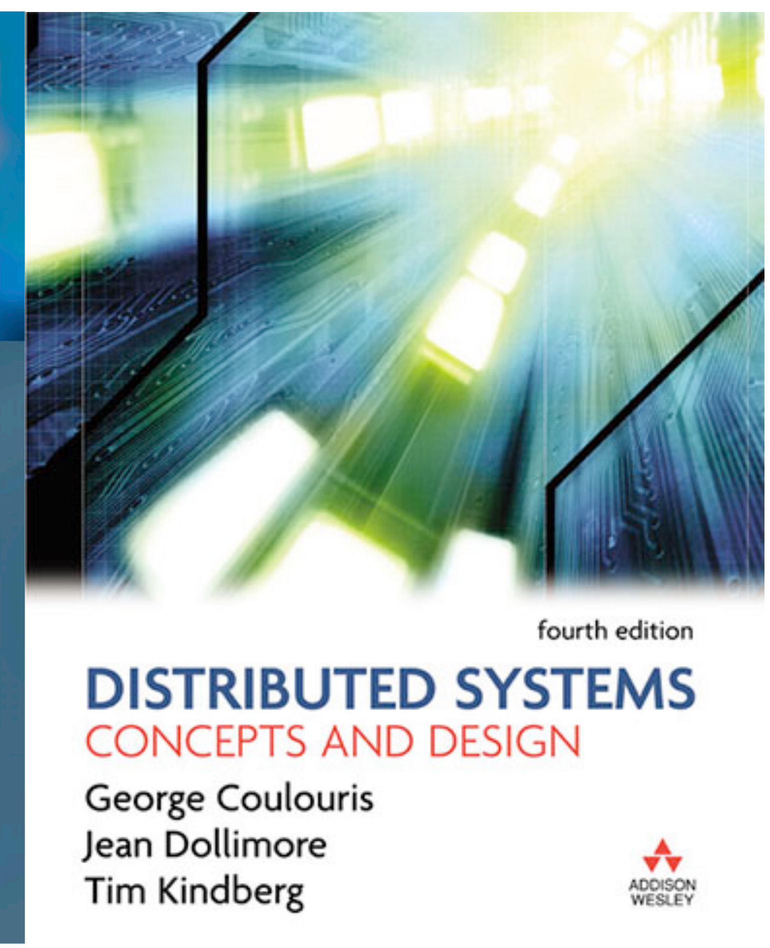
Logical Time and Global States

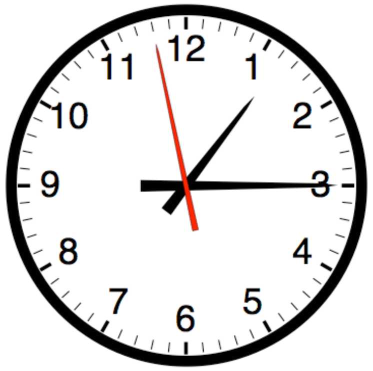
Nicola Dragoni
Embedded Systems Engineering
DTU Informatics

- Introduction
- Clock, Events and Process States
- Logical Clocks
- Global States





Why Is Time Interesting?



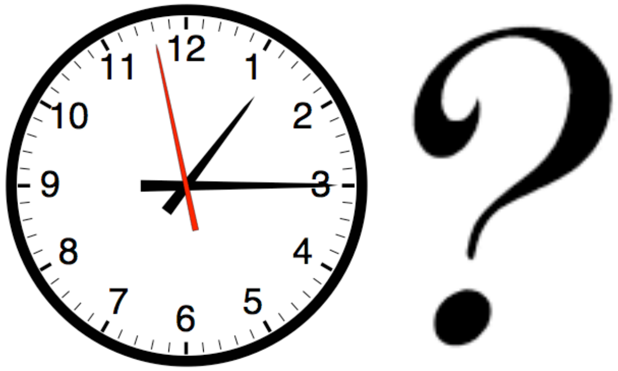
- Ordering of events: what happened first?
  - **Storage of data** in memory, file, database, ...
  - **Requests for exclusive access** - who asked first?
  - **Interactive exchanges** - who answered first?
  - **Debugging** - what could have caused the fault?
- Causality is linked to temporal ordering:

  if e_i causes e_j, it must happen before e_j

Distributed System Model

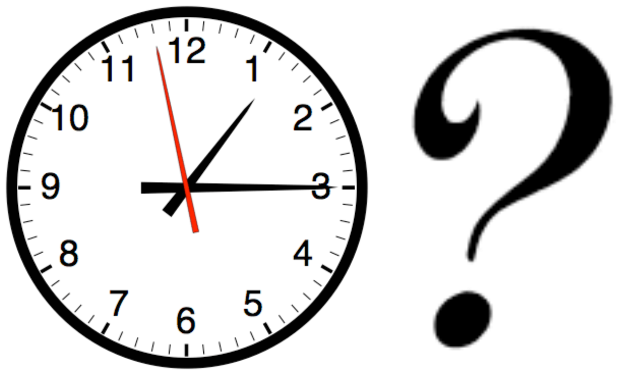
- We consider the following asynchronous distributed system:
  - N processes p_i, i = 1, ..., N
  - each process executes on a single processor
  - processors do **not** share memory --> processes communicate only by message passing
  - Actions of a process p_i: communicating actions (Send or Receive) or state transforming actions (such as changing the value of a variable)
- Event: occurrence of a single action that a process carries out as it executes

What Do We Know About Time?



- We **cannot** synchronize clocks *perfectly* across a distributed system
  - ➡ *We cannot in general use physical time to find out the order of any arbitrary pair of events occurring within a distributed system.* [Lamport, 1978]

What Do We Know About Time?



- We **cannot** synchronize clocks *perfectly* across a distributed system

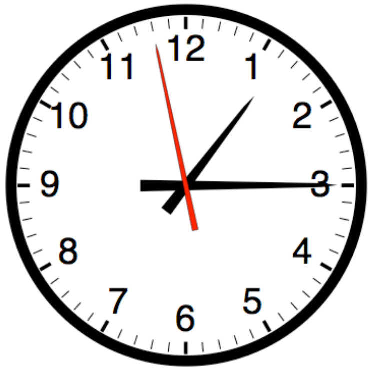
  ➡ *We cannot in general use physical time to find out the order of any arbitrary pair of events occurring within a distributed system.* [Lamport, 1978]

- The sequence of events within a single process p_i can be placed in a total ordering, denoted by the relation $\rightarrow_i$ ("occurs before)" between the events.

$$e \rightarrow_i e' \text{ if and only if the event } e \text{ occurs before } e' \text{ at } p_i$$

In other words: if two events occurred at the same process p_i, then they occurred in the order in which p_i observes them.

What Do We Know About Time?



- We **cannot** synchronize clocks *perfectly* across a distributed system

  ➡ *We cannot in general use physical time to find out the order of any arbitrary pair of events occurring within a distributed system.* [Lamport, 1978]

- The sequence of events within a single process p_i can be placed in a total ordering, denoted by the relation $\rightarrow_i$ ("occurs before)" between the events.

  e $\rightarrow_i$ e' if and only if the event e occurs before e' at p_i

  In other words: if two events occurred at the same process p_i, then they occurred in the order in which p_i observes them.

- Whenever a message is sent between two processes, the event of sending the message occurred before the event of receiving the message.

Happened-Before Relation ($\rightarrow$)

- Lamport's happened-before relation $\rightarrow$ (or causal ordering):

HB1: If $\exists$ process p_i : $e \rightarrow_i e'$, then $e \rightarrow e'$.

HB2: For any message m, send(m) $\rightarrow$ receive(m)

HB3: If e, e', e'' are events such that $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$.

Happened-Before Relation (→)

- Lamport's happened-before relation → (or causal ordering):

  HB1: If ∃ process p_i : $e \rightarrow_i e'$, then $e \rightarrow e'$.

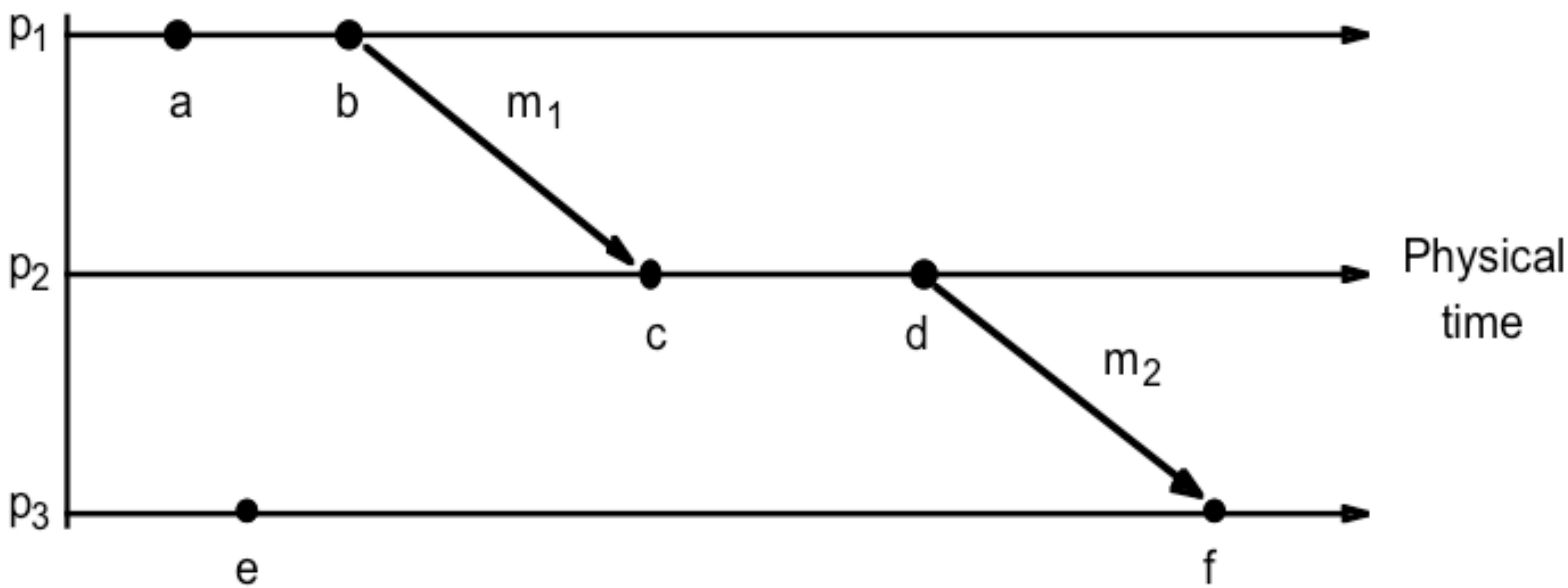
  HB2: For any message m, send(m) → receive(m)

  HB3: If e, e', e" are events such that e → e' and e' → e" then e → e".

- Thus, if e and e' are events, and if e → e', then we can find a series of events $e_1, e_2, ..., e_n$ occurring at one or more processes such that
  - $e = e_1$
  - $e' = e_n$
  - for i = 1, 2, ..., N-1 either HB1 or HB2 applies between e_i and e_{i+1}.

  In other words: either they occur in succession at the same process, or there is a message m such that e_i = send(m) and e_{i+1} = receive(m).

[Happened Before Relation] Example



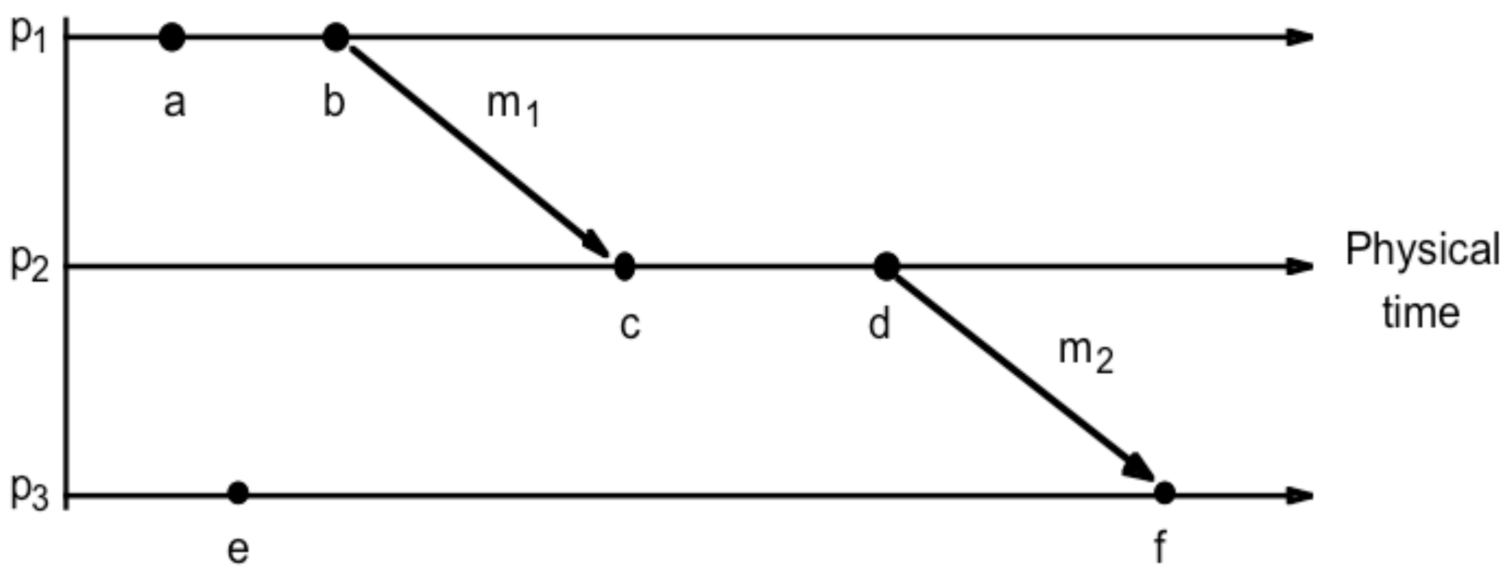
- $a \rightarrow b$, since the events occur in this order at process p_1 ($a \rightarrow_1 b$).
- $c \rightarrow d$.
- $b \rightarrow c$, since these events are the sending and reception of message m_1.
- $d \rightarrow f$, similarly.
- Combining these relations, we may also say that, for example, $a \rightarrow f$.

Happened-Before Relation ($\rightarrow$)

- Note that the $\rightarrow$ relation is an IRREFLEXIVE PARTIAL ORDERING on the set of all events in the distributed system.
  - Irreflexivity: $\neg(a \rightarrow a)$.
  - Partial ordering: not all the events can be related by $\rightarrow$.

Happened-Before Relation (→)

- Note that the → relation is an IRREFLEXIVE PARTIAL ORDERING on the set of all events in the distributed system.
  - Irreflexivity: ¬(a → a).
  - Partial ordering: not all the events can be related by →.



- ¬(a → e) and ¬(e → a) since they occur at different processes, and there is no chain of messages intervening between them.
- We say that a and e are not ordered by →; a and b are concurrent (a || b).

Logical Clocks

- Each process p_i keeps its own logical clock, L_i, which it uses to apply so-called *Lamport timestamps* to events.
- Intuition: a logical clock is a monotonically increasing software counter, which associates a value in an ordered domain with each event in a system.
- Ordering relation: $\rightarrow$

Logical Clocks

- Each process p_i keeps its own logical clock, L_i, which it uses to apply so-called *Lamport timestamps* to events.
- Intuition: a logical clock is a monotonically increasing software counter, which associates a value in an ordered domain with each event in a system.
- Ordering relation: $\rightarrow$

Definition: local logical clock L_i in process p_i is a function which associates a value, $L_i(e)$, in an ordered set V with each event e in p_i.

Logical Clocks

- Each process p_i keeps its own logical clock, L_i, which it uses to apply so-called *Lamport timestamps* to events.
- Intuition: a logical clock is a monotonically increasing software counter, which associates a value in an ordered domain with each event in a system.
- Ordering relation: $\rightarrow$

Definition: local logical clock L_i in process p_i is a function which associates a value, $L_i(e)$, in an ordered set V with each event e in p_i.

- Note that values of a logical clock need bear no particular relationship to any physical clock.

Logical Clocks Rules

- To match the definition of $\rightarrow$, we require the following clock rules:

CR1: If $\exists$ process p_i such that $e \rightarrow_i e'$, then $L_i(e) < L_i(e')$.

CR2: If a is the sending of a message by p_i and b is the receipt of the same message by p_j, then $L_i(a) < L_j(b)$.

CR3: If e, e', e'' are three events such that $L(e) < L(e')$ and $L(e') < L(e'')$ then $L(e) < L(e'')$.

Ok, but how to do that in practice?

Logical Clocks in Practice

- To capture the → relation, processes update their logical clocks and transmit the values of their logical clocks in messages as follows:

LC1: L_i is incremented *before* each event is issued at process p_i: $L_i := L_i + 1$.

LC2: (a) When p_i sends a msg m, it piggybacks on m the value $t = L_i$.

(b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event *receive(m)*.

Logical Clocks in Practice

- To capture the → relation, processes update their logical clocks and transmit the values of their logical clocks in messages as follows:

LC1: L_i is incremented *before* each event is issued at process p_i: $L_i := L_i + 1$.

LC2: (a) When p_i sends a msg m, it piggybacks on m the value $t = L_i$.

(b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event *receive(m)*.

- Although we increment clocks by 1, we could have chosen any positive value.

Logical Clocks in Practice

- To capture the → relation, processes update their logical clocks and transmit the values of their logical clocks in messages as follows:

  LC1: L_i is incremented *before* each event is issued at process p_i: $L_i := L_i + 1$.

  LC2: (a) When p_i sends a msg m, it piggybacks on m the value $t = L_i$.

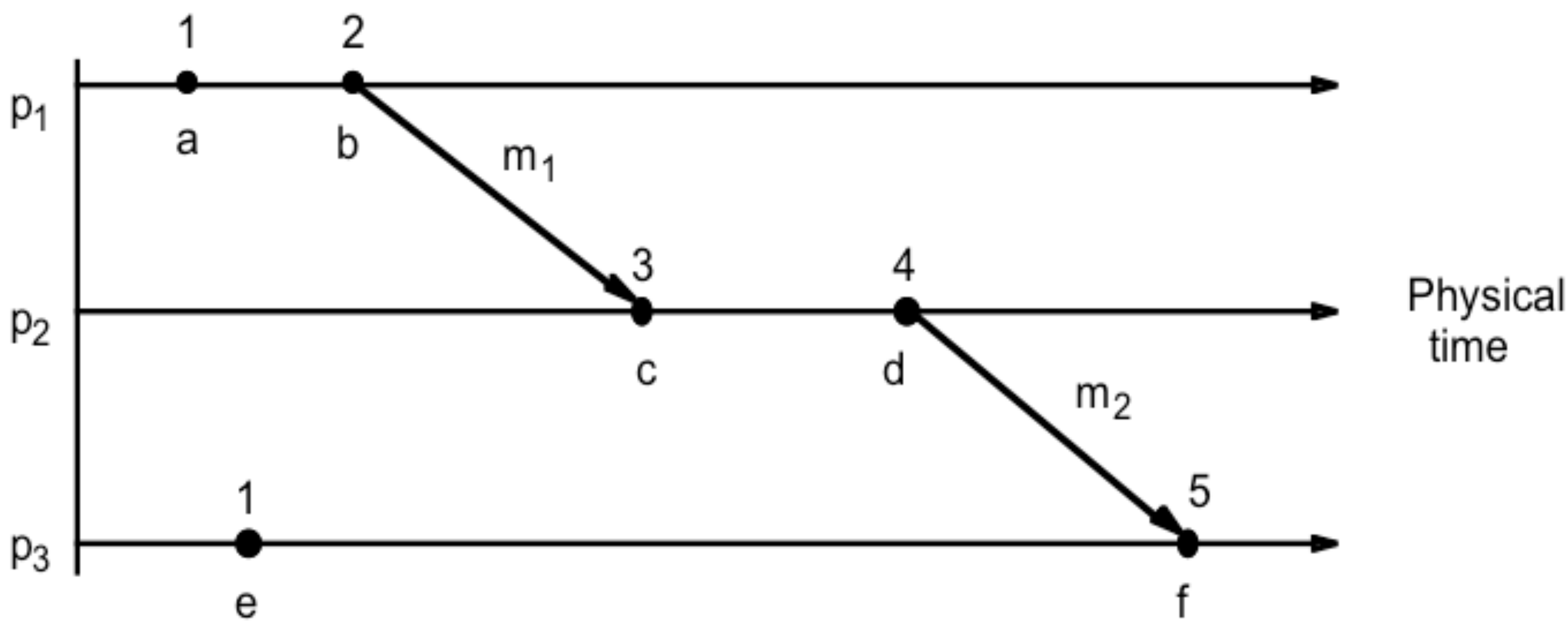
  (b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event *receive(m)*.

- Although we increment clocks by 1, we could have chosen any positive value.

- Clocks which follow these rules are known as LAMPORT LOGICAL CLOCKS.

$$\boxed{e \rightarrow e' \Rightarrow L(e) < L(e')}$$

[Lamport Clocks] Example 1

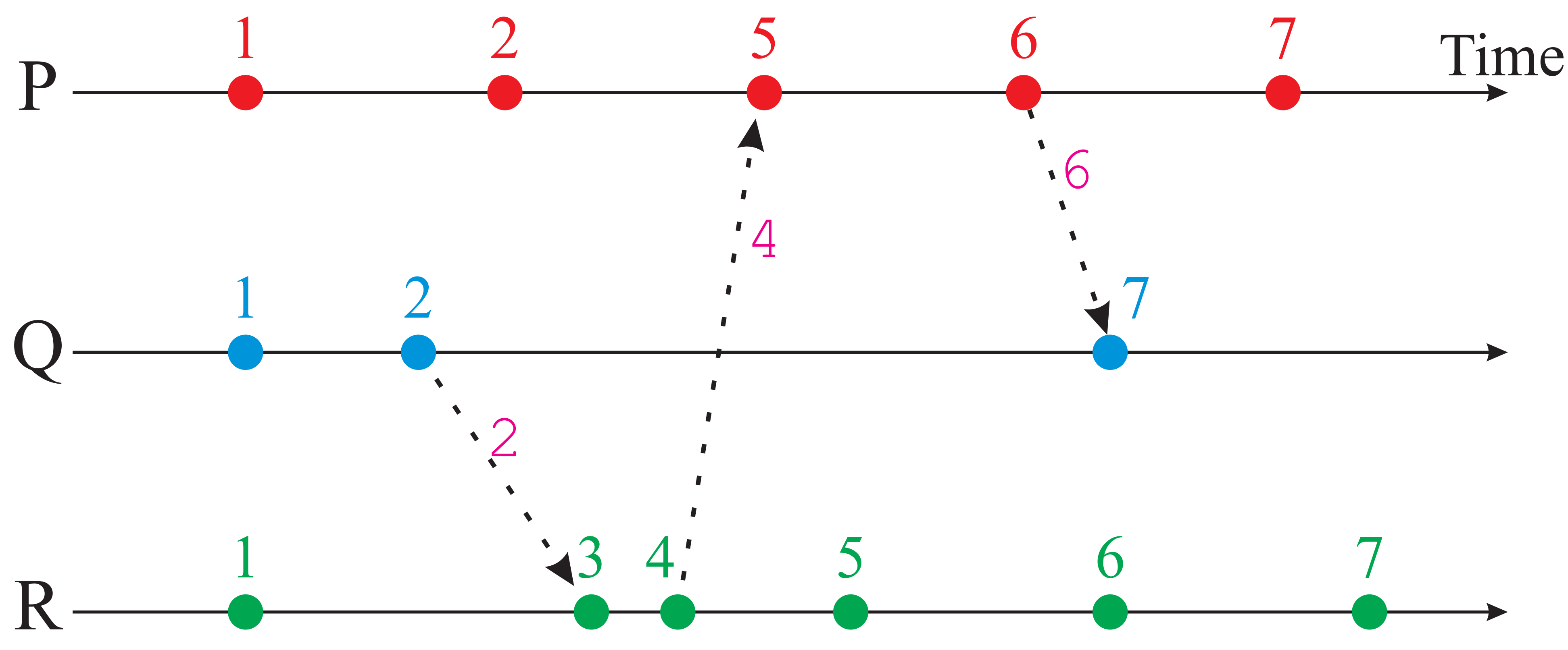


LC1: L_i is incremented *before* each event is issued at process p_i: $L_i := L_i + 1$.

LC2: (a) When p_i sends a msg m, it piggybacks on m the value $t = L_i$.

(b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event *receive(m)*.

[Lamport Clocks] Example 2

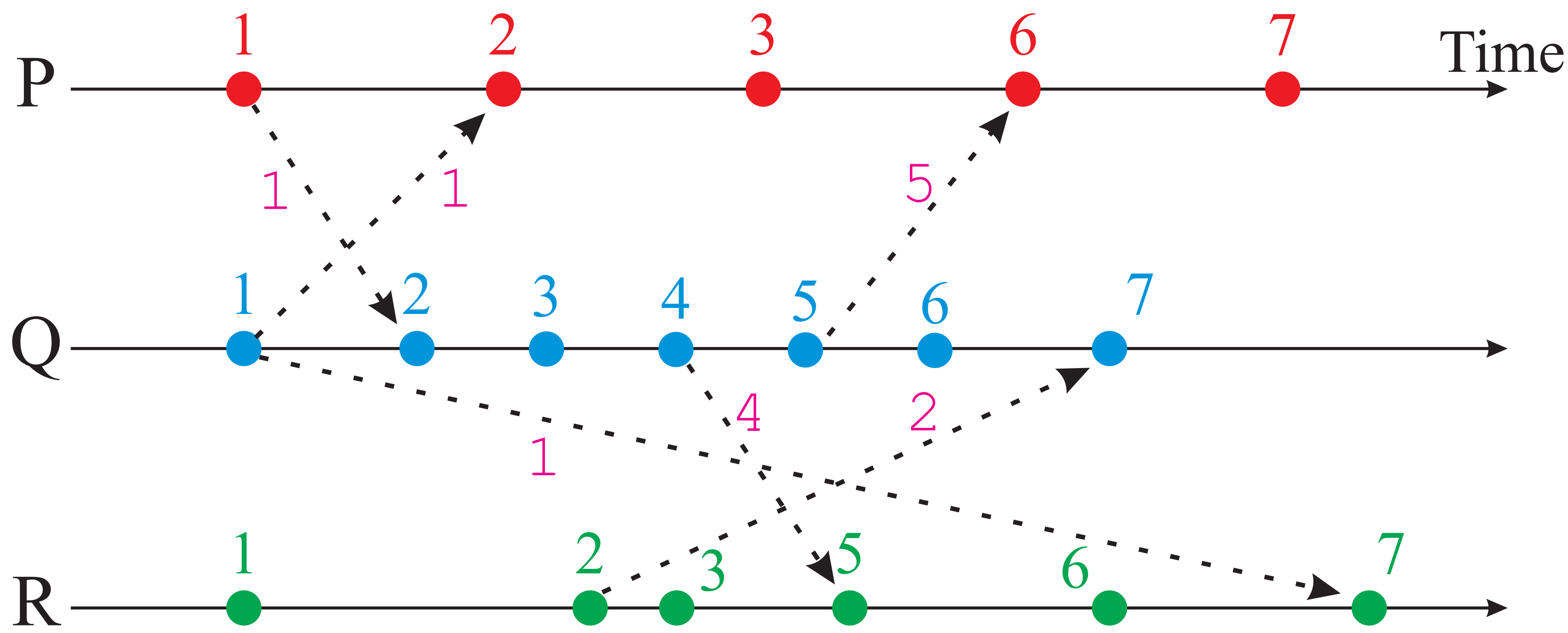


LC1: L_i is incremented *before* each event is issued at process p_i: $L_i := L_i + 1$.

LC2: (a) When p_i sends a msg m, it piggybacks on m the value $t = L_i$.

(b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event *receive(m)*.

[Lamport Clocks] Example 3



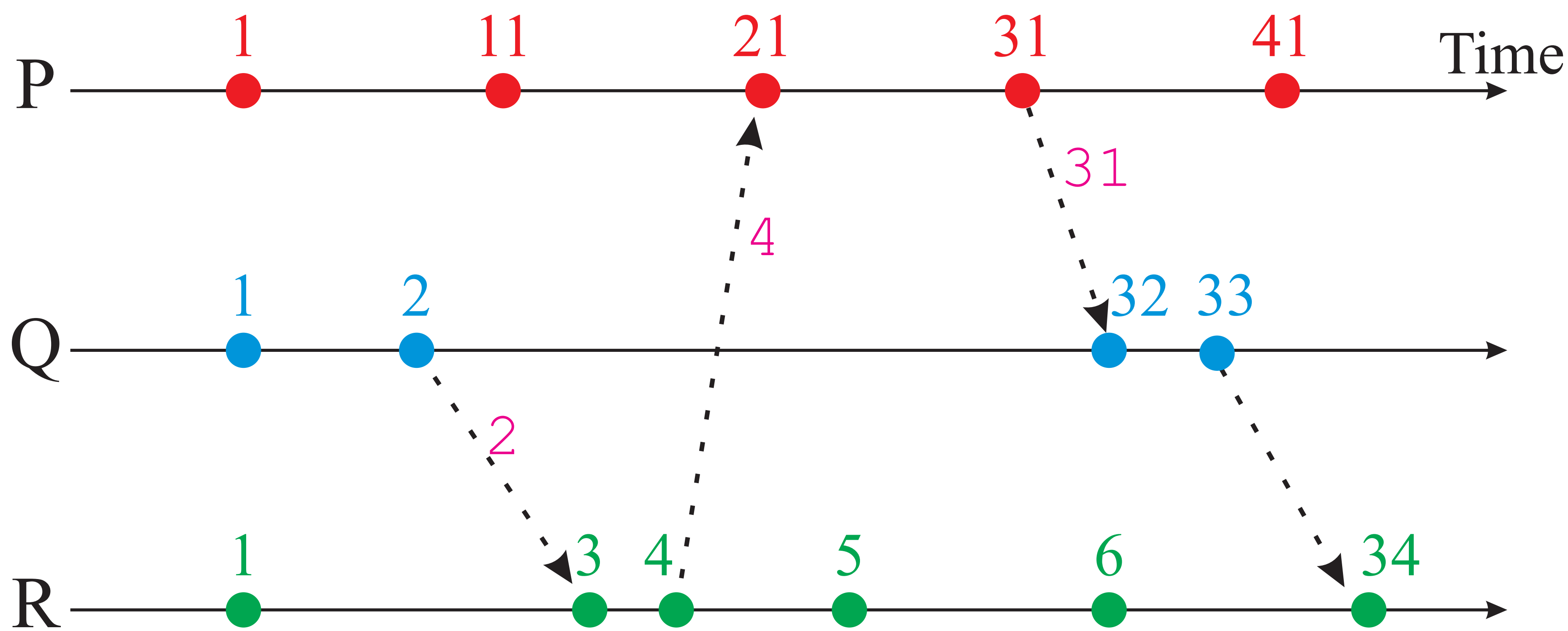
LC1: L_i is incremented *before* each event is issued at process p_i: $L_i := L_i + 1$.

LC2: (a) When p_i sends a msg m, it piggybacks on m the value $t = L_i$.

(b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event *receive(m)*.

[Lamport Clocks] Example 4

LOCAL CLOCKS TEND TO RUN AS FAST AS THE FASTEST OF THEM



LC1: L_i is incremented *before* each event is issued at process p_i: $L_i := L_i + 1$.

LC2: (a) When p_i sends a msg m, it piggybacks on m the value $t = L_i$.

(b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event *receive(m)*.

Homework

- By considering a chain of zero or more messages connecting events *e* and *e'* and using induction on the length of any sequence of events relating *e* and *e'*, show that $e \rightarrow e' \Rightarrow L(e) < L(e')$.

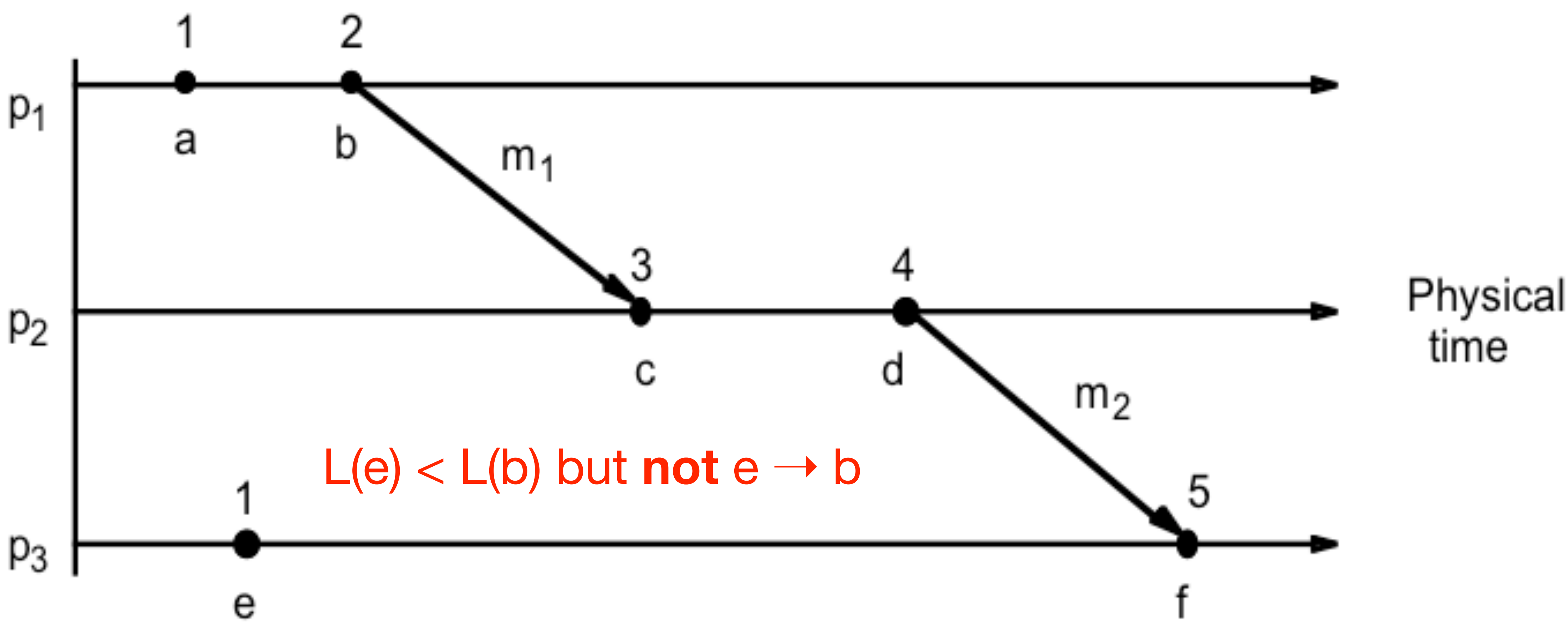
Homework

- The $\rightarrow$ relation is an IRREFLEXIVE PARTIAL ORDERING on the set of all events in the distributed system.
  - Irreflexivity: $\neg(a \rightarrow a)$.
  - Partial ordering: not all the events can be related by $\rightarrow$.

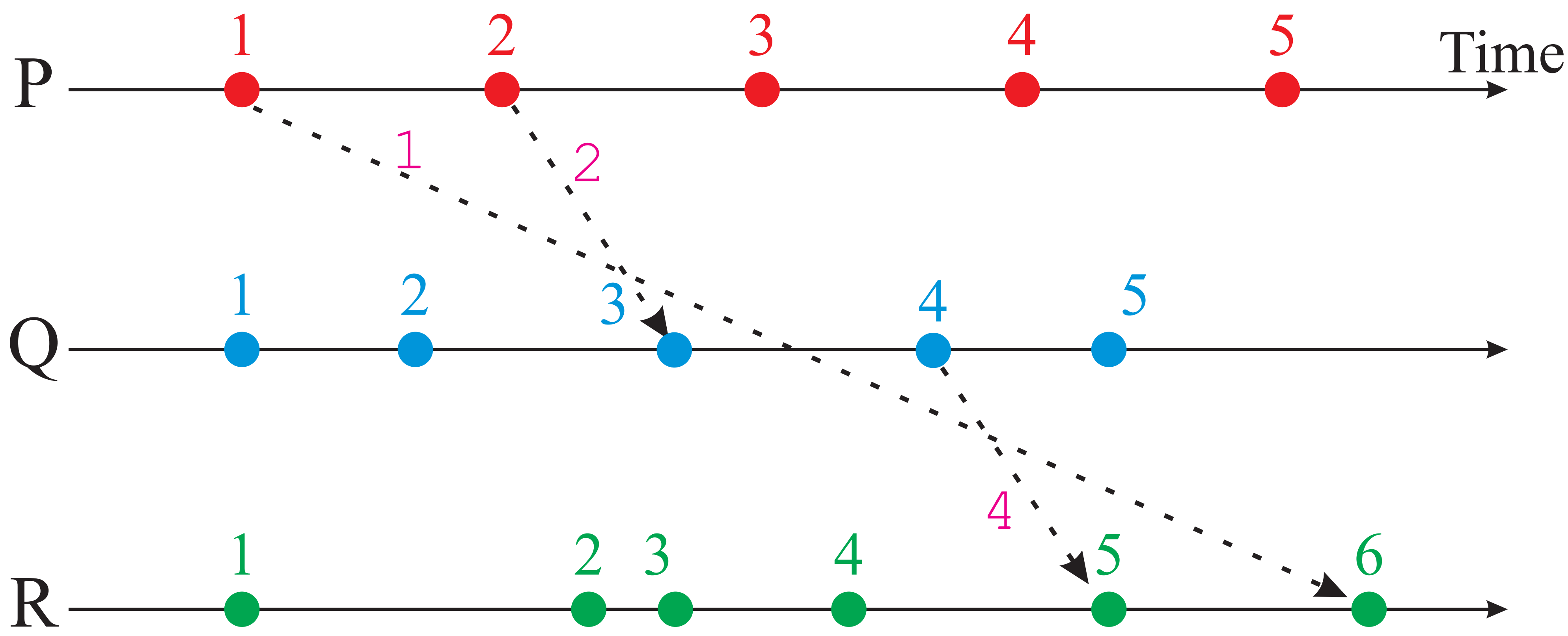
Extend the definition of the $\rightarrow$ relation to create a total ordering $\Rightarrow$ on events (that is, one for which all pairs of distinct events are ordered).

Shortcoming of Lamport clocks

A significant problem with Lamport clocks is that if $L(e) < L(e')$, then we **cannot** infer that $e \rightarrow e'$.

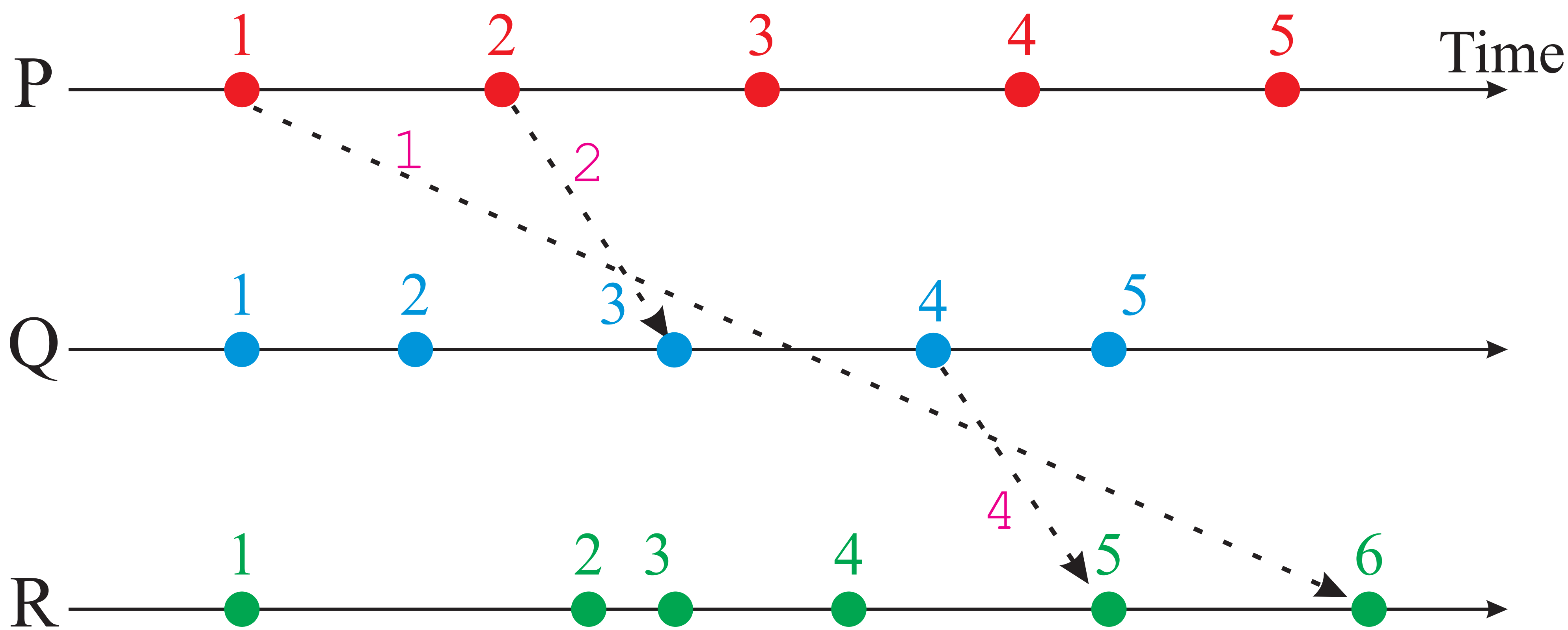


Another Problematic Scenario



- The message arriving at time 6 in R breaks the usual rules of causal ordering:
  - Event 1 in P causes event 5 in R
  - Event 1 in P causes event 6 in R

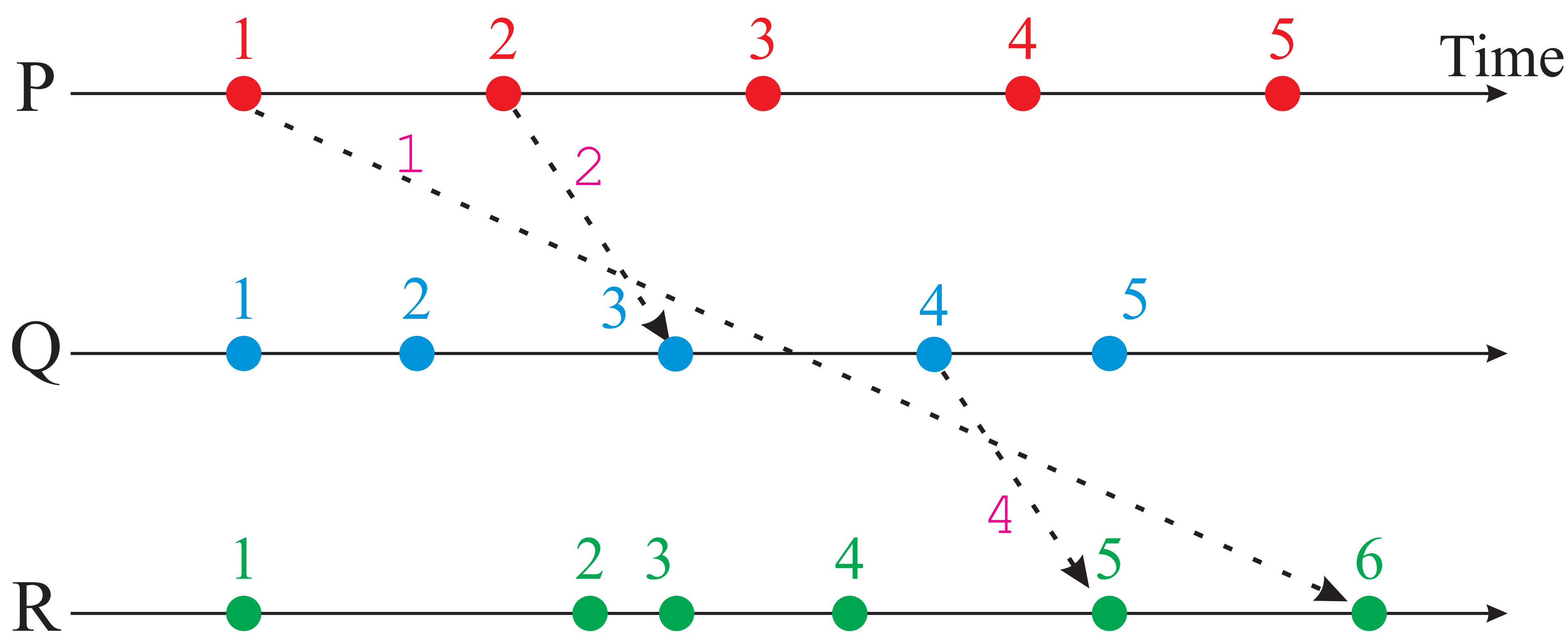
Another Problematic Scenario



- The message arriving at time 6 in R breaks the usual rules of causal ordering:
  - Event 1 in P causes event 5 in R
  - Event 1 in P causes event 6 in R

The message appears to be able to cause or be caused by the event at time 5!!

So... What Do We Need?



- This problem arises because only a single number is used to represent time.
- Idea: more info is needed to tell the receiving process what the sending process knew about the other clocks in the system when it sent the message.
- It would then become clear that the message arriving at time 6 in R was sent before the message arriving at time 5.

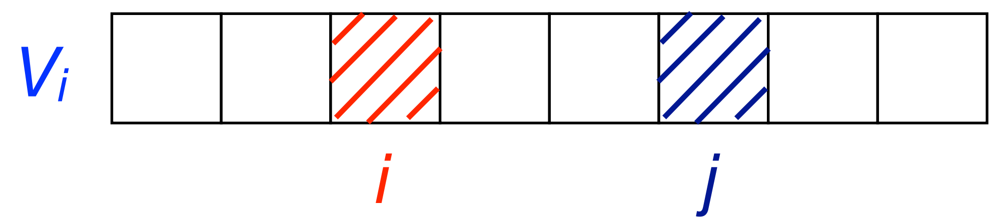
Mattern and Fidge Vector Clocks

- Developed to overcome the shortcoming of Lamport clocks
- Lamport clocks: $e \rightarrow f$ **then** $L(e) < L(f)$
- Vector clocks: $e \rightarrow f$ **iff** $V(e) < V(f)$
- Intuition: *Lamport clocks try to describe global time by a single number, which "hides" essential information.*
- Idea: *processes keep information on what they know about the other clocks in the system and use this information when sending a message*

Vector Clocks

- A vector clock for a system of N processes: array of N integers.

- Each process p_i keeps its own vector clock V_i, which it uses to timestamp local events.

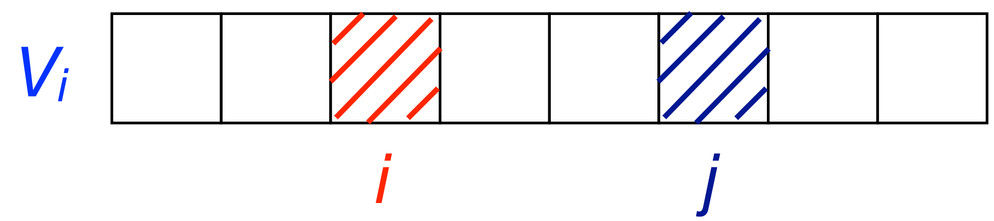


- Then $V_i[j]$ describes p_i's KNOWLEDGE of p_j's LOCAL LOGICAL CLOCK.

Vector Clocks

- A vector clock for a system of N processes: array of N integers.

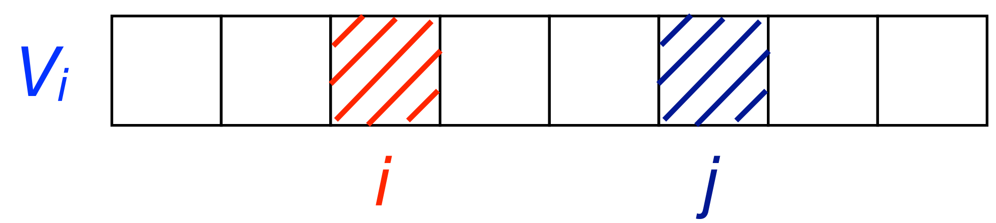
- Each process p_i keeps its own vector clock V_i, which it uses to timestamp local events.



- Then $V_i[j]$ describes p_i's KNOWLEDGE of p_j's LOCAL LOGICAL CLOCK.

- Example: if an event of p_2 is timestamped with (1, 1, 0) then p_2 knows that the value of the logical clocks are: 1 for p_1, 1 for p_2, 0 for p_3.

Note that...



- $V_i[j]$ $(j \neq i)$:
  - Latest clock value received by p_i from process p_j.
  - Number of events that have occurred at p_j that p_i has potentially been affected by.
    - *Process p_j may have timestamped more events by this point, but no information has flowed to p_i about them in messages yet!*

[Vector Clocks] Implementation Rules

VC1: Initially, $V_i[j] := 0$, for $i, j = 1, 2, \ldots., N$.

VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$.

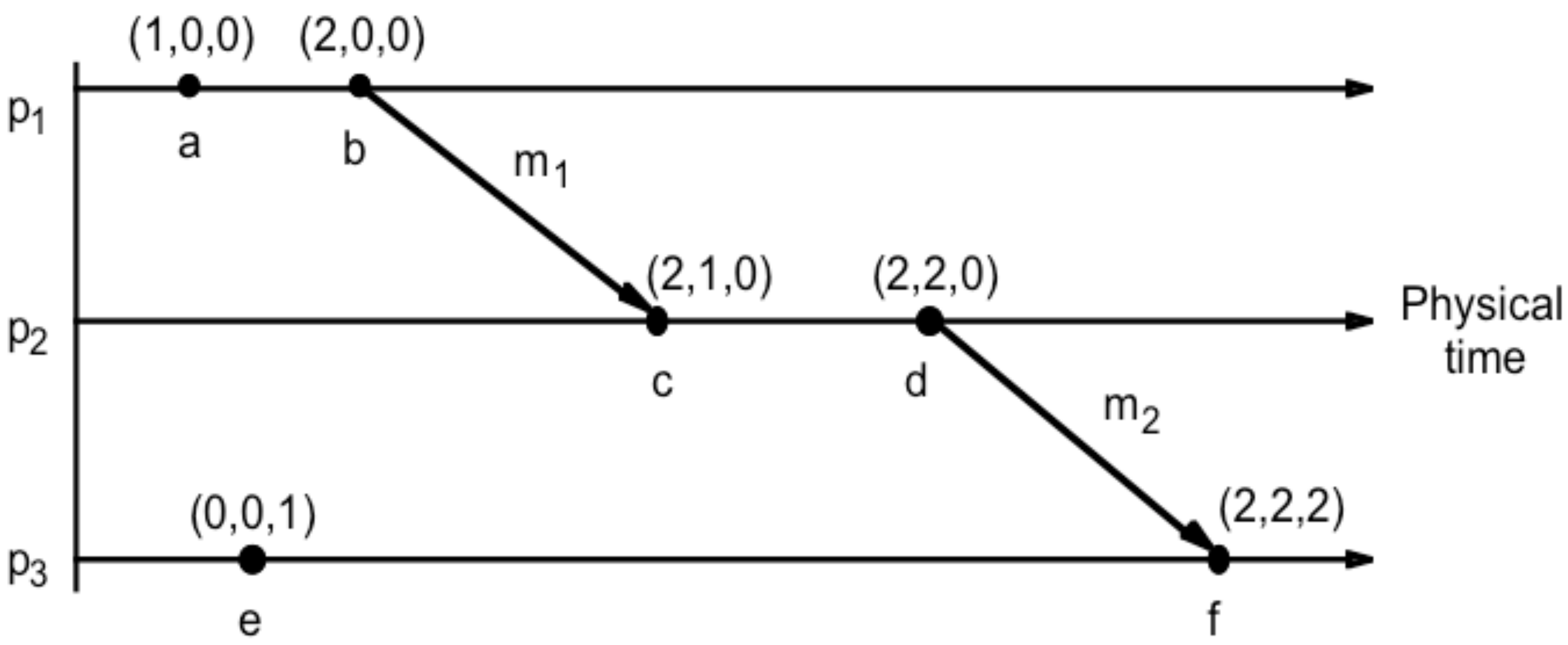
VC3: p_i includes the value $t = V_i$ in every message it sends.

VC4: When p_i receives a timestamp in a message, it sets

$V_i[j] := max(V_i[j], t[j])$ for $j = 1, 2, \ldots., N$

and then applies VC2 before timestamping the event *receive(m)*.

[Vector Clocks] Example



Ordering on Vectors

- For vector clocks using rules VC1-4, it follows that

- Ordering relation ($\leq$) on vectors:

$$\boxed{V \leq V' \Leftrightarrow V[j] \leq V'[j] \text{ for } j = 1, 2, \ldots, N}$$

- In particular:
  - $V = V' \Leftrightarrow V[j] = V'[j]$ for $j = 1, 2, \ldots, N$
  - $V < V' \Leftrightarrow V \leq V' \wedge V \neq V'$
  - $V \parallel V' \Leftrightarrow \neg(V < V') \wedge \neg(V' < V)$

Ordering on Vectors

- For vector clocks using rules VC1-4, it follows that

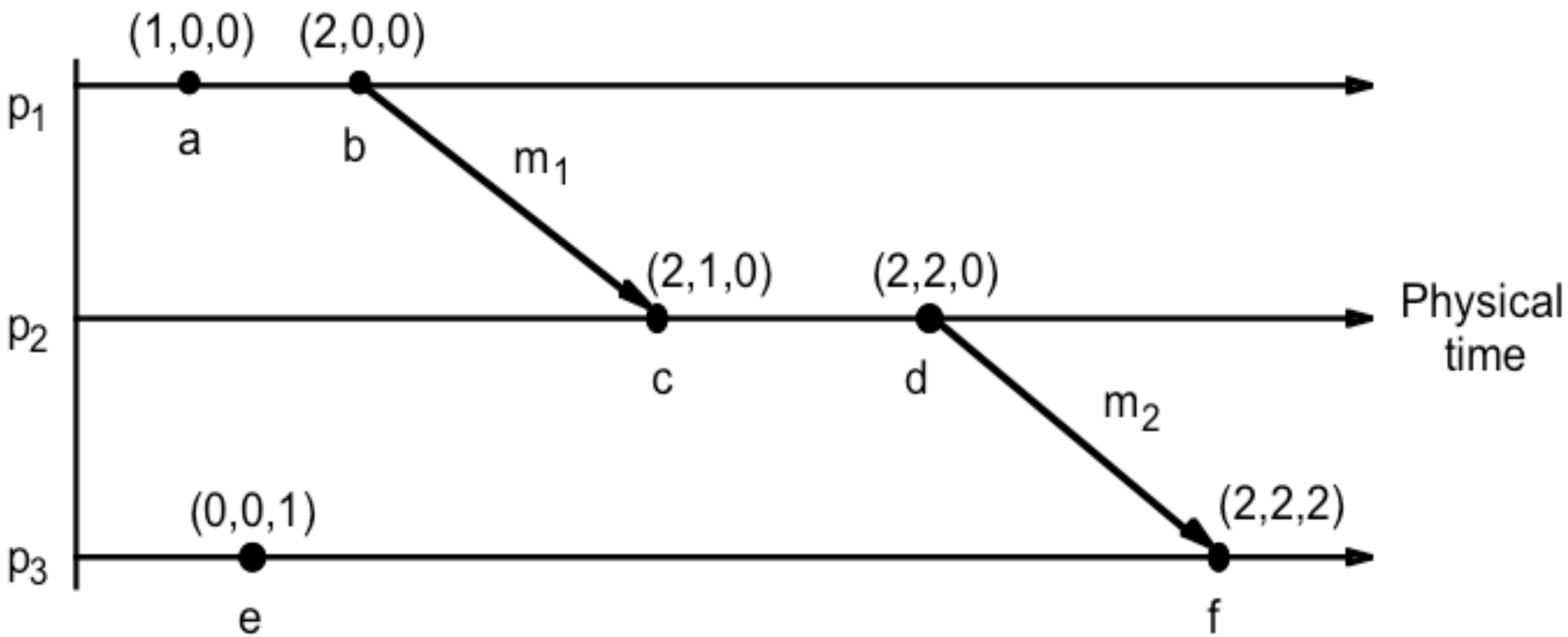
$$e \rightarrow e' \Leftrightarrow V(e) < V(e')$$

- Ordering relation ($\leq$) on vectors:

$$V \leq V' \Leftrightarrow V[j] \leq V'[j] \text{ for } j = 1, 2, \ldots, N$$

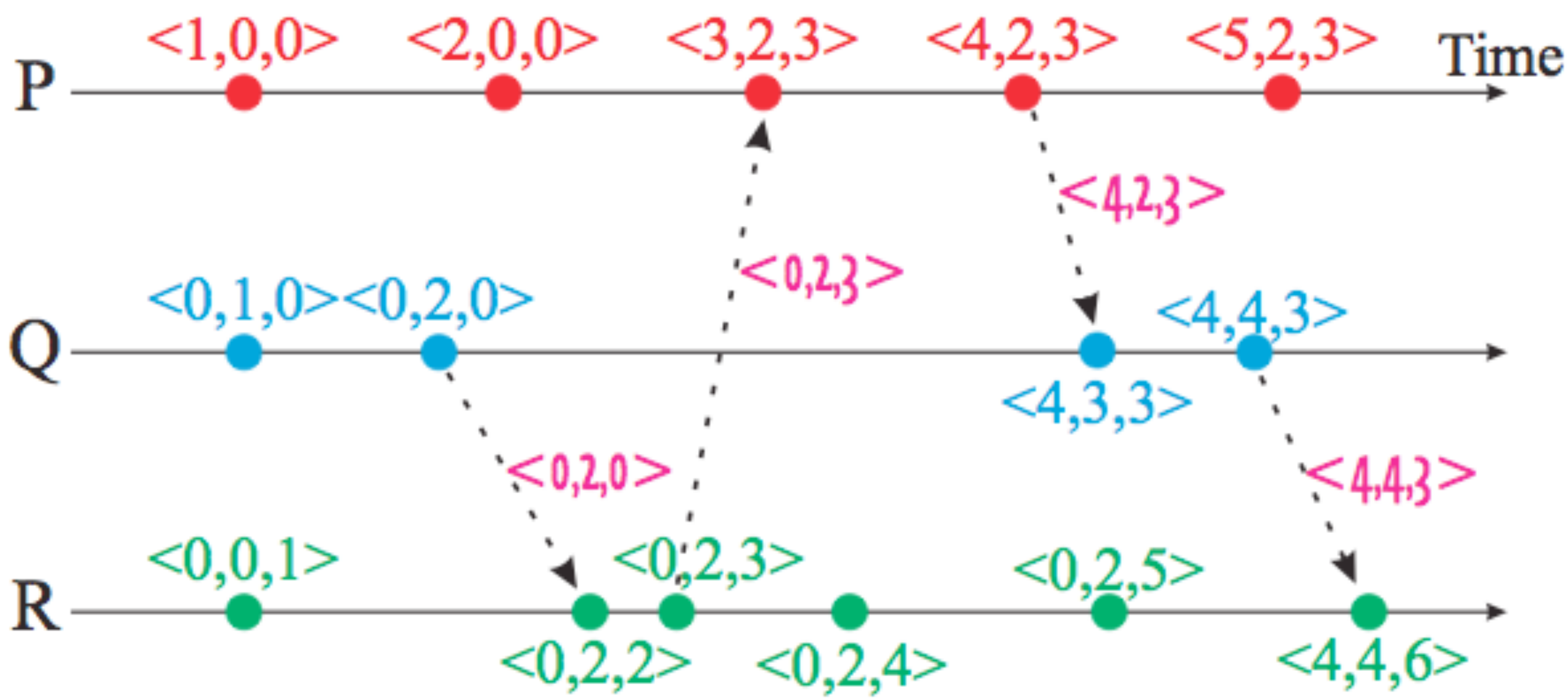
- In particular:
  - $V = V' \Leftrightarrow V[j] = V'[j]$ for $j = 1, 2, \ldots, N$
  - $V < V' \Leftrightarrow V \leq V' \wedge V \neq V'$
  - $V \parallel V' \Leftrightarrow \neg(V < V') \wedge \neg(V' < V)$

[Vector Clocks Ordering] Example



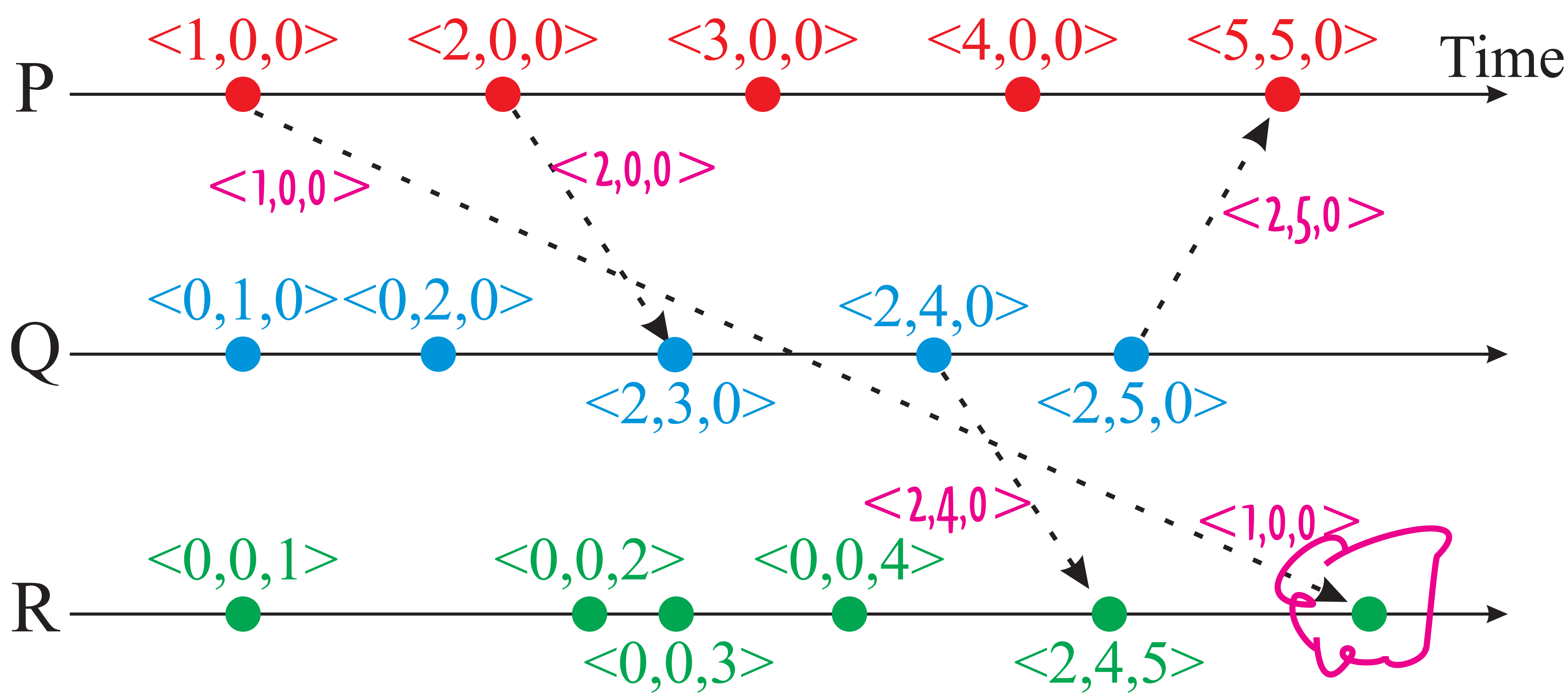
- *V(a)* < *V(f)*, reflecting the fact that *a* → *f*.
- *c* || *e* because neither *V(c)* ≤ *V(e)* nor *V(e)* ≤ *V(c)*.

[Vector Clocks] Example



[Vector Clocks] Violation of Causal Ordering

- Violation of causal ordering occurs if message M arrives with $V_M < V_i$.



- Here: $V_M[1] < V_R[1]$

Homework

1) Show that $V_j[i] \leq V_i[i]$.

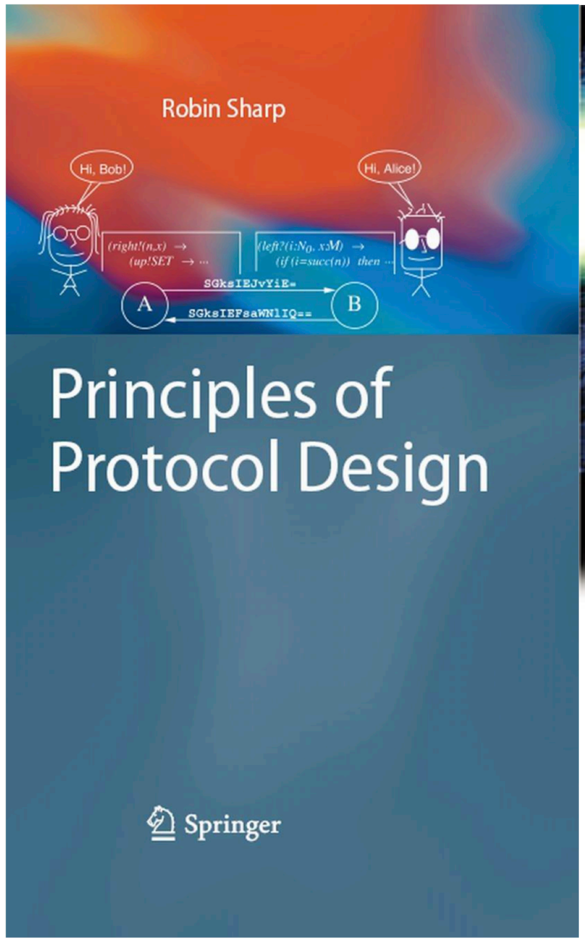
2) Show that $e \rightarrow e' \Rightarrow V(e) < V(e')$.

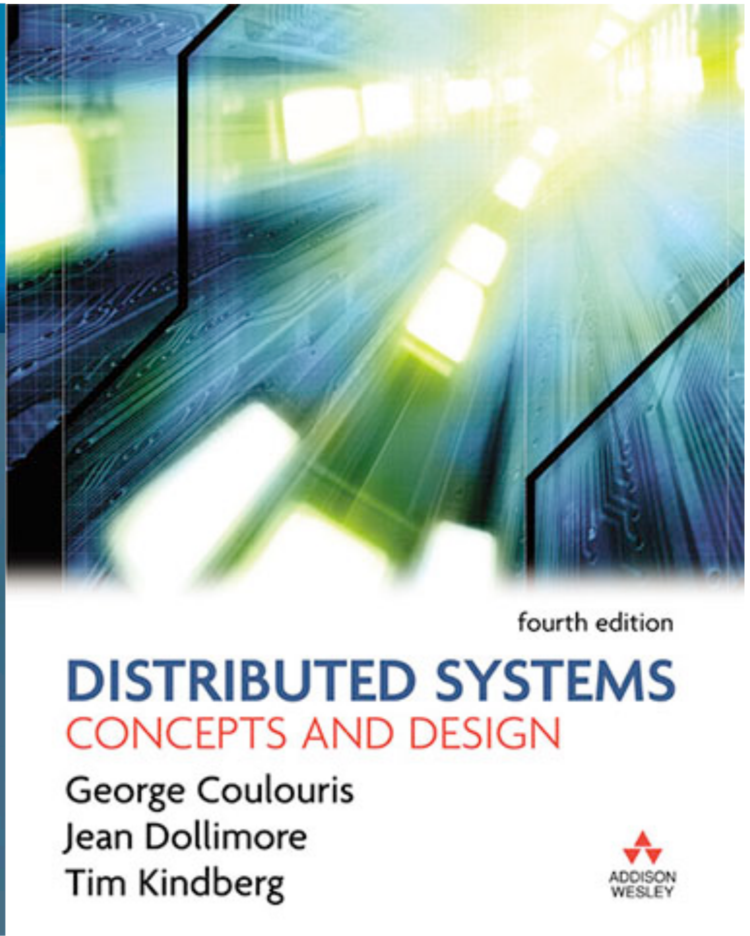
3) Using the result of Exercise 1), show that if events e and e' are concurrent then neither $V(e) \leq V(e')$ *nor* $V(e') \leq V(e)$.
   Hence show that if $V(e) < V(e')$ then $e \rightarrow e'$.

Logical Time and Global States

Nicola Dragoni
Embedded Systems Engineering
DTU Informatics

Introduction
Clock, Events and Process States
Logical Time and Logical Clocks
Global States





Problem: Finding the Global State

- Problem: to find the global state of a distributed system in which data items can move from one part of the system to another.

- Why? There are innumerable uses for this, for instance:
  - finding the *total number of files* in a distributed file system, where files may be moved from one file server to another
  - finding the *total space occupied by files* in such a distributed file system

Problem: Finding the Global State

- Problem: to find the global state of a distributed system in which data items can move from one part of the system to another.

- Why? There are innumerable uses for this, for instance:
  - finding the *total number of files* in a distributed file system, where files may be moved from one file server to another
  - finding the *total space occupied by files* in such a distributed file system

- Solution: distributed snapshot algorithm
  (Chandy and Lamport, 1985)

[Distributed Snapshots] Global State

- Idea: global states are described by
  - the states of the participating PROCESSES, together with
  - the states of the CHANNELS through which data (i.e., the files) pass when being transferred between these processes.

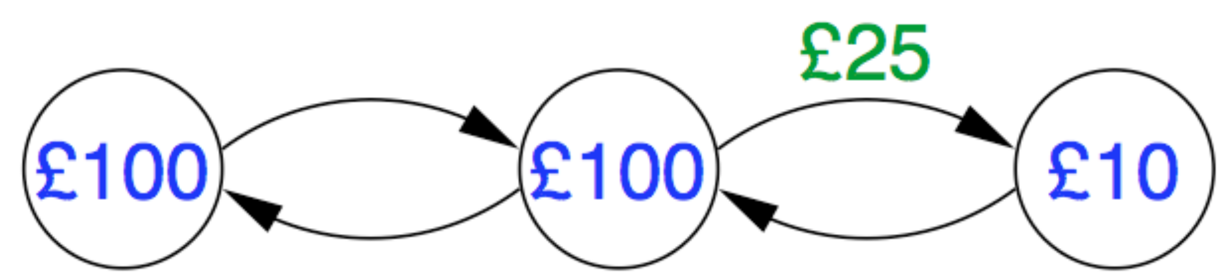
[Distributed Snapshots] Global State

- Idea: global states are described by
  - the states of the participating PROCESSES, together with
  - the states of the CHANNELS through which data (i.e., the files) pass when being transferred between these processes.

GLOBAL STATE: $\sum$ Money = £235

[Distributed Snapshots] Global State

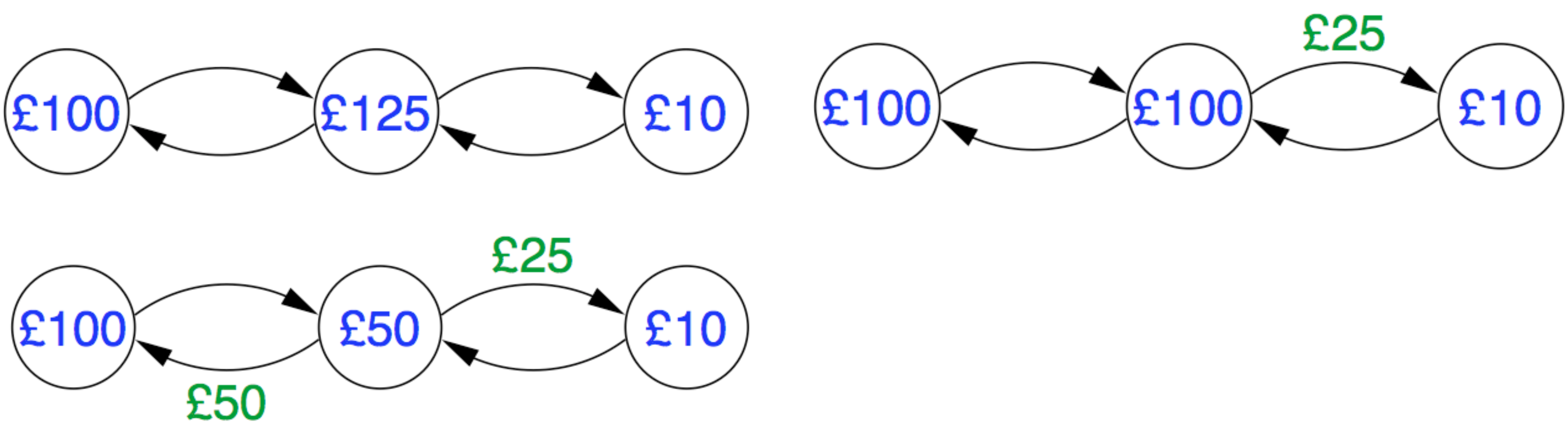
- Idea: global states are described by
  - the states of the participating PROCESSES, together with
  - the states of the CHANNELS through which data (i.e., the files) pass when being transferred between these processes.



GLOBAL STATE: $\sum$ Money = £235

[Distributed Snapshots] Global State

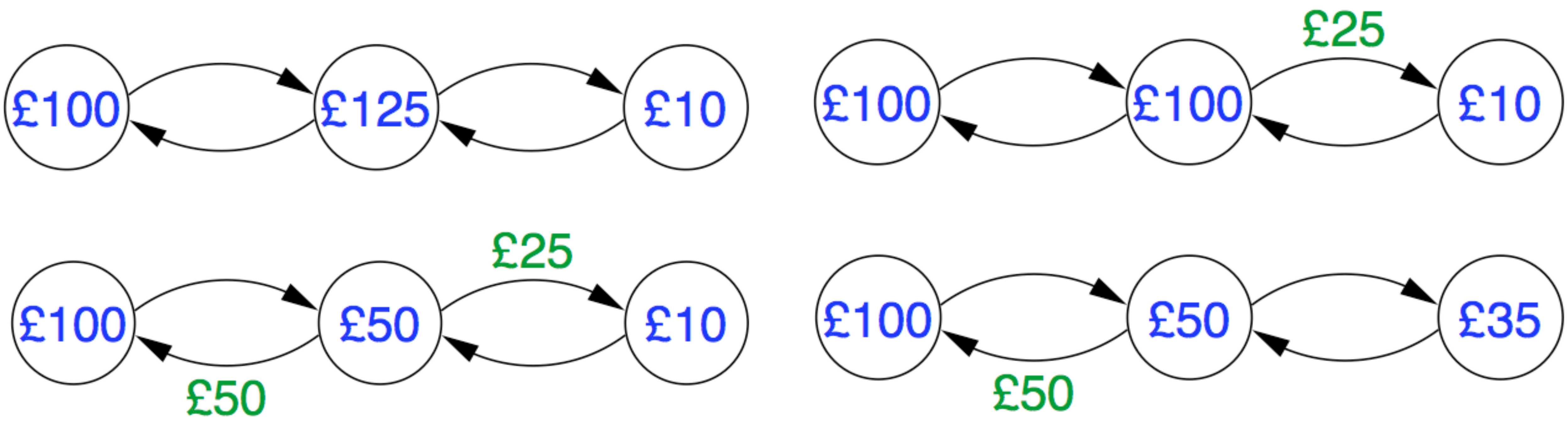
- Idea: global states are described by
  - the states of the participating PROCESSES, together with
  - the states of the CHANNELS through which data (i.e., the files) pass when being transferred between these processes.



GLOBAL STATE: $\sum$ Money = £235

[Distributed Snapshots] Global State

- Idea: global states are described by
  - the states of the participating PROCESSES, together with
  - the states of the CHANNELS through which data (i.e., the files) pass when being transferred between these processes.



GLOBAL STATE: $\sum$ Money = £235

[Distributed Snapshots] Assumptions

- The algorithm relies on two main assumptions:
    - Channels are ERROR-FREE and SEQUENCE PRESERVING (FIFO)
    - Channels deliver transmitted msgs after UNKNOWN BUT FINITE DELAY
- Other assumptions:
    - The only events in the system which can give rise to changes in the state are communicating events.
    - Simultaneous events are assumed **not** to occur, i.e., THE BEHAVIOR OF A DISTRIBUTED SYSTEM IS DESCRIBED BY A SEQUENCE WITH A TOTAL ORDERING OF ALL EVENTS.

[Distributed Snapshots] Events

- Each event is described by 5 components: $e = \langle p, s, s', M, c \rangle$
  - Process p goes from state s to state s'
  - Message M is sent or received on channel c

[Distributed Snapshots] Events

- Each event is described by 5 components: $e = \langle p, s, s', M, c \rangle$
  - Process p goes from state s to state s'
  - Message M is sent or received on channel c
- Event $e = \langle p, s, s', M, c \rangle$ is only possible in global state S if:

  1. p's state in S is just exactly s.
  2. If c is directed towards p, then c's state in S must be a sequence of messages with M at its head.

[Distributed Snapshots] Events

- Each event is described by 5 components: $e = <p, s, s', M, c>$
    - Process p goes from state s to state s'
    - Message M is sent or received on channel c
- Event $e = <p, s, s', M, c>$ is only possible in global state S if:
    1. p's state in S is just exactly s.
    2. If c is directed towards p, then c's state in S must be a sequence of messages with M at its head.
- A *possible computation* of the system is a *sequence of possible events*, starting from the *initial global state* of the system.

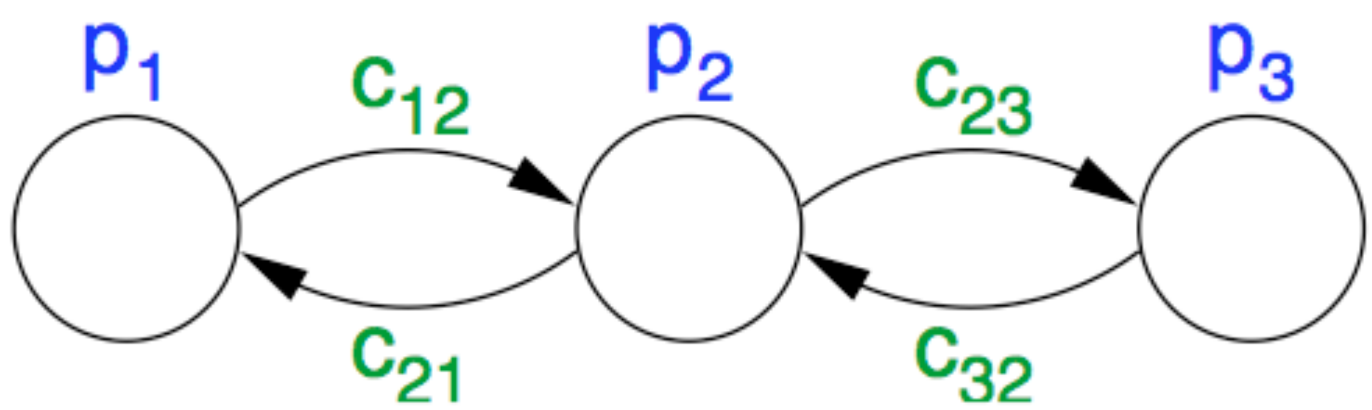
[Distributed Snapshots] Next Global State

- If *e* = <*p*, *s*, *s'*, *M*, *c*> takes place in global state *S*, then the following global state is *next*(*S*, *e*), where:

  1. *p*'s state in *next*(*S*, *e*) is *s'*
  2. If *c* is directed towards *p*, then *c*'s state in *next*(*S*, *e*) is *c*'s state in *S*, with *M* removed from the head of the message sequence
  3. If *c* is directed away from *p*, then *c*'s state in *next*(*S*, *e*) is *c*'s state in *S*, with *M* added to the tail of the message sequence.

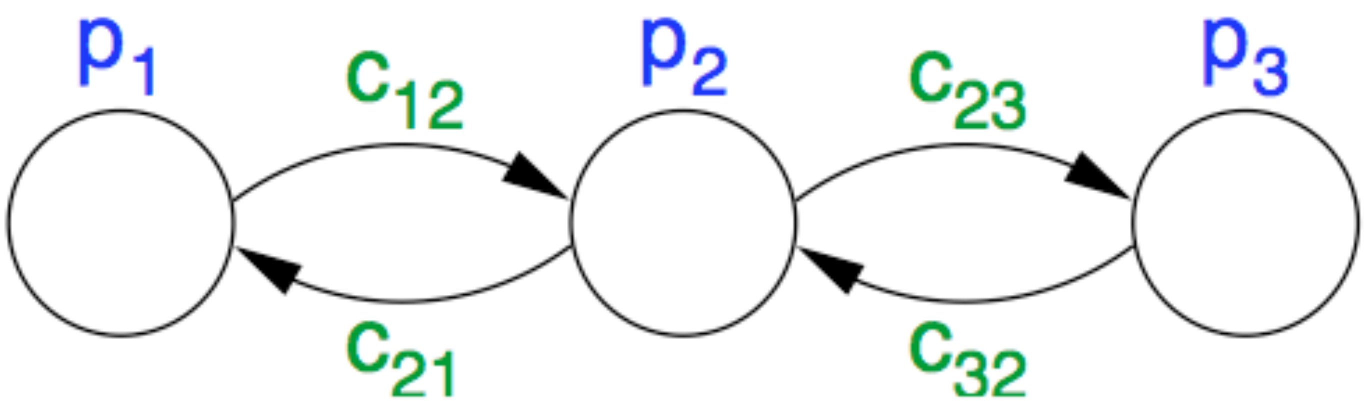
Example: A Possible Computation

- c_{ij} denotes the channel which can carry messages from p_i to p_j.
- System:



	Event						Global state S after event						
	$<P$	s	s'	M	$c>$	$\Rightarrow$	$<P_1$	P_2	P_3	c_{12}	c_{21}	c_{23}	$c_{32}>$
							<100	125	10	$<>$	$<>$	$<>$	$<>>$
e_1	$<P_1$	100	25	75	$c_{12}>$	$\Rightarrow$	<25	125	10	$\langle 75\rangle$	$<>$	$<>$	$<>>$
e_2	$<P_2$	125	100	25	$c_{23}>$	$\Rightarrow$	<25	100	10	$\langle 75\rangle$	$<>$	$\langle 25\rangle$	$<>>$
e_3	$<P_2$	100	175	75	$c_{12}>$	$\Rightarrow$	<25	175	10	$<>$	$<>$	$\langle 25\rangle$	$<>>$
e_4	$<P_2$	175	125	50	$c_{21}>$	$\Rightarrow$	<25	125	10	$<>$	$\langle 50\rangle$	$\langle 25\rangle$	$<>>$
e_5	$<P_3$	10	35	25	$c_{23}>$	$\Rightarrow$	<25	125	35	$<>$	$\langle 50\rangle$	$<>$	$<>>$
e_6	$<P_1$	25	75	50	$c_{21}>$	$\Rightarrow$	<75	125	35	$<>$	$<>$	$<>$	$<>>$

[Distributed Snapshots] The Question



Can we now find rules for
when to take snapshots of
the individual processes and channels
so as to build up a consistent picture of
the global state S?

[Distributed Snapshots] Consistent Picture

- Let us consider the happened before relation.

- If $e_1 \rightarrow e_2$ then e_1 happened before e_2 and could have caused it.

- A consistent picture of the global state is obtained if we include in our computation a set of possible events, H, such that

$$e_i \in H \wedge e_j \rightarrow e_i \Rightarrow e_j \in H$$

- If e_1 were in H, but e_i were not, then the set of events would include the effect of an event (for instance, the receipt of a file), but not the event causing it (the sending of the file), and an inconsistent picture would arise.

[Distributed Snapshots] Consistent Global State

- A consistent picture of the global state is obtained if we include in our computation a set of possible events, H, such that

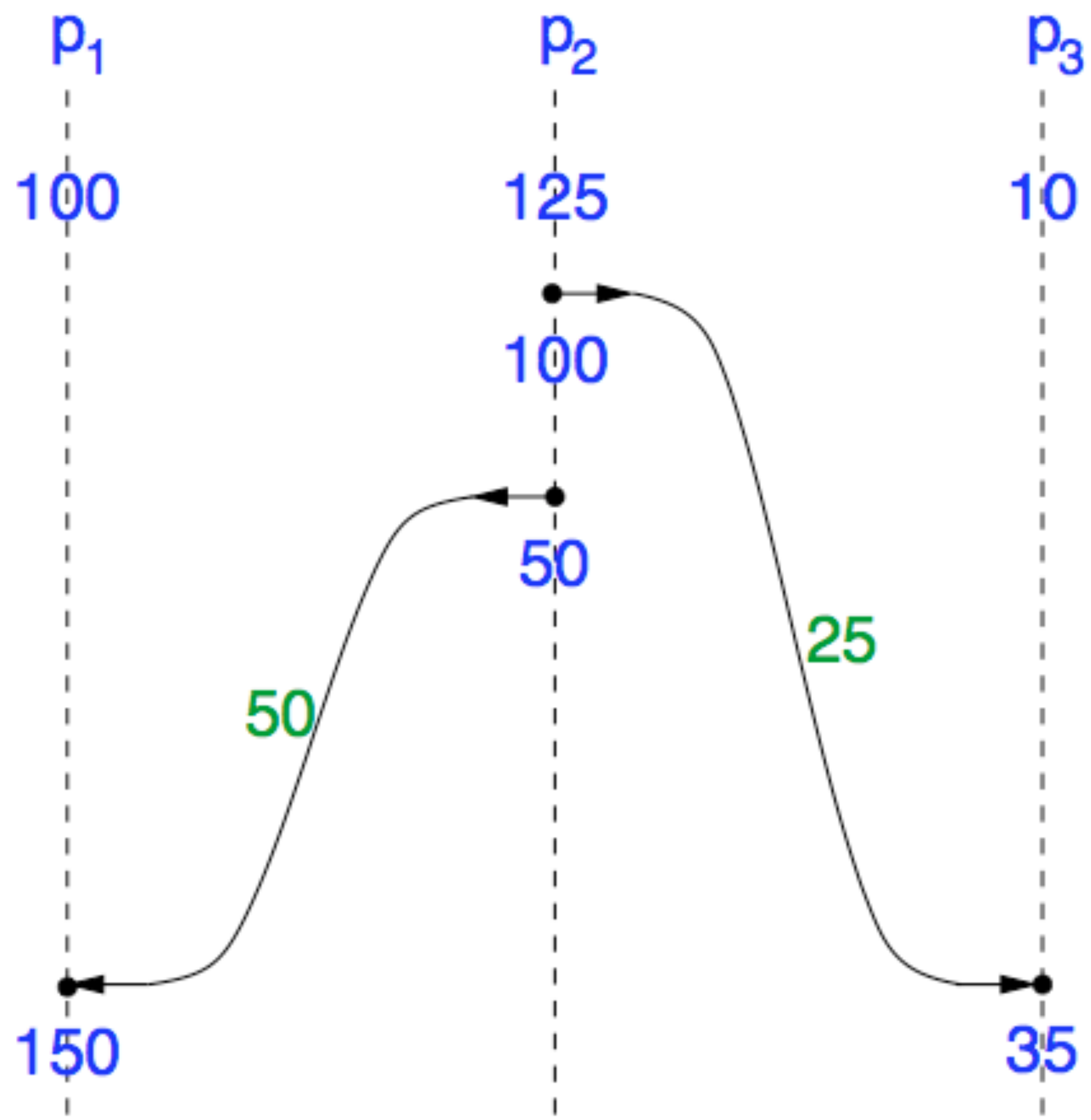
$$e_i \in H \wedge e_j \rightarrow e_i \Rightarrow e_j \in H$$

- The consistent GLOBAL STATE is then defined by

  $GS(H)$ = The state of each process p_i after p_i's last event in H
  + for each channel, the sequence of msgs sent in H but not received in H.

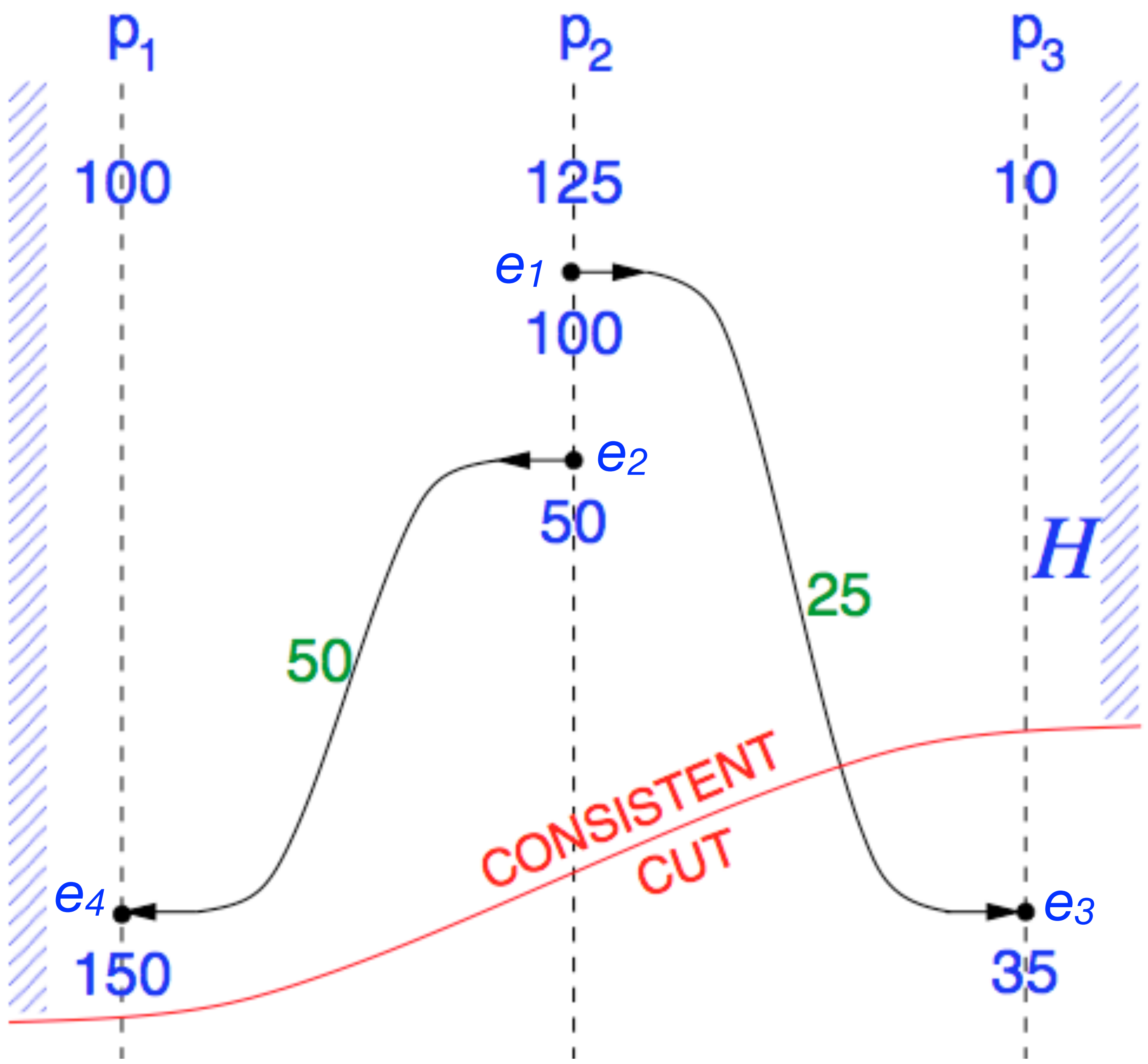
- In the distributed systems jargon, we say that *consistent global states are delimited by a "CUT" representing a consistent picture of the global state of the system*.

Example: A Possible Computation



Example: Consistent Cut

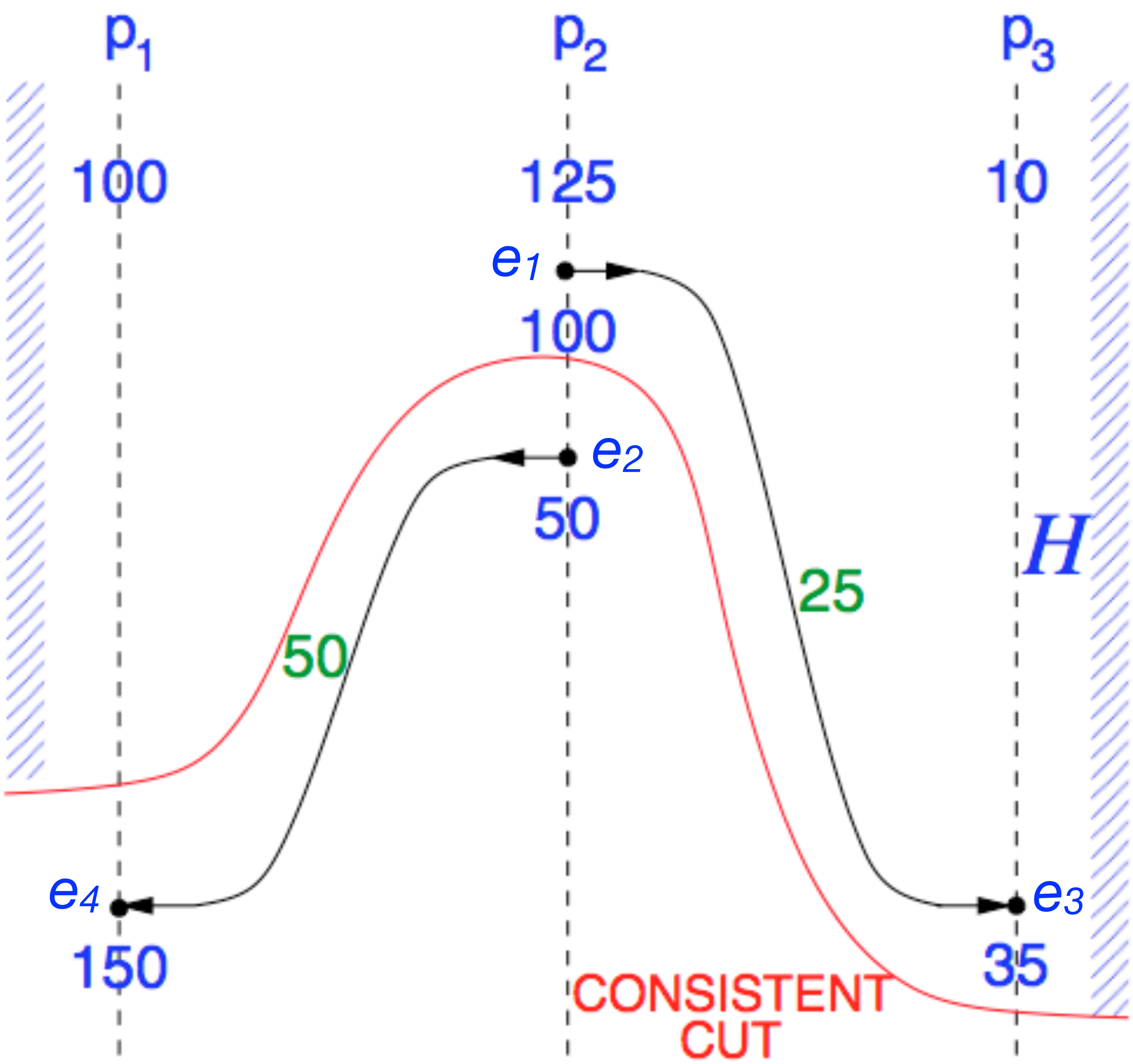
- REMEMBER: The CUT limiting H is defined by: $e_i \in H \wedge e_j \rightarrow e_i \Rightarrow e_j \in H$



H contains $\{e_1, e_2, e_3\}$

Example: Consistent Cut

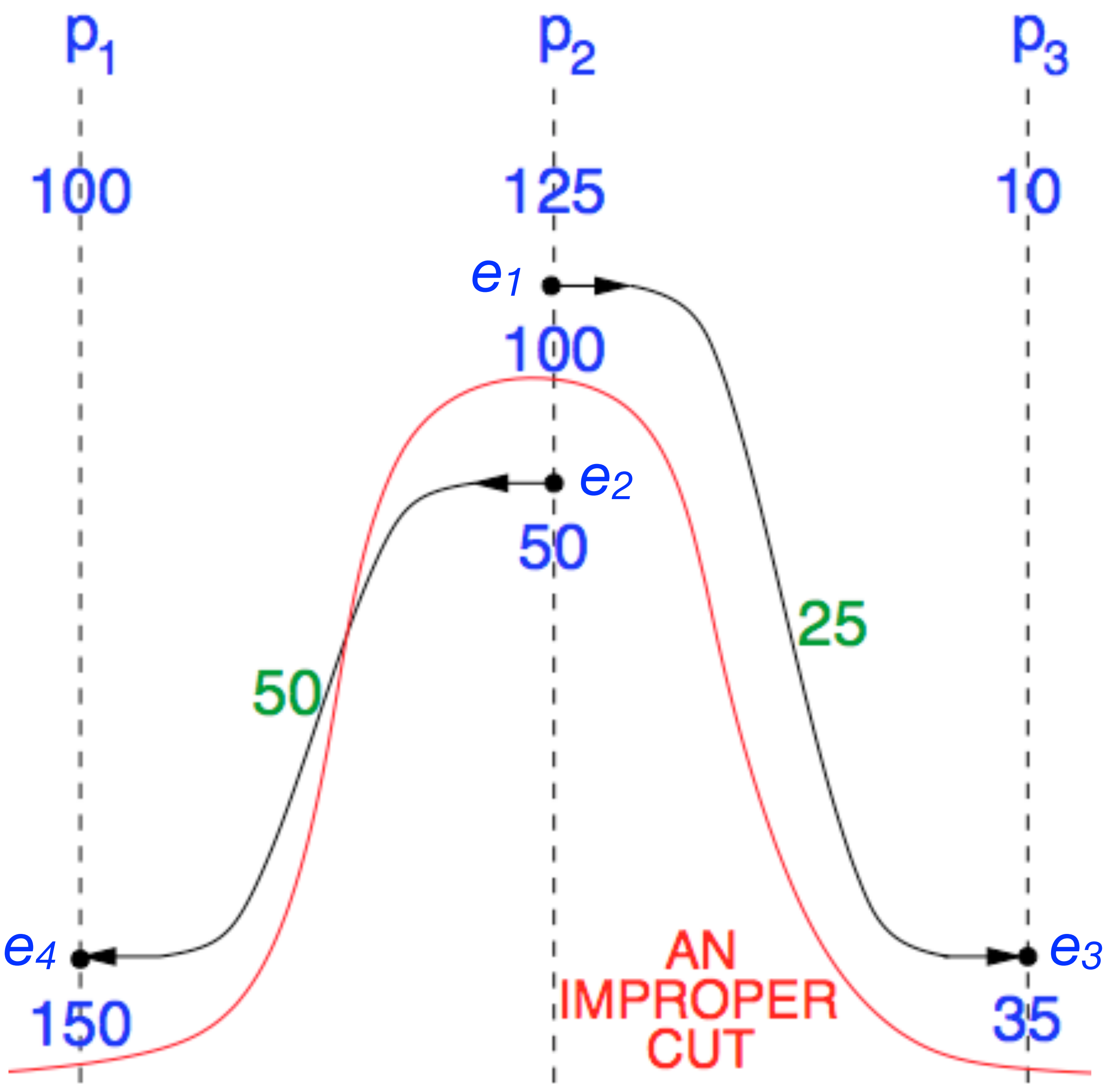
- REMEMBER: The CUT limiting H is defined by: $e_i \in H \wedge e_j \rightarrow e_i \Rightarrow e_j \in H$



H contains $\{e_1, e_3\}$

Example: Inconsistent Cut

- REMEMBER: The CUT limiting H is defined by: $e_i \in H \wedge e_j \rightarrow e_i \Rightarrow e_j \in H$



H contains $\{e_1, e_3, e_4\}$, but not e_2, where $e_2 \rightarrow e_4$

How to Construct *H*?

- Idea: The CUT and associated (consistent) set of events, *H*, are constructed by including specific control messages (MARKERS) in the stream of ordinary messages.

- Remember that we assume that:
  - Channels are all FIFO channels.
  - A transmitted marker will be received (and dealt with) within a FINITE TIME.

Chandy and Lamport's Algorithm to Construct *H*

- Process p_i follows two rules.

- SEND MARKERS

  Record p_i's state
  Before sending any more messages from p_i, send a marker on each channel c_{ij} directed away from p_i.

Chandy and Lamport's Algorithm to Construct *H*

- Process p_i follows two rules.

- SEND MARKERS

  Record p_i's state
  Before sending any more messages from p_i, send a marker on each channel c_{ij} directed away from p_i.

- RECEIVE MARKER

  On arrival of a marker via channel c_{ji}:

  IF p_i has not recorded its state

  THEN SEND MARKERS rule; record c_{ji}'s state as empty

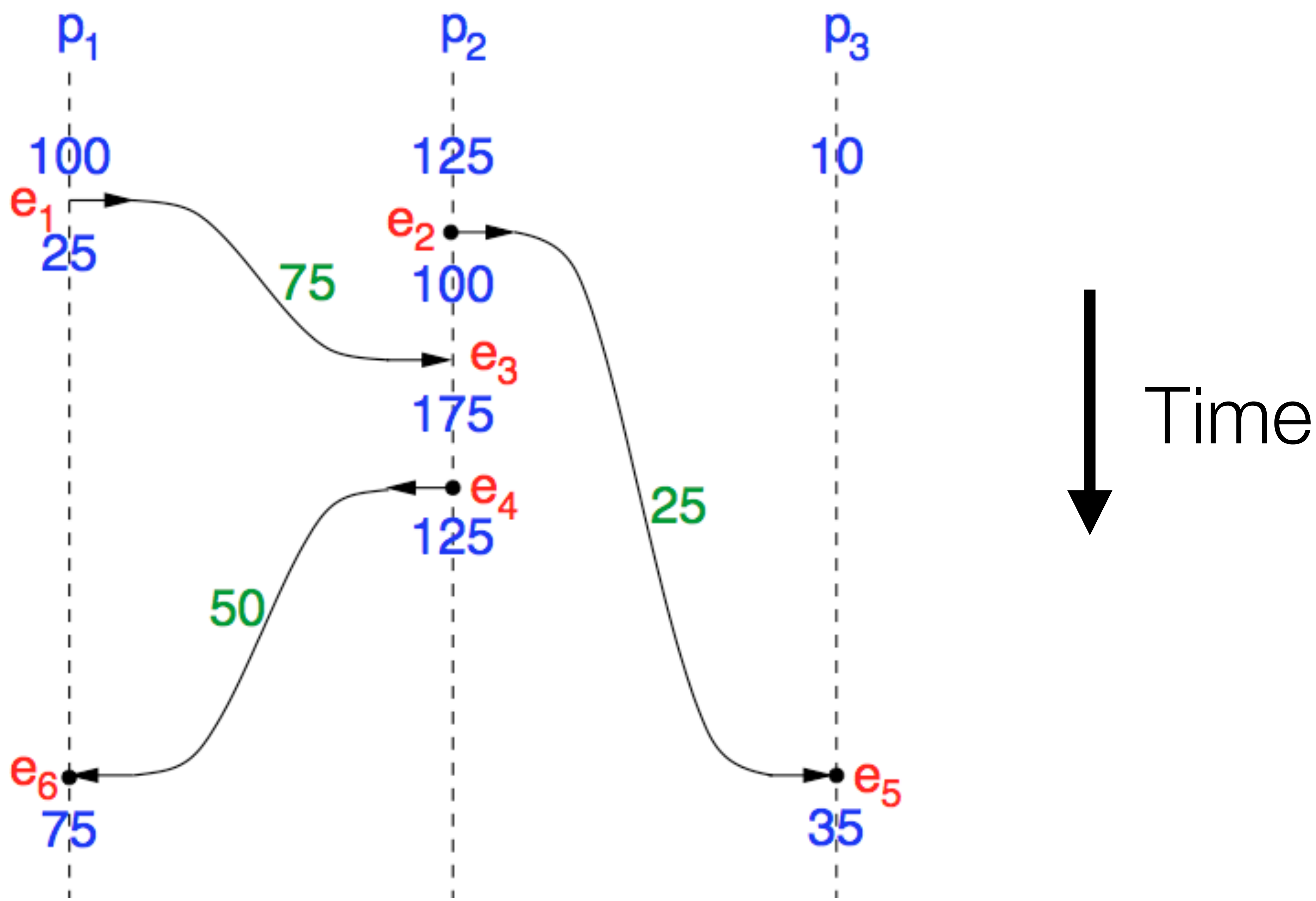
  ELSE record c_{ji}'s state as the sequence of messages received on c_{ji} since p_i last noted its state.

Chandy and Lamport's Algorithm to Construct *H*

- The algorithm can be initiated by any process by executing the rule SEND MARKERS.
    - Multiple processes can initiate the algorithm concurrently!
    - Each initiation needs to be distinguished by using unique markers.
    - Different initiations by a process are identified by a sequence number.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- Complexity of the algorithm: $O(E)$ messages, where E is the number of edges in the network.

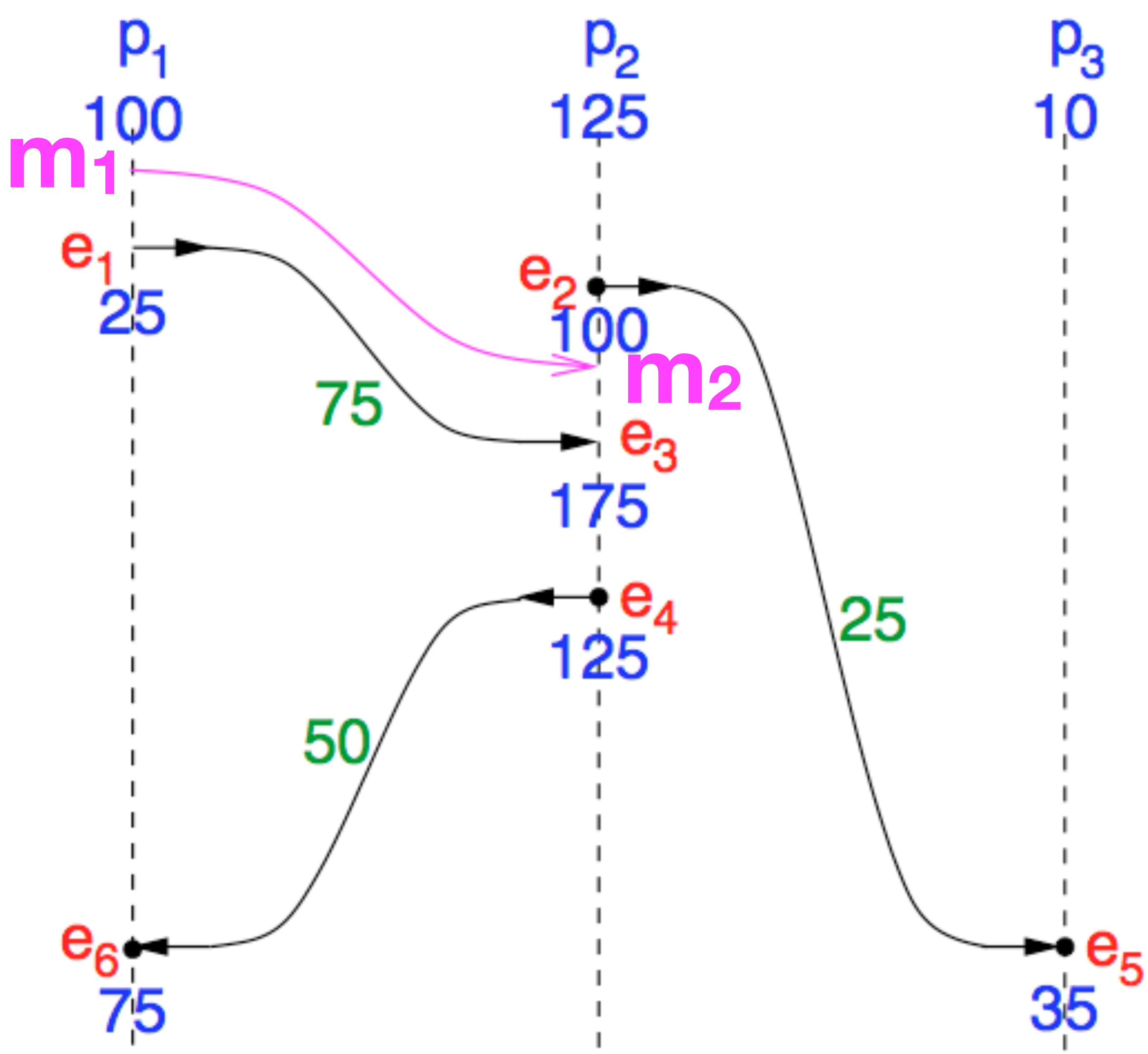
Example: The Algorithm In Action...

The computation



Example: The Algorithm In Action...

p_1 initiates the algorithm

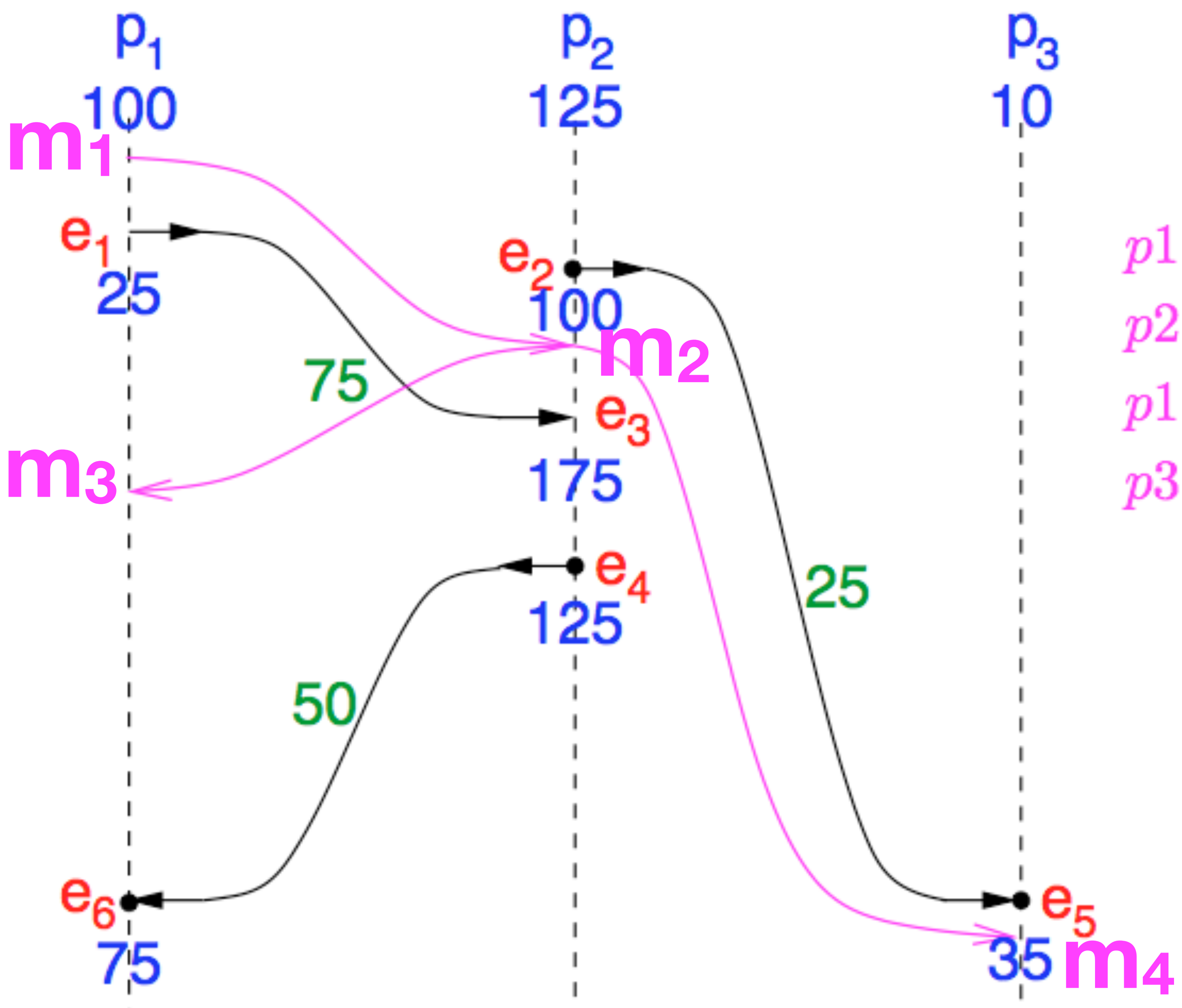


$p1: \quad 100$

$p2: \quad 100 \quad c_{12} = \quad <>$

Example: The Algorithm In Action...

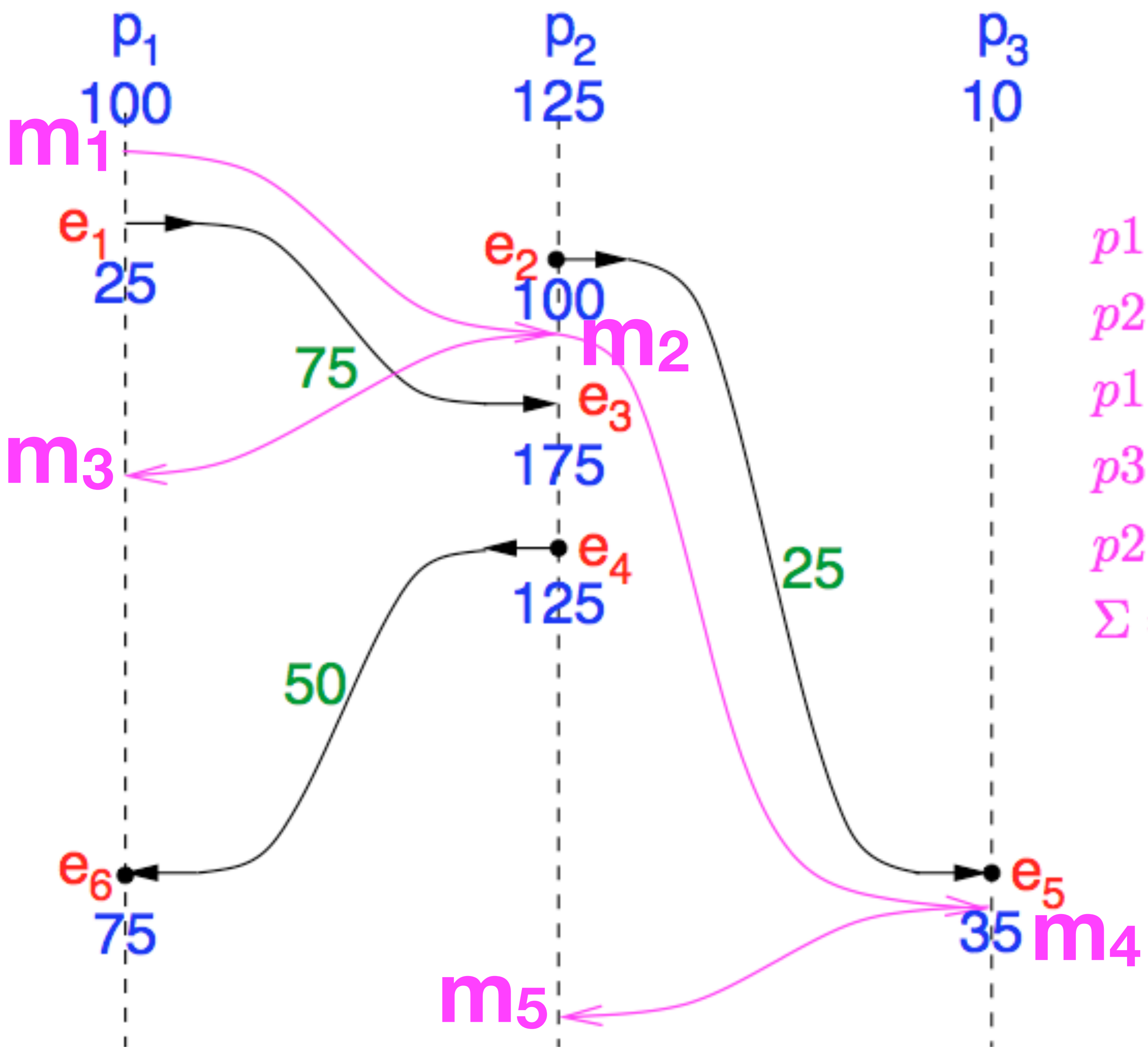
p_1 initiates the algorithm



$$
\begin{array}{lrll}
p1: & 100 & & \\
p2: & 100 & c_{12} = & \langle\rangle \\
p1: & - & c_{21} = & \langle\rangle \\
p3: & 35 & c_{23} = & \langle\rangle
\end{array}
$$

Example: The Algorithm In Action...

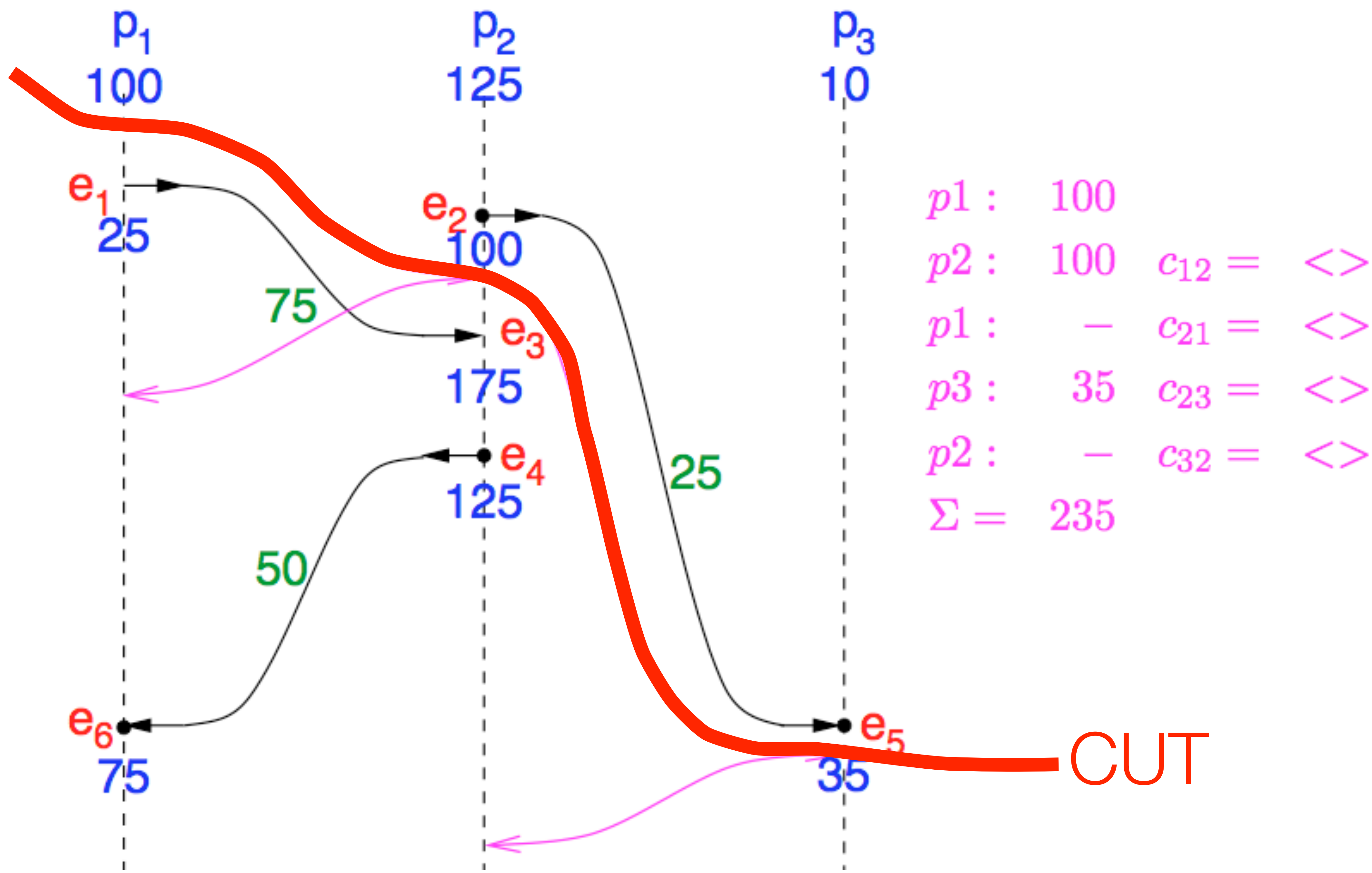
p_1 initiates the algorithm



$p1:$	100			
$p2:$	100	c_{12}	$=$	$<>$
$p1:$	$-$	c_{21}	$=$	$<>$
$p3:$	35	c_{23}	$=$	$<>$
$p2:$	$-$	c_{32}	$=$	$<>$
$\Sigma =$	235			

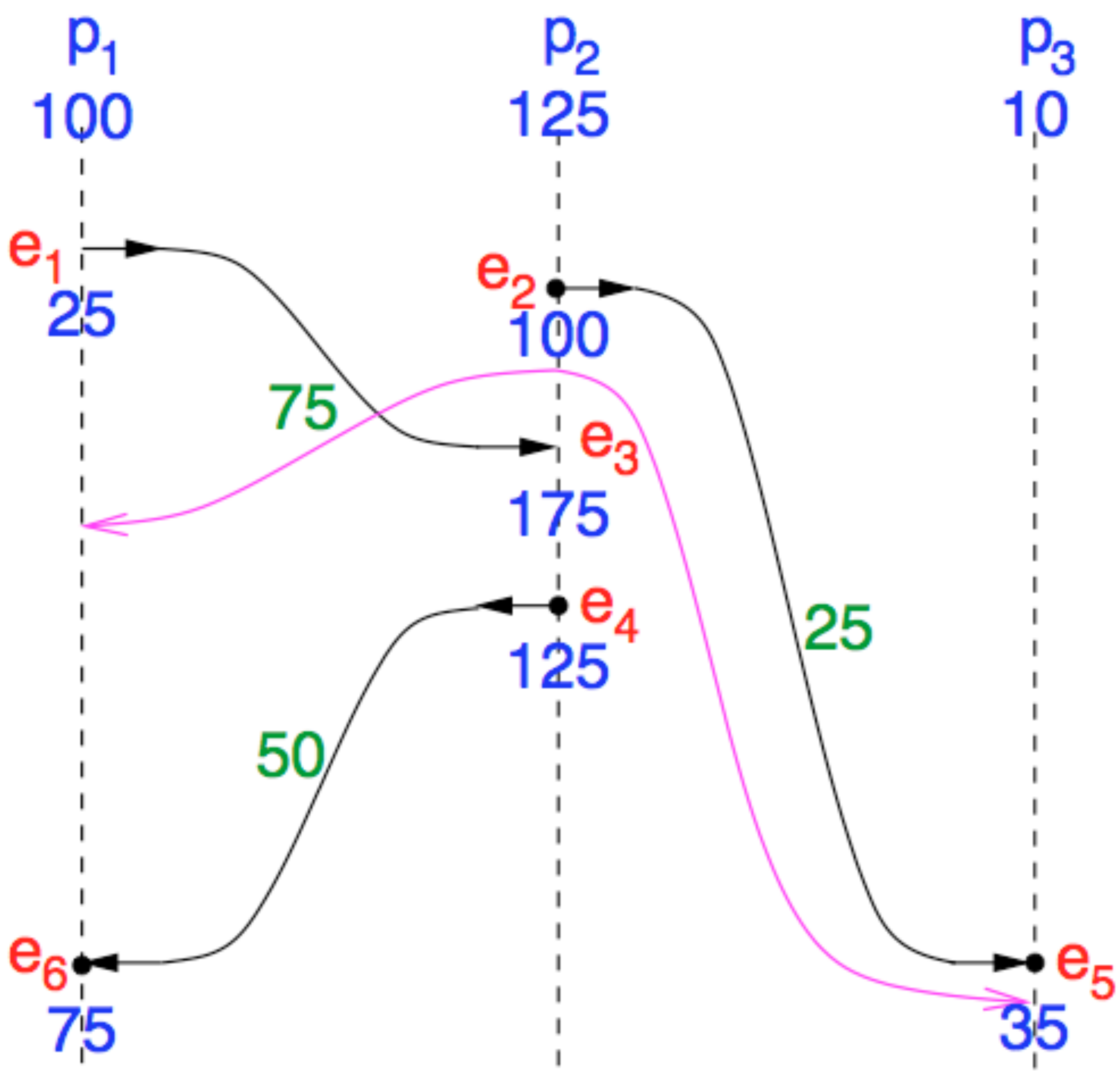
Example: The Algorithm In Action...

p_1 initiates the algorithm



Example: The Algorithm In Action...

*p*2 initiates the algorithm



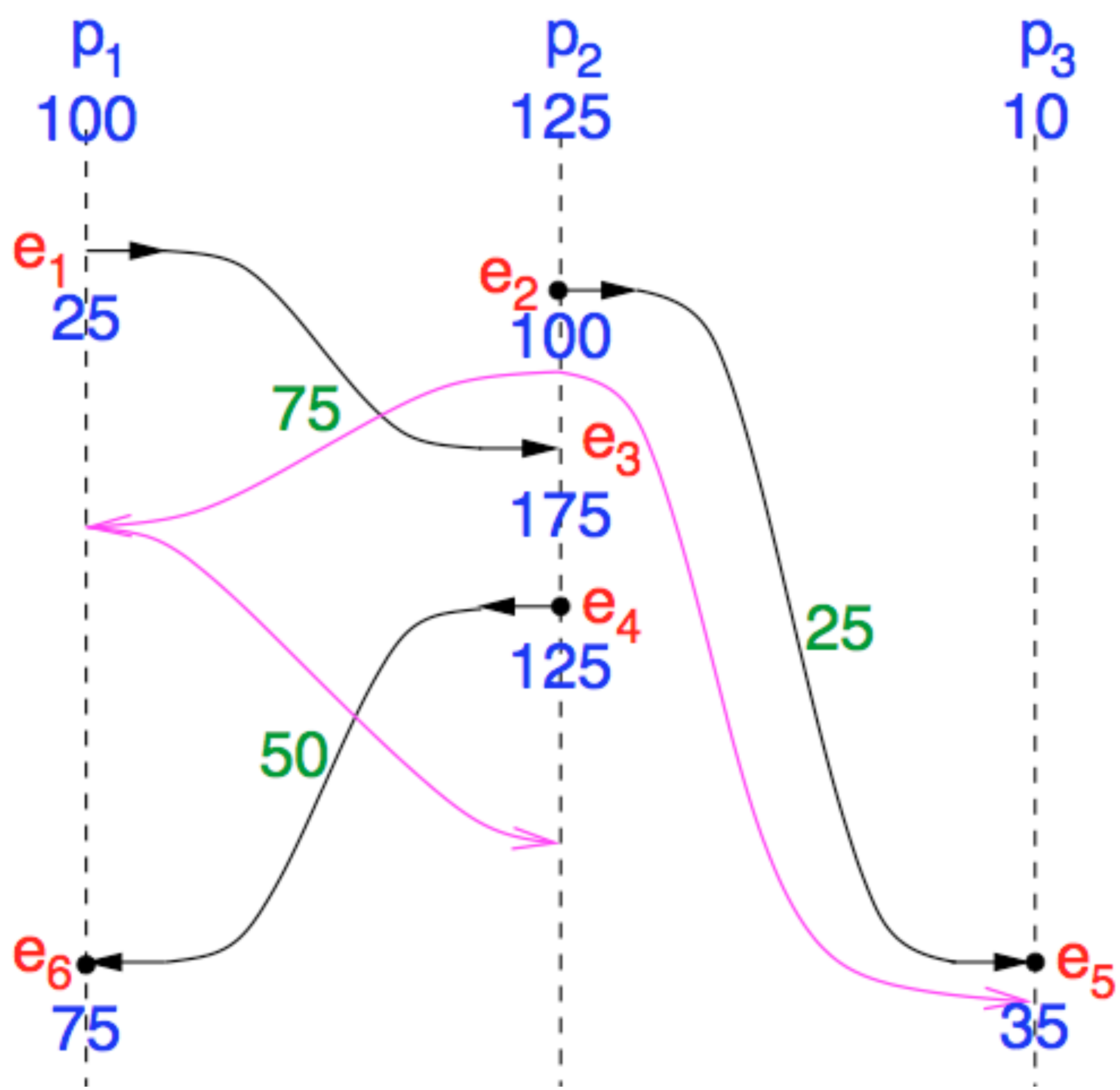
$p2: \quad 100$

$p1: \quad 25 \quad c_{21} = \quad \langle\rangle$

$p3: \quad 35 \quad c_{23} = \quad \langle\rangle$

Example: The Algorithm In Action...

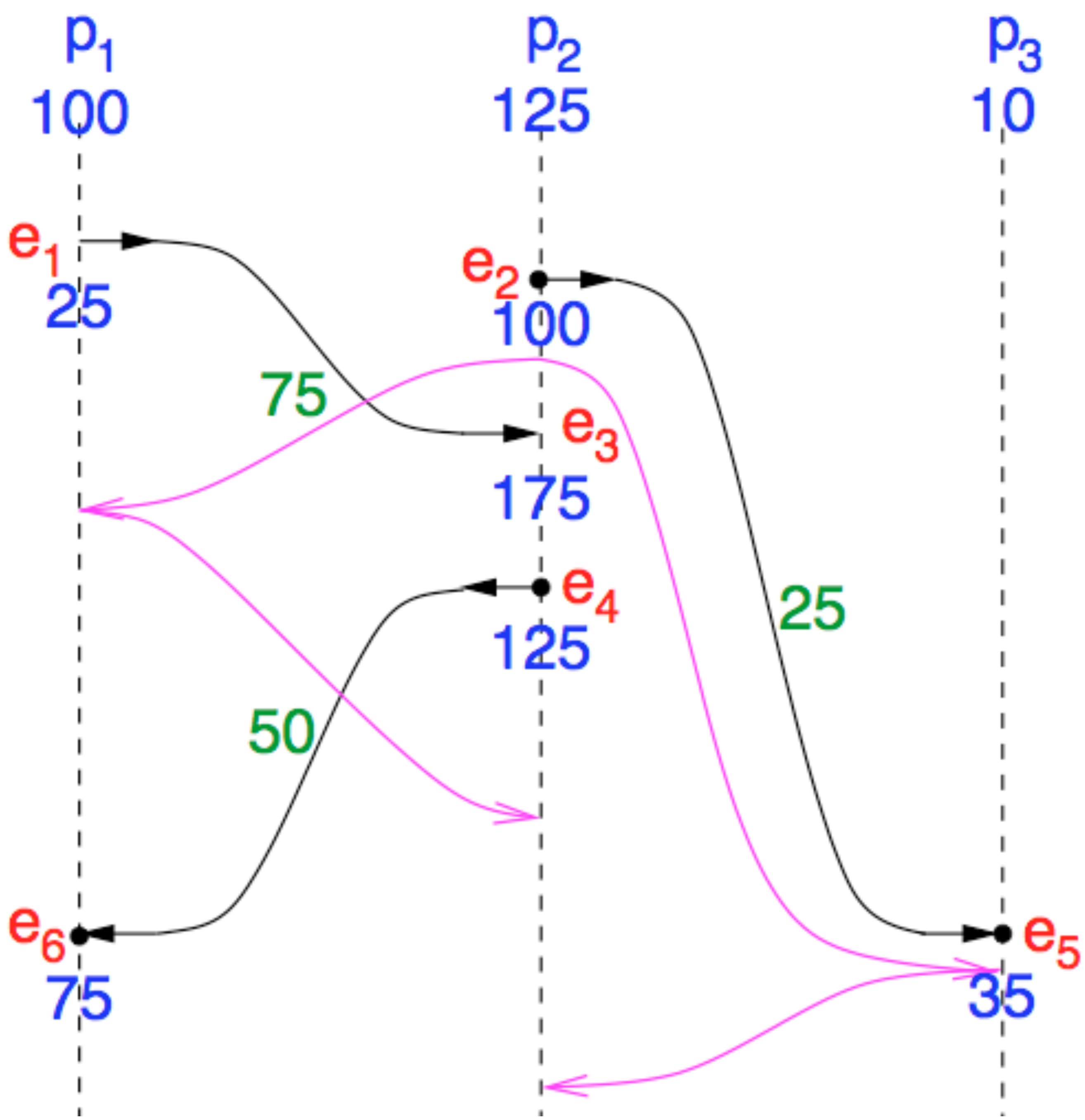
p_2 initiates the algorithm



$$\begin{array}{llll} p2: & 100 & & \\ p1: & 25 & c_{21} = & <> \\ p3: & 35 & c_{23} = & <> \\ p2: & - & c_{12} = & \langle 75 \rangle \end{array}$$

Example: The Algorithm In Action...

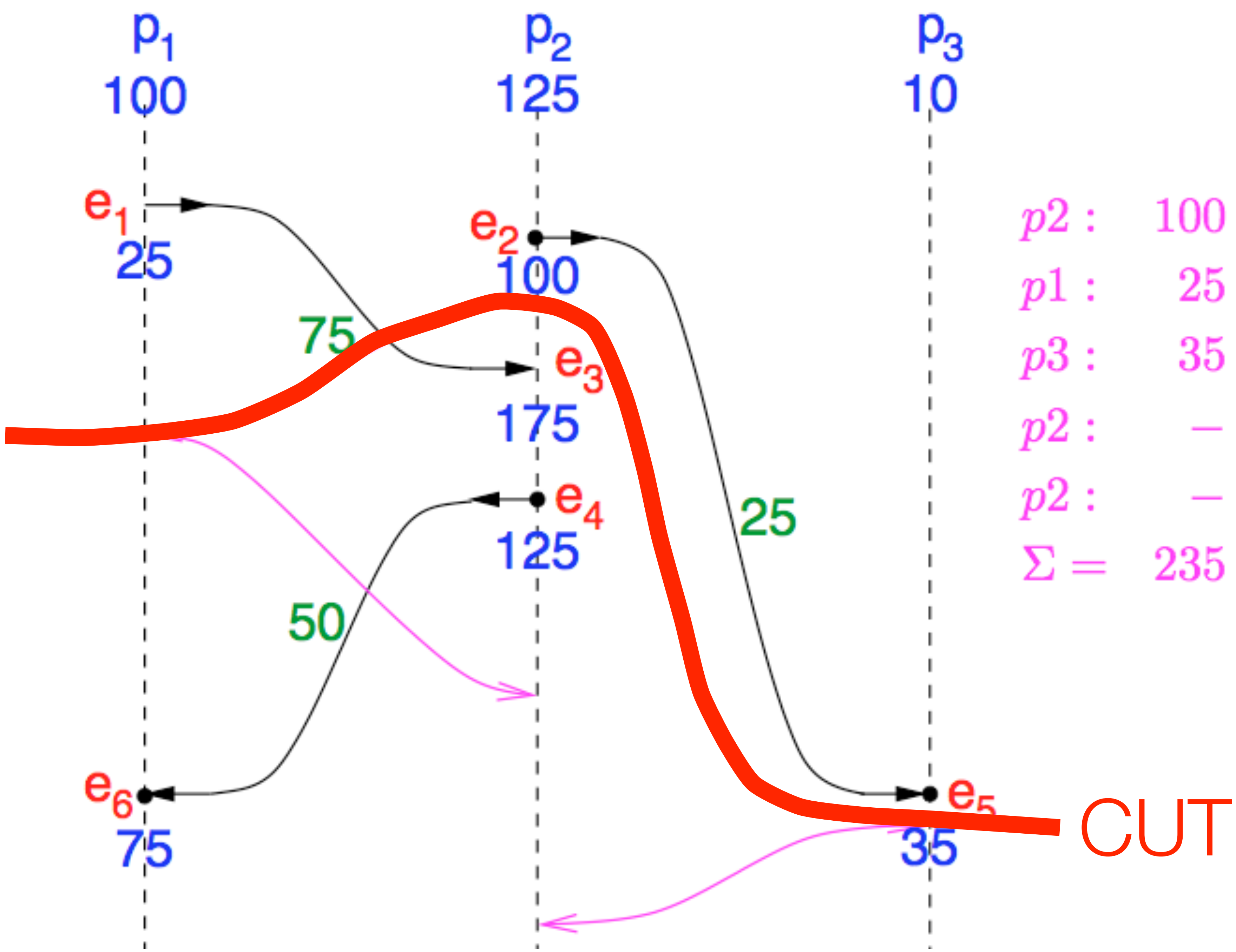
p_2 initiates the algorithm



$$
\begin{array}{llll}
p2: & 100 & & \\
p1: & 25 & c_{21} = & <> \\
p3: & 35 & c_{23} = & <> \\
p2: & - & c_{12} = & \langle 75 \rangle \\
p2: & - & c_{32} = & <> \\
\Sigma = & 235 & &
\end{array}
$$

Example: The Algorithm In Action...

p_2 initiates the algorithm



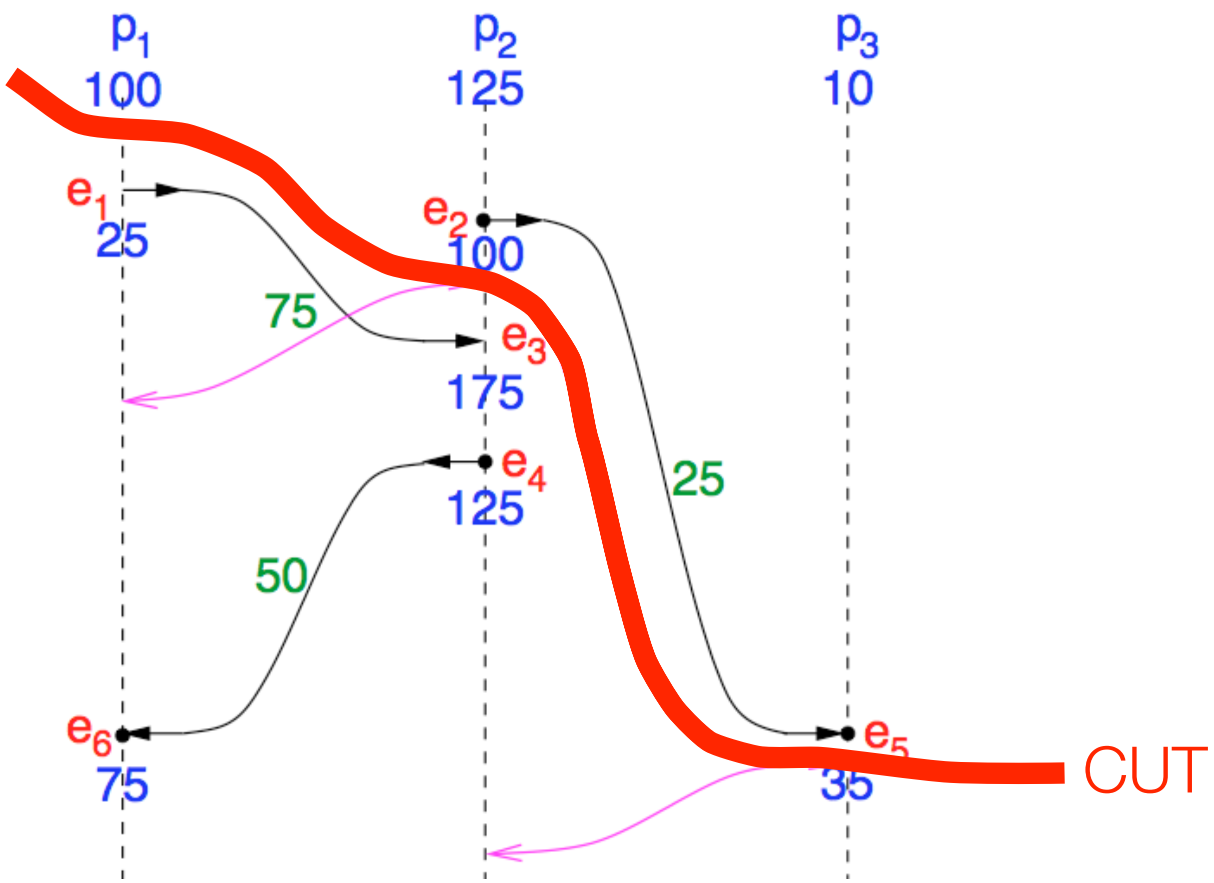
$$
\begin{array}{llll}
p2: & 100 & & \\
p1: & 25 & c_{21} = & <> \\
p3: & 35 & c_{23} = & <> \\
p2: & - & c_{12} = & \langle 75 \rangle \\
p2: & - & c_{32} = & <> \\
\Sigma = & 235 & &
\end{array}
$$

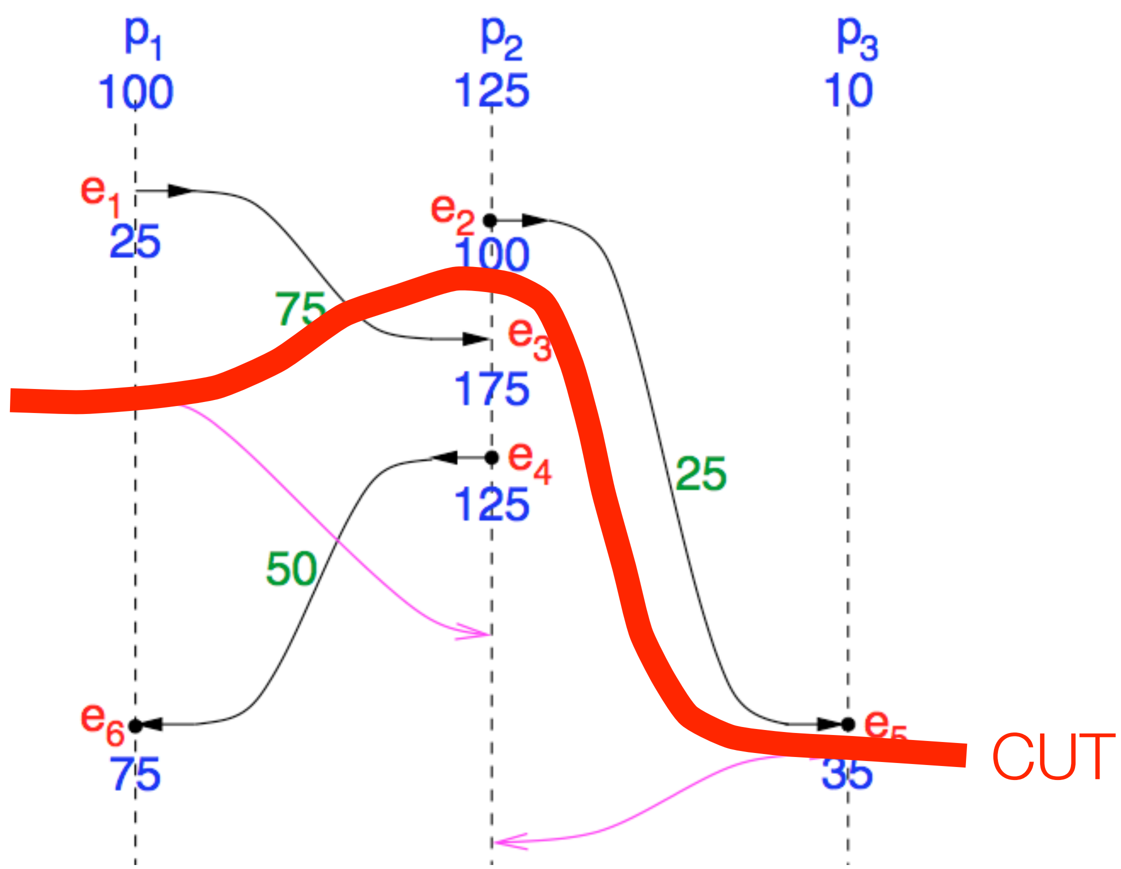
How the Global Snapshot is Then Created?

- In a practical implementation, the recorded local snapshots must be put together to create a global snapshot of the distributed system.

- How? Several policies:
  - each process sends its local snapshot to the initiator of the algorithm
  - each process sends the information it records along all outgoing channels and each process receiving such information for the first time propagates it along its outgoing channels

How is That Possible?





$$
\begin{array}{lll}
p1: & 100 & \\
p2: & 100 & c_{12} = \ <> \\
p1: & - & c_{21} = \ <> \\
p3: & 35 & c_{23} = \ <> \\
p2: & - & c_{32} = \ <> \\
\Sigma = & 235 &
\end{array}
$$

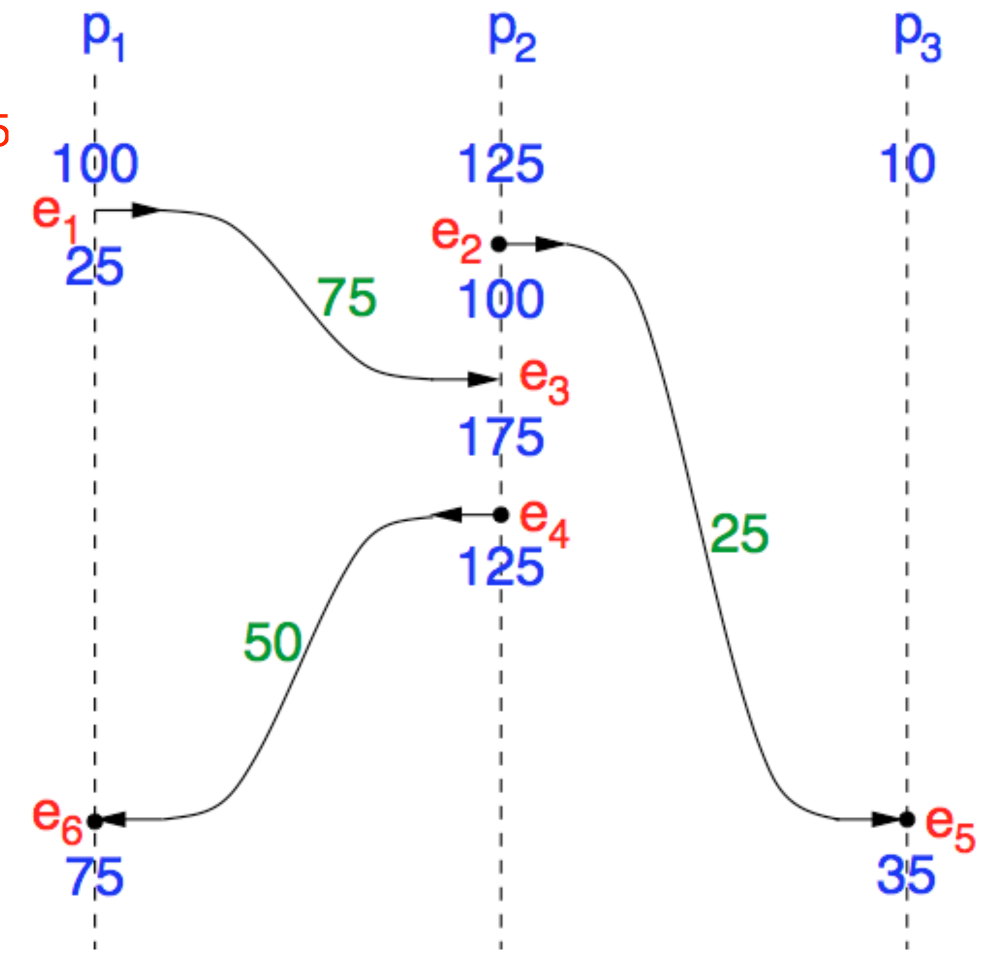
In both these possible runs of the algorithm, the recorded global states **NEVER** occurred in the execution!

$$
\begin{array}{lll}
p2: & 100 & \\
p1: & 25 & c_{21} = \ <> \\
p3: & 35 & c_{23} = \ <> \\
p2: & - & c_{12} = \langle 75 \rangle \\
p2: & - & c_{32} = \ <> \\
\Sigma = & 235 &
\end{array}
$$

Incomparable Events!

- The algorithm finds a global state based on a *partial ordering* → of events.

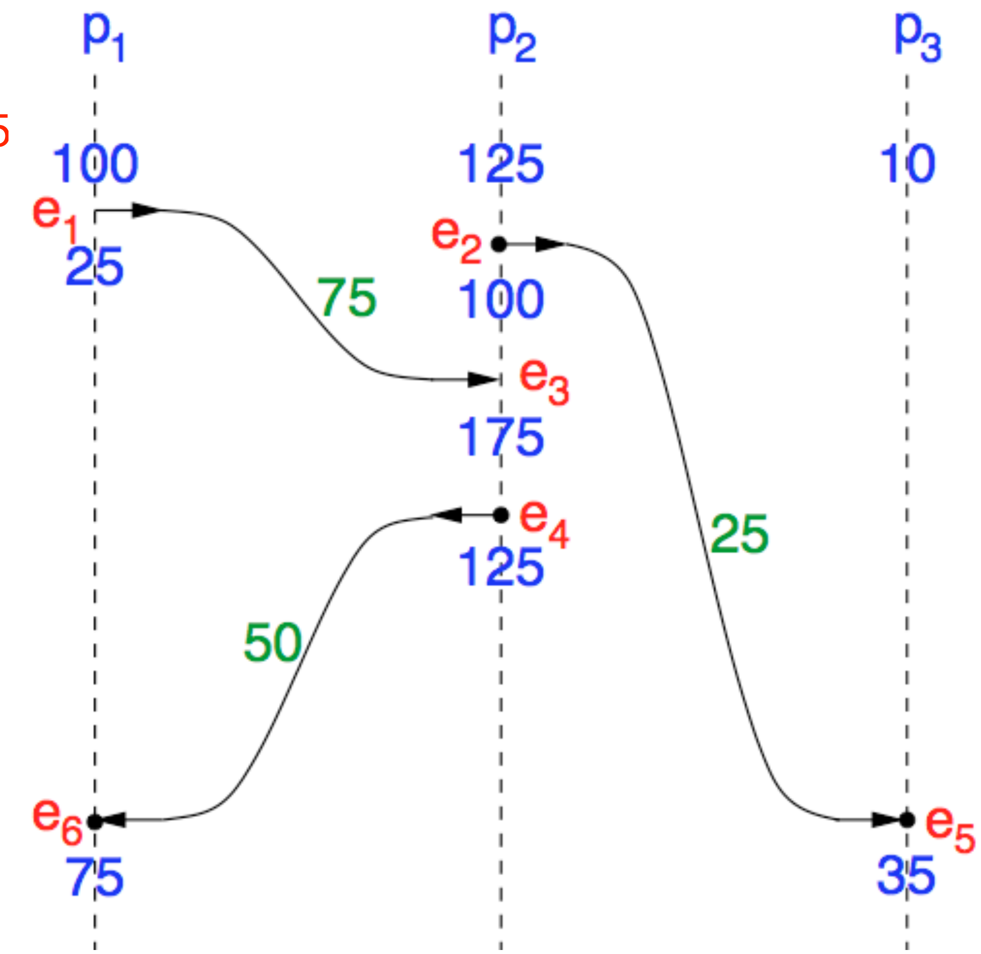
For instance, we know that $e_2 \rightarrow e_3$ and $e_2 \rightarrow e_5$ BUT we have no knowledge about the timing relationship of e_3 and e_5.
WIth respect to →, e_3 and e_5 are *incomparable*!



Incomparable Events!

- The algorithm finds a global state based on a *partial ordering* → of events.

For instance, we know that $e_2 \rightarrow e_3$ and $e_2 \rightarrow e_5$ BUT we have no knowledge about the timing relationship of e_3 and e_5.
WIth respect to →, e_3 and e_5 are *incomparable*!

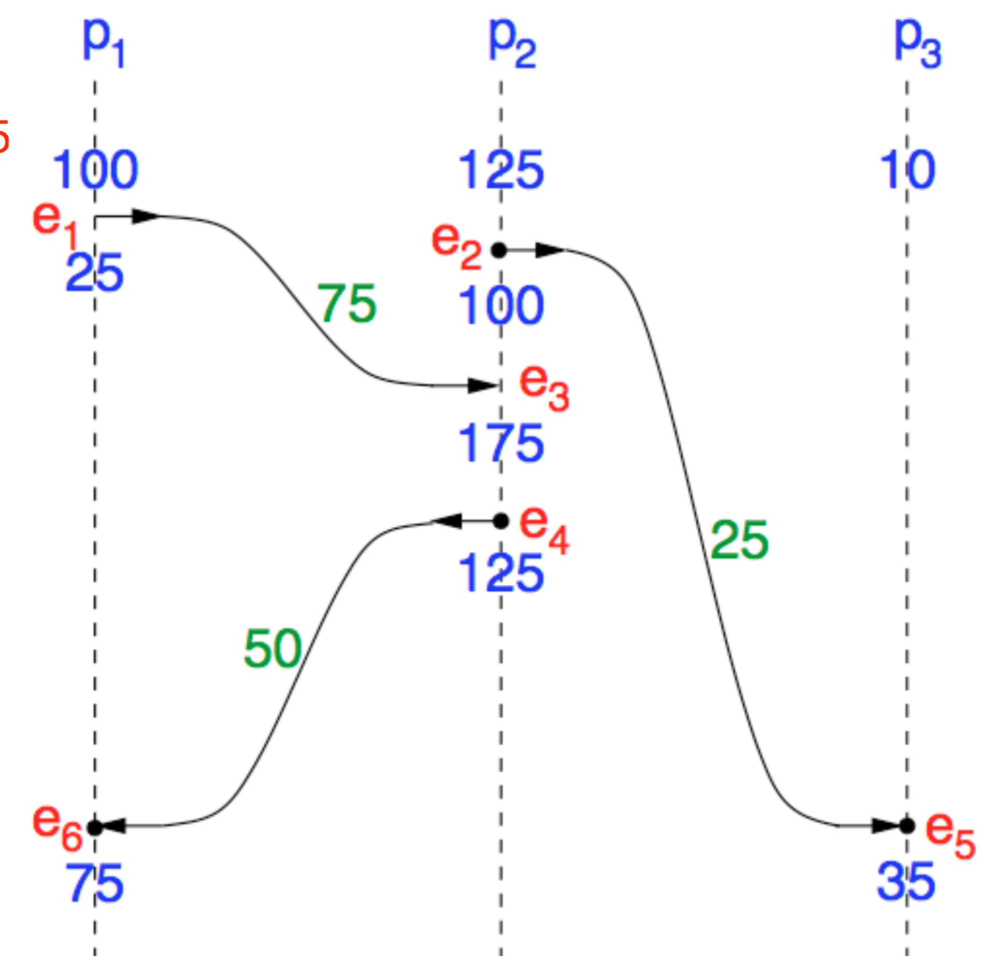


We cannot determine what the *true sequence* of these events is!

Incomparable Events!

- The algorithm finds a global state based on a *partial ordering* → of events.

For instance, we know that $e_2 \to e_3$ and $e_2 \to e_5$ BUT we have no knowledge about the timing relationship of e_3 and e_5.
WIth respect to →, e_3 and e_5 are *incomparable*!



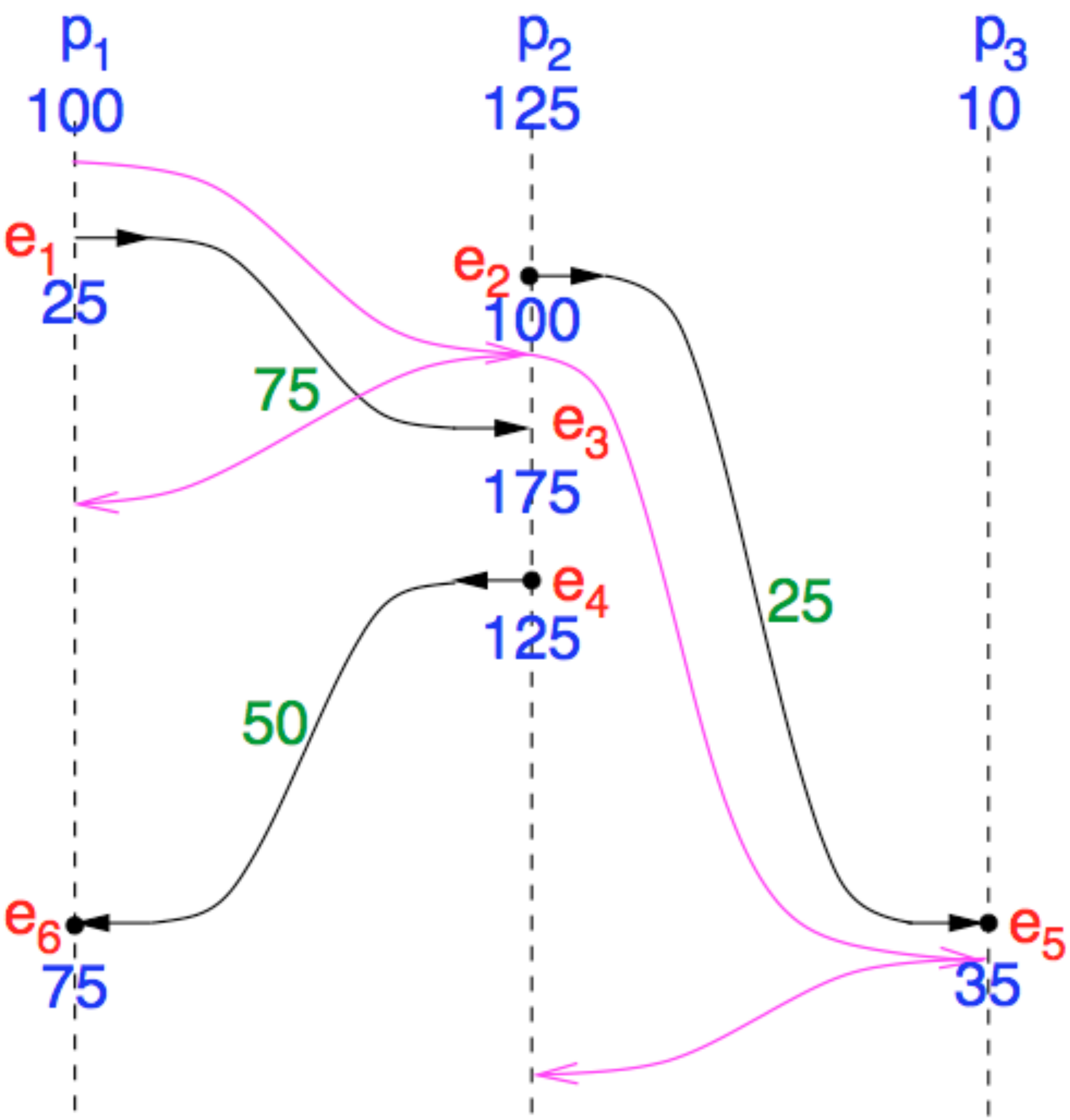
We cannot determine what the *true sequence* of these events is!

- *When we record a process' state, we are unable to know whether the events which we have already seen in this process lay before or after incomparable events in other processes.*

What Does the Algorithm Find?

- Pre-recording events: events in a computation which take place BEFORE the process in which they occur records its own state.
- Post-recording events: all other events.
- *The algorithm finds a global state which corresponds to a PERMUTATION of the actual order of the events, such that all pre-recording events come before all post-recording events.*
- The recorded global state, S*, is the one which would be found after all the pre-recording events and before all the post-recording events.

Example



$p1:$	100		
$p2:$	100	$c_{12} =$	$<>$
$p1:$	$-$	$c_{21} =$	$<>$
$p3:$	35	$c_{23} =$	$<>$
$p2:$	$-$	$c_{32} =$	$<>$
$\Sigma =$	235		

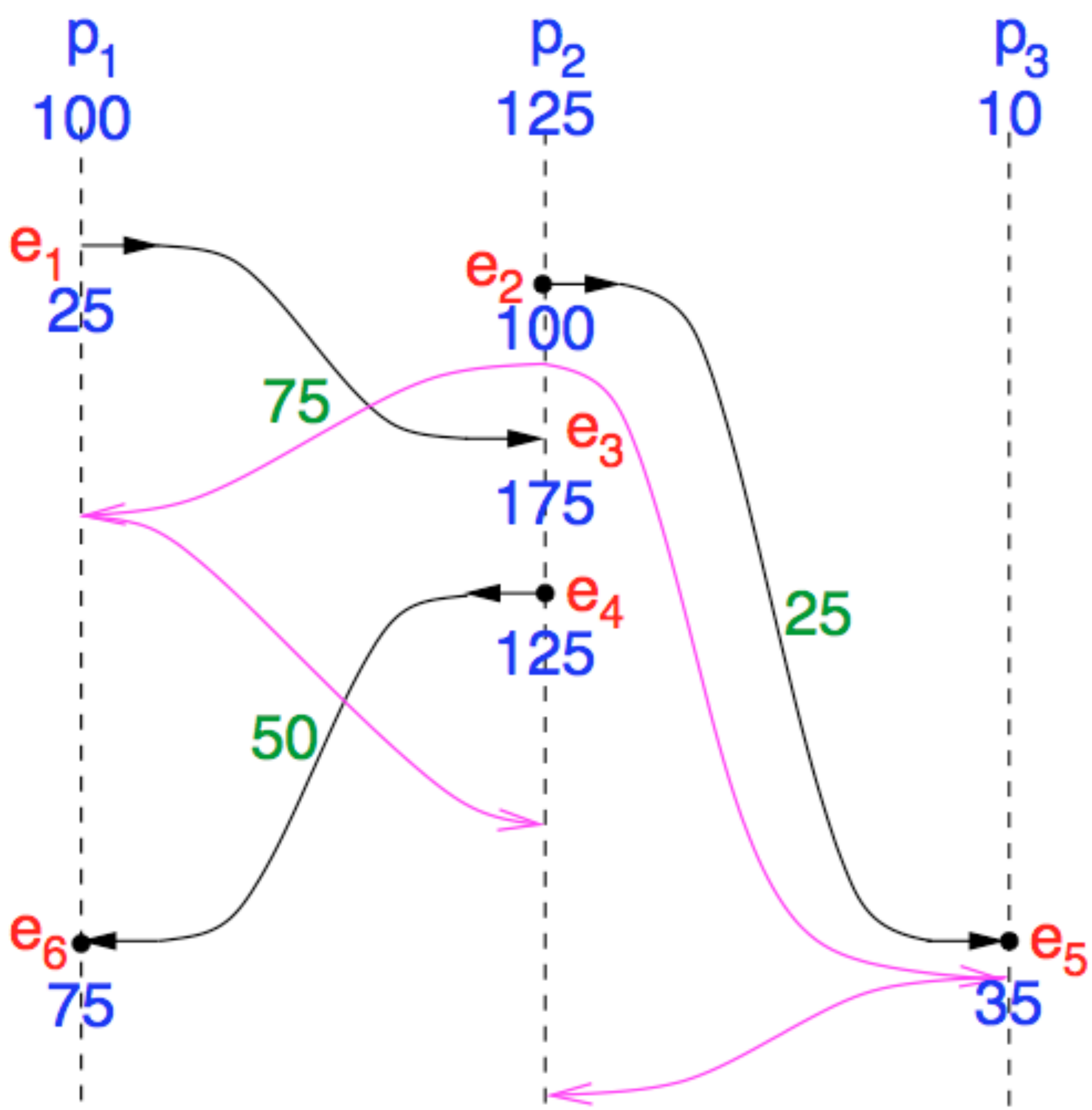
pre-recording events: $\{e_2, e_5\}$

$$seq = \langle e_1, e_2, e_3, e_4, e_5, e_6 \rangle$$

$$seq' = \langle e_2, e_5 | e_1, e_3, e_4, e_6 \rangle$$

S^* recorded global state

Example



$p2:$	100			
$p1:$	25	c_{21}	$=$	$<>$
$p3:$	35	c_{23}	$=$	$<>$
$p2:$	$-$	c_{12}	$=$	$\langle 75 \rangle$
$p2:$	$-$	c_{32}	$=$	$<>$
$\Sigma =$	235			

pre-recording events: {e_1, e_2, e_5}

$$seq = \langle e_1, e_2, e_3, e_4, e_5, e_6 \rangle$$

$$seq' = \langle e_1, e_2, e_5 \mid e_3, e_4, e_6 \rangle$$

S^* recorded global state

Global State Could Possibly Have Occurred!

- S^* is a state which *could possibly have occurred*, in the sense that:
  - It is possible to reach S^* via a sequence of *possible events* starting from the initial state of the system, S_i (in the previous example: <e_1, e_2, e_5>)
  - It is possible to reach the final state of the system, S_f, via a sequence of possible events starting from S^* (in the previous example: <e_3, e_4, e_6>)

$$seq' = \langle e_1, e_2, e_5 \mid e_3, e_4, e_6 \rangle$$

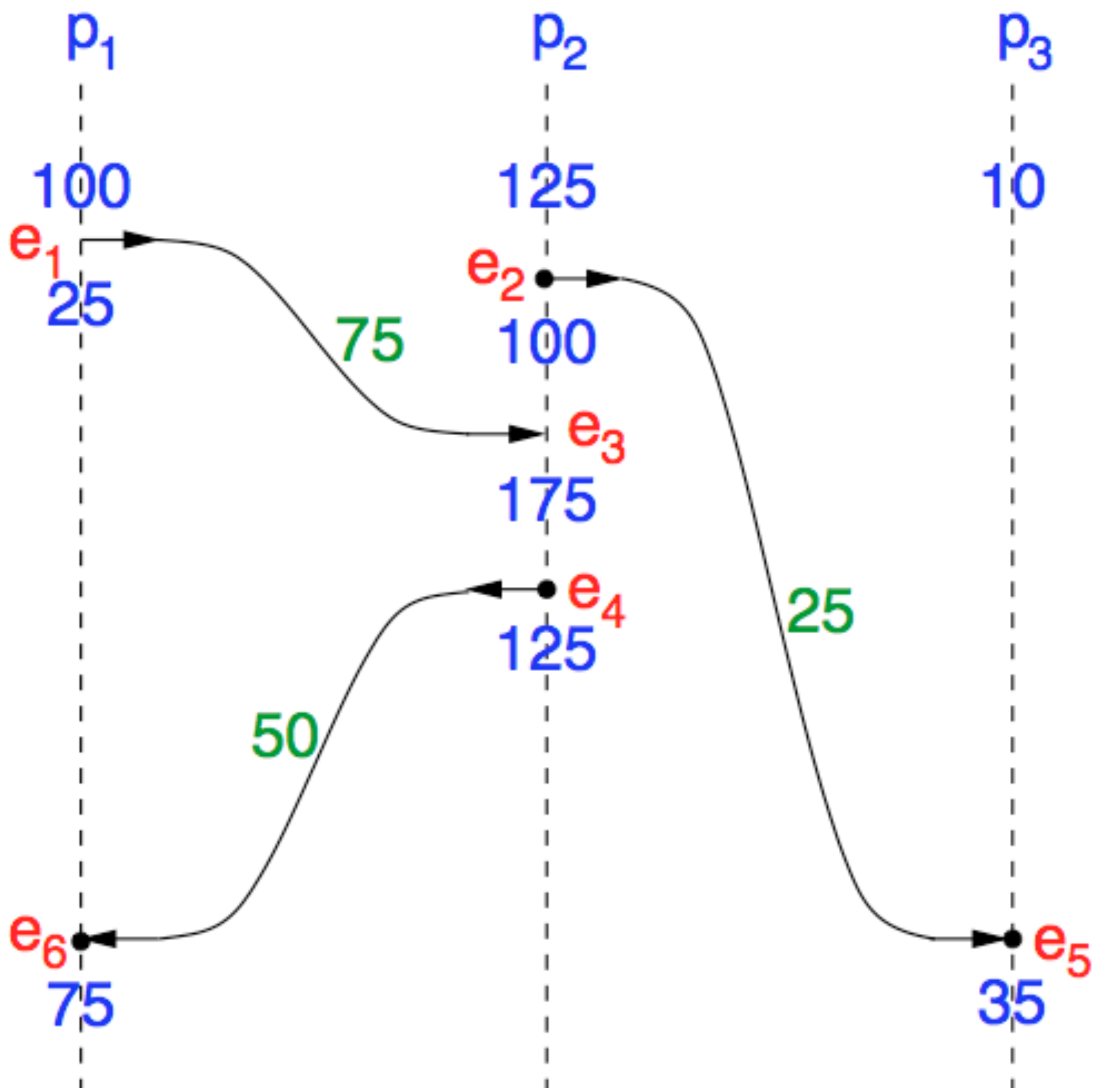
S^* recorded global state

So... Why Recording Global State?

- Stable property: a property that persists, such as termination or deadlock.
- Idea: *if a stable property holds in the system before the snapshot begins, it holds in the recorded global snapshot.*
- A recorded global state is useful in **DETECTING STABLE PROPERTIES**.
- Examples:
  - Failure recovery: a global state (checkpoint) is periodically saved and recovery from a process failure is done by restoring the system to the last saved global state.
  - Debugging: the system is restored to a consistent global state and the execution resumes from there in a controlled manner.

Homework

- Suppose Chandy and Lamport's distributed snapshot algorithm is initiated by process p_1 just after event e_1 in the following computation.



- Sketch how markers would be exchanged during the execution of the algorithm in this case.
- Which events are included in the set *H*?
- Which state components are noted down in the various processes, as the execution of the algorithm proceeds?
- Which global state S^* is discovered by the algorithm in this case?