Logical Time in Distributed Systems

Sistemi di Calcolo (II semestre) – Roberto Baldoni

Logical clock

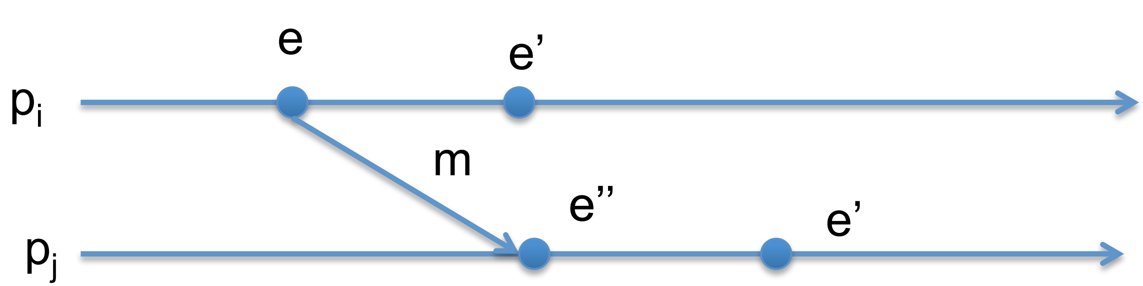
- Physical clock synchronization algorithms try to coordinate distributed clocks to reach a common value
  - based on the estimation of transmission times
    - it can be hard to find a good estimation.
  - In several applications it is not important when things happened but in which order they happened
- Reliable way of ordering events is required!

- **Notes:**
  1. Two events occurred at some process p_i happened in the same order as p_i observes them
  2. When p_i sends a message to p_j the *send* event happens before the *receive* event

- Lamport introduces the *happened-before* relation that captures the causal dependencies between events (*causal order relation*)
  - We note with $\rightarrow_i$ the ordering relation between events in a process p_i
  - We note with $\rightarrow$ the happened-before between any pair of events

Happened-Before Relation: Definition

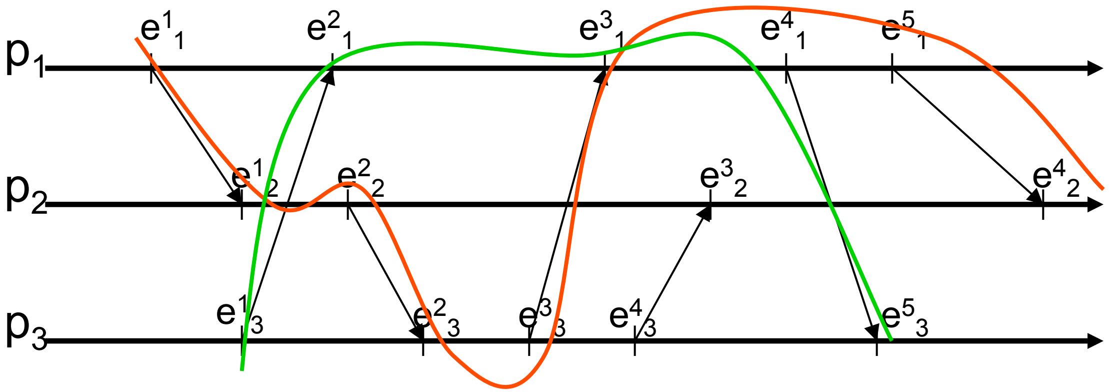
- Two events *e* and *e’* are related by happened-before relation (e → e’) if:
  - $\exists\, p_i \mid e \rightarrow_i e'$
  - $\forall$ message m send(m) → receive(m)
    - send(m) is the event of sending a message m
    - receive(m) is the event of receipt of the same message m
  - $\exists\, e, e', e'' \mid (e \rightarrow e'') \wedge (e'' \rightarrow e')$ (happened-before relation is transitive)



Happened-Before Relation

- Using the three rules is possible to define a causal ordered sequence of events e_1, e_2, … , e_n
- **Notes**:
  - The sequence e_1, e_2, …, e_n may not be unique
  - It may exists a pair of events $<e_1,e_2>$ such that e_1 and e_2 are not in happened-before relation
  - If e_1 and e_2 are not in happened-before relation then they are *concurrent* ($e_1||e_2$)
  - For any two events e_1 and e_2 in a distributed system, either $e_1 \rightarrow e_2$, $e_2 \rightarrow e_1$ or $e_1||e_2$

happened-before: example



e^j_i is j-th event of process p_i

$S_1 = < e^1_1, e^1_2, e^2_2, e^2_3, e^3_3, e^3_1, e^4_1, e^5_1, e^4_2 >$

$S_2 = < e^1_3, e^2_1, e^3_1, e^4_1, e^5_3 >$

Note:

e^1_3 and e^1_2 are concurrent

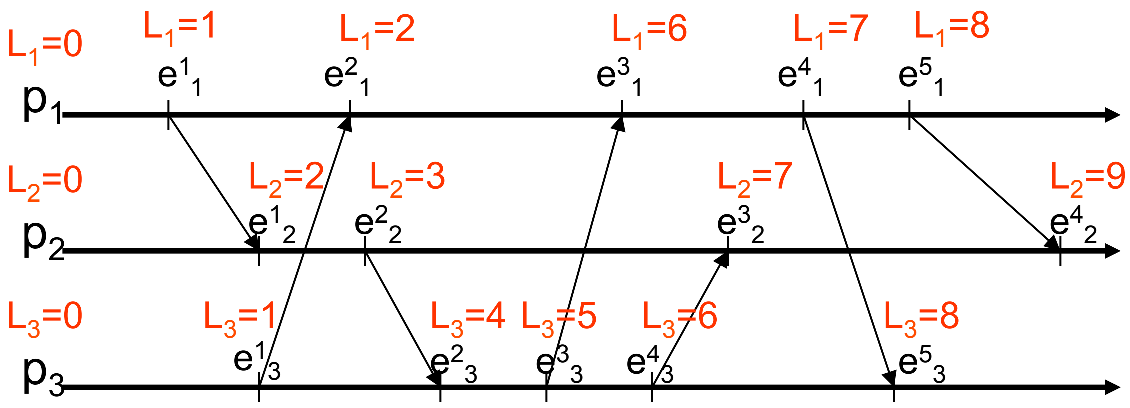
Logical Clock

- The Logical Clock, introduced by Lamport, is a software counting register *monotonically* increasing its value
  - Logical clock is not related to physical clock
- Each process p_i employs its logical clock L_i to apply a *timestamp* to events
- $L_i(e)$ is the "logical" timestamp assigned, using the logical clock, by a process p_i to events e.
- **Property**:
  - If $e \rightarrow e'$ then $L(e) < L(e')$
- **Observation**:
  - The ordering relation obtained through logical timestamps is only a partial order. Consequently timestamps could not be sufficient to relate two events

Scalar Logical Clock: an implementation

- Each process p_i initializes its logical clock L_i=0 ($\forall$ i = 1….N)
- p_i increases L_i of 1 when it generates an event (either *send* or *receive*)
  - $L_i = L_i + 1$
- When p_i sends a message *m*
  - creates an event *send(m)*
  - increases L_i
  - timestamps *m* with $t=L_i$
- When p_i receives a message *m* with timestamp *t*
  - Updates its logical clock $L_i = \max(t, L_i)$
  - Produces an event *receive(m)*
  - Increases L_i

Scalar Logical Clock: example



- e^j_i is j-th event of process pi
- L_i is the logical clock of p_i
- Note:
- $e^1_1 \rightarrow e^2_1$ and timestamps reflect this property
- $e^1_1 \parallel e^1_3$ and respective timestamps have the same value
- $e^1_2 \parallel e^1_3$ but respective timestamps have different values

Limits of Scalar Logical Clock

- Scalar logical clock can guarantee the following property
  - IF $e \rightarrow e'$ then $L(e) < L(e')$
- But it is not possible to guarantee
  - IF $L(e) < L(e')$ then $e \rightarrow e'$
- **Consequently**:
  - It is not possible to determine, analysing only scalar clocks, if two events are concurrent or correlated by the happened-before relation.
- Mattern [1989] and Fridge [1991] proposed an improved version of logical clock where events are time-stamped with local logical clock and node identifier
  - ***Vector Clock***

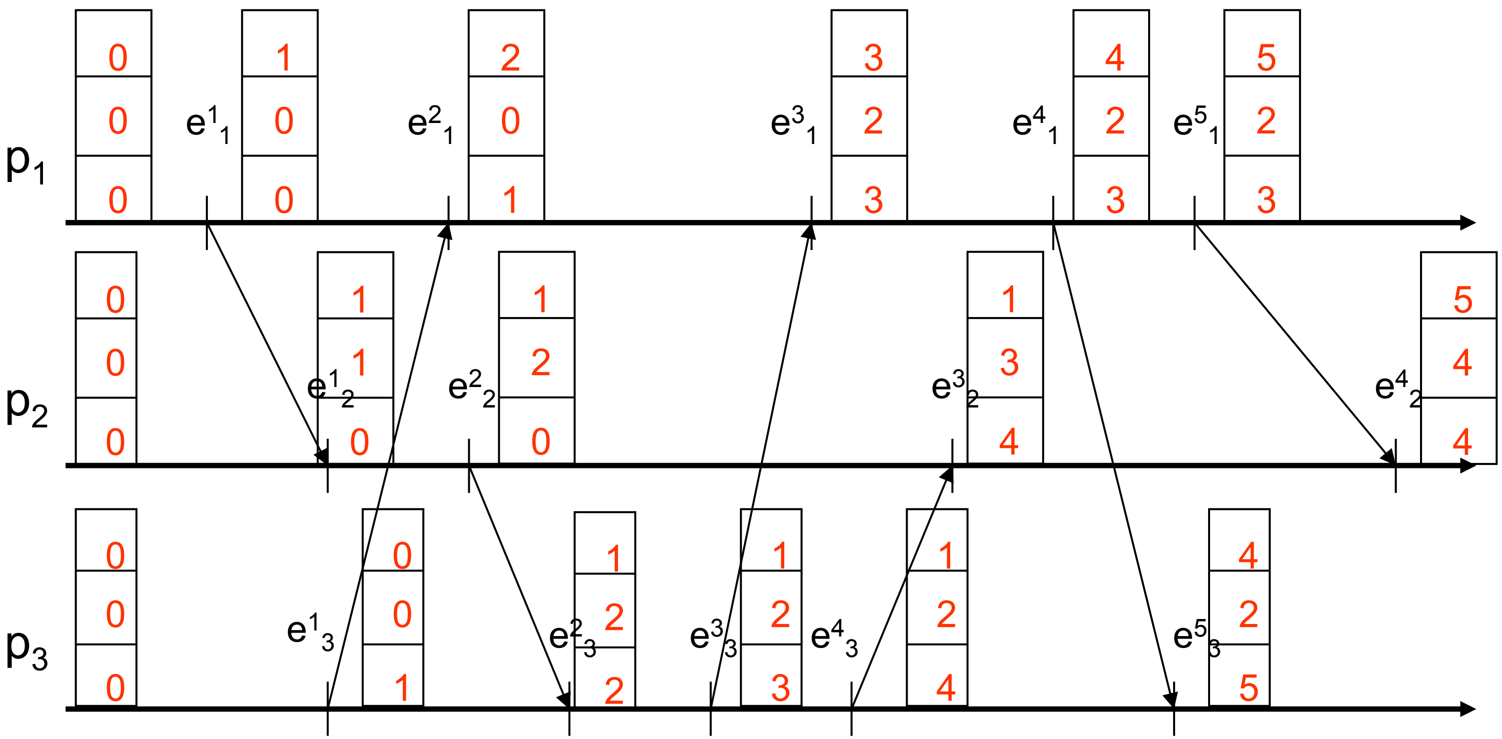
Vector Clock : definition

- Vector Clock for a set of N processes is composed by an array of N integer counters
- Each process p_i maintains a Vector Clock V_i and timestamps events by mean of its Vector Clock
- Similarly to scalar clock, Vector Clock is attached to message m (in this case we attach an array of integer)
- Vector Clock allows nodes to order events in happens-before order based on timestamps
  - Scalar clocks: $e \rightarrow e'$ implies $L(e) < L(e')$
  - Vector clocks: $e \rightarrow e'$ **iff** $L(e) < L(e')$

Vector Clock : a possible implementation

- Each process p_i initializes its Vector Clock V_i
  - $V_i[\ j\ ]=0 \ \ \forall\ j = 1 \ldots N$
- p_i increases $V_i[i]$ by1 when it generates an event
  - $V_i[\ i\ ]= V_i[\ i\ ] +1$
- When p_i sends a message *m*
  - Creates an event *send(m)*
  - Increases V_i
  - timestamps *m* with $t=V_i$
- When p_i receives a message *m* containing timestamp *t*
  - Updates it logical clock $V_i\,[\ j\ ] = \max(t[\ j\ ], V_i\,[\ j\ ]\,)\ \forall\ j = 1\ldots N$
  - Generates an event *receive(m)*
  - Increases V_i

Vector Clock: an example



Vector Clock: properties

- A Vector Clock V_i
  - $V_i[\ i\ ]$ represents the number of events produced by p_i
  - $V_i[\ j\ ]$ with $i \neq j$ represents the number of events generated by p_j that p_i can known
- V = V' if and only if
  - $V[\ j\ ] = V'\ [\ j\ ]\ \ \forall\ j = 1 \ldots N$
- V ≤ V' if and only if
  - $V[\ j\ ] \leq V'\ [\ j\ ]\ \ \forall\ j = 1 \ldots N$
- V < V' therefore the event associated to V happened before the event associated to V' if and only if
  - $V \leq V' \wedge V \neq V'$
    - $\forall\ i = 1 \ldots N\ V\ [\ i\ ] \leq V'\ [\ i\ ]$
    - $\exists\ i \in \{1 \ldots N\}\ |\ V\ [\ i\ ] < V'\ [\ i\ ]$

A comparison of Vector Clocks

$$V = \begin{array}{|c|}\hline 1 \\ \hline 0 \\ \hline 0 \\ \hline\end{array} \qquad V' = \begin{array}{|c|}\hline 1 \\ \hline 1 \\ \hline 0 \\ \hline\end{array}$$

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

$$V = \begin{array}{|c|}\hline 1 \\ \hline 0 \\ \hline 0 \\ \hline\end{array} \qquad V' = \begin{array}{|c|}\hline 0 \\ \hline 0 \\ \hline 1 \\ \hline\end{array}$$

V (e) $\neq$ V' (e') then e || e'

Differently from Scalar Clock, Vector Clock allows to determine if two events are concurrent or related by an happened-before relation

Logical Time and Ricart-Agrawal Mutual Exclusion Algorithm

Logical clock in distributed algorithms

- We have seen two mechanisms to represent logical time
  - Scalar Clock
  - Vector Clock

- Each mechanism can be used to solve different problems, depending on the problem specification
  - Scalar Timestamp → Lamport's Mutual Exclusion
  - Vector Timestamp → Causal Broadcast

Ricart-Agrawala's algorithm: implementation (see also lecture notes)

- Local variables
  - #replies (initially 0)
  - State $\in$ {Requesting, CS, NCS} (initially NCS)
  - Q pending requests queue (initially empty)
  - Last_Req
  - Num
- Algorithm

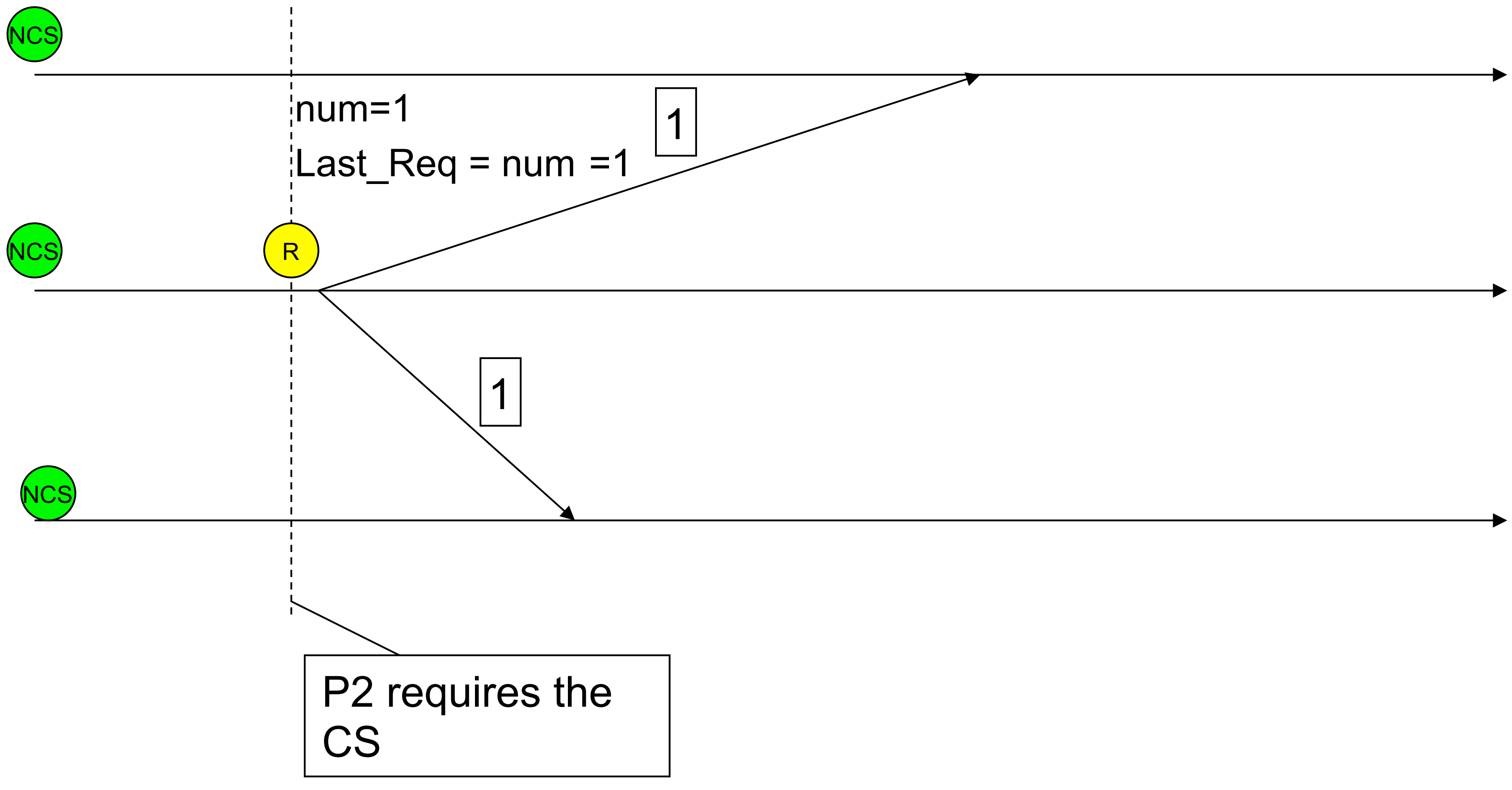
begin
1. State=Requesting
2. Num=num+1; Last_Req=num
3. ∀ i=1…N send REQUEST to pi
4. Wait until #replies=n-1
5. State=CS
6. CS
7. ∀ r∈Q send REPLY to r
8. Q= ∅; State=NCS; #replies=0

Upon receipt REQUEST(t) from pj
1. Num=max(t,num)
2. If State=CS or (State=Requesting and {Last_Req,i}<{t,j})
3. Then insert in Q{t, j}
4. Else send REPLY to pj

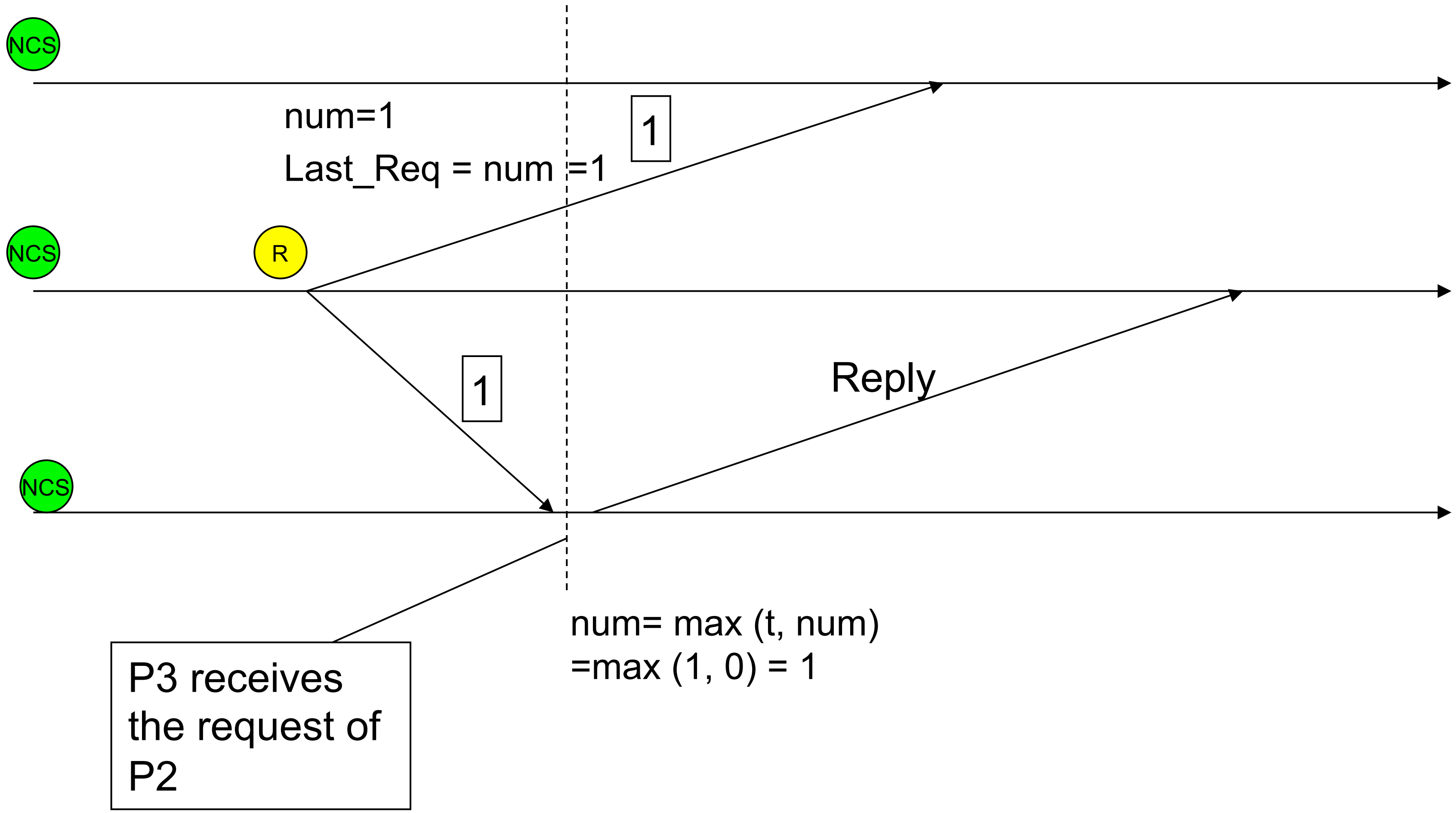
Upon receipt of REPLY from pj

1. #replies=#replies+1

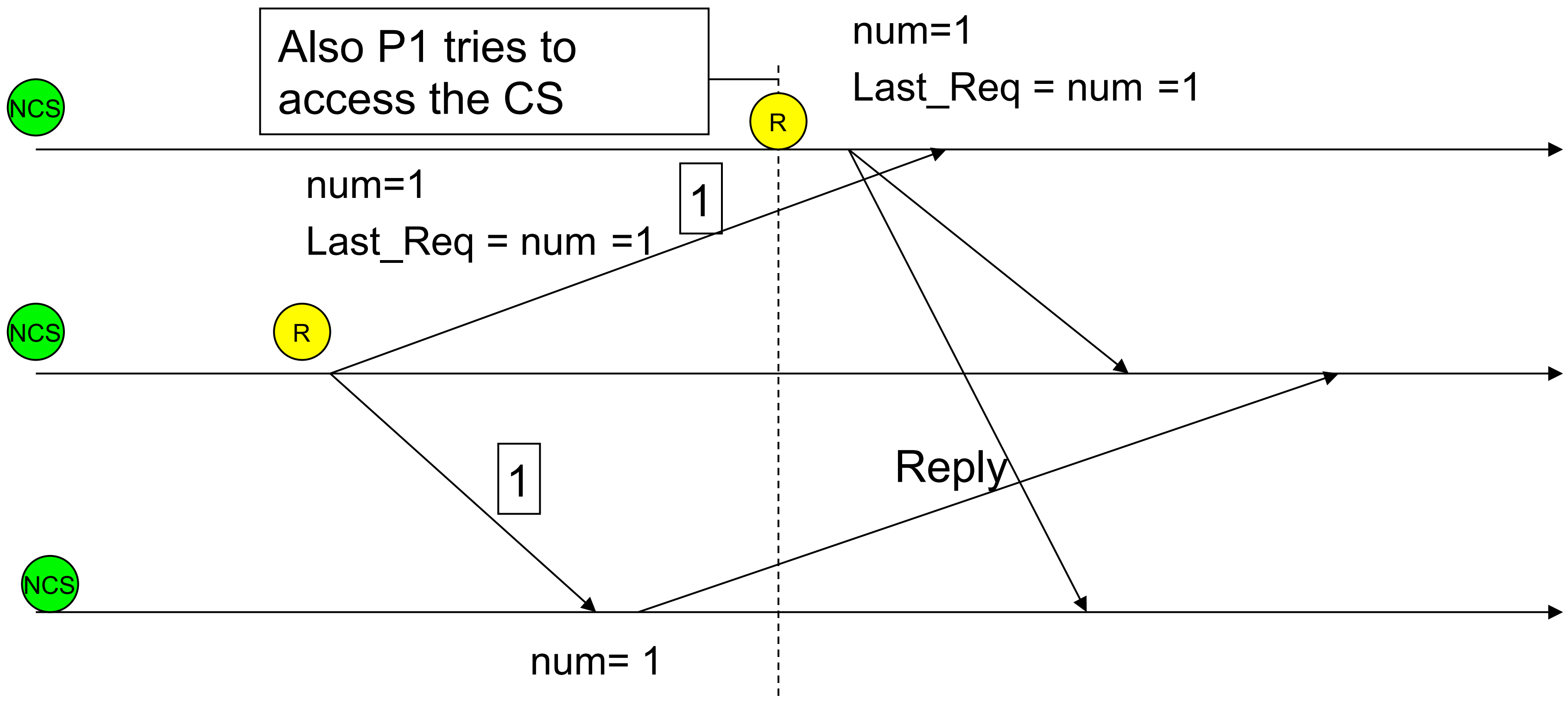
Ricart-Agrawala's algorithm: example



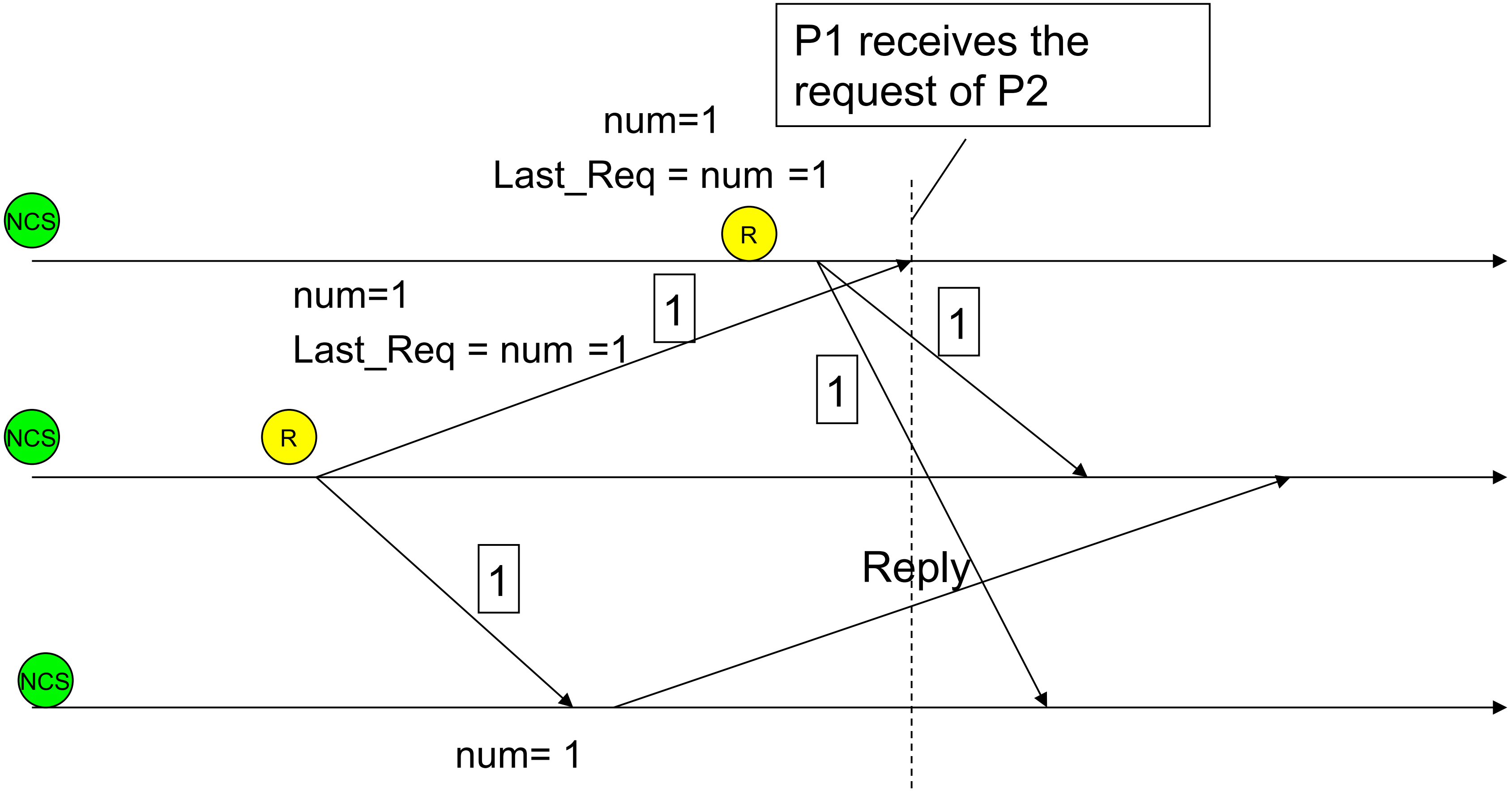
Ricart-Agrawala's algorithm: example



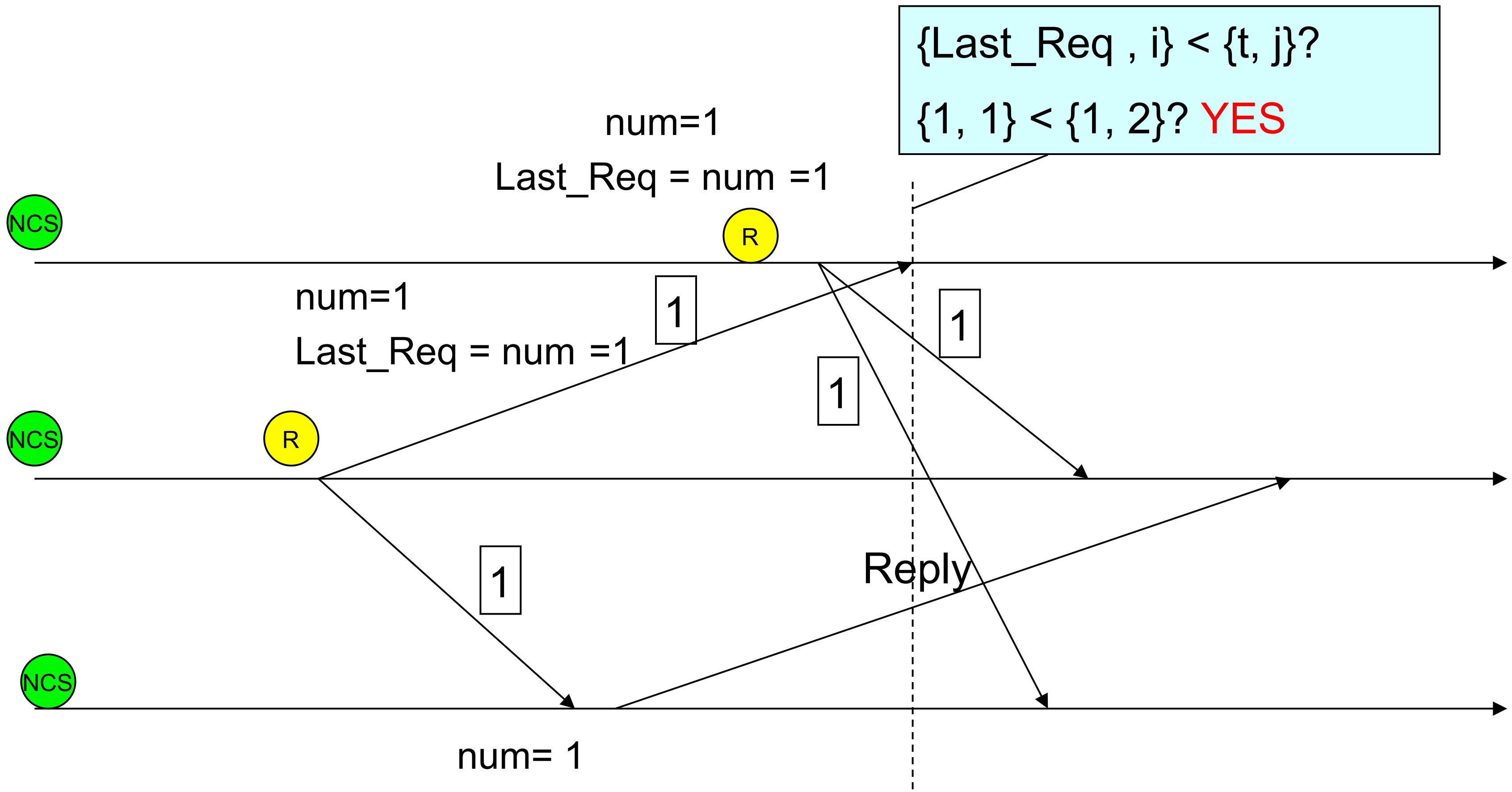
Ricart-Agrawala's algorithm: example



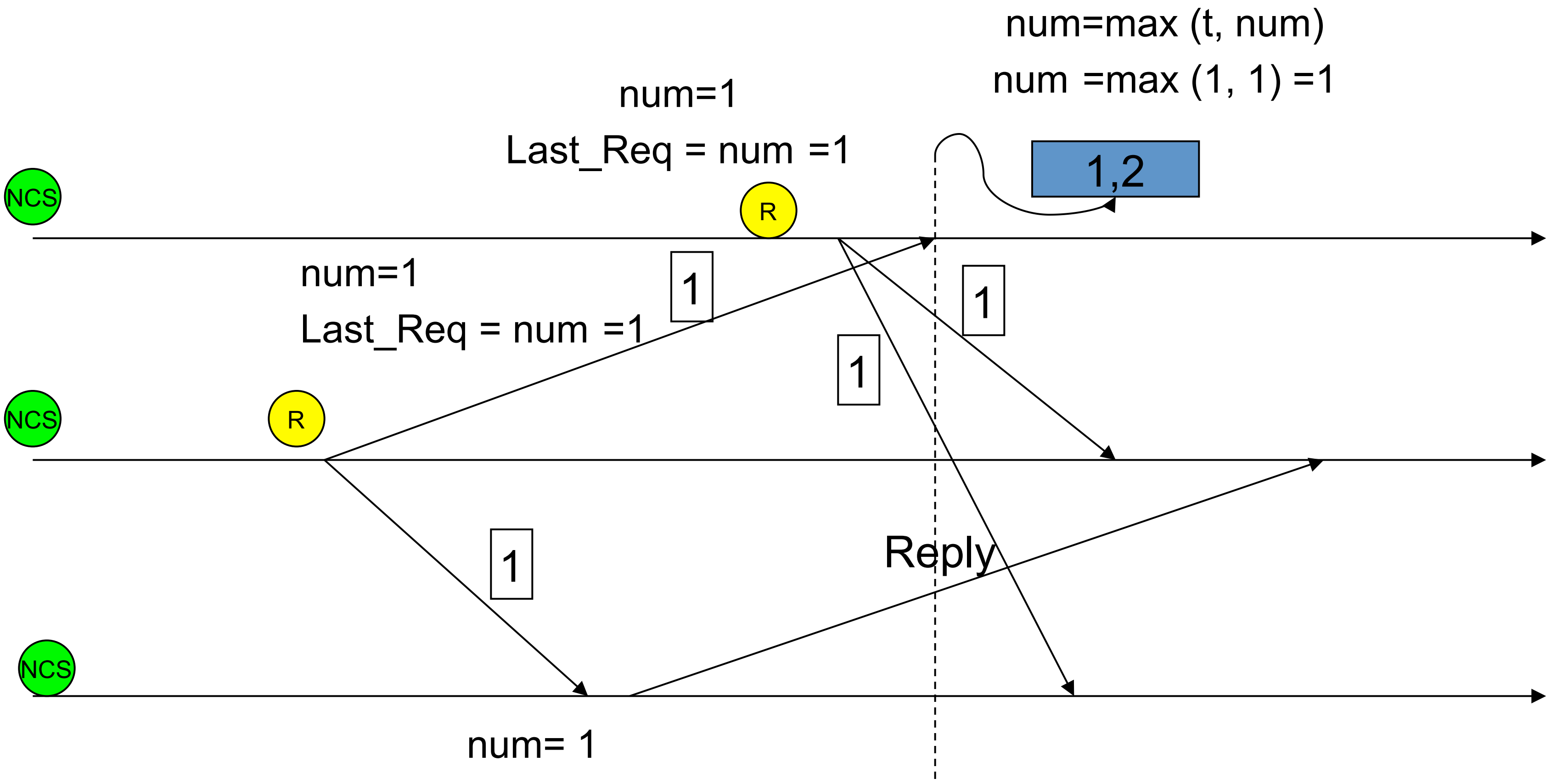
Ricart-Agrawala's algorithm: example



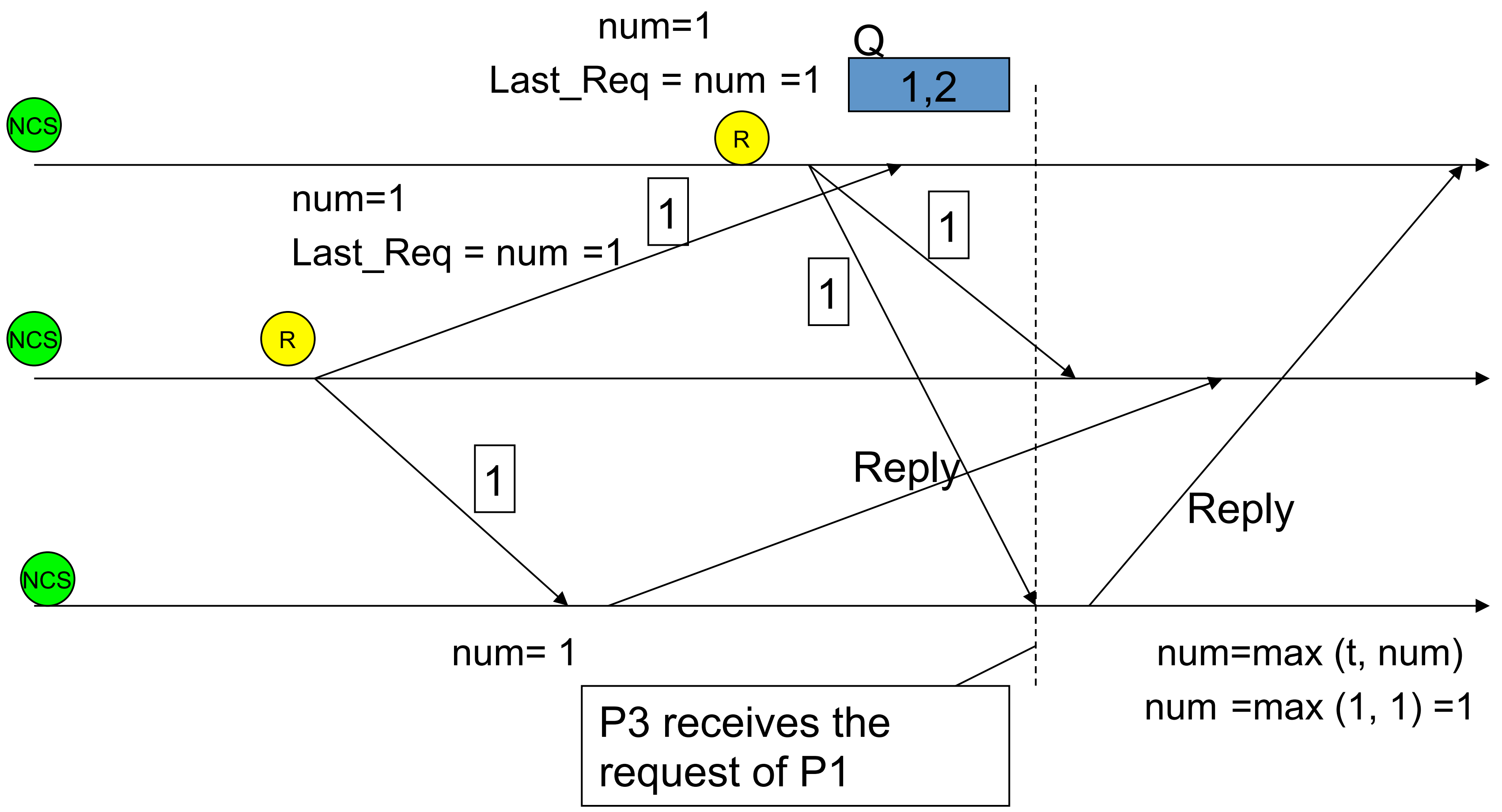
Ricart-Agrawala's algorithm: example



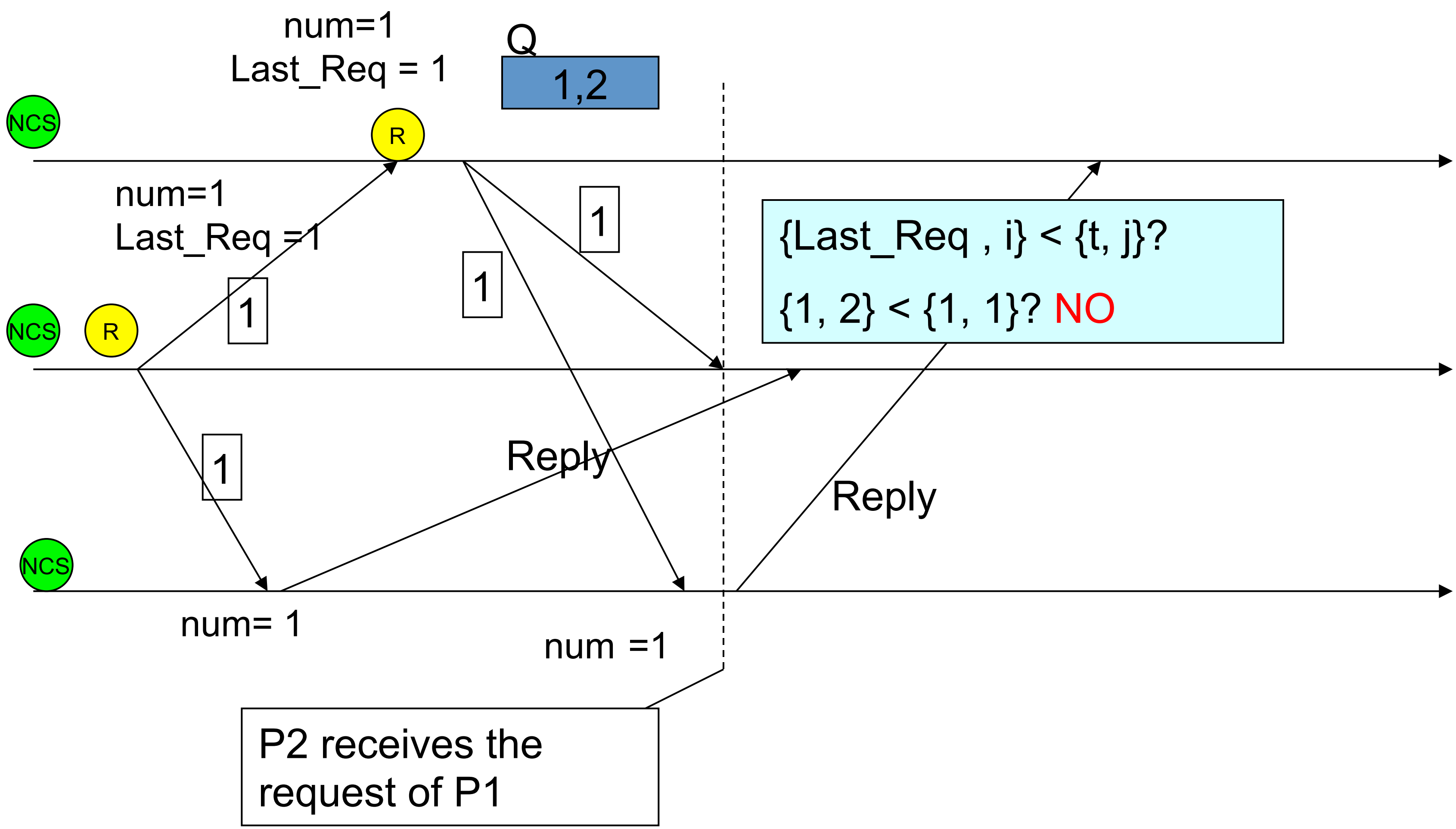
Ricart-Agrawala's algorithm: example



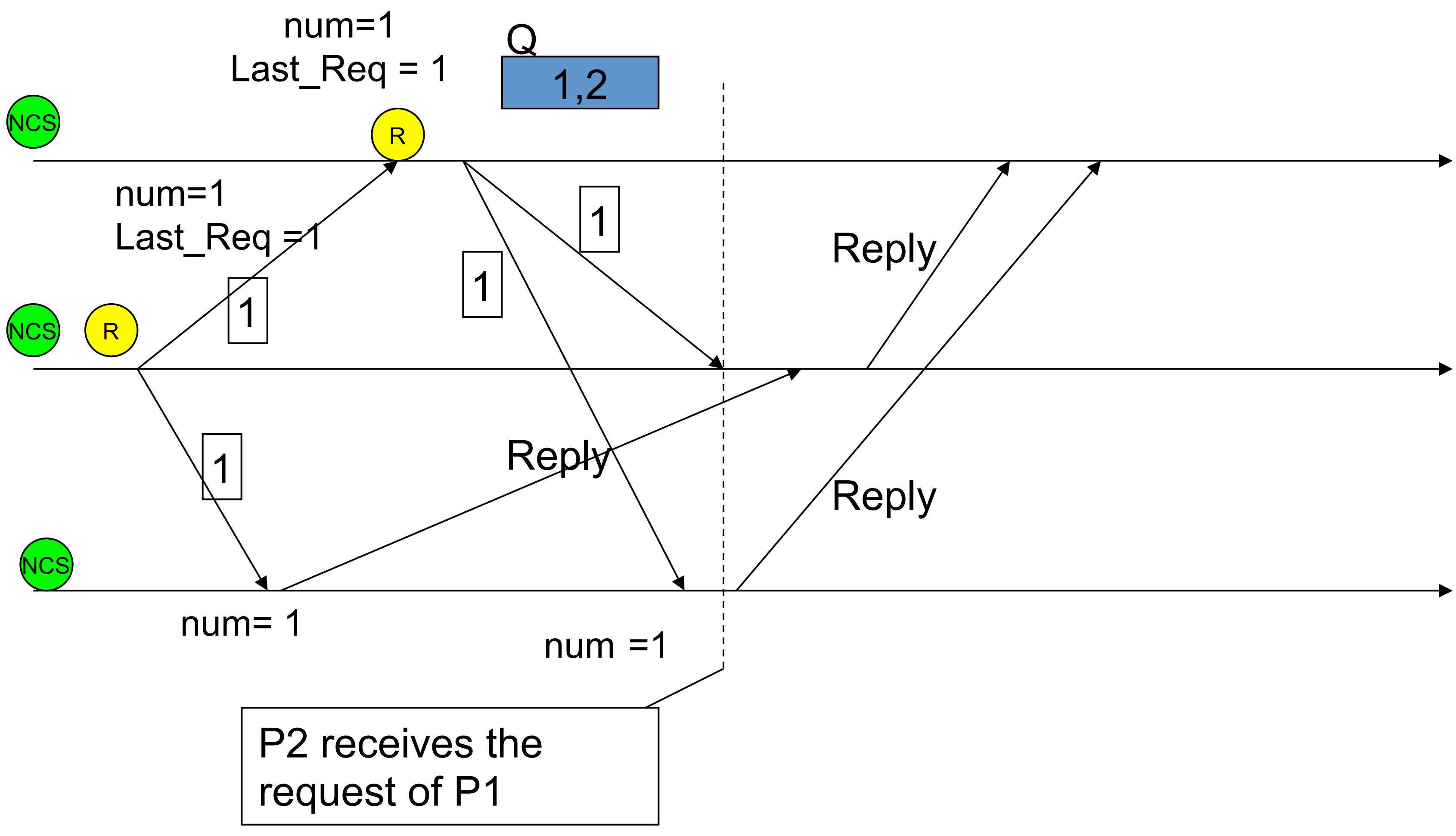
Ricart-Agrawala's algorithm: example



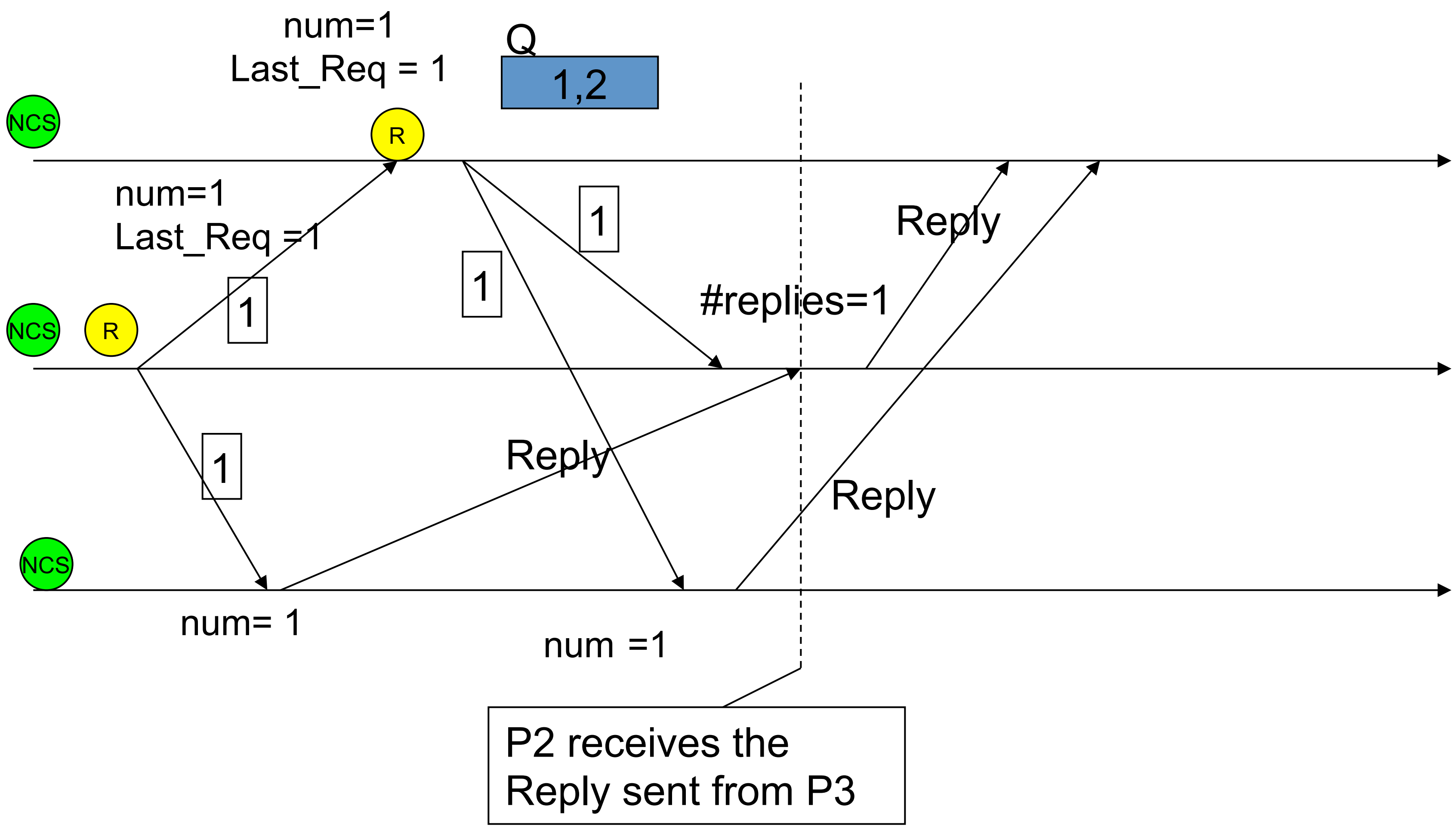
Ricart-Agrawala's algorithm: example



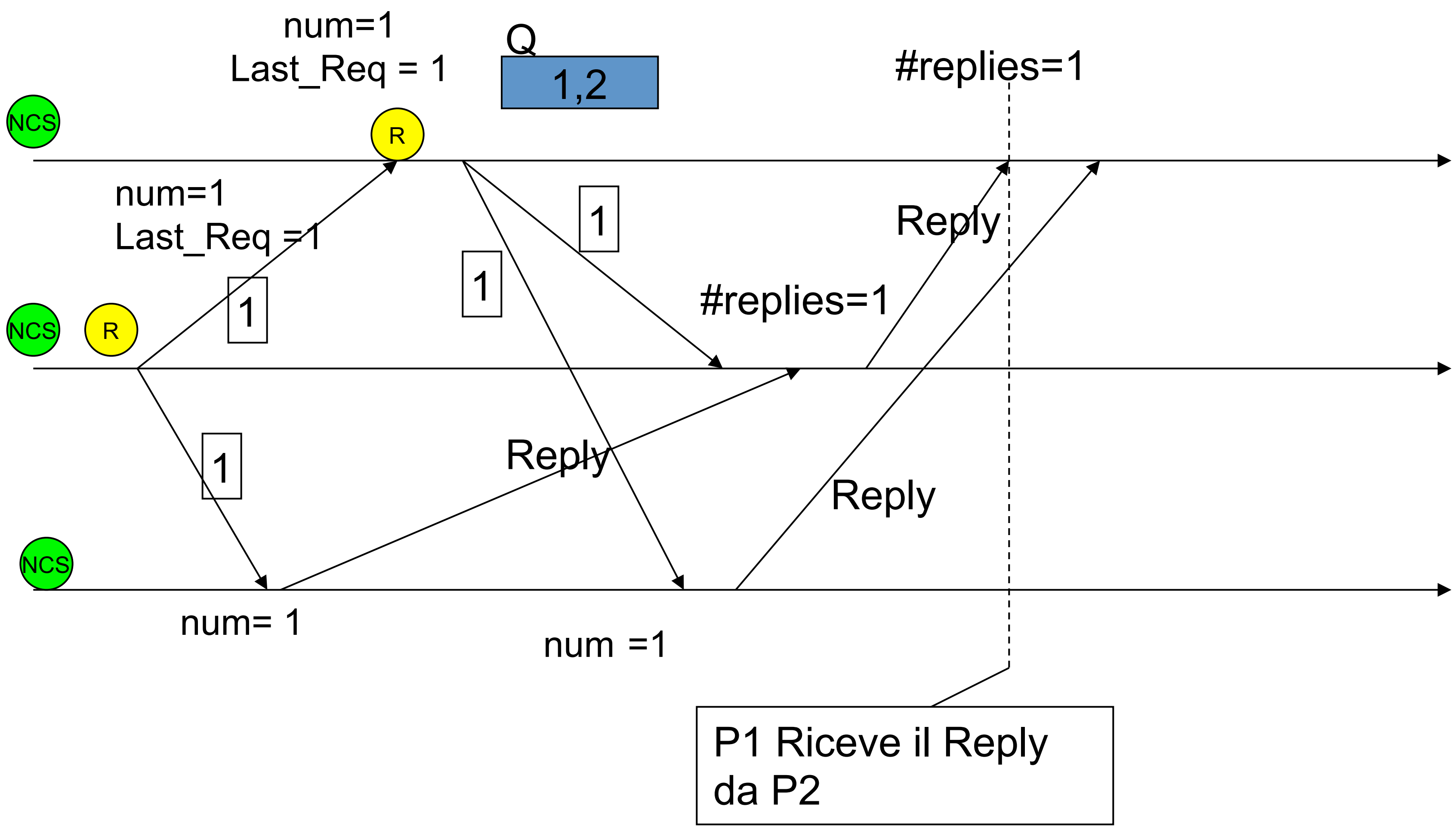
Ricart-Agrawala’s algorithm: example



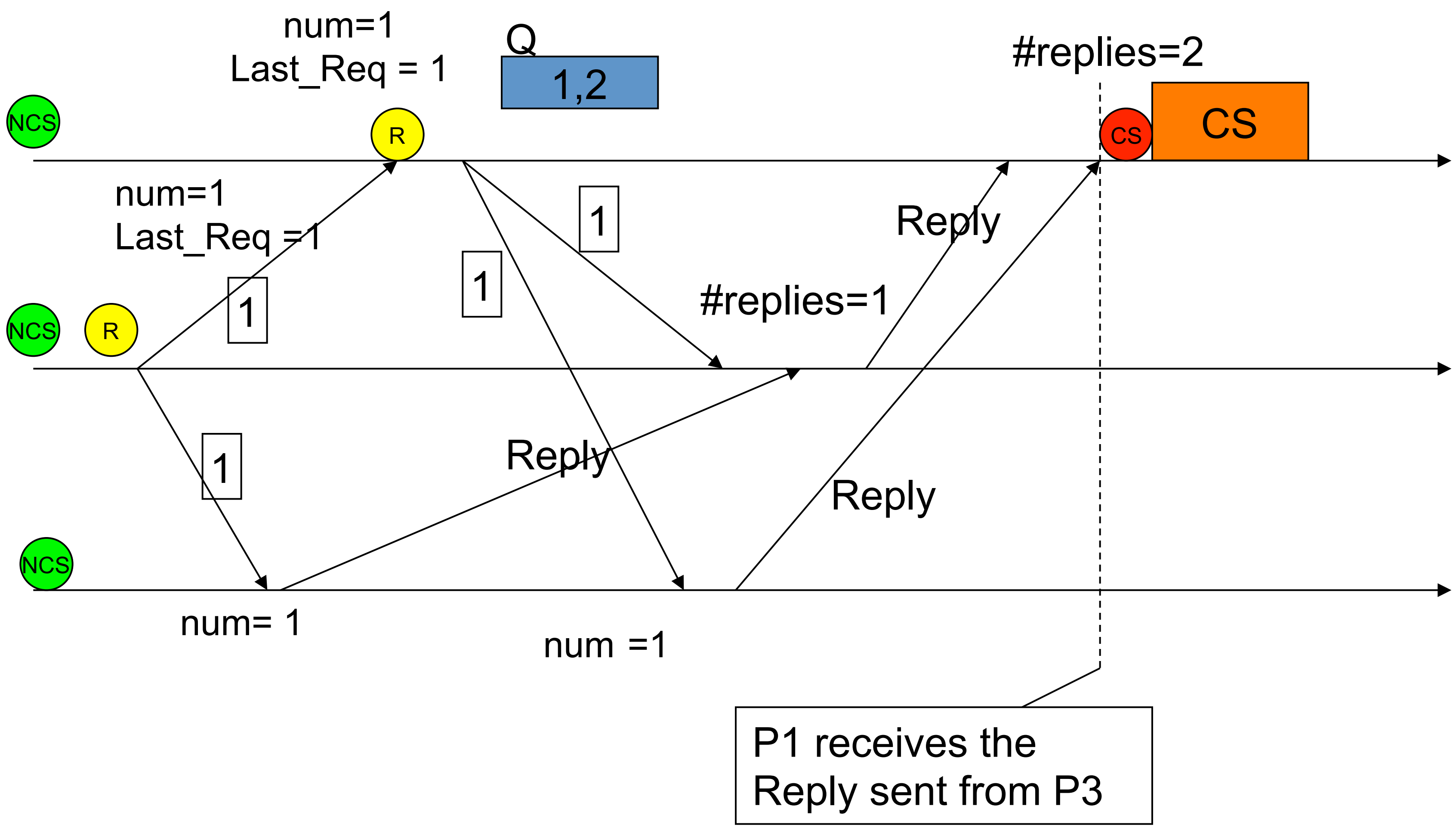
Ricart-Agrawala's algorithm: example



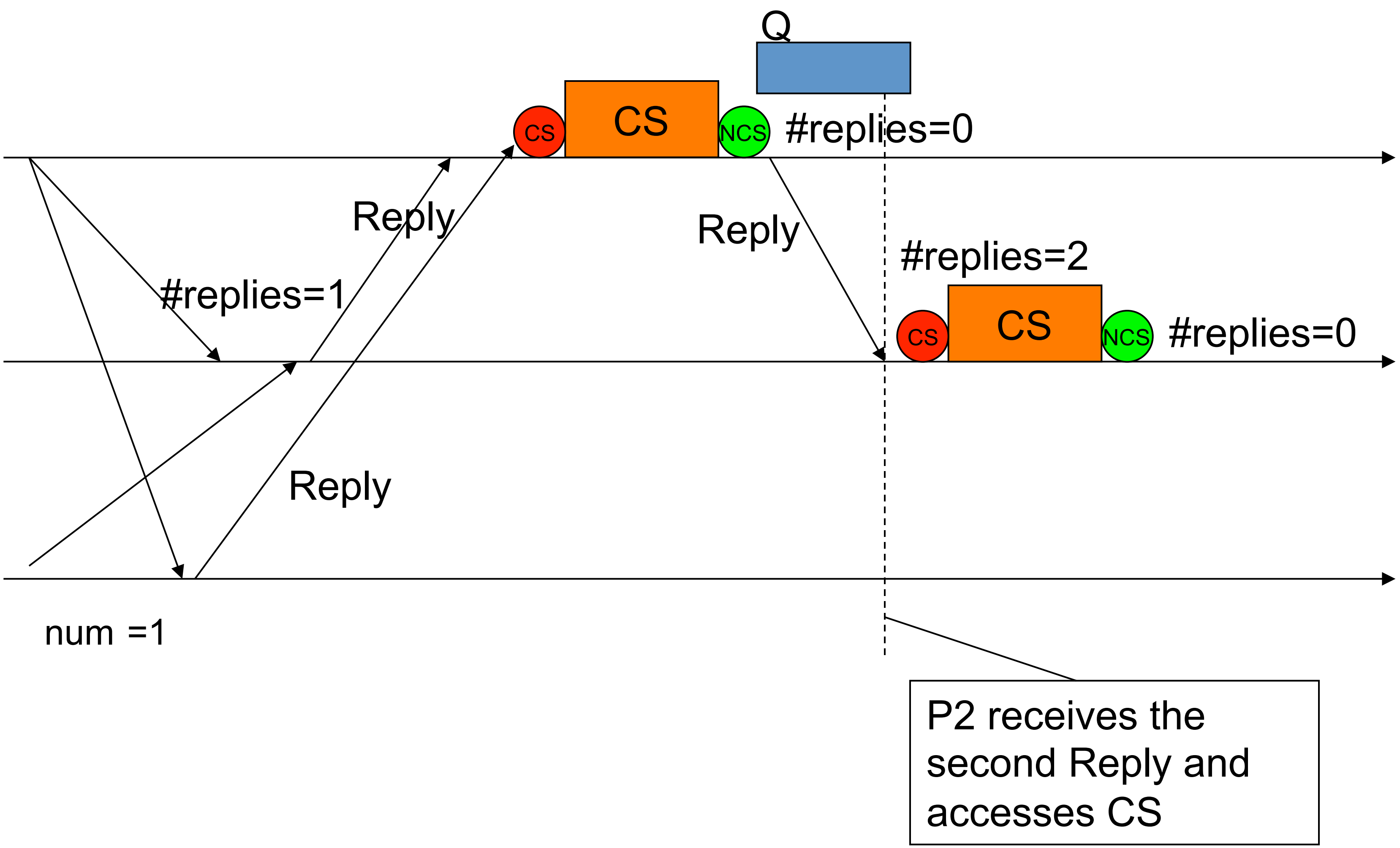
Ricart-Agrawala's algorithm: example



Ricart-Agrawala's algorithm: example



Ricart-Agrawala's algorithm: example



Causal Order among broadcast events

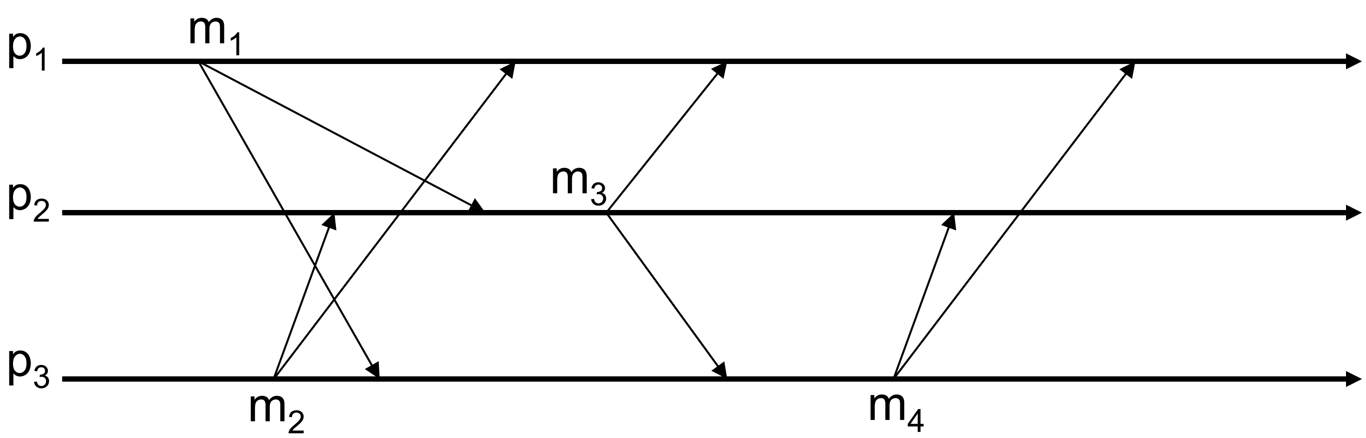
- When p_i issues a broadcast it sends a message *m* to each other process in the system
- An event *e=broadcast(m)* causally precedes another event *e'=broadcast(m')* if at least one of following condition is true
  - e and e' has been produced by the same process and broadcast(m) happens before broadcast(m')
  - e and e' was been produced by different processes but e' was produced only after the deliver of m
  - $\exists$ m'' | broadcast(m) $\rightarrow$ broadcast(m'') $\wedge$ broadcast(m'') $\rightarrow$ broadcast(m')

Causal Broadcast

- Causal broadcast was introduced by Birman and Joseph in order to reduce the asymmetry of communication channels as it was perceived by processes.

- **Specification**:
  - Let *m* and *m'* be two broadcast messages such that *broadcast(m)* → *broadcast(m')* then each process must deliver *m* before *m'*
  - Let *m* and *m'* two broadcast messages such that *broadcast(m)* || *broadcast(m')* then *m* and *m'* can be delivered in a different order by different processes

Causal Broadcast: an example



- In this scenario
  - $m_1 \rightarrow m_3 \Rightarrow$ every process must deliver m_1 before m_3
  - $m_1 \parallel m_2 \Rightarrow m_1$ and m_2 can be delivered in any order
  - $m_1 \rightarrow m_4 \Rightarrow$ every process must deliver m_1 before m_4

Causal broadcast: an implementation

- Asynchronous system without failures
- **Idea**:
  - A process p_i delays the deliver of a message m until every message that causally comes before m is delivered

- Each process p_i maintains a Vector Clock V_i containing the knowledge of p_i about the number of message sent by each process

- $V_i[\,j\,]$ represents the number of messages sent by p_j, and delivered by p_i

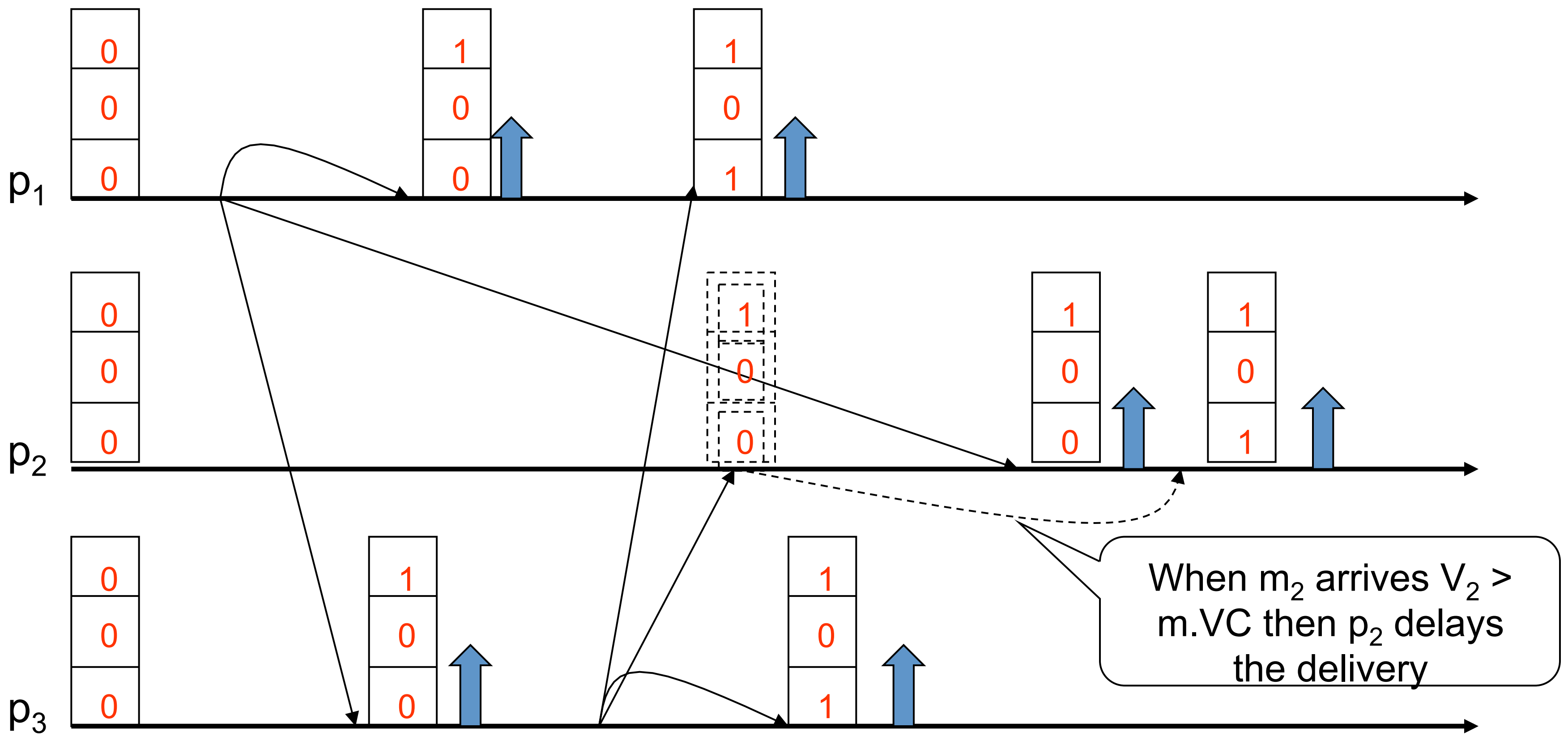
- Each broadcast message contains a vectorial timestamp m.VC

- When p_i receives a broadcast message m, delays the delivery of m until every message that causally precedes m is delivered
  - $\forall\ k \in \{1 \ldots N\}\ \ m.VC[\,k\,] \leq V_i\,[\,k\,]$

Causal Broadcast: pseudo-code

- Each process p_i implements the following rules to manage the casual broadcast
- **Procedure broadcast (m)**
  - m.VC = V_i // message timestamp
  - **for all** j $\in \{1 \dots N\}$
    - Send(m) to p_j // message broadcast
  - **endFor**
  - $V_i = V_i + 1$ //updating local clock
- **Upon receive m from p_j**
  - delay the delivery until $\forall\, k \in \{1 \dots N\}$ $m.VC[\,k\,] \leq V_i\,[\,k\,]$
  - if $i \neq j$
  - then $V_i\,[\,j\,] = V_i\,[\,j\,] + 1$ //updating local clock
  - deliver m to the upper layer //deliver event

Causal Broadcast: an example



Causal Broadcast: Safety

- **Property**:
  - Let two broadcast messages *m* and *m'* such that *broadcast(m)* → *broadcast(m')* then each process have to deliver *m* before *m'*
- **Observation:**
  - if m is the k-th message sent by p_i then m.Vc[i]= k-1
- Safety property can be proved by induction using the causal ordering relation among broadcast messages
- **Definition**:
  - Let two broadcast events b and b' with b → b'. These events have a causal distance k if ∃ a sequence of k broadcast events $b_1 \ldots b_k$ such that
    - $\forall$ I $\in \{1\ldots k\}$ $b_i \rightarrow b_{i+1} \wedge (\neg\exists)\ m^* \mid b_i \rightarrow m^* \rightarrow b_{i+1}$
    - $b \rightarrow b_1$
    - $b_k \rightarrow b'$

Proof – basic case (K=0)

- Given two messages m, m’ such that
  1. broadcast(m) → broadcast(m’)
  2. There does not exists broadcast(m’’) such that broadcast(m) → broadcast(m’’) ∧ broadcast(m’’) → broadcast(m’).

- We can have two distinct cases
  1. m and m’ have been issued by the same process
  2. m and m’ have been issued by distinct processes

Case 1 – brodacast produced by the same process

1. p_j is the receiver
2. For line 3 in broadcast procedure
   1. m'.VC[i]:= m.VC[i]+1.
      if m is the h-th message sent by p_i, m.VC[i]=h-1 and m'.VC[i]=h.
3. A process p_j that receives m' verifies the following delivery condition:
   1. $\forall$ x $\in$ {1,…,n} m'.VC[x] ≤ V_j[x] and m'.VC[i] ≤ V_j[i]
4. V_i[x] is equals to h if and only if the h-th message sent by p_x was delivered by p_i. (line 3 receive thread).
5. Consequently from 2,3,4, m' can be delivered only after the deliver of m.

Case 1 – brodacast produced by distinct processes

- m and m' was been sent by distinct processes, respectively p_i e p_j. P_k is the receiver.
- broadcast(m) → broadcast(m´), m' was broadcasted by p_j after the deliver of m.
  Without loss of generality m.VC[i]=h-1
  - For line 3 of reception thread e for assumption of k=0 we have m'.VC[i]=h.
- The receiver process p_k respects the following delivery condition:
  - $\forall\, x \in \{1,\ldots,n\}$ m'.VC[x] ≤ V_k[x] and m'.VC[i] ≤ V_k[i]
- To deliver the message, m'.VC[i] ≤ V_k[i], that is V_k[i] ≥ h
- V_k[i] is equals to h if and only if the h-th message sent by p_i has been delivered by p_k. (line 3 of reception thread thread).
- For 2,3,4, p_k can deliver m' only after the deliver of m

Proof – Inductive step($k>0$)

- $\exists$ a sequence of k broadcast events $b_1, b_2 \ldots b_k$ such that

  $b \rightarrow b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b_k \rightarrow b'$
- Inductive hypothesis : m has been delivered before m_k
- We have to prove that m_k has been delivered before m'.
  - It follows from the basic case.
- m has been delivered before m'.

Causal Broadcast: Liveness

- **Property**:
  - Eventually each message will be delivered

- Liveness is guaranteed by the following assumptions:
  - The number of broadcast events that precedes a certain event is finite
  - Channels are reliable