Distributed Systems
Master of Science in Engineering in Computer Science

AA 2020/2021

LECTURE 3 (PART 2): TIME IN DISTRIBUTED SYSTEMS

Logical Time

Logical clock

Physical clock synchronization algorithms try to coordinate distributed clocks to reach a common value

- Physical clock synchronization algorithms are based on the estimation of transmission delay but in several system 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

However in a Distributed System, each system has its own “logical clock”

- If clocks are not aligned it is not possible to order events generated by different processes

Reliable way of ordering events is required!

- **Notes**:
  - Two events occurred at some process p_i happened in the same order as p_i observes them
  - 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 ($e \rightarrow e'$) if:

- $\exists\, p_i \mid e \rightarrow_i e'$
- $\forall$ message m send(m) $\rightarrow$ 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)

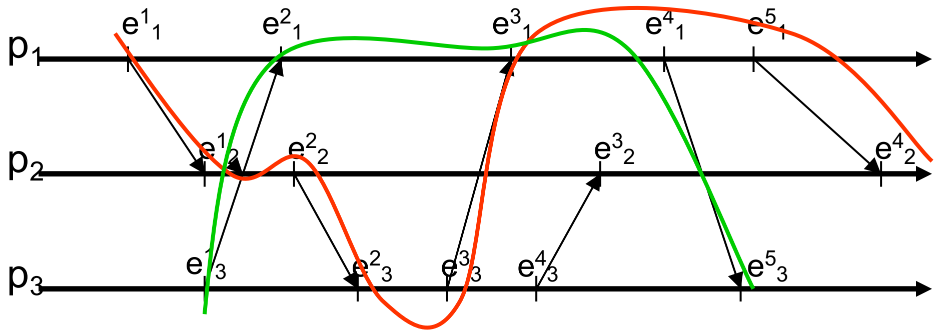
Happened-Before Relation

Using the three rules is possible to define a causal ordered sequence of events $e_1, e_2, \ldots, e_n$

Notes:

- The sequence $e_1, e_2, \ldots, e_n$ 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*)

- $L_i = L_i + 1$

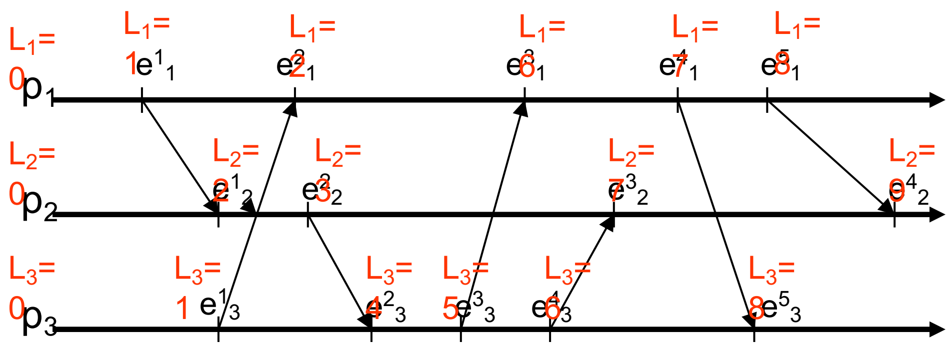
When p_i sends a message *m*

- 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 p_i
- L_i is the logical clock of p_i
- Note:
  - $e^1_1 \rightarrow e^2_1$ and timestamps reflect this property
  - $e^1_1 \,||\, e^1_3$ and respective timestamps have the same value
  - $e^1_2 \,||\, e^1_3$ 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, analyzing 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 timestamped 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 : an implementation

Each process p_i initializes its Vector Clock V_i

- $V_i[\ j\ ]=0\ \ \forall\ j = 1 \ldots N$

p_i increases $V_i[i]$ of 1 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: 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

- $V[\ j\ ] = V'\ [\ j\ ]\ \ \forall\ j = 1 \dots N$

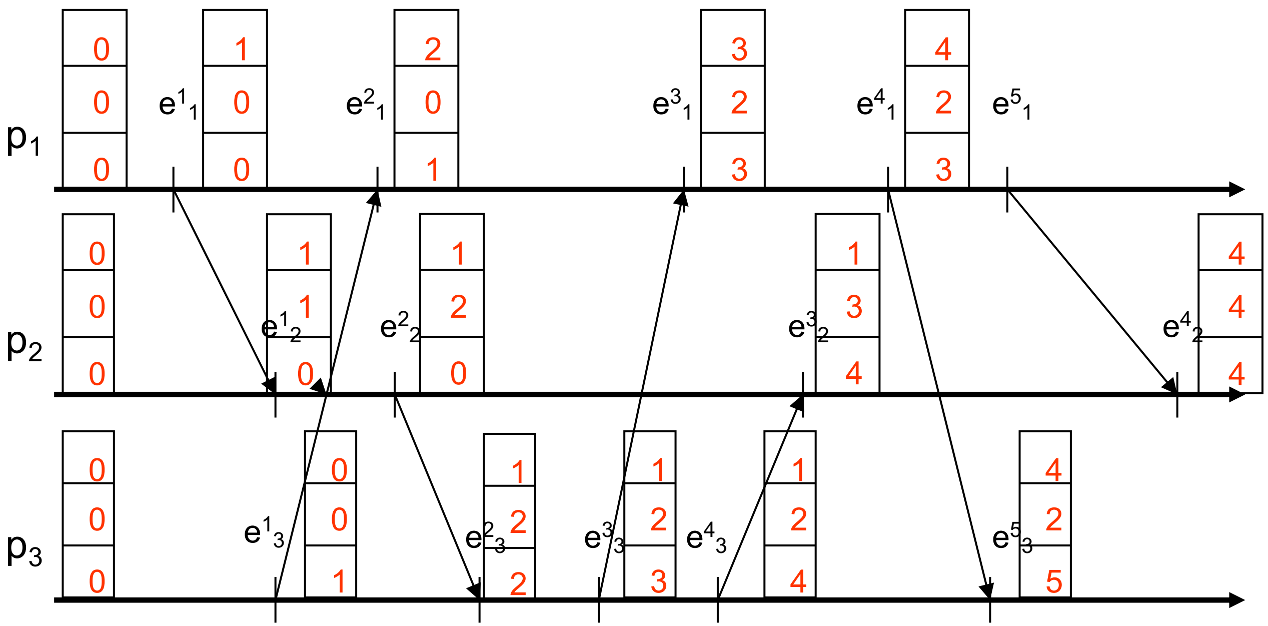
$V \leq V'$ if and only if

- $V[\ j\ ] \leq V'\ [\ j\ ]\ \ \forall\ j = 1 \dots N$

$V < V'$ therefore the event associated to V happened before the event associated to V' if and only if

- $V \leq V' \wedge V \neq V'$
  - $\forall\ i = 1 \dots N\ V'\ [\ i\ ] \geq V\ [\ i\ ]$
  - $\exists\ i \in \{1 \dots N\}\ |\ V'\ [\ i\ ] > V\ [\ i\ ]$

Vector Clock: an example



A comparison of Vector Clocks

$$V = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad V' = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

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

$$V = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad V' = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

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 Distributed Algorithms

Logical clock in distributed algorithms

We have seen two mechanisms to represent logical time

- Scalar Clock
- Vector Clock

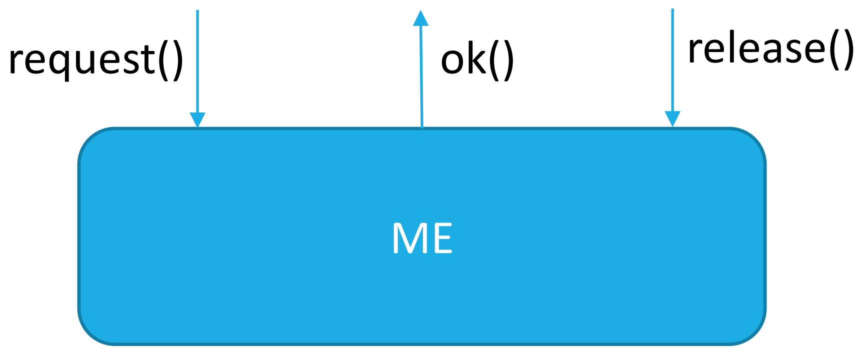
Each mechanism can be used to solve different problems, depending on the problem specification

- Scalar Timestamp $\rightarrow$ Lamport's Mutual Exclusion
- Vector Timestamp $\rightarrow$ Causal Broadcast

The Mutual Exclusion abstraction

Specification

- Mutual Exclusion: at every time t at most one process p is in critical section
- No-Deadlock: there always exists a process p able to enter the critical section.
- No-Starvation: every process p requesting the critical section eventually gets in.



Time stamp based algorithm: Lamport

Difference from concurrent system

- When a process wants to enter the CS sends a request message to all the other

An history of the operations is maintained by using a counter (time stamp)

Each transmission and reception event is relevant to the computation:

- The counter is incremented for each send and receive event
- The counter is incremented also when a message, not directly related to the mutual exclusion computation, is sent or received.

Lamport’s algorithm: implementation

Local data structures to each process pi

- ck
  - Is the counter for process pi
- Q
  - Is a queue maintained by pi where CS access requests are stored

Algorithm rules for a process pi

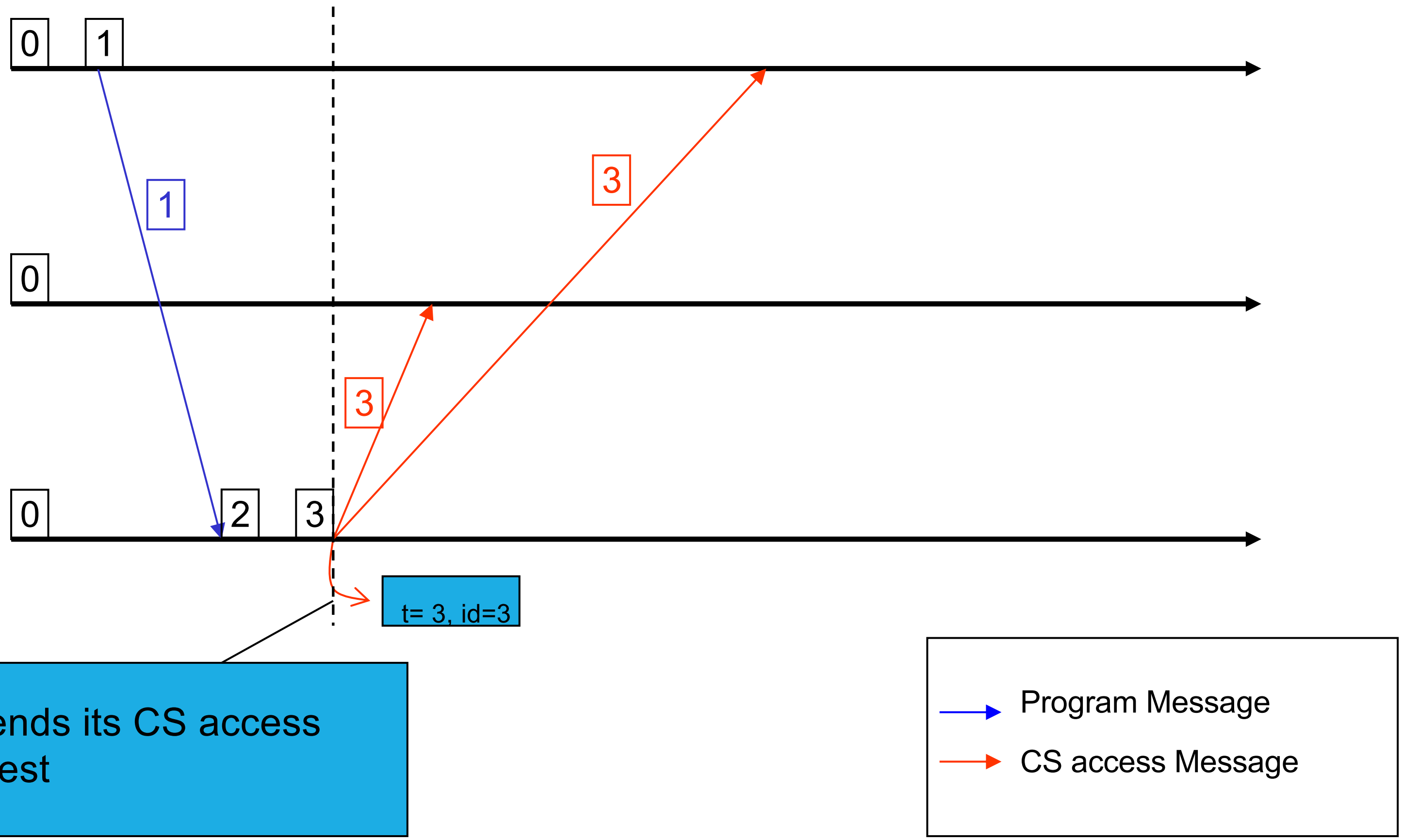
- Access the CS
  - pi sends a request message, attaching ck, to all the other processes
  - pi adds its request to Q
- Request reception from a process pj
  - pi puts pj request (including the timestamp) in its queue
  - pi sends back an ack to pj

Lamport’s algorithm: implementation

Algorithm rules for a process pi

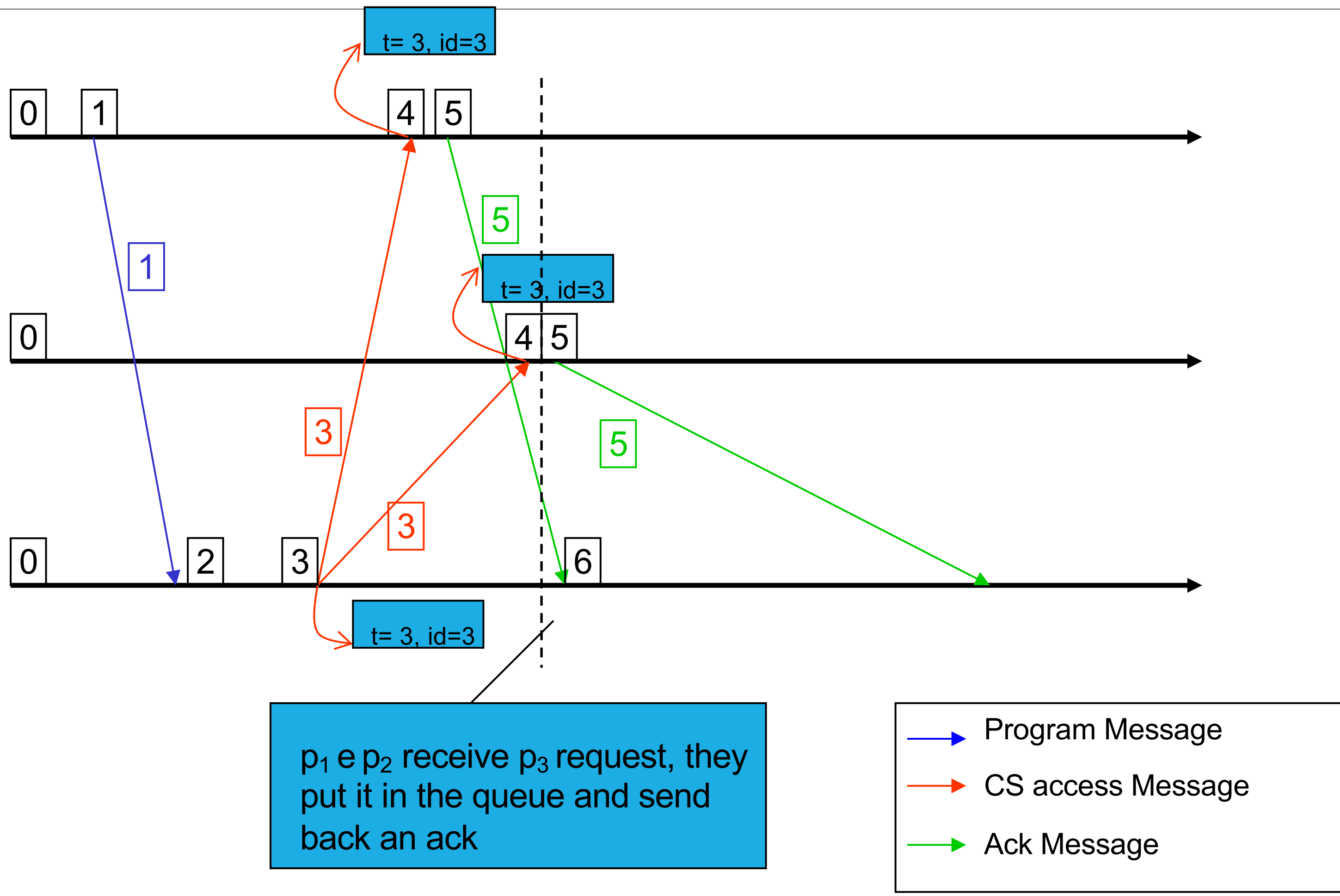
- p_i enters the CS iff
  - p_i has, in its queue, a request with timestamp t
  - t is the small timestamp in the queue
  - p_i has already received an ack with timestam t’ from any other process and t’>t
- Release of the CS
  - p_i sends a RELEASE message to all the other processes
  - p_i deletes its request from the queue
- Reception of a release message from a process pj
  - p_i deletes p_j’s request from the queue

Lamport's algorithm: example



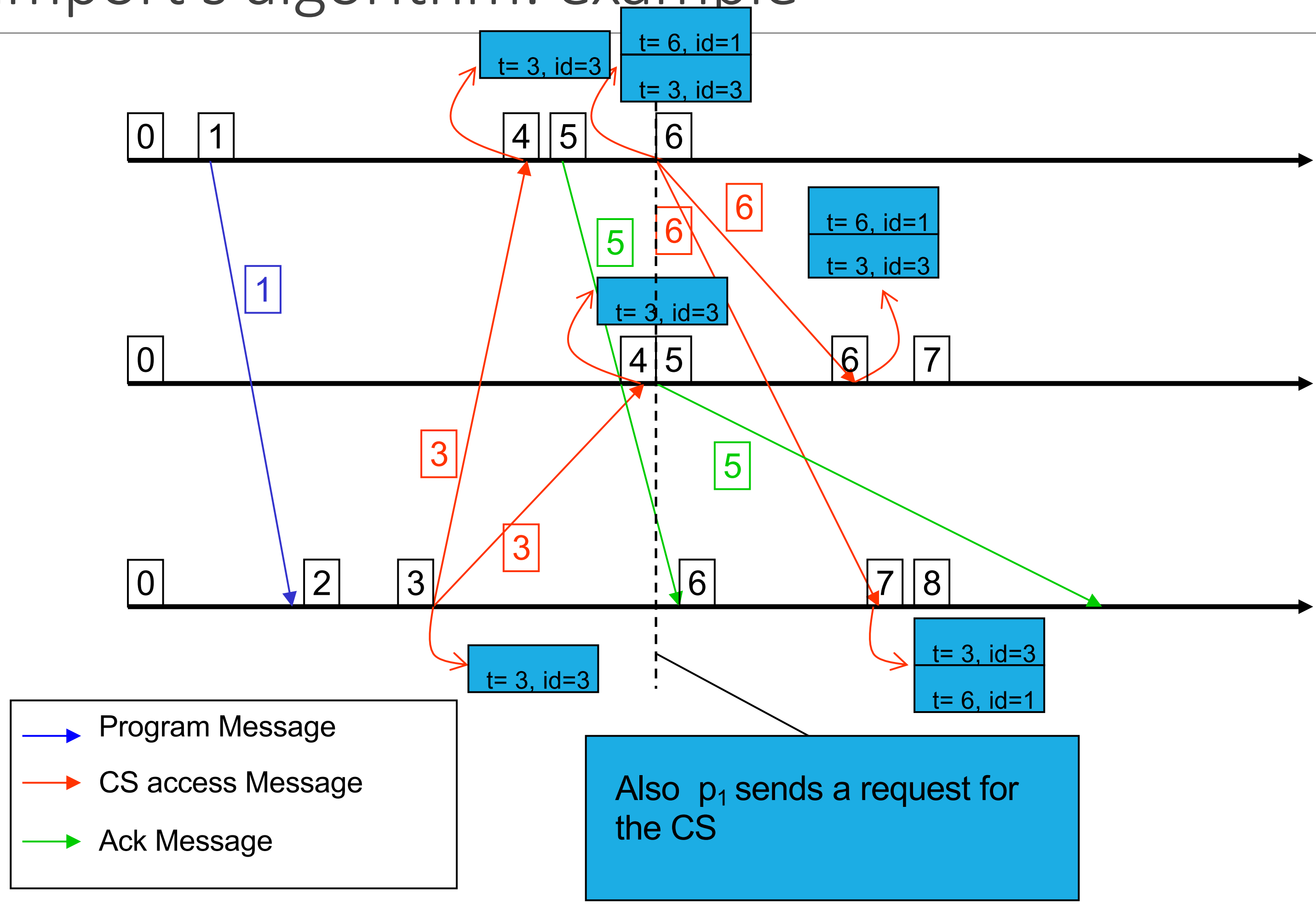
p_3 sends its CS access request

Lamport's algorithm: example

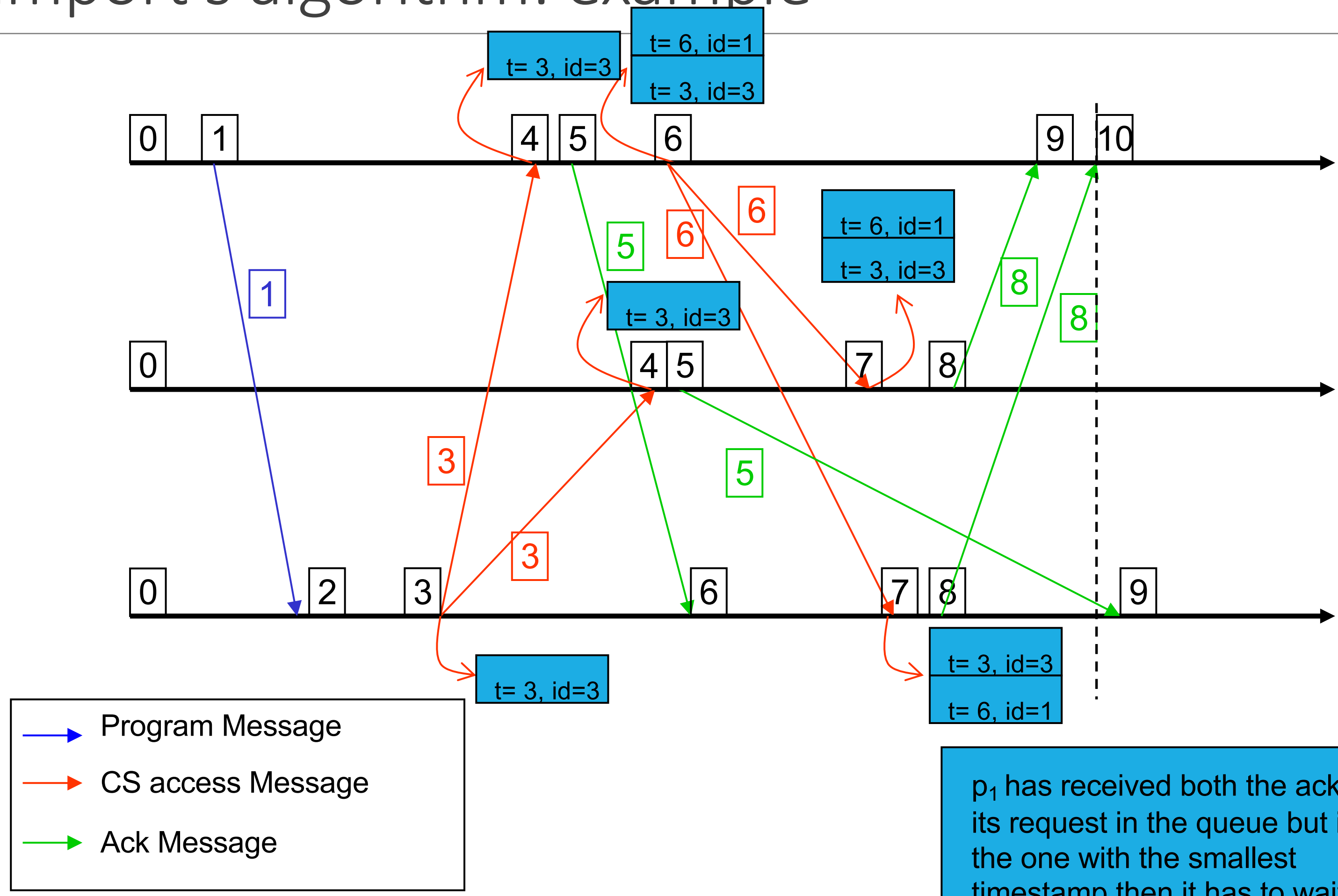


p_1 e p_2 receive p_3 request, they put it in the queue and send back an ack

Lamport's algorithm: example

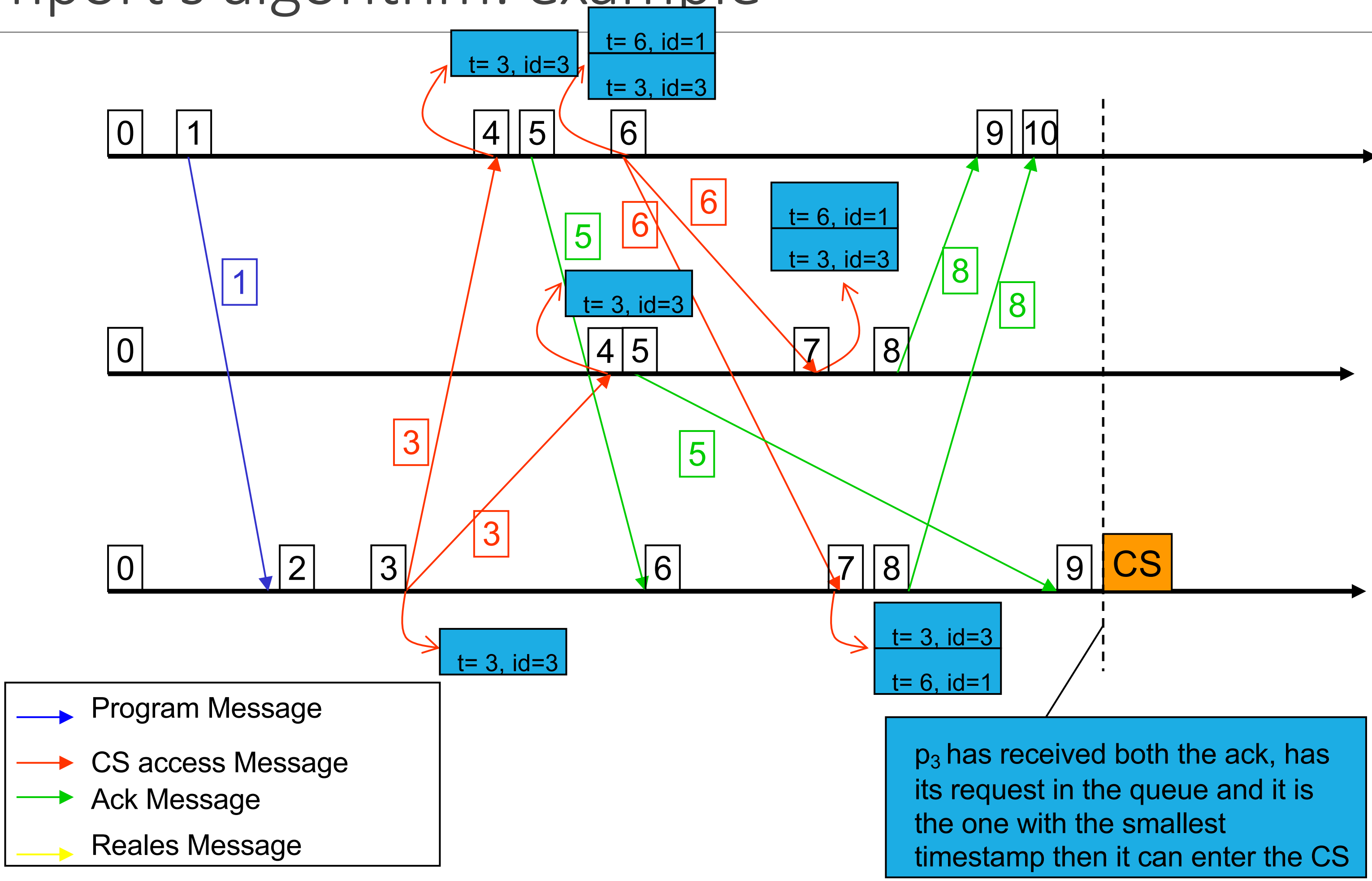


Lamport’s algorithm: example

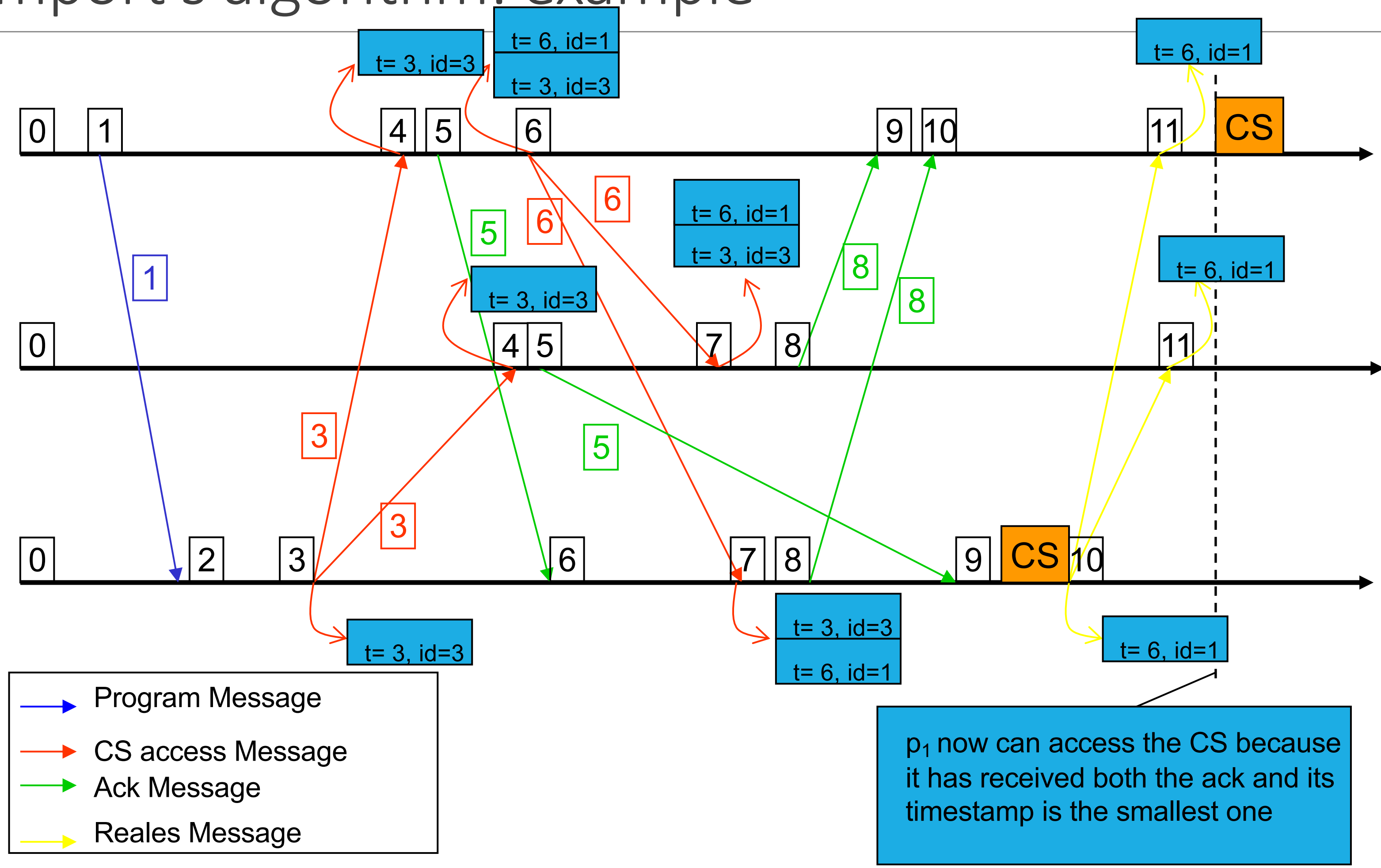


p_1 has received both the ack, has its request in the queue but it is not the one with the smallest timestamp then it has to wait

Lamport's algorithm: example



Lamport's algorithm: example

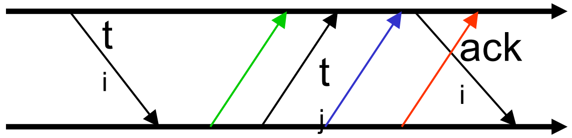


p_1 now can access the CS because it has received both the ack and its timestamp is the smallest one

Lamport’s algorithm: safety proof

Let us suppose by contradiction that both p_i and p_j enter the CS

- $\Rightarrow$ both the processes have received an ack from any other process and, to enter the CS, the timestamp has to be the smallest in the queue
  - $t_i < t_j < ack_i.ts$
  - $t_j < t_i < ack_j.ts$



Both processes receive the ack when the two requests are in the queue but ME is guaranteed by the total order on the timestamps

p_j's ack arrives after p_j's request but before p_i's ack then p_i enters the CS without any problem and sends its ack after executing the CS

p_j ack arrives before p_j request then p_i enters the CS without any problem

Lamport's algorithm: properties

Fairness is satisfied: different requests are satisfied in the same order as they are generated

- Such order comes from the happened-before relation:
  - If two requests are in happened-before relation then they are satisfied in the same order.
  - If two request are concurrent with respect to the happended before relation then the access can happen in any order

Lamport’s algorithm: performances

Lamport’s algorithm needs 3(N-1) messages for the CS execution

- N-1 requests
- N-1 acks
- N-1 releases

In the best case (none is in the CS and only one process ask for the CS) there is a delay (from the request to the access) of 2 messages

Ricart-Agrawala's algorithm: implementation

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. $\forall$ i=1…N send REQUEST(num) to pi
4. Wait until #replies=n-1
5. State=CS
6. CS
7. $\forall$ r$\in$Q send REPLY to r
8. Q= $\varnothing$; State=NCS; #replies=0

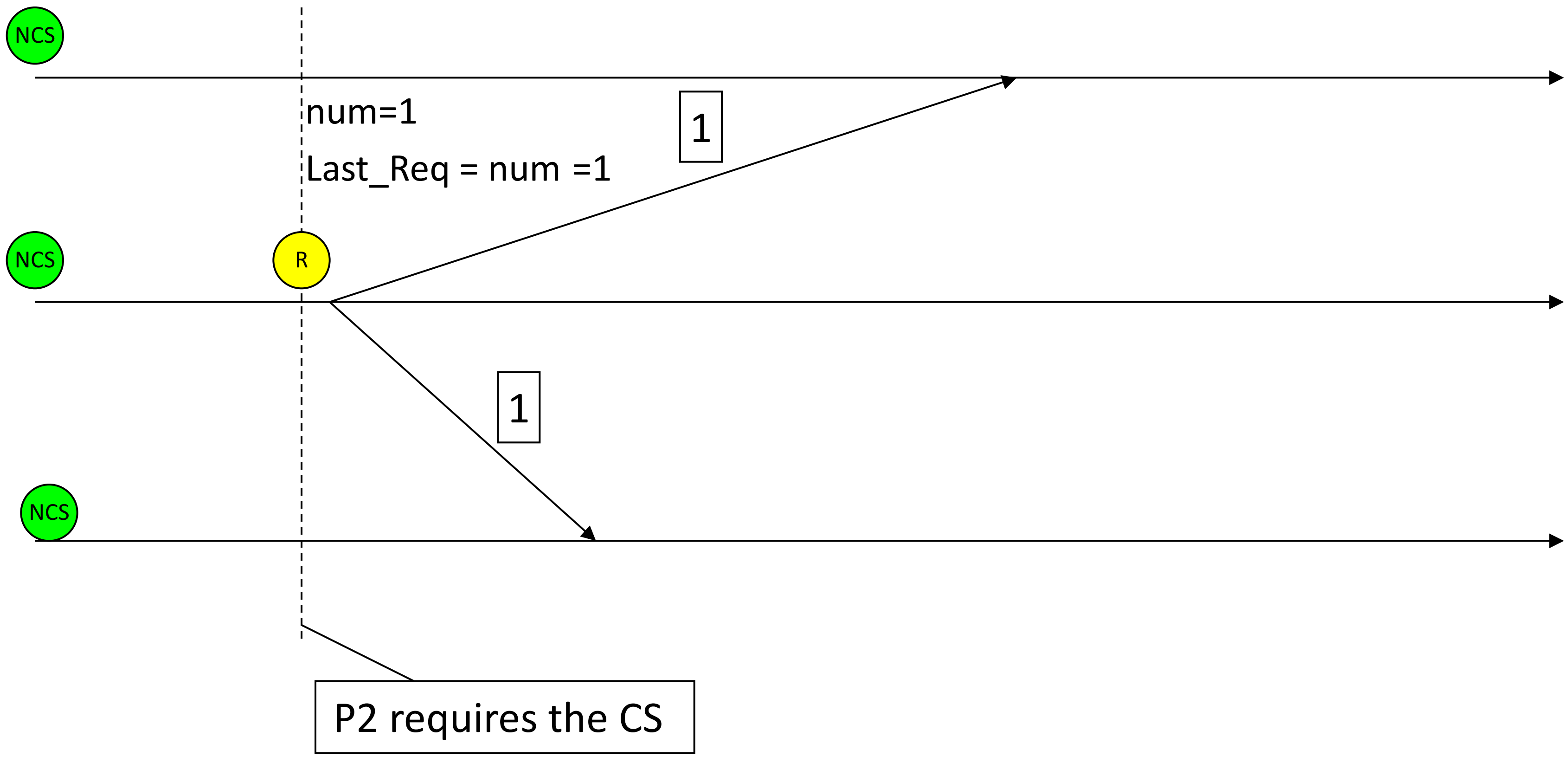
Upon receipt REQUEST(t) from pj

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

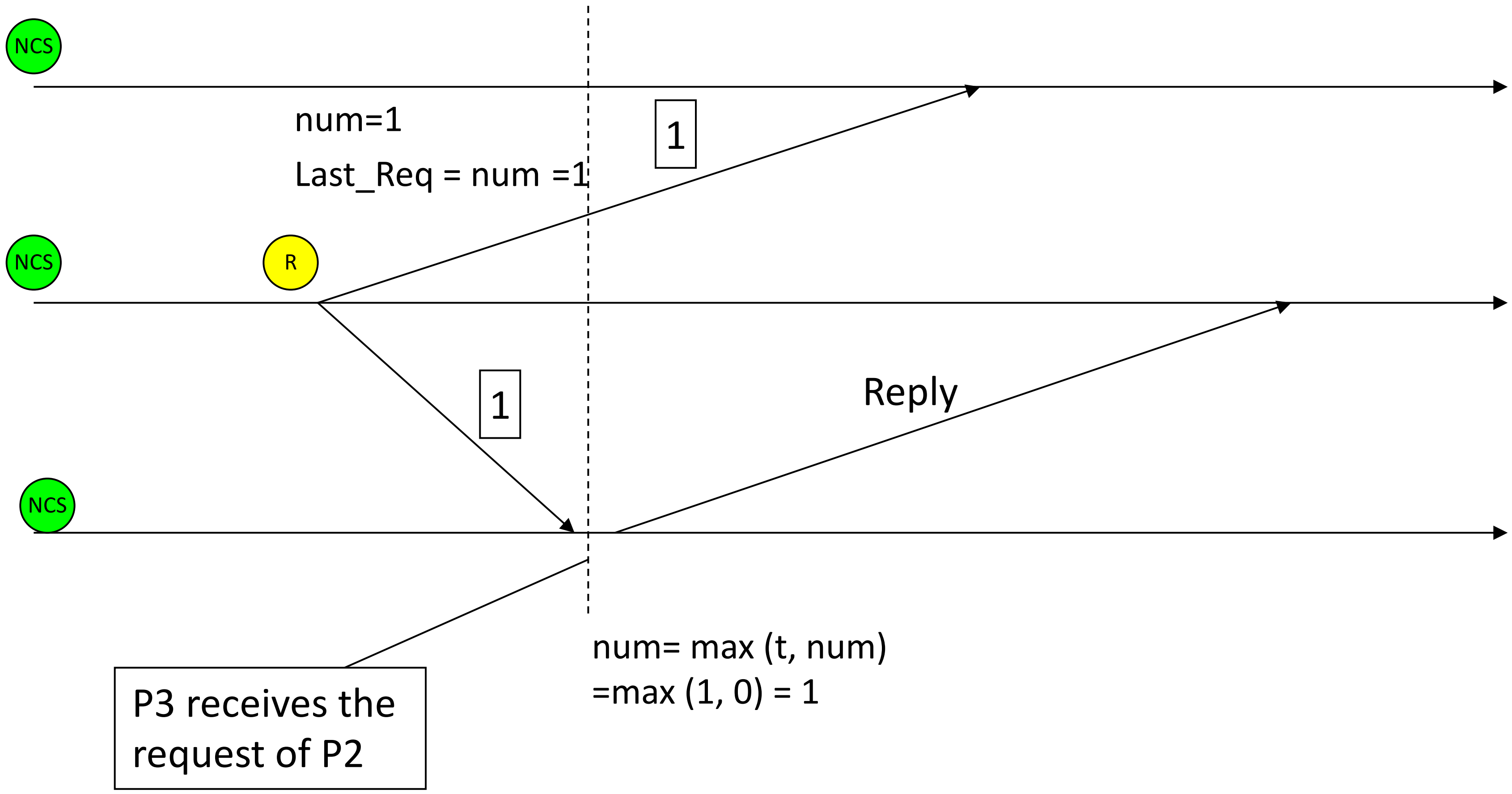
Upon receipt of REPLY from pj

1. #replies=#replies+1

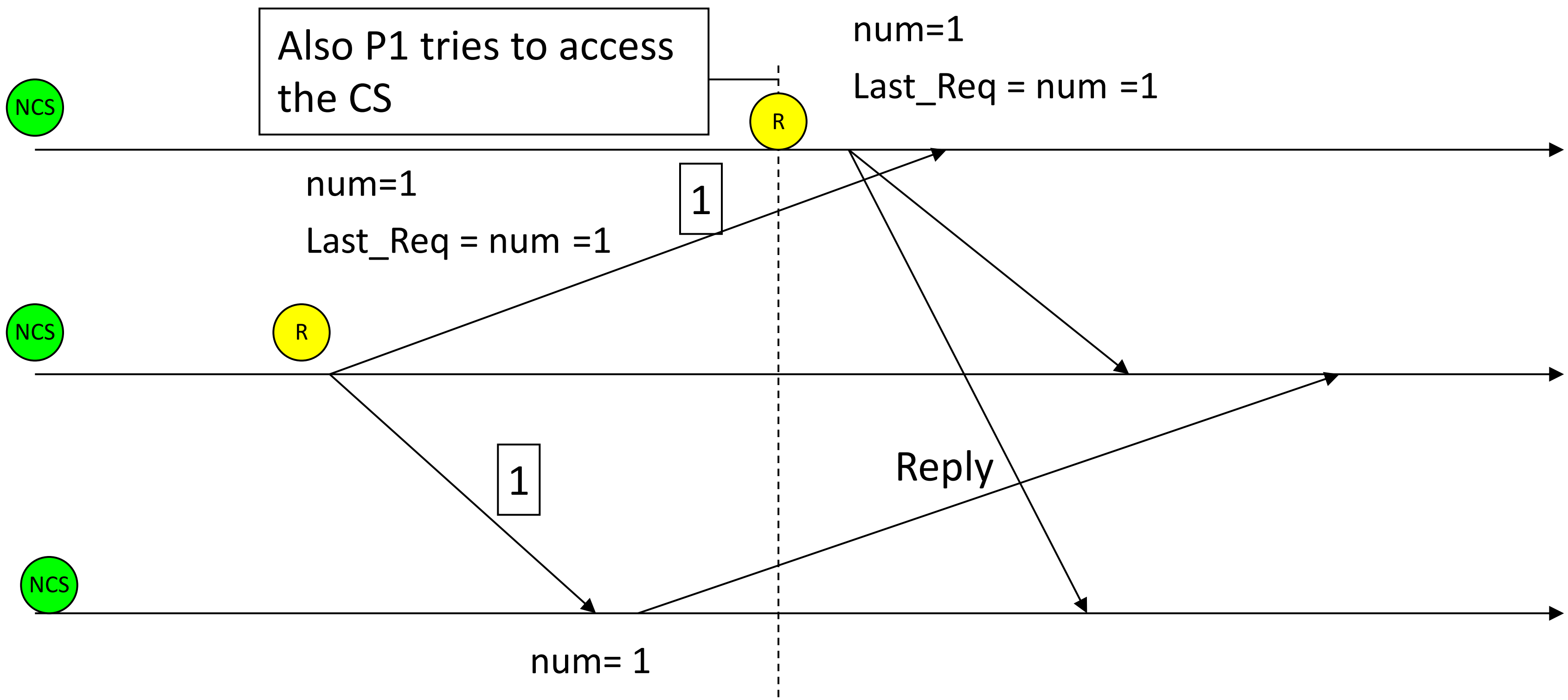
Ricart-Agrawala’s algorithm: example



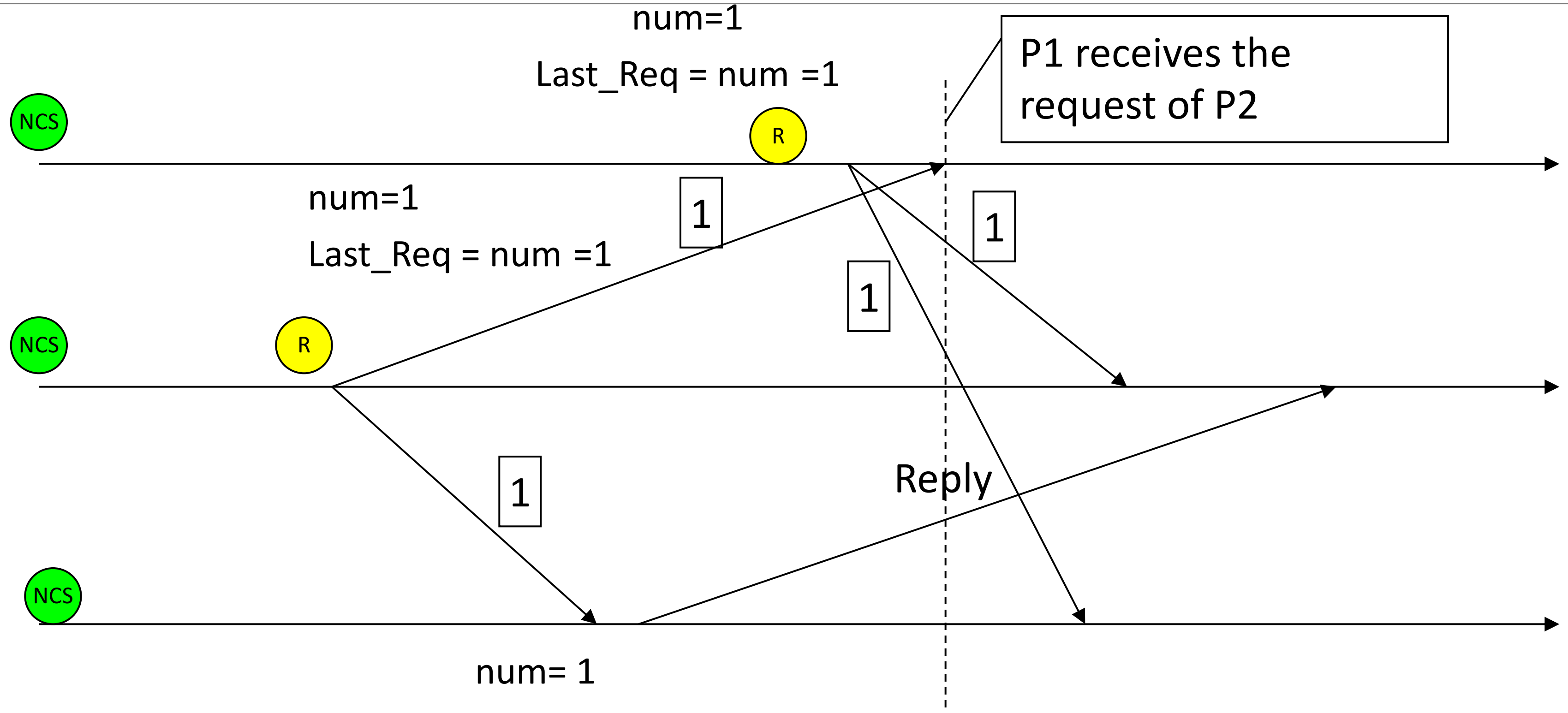
Ricart-Agrawala's algorithm: example



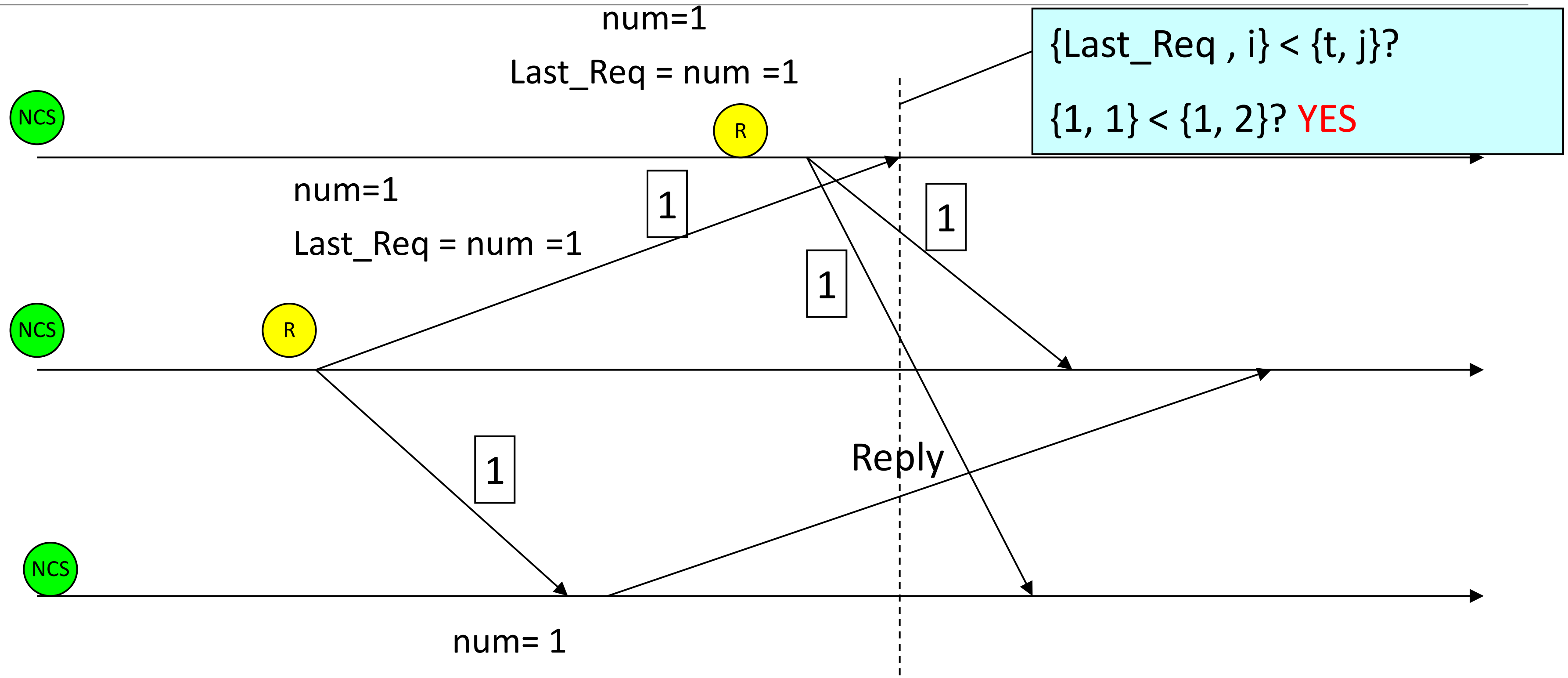
Ricart-Agrawala’s algorithm: example



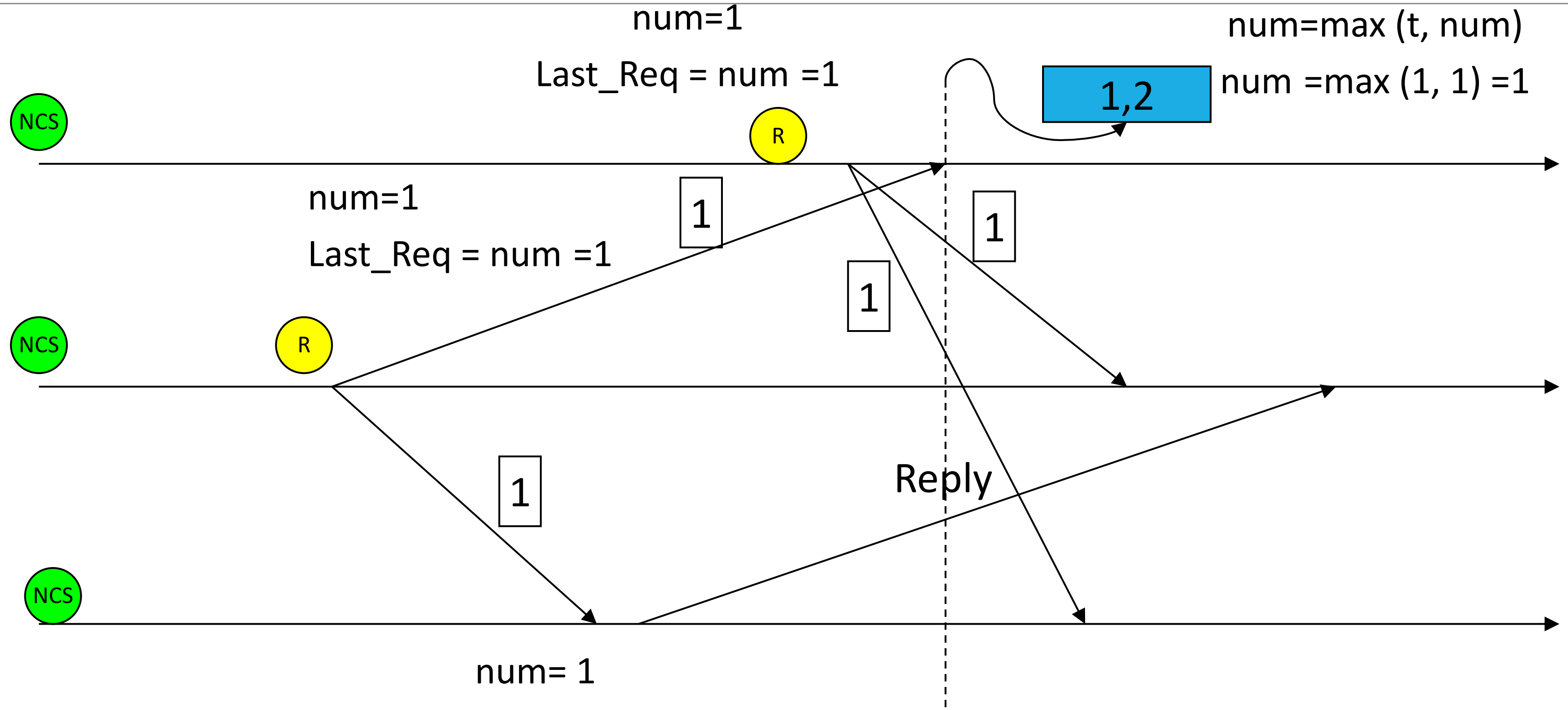
Ricart-Agrawala's algorithm: example



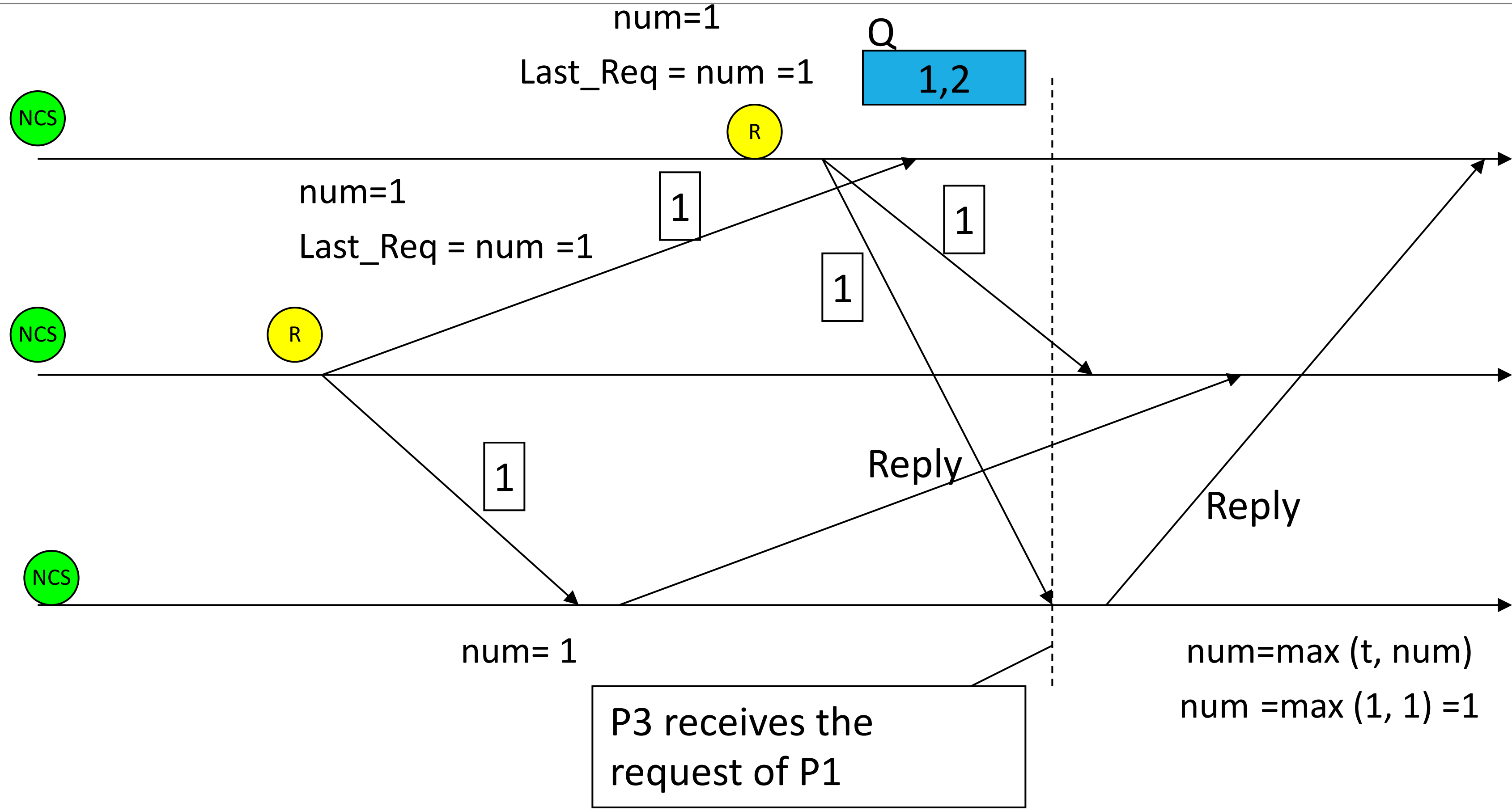
Ricart-Agrawala's algorithm: example



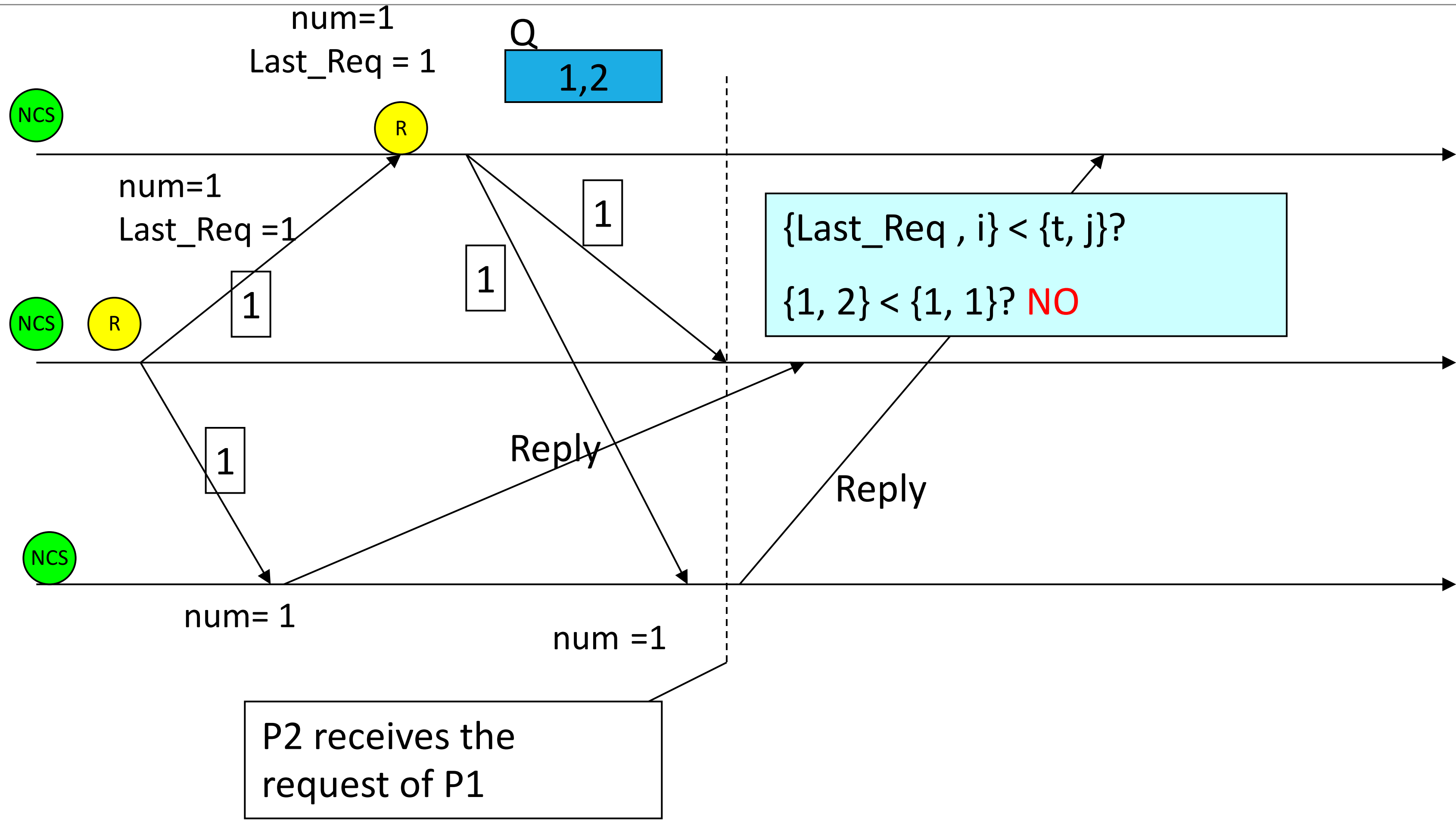
Ricart-Agrawala's algorithm: example



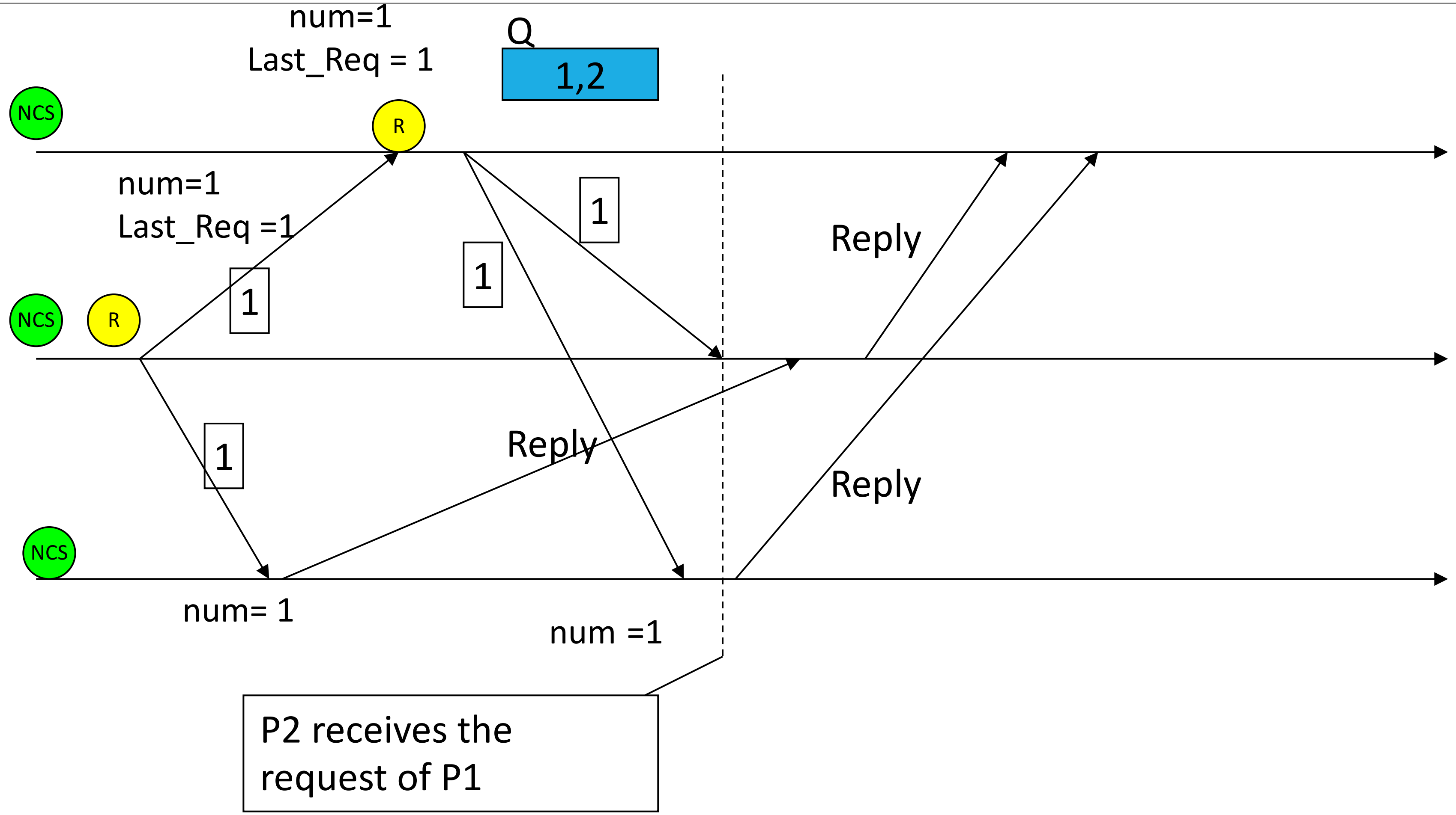
Ricart-Agrawala's algorithm: example



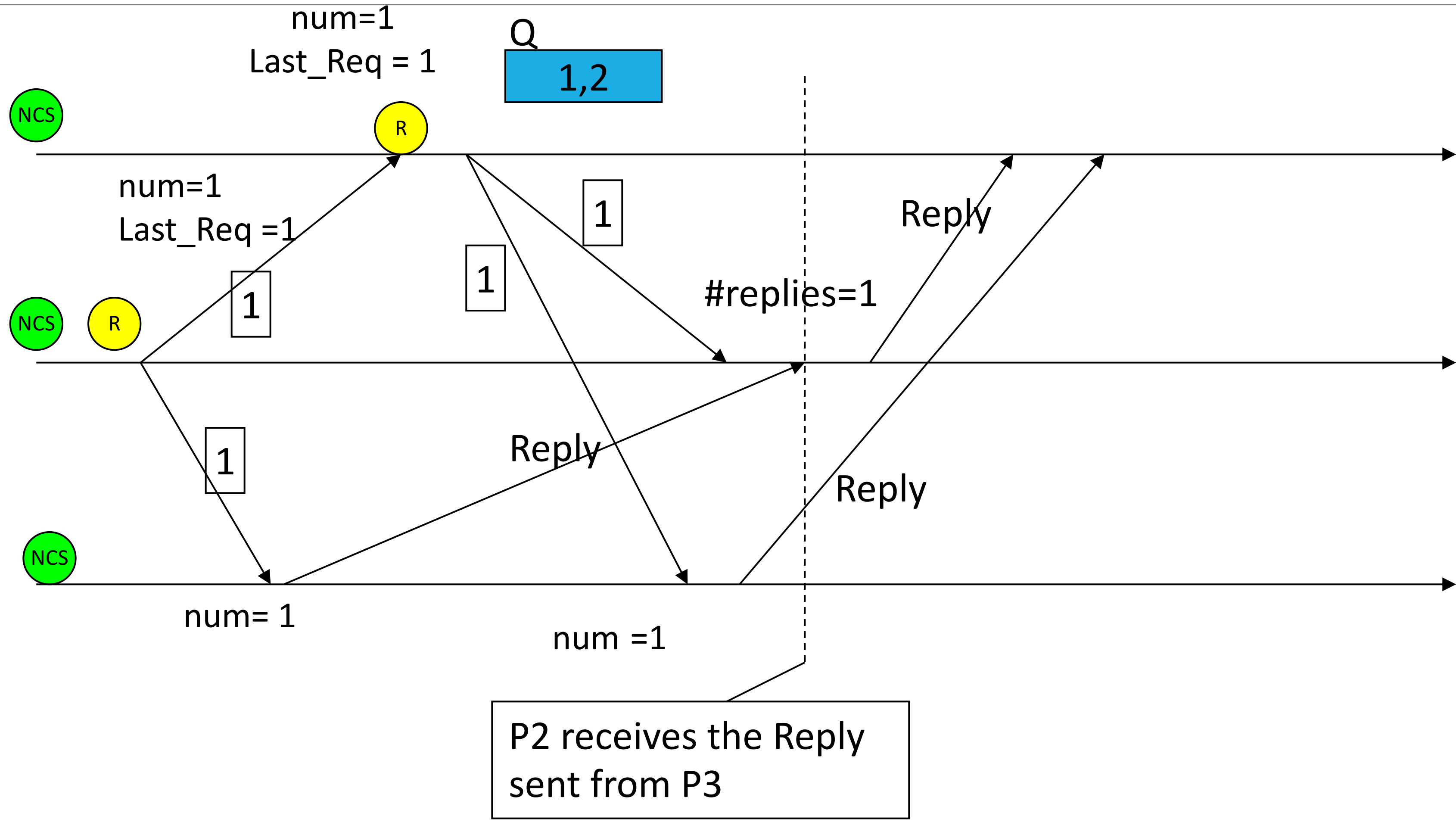
Ricart-Agrawala's algorithm: example



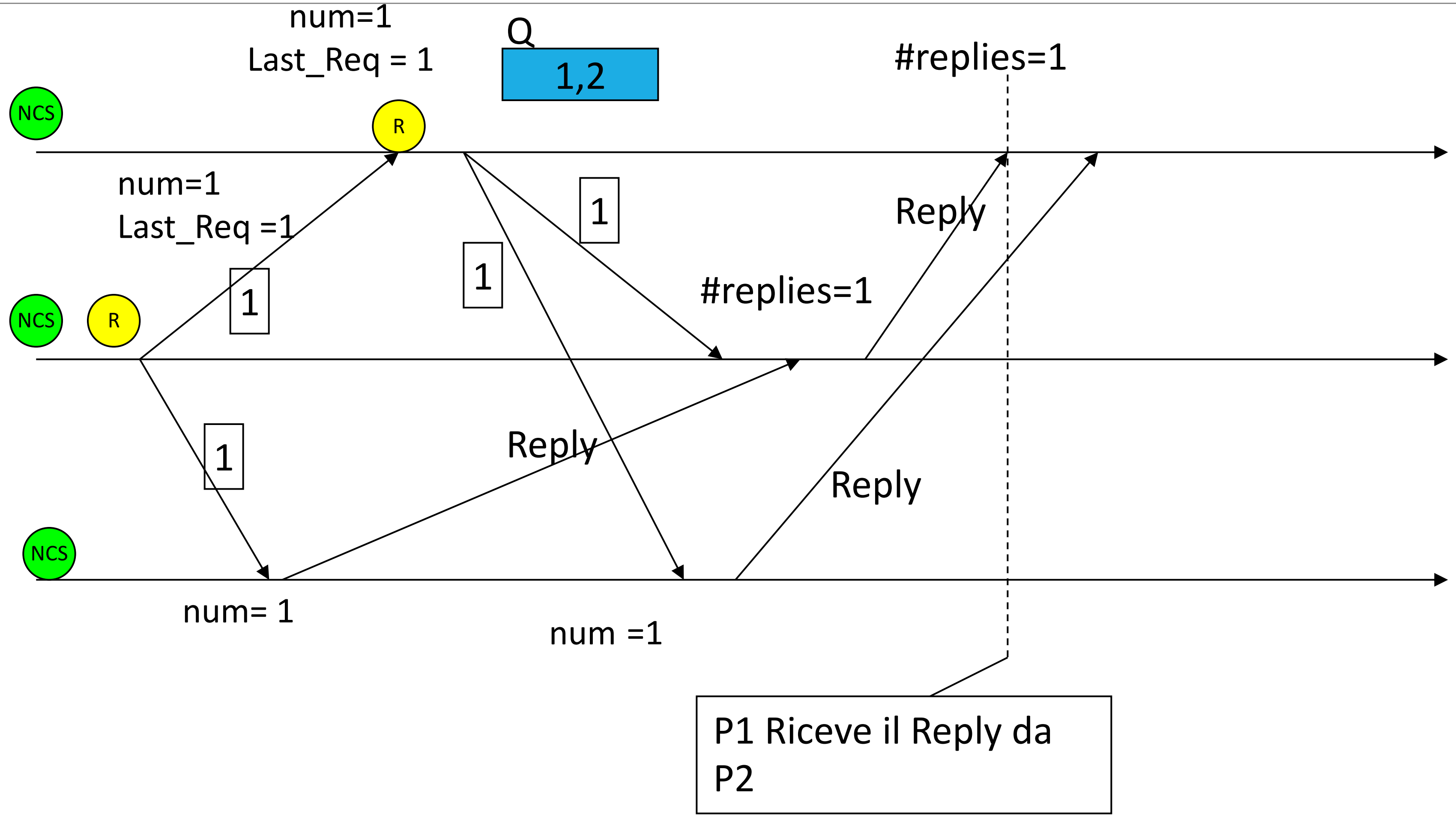
Ricart-Agrawala's algorithm: example



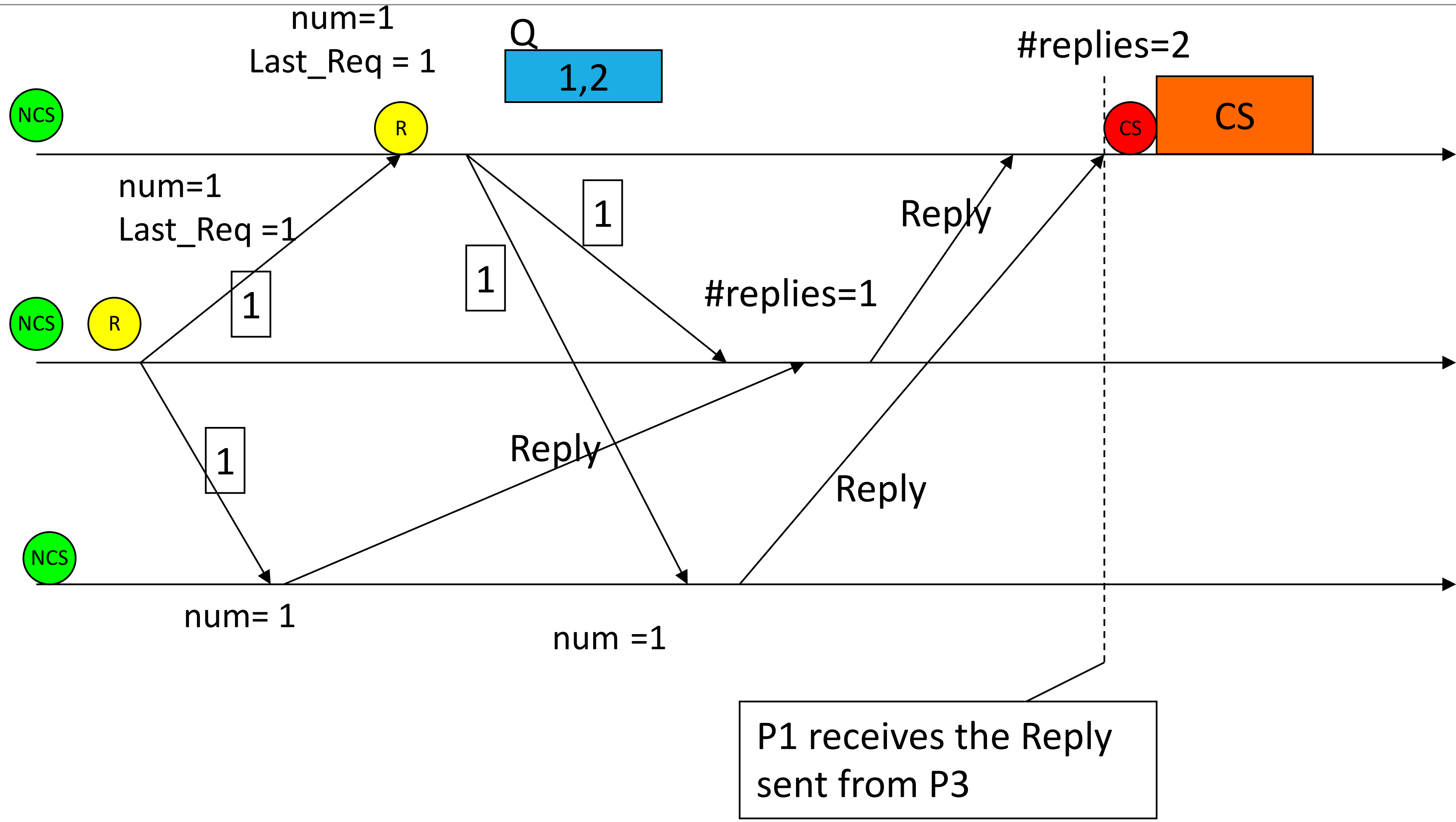
Ricart-Agrawala’s algorithm: example



Ricart-Agrawala’s algorithm: example



Ricart-Agrawala's algorithm: example



Ricart-Agrawala’s algorithm: example