

Dependable Distributed Systems

Master of Science in Engineering in Computer Science

AA 2022/2023

LECTURE 4: LOGICAL CLOCK

Recap

Physical clock synchronization algorithms have the aim to coordinate processes to reach an agreement on a common notion of time

The accuracy of the synchronization is strongly dependent on the estimation of transmission delay

- **ISSUE**: it can be hard to find a good estimation

OBSERVATION

- In several applications it is not important when things happened but in which order they happened

We need to find a reliable way to order events without using clock synchronization!

Happened-Before relation

OBSERVATION

- 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 deliver event

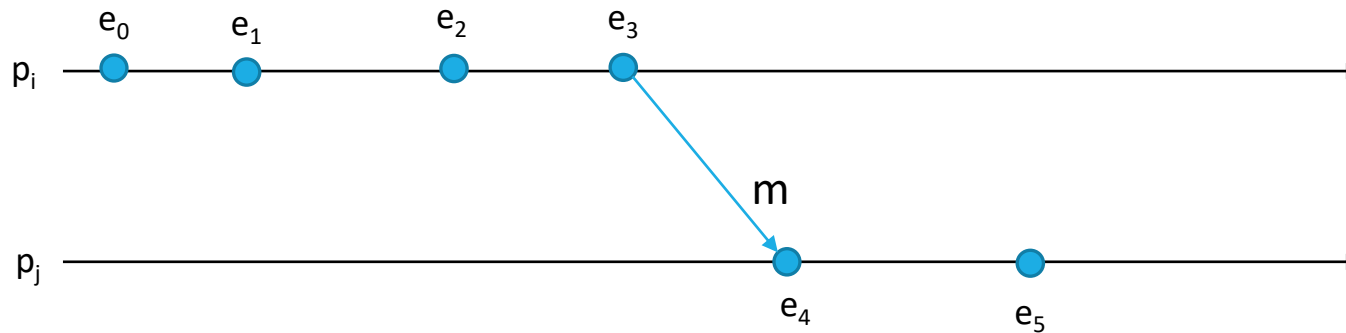
Lamport introduces the *happened-before relation* to capture causal dependencies between events (causal order relation)

- 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:

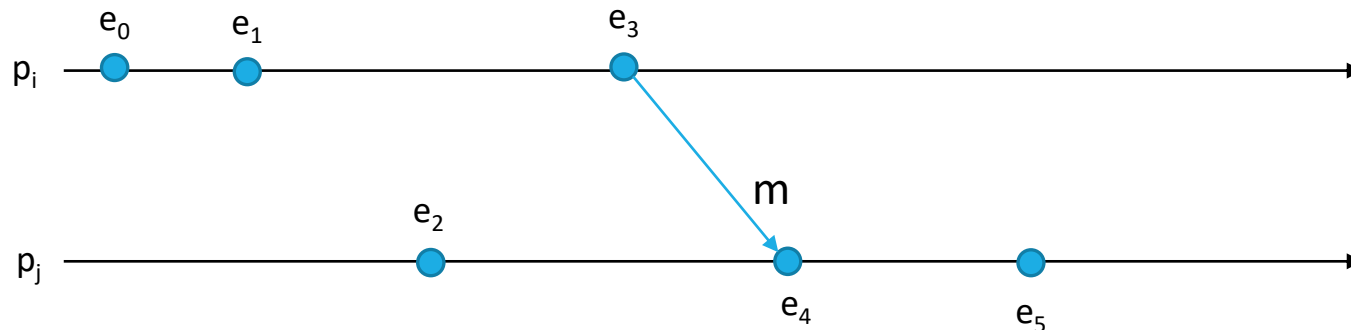
- $\exists p_i \mid e \rightarrow_i e'$
- $\exists m \mid e = \text{send}(m) \text{ and } e' = \text{deliver}(m)$
- $\exists e, e', e'' \mid (e \rightarrow e'') \wedge (e'' \rightarrow e')$ (happened-before relation is transitive)



Happened-Before Relation

OBSERVATIONS

- Happened-before relation imposes a partial order over events of the execution history
 - It may exist a pair of events $\langle e_i, e_j \rangle$ such that e_i and e_j are not in happened-before relation
 - If e_i and e_j are not in happened-before relation then they are concurrent ($e_i || e_j$)
- For any pair of events e_i and e_j in a distributed system only one of the following holds
 - $e_i \rightarrow e_j$,
 - $e_j \rightarrow e_i$
 - $e_i || e_j$



Logical Clock

The Logical Clock, introduced by Lamport, is a software counter that *monotonically* increases its value

A logical clock L_i can be used to *timestamp* events

$ts_e = L_i(e)$ is the “logical” timestamp assigned by a process p_i to events e using its current logical clock

PROPERTY

- If $e \rightarrow e'$ then $ts_e < ts_{e'}$

Observation

- The ordering relation obtained through logical timestamps is only a partial order

Scalar Logical Clock: an implementation

Each process p_i initializes its logical clock $L_i=0$ ($\forall i = 1 \dots N$)

p_i increases L_i of 1 when it generates an event (either *send* or *receive*)

- $L_i = L_i + 1$

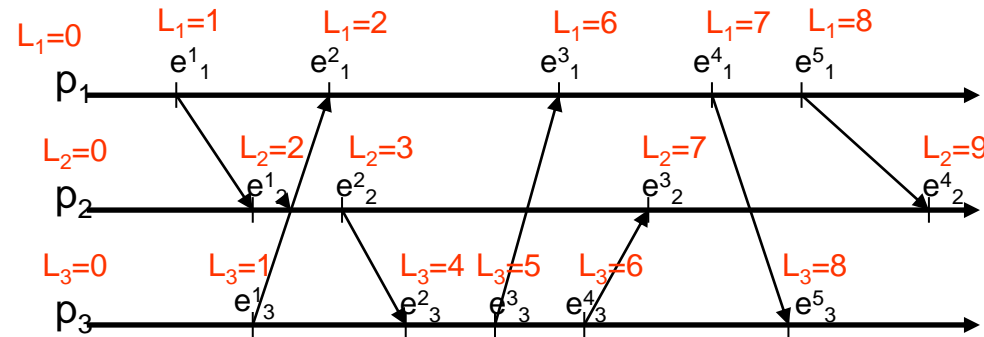
When p_i sends a message m

- creates an event *send*(m)
- increases L_i
- timestamps m with $ts=L_i$

When p_i receives a message m with timestamp ts

- Updates its logical clock $L_i = \max(ts, L_i)$
- 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
- $e^1_1 \parallel e^1_3$ 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 $ts_e < ts_{e'}$

But it is not possible to guarantee

- If $ts_e < ts_{e'}$, then $e \rightarrow e'$

Consequently:

- Using scalar logical clocks, it is not possible to determine if two events are concurrent or related 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 the timestamp will be an integer vector (i.e., an array of integer)

Vector Clock allows nodes to order events in happens-before just looking at their 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

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

p_i increases $V_i[i]$ of 1 when it generates an event

- $V_i[i] = V_i[i] + 1$

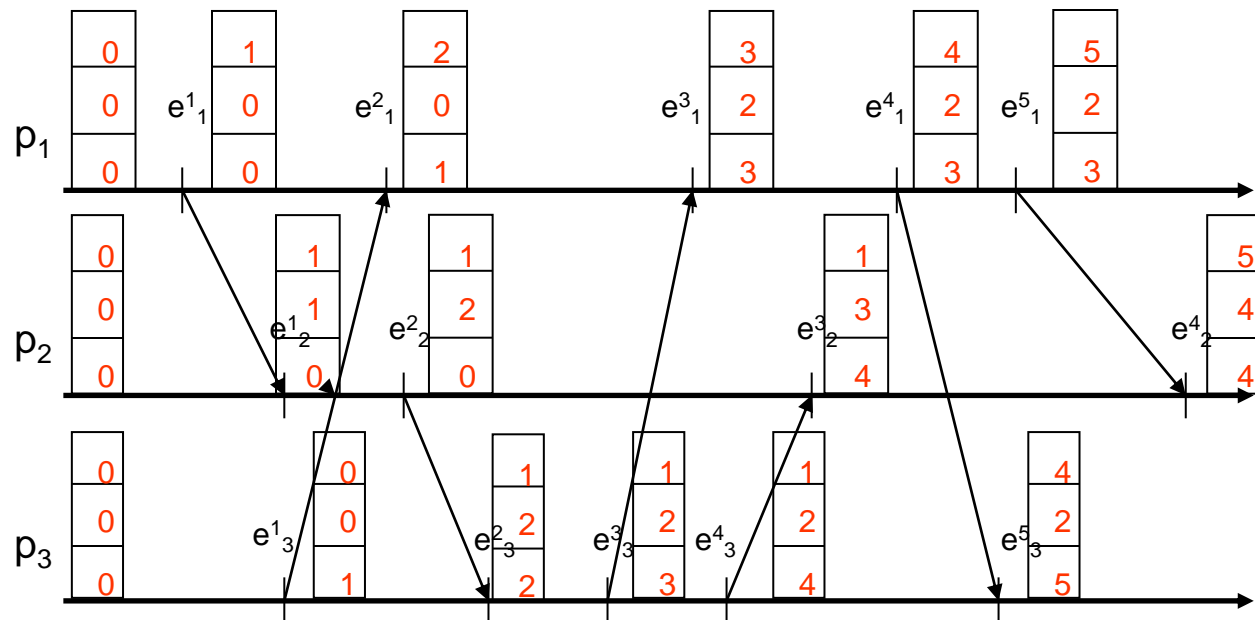
When p_i sends a message m

- Creates an event *send*(m)
- Increases V_i
- timestamps m with $ts = V_i$

When p_i receives a message m containing timestamp ts

- Updates its logical clock $V_i[j] = \max(ts[j], V_i[j]) \quad \forall j = 1 \dots N$
- 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 know

$V = V'$ if and only if

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

$V \leq V'$ if and only if

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

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

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

A comparison of Vector Clocks

1
0
0

V

1
1
0

V'

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

1
0
0

V

0
0
1

V'

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

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

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

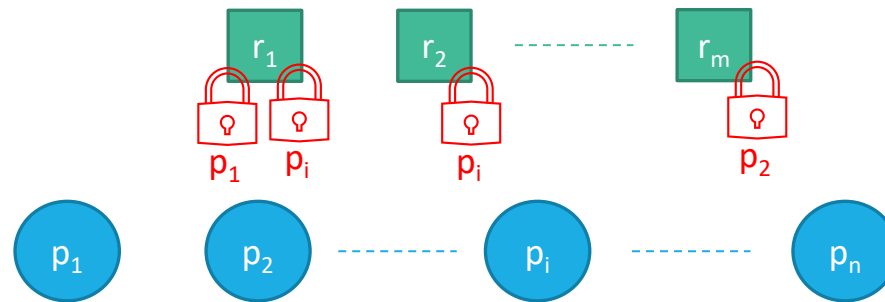
- Scalar Timestamp → Lamport's Mutual Exclusion
- Vector Timestamp → Causal Broadcast

Distributed Mutual Exclusion

The Mutual Exclusion Problem

Let us consider

- a set of processes $\Pi = \{p_1, p_2, \dots, p_n\}$
- a set of resources $R = \{r_1, r_2, \dots, r_m\}$



PROBLEM

- Processes need to access resources exclusively and we need to design a distributed abstraction that allows them to coordinate to get access to resources

System Model

Let us consider

- a set of processes $\Pi = \{p_1, p_2, \dots p_n\}$
- a set of resources $R = \{r_1, r_2, \dots r_m\}$
 - For the sake of simplicity let us assume $|R| = 1$

The system is asynchronous

Processes are not going to fail (they will be always correct)

Processes communicate by exchanging messages on top of perfect point-to-point links

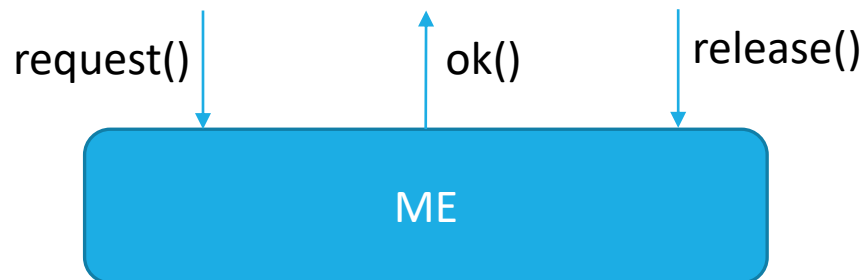
The Mutual Exclusion abstraction

EVENTS

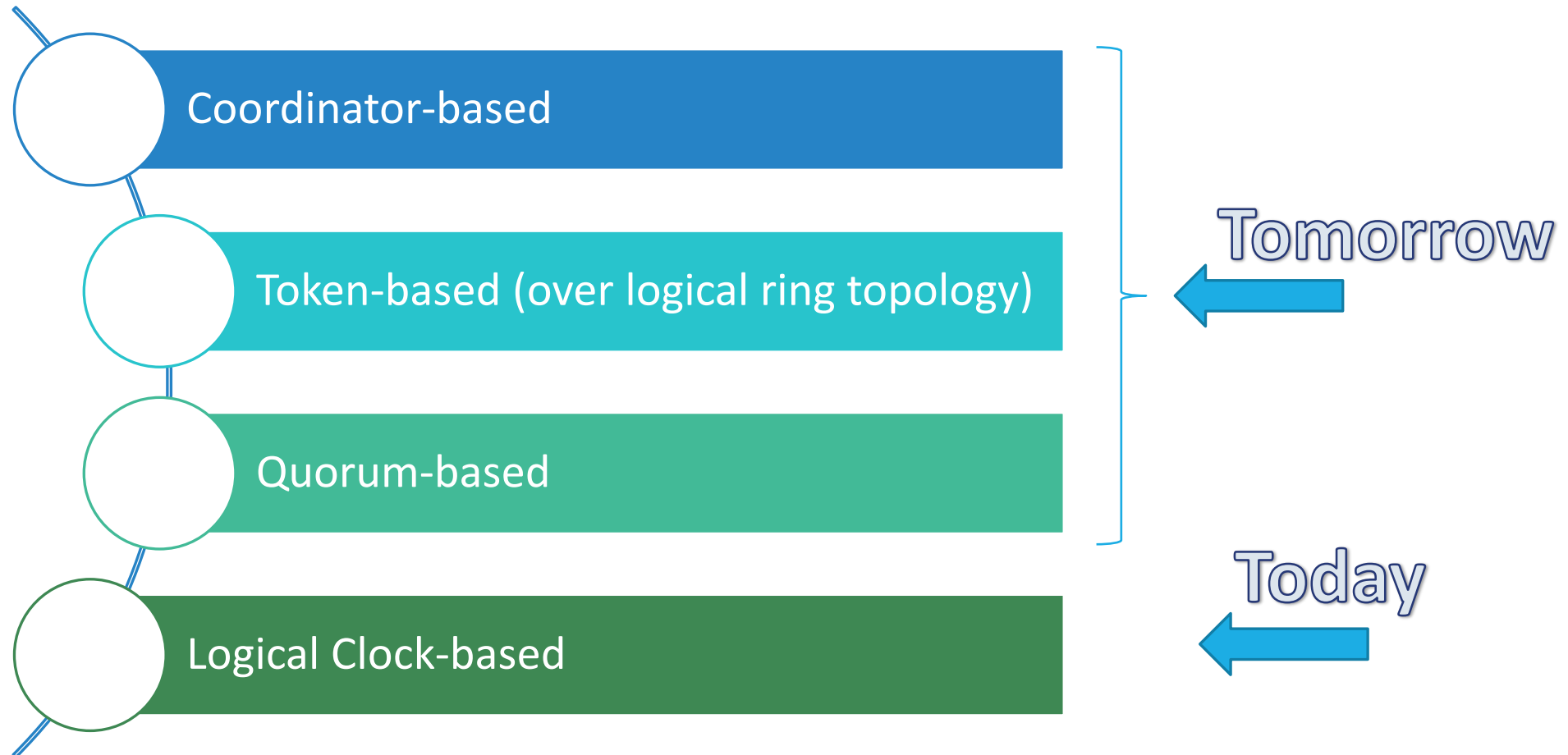
- `request()`: it issues a request to enter into the critical section
- `ok()`: it notifies the process that it can now access the critical section
- `release()`: it is invoked to leave the critical section and to allow someone else to enter

PROPERTIES

- **Mutual Exclusion**: at any time t , at most one process p is running the critical section
- **No-Deadlock**: there always exists a process p able to enter the critical section
- **No-Starvation**: every `request()` and `release()` operation eventually terminate



Different Approaches to Distributed Mutual Exclusion



Timestamp-based algorithm: Lamport's Distributed Mutual Exclusion

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 (timestamp)

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 p_i

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

Algorithm rules for a process p_i

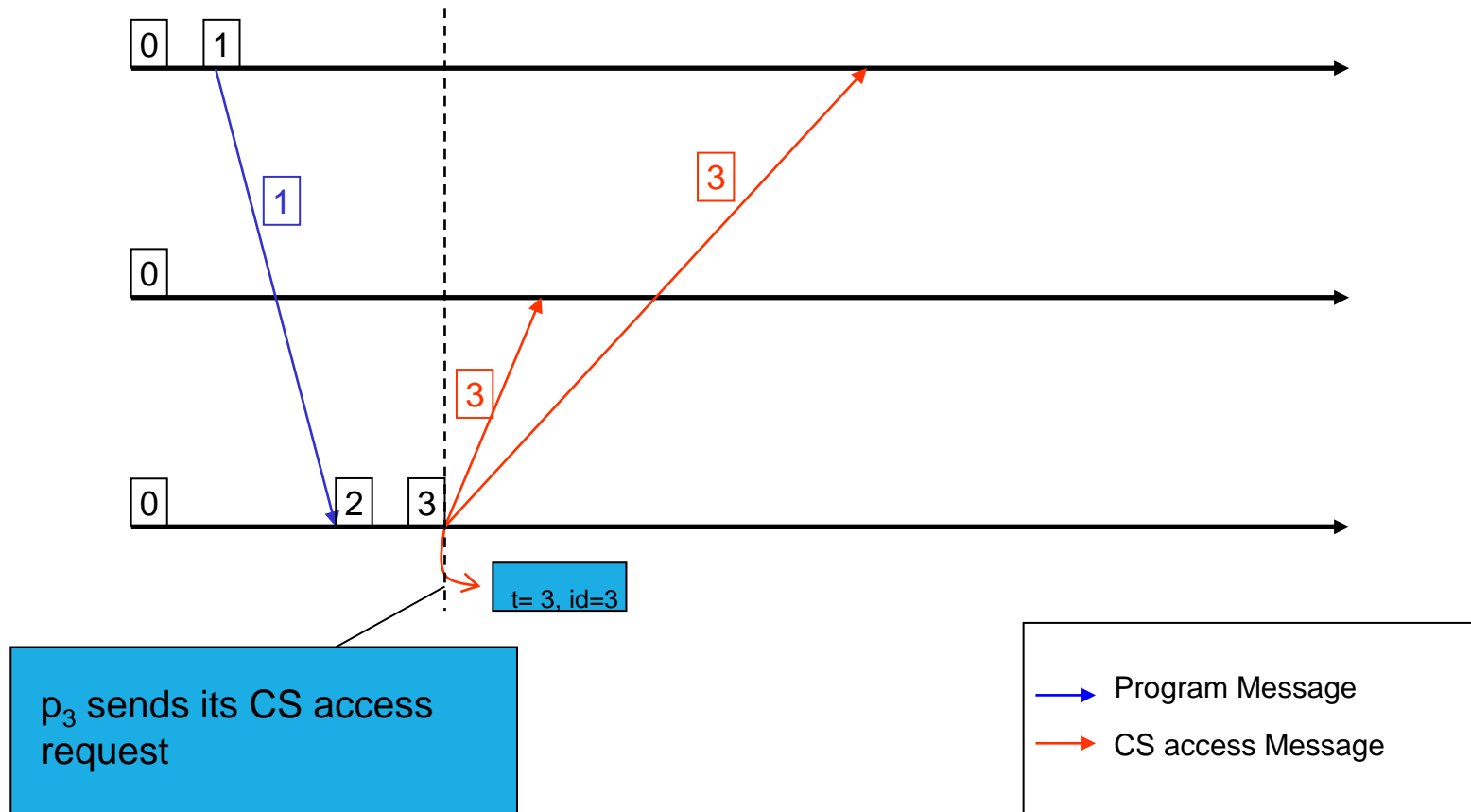
- **Request to access the CS**
 - p_i sends a request message, attaching ck , to all the other processes
 - p_i adds its request to Q
- **Request reception from a process p_j**
 - p_i puts p_j request (including the timestamp) in its queue
 - p_i sends back an ack to p_j

Lamport's algorithm: implementation

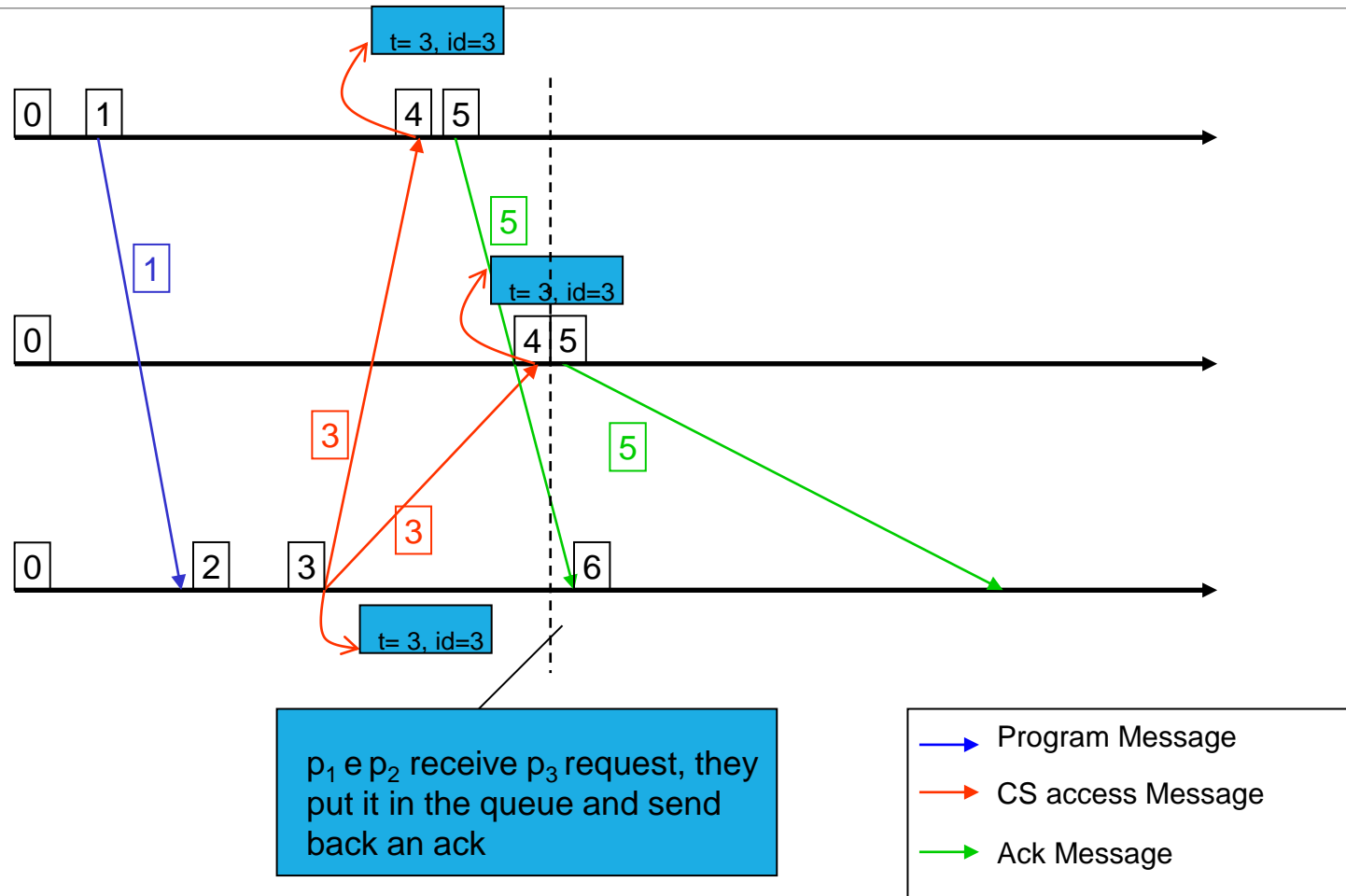
Algorithm rules for a process p_i

- **p_i enters the CS iff**
 1. p_i has, in its queue, a request with timestamp t
 2. t is the small timestamp in the queue
 3. p_i has already received an ack with timestamp 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 p_j**
 - p_i deletes p_j 's request from the queue

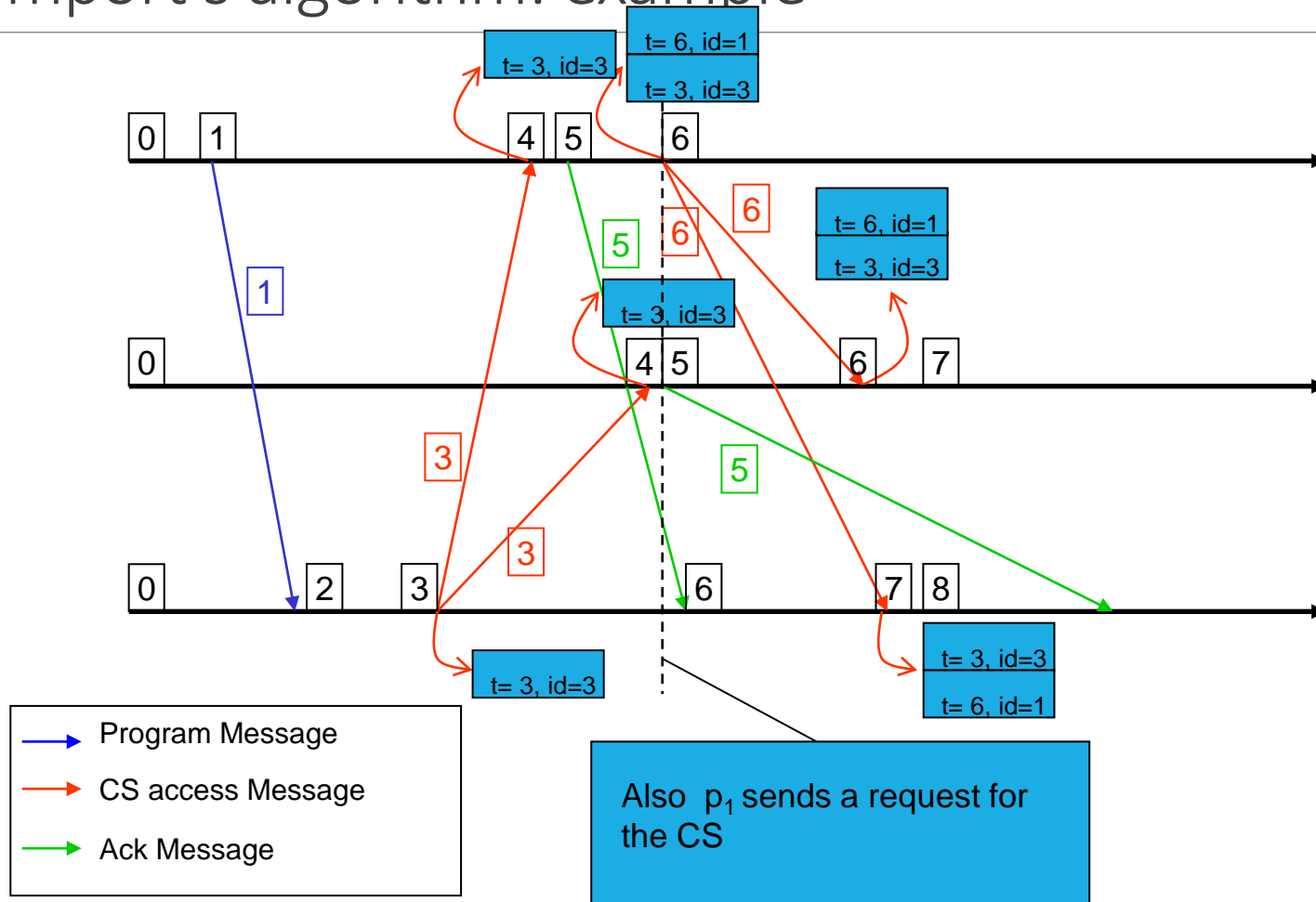
Lamport's algorithm: example



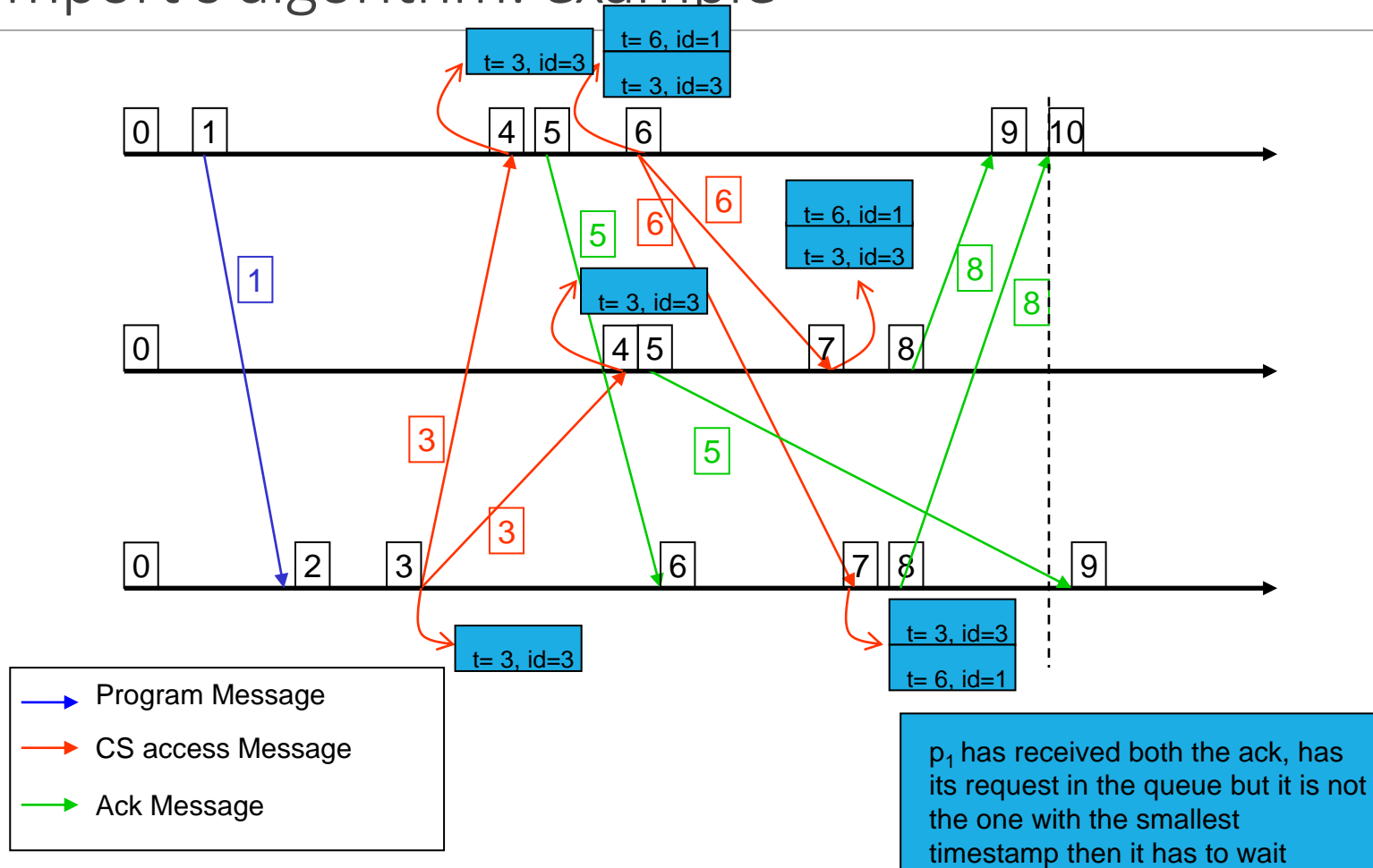
Lamport's algorithm: example



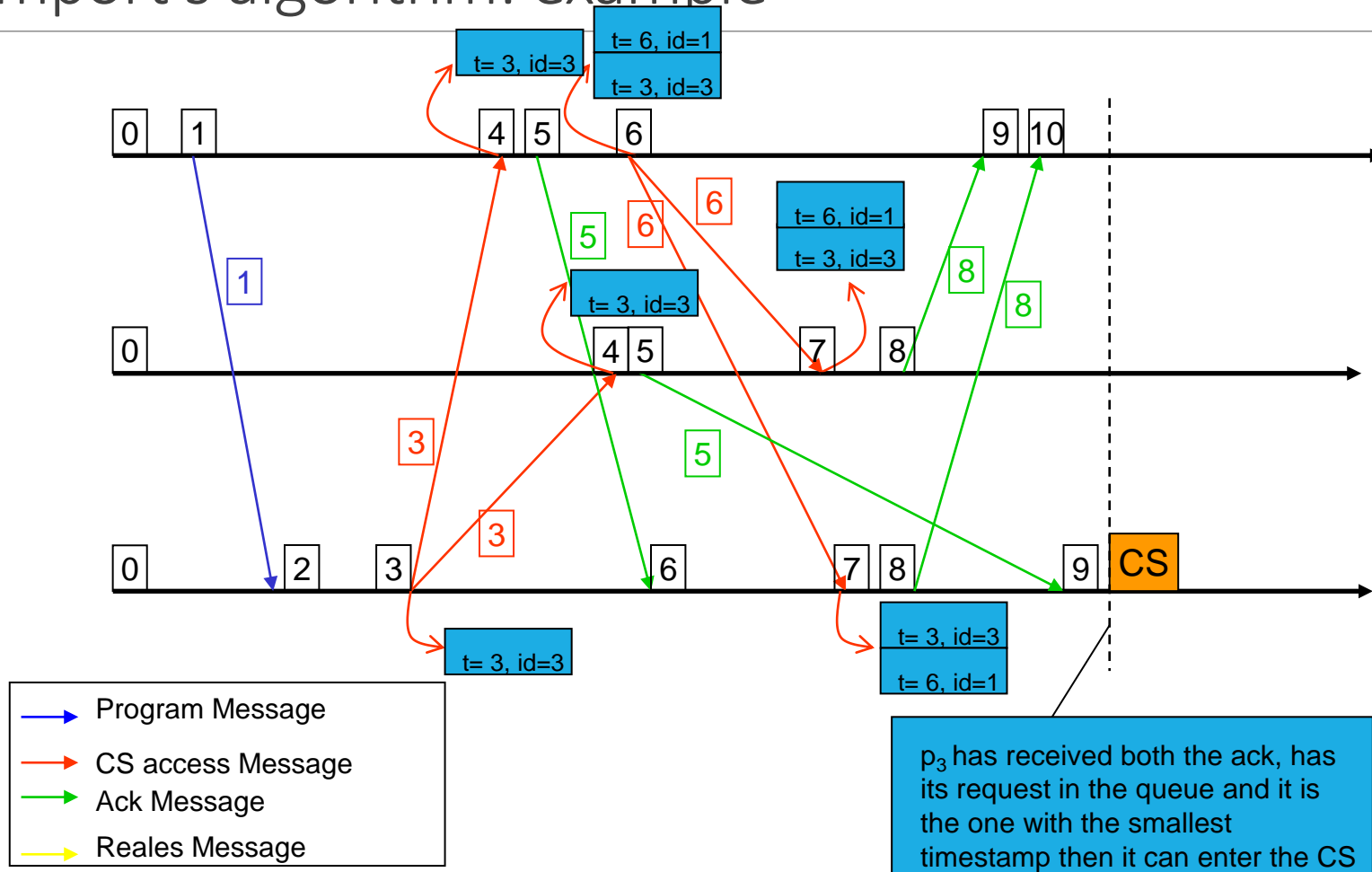
Lamport's algorithm: example



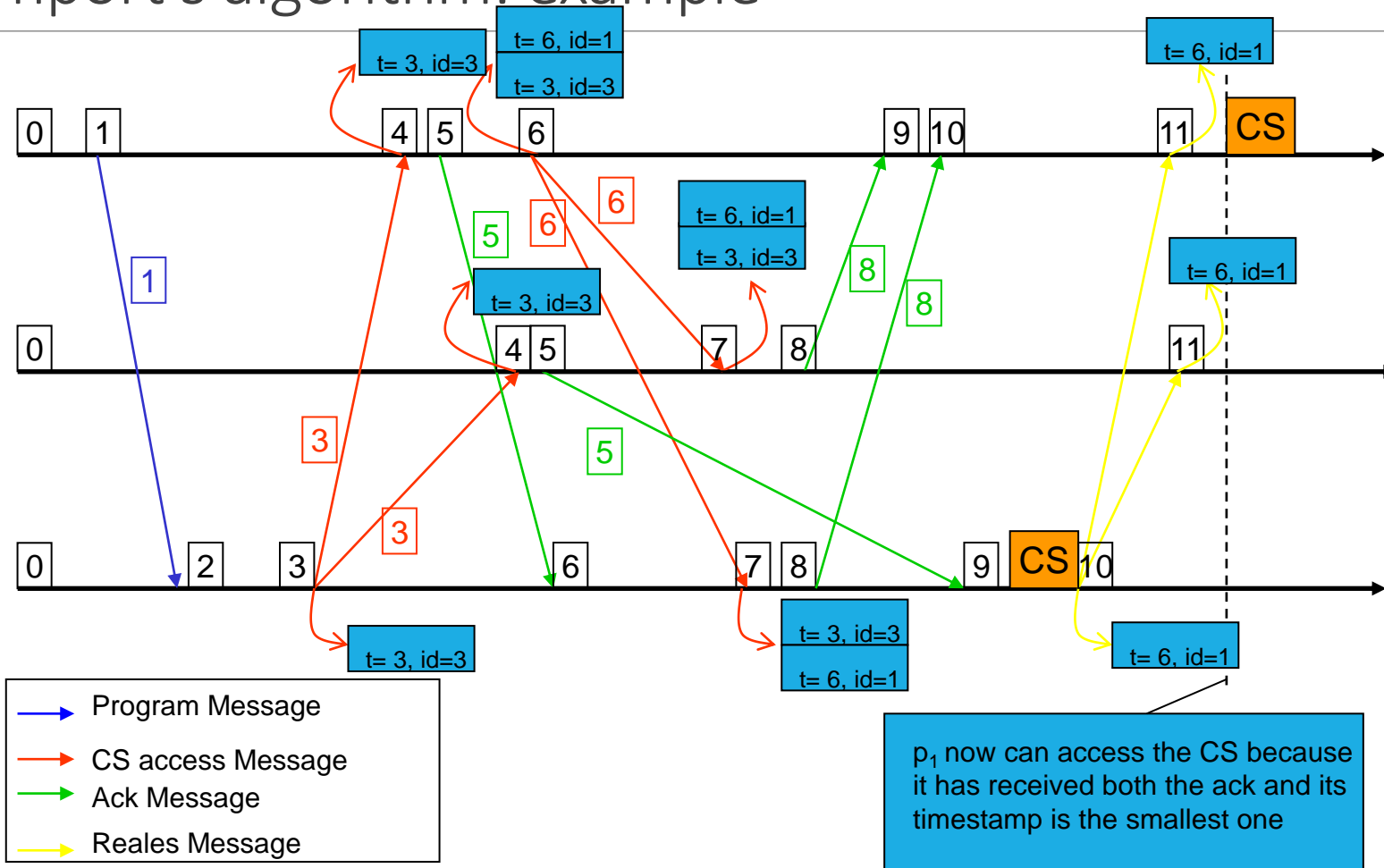
Lamport's algorithm: example



Lamport's algorithm: example



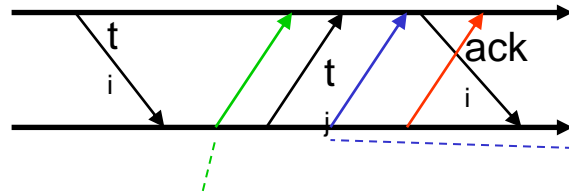
Lamport's algorithm: example



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
 - $t_i < t_j < \text{ack}_i.\text{ts}$
 - $t_j < t_i < \text{ack}_j.\text{ts}$



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

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

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 happened 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 $\in \{\text{Requesting}, \text{CS}, \text{NCS}\}$ (initially NCS)
- Q pending requests queue (initially empty)
- Last_Req
- Num

Algorithm

begin

1. State=Requesting
2. Num=num+1; Last_Req=num
3. $\forall i=1\dots N$ send REQUEST(num) to p_i
4. Wait until #replies=n-1
5. State=CS
6. CS
7. $\forall r \in Q$ send REPLY to r
8. $Q = \emptyset$; State=NCS; #replies=0

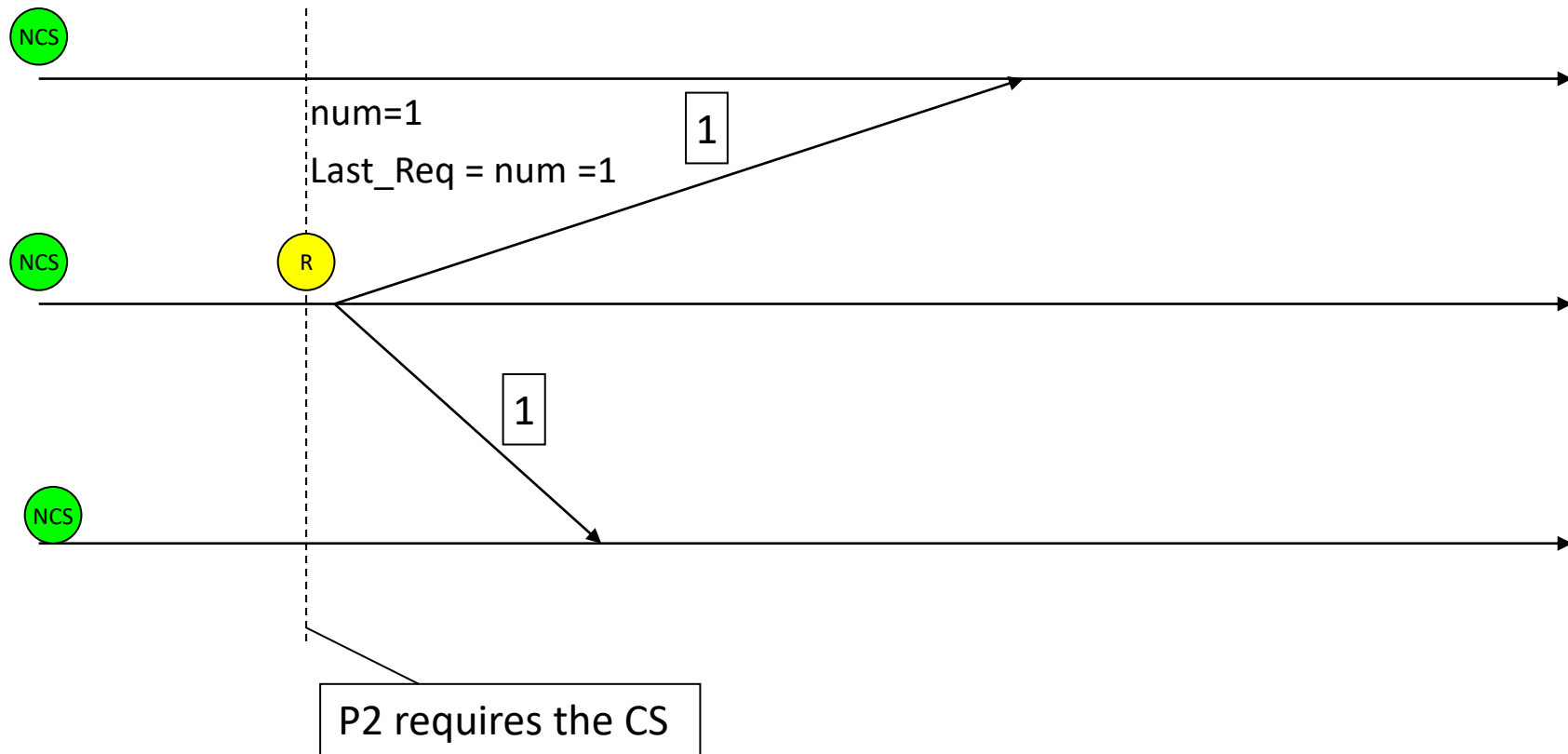
Upon receipt REQUEST(t) from p_j

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

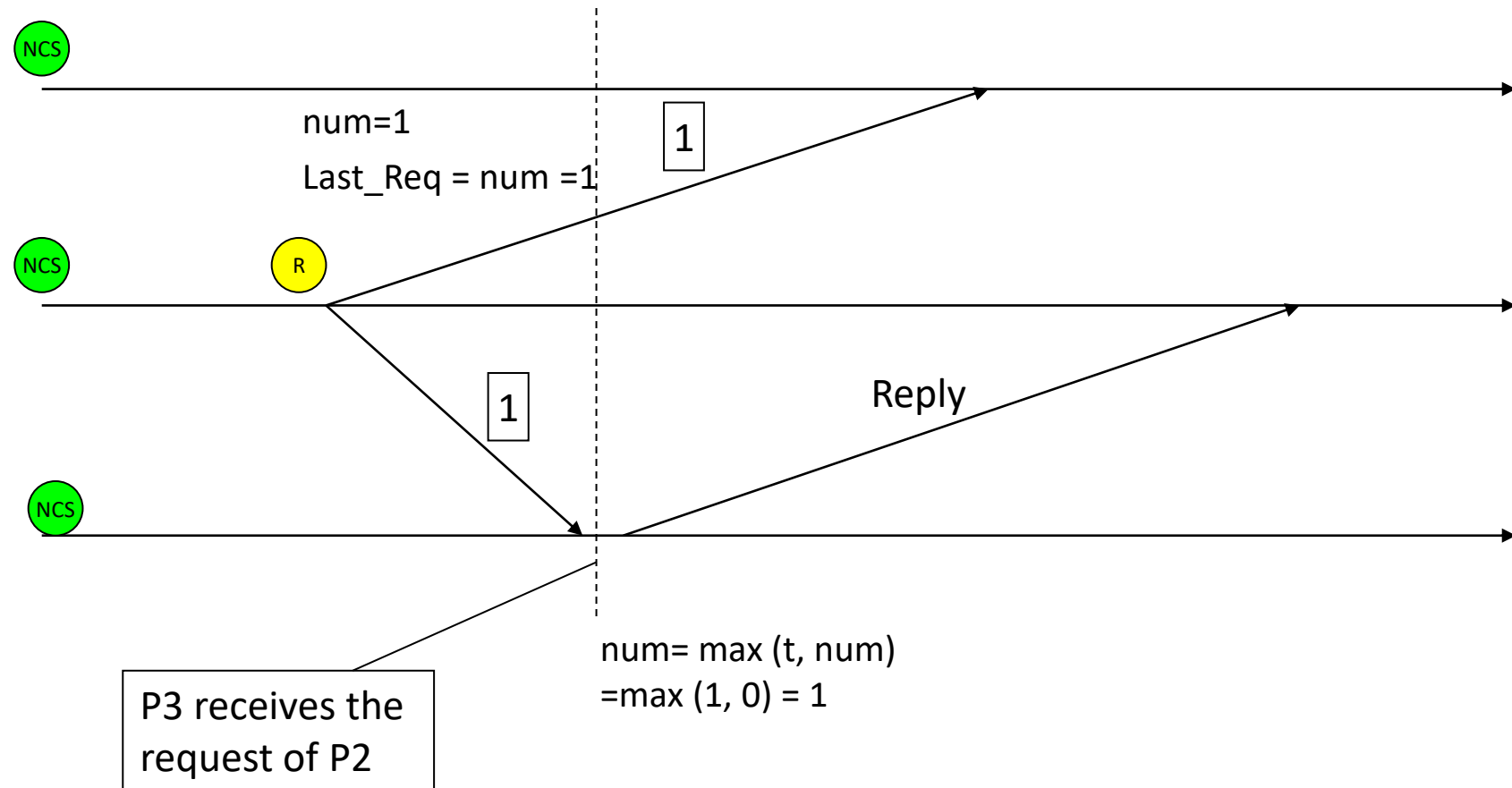
Upon receipt of REPLY from p_j

1. #replies=#replies+1

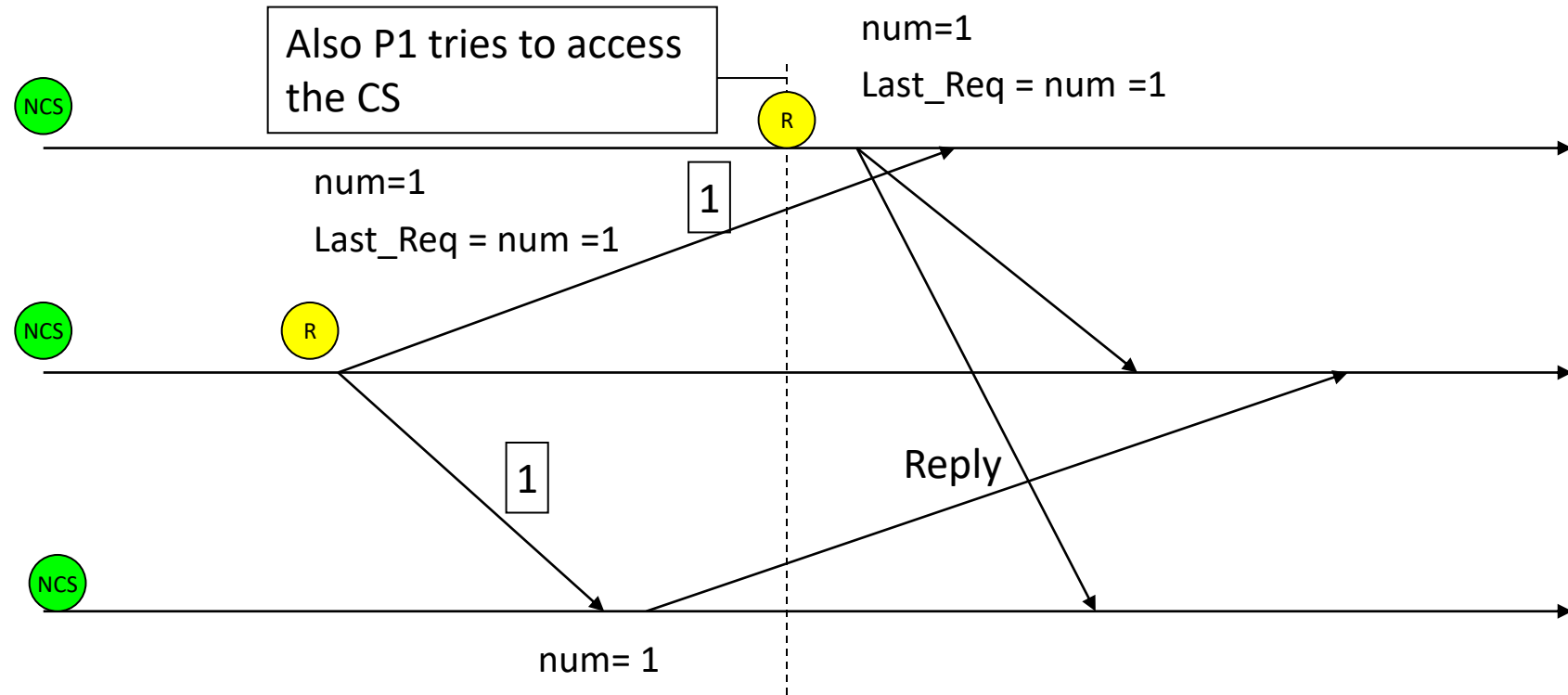
Ricart-Agrawala's algorithm: example



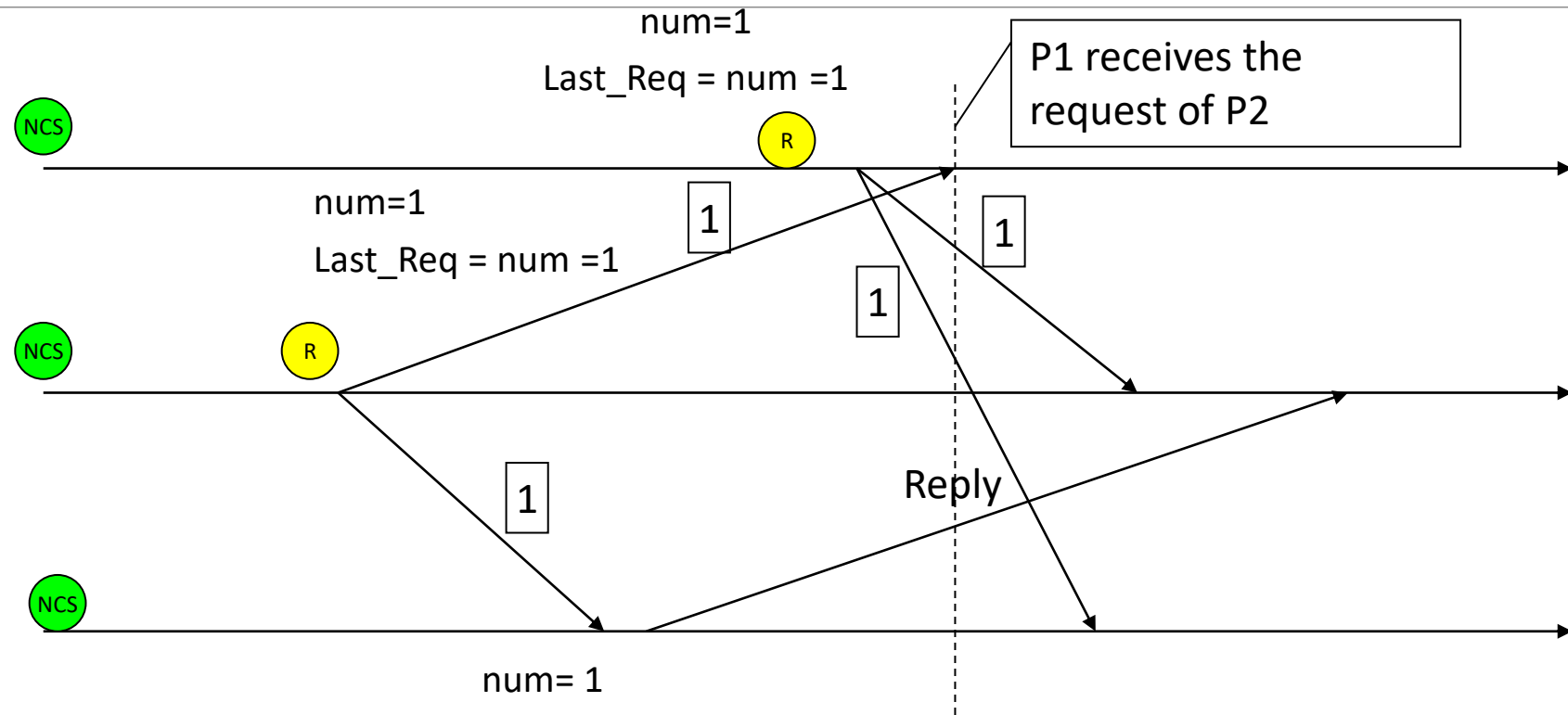
Ricart-Agrawala's algorithm: example



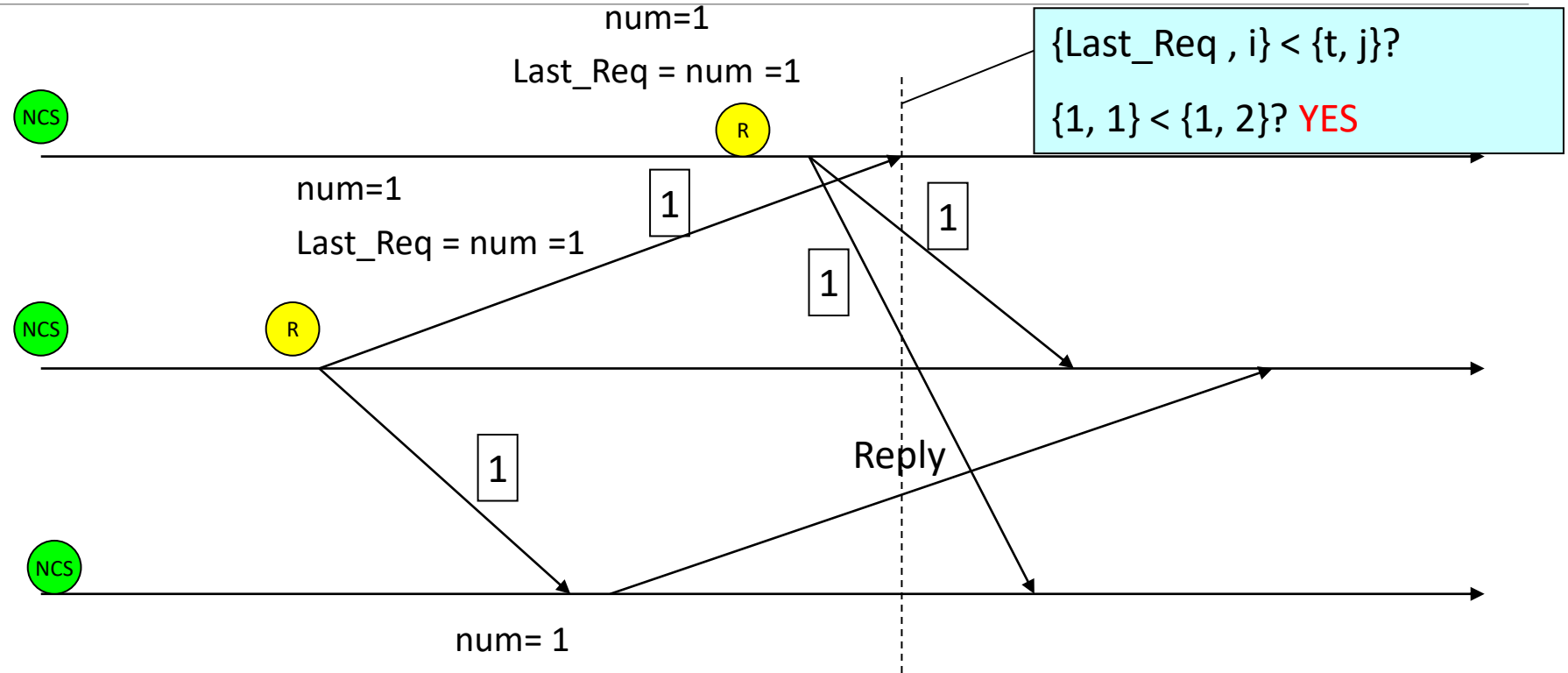
Ricart-Agrawala's algorithm: example



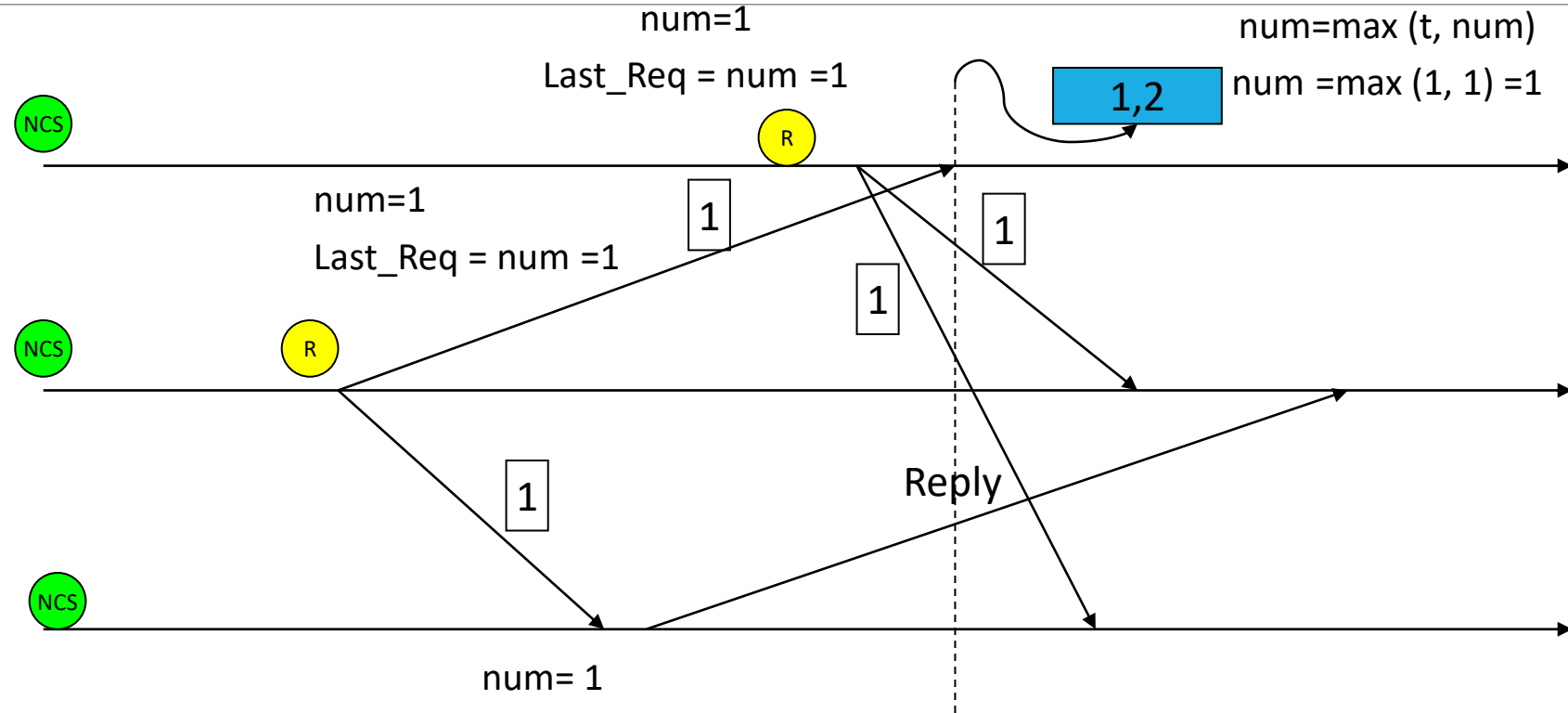
Ricart-Agrawala's algorithm: example



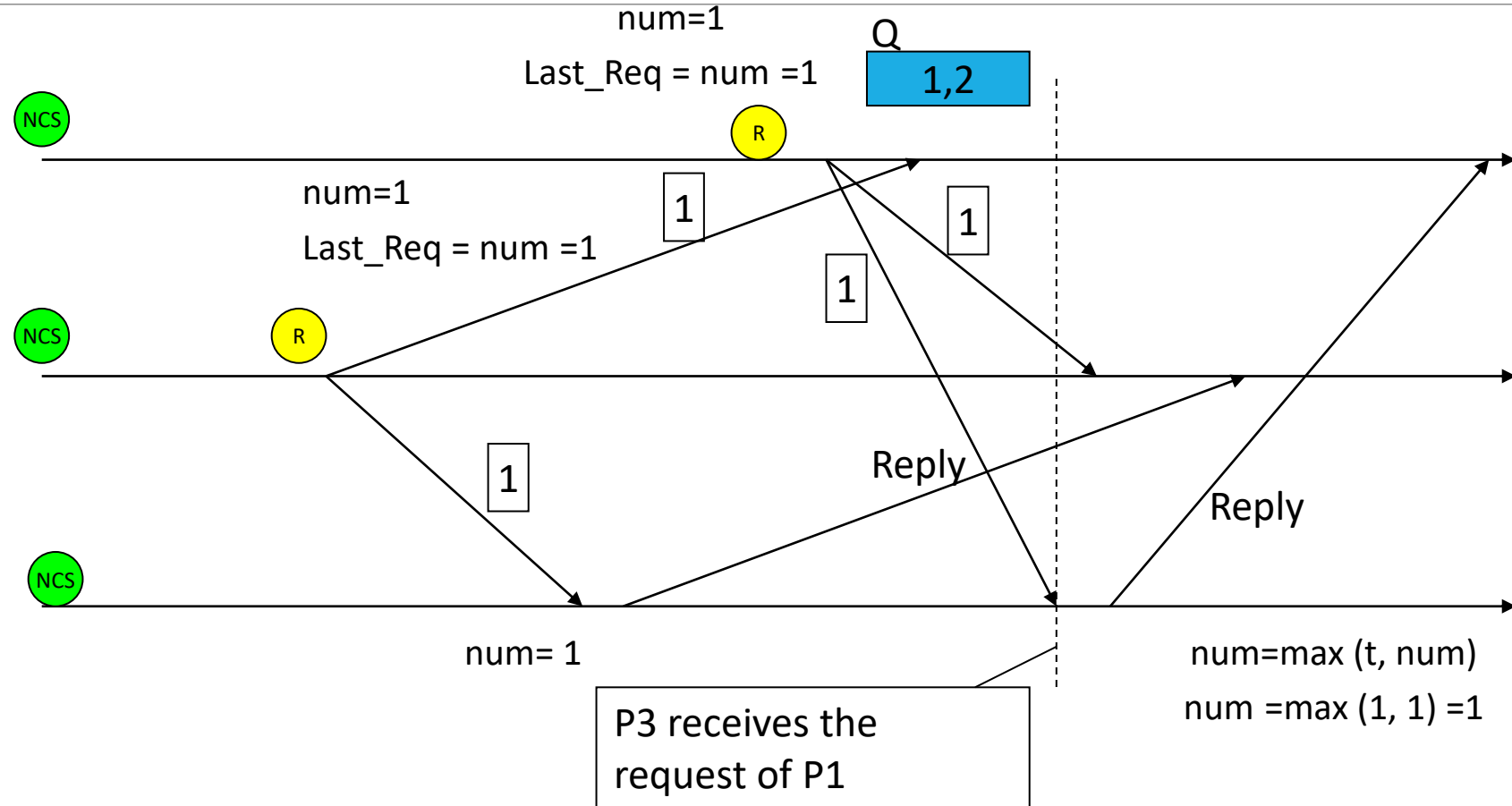
Ricart-Agrawala's algorithm: example



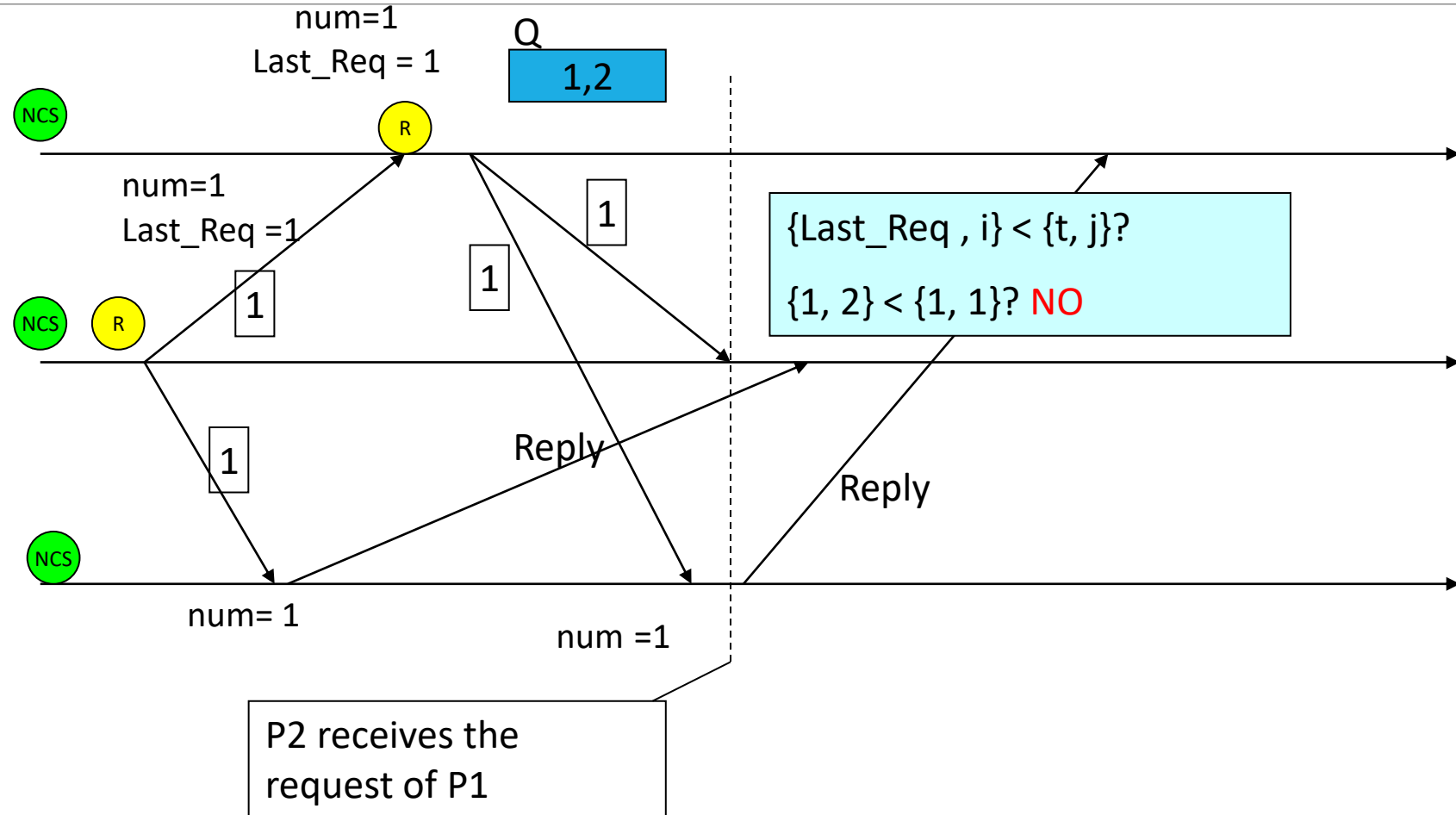
Ricart-Agrawala's algorithm: example



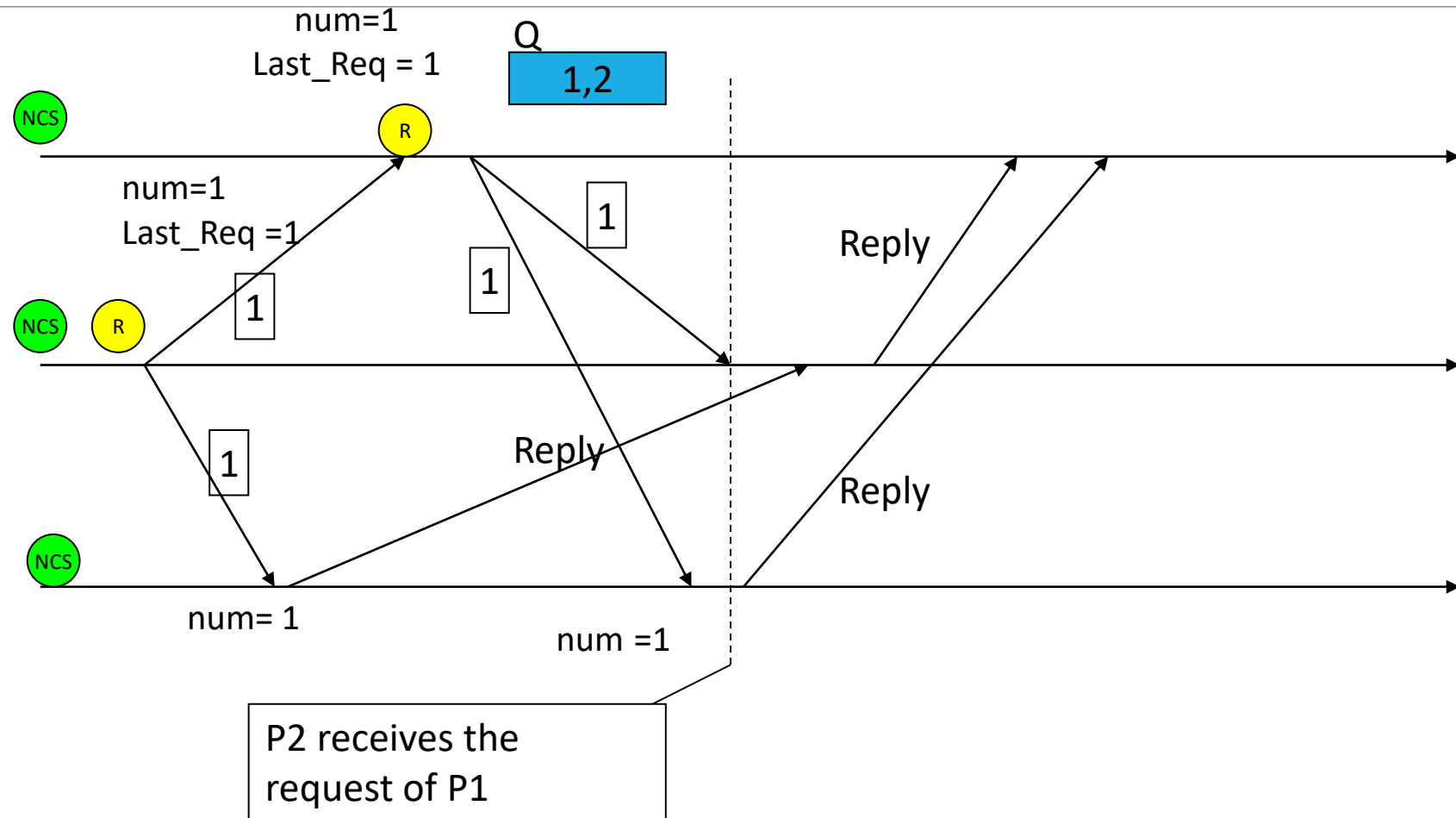
Ricart-Agrawala's algorithm: example



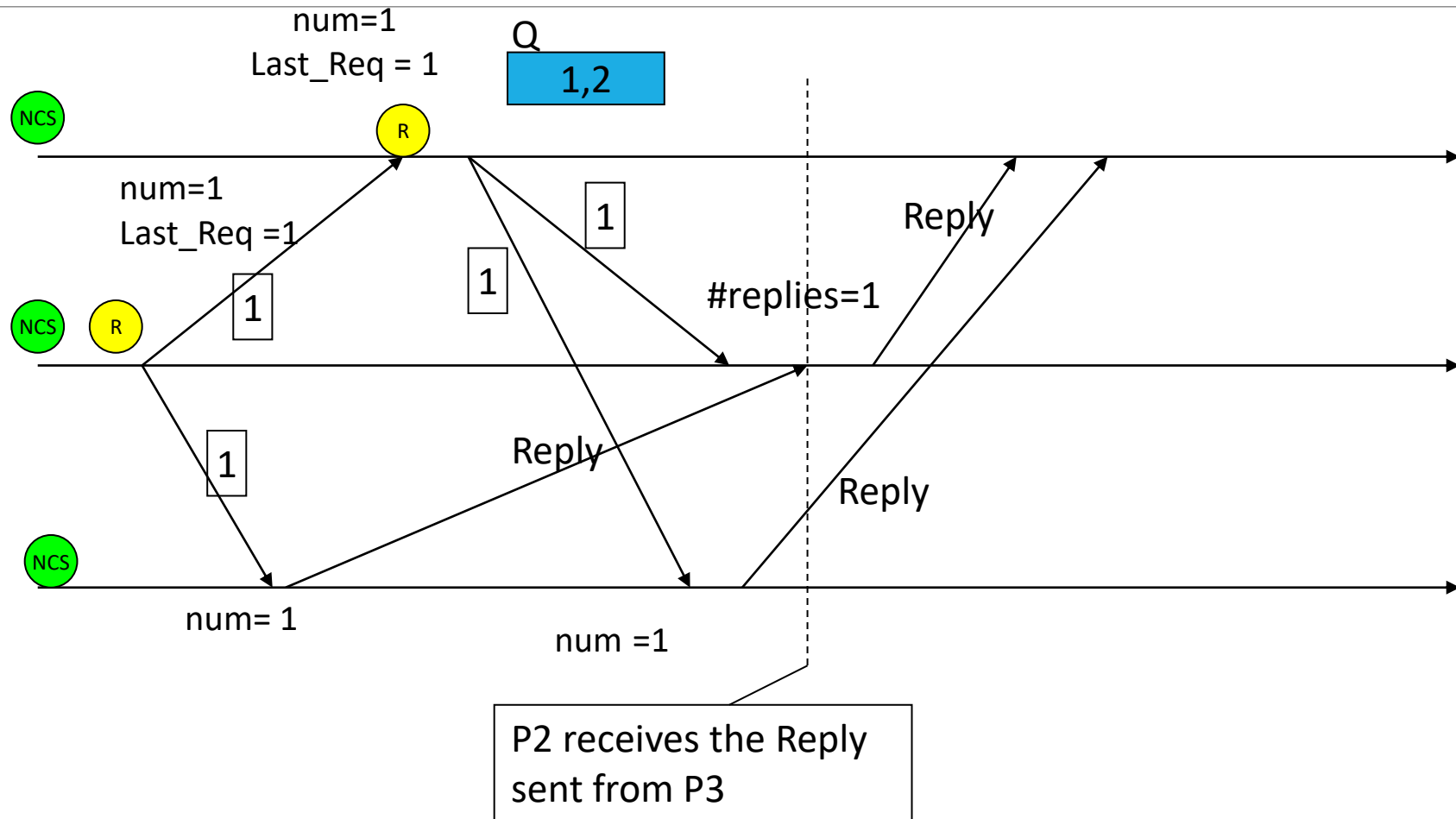
Ricart-Agrawala's algorithm: example



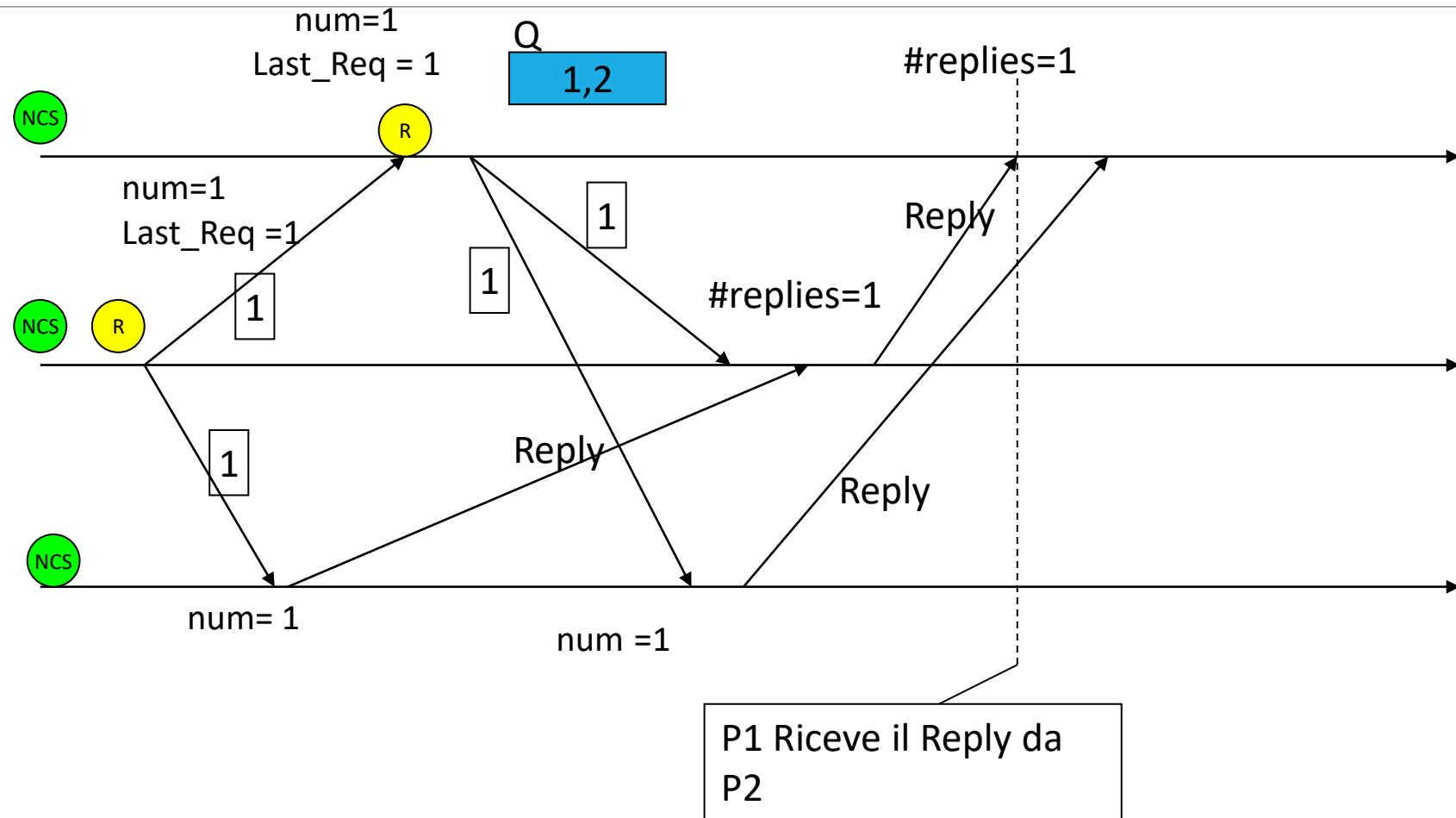
Ricart-Agrawala's algorithm: example



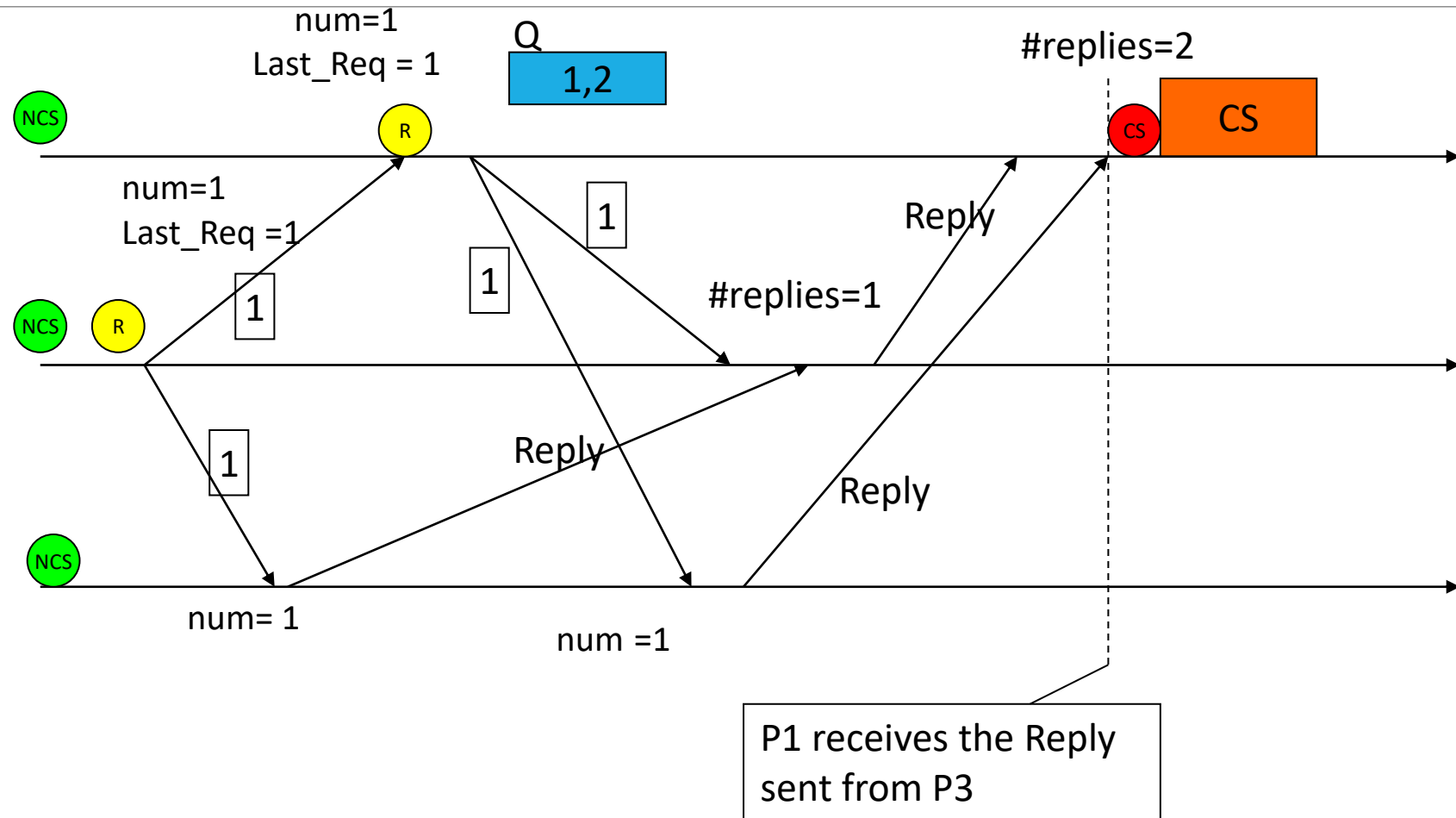
Ricart-Agrawala's algorithm: example



Ricart-Agrawala's algorithm: example



Ricart-Agrawala's algorithm: example



Ricart-Agrawala's algorithm: example

