03/10/23

Dependable Distributed Systems
Master of Science in Engineering in Computer Science

AA 2023/2024

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)

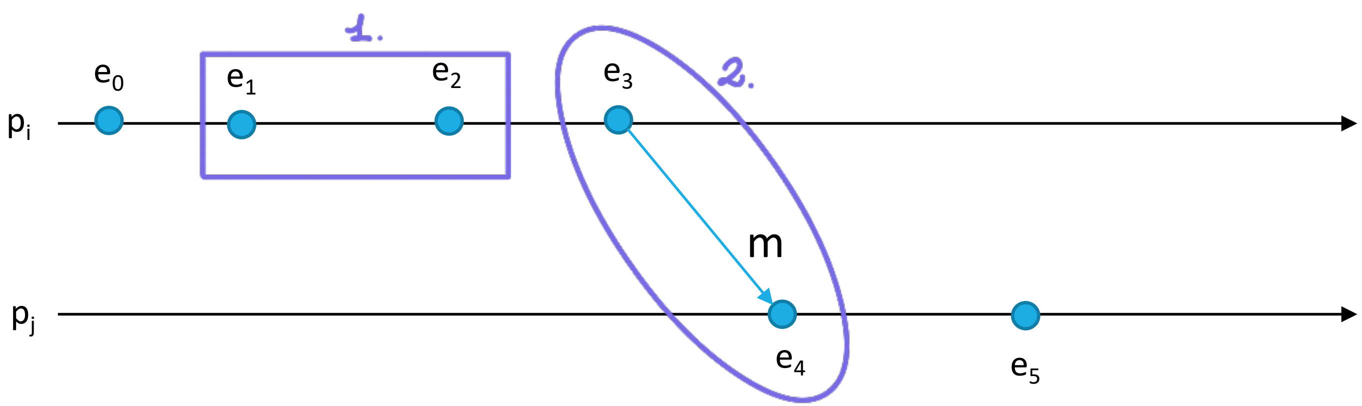
- We note with $\rightarrow_i$ the ordering relation between events in a process p_i
- We note with $\rightarrow$ the happened-before between any pair of events

Happened-Before Relation: Definition

Two events *e and e’* are related by happened-before relation ($e \rightarrow e'$) if:

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

= TRUE



$e_3 \rightarrow e_4$

RULE

2. $e_3 \rightarrow e_4$

1. $e_4 \rightarrow e_5$

Happened-Before Relation

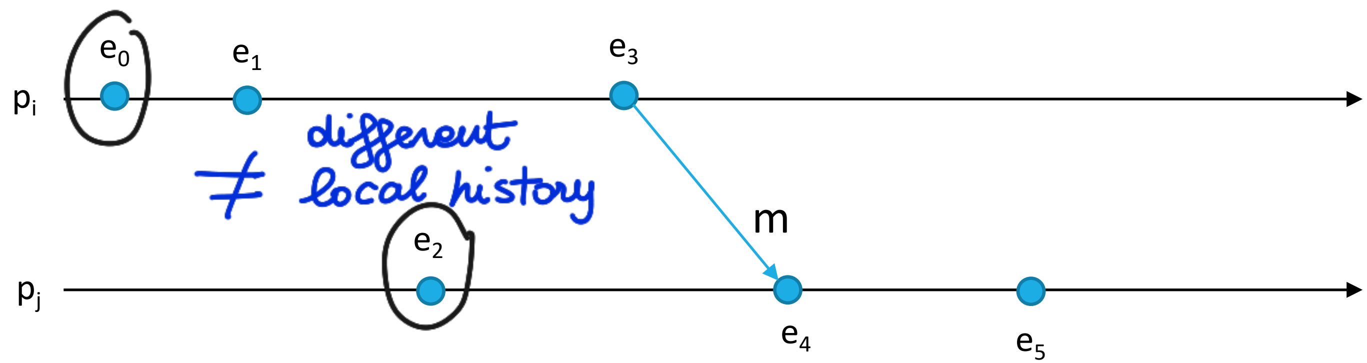
OBSERVATIONS

- Happened-before relation imposes a partial order over events of the execution history
  - It may exists a pair of events $<e_i,e_j>$ 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 ($e_i || e_j$) → NOT RELATED

- For any pair of events e_i and e_j in a distributed system only one of the following holds
  - $e_i \rightarrow e_j$,
  - $e_j \rightarrow e_i$
  - $e_i || e_j$

concurrent

$e_0 || e_2$



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 → INTEGER

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

PROPERTY

- $\boxed{\text{If } e \rightarrow e' \text{ then } ts_e < ts_{e'}}$ → Keep track of CAUSALITY

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….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*

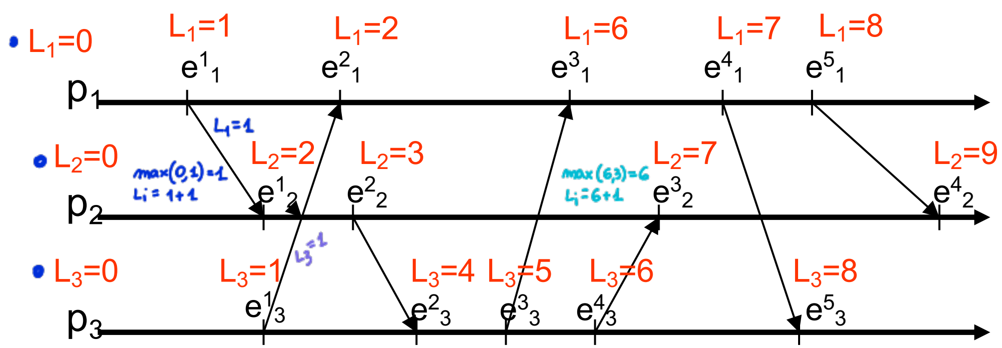
- creates an event *send(m)*
- increases L_i $= L_i + 1$
- timestamps *m* with ts=L_i

When p_i receives a message *m* with timestamp *ts*

ensure
MONOTONICITY

- Updates its logical clock $L_i = \max(ts, L_i)$
- Produces an event *receive(m)*
- Increases L_i $= L_i + 1$

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
- $e^1_1 \mid\mid e^1_3$ and respective timestamps have the same value
- $e^1_2 \mid\mid e^1_3$ but respective timestamps have different values

e^1_2 clock=2
e^1_3 clock=1
CONCURRENT

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

if PROCESSES = 10 → vector SIZE = 10

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

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

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

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

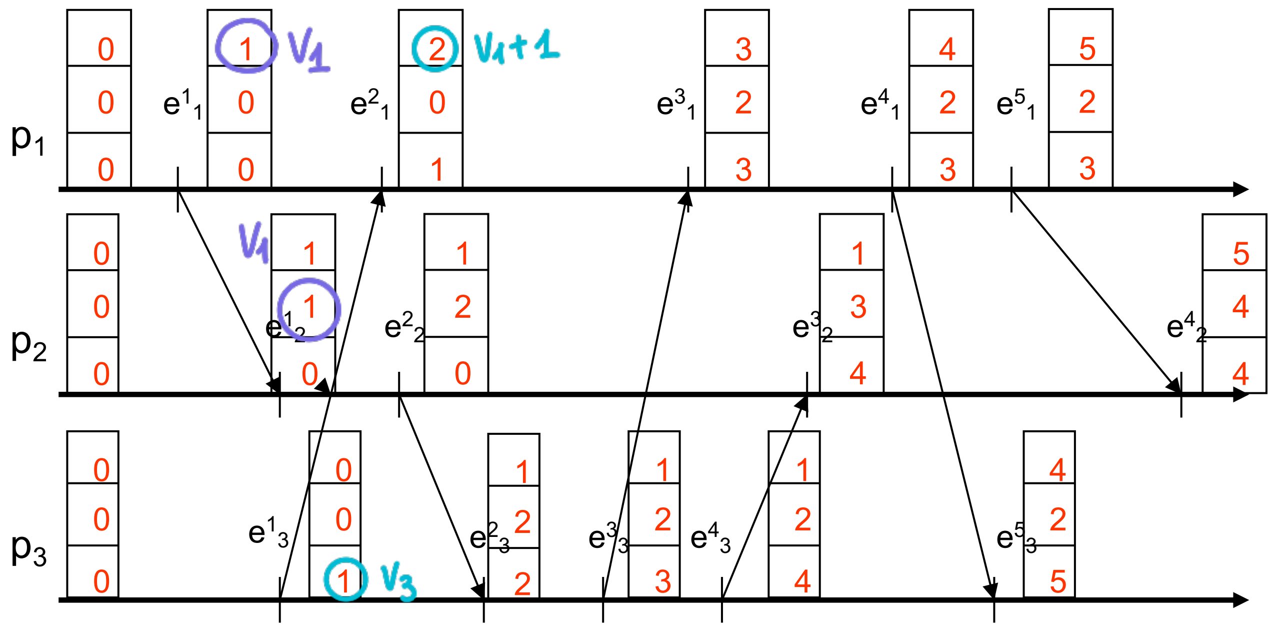
When p_i sends a message *m*

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

When p_i receives a message *m* containing timestamp *ts*

- Updates it logical clock $V_i\,[\ j\ ] = \max(ts[\ j\ ], V_i\,[\ j\ ]) \ \forall\ j = 1 \ldots N$
- 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

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

$V \leq V'$ 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

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

A comparison of Vector Clocks

$$\underset{V}{\begin{bmatrix}1\\0\\0\end{bmatrix}} < \underset{V'}{\begin{bmatrix}1\\1\\0\end{bmatrix}}$$

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

$$\underset{V}{\begin{bmatrix}1\\0\\0\end{bmatrix}} \quad \underset{V'}{\begin{bmatrix}0\\0\\1\end{bmatrix}}$$

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

concurrent

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 : timestamp
- 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

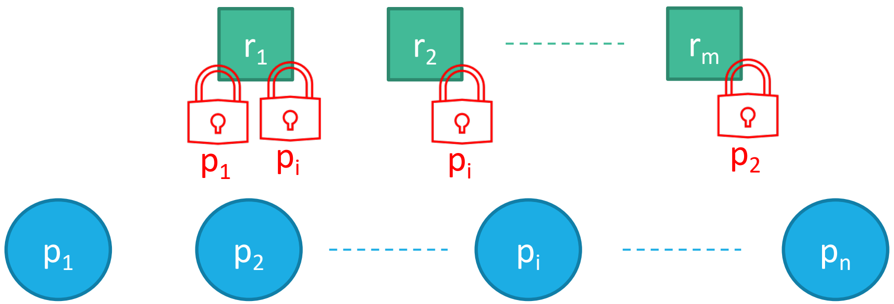
Distributed Mutual Exclusion

The Mutual Exclusion Problem

Let us consider

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

shared resources



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

The system is asynchronous not impo TIME

not FAILURE

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

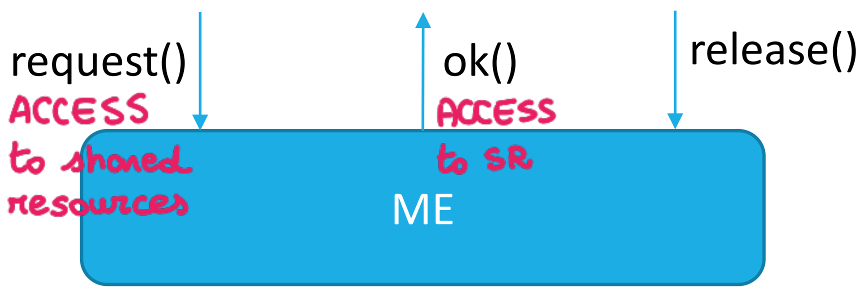
EVENTS

- `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

Critical section = Shared Resources



Different Approaches to Distributed Mutual Exclusion

- Coordinator-based
- Token-based (over logical ring topology)
- Quorum-based
- Logical Clock-based

Tomorrow

Today

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 timestamp
  - Is the counter for process p_i
- Q
  - Is a queue maintained by pi 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 pi

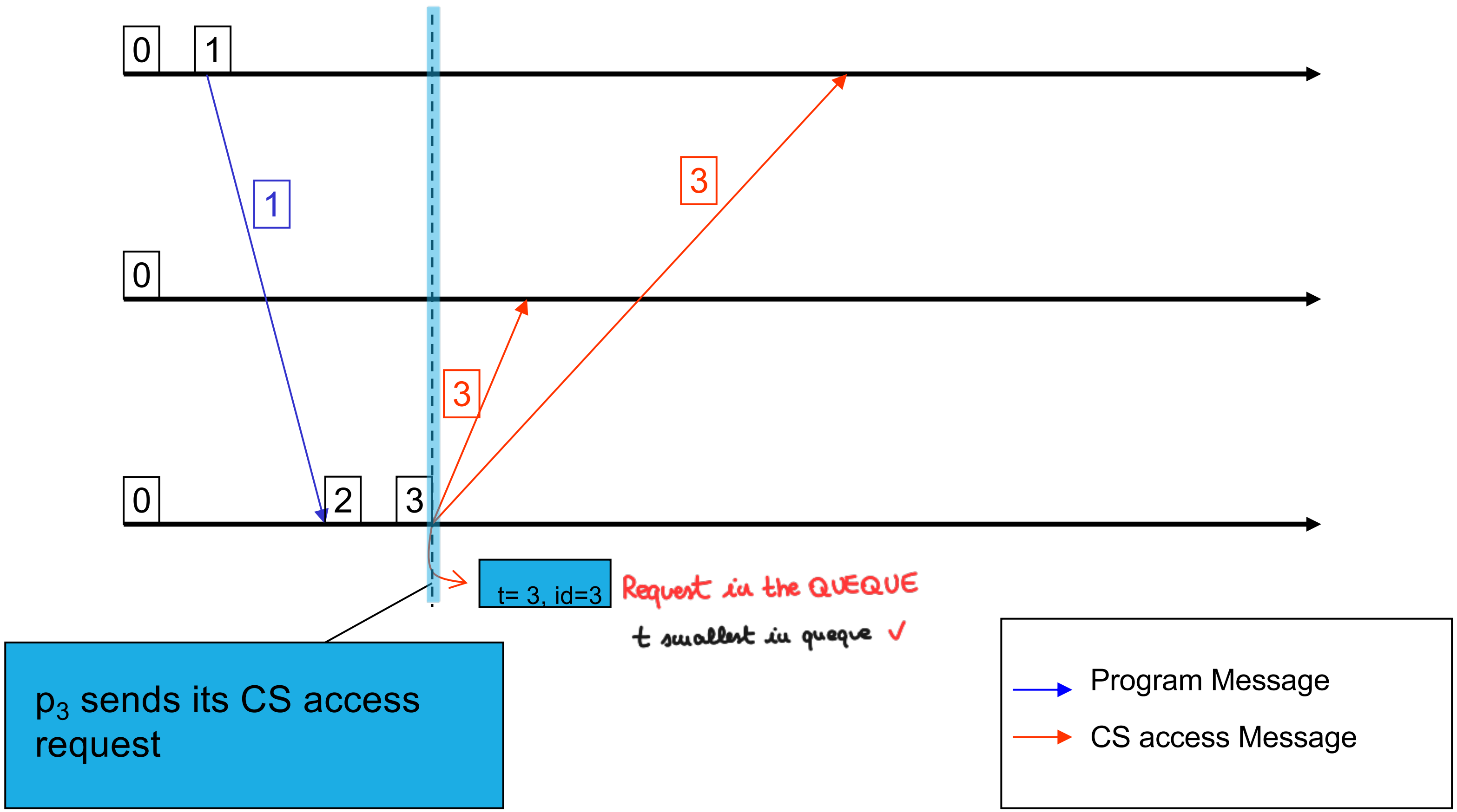
- **p_i enters the CS iff**
  1. p_i has, in its queue, a request with timestamp t
  2. t is the small timestamp in the queue
  3. p_i has already received an ack with timestamp t' from any other process and t'>t

if I have 3 TRUE
↓
entering in CS

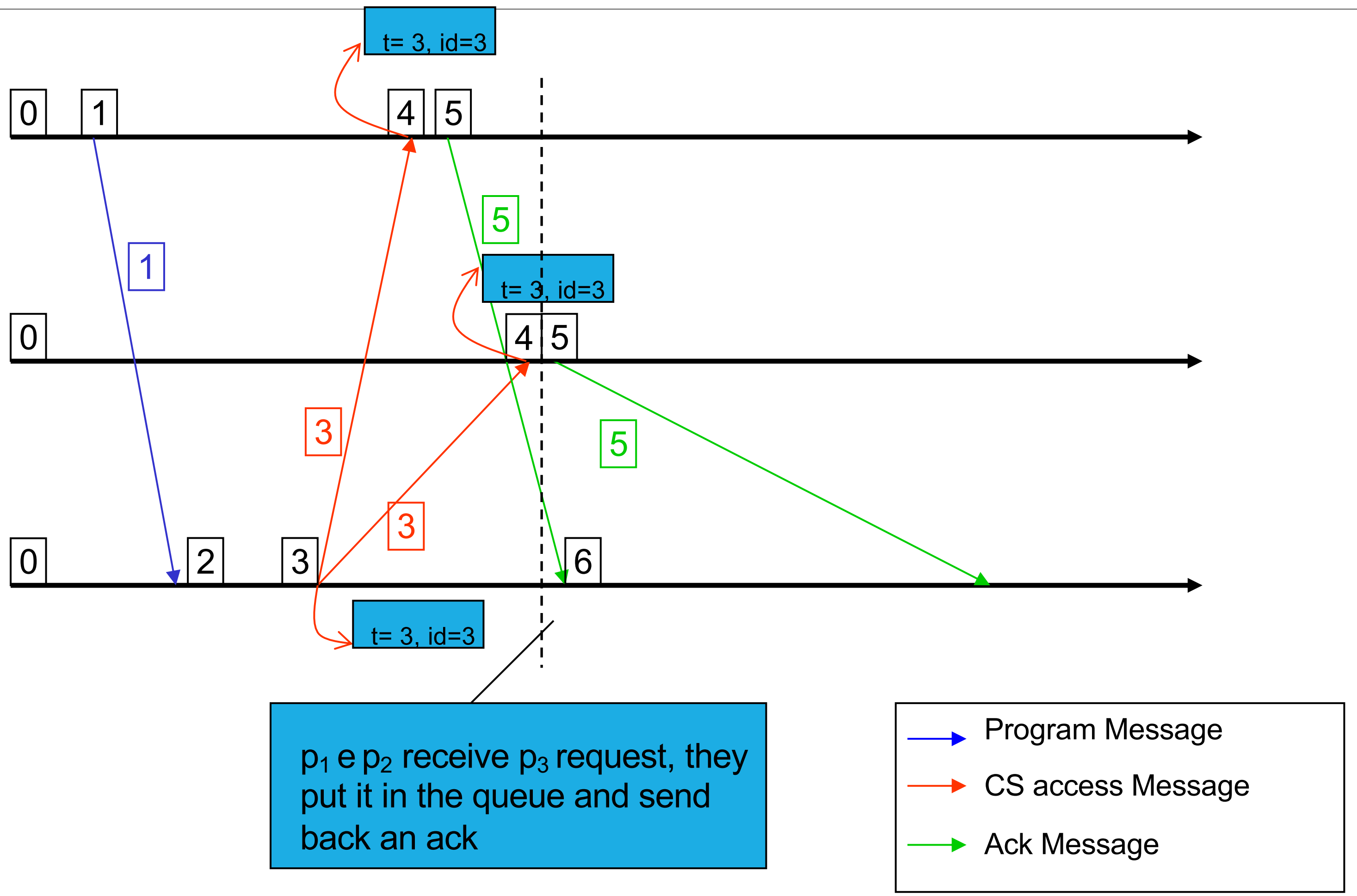
- **Release of the CS**
  - p_i sends a RELEASE message to all the other processes
  - p_i deletes its request from the queue

- **Reception of a release message from a process pj**
  - p_i deletes p_j's request from the queue

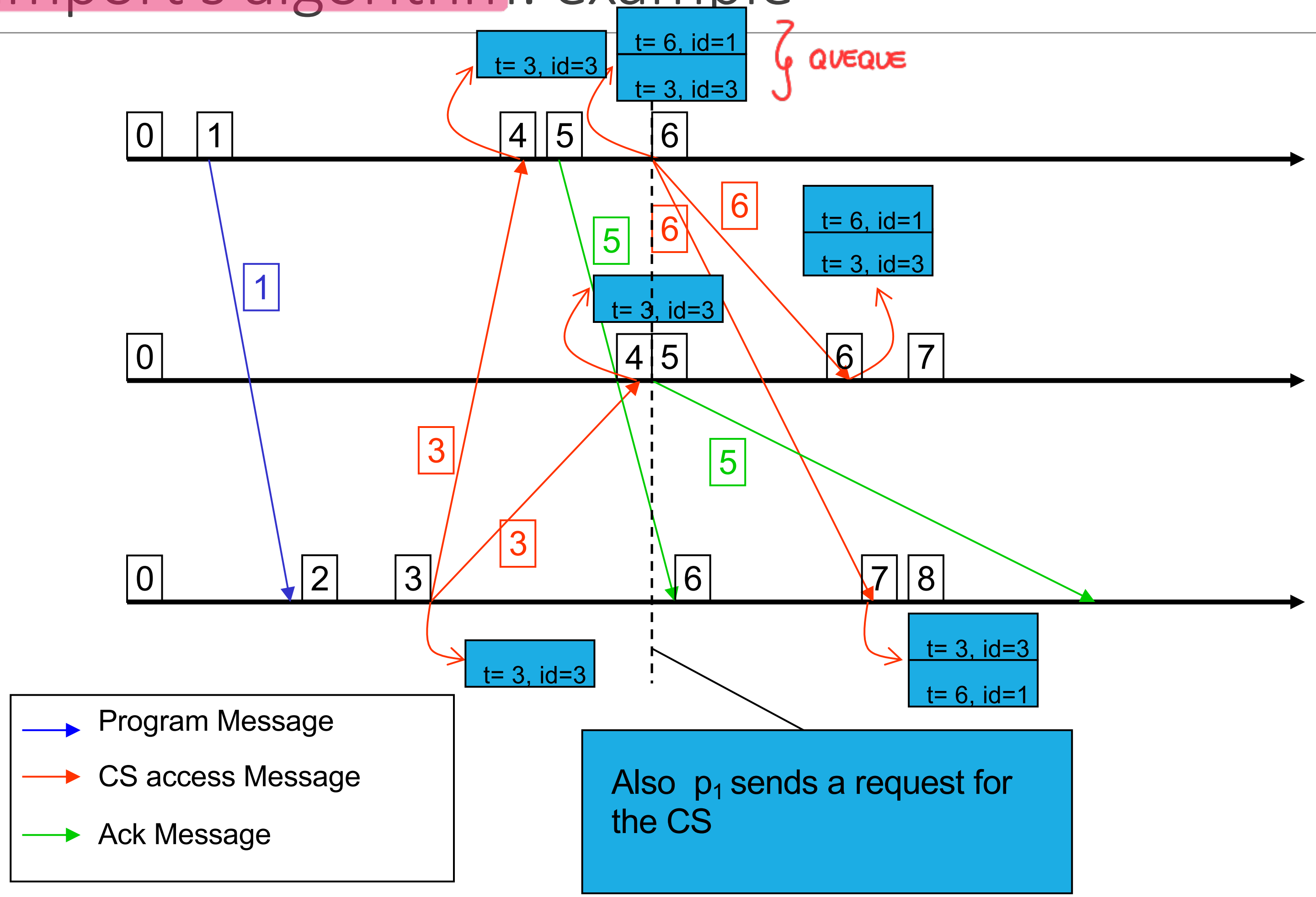
Lamport's algorithm: example



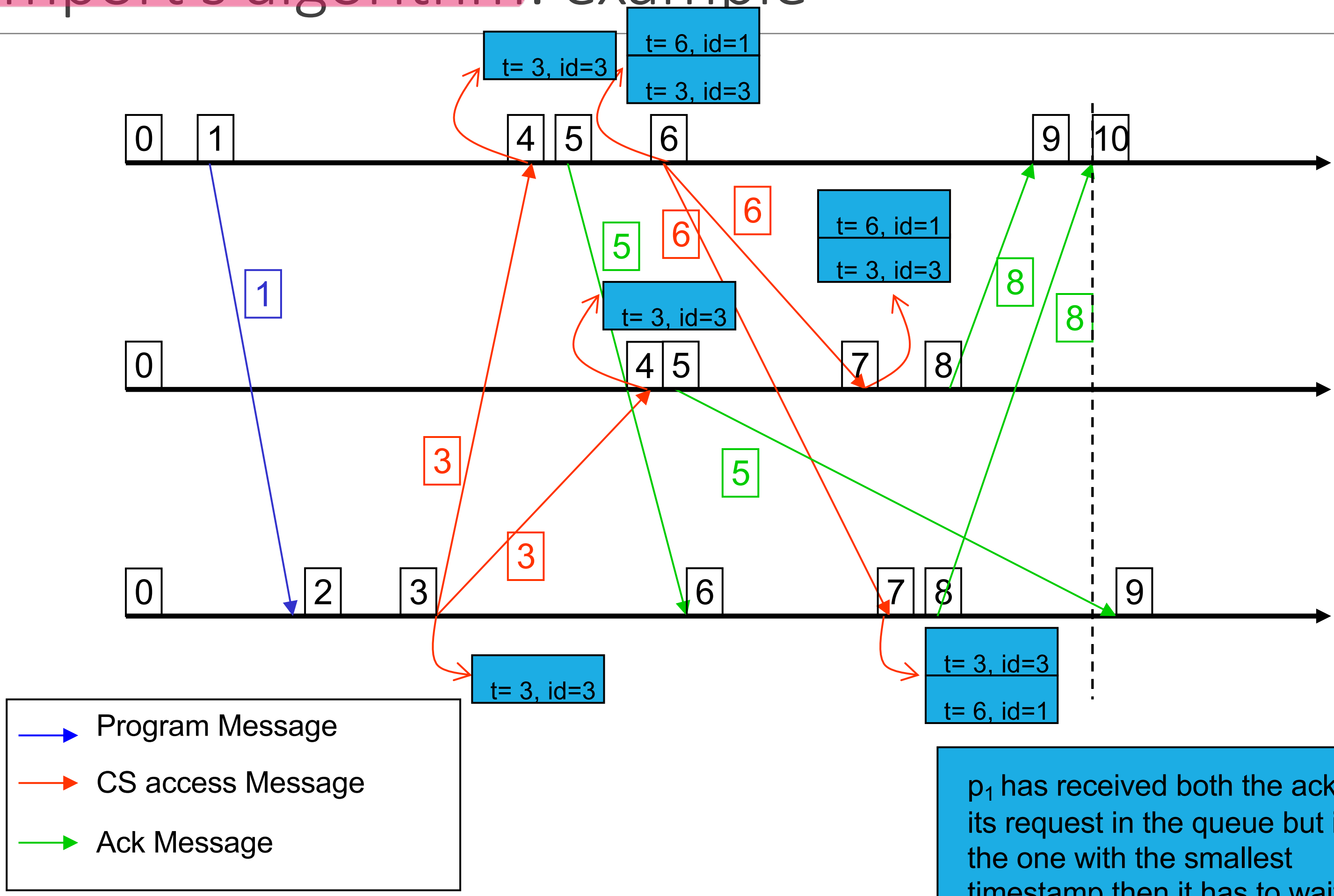
Lamport's algorithm: example



Lamport's algorithm: example

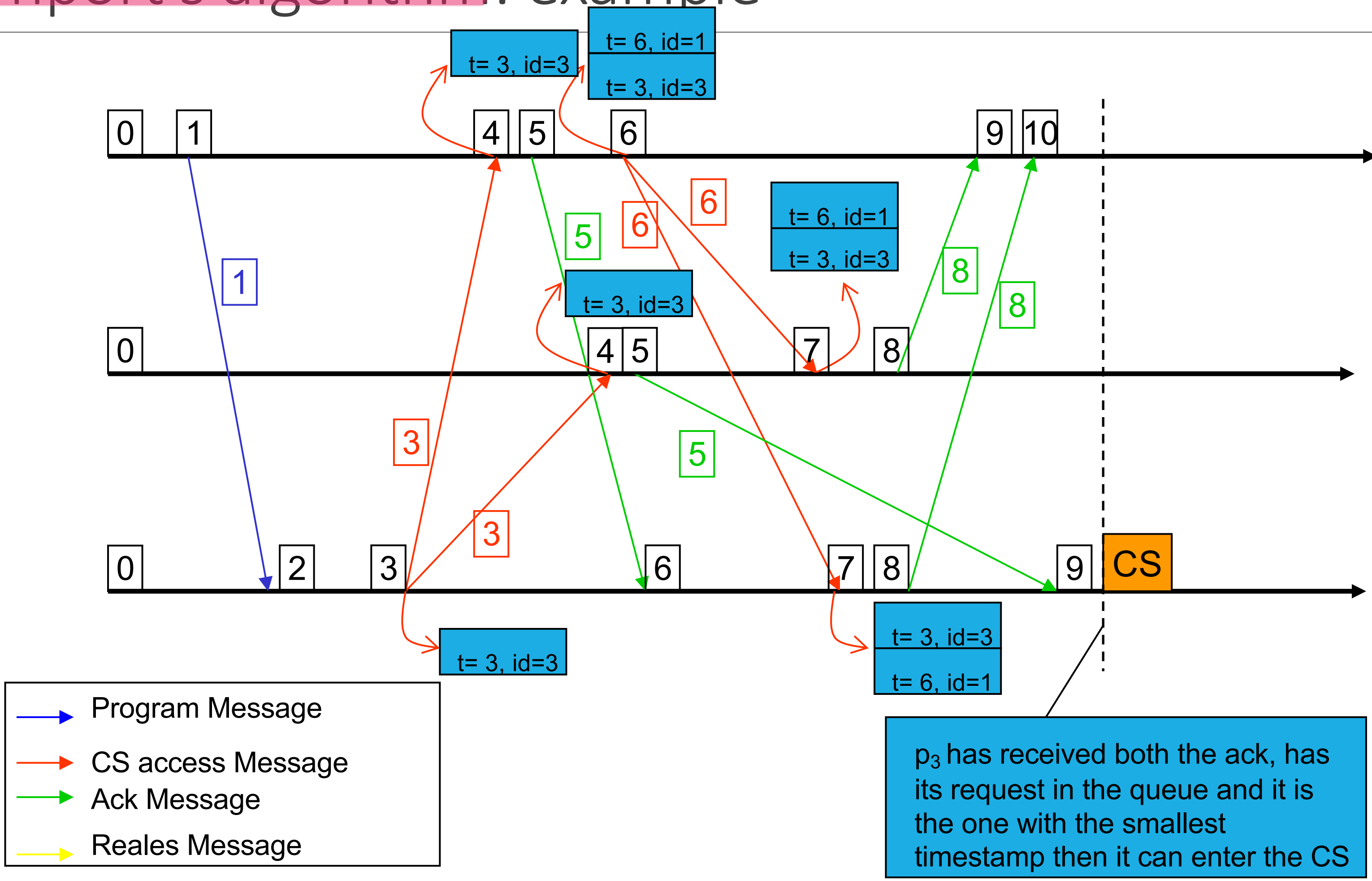


Lamport's algorithm: example

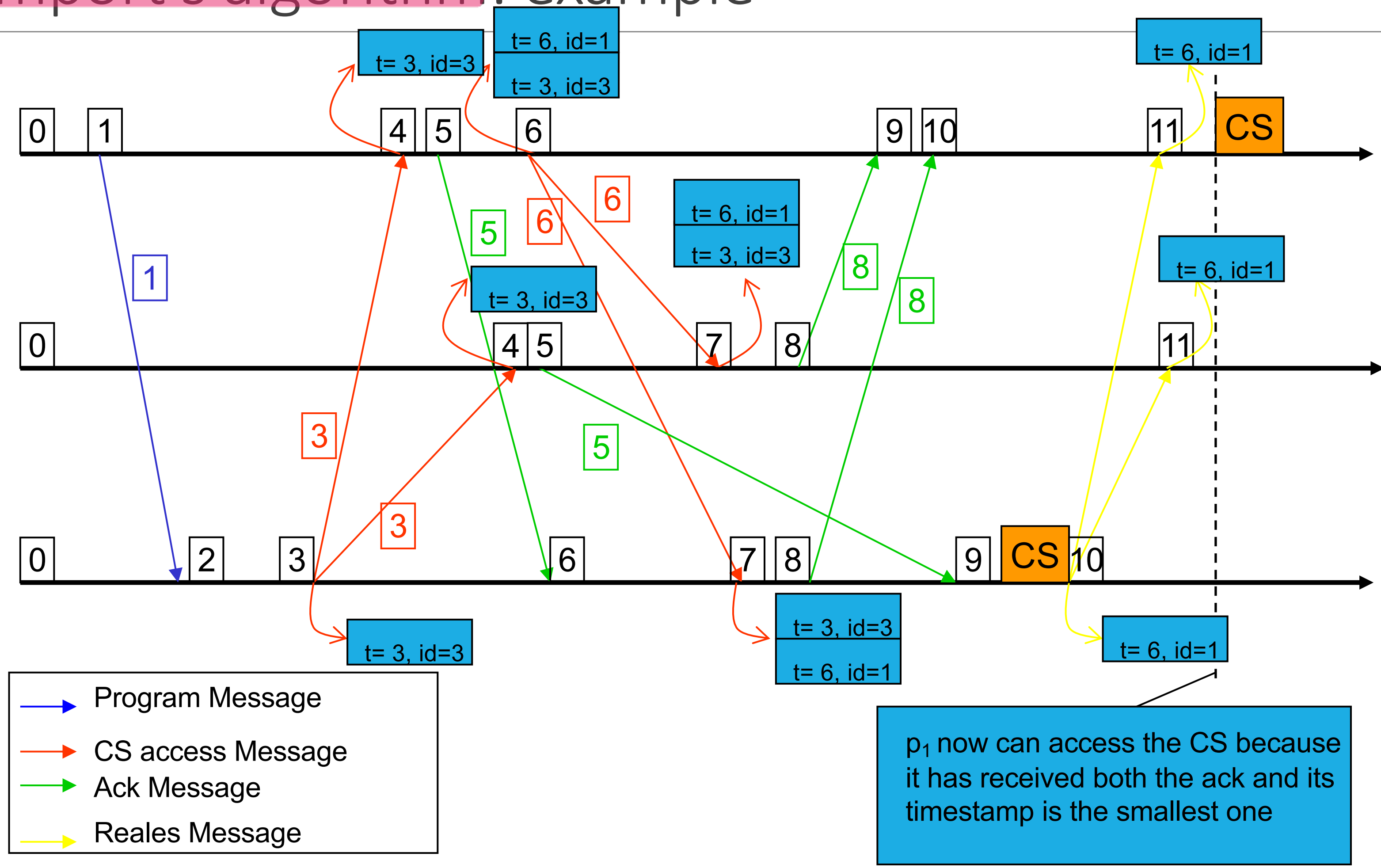


p_1 has received both the ack, has its request in the queue but it is not the one with the smallest timestamp then it has to wait

Lamport's algorithm: example



Lamport's algorithm: example

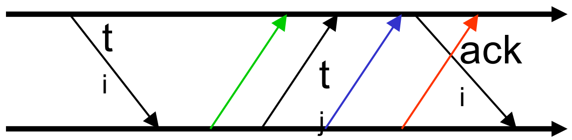


Lamport’s algorithm: safety proof

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

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

<u>Fairness is satisfied</u>: 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

P_1 P_2 concurrently, so $ts(p_1) = ts(p_2)$

id is unique and I use it to break it the symmetry

order by TS, if $ts(p_1) = ts(p_2)$

↓

order by ID

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 ∈ {Requesting, CS, NCS} (initially NCS) process that are
- 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(num) to pi
4. Wait until #replies=n-1
5. State=CS
6. CS
7. ∀ r∈Q send REPLY to r all processes that are waiting in the queue
8. Q= ∅; State=NCS; #replies=0

LOGICAL CLOCK

TIMESTAMP

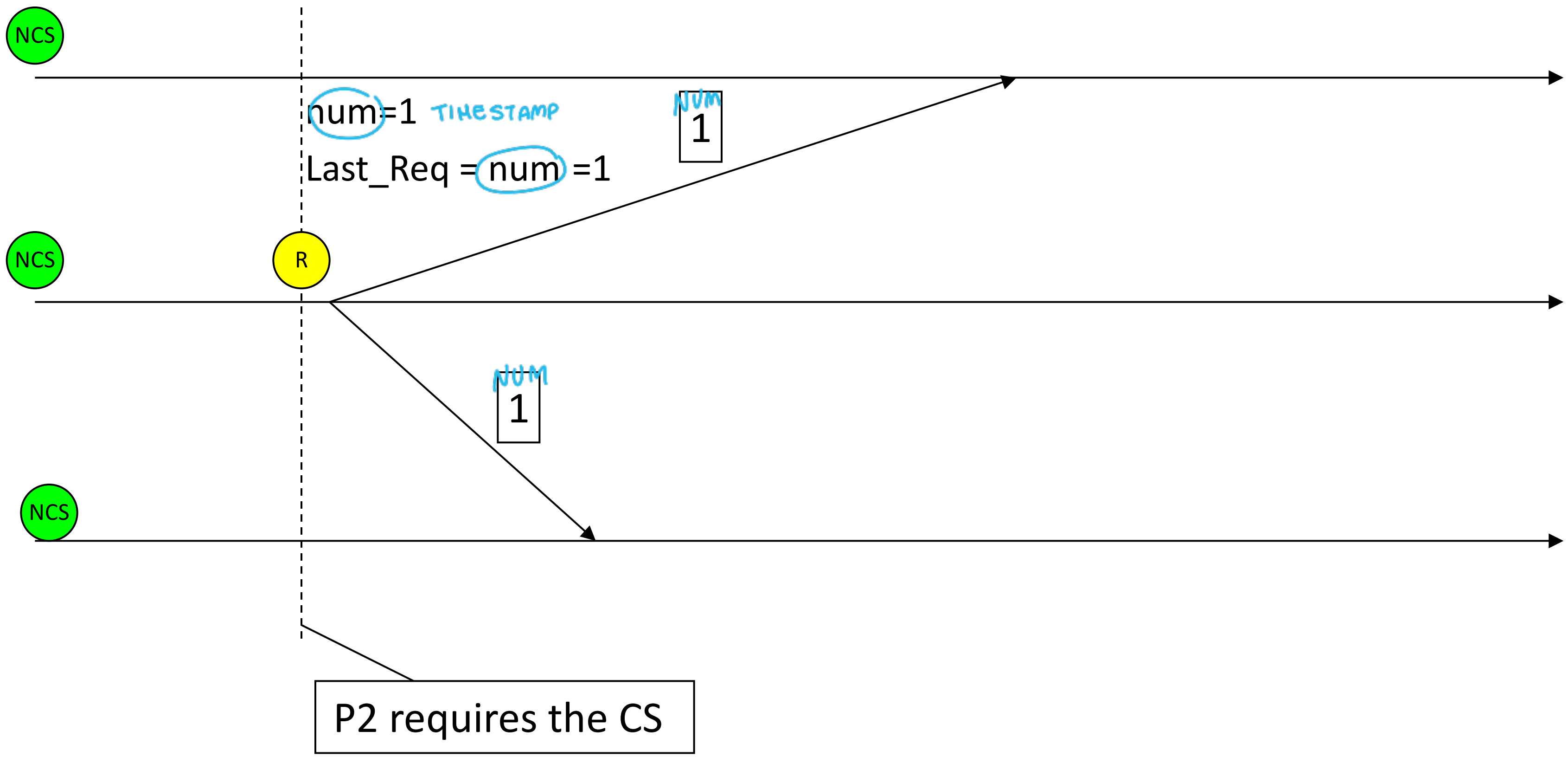
Upon receipt REQUEST(t) from pj

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)

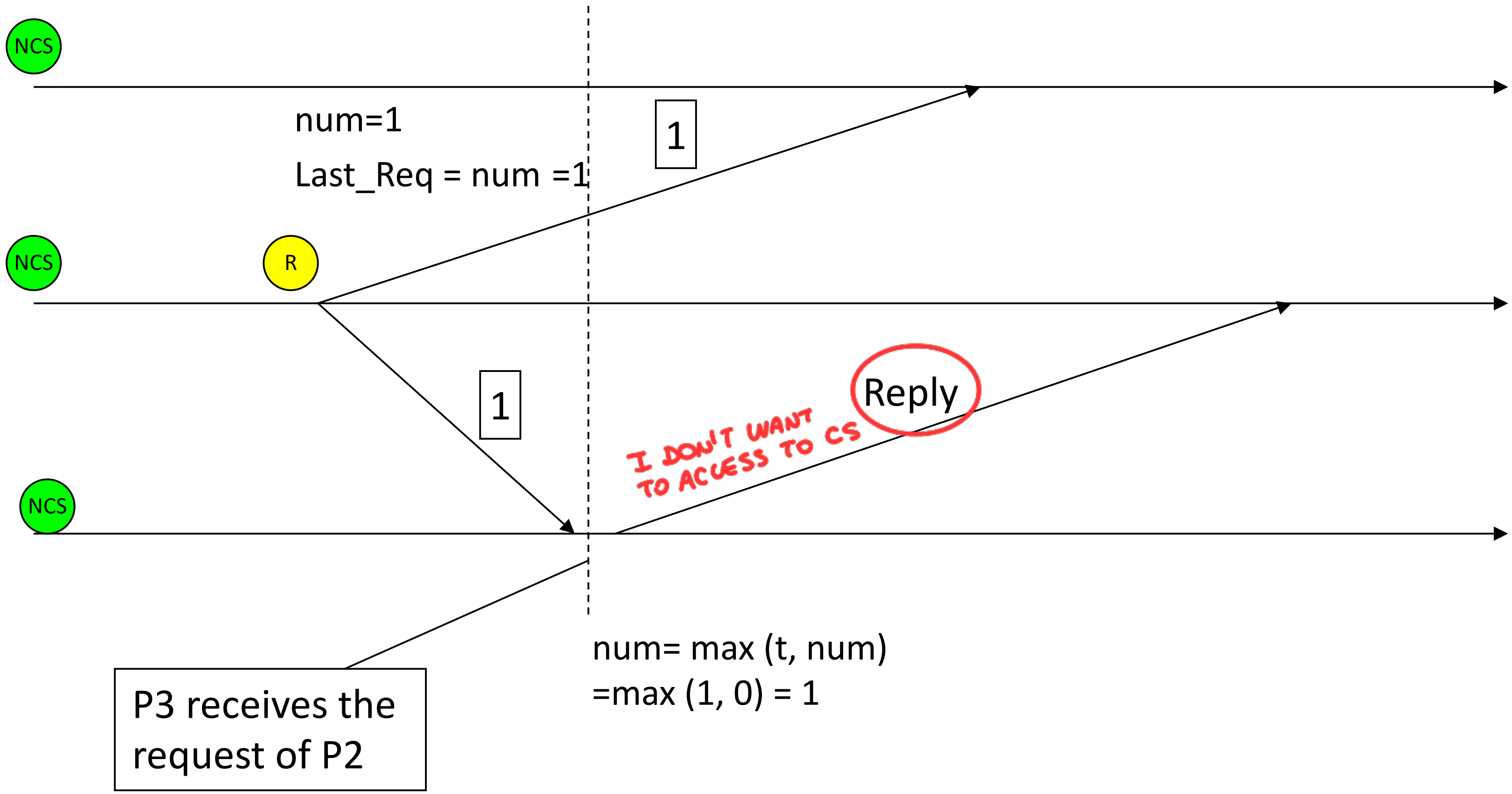
Upon receipt of REPLY from pj

1. #replies=#replies+1

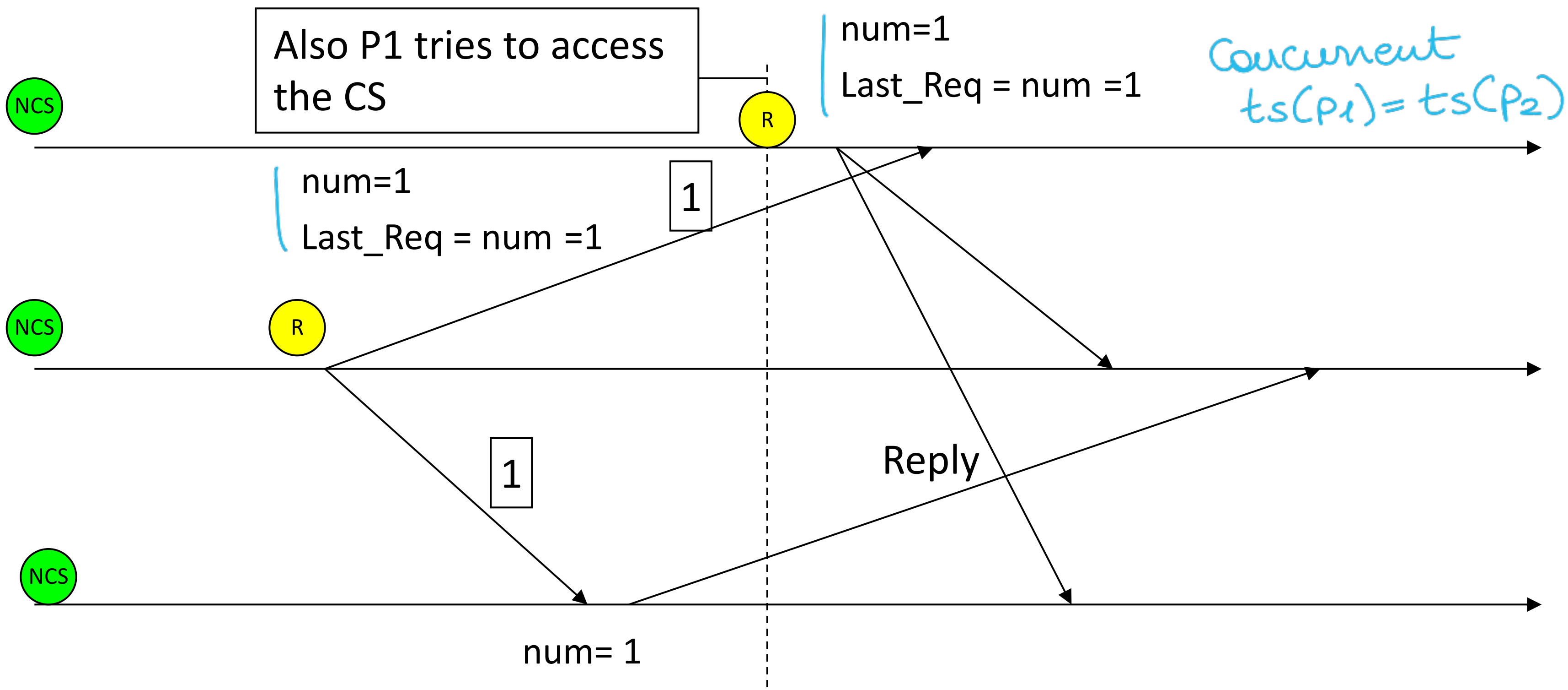
Ricart-Agrawala's algorithm: example



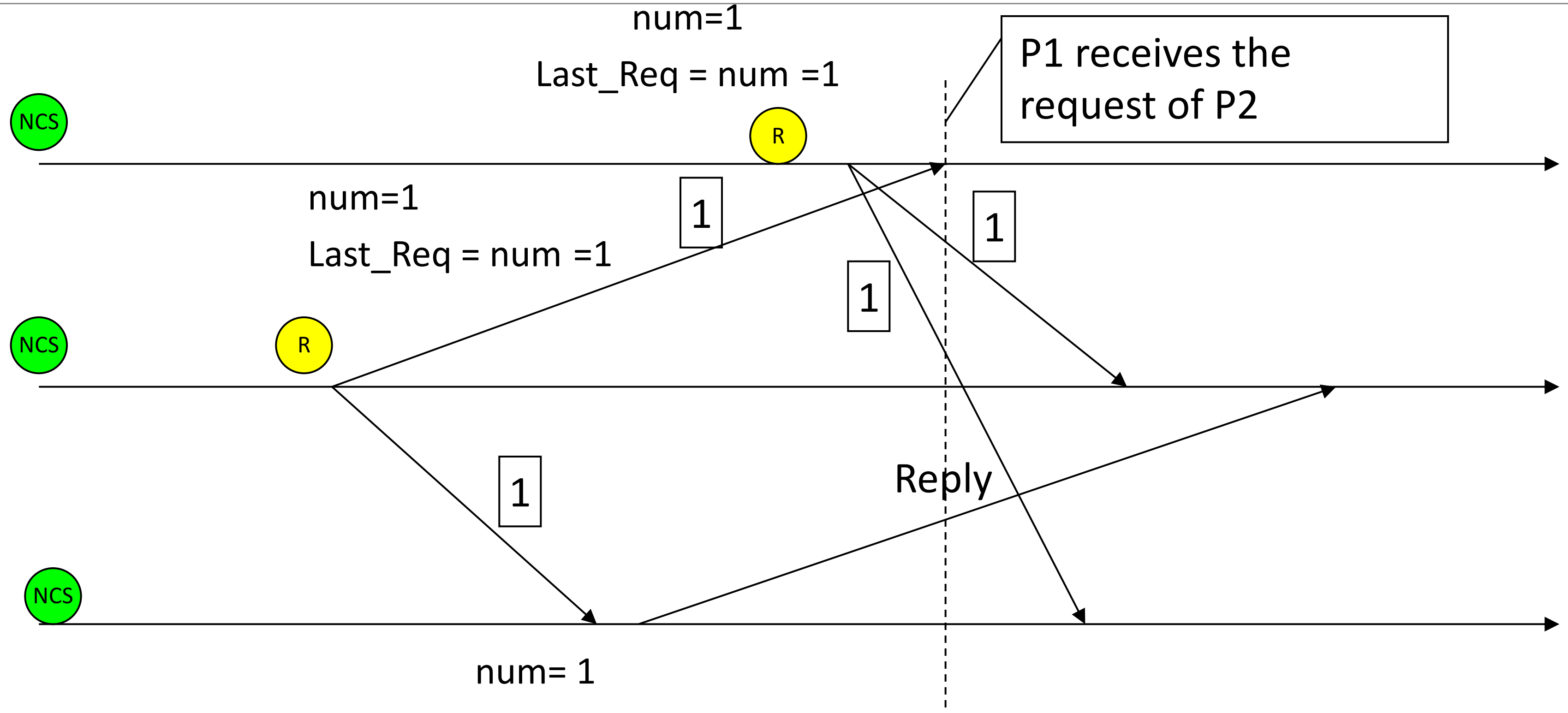
Ricart-Agrawala's algorithm: example



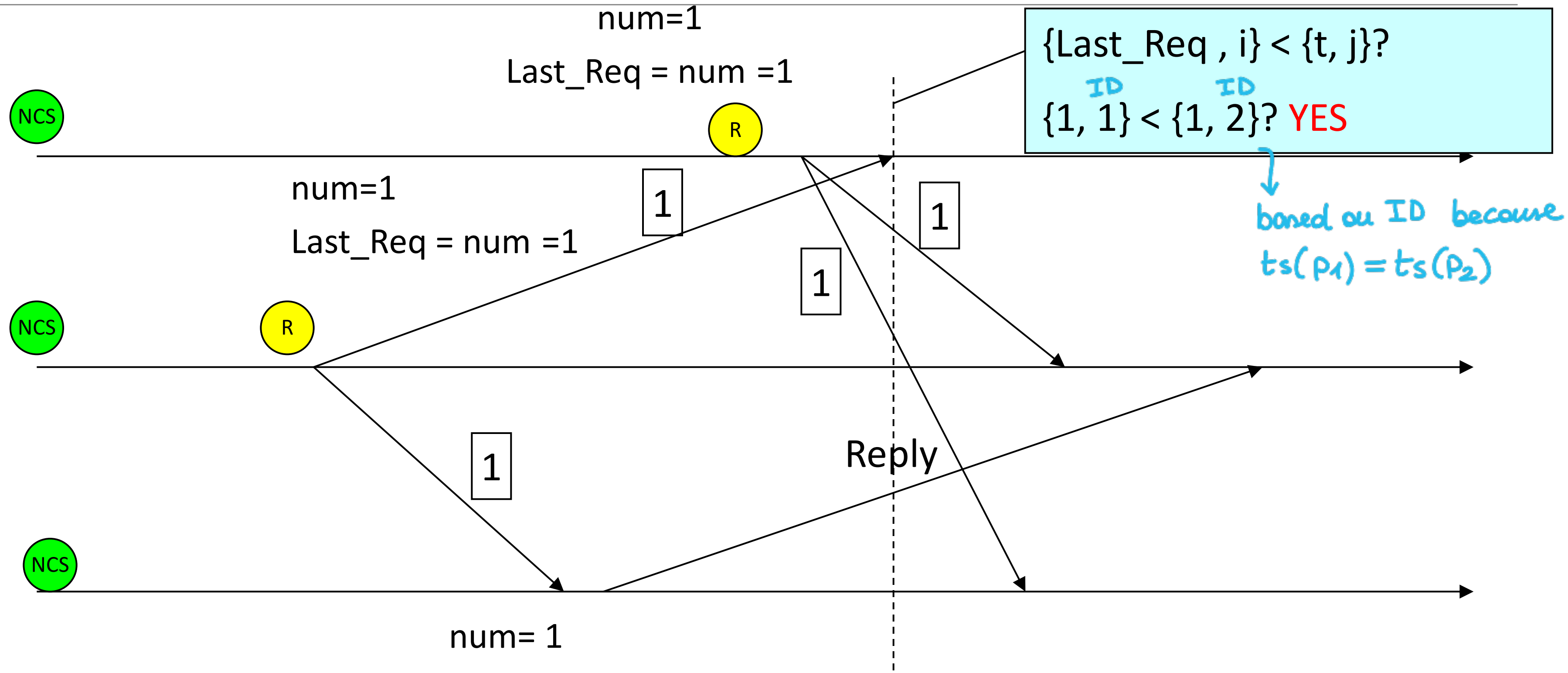
Ricart-Agrawala's algorithm: example



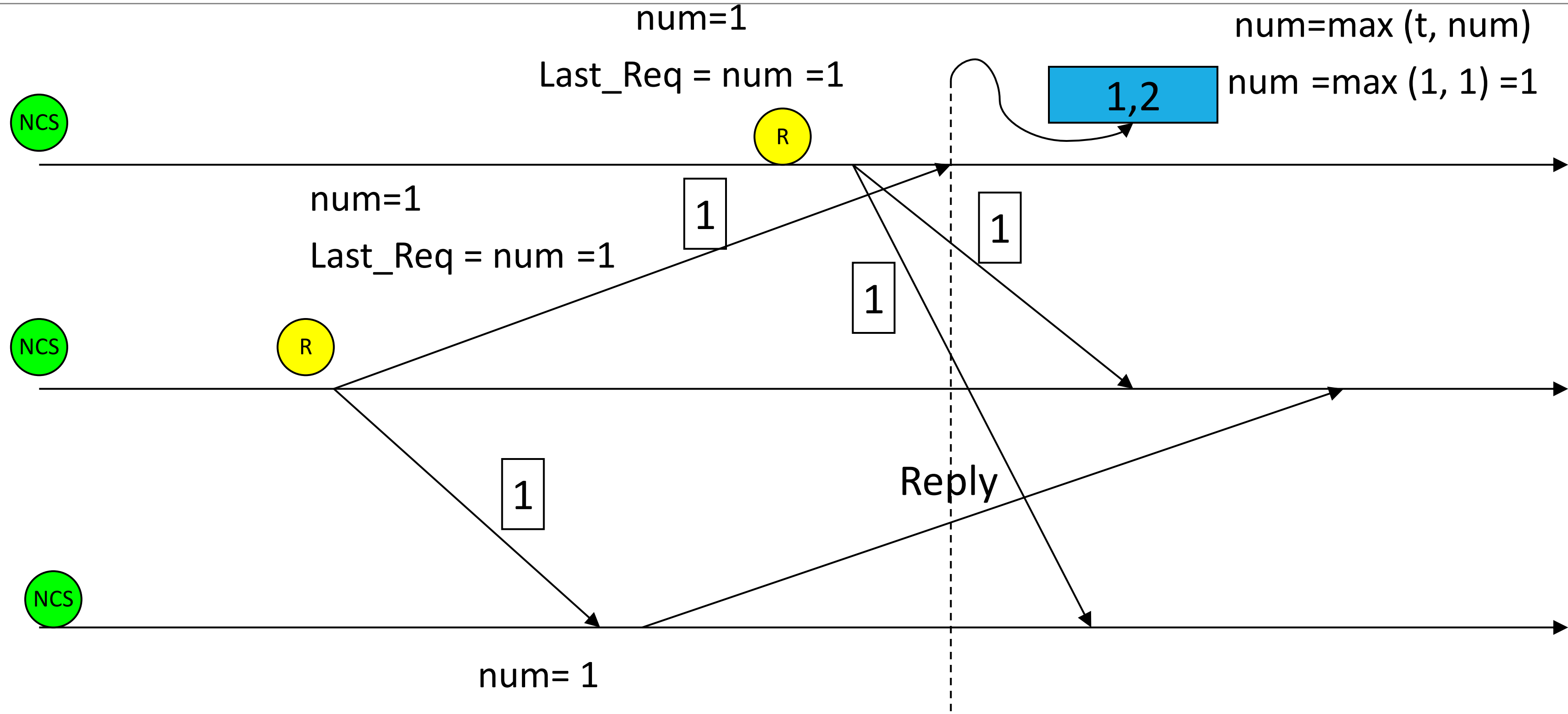
Ricart-Agrawala's algorithm: example



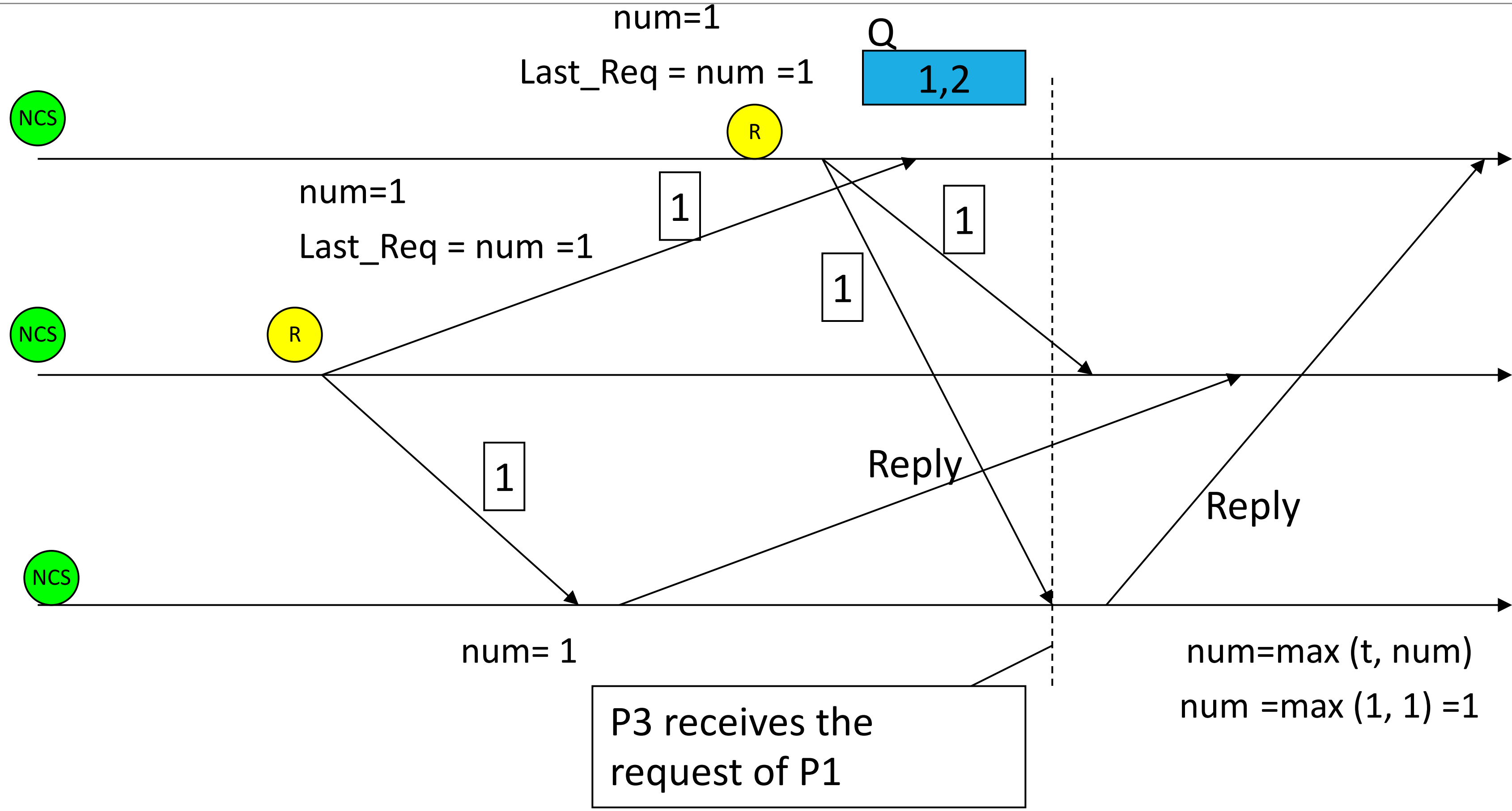
Ricart-Agrawala's algorithm: example



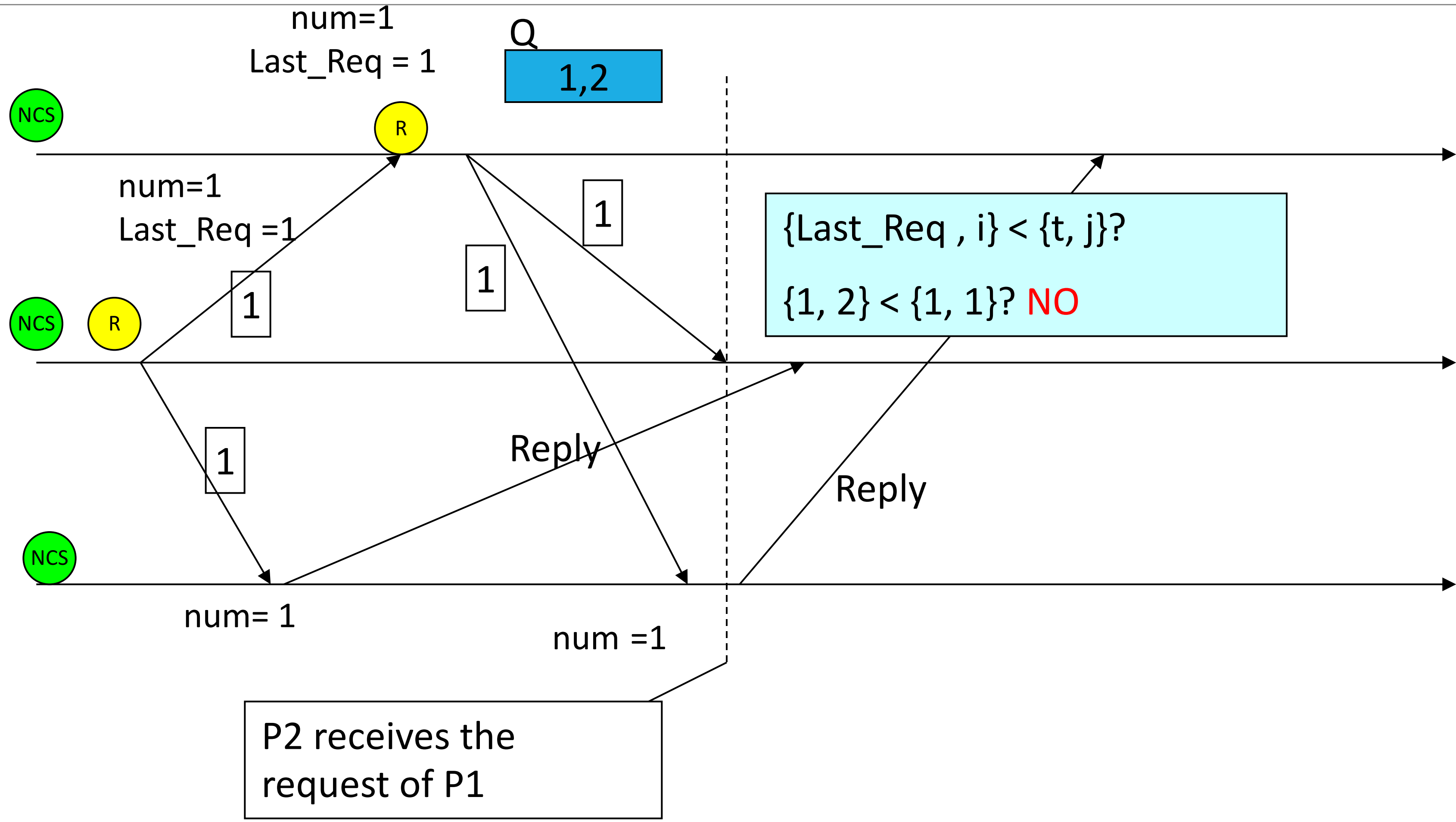
Ricart-Agrawala's algorithm: example



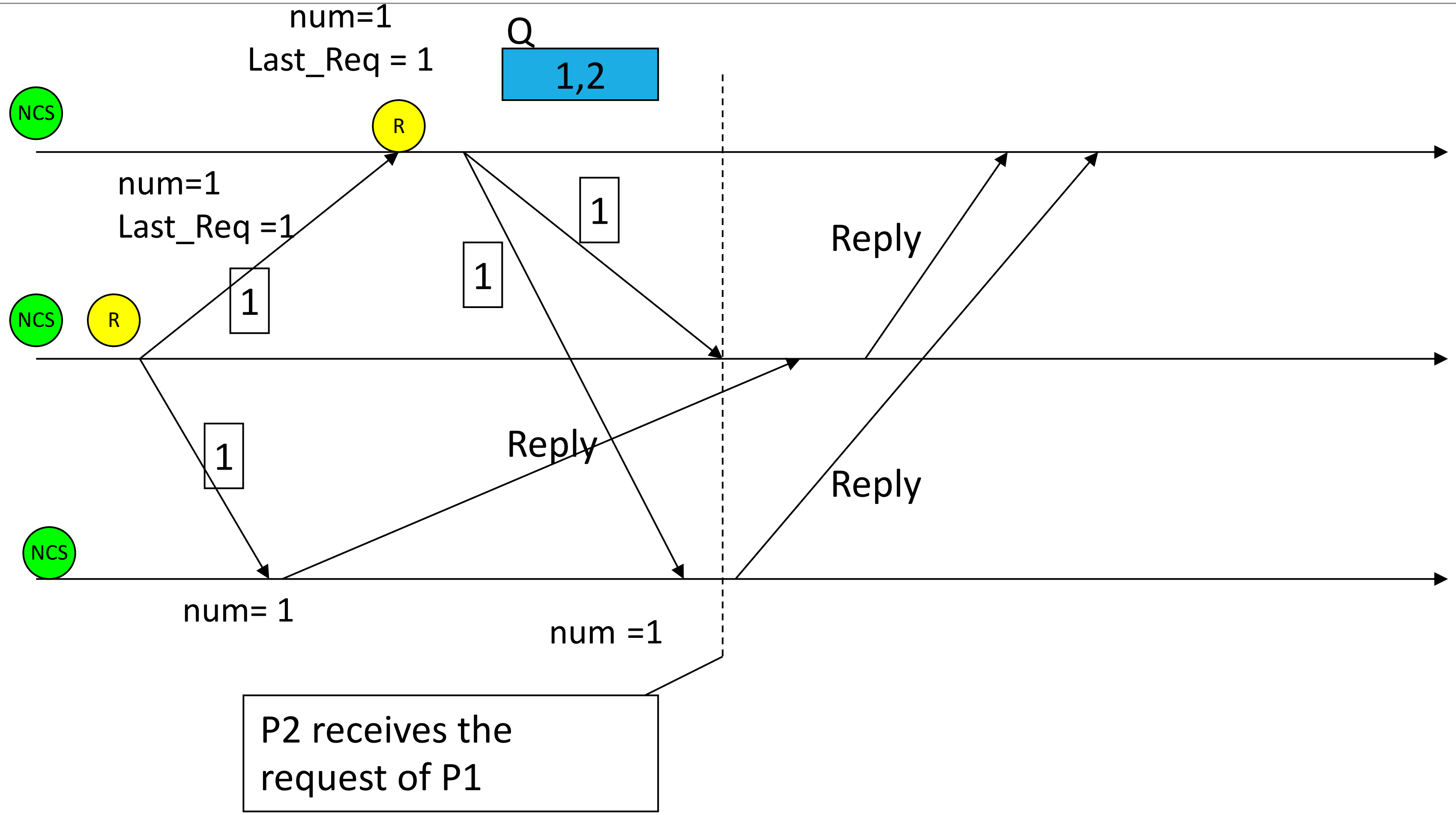
Ricart-Agrawala's algorithm: example



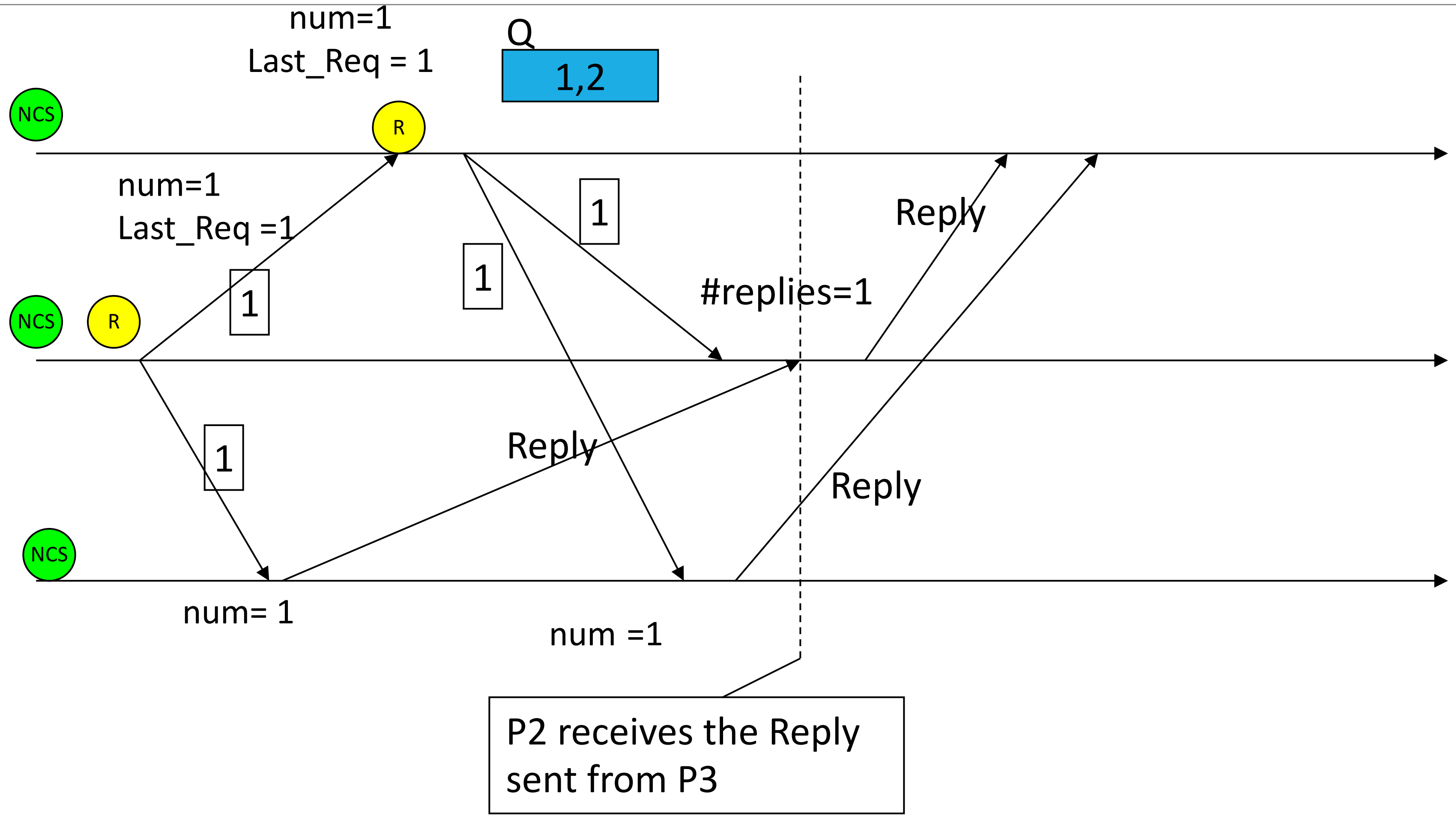
Ricart-Agrawala's algorithm: example



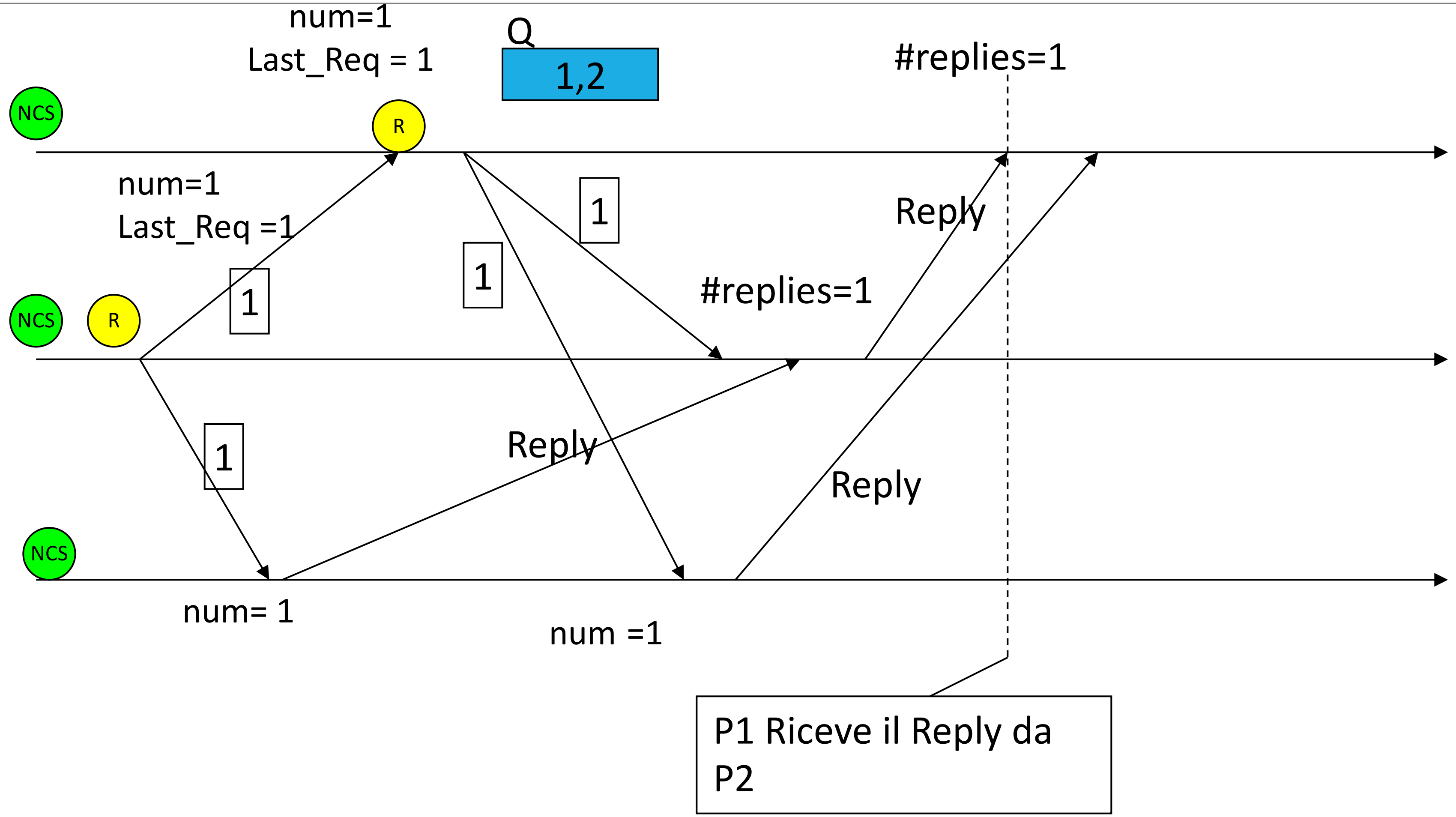
Ricart-Agrawala's algorithm: example



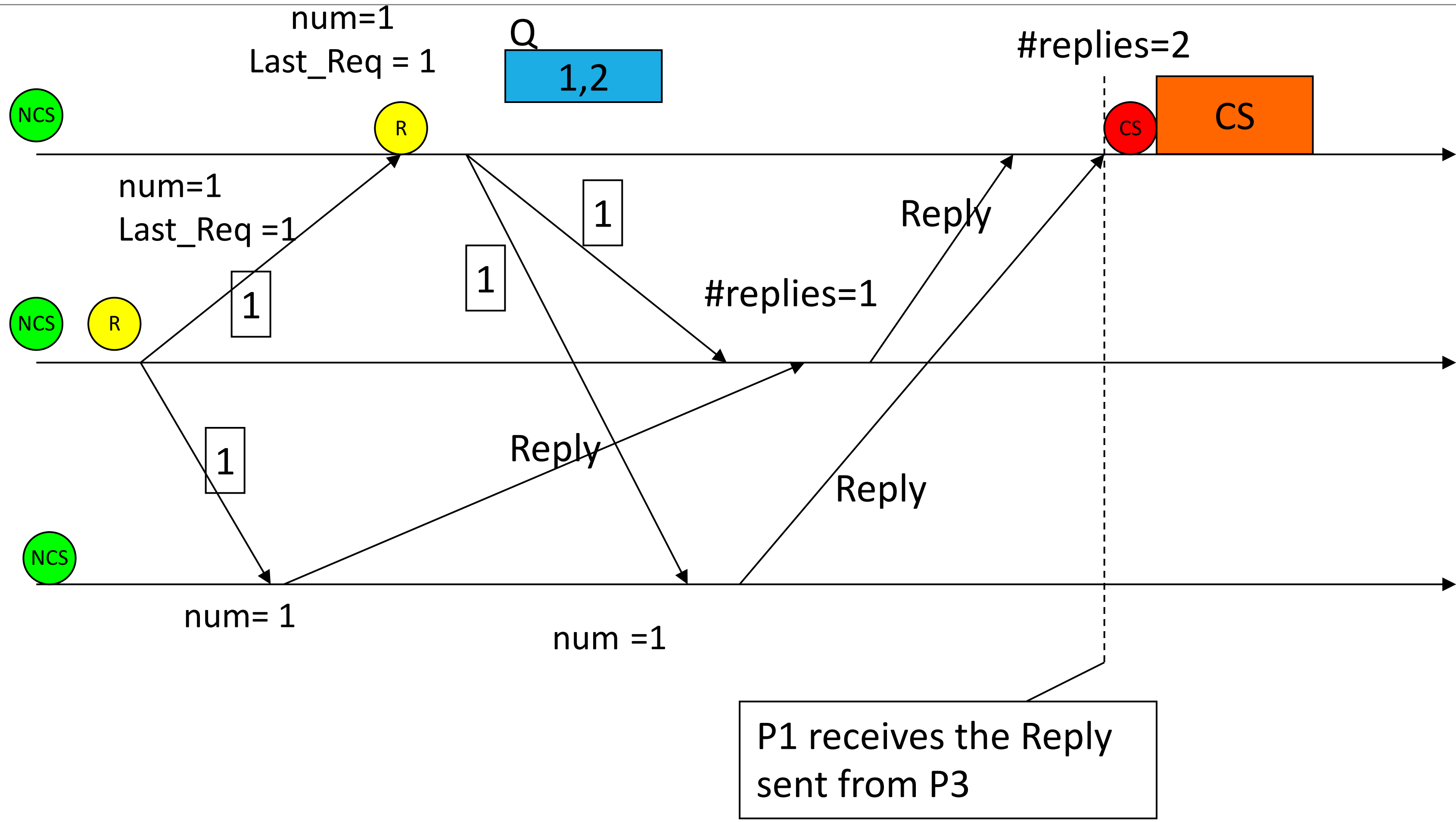
Ricart-Agrawala's algorithm: example



Ricart-Agrawala's algorithm: example



Ricart-Agrawala's algorithm: example



Ricart-Agrawala's algorithm: example