

Sistemi Distribuiti

La Mutua Esclusione Distribuita

Dott. Ing. Silvia Bonomi

bonomi@dis.uniroma1.it

La Mutua Esclusione Distribuita: introduzione al problema

- Si ha una risorsa condivisa tra i processi del sistema
- Ogni processo vuole acquisire la risorsa ed utilizzarla in maniera esclusiva per non avere interferenze con gli altri processi (*problema dell'accesso alla sezione critica*)
- In un sistema distribuito, per acquisire la risorsa
 - non si può fare affidamento sull'uso di variabili condivise o su procedure fornite dal sistema operativo
 - L'algoritmo che gestisce la mutua esclusione deve fare affidamento solo sullo scambio di messaggi tra processi

La Mutua Esclusione Distribuita: modello di sistema

- N processi $p_1 \dots p_n$ che non condividono variabili
- I processi accedono a risorse condivise ma solo all'interno della sezione critica
- Assumiamo (per semplicità) che ci sia un'unica sezione critica
- Sistema Asincrono
- No Guasti
- Canali affidabili (ogni messaggio inviato sarà *eventually* consegnato integro ed esattamente una volta)
- Rete completamente interconnessa

La Mutua Esclusione Distribuita: proprietà del problema

- **Safety:**

- In ciascun istante di tempo, al più un processo può essere nella sezione critica

- **Liveness:**

- Un processo che fa richiesta per entrare in CS eventually ottiene il permesso

- A volte viene richiesto anche il soddisfacimento della proprietà di *fairness*

- **Fairness:**

- Le richieste di accesso alla sezione critica devono essere soddisfatte secondo l'ordine in cui sono arrivate (l'ordine è stabilito dalla relazione happened-before)

Classificazione di Algoritmi per il problema della mutua esclusione distribuita

- Algoritmi basati su clock logico
 - Si fa l'assunzione aggiuntiva di avere canali FIFO
 - Lamport
 - Ricart-Agrawala
- Algoritmi Token-based:
 - Impiego di una risorsa aggiuntiva: *token*
 - Il token è unico nel sistema per ogni istante di tempo
 - Algoritmo centralizzato
 - Algoritmo decentralizzato (Suzuki-Kasami)
- Algoritmi basati su Quorum
 - Si richiede il permesso di accedere alla sezione critica ad un insieme di processi
 - Algoritmo di Maekawa

Algoritmo di Ricart-Agrawala: implementazione

- Variabili locali
 - #replies (inizializzata a 0)
 - Stato \in {Requesting, CS, NCS} (inizializzato a NCS)
 - Q coda di richieste pendenti (inizialmente vuota)
 - Last_Req
 - Num
- Algoritmo

begin

1. Stato=Requesting
2. Num=num+1; Last_Req=num
3. $\forall i=1\dots N$ send REQUEST 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

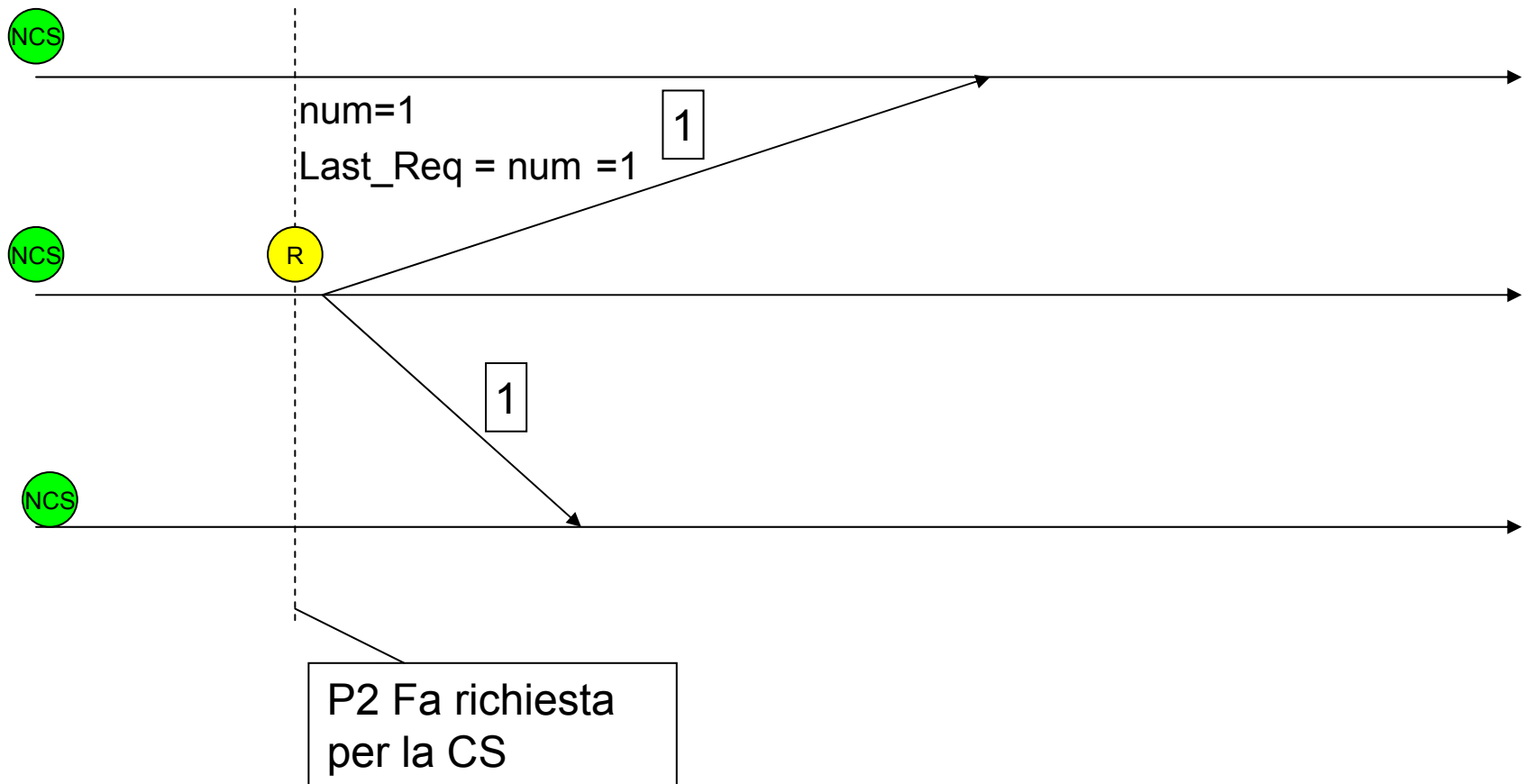
Upon receipt REQUEST(t) from p_j

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

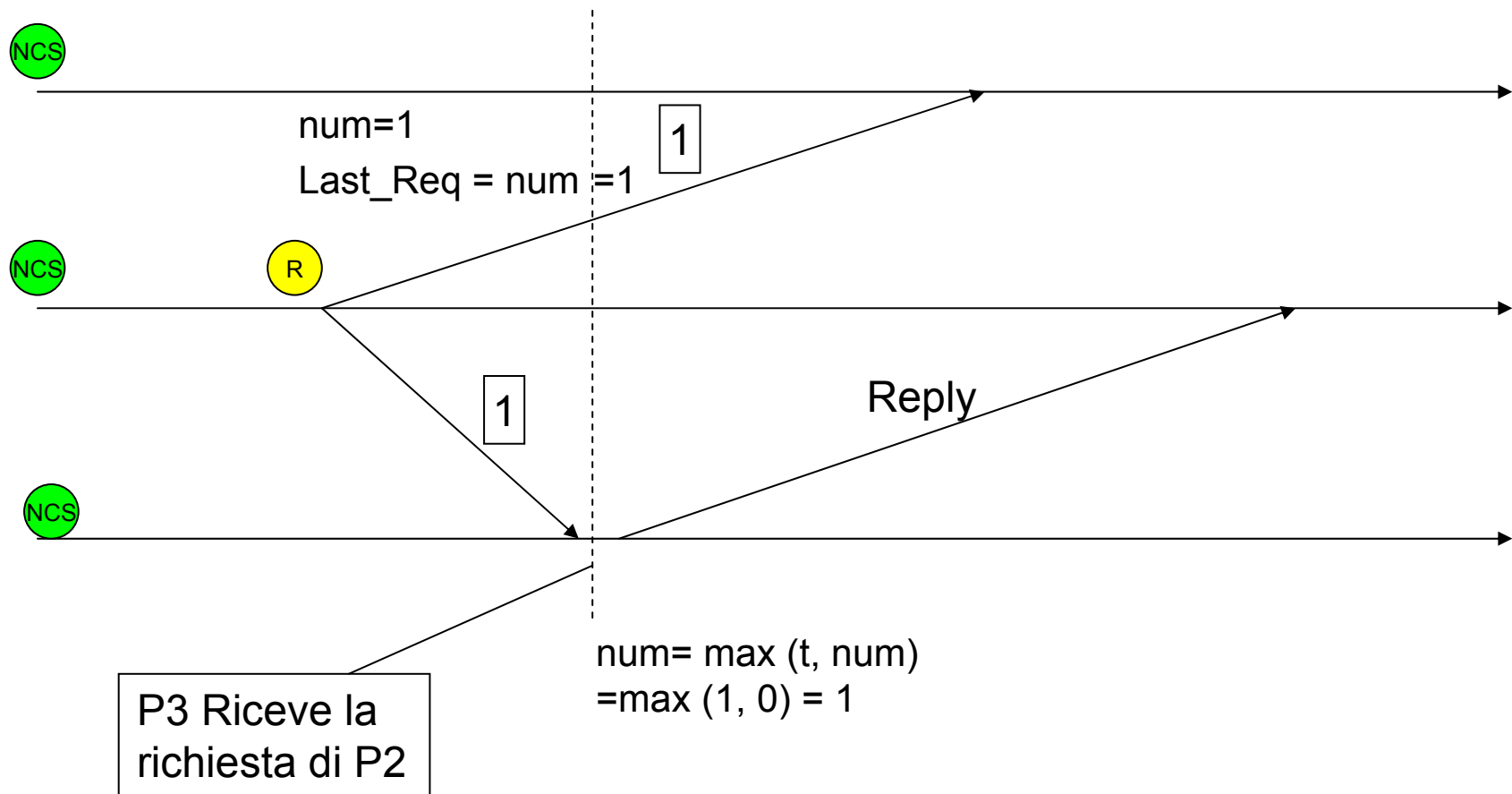
Upon receipt of REPLY from p_j

1. #replies=#replies+1

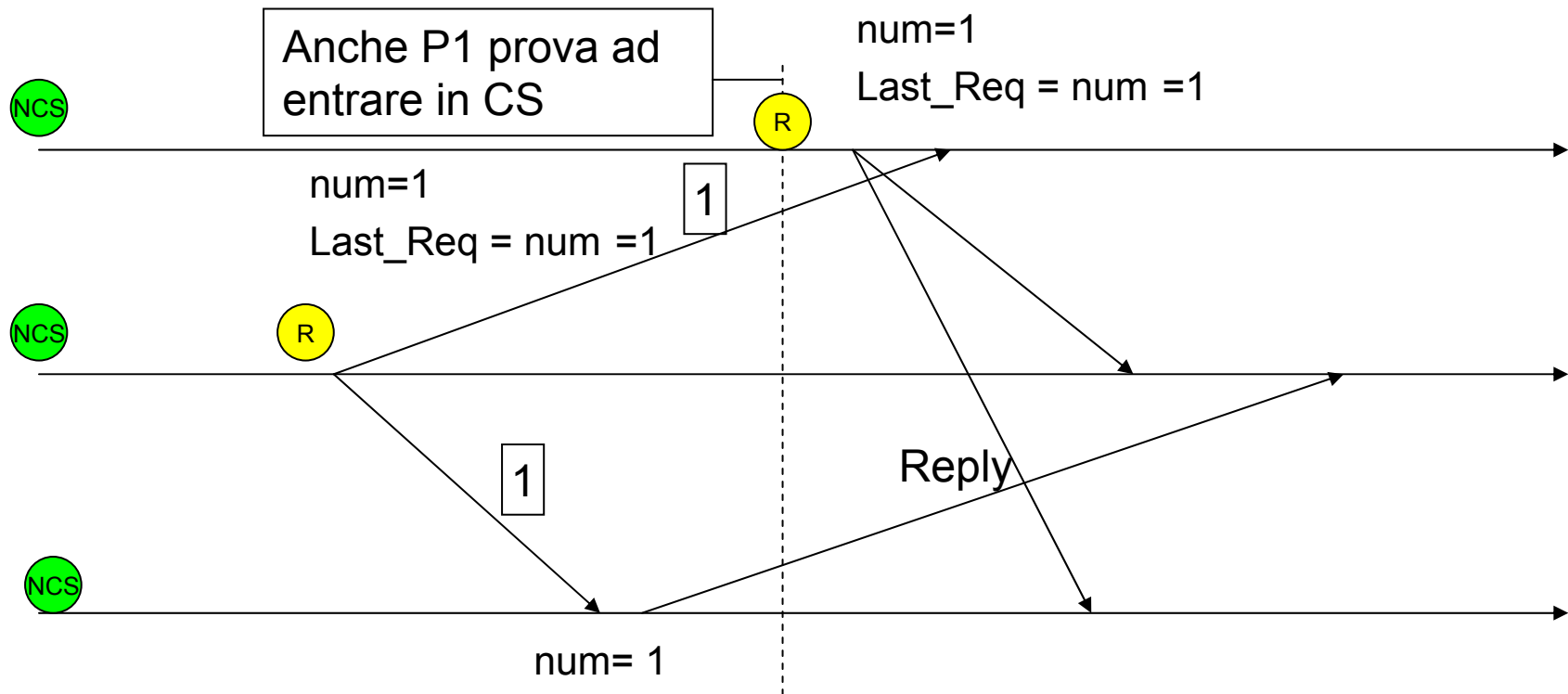
Algoritmo di Ricart-Agrawala: esempio



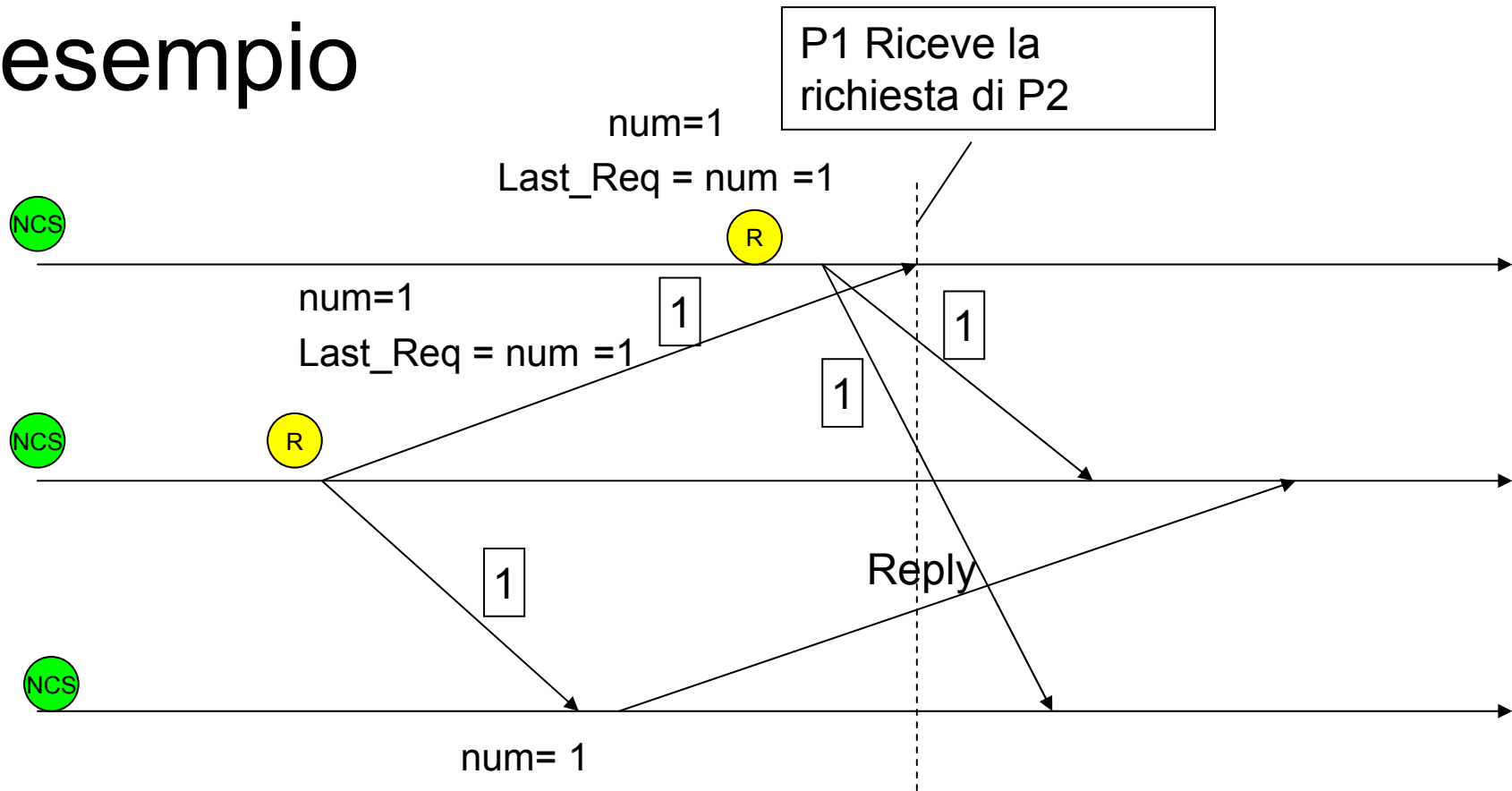
Algoritmo di Ricart-Agrawala: esempio



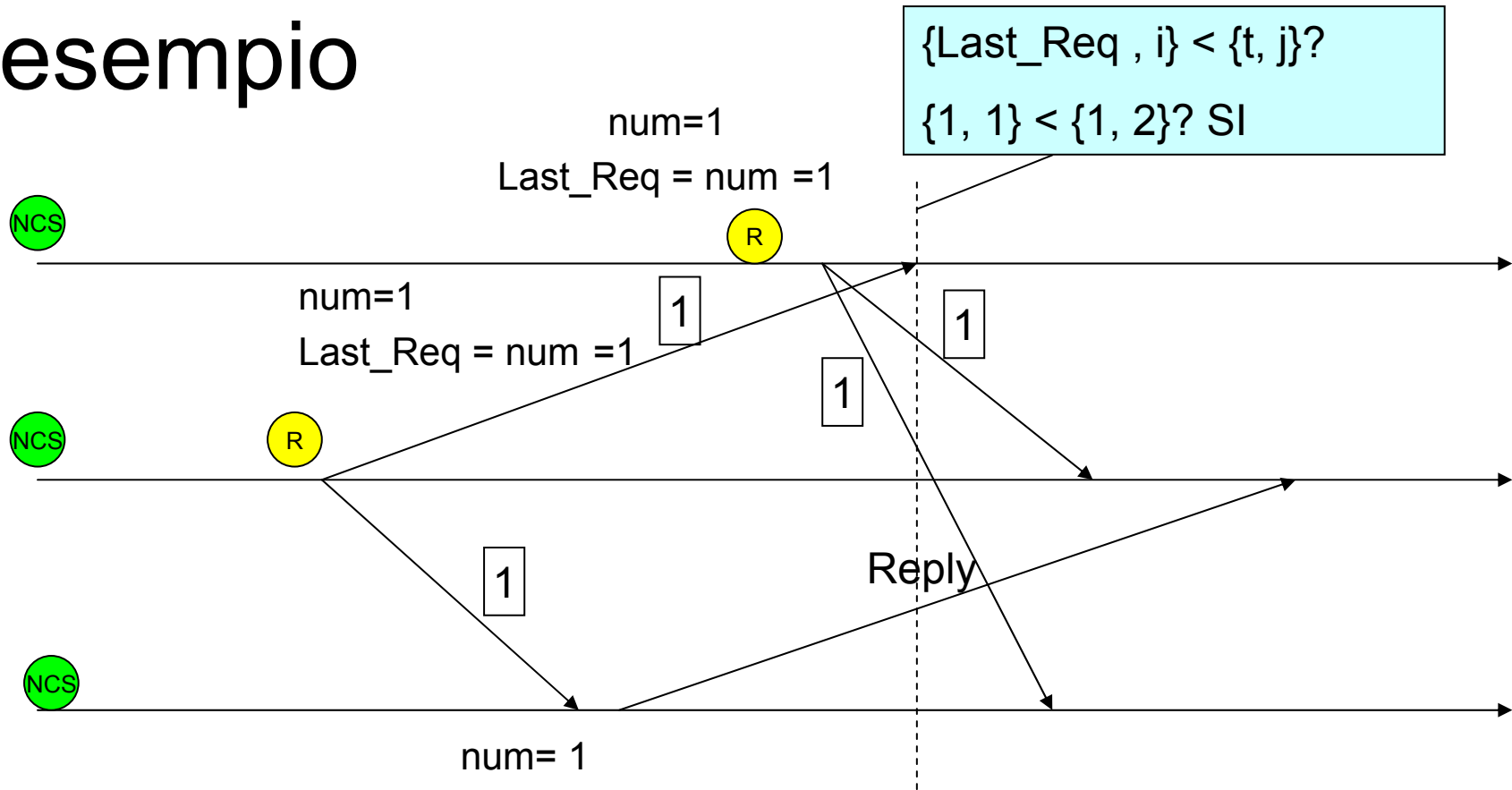
Algoritmo di Ricart-Agrawala: esempio



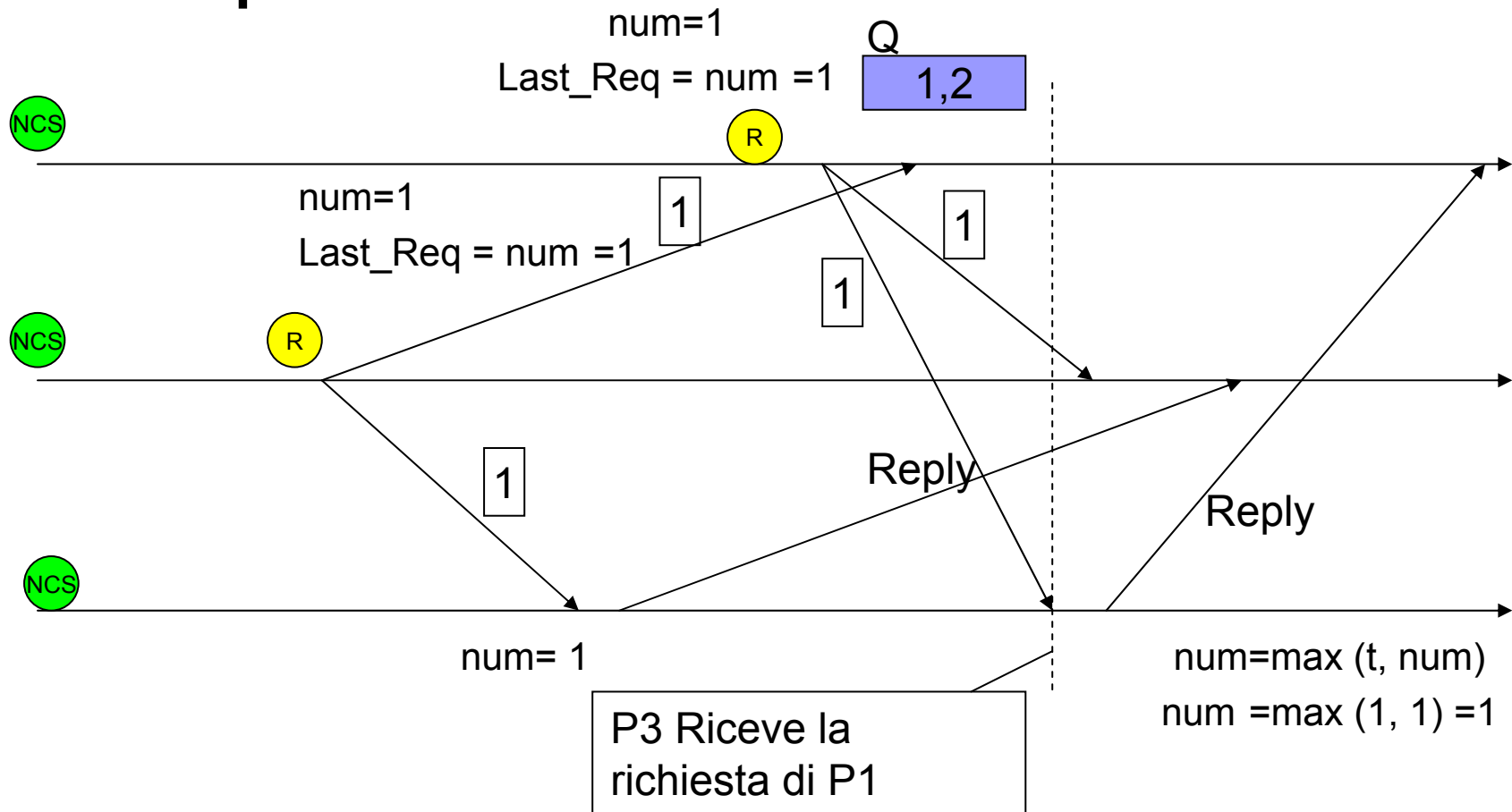
Algoritmo di Ricart-Agrawala: esempio



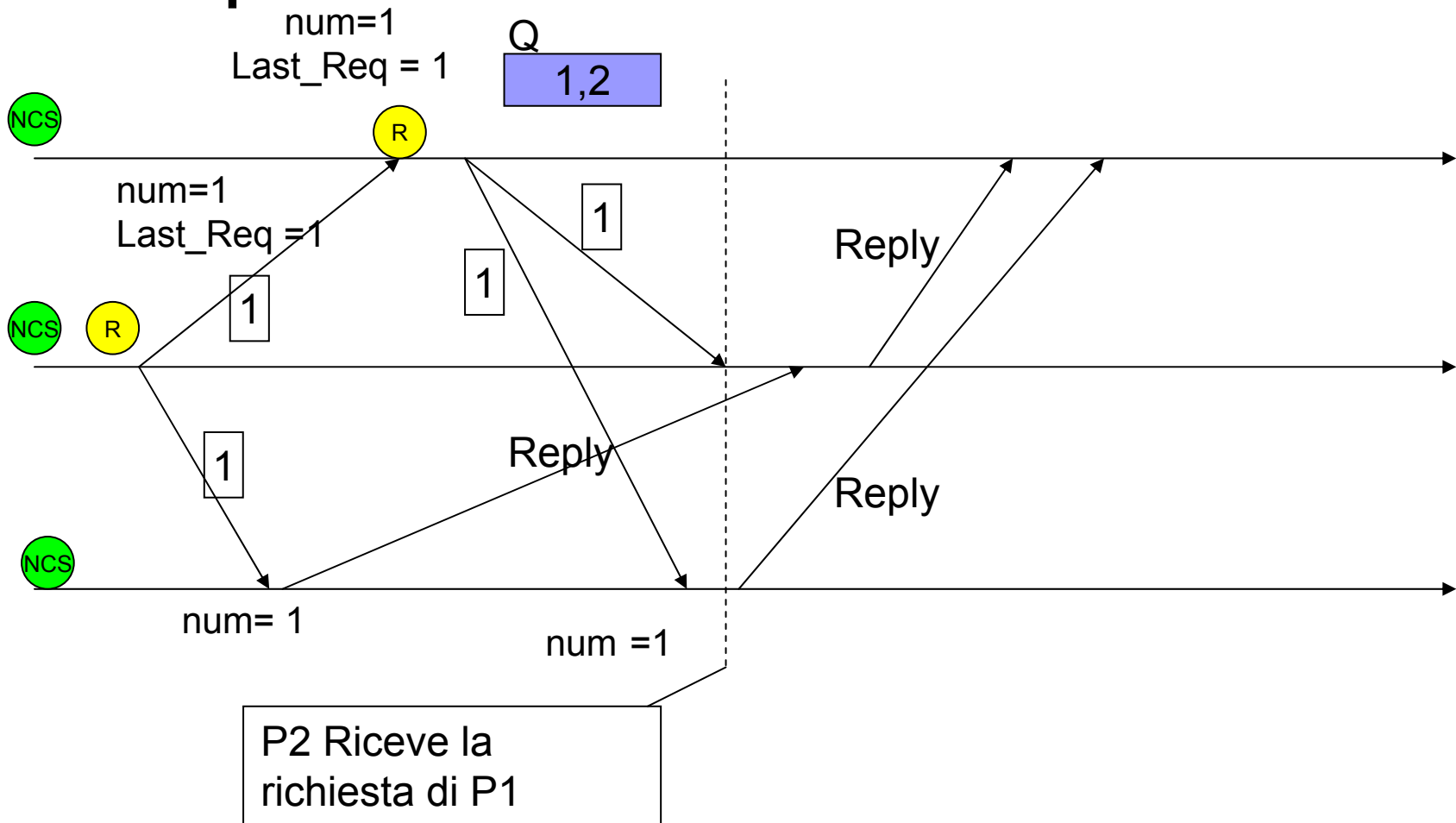
Algoritmo di Ricart-Agrawala: esempio



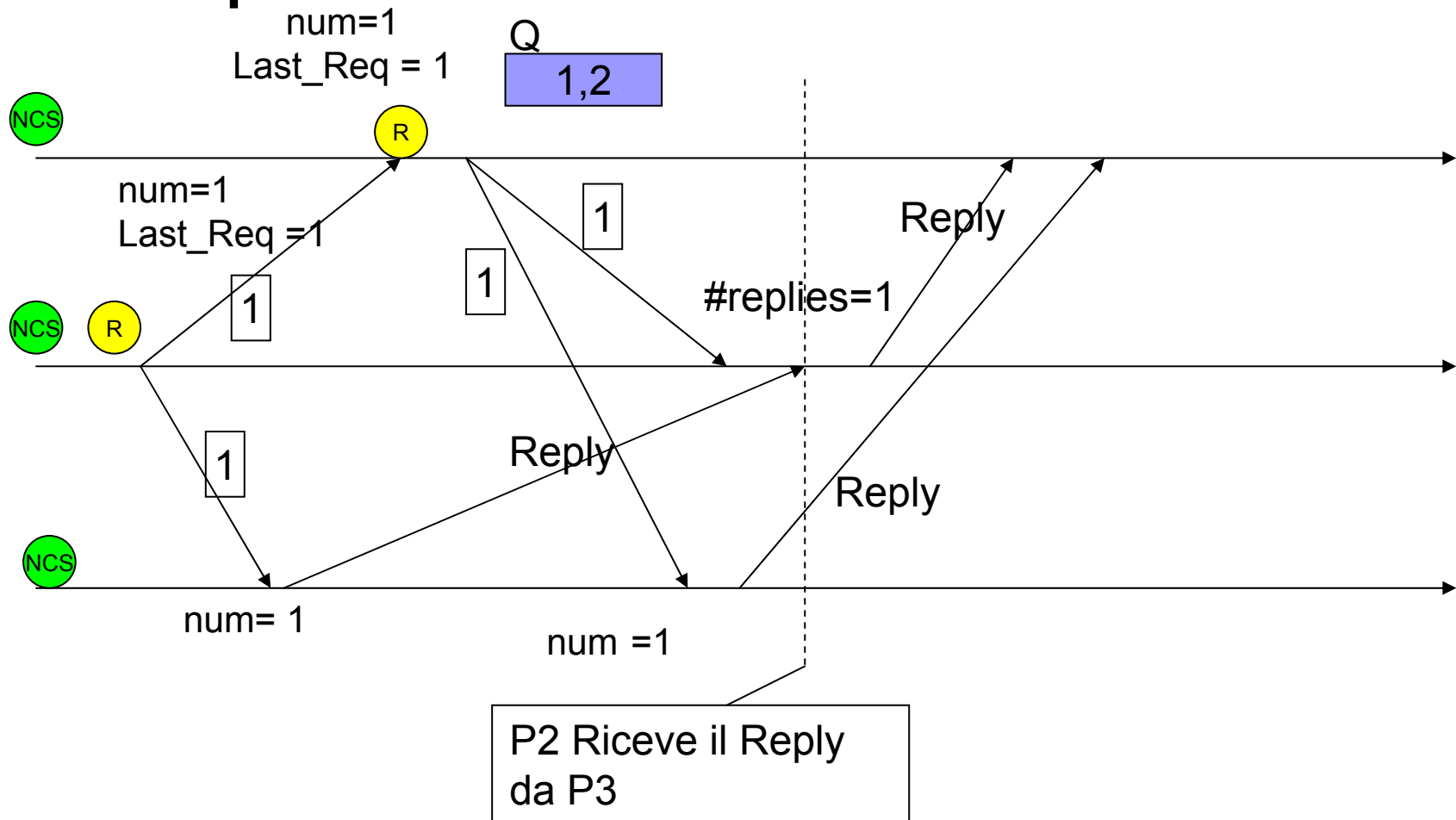
Algoritmo di Ricart-Agrawala: esempio



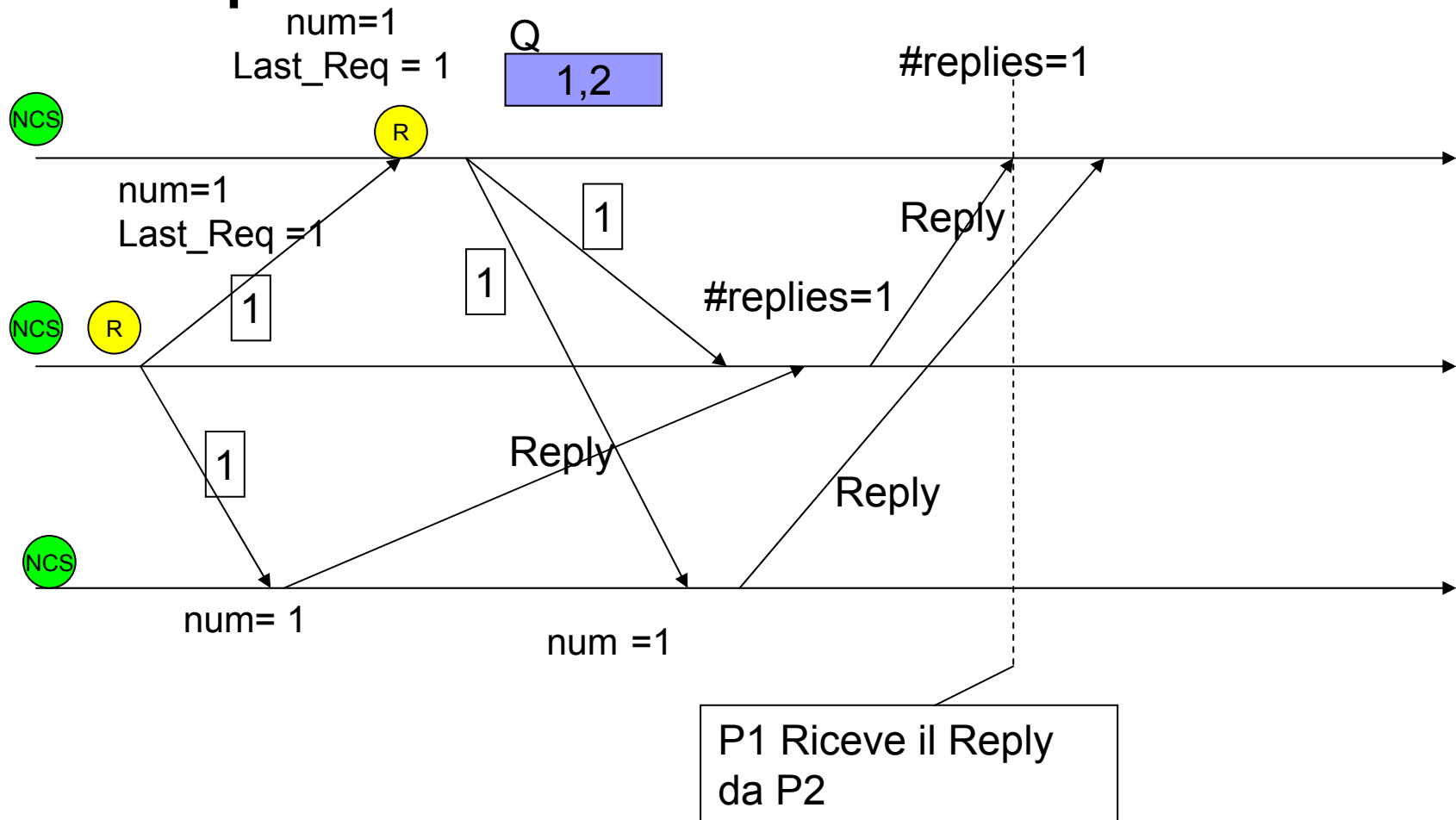
Algoritmo di Ricart-Agrawala: esempio



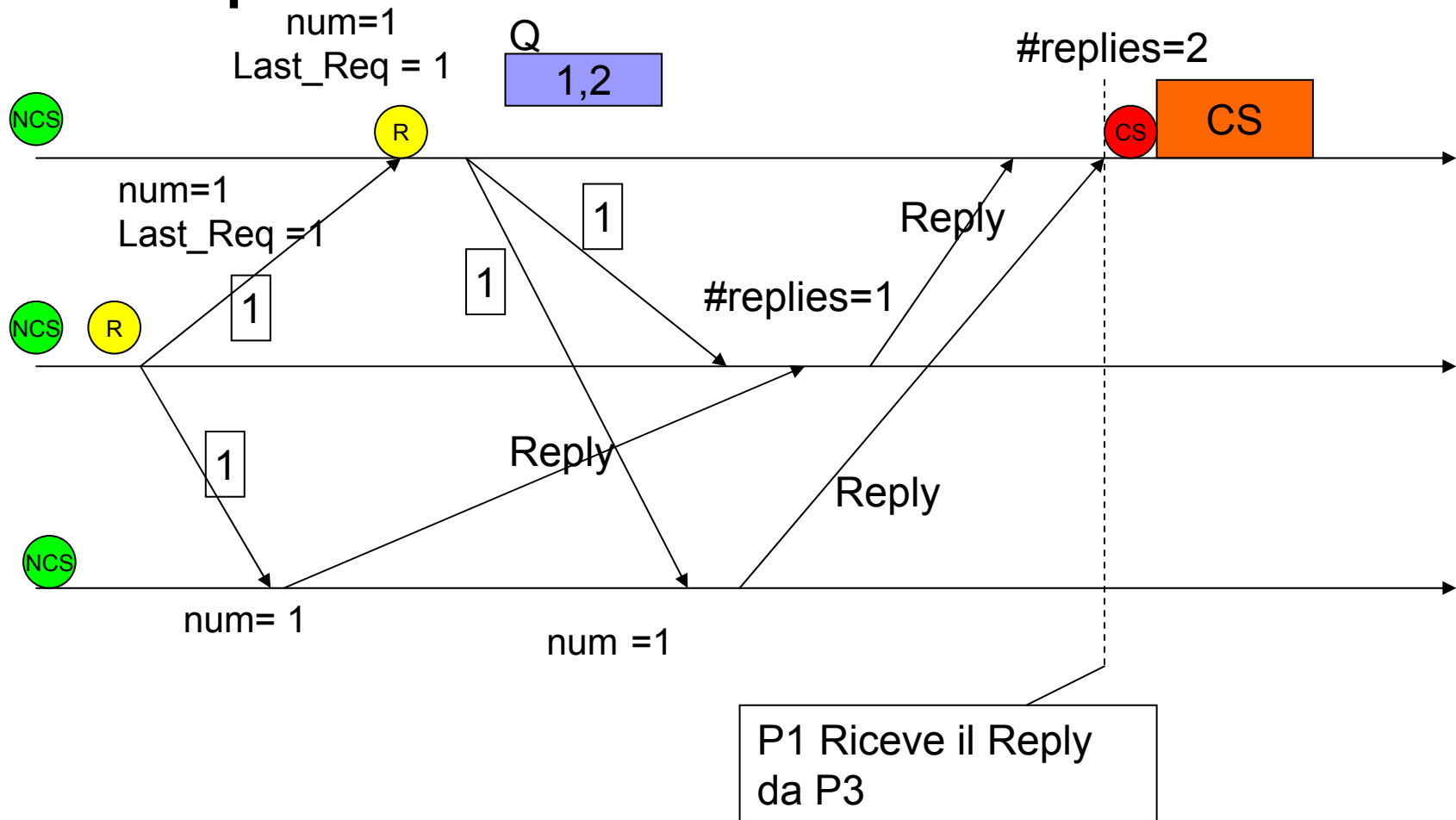
Algoritmo di Ricart-Agrawala: esempio



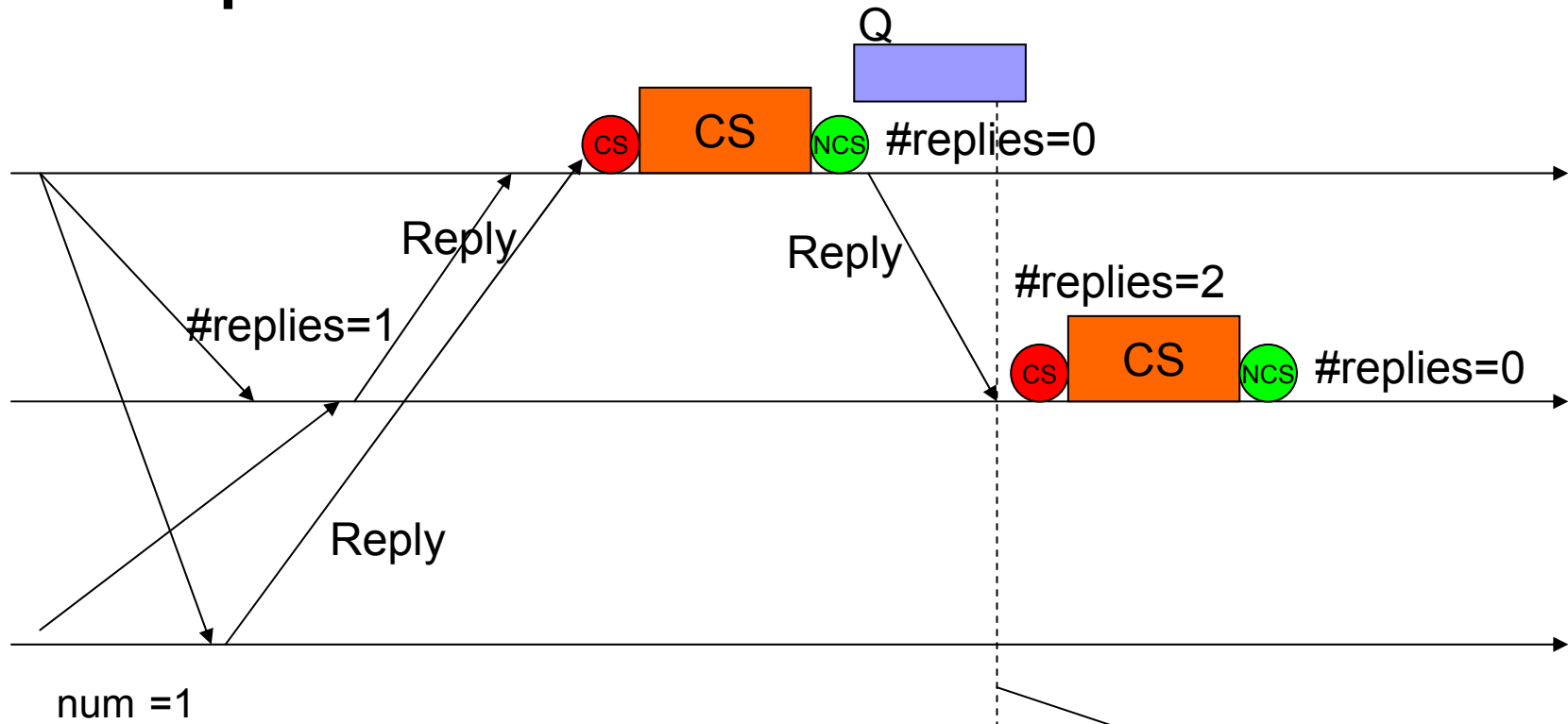
Algoritmo di Ricart-Agrawala: esempio



Algoritmo di Ricart-Agrawala: esempio



Algoritmo di Ricart-Agrawala: esempio



P2 riceve anche il
secondo Reply e
entra anche lui in
CS

Algoritmo token-based centralizzato

- Esiste un particolare processo che svolge il ruolo di *coordinatore* e che gestisce il token
- Il coordinatore tiene traccia delle richieste effettuate ed in particolare di:
 - Richieste effettuate ma non ancora servite
 - Richieste già servite
- Ogni processo p_i mantiene un suo vector clock in cui sono riportati i timestamp delle richieste.

Algoritmo token-based centralizzato

■ Idea:

- Quando un processo p_i vuole entrare in sezione critica fa richiesta al coordinatore
- Il coordinatore ne prende nota e lo inserisce nella lista delle richieste “pendenti” (ossia richieste effettuate ma non ancora servite)
- Quando la richiesta diventa eleggibile, il coordinatore invia il token a p_i che entrerà in CS
- Uscito dalla CS p_i restituirà il token al coordinatore

Algoritmo token-based centralizzato: implementazione del coordinatore

■ Strutture Dati:

- Reqlist
 - Memorizza la lista delle richieste ricevute ma non ancora servite
- V
 - Vettore di dimensione pari al numero di processi
 - Nella posizione $V[i]$ è memorizzato il numero di richieste di p_i già servite

■ Regole:

- Upon Token Request from p_i
 - $\text{Reqlist} = \text{Reqlist} \cup \{p_i, T_{p_i}\}$
- When Request of p_i is eligible ($T_{p_i} \leq V$) e il coordinatore ha il Token
 - Send Token to p_i

Algoritmo token-based centralizzato: implementazione dei client

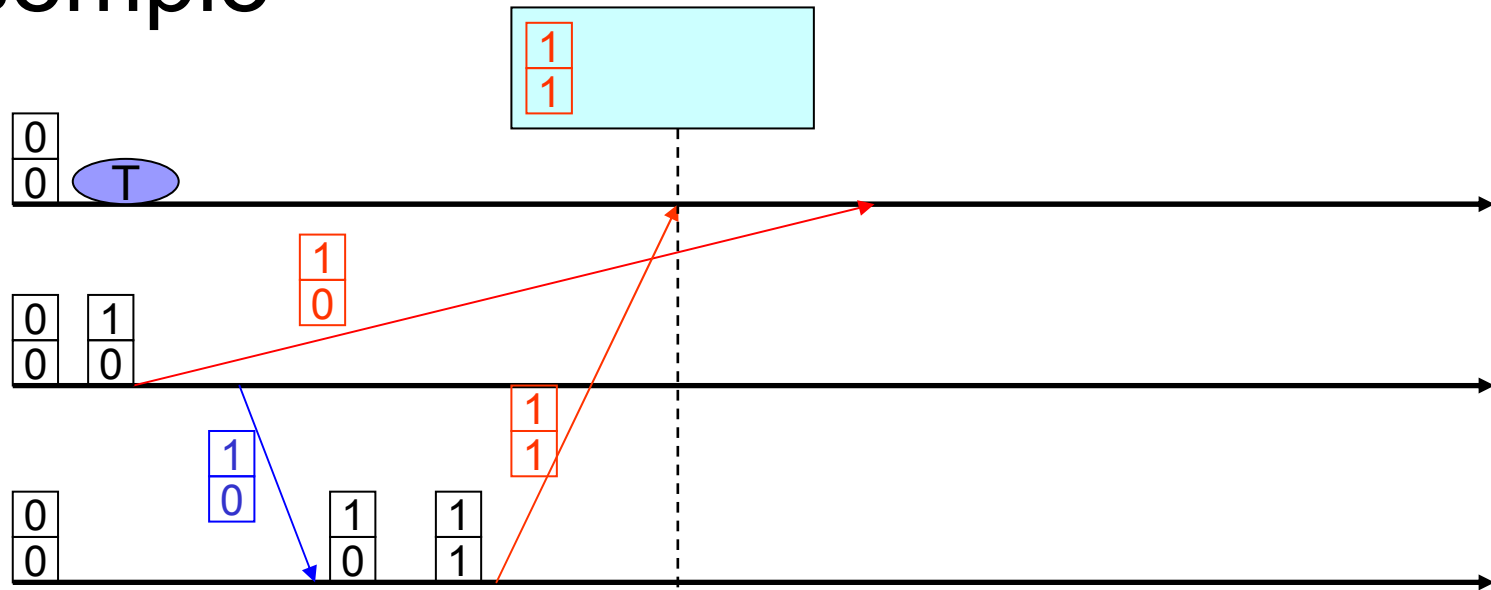
- **Strutture Dati:**

- VC
 - Vector Clock per i timestamp

- **Regole:**

- **Send Token-Request**
 - $VC[i] = VC[i] + 1$
 - Send m to Coordinato (with attached VC)
- **Token Reception**
 - Enter the CS
- **Exit CS**
 - Send token to Coordinator
- **Send Programm message m**
 - Attach VC to the message and send m
- **Programm Message Reception**
 - $VC = \max(VC, m.VC)$

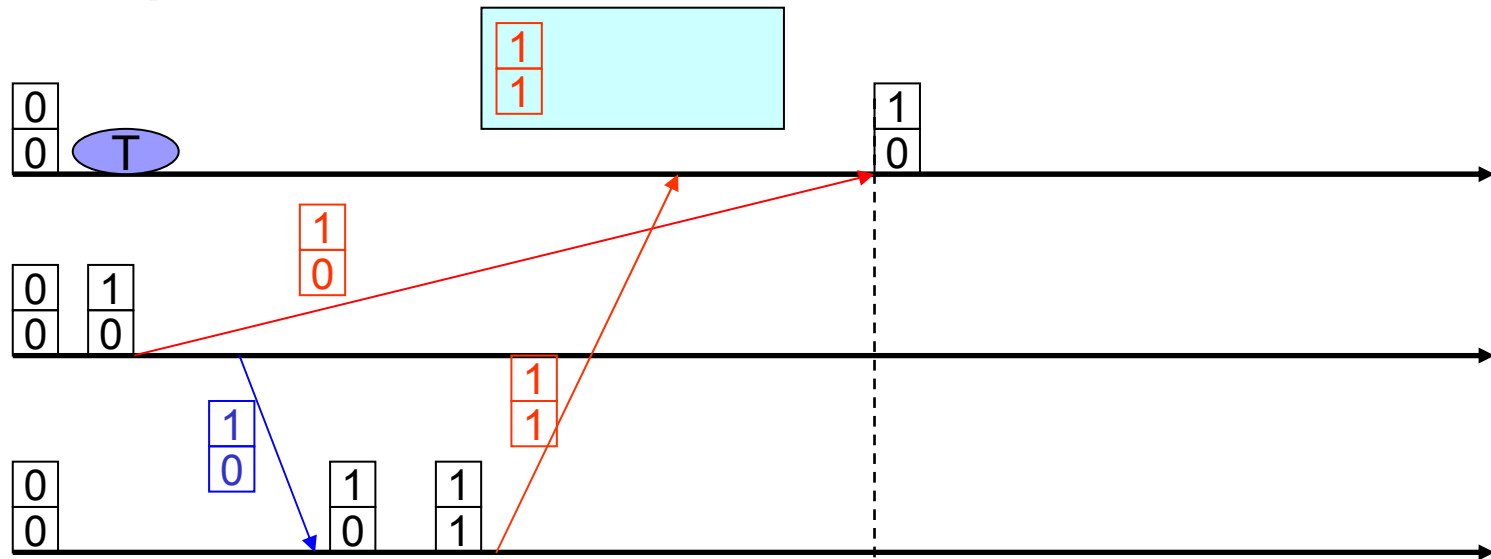
Algoritmo token-based centralizzato: esempio



→ Messaggio di Programma
→ Messaggio di Richiesta Token

$\begin{bmatrix} 0 \\ 0 \end{bmatrix} < \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ Il coordinatore memorizza la richiesta di p_2 perché non è ancora elegibile

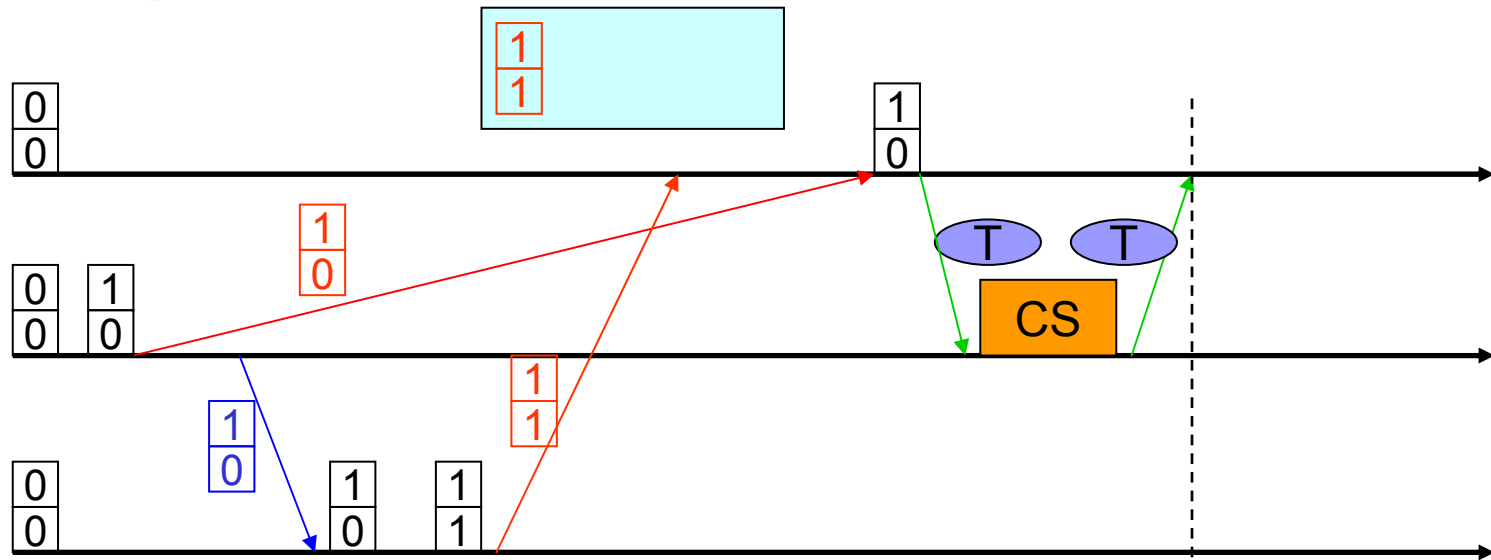
Algoritmo token-based centralizzato: esempio



$\begin{matrix} 0 \\ 0 \end{matrix} \leq \begin{matrix} 1 \\ 0 \end{matrix}$ Il coordinatore
 concede il token a p_1

→ Messaggio di Programma
→ Messaggio di Richiesta
 Token

Algoritmo token-based centralizzato: esempio

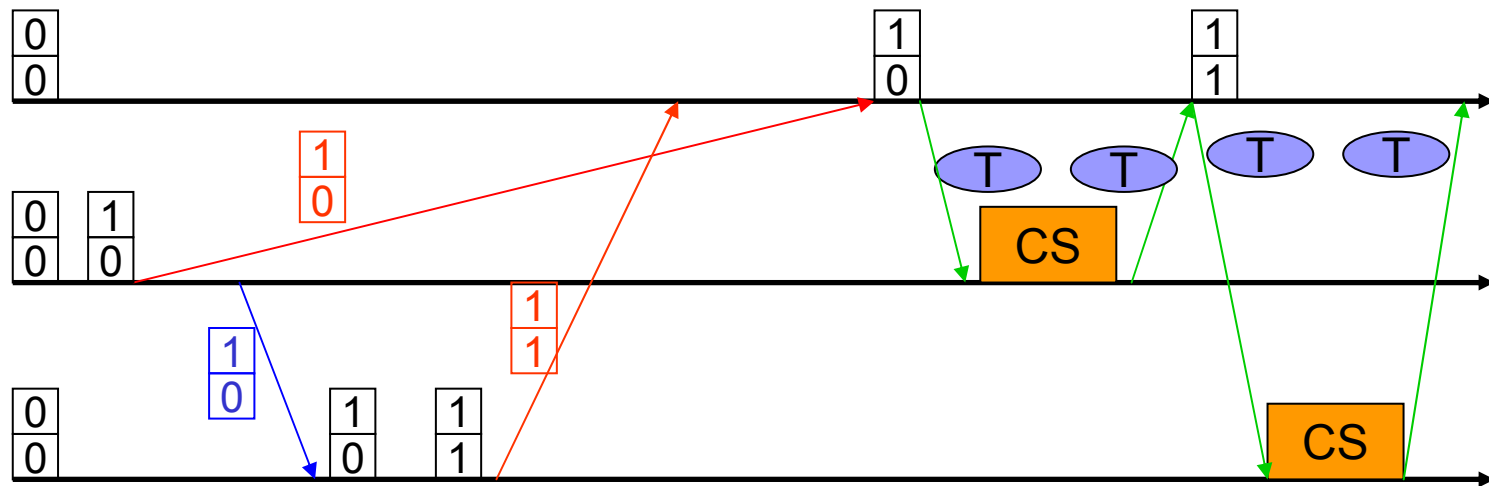


Il coordinatore controlla se la richiesta che aveva in coda è diventata elegibile e concede il token a p_2

$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

- Messaggio di Programma
- Messaggio di Richiesta Token
- Messaggio contenente il Token

Algoritmo token-based centralizzato: esempio



Entrambi i processi sono riusciti ad entrare nella sezione critica

- Messaggio di Programma
- Messaggio di Richiesta Token
- Messaggio contenente il Token

Algoritmo token-based centralizzato: verifica ME (safety)

- Supponiamo per assurdo che due processi p_i e p_j entrino contemporaneamente in sezione critica
- Per le regole dell'algoritmo p_i ha fatto richiesta a p_0 e ottenuto il token ma anche p_j ha fatto richiesta a p_0 e ottenuto il token *prima che p_i abbia rilasciato la sezione critica*
- Se p_i è in CS allora p_i non ha ancora inviato il messaggio di restituzione del token a p_0 (regola 3 del codice del client)
- Se p_i è in CS e per la proprietà dei canali di non duplicare nè generare messaggi spuri abbiamo che p_0 ha inviato il token a p_i (regola 2 del coordinatore e regola e regola 2 del client)
- Ma se p_0 ha già inviato il Token a p_i allora p_0 non possiede il token al momento della richiesta di p_j
- Quindi se p_0 non ha il token e per la proprietà dei canali di non duplicare nè generare messaggi spuri allora p_j non ha ricevuto il token da p_0 quindi non lo possiede e non può essere in CS.

⇒ **Contraddizione**

OSS: La safety è garantita dalla proprietà di UNICITA' del token

Algoritmo token-based centralizzato: verifica NS (Liveness)

- Supponiamo per assurdo che un processo p_i fa richiesta di accesso alle sezione critica ma non ottiene mai l'accesso, i.e. non ottiene mai il token
- Poichè i canali sono reliable l'unica ragione possibile è che il coordinatore non gli manda mai il token
- Il coordinatore mette in coda la richiesta di p_i ma non la serve mai
- La richiesta di p_i non è mai elegibile
- Qualche richiesta accaduta prima di quella di p_i (secondo l'happened before) non arriva mai al coordinatore. Ma da canali affidabili e no guasti tutte le richieste alla fine raggiungono il coordinatore
- Allora il numero di richieste che precedono la richiesta di p_i è infinito. Ma dalla proprietà di aciclicità dell'happened before segue che il numero di richieste che fanno parte del passato causale di quella di p_i è finito

⇒ **Contraddizione**

- Si noti che la liveness è garantita dalla proprietà di ESISTENZA del token

Algoritmo token-based centralizzato: commenti

- Il coordinatore rappresenta il collo di bottiglia del sistema ed è un single point of failure
- L'algoritmo è efficiente in termini di numero di messaggi scambiati, solo 3
- Si riesce a garantire anche la fairness

Algoritmo token-based decentralizzato: Suzuki-Kasami

- Non essendoci un processo coordinatore, tutte le strutture dati necessarie al mantenimento del Token sono contenute nel Token stesso
- Quando un processo vuole entrare in CS, e non è lui il detentore del token, manda un messaggio di richiesta a tutti gli altri
- Il processo che ha il token
 - se non è in CS risponde inviando il token a chi l'aveva richiesto
 - Se è in CS invia il token solo dopo esserne uscito
- **Problemi:**
 - Distinguere le richieste nuove da quelle obsolete
 - Determinare chi è il processo che ha la richiesta che deve essere servita

Algoritmo di Suzuki-Kasami: implementazione (1/2)

■ Strutture Dati del processo p_i

- $R[]$: array di dimensione N in cui nella posizione k è contenuto il numero di sequenza dell'ultima richiesta fatta da p_k

■ Strutture dati contenute nel Token

- $L[]$: array di dimensione N in cui nella posizione k è contenuta la richiesta, fatta da p_k , eseguita più di recente
- Q : coda di richieste pendenti

Algoritmo di Suzuki-Kasami: implementazione (2/2)

- Regole del processo p_i
 - Richiesta del Token
 - $R_i[i] = R_i[i] + 1$
 - $\forall j = 1 \dots N$ send (Request, i , $R_i[i]$) to p_j
 - Ricezione della richiesta del Token da p_j
 - $R_i[j] = R_i[j] + 1$
 - **If** (Token $\in p_i$) \wedge ($p_i \neg CS$)
 - **Then** send Token to p_j
 - Release della CS
 - $L[i] = R_i[i]$
 - **If** $R_i[j] = L[j] + 1$
 - **Then** $Q.enqueue(\text{Request}, j, R_j[j])$
 - **If** $Q \neq \emptyset$
 - **Then** $p_k = Q.dequeue$ and Send Token to p_k

Algoritmo di Suzuki-Kasami: correttezza

■ Proprietà di ME (Safety) - intuizione

- Un processo entra in sezione critica solo se ha il token ed il token è concesso solo quando il processo che era in CS esce.
- Poiché il token è unico nel sistema abbiamo che un solo processo alla volta può entrare in CS.

■ Proprietà di NS (Liveness) – intuizione

- I canali reliable assicurano che prima o poi la richiesta raggiungerà tutti i processi.
- L'esecuzione della CS ha un tempo finito quindi il processo che possiede il token prima o poi accoderà le richieste avvenute mentre era in CS.
- Il numero di richieste accodate è finito quindi prima o poi il processo verrà servito.

■ Osservazione:

- L'algoritmo non garantisce fairness
- Si può estendere in modo che supporti anche fairness

Extended Suzuki-Kasami

■ Strutture Dati del processo p_i

- **Vc []** : vector clock per i timestamp
- **Req**: lista delle richieste non ancora servite

■ Strutture dati contenute nel Token

- Il token contiene le strutture dati del processo coordinatore nel caso centralizzato
 - **Reqlist**: Memorizza la lista delle richieste ricevute ma non ancora servite
 - **V**: Vettore di dimensione pari al numero di processi; nella posizione $V[i]$ è memorizzato il numero di richieste di p_i già servite

Extended Suzuki-Kasami: implementazione

□ Richiesta token:

- incrementa il vector clock e invia rich.+timestamp a tutti i processi (incluso se stesso)

□ Ricezione richiesta token di P_j :

- aggiorna il vector clock prendendo il max tra il valore corrente e quello del timestamp della richiesta. Se non ha il token inserisce la rich. in reqlisti. Se ha il token accoda la richiesta nella reqlist del token quindi se non è nella sez. critica controlla se la rich. è elegibile.

□ Invio token a P_j :

- solo se la richiesta di P_j è elegibile e P_i non è in sezione critica. Le strutture dati del token vengono aggiornate prima dell'invio:
 - $reqlist = reqlist - richiesta\ elegibile\ di\ P_j$
 - $V[j] = V[j] + 1$

Extended Suzuki-Kasami: implementazione

■ Ricezione token:

- aggiorna il token (inserisce le richieste presenti in reqlisti che non sono ancora state servite in reqlist) ed entra nella sezione critica

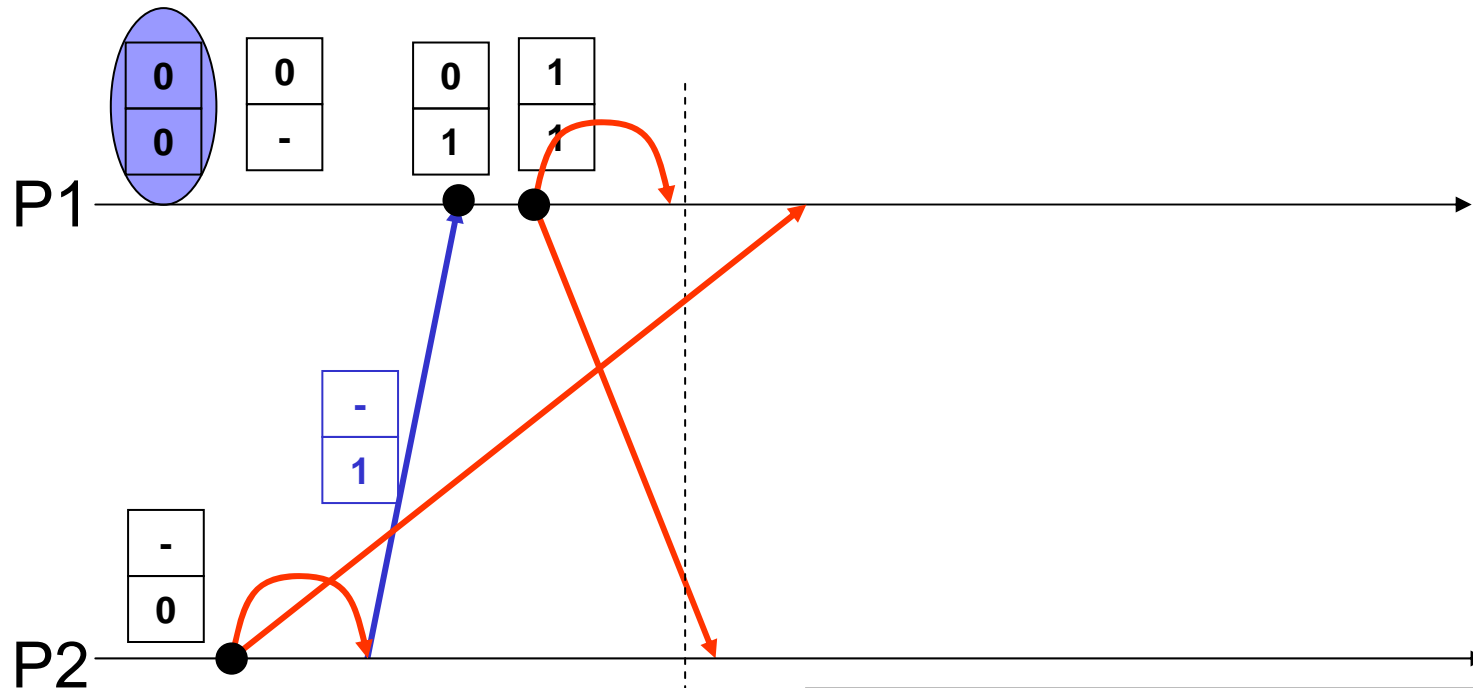
■ Invio messaggio di programma:

- piggyback del valore corrente del vector clock

■ Ricezione messaggio di programma:

- aggiorna il vector clock prendendo il max tra il valore corrente e quello del timestamp della richiesta

Extended Suzuki-Kasami: esempio



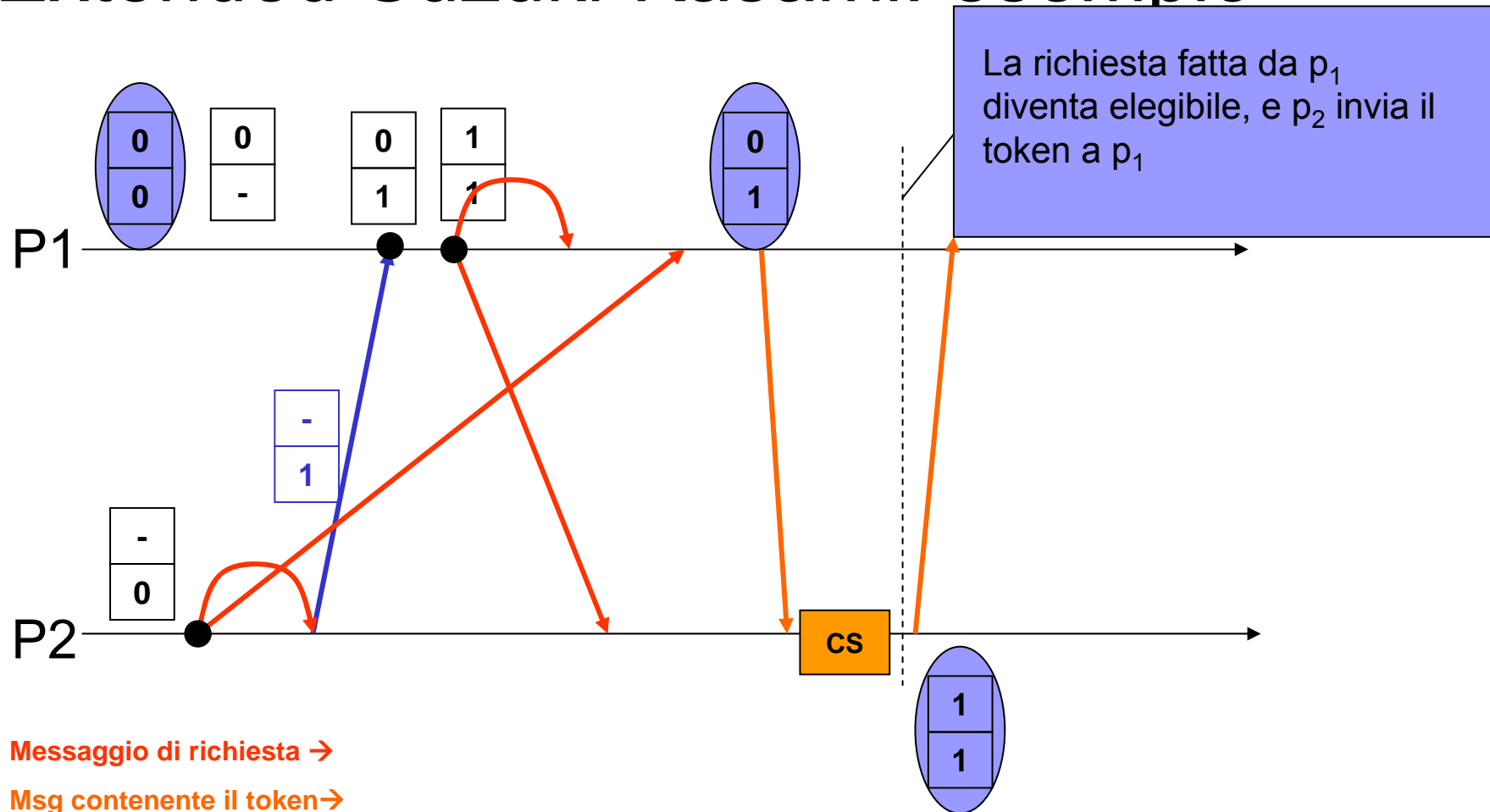
Messaggio di richiesta →

Msg contenente il token →

Messaggio del programma →

p_1 riceve una richiesta da se stesso *non* elegibile, il timestamp associato è infatti maggiore di V sulla 2° componente e non entra in CS

Extended Suzuki-Kasami: esempio



La richiesta fatta da p_1 diventa elegibile, e p_2 invia il token a p_1

Messaggio di richiesta →

Msg contenente il token →

Messaggio del programma →

Suzuki-Kasami: performance

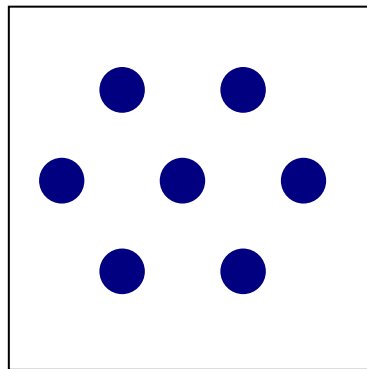
- Il numero di messaggi per l'esecuzione di una sezione critica è 0 o N , quindi nel caso peggiore la complessità relativa ai messaggi è N ($N-1$ per la richiesta e 1 messaggio di token)
- Sebbene sia decentralizzato è vulnerabile al guasto del processo che possiede il token
- Abbiamo visto come il token permetta facilmente di sequenzializzare l'accesso alla sezione critica

Algoritmi basati su Quorum

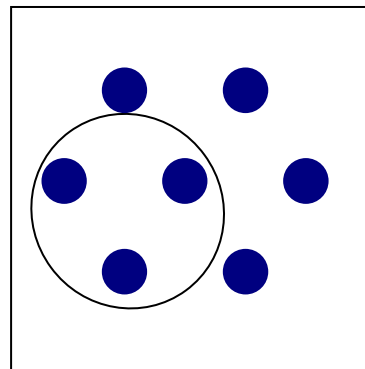
- L'idea di base di questi algoritmi è quella di chiedere il permesso per entrare in CS a un insieme di processi (chiamati *request set*)
- **Problema:**
 - Come scelgo chi sono i processi a cui chiedere?
- **Idea:**
 - Devo costruire dei request set a intersezione non vuota in modo da avere sempre almeno un processo a conoscenza di tutto

Sistemi a Quorum

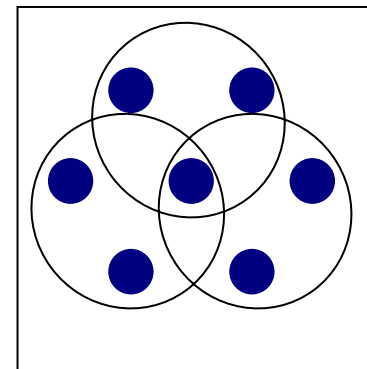
- P : insieme di processi p_1, p_2, \dots, p_n
- Quorum Q : sottoinsieme di P
- Coterie C : insieme di quorum Q_1, Q_2, \dots, Q_m
 - Proprietà di una coterie:
 - Minimalità: nessun quorum è un sottoinsieme di un altro quorum.
 - Intersezione: ogni coppia di quorum ha intersezione non nulla.



Insieme P



Quorum Q



Coterie C

Algoritmo di Maekawa

- Semplice strategia in un sistema a quorum: richiedere il permesso ad una maggioranza di processi. In questo caso il *request set di ogni processo* è qualsiasi sottoinsieme di processi con almeno $\lceil (N+1) / 2 \rceil$. In questo caso il numero di msg scambiati diminuisce...Ma si può fare di meglio?
- Maekawa: idee alla base dell'algoritmo:
 - se ogni nodo **pi** ottiene il permesso da un certo request set **Si**, allora trovare la **dimensione minima** di tale request set per un dato sistema significa trovare un lower bound sul numero di messaggi necessari per eseguire una sezione critica
 - Nota che la mutua esclusione completamente distribuita (no presenza di processi coordinatori) implica stesso *sforzo* e stessa *responsabilità* per ogni processo, qualsiasi sia la dimensione del request set dal quale ottenere il permesso. Quindi tutti i nodi dovrebbero avere assegnato un request set della stessa dimensione (sforzo) ed ogni nodo potrebbe partecipare allo stesso numero di request sets (responsabilità).

Algoritmo di Maekawa

- Per un numero di processi N , ad ogni processo p_i è associato un request set S_i tale che:
 - Per ogni i e j t.c. $1 \leq i, j \leq N$, $S_i \cap S_j \neq \emptyset$ (intersezione non nulla per ogni coppia di set)
 - Sforzo equo richiede che $|S_1| = \dots = |S_N| = K$, dove K è il numero di elementi di ogni set
 - Responsabilità equa richiede che ogni processo p_i , $1 \leq i \leq N$ sia nello stesso numero K di set S_j per $1 \leq j \leq N$
 - Per minimizzare la trasmissione dei messaggi è inoltre richiesto che un set S_j , $1 \leq j \leq N$ includa sempre il processo p_j come membro

Algoritmo di Maekawa

- Maekawa mostrò che la soluzione ottimale che minimizza K si ottiene dalla relazione tra N e K : $N = K(K - 1) + 1$ che è pari alla dimensione di un set moltiplicato il numero dei set ai quali un processo deve partecipare meno 1 (poichè partecipa al suo proprio set) più 1 (per contare il processo stesso)
- Quindi $K \approx \sqrt{N}$ (l'algoritmo di Maekawa è chiamato l'algoritmo della radice quadrata)
- Quindi solo per alcuni N esiste una soluzione ottimale
 - Per $K=1..\infty$ si potrebbero trovare tutti gli N per i quali $N=K(K-1)+1$
 - $K = 2, N = 2(2 - 1) + 1 = 3$
 - $K = 3, N = 3(3 - 1) + 1 = 7$

Algoritmo di Maekawa: esempio di coterie

- Nell'es. per $K = 3$ $N=7$:

$\{1,2,3\}$	S_1	$\{1,4,5\}$	S_4	$\{1,6,7\}$	S_6
$\{2,4,6\}$	S_2	$\{2,5,7\}$	S_5	$\{3,4,7\}$	S_7
$\{3,5,6\}$	S_3				

- Si noti che la coterie presa soddisfa tutt'e 4 le proprietà viste

Algoritmo di Maekawa: regole

- 3 tipi di messaggi:
 - REQUEST
 - REPLY
 - RELEASE
- Quando un p_i processo vuole accedere alla sezione critica manda un messaggio REQUEST agli altri $K-1$ membri di S_i .
- Quando un processo p_j riceve un messaggio REQUEST da un processo p_i , p_j manda immediatamente un messaggio di REPLY indietro, a meno che non si trovi esso stesso in sezione critica o abbia già inviato un messaggio di REPLY ad un altro processo che non ha ancora rilasciato la sezione critica (ha già votato). Nel caso non gli mandi immediatamente un messaggio di REPLY allora p_j accoda la richiesta di p_i .
- Per rilasciare la sezione critica un processo p_i invia un messaggio di RELEASE a tutti gli altri $K-1$ membri di S_i
- Quando un processo riceve un msg di RELEASE allora elimina la prima entry dalla coda e manda un msg di REPLY al processo la cui richiesta è stata appena cancellata

Algoritmo di Maekawa: pseudo-codice

On initialization

state := RELEASED;

voted := FALSE;

For p_i *to enter the critical section*

state := WANTED;

Multicast *request* to all processes in $S_i - \{p_i\}$;

Wait until (number of replies received = $(K - 1)$);

state := HELD;

On receipt of a request from p_i *at* p_j ($i \neq j$)

if (*state* = HELD or *voted* = TRUE)

then

 queue *request* from p_i without replying;

else

 send *reply* to p_i ;

voted := TRUE;

end if

For p_i *to exit the critical section*

state := RELEASED;

Multicast *release* to all processes in $S_i - \{p_i\}$

On receipt of a release from p_i *at* p_j ($i \neq j$)

if (queue of requests is non-empty)

then

 remove head of queue – from p_k , say;

 send *reply* to p_k ;

voted := TRUE;

else

voted := FALSE;

end if

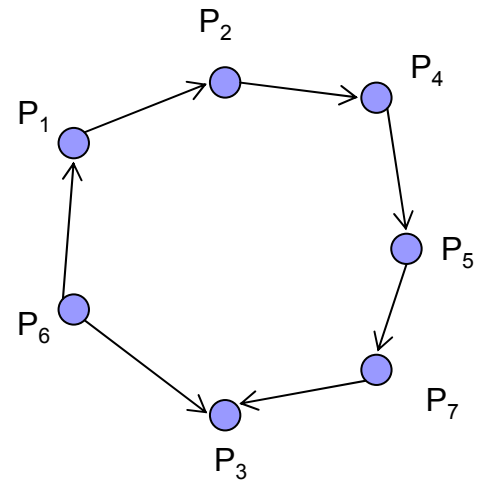
Algoritmo di Maekawa: correttezza

- **ME (sketch):** Supponi per contraddizione che due processi p_i e p_j stiano eseguendo contemporaneamente la sezione critica. Dalla proprietà di intersezione della coterie segue che $S_i \cap S_j = \{p_k\}$. Quindi p_k deve aver inviato due messaggi di REPLY sia a p_i che a p_j senza che nessuno dei due abbia rilasciato la sezione critica. Contraddizione.
- **Fairness:** non è supportata
 - **Saunders 1987:** algoritmo deadlock-free che serve le richieste in happened-before order

Algoritmo di Maekawa: correttezza

ND: l'algoritmo soffre di deadlock!!!

- si consideri il sistema visto nell'esempio precedente. Se tutti i processi richiedono contemporaneamente l'accesso alla sezione critica allora è possibile che
 - p1 riceve la REPLY da p6
 - p2 riceve la REPLY da p1
 - p3 riceve la REPLY da p7
 - p4 riceve la REPLY da p2
 - p5 riceve la REPLY da p4
 - p6 riceve la REPLY da p3
 - p7 riceve la REPLY da p5
- Ogni processo ha ricevuto una su due REPLY e nessuno può andare avanti!
- Estensioni dell'algoritmo risolvono il problema scambiando messaggi aggiuntivi



Algoritmo di Maekawa: performance

- numero di msg per sezione critica: $3\sqrt{N}$
- $3\sqrt{N}$ è minore di $2(N-1)$ (Ricart-Agrawala) per $N > 4$
- Anche con i msg addizionali per risolvere deadlock, la soluzione rimane migliore dal punto di vista della message complexity per reti di larga scala