

Logical Time in Distributed Systems

Sistemi di Calcolo (II semestre)

Daniele Cono D'Elia

Riccardo Lazzeretti

Material taken from:

Leonardo Querzoni, Roberto Baldoni - Distributed systems courses

Singhal Kshemkalyani, “Distributed Computing - Principles, Algorithms, and Systems”, Chapter 3

Il tempo nei sistemi distribuiti

- In un sistema distribuito è impossibile avere un unico clock fisico condiviso da tutti i processi.
- Eppure la computazione globale può essere vista come un ordine totale di eventi se si considera il tempo al quale sono stati generati.
- Per molti problemi risalire a questo tempo è di vitale importanza - o comunque importante - poter stabilire quale evento è stato generato prima di un altro.
 - esempio: analisi di logs per debugging

Il tempo nei sistemi distribuiti

- Soluzione 1:
 - Sincronizziamo con una certa approssimazione i clock fisici locali attraverso opportuni algoritmi.
 - Ogni processo poi allega un timestamp contenente il valore del proprio clock fisico locale ad ogni messaggio che invia.
 - I messaggi vengono ordinati secondo il valore dei loro timestamp
- Questo meccanismo funziona solo se l'errore di approssimazione dei clock fisici è limitato (e noto), altrimenti si rischia di alterare l'ordine degli eventi a causa di timestamps errati.

Il tempo nei sistemi distribuiti

- È sempre possibile mantenere l'approssimazione dei clock limitata?
- In un modello asincrono NO!
 - Un sistema distribuito è *asincrono* quando non è possibile stabilire un upper bound a:
 - tempo di esecuzione di ciascuno step di un processo
 - tempo di propagazione di un messaggio in rete
 - velocità di deriva di un clock
- In un modello asincrono il timestamping non si può basare sul concetto di tempo fisico.

Il tempo nei sistemi distribuiti

- Soluzione 2:
 - Il timestamping avviene etichettando gli eventi con il valore corrente di una variabile opportunamente aggiornata durante la computazione.
 - Questa variabile, poiché completamente scollegata dal comportamento del clock fisico locale, è chiamata **clock logico**.
- Questa soluzione, non essendo basata sul concetto di tempo reale e non richiedendo sincronizzazione tra i clock fisici, è adatta ad un uso in sistemi asincroni.

Clock fisico

All'istante di tempo reale t , il sistema operativo legge il tempo dal clock hardware $H_i(t)$ del computer, quindi produce il software clock:

$$C_i(t) = \alpha H_i(t) + \beta$$

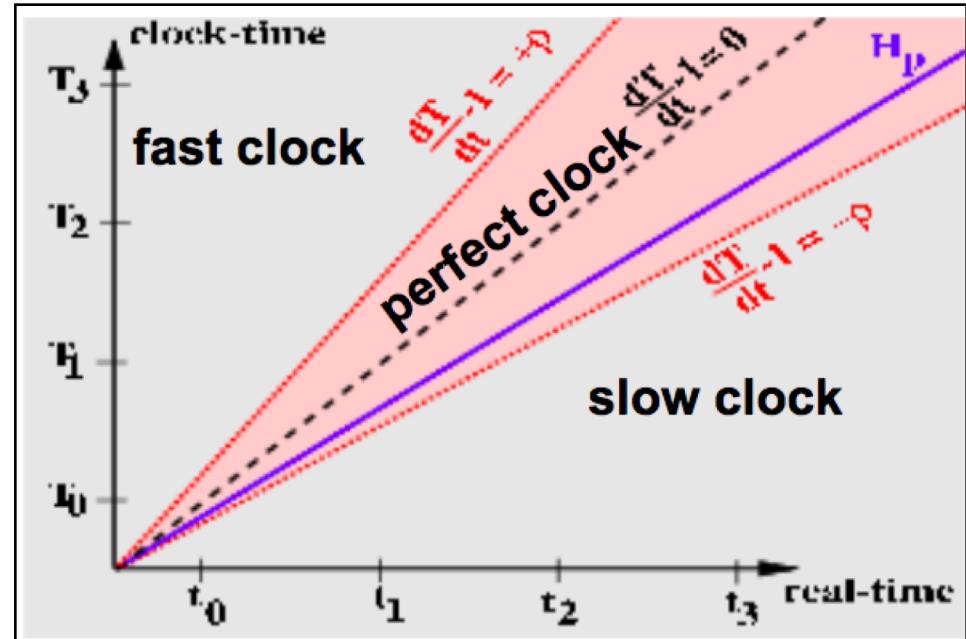
che approssimativamente misura l'istante di tempo fisico t per il processo P_i .

Clock fisico

- Il clock hardware $H_i(t)$ è un componente fisico (oscillatore al quarzo) caratterizzato da un parametro detto drift rate:
 - l'oscillatore è caratterizzato da una sua frequenza che fa divergere il clock hardware dal tempo reale.
 - il drift rate misura il disallineamento del clock hardware da un orologio ideale per unità di tempo.
 - Normali orologi al quarzo deviano di circa 1 sec in 11-12 giorni. (10^{-6} secs/sec).
 - Orologi al quarzo ad alta precisione hanno un drift rate di circa 10^{-7} o 10^{-8} secs/sec.
 - Gli orologi atomici hanno un drift rate di meno di 10^{-9} secs/giorno

Clock fisico

- Un clock hw $H_i(t)$ è corretto se il suo drift rate si mantiene all'interno di un limite $\rho > 0$ finito.
(es. 10^{-6} secs/ sec).



- Se il clock $H_i(t)$ è corretto allora l'errore che si commette nel misurare un intervallo di istanti reali $[t, t']$ è limitato:
 - » $(1 - \rho) (t' - t) \leq H_i(t') - H_i(t) \leq (1 + \rho) (t' - t)$ supposto $(t < t')$

Clock fisico

- Per il clock software $C_i(t)$ spesso basta una condizione di monotonicità
- $t' > t$ implica $C_i(t') > C_i(t)$
 - » Es. condizione richiesta da Unix make: 200 files compilati alle 17:00 e $C_i(17:00)=17:30$, 3 modificati alle 17:15. Se la monotonicità non fosse garantita e $C_i(17:15)=17:20$ nessun file viene ricompilato !
- Si potrebbe garantire monotonicità con un clock hardware non corretto scegliendo opportunamente i valori α e β .

Clock fisico

- Quanto deve essere la risoluzione del clock (periodo che intercorre tra gli aggiornamenti del valore del clock) per poter distinguere due differenti eventi?
 - » Tempo di risoluzione < intervallo di tempo che intercorre tra due eventi rilevanti
- Clock guasto: se non rispetta le condizioni di correttezza
 - crash failure - un clock che smette di funzionare
 - arbitrary failure – qualsiasi altro comportamento non previsto (es. Y2K bug che dopo il 31/1/1999 passa a 1/1/1900 invece di 1/1/2000)
- Nota: corretto ≠ accurato

UTC

- UTC (Coordinated Universal Time) è uno standard internazionale per mantenere il tempo
 - Basato su International Atomic Time, quindi è basato su orologi atomici ma è occasionalmente aggiustato utilizzando il tempo astronomico.
 - L'output dell'orologio atomico è inviato in broadcast da stazioni radio su terra e da satelliti (es. GPS).
 - Computer con ricevitori possono sincronizzare i loro clock con questi segnali
 - Segnali da stazioni radio su terra hanno un'accuratezza di circa 0.1-10 millisecondi.
 - Segnali da GPS hanno un'accuratezza di circa 1 microsecondo.

Sincronizzazione dei clock fisici

- È possibile sincronizzare i clock di processi appartenenti ad un sistema distribuito seguendo due differenti strategie:
 - **Sincronizzazione esterna** - I clock C_i (per $i=1, \dots, N$) sono sincronizzati con una sorgente di tempo S , in modo che, dato un intervallo I di tempo reale: $|S(t) - C_i(t)| < D$ per $i=1, \dots, N$ e per tutti gli istanti t in I , cioè i clock C_i hanno un'accuratezza compresa nell'intervallo D .
 - **Sincronizzazione interna** - I clock di due computer sono sincronizzati l'uno con l'altro in modo che $|C_i(t) - C_j(t)| < D$ per $i=1, \dots, N$ nell'intervallo I . In questo caso i due clock C_i e C_j si accordano all'interno dell'intervallo D .

Sincronizzazione dei clock fisici

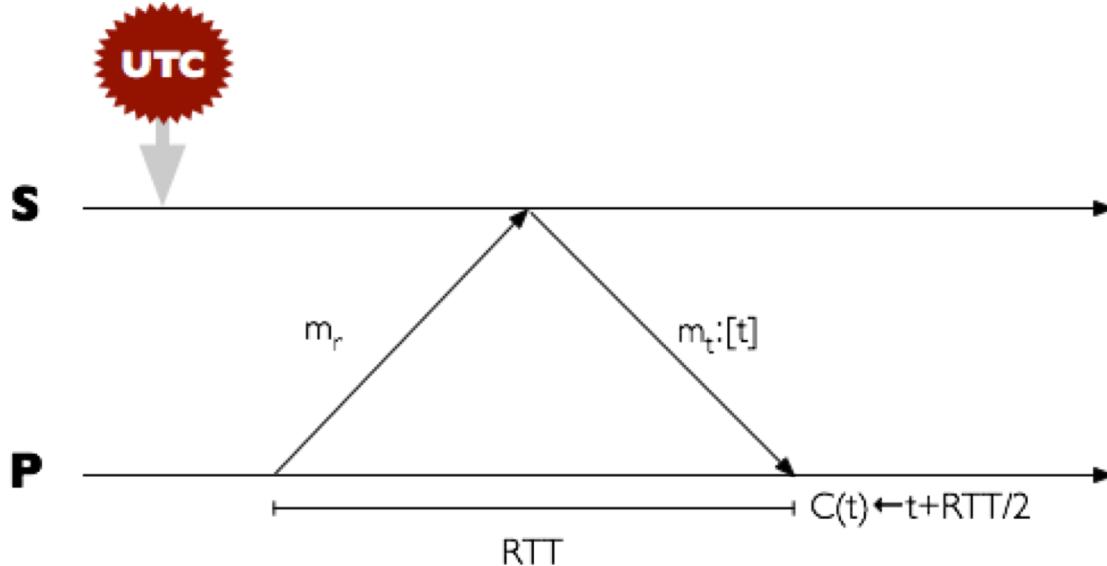
- I clock sincronizzati internamente non sono necessariamente esternamente sincronizzati.
 - Tutti i clock possono deviare collettivamente da una sorgente esterna sebbene rimangano sincronizzati tra loro entro l'intervallo D.
 - Se l'insieme dei processi è sincronizzato esternamente entro un intervallo D allora segue dalle definizioni che è anche internamente sincronizzato entro un intervallo $2D$.

Time services

- Il gruppo di processi che deve sincronizzarsi fa uso di un Time Service. Il Time Service può essere a sua volta implementato da un solo processo (server) oppure può essere implementato in modo decentralizzato da più processi
 - Time Service centralizzato
 - Request-driven (algoritmo di Cristian) - sync esterna
 - Broadcast-based (Berkeley Unix algorithm) - sync interna
 - Time Service decentralizzato
 - Network Time Protocol - sync esterna

Algoritmo di Cristian

- Time server centralizzato e passivo. Algoritmo:
 - Il time server S riceve il segnale da una sorgente UTC
 - Un processo P richiede il tempo con m_r e riceve t in $m_t:[t]$ da S
 - P imposta il suo clock a $t + RTT/2$
- RTT è il round trip time misurato tra P e S

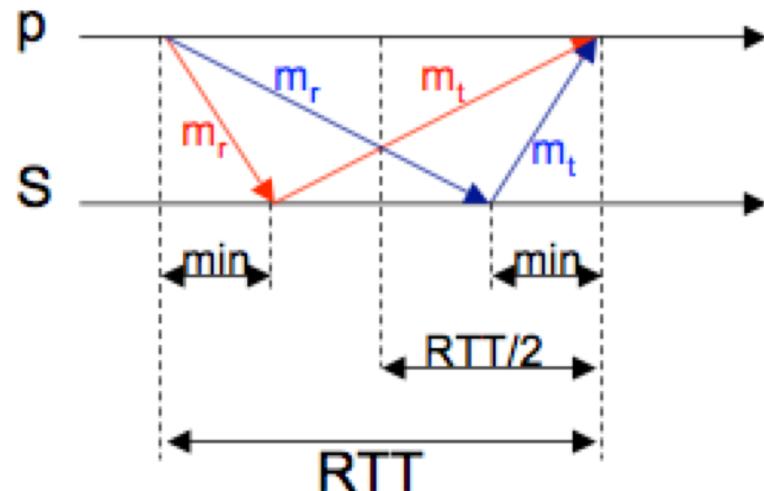


Algoritmo di Cristian

- Accuratezza

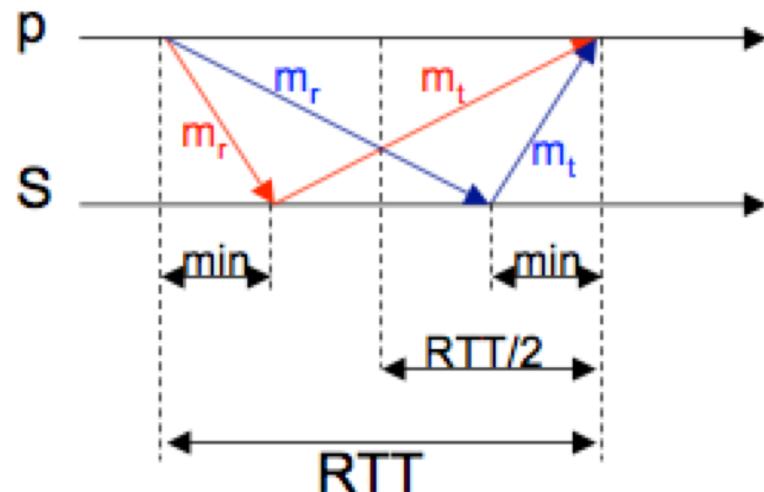
- Caso 1 - Il tempo impiegato dal messaggio di ritorno è maggiore rispetto alla stima fatta utilizzando RTT/2 ed in particolare è uguale a (RTT-min)

$$\Delta = \text{stima del messaggio di ritorno} - \text{tempo reale} = (RTT/2) - (RTT - \text{min}) = -RTT/2 + \text{min} = -(RTT/2 - \text{min})$$



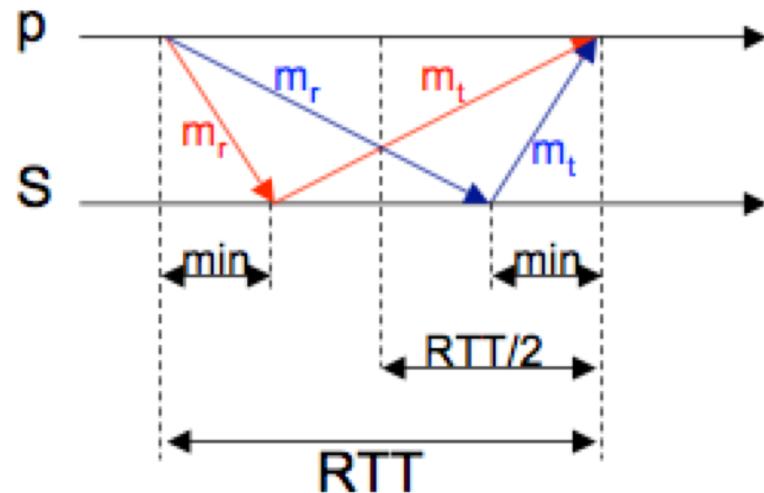
Algoritmo di Cristian

- Accuratezza
 - Caso 2 - Il tempo impiegato dal messaggio di ritorno è minore rispetto alla stima fatta utilizzando RTT/2 ed in particolare è uguale a (min)
 $\Delta = \text{stima del messaggio di ritorno} - \text{tempo reale} = (\text{RTT}/2) - \text{min} = +(\text{RTT}/2 - \text{min})$



Algoritmo di Cristian

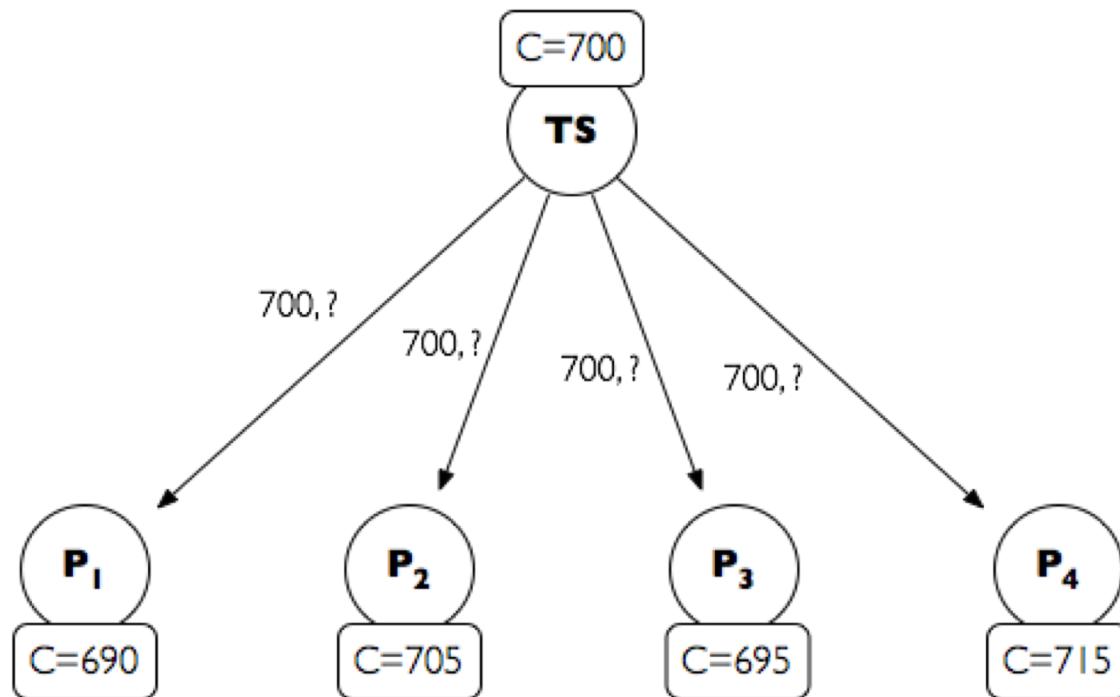
- Accuratezza
 - Il tempo di S quando m_t arriva a P è compreso nell'intervallo $[t+\min, t + \text{RTT} - \min]$
 - l'ampiezza di tale intervallo è $\text{RTT} - 2\min$
 - accuratezza $\leq \pm (\text{RTT}/2 - \min)$



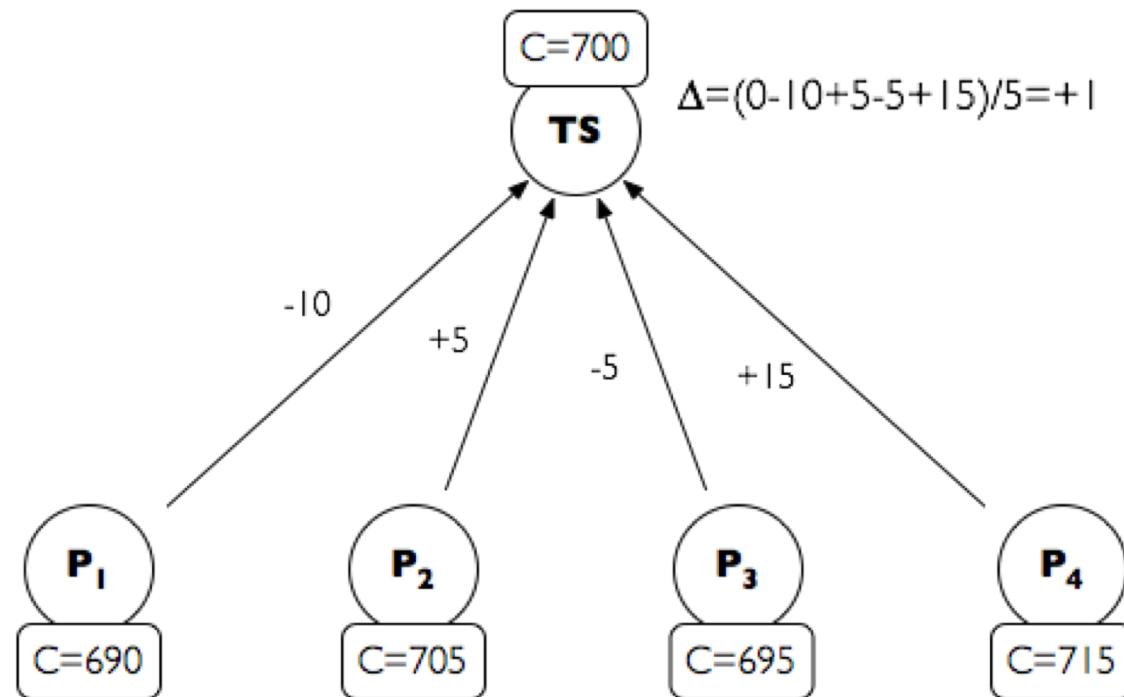
Algoritmo di Berkeley

- Algoritmo per la sincronizzazione interna di un gruppo di computer
 - Il time server è centralizzato e attivo (master)
 - Il master invia a tutti gli altri processi (slaves) il proprio valore locale e richiede gli scostamenti dei loro clock da questo valore;
 - riceve tutti gli scostamenti e usa il RTT per stimare il valore del clock di ciascun slave;
 - calcola il valore medio;
 - invia a tutti gli slaves gli scostamenti necessari per sincronizzare i clock.

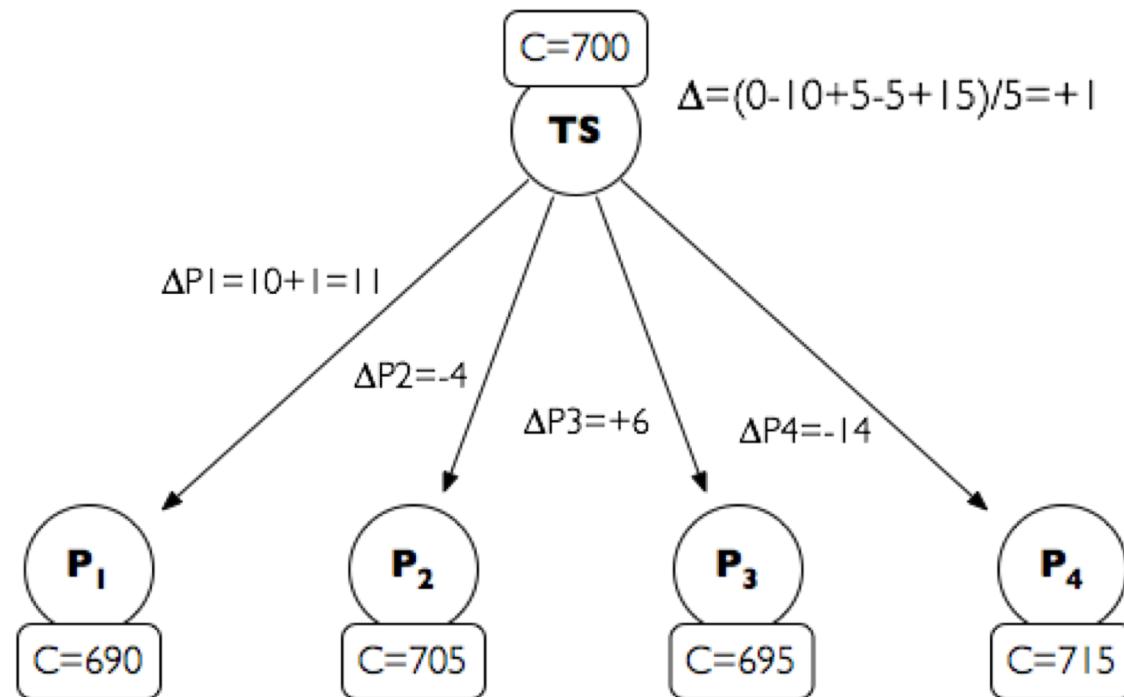
Algoritmo di Berkeley



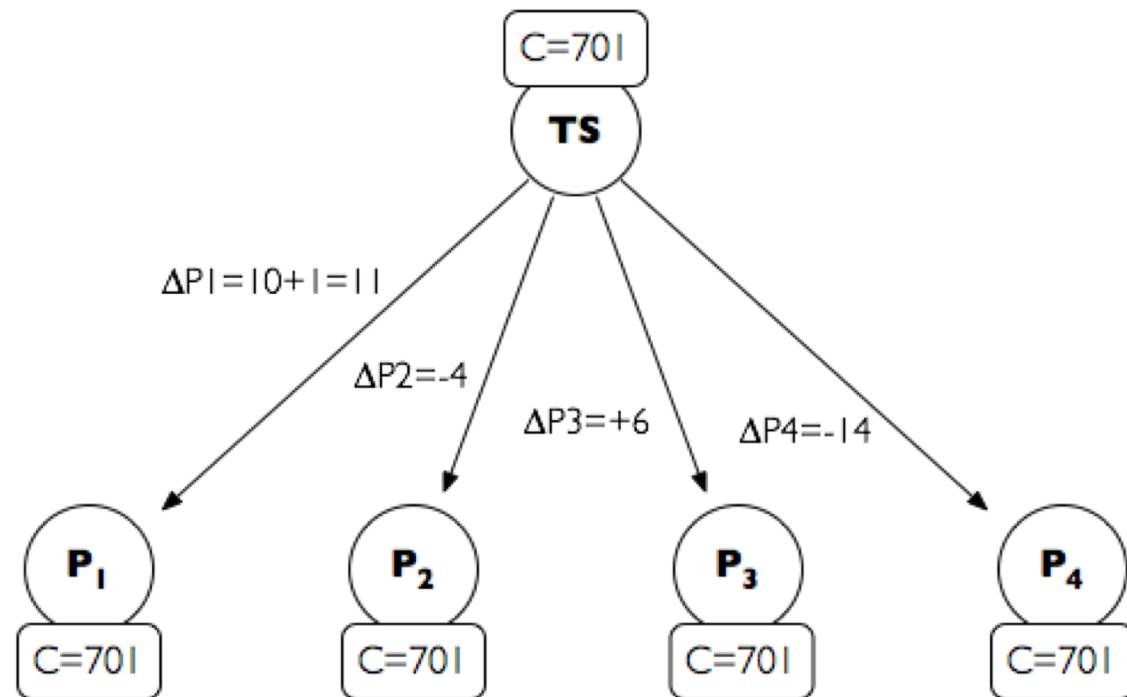
Algoritmo di Berkeley



Algoritmo di Berkeley



Algoritmo di Berkeley



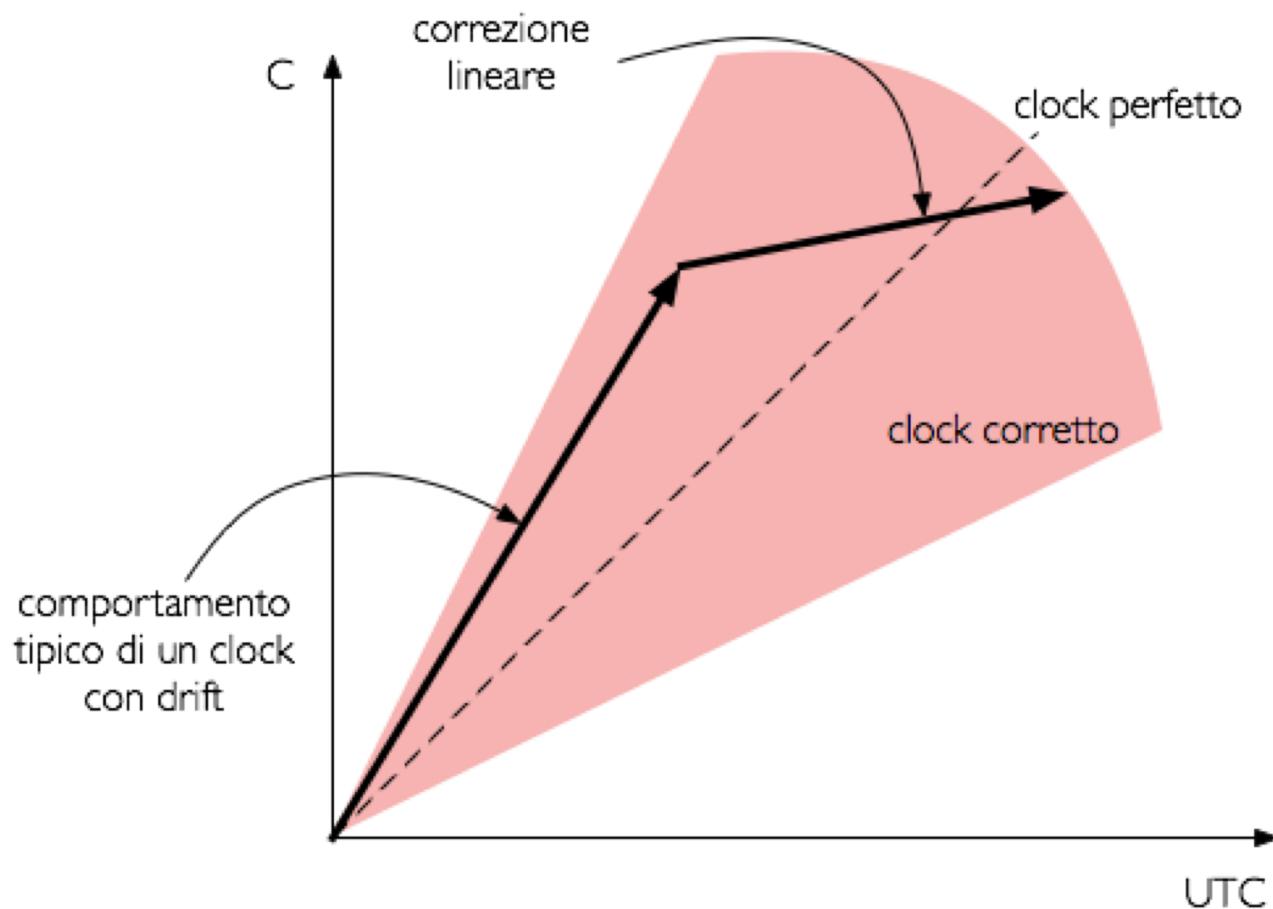
Algoritmo di Berkeley

- L'accuratezza dipende dalla variabilità del RTT dei vari canali che collegano slaves e master.
- Guasti:
 - se un master va in crash un'altra macchina viene eletta master (in un tempo non limitato a priori)
 - È tollerante a comportamenti arbitrari (slaves che inviano valori errati di clock)
 - Il master considera solo un sottoinsieme di valori ricevuti dagli slaves: scarta quelli con valori al di fuori di un intervallo prefissato.

Correzione locale del clock

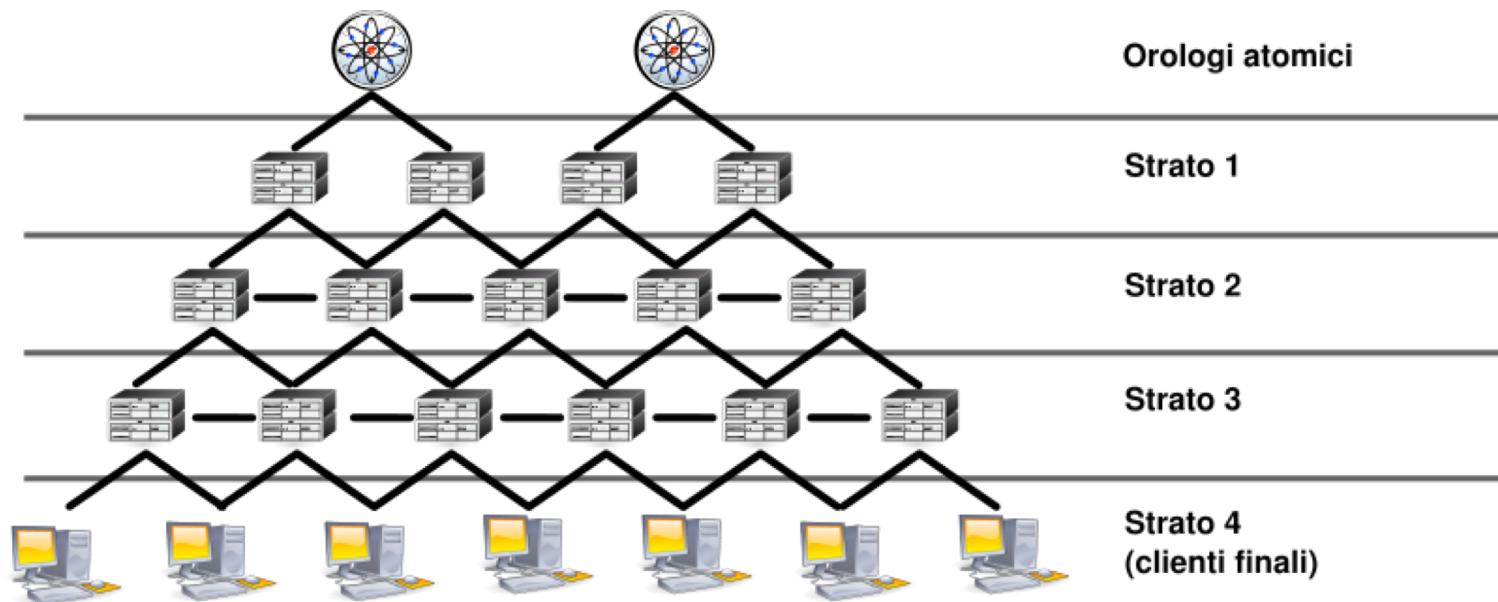
- Non si può pensare di imporre un valore di tempo passato ai processi slave con un valore di clock superiore a quello calcolato come valore di clock sincrono.
 - Ciò provocherebbe un problema di ordinamento causa/effetto di eventi e verrebbe violata la condizione di monotonicità del tempo.
 - La soluzione è quella di mascherare una serie di interrupt che fanno avanzare il clock locale in modo di rallentare l'avanzata del clock stesso.

Correzione locale del clock



Network Time Protocol

- Time service per Internet - sincronizza client a UTC
- Architettura: disponibile e scalabile poiché vengono usati server multipli e path ridondanti. Le sorgenti di tempo sono autenticate.



Network Time Protocol

- La sottorete di sincronizzazione (tutti i livelli che includono i server) si riconfigura in caso di guasti. Es.:
 - Un primary che perde la connessione alla sorgente UTC può diventare un server secondario
 - Un secondario che perde la connessione al suo primary (crash del primary) può usare un altro primary

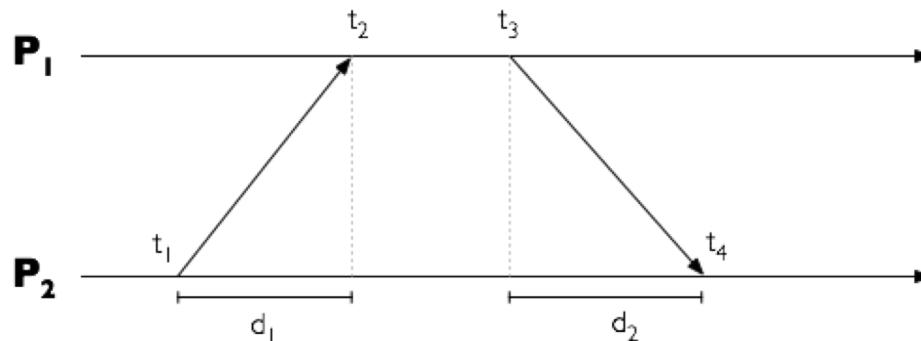
Network Time Protocol

- Modalità di sincronizzazione (sempre via UDP):
 - Multicast: un server manda in multicast il suo tempo agli altri che impostano il tempo ricevuto assumendo un certo ritardo prefissato (non molto accurato in mancanza di multicast hardware).
 - Procedure call: un server accetta richieste da altri computer (come algoritmo di Cristian). Alta accuratezza. Utile se non è disponibile multicast hw.
 - Simmetrico: coppie di server scambiano messaggi contenenti informazioni sul timing. Usata quando è necessaria un'accuratezza molto alta (quindi per gli alti livelli della gerarchia).

Approfondimento: <http://www.sematech.org/docubase/document/4736aeng.pdf>

Network Time Protocol

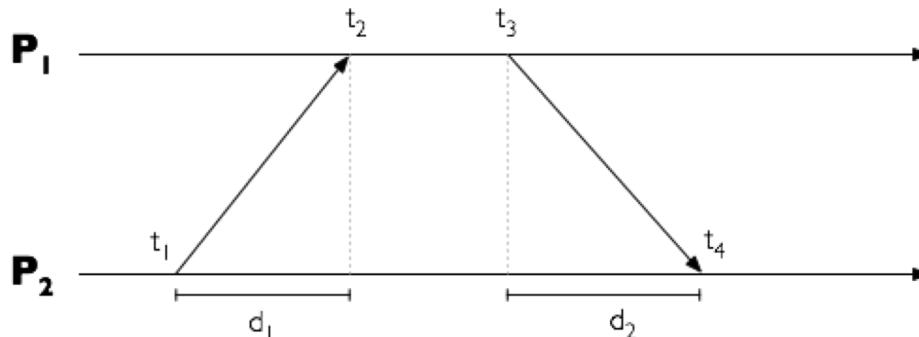
- Con il metodo simmetrico bisogna tenere in considerazione i ritardi dei canali.



- Per ogni coppia di messaggi scambiati tra i due server, NTP stima un offset O tra i 2 clock ed un ritardo D (tempo di trasmissione totale per i 2 msg).
- Immaginiamo che o sia il vero offset tra i due clock
- $t_2 = t_1 + d_1 + o$ $t_4 = t_3 + d_2 - o$
- $D = d_1 + d_2 = (t_2 - t_1) + (t_4 - t_3)$

Network Time Protocol

- Con il metodo simmetrico bisogna tenere in considerazione i ritardi dei canali.



- Sottraendo le equazioni:
$$t_2 - t_4 = (t_1 + d_1 + o) - (t_3 + d_2 - o) = t_1 + d_1 - t_3 - d_2 + 2o$$
$$\Rightarrow o = ((t_2 - t_1) + (t_3 - t_4)) / 2 + (d_2 - d_1) / 2$$
$$[O = ((t_2 - t_1) + (t_3 - t_4)) / 2] \Rightarrow o = O + (d_2 - d_1) / 2$$
- L'offset stimato O ha quindi un errore massimo pari a $\pm (d_2 - d_1) / 2$
- accuracy di 10msecs su Internet, 1msec su LAN.

Physical vs logical clock

- Algoritmi di sincronizzazione del clock fisico tentano di coordinare clock distribuiti per ottenere un valore comune
 - Basati sulla stima dei tempi di trasmissione
 - Ottenere una buona stima può essere difficile
 - Tuttavia, in diverse applicazioni non è importante quando gli eventi si sono verificati, ma piuttosto in che ordine sono avvenuti...

Tempo logico

- Claim: in molte applicazioni non è importante quando gli eventi accadono, ma in che ordine accadono
 - **Un ordinamento affidabile è necessario!**
- Nel caso in cui sincronizzare i clock non è possibile possiamo ordinare i messaggi sulla base di due ovvie assunzioni:
 - due eventi che accadono su uno stesso processo possono sempre essere ordinati
 - un evento di ricezione di un messaggio segue sempre l'evento di invio del messaggio stesso
- Gli eventi possono quindi essere ordinati secondo una nozione di causa-effetto (relazione happened before o ordinamento causale - Lamport)

Some notation

Lamport introduced the relation that captures the causal dependencies between events (*causal order relation*)

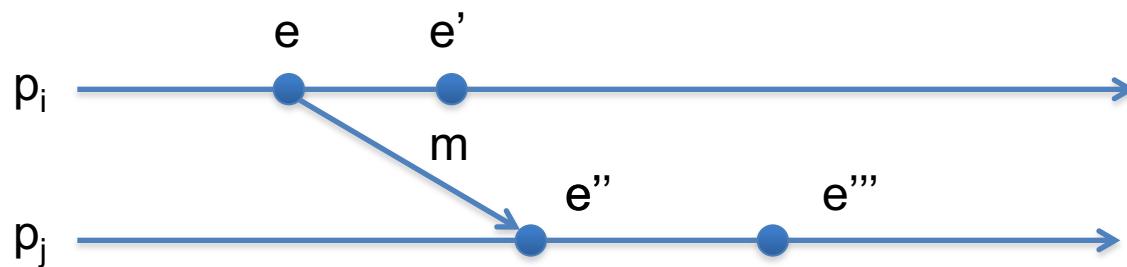
- We denote with \rightarrow_i the ordering relation between events in a process p_i
- We denote with \rightarrow the happened-before relation between any pair of events

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

Happened-Before Relation: Definition

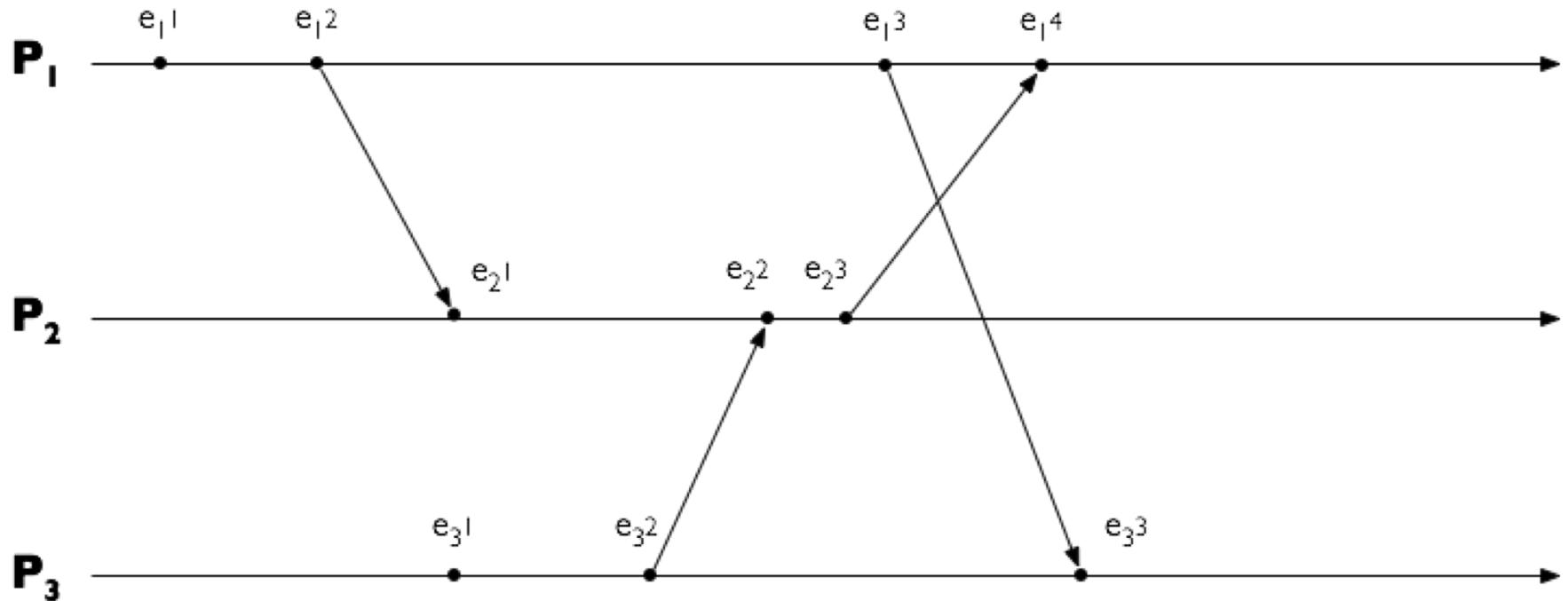
- Two events e and e' are related by happened-before relation ($e \rightarrow e'$) if:
 - $\exists p_i \mid e \rightarrow_i e'$
 - \forall message $m \ e_{\text{send}(m)} \rightarrow e_{\text{receive}(m)}$
 - $\text{send}(m)$ is the event of sending a message m
 - $\text{receive}(m)$ is the event of 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

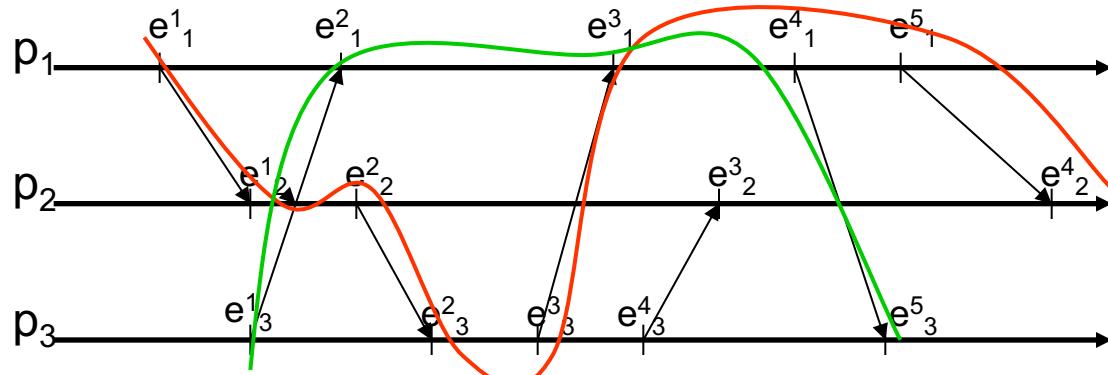
- Using these three rules it is possible to define a causal-ordered sequence of events e_1, e_2, \dots, e_n
- **Notes:**
 - The sequence e_1, e_2, \dots, e_n may not be unique
 - It may exist a pair of events $\langle e_1, e_2 \rangle$ such that e_1 and e_2 are not in happened-before relation
 - If e_1 and e_2 are not in happened-before relation then they are *concurrent* ($e_1 \parallel e_2$)
 - For any two events e_1 and e_2 in a distributed system, one of the following must hold:
 - $e_1 \rightarrow e_2$
 - $e_2 \rightarrow e_1$
 - $e_1 \parallel e_2$

Example



e_i^j is j-th event of process p_i

happened-before



e^j_i is j-th event of process p_i

$$S_1 = \langle e^1_1, e^1_2, e^2_1, e^2_3, e^3_3, e^3_1, e^4_1, e^5_1, e^4_2 \rangle$$

$$S_2 = \langle e^1_3, e^2_1, e^3_1, e^4_1, e^5_3 \rangle$$

Note: e¹₃ and e¹₂ are concurrent

Logical Clock

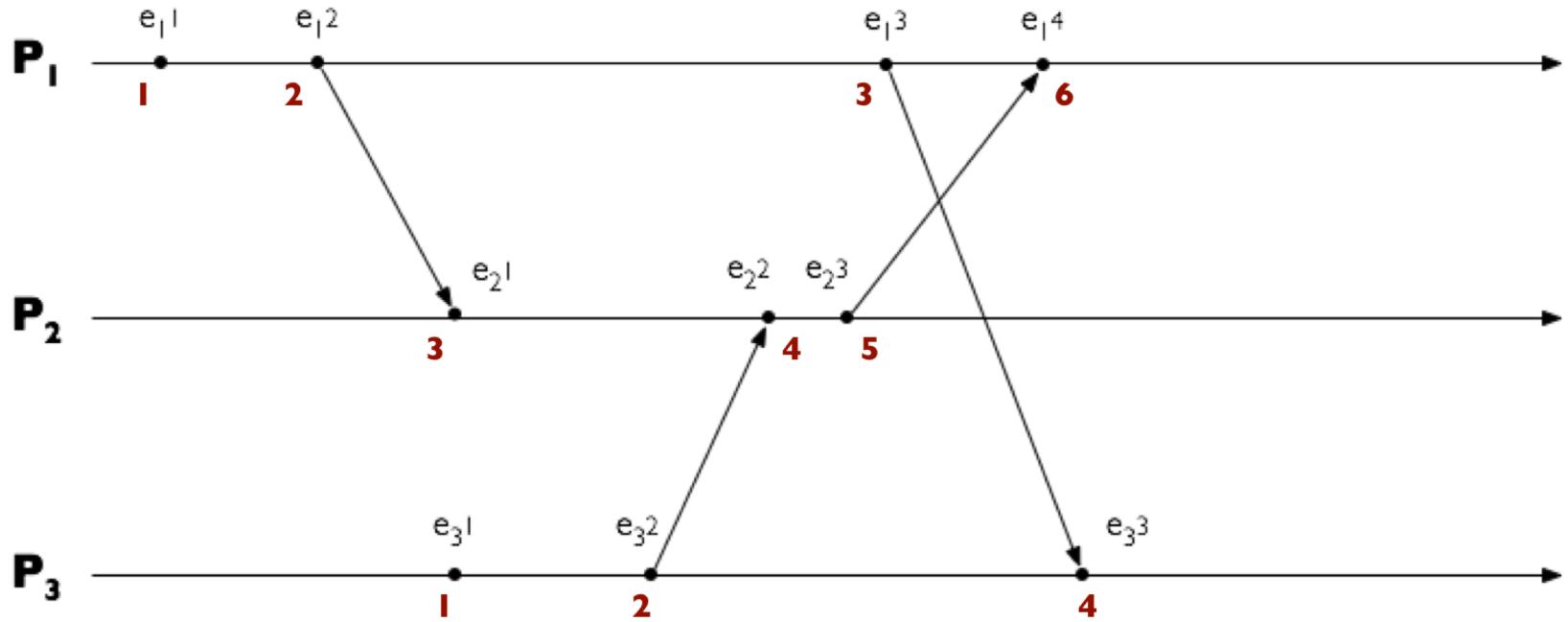
- The Logical Clock, introduced by Lamport, is a software counting register *monotonically* increasing its value
 - Logical clock is not related to physical clock
- Each process p_i employs its logical clock L_i to apply a *timestamp* to events
- $L_i(e)$ is the “logical” timestamp assigned, using the logical clock, by a process p_i to event 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 $L_i=0$ ($\forall i = 1 \dots N$)
- p_i increases L_i by 1 when it generates an event (*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

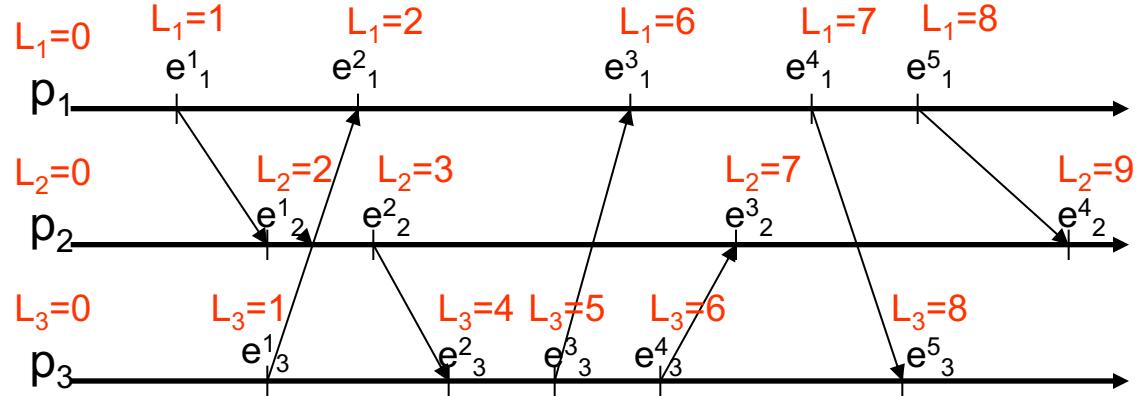
See also: https://en.wikipedia.org/wiki/Lamport_timestamps

Scalar Logical Clock: example



- $e_{1,1}^1 \rightarrow e_{1,2}^1 \rightarrow e_{2,1}^1$ (and $1 < 2 < 3$)
- $e_{1,1}^1 \rightarrow e_{1,2}^1 \rightarrow e_{1,3}^1 \rightarrow e_{3,3}^3$ (and $1 < 2 < 3 < 4$)
- What can we tell about $e_{3,3}^3$ and $e_{1,4}^1$?

Another example



- $e_{i,j}^j$ is j-th event of process p_i
- L_i is the logical clock of p_i
- Note:
 - $e_{1,1}^1 \rightarrow e_{2,1}^1$ and timestamps reflect this property
 - $e_{1,1}^1 \parallel e_{1,3}^3$ and respective timestamps have the same value
 - $e_{1,2}^2 \parallel e_{1,3}^3$ 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, by 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**

Clock vettoriale

- Ad ogni evento e viene assegnato un vettore $V(e)$ di dimensione pari al numero dei processi con la seguente proprietà:

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

- Che significato diamo all'operatore $<$ applicato a due vettori ?

$V(e') > V(e)$ se e solo se:

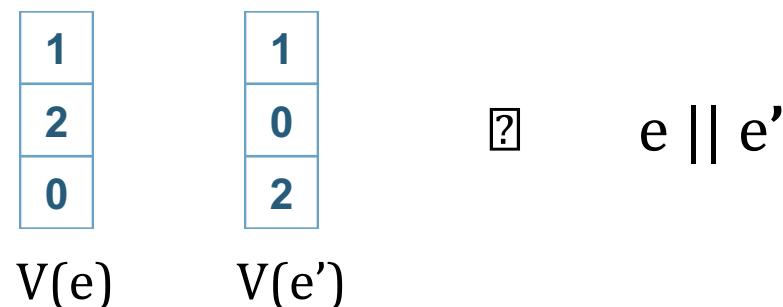
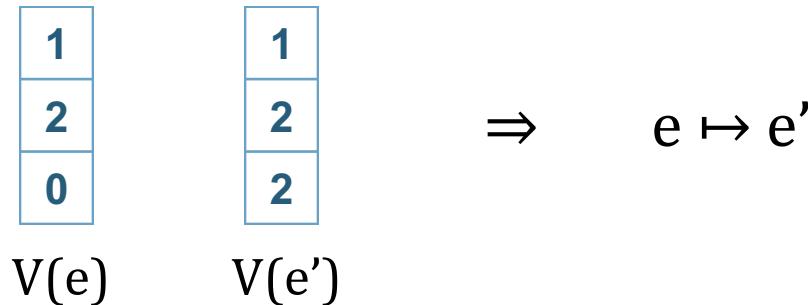
$$\forall x \in [1, \dots, n] \quad V(e')[x] \geq V(e)[x]$$

e

$$\exists x \in [1, \dots, n] \quad V(e')[x] > V(e)[x]$$

Clock vettoriale

- Comparare i valori di due clock vettoriali associati a due eventi distinti permette di capire la relazione che lega i due eventi (se uno precede l'altro o se sono concorrenti)



Clock vettoriale

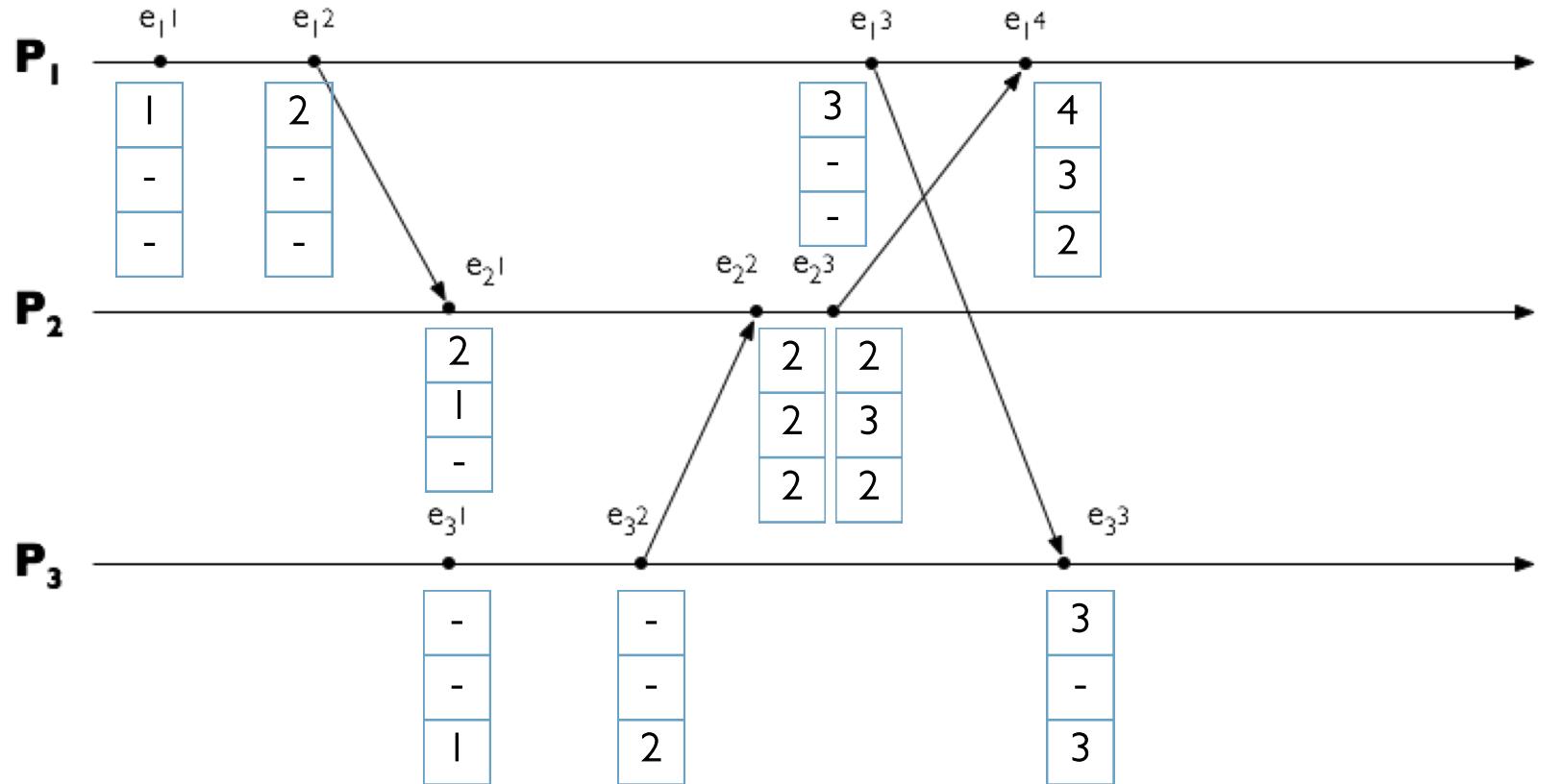
- Ogni processo P_i gestisce un vettore di interi V_i ad n componenti $V_i[1...n]$ ovvero una per ogni processo.
 - La componente $V_i[x]$ indica la stima che il processo P_i fa sul numero di eventi eseguiti dal processo P_x
 - Il vettore è inizializzato a $[-,-,...,0,...,-]$
 - $V_i[i]$ rappresenta il logical clock di P_i

Clock vettoriale

- Il vettore viene aggiornato in base alle seguenti regole:
 - quando P_i esegue un evento incrementa $V_i[i]$ di una unità e poi associa un timestamp T all'evento il cui valore è pari al valore corrente di V_i ;
 - quando P_i esegue un evento di invio messaggio, allega al messaggio il timestamp di quell'evento ottenuto dalla regola precedente;
 - quando arriva un messaggio a P_i con un timestamp T , P_i esegue la seguente operazione:
$$\forall x \in [1 \dots n] : V_i[x] = \max(V_i[x], T[x])$$
 - quindi esegue l'evento di consegna (esegue la prima regola incrementando il proprio logical clock).

Clock vettoriale

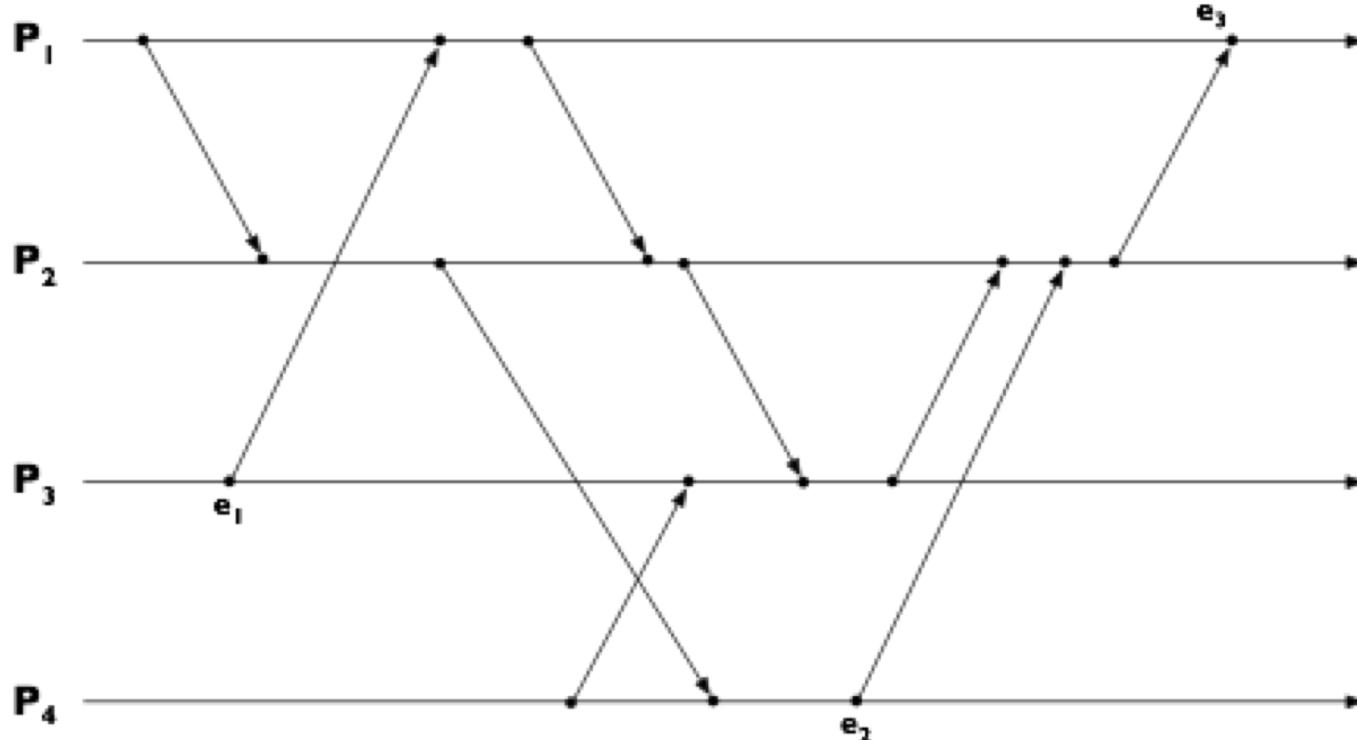
- Torniamo al nostro primo esempio:



- $[-, -, 1] < [4, 3, 2] \Rightarrow e_{3,1} \mapsto e_{1,4}$
- $[4, 3, 2] ? [3, -, 3] \Rightarrow e_{1,4} || e_{3,3}$

Esercizio

Domanda 1 Si consideri la seguente esecuzione globale:



Lo studente applichi a ciascun evento (evidenziato in figura con un puntino nero) il corrispondente vector clock e successivamente discuta quale delle seguenti affermazioni è falsa e perché:

1. $e_1 \rightarrow e_2$
2. $e_1 \rightarrow e_3$
3. $e_2 \rightarrow e_3$

La discussione deve fare riferimento ai valori contenuti nei vector clock

Ricart-Agrawala algorithm

(mutual exclusion in distributed setting)

- Scalar clocks may suggest a solution to the mutual exclusion problem in a distributed scenario
 - simply adapting Lamport's algorithm might yield an inefficient solution, as processes act independently (no coordination) in trying to access the critical section
- Intuition behind RA algorithm
 - each process enters the doorway suggesting a number
 - the process then sends the number to all the other processes, waiting for them to grant access to the CS

RA algorithm

Local variables

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

Assumptions

- processes don't fail
- messages are never lost
- finite channel latencies (value is unknown)

RA algorithm

repeat

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

Note: line 2 is executed atomically

RA algorithm

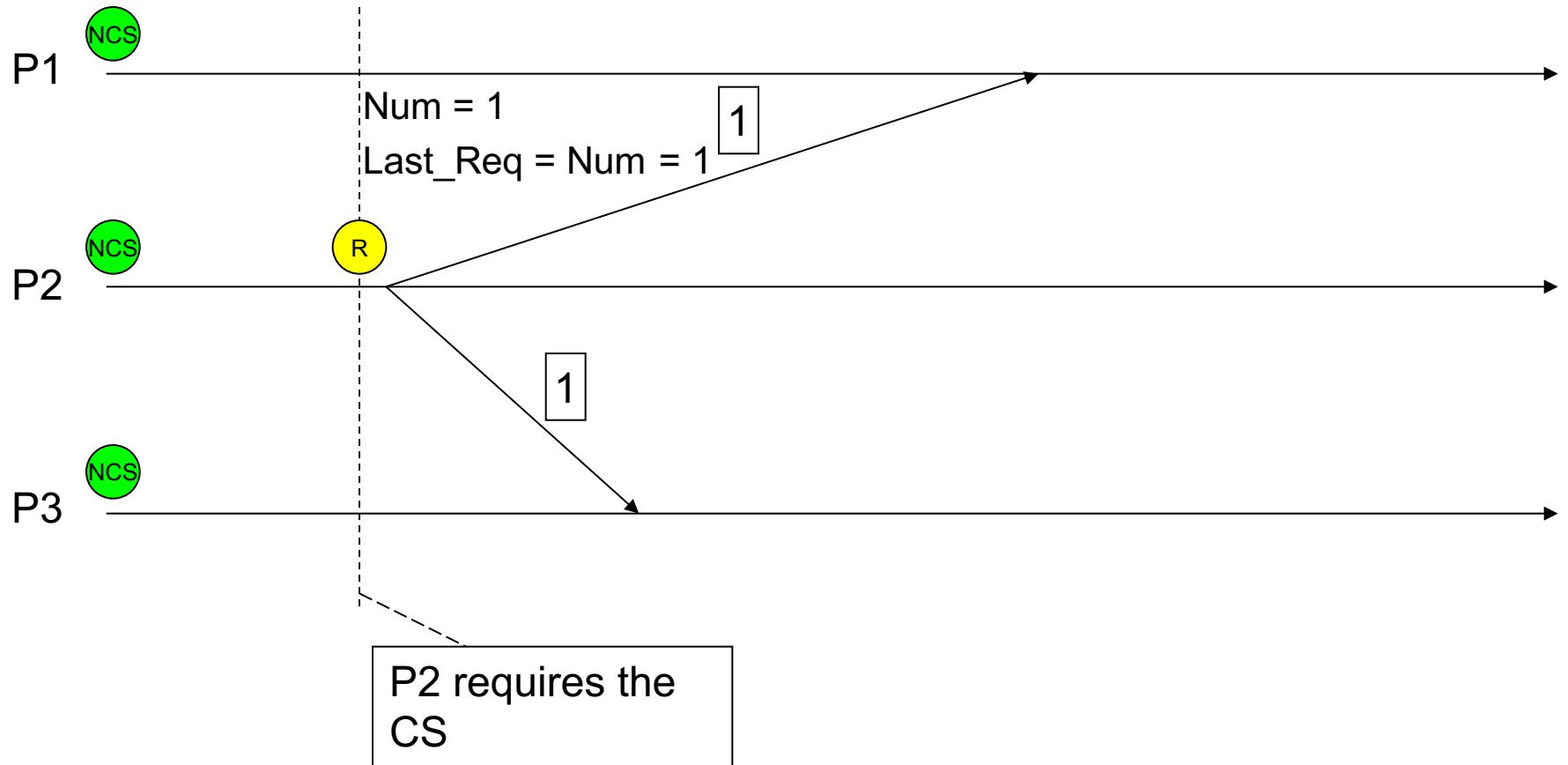
Upon receipt of REQUEST(t) from p_j

8. $\text{Num} = \max(t, \text{Num})$
9. If State == CS or (State == Requesting and $\{\text{Last_Req}, i\} < \{t, j\}$)
10. Then insert $\{t, j\}$ into Q
11. Else send REPLY to p_j

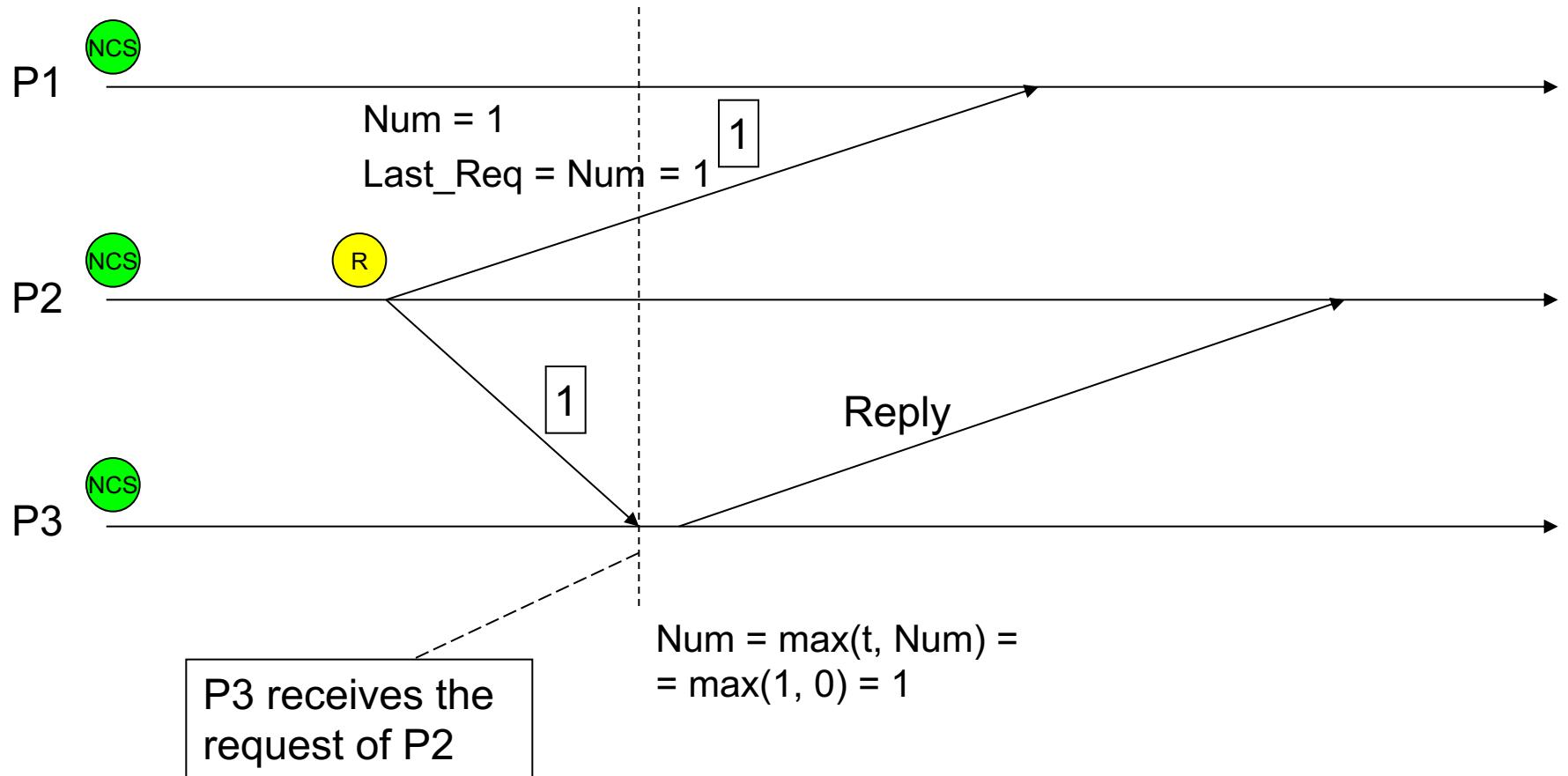
Upon receipt of REPLY from p_j

12. $\#\text{replies} = \#\text{replies} + 1$

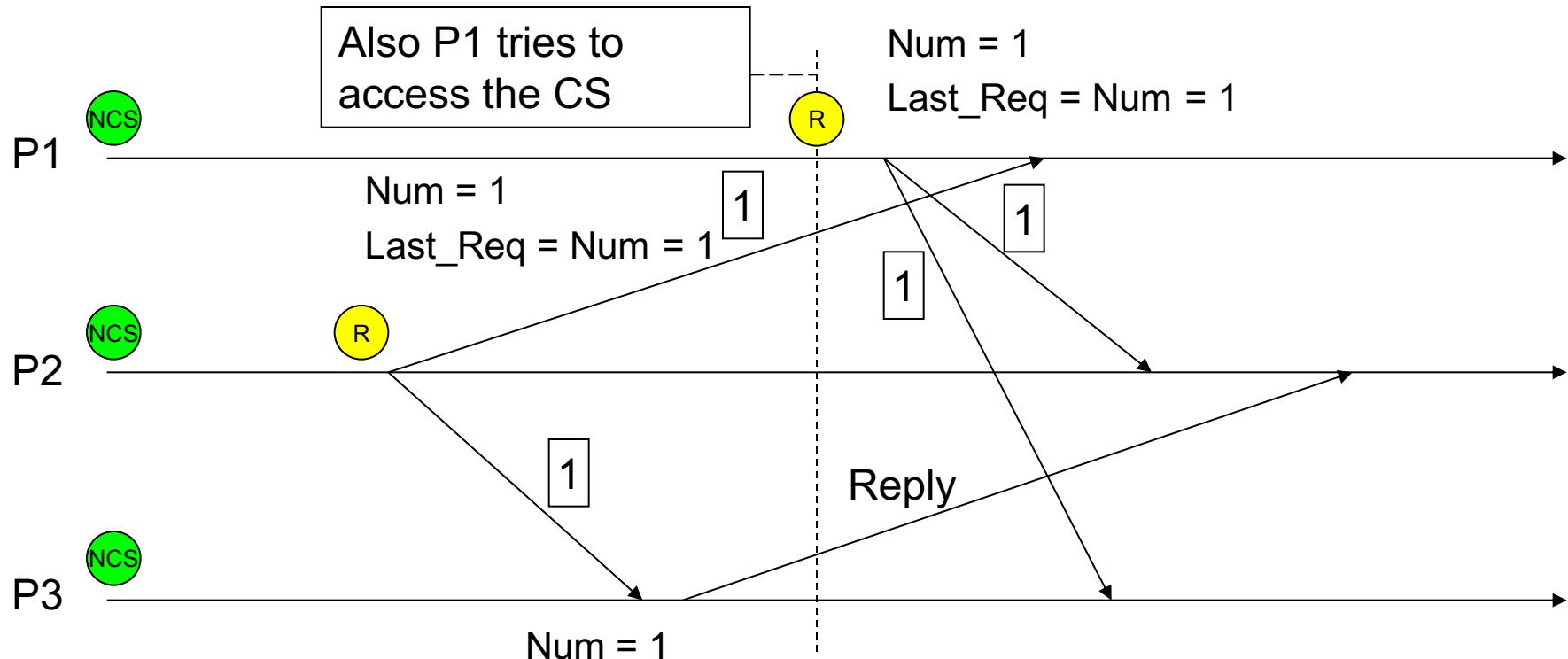
Ricart-Agrawala's algorithm: example



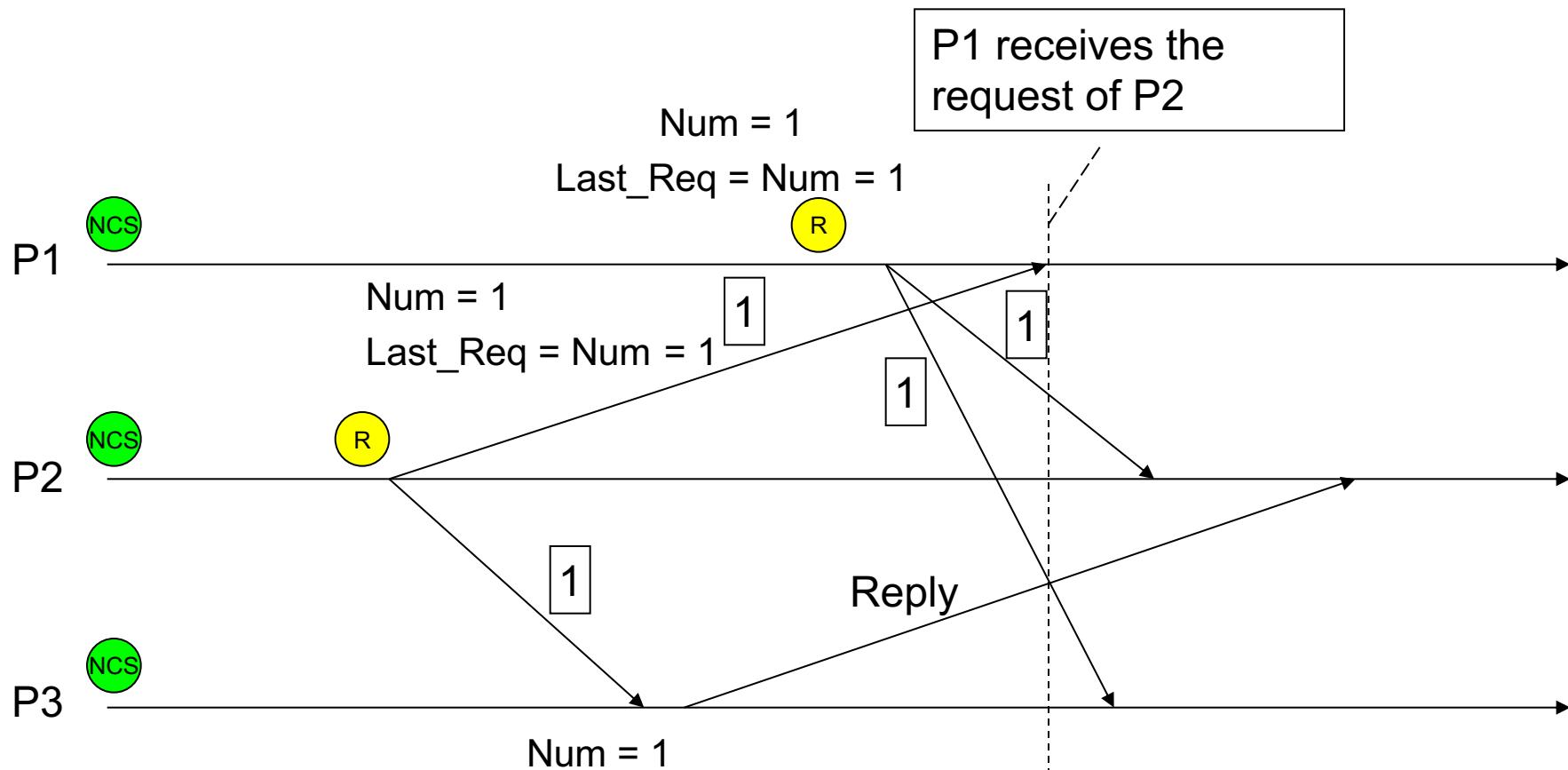
Ricart-Agrawala's algorithm: example



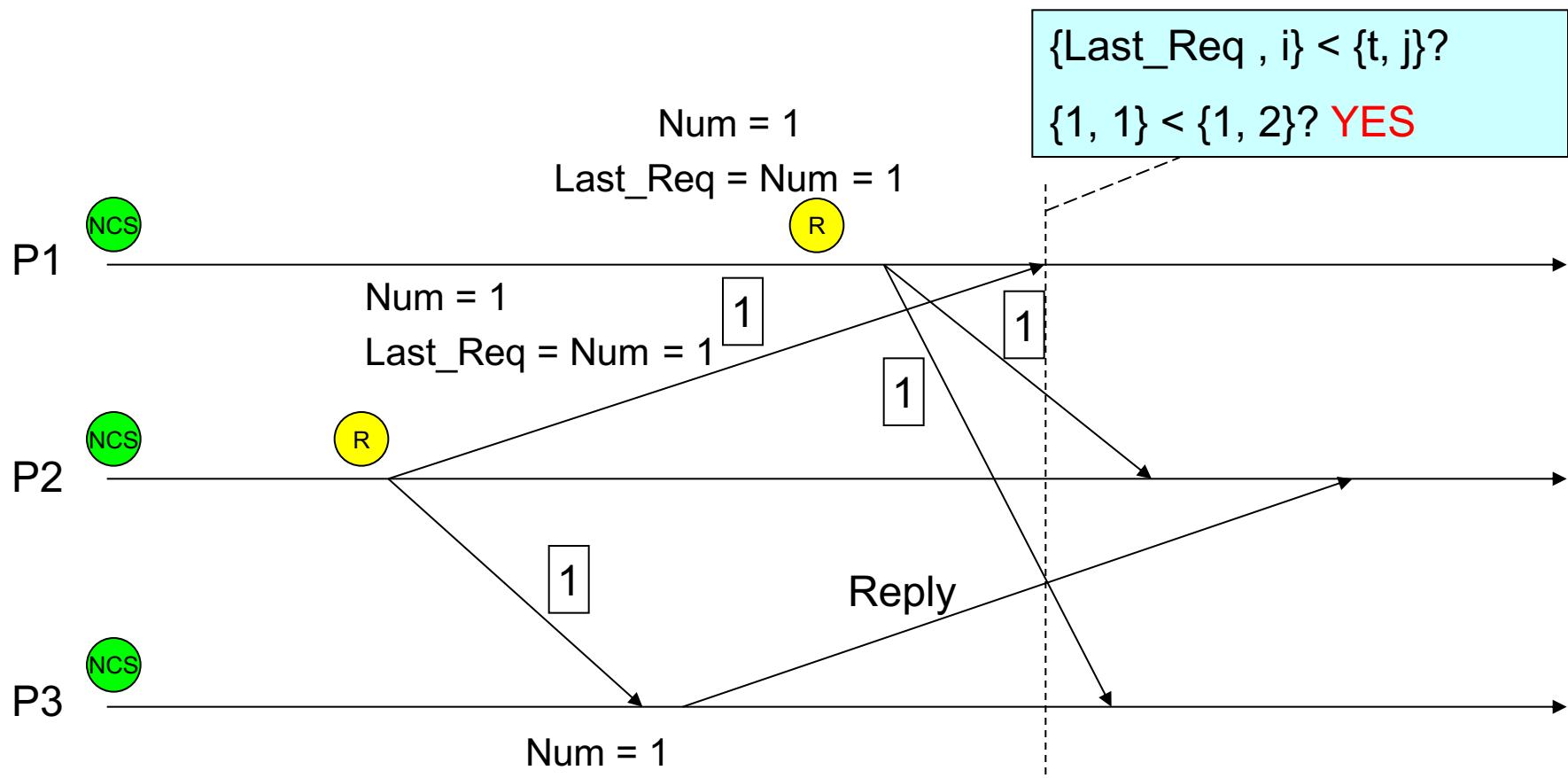
Ricart-Agrawala's algorithm: example



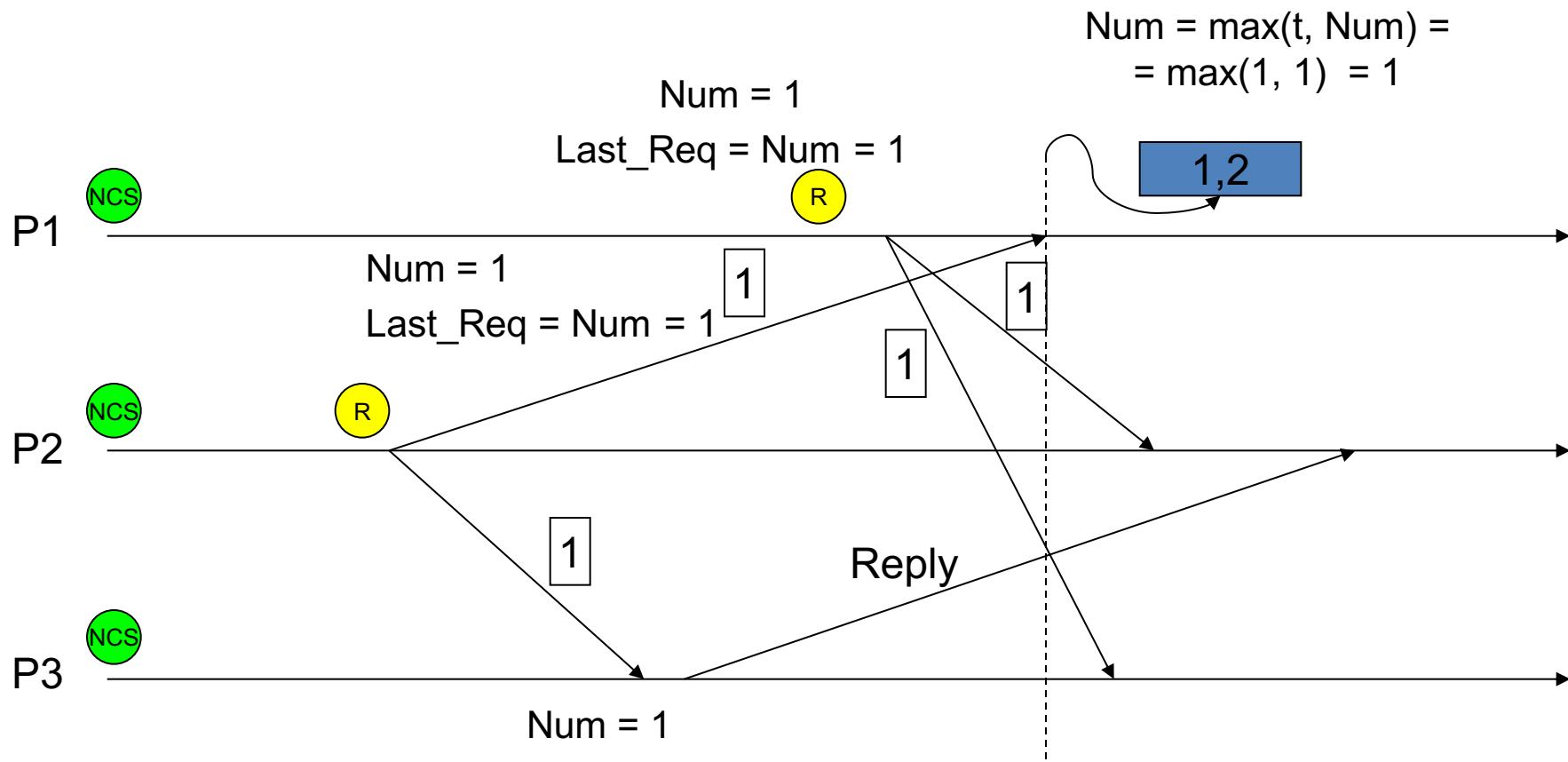
Ricart-Agrawala's algorithm: example



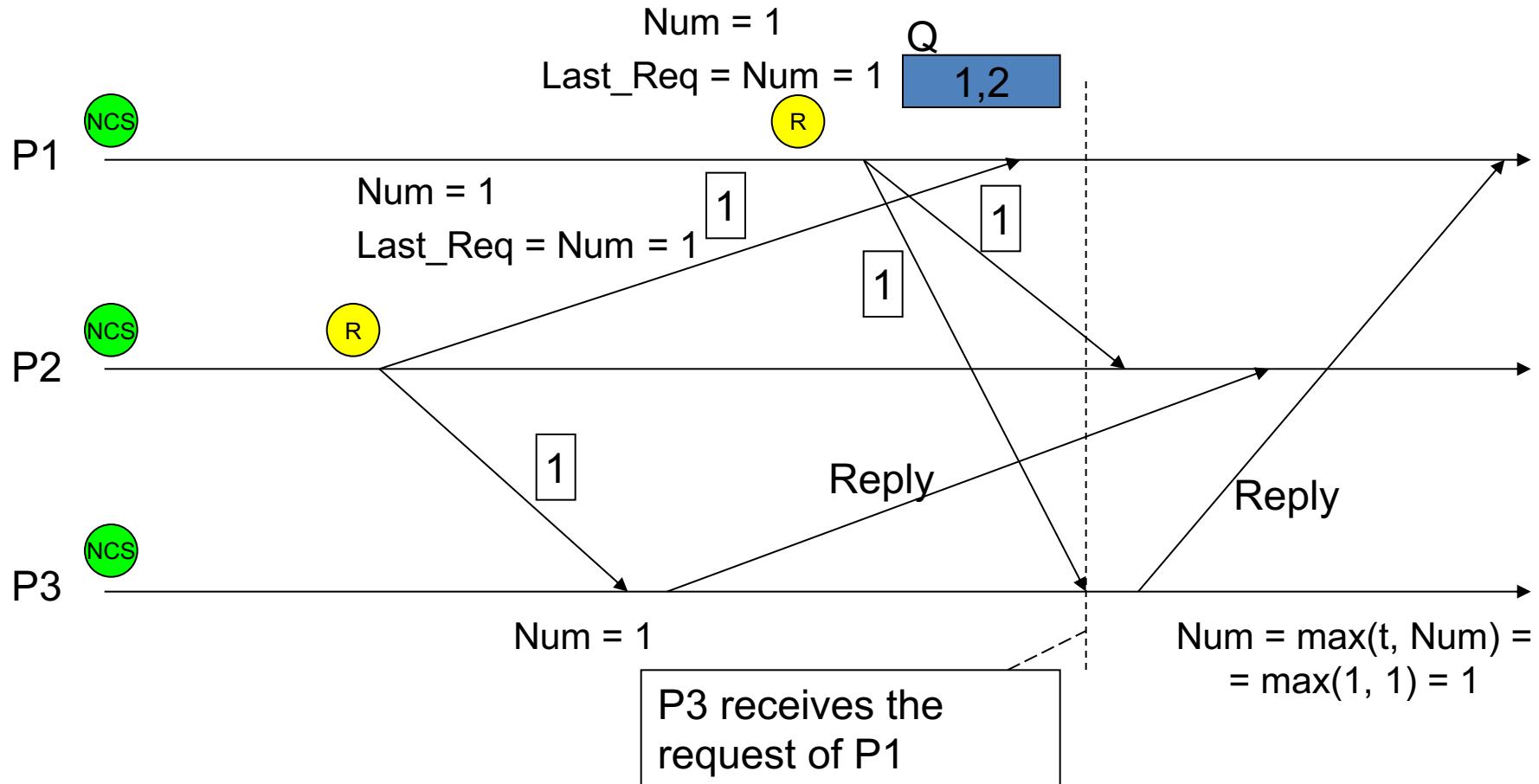
Ricart-Agrawala's algorithm: example



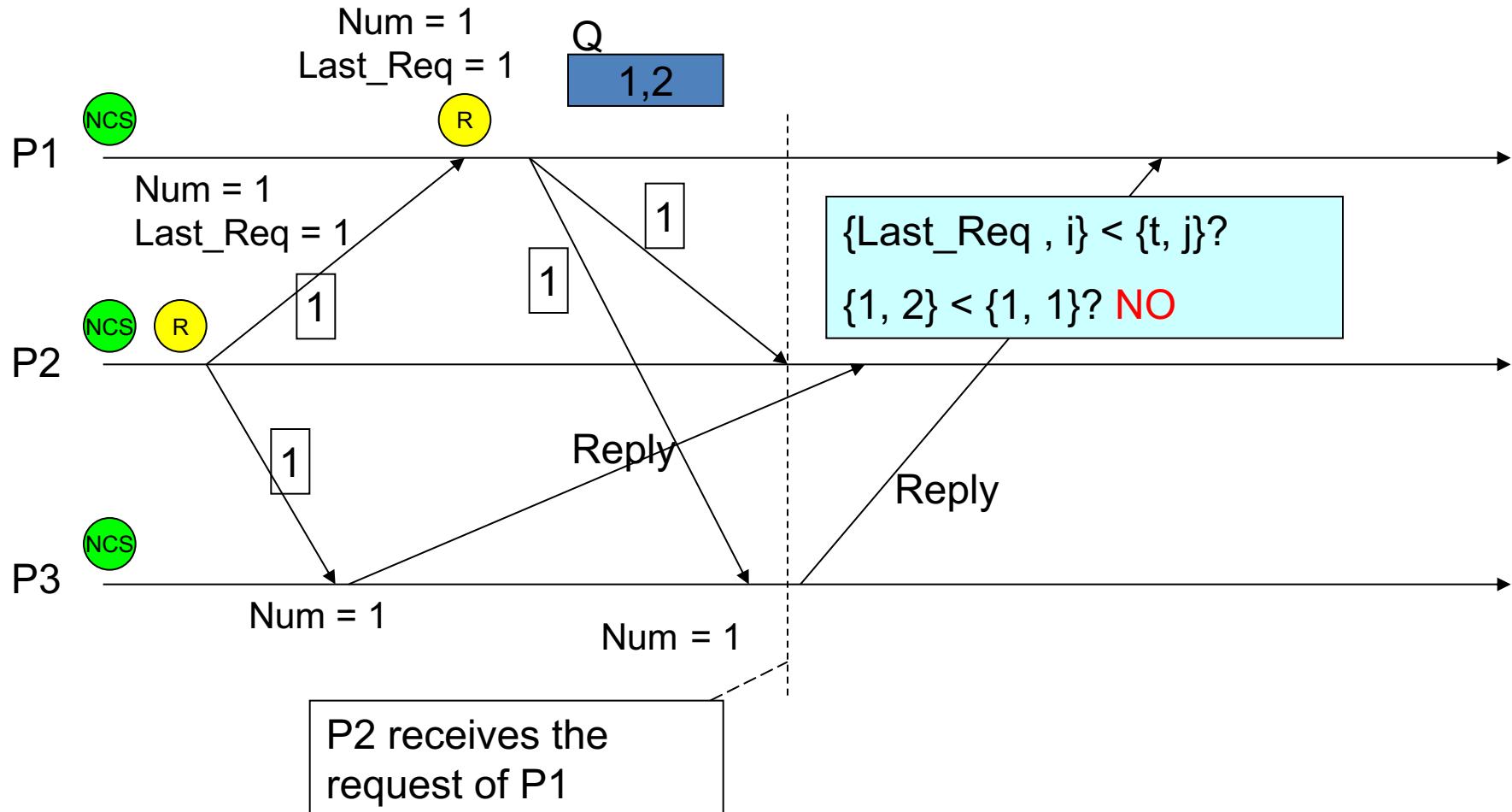
Ricart-Agrawala's algorithm: example



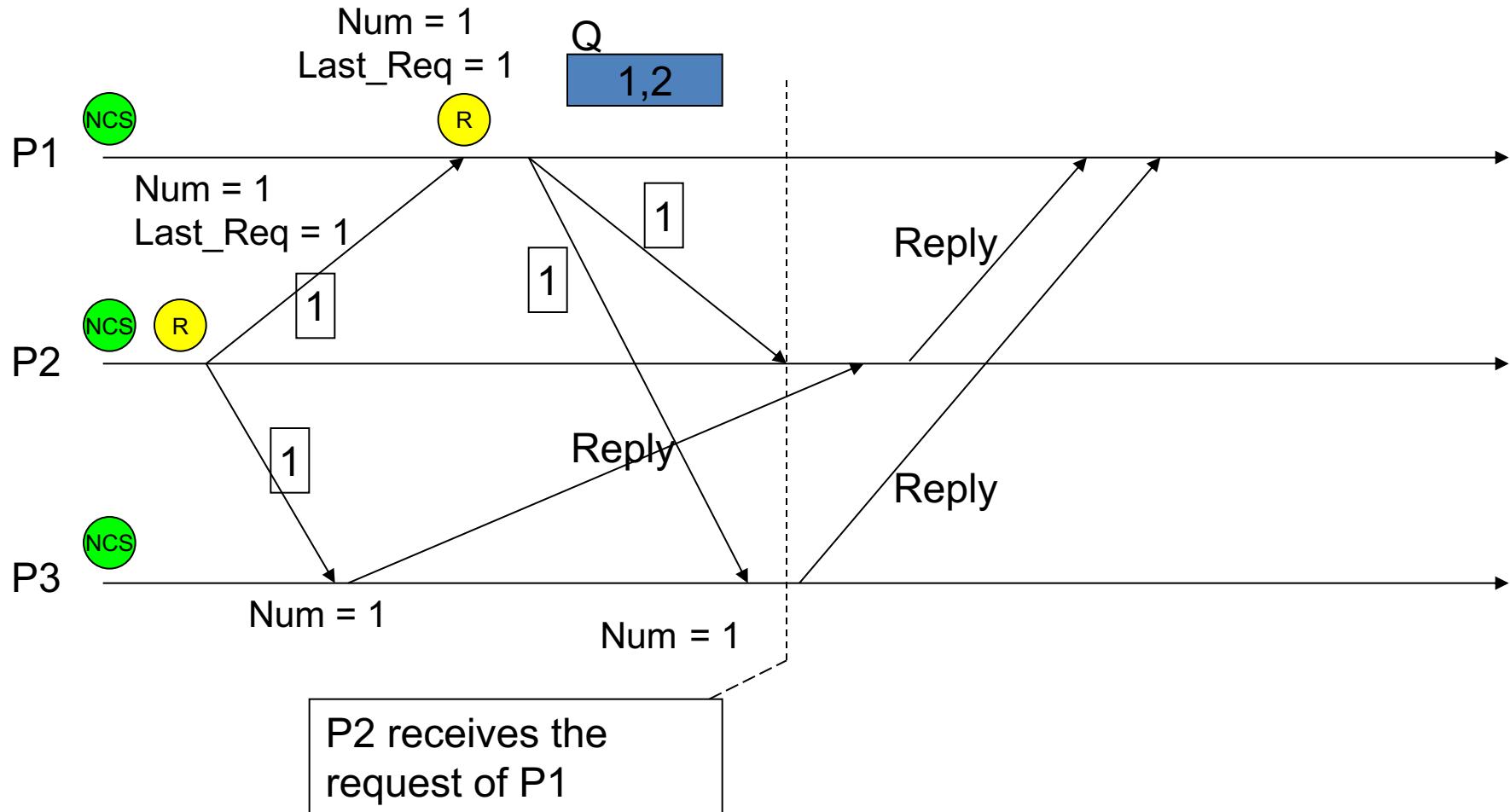
Ricart-Agrawala's algorithm: example



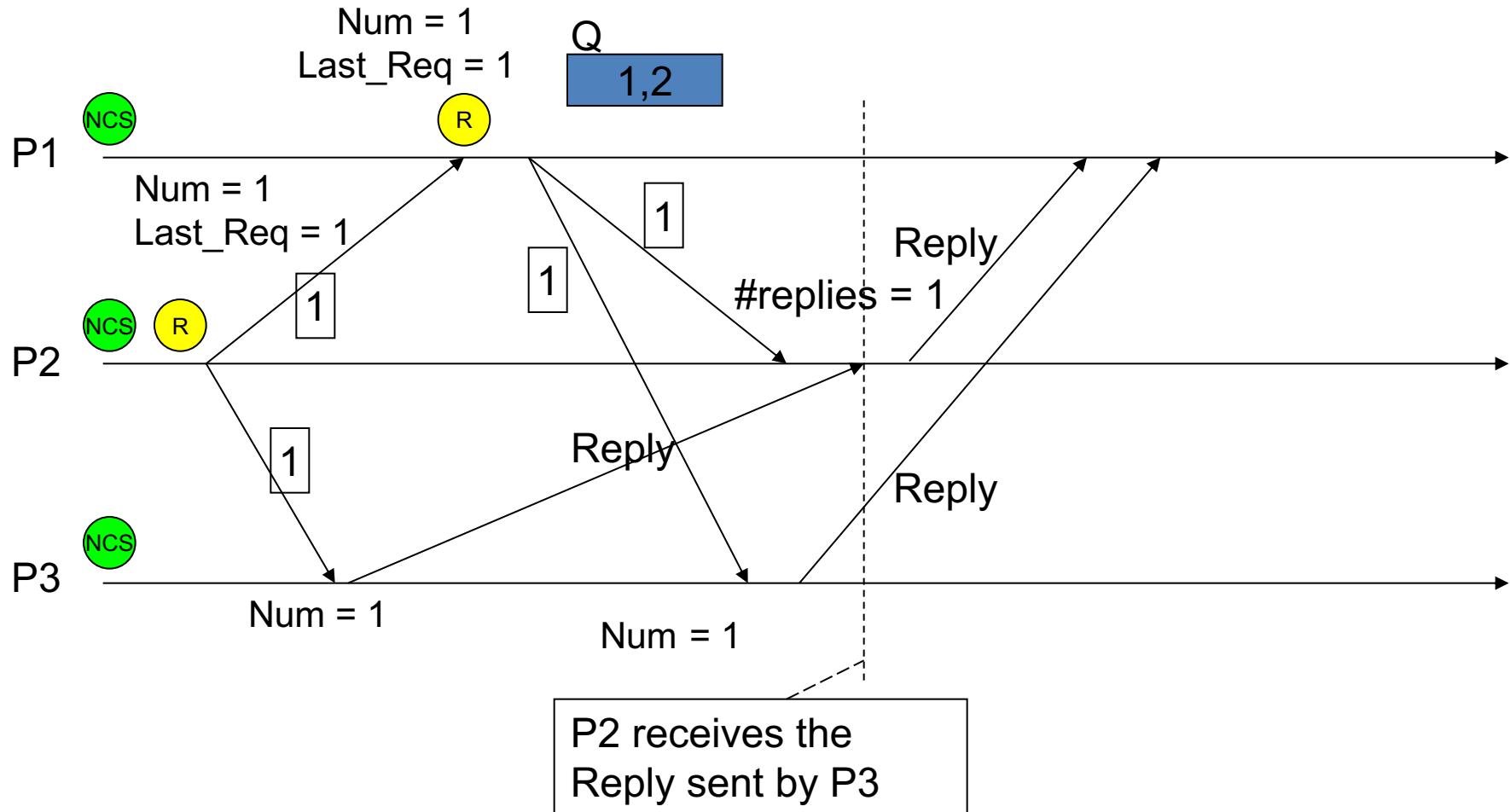
Ricart-Agrawala's algorithm: example



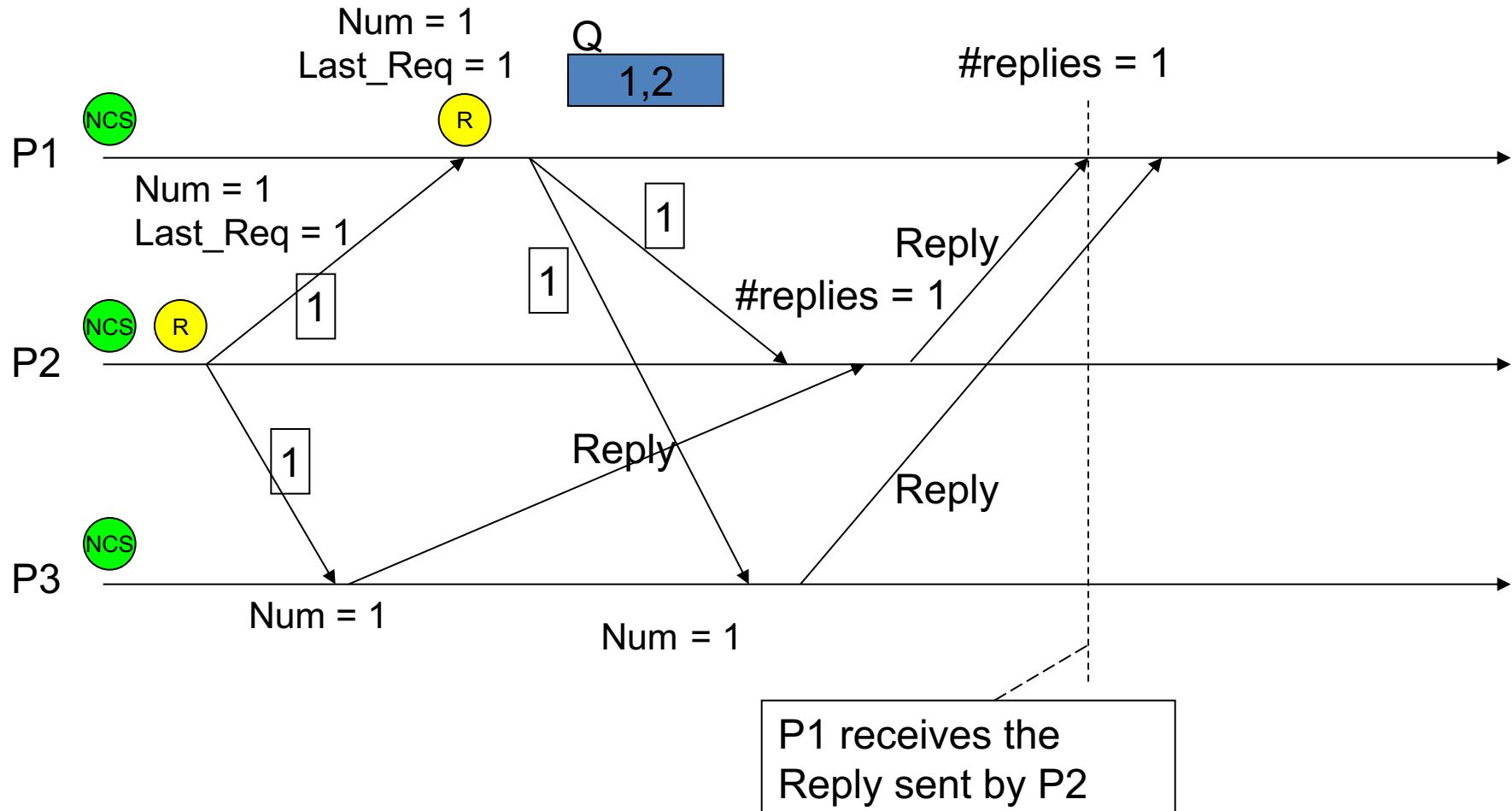
Ricart-Agrawala's algorithm: example



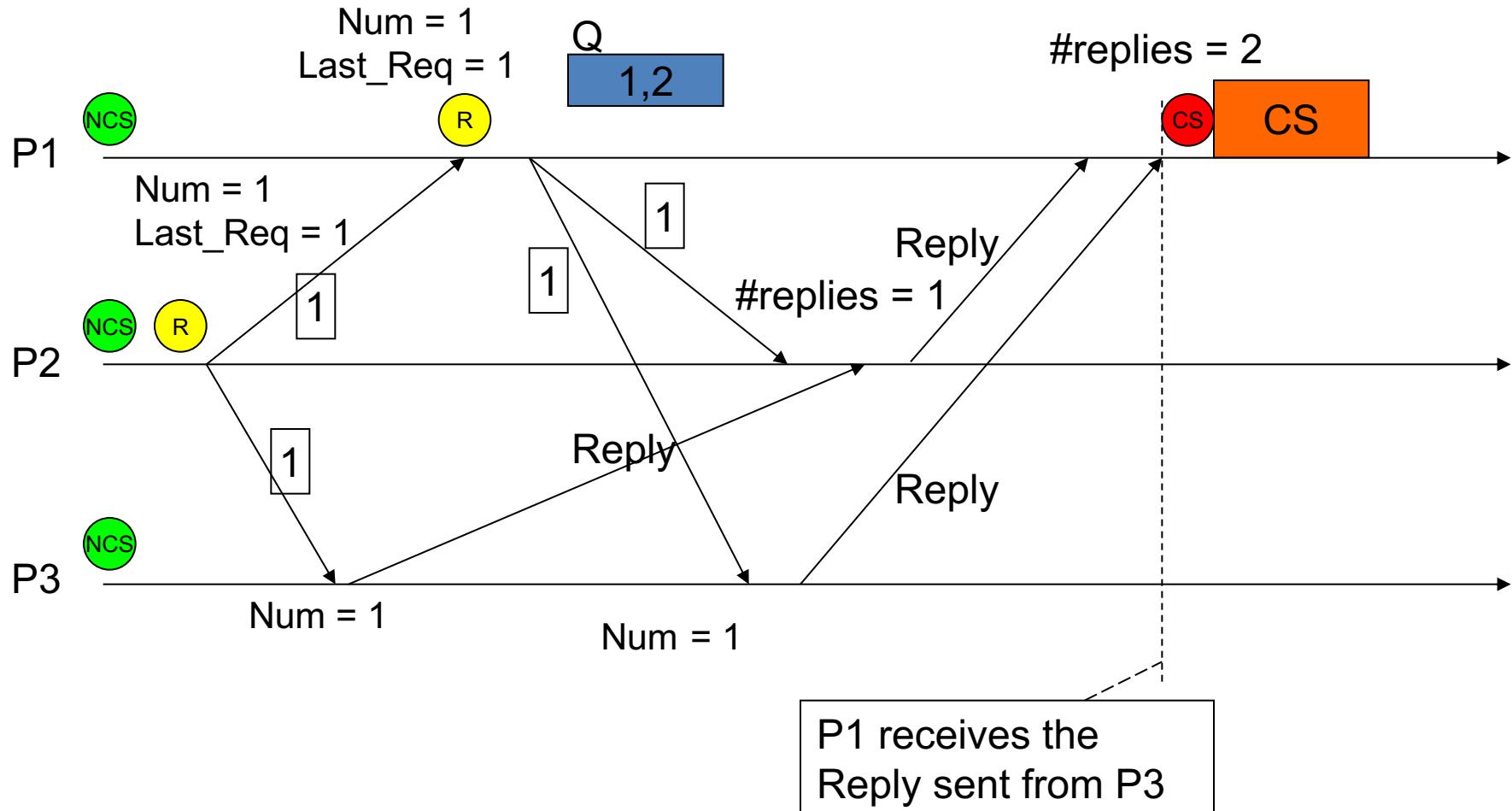
Ricart-Agrawala's algorithm: example



Ricart-Agrawala's algorithm: example



Ricart-Agrawala's algorithm: example



Ricart-Agrawala's algorithm: example

