



19/12/2022

## Sincronizzazione e Coordinazione nei Sistemi Distribuiti

**Corso di Sistemi Distribuiti e Cloud Computing**  
A.A. 2021/22

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

### Il tempo nei SD

---

- In un SD, i processi
  - vengono eseguiti su nodi connessi in rete
  - cooperano per portare a termine una computazione
  - comunicano tramite scambio di messaggi
- Osservazioni:
  - Molti algoritmi distribuiti richiedono **sincronizzazione**
    - Ovvero i processi in esecuzione su diversi nodi del SD devono avere una **nozione comune di tempo** per poter effettuare azioni sincronizzate rispetto al tempo
  - Molti algoritmi richiedono che gli **eventi** siano **ordinati**
    - Nei SD i messaggi arrivano con dei **timestamp** in modo che si possa sapere in che ordine devono essere eseguiti
- **Conseguenza:** nei SD **il tempo è un fattore critico**

# Il tempo nei SD

---

- In un **sistema centralizzato** è possibile stabilire l'ordine in cui gli eventi si sono verificati
  - Memoria comune e clock unico
- In un **sistema distribuito** è impossibile avere un unico clock fisico comune a tutti i processi
  - Eppure la computazione globale può essere vista come un ordine totale di eventi, se si considera il tempo in cui gli eventi sono stati generati
- Per numerosi problemi nei SD è di vitale importanza risalire a questo tempo, o comunque è importante stabilire l'**ordinamento degli eventi**, ovvero quale evento è stato generato prima di un altro
  - **Come?**

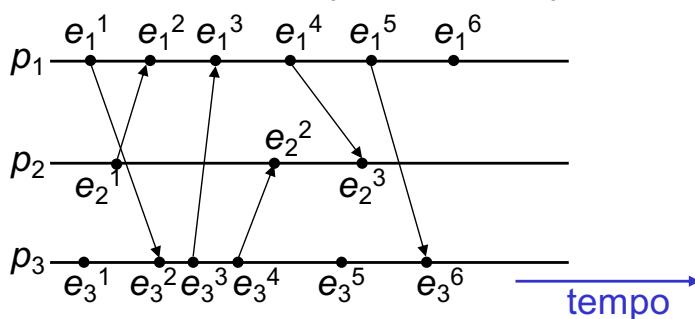
## Il tempo nei SD: soluzioni

---

- Soluzione 1: **sincronizzazione degli orologi fisici**
  - Il middleware di ogni nodo del SD aggiusta il valore del suo clock fisico in modo coerente con quello degli altri nodi o con quello di un clock di riferimento
- Soluzione 2: **sincronizzazione degli orologi logici**
  - Leslie Lamport ha dimostrato come in un SD non sia necessaria la sincronizzazione degli orologi fisici, ma lo sia solo l'ordinamento degli eventi

# Modello della computazione distribuita

- Componenti del SD:  $N$  processi e canali di comunicazione
- Ogni processo  $p_i$  ( $1 \leq i \leq N$ ) genera una sequenza di eventi
  - Eventi **interni** (cambiamento dello stato del processo) ed **esterni** (send/receive di messaggi)
  - $e_i^k$ :  $k$ -esimo evento generato da  $p_i$
- L'evoluzione della computazione può essere visualizzata con un **diagramma spazio-tempo**  
→ <sub>$i$</sub> : relazione di ordinamento tra due eventi in  $p_i$   
 $e \rightarrow_i e'$  se e solo se e è accaduto prima di  $e'$  in  $p_i$



Valeria Cardellini - SDCC 2021/22

4

## Timestamping

- Ogni processo etichetta gli eventi con un **timestamp**
  - In questo modo è possibile realizzare una storia globale del sistema
- Soluzione banale: ogni processo etichetta gli eventi in base al proprio orologio (clock fisico)
- Funziona?
  - E' facile ricostruire l'ordinamento degli eventi avvenuti su uno stesso processo
  - Ma l'ordinamento di eventi avvenuti su processi diversi?
  - In un sistema distribuito è *impossibile* avere un unico clock fisico condiviso da tutti i processi

# SD sincroni e asincroni

---

## • Proprietà di un **SD sincrono**

1. Esistono dei vincoli sulla velocità di esecuzione di ciascun processo
  - Il tempo di esecuzione di ciascuno passo è limitato, sia con lower bound che con upper bound
2. Ciascun messaggio trasmesso su un canale di comunicazione è ricevuto in un tempo limitato
3. Ciascun processo ha un clock fisico con un tasso di scostamento (*clock drift rate*) dal clock reale conosciuto e limitato

## • In un **SD asincrono**

- *Non ci sono vincoli* sulla velocità di esecuzione dei processi, sul ritardo di trasmissione dei messaggi e sul tasso di scostamento dei clock

# Soluzioni per sincronizzare i clock

---

## • Prima soluzione

- Tentare di **sincronizzare** con *una certa approssimazione* i **clock fisici** dei processi attraverso opportuni algoritmi
  - Ogni processo può etichettare gli eventi con il valore del suo clock fisico (che risulta sincronizzato con gli altri clock con una certa approssimazione)
  - Timestamping basato sul tempo fisico (**clock fisico**)
- 
- E' sempre possibile mantenere limitata l'approssimazione dei clock fisici?
    - **No** in un **SD asincrono**
    - Quindi in un SD asincrono il timestamping non può basarsi sul concetto di tempo fisico
    - Si introduce la nozione di clock basata su un tempo logico (**clock logico**)

## Clock fisico

---

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

$$C_i(t) = aH_i(t) + b$$

che approssimativamente misura l'istante di tempo fisico  $t$  per il processo  $p_i$

- Ad es.  $C_i(t)$  è un numero a 64 bit che fornisce i nsec trascorsi da un istante di riferimento fino all'istante  $t$
  - In generale il clock non è completamente accurato: può essere diverso da  $t$
  - Se  $C_i$  si comporta abbastanza bene, può essere usato per il timestamping degli eventi di  $p_i$
- Quale deve essere la risoluzione del clock per poter distinguere due eventi?

$$T_{\text{risoluzione}} < \Delta T \text{ tra due eventi rilevanti}$$

## Clock fisici in un SD

---

- In un SD clock fisici diversi con possibili valori diversi
- **Skew**: differenza istantanea fra il valore di due clock
- **Drift**: i clock contano il tempo con frequenze differenti (a causa di variazioni fisiche), quindi nel tempo **divergono** rispetto al tempo reale
- **Drift rate**: differenza per unità di tempo di un clock rispetto ad un orologio ideale
  - Ad es. drift rate di 2  $\mu\text{sec/sec}$  significa che il clock incrementa il suo valore di 1 sec+2  $\mu\text{sec}$  ogni secondo
  - Drift rate di normali orologi al quarzo:  $10^{-6} \text{ s/s}$  (circa 1 s in 11-12 giorni)
  - Drift rate di orologi al quarzo ad alta precisione:  $10^{-7}$  o  $10^{-8} \text{ s/s}$
  - A causa del drift rate, dopo un certo intervallo di tempo i clock di un SD saranno di nuovo disallineati → occorre eseguire una **sincronizzazione periodica** per riallineare i clock

# UTC

---

- **Universal Coordinated Time (UTC)**: standard internazionale per mantenere il tempo
- **Basato sul tempo atomico**, occasionalmente corretto utilizzando il tempo astronomico
  - 1 sec = tempo impiegato dall'atomo di cesio 133 per compiere 9192631770 transizioni di stato
- I clock fisici che usano oscillatori atomici sono i più accurati (drift rate pari a  $10^{-13}$  s/s)
- L'output dell'orologio atomico è inviato in broadcast da stazioni radio su terra e da satelliti (es. GPS)
  - In Italia: Istituto Galileo Ferraris
- Nodi con ricevitori possono sincronizzare i loro clock con questi segnali
  - Segnali da stazioni radio su terra: accuratezza nell'intervallo 1-10 ms
  - Segnali da satellite: accuratezza da 0.5 ms fino a 50 ns

Valeria Cardellini - SDCC 2021/22

10

21/12/2022

## Sincronizzazione di clock fisici

---

- Come sincronizzare i clock fisici con l'orologio atomico oppure tra di loro?
- **Sincronizzazione esterna**

I clock  $C_i$  (per  $i = 1, 2, \dots, N$ ) sono sincronizzati con una sorgente di tempo S (UTC), in modo che, dato un intervallo  $I$  di tempo reale:

$$|S(t) - C_i(t)| \leq \alpha \text{ per } 1 \leq i \leq N \text{ e per tutti gli istanti in } I$$

- I clock  $C_i$  hanno accuratezza  $\alpha$ , con  $\alpha > 0$

- **Sincronizzazione interna**

Due clock  $C_i$  e  $C_j$  sono sincronizzati l'uno con l'altro in modo che:

$$|C_i(t) - C_j(t)| \leq \pi \text{ per } 1 \leq i, j \leq N \text{ nell'intervallo } I$$

- I clock  $C_i$  e  $C_j$  hanno precisione  $\pi$ , con  $\pi > 0$

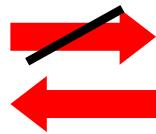
Quindi per sincronizzazione esterna parlo di ACCURATEZZA per sincronizzazione interna parlo di PRECISIONE

Valeria Cardellini - SDCC 2021/22

11

# Sincronizzazione di clock fisici

Sincronizzazione interna



Sincronizzazione esterna

- Clock sincronizzati internamente non sono necessariamente sincronizzati anche esternamente
  - Tutti i clock possono deviare collettivamente da una sorgente esterna sebbene rimangano tra loro sincronizzati entro il bound  $D$
- Se l'insieme dei processi è sincronizzato esternamente con accuratezza  $\alpha$ , allora è anche sincronizzato internamente con precisione  $2\alpha$

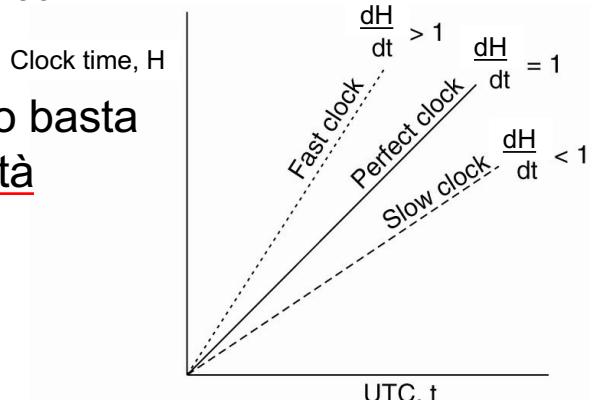
## Correttezza di clock fisici

- Un clock hardware  $H$  è *corretto* se il suo **drift rate** si mantiene in un intervallo limitato  $\rho > 0$ 
  - Il drift rate di un clock corretto è compreso tra  $-\rho$  e  $+\rho$
- Se il clock  $H$  è corretto, l'*errore* che si commette nel misurare un intervallo di istanti reali  $[t, t']$  (con  $t' > t$ ) è *limitato*:

$$(1 - \rho) (t' - t) \leq H(t') - H(t) \leq (1 + \rho) (t' - t)$$

- Si evitano “salti” del valore del clock

- Per il clock software  $C$  spesso basta una condizione di monotonicità  
 $t' > t$  implica  $C(t') > C(t)$



# Quando sincronizzare i clock fisici?

---

- Consideriamo 2 clock aventi lo stesso tasso di scostamento massimo (*maximum clock drift rate*) da UTC pari a  $\rho$
- Ipotizziamo che dopo la sincronizzazione i 2 clock si scostino da UTC in senso opposto
  - Dopo  $\Delta t$  si saranno scostati di  $2\rho \Delta t$
- Per garantire che i 2 clock non differiscano mai più di  $\delta$ , occorre sincronizzarli almeno ogni  $\delta/2\rho$  secondi

## Sincronizzazione interna in un SD sincrono

---

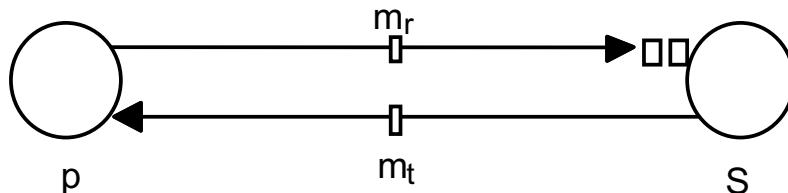
- Algoritmo di **sincronizzazione interna** tra due processi in un **SD sincrono**
  1. Un processo  $p_1$  manda il suo clock locale  $t$  ad un processo  $p_2$  tramite un messaggio  $m$
  2.  $p_2$  riceve  $m$  e imposta il suo clock a  $t + T_{trasm}$  dove  $T_{trasm}$  è il tempo di trasmissione di  $m$   
 $T_{trasm}$  non è noto ma, essendo il SD sincrono,  $T_{min} \leq T_{trasm} \leq T_{max}$   
Sia  $u = (T_{max} - T_{min})$  l'incertezza sul tempo di trasmissione (ovvero l'ampiezza dell'intervallo)
  3. Se  $p_2$  imposta il suo clock a  $t + (T_{max} + T_{min})/2$ , il lower bound ottimo sullo skew tra i due clock è pari a  $u/2$
- L'algoritmo può essere generalizzato per sincronizzare  $N$  processi, con lower bound ottimo sullo skew pari a  $u(1-1/N)$
- Per un **SD asincrono**  $T_{trasm} = T_{min} + x$ , con  $x \geq 0$  e non noto
  - Occorrono altri algoritmi di sincronizzazione dei clock fisici

# Sincronizzazione mediante time service

- Un **time service** può fornire l'ora con precisione
  - Dotato di un ricevitore UTC o di un clock accurato
- Il gruppo di processi che deve sincronizzarsi usa un **time service**
  - Time service implementato in modo centralizzato da un solo processo (time server) oppure in modo decentralizzato da più processi
- Time service **centralizzati**
  - Request-driven: algoritmo di Cristian (1989)
    - Sincronizzazione esterna
  - Broadcast-based: algoritmo di Berkeley Unix - Gusella & Zatti (1989)
    - Sincronizzazione interna
- Time service **distribuiti**
  - Network Time Protocol
    - Sincronizzazione esterna

## Algoritmo di Cristian

- Un **time server  $S$  (passivo)** riceve il segnale da una sorgente UTC (**sincronizzazione esterna**)
- Un processo  $p$  richiede il tempo con un messaggio  $m_r$ , e riceve  $t$  nel messaggio  $m_t$  da  $S$
- $p$  imposta il suo clock a  $t + T_{\text{round}}/2$ 
  - $T_{\text{round}}$  è il round trip time misurato da  $p$



### Osservazioni

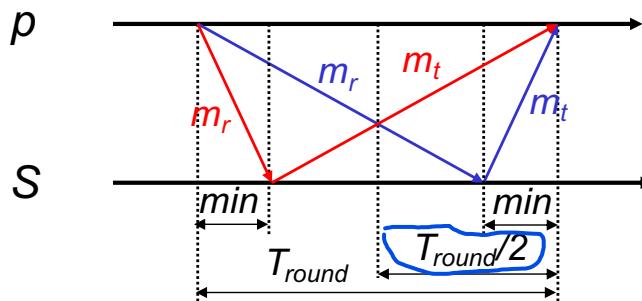
- Singolo time server potrebbe guastarsi
  - Soluzione: usare un gruppo di time server sincronizzati
- Non gestiti time server maliziosi
- Accuratezza ragionevole solo se  $T_{\text{round}}$  è breve → adatto per reti locali con bassa latenza

## Algoritmo di Cristian: accuratezza

- Caso 1:** S non può mettere  $t$  in  $m_t$  prima che sia trascorso  $min$  dall'istante in cui  $p$  ha inviato  $m_r$
- $min$  è il minimo tempo di trasmissione tra  $p$  e  $S$

- Caso 2:** S non può mettere  $t$  in  $m_t$  dopo il momento in cui  $m_t$  arriva a  $p$  meno  $min$

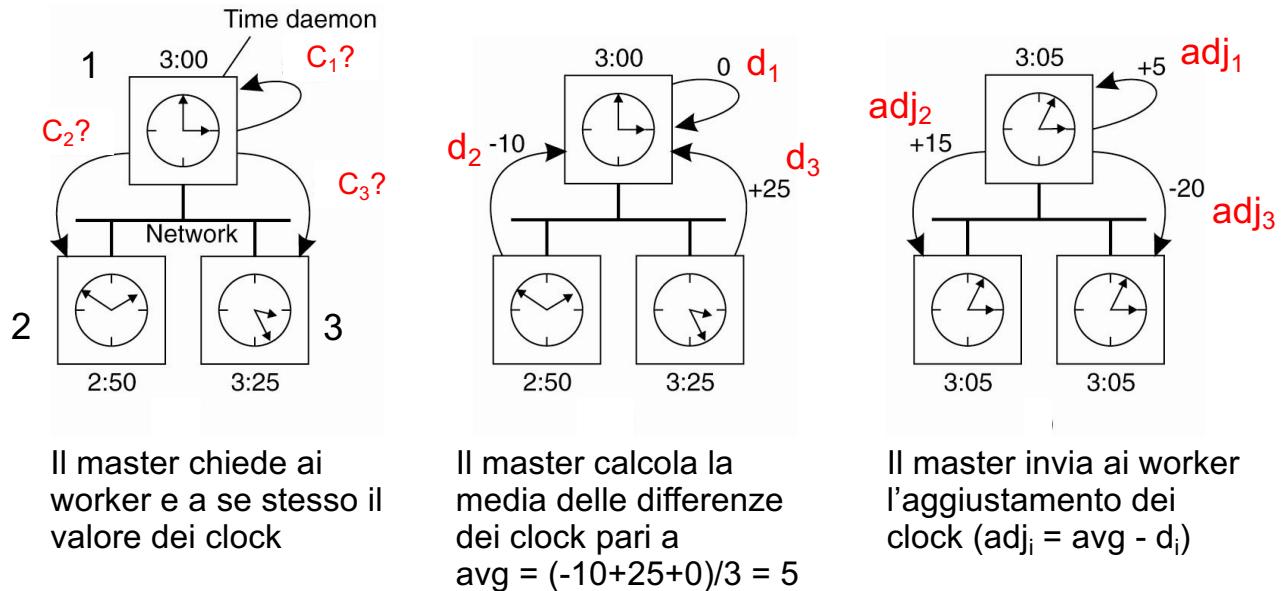
- Il tempo su  $S$  quando  $m_t$  arriva a  $p$  è compreso in  $[t + min, t + T_{round} - min]$ 
  - L'ampiezza di tale intervallo è  $T_{round} - 2 min$
- Accuratezza:  $\alpha \leq \pm (T_{round}/2 - min)$



## Algoritmo di Berkeley

- Algoritmo per la sincronizzazione interna di un gruppo di macchine
- Il *master* (time server attivo) richiede in broadcast il valore dei clock delle altre macchine (*worker*)
- Il master usa i RTT per stimare i valori dei clock dei worker
  - $d_i$  è la differenza tra clock del master  $M$  e clock del worker  $i$  (si calcola in modo simile all'algoritmo di Cristian):
$$d_i = (C_M(t_1) + C_M(t_3))/2 - C_i(t_2)$$
 dove:  $C_M(t_1)$  e  $C_M(t_3)$  sono i valori del clock sul master  $C_i(t_2)$  è il valore del clock sul worker  $i$
- Calcola la media delle differenze dei clock
- Invia un valore correttivo ai worker
  - Se il valore correttivo prevede un salto indietro nel tempo, il worker non imposta il nuovo valore ma rallenta il clock

# Algoritmo di Berkeley: esempio



## Algoritmo di Berkeley: caratteristiche

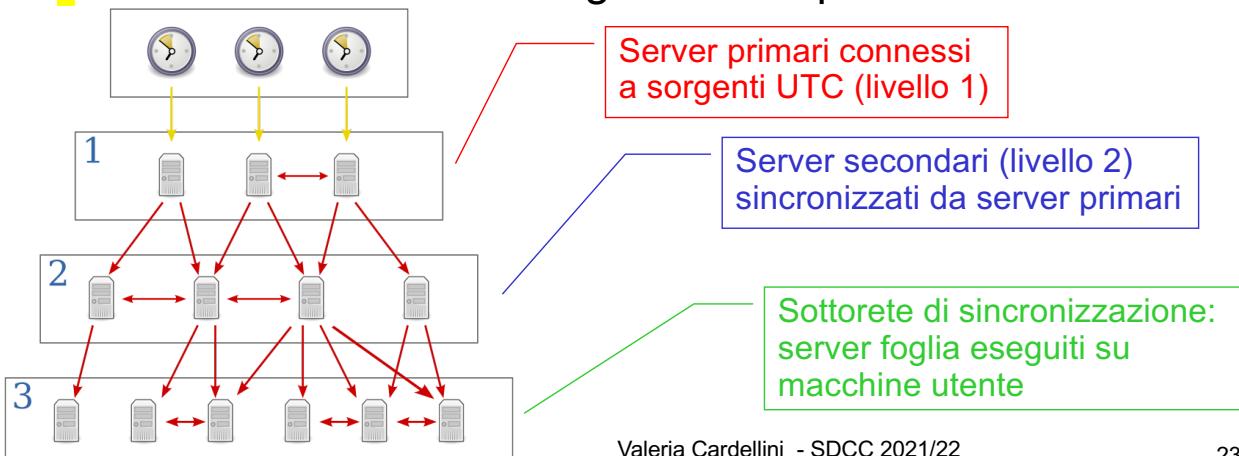
- L'accuratezza dipende da un RTT nominale massimo: il master non considera valori di clock associati a RTT superiori al massimo
  - Si scartano gli outlier
- Tolleranza ai guasti
  - Se il master cade, un'altra macchina viene eletta master
    - Tramite un algoritmo di elezione
  - Tollerante a comportamenti arbitrari (worker che inviano valori errati di clock)
    - Il master considera solo un certo numero valori di clock (da un sottoinsieme di worker); questi valori differiscono tra loro al più per una soglia specificata

# Algoritmo di Berkeley: slowdown del clock

- Che cosa significa rallentare un clock?
- Non è possibile imporre un valore di tempo passato ai worker che hanno un valore di clock superiore a quello calcolato come clock comune
  - Ciò provocherebbe un problema di ordinamento causa/effetto di eventi e verrebbe violata la condizione di monotonicità del tempo
- La soluzione consiste nel mascherare una serie di interrupt che fanno avanzare il clock locale in modo da rallentare l'avanzata del clock stesso
  - Il numero di interrupt mascherati è pari al tempo di slowdown diviso il periodo di interrupt del processore

## Network Time Protocol (NTP)

- Time service per Internet (standard in RFC 5905)
  - Sincronizzazione esterna accurata rispetto a UTC
  - Servizio configurabile su diversi SO. In GNU/Linux: demone ntpd, comando ntpdate per sincronizzare manualmente
- Architettura di time service disponibile e scalabile
  - Time server e path ridondanti
- Autenticazione delle sorgenti di tempo



# NTP: sincronizzazione

---

- La sottorete di sincronizzazione si riconfigura in caso di guasti
  - Server primario che perde la connessione alla sorgente UTC diventa server secondario
  - Server secondario che perde la connessione al suo primario (ad es. crash del primario) può usare un altro primario

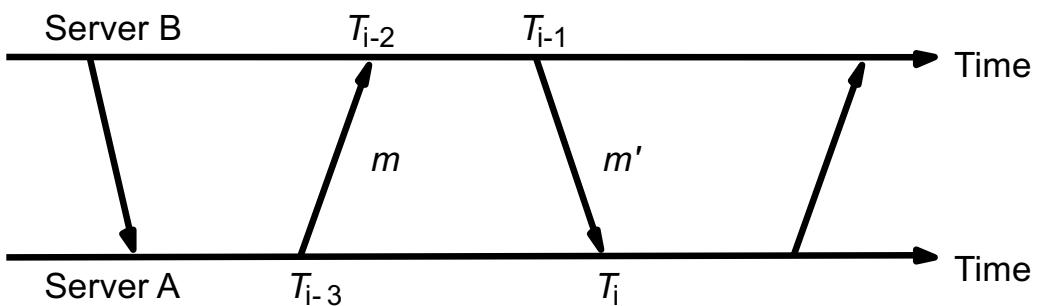
# NTP: modi di sincronizzazione

---

- Modi di sincronizzazione
  - **Multicast**: server all'interno di LAN ad alta velocità invia in multicast il suo tempo agli altri, che impostano il tempo ricevuto assumendo un certo ritardo di trasmissione
    - Accuratezza relativamente bassa
  - **Procedure call**: un server accetta richieste da altri (come algoritmo di Cristian)
    - Accuratezza maggiore rispetto a multicast
    - Utile se non è disponibile multicast
  - **Simmetrico**: coppie di server scambiano messaggi contenenti informazioni sul timing
    - Accuratezza molto alta (usato per livelli alti della gerarchia)
- Tutti i modi di sincronizzazione usano **UDP**

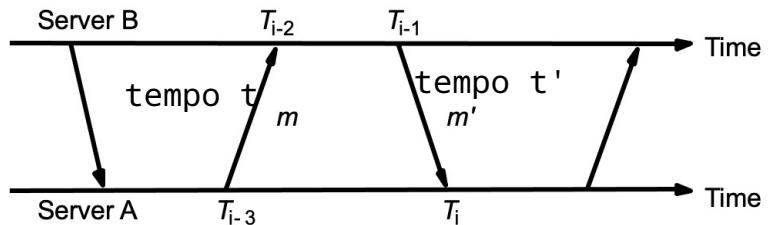
# NTP: modo simmetrico

- I server si scambiano coppie di messaggi ( $m, m'$ ) per migliorare l'accuratezza della loro sincronizzazione
- Ogni messaggio NTP contiene timestamp di eventi recenti:
  - Tempo locale di send ( $T_{i-1}$ ) del messaggio corrente  $m'$
  - Tempo locale di send ( $T_{i-3}$ ) e di receive ( $T_{i-2}$ ) del messaggio precedente  $m$
- Il ricevente di  $m'$  registra il tempo locale  $T_i$
- Il tempo tra l'arrivo di  $m$  e l'invio di  $m'$  ( $T_{i-1} - T_{i-2}$ ) può essere non trascurabile



# NTP: accuratezza

- Per ogni coppia di messaggi  $m$  ed  $m'$  scambiati tra 2 server, NTP stima l'offset  $o$ , tra i 2 clock e il ritardo  $d$ , (pari al tempo totale di trasmissione per  $m$  ed  $m'$ )



- Indicando con:
  - $o$ : offset reale del clock di B rispetto ad A ( $o = \text{clock}_B - \text{clock}_A$ )
  - $t$  e  $t'$ : tempi di trasmissione di  $m$  ed  $m'$  rispettivamente

- si ha

$$T_{i-2} = T_{i-3} + t + o \quad \text{e} \quad T_i = T_{i-1} + t' - o$$

$$\underline{d_i = t + t'} = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

- Sottraendo le equazioni:

$$o = o_i + (t' - t)/2 \quad \text{con } o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$$

- Poiché  $t, t' > 0$  si può dimostrare che

$$o_i - d_i/2 \leq o \leq o_i + d_i/2 \quad \text{monotonia}$$

- Quindi:  $o_i$  è la stima dell'offset e  $d_i/2$  l'accuratezza di questa stima

## NTP: accuratezza

---

- I server NTP applicano un algoritmo di filtraggio statistico sulle 8 coppie  $\langle o_i, d_i \rangle$  più recenti, scegliendo come stima di  $o$  il valore di  $o_j$  corrispondente al minimo  $d_j$
- Applicano poi un algoritmo di selezione dei peer per modificare eventualmente il peer da usare per sincronizzarsi <http://www.eecis.udel.edu/~mills/ntp/html/warp.html>
- Accuratezza di NTP:
  - 10 ms su Internet
  - 1 ms su LAN

Perfect synchronization over networks  
is actually impossible

## Google's TrueTime (TT)

---

- Distributed synchronized clock with bounded non-zero error
  - Proposed by Google for developing Spanner, a global-scale distributed NewSQL database
  - TT relies on a well engineered tight clock synchronization available at all servers thanks to GPS and atomic clocks
  - Cons: TT requires special hardware and a custom-build tight clock synchronization protocol, which is infeasible for many systems
    - Google also relies on its very high throughput, global fiber optic network linking its data centers

# Tempo nei SD asincroni

---

- Gli algoritmi per la sincronizzazione dei clock fisici si basano sulla stima dei tempi di trasmissione
  - Possiamo determinare l'accuratezza solo conoscendo upper e lower bound del tempo di trasmissione
- Ma in un SD asincrono i tempi di trasmissione non sono sempre predibili
- Quindi non è possibile ordinare gli eventi che accadono in processi diversi usando il tempo fisico
- Tuttavia, ciò che di solito interessa è che i processi concordino sull'ordine in cui si verificano gli eventi, piuttosto che sul tempo in cui sono avvenuti
  - Il tempo fisico non è più importante, usiamo il tempo logico

# Tempo logico

---

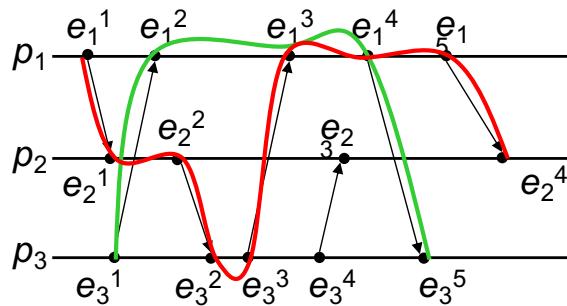
- Idea: usare un ordinamento degli eventi basato su due osservazioni intuitive:
  - Due eventi occorsi sullo stesso processo  $p_i$  si sono verificati esattamente nell'ordine in cui  $p_i$  li ha osservati
  - Quando un messaggio viene inviato da un processo  $p_i$  ad un processo  $p_j$ , l'evento di *send* precede l'evento di *receive*
- Lamport (1978) introduce il concetto di relazione di ***happened-before*** (anche detta relazione di ***precedenza*** o ***ordinamento causale***)
  - Indichiamo con  $\rightarrow_i$  la relazione di ordinamento su un processo  $p_i$
  - Indichiamo con  $\rightarrow$  la relazione di happened-before tra due eventi qualsiasi

## Relazione happened-before

- Due eventi  $e$  ed  $e'$  sono in relazione di happened-before (indicata con  $e \rightarrow e'$ ) se è vero uno dei seguenti casi:
  1.  $\exists p_i \mid e \rightarrow_i e'$
  2.  $e = \text{send}(m) \wedge e' = \text{receive}(m)$   
 $e$  è l'evento di invio del messaggio  $m$ ,  $e'$  è il corrispondente evento di ricezione
  3.  $\exists e, e', e'' \mid (e \rightarrow e'') \wedge (e'' \rightarrow e')$   
Ovvero la relazione happened-before è transitiva
- Applicando i tre casi è possibile costruire una sequenza di eventi  $e_1, e_2, \dots, e_n$  causalmente ordinati
- Osservazioni
  - La relazione happened-before rappresenta un **ordinamento parziale** (proprietà: *non riflessivo, antisimmetrico, transitivo*)
  - Non è detto che la sequenza  $e_1, e_2, \dots, e_n$  sia **unica**
  - Data una coppia di eventi, questa non è sempre legata da una relazione happened-before; in questo caso si dice che gli eventi sono **concorrenti** (indicato da  $\parallel$ )

32

## Relazione happened-before: esempio



- Sequenza  $s_1 = e_1^1, e_2^1, e_2^2, e_3^2, e_3^3, e_1^3, e_1^4, e_1^5, e_2^4$
- Sequenza  $s_2 = e_3^1, e_1^2, e_1^3, e_1^4, e_3^5$
- Gli eventi  $e_3^1$  ed  $e_2^1$  sono concorrenti  
 $e_3^1 \not\rightarrow e_2^1$  ed  $e_2^1 \not\rightarrow e_3^1$

# Clock logico scalare

---

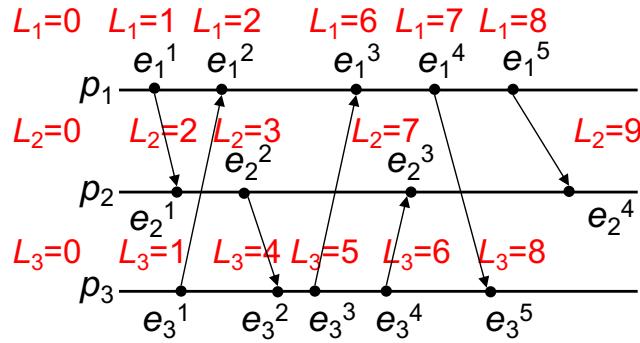
- Il clock logico è un contatore software **monotonicamente crescente**, il cui valore non ha alcuna relazione con il clock fisico
- Ogni processo  $p_i$  ha il proprio clock logico  $L_i$  e lo usa per applicare i **timestamp** agli eventi
- Denotiamo con  $L_i(e)$  il timestamp, basato sul clock logico, applicato dal processo  $p_i$  all'evento  $e$
- Proprietà: **se  $e \rightarrow e'$  allora  $L(e) < L(e')$**
- Osservazione:
  - Se  $L(e) < L(e')$  non è detto che  $e \rightarrow e'$ ; tuttavia,  $L(e) \neq L(e')$  implica che  $e \neq e'$
  - La relazione happened-before introduce un **ordinamento parziale** degli eventi: nel caso di eventi concorrenti non è possibile stabilire quale evento avviene effettivamente prima

## Clock logico scalare: implementazione

---

- **Algoritmo di Lamport**
- Ogni processo  $p_i$  inizializza il proprio clock logico  $L_i$  a 0 ( $\forall i = 1, \dots, N$ )
- Prima di eseguire un evento **interno**,  $p_i$  incrementa  $L_i$  di 1:  $L_i = L_i + 1$
- Quando  $p_i$  **invia** il messaggio  $m$  a  $p_j$ 
  - Incrementa il valore di  $L_i$ :  $L_i = L_i + 1$
  - Allega al messaggio  $m$  il timestamp  $t = L_i$
  - Esegue l'evento  $send(m)$
- Quando  $p_j$  **riceve** il messaggio  $m$  con timestamp  $t$ 
  - Aggiorna il proprio clock logico  $L_j = \max(t, L_j)$
  - Incrementa il valore di  $L_j$ :  $L_j = L_j + 1$
  - Esegue l'evento  $receive(m)$

## Clock logico scalare: esempio



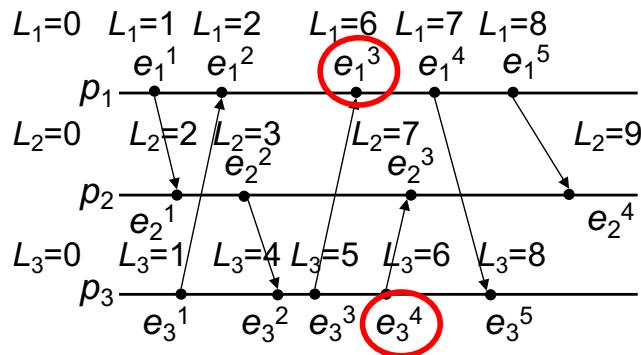
### • Osservazioni

- $e_1^1 \rightarrow e_2^1$  ed i relativi timestamp riflettono questa proprietà ( $L_1=1$  e  $L_2=2$ ); infatti if  $e_1^1 \rightarrow e_2^1$  then  $L(e_1^1) < L(e_2^1)$
- $e_1^1 \parallel e_3^1$  ed i relativi timestamp sono uguali ( $L_1=1$  e  $L_3=1$ ); infatti if  $L(e_1^1) \geq L(e_3^1)$  then  $e_1^1 \not\rightarrow e_3^1$
- $e_2^1 \parallel e_3^1$  ed i relativi timestamp sono diversi ( $L_2=2$  e  $L_3=1$ ); infatti if  $L(e_2^1) \geq L(e_3^1)$  then  $e_2^1 \not\rightarrow e_3^1$

## Relazione di ordine totale

- Usando il **clock logico scalare**, due o più eventi **possono avere stesso timestamp**: come realizzare un **ordinamento totale tra eventi**, evitando così che due eventi accadano nello *stesso tempo logico*?
- Soluzione: **usare**, oltre al **clock logico**, il **numero del processo** in cui è avvenuto l'evento
  - Si stabilisce un **ordinamento totale** ( $<$ ) tra i processi
- **Relazione di ordine totale** tra eventi (indicata con  $e \Rightarrow e'$ ): se  $e$  è un evento nel processo  $p_i$  ed  $e'$  è un evento nel processo  $p_j$  allora  $e \Rightarrow e'$  se e solo:
  1.  $L_i(e) < L_j(e')$  or
  2.  $L_i(e) = L_j(e')$  and  $p_i < p_j$
- Applicata negli algoritmi di Lamport distribuito e di Ricart-Agrawala per la mutua esclusione distribuita

## Relazione di ordine totale: esempio



- $e_1^3$  e  $e_3^4$  hanno lo stesso valore di clock logico: come ordinarli?
- $e_1^3 \Rightarrow e_3^4$  poiché  $L_1(e_1^3) = L_3(e_3^4)$  e  $p_1 < p_3$

## Clock logico scalare: limitazione

- Il clock logico scalare ha la seguente proprietà
  - Se  $e \rightarrow e'$  allora  $L(e) < L(e')$
- Ma **non è possibile assicurare** che
  - Se  $L(e) < L(e')$  allora  $e \rightarrow e'$   
Esempio slide 36:  $L(e_3^1) < L(e_2^1)$  ma  $e_3^1 \parallel e_2^1$
- **Conseguenza:** non è possibile stabilire, solo guardando i clock logici scalari, se due eventi sono concorrenti o meno
- Come superare questa limitazione?
- Introducendo i **clock logici vettoriali**: ad opera di Mattern (1989) e Fidge (1991)

# Clock logico vettoriale

---

- Il clock logico vettoriale per un sistema con  $N$  processi è un vettore di  $N$  interi
- Ciascun processo  $p_i$  mantiene il proprio clock vettoriale  $V_i$
- Per il processo  $p_i$   $V_i[i]$  è il clock logico locale
- Ciascun processo usa il suo clock vettoriale per assegnare il timestamp agli eventi
- Analogamente al clock scalare di Lamport, il clock vettoriale viene allegato al messaggio  $m$  ed il timestamp diviene vettoriale
- Con il clock vettoriale si catturano completamente le caratteristiche della relazione happened-before  
 $e \rightarrow e' \text{ se e solo se } V(e) < V(e')$

## Clock logico vettoriale: significato e confronto

---

- Dato il clock vettoriale  $V_i$ 
  - $V_i[i]$  è il numero di eventi generati da  $p_i$
  - $V_i[j]$  con  $i \neq j$  è il numero di eventi occorsi a  $p_j$  di cui  $p_i$  ha conoscenza
- Confronto di clock vettoriali
  - $V = V'$  se e solo se  $\forall j: V[j] = V'[j]$
  - $V \leq V'$  se e solo se  $\forall j: V[j] \leq V'[j]$
  - $V < V'$  (e quindi l'evento associato a  $V$  precede quello associato a  $V'$ ) se e solo se
    - $\forall i \in [1, \dots, N]: V[i] \leq V'[i]$   
and
    - $\exists j \in [1, \dots, N]: V[j] < V'[j]$
  - $V \parallel V'$  (e quindi l'evento associato a  $V$  è concorrente a quello associato a  $V'$ ) se e solo se
    - $\text{not}(V < V') \text{ and } \text{not}(V' < V)$

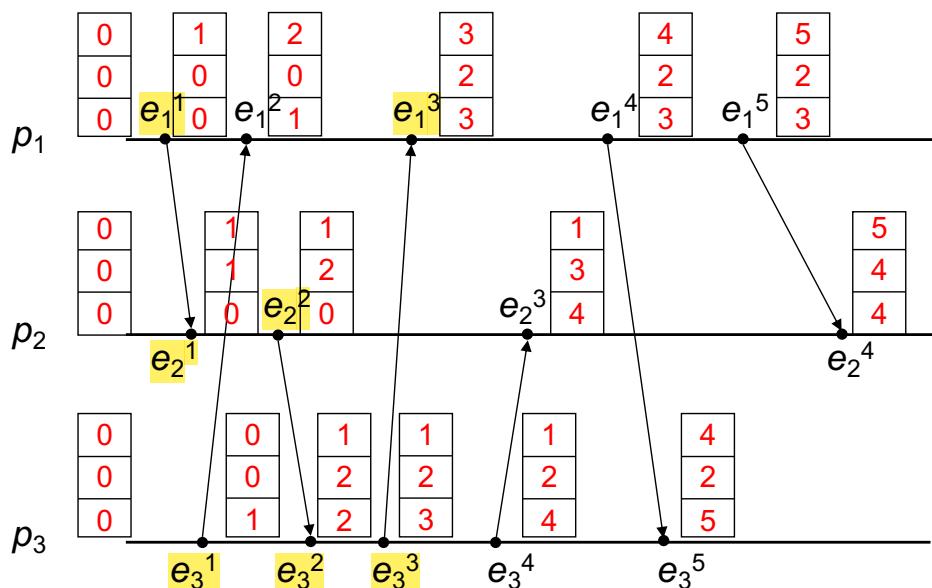
## Clock logico vettoriale: implementazione

- Ogni processo  $p_i$  inizializza il proprio clock vettoriale  $V_i$   
 $V_i[k]=0 \quad \forall k = 1, 2, \dots, N$
- Prima di eseguire un evento interno,  $p_i$  incrementa di 1 la sua componente  $V_i[i]$   
$$V_i[i] = V_i[i] + 1$$
- Quando  $p_i$  invia il messaggio  $m$  a  $p_j$ 
  - Incrementa di 1 la sua componente  $V_i[j]$ :  $V_i[j] = V_i[j] + 1$
  - Allega al messaggio  $m$  il timestamp vettoriale  $t = V_i$
  - Esegue l'evento  $send(m)$
- Quando  $p_j$  riceve il messaggio  $m$  con timestamp  $t$ 
  - Aggiorna il proprio clock logico  $V_j[k] = \max(t[k], V_j[k]) \quad \forall k = 1, 2, \dots, N$
  - Incrementa di 1 la sua componente  $V_j[j]$ :  $V_j[j] = V_j[j] + 1$
  - Esegue l'evento  $receive(m)$

Valeria Cardellini - SDCC 2021/22

42

## Clock logico vettoriale: esempio



# Confronto di clock vettoriali

- Confrontando i timestamp basati su clock vettoriale si può capire se gli eventi sono concorrenti o in relazione happened-before

1
2
0

1
2
2

V

V'

$$V(e) < V'(e) \text{ e quindi } e \rightarrow e'$$

1
2
0

1
0
2

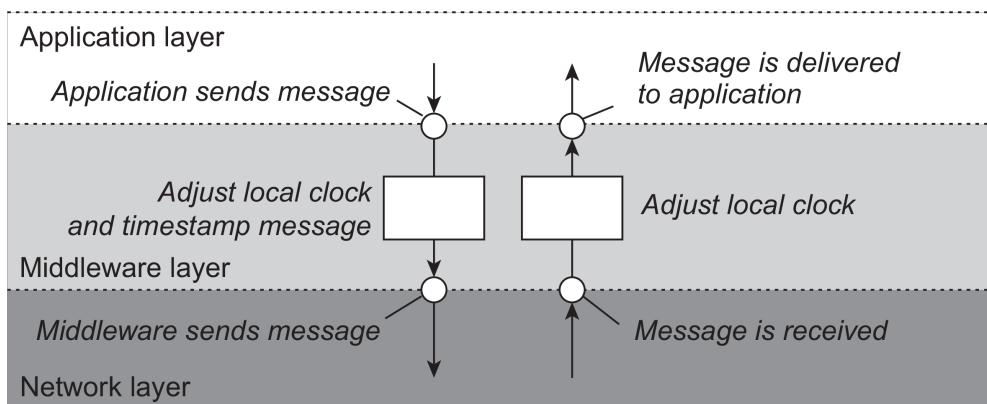
V

V'

$$V(e) \neq V'(e) \text{ e quindi } e \parallel e'$$

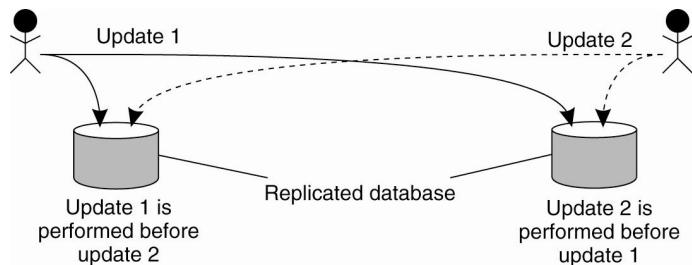
## Esempi di applicazione del clock logico

- Esaminiamo due applicazioni del clock logico scalare e vettoriale
1. Clock logico scalare per multicasting totalmente ordinato
  2. Clock logico vettoriale per multicasting causalmente ordinato



## Multicasting totalmente ordinato: problema

- Come garantire che aggiornamenti concorrenti su un database replicato siano visti nello stesso ordine da ogni replica?
  - $p_1$  aggiunge \$100 ad un conto (valore iniziale: \$1000)
  - $p_2$  incrementa il conto dell'1%
  - Ci sono due replicate
  - In assenza di sincronizzazione le due operazioni non vengono eseguite sulle due replicate nello stesso ordine
    - Replica #1  $\leftarrow 1111$  (prima  $p_1$  e poi  $p_2$ )
    - Replica #2  $\leftarrow 1110$  (prima  $p_2$  e poi  $p_1$ )

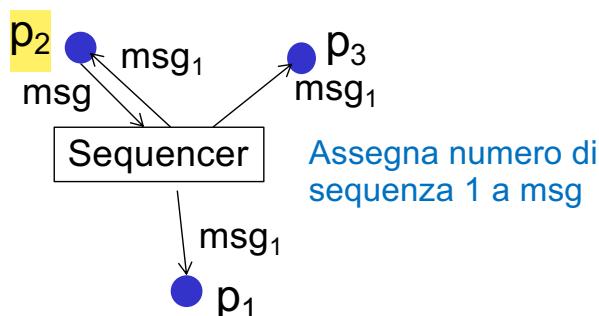


## Multicasting totalmente ordinato

- **Multicasting totalmente ordinato:** operazione di multicast con cui tutti i messaggi sono consegnati nello stesso ordine ad ogni destinatario
- Assunzioni: comunicazione *affidabile* e *FIFO ordered*
  - Comunicazione *affidabile*: no perdita di messaggi
  - Comunicazione *FIFO ordered*: messaggi inviati da  $p_i$  a  $p_j$  sono ricevuti da  $p_j$  nello stesso ordine in cui  $p_i$  li ha inviati

# Multicasting totalmente ordinato

- Algoritmo centralizzato: coordinatore centralizzato (**sequencer**)
  - Ogni processo invia il proprio messaggio di update al sequencer
  - Il sequencer assegna ad ogni messaggio di update un **numero di sequenza univoco** e poi invia in multicast il messaggio a tutti i processi, che eseguono gli aggiornamenti in ordine in base al numero di sequenza
  - Problemi di scalabilità e single point of failure



# Multicasting totalmente ordinato

- Algoritmo **distribuito**: applichiamo il **clock logico scalare** per risolvere il problema del multicasting totalmente ordinato in modo **completamente distribuito**
  - Ogni messaggio ha come timestamp il clock logico scalare del processo che lo invia

$p_i$  invia in multicast (incluso se stesso) il **messaggio di update**  $msg_i$

$msg_i$  viene posto da ogni processo ricevente  $p_j$  in una coda locale  $queue_j$ , ordinata in base al valore del **timestamp**

$p_j$  invia in multicast un **messaggio di ack** della ricezione di  $msg_i$   
 $p_j$  consegna  $msg_i$  all'applicazione se:

1.  $msg_i$  è **in testa a**  $queue_j$  (e tutti gli **ack** relativi a  $msg_i$  sono stati **ricevuti** da  $p_j$ )
2. **per ogni processo**  $p_k$  c'è un messaggio  $msg_k$  in  $queue_j$  con **timestamp maggiore** di quello di  $msg_i$

Ovvero  $msg_i$  viene consegnato solo quando  $p_j$  sa che nessun altro processo può inviare in multicast un messaggio con **timestamp minore o uguale** a quello di  $msg_i$

# Multicasting totalmente ordinato

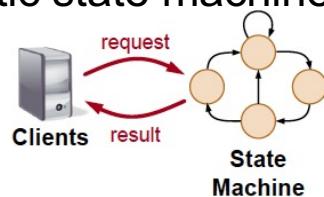
---

- Costo:
  - N processi e per ogni messaggio ricevuto viene inviato un ack in multicast → **N<sup>2</sup> ack**
  - Problemi di scalabilità in sistemi a larga scala
- Meccanismo alla base della **state-machine replication**

## State machine replication

---

- Software replication technique applied to services that can be implemented as a deterministic state machine  
Next state of the service =  
 $f(\text{current state}, \text{command executed})$
- To achieve fault-tolerance, service is replicated on several nodes, all of them running a state machine replication (SMR) middleware
  - Set of replicas behaves as a “centralized” server
- Every replica must execute commands in the same order
- If deterministic and by executing the same commands in the same order, **all replicas end up in same state**
- Service makes progress as long as any majority of the replicas are up



# Multicasting causalmente ordinato

---

- **Multicasting causalmente ordinato:** un messaggio viene consegnato solo se tutti i messaggi che lo precedono *causalmente* (relazione di causa-effetto) sono stati già consegnati
  - Relazione di causa-effetto tra due eventi: il secondo evento è *potenzialmente* influenzato dal primo evento
    - Obiettivo: **consegnare prima la causa e poi l'effetto**
  - Indebolimento del multicasting totalmente ordinato
  - Esempio:
    - $p_1$  invia i messaggi  $m_A$  ed  $m_B$
    - $p_2$  invia i messaggi  $m_C$  ed  $m_D$
    - **$m_A$  causa  $m_C$**
    - Alcune sequenze di consegna compatibili con l'ordinamento causale (e FIFO ordered) sono:  
 $m_A \ m_B \ m_C \ m_D$      $m_A \ m_C \ m_B \ m_D$      $m_A \ m_C \ m_D \ m_B$   
**ma NON  $m_C \ m_A \ m_B \ m_D$**

# Multicasting causalmente ordinato

---

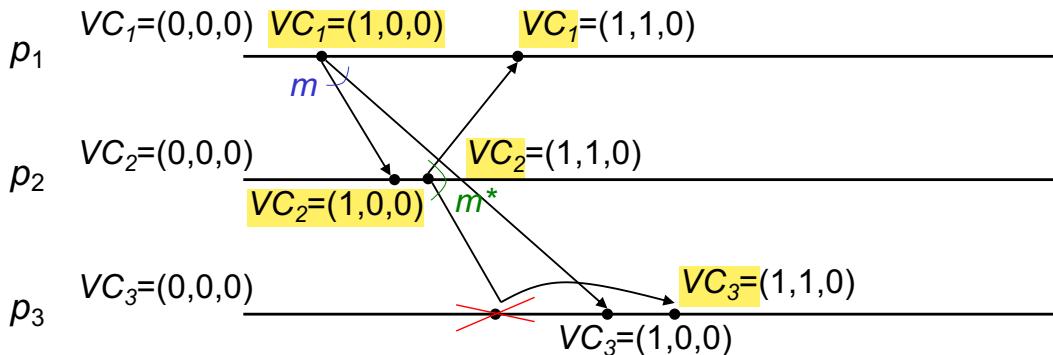
- Assunzioni: **comunicazione affidabile** e **FIFO ordered**
- Applichiamo il **clock logico vettoriale** per risolvere il problema del multicasting causalmente ordinato in modo **completamente distribuito**
  - $p_i$  invia il messaggio  $m$  con timestamp  $t(m)$  basato sul clock logico vettoriale  $V_i$ 
    - $V_j[i]$  conta il numero di messaggi da  $p_i$  a  $p_j$
  - $p_j$  riceve  $m$  da  $p_i$  e ne ritarda la consegna al livello applicativo (**ponendo  $m$  in una coda di attesa**) finché non si verificano **entrambe** le condizioni
    1.  $t(m)[i] = V_j[i] + 1$   
 **$m$  è il messaggio successivo che  $p_j$  si aspetta da  $p_i$**
    2.  $t(m)[k] \leq V_j[k]$  per ogni  $k \neq i$   
**per ogni processo  $p_k$ ,  $p_j$  ha visto almeno gli stessi messaggi visti da  $p_i$**

# Multicasting causalmente ordinato: esempio

- $p_1$  invia  $m$  a  $p_2$  e  $p_3$
- $p_2$ , dopo aver ricevuto  $m$ , invia  $m^*$  a  $p_1$  e  $p_3$
- Supponiamo che  $p_3$  riceva  $m^*$  prima di  $m$ : l'algoritmo evita la violazione della causalità tra  $m$  e  $m^*$ , facendo sì che su  $p_3$   $m^*$  sia consegnato al livello applicativo dopo  $m$

Aggiornamento clock

$$p_i \text{ invia msg: } VC_i[i] = VC_i[i] + 1$$
$$p_i \text{ riceve msg con } t(\text{msg}): VC_i[k] = \max\{VC_i[k], t(\text{msg})[k]\}$$



## Mutua esclusione e sistemi concorrenti

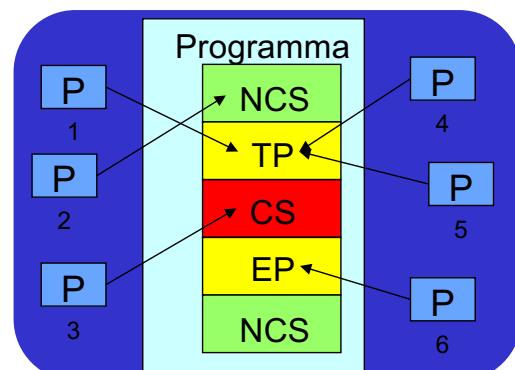
- La mutua esclusione nasce nei sistemi concorrenti
  - $N$  processi vogliono accedere ad una risorsa condivisa
  - Ogni processo vuole acquisire la risorsa ed utilizzarla in modo esclusivo senza avere interferenze con gli altri processi
- Ogni algoritmo di mutua esclusione comprende
  - Una sequenza di istruzioni chiamata **sezione critica (CS)**
    - L'esecuzione della sezione critica consiste nell'accesso alla risorsa condivisa
  - Una sequenza di istruzioni che precedono la sezione critica, chiamata **trying protocol (TP)**
  - Una sequenza di istruzioni che seguono la sezione critica, chiamata **exit protocol (EP)**

# Requisiti degli algoritmi di mutua esclusione

- Un algoritmo di mutua esclusione dovrebbe soddisfare le seguenti proprietà:
  - **Mutua esclusione (ME)** o proprietà di **safety**: al più un solo processo alla volta può eseguire la CS
  - **No deadlock (ND)**: se un processo rimane bloccato nella sua trying section, esistono uno o più processi che riescono ad entrare ed uscire dalla CS
  - **No starvation (NS)** o *assenza di posticipazione indefinita*: nessun processo può rimanere bloccato per sempre nella trying section, ovvero le richieste di ingresso e uscita dalla sezione critica sono prima o poi soddisfatte
- Osservazioni:
  - NS implica ND
  - NS è una proprietà di **liveness** (no ritardi infiniti)
  - L'assenza di starvation è una condizione di imparzialità (**fairness**) nei confronti dei processi
  - L'**ordering** è un ulteriore requisito di imparzialità
    - Richieste di accesso in CS servite secondo l'ordine d'arrivo

## Mutua esclusione e sistemi concorrenti

- La sezione critica è relativa ad una porzione di codice
- Soluzioni basate su variabili condivise per realizzare la mutua esclusione tra  $N$  processi
  - **Algoritmo di Dijkstra** (1965)
    - Ideato per sistemi a singolo processore
    - Garantisce mutua esclusione ed assenza di deadlock ma non garantisce assenza di starvation
  - **Algoritmo del panificio di Lamport** (1974)
    - Ideato per sistemi multiprocessore a memoria condivisa



# Algoritmo del panificio di Lamport

---

- Soluzione semplice ispirata ad una situazione reale
  - Attesa di essere serviti in un panificio
- Modello di sistema concorrente per Lamport
  - I processi comunicano leggendo/scrivendo variabili condivise
  - Lettura e scrittura di una variabile non sono operazioni atomiche: un processo può scrivere mentre un altro processo sta leggendo
  - Ogni variabile condivisa è di proprietà di un processo
    - Tutti possono leggere la sua variabile
    - Solo il processo può scrivere la sua variabile
  - Nessun processo può eseguire due scritture contemporaneamente
  - Le velocità di esecuzione dei processi non sono correlate tra loro

# Algoritmo del panificio di Lamport

---

- Variabili condivise
  - num[1,...,N] : array di interi inizializzati a 0
  - choosing[1,...,N] : array di booleani inizializzati a false
- Variabile locale
  - j: intero compreso in [1,...,N]

# Algoritmo del panificio di Lamport

- Ciclo ripetuto all'infinito dal processo  $p_i$ :

// sezione non critica

// prende un biglietto

choosing[i] = true; // inizio della selezione del biglietto

num[i] = 1 + max(num[x]: 1 ≤ x ≤ N);

choosing[i] = false; // fine della selezione del biglietto

// attende che il numero sia chiamato confrontando il suo biglietto

// con quello degli altri

doorway

bakery

for j = 1 to N do

// busy waiting mentre j sta scegliendo

while choosing[j] do NoOp();

// busy waiting finché il valore del biglietto non è il più basso

// viene favorito il processo con identificativo più piccolo

while num[j] ≠ 0 and {num[j], j} < {num[i], i} do NoOp();

// sezione critica

num[i] = 0;

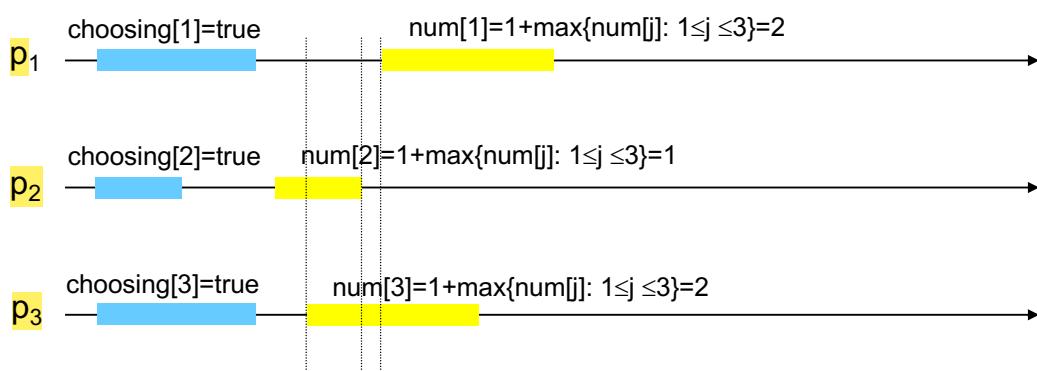
// fine sezione critica

Relazione di precedenza < su coppie ordinate di interi definita da:  $(a,b) < (c,d)$  se  $a < c$ , o se  $a = c$  e  $b < d$

# Algoritmo del panificio di Lamport

- Doorway

- Ogni processo  $p_i$  che entra lo segnala agli altri tramite choosing[i]
- Prende un numero di biglietto pari al massimo dei numeri scelti dai processi precedenti più 1
- Altri processi possono accedere concorrentemente alla doorway
- Esempio di esecuzione della doorway



# Algoritmo del panificio di Lamport

## Bakery

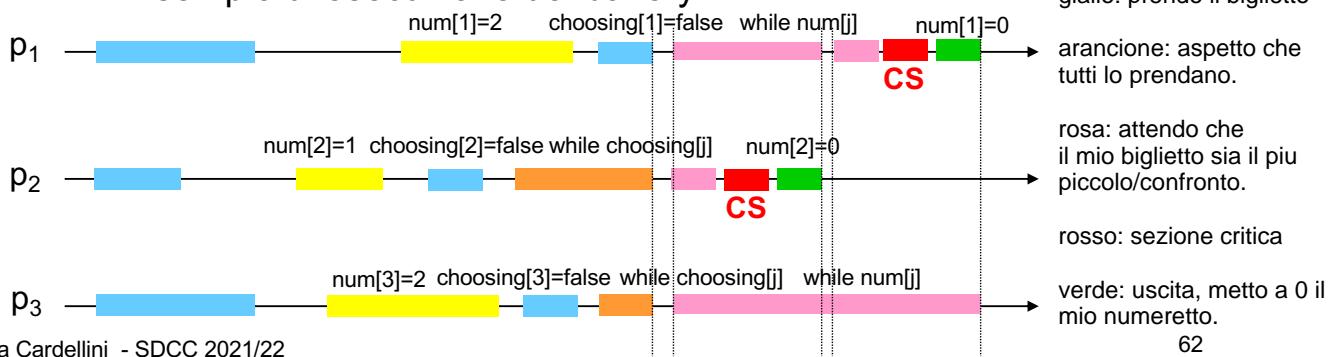
- Ogni processo  $p_i$  deve controllare che tra i processi in attesa sia lui il prossimo a poter accedere alla CS
- Il primo ciclo permette a tutti i processi nella doorway di terminare la scelta del biglietto
- Il secondo ciclo lascia  $p_i$  in attesa finché:
  - Il suo numero di biglietto non diventa il più piccolo
  - Tutti i processi che hanno scelto un numero di biglietto uguale al suo non hanno identificativo maggiore

### Osservazione:

- I casi in cui viene scelto lo stesso numero di biglietto sono risolti basandosi sull'identificativo del processo

blu: scelgo il biglietto

### Esempio di esecuzione del bakery



# Algoritmo del panificio di Lamport: proprietà

- Proprietà di mutua esclusione
  - Deriva da: se  $p_i$  è nella doorway e  $p_j$  è nel bakery allora  $\{num[j], j\} < \{num[i], i\}$
- Proprietà di no starvation
  - Nessun processo attende per sempre, poiché prima o poi avrà il numero di biglietto più piccolo
- L'algoritmo gode anche della proprietà FCFS
  - Se  $p_i$  entra nel bakery prima che  $p_j$  entri nella doorway, allora  $p_i$  entrerà in CS prima di  $p_j$

## Mutua esclusione distribuita

---

- Rispetto al modello di Lamport (slide 59) la comunicazione avviene tramite scambio di messaggi:
  - Un processo non può leggere direttamente il valore di una variabile di proprietà di un altro processo, ma deve inviare un messaggio di richiesta ed attendere un messaggio di risposta contenente il valore
- La comunicazione tra processi avviene in base alle seguenti assunzioni:
  - I processi non hanno variabili condivise ma comunicano tramite scambio di messaggi
  - Il ritardo di trasmissione di un messaggio è impredicibile ma finito
  - I canali di comunicazione tra i processi sono affidabili
    - Un messaggio inviato viene ricevuto correttamente dal suo destinatario
    - Non ci sono messaggi duplicati o messaggi spuri (ricevuti ma mai trasmessi)

## Mutua esclusione distribuita: modello del sistema

---

- Consideriamo un sistema con  $N$  processi  $p_i$ ,  $i = 1, \dots, N$
- Il sistema è asincrono
- I processi non sono soggetti a fallimenti
- La comunicazione è affidabile e FIFO ordered
- I processi trascorrono un tempo finito nella CS

## Adattamento dell'algoritmo del panificio

---

- Proviamo ad adattare ai SD l'algoritmo del panificio di Lamport ideato per sistemi concorrenti

Ogni processo  $p_i$  si comporta da server rispetto alle proprie variabili  $num[i]$  e  $choosing[i]$

Comunicazione basata non su memoria condivisa ma su scambio di messaggi

- Ogni processo legge i valori locali agli altri processi tramite messaggi di richiesta-risposta

## Adattamento dell'algoritmo del panificio

---

- E' una buona soluzione? Funzione, ma...
- Costo totale pari a  **$6N$**  messaggi
  - Per accedere alla CS servono 3 scambi di messaggi (1 per doorway, 2 per bakery)
  - Per ogni "lettura" vengono scambiati  $2N$  messaggi
- La latenza per un singolo scambio di messaggi è uguale al tempo impiegato dalla combinazione *canale di comunicazione-processo* più lenta
- Quindi: **scarsa efficienza e scalabilità**
- I problemi derivano dalla mancanza di cooperazione tra processi

# Panoramica sulle algoritmi per ME distribuita

---

- Algoritmi basati su **autorizzazioni**
  - Un processo che vuole accedere ad una risorsa condivisa chiede l'autorizzazione
  - Autorizzazione gestita
    - In modo *centralizzato* (unico coordinatore)
    - In modo *completamente distribuito* (algoritmo di *Lamport distribuito*, algoritmo di *Ricart e Agrawala*)
- Algoritmi basati su **token**
  - Tra i processi circola un messaggio speciale, detto *token*
    - Il token è unico in ogni istante di tempo
    - Solo chi detiene il token può accedere alla risorsa condivisa
  - Algoritmo *centralizzato* e *decentralizzato*
- Algoritmi basati su **quorum** (o votazione)
  - Si richiede il permesso di accedere ad una risorsa condivisa solo ad un sottoinsieme di processi
    - Algoritmo di *Maekawa*

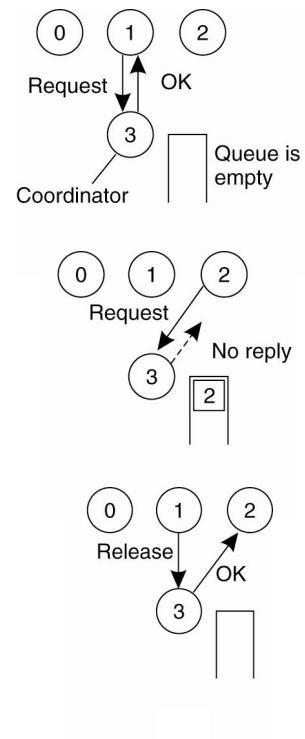
# Valutazione degli algoritmi per ME distribuita

---

- Principali criteri usati per valutare gli algoritmi di ME distribuita:
  - **Numero di messaggi scambiati per l'ingresso e l'uscita dalla CS:** è una misura indiretta della banda di rete consumata
  - **Numero di messaggi scambiati per l'ingresso in CS:** è una misura indiretta del tempo di attesa

## Autorizzazioni: algoritmo centralizzato

- La richiesta di accesso (ENTER) ad una risorsa in mutua esclusione viene inviata al **coordinatore centrale**
- Se la risorsa è libera, il coordinatore informa il mittente che l'accesso è consentito (GRANTED)
- Altrimenti, il coordinatore accoda la richiesta con **politica FIFO** e informa il mittente che l'accesso non è consentito (DENIED)
  - Oppure, nel caso di **SD sincrono, non risponde** (vedi figura)
- Il processo che rilascia la risorsa ne informa il coordinatore (RELEASED)
- Il coordinatore preleva dalla coda la prima richiesta in attesa e invia GRANTED al suo mittente



## Autorizzazioni: algoritmo centralizzato

- **Vantaggi**
  - Garantisce la mutua esclusione ed assenza di starvation
  - Garantisce fairness (coda gestita con disciplina FIFO)
  - Semplice e facile da implementare
    - Solo **3 messaggi** (richiesta, risposta e rilascio) per ingresso e uscita dalla CS
- **Svantaggi**
  - Il coordinatore è il singolo point of failure dell'algoritmo
  - Il coordinatore può diventare il collo di bottiglia per le prestazioni
  - Se fallisce un processo mentre è in CS, si perde il messaggio di rilascio

## Algoritmo di Lamport distribuito

---

- Ogni processo mantiene un **clock logico scalare** ed una **coda locale** (per memorizzare le richieste di accesso alla CS)
  - Si applica anche la **relazione di ordine totale**  $\Rightarrow$
- Regole dell'algoritmo:
  - $p_i$  richiede l'accesso in CS:  $p_i$  invia a tutti gli altri processi un **messaggio di richiesta** avente come timestamp il suo clock scalare ed aggiunge la coppia <richiesta, timestamp> alla sua coda
  - $p_j$  riceve una richiesta da  $p_i$ :  $p_j$  memorizza la richiesta (con il timestamp) nella sua coda ed invia a  $p_i$  un **messaggio di ack**
  - $p_i$  accede alla CS se e solo se:
    - la richiesta di  $p_i$  con timestamp  $t$  precede tutti gli altri messaggi di richiesta in coda (ossia  $t$  è il minimo applicando  $\Rightarrow$ )
    - $p_i$  ha ricevuto da ogni altro processo un messaggio (di ack o nuova richiesta) con timestamp maggiore di  $t$  (applicando  $\Rightarrow$ )
  - $p_i$  rilascia la CS:  $p_i$  elimina la richiesta dalla sua coda ed invia un **messaggio di release** a *tutti* gli altri processi
  - $p_j$  riceve un messaggio di release: elimina la richiesta corrispondente dalla sua coda

## Algoritmo di Lamport distribuito

---

- Somiglianza con l'algoritmo distribuito per il multicast totalmente ordinato: perché?
- Soddisfa le proprietà di mutua esclusione (**safety**, **liveness**, **fairness**)
  - Fair perché le richieste vengono servite secondo l'ordine del timestamp ed il tempo è determinato dai clock logici
- Richiede **3( $N-1$ )** messaggi per accedere alla sezione critica:
  - $N-1$  messaggi di richiesta
  - $N-1$  messaggi di ack
  - $N-1$  messaggi di release

# Algoritmo di Ricart e Agrawala

- Estensione ed ottimizzazione dell'algoritmo di Lamport distribuito

– Basato su **clock logico scalare** e **relazione d'ordine totale**

- Un processo che vuole accedere alla CS manda un **messaggio di REQUEST** a tutti gli altri contenente:
  - proprio identificatore
  - *timestamp* basato su clock logico scalare
- Si pone in attesa della risposta da tutti gli altri
- Ottenuti tutti i **messaggi di REPLY** entra nella CS
- All'uscita dalla CS manda REPLY a *tutti* i processi nella coda locale

- Un processo che riceve il messaggio di REQUEST può
  - non essere nella CS e non volervi accedere → manda **REPLY** al mittente
  - essere nella CS → non **risponde** e mette il messaggio in coda locale
  - voler accedere alla CS → **confronta il suo timestamp** con quello ricevuto e vince quello minore: se è l'altro, invia REPLY; se è lui, non risponde e mette il messaggio in coda

# Algoritmo di Ricart e Agrawala

- Variabili locali per ciascun processo
  - #replies: numero di risposte ricevute (inizializzata a 0)
  - State ∈ {Requesting, CS, NCS} (inizializzata a NCS)
  - Q: coda di richieste pendenti (inizialmente vuota)
  - Last\_Req: timestamp del messaggio di richiesta (inizializzata a 0)
  - Num (inizializzata a 0): clock logico scalare
- Ogni processo  $p_i$  esegue il seguente algoritmo

## Begin

1. State = Requesting;
2. Num = Num+1; Last\_Req = Num;
3. for j=1 to N-1 send REQUEST to  $p_j$ ;
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;

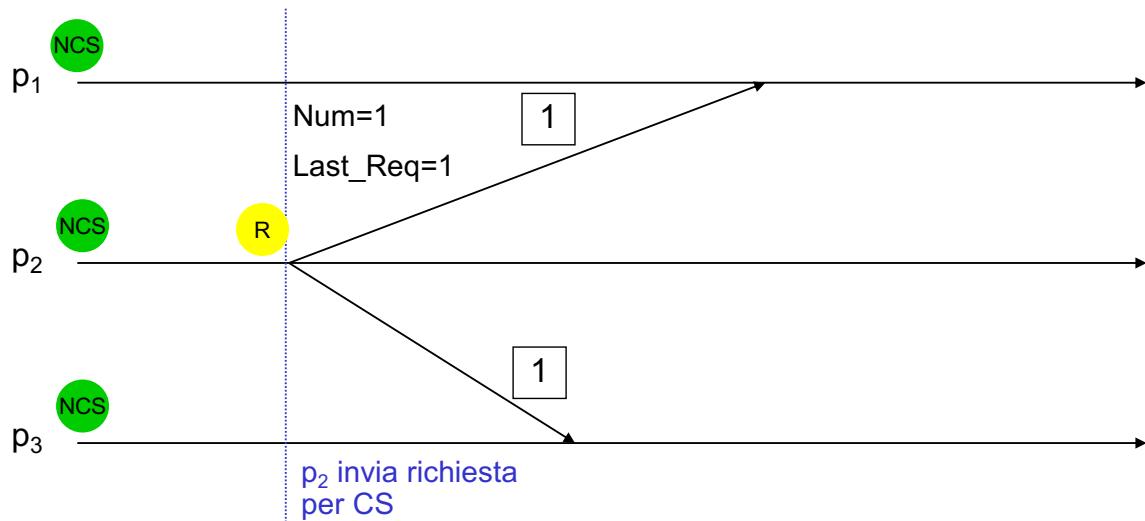
## Upon receipt of REQUEST(t) from $p_j$

1. if State=CS or (State=Requesting and  $\{Last\_Req, i\} < \{t, j\}$ )
2. then insert  $\{t, j\}$  in Q
3. else send REPLY to  $p_j$
4. Num = max(t, Num)

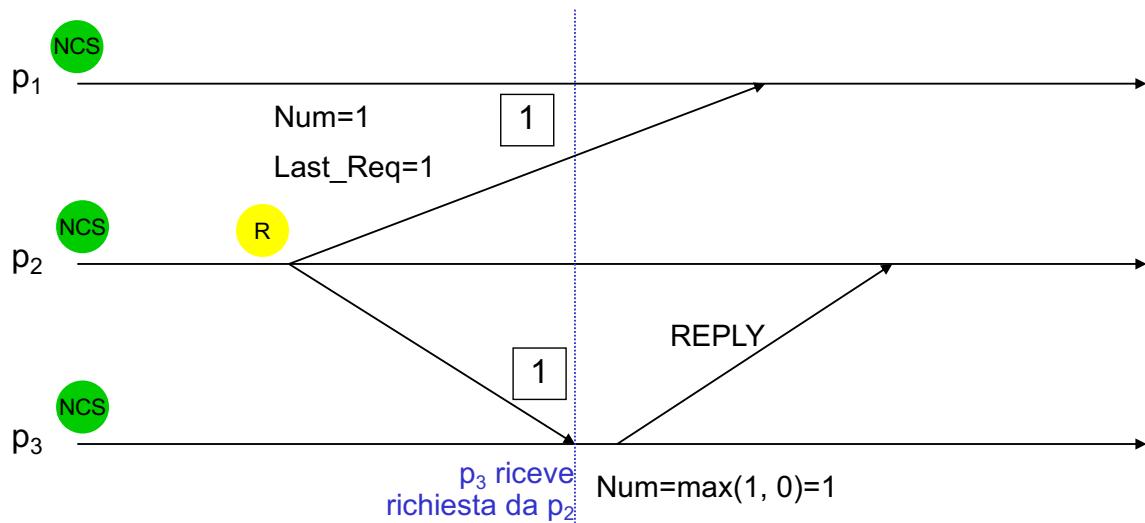
## Upon receipt of REPLY from $p_j$

1. #replies = #replies+1

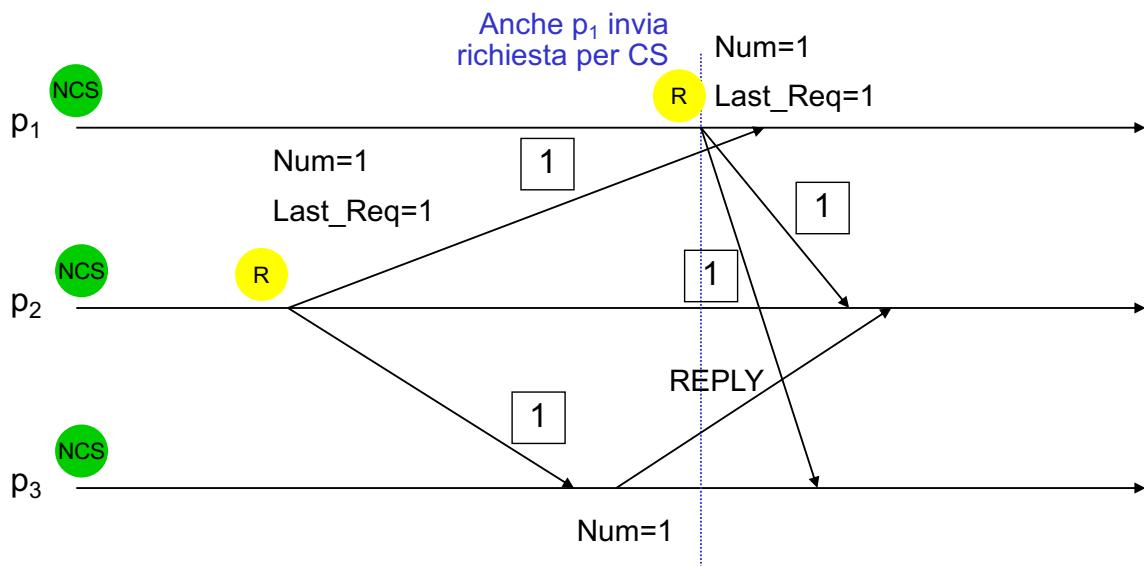
## Algoritmo di Ricart e Agrawala: esempio



## Algoritmo di Ricart e Agrawala: esempio

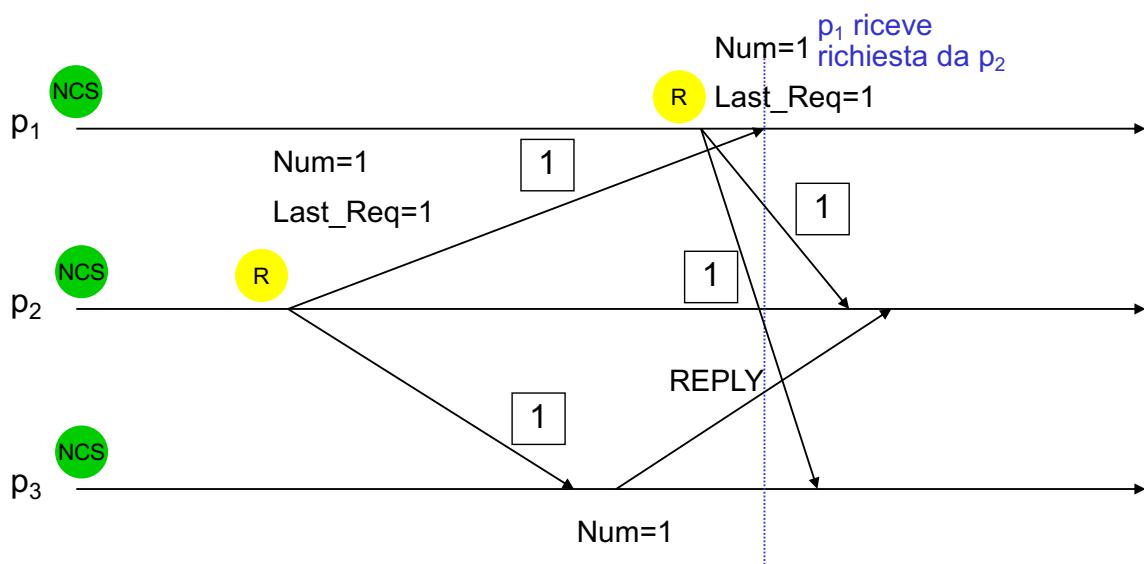


## Algoritmo di Ricart e Agrawala: esempio

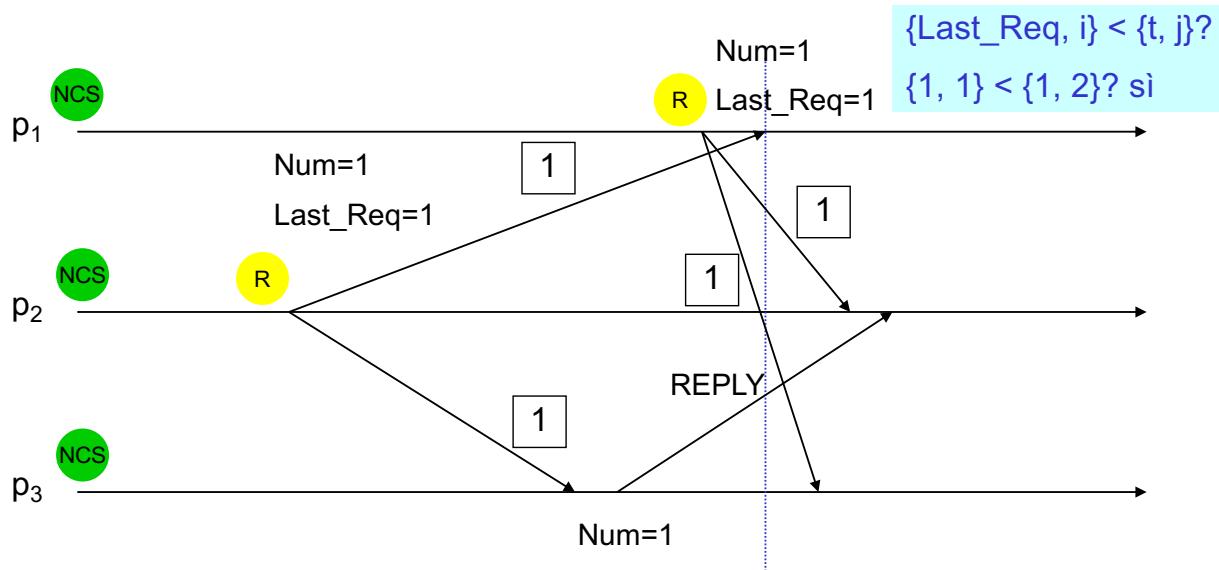


Anche p<sub>1</sub> vuole entrare in sezione critica (requesting), e manda suo msg di richiesta a p<sub>2</sub> e p<sub>3</sub>, p<sub>1</sub> ha Num=1 e Last\_Req=1

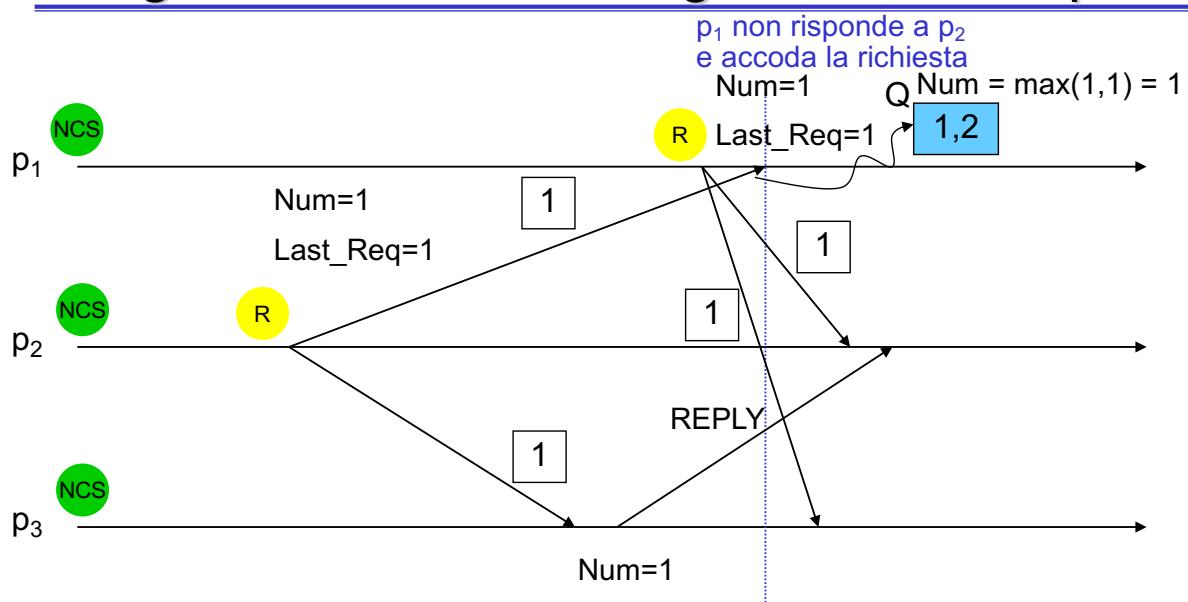
## Algoritmo di Ricart e Agrawala: esempio



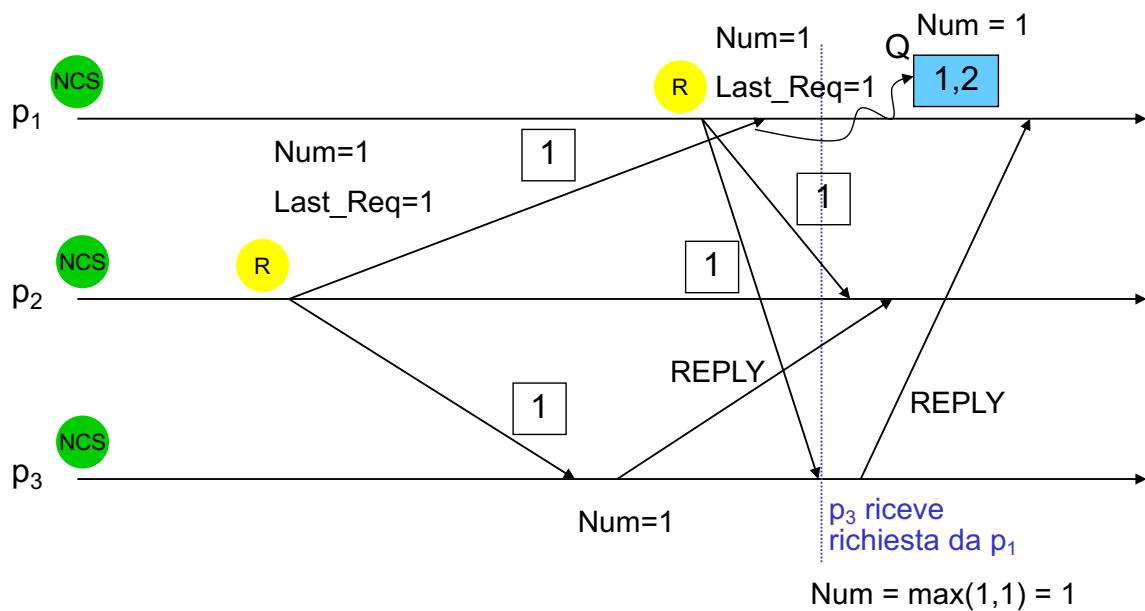
## Algoritmo di Ricart e Agrawala: esempio



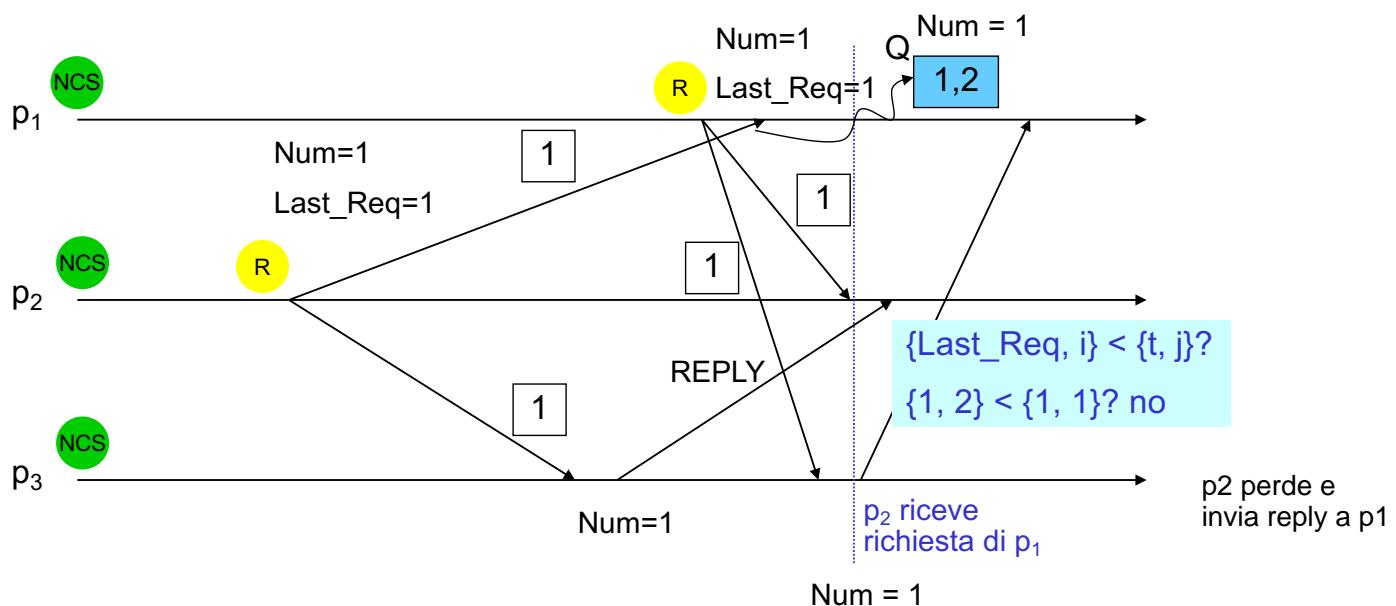
## Algoritmo di Ricart e Agrawala: esempio



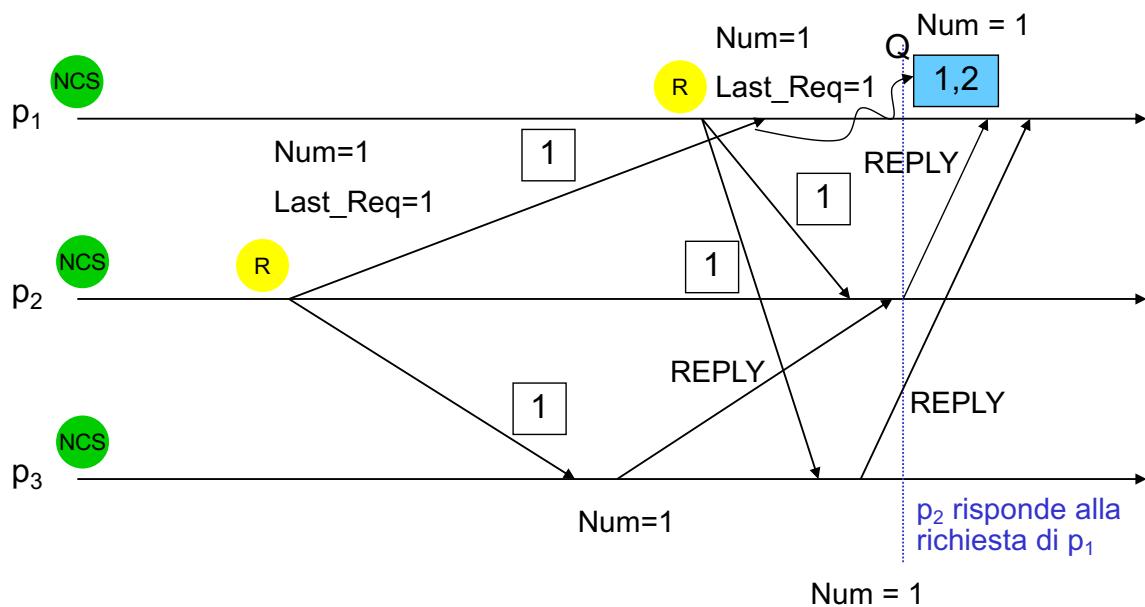
## Algoritmo di Ricart e Agrawala: esempio



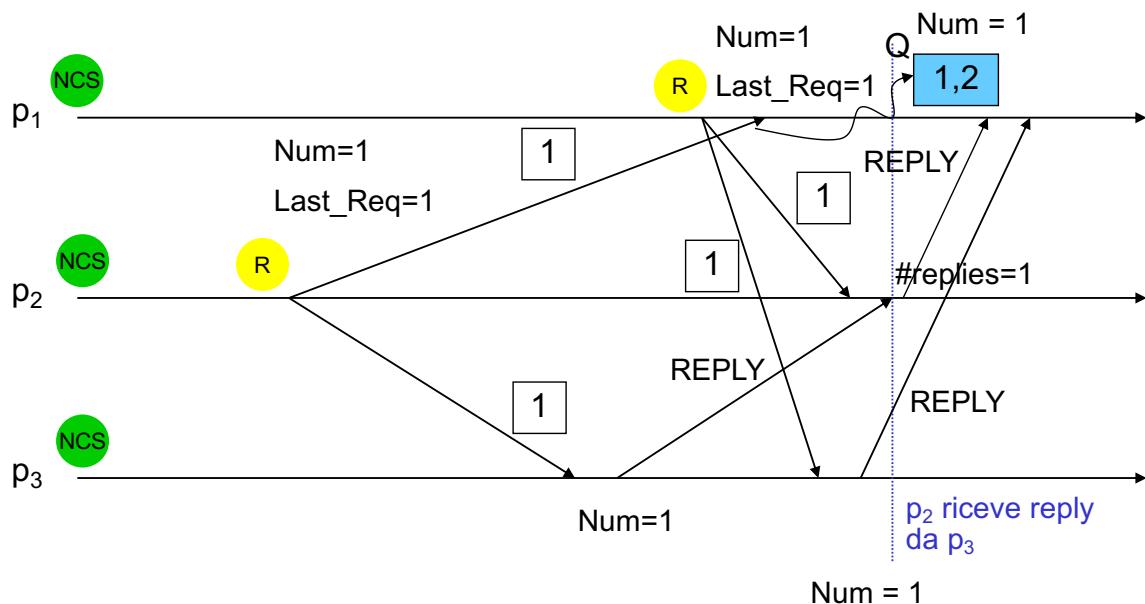
## Algoritmo di Ricart e Agrawala: esempio



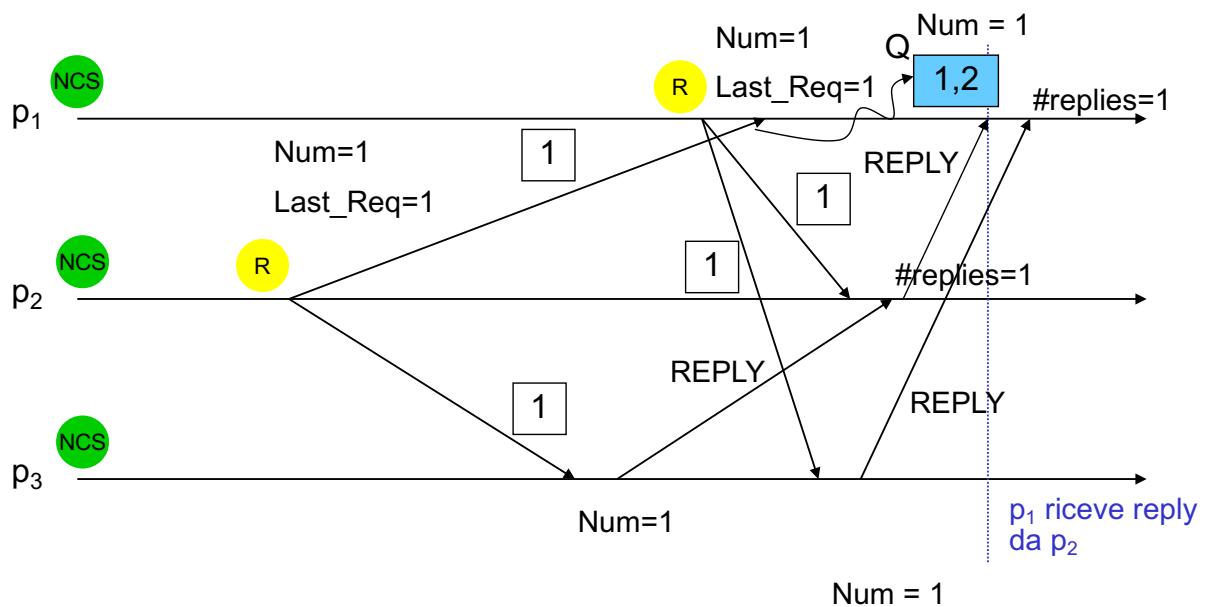
## Algoritmo di Ricart e Agrawala: esempio



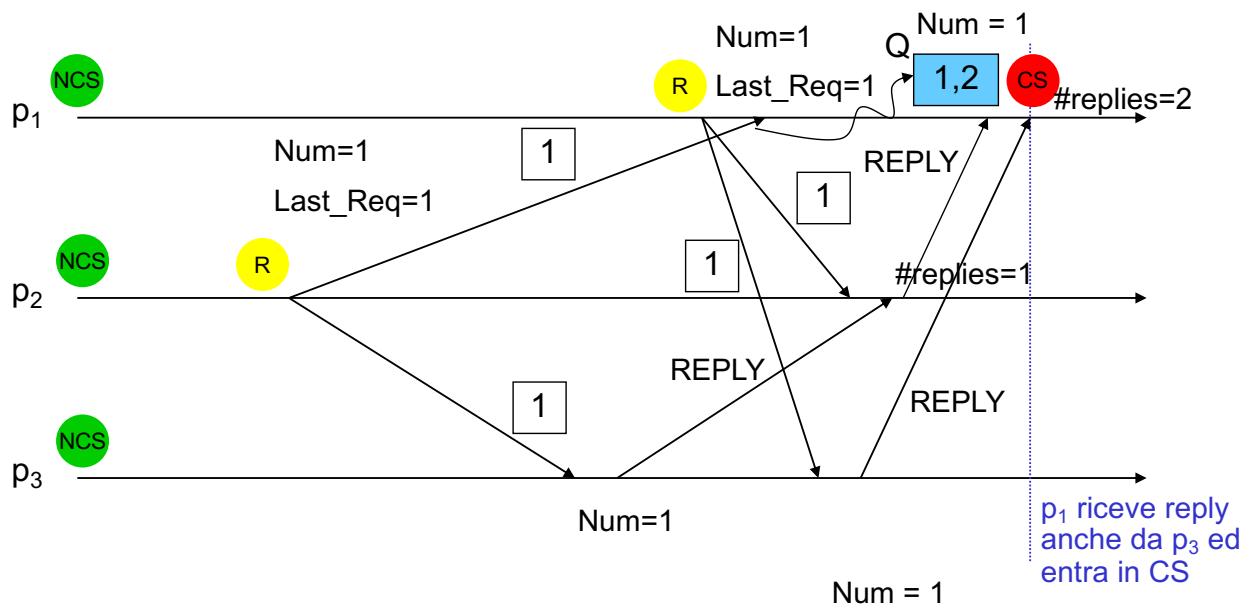
## Algoritmo di Ricart e Agrawala: esempio



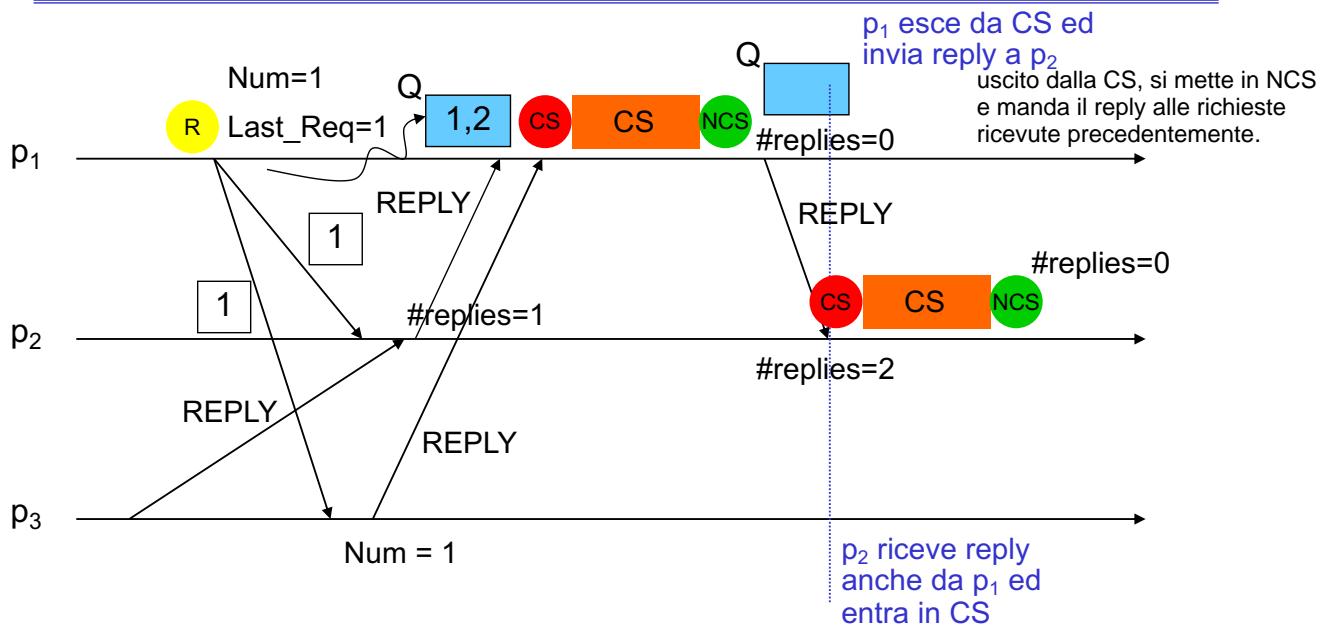
## Algoritmo di Ricart e Agrawala: esempio



## Algoritmo di Ricart e Agrawala: esempio



# Algoritmo di Ricart e Agrawala: esempio



# Algoritmo di Ricart e Agrawala

- **Vantaggi**
  - Come Lamport distribuito, è **completamente distribuito**
    - Nessun elemento centrale
  - Rispetto a Lamport distribuito, si risparmiano messaggi
    - **No messaggio di release**, msg di ack differito all'uscita della CS
    - Solo **2(N-1)** messaggi per accedere alla CS:  $N-1$  messaggi di richiesta, seguiti da  $N-1$  messaggi di reply
  - In letteratura ottimizzazioni (non esaminate) che riducono il numero di messaggi a  $N$
- **Svantaggi (come Lamport distribuito)**
  - Se un **processo fallisce**, nessun altro potrà entrare nella CS: occorre aggiungere un meccanismo di **failure detection**
  - Tutti i processi possono **essere collo di bottiglia**
    - Ogni processo partecipa ad ogni decisione

# Algoritmi basati su token

---

- Viene usata una risorsa ausiliaria, chiamata **token**
  - Token usato non solo per mutua esclusione, ma in diversi altri algoritmi distribuiti
- L'algoritmo deve definire: come vengono fatte le richieste per il token, mantenute e servite
- In un algoritmo basato su token in ogni istante esiste un **solo possessore di token**
  - Si garantisce così la **proprietà di safety** (mutua esclusione)
- Tipi di algoritmi basati su token:
  - **Centralizzato** (o approccio *token-asking*): gestione centralizzata del token da parte di un coordinatore
  - **Decentralizzato** (o approccio *perpetuum mobile*): gestione decentralizzata del token, che si muove nel sistema

## Algoritmo basato su token centralizzato

---

- Uno dei processi svolge il ruolo di **coordinatore** e gestisce il token
- Il coordinatore tiene traccia delle richieste, in particolare di:
  - Richieste ricevute ma non ancora servite
    - Richieste già servite
- Ogni processo  $p_i$  mantiene un suo **clock vettoriale**
- Quando  $p_i$  vuole entrare in CS invia un messaggio di richiesta al coordinatore avente come timestamp il proprio clock vettoriale
- Il coordinatore inserisce il msg nella lista delle richieste pendenti
- Quando la richiesta di  $p_i$  diventa eleggibile, il coordinatore invia il token a  $p_i$  che entrerà in CS
- Uscito dalla CS  $p_i$  restituisce il token al coordinatore

# Algoritmo basato su token centralizzato: coordinatore

---

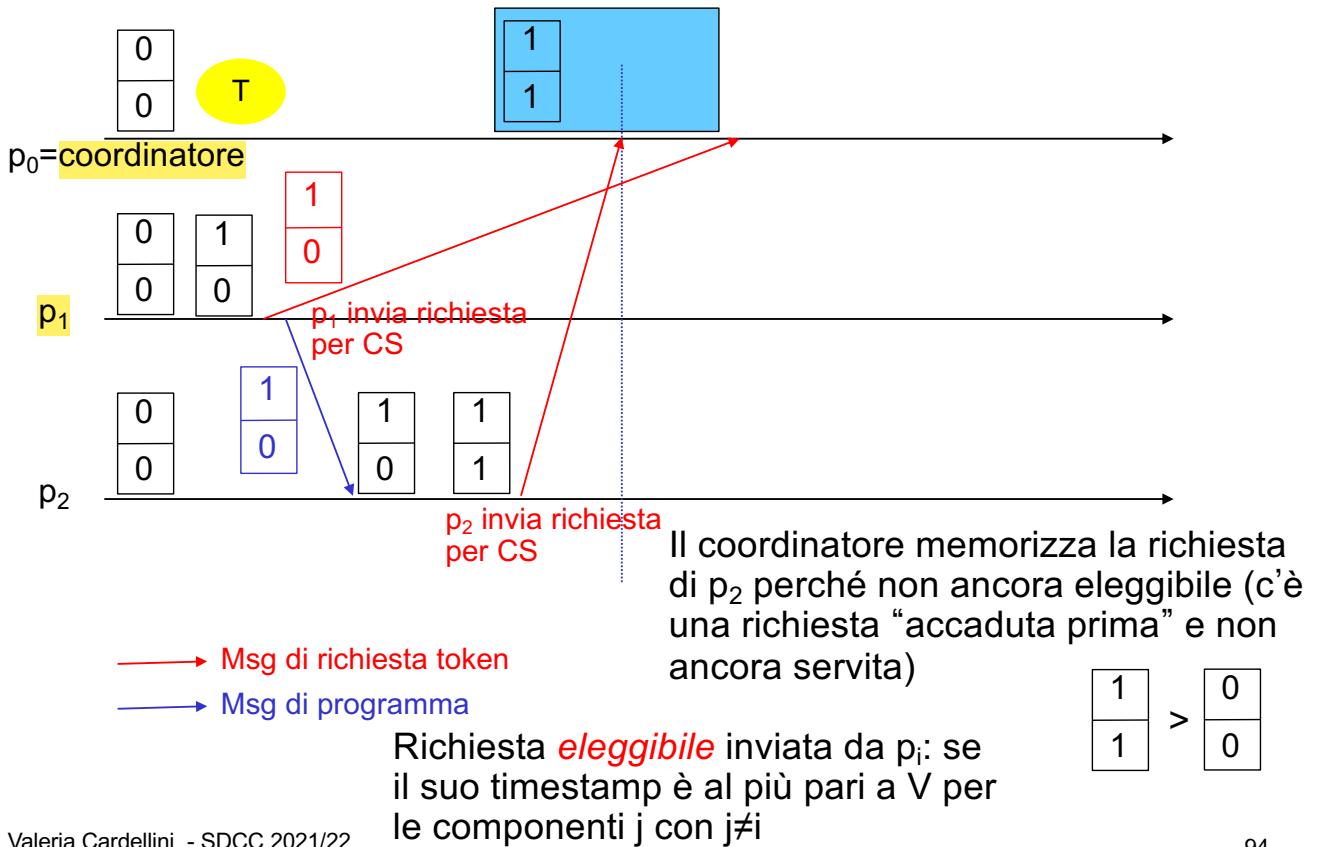
- Strutture dati del coordinatore:
  - Reqlist
    - **Lista delle richieste pendenti**, ricevute dal coordinatore ma non ancora servite
  - V
    - Array di dimensione pari al numero di processi
    - $V[i]$ : numero di richieste di  $p_i$  già servite
- Regole:
  - All'arrivo della richiesta di token da parte di  $p_i$  con timestamp  $T_{pi}$   
$$\text{Reqlist} = \text{Reqlist} \cup \{p_i, T_{pi}\}$$
  - Quando la richiesta di  $p_i$  diventa eleggibile ( $T_{pi} < V$ ) e il coordinatore ha il token  
    Invia token a  $p_i$

# Algoritmo basato su token centralizzato: processo

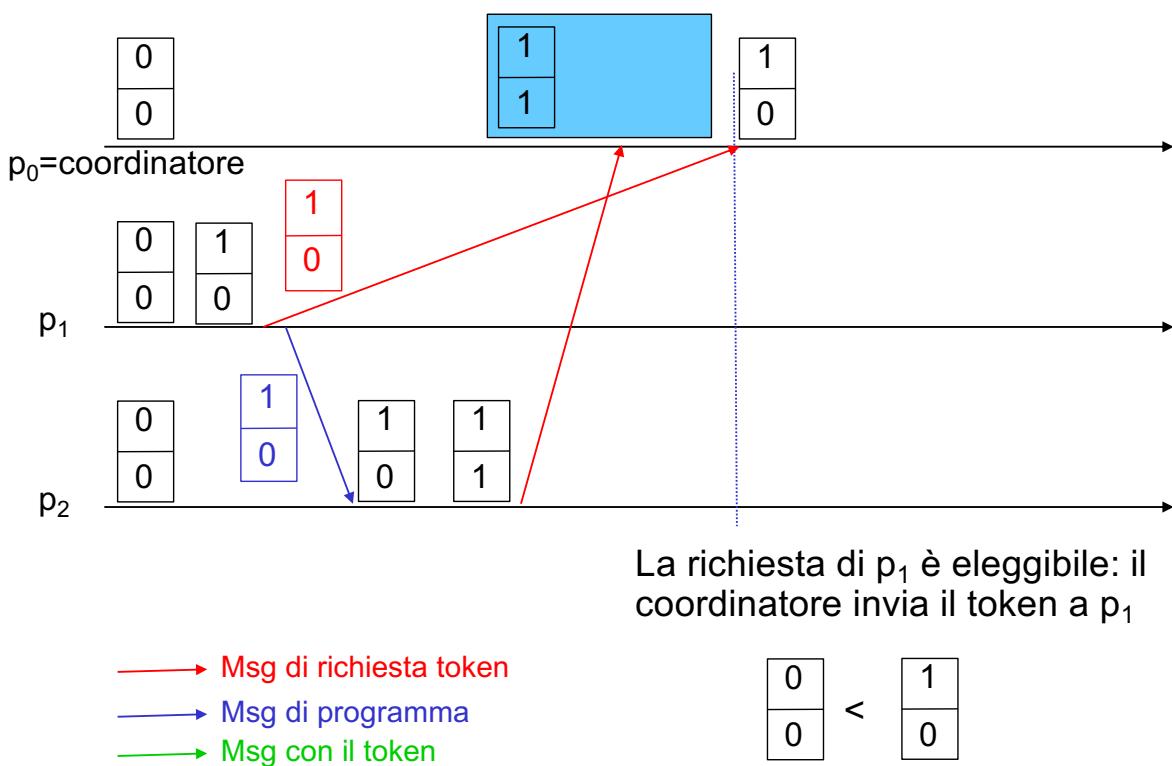
---

- Strutture dati di ogni processo:
  - VC
    - Clock vettoriale per il timestamp dei messaggi
- Regole:
  - All'invio della richiesta di token:  
$$VC[i] = VC[i] + 1$$
  
    Invia msg al coordinatore (con timestamp VC)
  - Alla ricezione del token  
    Entra nella CS
  - All'uscita dalla CS  
    Invia token al coordinatore
  - All'invio del **messaggio di programma**  
    Invia msg (con timestamp VC)
  - Alla ricezione del messaggio di programma  
    
$$VC = \max(VC, \text{msg.VC})$$

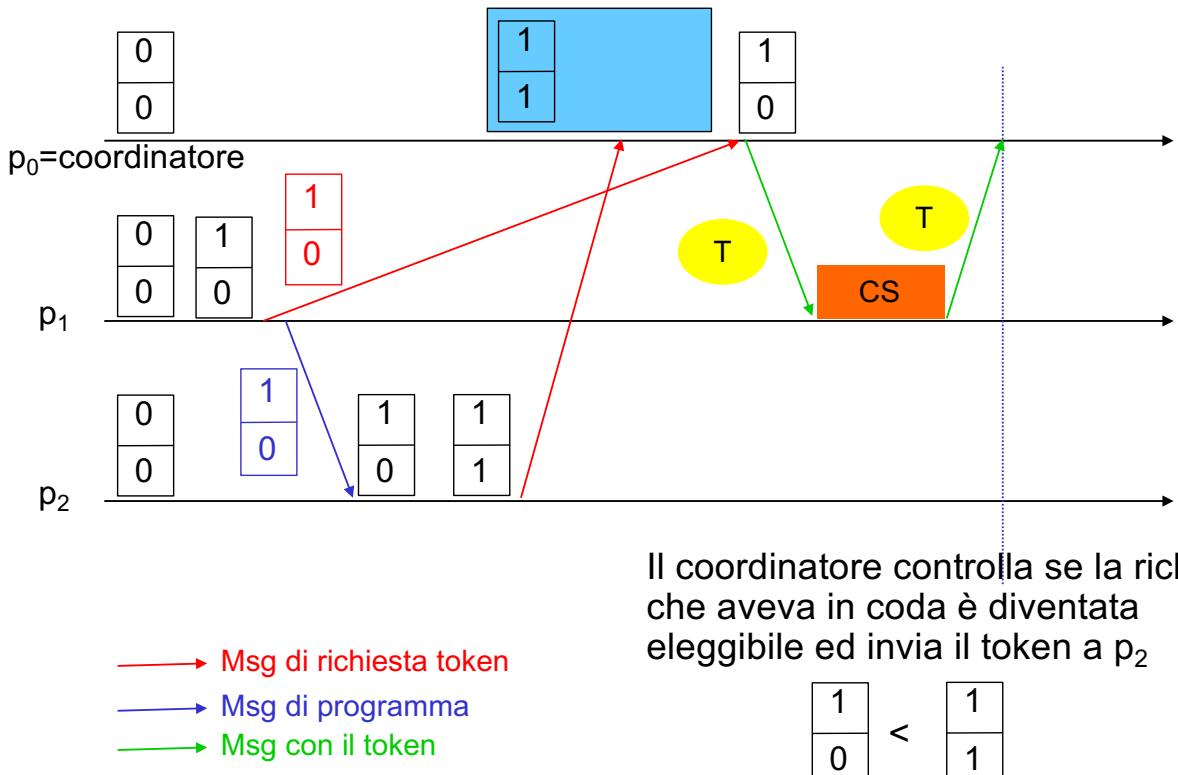
## Algoritmo basato su token centralizzato: esempio



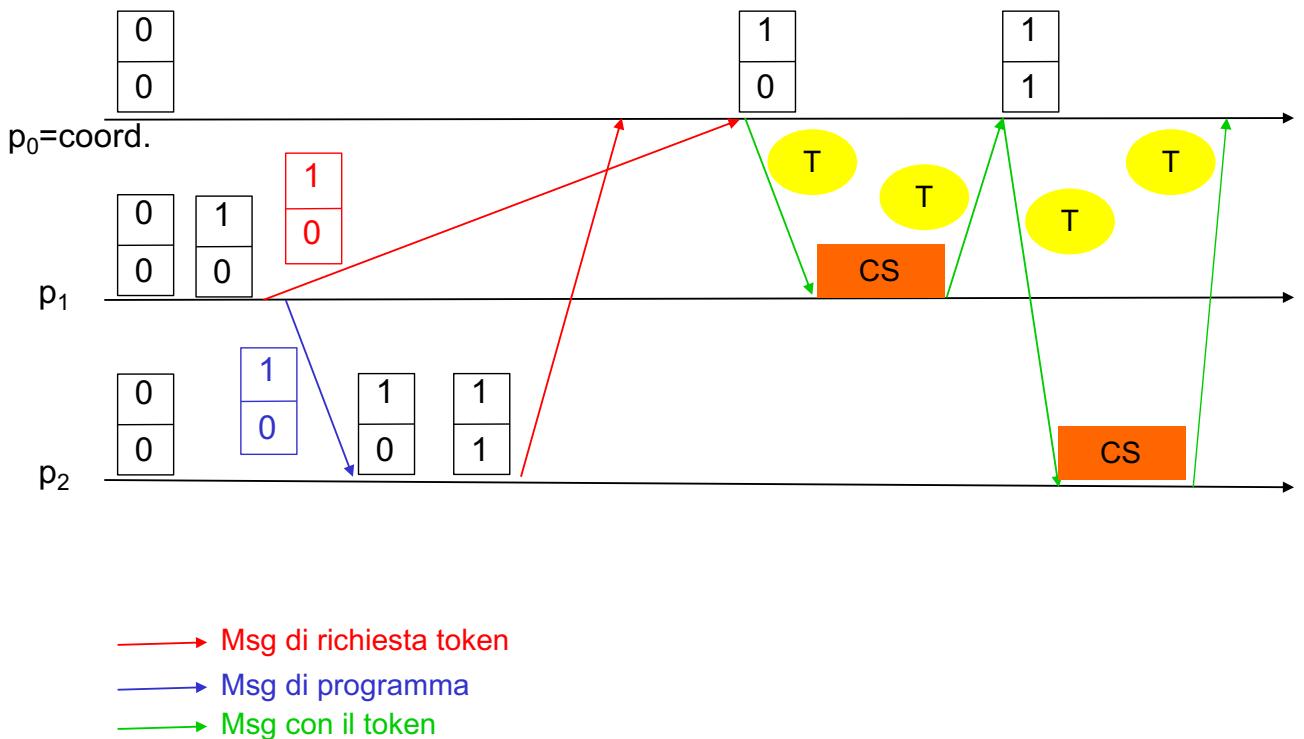
## Algoritmo basato su token centralizzato: esempio



# Algoritmo basato su token centralizzato: esempio



# Algoritmo basato su token centralizzato: esempio

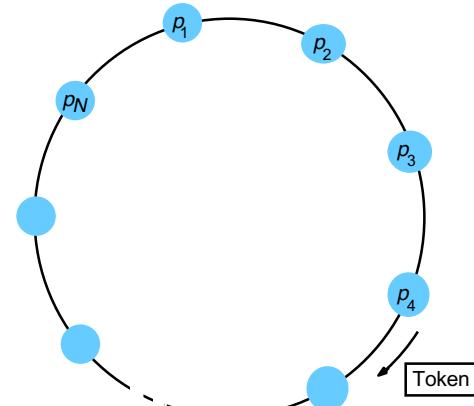


## Algoritmo basato su token centralizzato

- Vantaggi
  - Efficiente in termini di numero di msg scambiati con il coordinatore
    - Considerando anche i msg di programma?
  - Garantite anche proprietà di **fairness** e **ordering**
- Svantaggi
  - Coordinatore: SPoF e collo di bottiglia per scalabilità
  - In caso di crash del coordinatore occorre eleggere il nuovo coordinatore

## Algoritmo basato su token decentralizzato

- Processi organizzati logicamente ad anello (unidirezionale)
  - Nessuna relazione tra topologia dell'anello e interconnessione fisica tra i nodi
- Il token **viaggia da un processo all'altro**
  - Il token viene passato dal processo  $p_k$  al processo  $p_{(k+1) \bmod N}$
- Il processo in possesso del token può entrare in CS
- Se un processo riceve il token ma **non vuole entrare in CS, passa il token al processo successivo**



# Algoritmo basato su token decentralizzato

- Vantaggi
  - Se l'anello è unidirezionale, viene garantita anche la fairness
  - Rispetto al centralizzato, migliora il bilanciamento del carico
- Svantaggi
  - Consumo di banda di rete per trasmettere il token anche quando nessuno chiede l'accesso alla CS
  - In caso di perdita del token occorre rigenerarlo
    - Perdita anche per malfunzionamenti hw/sw
  - Crash dei singoli processi
    - Se un processo fallisce, occorre riconfigurare l'anello
    - Se fallisce il processo che possiede il token, occorre eleggere il prossimo processo che avrà il token
  - Guasti temporanei possono portare alla creazione di token multipli

11/01/2023

## Algoritmi basati su quorum

- Idea: per entrare in CS occorre raccogliere voti solo da un sottoinsieme di processi (**quorum**), non da tutti
- Votazione all'interno del sottoinsieme
  - I processi votano per stabilire chi è autorizzato ad entrare in CS
  - Un processo può votare per un solo processo
- **Insieme di votazione**  $V_i$ : sottoinsieme di  $\{p_1, \dots, p_N\}$ , associato ad ogni processo  $p_i$
- Un processo  $p_i$  per accedere alla CS
  - Invia *request* a tutti gli altri membri di  $V_i$
  - Attende *reply* da tutti i membri di  $V_i$
  - Ricevute tutte le *reply* dai membri di  $V_i$  usa la CS
  - Al rilascio della CS invia *release* a tutti gli altri membri di  $V_i$
- Un processo  $p_j$  in  $V_i$  che riceve *request*
  - Se è in CS o ha già risposto dopo aver ricevuto l'ultimo *release*, non risponde e accoda la richiesta
  - Altrimenti risponde subito con *reply*
- Un processo che riceve *release* estrae **una** richiesta dalla coda e invia *reply*

# Algoritmo di Maekawa

---

- Ogni processo  $p_i$  esegue il seguente algoritmo:

## Inizializzazione

```
state = RELEASED;  
voted = FALSE;
```

## Protocollo di ingresso in CS per $p_i$

```
state = WANTED;  
invia in multicast request a tutti i processi in  $V_i$  (incluso se stesso);  
wait until (numero di reply ricevute =  $K$ );  
state = HELD;
```

## Alla ricezione di *request* da $p_j$ ( $i \neq j$ )

```
if (state = HELD or voted = TRUE) then  
    accoda request da  $p_j$  senza rispondere;  
else  
    invia reply a  $p_j$ ; // vota a favore di  $p_j$   
    voted = TRUE;  
end if
```

Valeria Cardellini - SDCC 2021/22

102

# Algoritmo di Maekawa

---

## Protocollo di uscita da CS per $p_i$

```
state = RELEASED;  
invia in multicast release a tutti i processi in  $V_i$ ;  
if (coda di richieste non vuota) then  
    estrai la prima richiesta in coda, ad es. da  $p_k$ ;  
    invia reply a  $p_k$ ; // vota a favore di  $p_k$   
    voted = TRUE;  
else  
    voted = FALSE;  
end if
```

## Alla ricezione di un messaggio di release da $p_j$ ( $i \neq j$ )

```
if (coda di richieste non vuota) then  
    estrai la prima richiesta in coda, ad es. da  $p_k$ ;  
    invia reply a  $p_k$ ;  
    voted = TRUE;  
else  
    voted = FALSE;  
end if
```

## Algoritmo di Maekawa: insieme di votazione

- Come è definito l'insieme di votazione  $V_i$  per  $p_i$ ?

1.  $V_i \cap V_j \neq \emptyset \quad \forall i, j$

- Insiemi di votazione ad intersezione non nulla

2.  $|V_i| = K \quad \forall i$

- Tutti i processi hanno insiemi di votazione con stessa cardinalità  $K$  (stesso sforzo per ogni processo)

3. Ogni processo  $p_i$  è contenuto in  $K$  insiemi di votazione

- Stessa *responsabilità* per ogni processo

4.  $p_i \in V_i$

- Per ridurre il numero di messaggi trasmessi

• Si dimostra che la soluzione ottima che **minimizza  $K$**  è  $K \approx \sqrt{N}$

- Essendo  $N = K(K-1) + 1$

Valeria Cardellini - SDCC 2021/22

Esempio:  $N=7$

$V_1=\{1,2,3\}$
$V_4=\{1,4,5\}$
$V_6=\{1,6,7\}$
$V_2=\{2,4,6\}$
$V_5=\{2,5,7\}$
$V_7=\{3,4,7\}$
$V_3=\{3,5,6\}$

104

## Algoritmo di Maekawa

- Come scegliere quali sono i processi a cui inviare la richiesta?

- Si costruiscono gli insiemi di votazione in modo tale che abbiano intersezione non vuota
- Se un quorum accorda l'accesso in CS ad un processo, **nessun altro quorum potrà accordare lo stesso permesso**

- Proprietà

- Soddisfa la proprietà di safety ma non di **liveness**
  - Si può verificare deadlock
  - Si può rendere l'algoritmo deadlock-free con messaggi aggiuntivi

- Prestazioni

- Per accedere alla CS occorrono un numero di messaggi pari a  **$3\sqrt{N}$**  ( $2\sqrt{N}$  per ingresso in CS e  $\sqrt{N}$  per uscita dalla CS)
  - Soluzione migliore di Ricart-Agrawala per reti a larga scala essendo  $3\sqrt{N}$  minore di  $2(N-1)$  per  $N > 4$

Valeria Cardellini - SDCC 2021/22

105

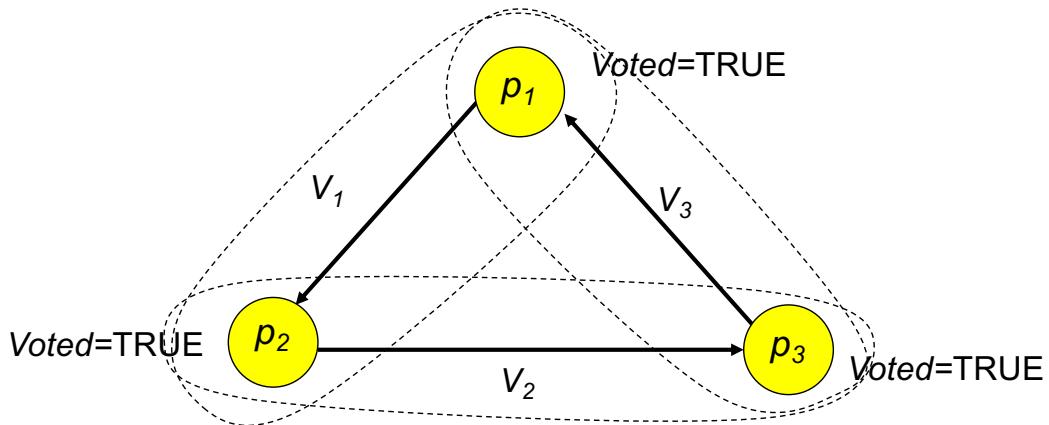
# Algoritmo di Maekawa: esempio di deadlock

$$P = \{p_1, p_2, p_3\}$$

$$V_1 = \{p_1, p_2\}, V_2 = \{p_2, p_3\}, V_3 = \{p_3, p_1\}$$

## Situazione di deadlock

- $p_1, p_2$  e  $p_3$  richiedono contemporaneamente l'accesso in CS
- $p_1, p_2$  e  $p_3$  impostano ognuno  $voted=TRUE$  ed aspettano la risposta dell'altro



## Confronto tra algoritmi per ME distribuita

Algoritmo	#msg per ingresso/uscita in/da CS	#msg prima dell'ingresso in CS	Problemi
Autorizzazione centralizzata	3	2	Crash del coordinatore
Ricart-Agrawala	$2(N-1)$	$2(N-1)$	Crash di un qualunque processo
Token centralizzato	$3 + (N-1)$ inclusi msg di programma	$2 + (N-1)$ inclusi msg di programma	Crash del coordinatore
Token decentralizzato	Da 1 a $\infty$ (se anello bidirezionale)	Da 0 a $N-1$	Perdita del token Crash di un qualunque processo
Maekawa	$3\sqrt{N}$	$2\sqrt{N}$	Possibile deadlock

# Algoritmi di elezione distribuita

---

- Molti algoritmi distribuiti richiedono che un processo agisca da **coordinatore** o **leader**
  - Sequencer nel multicast totalmente ordinato
  - Coordinatore nella mutua esclusione
- Come eleggere il coordinatore in modo **dinamico**?
  - Eleggere un coordinatore richiede **accordo distribuito**
  - L'obiettivo dell'algoritmo di elezione è assicurarne la terminazione con l'accordo di tutti i partecipanti
- Algoritmi esaminati
  - Algoritmo bully (del prepotente)
  - Algoritmo ad anello di Fredrickson & Lynch

## Elezione distribuita: modello del sistema

---

- Consideriamo un sistema con  $N$  processi  $p_i$ ,  $i = 1, \dots, N$
- I processi sono **soggetti a fallimenti di tipo crash**
- La comunicazione è affidabile
- Un processo non indice più di un'elezione alla volta
- Ogni processo ha un id univoco e viene eletto il processo non guasto con l'id più elevato

# Distributed election: properties

---

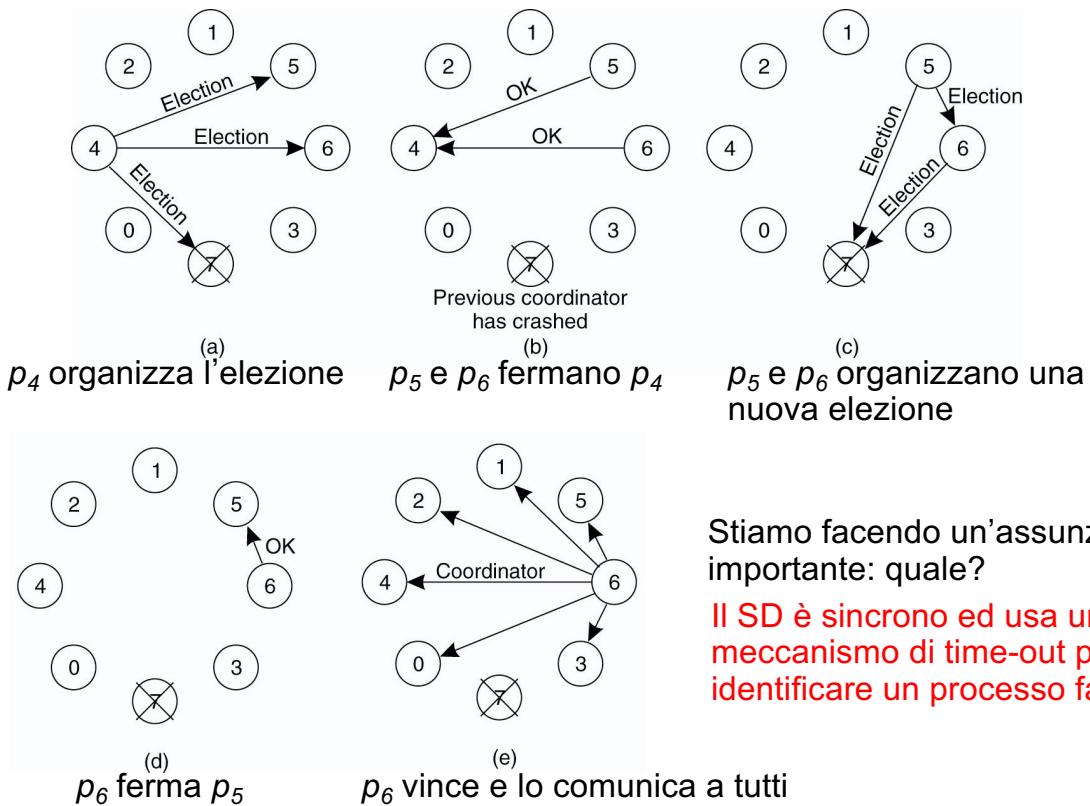
- **Safety**: only the non-faulty process with the highest id is elected as leader
  - The election result does not depend on the process that started the election
  - When multiple processes start an election at the same time, only a single winner will be announced
- **Liveness**: at any time, some process is eventually leader

## Algoritmo bully

---

- Idea: “Node with highest ID bullies his way into leadership” (Garcia-Molina)
- Il processo  $p_i$  che rileva l’assenza del coordinatore promuove una nuova elezione
  - $p_i$  invia un messaggio ELEZIONE a tutti i processi con id maggiore del suo
  - Se nessuno risponde,  $p_i$  si autoprolama vincitore, invia un messaggio di proclamazione a tutti i processi con id minore del suo e diventa il nuovo coordinatore
  - Se  $p_k$  (con  $k > i$ ) riceve il messaggio ELEZIONE da  $p_i$  (che ha id minore del suo) risponde con il messaggio OK e indice una nuova elezione
  - Se  $p_i$  riceve un messaggio OK, abbandona il suo proposito di diventare il nuovo coordinatore
- L’algoritmo designa sempre come coordinatore il processo non-faulty con id più alto
- Un processo appena (ri-)creato non conosce il coordinatore e dunque promuove un’elezione

## Algoritmo bully: esempio



Valeria Cardellini - SDCC 2021/22

112

Stiamo facendo un'assunzione importante: quale?

Il SD è sincrono ed usa un meccanismo di time-out per identificare un processo faulty

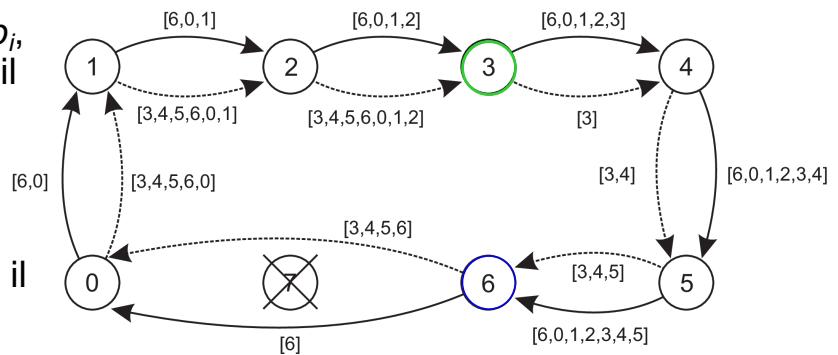
## Bully algorithm: communication cost

- Which is the communication cost in terms of exchanged messages?
- **Best case:** the process with the second highest identifier notices the coordinator's failure
  - It can immediately select itself as coordinator and then send  $N-2$  coordinator messages  $\Rightarrow O(N)$  messages
- **Worst case** (assuming no process failure during election): the process with the lowest identifier initiates the election
  - It sends  $N-1$  election messages to processes, which themselves initiate each one an election  $((N-1) + (N-2) + \dots + 1) + N-1 \Rightarrow O(N^2)$  messages

# Algoritmo ad anello di Fredrickson & Lynch

- Processi organizzati logicamente in **anello (unidirezionale)**
  - Ogni processo conosce almeno il suo successore
- Il processo  $p_i$  che rileva l'assenza del coordinatore promuove una nuova elezione
  - $p_i$  invia un **messaggio di ELEZIONE** a  $p_{(i+1) \bmod N}$
  - Se  $p_{(i+1) \bmod N}$  è faulty,  $p_i$  lo salta e passa al successivo lungo l'anello, finché non trova un processo non-faulty
  - Ad ogni passo, il processo che riceve il messaggio di ELEZIONE vi aggiunge il suo id e lo inoltra
- Al termine, il messaggio di ELEZIONE ritorna a  $p_i$ , che identifica nella lista il processo con id più alto ed invia sull'anello un **messaggio di COORDINATORE** per informare tutti su chi sia il nuovo coordinatore

Valeria Cardellini - SDCC 2021/22



114

## Election algorithms: cost

- Communication cost as number of required messages
- Bully:
  - $O(N^2)$  in the worst case
  - $O(N)$  in the best case
- Fredrickson & Lynch ring:
  - $O(2N)$
  - But requires larger messages than Bully

# Election algorithms: properties

---

- Assumption common to all election algorithms
  - **Communication is reliable** (messages are neither dropped nor corrupted)
- Ring election algorithm:
  - Works both for synchronous and asynchronous communications
  - **Works for any N** and does not require any process to know how many processes are in the ring
- Fault tolerance with respect to process failure
  - What happens if a process crashes during the election? It depends on the algorithm and the crashed process
  - Additional mechanisms may be needed, e.g., ring reconfiguration
- Something to consider:
  - What happens in case of **network partition**?
  - Multiple nodes may decide they are the new leader

