Università degli Studi di Roma "Tor Vergata"
Facoltà di Ingegneria

Sincronizzazione nei Sistemi Distribuiti
(parte 2)

Corso di Sistemi Distribuiti

Valeria Cardellini

Anno accademico 2009/10

Mutua esclusione e sistemi concorrenti

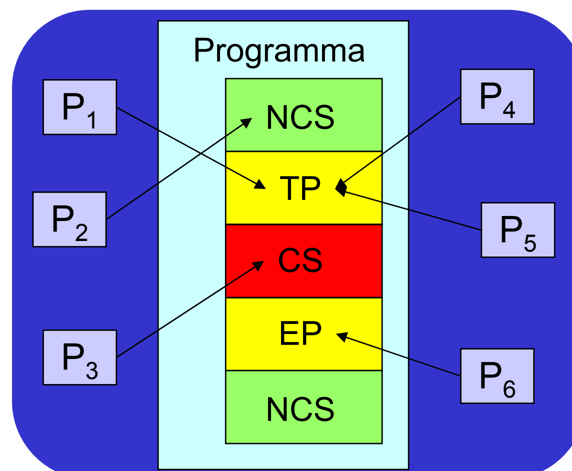
- La mutua esclusione nasce nei sistemi concorrenti
  - *N* processi vogliono accedere ad una risorsa condivisa
  - Ogni processo vuole acquisire la risorsa ed utilizzarla in modo esclusivo senza avere interferenze con gli altri processi
- Ogni algoritmo di mutua esclusione comprende
  - Una sequenza 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)**
- Uno scheduler sceglie di volta in volta a quale processo consentire l'esecuzione della prossima istruzione
  - La sequenza decisa dallo scheduler e comprendente tutte le istruzioni dei diversi processi è chiamata schedule

Mutua esclusione e sistemi concorrenti (2)

- Il problema è specificato attraverso le seguenti tre proprietà:
  - **Mutua esclusione (ME)** o ***safety*** (no azioni sbagliate): al più un processo per volta è nella sezione critica
  - **No deadlock (ND)**: se un processo rimane bloccato nella sua trying section, esistono uno o più processi che riescono ad entrare ed uscire dalla sezione critica
  - **No starvation (NS)** o *posticipazione indefinita*: nessun processo può rimanere bloccato per sempre nella trying section
- Osservazione:
  - NS implica ND
  - NS, oltre ad implicare una proprietà di ***liveness*** (no ritardi indefiniti), specifica un comportamento paritetico dei vari processi (***fairness***)

Mutua esclusione e sistemi concorrenti (3)

- 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
  - **Algoritmo del panificio di Lamport** (1974)
    - Ideato per sistemi multiprocessore



Algoritmo di Dijkstra

- Modello di sistema concorrente di Dijkstra
  - I processi comunicano leggendo e scrivendo variabili condivise
  - La lettura e scrittura di una variabile sono operazioni atomiche
  - Non ci sono assunzioni sul tempo che ogni processo impiega ad eseguire un'azione atomica
- Variabili condivise
  - b[1,· · ·,N] : array di booleani inizializzati a true
  - c[1,· · ·,N] : array di booleani inizializzati a true
  - k : intero inizializzato in modo casuale
- Variabili locali
  - j : intero

Algoritmo di Dijkstra (2)

- Ogni processo p_i esegue il seguente algoritmo:

trying protocol

ciclo della sentinella

```
Li0:  b[i] = false       // p_i si prenota
Li1:  while (k ≠ i)      // la sentinella k seleziona uno
                         // dei processi prenotati
Li2:     c[i] = true     // p_i cancella una sua eventuale occupazione
         if (b[k] = true) then k = i
Li4:  c[i] = false       // p_i occupa
      for j = 1 to N do      // p_i controlla se ci sono conflitti:
                             // almeno un altro processo che ha occupato
         if (j ≠ i and c[j] = false) then goto Li1
      // sezione critica
      c[i] = true
      b[i] = true
      // Altre istruzioni
      goto Li0:
```

exit protocol

Un possibile schedule - 4 processi, k=4

p_1	p_2	p_3
b[1] = false		
while (k ≠ 1)		
c[1] = true		
	b[2] = false	
	while (k ≠ 2)	
	c[2] = true	
		b[3] = false
		while (k ≠ 3)
		c[3] = true
if (b[4] = true)		
	if (b[4] = true)	
		if (b[4] = true)
k=1		
	k = 2	
		k = 3

Un possibile schedule - 4 processi, k=4 (2)

- A p_4 lo scheduler non concede di eseguire nessuna istruzione
- Alla fine di questa parte di schedule la situazione è la seguente:
  - La prossima istruzione da eseguire per tutti i processi (tranne p_4) è l'istruzione Li1
  - Il valore di k=3
  - A questo punto p_3 occupa (istruzione Li4), gli altri non occupano e vedono che p_3 è comunque prenotato, quindi lasciano inalterato il valore di k
  - p_3 entra in sezione critica (vedi prossimo lucido) perché non rileva conflitti (altri processi che occupano la variabile c). Quando p_3 esce dalla sezione critica, rilascia sia le variabili c e b: p_1 e p_2 rimasti nella trying section possono provare a passare il ciclo della sentinella

Un possibile schedule - 4 processi, k=4 (3)

p_1	p_2	p_3
		c[3] = false
		for j = 1 to N do
		if (j ≠ 3 and c[j] = false)
		then goto Li1
		Sezione critica
		b[3] = true
		c[3] = true

- Da questo schedule sembra che il primo processo che passa il ciclo della sentinella è anche il primo ad entrare in sezione critica
- Ciò accade solo se non ci sono conflitti (nessun processo che ha contemporaneamente occupato)
- In caso di conflitto l'ordine di entrata in sezione critica lo decide lo scheduler

Un possibile schedule - 4 processi, k=4 (4)

p_1	p_2	p_3
while (k ≠ 1)		
c[1] = true		
	while (k ≠ 2)	
	c[2] = true	
if (b[3] = true) *(sì, p_3 uscito da CS)*		
	if (b[3] = true) *(sì, p_3 uscito da CS)*	
then k=1		
while (k ≠ 1)		
c[1] = false *(occupa)*		
	then k=2	
	while (k ≠ 2)	
	c[2] = false *(occupa)*	
if (j ≠ i and c[j] = false) then goto Li1		
	if (j ≠ i and c[j] = false) then goto Li1	

- In caso di conflitto (almeno due processi che occupano allo stesso tempo), si torna indietro e si cancellano le occupazioni, ma uno di loro la mantiene, in particolare quello che per ultimo aveva posto il suo nome su k (p_2)

Algoritmo di Dijkstra (3)

- L'algoritmo di Dijkstra soddisfa le proprietà di
  - Mutua esclusione
  - No deadlock
- Tuttavia, non soddisfa la proprietà di starvation

Algoritmo del panificio di Lamport

- Soluzione semplice ispirata ad una situazione reale
  - L'attesa di essere serviti in un panificio
- Modello di Lamport
  - I processi comunicano leggendo/scrivendo variabili condivise
  - La lettura e la scrittura di una variabile non sono operazioni atomiche; un processo potrebbe 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 sono non correlate

Algoritmo del panificio di Lamport (2)

- Variabili condivise
  - num[1,· · ·,N] : array di interi inizializzati a 0
  - choosing[1,· · ·,N] : array di booleani inizializzati a *false*
- Variabili locali
  - j: intero compreso in [1, …,N]

Algoritmo del panificio di Lamport (3)

- 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
for j = 1 to N do
    // busy waiting mentre j sta scegliendo
    while choosing[j] do NoOp();
    // busy waiting finchè il valore del biglietto non è il più basso
    // meccanismo che favorisce 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
```

Algoritmo del panificio di Lamport (4)

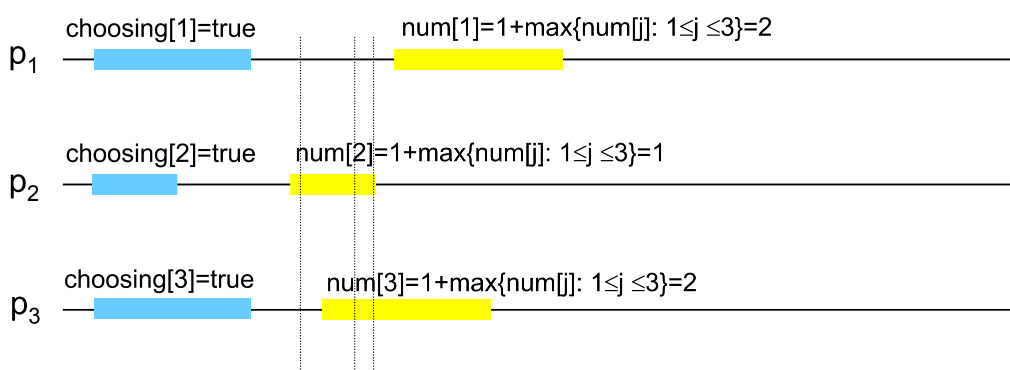
- Ciclo ripetuto all'infinito dal processo p_i

```
// sezione non critica
// prende un biglietto                                    doorway
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                                  bakery
for j = 1 to N do
    // busy waiting mentre j sta scegliendo
    while choosing[j] do NoOp();
    // busy waiting finchè il valore del biglietto non è il più basso
    // meccanismo che favorisce 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 $b < d$

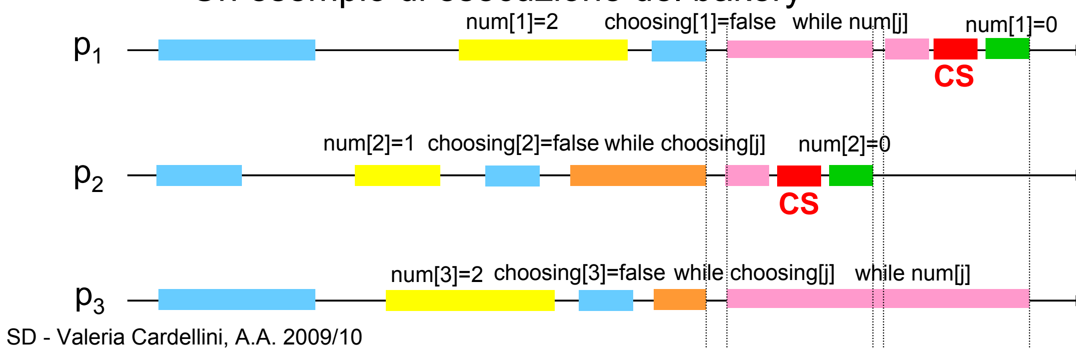
Algoritmo del panificio di Lamport (5)

- 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 dagli altri più 1
  - Altri processi possono accedere concorrentemente alla doorway
  - Un esempio di esecuzione della doorway



Algoritmo del panificio di Lamport (6)

- Bakery
  - Ogni processo deve controllare che tra i processi in attesa lui sia il prossimo ad avere accesso alla CS
  - Il primo ciclo permette a tutti i processi nella doorway di terminare la loro scelta del biglietto
  - Il secondo ciclo lascia un processo 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 sono risolti basandosi sull'identificativo del processo
  - Un esempio di esecuzione del bakery



Algoritmo del panificio di Lamport (7)

- 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 nei SD

- Rispetto al modello di Lamport aggiungiamo il vincolo:
  - Un processo non può leggere direttamente il valore di una variabile di proprietà degli altri processi, ma deve esplicitamente inviare un messaggio ed attendere una risposta contenente il valore
- La comunicazione tra processi avviene in base alle seguenti assunzioni:
  - I processi comunicano leggendo e scrivendo variabili attraverso scambio di messaggi
  - Il ritardo di trasmissione di un messaggio è impredicibile ma finito
  - I canali di comunicazione sono affidabili
    - Un messaggio inviato viene ricevuto correttamente dal suo destinatario
    - Inoltre, non ci sono duplicazioni o messaggi spuri (ricevuti ma mai trasmessi)

Adattamento dell'algoritmo del panificio

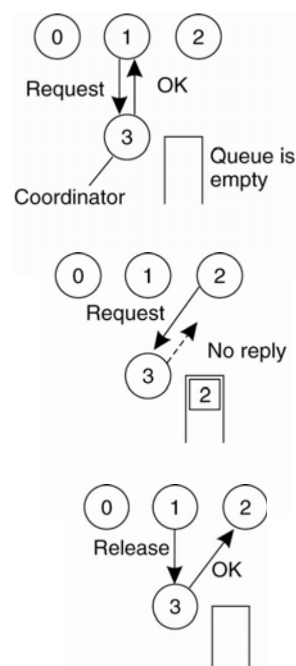
- Proviamo ad adattare ai SD l'algoritmo del panificio di Lamport ideato per sistemi concorrenti
  - Ogni processo p_i si comporta da server rispetto alle proprie variabili num[i] e choosing[i]
  - Comunicazione basata non su memoria condivisa ma su scambio di messaggi
    - Ogni processo legge i valori locali agli altri processi tramite messaggi di richiesta-risposta
- Non è una buona soluzione
  - Per ogni "lettura" vengono scambiati $2N$ messaggi
  - Per accedere alla CS servono 3 scambi di messaggi (1 per doorway, 2 per bakery) con un costo totale di $6N$ messaggi
  - La latenza per un singolo scambio di messaggi è uguale al tempo impiegato all'accoppiata canale di comunicazione-processo più lenti
  - Quindi: scarsa efficienza e scalabilità
  - Problema derivante 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

Autorizzazioni: algoritmo centralizzato

- La richiesta di accesso ("ENTER") ad una risorsa in mutua esclusione viene inviata ad un 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 non risponde nel caso di SD sincrono
- Il processo che rilascia la risorsa ne informa il coordinatore ("RELEASED")
- Il coordinatore preleva la prima richiesta in attesa e invia "GRANTED" al suo mittente



Autorizzazioni: algoritmo centralizzato (2)

- Vantaggi
  - Garantisce la mutua esclusione
  - No starvation
  - Semplice e facile da implementare
    - Solo 3 messaggi (richiesta, risposta e rilascio) per ogni accesso alla CS
- Svantaggi
  - Il coordinatore è il singolo point of failure dell'algoritmo
  - Il coordinatore può diventare il collo di bottiglia per le prestazioni

Algoritmo di Lamport distribuito

- Ogni processo mantiene un clock logico scalare ed una coda (per memorizzare le richieste di accesso alla CS)
- Regole dell'algoritmo:
  - Richiesta di CS: p_i manda un messaggio di richiesta con timestamp a tutti gli altri processi e aggiunge la coppia <richiesta, timestamp> alla sua coda locale
  - Ricezione di una richiesta: la richiesta (con il suo timestamp) viene memorizzata nella coda locale ed un messaggio di ack è inviato al mittente
  - Accesso di p_i alla CS se e solo se:
    - p_i ha una richiesta in coda con timestamp t
    - t è il timestamp più piccolo tra quelli presenti in coda
    - p_i ha già ricevuto da ogni altro processo un messaggio di ack con timestamp più grande di t
  - Rilascio della CS: p_i manda un messaggio di release a tutti gli altri ed elimina la richiesta dalla coda
  - Ricezione di release: la richiesta è eliminata dalla coda

Algoritmo di Lamport distribuito (2)

- L'algoritmo rispetta le proprietà per la mutua esclusione (safety, liveness, fairness)
- Tuttavia 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

- Richiede che ci sia un ordinamento totale di tutti gli eventi del sistema
  - Basato su clock logici (estensione di Lamport senza release)

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

- Un processo che riceve può
  - non essere nella CS richiesta e non vuole accedervi → manda REPLY al mittente
  - essere nella CS → non risponde e mette il messaggio in coda locale
  - voler accedere alla CS → confronta i *timestamp* e vince quello minore: se è l'altro invia REPLY, se è lui non risponde e pone il messaggio in coda

Algoritmo di Ricart e Agrawala (2)

- Variabili locali per ciascun processo
  - #replies (inizializzata a 0)
  - State ∈ {Requesting, CS, NCS} (inizializzato a NCS)
  - Q coda di richieste pendenti (inizialmente vuota)
  - Last_Req (inizializzata a 0)
  - Num (inizializzata a 0)
- 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 pj;
4. Wait until #replies=N-1;
5. State = CS;
6. CS
7. ∀ r∈Q send REPLY to r
8. Q=∅; State=NCS; #replies=0;

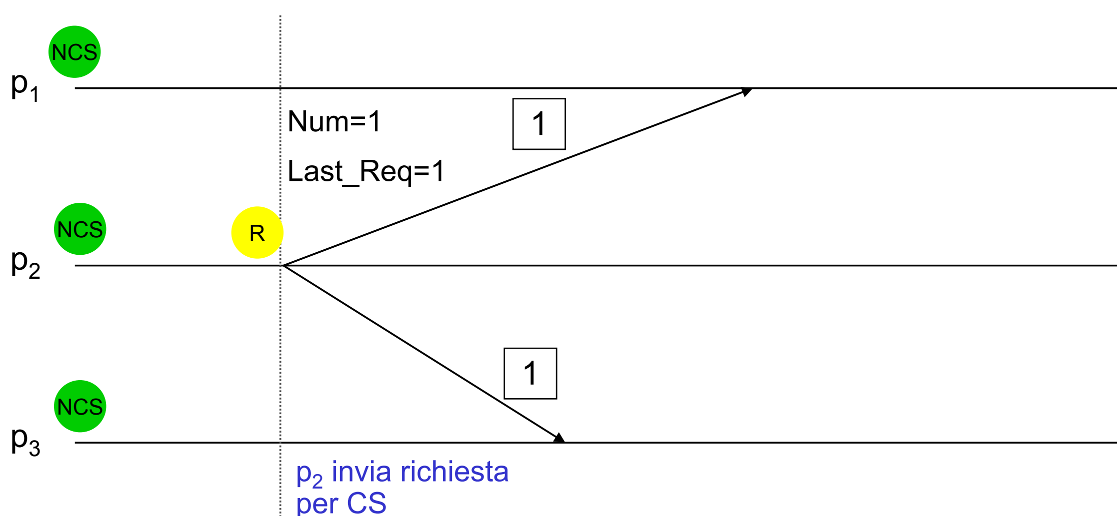
Upon receipt REQUEST(t) from pj

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 pj
4. Num = max(t, Num)

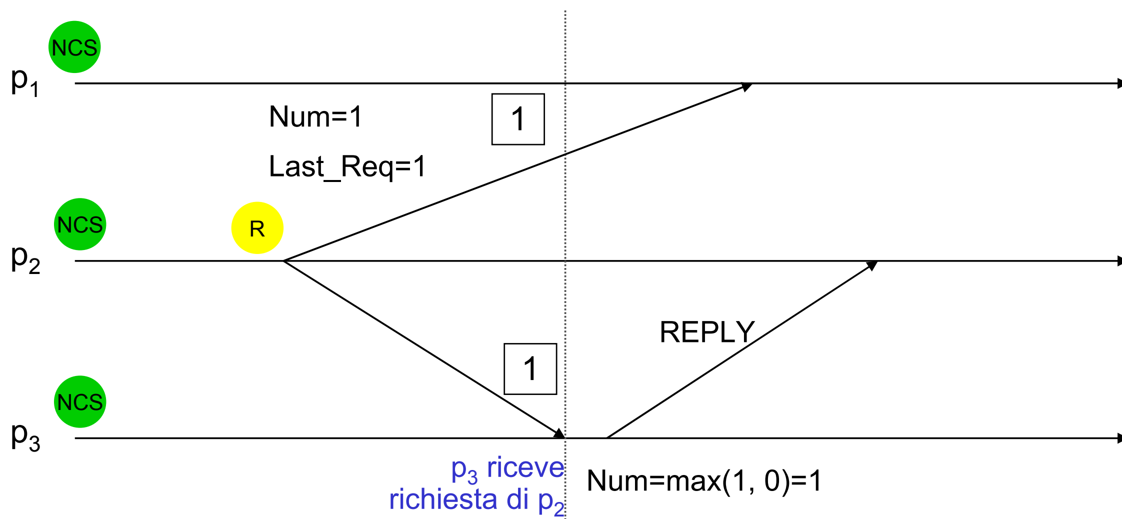
Upon receipt REPLY from pj

1. #replies = #replies+1

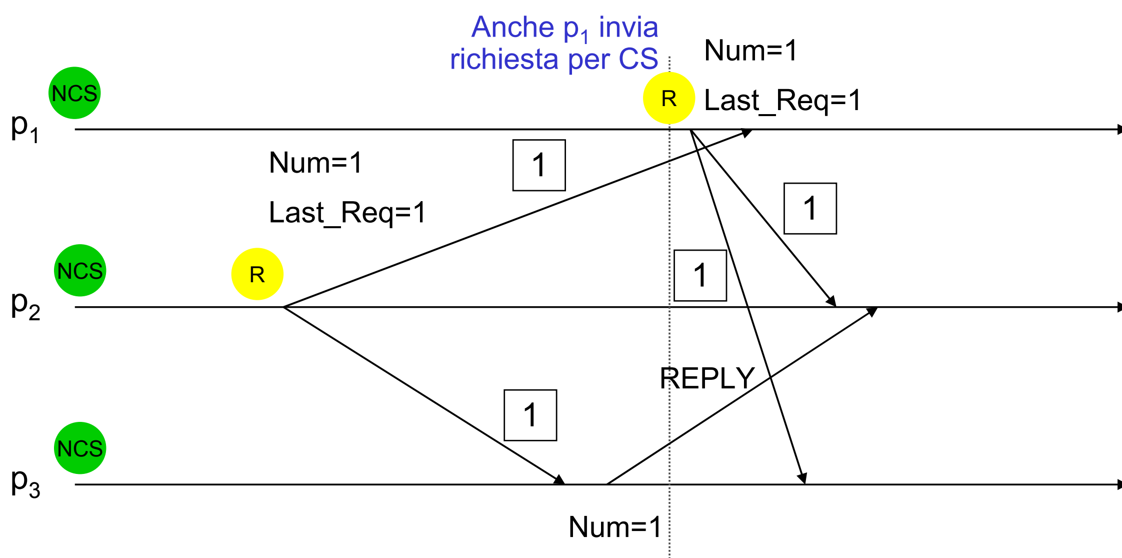
Algoritmo di Ricart e Agrawala: esempio



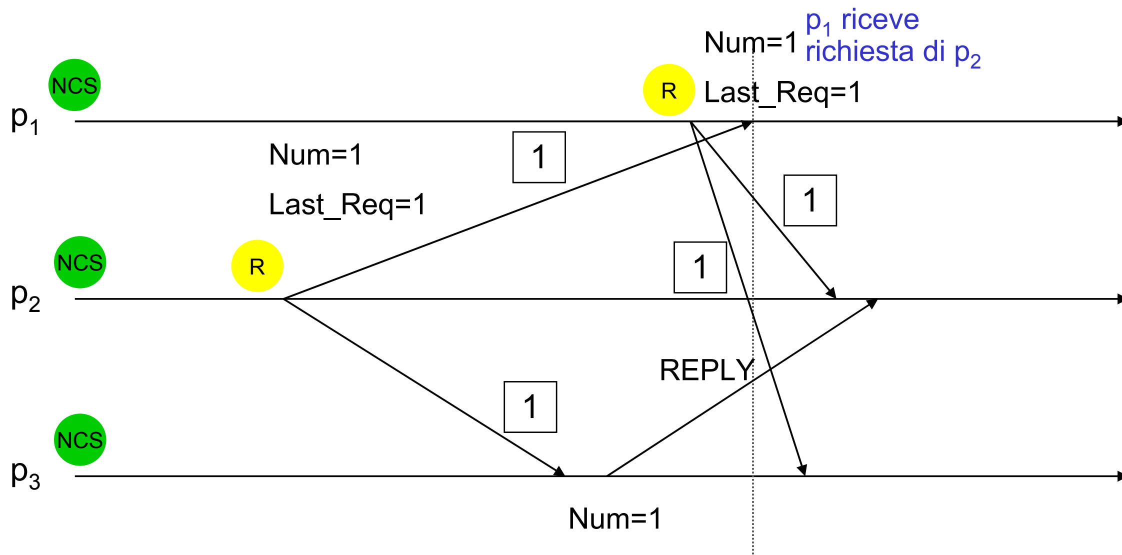
Algoritmo di Ricart e Agrawala: esempio



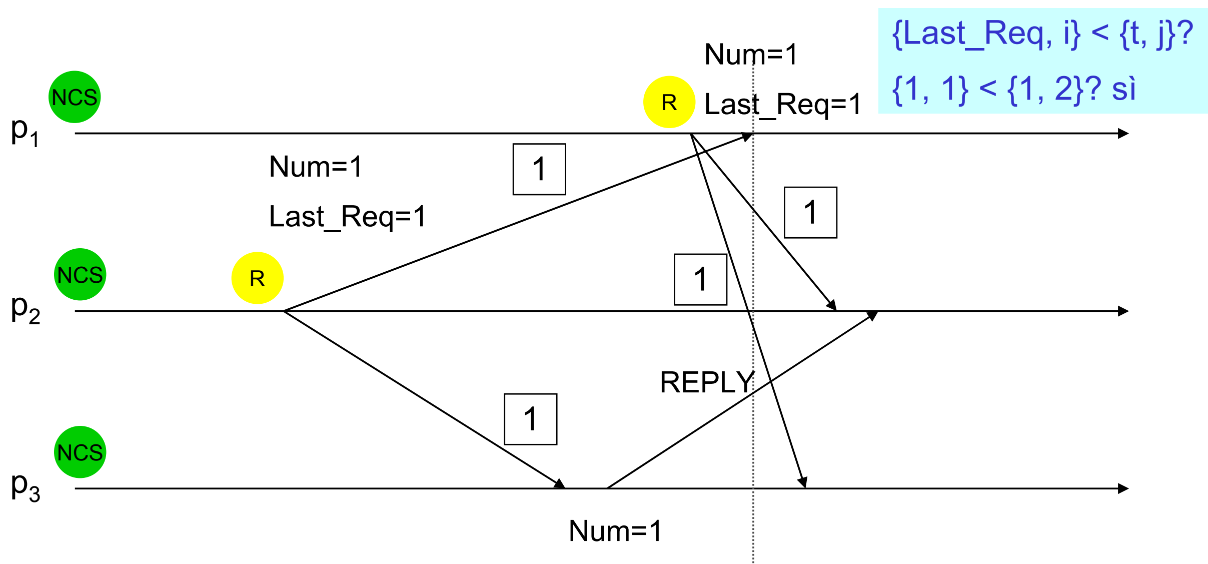
Algoritmo di Ricart e Agrawala: esempio



Algoritmo di Ricart e Agrawala: esempio



Algoritmo di Ricart e Agrawala: esempio



Algoritmo di Ricart e Agrawala: esempio

p_1 NCS R Num=1 Last_Req=1 Q 1,2 Num = max(1,1) = 1

p_2 NCS R Num=1 Last_Req=1

p_3 NCS Num=1

1 1 1 1 REPLY

Algoritmo di Ricart e Agrawala: esempio

p_1 NCS R Num=1 Last_Req=1 Q 1,2 Num = 1

p_2 NCS R Num=1 Last_Req=1

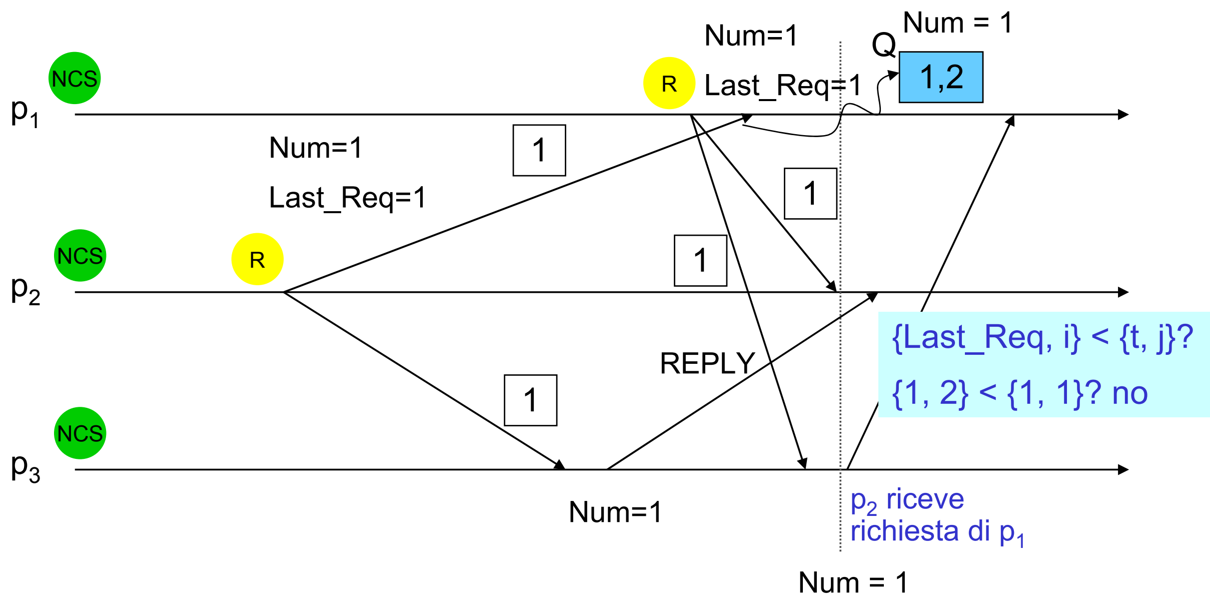
p_3 NCS Num=1

1 1 1 1 REPLY REPLY

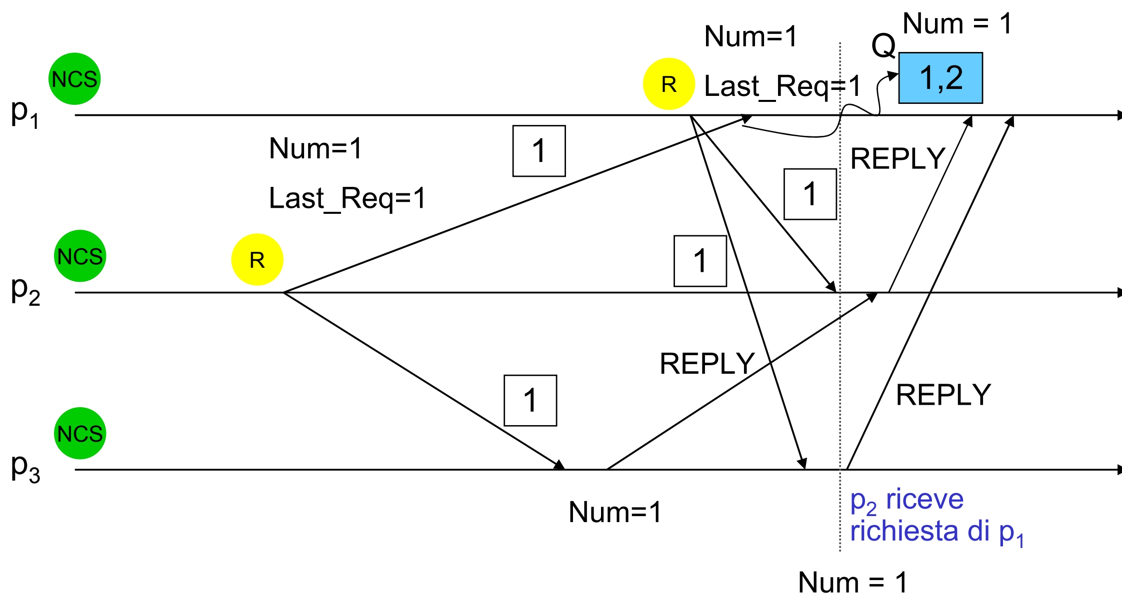
p_3 riceve richiesta di p_1

Num = max(1,1) = 1

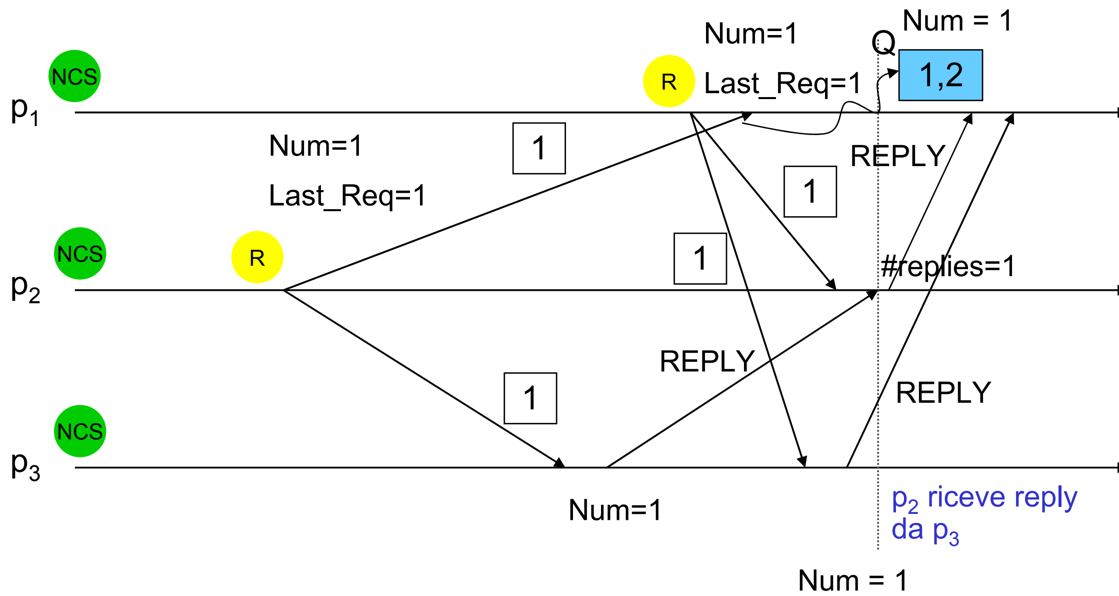
Algoritmo di Ricart e Agrawala: esempio



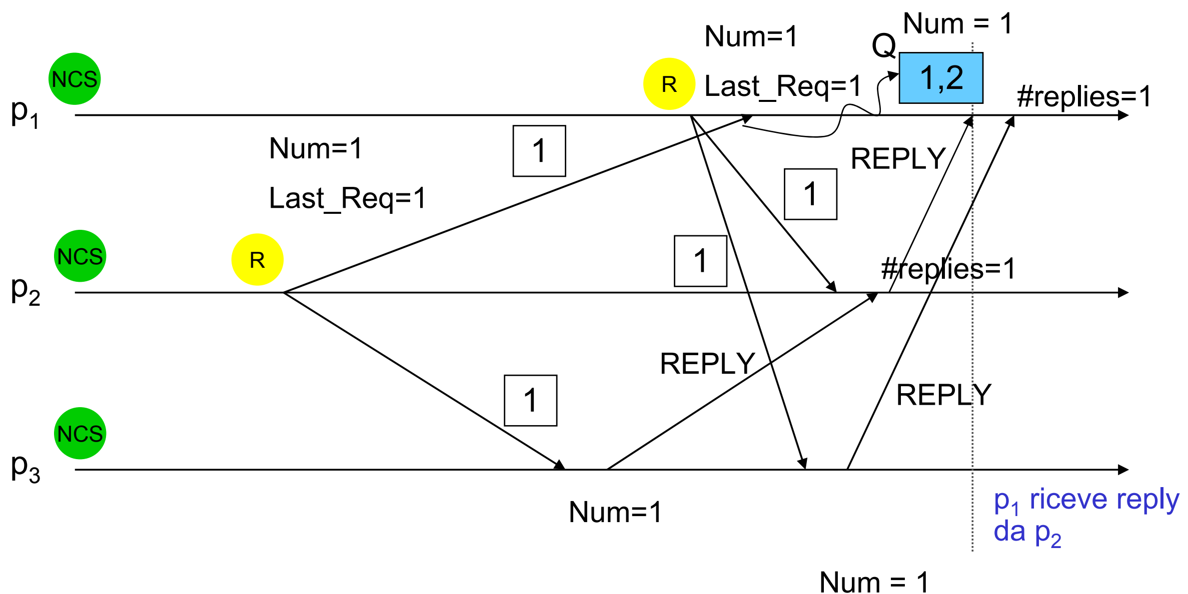
Algoritmo di Ricart e Agrawala: esempio



Algoritmo di Ricart e Agrawala: esempio



Algoritmo di Ricart e Agrawala: esempio



Algoritmo di Ricart e Agrawala: esempio

p_1 riceve reply anche da p_3 ed entra in CS

Algoritmo di Ricart e Agrawala: esempio

p_2 riceve reply anche da p_1 ed entra in CS

Algoritmo di Ricart e Agrawala (3)

- Vantaggi
  - L'algoritmo è completamente distribuito
    - Nessun elemento centrale
  - Rispetto a Lamport si risparmiano messaggi
    - Il messaggio di reply è inviato solo quando tutte le richieste che precedono quella corrente sono state servite
    - Quindi Ricart-Agrawala permette di usare solo 2(*N*-1) messaggi per accedere alla CS
- Svantaggi
  - Se un processo fallisce nessun altro potrà entrare nella CS
  - Tutti i processi possono essere collo di bottiglia
    - Ogni processo partecipa ad ogni decisione
  - Come Lamport, assume che la comunicazione sia affidabile e FIFO ordered

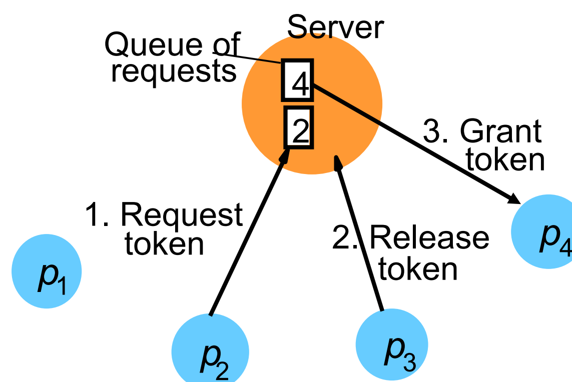
Algoritmi basati su token

- Viene usata una risorsa ausiliaria, chiamata **token**
  - L'uso di token non è utile solo per la mutua esclusione: molti problemi distribuiti vengono risolti con questa tecnica
- 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
  - Il meccanismo garantisce la safety della mutua esclusione
- Tipi di algoritmi basati su token:
  - *Centralizzato* (o approccio *token-asking*): esiste un processo coordinatore unico, responsabile di gestire il token
  - *Decentralizzato* (o approccio *perpetuum mobile*): il token si muove nel sistema e porta con sé tutte le strutture dati richieste dal coordinatore nel caso centralizzato

Algoritmo basato su token centralizzato

- Esiste un processo che svolge il ruolo di *coordinatore* e che gestisce il token
- Il coordinatore tiene traccia delle richieste, in particolare di:
  - Richieste effettuate ma non ancora servite
  - Richieste già servite
- Ogni processo p_i mantiene un suo clock vettoriale in cui sono riportati i timestamp delle richieste
- Quando p_i vuole entrare in CS invia una richiesta al coordinatore contenente il proprio clock vettoriale
- Il coordinatore ne prende nota e lo inserisce 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

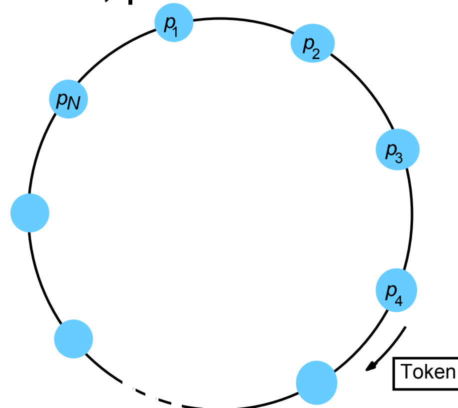
Algoritmo basato su token centralizzato (2)



- Vantaggi
  - L'algoritmo è efficiente in termini di numero di messaggi scambiati (solo 3) per accedere alla CS
  - Viene garantita anche la fairness
- Svantaggi
  - Il coordinatore rappresenta il collo di bottiglia del sistema ed è un singolo point of failure

Algoritmo basato su token decentralizzato

- Il token viaggia da un processo all'altro
- I processi sono organizzati in un anello logico
  - Il token viene passato dal processo p_k al processo $p_{(k+1) mod\ N}$
- Il processo che ha il *token* è abilitato all'accesso alla CS
- Se un processo riceve il token ma non ha necessità di accedere alla CS, passa il token lungo l'anello



Algoritmo basato su token decentralizzato (2)

- Vantaggi
  - Se l'anello è unidirezionale, viene garantita anche la fairness
  - Rispetto al centralizzato, migliora il bilanciamento del carico
- Svantaggi
  - Il token si può perdere; in questo caso, occorre rigenerarlo
    - Perdita di token anche per malfunzionamenti hw/sw
  - Soggetto al crash dei singoli processi
    - Se un processo fallisce occorre riconfigurare l'anello logico
    - Se fallisce il processo che possiede il token occorre eleggere il prossimo processo che avrà il token
  - Rischio e attenzione ai guasti temporanei che possono portare alla creazione di token multipli
  - L'algoritmo usa sempre banda per trasmettere il token anche quando nessuno chiede l'accesso alla CS

Algoritmi basati su quorum

- Per entrare in una CS occorre sincronizzarsi solo con il sottoinsieme dei processi interessati
- Algoritmi di votazione all'interno del sottoinsieme
  - I processi votano per stabilire chi è autorizzato ad entrare in CS
- Insieme di votazione V_i: sottoinsieme di $\{p_1,\ldots, p_N\}$, associato ad ogni processo p_i

- Un processo p_i per accedere alla CS
  - Invia una richiesta a tutti gli altri membri di V_i
  - Attende le risposte
  - Ricevute tutte le risposte dai membri di V_i usa la CS
  - Al rilascio della CS invia un *release* a tutti gli altri membri di V_i

- Un processo p_j in V_i che riceve la richiesta
  - Se è nella CS o ha già risposto dopo aver ricevuto l'ultimo *release* non risponde e accoda la richiesta
  - Altrimenti risponde subito con un *reply*

- Un processo che riceve un *release* estrae una richiesta dalla coda e invia un *reply*

Algoritmo di Maekawa

- Ogni processo p_i esegue il seguente algoritmo:

Inizializzazione

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

Protocollo di ingresso nella sezione critica per p_i

```
state = WANTED;
Multicast request a tutti i processi in Vi – {pi};
Wait until (numero di risposte ricevute = (K – 1));
state = HELD;
```

Alla ricezione di una richiesta da p_j ($i \neq j$)

```
if (state = HELD or voted = TRUE)
then
        accoda request da pj senza rispondere;
else
        invia reply a pj;
        voted = TRUE;
end if
```

Algoritmo di Maekawa (2)

Protocollo di uscita dalla sezione per p_i

```
state = RELEASED;
Multicast release a tutti i processi in V_i – {p_i};
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
```

Alla ricezione di un messaggio 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
```

- L'algoritmo garantisce la safety ma non l'assenza di deadlock
  - Si può rendere l'algoritmo deadlock-free con messaggi aggiuntivi

Algoritmo di Maekawa (3)

- Come è definito l'insieme di votazione V_i per p_i?
- $V_i \cap V_j \neq \emptyset \ \forall \ i, j$
  - Insiemi di votazione ad intersezione non nulla
- $|V_i| = K \ \forall \ i$
  - Tutti i processi hanno insiemi di votazione con stessa cardinalità (stesso *sforzo* per ogni processo)
- Ogni processo p_i è contenuto in M insiemi di votazione
  - Stessa *responsabilità* per ogni processo
- $p_i \in V_i$
  - Per minimizzare la trasmissione dei messaggi
- Si dimostra che la soluzione ottimale che minimizza K è $K \approx \sqrt{N}$
  - Essendo N = K(K-1) + 1

S1={1,2}
S3={1,3}
S2={2,3}

S1={1,2,3}
S4={1,4,5}
S6={1,6,7}
S2={2,4,6}
S5={2,5,7}
S7={3,4,7}
S3={3,5,6}

Algoritmo di Maekawa (4)

- Problema: come scegliere chi sono i processi a cui inviare la richiesta?
  - Si costruiscono dei sottoinsiemi in modo tale che tra di loro abbiano intersezione non vuota
  - Se un quorum accorda l'accesso in CS ad un processo, nessun altro quorum (anche diverso) potrà accordare lo stesso permesso
- 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 $N > 4$

Confronto tra algoritmi per ME distribuita

Algoritmo	Messaggi per ingresso/uscita dalla CS	Ritardo prima dell'ingresso (in messaggi)	Problemi
Autorizzazione o token centralizzato	3	2	Crash del coordinatore
Ricart-Agrawala	$2(N-1)$	$2(N-1)$	Crash di un qualunque processo
Token decentralizzato	1 a ∞	0 a $N-1$	Perdita del token Crash di un qualunque processo
Maekawa	$3\sqrt{N}$	$2\sqrt{N}$	Possibile deadlock

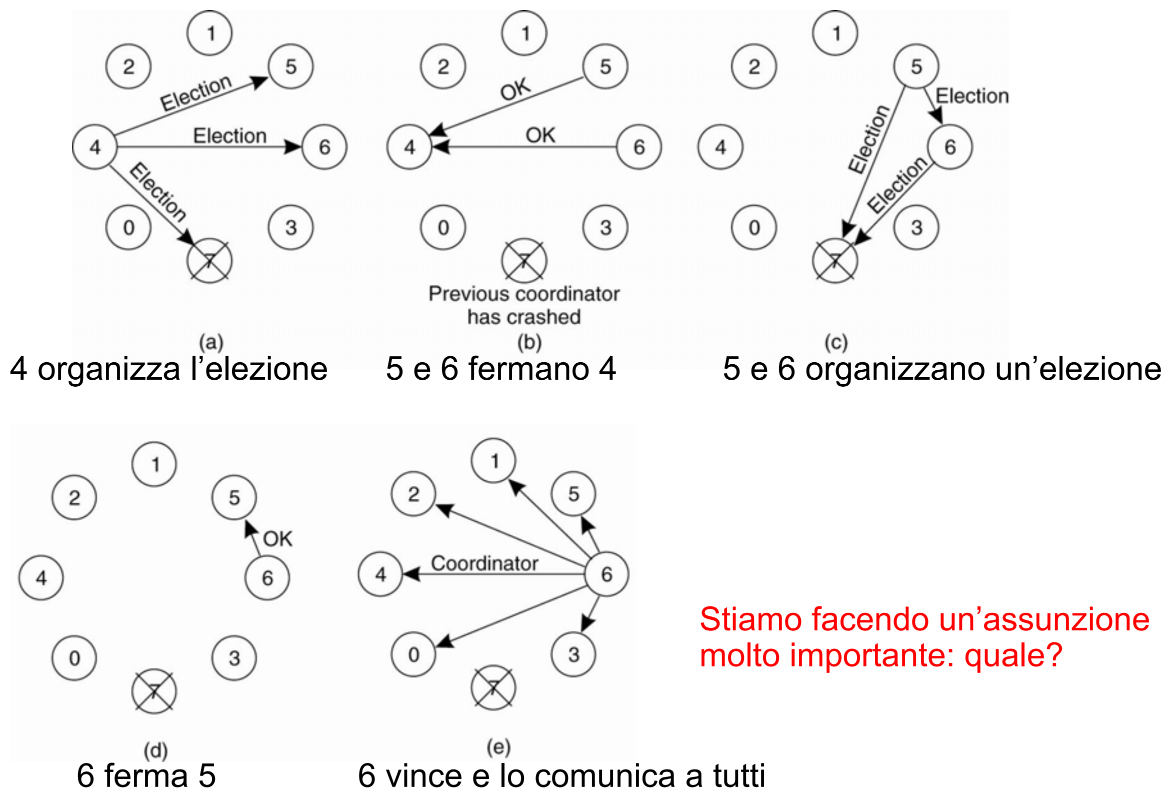
Algoritmi di elezione

- Molti algoritmi distribuiti richiedono che un processo agisca da *coordinatore* o *leader* oppure che abbia un ruolo speciale: come eleggere questo 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
- Se i processi sono identici, basta uno qualsiasi
  - Ipotesi: ogni processo ha un id univoco
  - Obiettivo: eleggere il processo con il maggior id
- Algoritmi di elezione tradizionali
  - Algoritmo bully (del prepotente)
  - Algoritmo ad anello
- Algoritmi di elezione di superpeer

Algoritmo bully

- Il processo P che rileva l'assenza del coordinatore promuove una nuova elezione
  - P invia un messaggio "ELEZIONE" a tutti i processi di identificatore maggiore
  - Se nessuno risponde, P si autoproclama vincitore e diventa coordinatore
  - Un processo che riceva il messaggio "ELEZIONE" da un processo di identificatore minore risponde con il messaggio "OK" e rileva la gestione dell'elezione
  - Se P riceve un messaggio "OK" ha finito il suo lavoro
- Un processo appena (ri-)creato non conosce il coordinatore e dunque promuove una elezione
- L'algoritmo designa sempre come coordinatore il processo in vita con identificatore maggiore
  - Il vincente informa tutti i processi del sistema che hanno un nuovo coordinatore

Algoritmo bully (2)



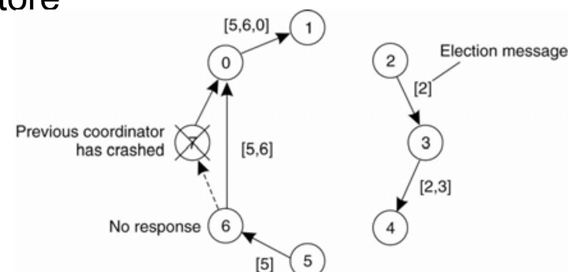
4 organizza l'elezione — 5 e 6 fermano 4 — 5 e 6 organizzano un'elezione

6 ferma 5 — 6 vince e lo comunica a tutti

Stiamo facendo un'assunzione molto importante: quale?

Algoritmo ad anello

- I processi sono ordinati logicamente in un anello
  - Ogni processo conosce il suo successore
- Il processo P che rileva l'assenza del coordinatore promuove una nuova elezione
  - P invia un messaggio di tipo ELEZIONE al suo successore
  - Se il successore non è attivo, il mittente lo salta e passa al successivo lungo l'anello, finché non trova un processo attivo
  - Ad ogni passo, il mittente aggiunge il suo numero di processo nel messaggio
  - Alla fine, il messaggio ritorna al processo P, che identifica il processo con il numero più alto ed invia sull'anello un messaggio di tipo COORDINATORE per informare tutti su chi sia il nuovo coordinatore



Algoritmi di elezione di superpeer

- Come selezionare superpeer che soddisfino uno o più tra i seguenti criteri:
  - *Accesso*: i peer normali abbiano un basso tempo di latenza verso i superpeer
  - *Distribuzione*: i superpeer siano distribuiti uniformemente nella rete overlay
  - *Proporzione*: ci sia una quantità predefinita di superpeer in base al numero totale di peer nella rete overlay
  - *Bilanciamento del carico*: ogni superpeer non serva più di un certo numero di peer
- Si possono applicare algoritmi derivanti dalla teoria dei grafi e dagli algoritmi di elezione tradizionali
  - Esempi: problema dell'insieme dominante, dominanza della distanza, problema del p-center

Riferimento: Lo et al., "Scalable supernode selection in peer-to-peer overlay networks", *Proc. of HOT-P2P*, 2005.