TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Mutua Esclusione ed Elezione nei Sistemi Distribuiti

Corso di Sistemi Distribuiti e Cloud Computing

A.A. 2023/24

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Main properties of algorithms for concurrent and distributed systems

- **Safety**: nothing bad will happen
- **Liveness**: eventually something good will happen

Mutua esclusione

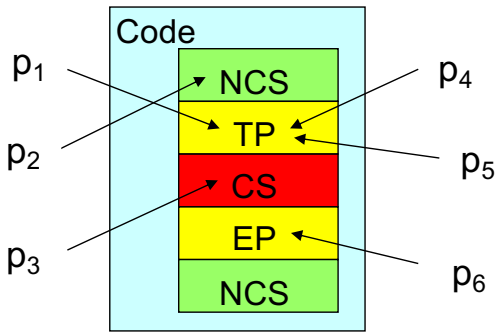
- *N* processi vogliono accedere ad una risorsa condivisa
  - Ogni processo vuole acquisire la risorsa ed usarla in modo esclusivo senza interferenze con gli altri processi
- Ogni algoritmo di mutua esclusione comprende
  - Una sequenza di istruzioni chiamata **sezione critica (CS)**
    - L'esecuzione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

Proprietà degli algoritmi di mutua esclusione

- **Mutua esclusione (ME)** o safety: al più un solo processo alla volta può eseguire in CS
- **No deadlock (ND)**: se un processo rimane bloccato nella sua trying section, almeno un altro processo riesce ad entrare ed uscire da CS
- **No starvation (NS)** o *assenza di posticipazione indefinita*: nessun processo rimane bloccato per sempre nella trying section, ovvero tutte le richieste di entrata in CS sono prima o poi soddisfatte
- Osservazioni:
  - NS implica ND (non vale il viceversa)
  - NS e ND sono proprietà di liveness
  - NS è anche una condizione di fairness
- **Ordering:** richieste di entrata in CS sono servite in ordine d'arrivo (in senso *happened-before*)
  - E' un'altra condizione di fairness

Mutua esclusione e sistemi concorrenti

- La mutua esclusione nasce nei sistemi concorrenti
- Algoritmi basati sull'uso di variabili condivise per realizzare la mutua esclusione tra *N* processi
  - **Algoritmo di Dijkstra** (1965)
    - Sistemi a singolo processore
    - Garantisce ME e ND ma non garantisce NS
  - **Algoritmo del panificio di Lamport** (1974)
    - Sistemi multiprocessore a memoria condivisa
    - Garantisce ME, ND e NS



Algoritmo del panificio di Lamport

- Soluzione ispirata ad una situazione reale
  - Attesa di essere serviti in un panificio
- Assunzioni sul modello di sistema concorrente
  - I processi comunicano leggendo/scrivendo variabili condivise
  - Lettura e scrittura di una variabile condivisa non sono operazioni atomiche: un processo può scrivere mentre un altro processo sta leggendo la stessa variabile
  - Ogni variabile condivisa è di proprietà di un processo
    - Tutti possono leggerla
    - Solo il proprietario può scriverla
  - 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 int inizializzati a 0
  - `choosing[1,…,N]`: array di boolean inizializzati a *false*
- Variabile locale
  - `j`: intero compreso in [1,…,N]

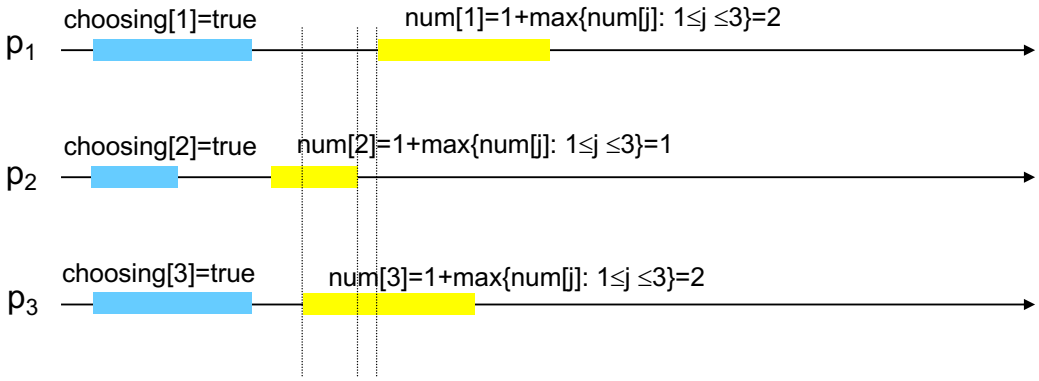
Algoritmo del panificio di Lamport

```
// Ciclo ripetuto all'infinito dal processo p_i
// sezione non critica
// prende un biglietto                                          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 suo numero sia chiamato confrontandolo con
// quello degli altri
for j = 1 to N do                                               bakery
   // busy waiting mentre j sta scegliendo
   while choosing[j] do NoOp();
   // busy waiting finché il numero di p_i non è il minimo
   // viene favorito il processo con id minore
   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: $\{a,b\} < \{c,d\}$ se a < c OR se $a = c$ AND $b < d$

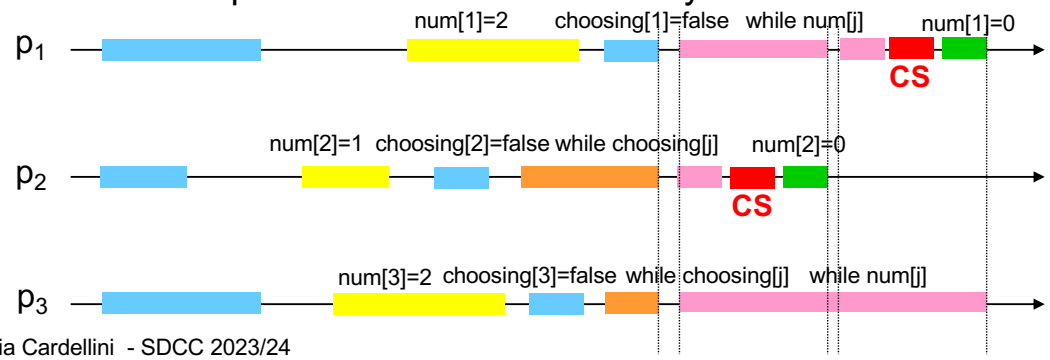
Algoritmo del panificio di Lamport

- Doorway
  - Quando p_i entra, lo segnala agli altri tramite `choosing[i]`
  - p_i prende un numero di biglietto pari al massimo dei numeri scelti dai processi precedenti più 1
  - Altri processi possono accedere in modo concorrente alla doorway
  - Esempio di esecuzione della doorway



Algoritmo del panificio di Lamport

- Bakery
  - p_i deve controllare che tra i processi in attesa sia lui il prossimo a poter entrare in CS
  - Il primo while permette a tutti i processi nella doorway di terminare la scelta del biglietto
  - Il secondo while lascia p_i in attesa finché:
    - Il suo numero di biglietto non diventa il minimo
    - 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
  - Esempio di esecuzione del bakery



Algoritmo del panificio di Lamport: proprietà

- Proprietà di ME
  - Deriva da: *se p_i è nella doorway e p_j è nel bakery allora {num[j], j} < {num[i],i}*
- Proprietà di NS
  - Nessun processo attende per sempre, poiché prima o poi il suo numero di biglietto diventerà il minimo
- Soddisfa anche proprietà di ordering
  - 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: modello della comunicazione

- Rispetto al modello di Lamport (slide 5) in un SD comunicazione tramite scambio di messaggi
  - Un processo non può leggere direttamente una variabile di un altro processo, ma deve inviare un messaggio di richiesta e ricevere un messaggio di risposta contenente il valore
- Assunzioni sulla comunicazione tra processi
  - Processi comunicano tramite scambio di messaggi
  - Ritardo di trasmissione dei messaggi non noto ma finito
  - Canali di comunicazione tra processi affidabili
    - Messaggio inviato ricevuto correttamente dal suo destinatario
    - No messaggi duplicati o spuri (ricevuti ma mai trasmessi)

Mutua esclusione distribuita: modello del sistema

- Sistema con N processi p_i, $i = 1, \ldots, N$
- 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

- Adattiamo ai SD l'algoritmo del panificio di Lamport

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

  Comunicazione basata su scambio di messaggi
  - Ogni processo legge i valori locali agli altri processi tramite messaggi di richiesta-risposta
- E' una buona soluzione? Funziona, ma
  - Costo di comunicazione per entrare in CS: $6N$ messaggi
    - Occorre leggere $3N$ variabili (N per doorway, $2N$ per bakery)
    - Per ogni lettura vengono scambiati 2 messaggi
  - Latenza determinata dalla combinazione *canale di comunicazione-processo* più lenta

  ✗ Limitata efficienza e scalabilità
    - Manca la cooperazione tra processi

Tassonomia degli algoritmi per ME distribuita

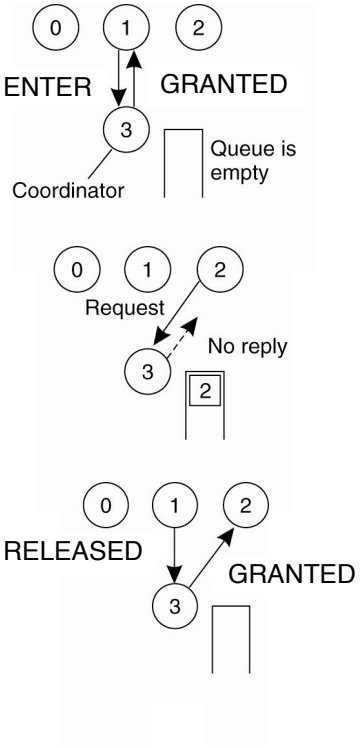
- Algoritmi basati su **autorizzazione**
  - Un processo che vuole accedere alla risorsa condivisa chiede l'autorizzazione, che è gestita
    - In modo *centralizzato* (coordinatore)
    - In modo distribuito (algoritmo di Lamport distribuito, algoritmo di Ricart e Agrawala)
- Algoritmi basati su **token**
  - 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
  - Token gestito
    - In modo distribuito
- Algoritmi basati su **quorum** (o votazione)
  - Un processo che vuole accedere alla risorsa condivisa chiede il voto ad un sottoinsieme di processi
    - Algoritmo di Maekawa

Prestazioni di algoritmi per ME distribuita

- Metriche per valutare le prestazioni di algoritmi di ME distribuita:
  - Numero di messaggi per entrare in CS e uscire da CS: misura indiretta della banda di rete consumata
  - Numero di messaggi per entrare in CS: misura indiretta del tempo di attesa

Autorizzazione: algoritmo centralizzato

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



Autorizzazione: algoritmo centralizzato

- Vantaggi e svantaggi
  - ✓ Garantisce safety e liveness

  Ordering?
  - Ordinamento FIFO garantito in ordine di richieste pervenute al coordinatore
  - ✗ Non nell'ordine in cui sono state inviate le richieste o nell'ordine in cui i processi richiedono l'entrata in CS
  - ✓ Il più semplice da implementare
  - ✓ Il più efficiente rispetto al numero di messaggi
    - Solo 3 messaggi (richiesta, risposta e rilascio) per entrare e uscire da CS
  - ✗ Coordinatore è SPoF e possibile collo di bottiglia per le prestazioni
  - ✗ Se un processo fallisce mentre è in CS, si perde il messaggio di rilascio

Autorizzazione: Algoritmo di Lamport distribuito

- Ogni processo p_i usa clock logico scalare per timestamp dei messaggi inviati (e relazione di ordine totale $\Rightarrow$) e gestisce coda locale
  - In coda memorizza richieste di accesso in CS degli altri processi
- Regole dell'algoritmo:
  - p_i richiede l'accesso in CS: p_i invia a tutti gli altri processi un messaggio di richiesta REQ(t_i, p_i) con timestamp il suo clock scalare t_i ed inserisce REQ(t_i, p_i) in coda
  - p_j riceve una richiesta da p_i: p_j inserisce in coda REQ(t_i, p_i) ricevuto da p_i e gli invia un messaggio di ack
  - p_i entra in CS *se e solo se*:
    - REQ(t_i, p_i) precede tutti gli altri messaggi di richiesta in coda (i.e., {t_i, p_i} è il minimo applicando $\Rightarrow$)
    - p_i ha ricevuto da ogni altro processo un messaggio (di ack o di richiesta) con timestamp maggiore di t_i (applicando $\Rightarrow$)
  - p_i esce da CS: p_i elimina REQ(t_i, p_i) dalla sua coda ed invia un messaggio di release *a tutti* gli altri processi
  - p_j riceve un messaggio di release da p_i: elimina REQ(t_i, p_i) dalla sua coda

Autorizzazione: Algoritmo di Lamport distribuito

- Simile ad algoritmo distribuito per multicast totalmente ordinato 🧐
- Garantisce safety, liveness e ordering
  - Ordering: richieste servite in base al timestamp e tempo basato su clock logico
- Richiede 3(*N*-1) messaggi per entrare in CS e uscirne:
  - *N*-1 messaggi di richiesta
  - *N*-1 messaggi di ack
  - *N*-1 messaggi di release

Autorizzazione: Algoritmo di Ricart e Agrawala

- Ottimizzazione dell'algoritmo di Lamport distribuito
  - Clock logico scalare (e $\Rightarrow$) più coda locale

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

- Un processo che riceve il messaggio di REQUEST può
  - non essere in CS e non volervi entrare $\rightarrow$ manda REPLY al mittente
  - essere in CS $\rightarrow$ non risponde e mette il messaggio in coda locale
  - voler entrare in CS $\rightarrow$ confronta il suo timestamp e id con timestamp e id ricevuti e vince la coppia minore: se è l'altro, invia REPLY; se è lui, non risponde e mette il messaggio in coda

Algoritmo di Ricart e Agrawala

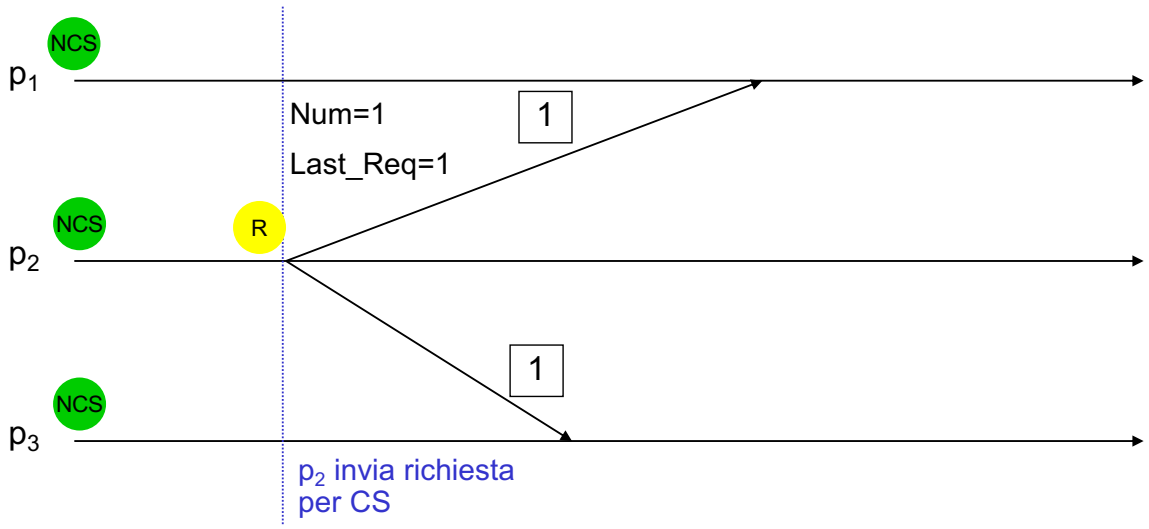
- Variabili locali per ciascun processo
  - #replies: numero di risposte ricevute (inizializzata a 0)
  - State $\in$ {Requesting, CS, NCS} (inizializzata a NCS)
  - Q: coda di richieste pendenti (inizialmente vuota)
  - Last_Req: timestamp del messaggio di richiesta (inizializzata a 0)
  - Num (inizializzata 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 pj;
4. Wait until #replies=N-1;
5. State = CS;
6. CS
7. $\forall$ r$\in$Q send REPLY to r
8. Q=$\varnothing$; State=NCS; #replies=0;

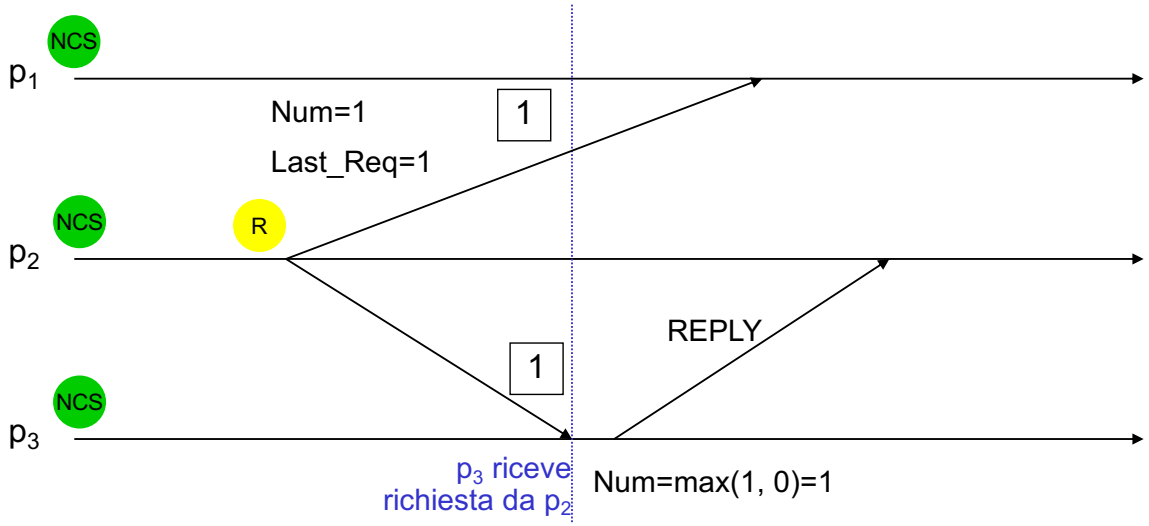
Upon receipt of REQUEST(t) from pj
1. if State=CS or (State=Requesting and {Last_Req, i} < {t, j})
2. then insert {t, j} in Q
3. else send REPLY to pj
4. Num = max(t, Num)

Upon receipt of REPLY from pj
1. #replies = #replies+1

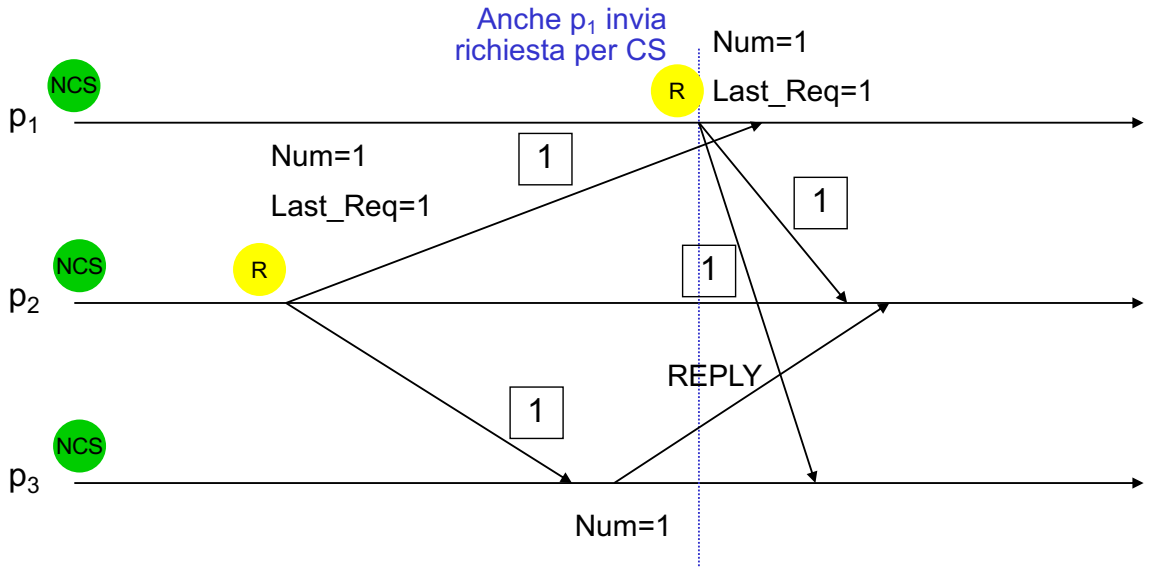
Algoritmo di Ricart e Agrawala: esempio



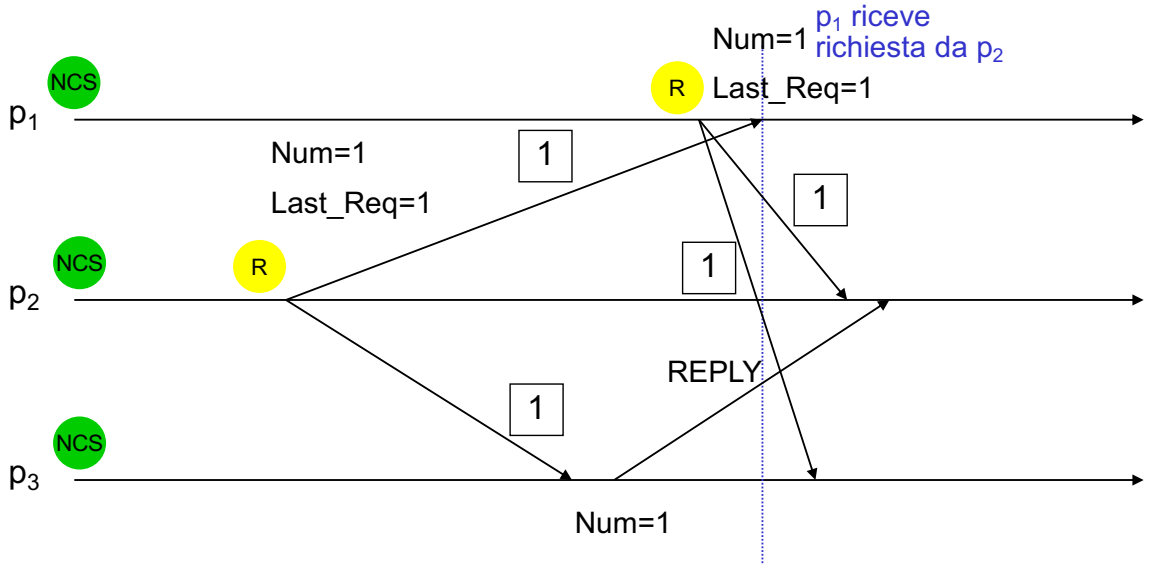
Algoritmo di Ricart e Agrawala: esempio



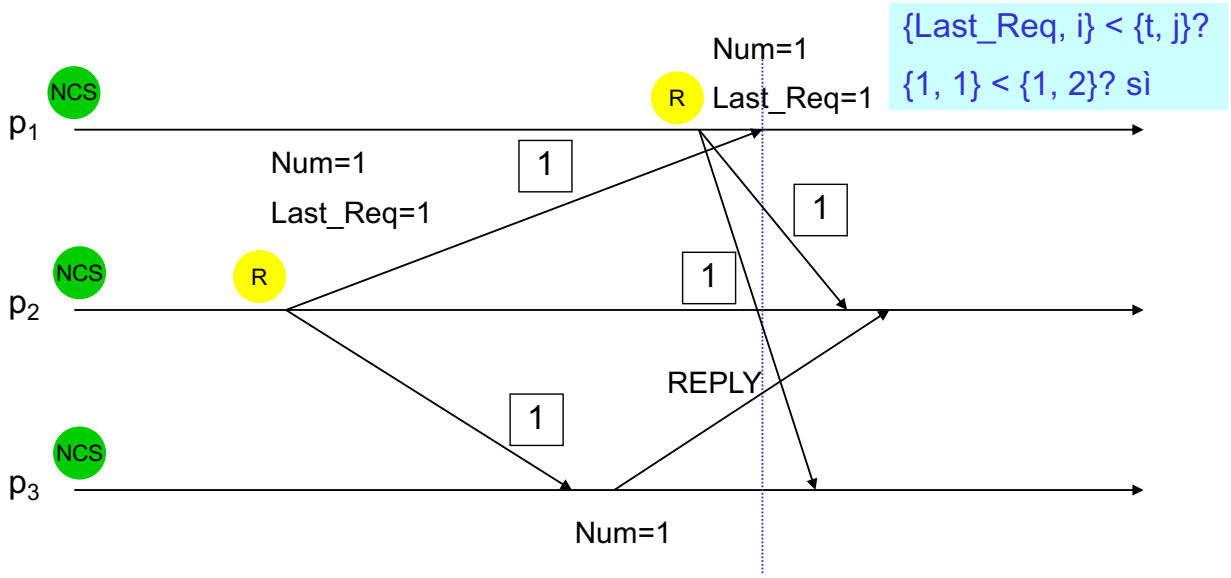
Algoritmo di Ricart e Agrawala: esempio



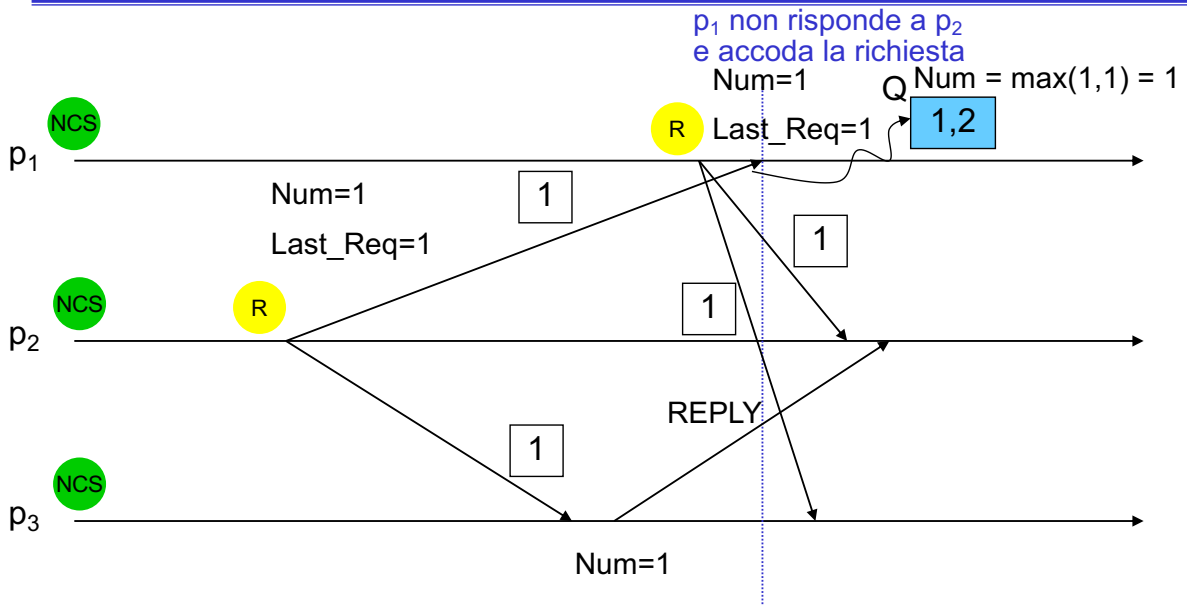
Algoritmo di Ricart e Agrawala: esempio



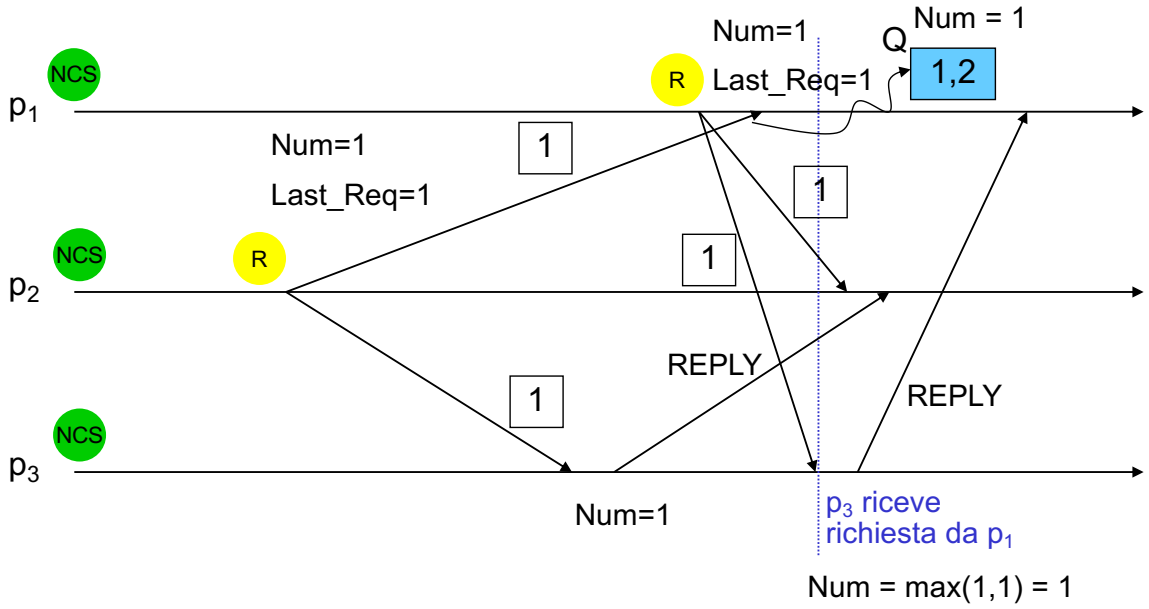
Algoritmo di Ricart e Agrawala: esempio



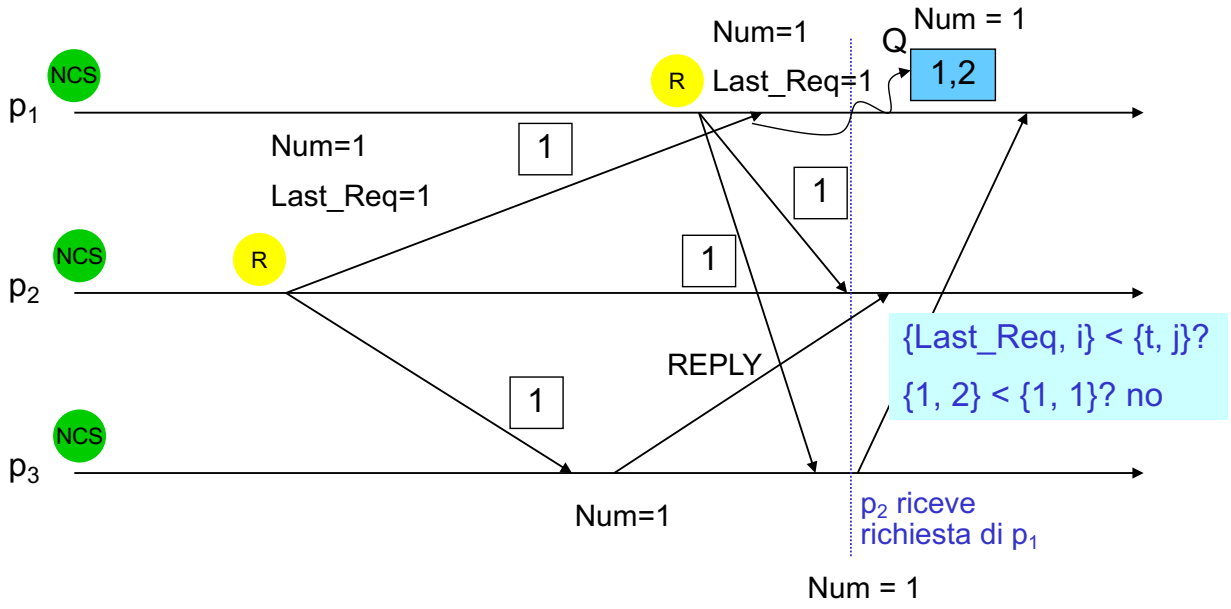
Algoritmo di Ricart e Agrawala: esempio



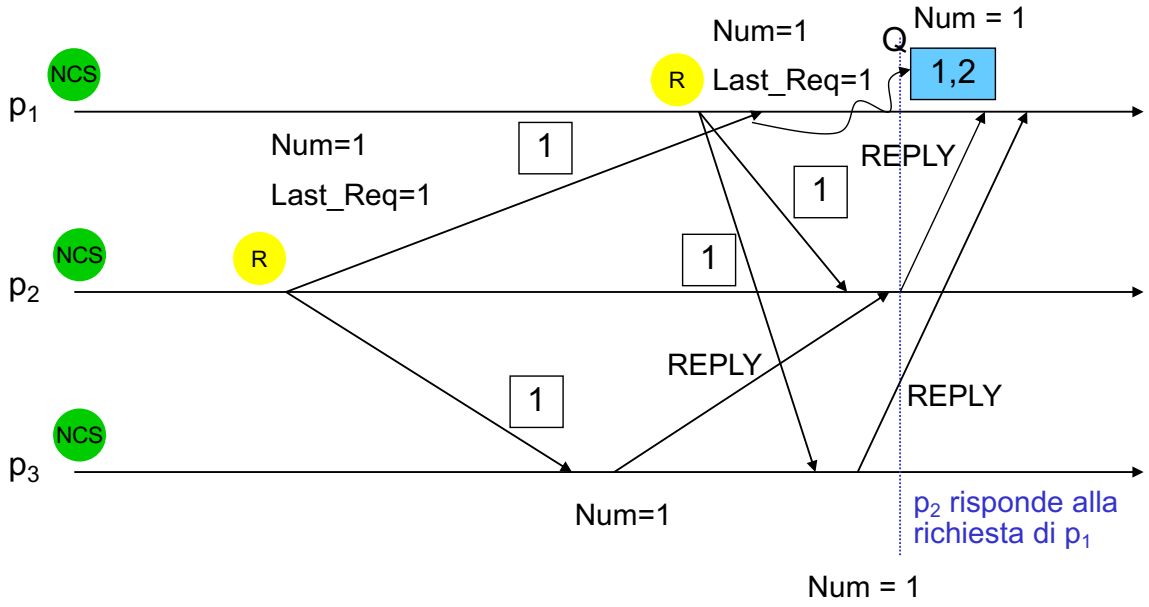
Algoritmo di Ricart e Agrawala: esempio



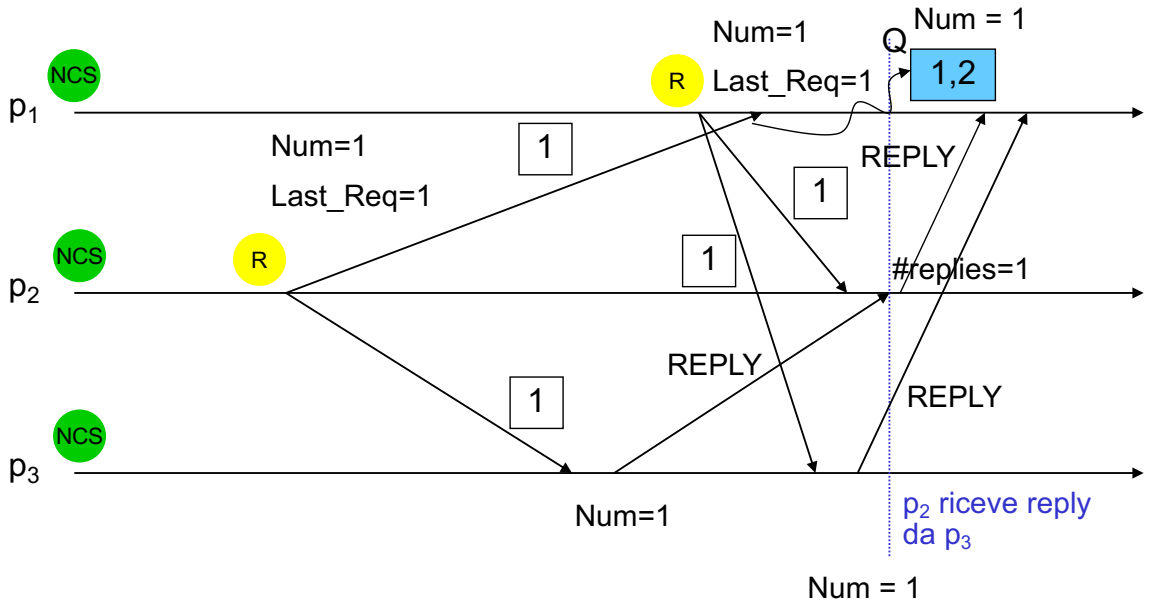
Algoritmo di Ricart e Agrawala: esempio



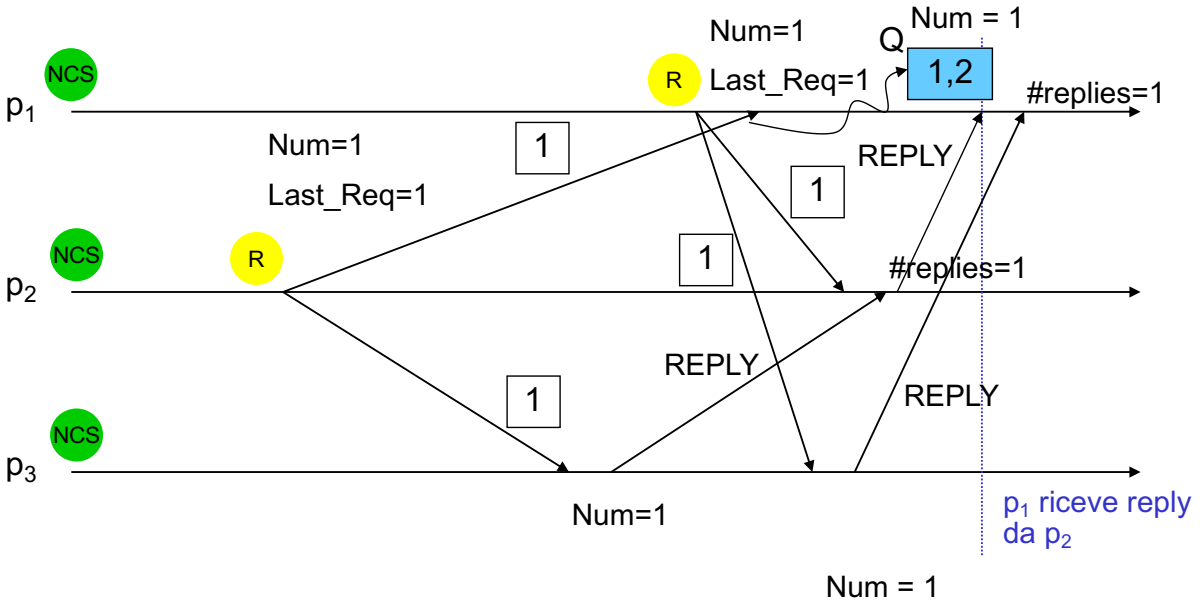
Algoritmo di Ricart e Agrawala: esempio



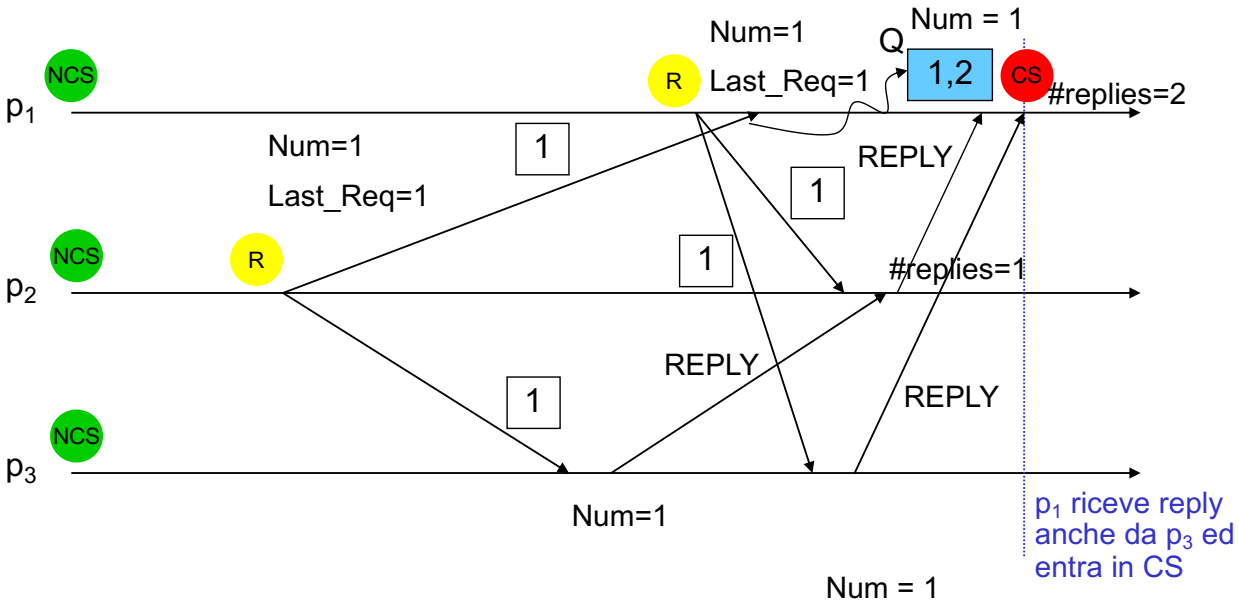
Algoritmo di Ricart e Agrawala: esempio



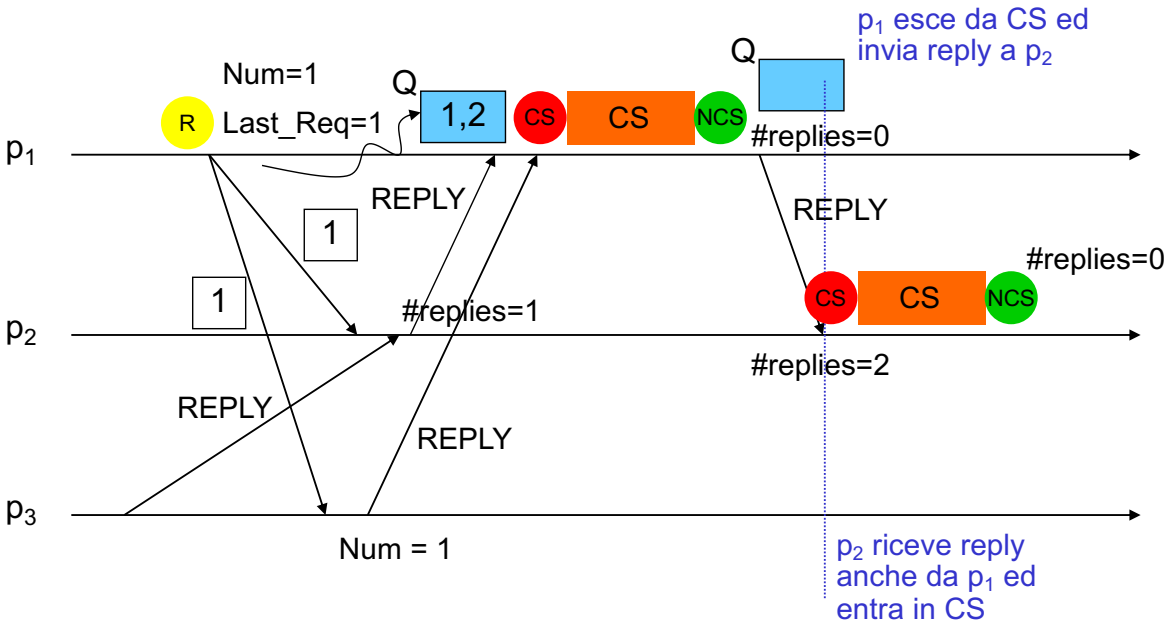
Algoritmo di Ricart e Agrawala: esempio



Algoritmo di Ricart e Agrawala: esempio



Algoritmo di Ricart e Agrawala: esempio



Algoritmo di Ricart e Agrawala

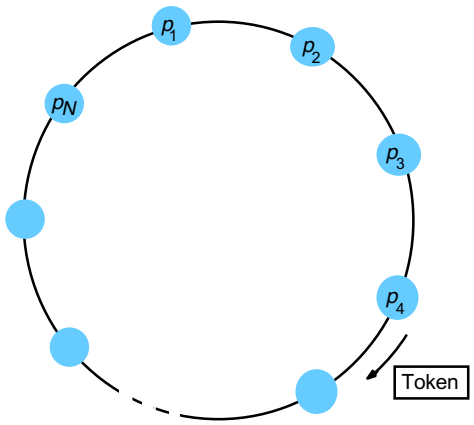
- Vantaggi
  - ✓ Come Lamport distribuito, è completamente distribuito
    - Nessun elemento centrale
  - ✓ Rispetto a Lamport distribuito, si risparmiano messaggi
    - No messaggio di release, msg di ack differito all'uscita da CS
    - Solo 2(*N*-1) messaggi per entrare in CS e uscirne: *N*-1 messaggi 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 qualunque processo fallisce, nessuno può entrare in CS: occorre aggiungere un meccanismo di *failure detection*
  - ✗ Ogni processo può essere collo di bottiglia
    - Ogni processo partecipa ad ogni decisione
  - ✗ Occorre conoscere chi appartiene al gruppo di multicast

Algoritmi basati su token

- Viene usata una risorsa ausiliaria, chiamata **token**
  - Anche altri algoritmi distribuiti usano il token (es. elezione)
- 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)
- Molteplici algoritmi basati su token, analizziamo
  - Distribuito (o *perpetuum mobile*): gestione decentralizzata del token, che si muove nel sistema

Algoritmo decentralizzato basato su token

- 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
  - Passa da p_i a $p_{(i+1) mod\ N}$
- 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

- ✓ Garantisce safety
- ✓ Garantisce NS (se anello unidirezionale)
- ✓ Garantisce ND (se il token non si perde)
- • Garantisce ordering?
- ✗ Consumo di banda di rete per trasmettere il token anche quando nessuno vuole entrare in CS
- ✗ In caso di perdita del token occorre rigenerarlo
- ✗ Guasti temporanei possono portare alla creazione di token multipli
- • In caso di crash di singoli processi
  - – Se fallisce un processo, occorre riconfigurare l'anello
  - – Se fallisce il processo che possiede il token, occorre rigenerare il token ed eleggere il prossimo processo che avrà il token

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 per turno
- Insieme di votazione V_i: sottoinsieme di $\{p_1, \ldots, p_N\}$, associato ad ogni processo p_i

- Un processo p_i per entrare in 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 entra in CS
  - All'uscita da 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 entrata 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
```


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. $V_i \cap V_j \neq \varnothing \quad \forall\, i, j$
   - 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 $K = \lceil \sqrt{N} \rceil$

Esempio: N=3

V_1={1,2}
V_3={1,3}
V_2={2,3}

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}

Algoritmo di Maekawa: proprietà e prestazioni

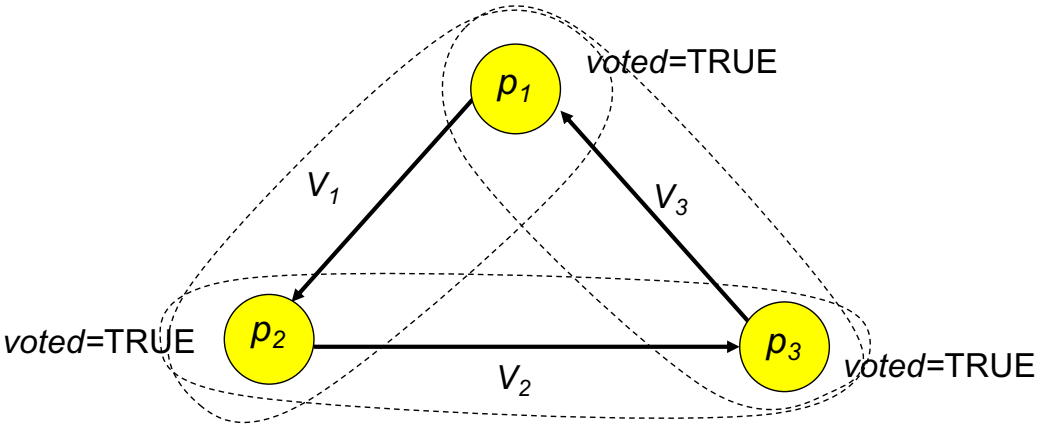
- Come scegliere quali sono i processi a cui inviare la richiesta?
  - Si costruiscono gli insiemi di votazione in modo tale che abbiano intersezione non vuota
  - Se un quorum autorizza l'accesso in CS ad un processo, nessun altro quorum potrà accordare lo stesso permesso
- Proprietà
  - Soddisfa safety ma non liveness
    - Si può verificare deadlock
    - Si può rendere l'algoritmo deadlock-free con messaggi aggiuntivi
- Prestazioni
  - Per entrare in CS e uscirne occorrono $3\sqrt{N}$ messaggi ($2\sqrt{N}$ per entrata e $\sqrt{N}$ per uscita)
    - Più efficiente di Ricart-Agrawala per reti a larga scala essendo $3\sqrt{N} < 2(N-1)$ per $N > 4$

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'entrata 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 entrata/uscita in/da CS	#msg per entrata in CS	Problemi
Autorizzazione centralizzato	3	2	Crash del coordinatore
Ricart-Agrawala	$2(N-1)$	$2(N-1)$	Crash di un qualunque processo
Token decentralizzato	Da 1 a ∞ (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

Distributed election algorithms

- Many distributed algorithms require that a process acts as *coordinator* (or *leader*), e.g.,
  - Sequencer in totally ordered multicast
  - Coordinator in mutual exclusion
- How to elect the coordinator at runtime?
  - Existing coordinator can crash
  - Election requires to reach a distributed consensus
- Let's study two election algorithms
  - **Bully algorithm**
  - **Ring election algorithm** by Fredrickson & Lynch

Distributed election: system model

- System with N processes p_i, $i = 1, \ldots, N$
- Processes can crash
- Communication is reliable
- Each process does not hold more than one election at time
- Each process has a unique id and the non-faulty process with the highest id is elected

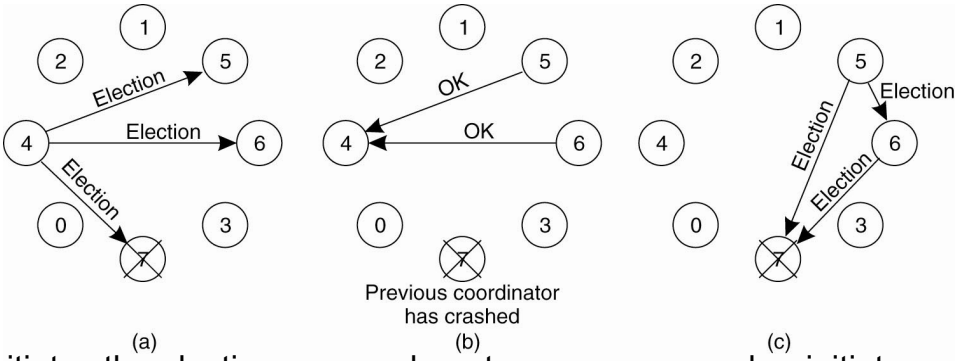
Distributed election: properties

- *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

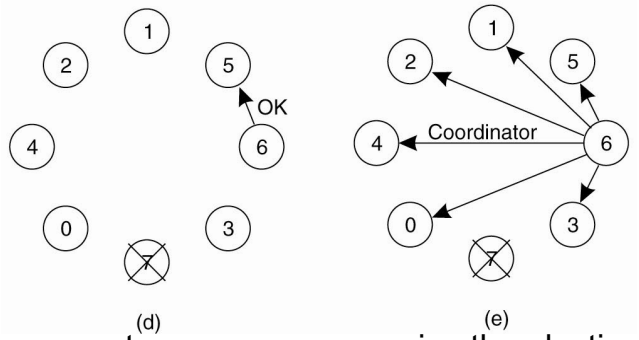
Bully algorithm

- Idea: "Node with highest ID bullies its way into leadership" (Garcia-Molina)
- p_i notices that coordinator is no longer responding and initiates an election
  - p_i sends an ELECTION message to all processes with higher id (p_{i+1}, p_{i+2}, …, p_N)
  - If no one responds, p_i wins the election and becomes the coordinator, announces the victory by sending all processes a message
  - If p_k (with $k>i$) receives the ELECTION message from p_i, it answers sending OK message, takes over and holds an election
  - If p_i receives OK message, it sits back
- This algorithms always selects as new leader the non-faulty process with the highest id
- A new process (or a process that restarts after crash) does not know the coordinator and therefore holds an election

Bully algorithm: example



p_4 initiates the election — p_5 and p_6 stop p_4 — p_5 and p_6 initiate a new election



p_6 stops p_5 — p_6 wins the election and becomes the new coordinator

Which is an important assumption?

Synchronous DS and we use a timeout-based mechanism to identify a faulty process

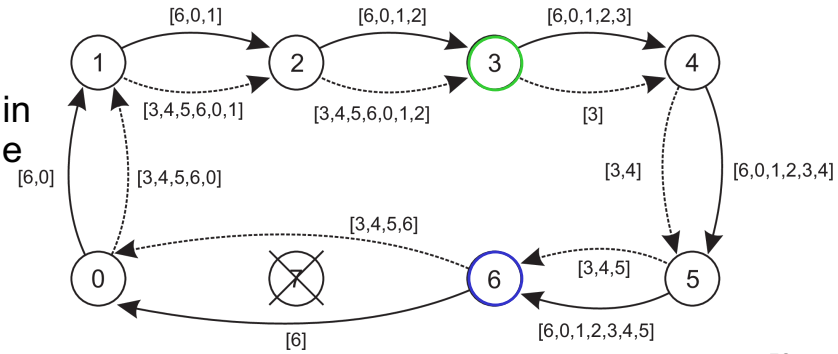
Bully algorithm: communication cost

- 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 ⇒ O(N) messages
- Worst case (assuming no process fails during election): the process with the lowest id initiates the election
  - It sends N-1 election messages to processes, which themselves initiate each one an election

    ((N-1) + (N-2) + … + 1) + N-1 ⇒ $O(N^2)$ messages

Ring algorithm by Fredrickson & Lynch

- Processes are organized in a logical ring (unidirectional)
  - Each process knows at least its successor
- Process p_i notices that coordinator is no longer responding and initiates an election
  - p_i sends an ELECTION message to $p_{(i+1) mod\ N}$ with its own id
  - If $p_{(i+1) mod\ N}$ *is* faulty, p_i skips over it and goes to the next process along the ring, until a non-faulty process is located
  - At each step, the receiver adds its own id to the list in the ELECTION message and forwards the message
- Eventually, the ELECTION message gets back to p_i, which identifies the highest id in the list and circulates the COORDINATOR message to inform everyone else who the coordinator is

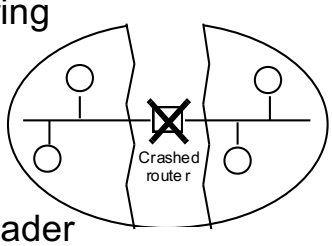


Election algorithms: cost

- 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

- Common assumption
  - 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 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



References

- Sections 5.3 and 5.4 of van Steen & Tanenbaum book
- Sections 15.2 and 15.3 of Coulouris et al. book