DIETI UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

*Corso di Laurea Magistrale in Ingegneria Informatica*

Corso di

# Distributed Systems and IoT

# Algoritmi di mutua esclusione nei sistemi distribuiti

1

## Sommario

- Formulazione del problema della mutua esclusione distribuita
- Algoritmo del server centrale
- Algoritmo ad anello
- Algoritmo con *multicast* e orologi logici
- Algoritmo di *voting* di Maekawa
- Valutazione degli algoritmi

Riferimenti:

G. Coulouris et al.: *Distributed Systems: Concepts and Design*, V ed., 2012, **Cap. 15 § 2**.

2

2

## Il problema della mutua esclusione: formulazione

Informalmente, il problema della mutua esclusione distribuita consiste nel far sì che un insieme di processi distribuiti che condividono una risorsa (o una collezione di risorse) possano accedervi senza interferenze e senza determinarne stati inconsistenti.

Spesso le risorse sono gestite (e incapsulate) da processi *server*, che possono offrire meccanismi di accesso in mutua esclusione. In tal caso si applicano gli algoritmi di mutua esclusione (non distribuita) noti dalla teoria dei sistemi operativi.

Nella teoria dei sistemi distribuiti, il caso di interesse è quello in cui la risorsa non è gestita da un singolo server locale, e un insieme di processi (*peers*) devono coordinare gli accessi alle risorse tra essi condivise.

Due esempi applicativi:

- rete *wireless* in modalità *ad hoc*;

- sistema di controllo dei posti vacanti in un parcheggio multiaccesso.

3

3

## Mutua esclusione distribuita: modello di sistema

Si considera un sistema con $N$ processi $p_i$, $i = 1, \ldots, N$.

I processi non hanno variabili condivise, ed accedono alle risorse comuni in una **sezione critica**.

Esiste un'unica sezione critica.

Il sistema è asincrono.

I processi non sono soggetti a fallimenti.

I canali sono affidabili, e i messaggi prima o poi sono consegnati intatti una e una sola volta.

I processi trascorrono un tempo finito nella sezione critica, rilasciando quindi le risorse comuni dopo un tempo limitato.

4

4

## Mutua esclusione distribuita: requisiti

I requisiti che devono essere soddisfatti da un algoritmo di mutua esclusione distribuita in ogni sua esecuzione sono:

- **ME1** (***safety***): al più un processo alla volta può trovarsi nella sezione critica;
- **ME2** (***liveness***): le richieste di ingresso e uscita dalla sezione critica devono prima o poi essere soddisfatte.

La *liveness* garantisce l'assenza di ***deadlock*** e ***starvation***.

L'assenza di *starvation* è una condizione di imparzialità (***fairness***). Spesso è richiesto l'ulteriore requisito di imparzialità:

- **ME3** (***ordering***): se la richiesta $x$ è in relazione HB con la richiesta $y$ ($x \rightarrow y$), l'accesso alla sezione critica da parte di $x$ deve avvenire prima dell'accesso di $y$.

5

5

## Mutua esclusione distribuita: criteri di valutazione

Gli algoritmi di mutua esclusione distribuita possono essere valutati in base ai seguenti criteri:

A. **banda di rete** (***network bandwidth***) consumata, proporzionale al numero di messaggi inviati per l'ingresso e l'uscita dalla sezione critica;

B. **ritardo di un processo** (***client delay***) in ingresso e uscita dalla sezione critica;

C. **ritardo di sincronizzazione** (***synchronization delay***) tra due processi che si succedono nella sezione critica: è una misura dell'effetto dell'algoritmo sulla produttività (***throughput***) del sistema; quest'ultima determina il numero massimo di processi che complessivamente possono accedere alla sezione critica nell'unità di tempo.

6

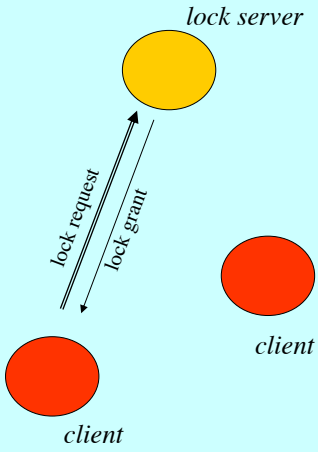
6

## Algoritmo del *server* centrale (1/2)

Una soluzione distribuita basata su controllo centralizzato è quella di prevedere un server centrale (*lock server*) cui i processi inviano le richieste d'accesso alla risorsa condivisa.

Questa soluzione presenta diversi svantaggi:

- il *lock server* è un punto-di-guasto-unico: in caso fallisca, si perde conoscenza di quale processo detiene la risorsa, nonché dei processi in coda e del loro ordine;
- in caso di fallimento del cliente nella sezione critica, si perde il messaggio di rilascio;
- il *lock server* può essere un collo-di-bottiglia delle prestazioni.




7

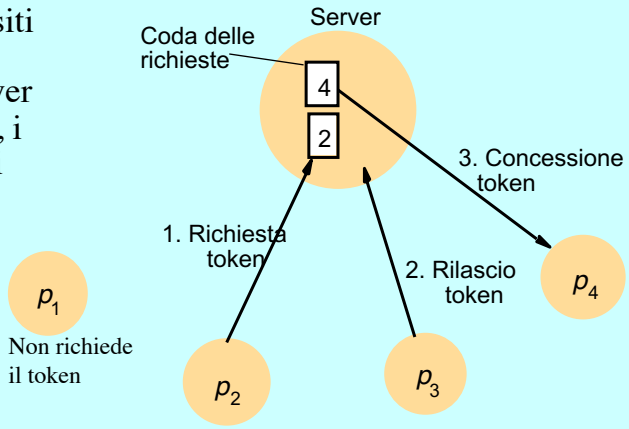
7

## Algoritmo del *server* centrale (2/2)

Per entrare nella sezione critica, un processo invia un messaggio a un apposito *server* ed attende finché ottiene in risposta il permesso.

Sono soddisfatti i requisiti ME1 (*safety*), ME2 (*liveness*), poiché il server gestisce una coda FIFO, i canali sono affidabili e i processi non vanno in *crash* e trascorrono un tempo limitato nella sezione critica.

Il requisito ME3 (*ordering*) non è soddisfatto.




8

8

## Algoritmo del *server* centrale - valutazione

**A.** Il numero dei messaggi è proporzionale alle richieste di accesso alla sezione critica.

L'algoritmo non consuma banda di rete quando nessun processo richiede l'accesso alle risorse comuni.

**B.** In ingresso nella sezione critica, il *client delay* è di 2 messaggi (*request*, *grant*).

In uscita dalla sezione critica, il *client delay* è di un solo messaggio (*release*).

**C.** Il ritardo di sincronizzazione tra il processo uscente e un processo entrante è di 2 messaggi (*release*, poi *grant* al successivo).

9

9

## Algoritmo di Lamport (1/3)

Si potrebbe immaginare una soluzione ispirata all'algoritmo del *server* centrale, ma completamente decentralizzata, replicando in ciascun processo lo stato del *lock server*: per richiedere la risorsa, un processo dovrebbe inviare la richiesta di *lock* a tutti gli altri, e attendere i relativi *lock grant*.

Naturalmente tale soluzione sarebbe inadeguata.

Si supponga di avere 4 processi:

- $p_1$ e $p_2$ inviano simultaneamente una *lock request*;

- la richiesta di $p_1$ è ricevuta prima da $p_3$, che invia un *lock grant*;

- la richiesta di $p_2$ è ricevuta da $p_4$ prima di quella di $p_1$; $p_4$ invia un *lock grant* a $p_2$, con una decisione inconsistente con quella di $p_3$;

Inoltre l'algoritmo causerebbe un *deadlock*, perchè $p_1$ e $p_2$ non solo non otterrebbero il *lock*, ma impedirebbero ad altri di ottenerlo.

10

10

## Algoritmo di Lamport (2/3)

I problemi di tale soluzione decentralizzata possono essere risolti:

- con l'uso di multicast totalmente ordinati;

- con l'uso di orologi logici.

In quest'ultimo caso (Lamport, 1978), tutti i messaggi vanno marcati temporalmente con l'orologio logico del mittente.
Le richieste ricevute sono immesse da ciascun processo in una coda, ordinata secondo la loro marcatura temporale.

Alla ricezione di una richiesta, viene inviato un ACK a tutti gli altri processi. Quando una richiesta ha ricevuto ACK da tutti i processi, viene marcata come stabile.

11

11

## Algoritmo di Lamport (3/3)

Un processo entra nella sezione critica quando:

- la risorsa è libera,

- la propria richiesta è in cima alla coda, e

- la propria richiesta è stabile.

All'uscita dalla sezione critica si inviano messaggi di *release*.

Non è necessario inviare messaggi di *lock grant*.

12

12

## Algoritmo ad anello

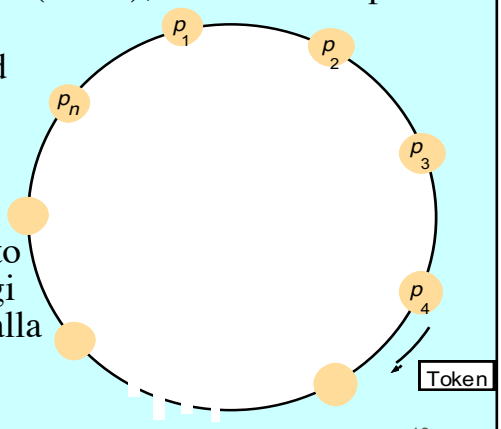
I processi sono organizzati logicamente ad anello: il processo $p_i$ ha un canale di comunicazione con il successivo $p_{(i+1) mod N}$.

I processi si scambiano un gettone (***token***), che ruota sempre nella stessa direzione.

Il possesso del *token* dà diritto ad entrare nella sezione critica.

Sono soddisfatti i requisiti ME1, ME2.

Il requisito ME3 non è soddisfatto (i processi si scambiano messaggi applicativi indipendentemente dalla rotazione del *token*).




13

13

## Algoritmo ad anello - valutazione

**A.** Il *token* ruota in continuazione, tranne quando un processo è nella sezione critica, perciò l'algoritmo consuma banda di rete, anche quando nessun processo richiede l'accesso alle risorse comuni.

**B.** In ingresso nella sezione critica, il *client delay* varia da 0 (se ha appena ricevuto il *token*) a *N* messaggi (se l'ha appena ceduto).

In uscita dalla sezione critica, il *client delay* è di un solo messaggio.

**C.** Il ritardo di sincronizzazione tra il processo uscente e un processo entrante varia da 1 a *N*-1 messaggi.

14

14

## Algoritmo di Ricart e Agrawala (1/5)

È un algoritmo basato su *multicast* e orologi logici (1981).

Un processo che vuole entrare nella sezione critica invia in *multicast* una richiesta, ed accede effettivamente alla risorsa solo dopo aver ricevuto risposta da tutti gli altri processi.

I processi hanno identificativi univoci numerici interi $p_i$, e possiedono ciascuno un orologio logico di Lamport.

I messaggi di richiesta sono nella forma $<T, p_i>$, dove $T$ è la marcatura temporale (*timestamp*) di Lamport.

Le condizioni in base alle quali i processi rispondono garantiscono il soddisfacimento dei requisiti ME1, ME2, ME3.

15

15

## Algoritmo di Ricart e Agrawala (2/5)

Ogni processo ha una variabile *state* con i possibili valori:

-RELEASED: il processo è all'esterno della sezione critica;

-HELD: il processo è nella sezione critica;

-WANTED: il processo vuole accedere alla sezione critica.

Un processo nello stato WANTED invia in multicast la richiesta $<T, p_i>$, ed attende di raccogliere $N$-1 risposte.

Un processo nello stato RELEASED risponde immediatamente alle richieste.

Il processo nello stato HELD risponde alle richieste solo al termine della sezione critica, mettendo quindi in attesa il/i richiedente/i.

16

16

## Algoritmo di Ricart e Agrawala (3/5)

***On initialization***
*state* := RELEASED;
***To enter the section***
*state* := WANTED;
Multicast *request* to all processes;
*T* := request's timestamp;
*Wait until* (number of replies received = (*N* – 1));
*state* := HELD;

request processing deferred here

***On receipt of a request $\langle T_i, p_i \rangle$ at $p_j$ ($i \neq j$)***
*if* (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
*then*
queue *request* from $p_i$ without replying;
*else*
reply immediately to $p_i$;
*end if*
***To exit the critical section***
*state* := RELEASED;
reply to any queued requests;

17

17

## Algoritmo di Ricart e Agrawala (4/5)

Se un solo processo è nello stato WANTED e gli altri nello stato RELEASED, il richiedente riceve subito *N*-1 risposte ed entra nella sezione critica.

Se un solo processo è nello stato WANTED ed un processo è nello stato HELD, allora *N*-2 processi sono nello stato RELEASED, il richiedente riceve subito *N*-2 risposte, ed entra nella sezione critica appena il processo HELD rilascia la risorsa.

Se due o più processi sono nello stato WANTED, il primo a raccogliere *N*-1 repliche è il processo la cui richiesta ha il *timestamp* minore. A parità di *timestamp* logico, ha la precedenza il processo con l'identificativo minore.

È essenziale che nella fase di invio della richiesta ciascun processo sospenda l'elaborazione delle richieste altrui.

18

18

## Algoritmo di Ricart e Agrawala (5/5)

L'algoritmo di Ricart e Agrawala è una ottimizzazione dell'algoritmo di Lamport, basata sull'osservazione che un processo non può entrare nella sezione critica se la sua richiesta non è stabile, cioè non ha avuto N-1 *grant*.

Pertanto è possibile risparmiare i messaggi espliciti di *release*, se il processo che detiene la risorsa differisce l'invio degli ACK al termine della sezione critica.

19

19

## Algoritmo di Ricart e Agrawala - valutazione

**A.** L'ingresso nella sezione critica comporta 2(*N*-1) messaggi: *N*-1 per il *multicast* della richiesta, e *N*-1 repliche. Se il *multicast* è supportato in hardware, occorrono *N* messaggi.

L'algoritmo non consuma banda di rete quando nessun processo richiede l'accesso alle risorse comuni.

**B.** Se il *multicast* è in hardware, in ingresso il *client delay* è di due tempi di trasmissione di messaggi (richiesta in *multicast*, e risposta dal processo uscente).

In uscita dalla sezione critica, il *client delay* è pari al numero di processi in attesa. Se il *multicast* è in hardware, il *client delay* è di un solo messaggio.

**C.** Il ritardo di sincronizzazione tra il processo uscente e un processo entrante è di un solo messaggio.

20

20

## Algoritmo di Maekawa (1/5)

Si tratta di un algoritmo basato sull'osservazione che, affinché un processo possa entrare nella sezione critica, non è necessario che attenda il permesso di tutti gli altri: è sufficiente che ciascuno attenda e riceva il permesso da sottoinsiemi dei suoi pari, a condizione che i sottoinsiemi di due qualsiasi processi non siano disgiunti.

Si può pensare a tale algoritmo come a una votazione distribuita: i processi votano per entrare nella sezione critica, e per entrarvi un candidato deve raccogliere un numero sufficiente di voti.

I processi comuni a due sottoinsiemi garantiscono il requisito di *safety* ME1, in quanto sono potenziali elettori di più processi, ma devono votare per uno solo di essi.

21

21

## Algoritmo di Maekawa (2/5)

A ciascun processo $p_i$ è associato un ***voting set*** $V_i$, con:

- $V_i \subseteq \{p_1, p_2, ..., p_N\}$,
- $p_i \in V_i$ - *ogni processo appartiene al proprio voting set*;
- $V_i \cap V_j \neq \emptyset$ - *due qualsiasi voting set non sono disgiunti*;
- $|V_i| = K$ - *i voting set sono di pari dimensioni (fairness)*;
- Ogni processo $p_j$ è contenuto in $M$ dei *voting set* $V_i$.

La soluzione ottima – che minimizza $K$ – prevede:

$$K \sim \sqrt{N}, \quad M = K,$$

ed ogni processo appartiene a un numero di *voting set* pari alla cardinalità dei *voting set*.

22

22

## Algoritmo di Maekawa (3/5)

Ogni processo ha una variabile *state* ∈{RELEASED, WANTED, HELD} e una var. booleana *voted*, inizialmente pari a RELEASED e FALSE, rispettivamente.

**RICHIESTA.** Un processo $p_i$ che transita nello stato WANTED invia in *multicast* un messaggio di richiesta a tutti i $K$ membri di $V_i$, incluso sé stesso, e attende di ricevere tutte le $K$ repliche prima di entrare nello stato HELD.

Un processo $p_j$ in $V_i$ che riceve una richiesta:

- se si trova nello stato RELEASED: risponde (vota) e pone *voted=TRUE*;
- se si trova nello stato HELD, o se ha già votato dal più recente messaggio di rilascio ricevuto: accoda la richiesta.

**RILASCIO.** All'uscita dalla sezione critica, $p_i$ transita nello stato RELEASED, e invia in *multicast* un messaggio di rilascio ai processi nel suo *voting set*.

Un processo $p_j$ che riceve un messaggio di rilascio:

- se la coda è vuota: definisce *voted=FALSE*,
- altrimenti: replica a una richiesta pendente, rimuovendola dalla coda, e pone *voted=TRUE*.

23



## Algoritmo di Maekawa (4/5)

***On initialization***
*state* := RELEASED;
*voted* := FALSE;

***For $p_i$ to enter the critical section***
*state* := WANTED;
Multicast *request* to all processes in $V_i$;
*Wait until* (number of replies received = $K$);
*state* := HELD;

***On receipt of a request from $p_i$ at $p_j$***
*if* (*state* = HELD *or voted* = TRUE)
*then*
queue *request* from $p_i$ without replying;
*else*
send *reply* to $p_i$;
*voted* := TRUE;
*end if*

***For $p_i$ to exit the critical section***
*state* := RELEASED;
Multicast *release* to all processes in $V_i$;

***On receipt of a release from $p_i$ at $p_j$***
*if* (queue of requests is non-empty)
*then*
remove head of queue – from $p_k$, say;
send *reply* to $p_k$;
*voted* := TRUE;
*else*
*voted* := FALSE;
*end if*

24

## Algoritmo di Maekawa (5/5)

L'algoritmo base soddisfa i requisito di *safety* ME1, ma non quello di *liveness* ME2: è infatti possibile che si verifichi un *deadlock*.

Se infatti tre processi $p_1, p_2, p_3$ richiedono concorrentemente l'accesso, e $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$, $V_3 = \{p_3, p_1\}$, è possibile che ogni processo replichi a sé stesso e tenga fuori l'altro: ogni processo riceverebbe una replica su due, e nessuno può procedere oltre.

L'algoritmo può essere adattato in modo da soddisfare ME2 (Saunders, 1987). L'adattamento di Saunders si basa sull'uso di orologi logici, e sull'accodamento delle richieste secondo l'ordinamento *happened-before*; in questo modo anche il requisito ME3 è soddisfatto.

25

25

## Algoritmo di Maekawa - valutazione

**A.** Se il *multicast* non è supportato in hardware, l'ingresso nella sezione critica comporta $2\sqrt{N}$ messaggi, e l'uscita comporta $\sqrt{N}$ messaggi. Per quanto riguarda il consumo di banda, il totale dei messaggi scambiati è $3\sqrt{N}$.

Se $N > 4$, $3\sqrt{N} < 2(N-1)$, e il consumo di banda è migliore di quello dell'algoritmo di Ricart e Agrawala.

**B.** Il *client delay* è lo stesso dell'algoritmo di Ricart e Agrawala.

**C.** Il ritardo di sincronizzazione tra il processo uscente e un processo entrante è di due messaggi, peggiore di quello di Ricart e Agrawala.

26

26