

I Sistemi Distribuiti

Un sistema distribuito è un insieme di entità geograficamente separate, ciascuno di questi ha una certa potenza computazionale. Essi sono in grado di comunicare e coordinarsi tra loro per raggiungere un obiettivo comune. Il loro funzionamento ricorda quello del Client e Server in cui vengono coordinati affinché il primo risolva una RPC, invece il secondo soddisfi delle richieste. La fondamentale differenza è nel numero delle entità in gioco e i problemi (perdita messaggi, latenze → tempo che ci impiega il pacchetto ad arrivare, ...) si moltiplicano.

Gli obiettivi comuni, solitamente, sono la condivisione di dati e risorse → problema di mutua esclusione. Per superarlo si potrebbe mettere un server che regola l'accesso. Però se si rompesse, non si potrebbe accedere alle risorse. Esso potrebbe morire per un problema fisico o per un sovraccarico di richieste (fenomeno di trashing).

Da questo discorso si capisce che è un approccio troppo cagionevole. Perciò è bene considerare i sistemi distribuiti come un insieme di elementi dello stesso livello.

I problemi principali sono legati alla coordinazione e alla sincronizzazione. Per cercare di implementare questi aspetti, bisogna tenere conto delle seguenti differenze con i sistemi concorrenti:

1. Concorrenza temporale e spaziale → i processi avvengono su diverse macchine
2. Non esiste un clock globale → non può essere stabilito quale processo procede un altro, a meno che uno non implichi l'altro (SEND prima della RECEIVE) → non esiste un ordine totale, solo all'interno del processo stesso (dovuto al clock e al processore → esecuzione sequenziale delle istruzioni)
3. Failures (Guasti) (non trattati nel corso) → se una macchina si rompe, tutto il suo lavoro muore, in un sistema distribuito, si potrebbe trovare un algoritmo che eviti questo problema → più host si rompono, più ne risentono le prestazioni, ma si evitano le morti dei loro programmi
4. Latenze → il programmatore deve scrivere algoritmi che tengano conto dei diversi tempi di rete, e quindi, dei ritardi dei pacchetti

Questi 4 punti portano a delle restrizioni nel risolvere i problemi con un'impostazione distribuita.

Dai sistemi concorrenti a quelli distribuiti

Il passaggio tra sistemi concorrenti con quelli distribuiti si concentra sullo studio della mutua esclusione.

Per risolvere il seguente problema, vengono enunciate delle ASSUNZIONI su cui verrà costruito il PROTOCOLLO risolutivo che ha delle determinate PROPRIETA'.

Le assunzioni per sistemi centralizzati si basano sullo scheduler e sull'accesso alla memoria, invece per quelli distribuiti, sullo scambio di messaggi.

Formuliamo il problema: ci sono N processi che eseguono, ripetutamente, il seguente schema di codice:

```
1: <non in sezione critica> // codice utente  
    <trying sectionl> // prova ad entrare in sc  
    sezione critica // zona in cui il processo accede a risorse condivise  
    <exit protocol> // fine protocollo  
    <exit sezione critica> //uscita da sc  
    Go to 1 //riesecuzione del codice
```

Affinché le proprietà vengano rispettate, bisogna coordinare i vari processi.

Le istruzioni dopo <non in sezione critica> vengono gestite da un protocollo fa riferimento alle assunzioni

Il primo algoritmo che studieremo (Dijkstra) fa riferimento ad un modello di sistema primitivo, concorrente, composto da:

- CPU

- Scheduler che permette l'accesso ai processi alla CPU
- Memoria collegati da un bus

Proprietà di Correttezza che un protocollo deve necessariamente possedere:

P1 → MUTEX → al più un processo può entrare in sezione critica, uno alla volta (due processi non possono entrare in sezione critica contemporaneamente)

P2 → NO_DEADLOCK → un processo tra N rimane bloccato e non accede alla sezione critica, mentre ne esiste almeno uno che vi accede.

È interessante aggiungere un'altra proprietà:

P3 → No Starvation → nessun processo può rimanere bloccato per sempre nella sua trying section.

Da notare che NS ⇒ ND.

Assunzioni riguardanti l'accesso alla memoria e tipo di scheduler:

A1 → i processi leggono e scrivono variabili condivise contenute in un certo indirizzo di memoria

A2 → lettura e scrittura sono operazioni atomiche → perciò il processo che sta eseguendo la read e la write, lo scheduler non può sosponderlo

A3 → lo scheduler è fair → entro un certo tempo finito, ogni processo sarà scelto dallo scheduler e avrà la possibilità di progredire

Nessuna assunzione viene fatta sulla velocità dei processi e sul tempo che i processi ci mettono ad eseguire una singola azione atomica. Questa è la tipica situazione di più processi che condividono una parte della stessa memoria.

Algoritmo di Dijkstra (1965)

Dijkstra nel 1965 formulò il problema scritto sopra e ne trovò anche una soluzione, instanziò, quindi, il trying protocol e l'exit protocol, per modelli concorrenti.

Lo scheduler è centralizzato e permette ad un solo processo alla volta di entrare in esecuzione e quindi di evolvere secondo il suo codice. Di fatto quindi lo scheduler esegue una linearizzazione di tutte le istruzioni elementari effettuate dai vari processi. La linearizzazione creata dalla singola esecuzione dell'algoritmo è chiamata "schedule".

Shared variables

```
x[1..n]: array of Boolean, initially all false
y[1..n]: array of Boolean, initially all false
k: integer in range 1..N, initially any value in its range
```

Local variables

```
j: integer in range 1..N
```

repeat

```
1      NCS
2      y[i]:= true                      % inizio trying protocol %
3      x[i]:= false
4      while k≠i do                    % ciclo della sentinella (k) → busy waiting %
5          if not y[k] then k:=i
6          x[i]:= true
7          for j:= 1 to n do
```

```

8      if i≠j and x[j] then goto 3 % fine trying protocol %
9      CS
10     y[i]:=x[i]:= false;           % exit protocol %
forever

```

I processi sono considerati paritari, e grazie all'algoritmo si riesce a trovare un modo per coordinarli.

Osservazione preliminare:

Si può vedere che le prime 2 operazioni sono di assegnazione, quindi scrittura, su due caselle di due array identificate da i (id processo). Nella istr. 4 viene introdotta la variabile k, che viene confrontata, letta, con l'id di processo. Nella 5 viene fatta un'operazione di lettura e una di scrittura. Nella 6 di scrittura...

L'osservazione è nel fatto che le operazioni di scrittura sono fatte su variabili collegate all'id del processo, quindi su quelle che si trovano nel loro spazio di memoria, quelle di lettura sono fatte su tutte le componenti. Solo una variabile può essere scritta e letta da tutti → k.

Funzionamento:

Quando il processo i vuole entrare in sezione critica prova prima ad impostare k al suo id i, passando quello che si chiama il ciclo di sentinella. Il valore di k ci dice chi è l'ultimo processo (pi) che è uscito dal ciclo della sentinella. Da notare che ciò non significa necessariamente che pj è l'unico processo che supera il ciclo.

Nel secondo ciclo (for) si controlla se ci sono stati passaggi concorrenti attraverso il primo, per far questo si deve eseguire una ulteriore scrematura per permettere ad un solo processo di entrare in sezione critica tra quelli passati per il ciclo della sentinella.

Esempio:

Consideriamo infatti un sistema di 3 processi, p1, p2 e p3. Tutti e tre i processi iniziano ad eseguire la loro trying section. P1 entra nel ciclo della sentinella (esegue il test sul while) e poi viene interrotto dallo scheduler che mette in esecuzione p2. Anche p2 entra nel ciclo e viene bloccato dallo scheduler. Infine viene attivato p3 che esegue con successo tutto il ciclo della sentinella (settando il valore di k a 3). A questo punto lo scheduler sveglia prima p1 e poi p2, quindi quando tutti e tre i processi sono usciti dal ciclo della sentinella k ha assunto il valore 2.

Diversi casi possono accadere una volta p1, p2 e p3 superando il ciclo della sentinella. Analizziamo i due casi estremi particolarmente significativi. Se grazie alle attivazioni dello scheduler un processo, per esempio p1, riesce ad eseguire con successo il ciclo for (ciò significa che p2 e p3 sono stati bloccati dallo scheduler prima di eseguire l'istruzione 6), allora entrerà nelle sua CS indipendentemente dal fatto che k è stato settato da p2. Nel caso in cui tutti e tre i processi non riescono a completare il ciclo for, ritorneranno alla linea 3 (goto statement). A questo punto mentre p1 e p3 si bloccheranno nel ciclo della sentinella, p2 sarà l'unico processo in grado di superarlo e quindi di portare con successo l'ingresso in sezione critica poiché p2 sarà l'unico processo ad eseguire il ciclo for.

La forza di questo codice è proprio in k. Infatti molti processi la settano, ma solo l'impostazione dell'ultimo rimarrà. Possiamo parlare di "ultimo" in quanto lo scheduler esegue un processo per volta. Comunque è bene notare che non sempre entra in sezione il processo che ha settato la variabile alla fine.

Questo algoritmo soddisfa P1, P2 ma non P3.

Prova P1 → Si basa sul fatto che l'ordinamento delle operazioni dentro ad un processo è totale. Per contraddizione supponiamo che due processi i e j siano in sezione critica. Poiché i è in sezione critica, precedentemente ha trovato $x[j]=\text{false}$ durante il test di linea 8. Questo implica che p_i ha eseguito la linea 8 prima che p_j eseguisse la linea 6, pertanto $i.8 \rightarrow j.6$. Chiaramente $i.6 \rightarrow i.8$ quindi si ha $i.6 \rightarrow j.6$. Scambiando i e j e seguendo lo stesso ragionamento otteniamo $j.6 \rightarrow i.6$ pertanto $i.6 \rightarrow i.6$, ovvero una contraddizione (un'operazione non può precedere se stessa). •

Prova P2 → Supponiamo per contraddizione che esiste un deadlock che coinvolge un insieme di processi D (ovvero tutti i processi in D sono bloccati nella trying e nessuno può entrare nella propria CS). Per tutti i processi i' in D , $y[i']=\text{true}$. Prendiamo il processo i che fa l'ultima assegnazione della variabile k (linea 5). Ovvero dopo questa assegnazione $k=i$ per sempre. Questo processo i appartiene a D altrimenti qualche altro processo j in D troverebbe $y[i]=\text{false}$ e quindi porrebbe a j il valore della variabile k a linea 5. A questo punto ogni processo i' in $D \setminus \{i\}$ avrà prima o poi (eventually) $x[i']=\text{false}$, di conseguenza questi processi si bloccano nel while (linee 4-5). Inoltre la variabile $x[i']$ sarà prima o poi falsa anche per tutti quei processi che non appartengono a D e che sono in NCS. Consideriamo ora il processo i . Questo processo salta il ciclo while (linea 4) poiché $k=i$ ed entrerà in sezione critica poiché tutti i valori $x[i']$ dei processi diversi da i sono falsi. Ma allora il processo i non può appartenere a D , contraddizione all'ipotesi che i deve necessariamente appartenere a D . •

Prova P3 → Essa non viene rispettata in quanto non si ha la sicurezza che prima o poi tutti i processi entrino in SC. Infatti, anche se lo scheduler è Fair, il processo potrebbe essere fermato in modo che non riesca mai a settare k per ultimo, o comunque, in modo che trovi libero la SC. Ciò è legato al fatto che non viene usato un meccanismo deterministico nella scelta del processo da far entrare. Il compito dello scheduler è far fare infinite istruzioni a tutti i processi in un tempo infinito, senza prediligere nessuno.

Il primo algoritmo derivato da quello di Dijkstra e starvation-free è dovuto a Knut (1966).

Domanda: Un processo che setta con successo la variabile k quante volte può eseguire l'istruzione "goto 3" prima di entrare in sezione critica?

Domanda: Fornire un particolare scenario in cui si viola NS.

L'algoritmo del panettiere di Lamport (1975)

Nell'algoritmo di Dijkstra si assume che le read e le write siano atomiche ovvero o si esegue una o l'altra. Inoltre poiché esiste una variabile che viene letta e scritta da tutti (variabile k), questo imporrebbe che processi accedano alla stessa memoria fisica. Se si ragiona in un sistema con singola CPU (a livello hardware) a memoria unica questo modello è assolutamente realistico. Una CPU o organizza un accesso in memoria in scrittura o in lettura. Se pensiamo ad un livello di astrazione più alto per esempio un sistema distribuito classico basato su reti di calcolatori o ad un sistema multiprocessore in cui ogni processore ha una memoria locale dove tiene le proprie variabili, questa assunzione comincia ad essere un po' troppo forte. Potremmo avere copie di una stessa variabile su diverse macchine o accessi concorrenti in lettura ed in scrittura ad una variabile condivisa o file system che permettono ad utenti di leggere e scrivere file contemporaneamente. Questo potrebbe portare una operazione di write a prendere un tempo considerevole per la sua esecuzione su diverse macchine.

Quindi, ciò che si propone di fare questo algoritmo è di risolvere il problema della mutua esclusione in sistema multiprocessore.

Le assunzioni su cui si basa Lamport sono:

- (L1) La lettura e la scrittura di una variabile non è una azione atomica. Uno scrittore potrebbe scivere mentre un lettore sta leggendo e nessuno (lettore o scrittore) viene notificato di tale interferenza.
- (L2) Ogni variabile condivisa è di proprietà di un processo. Questo processo è l'unico che può modificarla, tutti gli altri possono solo leggere (differentemente dalla variabile k dell'algoritmo di Dijkstra).
- (L3) Nessun processo può emettere due scritture concorrentemente
- (L4) Le velocità di esecuzione dei processi sono non correlate. In un tempo infinito ogni processo esegue infiniti step elementari mentre in un tempo finito esegue un numero finito di passi.

Le proprietà che dovrà rispettare sono P1 (mutua esclusione) e P3 (no starvation).

L'algoritmo si basa su un semplice concetto: quando vado in un panificio, prendo un numeretto che è più grande di uno rispetto a quello posseduto dall'ultimo in fila → ogni processo ha una propria variabile, leggibile da tutti ma modificabile solo da lui, quando la deve settare, legge tutti i numeretti degli altri, estrae il massimo e aggiunge 1.

Shared variable

```
num[1..n]: array of integer, initially all 0  
choosing[1..n]: array of Boolean, initially all false  
%process i owns num[i] and choosing[i]%
```

Local variable

j: integer in range 1..N

repeat

```
1      NCS  
2      choosing[i]:= true  
3      num[i]:= 1+ max {num[j] : n≥ j ≥1}           %DOORWAY%  
4      choosing[i]:= false  
5      for j:= 1 to n do begin  
6          while choosing[j] do skip  
7          while num[j] ≠0 and {num[j], j } < {num[i],i} do skip           %BAKERY%  
8      end  
9      CS  
10     num[i]:=0;
```

forever

L'algoritmo del Panettiere (1979)

In questo algoritmo la trying section è divisa in due parti: la *doorway* (da linea 2 a linea 4) e il *bakery* (dal linea 5 a linea 8). Quando un processo entra nella doorway lo segnala agli altri attraverso la variabile choosing. Nella doorway il processo i legge tutti gli ultimi numeri usati dagli altri processi durante la loro ultima richiesta di accesso e definisce il numero di sequenza della sua corrente richiesta (linea 3). Le cose vanno come se un cliente entra in una panetteria portando da casa un biglietto eliminacoda in bianco. Il cliente è in grado di leggere il numero di

attesa di ogni cliente. A questo punto scrive nel proprio biglietto un valore più grande tra quelli letti. Chiaramente si deve avere ben presente che nella doorway possono esserci più processi contemporaneamente.

Uscito dalla doorway il processo i si avvicina al banco del panettiere (bakery section). A questo punto deve assicurarsi che tra i processi che stanno in attesa lui è il prossimo a dover entrare nella sezione critica. Il ciclo **while** serve proprio a questo. Il processo i cicla fino a quando non è sicuro che ogni altro processo j o (a) non è nella doorway (quindi, controlla che nessuno stia facendo un'operazione di scrittura) o (b) ha un numero di coda maggiore del suo, o se lo ha uguale (caso in cui i e j hanno eseguito il settaggio di num concorrentemente) $j > i \rightarrow$ vince il processo con la coppia più piccola. Il primo while è necessario in quanto, se j viene fermato mentre stava per scrivere il suo num, i, che è uscito dalla doorway, non sa se j avrà un num più grande. Ciò è legato allo scheduler fair. Il secondo while blocca il processo i sulla prima j che ha un num più piccolo, fintanto che questo non sia entrato ed uscito dalla SC.

A questo punto il processo i accede alla sezione critica. Uscendo dalla sezione critica il processo i cancella dal suo biglietto di coda il numero usato scrivendo zero.

P1 deriva dalla seguente proprietà: "*se un processo i è nella doorway ed un processo j è nella bakery section allora $\{num[j], j\} < \{num[i], i\}$* ". Quindi se due processi i e j sono nella CS contemporaneamente, allora per la precedente proprietà otteniamo $\{num[i], i\} < \{num[j], j\}$ e $\{num[j], j\} < \{num[i], i\}$ chiaramente un assurdo. (La prova formale non è in programma). Comunque è da sottolineare che anche l'istruzione 6 ha un ruolo importantissimo. Infatti se i entra in doorway, trova il max ma poi viene sospeso. Nel frattempo j setta num, con un numero maggiore, e verifica il while (b). Il processo j non sa che i ha un num minore che sta finendo di scrivere, quindi supera i controlli ed entra. Lo scheduler riattiva i, il quale finisce di settare num e supera, di diritto, il while \rightarrow i e j si ritrovano contemporaneamente in SC \rightarrow choosing è un flag necessario.

P3 è garantita dal fatto che nessun processo attende per sempre poiché prima o poi avrà il numero di attesa più piccolo \rightarrow se un processo entra nella doorway, allora prima o poi entrerà in sc.

Dijkstra si basava sul concetto dell'atomicità della write e della read, quindi sull'ordine totale che ne derivava. Lamport, invece, ha considerato sistemi in cui la write poteva sovrapporsi ad una read, quindi necessitava di crearsi un ordine. Ci è riuscito grazie alla variabile num.

Domanda: indicare il range della variabile num[i] nell'algoritmo del panettiere

L'algoritmo del panettiere gode, inoltre, di un'altra proprietà:

FCFS(first-Come-First-Served): Se il processo i entra nella sezione bakery prima che j entra nella doorway allora i entrerà in sezione critica prima di j.

Domanda: provare formalmente che l'algoritmo del panettiere assicura FCFS.

Domanda: perchè non vale la seguente (più intuitiva) nozione FCFS:

if il processo i entra nella doorway prima di j allora i entrerà in sezione critica prima di j

L'algoritmo del panettiere vale anche sui modelli monoprocessori, infatti è più debole rispetto a quello di Dijkstra.

L'algoritmo del panettiere in ambiente distribuito Client/Server

In questo caso i processi diventano separati spazialmente, cioè girano su due macchine differenti. Quindi non sono più divisi da un bus, bensì dalla rete. Per leggere il valore di una variabile un processo deve esplicitamente inviare un messaggio di richiesta ed attendere una risposta che conterrà questo valore.

Local variable

j: integer in range 1..N; num: integer, initially all 0;

choosing: Boolean, initially false;

repeat

```
1      NCS
2      choosing:= true                      %inizio doorway%      %inizio trying%
3      for j#i do
4          send num to pj                  //richiesta
5          receive reply(v) from pj
6          num:=max(v,num)
7          num:= num+1
8          choosing:= false                 %fine doorway%
9          for j:= 1 to n do begin           %inizio bakery%
10         repeat
11         send choosing to pj
12         receive reply(v) from pj
13         untill v
14         repeat
15         send num to pj
16         receive reply(v) from pj
17         untill v=0 or {num,i}>{v, j }
18         end                           %fine bakery%  %fine trying%
19         CS
20         num:=0;                      %exit protocol%
```

forever

Compiti gestiti dai thread:

Upon the arrival of a num message from process j

send reply(num) **to** process j

Upon the arrival of a choosing message from process j

send reply(choosing) **to** process j

Algoritmo del panettiere in un sistema distribuito non cooperante

Di fatto ogni processo si comporta da server rispetto alle variabili num e choosing di sua proprietà e risponde alle richieste di lettura di client (altri processi).

Oltre alle assunzione L1-L4, in questo ambiente si aggiungono le seguenti assunzioni:

- (A1) I processi comunicano leggendo e scrivendo variabili attraverso scambio di messaggi. Il ritardo di trasmissione di un messaggio è impredicibile ma finito;
- (A2) I canali di comunicazione sono affidabili. Un messaggio inviato viene ricevuto correttamente dal giusto destinatario. Inoltre non ci sono duplicazioni o messaggi spuri (ricevuti ma mai trasmessi)

Per l'assunzione A1 cambiano radicalmente le operazioni alle linee 3, 6 e 7 dell'algoritmo presentato nella precedente sezione. In questo caso un messaggio deve essere inviato ai vari processi per avere il valore della variabile corrispondente. I quali rispondono inviando il valore della variabile. Quindi il codice diventa quello riportato nella pagina seguente. Il codice della doorway va da riga 2 a riga 8 mentre quello del bakery da linea 9 a linea 20.

Concettualmente, funziona come l'algoritmo di prima ma la richiesta del num e del choosing avviene tramite messaggi. Differenze:

Dalla 3 all 8 → riga 3 → doorway aumenta in quanto è necessario chiedere a tutti i processi il proprio num
Dalla 9 alla 13 → righe 5 e 6 → non so quanti messaggi saranno scambiati, il processo verrà sbloccato quando tutti sono usciti dalla doorway
Dalla 14 alla 17 → riga 7

L'esistenza di messaggi implica l'inserimento di thread concorrenti al codice di mutua escusione in grado di rispondere alle richieste esterne. L'esecuzione di un tale algoritmo genera un message pattern simile a quello generato mostrato nella figura successiva.

Message pattern dell'algoritmo del bakery in ambiente distribuito non-cooperante

I messaggi portano anche ad avere una computazione pesante, infatti oltre al codice da eseguire, vi è la gestione dello scambio di informazioni. Su di essi viene calcolata la complessità, infatti rappresentano l'istruzione più gravosa. Nell'algoritmo semplice, dove si lavorava all'interno di uno stesso computer, leggere il num e il choosing richiedeva un tempo dell'ordine del nanosecondo, ora, tramite messaggi, si parla di millisecondo. Questo comporta ad un rallentamento notevole legato anche al fatto che, a priori, non si sa il numero dei pacchetti scambiati. Per i sistemi distribuiti, quindi, è un algoritmo non efficiente. Si è cercato di creare un pattern ad hoc basato sulla cooperazione. Infatti prima, il processo leggeva le variabili e poi autonomamente sceglieva come agire, gli altri processi non hanno un ruolo attivo in merito se non quello di inviare un valore.

doorway

L'algoritmo del panettiere in ambiente distribuito peer-to-peer

Questo algoritmo è stato scritto da Ricart-Agrawala nel 1981.

p1

p2

p3

Un approccio più adatto a questo tipo di ambiente è quello in cui i processi cooperano in modo esplicito per fare accedere ogni singolo processo alla sezione critica senza l'uso di un coordinatore centrale (peer-to-peer). Lo scopo di questa cooperazione è di avere dei message pattern più snelli in termini di messaggi scambiati per accesso alla sezione critica. La cooperazione tra i processi, per esempio, può ribaltare lo schema di scrittura del biglietto eliminacode. Si pensi ad esempio ad un cliente che entri dentro il negozio del panettiere con un numero già in tasca ed aspetti un invito esplicito da parte di tutti i clienti che individuano in lui il prossimo cliente che deve essere servito.

A questo punto il problema diventa come assicurare che il numero che porta da casa il cliente rispetta un ordine totale imposto dall'accesso in mutua esclusione della propria sezione critica. Questo può essere realizzato pensando alla variabile num come unica nel sistema di cui ogni processo ne possiede una copia ed ogni processo la aggiorna e comunica questo aggiornamento in modo che anche gli altri possano farlo. Mentre nel bakery originale colui che sta entrando nella doorway vuole leggere i valori dei biglietti eliminacode degli altri processi per scrivere il suo numero, nei sistemi distribuiti cooperanti un processo quando entra nella doorway (linea 1-3) comunica agli altri il suo numero. Chiaramente per fare ciò, bisogna rispettare un ordinamento totale; quando questo processo riceve messaggi di altri che desiderano entrare nella doorway deve mantenere in memoria il massimo numero visto. A questo punto la condizione di accesso non è computata in modo locale, come nel caso precedente. Una volta recuperate tutte le informazioni (num e choosing) dagli altri processi, la condizione (linea 4) è dovuta ad un messaggio di consapevole riscontro di ciascuno degli altri processi (messaggi di reply). Questi processi in piena autonomia quando vedono la richiesta di un certo nodo i essere in testa all'ordine totale (ovvero la prossima richiesta che deve essere servita al banco del panettiere) inviano un REPLY al processo i.

Ricapitolando, l'obiettivo è avere un algoritmo che permetta di far entrare in SC un processo alla volta, tramite un numero deterministico di messaggi (dell'ordine di $n \rightarrow$ numero processi), e che venga realizzato grazie alla cooperazione. In più devono essere rispettate P1 (mutex) e P3 (no starvation).

Esempio dell'idea: appena il processo con num=1 esce da SC invia un messaggio con scritto "uscito 1". I processi settano num=2 e quello interessati ad entrare in sezione, inviano una richiesta. Esso, per entrarem deve ricevere un messaggio di conferma da tutti gli altri.

Calcolo numero messaggi necessario $\rightarrow 2(n-1)$ [numero mex, ricevuti ed inviati, necessari per entrare in SC] + $(n-1)$ [mex inviati all'uscita da SC] = $3(n-1)$

Num viene costruito insieme, anche se un processo non entra da tanto tempo in SC.

Local variable

```
#replies: integer initially set to zero
state: in set {requesting, CS, NCS} initially set to NCS
Q: queue of pending request {T,i} initially set to empty
Last_req: integer ;
Num: integer initially set to zero

begin
    1      state:= requesting
    2      num:=num+1; last_req:=num;
```

```

3   send REQUEST(last_req) to p1..pn
4   wait until #replies=(n-1)
5   state:= CS
6   CS
7   send REPLY to any request in Q; Q:=∅; state:=NCS; #replies:=0
end

```

Handler:

Upon the receipt of REQUEST(t) from process j

```

8   if state=CS or (state=requesting and {last_req, i } < {t,j})
9   then insert.Q({t, j})
10  else send REPLY to Pj
11  num:=max(t,num)

```

Upon the receipt of REPLY from process j

12. #replies++;

Algoritmo di Ricart-Agrawala

Modello di sistema:

1. I messaggi sono affidabili → ciò che viene inviato, arriva sempre a destinazione, con un tempo finito ma impredicibile, dipende da quanto sono lontani i terminali
2. Le variabili sono proprietà dei processi → non possono essere né letti né scritti da altri
3. Lo scheduler è fair → legato al fatto che gli handler sono gestiti da due thread

Lo scambio di messaggi potrebbe avvenire tramite socket, e la loro gestione potrebbe essere fatta da dei thread concorrenti.

Situazione iniziale: abbiamo 3 processi (p1, p2, p3) che si trovano su 3 terminali diversi.

- Inizialmente num viene settato da tutti con 0.
- p1 invia una request REQ(1) a p2 e p3 che settano num=1 (coscienza del sistema)
- p2 e p3 si chiedono: sono richiedente? No, sono in sezione critica? No
- p2 e p3 inviano una REPLY
- Solo quando tutte le REPLY sono arrivate, p1 entra in SC
- Se p3 fa una richiesta, il num che invia, sarà pari a 2

Quando il processo i ha collezionato un REPLY da ogni altro processo può entrare in CS. Per evitare violazione di P1 un processo che è (1) in sezione critica o (2) in attesa ma con un numero di attesa inferiore a quello della richiesta del processo i, ritarda l'invio del messaggio di reply fino alla sua uscita dalla sezione critica. Queste richieste ritardate vengono inserite in una coda Q.

Caso in cui arriva una richiesta mentre un processo è in SC:

- p2 invia una request REQ(1)
- p3 setta num e si chiede: sono richiedente? No, sono in SC? No → invia reply
- p1 setta num e si chiede: sono richiedente? No, sono in SC? Si → inserisce REQ(2) nella coda Q

- Appena p1 esce da SC ed invia una reply a qualsiasi REQ in Q → spedisce REPLY a p2
- p2 entra in SC

Caso in cui vengono inviate più request contemporaneamente:

- p1 e p3 inviano REQ(1), rispettivamente, a p2 e p3 e p1 e p2
- p2 setta num, si fa le solite domande ed invia una REPLY a tutti e due
- p3 setta num e si chiede: sono richiedente? Si → il mio num è più piccolo? No, sono uguali → il mio identificatore è più piccolo? No → invia REPLY a p1
- p1 fa lo stesso procedimento di p3, ma scopre di avere l'identificativo più piccolo → inserisce la REQ di p3 in Q
- p1 entra in SC e quando esce risponde a p3
- p3 entra in SC

Quanti messaggi sono stati scambiati per far entrare in SC un processo? Sempre 2(n-1)!!!

Differenza, da sottolineare, con il bakery semplice è la costruzione di num → nessuno entra nella bakery section senza num, che è più grande dell'ultima richiesta vista.

Assumiamo che la linea (istruzione) 2 venga eseguita atomicamente.

Prova P1 → (Per contraddizione). Supponiamo che i e j siano in CS contemporaneamente, allora i ha inviato un REPLY a j e viceversa. Sono possibili tre casi analizzando il message pattern associato all'algoritmo.

Caso 1. Il processo i spedisce il REPLY prima di scegliere il proprio num alla linea 2 (timestamp della richiesta). Quindi la richiesta di i avrà necessariamente un timestamp più alto di quello della richiesta di j. Poichè i sulla ricezione della richiesta di j avrà eseguito linea 11 e prima di spedire la richiesta i eseguirà linea 2. Una volta che la richiesta di i arriva a j, quest'ultimo metterà la richiesta in coda poichè o e' già nella propria CS oppure è nello stato requesting ma il timestamp associato alla sua richiesta è minore di quello di i. Di conseguenza i e j non possono essere entrambi in CS

Caso 2. Identico al caso 1 invertendo i e j.

Caso 3. i e j spediscono un REPLY all'altro dopo aver scelto il timestamp per la propria richiesta. In questo caso la regola di ordinamento delle richieste è unica e deterministica in ogni processo (se hanno stesso timestamp si confrontano gli identificatori). Quindi quando viene ricevuto il messaggio di richiesta, uno dei due spedisce un REPLY e l'altro metterà la richiesta in coda (test di linea 8). Di conseguenza non possono essere entrambi in CS.

Caso 4. Assumiamo che la linea 2 non venga eseguita atomicamente. Lo scheduler potrebbe sospendere il processo in esecuzione, cambiando il valore di num e last req provocando la violazione della mutua esclusione. •

Prova P3 → Assumiamo che esista un processo i che attende indefinitivamente l'accesso in sezione critica (linea 4) dopo aver inviato la richiesta con timestamp num_i. Poichè ogni messaggio arriva a destinazione in un tempo finito, ciò significa che esiste un set non vuoto di processi S che non invia il messaggio di REPLY a i. Sia j un processo appartenente a S. Questo accade solo se j è in sezione critica o è in attesa e la sua richiesta ha priorità rispetto a i ovvero {num_j, j} < {num_i, i}. Nel primo caso in un tempo finito j esce dalla sezione critica e invia un REPLY a i sbloccandolo e contraddicendo l'assunzione iniziale. Nel secondo caso esiste un processo k appartenente a S \ {j}

che blocca j. Usando gli stessi argomenti, si definisce una sequenza di attesa di lunghezza pari alla cardinalità di S ($\max n$ processi) $\langle i, j, k, \dots, u \rangle$ tale che un processo blocca il suo precedente nella sequenza e la richiesta del successivo ha priorità sul precedente quindi per transitività $\{num_u, u\} < \{num_i, i\}$ ($\rightarrow S$ viene linearizzato). In questa sequenza l'ultimo processo u è bloccato necessariamente da un processo che lo precede nella sequenza. Sia i questo processo, quindi $\{num_i, i\} < \{num_u, u\}$ da ciò segue l'assurdo $\{num_i, i\} < \{num_i, i\}$. •

Domanda. Consideriamo un sistema in cui al più k processi possono essere nelle loro sezioni critiche contemporaneamente (k-mutex). Come cambia il precedente algoritmo per migliorare il grado di concorrenza?