

## Sincronizzazione nei sistemi distribuiti

## Sommario

- Sincronizzazione dei clock
- Mutua esclusione
- Transazioni
- Deadlock
- Le tecniche adottate per i S.O. tradizionali non vanno bene
  - Suppongono la presenza di memoria condivisa!

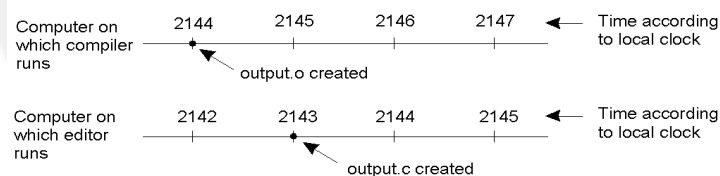
## Proprietà degli algoritmi distribuiti

- Le informazioni sono distribuite fra molte macchine
- I processi prendono decisioni in base alle informazioni locali
- Evitare la presenza di un “unico punto di fallimento”
- Non esiste clock comune

## SINCRONIZZAZIONE DEI CLOCK

## Sincronizzazione dei clock

- Quando ogni host possiede un proprio clock, l'ordine degli eventi può essere erroneamente invertito
- Esempio
  - Iterazione editing/compilazione + make



**Necessario meccanismo di sincronizzazione**

5

## Assunzioni

- Il sistema distribuito è composto da un insieme di processori  $P = \{P_1, P_2, \dots, P_n\}$ 
  - $P_i$  reagisce ad eventi
    - Eventi esterni: invio, ricezione di messaggi
    - Eventi interni: I/O locale, arrivo di segnali, ...
- $E$  = insieme dei possibili eventi nel sistema
- $E_{P_i}$  = insieme dei possibili eventi in  $P_i$
- Vogliamo definire un ordine per  $E$

6

## Sincronizzazione dei clock

- Si possono sincronizzare tutti i clock di un sistema?
  - In linea di principio, NO
    - Necessario un concetto di tempo globale
    - Non realizzabile
      - No clock globale
      - Ritardo di comunicazione (non deterministico) tra nodi
  - Possibile una versione “rilassata” di sincronizzazione
    - Basata sul concetto di tempo logico
      - Sorta di tempo relativo (non assoluto)

7

## Clock logici o virtuali

- Non necessariamente legato al tempo effettivo (reale)
- Sincronizzazione di clock logici è più semplice
  - Idea: l'ordine degli eventi è più importante del tempo esatto in cui gli eventi si verificano
- Modello del sistema
  - Esecuzione dei processi = sequenza di eventi
  - Granularità = evento (istruzione, chiamata a procedura, invio di un messaggio ...)

8

## Relazione “ $\rightarrow$ ”

- Cattura dipendenze causali tra eventi
- Relazione “è avvenuto prima” (*happened-before*)
- Definizione
  - Se A e B sono eventi nello stesso processore, e A è stato eseguito prima di B, allora  $A \rightarrow B$ 
    - Nello stesso processore gli eventi sono totalmente ordinati
  - Se A è l'evento corrispondente a mandare un messaggio da parte di un processore, e B è l'evento corrispondente a ricevere il messaggio da parte di un altro processore, allora  $A \rightarrow B$
- Evento A ha effetto causale su B se  $A \rightarrow B$

9

## Relazione “ $\rightarrow$ ”

- $\rightarrow$  è una relazione di ordine parziale irreflessiva
  - Irriflessività
    - $\neg (A \rightarrow A)$
  - Asimmetria
    - se  $A \rightarrow B$  allora  $\neg (B \rightarrow A)$
  - Transitività
    - se  $A \rightarrow B$  e  $B \rightarrow C$  e...e  $V \rightarrow Z$  allora  $A \rightarrow Z$

10

## Relazione “ $\rightarrow$ ”

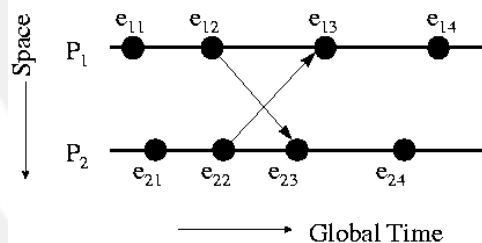
- Sono ammessi eventi concorrenti:
  - $A \parallel B$  se  $\neg (A \rightarrow B)$  e  $\neg (B \rightarrow A)$
  - Quando due processi su processori diversi non si scambiano messaggi
- Per ogni coppia di eventi A e B in un sistema
  - $A \parallel B$  o  $A \rightarrow B$  o  $B \rightarrow A$

11

## Relazione “ $\rightarrow$ ”

- Esempio (diagramma spazio-temporale)

- $e_{12} \rightarrow e_{23}$
- $e_{22} \rightarrow e_{13}$
- $e_{23} \rightarrow e_{24}$
- $e_{12} \rightarrow e_{24}$
- $e_{11} \parallel e_{22}$



12

## Implementazione di “ $\rightarrow$ ”

- Obiettivo:
  - per ogni evento  $a$ , disporre di un valore di tempo  $C(a)$  su cui ogni processore è d'accordo
- Sistema di clock logici [Lamport 78]
- Vettori di clock [Fidge 91 / Mattern 88 / Raynal&Singhal 96]

13

## Clock logici di Lamport

- Ogni processore  $P_i$  possiede un clock  $C_i$
- $C_i$  assegna dei valori  $C_i(a)$  (*timestamp*) ad ogni evento  $a$  di  $P_i$
- Valori di  $C_i$ 
  - Nessuna relazione con il tempo reale
  - Monotonicamente crescenti
- Implementazione tramite contatori (locali!)

14

## Clock logici di Lamport

- Condizione affinché un sistema di clock logici sia “corretto”:
  - Se  $a \rightarrow b$  allora  $C(a) < C(b)$   
(indipendentemente da dove  $a$  e  $b$  si trovano)
- Realizzabile se sono verificate le seguenti regole:
  - Condizioni di correttezza C1, C2
  - Regole di implementazione IR1, IR2

15

## Clock logici di Lamport

- Condizioni di correttezza
  - [C1] Per ogni coppia di eventi  $a$  e  $b$  in  $P_i$ , se  $a$  si verifica prima di  $b$  allora  $C_i(a) < C_i(b)$
  - [C2] Se  $a$  è l'evento corrispondente a mandare un messaggio  $m$  nel processore  $P_i$ , e  $b$  è l'evento corrispondente a ricevere il messaggio  $m$  nel processore  $P_j$  allora  $C_i(a) < C_j(b)$

16



## Clock logici di Lamport

- Regole di implementazione
  - [IR1] Il clock  $C_i$  viene incrementato tra due eventi successivi nel processore  $P_i$  come segue:
    - $C_i = C_i + d$  ( $d > 0$ )
  - [IR2] Se  $a$  è l'evento corrispondente a mandare un messaggio  $m$  da parte del processore  $P_i$ , allora  $a$  viene assegnato un timestamp  $t_m = C_i(a)$  (dopo l'applicazione di IR1)
    - Quando  $m$  viene ricevuto dal processore  $P_j$ ,  $C_j$  viene calcolato come segue:
      - $C_j = \max(C_j, t_m) + d$

17

## Esempio

P1	P2	P3		P1	P2	P3
0	0	0		0	0	0
6	8	10		6	8	10
12	16	20		12	16	20
18	24	30		18	24	30
24	32	40		24	32	40
30	40	50		30	40	50
36	48	60		36	48	60
42	56	70		42	70	70
48	64	80		48	78	80
54	72	90		86	86	90
60	80	100		92	94	100

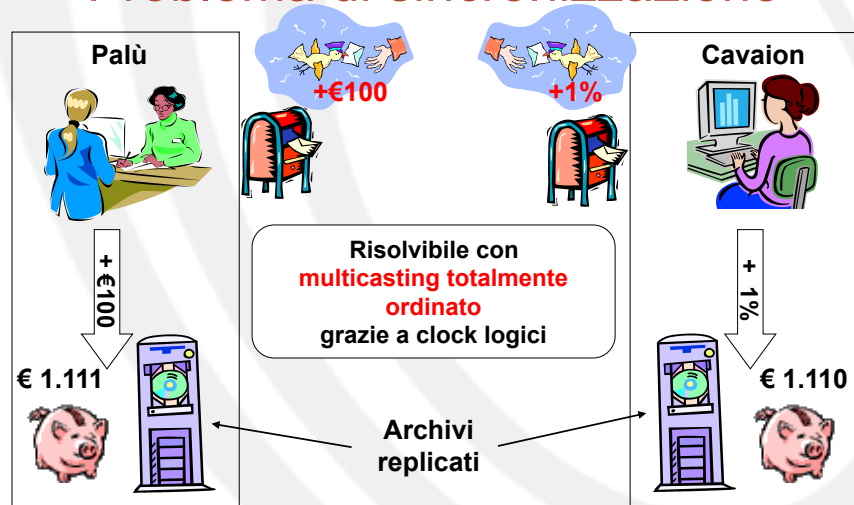
18

## Ordine totale?

- → è una relazione di ordine parziale
- Per ottenere ordine totale basta aggiungere al timestamp di un evento l'ID del corrispondente processore
- Esempio:
  - Se A accade al tempo 10 su P1, e B al tempo 10 su P2,  $A \rightarrow B$  perché  $P1 < P2$
- Ordine totale usato in molti algoritmi di sincronizzazione

19

## Problema di sincronizzazione



20

## Multicasting totalmente ordinato

- Ogni processo invia messaggi a tutti gli altri processi del gruppo (anche a se stesso)
- Ogni messaggio  $m$  ha un timestamp  $t_m$  corrispondente al clock logico del mittente al momento dell'invio
- I messaggi inviati da un processo arrivano nello stesso ordine di invio e non ci sono perdite
  - Condizione ottenibile tramite *ack* e numerazione dei messaggi
- Quando un processo  $P_i$  riceve  $m$ 
  - $m$  viene archiviato in una coda totalmente ordinata in base a  $t_m$
  - $P_i$  invia *ack* a tutti ( $t_{ack} > t_m$ )
- $m$  viene rimosso dalla coda ed elaborato da  $P_i$  solo quando si trova in testa alla coda e  $P_i$  ha ricevuto il relativo *ack* da tutti gli altri processi

**N.B: la coda è uguale per tutti i processi**

21

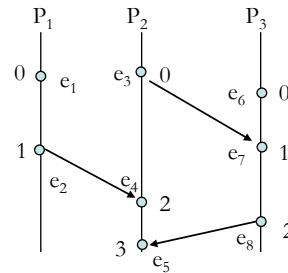
## Limitazioni di “Lamport”

- Lamport assicura che se  $a \rightarrow b$ , allora  $C(a) < C(b)$
- L'algoritmo è:
  - completamente distribuito
  - Semplice
  - tollerante ai guasti
- Ma se  $C(a) < C(b)$ ,  $a \rightarrow b$ ?
  - In generale NO! (per eventi su processi remoti)
  - Problema: vogliamo capire anche se 2 eventi sono legati da una relazione di causa ed effetto guardando i timestamp!

22

## Limitazioni di “Lamport”

- Se  $C(a) < C(b)$  allora  $a \rightarrow b$  ?
  - In generale NO!
- Lamport non cattura la seguente relazione di causalità:
  - se  $C(e_i) < C(e_j)$  allora  $e_i \rightarrow e_j$



$C(e_3) < C(e_2)$   
ma  $e_3 \not\rightarrow e_2$  !!  
In realtà  
 $e_3$  ed  $e_2$  sono  
concorrenti

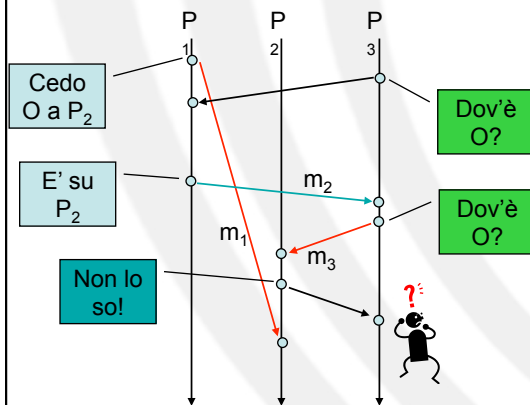
23

## Relazione di causalità

- Utile perché permette:
  - debugging
  - liveness e fairness per la mutua esclusione
  - consistenza per database replicati
  - rilevazione dei deadlock
  - costruzione di stati globali consistenti per il recovery da crash
  - determinazione del grado di parallelismo tra processi
  - ...

24

## Violazione della causalità



- $S(m)$  = evento in cui  $m$  è spedito
- $R(m)$  = evento in cui  $m$  è ricevuto
- Violazione della causalità:
  - $m_1$  arriva a destinazione dopo  $m_3$ !
  - $S(m_1) \rightarrow S(m_3)$  ma  $R(m_3) \rightarrow R(m_1)$

25

## Violazione della causalità

- Esempio tipico
  - Un processore  $P_i$  invia due messaggi  $m_1, m_2$
  - Un processore  $P_j$  riceve i due messaggi  $m_1, m_2$
  - $S(m_1) \rightarrow S(m_2)$
  - $R(m_2) \rightarrow R(m_1)$
  - In generale, per evitare la violazione della causalità dobbiamo sapere se 2 eventi sono in relazione causale tra loro
  - Soluzione: Vettori di clock logici

26

## Vettori di clock

- [Fidge 91 / Mattern 88 / Raynal&Singhal 96]
- Simili a clock logici, ma più valori (vettore) anziché uno solo
  - $n$  processori  $\rightarrow n$  elementi del vettore
  - $C_i[i]$  : tempo logico di  $P_i$
  - $C_i[j]$  : la migliore stima del tempo logico di  $P_j$  da parte di  $P_i$ 
    - $C_i[j]$  = tempo dell'ultimo evento in  $P_j$  che "è successo prima" del tempo corrente di  $P_i$

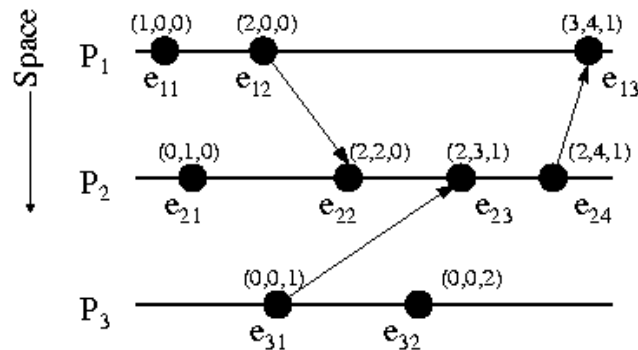
27

## Vettori di clock

- Regole di implementazione
  - [IR1] L'elemento  $i$  del clock  $C_i$  viene incrementato tra due eventi successivi nel processore  $P_i$ 
    - $C_i[i] = C_i[i] + d$  ( $d > 0$ )
  - [IR2] Se  $a$  è l'evento corrispondente a mandare un messaggio  $m$  nel processore  $P_j$ , allora
    - $a$  viene assegnato vector timestamp  
 $t_m = C_i(a) = (C_i[1], \dots, C_i[n])$  (dopo aver applicato IR1)
    - quando  $m$  viene ricevuto dal processore  $P_j$ ,  $C_j$  è calcolato come
    - $\forall k, C_j[k] = \max(C_j[k], t_m[k]) + d$

28

## Vettori di clock - esempio



29

## Vettori di clock

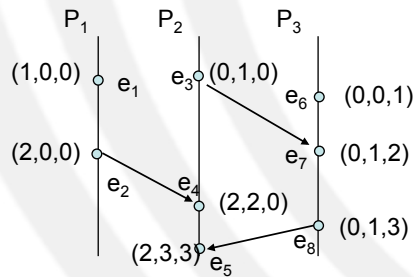
- Permette di determinare relazioni causa-effetto tra eventi
  - Non possibile con i clock di Lamport
- Definizione
  - $t_a < t_b$  se e solo se
    - $t_a[i] \leq t_b[i]$  per ogni  $i$
    - Esiste  $j$  tale che  $t_a[j] < t_b[j]$
- Teorema
  - se  $t_a < t_b$ , allora  $a \rightarrow b$  e viceversa
- Permettono di verificare qualunque tipo di ordine parziale ( $=$ ,  $\leq$ ,  $\geq$ ,  $\parallel$ )

Esempi
$(1,0,3) \rightarrow (2,0,5)$
$(1,1,3) \not\rightarrow (1,0,3)$
$(1,1,3) \not\rightarrow (1,1,3)$

30

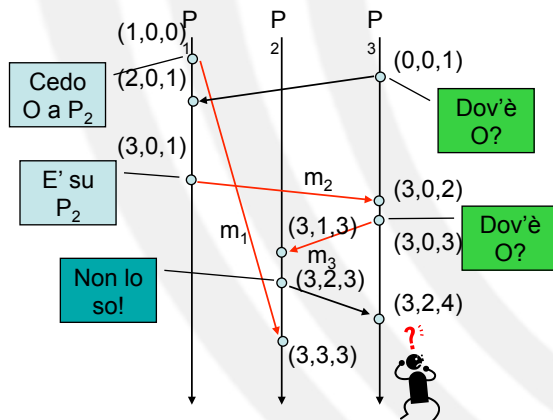
Se  $t_a < t_b$ , allora  $a \rightarrow b$

• Esempio



$C(e_3) < C(e_2)$ ?  
NO, infatti  
 $e_3 \not\rightarrow e_2$  !!  
 $e_3$  ed  $e_2$  sono  
concorrenti

Violazione della causalità



- Quando arriva  $m_1$ ,  $P_2$  capisce che c'è stata violazione della causalità perché:  
 $t_{\text{send}}(m_1) < t_{\text{send}}(m_3)$   
ma  
 $t_{\text{receive}}(m_3) < t_{\text{receive}}(m_1)$



## Clock fisici

- In molti casi è necessario che i singoli clock siano in qualche modo legati al tempo reale
  - Ambienti Real-time!
- Clock fisici vs. logici
- Come garantire la sincronizzazione di clock fisici?
  - Soluzioni centralizzate
  - Soluzioni distribuite

33

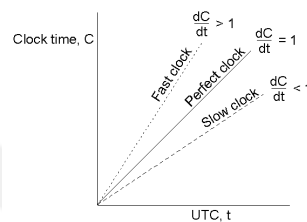
## Coordinated Universal Time (UTC)

- Standard internazionale per mantenere il tempo
- Basato su International Atomic Time (TAI)
  - 1s = tempo impiegato da un atomo di cesio 133 per compiere 9.192.631.770 transizioni
  - 50 laboratori calcolano il tempo → media = TAI
  - 86400s TAI durano 3ms meno rispetto al giorno solare medio → richiesta compensazione
- L'output dell'orologio atomico e' inviato in broadcast da stazioni radio e satelliti (GPS) sulla terra
- Segnali da stazioni radio su terra hanno un'accuratezza di circa 0.1-10 ms; segnali da GPS hanno un'accuratezza di circa 1  $\mu$ s
- Computer con ricevitori possono sincronizzare i loro clock con questi segnali → le altre macchine vanno sincronizzate

34

## Sincronizzazione di clock fisici

- Modello del sistema
  - Ogni host possiede un timer che causa un interrupt  $H$  volte al secondo (tick)
  - Quando scatta il timer viene aggiunto 1 a un clock SW  $C(t)$
- Detto  $t$  il tempo universale (UTC), idealmente  $C(t) = t$ 
  - In pratica  $dC/dt \neq 1$
  - Tolleranza  $r$ 
    - Maximum drift rate (tasso di scostamento massimo)
  - $1-r < dC/dt < 1+r$



35

## Sincronizzazione di clock fisici

- Assunzione
  - Esiste un processo (time server) che è in grado di fornire un tempo di riferimento
- Periodicamente, gli host dialogano con il time server per sincronizzarsi
- Quanto spesso?
  - Dopo tempo  $dt$ , 2 clock possono discordare di  $2r dt$
  - $r$  = tolleranza del clock
  - Se  $d$  = errore max consentito rispetto tempo universale  $t$ 
    - $2r dt < d \rightarrow dt < d/2r$
  - Allora  $Dt$  = distanza tra due sincronizzazioni =  $d/2r$  secondi

36

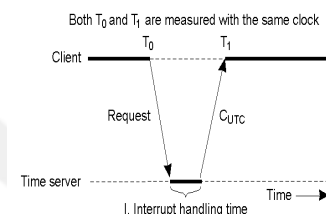
## Sincronizzazione di clock fisici

- Metodi centralizzati
  - Algoritmo di Cristian [Cristian 89]
  - Algoritmo di Berkeley [Gusella&Zatti 89]
- Metodo distribuito

37

## Algoritmo di Cristian

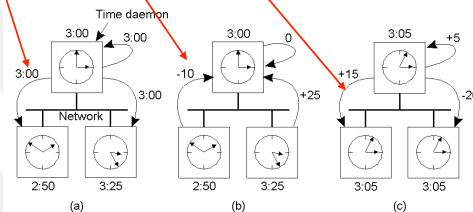
- Host del time server possiede una copia di UTC
- Periodicamente, i processi interrogano il time server
- Difficoltà
  - Processo da sincronizzare può avere clock “veloce”
    - Soluzione: rallentamento progressivo del clock da sincronizzare
  - Va considerato il tempo di propagazione della richiesta
    - La sua misurazione è possibile perché si usa il clock “locale”
    - Stima:  $(T_1 - T_0)/2$



38

## Algoritmo di Berkeley

- Time server (daemon) attivo
  - Chiede ai vari host il tempo e fornisce il proprio
  - Gli host forniscono la differenza tra il loro tempo e quello del server
  - Il daemon dice agli host come regolare i clock
    - Es: facendo una media
- Adatto per sistemi senza riferimento UTC, dove il server viene regolato a mano
- Usato nel Berkeley Unix



39

## Metodi distribuiti

- Tempo suddiviso in periodi di lunghezza fissa
- Host inviano proprio tempo in broadcast all'inizio di ogni periodo
  - Invii non simultanei a causa di differenze nei vari clock
    - Quindi la ricezione del broadcast rimane attiva per un certo tempo S
  - Quando scade S il clock locale è modificato in base a
    - Media dei valori ricevuti
    - Media dei valori, esclusi n valori estremi
  - E' possibile tener conto anche della stima del tempo di propagazione del broadcast da una data sorgente

40

## DETERMINAZIONE DELLO STATO GLOBALE

41

### Stato globale

- Problemi classici dei S.O. tradizionali si presentano anche nel caso di S.O. distribuiti
  - Mutua esclusione
  - Starvation
  - Deadlock
- Complicazione
  - Non c'è uno stato globale

42

## Stato globale

- Supponiamo di interrompere una computazione distribuita mediante interruzione simultanea di tutti i processi
- Lo stato globale è dato da:
  - stato di ogni singolo processo
  - contenuto di ogni canale di comunicazione
- Osservazione dello stato globale di un sistema distribuito può essere semplice per un osservatore esterno ma difficile all'interno del sistema

43

## Stato globale

- Esempi di casi in cui la conoscenza dello stato globale è importante:
  - Checkpoint per rollback in caso di malfunzionamenti
  - Rilevazione di proprietà stabili
    - La computazione ha terminato
    - Il sistema è in deadlock (tramite RAG)
    - Tutti i token in una rete token-ring sono spariti
  - ...

44

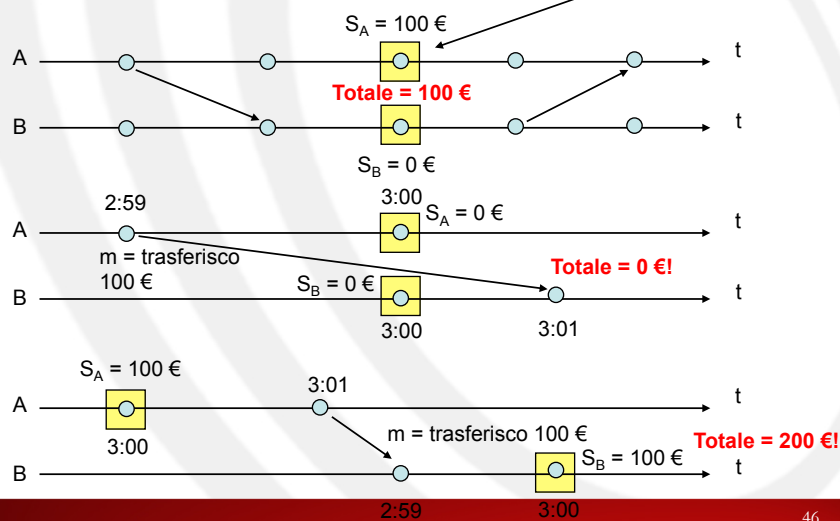
## Stato globale (difficoltà)

- Stato distribuito
  - Necessario raccogliere informazioni sparse su macchine diverse
- Conoscenza solo locale
  - Un processo non conosce lo stato di un altro processo!
- Registrazione “istantanea” impossibile
  - No clock globale
    - La registrazione degli stati locali non può essere sincronizzata in base al tempo
    - Conosciamo lo stato locale degli altri processi con un “certo ritardo”
      - Es.: Stelle lontane
- Ci accontentiamo del passato

45

## Stato globale

Snapshot



46

## Definizioni

- Canale
  - Tra P1 e P2 esiste un canale se P1 e P2 scambiano messaggi
  - Può essere visto come il cammino tramite cui i messaggi vengono trasferiti
  - È unidirezionale
- Stato locale
  - Lista di *send* e *receive* (di messaggi) in corso
  - Definito per ogni tempo locale (processo locale)
- Snapshot
  - Registra lo stato di un processo
  - Contiene record di tutti i messaggi spediti e ricevuti su tutti i canali dall'ultimo snapshot

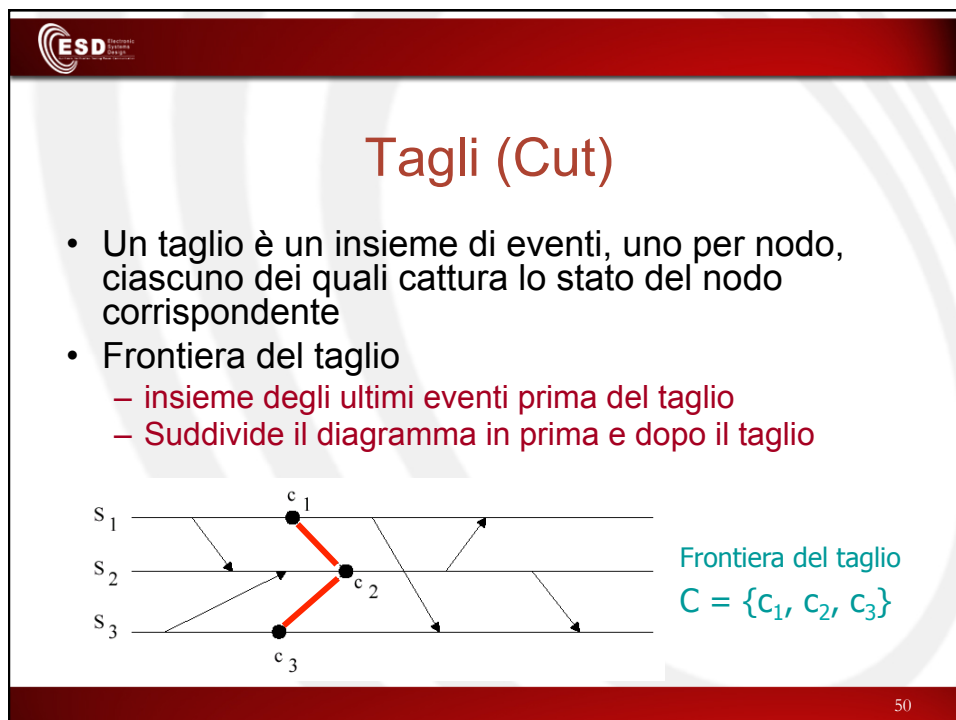
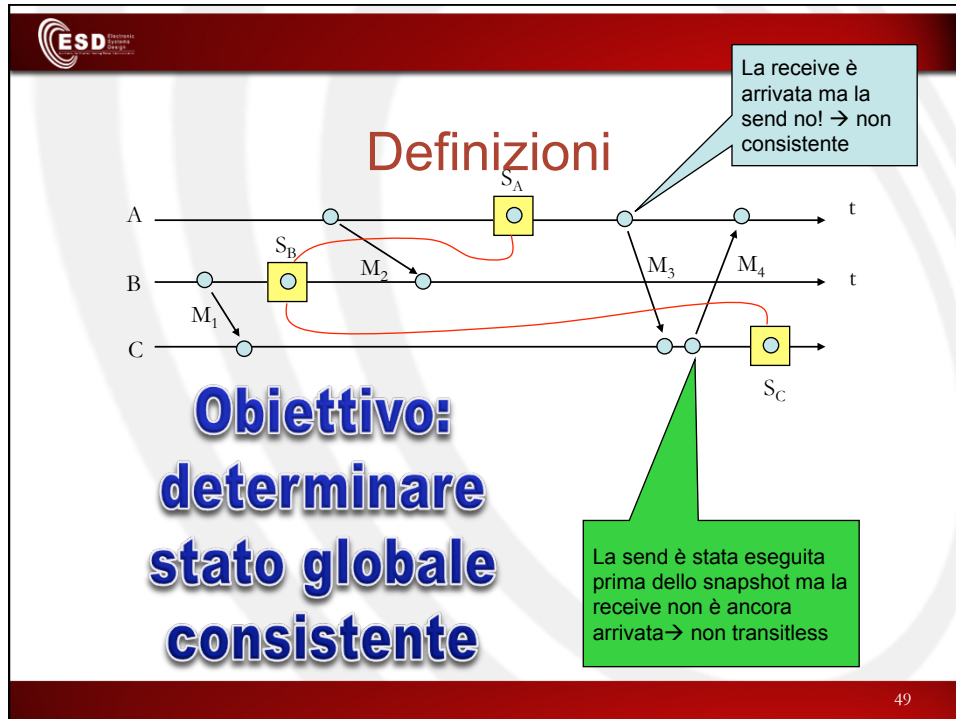
47

## Definizioni

- Stato globale
  - Stato locale di ogni processo + stato dei canali di comunicazione al momento dello snapshot
  - Consistente
    - Per ogni messaggio ricevuto “esiste” la send corrispondente
    - In pratica, ogni evento deve essere preceduto dagli eventi causalmente correlati ad esso
  - Transitless
    - Tutte le send sono state ricevute
  - Fortemente consistente
    - Consistente + transitless

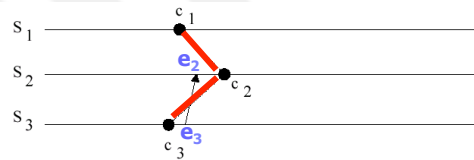
48





## Tagli

- Un taglio  $C = \{c_1, c_2, c_3, \dots\}$  si dice **consistente** se per ciascun nodo non esistono due eventi  $e_i$  ed  $e_j$  per cui  $e_i \rightarrow e_j$  AND  $e_j \rightarrow c_j$  AND  $e_i \rightarrow c_i$
- In pratica, un taglio è consistente se quando include un evento, include anche gli eventi che lo precedono
- Esempio di inconsistenza



51

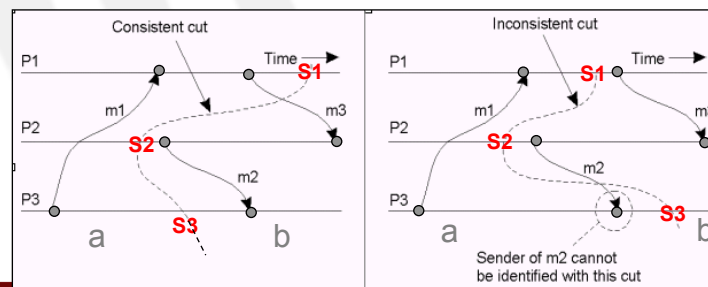
## Stato globale

- Uno stato globale è consistente se corrisponde ad un taglio consistente
- I termini stato globale, taglio o snapshot sono intercambiabili

52

## Stato globale: esempio

- $GSa = \{S1=\{(m1,R), (m3,S)\} \ S3=\{(m1,S)\}\}$
- $GSb = \{S2=\{(m2,S), (m3,R)\} \ S3=\{(m2,R)\}\}$
- $GSa = \{S1=\{(m1,R)\}, S3=\{(m1,S), (m2,R)\}\}$
- $GSb = \{S1=\{(m3,S)\}, S2=\{(m2,S), (m3,R)\}\}$



53

## Determinazione dello stato globale

- Necessario un algoritmo specifico
- Chiave
  - Essenziale mantenere lo stato dei messaggi e del canale!
- Esempio
  - Algoritmo banale
    - I processi registrano il loro stato in un istante arbitrario
    - Un processo designato raccoglie i vari stati
  - Semplice, ma non corretto!

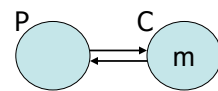
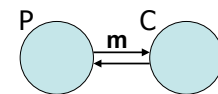
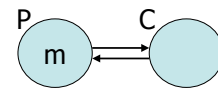
54

## Determinazione dello stato globale

- Esempio:

- produttore-consumatore

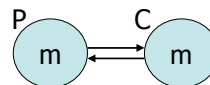
- P registra il suo stato
- Il messaggio transita sul canale
- C riceve il messaggio e registra il suo stato



55

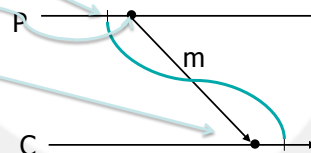
## Determinazione dello stato globale

- Stato globale



- Cosa abbiamo sbagliato?

- P registra il suo stato
- P non ha registrato l'invio di m
- C ha registrato la ricezione



56

## Algoritmo dei distributed snapshot

- Idea [Chandy&Lamport 85]
    - Utilizzare un messaggio speciale (marker) come elemento di sincronizzazione
  - Assunzioni ← Es.: TCP
    - Canali FIFO (implica relazioni di precedenza)
    - Nessuna perdita di messaggi
  - Algoritmo eseguito inizialmente da un iniziatore
    - Registra il proprio stato (locale)
    - Invia il marker su tutti i suoi canali di uscita
- atomica  
Nessun messaggio ricevuto o spedito durante queste operazioni

57

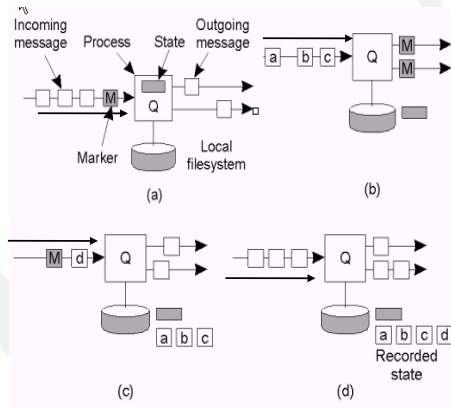
## Algoritmo dei distributed snapshot

- Comportamento del generico processo:
    - Alla prima ricezione del marker su un canale
      - Registra il suo stato locale
      - Indica come vuoto il canale da cui ha ricevuto il marker
      - Propaga il marker in uscita su tutti i suoi canali
      - Si mette "in ascolto" sui canali di ingresso
    - Alle successive ricezioni del marker su un canale
      - Stato del canale = {sequenza di messaggi ricevuti a partire dalla prima ricezione del marker}
  - Algoritmo termina quando:
    - Tutti i processi hanno ricevuto il marker su tutti i canali di ingresso
    - N.B.: raccolta dello stato globale richiede un ulteriore processo
  - Garantisce uno stato globale consistente
- atomica

58

## Distributed snapshot

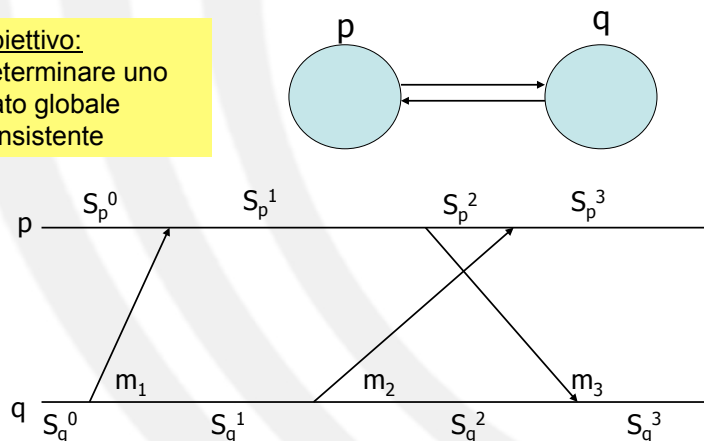
- Prima ricezione del marker
- Q inizia a registrare il proprio stato ed invia il marker sui suoi canali di uscita
- Q riceve un marker
- Q invia il suo stato  $S = \{a,b,c,d\}$



59

## Esempio di esecuzione (1)

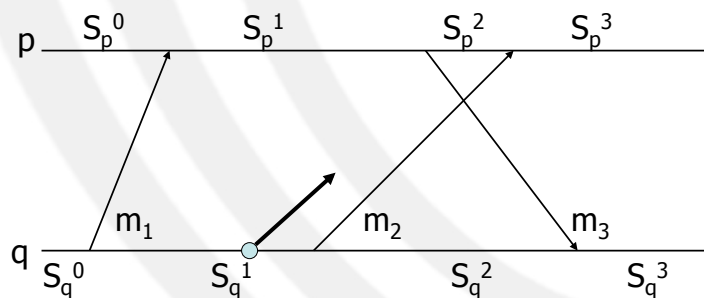
**Obiettivo:**  
Determinare uno  
stato globale  
consistente



60

## Esempio di esecuzione (2)

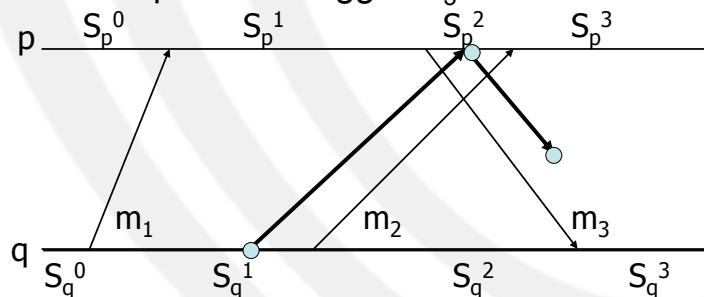
- q inizia l'algoritmo e registra lo stato  $S_q^1$
- q invia il marker a p



61

## Esempio di esecuzione (3)

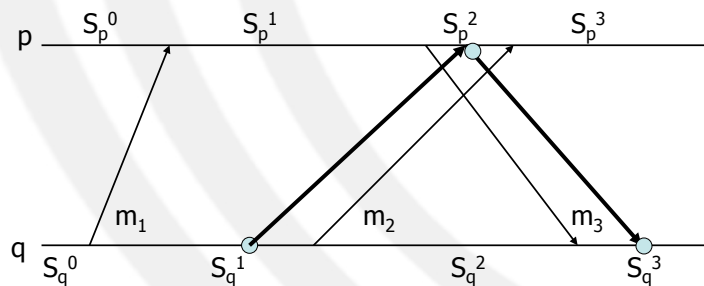
- p riceve il marker e registra il suo stato come  $S_p^2$
- Stato del canale =  $\{\}$
- p rispedisce il marker sul canale di uscita
- nel frattempo il messaggio  $m_3$  viene ricevuto da q



62

## Esempio di esecuzione (4)

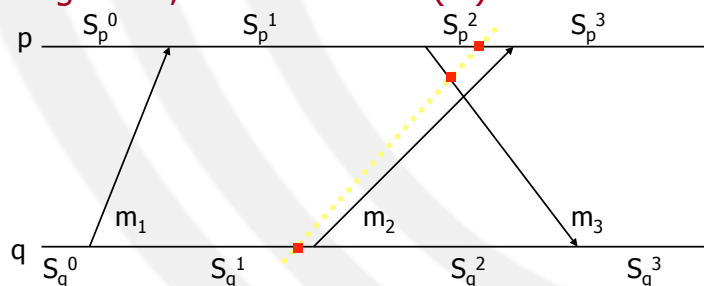
- q riceve il marker per la seconda volta
- q registra lo stato del canale =  $\{m_3\}$



63

## Esempio di esecuzione (5)

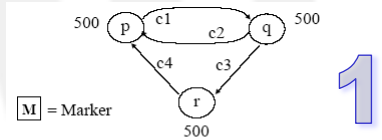
- STATO GLOBALE =  $\{(S_p^2, S_q^1), (0, m_3)\}$
- Consistente:
  - si può dimostrare che se la receive(m) è stata registrata, lo è anche send(m)



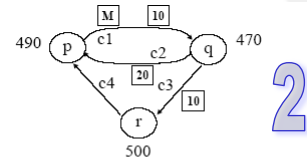
64



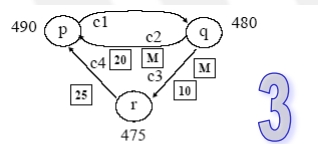
## Altro esempio (1)



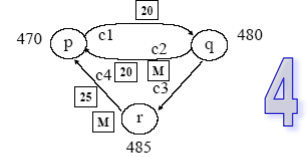
Node	Recorded state			
	c1	c2	c3	c4
p		{}		{}
q		{}		{}
r			{}	



Node	state	c1	c2	c3	c4
p	490		{}		{}
q					{}
r				{}	

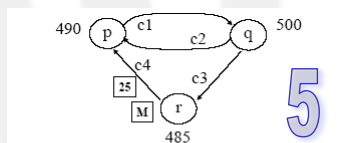


Node	state	c1	c2	c3	c4
p	490		{}		{}
q	480				{}
r				{}	

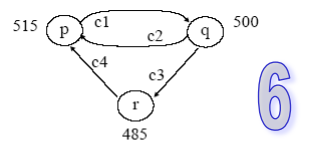


Node	state	c1	c2	c3	c4
p	490		{}		{}
q	480				{}
r	485			{}	

## Altro esempio (2)



Node	state	c1	c2	c3	c4
p	490		{20}		{}
q	480				{}
r	485			{}	



Node	state	c1	c2	c3	c4
p	490		{20}		{25}
q	480				{}
r	485			{}	

## Proprietà

- Il distributed snapshot garantisce che
  - se  $S_i$  e  $S_j$  sono gli stati globali al momento dell'inizio e della fine dell'algoritmo, e  $S^*$  è lo stato globale registrato dall'algoritmo, allora
    - $S^*$  è raggiungibile da  $S_i$
    - $S_j$  è raggiungibile da  $S^*$

67

## MUTUA ESCLUSIONE DISTRIBUITA

68

## Mutua esclusione

- Assunzioni
  - $n$  processi; ogni processo  $P_i$  in esecuzione su un diverso processore
  - Ogni processo ha una sezione critica da eseguire in mutua esclusione
- Requisiti
  - Se  $P_i$  sta eseguendo la sua sezione critica, nessun altro processo  $P_j$  sta eseguendo la sua sezione critica
- Due classi di approcci
  - Centralizzati
  - Distribuiti

69

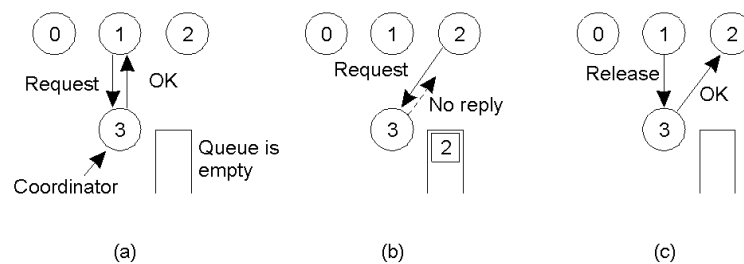
## Approccio centralizzato

- Uno dei processi nel sistema si occupa di coordinare gli ingressi nella sezione critica
  - Un processo che vuole entrare nella SC invia un messaggio *request* al coordinatore
  - Il coordinatore decide quale processo entra e invia un messaggio *reply*
  - Il processo che riceve la *reply* entra nella SC
  - Quando esce dalla SC, invia un messaggio *release* al coordinatore
- Tre messaggi per ogni ingresso nella SC

70

## Approccio centralizzato - esempio

- P1: request
- Coordinator: reply OK
- P2: request ma il coordinator non invia replay
- P1: release
- Coordinator: reply per P2



## Approccio centralizzato

- Vantaggi
  - Semplice (3 messaggi per richiesta)
  - Garantisce le tre condizioni (mutua esclusione, progresso, attesa limitata) senza deadlock/starvation
  - Utilizzabile per ogni tipo di risorsa
- Svantaggi
  - Criticità del coordinatore
    - Tolleranza ai guasti (se va in crash?)
    - Prestazioni (possibile collo di bottiglia)

## Approccio distribuito

- Basato su broadcast e sull'ordinamento totale degli eventi nel sistema [Ricart & Agrawala 81]
  - Data una coppia di eventi deve essere univocamente definito quale dei due si è verificato prima
  - Utilizzabili timestamp alla Lamport

73

## Approccio distribuito: algoritmo

- Quando un processo  $P_i$  vuole entrare nella SC
  - genera un timestamp  $TS$
  - invia il messaggio *request* ( $P_i, CS_k, TS$ ) a tutti i processi del sistema
    - invio dei messaggi affidabile (ACK per ogni messaggio)
- Quando un processo  $P_j$  riceve un messaggio di *request*,
  - può mandare un messaggio di *reply* (OK)
  - oppure mettere la richiesta in coda
- Quando un processo  $P_i$  riceve un messaggio di *reply* (OK) da tutti gli altri processi
  - può entrare nella SC
- Quando  $P_i$  esce dalla SC
  - invia un messaggio di *reply* (OK) a tutte le richieste accodate

74

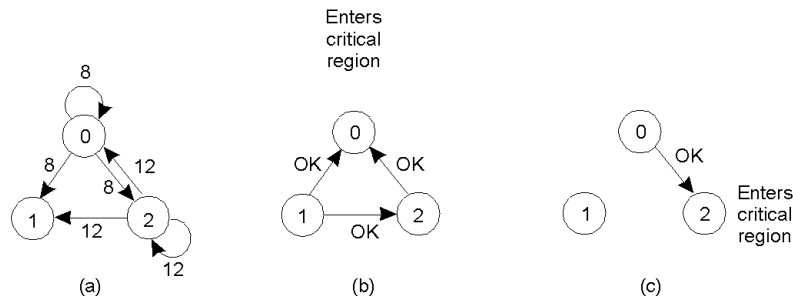
## Approccio distribuito: algoritmo

- La decisione di replicare o meno in seguito alla ricezione di *request* ( $P_i$ ,  $CS_k$ ,  $TS$ ) da parte di  $P_j$  dipende da tre situazioni:
  - Se  $P_j$  è nella SC
    - accoda il messaggio di reply (OK) a  $P_i$
  - Se  $P_j$  non è nella SC e non vuole entrare nella SC
    - invia immediatamente un messaggio di reply (OK) a  $P_i$ .
  - Se  $P_j$  vuole entrare nella SC
    - confronta il timestamp della propria richiesta  $TS_j$  con  $TS$ 
      - Se  $TS_j > TS$  invia immediatamente un messaggio di reply (OK) a  $P_i$  ( $P_i$  ha richiesto prima)
      - Altrimenti, accoda il messaggio di reply (OK) a  $P_i$

75

## Approccio distribuito: esempio

- $P_0$  e  $P_2$  richiedono l'accesso alla stessa risorsa
- $P_0$  entra prima ( $8 < 12$ )
- Una volta finito invia OK a  $P_2$



76

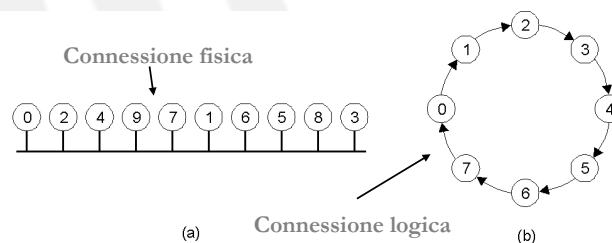
## Approccio distribuito

- Vantaggi
  - Mutua esclusione garantita senza deadlock/starvation
  - $2(n-1)$  messaggi per entrata nella SC
  - Nessun singolo punto di criticità
- Svantaggi
  - $n$  punti di criticità
    - Crash di un processo = divieto di accesso a SC!
  - $n$  colli di bottiglia
  - Necessità di primitive di comunicazione di gruppo
    - I processi devono conoscere l'identità di tutti gli altri
- Variante
  - Modificare l'accodamento con un messaggio di rifiuto
  - Funge da ACK e risolve il problema degli  $n$  punti di criticità

77

## Algoritmo token-ring

- Sorta di approccio distribuito
- Assume una specifica organizzazione logica dei processi (ad anello)
- Basato sul passaggio a rotazione di un token tra i vari processi



78

## Algoritmo token-ring

- Token originariamente a processo 0
- Passaggio tramite messaggi punto-punto da processo  $i$  a  $i+1$  (modulo  $N$ )
- Quando un processo riceve il token
  - Se vuole entrare nella SC
    - Entra nella SC
    - In uscita, passa il token al processo successivo
  - Se non vuole entrare
    - Passa subito il token

79

## Algoritmo token-ring

- Vantaggi
  - No deadlock/starvation
  - Criticità dei singoli nodi limitata
    - In caso di crash, è sufficiente “bypassare” (aggiornando la struttura della rete)
    - Ci accorgiamo del crash se non arriva ACK del token
- Svantaggi
  - Perdita del token
    - Come individuarla (SC lunga vs. token perso)?

80



## Confronto

<i>Algoritmo</i>	<i>Msg per entry/ exit</i>	<i>Ritardo prima di entrare (in tempi di msg)</i>	<i>Problemi</i>
Centralizzato	3	2	Crash del coordinatore
Distribuito	$2 (n - 1)$	$2 (n - 1)$	Crash di ogni processo
Token ring	Da 1 a $\infty$	Da 0 a $n - 1$	Perdita del token, crash di un processo

NOTA: ( $\infty \rightarrow$  nessun processo vuole entrare)

81

## TRANSAZIONI

82

## Transazioni atomiche

- Le tecniche precedenti sono da considerarsi “a basso livello”
  - Richiedono la conoscenza da parte del programmatore dei dettagli della mutua esclusione, deadlock, crash, etc.
- Necessario meccanismo più ad alto livello
  - transazioni atomiche (o transazioni)
- Derivate da modello “commerciale”
  - Proprietà tutto-o-niente (all-or-nothing)

83

## Transazioni atomiche: modello

- Esempi di tipiche primitive (offerte da S.O. o da un linguaggio)

Primitiva	Descrizione
<b>BEGIN_TRANSACTION</b>	<i>Inizio di una transazione</i>
<b>END_TRANSACTION</b>	<i>Termine di una transazione e tentativo di conferma (commit)</i>
<b>ABORT_TRANSACTION</b>	<i>Cancellazione della transazione e ripristino dei valori iniziali</i>
<b>READ</b>	<i>Generica lettura</i>
<b>WRITE</b>	<i>Generica scrittura</i>

- BEGIN/END essenziali
  - delimitano il contesto di una transazione
  - Fondamentali per garantire atomicità

84

## Transazioni atomiche: esempio

- Prenotazione aerea
  - Transazione (a) ok
  - Transazione (b) abortita quando la terza prenotazione non è fattibile

```
BEGIN_TRANSACTION
reserve WP -> JFK;
reserve JFK -> Nairobi;
reserve Nairobi -> Malindi;
END_TRANSACTION
(a)
```

```
BEGIN_TRANSACTION
reserve WP -> JFK;
reserve JFK -> Nairobi;
reserve Nairobi -> Malindi NO
=>ABORT_TRANSACTION
(b)
```

85

## Transazioni atomiche: proprietà

- Serializzabilità
  - Transazioni concorrenti non interferiscono con altre
    - In pratica, l'esecuzione appare al sistema come se le varie transazioni avvenissero sequenzialmente in un qualche ordine
- Atomicità
  - Transazioni avvengono in modo indivisibile (tutto o niente)
- Permanenza
  - Una volta confermata (commit), le modifiche sono permanenti
- Consistenza
  - Se qualche proprietà invariante è vera prima della transizione, lo deve essere anche dopo

86

## Serializzabilità - esempio

- Esecuzione “seriale”: valore finale dipende dalla sequenza (1, 2 o 3)

BEGIN_TRANSACTION $x = 0;$ $x = x + 1;$ END_TRANSACTION (a)	BEGIN_TRANSACTION $x = 0;$ $x = x + 2;$ END_TRANSACTION (b)	BEGIN_TRANSACTION $x = 0;$ $x = x + 3;$ END_TRANSACTION (c)
---	---	---

Schedule 1	$x = 0;$ $x = x + 1;$ $x = 0;$ $x = x + 2;$ $x = 0;$ $x = x + 3;$	Legale
Schedule 2	$x = 0;$ $x = 0;$ $x = x + 1;$ $x = x + 2;$ $x = 0;$ $x = x + 3;$	“Legale”
Schedule 3	$x = 0;$ $x = 0;$ $x = x + 1;$ $x = 0;$ $x = x + 2;$ $x = x + 3;$	Illegale

87

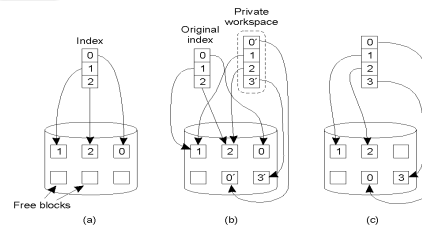
## Transazioni atomiche: implementazione

- Implementazione deve rispettare le 4 proprietà
- Concetto
  - Un processo che esegue una qualunque transazione non deve modificare gli oggetti originali
- Due metodi
  - Utilizzo di uno spazio di lavoro (workspace) privato
  - Write-ahead log

88

## Spazio di lavoro privato

- Uso di uno spazio privato (temporaneo) sul quale operare la transazione
- Commit = copia delle modifiche su spazio comune
- Ottimizzazioni per abbattere il costo
  - No copia per accessi non modificanti
  - Copia di strutture dati (i-node), non di dati
- Esempio:
  - 1,2 = read
  - 0 = write
  - 3 = append



89

## Write-ahead Log

- File di log contenente una sorta di “dichiarazione di intenti”
- Modifiche effettive, ma il log contiene, per ogni transazione, l'effetto della modifica
- Esempio

	Vecchio valore	Nuovo valore
<pre> x = 0; y = 0; BEGIN_TRANSACTION;   x = x + 1;   y = y + 2;   x = y * y; END_TRANSACTION;</pre>	<pre> Log [x = 0/1]</pre>	<pre> Log [x = 0/1] [y = 0/2] [x = 1/4]</pre>
(a)	(b)	(c) (d)

90

## Write-ahead Log

- Al termine delle operazioni
  - Se tutto OK, le modifiche sono già state fatte
  - Se abort, si usa il log (al contrario) per ripristinare (rollback) lo stato precedente alla transazione

91

## Problematiche

- Commit
  - L'operazione di commit deve essere atomica
    - Necessità di opportuno protocollo in ambienti distribuiti
- Controllo della concorrenza
  - Necessario per impedire che transazioni simultanee creino situazioni indesiderate
  - Gestione tramite un transaction manager
    - Two-phase locking, pessimistic timestamp ordering, optimistic timestamp ordering

92

## DEADLOCK IN SISTEMI DISTRIBUITI

93

### Definizione

- Processi utilizzano risorse
- Sequenza di utilizzo
  - Richiesta
    - Se non può essere immediatamente soddisfatta, il processo deve attendere
  - Utilizzo
  - Rilascio
- DEADLOCK  
Un insieme di processi è in deadlock quando ogni processo è in attesa di un evento che può essere causato da un processo dello stesso insieme

94

## Condizioni necessarie

- **Mutua esclusione**
  - Almeno una risorsa deve essere non condivisibile
- **Hold and Wait**
  - Deve esistere un processo che detiene una risorsa e che attende di acquisirne un'altra, detenuta da un altro
- **No preemption**
  - Le risorse non possono essere rilasciate se non "volontariamente" dal processo che le usa
- **Attesa circolare**
  - Deve esistere un insieme di processi che attendono ciclicamente il liberarsi di una risorsa
- **Devono essere vere contemporaneamente**
  - Se una non si verifica, non si ha deadlock

95

## Gestione dei deadlock

- **Stesse opzioni dei sistemi centralizzati**
  - **Algoritmo dello struzzo**
  - **Prevenzione statica**
    - Evitare che si possa verificare una delle quattro condizioni
  - **Prevenzione dinamica basata su allocazione delle risorse**
    - Mai usata: vincolo sulla conoscenza delle richieste di risorse
  - **Rivelazione e ripristino**
    - Permettere che si verifichino
    - Prevedere metodi per riportare il sistema al funzionamento normale (recovery)

96



## Gestione dei deadlock

- Noi vedremo
  - Rivelazione
    - Centralizzata
    - Distribuita [Chandy&Misra&Haas 83]
  - Prevenzione statica
    - Evitare attesa circolare
    - Distribuita
- Rilevazione e ripristino è accettabile grazie all'uso delle transazioni che permettono di fare rollback

97

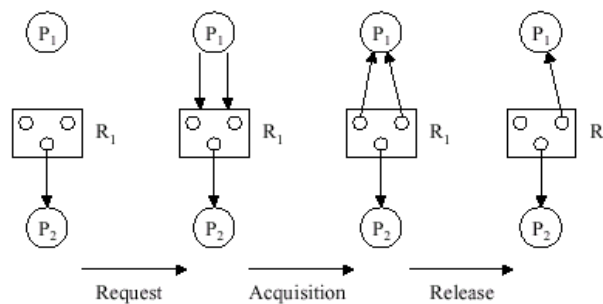
## Modello RAG

- Grafo di allocazione delle risorse (RAG)  $G(V,E)$ 
  - $V$  = nodi
  - $E$  = archi
- Nodi
  - Cerchi = processi (CPU, I/O, memoria)
  - Rettangoli = risorse
    - Nei rettangoli vi sono tanti “•” quante sono le istanze della corrispondente risorsa
- Archi
  - Da processi a risorse: processo richiede risorsa
  - Da risorse a processi: processo detiene risorsa

98

## Esempio di RAG

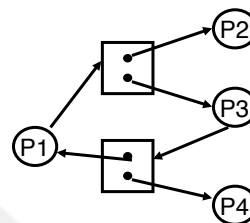
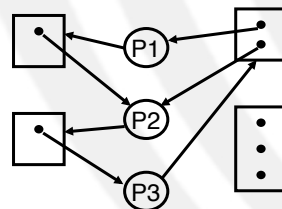
- $V = \{P_1, P_2, R_1\}$
- $E_{iniziale} = \{(R_1, P_2)\}$   $E_{finale} = \{(R_1, P_1), (R_1, P_2)\}$



99

## RAG e deadlock

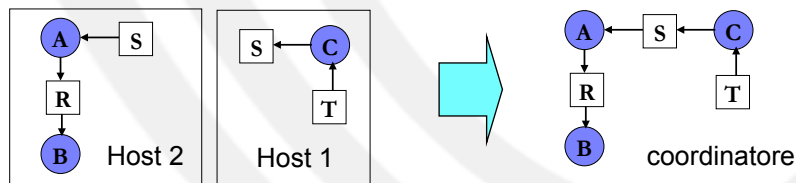
- Se il RAG non contiene cicli, non ci sono deadlock
- Se contiene cicli
  - Se si ha una sola istanza per risorsa → deadlock
  - Se ci sono più istanze, dipende dallo schema di allocazione



100

## Rivelazione: approccio centralizzato

- Emulazione del caso non distribuito
- Un coordinatore mantiene un RAG globale
  - Unione dei singoli RAG dei singoli host
  - Se trova un ciclo uccide un processo
- Esempio



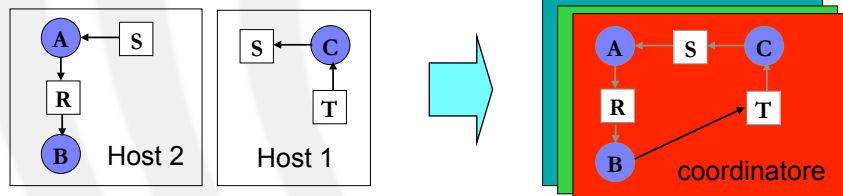
101

## Rivelazione: approccio centralizzato

- Come e quando costruire il RAG globale?  
(quando trasmettere al coordinatore?)
  - Dopo ogni aggiunta/rimozione di un arco nei grafi locali
  - Periodicamente
  - Su richiesta del coordinatore

102

## Falsi deadlock – Esempio



- B rilascia R e richiede T (operazione legale)
- Qual è l'ordine dei messaggi?
  - Msg1: Host 2 annuncia al coordinatore che B lascia R
  - Msg2: Host 1 annuncia al coordinatore che B chiede T
- Se i messaggi arrivano in ordine opposto → falso deadlock

103

## Falsi deadlock – Soluzione

- Aggiornamento del grafo globale deve essere indipendente dall'ordine di ricezione dei messaggi
  - Associazione di timestamp (alla Lamport) alle modifiche dei grafi locali
  - Su ricezione di una modifica con timestamp T, il coordinatore sollecita l'invio di eventuali altre modifiche in corso con timestamp minore di T

104

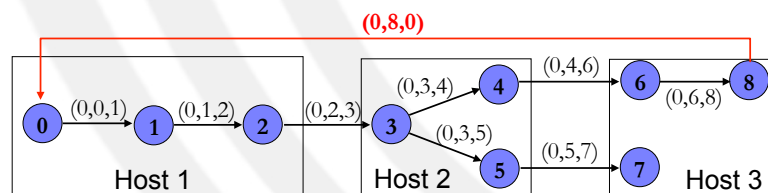
## Rivelazione: approccio distribuito

- [Chandy&Misra&Haas 83]
  - Invocato quando un processo si mette in attesa per una qualche risorsa
    - Invia messaggi di probe ai processi che detengono la risorsa
      - Messaggio = (id proc bloccato, id proc mittente, id proc destinatario)
    - probe propagato a catena ai processi (eventualmente remoti) a loro volta in attesa
      - Se il ricevente è in attesa rispedisce il probe aggiornando mittente e destinatario
    - Se il messaggio raggiunge il mittente → deadlock
  - Recovery
    - Uccisione dell'iniziatore del probe → possibili molte uccisioni se + processi iniziano l'algoritmo contemporaneamente
    - Uccisione del processo con id più altro

105

## Rivelazione: approccio distribuito

- Esempio:
- Il sistema è in deadlock perché il processo 0 (che inizia l'algoritmo) riceve un messaggio (0,8,0) in cui l'id del processo bloccato è proprio 0.



106

## Prevenzione statica

- Prevenzione della condizione di attesa circolare
  - Definizione di un ordine globale tra le risorse del sistema
  - Es: assegnazione di un unico numero ad ogni risorsa
    - Un processo può richiedere una risorsa con un numero  $n$  solo se non possiede risorse con numeri più alti di  $n$
    - In tal modo è impossibile che si verifichi attesa circolare
  - Implementazione semplice e a basso costo

107

## Prevenzione statica

- In presenza di:
  - Transazioni atomiche
  - Ordinamento globale del tempo (Lamport)
- possibili altri schemi:
  - Wait-die (aspetta-muori)
  - Wound-wait (ferisci-aspetta)

108

## Prevenzione statica basata su timestamp

- Timestamp usato come valore di priorità di ogni processo
- Uso dei timestamp per regolare l'attesa di un processo
  - Schema non-preemptive (wait-die)
  - Schema preemptive (wound-wait)
- L'uso dei timestamp previene starvation

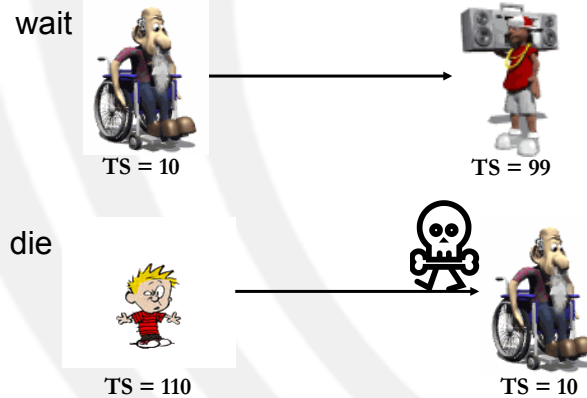
109

## Schema Wait-Die

- Se un processo  $P_i$  necessita di una risorsa detenuta da un processo  $P_j$ 
  - se  $TS(P_i) < TS(P_j) \Rightarrow P_i$  si mette in attesa (wait)
  - se  $TS(P_i) > TS(P_j) \Rightarrow$  rollback di  $P_i$  (die)

110

## Wait-die – esempio



→ Sempre nella direzione di timestamp crescenti

111

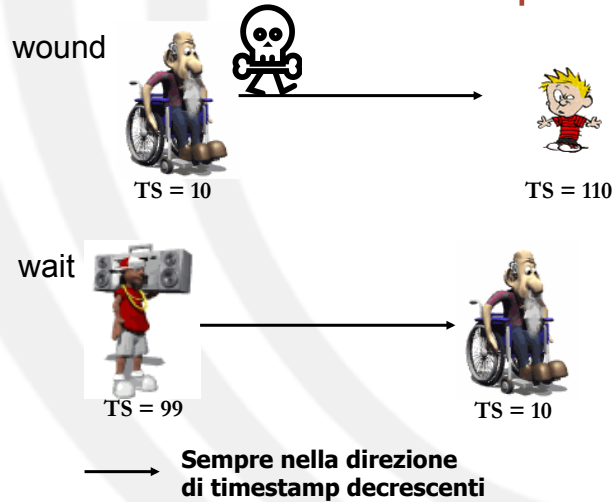
## Schema Wound-Wait

- Se un processo  $P_i$  necessita di una risorsa detenuta da un processo  $P_j$ 
  - se  $TS(P_i) < TS(P_j) \Rightarrow P_j$  viene ferito e ucciso (wound)
  - se  $TS(P_i) > TS(P_j) \Rightarrow$  si mette in attesa (wait)

112



## Wound-wait – esempio



113

## Prevenzione statica basata su timestamp

- Rollback possibile grazie a transazioni ➡ possibile sottrarre risorse senza danni!
- Wait-die tende ad ammazzare tanti processi giovani
  - Vecchio rispettoso aspetta
  - Giovane presuntuoso viene ripetutamente ucciso
- Wound-wait ammazza una sola volta
  - Vecchio saggio prelaiona il giovane
  - Il giovane riparte e attende pazientemente

114

# ALGORITMI DI ELEZIONE

115

## Algoritmi di elezione

- Alcuni algoritmi distribuiti richiedono la presenza di un processo speciale (es.: coordinatore, iniziatore)
- Necessità di algoritmi specifici per stabilire chi eleggere (o rieleggere)
- Assunzioni
  - Un valore unico di priorità  $i$  associato ad ogni processo  $P_i$  del sistema (solitamente PID)
  - Corrispondenza uno a uno tra host e processo (1 processo/1 host)
- Coordinatore = processo con valore di priorità più alto

116

## Algoritmo del bullo

- [Garcia-Molina 82]
- Applicabile a sistemi in cui ogni processo può mandare un messaggio ad ogni altro processo
- Schema
  - Iniziato da un processo  $P_i$  nel caso di richiesta non risposta dal coordinatore entro un tempo  $T$
  - $P_i$  assume che il coordinatore non è attivo e cerca di eleggere se stesso come coordinatore

117

## Algoritmo del bullo

- Processo di elezione
  - $P_i$  manda un messaggio (ELECTION) ad ogni processo con un valore di priorità più alto, ed attende una replica entro un tempo  $T$
  - Se non riceve risposta entro  $T$ , assume che tutti i processi destinazione non siano attivi;  $P_i$  elegge se stesso come coordinatore
  - Se riceve risposta (OK), i processi che hanno ricevuto il messaggio lanciano a loro volta un'elezione  
 $P_i$  resta in attesa di un messaggio da parte del nuovo coordinatore (COORDINATOR)
    - Attesa legata a timeout ( $T'$ )
    - Se non riceve messaggi entro  $T'$ ,  $P_i$  rilancia un'elezione

118

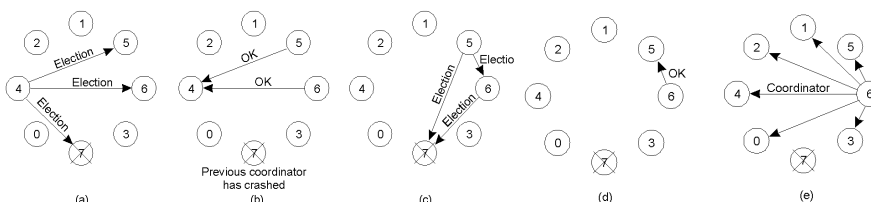
## Algoritmo del bullo

- L'algoritmo "favorisce" processi che ripartono dopo un crash
  - Il processo esegue l'algoritmo
  - Se non ci sono processi con valori più alti, il processo ripartito (bullo) obbliga tutti i processi con valori più bassi a farlo diventare il coordinatore
  - Generalmente il processo che riparte ha un PID maggiore rispetto a quello dei processi già esistenti

119

## Algoritmo del bullo: esempio

1. P7 crash
2. P4 lancia un'elezione
3. P5 e P6 rispondono a P4 (OK)
4. P5 e P6 lanciano un'elezione
5. P6 risponde a P5 (OK)
6. P6 comunica a tutti che e' il nuovo coordinatore



120

## Algoritmo dell'anello

- Applicabile a sistemi organizzati ad anello (logico)
- Assunzioni
  - Link unidirezionali
  - Processi mandano messaggi al processo adiacente
  - Ogni processo mantiene una lista di tutti i valori di priorità di tutti i processi nel sistema
- Se il processo  $P_i$  osserva un problema nel coordinatore, lancia un'elezione a cascata
  - Messaggio ELECTION al primo successore attivo
  - Aggiunge il valore  $i$  (la propria priorità) alla lista

121

## Algoritmo dell'anello

- Prima o poi,  $P_i$  riceverà un messaggio contenente il proprio numero
- A questo punto  $P_i$  è in grado di decidere chi è il coordinatore
  - Fa circolare lo stesso messaggio, stavolta di tipo COORDINATOR
  - Tutti i processi, analizzando la lista, determinano il nuovo coordinatore (processo con priorità massima)
  - Quando i vari messaggi COORDINATOR sono circolati una volta, vengono eliminati
- Messaggi ridondanti!

122

## Algoritmo dell'anello: esempio

- P7 crash
- P5 e P2 lanciano un'elezione simultaneamente
- Quando P5 e P2 ricevono un messaggio contenente 2 e 5 rispettivamente, lanciano un messaggio COORDINATOR
- P6 diventa nuovo coordinatore

