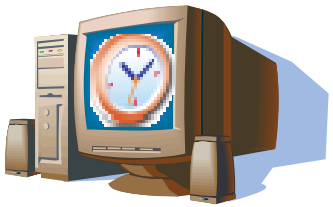
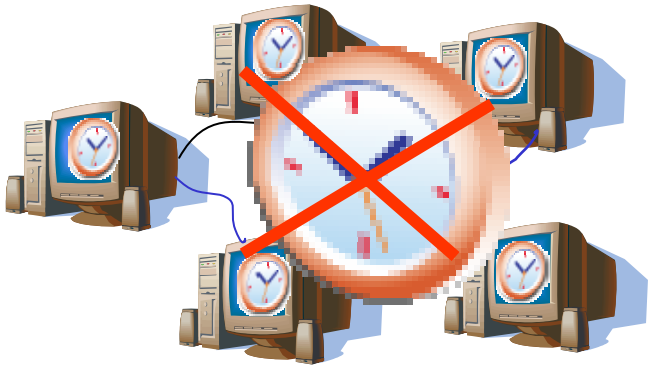
Sincronizzazione nei Sistemi Distribuiti

Sincronizzazione dei Clock

- In un sistema centralizzato la misurazione del tempo non presenta ambiguità.
  (*ogni computer ha il proprio clock*)



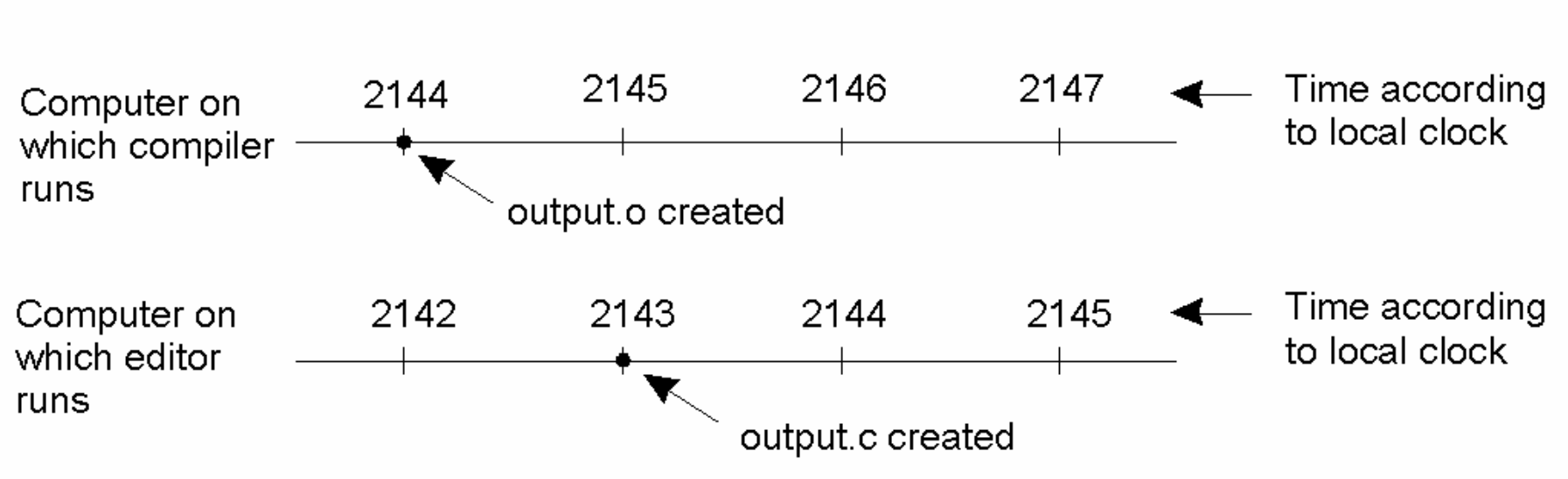
- In un sistema distribuito definire un tempo globale non è semplice.
  (*è impossibile garantire che i clocks avanzino tutti alla stessa esatta frequenza*)



- Soluzioni:
  - Clock synchronization
  - Logical clocks

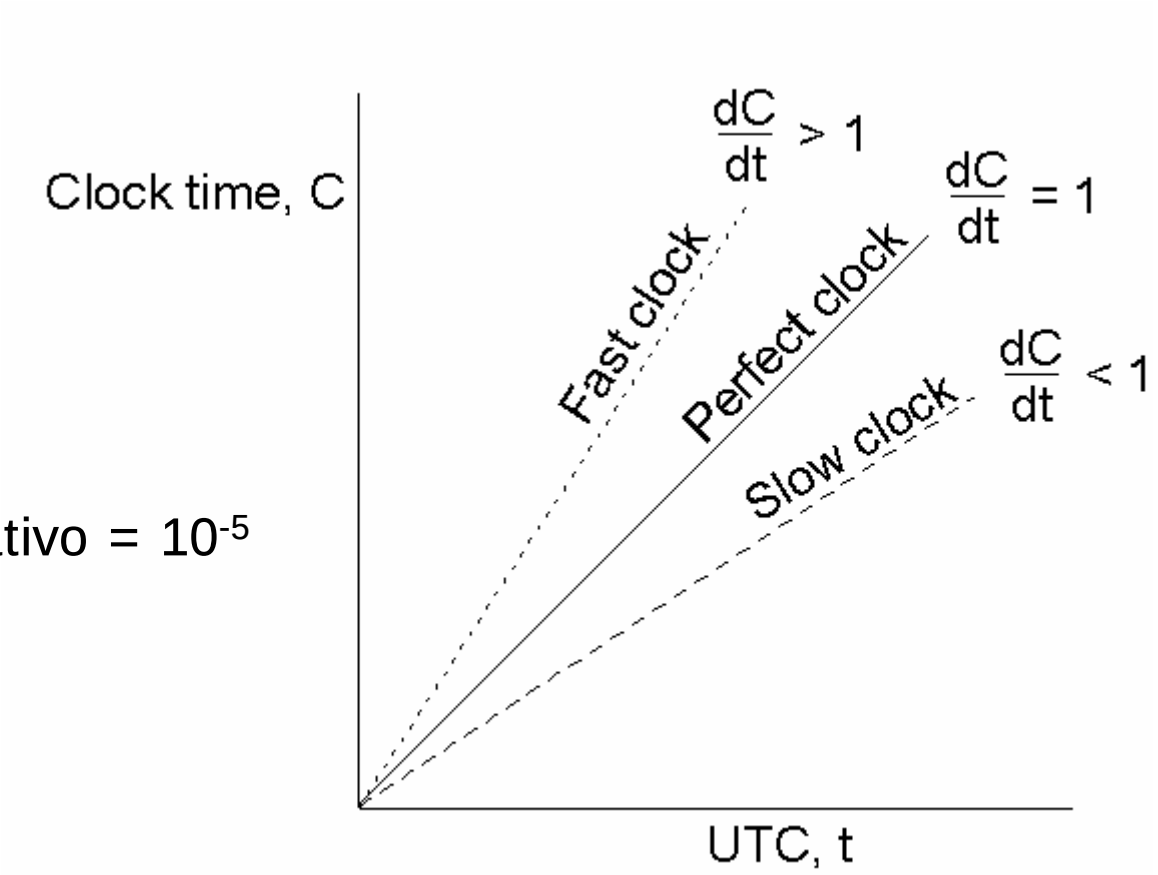
Sincronizzazione dei Clock

Esempio: il programma *make*



- Quando ogni macchina ha il proprio clock, ad un evento che avviene dopo un altro evento gli può essere assegnato un tempo anteriore.

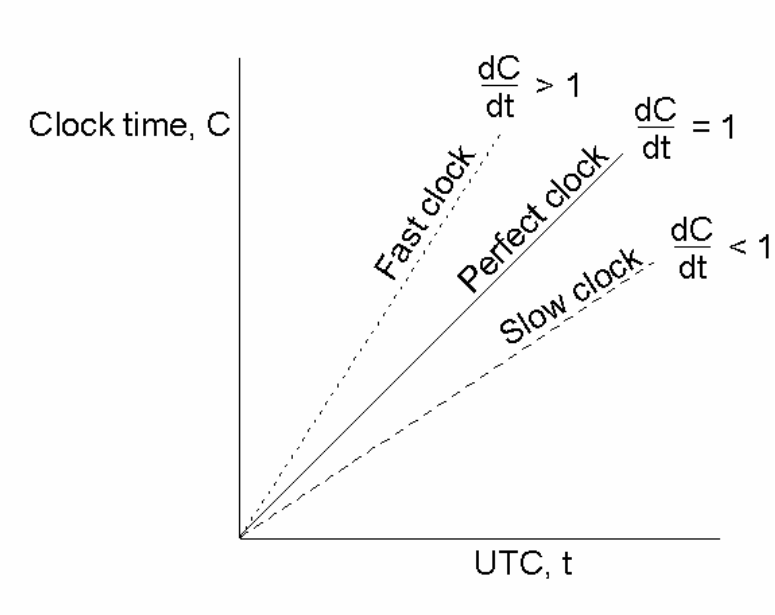
Algoritmi di Sincronizzazione dei Clock



Errore relativo = 10^{-5}

Clock time e UTC (Universal Coordinated Time) con i clocks tick a differenti velocità.

Algoritmi di Sincronizzazione dei Clock



Se esiste una costante ρ tale che

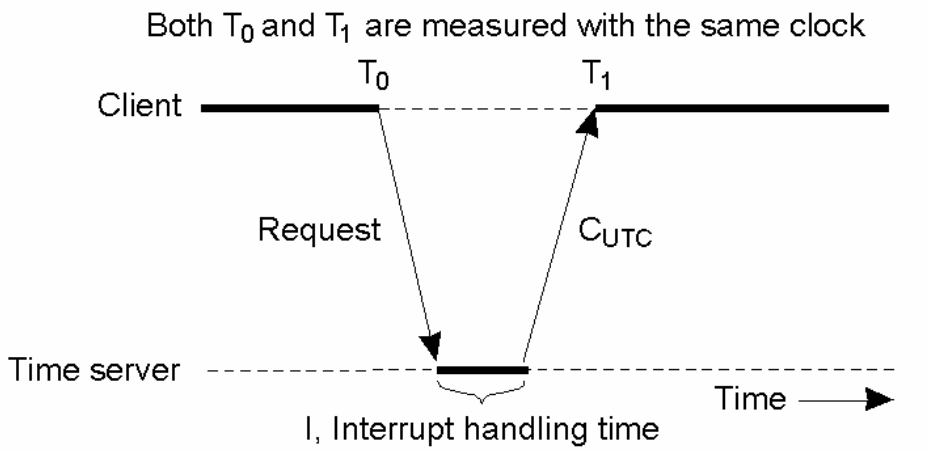
$1 - \rho <= dC/dt <= 1 + \rho$ (maximum drift rate)

dopo Δt la differenza tra due clock può essere al massimo:

$2\rho\,\Delta t$

Algoritmo di Cristian

Ipotesi: I computer ricevono periodicamente il tempo corrente da un **time server**.



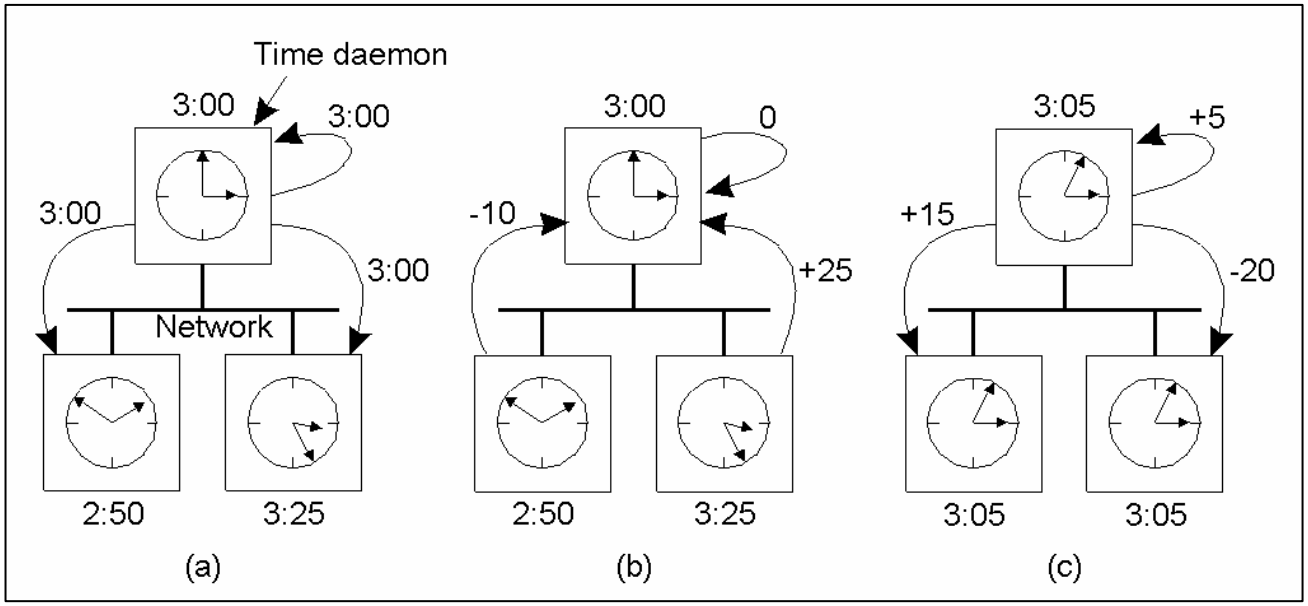
Due problemi:

- Il tempo non deve mai scorrere all'indietro (per il clock del server più lento)
- La risposta del server del C_{UTC} richiede un tempo pari a : $(T_1-T_0-I)/2$.

Algoritmo di Berkeley

- Il server ha un ruolo attivo, ma non ha il valore esatto del tempo da fornire alle macchine.

a) Il server chiede a tutte le macchine il valore del loro clock.
b) Ogni macchina risponde al server.
c) Il server invia a tutte le macchine il nuovo valore medio del clock.



Clock Logici

- I Clock Logici sono usati quando è necessario avere un **valore del tempo consistente per tutti i nodi** del sistema distribuito, ma questo non nececessariamente deve essere il valore del tempo reale assoluto.

- Proposta di Lamport:
  a) *Se due processi non interagiscono non è necessario sincronizzare i loro clock.*
  b) *Quello che è importante per due o più processi interagenti è rispettare l'ordine corretto in cui gli eventi avvengono.*

Timestamp di Lamport

- Per sincronizzare i clock logici è stat definita la relazione:

  happens-before ($\rightarrow$)

- $a \rightarrow b$ significa “a avviene prima di b”

- Se a e b sono due eventi nello stesso processo e a avviene prima di b, allora: $a \rightarrow b$ è vera

- In due processi, se a è l’evento di invio di un messaggio m e b è l’evento di ricezione di un messaggio m, allora: $a \rightarrow b$ è vera

- Se $a \rightarrow b$ e $b \rightarrow c$, allora: $a \rightarrow c$

Timestamp di Lamport

- Considerando due eventi x e y in due processi non-interagenti, allora $x \rightarrow y$ non è vero, ma neanche $y \rightarrow x$ è vero.

- x e y sono detti **concorrenti**.

- Per ogni evento non concorrente a è necessaria una misura globale del tempo da assegnare ad a : $C(a)$ valido in tutti **i processi / processori**

- Se $a \rightarrow b$ allora $C(a) < C(b)$.

Timestamp di Lamport

Total ordering può essere definito se :

- Ogni messaggio contiene il tempo del suo invio sul mittente (basato sul clock del nodo mittente)
- Quando un messaggio arriva il clock del ricevente deve essere maggiore di almeno un tick del tempo del mittente (segnato sul messaggio).
- Tra due eventi il clock deve avanzare almeno di un tick.

Requisito addizionale:

- Non si possono avere due eventi che accadono nello stesso esatto istante di tempo.

Algoritmo di Lamport

/ Cp è il valore del clock logico del processo*

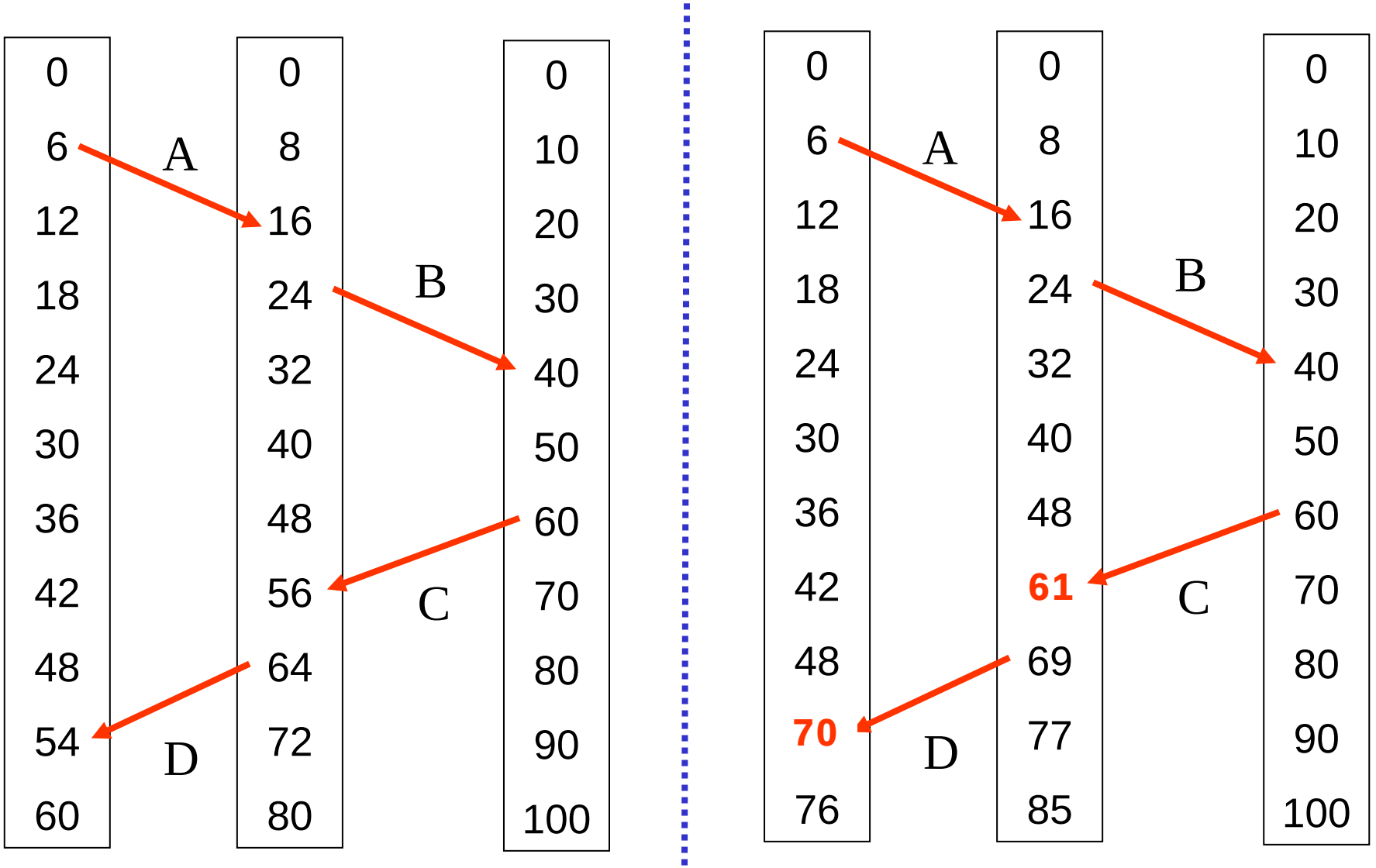
/ Cr è il valore del clock ricevuto dal processo remoto*

```
Var Cp: integer

Cp = 0;

if (a è un evento interno)
        Cp = Cp +1;
if (a è l'invio di un messaggio)
     {  Cp = Cp +1;
        send(messg, Cp); }
if (a è la ricezione di un messaggio)
     {  receive(messg, Cr);
        Cp=max(Cp, Cr)+1; }
```

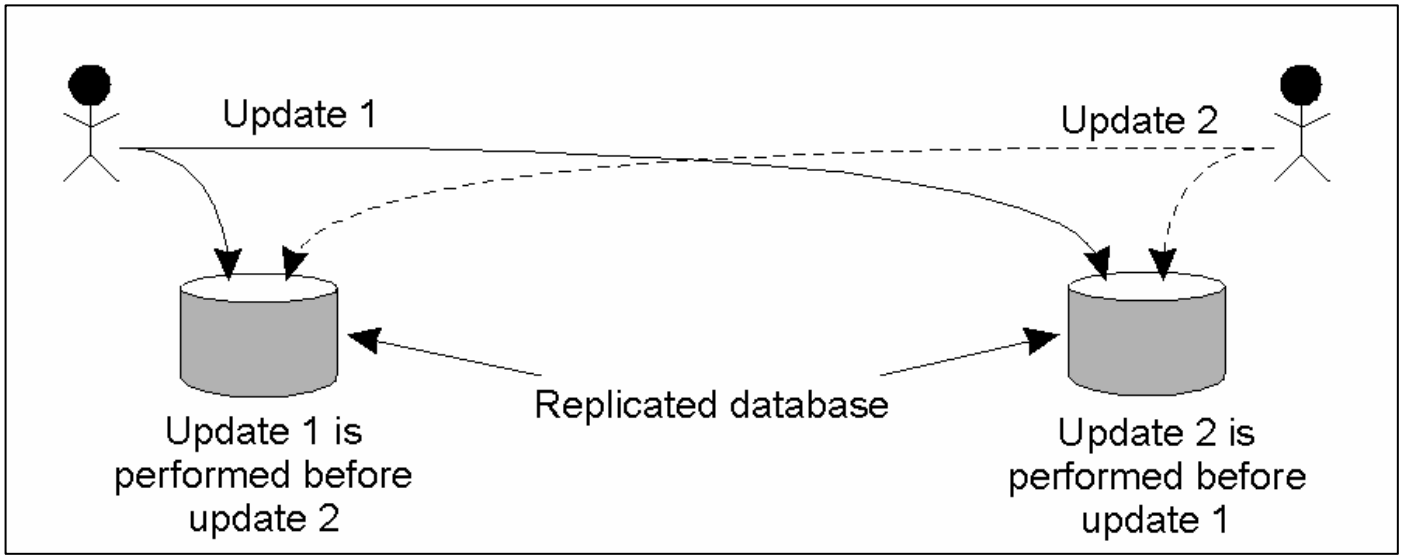
Timestamp di Lamport



Tre processi, ognuno con il proprio clock. I clock hanno diverse velocità.
L'algoritmo di Lamport corregge o clock.

Esempio: Totally-Ordered Multicasting

Database Replicato in due siti



Se si fanno due incrementi contemporanei, l'aggiornamento del database replicato può portare ad uno stato inconsistente.

Un meccanismo di **totally-ordered multicast** (tutti i messaggi consegnati a tutti nello stesso ordine) è necessario e può essere implementato con i timestamp di Lamport.

Totally-Ordered Multicasting

- Un gruppo di processi comunica tramite multicast tra loro:

  1. Ogni messaggio è inviato a tutti i processi con una multicast e con un timestamp del ***logical time*** del mittente e messo in coda nell'ordine del timestamp.

  2. I messaggi sono consegnati nell'ordine in cui vengono inviati

  3. Ogni messaggio richiede l'invio di un acknowledge

  4. Non è possibile che due messaggi abbiamo lo stesso timestamp

  5. Ogni processo ha la stessa copia della coda.

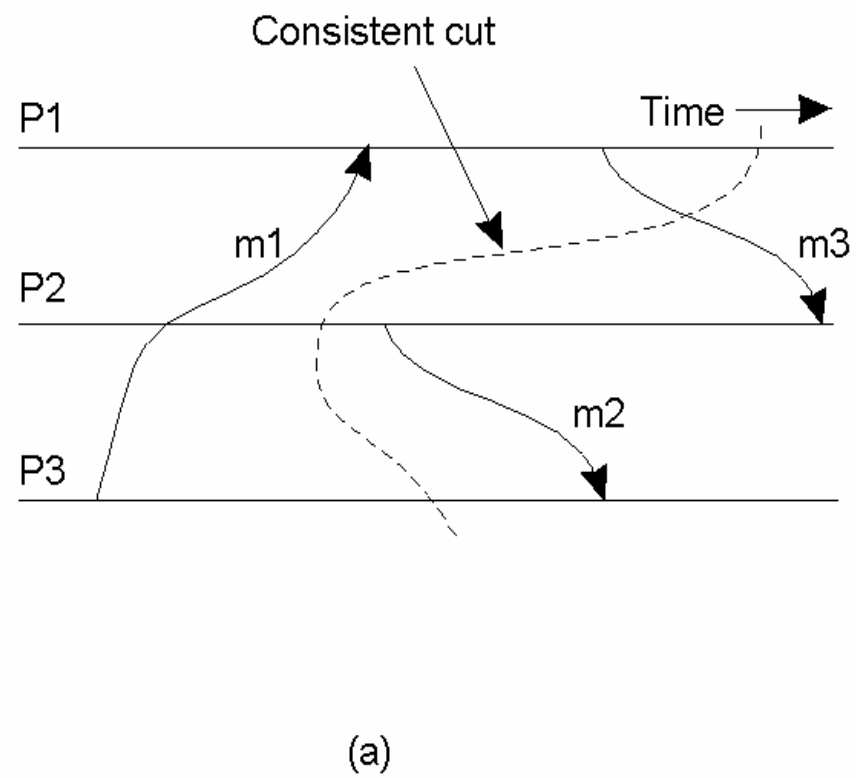
Global State (1)

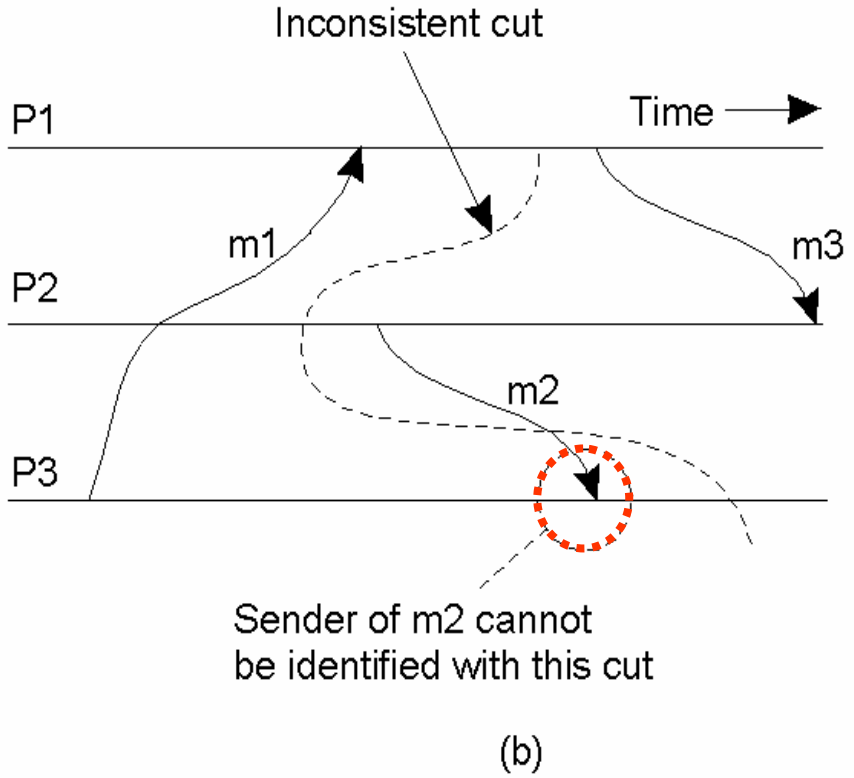
a) Lo **stato globale** di un sistema distribuito è dato dalla ***collezione degli stati locali*** di ogni processo più i ***messaggi in transito***.

b) La conoscenza dello stato globale è utile in molti casi.

c) Uno **snapshot distribuito** è uno stato in cui un sistema distribuito si può trovare (uno stato globale consistente).

Global State (2)





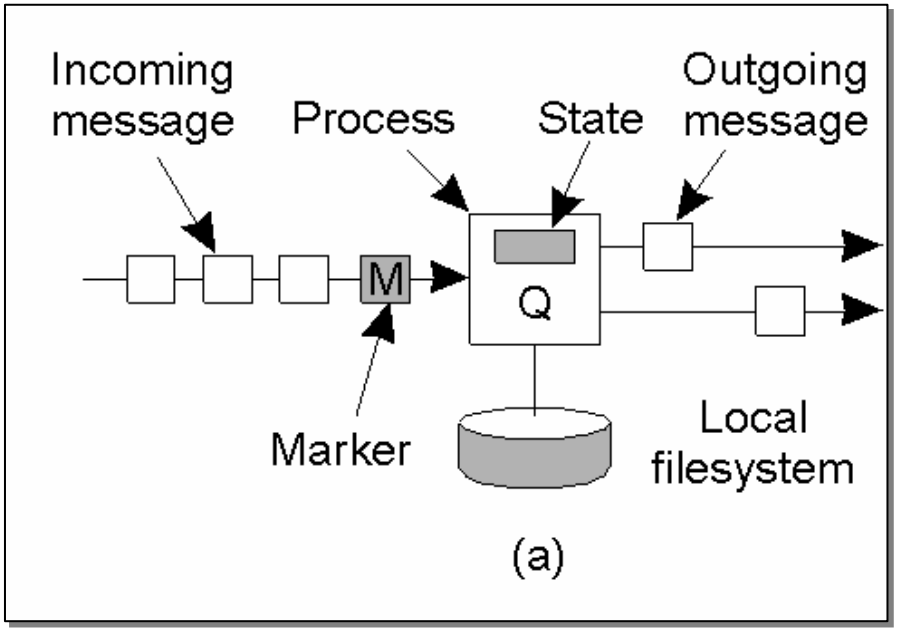
(a) Un “taglio” consistente

(b) Un “taglio” inconsistente

Global State (3)

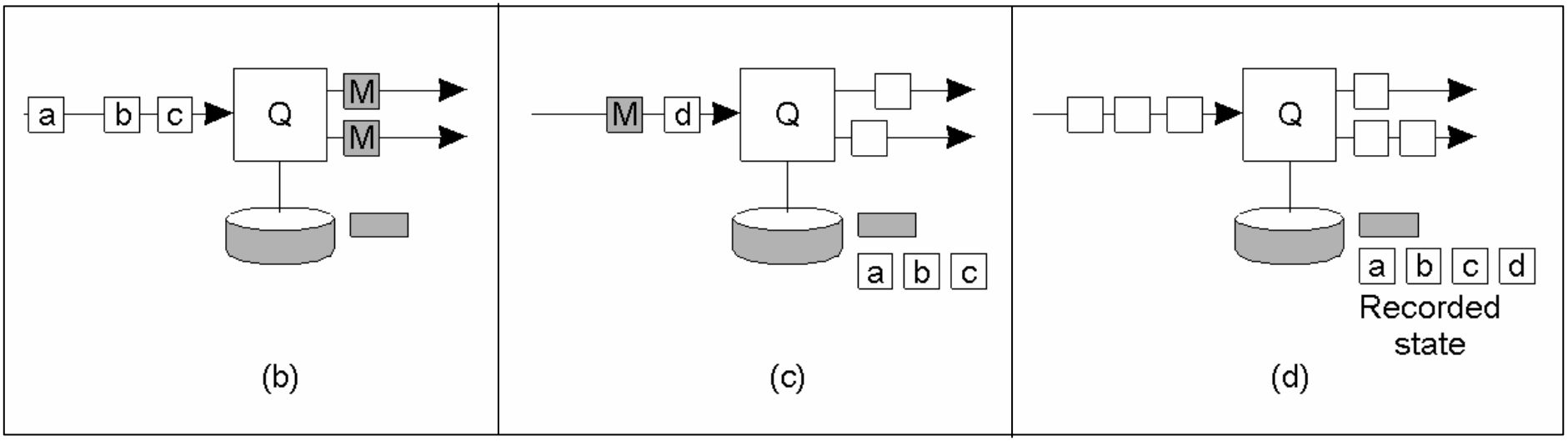
- Usando i distributed snapshots è possibile memorizzare uno stato globale.

a. Un processo *P* inizia l'algoritmo memorizzando il proprio stato e invia un marker nei canali di uscita indicando al destinatario che deve participare per memorizzare lo stato globale.



Organizzazione di un processo Q e dei canali per uno snapshot distribuito

Global State (4)



b. Quando un processo *Q* riceve un marker per la prima volta memorizza il suo stato locale e invia il marker nei suoi canali di uscita.

c. *Q* memorizza tutti i messaggi in arrivo

d. *Q* riceve un marker per i suoi canali di input e finisce memorizzando lo stato dei canali in ingresso.

Global State (5)

- Quando un processo ha ricevuto ed elaborato tutti i marker nei suoi canali di ingresso completa il suo compito per l'algoritmo e invia lo stato che ha memorizzato.

- Un processo qualsiasi può iniziare l'algoritmo e il marker sarà etichettato con l'identificatore del processo iniziatore.

Terminazione Distribuita (1)

- Identificare e gestire la terminazione di un algoritmo distribuito non è banale (a volte è complesso).
- Uno snapshot distribuito può non mostrare uno stato di terminazione a causa dei messaggi che possono essere in transito.
- Per la rilevazione della terminazione tramite uno snapshot distribuito è necessario che tutti I canali siano vuoti.

Terminazione Distribuita (2)

- Quando un processo Q completa la sua parte dello snapshot, può inviare un messaggio *DONE* ai suoi predessori se due condizioni sono sono verificate:
  1. tutti i successori di Q hanno ritornato un messaggio DONE
  2. Q non ha ricevuto messaggi tra l'istante di tempo di memorizzazione dello stato e la ricezione del marker su tutti i suoi canali di ingresso.
- In tutti gli altri casi, Q invia un messaggio *CONTINUE* al suo predecessore.
- Solo quando tutti i messaggi *DONE* sono stati ricevuti dal processo iniziatore l'elaborazione è terminata.

Algoritmi di Elezione

Algoritmi per **eleggere un coordinatore** (con un ruolo speciale) tra i processi che compongono una applicazione distribuita.

- Ogni processo è identificato da un identificatore numerico (ID).
- Ogni processo conosce l'identificatore di tutti gli altri processi.
- Ma non sa quali sono attivi e quali non lo sono.
- Un algoritmo di elezione termina quando tutti i processi condordano su un coordinatore.

Algoritmo Bully (1)

Un processo P gestisce una elezione come di seguito:

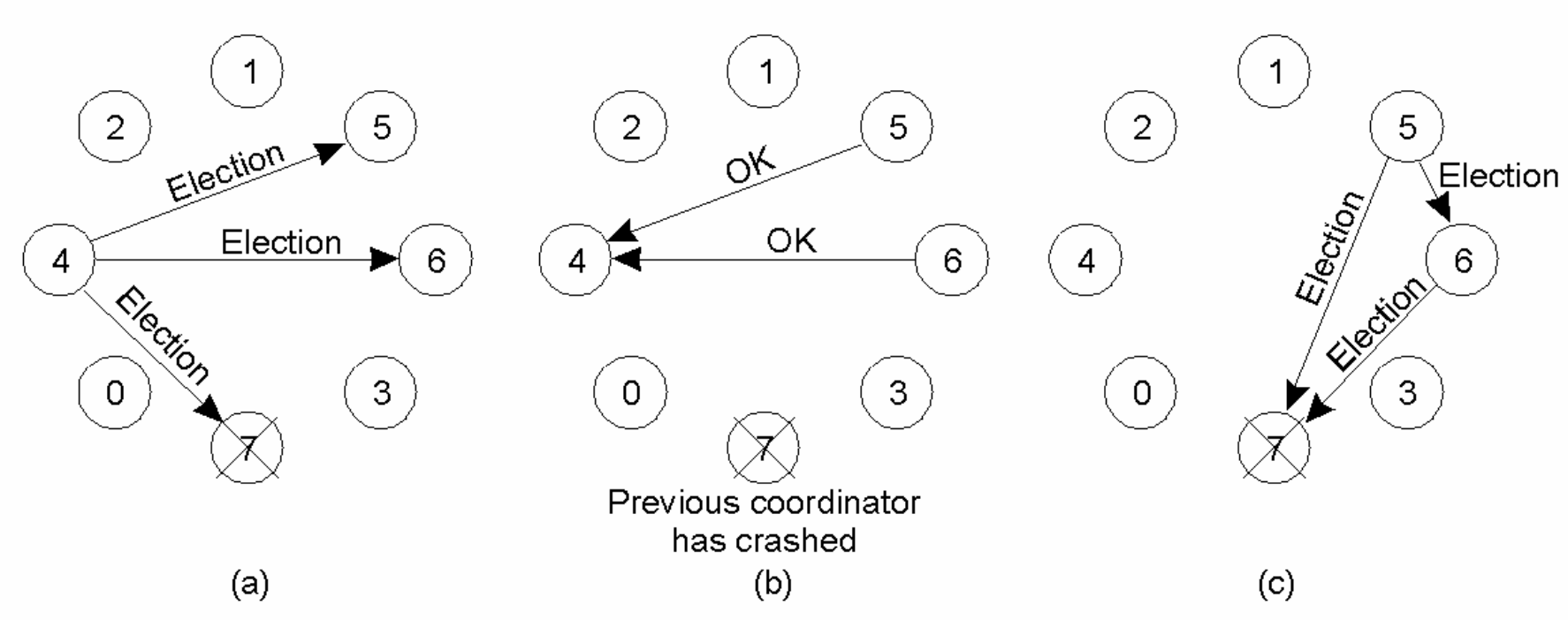
1. P invia un messaggio *ELECTION* a tutti i processi con ID maggiore del proprio.

2. Se nessuno risponde, P diventa in nuovo coordinatore.

3. Se un processo con ID maggiore risponde, questo continua l'algoritmo di elezione.

4. Il nuovo coordinatore informa tutti i processi.

Algoritmo Bully (2)



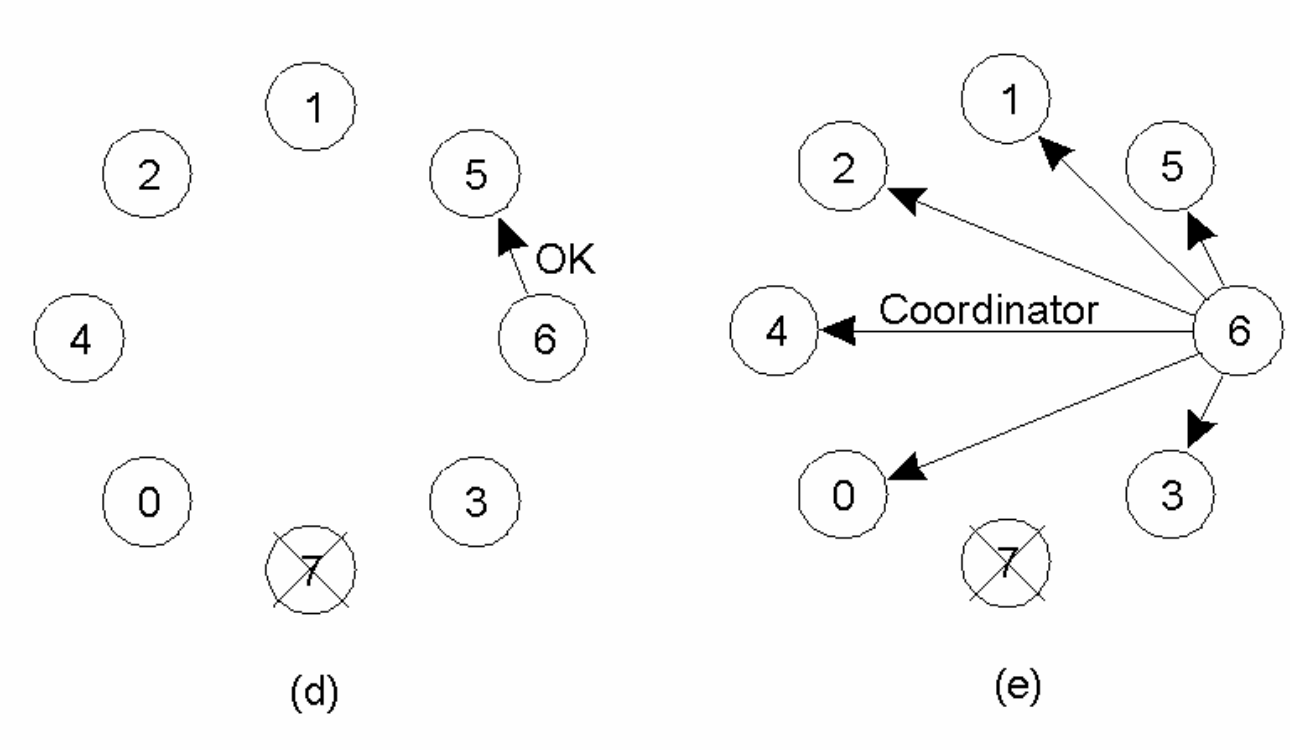
L'algoritmo di elezione Bully

a) Il processo 4 inzia l'algoritmo di elezione

b) I processi 5 e 6 rispondono, informando 4 di fermarsi

c) Adesso 5 e 6 prendono in carico la continuazione dell'algoritmo.

Algoritmo Bully (3)



d) Il processo 6 informa il processo 5 di fermarsi

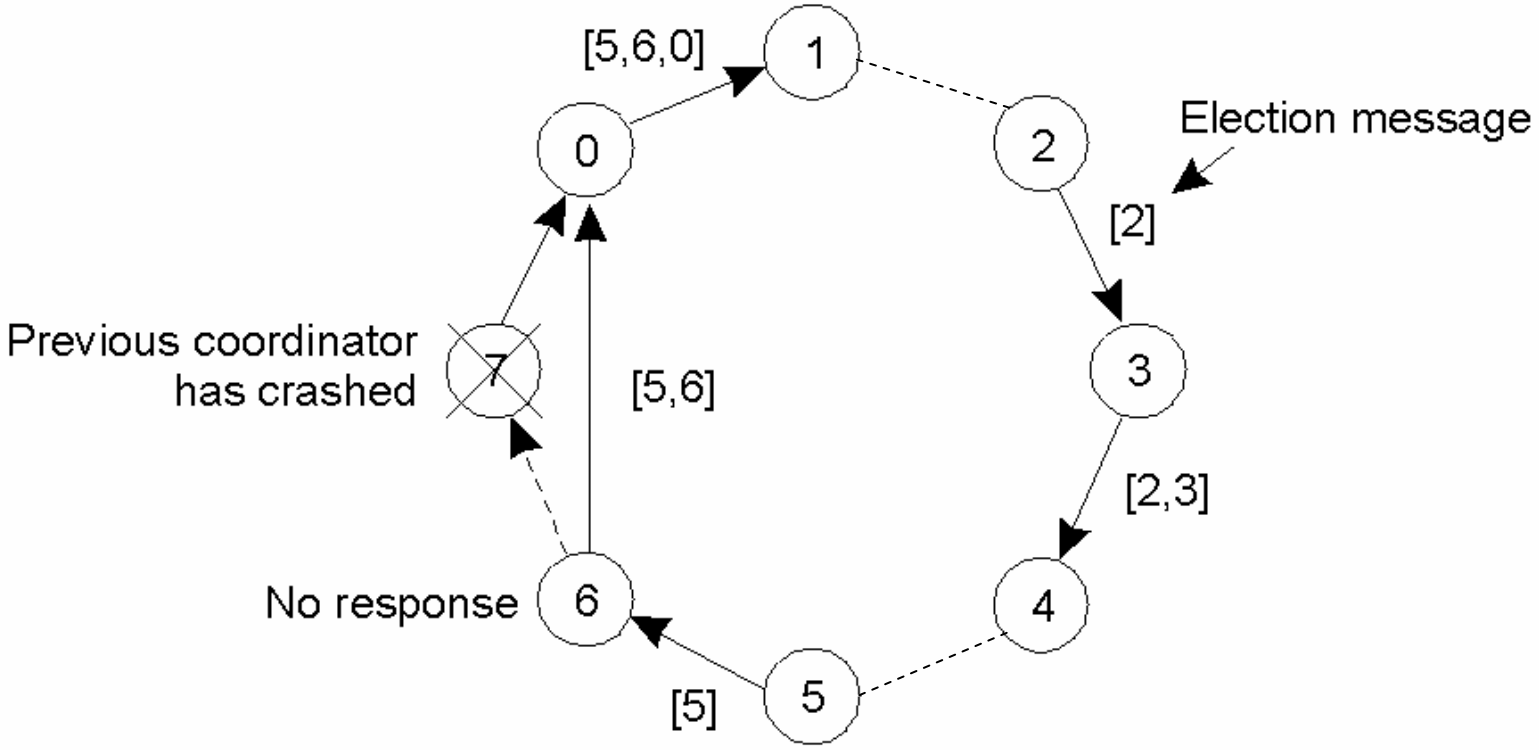
e) Il processo 6 diventa il coordinatore e informa tutti

Algoritmo Ring (1)

Algoritmo di elezione che fa uso di un anello:

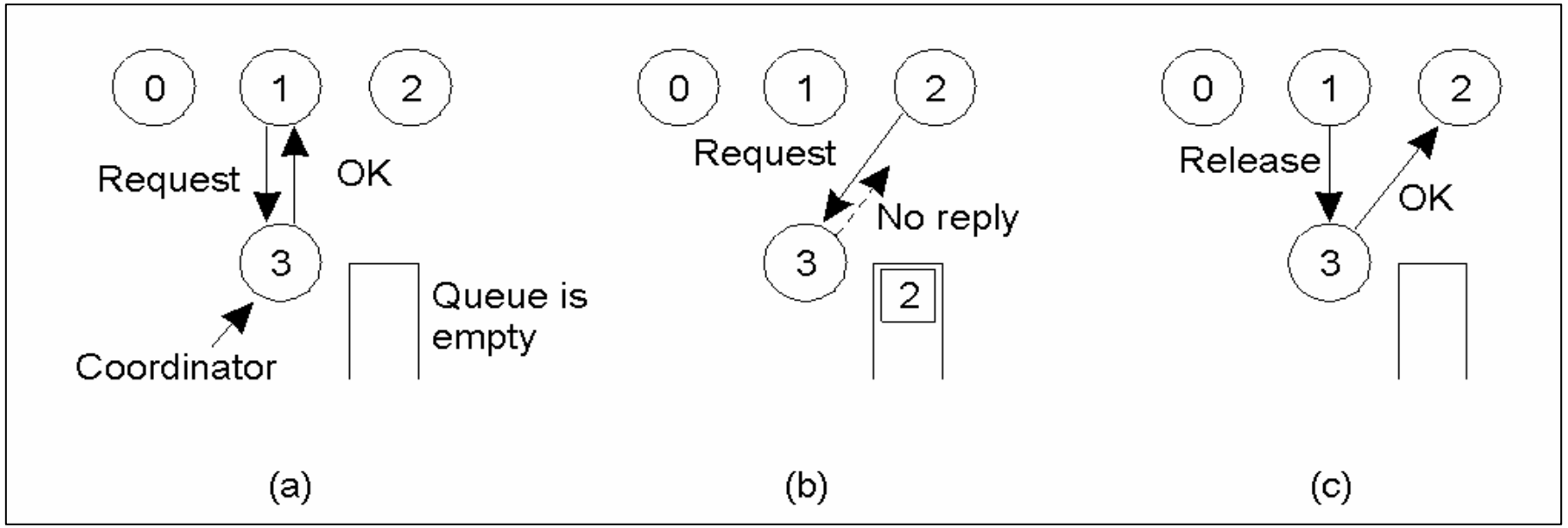
1. Ogni processo conosce chi è il suo successore
2. L'algoritmo di elezione è iniziato da un processo che invia un messaggio *ELECTION* con il suo ID al suo successore.
3. Ogni mittente aggiunge il suo ID al messaggio.
4. Quando il messaggio ritorna all'iniziatore, esso controlla il valore maggiore e invia il messaggio *COORDINATOR* sull'anello con il numero del nuovo coordinatore.

Algorithm Ring (2)



Algoritmo di elezione che fa uso di un anello: I processi 2 e 5 rilevano il crash del coordinatore e avviano l'algoritmo. Alla fine il processo 6 sarà eletto coordinatore.

Mutua Esclusione: Un Algoritmo Centralizzato



a) Il processo 1 chiede al coordinatore il permesso per entrare in una regione critica. Il permesso è concesso

b) Il processo 2 chiede al coordinatore il permesso per entrare in una regione critica. Il coordinatore non risponde.

c) Quando il processo 1 esce dalla regione critica, informa il coordinatore, quindi questo risponde al processo 2

Un Algoritmo Distribuito (1)

Ipotesi: La trasmissione dei messaggi è affidabile ed esiste un ordinamento totale del tempo.

a) Quando un processo vuole entrare in una regione critica invia a tutti i processi

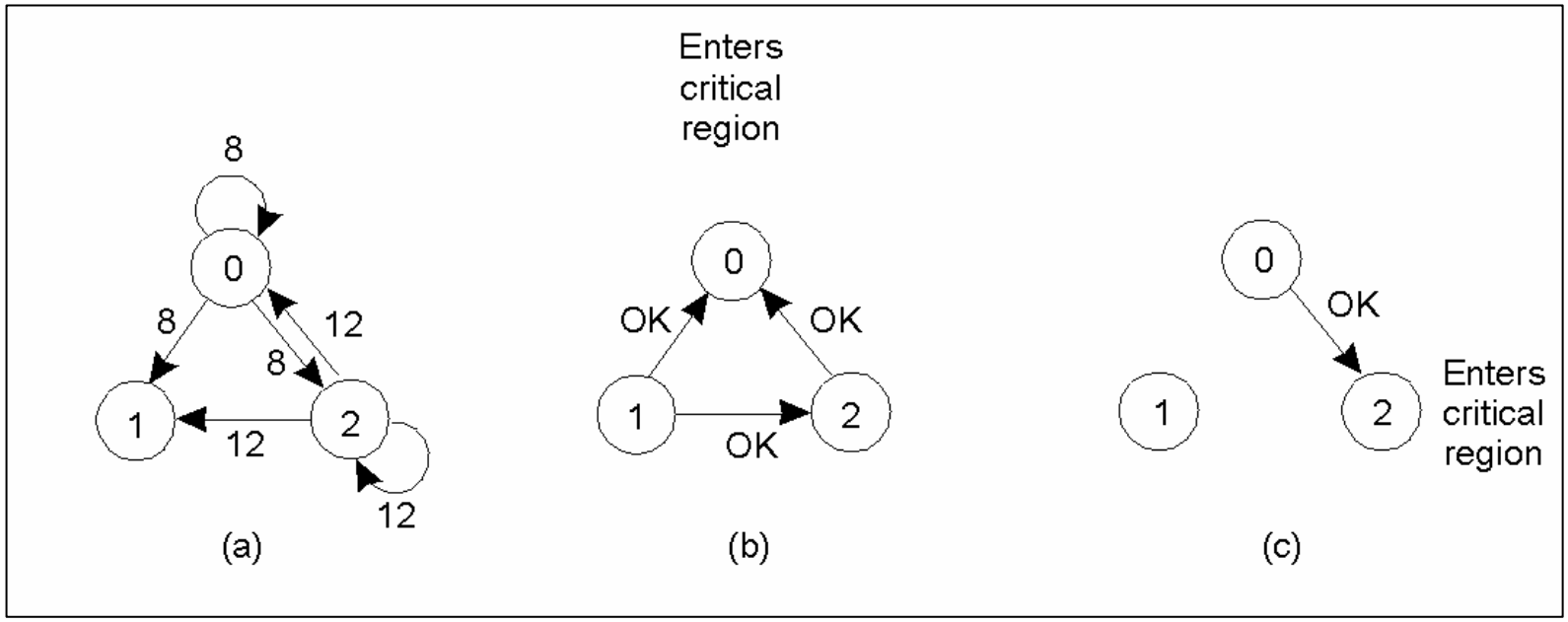
< *cr_name*, *proc_id*, *time* >

b) Quando un processo riceve il messaggio

1. Se non è in una regione critica e non vuole entrarci, invia un OK
2. Se è in una regione critica non risponde e accoda il messagio
3. Se vuole entrare in una regione critica, confronta il timestamp della sua richiesta con il timestamp del messaggio ricevuto, il più basso vince
4. Quando un processo esce da una regione critica invia OK a tutti i processi i cui messaggi erano stati accodati.

Funziona ma non è efficiente!

Un Algoritmo Distribuito (2)

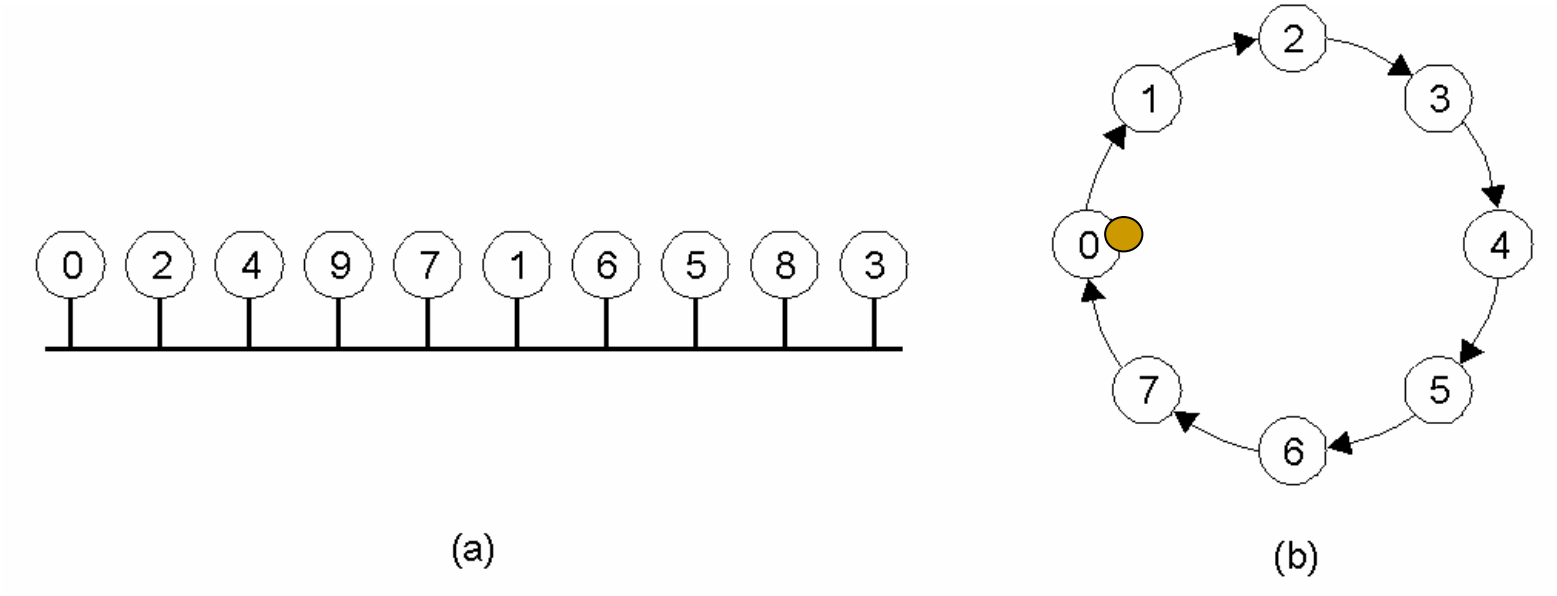


a) Due processi vogliono entrare nella stessa regione critica nello stesso istante

b) Il processo 0 ha il timestamp più basso (Ts=8), e vince.

c) Quando il processo 0 ha finito, invia un OK, quindi il processo 2 può accedere alla regione critica.

Algoritmo Token Ring



(a) Un gruppo di processi non ordinati in una rete.

(b) Un anello logico ordinato costruito etichettando i processi

1. Il processo 0 ha un **token** che fa circolare sull'anello.
2. Un processo *N* che possiede il token può accedere alla regione critica o può passarlo al processo *N+1*.

Confronto

Algoritmo	Messaggi per entrare / uscire	Ritardo prima di entrare (in messaggi)	Problemi
Centralizzato	3	2	Crash del coordinatore
Distribuito	2 (n – 1)	2 (n – 1)	Crash di un processo
Token ring	1 a ∞	Da 0 a n – 1	Token perso, processo in crash

Una comparazione dei tre algoritmi di mutua esclusione.

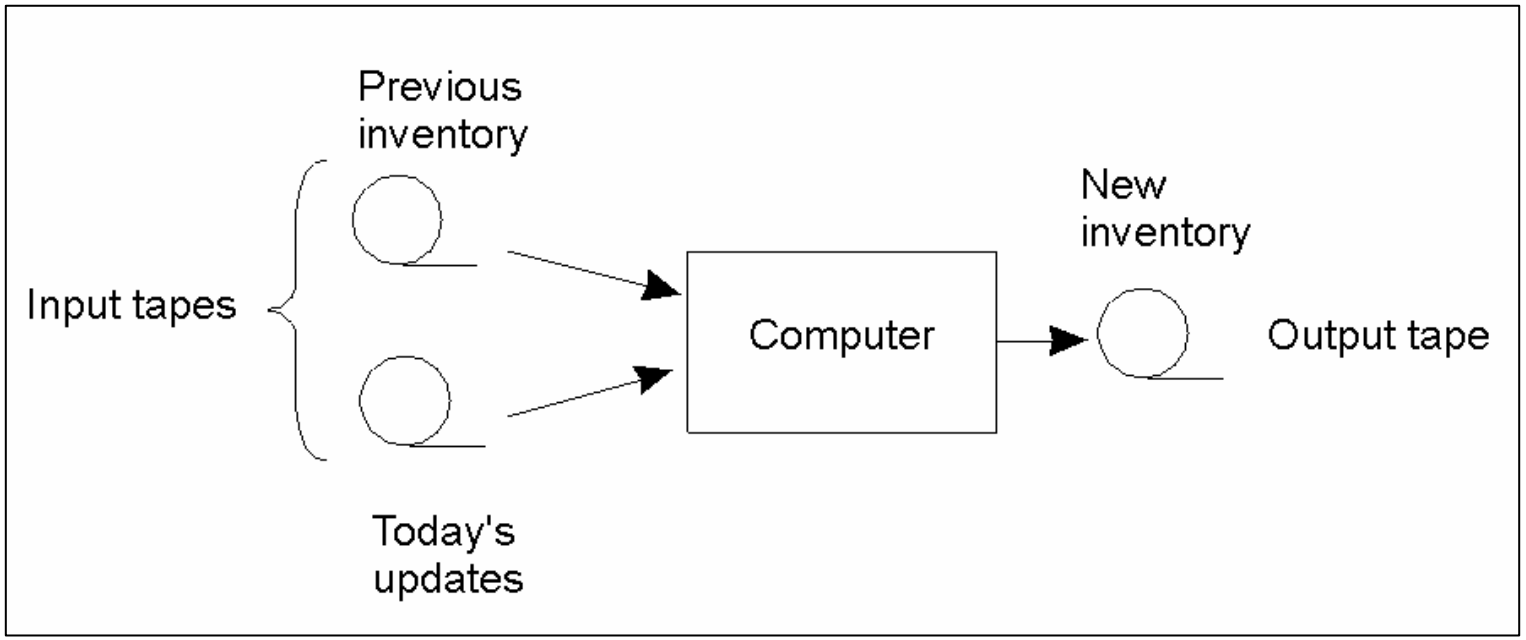
Transazioni (1)

- Le Transazioni sono composte da un insieme di operazioni che rispettano la proprietà tutto-o-niente (all-or-nothing).

- Esempio di transazione con 2 operazioni:
  - Op1: Prelievo 1000 € dal conto 1
  - Op2: Deposito 1000 € dal conto 2.

  Se si ha un fallimento tra Op1 and Op2, la transazione deve essere annullata (transaction abort).

Transazioni (2)



Aggiornamento di un nastro è fault tolerant.

Transazioni (3)

Primitive speciali sono definite per le transazioni.

Primitive	Description
BEGIN_TRANSACTION	Inzia una transazione
END_TRANSACTION	Termina una transazione e effettua il commit
ABORT_TRANSACTION	Annulla la transazione e riporta i valori precedenti
READ	Legge i dati da un file, una tabella, o altro
WRITE	Scrive i dati da un file, una tabella, o altro

Esempi di primitive per transazioni

Transazioni (4)

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 full =>
ABORT_TRANSACTION

(b)

(a) Una Transazione per prenotare tre voli ha successo

(b) Una Transazione fallisce per la impossibilità di prenotare il terzo volo.

Transazioni (5)

ACID PROPERTIES

- **ATOMIC**: la transazione è indivisibile
- **CONSISTENT**: la transazione non viola gli invarianti del sistema
- **ISOLATED**: transazioni concorrenti non interferiscono tra loto (SERIALIZZABILE)
- **DURABLE**: dopo il commit, le modifiche sono permanenti.

Transazioni Innestate e Distribuite

- Oltre alle transazioni “piatte” vengono usate altri modelli di transazioni

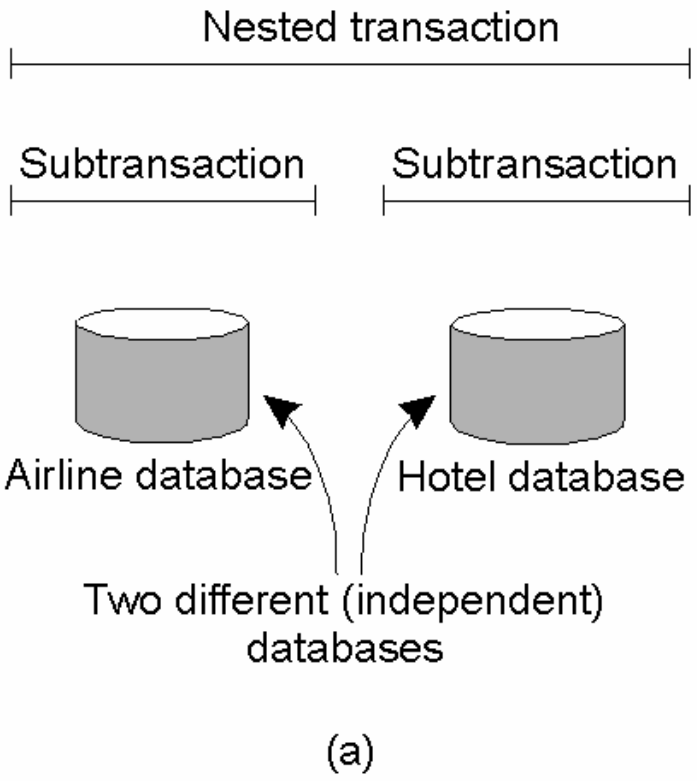
  Una **nested transaction** (transazione innestata) è una transazione che è logicamente decomposta in un insieme di sotto-transazioni

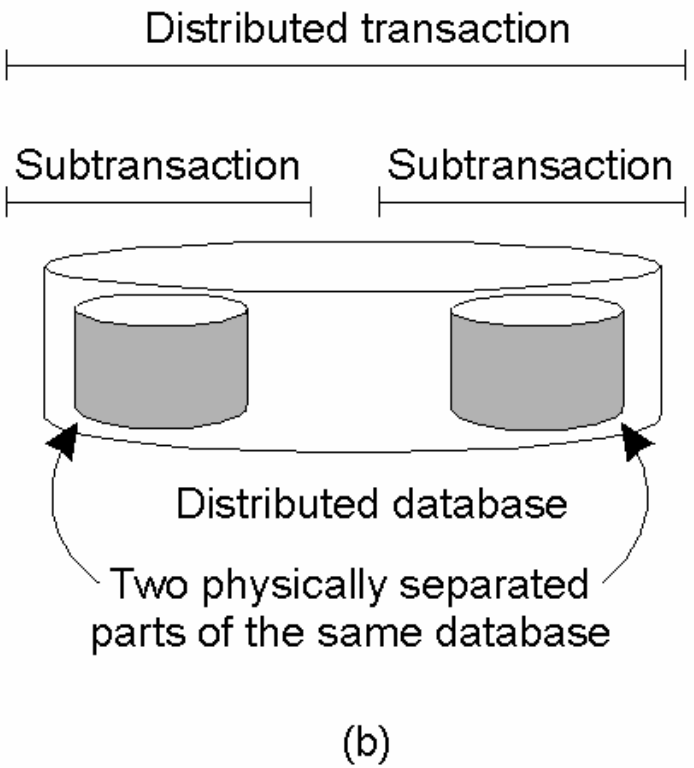
  Un meccanismo di *hierarchical abort* deve essere previsto.

  Una **distributed transaction** (transazione distribuita) è una transazione “piatta” che opera su dati distribuiti.

  Un meccanismo di *distributed locking* è necessario.

Transazioni Distribuite



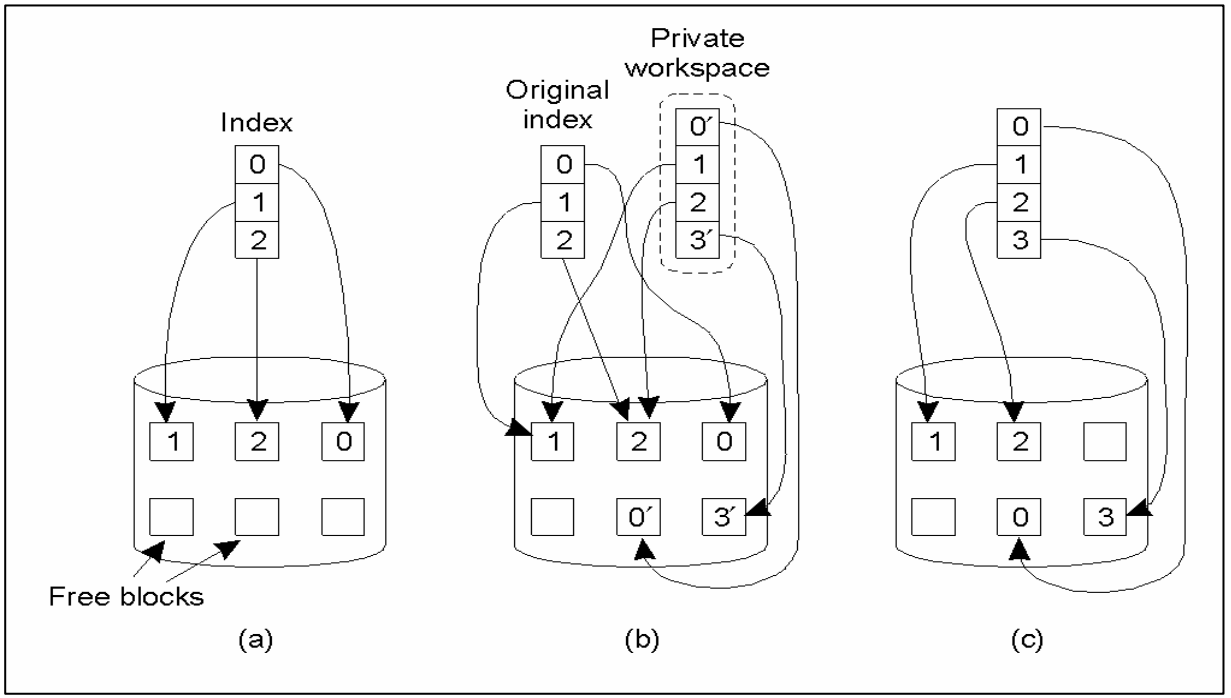


(a) Una transazione innestata

(b) Una transazione distribuita

Workspace Privato

Il *Private workspace* è un metodo per implementare transazioni **atomiche**.



(a) Il file index e blocchi di disco per un file di tre blocchi

(b) La situazione dopo una transazione ha modificato il blocco 0 e aggiunto il blocco 3

(c) Dopo il commit della transazione.

Writeahead Log

Il *Writeahead log* è un altro metodo per implementare le transazioni atomiche

x = 0;
y = 0;
BEGIN_TRANSACTION;
x = x + 1;
y = y + 2
x = y * y;
END_TRANSACTION;
(a)
	Log
	
	[x = 0 / 1]
	
	(b)
		Log
		
		[x = 0 / 1]
		[y = 0/2]
		
		(c)
			Log
			
			[x = 0 / 1]
			[y = 0/2]
			[x = 1/4]
			
			(d)

(a) Una transazione

(b) – (d) Il log prima che ogni statement sia eseguito

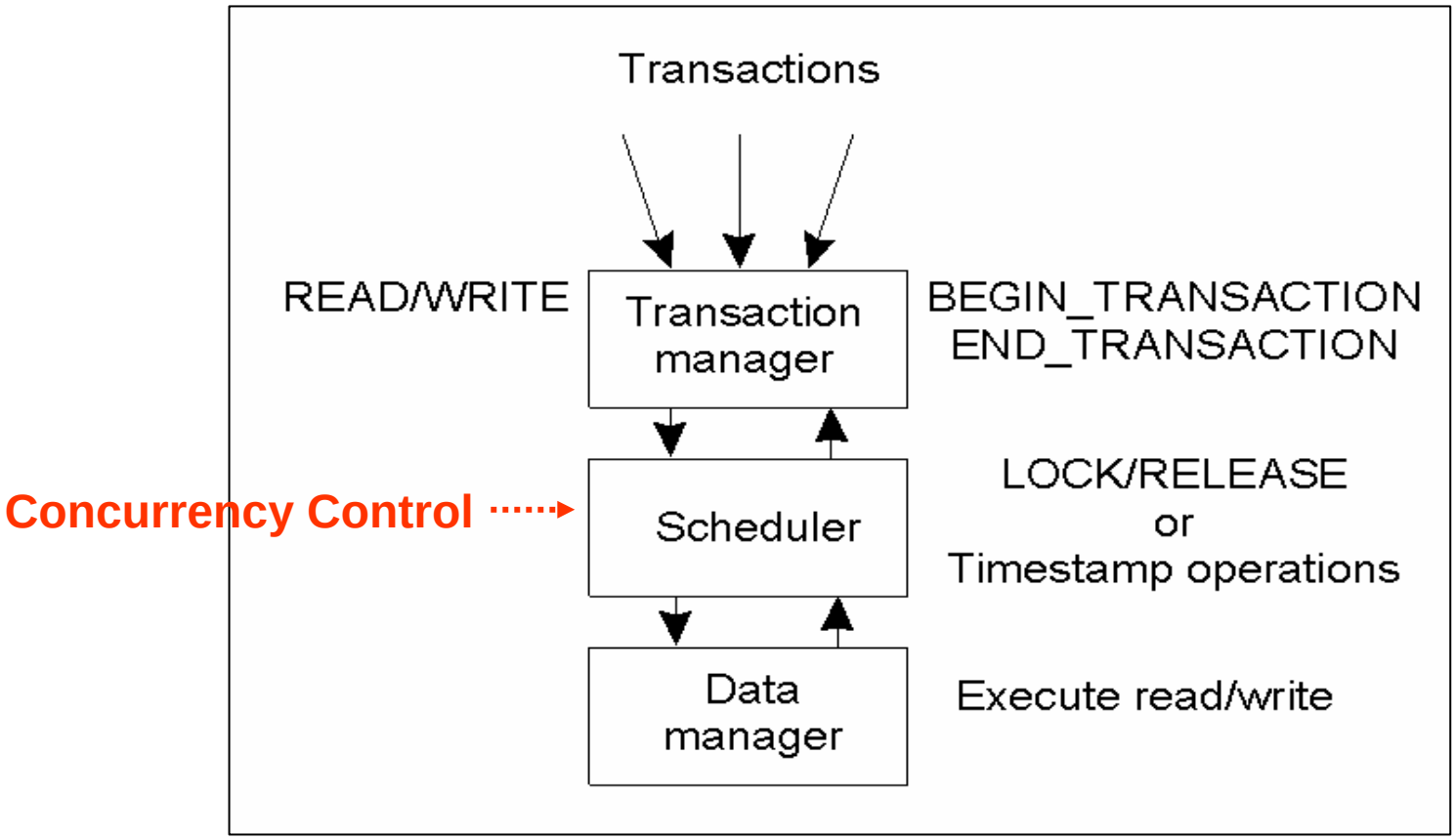
In caso di un abort viene eseguita l'operazione di **Rollback**

Controllo della Concorrenza (1)

- Il **controllo della Concorrenza** è usato per assicurare la **SERIALIZZABILITA'** : transazioni concorrenti non interferiscono tra loro.

- Il risultato finale dovrà essere uguale a quello ottenuto da una esecuzione sequenziale (in un'ordine qualsiasi) delle due transazioni.

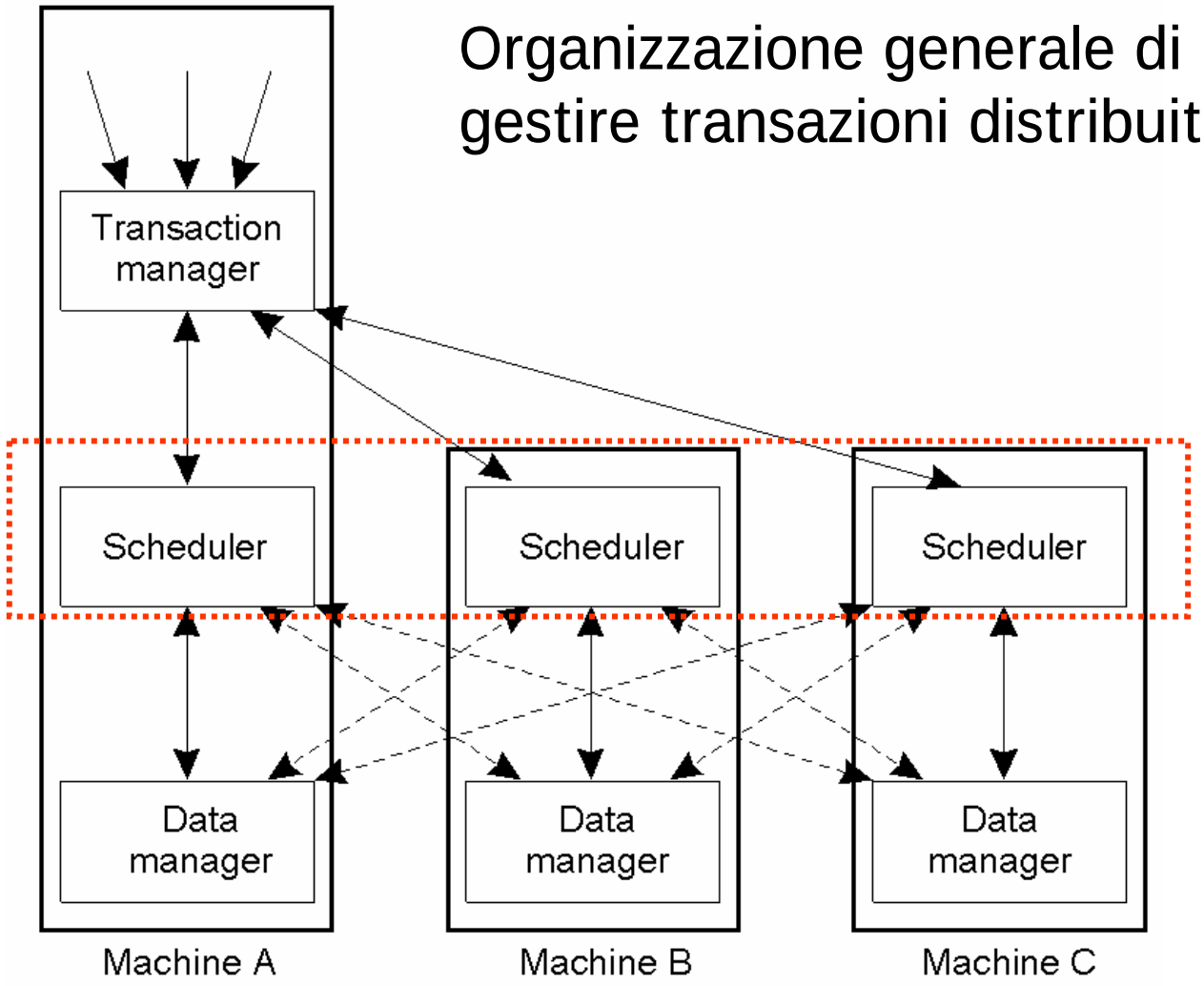
Controllo della Concorrenza (2)



Organizzazione generale di manager per gestire le transazioni.

Controllo della Concorrenza (3)

Organizzazione generale di manager per gestire transazioni distribuite.



Serializzabilità

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)

(a) – (c) Tre transazioni T1, T2, e T3

Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	**Illegal**

Tempo -->

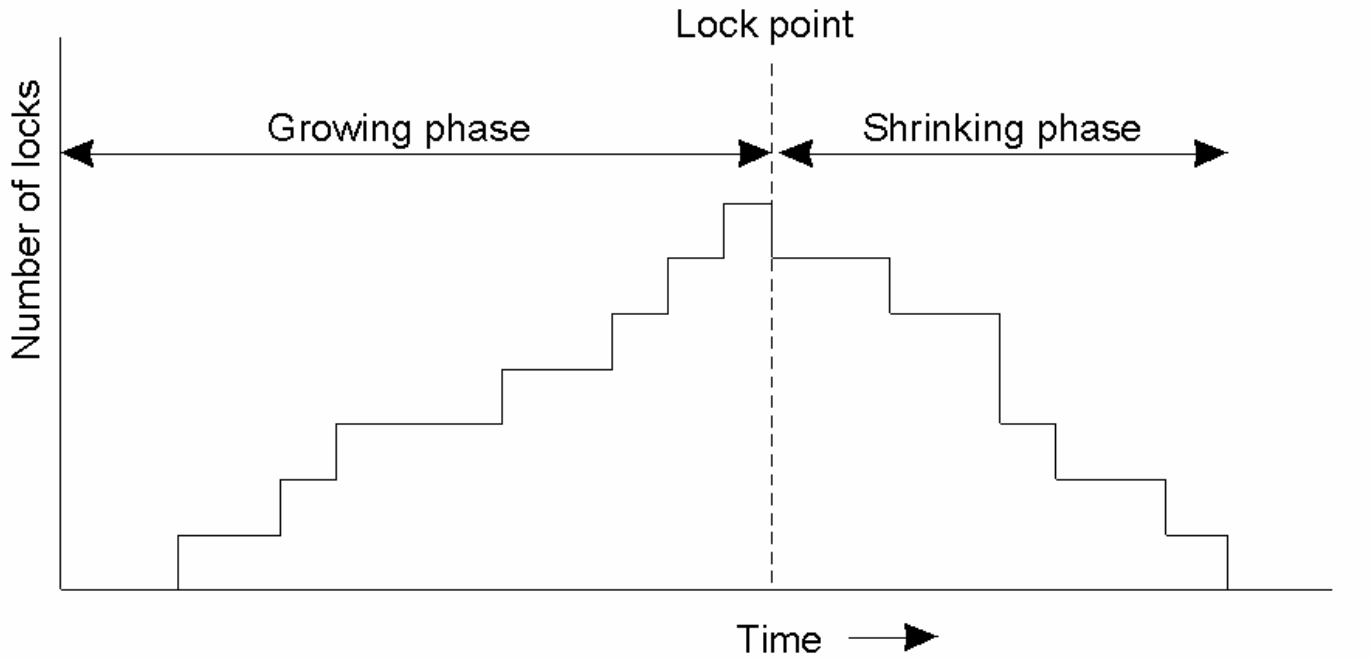
Possibili sequenze di esecuzione

Conflitto tra Operazioni

- Due operazioni sono in conflitto se operano sulla stessa istanza di un dato e una di queste operazionie è una **write**. (write-read, write-write)
- Il Controllo della concorrenza deve trovare uno schedule per le **operazioni in conflitto** (attraverso una corretta **sincronizzazione**).
- Tecniche usate:
  - **Two-phase locking**
  - **Timestamp ordering**

Two-Phase Locking (1)

- Nel **Two-phase locking** lo scheduler prima acquisisce tutti i locks necessari durante la **growing phase** e li rilascia nella **shrinking phase**.



Two-Phase Locking (2)

Regole di base:

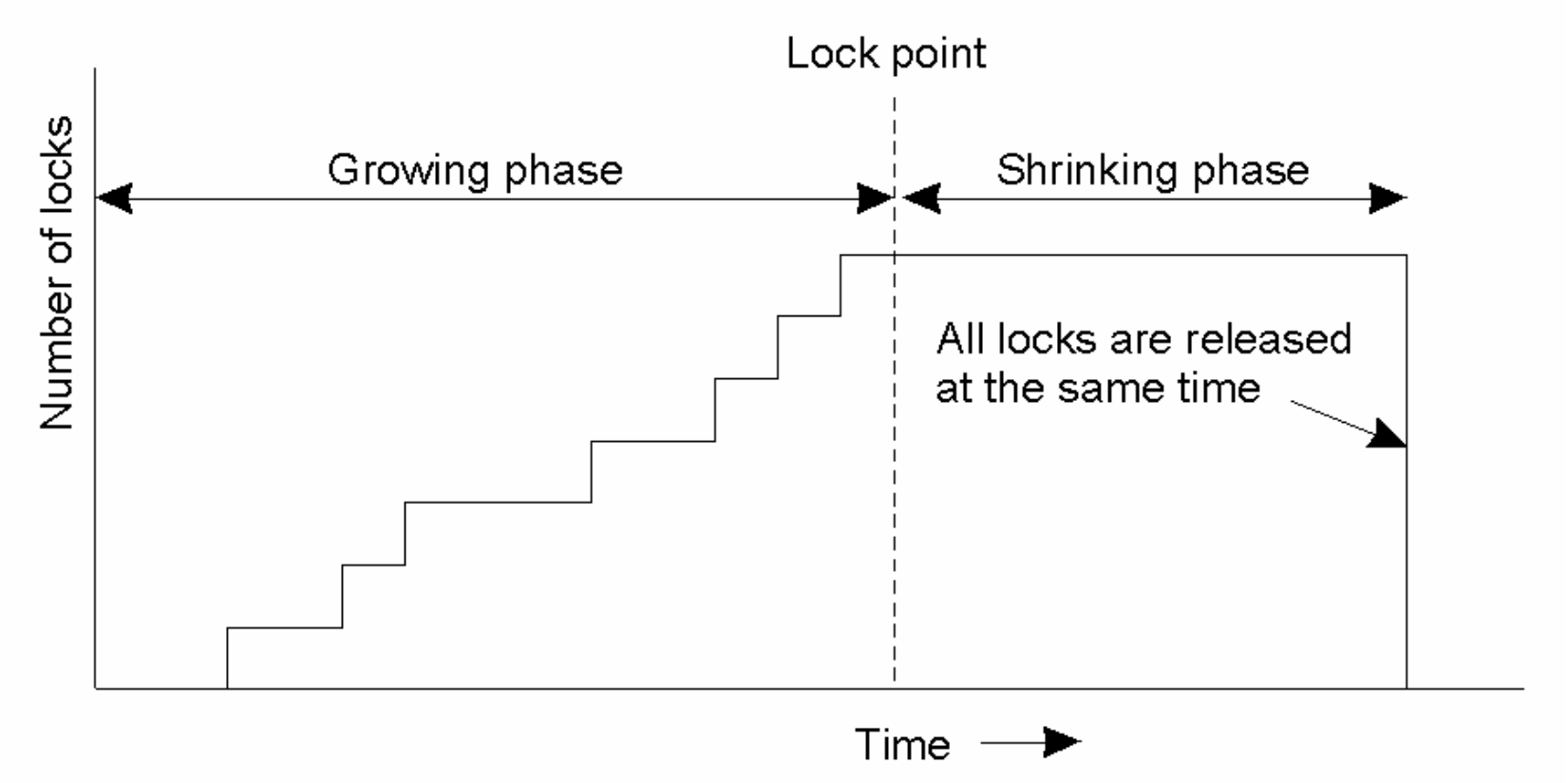
1. Quando lo scheduler riceve una operazione su *x* controlla se l'operazione confligge con un'altra operazione alla quale è stato garantito un lock. Se non vi è conflitto, lo scheduler acquisice il lock per *x* e chiede al data manager di eseguire l'operazione.

2. Lo scheduler non rilascia il lock per x finchè il data manager non abbia eseguito l'operazione.

3. Quando lo scheduler ha rilasciato un lock per conto di T, non acquisisce un altro lock rischiesto eventualmente da T.

Queste tre regole garantiscono la serializzabilità.

Strict Two-Phase Locking



Nel Strict two-phase locking i lock sono rilasciati solo quando una transazioneè completata. Si evitano **abort in cascata**.

Two-Phase Locking (3)

- Si può avere Deadlock.
  - Potrebbe essere evitato acquisendo tutti i lock necessari in un ordine prestabilito.

    oppure
  - Costruendo un grafo di lock e determinando i cicli nel grafo.
- Il Timeout può essere usato per far rilasciare rilasciare i lock ad un processo dopo un dato intervallo di tempo.

Two-Phase Locking (4)

- 2PL Centralizzato vs 2PL Distribuito

- Nel 2PL Centralizzato viene usato un singolo lock manager responsabile di acquisire e rilasciare I lock.

- Nel 2PL Distribuito sono usati più lock manager che gestiscono le operazioni sui lock e comunicano con i data manager locali e/o remoti.

Pessimistic Timestamp Ordering (1)

- Nel controllo della concorrenza usando timestamp, ogni transazione ha un timestamp ***ts(T)*** all'istante in cui inizia.

- Ogni dato ha un **read timestamp** $ts_{RD}(T)$ e un **write timestamp** $ts_{WR}(T)$

- Se due operazioni sono in conflitto, il data manager elabora quella con il timestamp più basso.

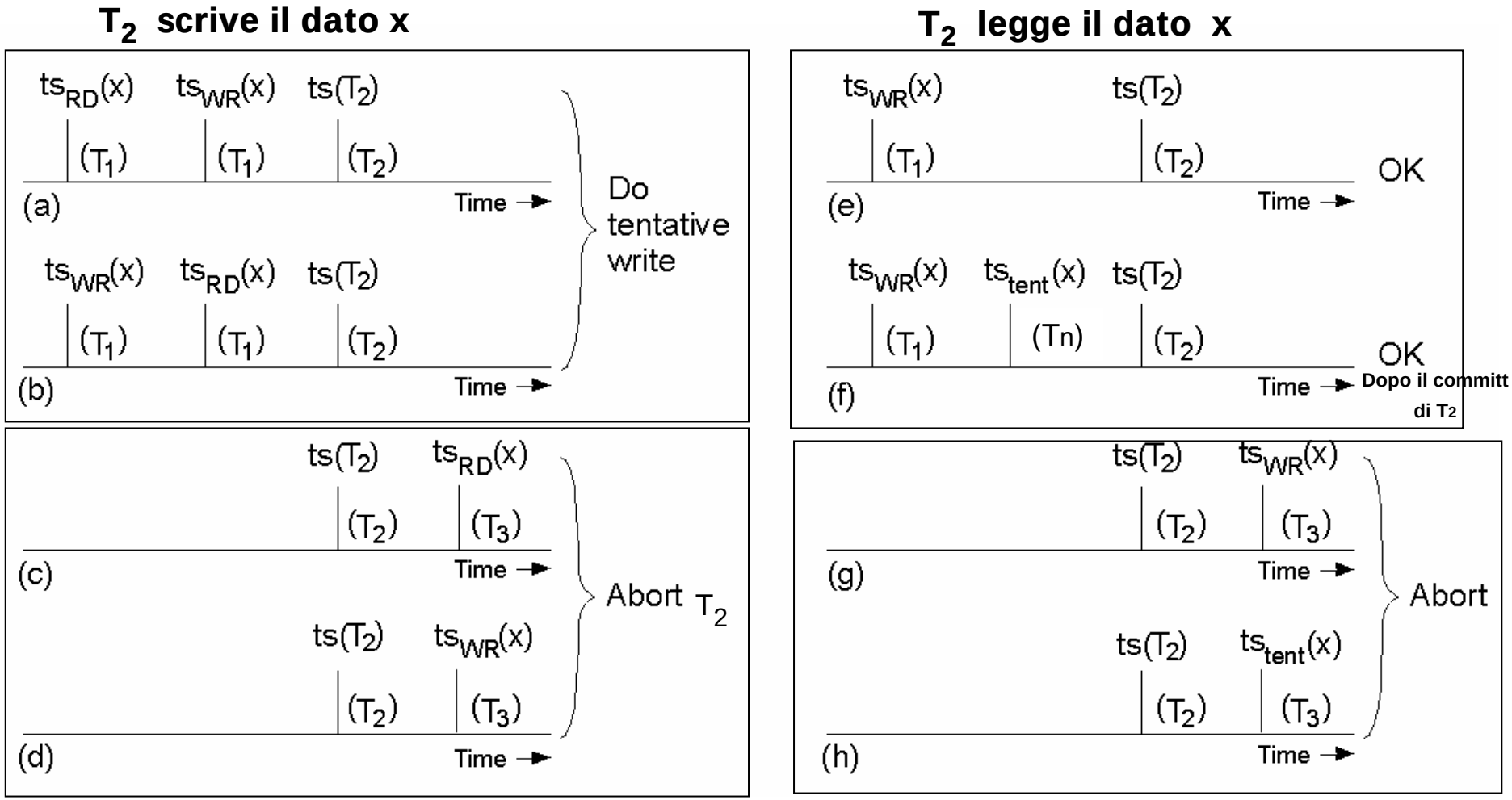
Pessimistic Timestamp Ordering (2)

- I timestamp sono usati per l'abort di operazioni: quando una transazione rileva un timestamp più grande essa fallisce (genera un abort).
- Vediamo un esempio con tre transazioni: $\mathbf{T_1}$, $\mathbf{T_2}$, $\mathbf{T_3}$

$$ts(T_1) << ts(T_2) < ts(T_3)$$

- T_1 è stata eseguita molto prima di T_2 e T_3 che sono eseguite in concorrenza e T_1 ha usato i dati che verranno usati da T_2 e T_3.

Pessimistic Timestamp Ordering (3)



Esempi di concurrency control usando timestamp.

2PL e Timestamp Ordering

- Il Two-phase locking può causare deadlock, sono quindi necessarie tecniche di deadlock detection.
- L'ordinamento con timestamp, al contrario, è deadlock free.
- L'**Optimistic concurrency control** è un approccio alternativo alla strategia pessimistica. I conflitti sono verificati prima del commit delle transazioni.
- Se qualche dato è stato modificato dopo l'inizio della transazione, la modifica viene annullata per l'abort della transazione.