Coordinamento e Sincronizzazione nei Sistemi Distribuiti

Cooperazione e Sincronizzazione

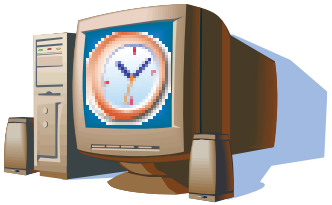
- In un sistema distribuito i processi, oltre a dover comunicare, spesso hanno necessità di coordinarsi e di sincronizzarsi.

- Accesso coordinato alle risorse, ordinamento di eventi, cooperazione tramite sincronizzazioni sono funzionalità necessarie nei sistemi distribuiti.

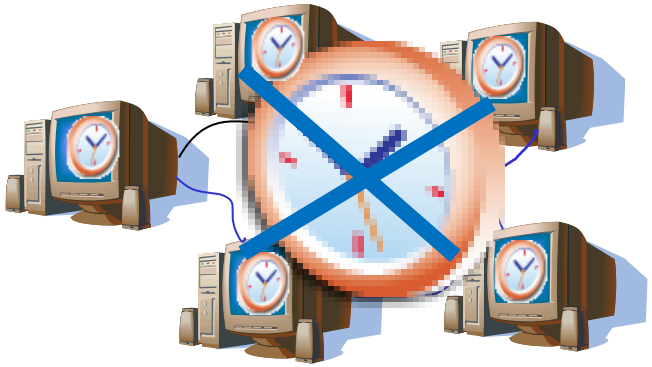
- Molte soluzioni distribuite hanno bisogno di meccanismi e protocolli di sincronizzazione e coordinamento. Senza di questi molte applicazioni non sarebbero possibili.

Sincronizzazione dei Clock

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



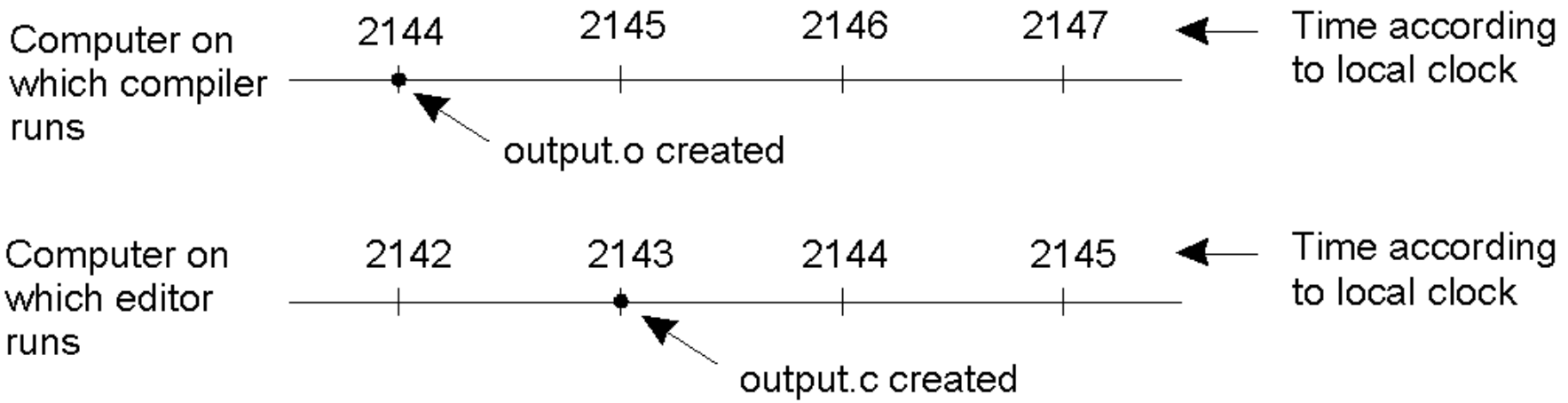
- In un sistema distribuito definire un tempo globale non è semplice.
  (*È impossibile garantire che i clocks avanzino tutti alla stessa esatta frequenza*)



- Soluzioni:
  - Clock synchronization
  - Logical clocks

Sincronizzazione dei Clock

Esempio: il programma *make*



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

Algoritmi di Sincronizzazione dei Clock

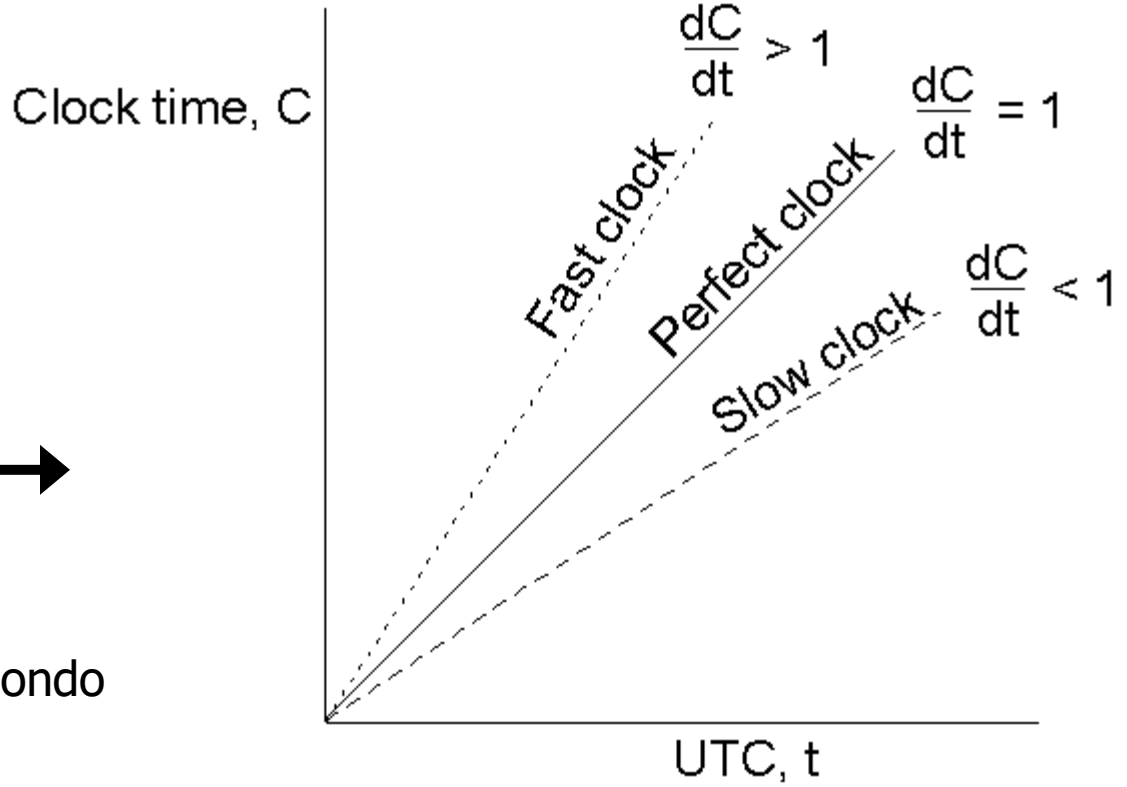
In un mondo perfetto:

$C(t) = t$

Invece →

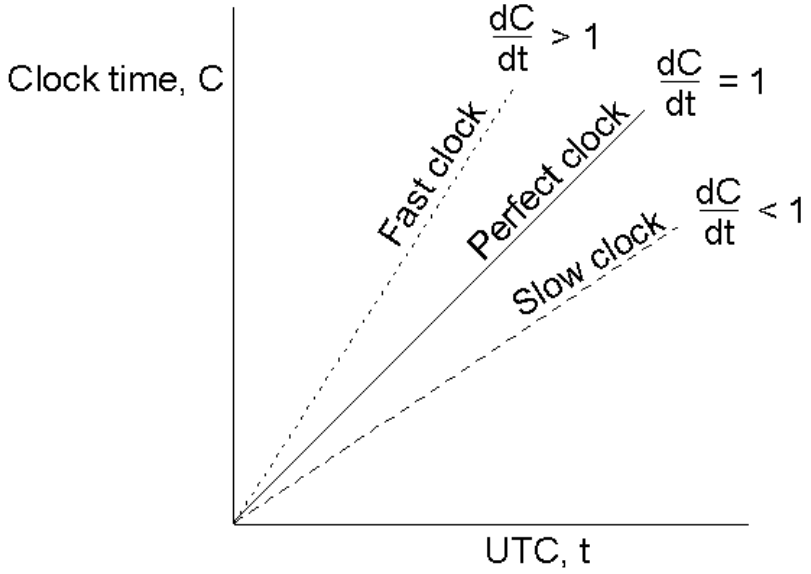
↓

Errore relativo = 10^{-6} sec al secondo
(31,5 sec/anno)



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

Algoritmi di Sincronizzazione dei Clock



Se esiste una costante ρ (scostamento massimo) tale che

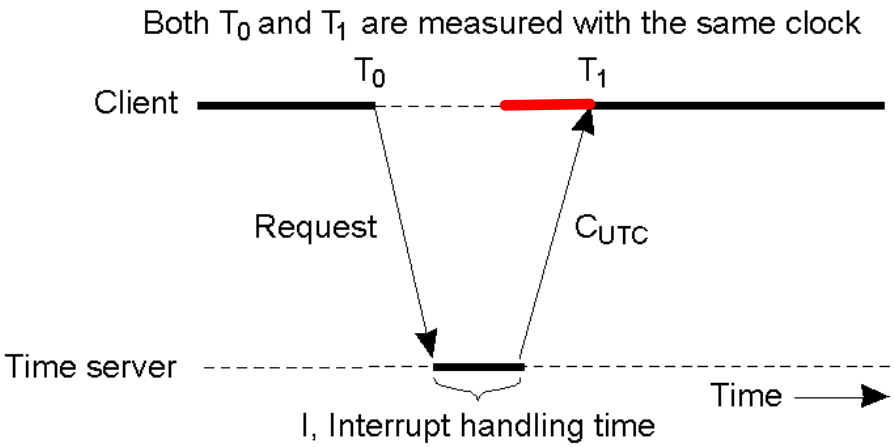
$$1-\rho \leq dC/dt \leq 1+\rho$$ (maximum drift rate)

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**.



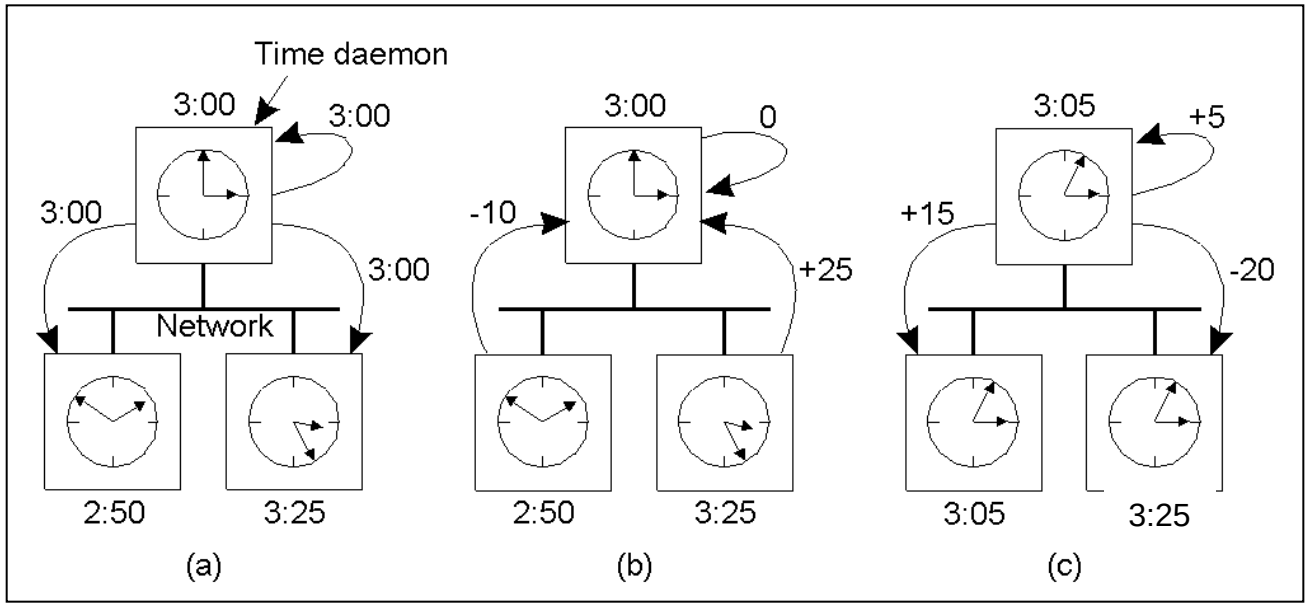
Due problemi:

- Il tempo non deve mai scorrere all' indietro (a causa del 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.

1. Il server chiede a tutte le macchine il valore del loro clock.
2. Ogni macchina risponde al server.
3. Il server invia a tutte le macchine il nuovo valore medio del clock.

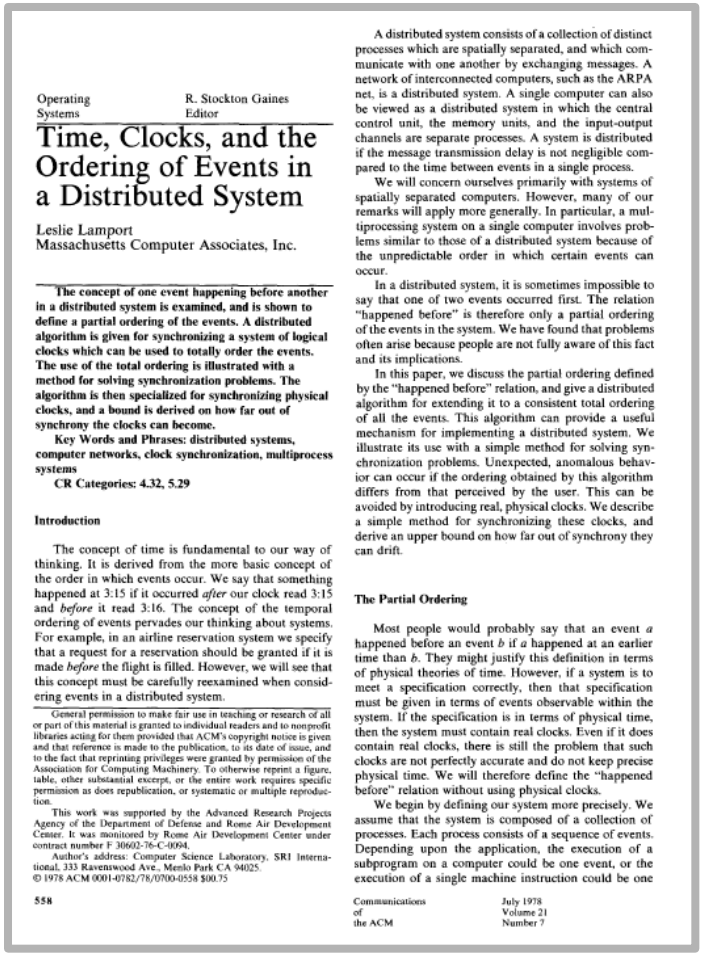


Clock Logici

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

- Proposta di **Lamport**:

1. **Se due processi non interagiscono non è necessario sincronizzare i loro clock.**
2. Quello che è importante per due o più processi interagenti è rispettare **l'ordine corretto in cui gli eventi realmente avvengono.**



Operating Systems — R. Stockton Gaines, Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems
CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event *a* happened before an event *b* if *a* happened at an earlier time than *b*. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

Timestamp di Lamport

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

  $happened\text{-}before$ ($\rightarrow$)

- $a \rightarrow b$ 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

- Consideriamo 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 nodi** di un sistema distribuito o concorrente (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*

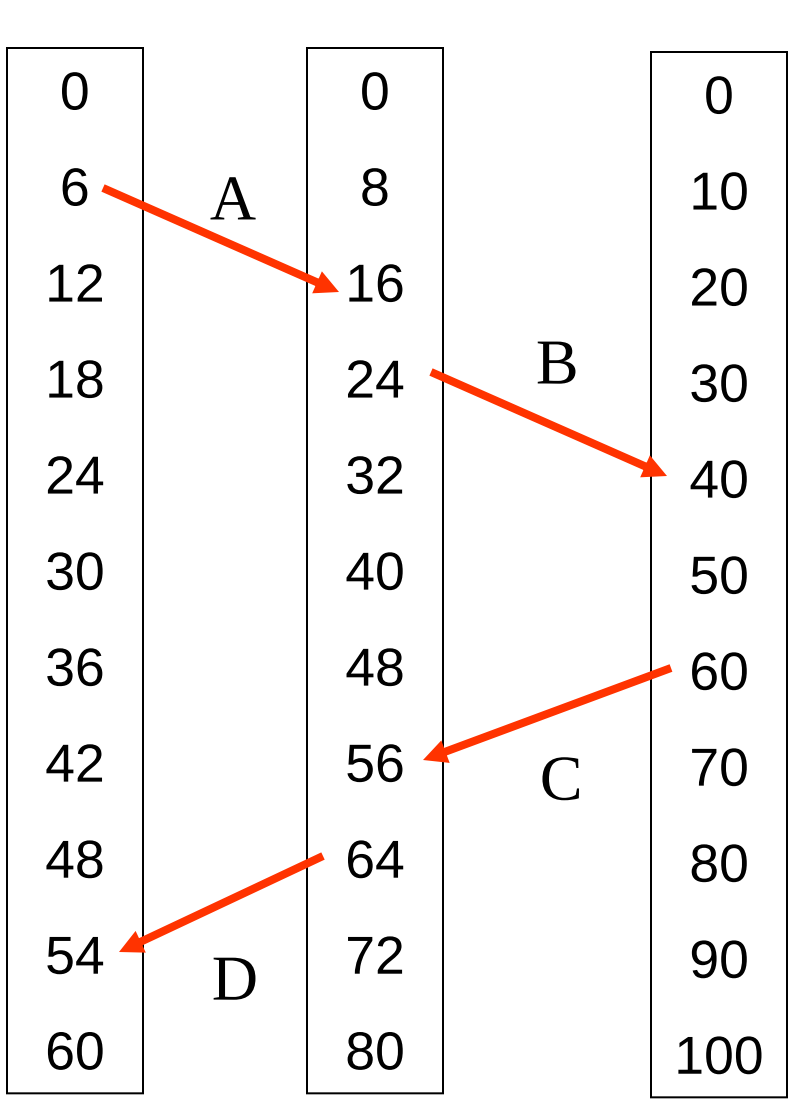
/ Cr è il valore del clock ricevuto dal processo remoto*

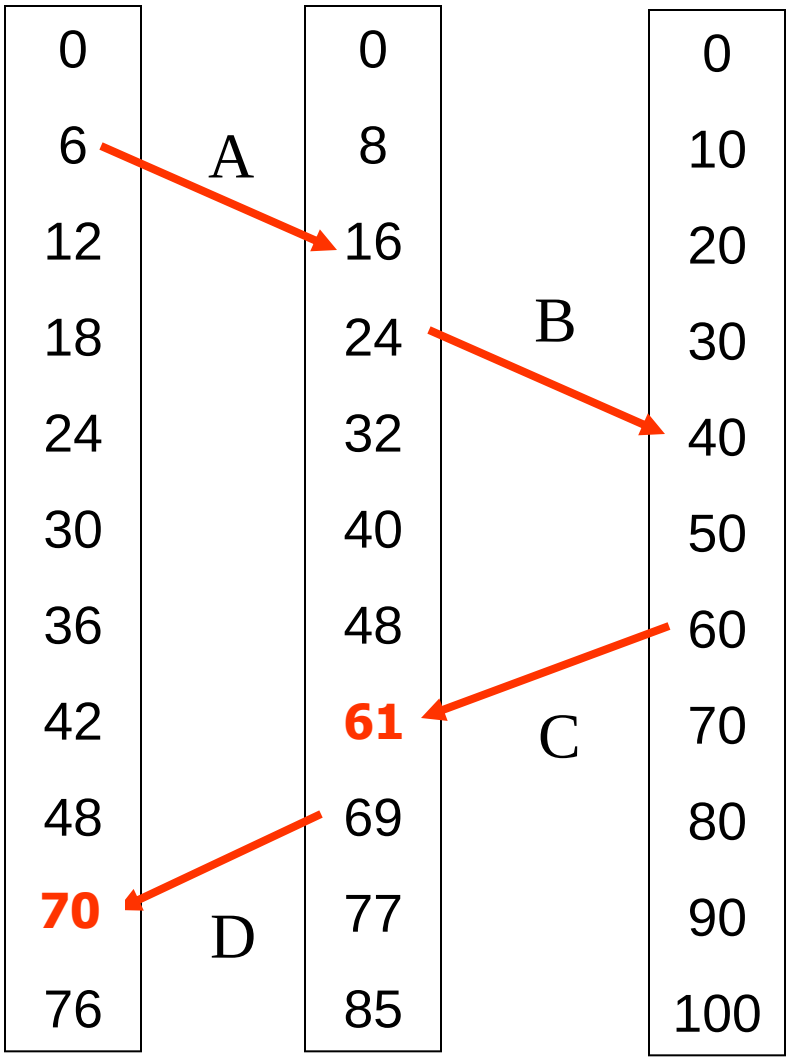
```
Var Cp: integer;

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

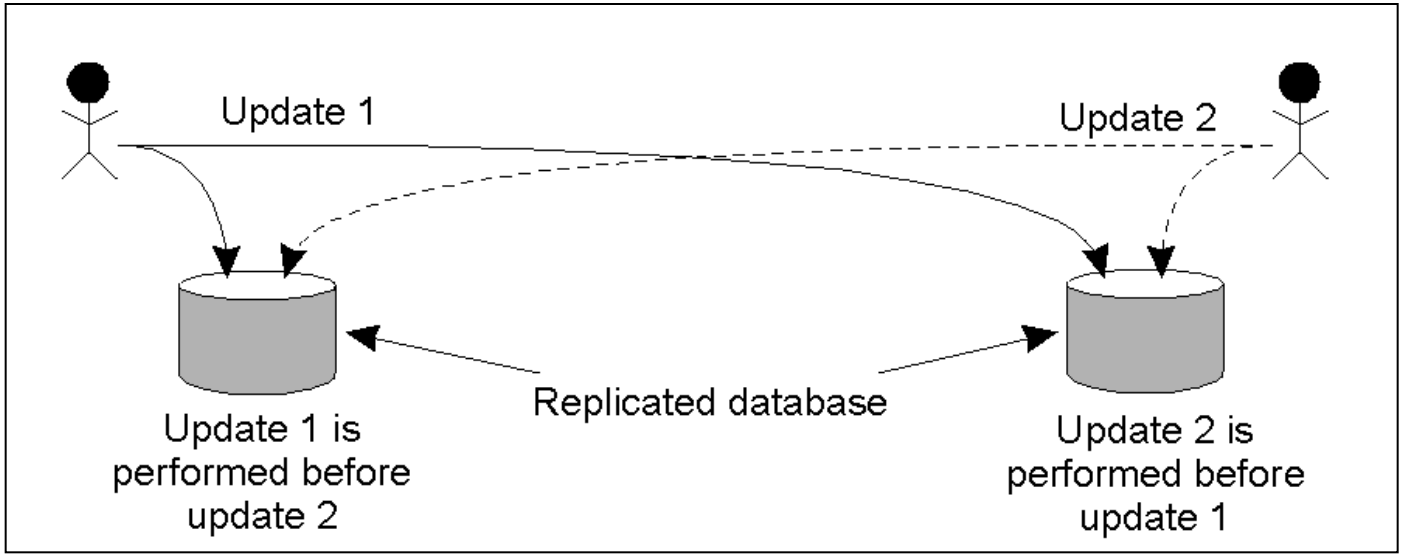




Tre processi, ognuno con il proprio clock. I clock hanno diverse velocità.
L' algoritmo di Lamport corregge il valore del clock.

Esempio: Totally-Ordered Multicasting

Database Replicato in due siti



Se si fanno due aggiornamenti 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 e non vengono persi;

  3. Ogni messaggio richiede l'invio di un acknowledgement;

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

Ogni processo ha la stessa copia della coda dei messaggi.

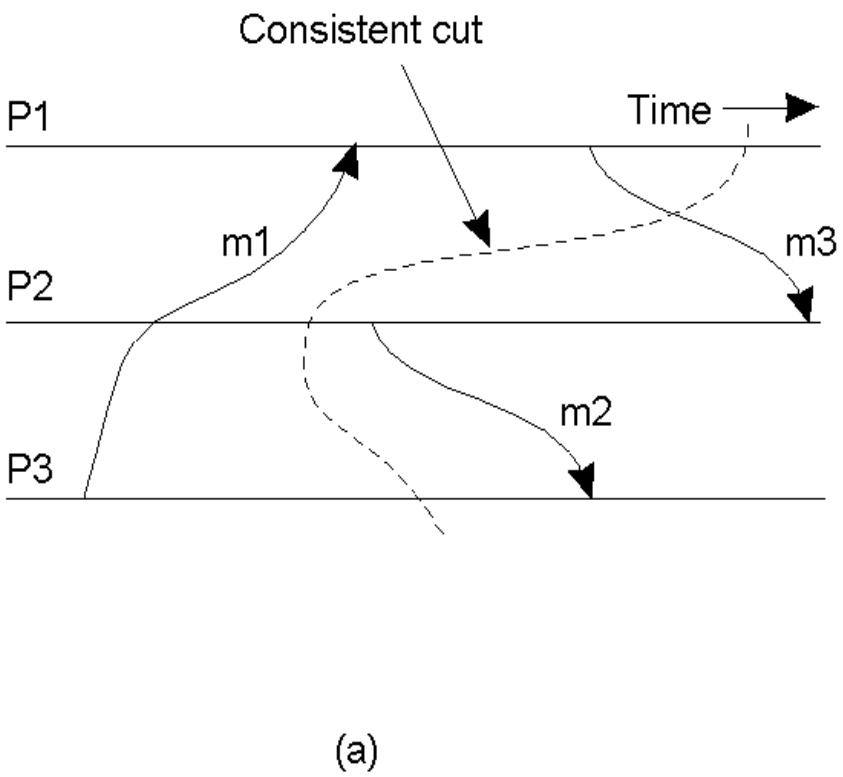
Global State (1)

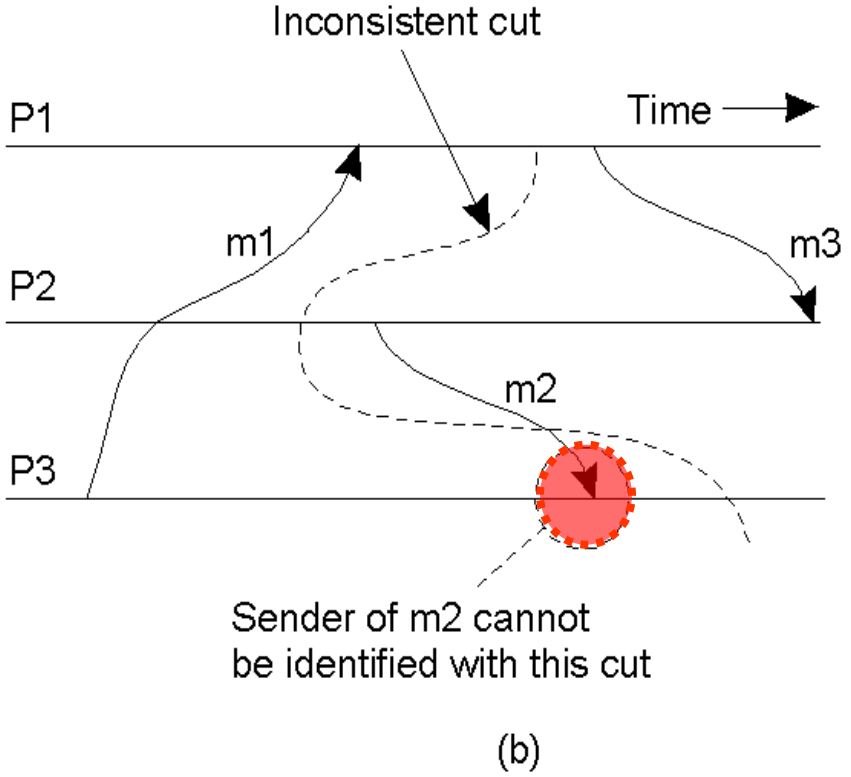
a) Lo **stato globale** di un sistema distribuito è dato dalla ***collezione degli stati locali*** di ogni processo più i ***messaggi in transito***.

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)





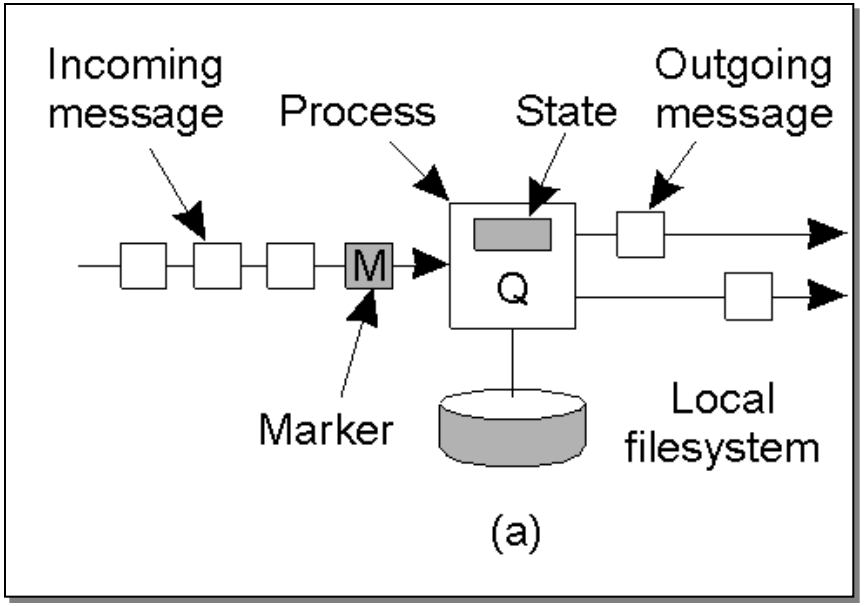
(a) Un "taglio" consistente

(b) Un "taglio" inconsistente

Global State (3)

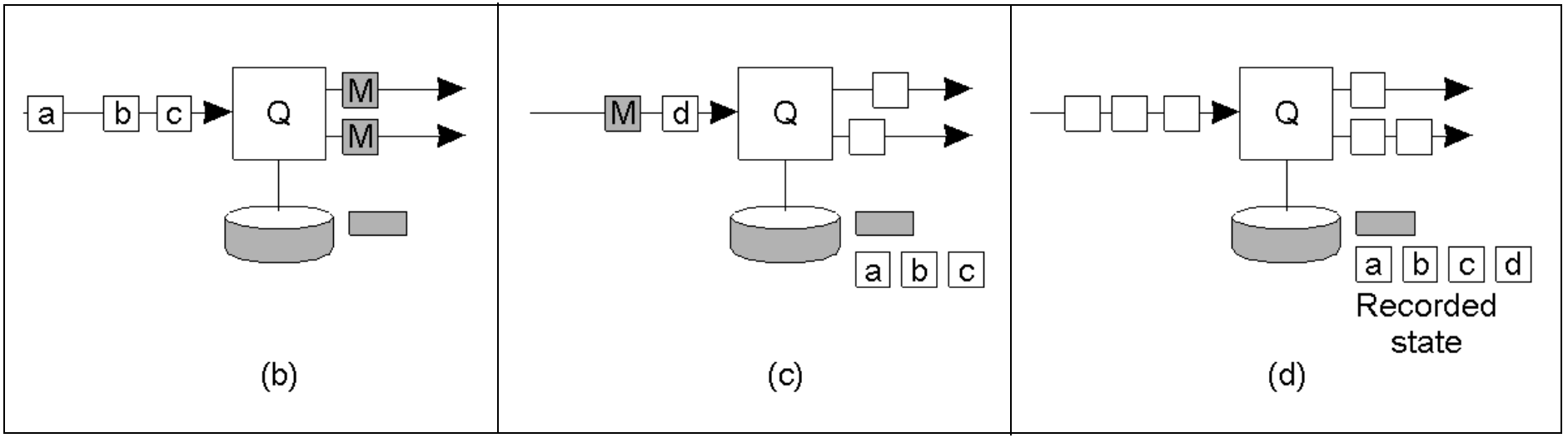
- Usando i distributed snapshots è possibile memorizzare uno stato globale.

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

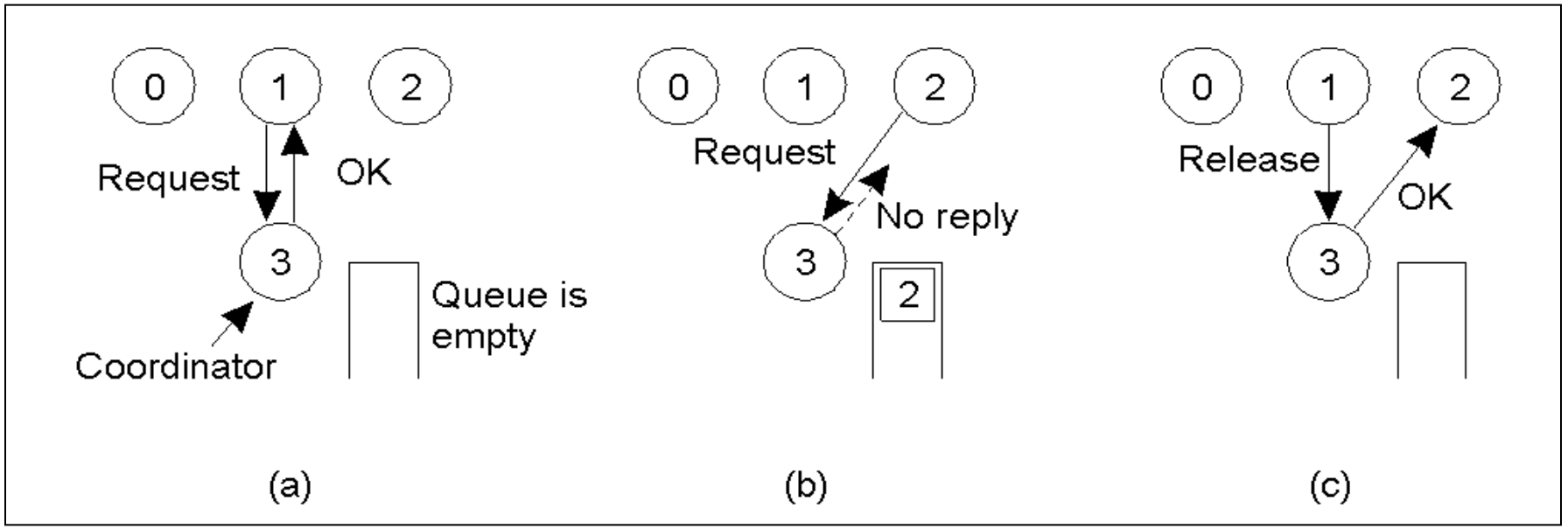
d. *Q* riceve un marker per i suoi canali di input e finisce memorizzando lo stato dei canali in ingresso.

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**.

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

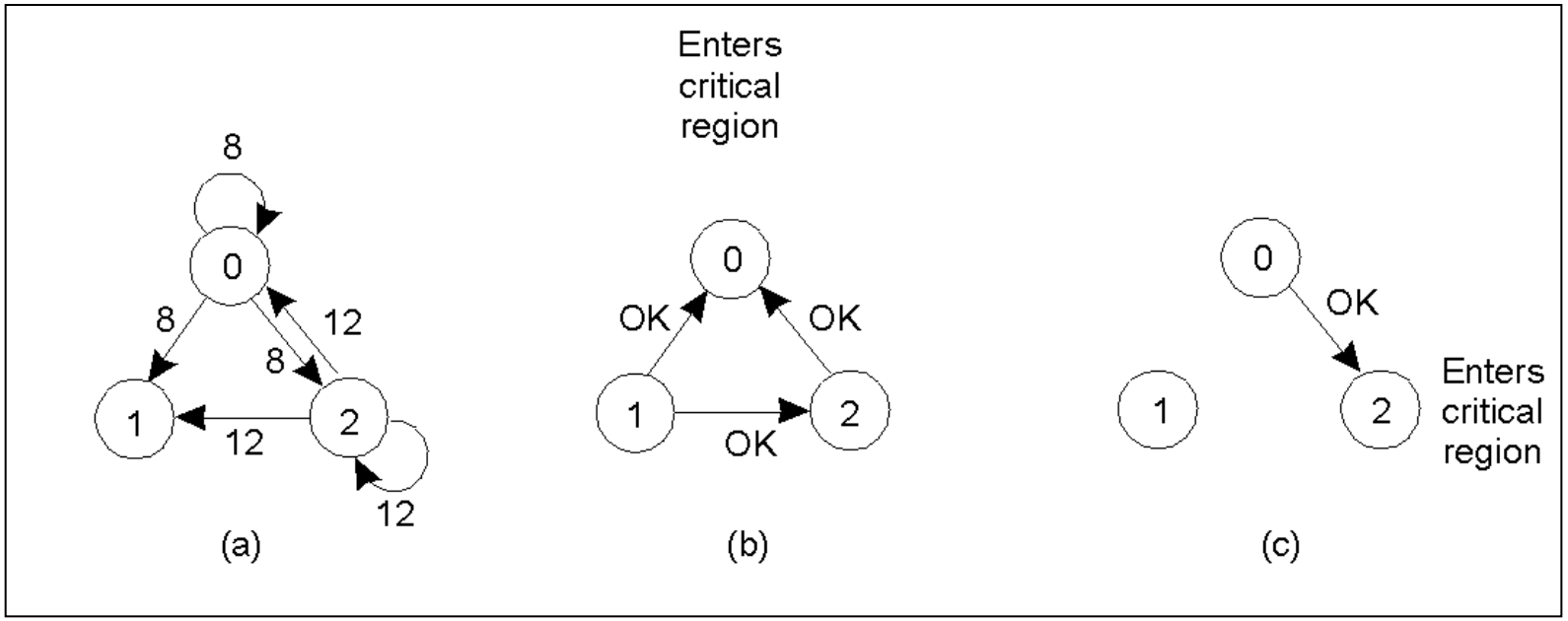
< *cr_name*, *proc_id*, *time* >

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 messaggio;
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)

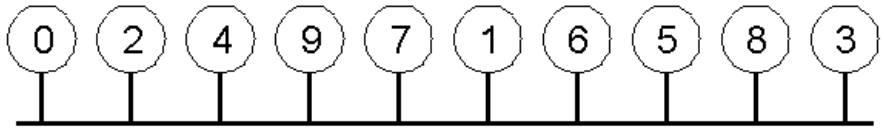


a) Due processi (0 e 2) vogliono entrare nella stessa regione critica nello stesso istante

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

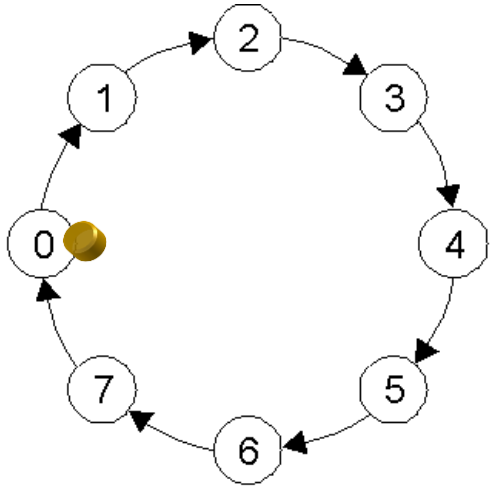
c) Quando il processo 0 ha finito, invia un OK, quindi il processo 2 può accedere alla regione critica.

Algoritmo Token Ring



(a)

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



(b)

(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

Un confronto dei tre algoritmi di mutua esclusione.

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)

- In un sistema distribuito un processo non deve terminare senza segnalare la sua terminazione.
- La terminazione di un processo non segnalata può bloccare la terminazione di altri processi.
- La terminazione distribuita richiede un algoritmo/protocollo specifico che permetta di determinare (in maniera distribuita!) quando tutti i processi non sono più in esecuzione.
- Strutture di terminazione ad hoc (es: anello, albero, grafo).

Algoritmi di Elezione Distribuiti

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 concordano su un coordinatore.

Algoritmo Bully (1)

Un processo P gestisce una elezione come di seguito:

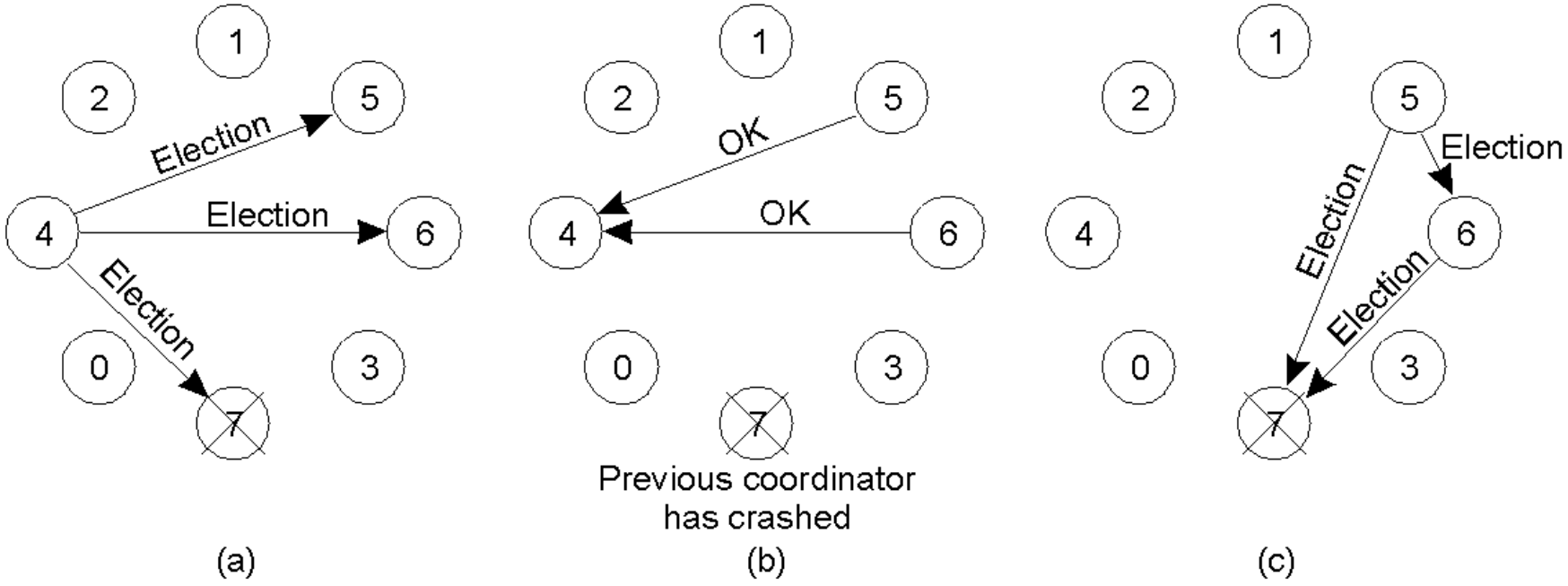
1. P_i invia un messaggio *ELECTION* a tutti i processi con ID maggiore del proprio: P_{i+1}, P_{i+2}, … P_N.

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

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

4. Il nuovo coordinatore informa tutti i processi.

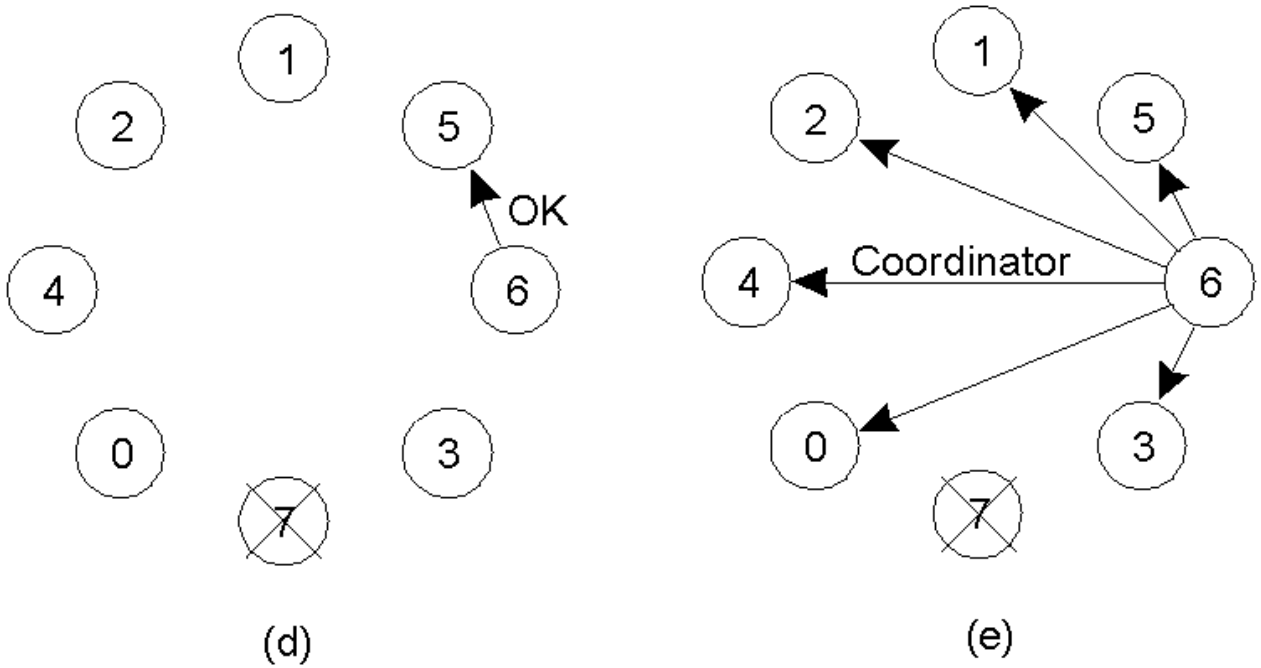
Algoritmo Bully (2)



L' algoritmo di elezione Bully

a) Il processo 4 inzia l' algoritmo di elezione (poichè il 7 non risponde),
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:

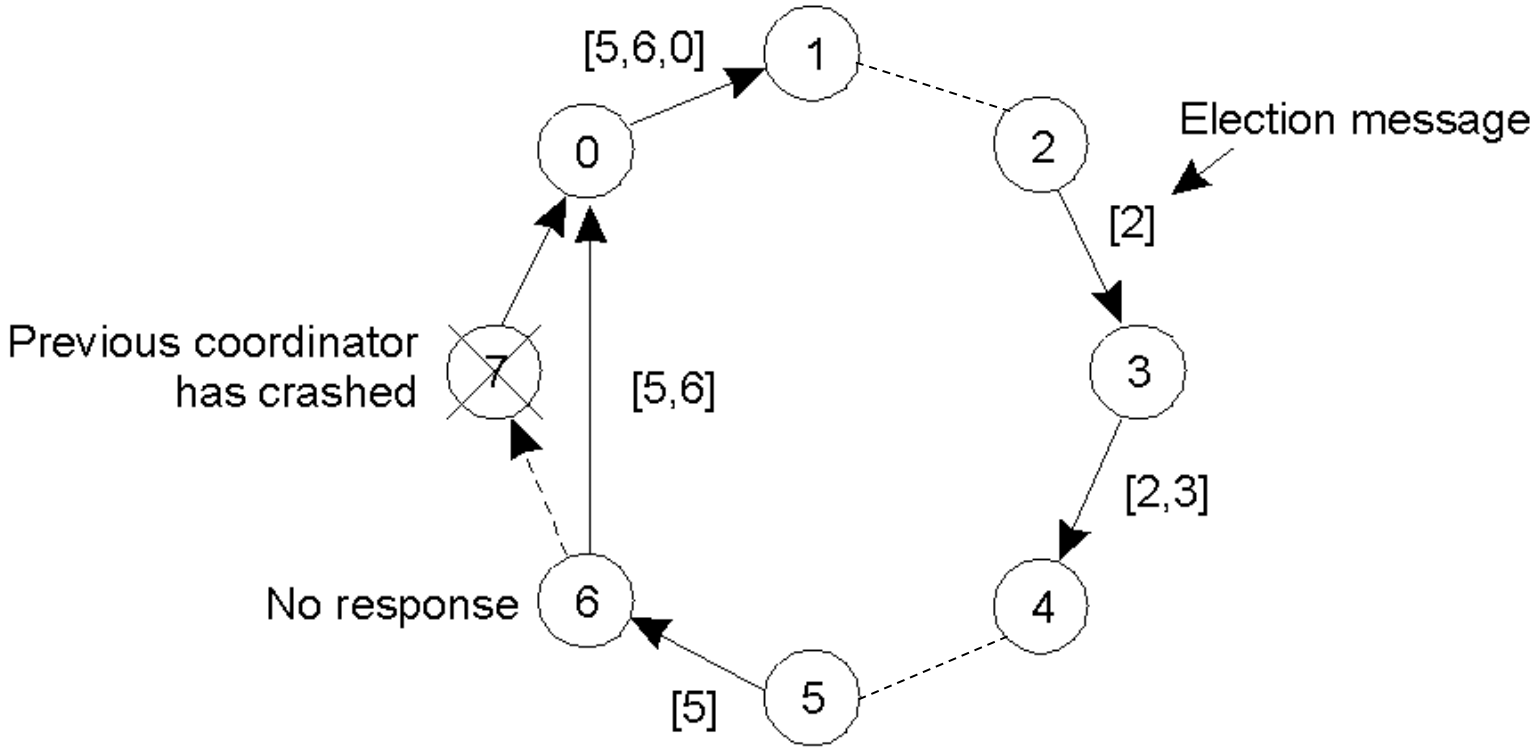
1. Ogni processo conosce chi è il suo successore.

2. L'algoritmo di elezione è iniziato da un processo P_i 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.