

# Sistemi Distribuiti

## Il Tempo Logico

Dott. Ing. Silvia Bonomi  
bonomi@dis.uniroma1.it

# Motivazioni

- Gli algoritmi per la sincronizzazione dei clock fisici si basano sulla stima dei tempi di trasmissione dei canali
- Nei sistemi reali non sempre i tempi di trasmissione dei canali sono predicibili e quindi viene a mancare l'accuratezza della sincronizzazione
- L'assenza di sincronizzazione implica l'impossibilità di ordinare gli eventi appartenenti a processi diversi



- **Osservazioni:**

- Due eventi occorsi sullo stesso processo  $p_i$  si sono verificati esattamente nell'ordine in cui  $p_i$  li ha osservati
- Quando un messaggio viene inviato da un processo  $p_i$  ad un processo  $p_j$ , l'evento di *send* precede l'evento di *receive*

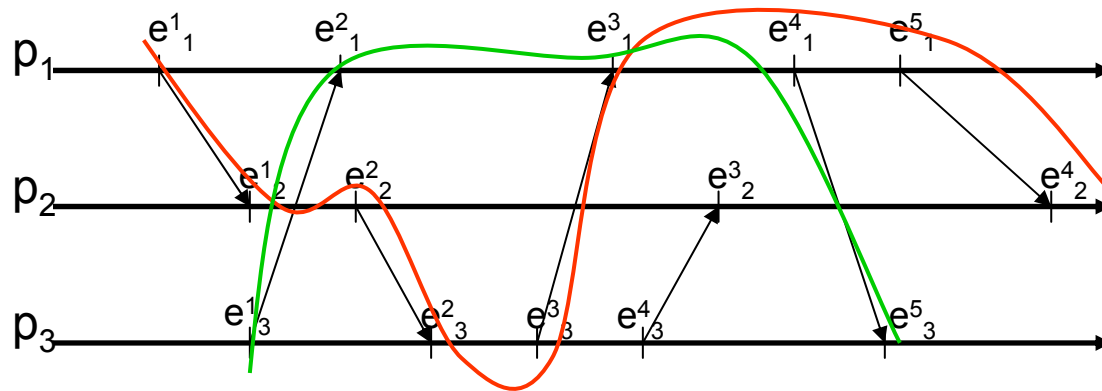
- Lamport introduce il concetto di relazione di *happened-before* (anche detta relazione di *ordinamento causale*)

- Indichiamo con  $\rightarrow_i$  la relazione di ordinamento su un processo  $p_i$
- Indichiamo con  $\rightarrow$  la relazione di *happened-before* tra due eventi qualsiasi

# Relazione Happened-before: Definizione

- Due eventi  $e$  ed  $e'$  sono in relazione di happened-before ( $e \rightarrow e'$ ) se:
  - $\exists p_i \mid e \rightarrow_i e'$
  - $\forall$  messaggio  $m$   $\text{send}(m) \rightarrow \text{receive}(m)$ 
    - Con  $\text{send}(m)$  identifichiamo l'evento di invio del messaggio  $m$
    - Con  $\text{receive}(m)$  identifichiamo l'evento di ricezione del messaggio  $m$
  - $\exists e, e', e'' \mid (e \rightarrow e'') \wedge (e'' \rightarrow e')$
- Applicando le tre regole è possibile costruire una sequenza di eventi  $e_1, e_2, \dots, e_n$  causalmente ordinati
- **OSSERVAZIONE:**
  - Non è detta che la sequenza  $e_1, e_2, \dots, e_n$  sia unica
  - Non è detto che comunque presa una coppia di eventi questa sia legata da una relazione happened-before; in questo caso si dice che gli eventi sono *concorrenti*

# Relazione happened-before: esempio



Indichiamo con  $e^j_i$  il  $j$ -esimo evento del processo  $i$

$$S_1 = \langle e^1_1, e^1_2, e^2_2, e^2_3, e^3_3, e^3_1, e^4_1, e^5_1, e^4_2 \rangle$$

$$S_2 = \langle e^1_3, e^2_1, e^3_1, e^4_1, e^5_3 \rangle$$

Osservazione:

Gli eventi  $e^1_3$  e  $e^1_2$  sono concorrenti

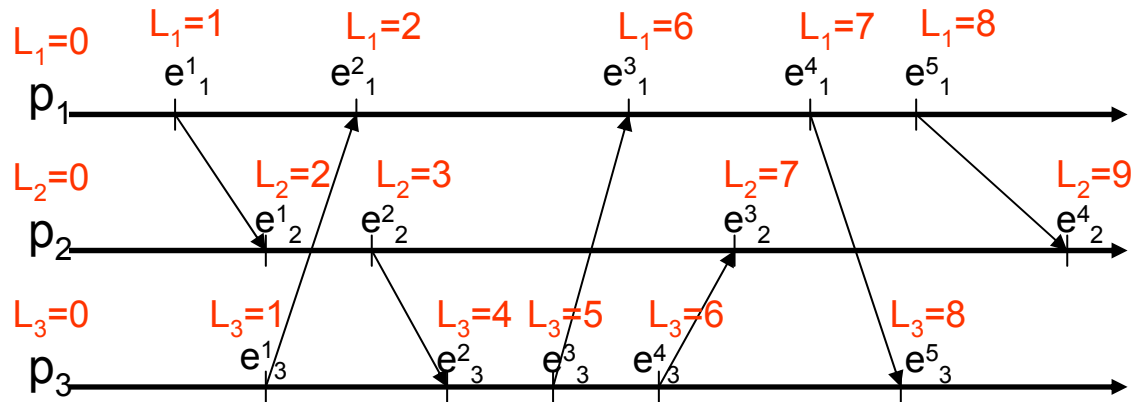
# Clock Logici

- Il Clock Logico, introdotto da Lamport, è un contatore software *monotonicamente crescente* il cui valore non ha alcuna relazione con il clock fisico
- Ogni processo  $p_i$  ha il proprio clock logico  $L_i$  usato per applicare i *timestamp* agli eventi
- Denotiamo con  $L_i(e)$  il timestamp, basato sul clock logico, applicato dal processo  $p_i$  all'evento  $e$ .
- **Proprietà:**
  - Se  $e \rightarrow e'$  allora  $L(e) < L(e')$
- **Osservazione:**
  - L'ordinamento che si ottiene è un ordinamento parziale quindi non è detto che guardando i timestamp di due eventi riesca a capire in che relazione sono

# Clock Logico Scalare: implementazione

- Ogni processo  $p_i$  inizializza il proprio clock logico  $L_i$  a 0 ( $\forall i = 1 \dots N$ )
- $L_i$  è incrementato di 1 che il processo  $p_i$  generi l'evento (sia che l'evento sia una *send* che sia una *receive*)
  - $L_i = L_i + 1$
- Quando  $p_i$  invia un messaggio  $m$ 
  - Genera l'evento di *send*( $m$ )
  - Incrementa il valore di  $L_i$  (conseguenza della regola precedente)
  - Allega al messaggio  $m$  il timestamp  $t=L_i$
- Quando  $p_i$  riceve un messaggio  $m$  con timestamp  $t$ 
  - Aggiorna il proprio clock logico  $L_i = \max(t, L_i)$
  - Genera l'evento di *receive*( $m$ )
  - Incrementa il valore di  $L_i$  (conseguenza della prima regola)

# Clock Logico Scalare: esempio



- Indichiamo con  $e_j^i$  il  $j$ -esimo evento del processo  $p_i$
- Indichiamo con  $L_i$  il clock logico del processo  $p_i$
- Ossevazione:
  - $e_1^1 \rightarrow e_2^1$  e i relativi timestamp riflettono questa proprietà
  - $e_1^1 \parallel e_1^3$  e i relativi timestamp sono uguali
  - $e_2^1 \parallel e_1^3$  e i relativi timestamp sono diversi

# Problema

- Il clock logico scalare ha la seguente proprietà
  - SE  $e \rightarrow e'$  allora  $L(e) < L(e')$
- Ma non è possibile assicurare che
  - SE  $L(e) < L(e')$  allora  $e \rightarrow e'$
- **Conseguenza:**
  - Non è possibile stabilire, solo guardando i clock logici scalari, se due eventi sono concorrenti o meno
- Mattern [1989] e Fridge [1991] propongono di ovviare al problema considerando anche l'identificativo del processo in cui l'evento occorre
  - Introduzione dei ***Vector Clock***

# Vector Clock : definizione

- Un Vector Clock per un sistema di  $N$  processi è dato da un array di  $N$  interi
- Ciascun processo  $p_i$  detiene il proprio Vector Clock  $V_i$
- Ciascun processo usa il suo Vector Clock per assegnare i timestamp agli eventi
- Analogamente al caso di Lamport, il Vector Clock viene allegato al messaggio  $m$  ed il timestamp diviene vettoriale
- Con i Vector Clock si catturano a pieno le caratteristiche della relazione happened-before

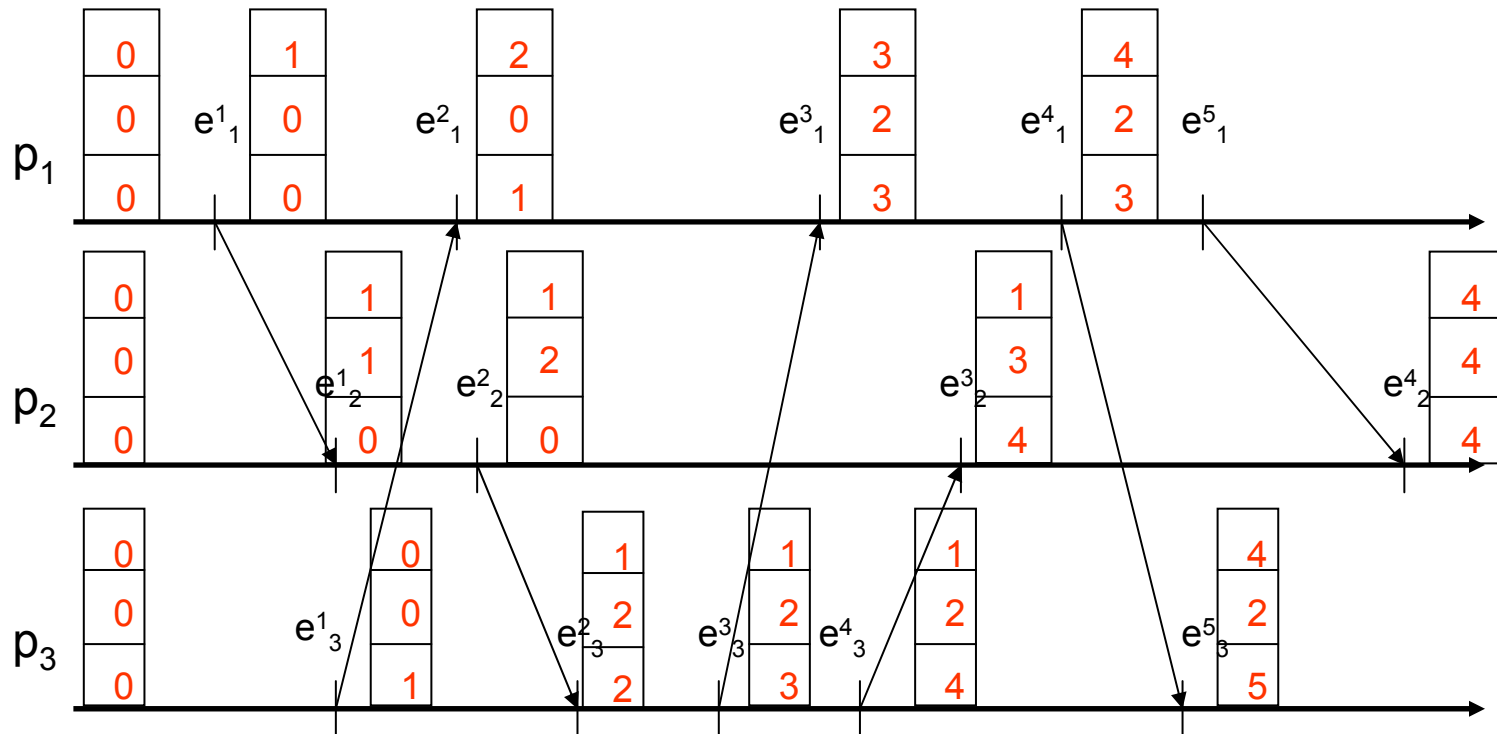
# Vector Clock : implementazione

- Ogni processo  $p_i$  inizializza il proprio Vector Clock  $V_i$ 
  - $V_i[j]=0 \quad \forall j = 1 \dots N$
- Prima di generare un evento  $p_i$  incrementa la sua componente del Vector Clock
  - $V_i[i]=V_i[i]+1$
- Quando  $p_i$  invia un messaggio  $m$ 
  - Genera l'evento di *send*( $m$ )
  - Incrementa il valore di  $V_i$  (conseguenza della regola precedente)
  - Allega al messaggio  $m$  il timestamp  $t=V_i$
- Quando  $p_i$  riceve un messaggio  $m$  con timestamp  $t$ 
  - Aggiorna il proprio clock logico  $V_i[j]=\max(t[j], V_i[j]) \quad \forall j = 1 \dots N$
  - Genera l'evento di *receive*( $m$ )
  - Incrementa il valore di  $V_i$  (conseguenza della prima regola)

# Vector Clock: proprietà

- Dato un Vector Clock  $V_i$ 
  - $V_i[i]$  indica il numero di eventi generati da  $p_i$
  - $V_i[j]$  con  $i \neq j$  indica il numero di eventi occorsi a  $p_j$  di cui  $p_i$  potrebbe avere conoscenza
- $V = V'$  se e solo se
  - $V[j] = V'[j] \quad \forall j = 1 \dots N$
- $V \leq V'$  se e solo se
  - $V[j] \leq V'[j] \quad \forall j = 1 \dots N$
- $V < V'$  e quindi l'evento associato a  $V$  precede quello associato a  $V'$  se e solo se
  - $V \leq V' \wedge V \neq V'$ 
    - $\forall i = 1 \dots N \quad V'[i] \geq V[i]$
    - $\exists i \in \{1 \dots N\} \mid V'[i] > V[i]$

# Vector Clock: esempio



# Comparazione di Vector Clock

1	1
0	1
0	0

V                  V'

In base alle proprietà viste in precedenza abbiamo che  $V(e) < V'(e')$  quindi  $e \rightarrow e'$

1	0
0	0
0	1

V                  V'

In base alle proprietà viste in precedenza abbiamo che  $V(e) \neq V'(e')$  quindi  $e \parallel e'$

Solo guardando i timestamp riusciamo a capire se due eventi sono concorrenti o se sono in relazione happened-before

# Il concetto di Knowledge

- Assumiamo che la “knowledge” sia una collezione di fatti.
- Con un’appropriata codifica una quantità finita di “knowledge” può essere rappresentata da un intero.
- Assumiamo che:
  - la knowledge aumenti con il tempo per ogni processo (un processo “non dimentica mai”).
  - l’unico modo in cui la knowledge può essere comunicata ad altri processi è attraverso messaggi.
- Se ogni processo include tutto ciò che sa in un messaggio e il ricevente aggiorna la propria knowledge alla ricezione dello stesso, allora il ricevente avrà più knowledge sia rispetto al mittente che rispetto a se stesso prima di ricevere il messaggio.

# Diffusione della conoscenza

- Maggiore è la dimensione dei clock maggiore è la conoscenza (“knowledge”) che possono trasmettere ai processi.
- Un sistema di clock logici mi codifica la conoscenza di ogni singolo processo
- Un sistema di vector clock permette ad un processo di avere visione della sua conoscenza e della conoscenza degli altri processi nel sistema. Questa knowledge” deve essere codificata con un vettore di dimensione pari al numero dei processi.
- Accresciamo la conoscenza dei processi utilizzando Matrix Clock

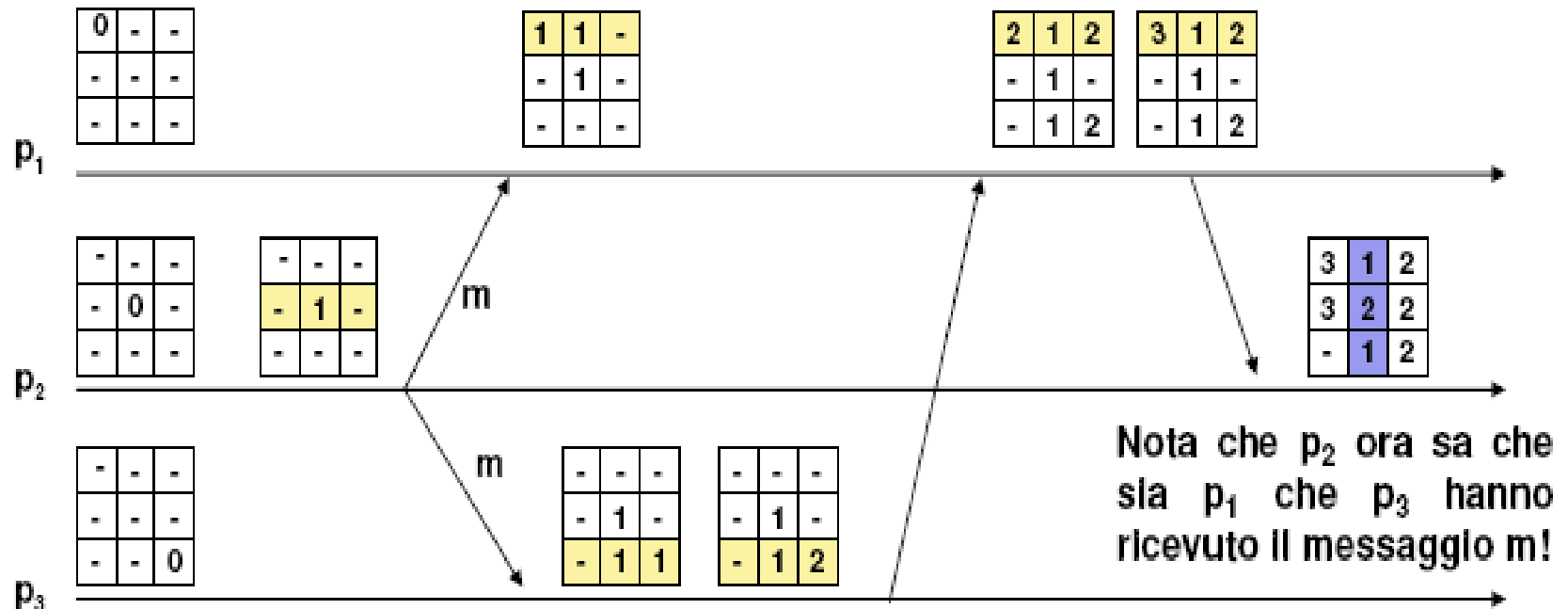
# I Matrix Clock

- Un Matrix Clock è un vettore di dimensione  $N$  di Vector Clock
- L'elemento  $(i,j)$  della matrice  $M_k$  rappresenta ciò che  $p_k$  sa a proposito di ciò che  $p_i$  sa di  $p_j$
- Se  $P_i$  nel suo matrix clock ha tutta la colonna relativa a se stesso (colonna  $i$ ) maggiore di un certo  $k$ , allora può concludere che tutti sanno che  $P_i$  è arrivato almeno all'evento  $k$ .
- Questo livello di conoscenza dell'informazione all'interno del sistema è utile quando si vuole controllare che alcune informazioni siano ricevute da tutti i processi del sistema (es. canali non affidabili in cui è possibile la perdita di pacchetti)
- il mittente  $p_i$  può rilevare se l'informazione è stata ricevuta da tutti i processi (*informazione stabile*) ispezionando la colonna  $i$  del matrix clock. In caso affermativo può scartare la suddetta informazione (sa che non la deve re-inviare più).

# Matrix Clock: implementazione

- Ogni processo  $p_k$  ha il suo Matrix Clock  $M_k$  di dimensioni  $N \times N$
- Ogni processo  $p_k$  inizializza il proprio Matrix Clock a “-” tranne la posizione  $M_k(k,k) = 0$
- Prima di generare un evento  $p_i$  incrementa la sua componente del Vector Clock
  - $M_k(k,k) = M_k(k,k) + 1$
- Quando  $p_k$  invia un messaggio  $m$ 
  - Genera l'evento di *send*( $m$ )
  - Incrementa il valore di  $M_k$  (conseguenza della regola precedente)
  - Allega al messaggio  $m$  il timestamp  $t=M_k$
- Quando  $p_i$  riceve un messaggio  $m$  con timestamp  $t$  da  $p_j$ 
  - $\forall x \in [1, \dots, n]$  e  $x \neq i$   $M_i[x, *] = \max [M_i(x, *), T(x, *)]$  (aggiorna ciò che  $P_i$  conosce della conoscenza degli processi)
  - $\forall y \in [1, \dots, n]$   $M_i[i, y] = \max [M_i(i, y), T(j, y)]$  (aggiorna la conoscenza di  $P_i$ )
  - Genera l'evento di *receive*( $m$ )
  - Incrementa il valore di  $M_i$  (conseguenza della prima regola)

# Matrix Clock: esempio





# Tempo Logico e Algoritmi Distribuiti

Applicazione di Vector Clock e Matrix  
Clock

# Il tempo logico negli algoritmi distribuiti

- Abbiamo visto tre meccanismi per rappresentare il tempo logico
  - Clock salare
  - Vector Clock
  - Matrix Clock
- A seconda della specifica del problema è possibile utilizzare un diverso meccanismo per assegnare i timestamp agli eventi
  - Timestamp scalare → Mutua Esclusione di Lamport
  - Timestamp vettoriale → Causal Broadcast
  - Timestamp matriciale → Rilevazione della stabilità dei messaggi

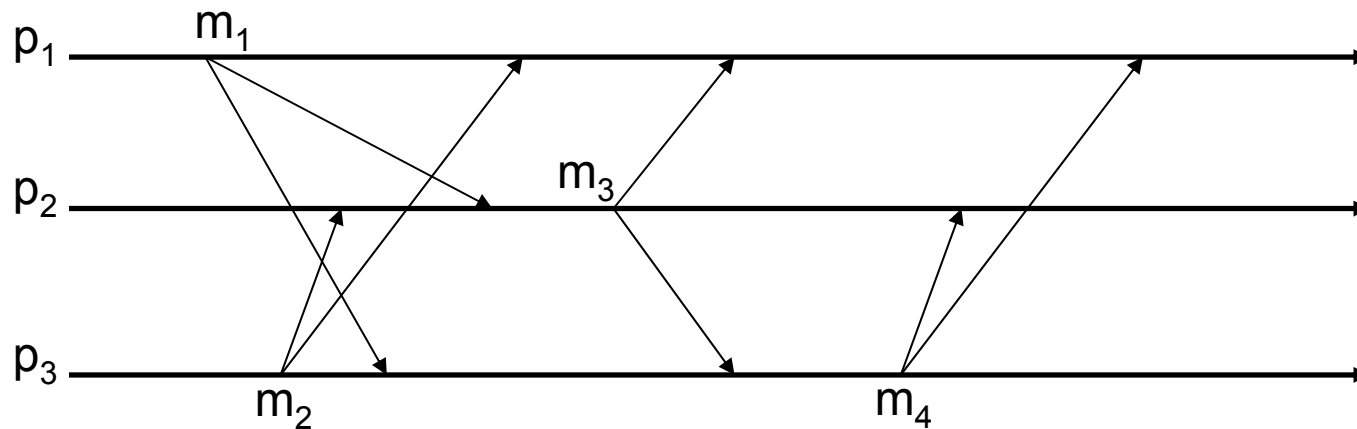
# La precedenza causale tra eventi di broadcast

- Un evento di broadcast è dato dall'invio di un messaggio  $m$  da parte di un processo  $p_i$  a tutti i processi del sistema
- Un evento  $e = \text{broadcast}(m)$  precede causalmente un evento  $e' = \text{broadcast}(m')$  se una delle seguenti condizioni è verificata
  - $e$  ed  $e'$  sono eventi generati dallo stesso processo e  $\text{broadcast}(m)$  avviene prima di  $\text{broadcast}(m')$
  - $e$  ed  $e'$  sono generati da processi diversi e l'evento  $e'$  è generato dopo aver consegnato  $m$
  - $\exists m''$  tale che  $\text{broadcast}(m) \rightarrow \text{broadcast}(m'') \wedge \text{broadcast}(m'') \rightarrow \text{broadcast}(m')$

# Causal Broadcast

- Il concetto di causal broadcast viene introdotto da Birman e Joseph con lo scopo di ridurre l'asincronia dei canali di comunicazione percepita dai processi
- **Specifica:**
  - Dati due messaggi di broadcast  $m$  ed  $m'$  tali che  $broadcast(m) \rightarrow broadcast(m')$  allora ogni processo deve consegnare  $m$  prima di  $m'$
  - Dati due messaggi di broadcast  $m$  ed  $m'$  tali che  $broadcast(m) \parallel broadcast(m')$  allora  $m$  ed  $m'$  possono essere consegnati in un certo ordine da alcuni processi e in un altro ordine da altri

# Causal Broadcast: esempio



- In questo scenario

- $m_1 \rightarrow m_3 \Rightarrow$  tutti i processi devono consegnare  $m_1$  prima di consegnare  $m_3$
- $m_1 \parallel m_2 \Rightarrow$   $m_1$  e  $m_2$  possono essere consegnati diversamente da tutti i processi
- $m_1 \rightarrow m_4 \Rightarrow$  tutti i processi devono consegnare  $m_1$  prima di consegnare  $m_4$

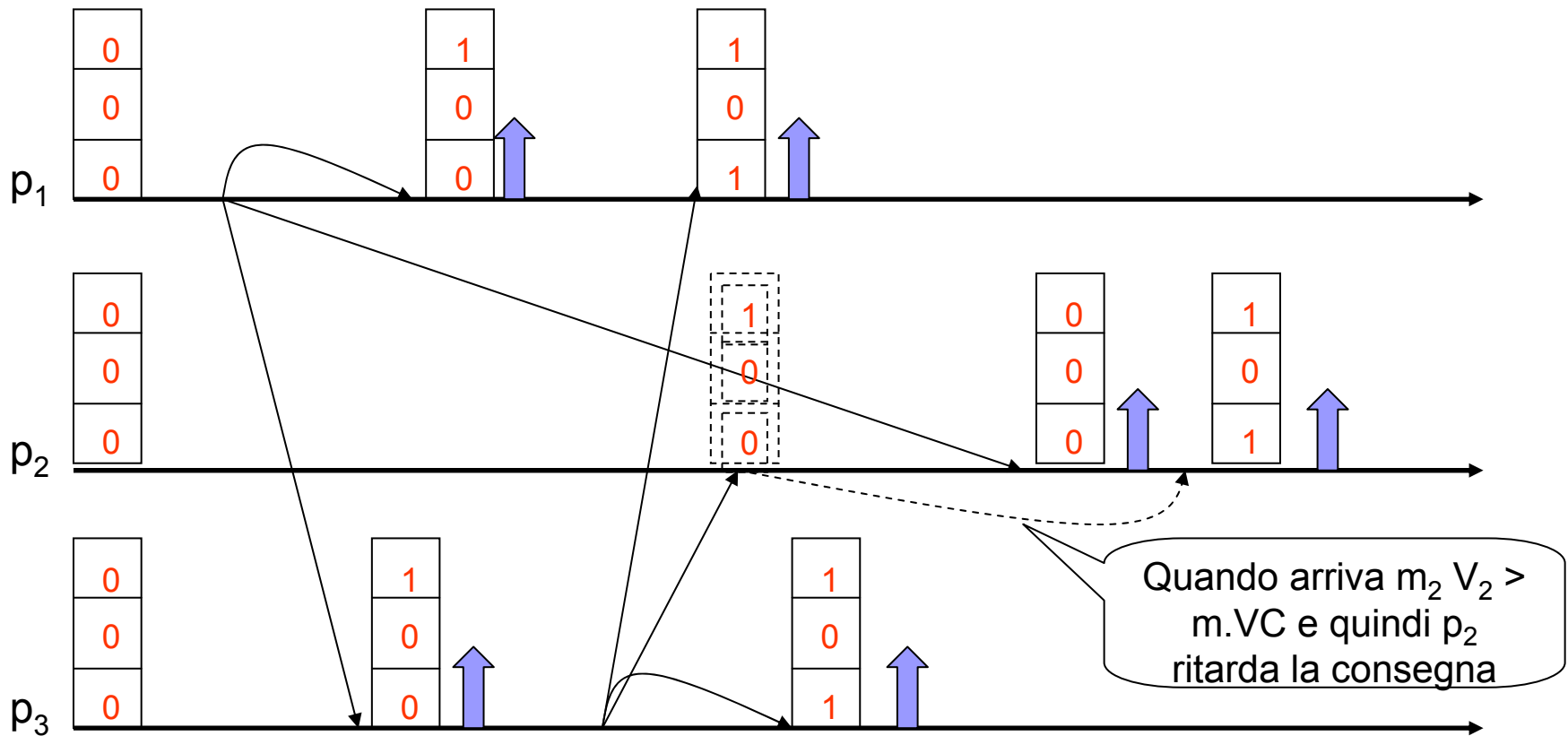
# Causal broadcast: implementazione

- Il modello di sistema è asincrono e non ci sono guasti
- **Idea:**
  - Un processo  $p_i$  ritarda la consegna di un messaggio  $m$  finchè tutti i messaggi che predono causalmente  $m$  non vengono consegnati da  $p_i$
- Ogni processo  $p_i$  mantiene un Vector clock  $V_i$  che tiene traccia della conoscenza che  $p_i$  ha del numero di messaggi inviati da ciascun processo
- $V_i [ j ]$  rappresenta il numero di messaggi, inviati da  $p_j$ , di cui  $p_i$  ha conoscenza e che sono stati consegnati da  $p_i$
- Ogni messaggio di broadcast  $m$  ha allegato un timestamp vettoriale  $m.VC$
- Quando  $p_i$  riceve un messaggio di broadcast  $m$ , ritarda la consegna di  $m$  finche tutti i messaggi del “passato causale” di  $m$  non sono stati consegnati
  - $\forall k \in \{1 \dots N\} \ m.VC[k] \leq V_i [ k ]$

# Causal Broadcast: pseudo-codice

- Ogni processo  $p_i$  implementa le seguenti regole per gestire il causal broadcast
- **Procedure broadcast (m)**
  - $m.VC = V_i$  // timestamp da allegare a m
  - for all  $j \in \{1 \dots N\}$ 
    - Send(m) to  $p_j$  // invio del messaggio di broadcast
  - and for
  - $V_i = V_i + 1$  //aggiornamento del clock locale
- **Upon receive m from  $p_j$** 
  - delay the delivery until  $\forall k \in \{1 \dots N\} m.VC[k] \leq V_i[k]$
  - if  $i \neq j$
  - then  $V_i[j] = V_i[j] + 1$  //aggiornamento del clock locale
  - deliver m to the upper layer //evento di deliver

# Causal Broadcast: esempio



# Causal Broadcast: Safety

## ■ Proprietà:

- Dati due messaggi di broadcast  $m$  ed  $m'$  tali che  $broadcast(m) \rightarrow broadcast(m')$  allora ogni processo deve consegnare  $m$  prima di  $m'$

## ■ Osservazione:

- se  $m$  è il  $k$ -esimo messaggio inviato da  $p_i$  allora  $m.Vc[i] = k-1$

- Per verificare che la proprietà di safety sia soddisfatta dall'algoritmo proposto facciamo una prova per induzione sulla relazione di precedenza causale da messaggi di broadcast

## ■ Definizione:

- Dati due eventi di broadcast  $b$  ed  $b'$  con  $b \rightarrow b'$  diciamo che tali eventi hanno distanza causale  $k$  se  $\exists$  una sequenza di  $k$  eventi di broadcast  $b_1 \dots b_k$  tali che
  - $\forall i \in \{1 \dots k\} b_i \rightarrow b_{i+1} \wedge (\neg \exists) m^* | b_i \rightarrow m^* \rightarrow b_{i+1}$
  - $b \rightarrow b_1$
  - $b_k \rightarrow b'$

# Dimostrazione – Passo Base (K=0)

- Siano  $m, m'$  due messaggi tali che
  - $\text{broadcast}(m) \rightarrow \text{broadcast}(m')$
  - non esiste alcun evento  $\text{broadcast}(m'')$  t.c.  
 $\text{broadcast}(m) \rightarrow \text{broadcast}(m'') \wedge \text{broadcast}(m'') \rightarrow \text{broadcast}(m')$ .
- Si possono verificare due casi
  - $m$  e  $m'$  sono generati dallo stesso processo
  - $m$  e  $m'$  sono generato da processi diversi

# Caso 1 – broadcast generati dallo stesso processo

1. Sia  $p_j$  il processo ricevente.
2. Per la linea 3 della procedura di broadcast
  1.  $m'.VC[i] := m.VC[i] + 1$ .  
Se  $m$  è il  $h$ -esimo messaggio inviato da  $p_i$ ,  $m.VC[i] = h - 1$  e  $m'.VC[i] = h$ .
3. Un processo  $p_j$  che riceve  $m'$  verifica la seguente condizione di delivery:
  1.  $\forall x \in \{1, \dots, n\} m'.VC[x] \leq V_j[x]$  e in particolare  $m'.VC[i] \leq V_j[i]$
4.  $V_j[x]$  è uguale ad  $h$  se e solo se l'  $h$ -esimo messaggio inviato dal processo  $p_x$  è stato consegnato da  $p_i$ . (linea 3 thread di ricezione).
5. Dai punti 2,3,4 deriva che  $m'$  può essere consegnato solo dopo che è stato consegnato  $m$ .

# Caso 1 – broadcast generati da processi diverso

- $m$  ed  $m'$  sono stati inviati da due processi diversi, rispettivamente  $p_i$  e  $p_j$ . Sia  $p_k$  il processo ricevente.
- Poiché  $\text{broadcast}(m) \rightarrow \text{broadcast}(m')$ ,  $m'$  è stato mandato in broadcast dal processo  $p_j$  dopo aver consegnato  $m$ .  
Senza perdita di generalità sia  $m.VC[i]=h-1$ 
  - per la linea 3 del thread di ricezione e per l'assunzione di  $k=0$  si ha  $m'.VC[i]=h$ .
- Un processo  $p_k$  che riceve  $m'$  verifica la seguente condizione di delivery:
  - $\forall x \in \{1, \dots, n\} m'.VC[x] \leq V_k[x]$  e in particolare  $m'.VC[i] \leq V_k[i]$
- In particolare  $m'.VC[i] \leq V_k[i]$ , cioè  $V_k[i] \geq h$
- $V_k[i]$  è uguale ad  $h$  se e solo se l' $h$ -esimo messaggio inviato dal processo  $p_i$  è stato consegnato dal processo  $p_k$ . (linea 3 thread di ricezione).
- Dai punti 2,3,4 deriva che  $p_k$  può consegnare il messaggio  $m'$  solo dopo aver consegnato il messaggio  $m$ .

# Dimostrazione – Passo Induttivo ( $k > 0$ )

- $\exists$  una sequenza di  $k$  eventi di broadcast  $b_1, b_2 \dots b_k$  tale che
  - $b \rightarrow b_1 \rightarrow b_2 \rightarrow \dots \rightarrow b_k \rightarrow b'$
- Ipotesi induttiva:  $m$  è consegnato prima di  $m_k$
- Devo dimostrare che  $m_k$  è consegnato prima di  $m'$ .
  - dimostrato dal passo base.
- Segue che  $m$  è consegnato prima di  $m'$ .

# Causal Broadcast: Liveness

- **Proprietà:**
  - ogni messaggio viene prima o poi consegnato.
- **Garantita dalla seguente coppia di assunzioni:**
  - il numero di eventi di broadcast di messaggi che precedono causalmente un certo evento di broadcast è finito
  - i canali sono affidabili.

# Stabilità dei messaggi

- Considera applicazioni in cui i processi fanno broadcast di operazioni a tutti gli altri processi, e dove ogni processo deve alla fine ricevere lo stesso insieme di operazioni che i processi corretti inviano. Questo problema astrae la nozione di reliable broadcast in cui le operazioni corrispondono a messaggi.
- Modello di sistema:
  - crash and network partition (send/receive omission)

- Quindi per garantire reliable broadcast in questo sistema ogni processo deve bufferizzare una copia di ogni messaggio che manda o che riceve. In caso di necessità, es. guasto di un processo  $p$  e mancato recapito da parte di alcuni processi del messaggio  $m$  inviato da  $p$ , la copia del messaggio  $m$  viene inoltrata da i processi vivi a quelli che non hanno ricevuto  $m$
- Rapida crescita del buffer! Rischio di overflow
- **Osservazione:**
  - Un messaggio consegnato da tutti i processi non è più necessario. Tale messaggio è chiamato messaggio stabile.
- I messaggi stabili possono essere eliminati dai buffer.

# Protocollo per la rilevazione della stabilità dei messaggi

- Un protocollo per la rilevazione della message stability gestisce i buffer dei processi.
- Implementazione basata su matrix clock:
- Modello di sistema: (per semplicità)
  - canali FIFO
  - no guasti
- Gli eventi di broadcast sono gli eventi rilevanti della computazione. Ogni processo  $P_i$  mantiene un matrix clock  $MC_i$ .  $MC_i[k]$  indica qual'è la conoscenza di  $P_i$  circa i messaggi consegnati da  $P_k$ .
- In particolare:
  - $M_i[k][l]$  con  $i, k, l$  rappresenta la conoscenza di  $P_i$  del numero di messaggi inviati da  $P_l$  che  $P_k$  ha consegnato;
  - $MC_i[i][i]$  rappresenta il numero di sequenza del prossimo messaggio inviato da  $P_i$ .
- Quindi il valore minimo sulla colonna  $j$  di  $MC_i$  —cioè, rappresenta la conoscenza di  $P_i$  a proposito del numero di sequenza dell'ultimo messaggio stabile che  $P_j$  ha inviato.

- Per propagare stability information, quando  $P_i$  invia un messaggio  $m$  inserisce in  $m$ :
  - l'identità del mittente ( $m.sender$ )
  - un timestamp  $m.VC$ :  $m.VC$  corrisponde al vettore  $MC_i[i][*]$  e indica quanti messaggi  $P_i$  ha consegnato relativamente ad ogni processo mittente.
- Due operazioni aggiornano il buffer locale (bufferi):
  - $deposit(m)$ , inserisce un messaggio  $m$  nel buffer
  - $discard(m)$ , rimuove  $m$  dal buffer
- Un processo inserisce un messaggio nel buffer immediatamente dopo la sua ricezione e lo elimina dal buffer appena il messaggio diventa stabile
- predicato di stabilità per il messaggio  $m$ 
  - $m.VC[m.sender]$  rappresenta il numero di sequenza di  $m$

# Stabilità dei messaggi: Pseudocodice

```
procedure rel-broadcast( $m$ ) % issued by  $P_i$  %  
   $m.VC := MC_i[i][*]$ ; % construct the timestamp of  $m$  %  
   $m.sender := i$ ;  
   $\forall x \in \{1, \dots, n\}$  do send( $m$ ) to  $P_x$  enddo % broadcast event %  
   $MC_i[i][i] := MC_i[i][i] + 1$  % one more broadcast by  $P_i$  %  
  
when  $P_i$  receives  $m$  from  $P_j$  %  $m$  piggybacks its vector timestamp  $m.VC$  %  
  deposit( $m$ ); % add  $m$  to the local buffer %  
   $MC_i[j][*] := m.VC$ ; % update of  $P_i$ 's view of  $P_j$ 's vector %  
  if  $i \neq j$  then  $MC_i[i][j] := MC_i[i][j] + 1$ ; % one more message delivered from  $P_j$  %  
  deliver  $m$  to the upper layer % produce the delivery event %  
  
when  $(\exists m \in buffer_i : m.VC[m.sender] \leq \min_{1 \leq x \leq n} (MC_i[x][m.sender]))$   
  discard( $m$ ) % suppress  $m$  from the local buffer %
```

# Stabilità dei messaggi: Esempio

