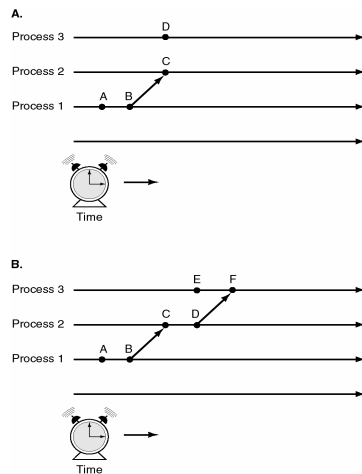


Nozione di Tempo Logico



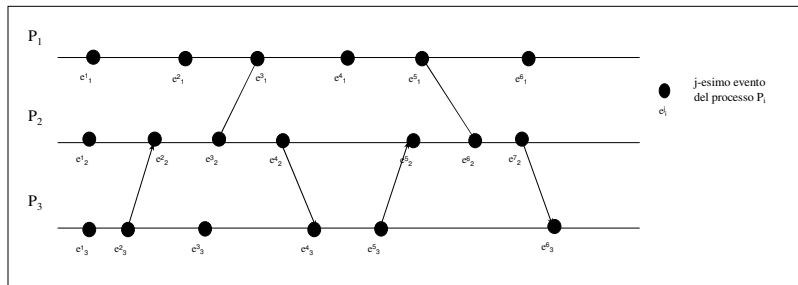
- Basato sulle seguenti ovvie assunzioni:
 - Due eventi nello stesso processo sono “naturalmente” ordinati
 - Una trasmissione precede sempre una ricezione
 - Gli eventi sono così ordinati secondo la nozione di causa-effetto (precedenza causale o *happened before*)

Happened-before

- Dati due eventi e ed e' allora e precede e' , indicandolo con $e \rightarrow e'$ se:
 1. gli eventi e ed e' appartengono allo stesso processo ed e accade prima di e' ;
 2. gli eventi e ed e' appartengono invece a processi distinti, e è l'evento di invio di un messaggio ed e' l'evento di ricezione di tale messaggio;
 3. se esiste un evento e'' t.c. $e \rightarrow e''$ e $e'' \rightarrow e'$
- Dati due eventi e ed e' se $\neg(e \rightarrow e')$ ed $\neg(e' \rightarrow e)$, i due eventi sono detti *concorrenti*: $e \parallel e'$

Happened-Before/esempio

- Dato un diagramma spazio-tempo allora $e \rightarrow e'$ se è possibile tracciare un percorso da e ad e' , procedendo da sinistra verso destra, altrimenti sono concorrenti
- Nell' esempio: $e_3^2 \rightarrow e_2^2$, $e_2^3 \rightarrow e_1^3$ e quindi $e_3^2 \rightarrow e_1^3$, mentre gli eventi e_2^1 e e_3^3 sono concorrenti.



Clock Logic

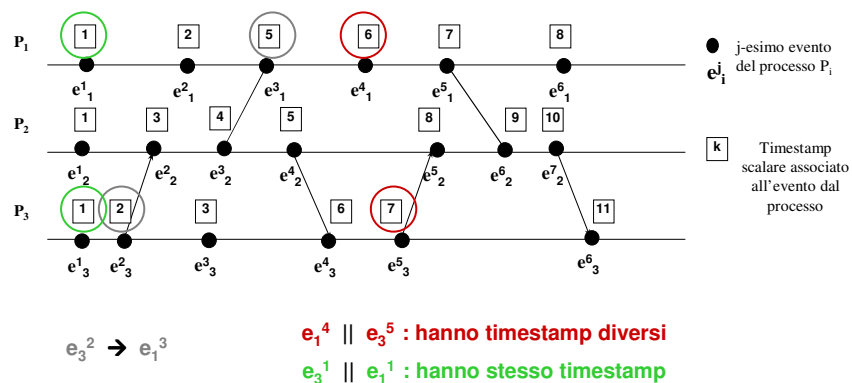
- L'idea è di ordinare gli eventi del sistema assegnando un numero naturale ad ogni evento (*timestamping scalare*)
- L'ordinamento è basato sulla relazione "happened before \rightarrow "
- Ad ogni evento e del sistema viene associato un timestamp $C(e)$, tale che:
se $e \rightarrow e'$ allora $C(e) < C(e')$

Timestamping scalare\implementazione

P_i mantiene un contatore C_i inizializzato a 0 e segue le seguenti regole di aggiornamento:

1. quando P_i processa un evento, prima incrementa il contatore C_i di una unità ($C_i := C_i + 1$) e quindi associa un timestamp T_i all'evento il cui valore è pari al valore corrente di C_i ;
2. quando P_i invia un messaggio, esegue l'evento di trasmissione e allega al messaggio il timestamp T_i associato a tale evento ricavato dalla regola 1;
3. quando a P_i arriva un messaggio m con timestamp T , esso pone $C_i = \max(C_i, T)$ e quindi esegue l'evento di ricezione del messaggio (regola 1).

Timestamping scalare\esempio



Vector Clock

- Clock logici: non catturano completamente la relazione happened-before. Infatti pur soddisfacendo la seguente proprietà: se $e \rightarrow e'$ allora $C(e) < C(e')$, non soddisfano il viceversa: $C(e) < C(e')$ non implica $e \rightarrow e'$
- In sostanza i clock logici non permettono di stabilire se due eventi sono concorrenti
- Mattern nel 1988 ha introdotto la nozione di *vector clock*, che invece caratterizza completamente la relazione di causalità.

Vector Clock

- Ad ogni evento e viene assegnato un vettore $V(e)$ di dimensione pari al numero dei processi con la seguente proprietà:

$$e \rightarrow e' \text{ se e solo se } V(e) < V(e')$$

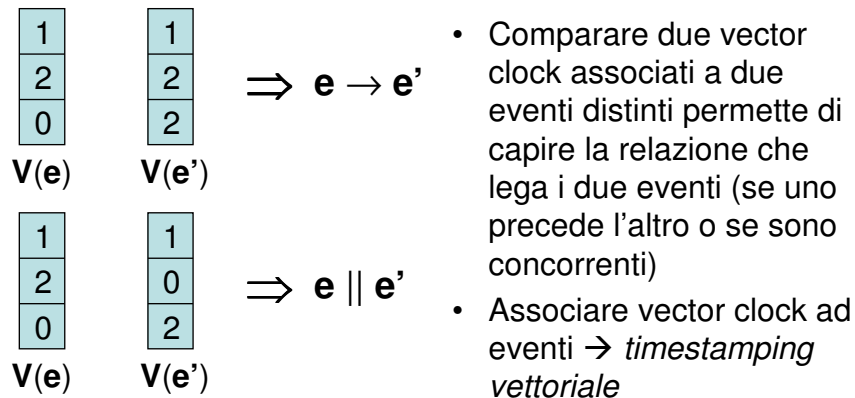
- Che significato ha il comparatore di minoranza tra vettori?

$V(e) < V(e')$ se e solo se

$$\forall x \in [1, \dots, n]: V(e')[x] \geq V(e)[x] \wedge$$

$$\exists x \in [1, \dots, n]: V(e')[x] > V(e)[x]$$

Vector Clock

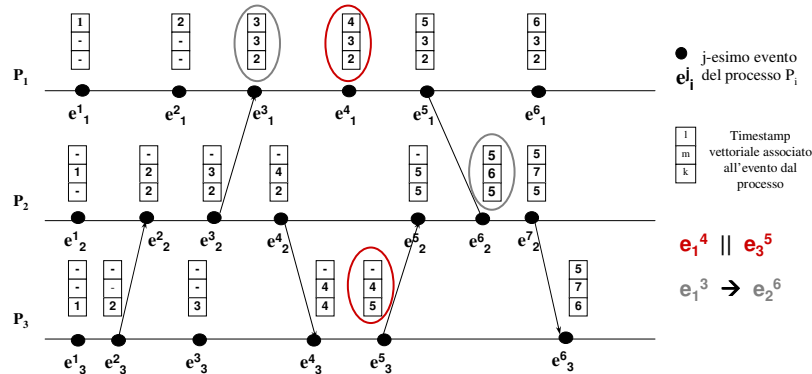


Timestamping vettoriale\implementazione

Un sistema di vector clock è formato da n vettori di interi V ad n componenti, uno per ogni processo. La componente $V_i[x]$ indica il numero di eventi del processo P_x osservati dal processo P_i . In particolare, ogni processo $P_i \in [1, \dots, n]$ gestisce un vettore di interi $V_i[1 \dots n]$ (inizializzato a $[-, \dots, 0, \dots, -]$) in base alle seguenti regole:

1. quando P_i processa un evento, incrementa $V_i[i]$ di una unità e poi associa un timestamp T all'evento il cui valore è pari al valore corrente di V_i ;
2. quando P_i esegue un evento di trasmissione, allega al msg il timestamp di quell'evento ottenuto dalla regola 1;
3. quando arriva un msg a P_i da P_j con un timestamp T , P_i esegue la seguente operazione: $\forall x \in [1, \dots, n]: V_i[x] := \max(V_j[x], T[x])$, quindi esegue l'evento di ricezione (esegue la regola 1);

Timestamping vettoriale\esempio



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.
- La “knowledge” che un processo ha di se stesso è rappresentata da questo intero.
- Assumiamo che un processo “non dimentica mai”, cioè che la knowledge aumenti con il tempo per ogni processo. Inoltre l’unico modo in cui la knowledge può essere comunicata a differenti 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. Ma questo meccanismo è quello dei logical clock!
- Se un processo vuole sapere non solo ciò che lui sa ma anche cosa gli altri processi fanno, questa “knowledge” deve essere codificata con un vettore di dimensione pari al numero dei processi. Vector clock!
- E’ naturale chiedersi se clock con dimensione superiore possano fornire ai processi più “knowledge”. La risposta è ovviamente **si**.

Matrix clock

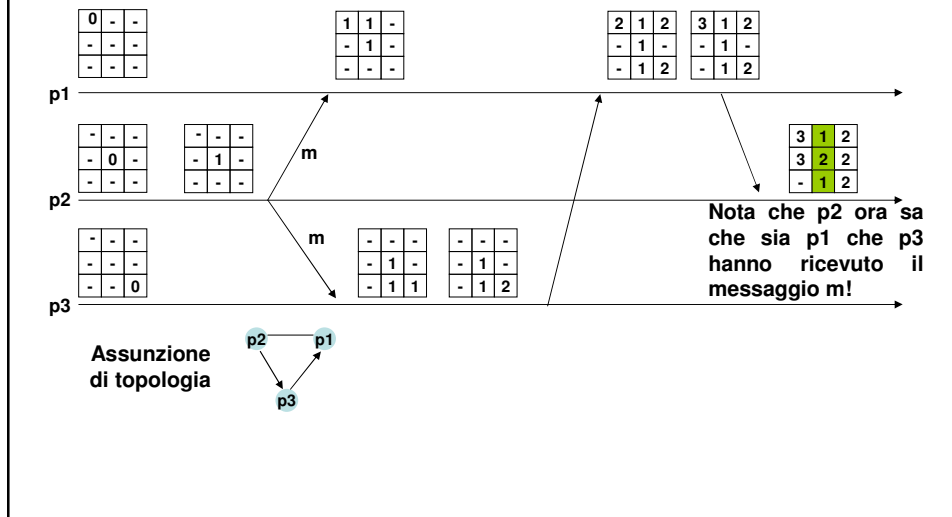
- Matrix clock, ossia un vettore di dimensione n di vector clock. Codifica un livello superiore di knowledge rispetto a un vector clock.
- L'idea è che l'elemento della matrice (i,j) rappresenta ciò che il processo p_i sa a proposito di ciò che il processo p_j sa a proposito del processo p_j .
- Se p_i 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_j è arrivato almeno all'evento k .
- Questo meccanismo è utile quando si vuole assicurare che una certa informazione venga ricevuta da tutti i processi (in situazione di comunicazione "incerta", es. perdita di messaggi). In questo caso 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 reinviare più).

Timestamping con matrix clock\implementazione

Un sistema di matrix clock è formato da n matrici M di interi di dimensione $n \times n$. In particolare, ogni processo P_i gestisce una matrice $M_i[1 \dots n]$ (con tutte le componenti inizializzate a $-$, tranne $M_i[i,i]=0$) in base alle seguenti regole:

1. Quando P_i processa un evento, $M_i[i,i]$ si incrementa di una unità e quindi associa un timestamp T all'evento il cui valore è pari al valore corrente di M_i ;
2. Quando P_i esegue un evento di trasmissione di un messaggio, egli allega al messaggio il timestamp di quell'evento ottenuto dalla regola 1;
3. Quando arriva un messaggio a P_i da P_j con allegato un timestamp T , P_i esegue le seguenti operazioni:
 1. $\forall x \in [1, \dots, n]$ e $x \neq i$: $M_i[x,*] := \max(M_i[x,*], T[x,*])$
 2. $\forall y \in [1, \dots, n]$: $M_i[i,y] := \max(M_i[i,y], T[j,y])$
 3. esegue l'evento di ricezione (esegue la regola 1).

Timestamping con matrix clock\esempio



Tempo Logico e Algoritmi distribuiti

- Abbiamo visto tre meccanismi per ordinare eventi in un sistema distribuito.
- Questi meccanismi sono utili per sviluppare algoritmi distribuiti dato un certo problema.
- Es. l'algoritmo di Lamport per la mutua esclusione utilizza il timestamping scalare mentre l'algoritmo che implementa la comunicazione ordinata causale utilizza il timestamping vettoriale. La rilevazione della stabilità dei messaggi è un problema in cui viene utilizzato il matrix clock.

Causal Broadcast

- **Causal broadcast:** per ridurre l'asincronia dei canali di comunicazione percepita dai processi dell'applicazione.
- Garantisce che l'ordine in cui i processi consegnano i messaggi al livello applicativo non possano violare l'ordine indotto dalla happen-before dei corrispondenti eventi di broadcast.
- **Specifica:**
 - Se 2 messaggi di broadcast m e m' sono tali che $\text{broadcast}(m) \rightarrow \text{broadcast}(m')$, allora ogni processo deve consegnare m prima di m' .
 - Se i broadcast di m e m' sono concorrenti, allora i processi sono liberi di consegnare m e m' in qualsiasi ordine.

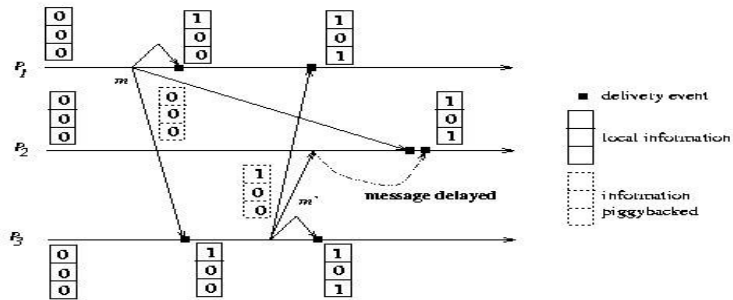
Implementazione basata su vector clock

- ✓ Modello di sistema: asincrono, no guasti
- ✓ ogni processo P_i gestisce un vector clock VC_i che traccia la conoscenza corrente del numero di messaggi che ogni processo ha inviato. In particolare $VC_i[j]$ rappresenta la conoscenza del numero di messaggi che P_j ha inviato in broadcast e consegnati da P_i
- ✓ Ogni messaggio m ha in piggyback un timestamp $m.VC$, che rivela quanti messaggi ogni processo ha inviato in broadcast nel passato causale del broadcast di m
- ✓ un processo ricevente P_i deve ritardare la consegna di un messaggio m fino a che tutti i messaggi inviati in broadcast nel passato causale di m sono consegnati da P_i .

Causal Broadcast\implementazione

```
procedure broadcast(m) % issued by  $P_i$  %  
   $m.VC := VC_i$ ; % construct the timestamp of  $m$  %  
   $\forall x \in \{1, \dots, n\}$  do send( $m$ ) to  $P_x$  enddo % broadcast event %  
   $VC_i[i] := VC_i[i] + 1$  % one more broadcast by  $P_i$  %  
  
when  $P_i$  receives  $m$  from  $P_j$  %  $m$  piggybacks its vector timestamp  $m.VC$  %  
  delay the delivery until ( $\forall x \in \{1, \dots, n\} : m.VC[x] \leq VC_i[x]$ );  
  if  $i \neq j$  then  $VC_i[j] := VC_i[j] + 1$ ; % update control variable %  
  deliver  $m$  to the upper layer % produce the delivery event %
```

Causal Broadcast\esempio



Quando m' arriva a P_2 , la sua consegna deve essere ritardata poichè m' è arrivato a P_2 primadi m , e l'invio in broadcast di m precede causalmente m' .

Causal Broadcast\correttezza

SAFETY. Dobbiamo dimostrare che

Se 2 messaggi di broadcast m e m' sono tali che $\text{broadcast}(m) \rightarrow \text{broadcast}(m')$, allora ogni processo deve consegnare m prima di m'

Dim. Per assurdo. Supponi che $\text{broadcast}(m) \rightarrow \text{broadcast}(m')$, e che esista un processo p che consegna m' senza aver prima consegnato m

- Caso 1. Messaggi inviati dallo stesso processo p_i .
- Caso 2. Messaggi inviati da processi diversi

Causal Broadcast\correttezza

- Caso 1. $m.V[i] < m'.V[i]$ (terza riga della procedura di broadcast)
Un processo ricevente che riceve m' lo consegna solo se è verificata la condizione di delivery. Se la condizione di delivery è verificata significa che sicuramente p ha consegnato tutti i messaggi che p_i ha inviato prima di m' . Poiché i vector clock sono unici per ogni messaggio (due diversi messaggi hanno timestamp diversi e due timestamp diversi sono associati a messaggi diversi), allora p_i ha già consegnato m . Contraddizione.
- Caso 2. Induzione sulla relazione d'ordine happened-before. K indica la distanza tra due eventi, nei termini del numero di eventi compresi per l'ordine causale tra i due eventi
 - Supponi $k=0$, i.e. $\text{broadcast}(m) \rightarrow \text{broadcast}(m')$ e **non esiste** alcun evento di $\text{broadcast}(m'')$ t.c. $\text{broadcast}(m) \rightarrow \text{broadcast}(m'')$ e $\text{broadcast}(m'') \rightarrow \text{broadcast}(m')$. Se un processo p_i ha fatto il broadcast di m' allora ha consegnato m (visto che sono in relazione), ciò significa che $V[i] \geq m.V[i]$ alla consegna di m . Quando p_i invia m' il timestamp associato è tale che: $m'.V > m.V$. Quindi un processo ricevente (come sopra) non può consegnare m' se non ha già consegnato m . Contraddizione.
 - Per $k > 1$ vale il caso 1, il caso 2 e la proprietà transitiva della happened-before

LIVENESS: ogni messaggio viene alla fine consegnato.

Garantita grazie a:

- 1) il numero di eventi di broadcast di messaggi che precedono causalmente un certo evento di broadcast è finito e
- 2) assunzione di canali 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 a proposito dei messaggi consegnati da P_k . In particolare:

$$MC_i[k][\ell] \quad (i \neq k, \ell)$$

rappresenta la conoscenza di P_i del numero di messaggi che P_k ha consegnato e P_i inviato; $MC_i[j][j]$ rappresenta il numero di sequenza del prossimo messaggio inviato da P_i . Quindi il minimo valore sulla colonna j di MC_i —cioè,

$$\min_{1 \leq x \leq n} (MC_i[x][j])$$

rappresenta la conoscenza di P_i a proposito del numero di sequenza dell'ultimo messaggio stabile che P_j ha inviato.

Protocollo per la rilevazione della stabilità dei messaggi

- ✓ Per propagare stability information, ogni messaggio m che P_i invia ha in piggyback l'identità del suo mittente ($m.sender$) e un timestamp $m.VC$, indicante quanti messaggi P_i ha consegnato da ogni altro processo P_l , ($m.VC$ corrisponde al vettore $MC_i[l][*]$).
- ✓ Due operazioni aggiornano il buffer locale (*buffer*):
 - ✓ $deposit(m)$ inserisce un messaggio m nel buffer
 - ✓ $discard(m)$ rimuove m dal buffer
- ✓ Un processo inserisce un messaggio immediatamente dopo la sua ricezione e lo elimina dal buffer appena il messaggio diventa stabile
- ✓ predicato di stabilità per il messaggio m

$$m.VC[m.sender] \leq \min_{1 \leq x \leq n} (MC_x[x][m.sender]),$$

- ✓ $m.VC[m.sender]$ rappresenta il numero di sequenza di m

Protocollo per la rilevazione della stabilità dei messaggi/implementazione

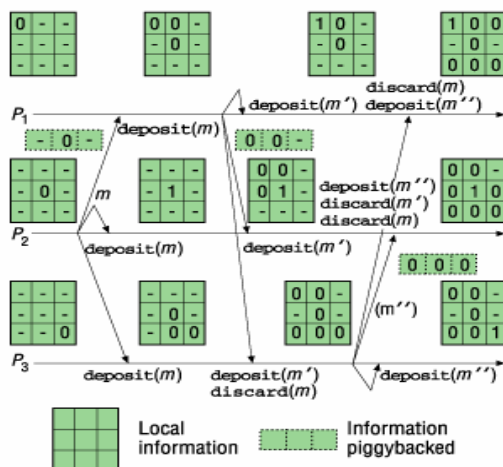
```

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  $P_j$  receives m from  $P_i$  % m piggybacks its vector timestamp m.VC %
  deposit(m); % add m to the local buffer %
   $MC_j[j][*] := m.VC$ ; % update of  $P_j$ 's view of  $P_i$ 's vector %
  if  $i \neq j$  then  $MC_j[i][j] := MC_j[i][j] + 1$ ; % one more message delivered from  $P_i$  %
  deliver m to the upper layer % produce the delivery event %

when  $(\exists m \in buffer_j : m.VC[m.sender] \leq \min_{1 \leq x \leq n} (MC_j[x][m.sender]))$ 
  discard(m) % suppress m from the local buffer %
  
```

Protocollo per la rilevazione della stabilità dei messaggi/esempio



P_3 scarta immediatamente m dopo la ricezione di m' , questo perchè

$$\min_{1 \leq x \leq 3} (MC_j[x][m.sender]) = 0,$$

che corrisponde al numero di sequenza di m .

Alla fine della computazione i buffer di P_1 e P_3 contengono m' e m'' , mentre il buffer di P_2 contiene m'' .