Ordinamento causale e stati globali nei Sistemi Distribuiti

Moreno Marzolla

E-mail: marzolla@dsi.unive.it

Web: http://www.dsi.unive.it/~ marzolla

Introduzione

Supponiamo di avere una *computazione distribuita* attiva, e di voler rispondere a domande del tipo:

- Il sistema è in deadlock?
- Quanti processi stanno leggendo un certo file?
- Quale è il bilancio della banca?

Per rispondere a simili domande, occorre spedire dei messaggi ai processi. Il problema è che, mentre rispondono alla domanda, i processi continuano a scambiarsi altri messaggi.

Sistemi asincroni

Cercheremo di capire cosa possiamo dire a proposito degli *stati globali* di un *sistema distribuito asincrono*.

Definizione: Un sistema distribuito *asincrono* è composto da *processi sequenziali* $P_1, P_2, \ldots, P_n$, e da *canali di comunicazione* che collegano coppie di processi.

La comunicazione si presume *affidabile* (non ci sono messaggi persi), ma i messaggi possono subire ritardi *arbitrariamente lunghi* prima di essere ricevuti. Ogni processo può essere più lento o veloce di un altro.

Computazioni distribuite

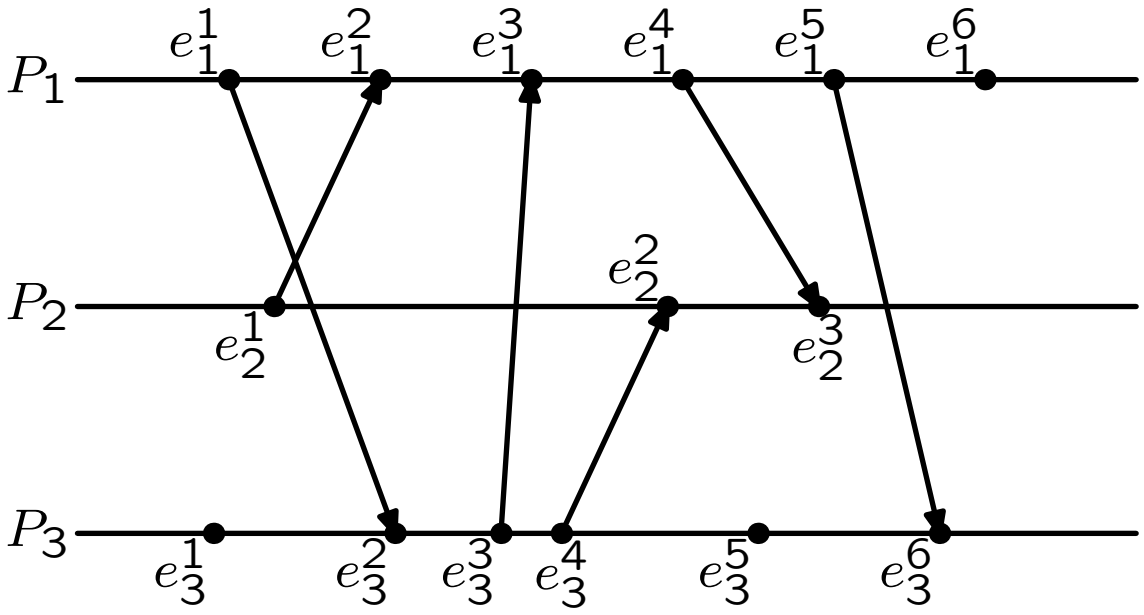
Una computazione distribuita corrisponde ad una *singola esecuzione* di un *programma distribuito* da parte di un insieme di processi.

Ogni processo genera una sequenza di *eventi* che possono essere *eventi interni*, oppure *eventi di comunicazione*.

Computazioni distribuite

Indichiamo con e_i^k l'evento k-esimo eseguito dal processo P_i.

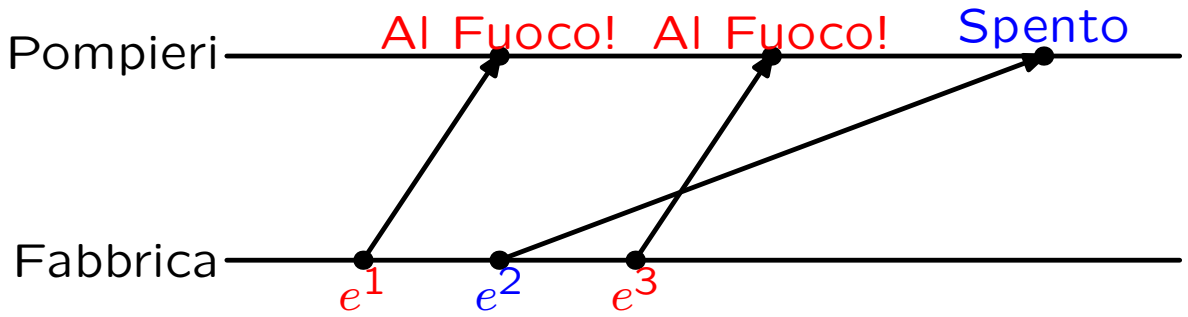
L'evoluzione di una computazione distribuita può essere visualizzata con un *diagramma spazio-tempo*:



I problemi da risolvere: I

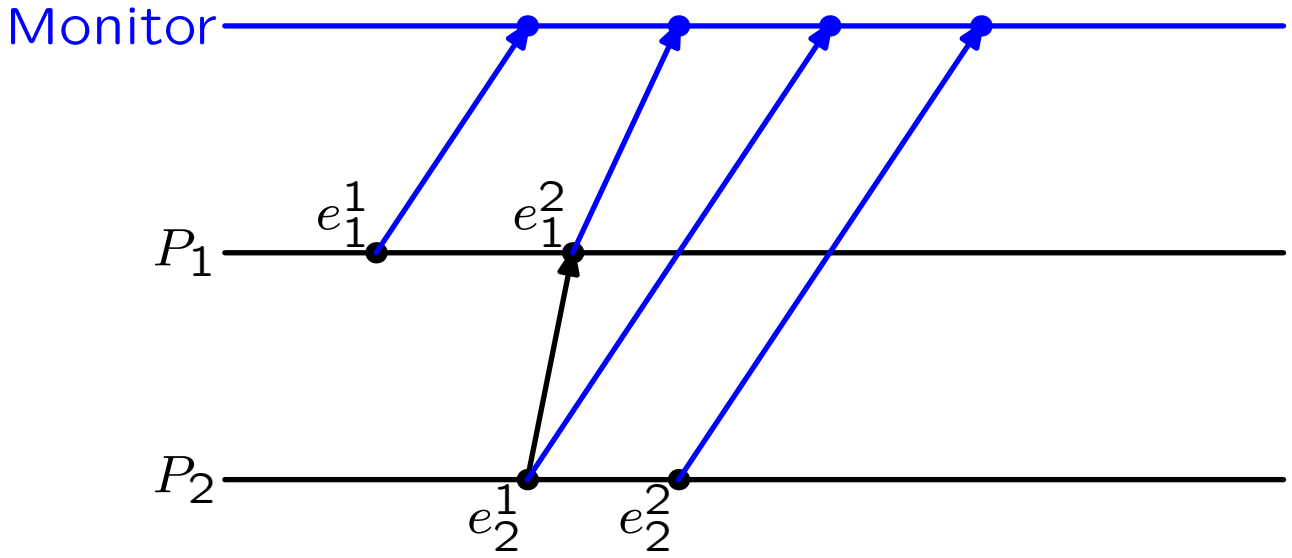
Una fabbrica ha installato un nuovo sistema antincendio, collegato direttamente con la stazione dei pompieri. Il sistema manda un messaggio ai pompieri per avvisare se c'è un incendio, e nel frattempo attiva gli estintori. Se l'incendio viene spento dagli estintori automatici, un nuovo messaggio viene spedito ai pompieri.

Nonostante tutto, la fabbrica è stata distrutta da un incendio. Come è potuto succedere?



I problemi da risolvere: II

Vogliamo osservare il comportamento di un sistema, composto da due processi P_1 e P_2. Usiamo un algoritmo *naif* che *non funziona!*. Per ogni evento compiuto da P_1 e P_2 viene mandato un messaggio al monitor:



Da quanto osserva il monitor, appare che il messaggio spedito da P_2 a P_1 è arrivato prima ancora di essere stato trasmesso!

Storie

La *storia locale* di un processo P_i durante la computazione è la sequenza (possibilmente infinita) di eventi $h_i = e_i^1 e_i^2 \ldots$

La *storia locale parziale* di un processo è un prefisso della storia locale: $h_i^n = e_i^1 e_i^2 \ldots e_i^n$

La *storia globale* della computazione è l'insieme $H = \cup_{i=1}^{n} h_i$ delle storie locali.

Tempo in un sistema asincrono

In un sistema asincrono ciascun processo ha un proprio orologio interno. Tuttavia, è in generale impossibile sincronizzare *esattamente* gli orologi di tutti i processi.

Di conseguenza, il tempo di due eventi che accadono in processi diversi non può generalmente essere utilizzato per decidere quando un evento precede l'altro.

Stato di una computazione

Supponiamo di interrompere una computazione distribuita mediante interruzione *simultanea* di tutti i processi. Lo stato *globale* della computazione è dato da:

- Stato di ogni singolo processo,
- Contenuto di ogni messaggio ancora in transito.

Sapendo lo stato globale, possiamo riconoscere se il sistema è o no in deadlock, oppure possiamo conoscere il bilancio complessivo di una banca.

Il problema è che in un sistema asincrono *non esiste alcun concetto di simultaneità*.

Causa ed effetto

Mancando la nozione di simultaneità in un sistema asincrono, abbiamo bisogno di qualche alternativa.

Disponiamo della nozione di *causa ed effetto*. Se un evento e_i ha *causato* un altro evento e_j, allora e_i ed e_j non possono essere simultanei: e_i *precede* e_j.

Dato che non possiamo guardare dentro ad una computazione distribuita, non c'è modo di sapere se un evento *ha causato* un altro eventi, ma solo se *potrebbe averlo causato*.

Precedenza causale

Definiamo una relazione binaria $\rightarrow$ sull'insieme degli eventi, per formalizzare la nostra nozione intuitiva di "dipendenza causale":

1. se $e_i^k, e_i^l \in h_i$, e $k < l$, allora $e_i^k \rightarrow e_i^l$
2. Se $e_i = \text{send}(m)$, e $e_j = \text{receive}(m)$, allora $e_i \rightarrow e_j$
3. se $e \rightarrow e'$, e $e' \rightarrow e''$, allora $e \rightarrow e''$.

Se $e \rightarrow e'$, allora *e precede causalmente e'*, oppure *e è accaduto prima di e'*.

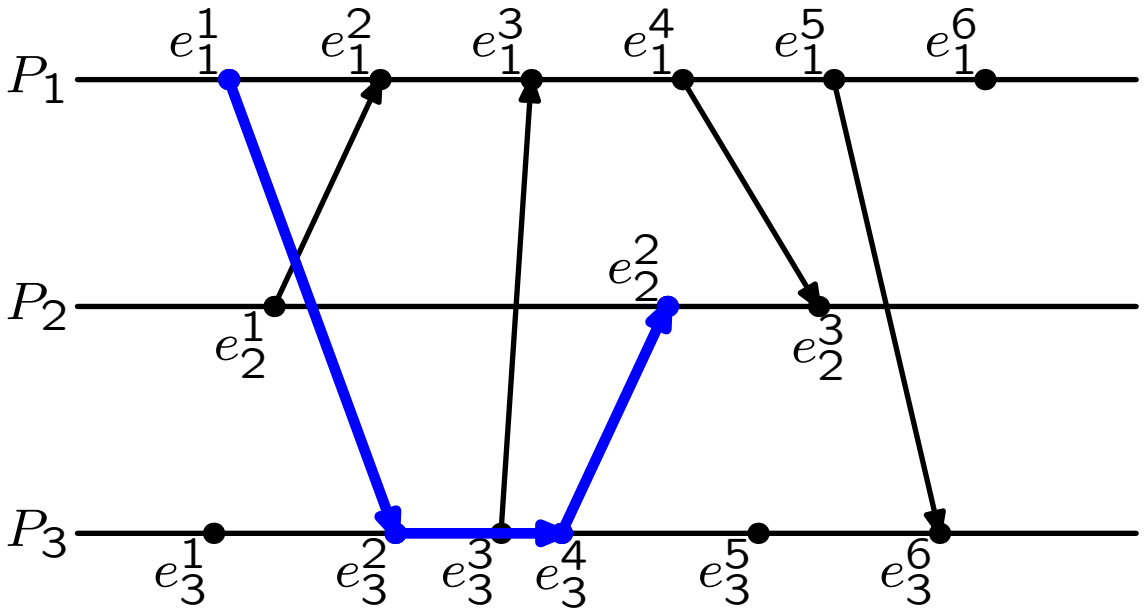
Due eventi sono *concorrenti* se

$$e \parallel e' \equiv \neg(e \rightarrow e') \wedge \neg(e' \rightarrow e)$$

Una computazione distribuita è un *insieme parzialmente ordinato* $\gamma = (H, \rightarrow)$

Diagrammi spazio-tempo

Dato un diagramma spazio-tempo, è facile vedere se due eventi sono causalmente correlati: $e \rightarrow e'$ se è possibile tracciare un percorso da e ad e', procedendo da sinistra verso destra, altrimenti sono concorrenti.



Nell'esempio, si vede che $e_1^1 \rightarrow e_2^2$, ma $e_1^3 \parallel e_3^5$.

Stati Globali

Definizione: Lo *stato locale* del processo P_i subito dopo aver eseguito l'evento e_i^k si denota con σ_i^k. Lo stato iniziale di P_i si indica con σ_i^0.

Lo *stato globale* è l'insieme degli stati locali:

$$\Sigma = \bigcup_{i=1}^{n} \sigma_i$$

Tagli

Definizione: Un *taglio* (*cut*) $\mathcal{C}$ in una computazione distribuita è un insieme di storie locali parziali:

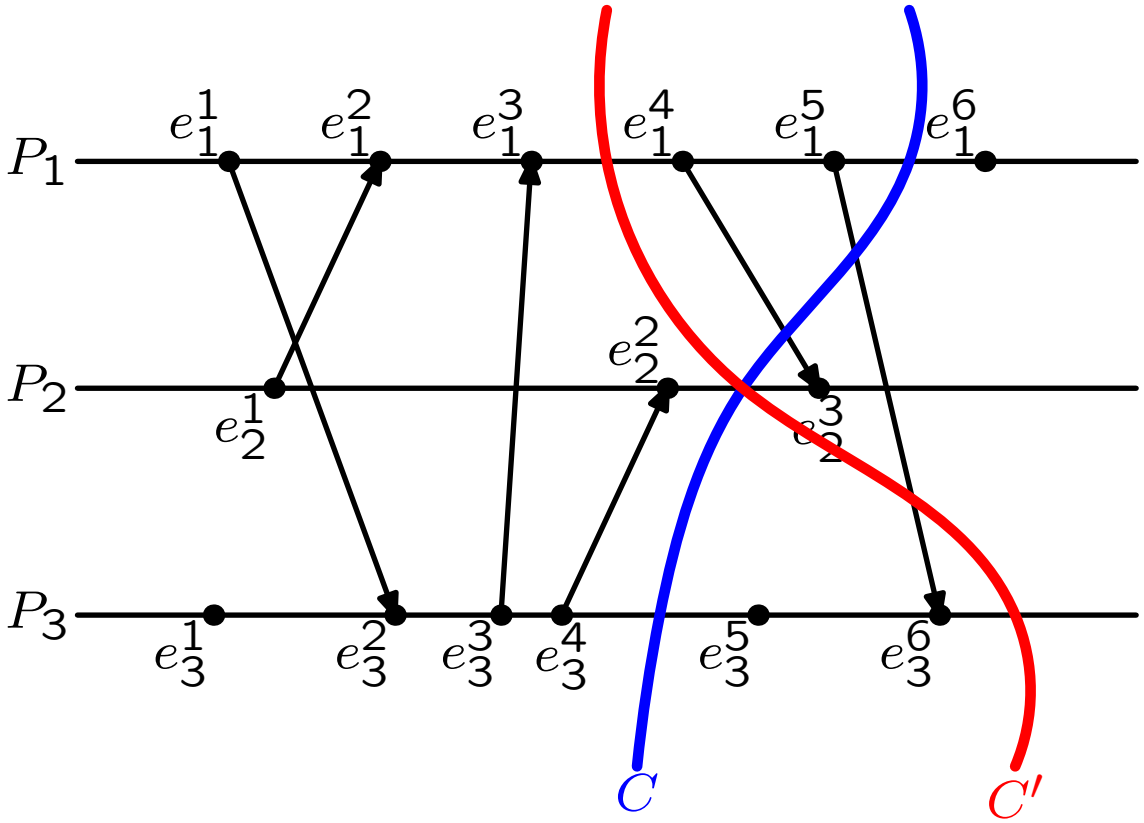
$$\mathcal{C} = \bigcup_{i=1}^{n} h_i^{c_i}$$

Ricordiamo che $h_i^{c_i} = e_i^1 e_i^2 \ldots e_i^{c_i}$.

La *frontiera* del taglio è l'insieme degli ultimi eventi $e_i^{c_i}$, $(i = 1, \ldots, n)$. Per brevità, indicheremo un taglio con l'indice degli eventi della frontiera: $\mathcal{C} = (c_1, c_2, \ldots, c_n)$.

Esempio

Esempio



$$\mathcal{C} = h_1^5 \cup h_2^2 \cup h_3^4 \qquad \mathcal{C}' = h_1^3 \cup h_2^2 \cup h_3^6$$

La frontiera di $\mathcal{C}$ è l'insieme $\{e_1^5, e_2^2, e_3^4\}$, la frontiera di $\mathcal{C}'$ è $\{e_1^3, e_2^2, e_3^6\}$.

Run

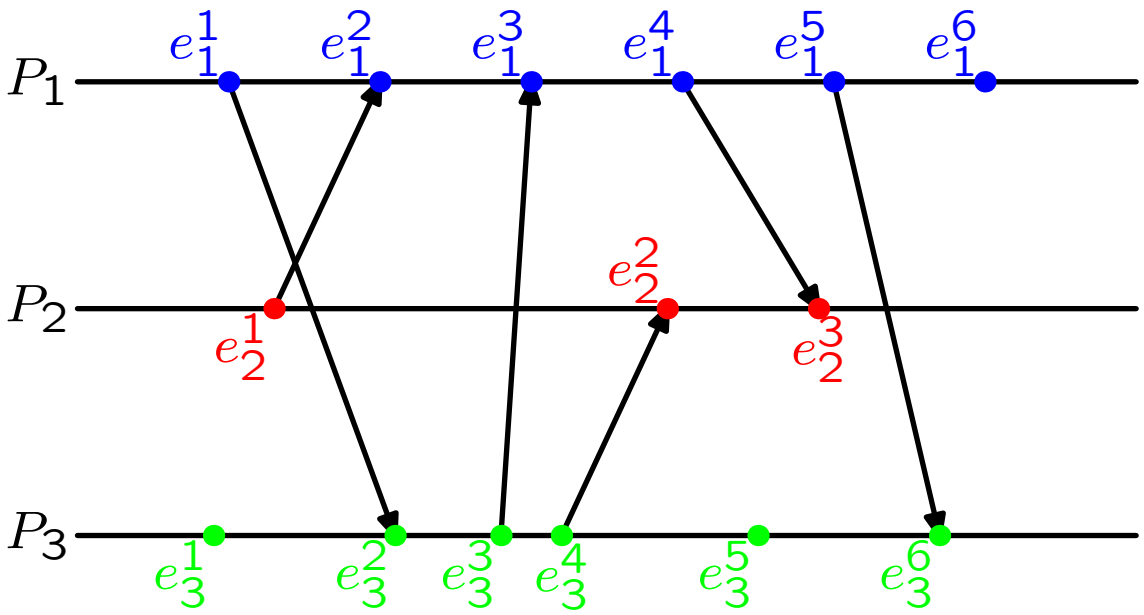
Definizione: Un *run* in una computazione distribuita $(H, \rightarrow)$ è un *ordinamento totale* R di tutti gli eventi che rispetta ciascuna storia locale. Un run è *consistente* se in più rispetta anche la relazione $\rightarrow$

Quindi, in un run consistente:

- Gli eventi di ciascuna storia locale h_i devono comparire in R esattamente con l'ordine in cui compaiono in h_i;
- Se $e \rightarrow e'$, allora e precede e' in R.

Esempio

Esempio



Un possibile run in questo caso è:

$$R = e_3^1 e_1^1 e_3^2 e_2^1 e_3^3 e_3^4 e_2^2 e_1^2 e_3^5 e_1^3 e_1^4 e_1^5 e_3^6 e_2^3 e_1^6$$

Nota bene...

Si noti che la definizione di run (non necessariamente consistente) richiede di rispettare *solo* ogni singola storia locale. In generale:

- Un run può *non* coincidere con una reale esecuzione della computazione distribuita;
- La stessa computazione distribuita può avere più run diversi

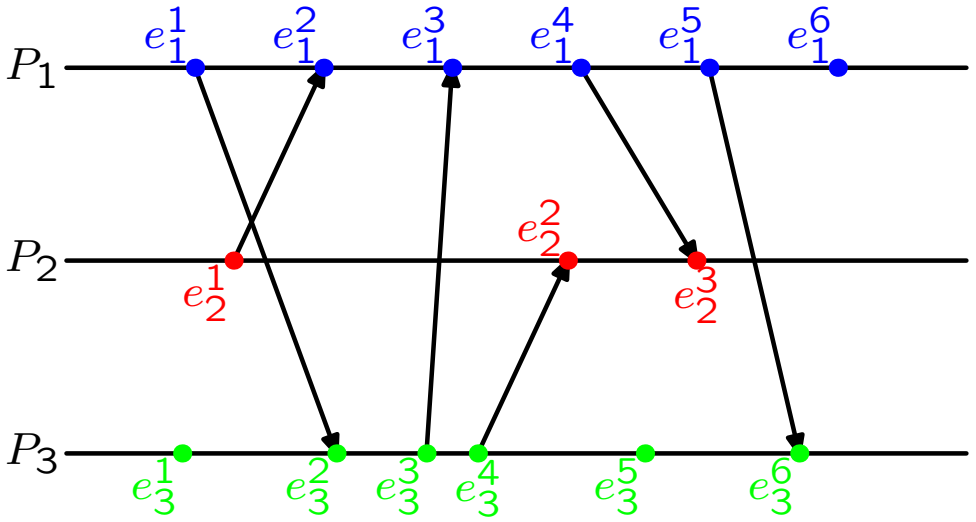
Osservazioni

Definizione: Una *osservazione* è un ordinamento totale Ω di eventi, ricostruito dall'interno del sistema. Un singolo run può avere diverse osservazioni.

La definizione di osservazione è meno restrittiva di quella di run: *qualsiasi* permutazione degli eventi di un run può essere una osservazione.

Esempio

Esempio



$$R = e_3^1 e_1^1 e_3^2 e_2^1 e_3^3 e_3^4 e_2^2 e_1^2 e_3^5 e_1^3 e_1^4 e_1^5 e_3^6 e_2^3 e_1^6$$

$$\begin{aligned} O_1 &= e_2^1 e_1^1 e_3^1 e_3^2 e_3^4 e_1^2 e_2^2 e_3^3 e_1^3 e_1^4 e_5^3 \ldots \\ O_2 &= e_1^1 e_3^1 e_2^1 e_3^2 e_1^2 e_3^3 e_3^4 e_1^3 e_2^2 e_3^5 e_3^6 \ldots \\ O_3 &= e_3^1 e_2^1 e_1^1 e_1^2 e_3^2 e_3^3 e_1^3 e_3^4 e_1^4 e_2^2 e_1^5 \ldots \end{aligned}$$

O_1 non corrisponde ad alcun run poiché gli eventi del processo P_3 non sono elencati nell'ordine in cui vi compaiono.

Consistenza

Osservando lo stato di un sistema distribuito asincrono dal suo interno, in generale è impossibile dire se un particolare stato globale sia mai successo.

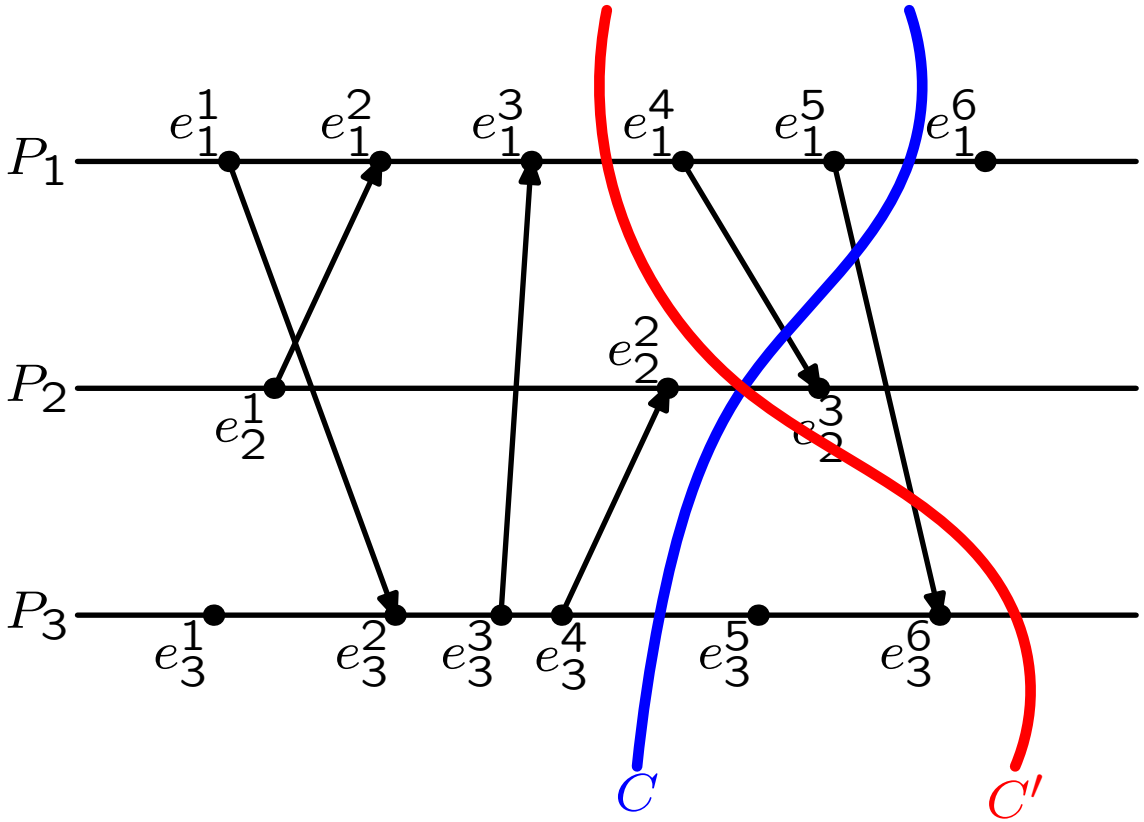
Invece, definiamo *stato consistente* uno stato che *avrebbe potuto* manifestarsi:

Definizione: Un taglio $\mathcal{C}$ è consistente se

$$\forall e, e' : (e \in \mathcal{C}) \wedge (e' \rightarrow e) \Rightarrow e' \in \mathcal{C}$$

Ogni taglio $\mathcal{C} = (c_1, c_2, \ldots, c_n)$ è associato ad uno stato globale $\Sigma = (\sigma_1, \sigma_n, \ldots, \sigma_n)$.

Esempio



Il taglio $\mathcal{C}$ è consistente, mentre $\mathcal{C}'$ no.

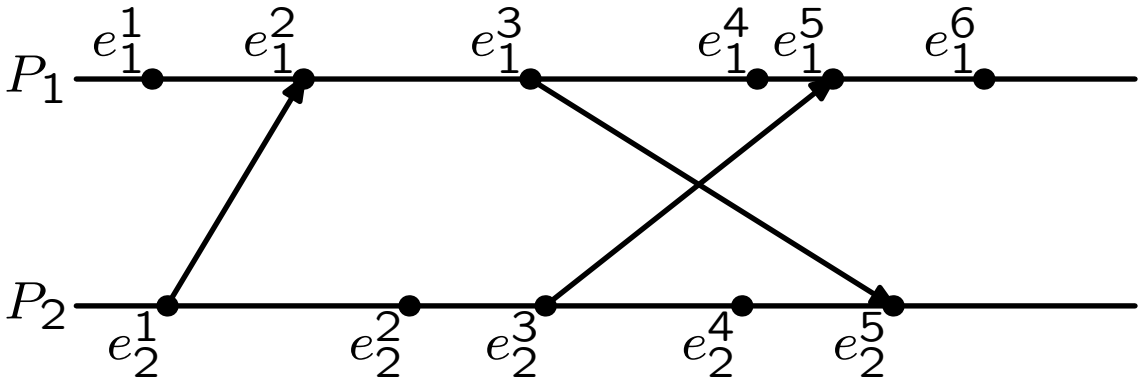
Consistenza (cont.)

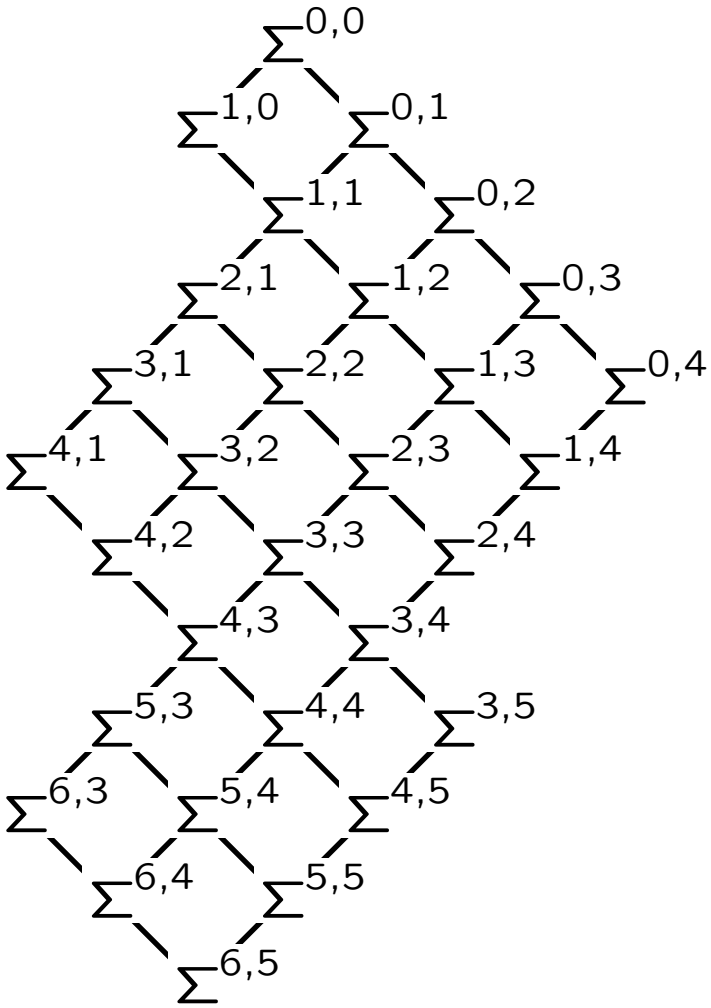
Uno *stato globale consistente* è uno stato che corrisponde ad un taglio consistente.

Una *osservazione consistente* è un ordinamento totale degli eventi in H che sia consistente con l'ordinamento parziale definito dalla relazione di precedenza causale.

Una osservazione consistente $\Omega = e^1 e^2 \ldots$ corrisponde ad una sequenza di stati globali $\Sigma^0, \Sigma^1, \ldots$ (consistenti), dove Σ^i deriva da Σ^{i-1} con in più l'esecuzione del singolo evento e^i.

Ordinamento di Stati Globali





Algoritmo di monitoring
(non funzionante)

Osservare una computazione distribuita dal suo interno si chiama *monitoring*. P_0 è il processo monitor. Il suo scopo è costruisce uno stato globale della computazione.

- Supponiamo che *ogni processo* $P_1, P_2, \ldots$, *ogni volta che esegue un evento*, mandi anche una notifica (un evento extra) al monitor P_0.
- Quando il monitor riceve una notifica, la aggiunge all'osservazione che sta costruendo: per ogni evento di P_i, il monitor aggiorna lo stato locale σ_i.
- A ogni istante, lo stato globale è costituito dall'unione degli stati locali più recenti.

Ricezione e Consegna

Quando il monitor riceve la notifica di un evento, dovrebbe prima assicurarsi che non ci siano altri messaggi in giro che dovrebbero avere la precedenza.

Distinguiamo tra la *ricezione* di un messaggio e la sua *consegna* (*delivery*): quando un messaggio arriva ad un processo, viene ricevuto. Successivamente, una *regola di consegna* deve stabilire quando il messaggio può essere processato.

FIFO

La comunicazione tra i processi P_i e P_j è *First-in First-Out* (FIFO) se per ogni messaggio m e m':

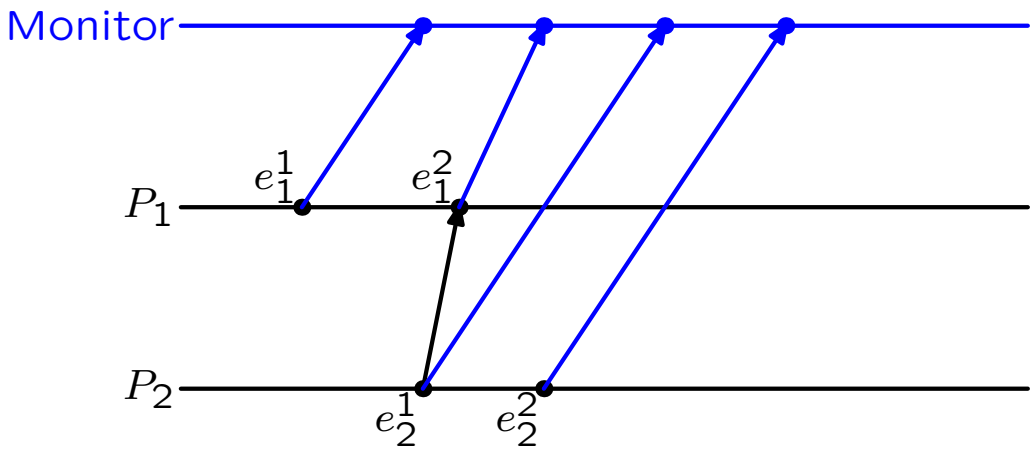
$$\text{send}_i(m) \rightarrow \text{send}_i(m') \Rightarrow \text{deliver}_j(m) \rightarrow \text{deliver}_j(m')$$

La consegna FIFO può essere implementata banalmente (inserendo in ogni messaggio un numero di sequenza).

La consegna FIFO garantisce che le osservazioni fatte dal monitor corrispondano ad un run, ma non è da sola sufficiente a garantire osservazioni *consistenti*.

FIFO non basta...

...come dimostra l'esempio seguente:



Qui l'osservazione ricostruita è

$$O = e_1^1 e_1^2 e_2^1 e_2^2$$

e non è consistente.

Per risolvere questo ulteriore problema dobbiamo introdurre un ordinamento negli eventi, e costruire un monitor che raccolga le notifica rispettando questo ordinamento.

Sistemi Sincroni: Clock

Supponiamo che tutti i processi abbiano accesso ad un *real-time clock globale* RC e che il ritardo che i messaggi possono subire sia limitato superiormente da δ. Ogni messaggio m che notifica un evento e contiene un *timestamp* $TS(m) = RC(e)$.

Delivery Rule 1 Al tempo t, consegna tutti i messaggi ricevuto con timestamp $\leq t - \delta$ in ordine crescente di timestamp.

Le osservazioni così costruite sono consistenti perché soddisfano la

Clock Condition $e \rightarrow e' \Rightarrow RC(e) < RC(e')$;

I clock *real-time* soddisfano la clock condition.

Logical Clock
(Lamport 1978)

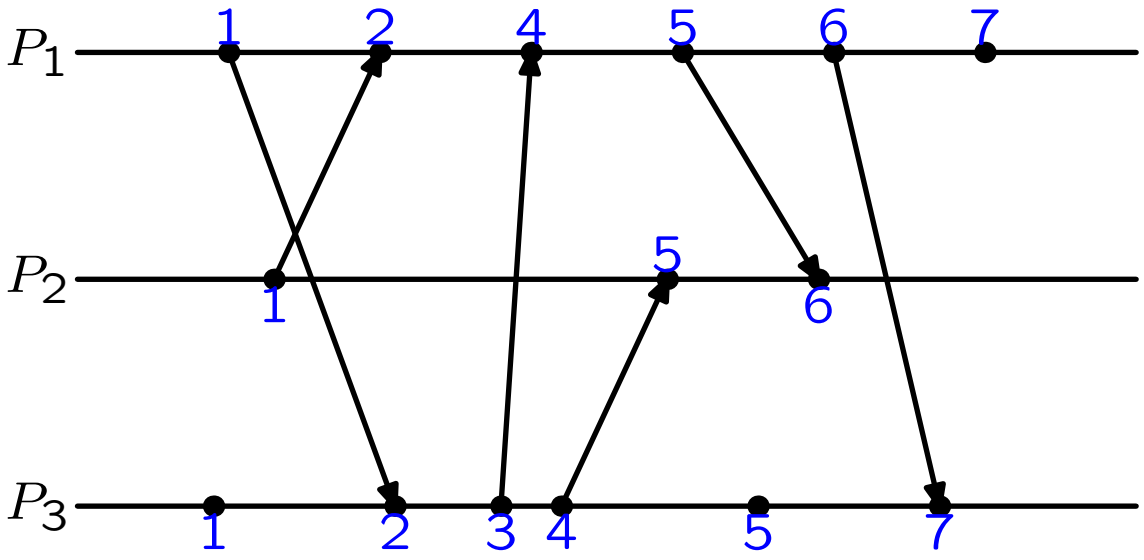
Dobbiamo inventare un orologio per un sistema asincrono che soddisfi la clock condition.

Ogni processo mantiene una variabile intera positiva LC, il *Logical Clock* del processo. $LC(e_i)$ è il valore del clock di P_i quando esegue l'evento e_i.

L'orologio viene aggiornato come segue:

$$LC(e_i) = \begin{cases} LC + 1 \\ \qquad \text{se } e_i \text{ è interno, oppure send} \\ \\ \max\{LC, TS(m)\} + 1 \\ \qquad \text{se } e_i = \text{receive}(m) \end{cases}$$

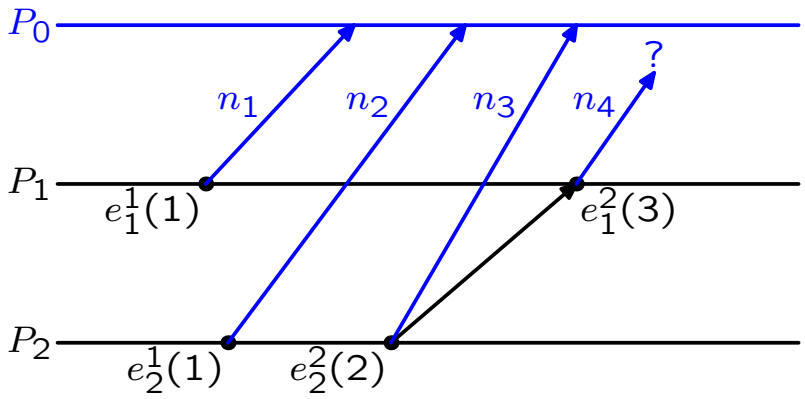
Esempio Logical Clock



$$LC(e_i) = \begin{cases} LC + 1 \\ \qquad \text{se } e_i \text{ è interno, oppure send} \\ \\ \max\{LC, TS(m)\} + 1 \\ \qquad \text{se } e_i = \text{receive}(m) \end{cases}$$

Gap Detection

Il Logical Clock soddisfa la *Clock condition*: $e \rightarrow e' \Rightarrow LC(e) < LC(e')$
Però presenta un altro problema:



Per consegnare un messaggio con $TS = t$ bisogna assicurarsi che nessun messaggio con $TS < t$ sarà ricevuto.

Gap Detection: Dati due eventi e, e' con timestamp $LC(e) < LC(e')$, determinare se esiste un evento e'' tale che

$$LC(e) < LC(e'') < LC(e')$$

Messaggi Stabili

Un messaggio m ricevuto da un processo P è *stabile in P* se nessun messaggio con timestamp più piccolo di $TS(m)$ potrà essere ricevuto da P.

Delivery Rule 2 Consegna tutti i messaggi ricevuti che sono stabili in P_0, in ordine crescente di timestamp

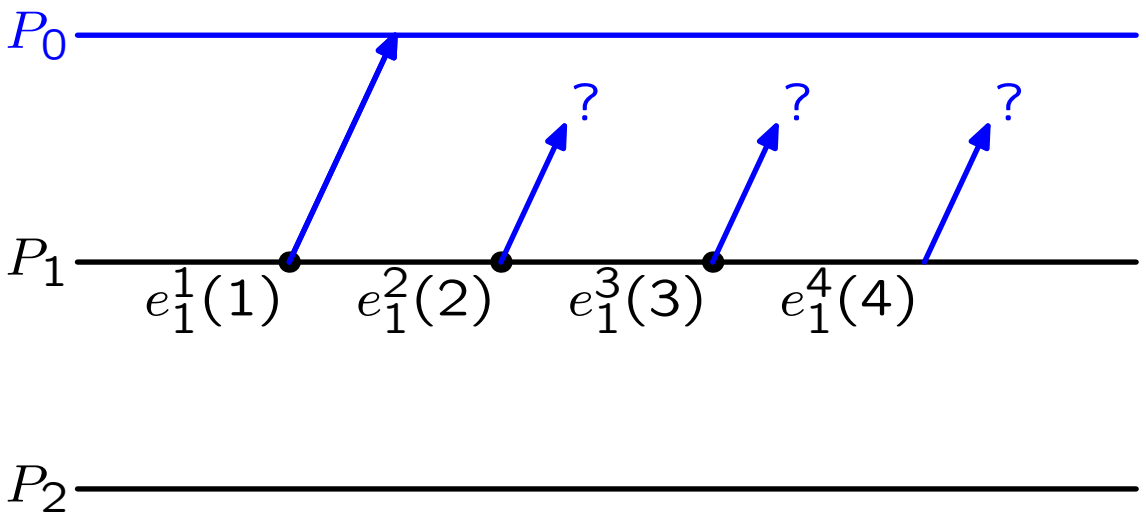
Se il canale tra P_0 e P_i è FIFO, e P_0 riceve m da P_i, P_0 non potrà ricevere successivamente un messaggio m' (sempre da P_i) con $TS(m') < TS(m)$. In sostanza:

Logical Clock e ordinamento FIFO permettono al monitor di costruire sempre osservazioni consistenti.

Però...

Ancora sui messaggi stabili

...Cosa succede se un processo (ad esempio P_2) non manda mai messaggi?



Cambiamo strategia (e orologio)...

Strong Clock Condition

Sappiamo che, con un Logical Clock o un orologio real-time, vale la

Clock Condition

$$e \rightarrow e' \Rightarrow RC(e) < RC(e')$$

Consegnare messaggi secondo l'ordinamento del timestamp è troppo restrittivo perché è possibile che $RC(e) < RC(e')$, ma $e \nrightarrow e'$. Sarebbe preferibile avere un orologio T che soddisfa la

Strong Clock Condition

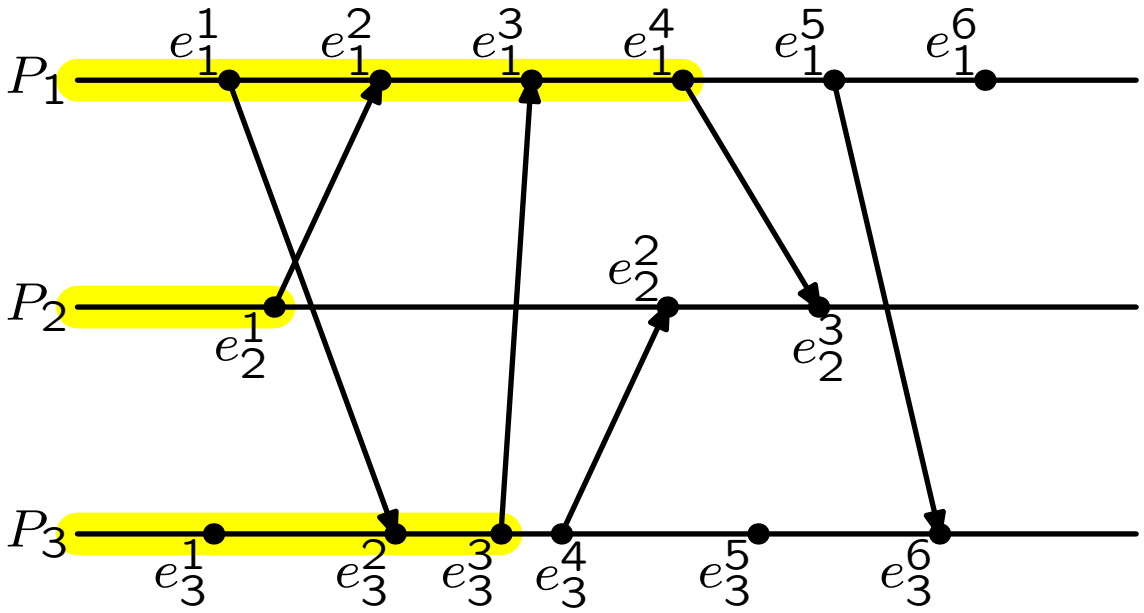
$$e \rightarrow e' \equiv T(e) < T(e')$$

Storia Causale

La *storia causale* (*causal history*) dell'evento e in una computazione distribuita $(H, \rightarrow)$ è l'insieme

$$\theta(e) = \{e' \in H | e' \rightarrow e\} \cup \{e\}$$

Esempio: Storia causale dell'evento e_1^4:



Storia Causale

Le storie causali possono essere usate per implementare la strong clock condition. Infatti:

$$e \rightarrow e' \equiv \theta(e) \subset \theta(e')$$

Oppure, se $e \neq e'$:

$$e \rightarrow e' \equiv e \in \theta(e')$$

La *proiezione* di $\theta(e)$ sul processo P_i è l'insieme $\theta_i(e) = \theta(e) \cap h_i^k = h_i^k$.

$$e_i^k \in \theta_i(e) \Rightarrow \forall j < k : e_i^j \in \theta_i(e)$$

Vector Clock

Associare l'intera sua storia causale ad un evento non è pratico.

Notiamo però quanto segue:

- La proiezione di $\theta(e)$ sul processo P_i è uguale ad un prefisso h_i^k della sua storia causale;
- Per definizione, $\theta(e) = \bigcup_{i=1}^{n} \theta_i(e)$

quindi l'intera storia causale dell'evento e può essere rappresentata da un vettore n-dimensionale, il *vector clock*, in cui la i-esima componente è definita come:

$$VC(e)[i] = k \Leftrightarrow \theta_i(e) = h_i^k$$

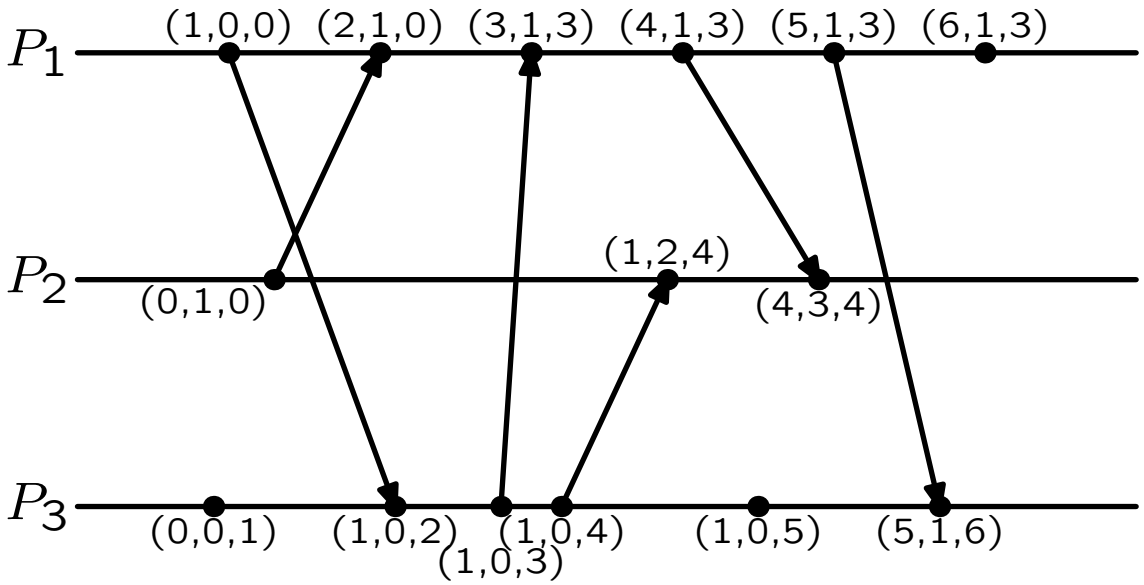
Regole di aggiornamento

Se $e_i = \text{send}(m)$ o è un evento interno:

$$VC(e_i)[i] := VC[i] + 1$$

Se $e_i = \text{receive}(m)$:

$$\begin{aligned} VC(e_i) &:= \max\{VC, TS(m)\} \\ VC(e_i)[i] &:= VC[i] + 1 \end{aligned}$$



Interpretazione del Vector Clock

$VC(e_i)[i]$ rappresenta il numero di eventi che P_i ha eseguito incluso e_i.

$VC(e_i)[j]$, $i \neq j$, rappresenta il numero di eventi di P_j che precedono causalmente l'evento e_i di P_i

Proprietà del Vector Clock

Si definisce la relazione *minore di* nel modo seguente:

$$V < V' \equiv (V \neq V') \wedge (\forall k, 1 \leq k \leq n : V[k] \leq V'[k])$$

Proprietà 1 (Strong Clock Condition).

$$e \rightarrow e' \equiv VC(e) < VC(e')$$

Proprietà 2 (Simple Strong Clock Condition). Dati eventi e_i in P_i ed e_j in P_j, $i \neq j$:

$$e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i]$$

Proprietà del Vector Clock

Gli eventi e_i e e_j sono *mutuamente inconsistenti* se non possono appartenere alla frontiera dello stesso taglio consistente

Proprietà 3 (Concorrenza). Dati eventi e_i in P_i ed e_j in P_j:

$$\begin{aligned} e_i \parallel e_j \quad = \quad & (VC(e_i)[i] > VC(e_j)[i]) \wedge \\ & (VC(e_j)[j] > VC(e_i)[j]) \end{aligned}$$

Proprietà 4 (Mutua inconsistenza). L'evento e_i di P_i è *mutuamente inconsistente* con l'evento e_j di P_j $(i \neq j)$ se e solo se

$$(VC(e_i)[i] < VC(e_j)[i]) \vee (VC(e_j)[j] < VC(e_i)[j])$$

Proprietà del Vector Clock

Proprietà 5 (Taglio consistente). Un taglio definito da $(c_1, \ldots, c_n)$ è consistente se e solo se:

$$\forall i, j : 1 \leq i \leq n, 1 \leq j \leq n : VC(e_i^{c_i})[i] \geq VC(e_j^{c_j})[i]$$

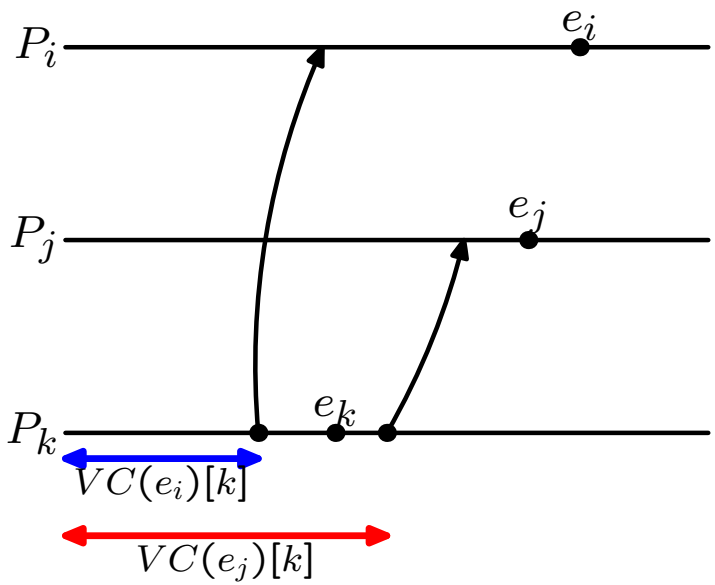
Proprietà 6 (Weak Gap Detection). Dati gli eventi e_i in P_i e e_j in P_j, se esiste un $k \neq j$ tale che

$$VC(e_i)[k] < VC(e_j)[k]$$

allora esiste un evento e_k

$$\neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$$

Dimostrazione Weak Gap Detection



Weak Gap Detection. Dati gli eventi e_i in P_i e e_j in P_j, se esiste un $k \neq j$ tale che

$$VC(e_i)[k] < VC(e_j)[k]$$

allora esiste un evento e_k

$$\neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$$

Implementare Causal Delivery

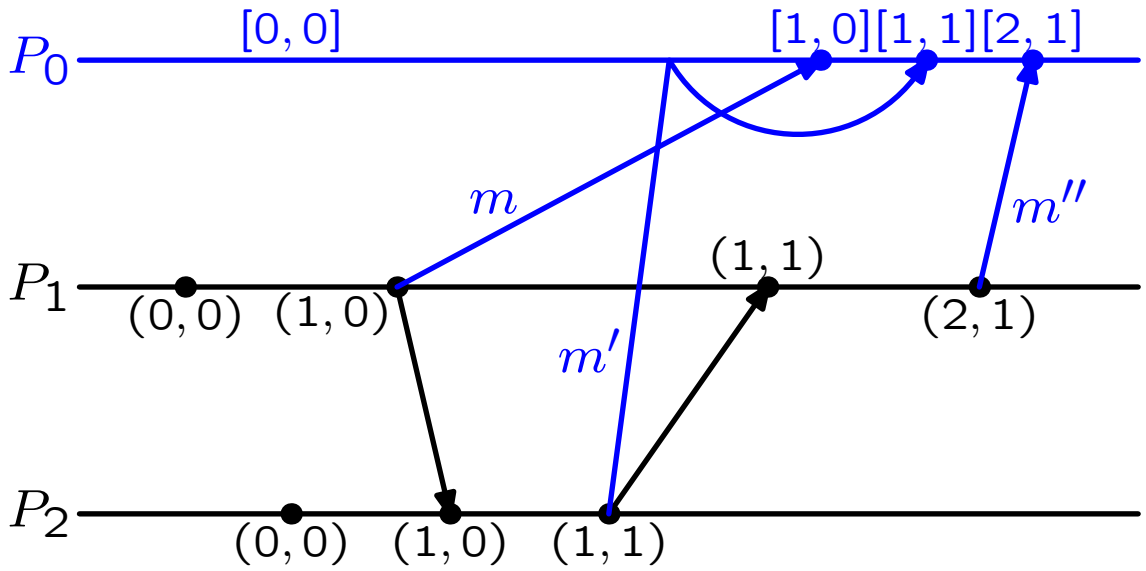
- I processi incrementano la componente locale del loro vector clock solo per gli eventi che sono notificati al monitor
- Ogni messaggio m porta un timestamp $TS(m)$ che è il vector clock dell'evento che viene notificato da m
- Tutti i messaggi ricevuti da P_0 (ma non ancora processati) sono mantenuti in un insieme $\mathcal{M}$, inizialmente vuoto
- Il processo P_0 mantiene un vettore $D[1 \ldots n]$ di contatori, inizialmente tutti zero, tale che $D[i] = TS(m)[i]$, dove m è l'ultimo messaggio consegnato da P_i

Implementare Causal Delivery

Delivery Rule 3 (Causal Delivery). Consegna il messaggio m da P_j quando entrambe le condizioni seguenti sono verificate:

$$\begin{aligned} D[j] &= TS(m)[j] - 1 \\ D[k] &\geq TS(m)[k], \forall k \neq j \end{aligned}$$

Quando P_0 consegna m, setta $D[j]$ a $TS(m)[j]$



Snapshot Distribuiti

Quando il monitor prende uno snapshot degli stati locali in modo non coordinato, non è garantito che lo stato globale risultante sia consistente.

È necessario sviluppare un protocollo di snapshot che costruisce solamente stati globali consistenti.

Ipotesi di lavoro

- I canali di comunicazione preservano l'ordine FIFO dei messaggi
- Lo *stato del canale* $\chi_{i,j}$ che collega P_i con P_j è costituito dai messaggi che P_i ha spedito a P_j, e quest'ultimo non ha ancora ricevuto.
- L'insieme dei canali entranti in P_i è indicaton con IN_i. L'insieme dei canali uscenti da P_i è OUT_i.
- Ogni processo P_i registra il proprio stato locale σ_i e lo stato dei canali entranti $\chi_{j,i}, \forall P_j \in IN_i$.

Algoritmo di Snapshot 1

Supponiamo che tutti i processi abbiano accesso ad un orologio real-time RC, che i messaggi vengano consegnati tutti entro un tempo massimo noto, e che i processi abbiano all'incirca la stessa velocità

1. P_0 manda un messaggio "*Acquisire snapshot al tempo* t_{ss}" a tutti i processi, ove t_{ss} è sufficientemente lontano nel futuro

2. Quando RC segna t_{ss}, ogni P_i registra il suo stato locale σ_i, manda un messaggio vuoto su tutti i canali in uscita, e inizia a registrare i messaggi nei suoi canali in ingresso. Nel frattempo, P_i non esegue alcun altro evento.

3. Quando P_i riceve un messaggio da P_j con timestamp $\geq t_{ss}$, P_i dichiara lo stato $\chi_{j,i}$ a P_0.

Algoritmo di Snapshot 2

...È sufficiente sostituire *Logical Clock* al posto di *real-time clock* nell'algoritmo precedente.

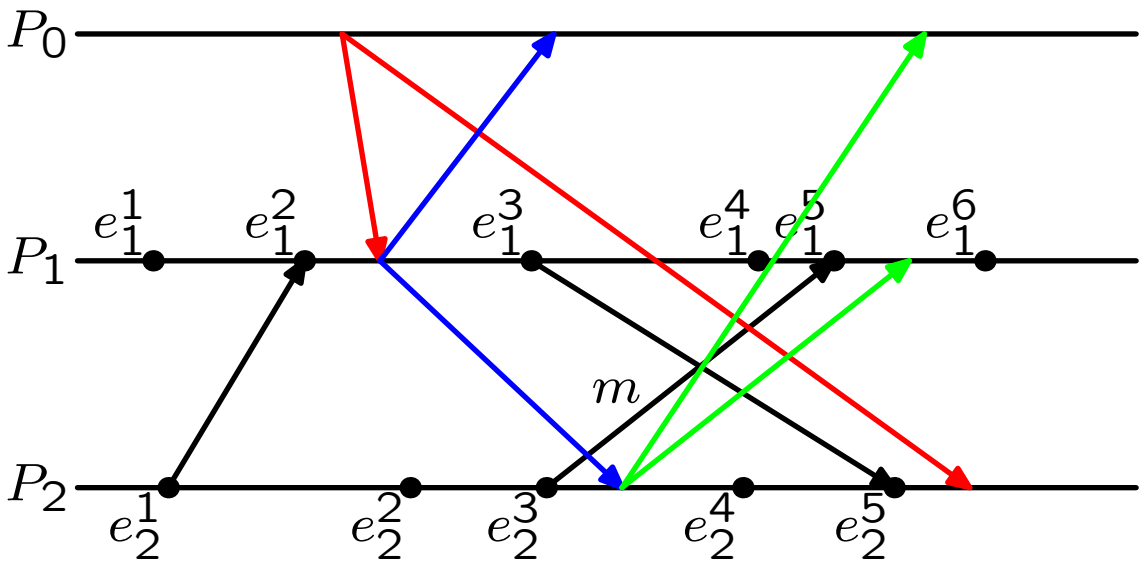
Algoritmo di Snapshot 3
Chandy e Lamport (1986)

1. P_0 si manda un messaggio "*Acquisire snapshot*"

2. Quando P_i riceve il *primo* messaggio "Acquisire snapshot" (diciamo da P_j), registra il suo stato locale σ_i e manda copia del messaggio su tutti i canali di output. $\chi_{j,i}$ è dichiarato essere vuoto e P_i inizia a registrare i messaggi che riceve su tutti i canali in input.

3. Quando P_i riceve un *nuovo* messaggio "Acquisire snapshot" (diciamo da P_s), dichiara che $\chi_{s,i}$ è l'insieme dei messaggi ricevuti da P_s.

Esempio dell'algoritmo Chandy-Lamport



Proprietà dell'algoritmo Chandy-Lamport

Sia Σ^s lo stato globale ottenuto dall'algoritmo Chandy-Lamport. È possibile dimostrare che Σ^s è uno stato consistente.

Purtroppo, in generale non è vero che il run corrispondente all'esecuzione del sistema distribuito sia passato per lo stato Σ^s.

Vedremo comunque che lo stato globale Σ^s ha delle utili proprietà.

Supponiamo che Σ^{start} e Σ^{end} siano gli stati globali risp. all'inizio e al termine dell'algoritmo di Chandy-Lamport. Allora vale la seguente relazione:

$$\Sigma^{start} \rightsquigarrow \Sigma^s \rightsquigarrow \Sigma^{end}$$

Proprietà di Stati Globali

Consideriamo un qualunque algoritmo di snapshot. L'algoritmo in generale inizia con il sistema nello stato Σ^a, termina con il sistema nello stato Σ^f e fornisce lo snapshot Σ^s.

Per l'algoritmo di Chandy-Lamport si ha

$$\Sigma^a \rightsquigarrow \Sigma^s \rightsquigarrow \Sigma^f$$

Definizione Le proprietà Φ che una volta vere, rimangono vere anche se lo stato evolve, sono dette *stabili*.

Esempio di proprietà stabili: deadlock, perdita di tutti i tokens, irraggiungibilità dello storage (*garbage collection*)...

Proprietà stabili

$$\Sigma^a \rightsquigarrow \Sigma^s \rightsquigarrow \Sigma^f$$

Se Φ è una proprietà stabile, allora le seguenti conclusioni sono possibili:

$$(\Phi \text{ vera in } \Sigma^s) \Rightarrow (\Phi \text{ vera in } \Sigma^f)$$

e anche

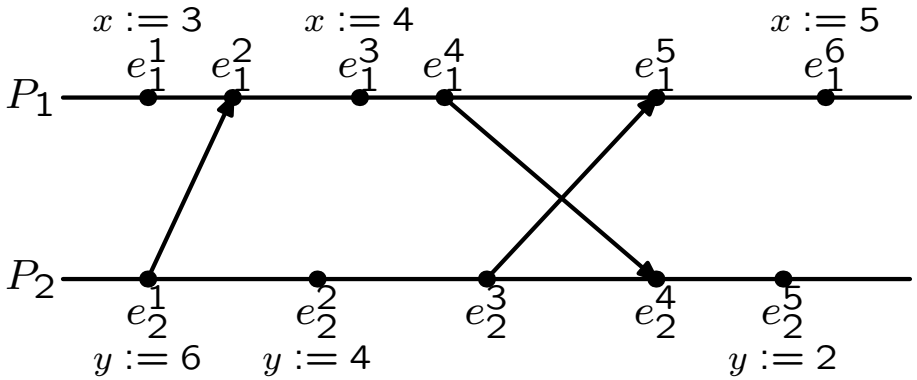
$$(\Phi \text{ falsa in } \Sigma^s) \Rightarrow (\Phi \text{ falsa in } \Sigma^a)$$

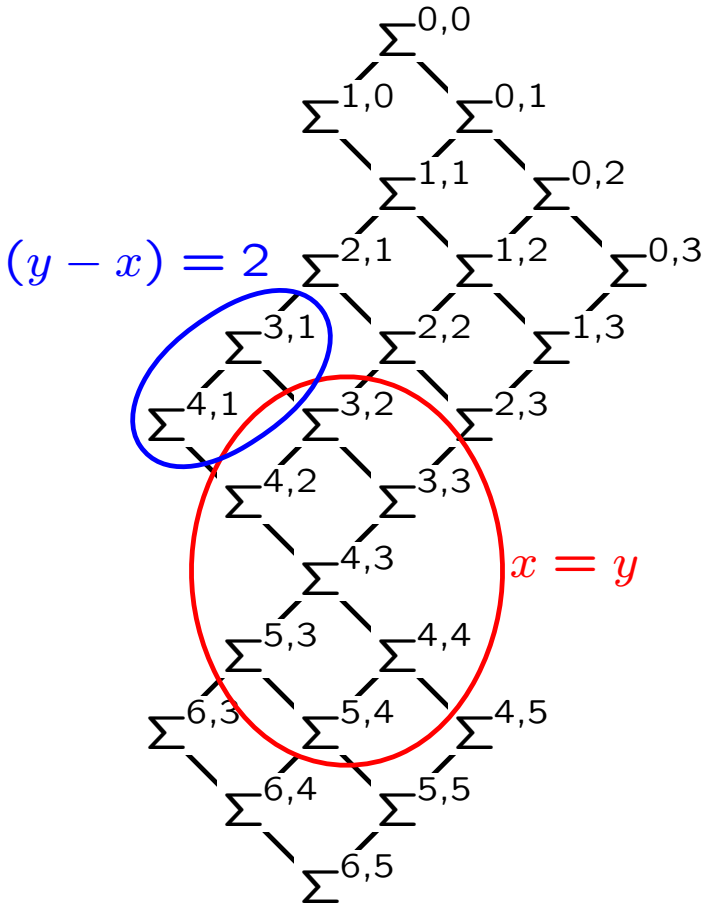
Sfortunatamente, molti predicati che ci possono interessare non godono di questa proprietà

Esempio

Predicati non stabili

Consideriamo la computazione seguente:





Possibly e Definitely

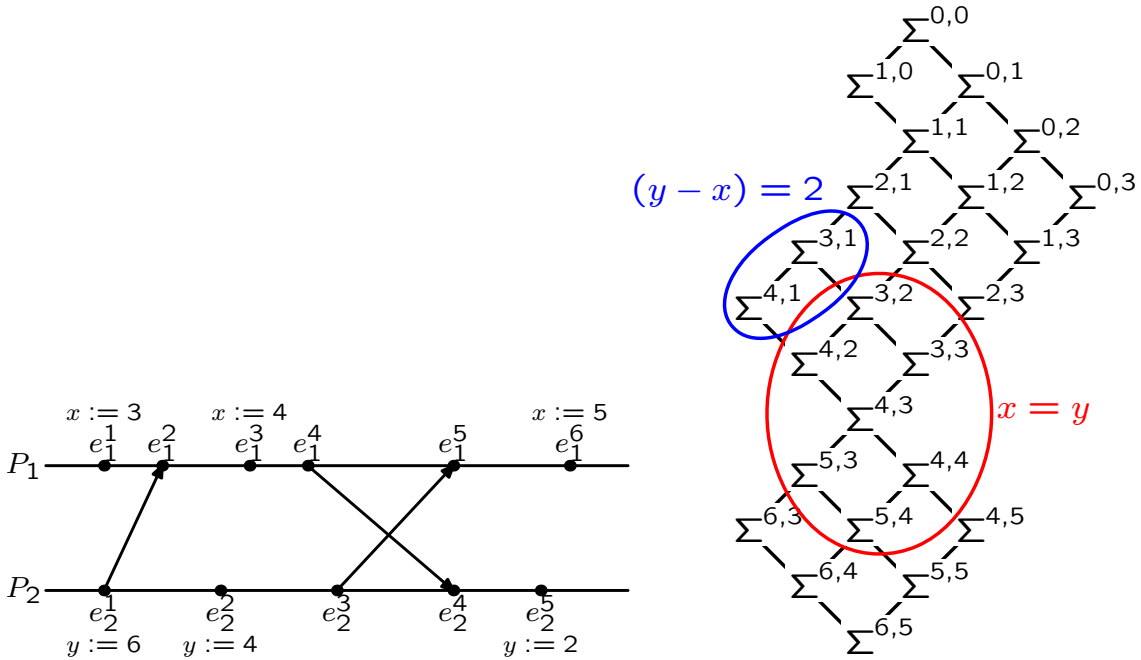
C'è un modo per dare un senso anche ai predicati non stabili. Definiamo:

Possibly(Φ) Esiste una osservazione consistente O della computazione tale che Φ vale in uno stato globale di O

Definitely(Φ) Per ogni osservazione consistente O della computazione, esiste uno stato globale di O in cui Φ vale.

Esempio

Nel solito esempio:



Valgono:

$$\text{Possibly}((y - x) = 2), \quad \text{Definitely}(x = y)$$

Dalla definizione, costruiamo due algoritmi per Possibly(Φ) e Definitely(Φ); entrambi sfruttano l'ordinamento degli stati.

Possibly(Φ)

current := $\{\Sigma^{0\ldots0}\}$
$l := 0$
while nessuno stato in current soddisfa Φ **do**
  if *current* = {stato_finale} **then**
    return false
  else
    $l := l + 1$
    current := {stati di livello l}
  end if
end while
return true

Definitely(Φ)

$last := \{\Sigma^{0\ldots0}\}$
Rimuovi gli stati di *last* che soddisfano Φ
$l := 1$
while $last \neq \emptyset$ **do**
  current := {stati di livello l raggiungibili da *last*}
  Rimuovi gli stati di *current* che soddisfano Φ
  if *current* = {stato_finale} **then**
    return false
  else
    $l := l + 1$
    $last := current$
  end if
end while
return true

Riferimenti

- Sape Mullender, ed. **Distributed Systems (2nd edition)**, Addison-Wesley, pp. 55–96
- Colouris, Dollimore, Kindberg, **Distributed Systems: conecepts and design**, pp. 397–415
- L. Lamport, **Time, Clocks, and the Ordering of Events in a Distributed System**, Comm ACM 21:7, 1978
- K. Mani Chandy, L. Lamport, **Distributed Snapshots: Determining Global States of Distributed Systems**, ACM Trans. on Comp. Systems 3:1, 1985
- Queste trasparenze: pagina del corso oppure `http://www.dsi.unive.it/~marzolla`