Università degli Studi di Roma “Tor Vergata”

Facoltà di Ingegneria

Sincronizzazione nei Sistemi Distribuiti

(parte 1)

Corso di Sistemi Distribuiti

Valeria Cardellini

Anno accademico 2009/10

La nozione di tempo nei SD

- In un SD
  - I processi girano su macchine diverse connesse in rete
  - I processi cooperano per portare al termine una computazione
  - La comunicazione avviene esclusivamente attraverso lo scambio di messaggi
- *Osservazioni:*
  - Molti algoritmi richiedono sincronizzazione per essere portati a termine
    - I nodi di un SD possono avere necessità di effettuare azioni sincronizzate rispetto allo stesso tempo assoluto
  - Molti algoritmi richiedono che gli eventi siano ordinati
    - Nei SD i messaggi arrivano con dei timestamp in modo che si possa sapere in che ordine devono essere eseguiti
- *Conseguenza*: nei SD il tempo è un fattore critico!

La nozione di tempo nei SD (2)

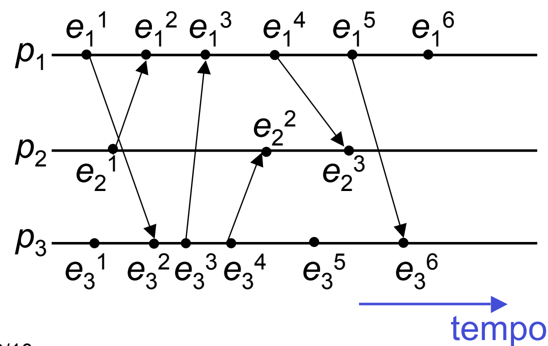
- In un sistema centralizzato è possibile stabilire l'ordine in cui gli eventi si sono verificati
  - Esiste un'unica memoria comune e un unico clock
- In un sistema distribuito è impossibile avere un unico clock fisico comune a tutti i processi
  - Eppure la computazione globale può essere vista come un ordine totale di eventi se si considera il tempo al quale sono stati generati
- Per molti problemi nei SD risalire a questo tempo è di vitale importanza, o comunque è importante stabilire quale evento è stato generato prima di un altro

La nozione di tempo nei SD (3)

- Soluzione 1: **sincronizzazione degli orologi fisici**
  - Il middleware di ogni nodo del SD aggiusta il valore del suo clock fisico in modo coerente con quello degli altri o con quello di un clock di riferimento
- Soluzione 2: **sincronizzazione degli orologi logici**
  - Lamport ha mostrato come l'accordo degli orologi fisici non sia necessario, ma lo sia solo l'ordinamento degli eventi

Modello della computazione

- Componenti del SD: *N* processi e canali di comunicazione
- Ogni processo *p* genera una sequenza di eventi
  - Eventi ***interni*** (cambiamento dello stato del processo) ed ***esterni*** (send/receive di messaggi)
  - Denotiamo con e_i^k il *k*-esimo evento generato da p_i
- L'evoluzione di una computazione può essere visualizzata con un *diagramma spazio-tempo*



Storia della computazione

- Denotiamo con $\rightarrow_i$ la relazione di ordinamento su un processo P_i tra due eventi:

  $e \rightarrow_i e'$ se e solo se *e* è accaduto prima di *e'* in P_i

- Storia locale: sequenza di eventi generati da un singolo processo
  - Ad es.: history(P_1) = h_1 = $<e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_1^6>$
- Storia locale parziale: prefisso della storia locale
  - Ad es.: $h_1^m = e_1^1 \dots e_1^m$
- Storia globale: insieme delle storie locali

  $H = \cup_i h_i$ per $1 \leq i \leq n$

Timestamping

- Tecnica del ***timestamping***: ogni processo etichetta (con un timestamp) gli eventi
  – In questo modo è possibile realizzare una storia globale del sistema
- Soluzione banale: ogni processo etichetta gli eventi con il proprio clock fisico
- Funziona?
  – L'ordinamento degli eventi di uno stesso processo si può ricostruire
  – Ma l'ordinamento di eventi tra processi diversi?
  – In un sistema distribuito è *impossibile* avere un unico clock fisico condiviso da tutti i processi

SD sincroni e asincroni

- Caratteristiche di un **SD sincrono**
  – Esistono dei vincoli sulla velocità di esecuzione di ciascun processo
    - Il tempo di esecuzione di ciascuno passo è limitato, sia con lower bound che con upper bound
  – Ciascun messaggio trasmesso su un canale di comunicazione è ricevuto in un tempo limitato
  – Ciascun processo ha un clock locale con un tasso di scostamento del clock (*clock drift rate*) dal clock reale conosciuto e limitato
- In un **SD asincrono**
  – *Non ci sono vincoli* sulla velocità di esecuzione dei processi, sul ritardo di trasmissione dei messaggi e sul tasso di scostamento dei clock

Soluzioni per sincronizzare gli orologi

- Prima soluzione
  - Tentare di sincronizzare con *una certa approssimazione* i clock fisici locali ad ogni processo attraverso opportuni algoritmi
  - Il processo può etichettare gli eventi con il valore del suo clock fisico (che risulta sincronizzato con gli altri con una certa approssimazione)
  - Il timestamping è quindi basato sulla nozione di tempo fisico (***clock fisico***)

- E' sempre possibile mantenere l'approssimazione dei clock limitata?
  - **NO** in un SD asincrono
  - In un modello asincrono il timestamping non si può basare sul concetto di tempo fisico
  - Si introduce la nozione di clock basata su un tempo logico *(**clock logico**)*

Clock fisico

- All'istante di tempo reale t, il sistema operativo legge il tempo dal clock hardware $H_i(t)$ del computer
- Quindi produce il clock software

$$C_i(t)= aH_i(t) + b$$

  che approssimativamente misura l'istante di tempo fisico t per il processo P_i
  - Ad es. $C_i(t)$ è un numero a 64 bit che fornisce i nsec trascorsi all'istante t da un istante di riferimento
  - In generale il clock non è completamente accurato: può essere diverso da t
  - Se C_i si comporta abbastanza bene, può essere usato come timestamp per gli eventi che occorrono in P_i
- Quanto deve essere la risoluzione del clock (periodo che intercorre tra gli aggiornamenti del valore del clock) per poter distinguere due eventi?
  - $T_{risoluzione} < \Delta T$ tra due eventi rilevanti

Clock fisici in un SD

- Diversi clock locali possono avere valori diversi
- Skew: differenza istantanea fra il valore di due qualsiasi clock
- Drift: i clock contano il tempo con differenti frequenze (fenomeno dovuto a variazioni fisiche dell'orologio), quindi divergono nel tempo rispetto al tempo reale
- Drift rate: differenza per unità di tempo di un clock rispetto ad un orologio ideale
  - Ad es. drift rate di 2 μsec/sec significa che il clock incrementa il suo valore di 1 sec+2 μsec ogni secondo
  - I normali orologi al quarzo deviano di circa 1 sec in 11-12 giorni (10^{-6} sec/sec)
  - Orologi al quarzo ad alta precisione hanno un drift rate di circa 10^{-7} o 10^{-8} sec/sec
  - Anche se i differenti clock di un SD vengono sincronizzati in un certo istante, a causa del drift rate, dopo un certo intervallo di tempo, saranno di nuovo disallineati → occorre eseguire una sincronizzazione periodica per riallineare i clock

UTC

- Coordinated Universal Time (UTC) è uno standard internazionale per mantenere il tempo
- Basato sul tempo atomico ma occasionalmente aggiustato utilizzando il tempo astronomico
  - Numero di transizioni al secondo dell'atomo di cesio 133
- I clock fisici che usano oscillatori atomici sono i più accurati (drift rate pari a 10^{-13})
- L'output dell'orologio atomico è inviato in broadcast da stazioni radio su terra e da satelliti (es. GPS)
  - In Italia: Istituto Galileo Ferraris
- Computer con ricevitori possono sincronizzare i loro clock con questi segnali
  - Segnali da stazioni radio su terra hanno un'accuratezza di circa 0,1-10 msec
  - Segnali da GPS hanno un'accuratezza di circa 1 μsec

Sincronizzazione di clock fisici

- Come si possono sincronizzare i clock fisici con gli orologi del mondo reale?
- Come si possono sincronizzare i clock fisici tra di loro?
- Sincronizzazione **esterna**

  I clock C_i (per $i = 1, 2, \dots N$) sono sincronizzati con una sorgente di tempo S (UTC), in modo che, dato un intervallo I di tempo reale:

  $|S(t) - C_i(t)| < D$ per $1 \leq i \leq N$ e per tutti gli istanti in I

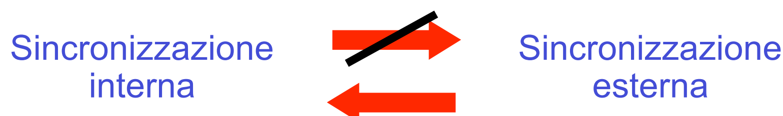
  - I clock C_i hanno un'accuratezza che si mantiene all'interno del bound (*precisione*) D
- Sincronizzazione **interna**

  Due clock C_i e C_j sono sincronizzati l'uno con l'altro in modo che:

  $|C_i(t) - C_j(t)| < D$ per $1 \leq i,j \leq N$ nell'intervallo I

  - I clock C_i e C_j si accordano all'interno del bound D

Sincronizzazione di clock fisici (2)



- I clock sincronizzati internamente non sono necessariamente sincronizzati anche esternamente
  - Tutti i clock possono deviare collettivamente da una sorgente esterna sebbene rimangano tra loro sincronizzati entro il bound D

- Se l'insieme dei processi è sincronizzato esternamente entro un bound D, allora è anche sincronizzato internamente entro un bound 2D

Correttezza di clock fisici

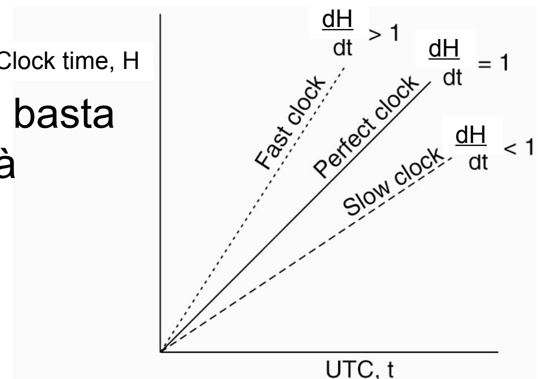
- Un clock hardware *H* è ***corretto*** se il suo drift rate si mantiene all'interno di un bound $\rho > 0$ limitato
  - Il drift rate di un clock corretto è compreso tra $-\rho$ e $+\rho$
- Se il clock *H* è corretto, allora l'***errore*** che si commette nel misurare un intervallo di istanti reali $[t, t']$ (con $t'>t$) è ***limitato***:

$$(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 + \rho)(t' - t)$$

  - Si evitano "salti" del valore del clock

- Per il clock software *C* spesso basta una condizione di monotonicità

$$t' > t \text{ implica } C(t') > C(t)$$



Quando sincronizzare?

- Consideriamo 2 clock con stesso tasso di scostamento massimo pari a ρ
- Ipotizziamo che dopo la sincronizzazione i 2 clock si scostino dall'UTC in senso opposto
  - All'istante Δt si saranno scostati di $2\rho \Delta t$
- Per garantire che i 2 clock non differiscano mai più di δ, occorre sincronizzarli almeno ogni $\delta/2\rho$ secondi

Sincronizzazione interna in un SD sincrono

- Algoritmo di sincronizzazione tra due processi in un SD sincrono

1. Un processo P_1 manda il suo clock locale t ad un processo P_2 tramite un messaggio m
2. P_2 riceve m e imposta il suo clock a $t + T_{trasm}$ dove T_{trasm} è il tempo di trasmissione del messaggio m

   T_{trasm} non è noto ma, essendo il SD sincrono, $T_{min} \leq T_{trasm} \leq T_{max}$

   Sia $u = (T_{max} - T_{min})$ l'incertezza sul tempo di trasmissione (ovveo l'ampiezza dell'intervallo)
3. Se P_2 imposta il suo clock a $t + (T_{max} + T_{min})/2$ è possibile mantenere lo skew tra i due clock al più pari a $u/2$
   - In generale, con N clock il bound ottimo sullo skew è pari a $u(1-1/N)$

- Per un SD asincrono

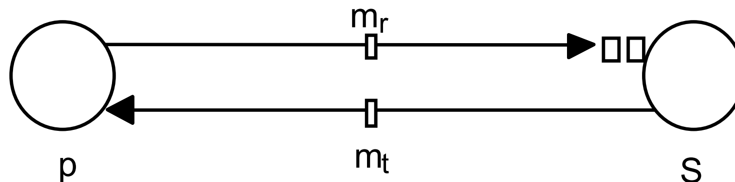
  $T_{trasm} = T_{min} + x$, con $x \geq 0$ e non noto
  - Occorrono altri algoritmi di sincronizzazione dei clock fisici

Sincronizzazione mediante time service

- Un time service può fornire l'ora con precisione
  - Dotato di un ricevitore UTC o di un clock accurato
- Il gruppo di processi che deve sincronizzarsi usa un time service
  - Time service implementato in modo centralizzato da un solo processo (time server) oppure in modo decentralizzato da più processi
- Time service centralizzati
  - Request-driven: algoritmo di Cristian (1989)
    - Sincronizzazione esterna
  - Broadcast-based: algoritmo di Berkeley Unix - Gusella & Zatti (1989)
    - Sincronizzazione interna
- Time service distribuiti
  - Network Time Protocol
    - Sincronizzazione esterna

Algoritmo di Cristian

- Un time server *S* (*passivo*) riceve il segnale da una sorgente UTC (sincronizzazione esterna)
- Un processo *p* richiede il tempo con un messaggio m_r e riceve *t* nel messaggio m_t da *S*
- *p* imposta il suo clock a $t + T_{round}/2$
  - T_{round} è il round trip time registrato da *p*



- Osservazioni
  - Un singolo time server potrebbe guastarsi
    - Soluzione: usare un gruppo di time server sincronizzati
  - Non prevede time server maliziosi
  - Permette la sincronizzazione solo se T_{round} è breve → adatto per reti locali con bassa latenza

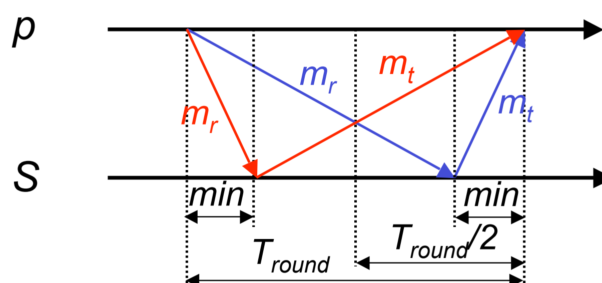
Accuratezza dell'algoritmo di Cristian

Caso 1: *S* non può mettere *t* in m_t prima che sia trascorso *min* dall'istante in cui *p* ha inviato m_r

- *min* è il minimo tempo di trasmissione tra *p* e *S*

Caso 2: *S* non può mettere *t* in m_t dopo il momento in cui m_t arriva a *p* meno *min*

- Il tempo di S quando arriva m_t è compreso in $[t + min, t + T_{round} - min]$
  - L'ampiezza di tale intervallo è $T_{round} - 2\ min$
- accuratezza $\leq \pm (T_{round}/2 - min)$



Algoritmo di Berkeley

- Algoritmo per la sincronizzazione interna di un gruppo di macchine
- Il *master* (time server *attivo*) richiede, attraverso broadcast, il valore dei clock delle altre macchine (*slave*)
- Il master usa i RTT per stimare i valori dei clock degli slave
  - In modo simile all'algoritmo di Cristian:

    $$d_i = (C_M(t_1) + C_M(t_3))/2 - C_i(t_2)$$

    dove d_i è la differenza tra il clock del master e quello dello slave *i*, $C_M(t_1)$ e $C_M(t_3)$ sono i valori del clock sul master, $C_i(t_2)$ è il valore del clock sullo slave *i*
- Calcola la media delle differenze dei clock
- Invia un valore correttivo opportuno agli slave
  - Se il valore correttivo prevede un salto indietro nel tempo, lo slave non imposta il nuovo valore ma rallenta (monotonicità)

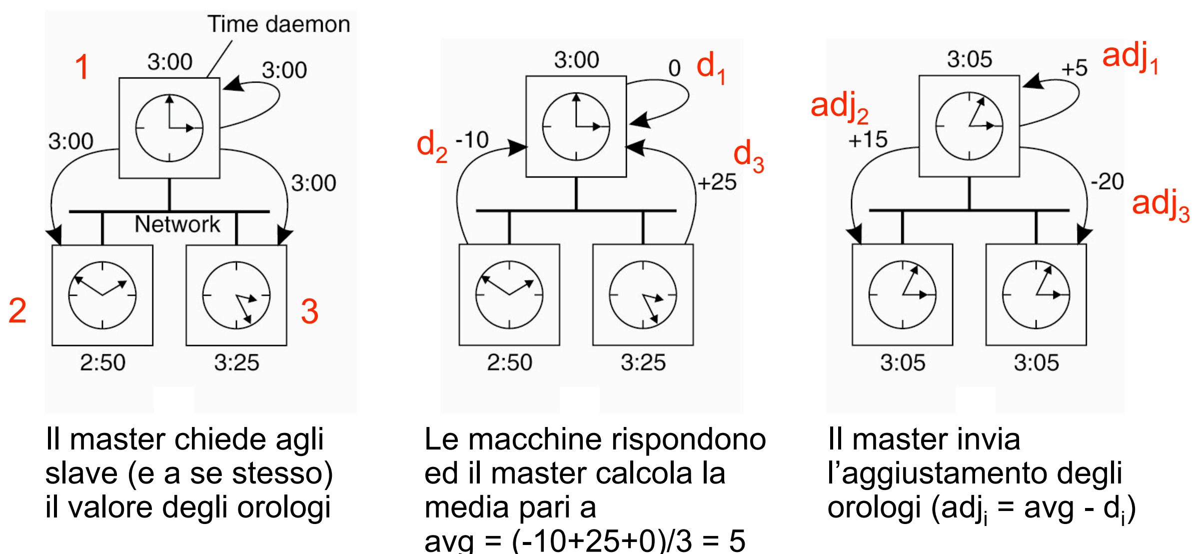
Algoritmo di Berkeley (2)

- L'accuratezza dell'algoritmo dipende da un RTT nominale massimo: il master non considera valori di clock associati a RTT superiori al massimo
- Tolleranza ai guasti
  - Se il master cade, un'altra macchina viene eletta master
  - Tollerante a comportamenti arbitrari (slave che inviano valori errati di clock)
    - Il master prende un certo numero valori di clock (da un sottoinsieme di slave); questi valori non differiscono tra loro per una quantità specificata

Algoritmo di Berkeley (3)

- Che cosa significa rallentare un clock?
- Non è possibile imporre un valore di tempo passato agli slave che si trovano con un valore di clock superiore a quello calcolato come clock sincrono
  – Ciò provocherebbe un problema di ordinamento causa/effetto di eventi e verrebbe violata la condizione di monotonicità del tempo
- La soluzione consiste nel mascherare una serie di interrupt che fanno avanzare il clock locale in modo da rallentare l'avanzata del clock stesso
  – Il numero di interrupt mascherati è pari al tempo di slowdown diviso il periodo di interrupt del processore

Algoritmo di Berkeley: esempio



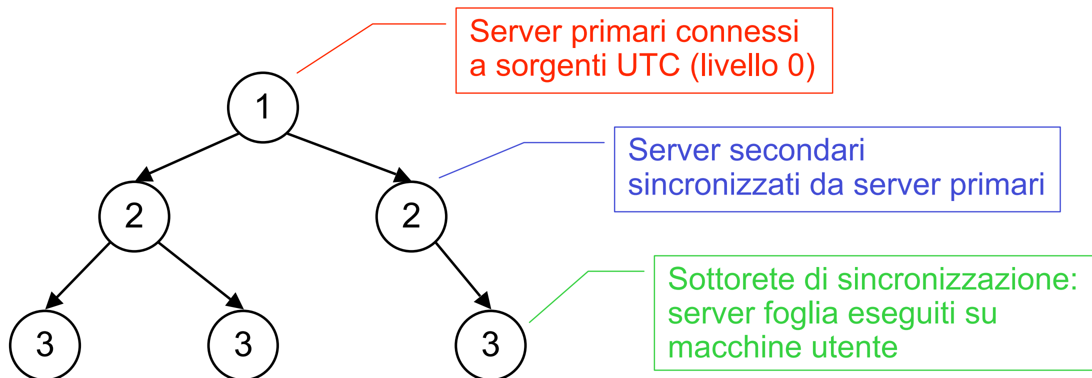
Il master chiede agli slave (e a se stesso) il valore degli orologi

Le macchine rispondono ed il master calcola la media pari a avg = (-10+25+0)/3 = 5

Il master invia l'aggiustamento degli orologi (adj_i = avg - d_i)

Network Time Protocol (NTP)

- Time service per Internet
  - Sincronizzazione esterna accurata rispetto a UTC
- Architettura di service time disponibile e scalabile
  - Server e path ridondanti
- Autenticazione delle sorgenti di tempo

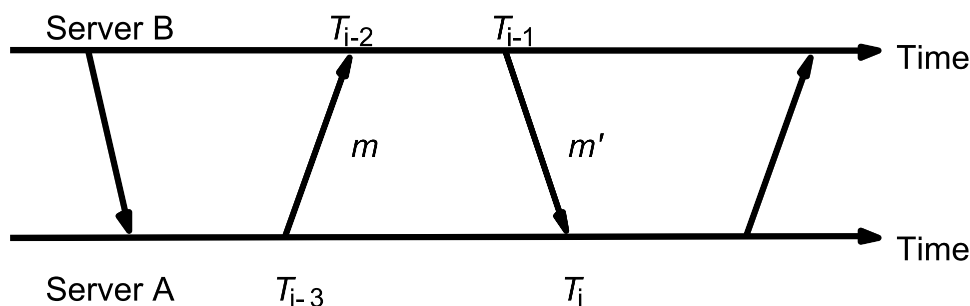


NTP: sincronizzazione

- La sottorete di sincronizzazione si riconfigura in caso di guasti
  - Server primario che perde la connessione alla sorgente UTC diventa server secondario
  - Server secondario che perde la connessione al suo primario (ad es. crash del primario) può usare un altro primario
- Modi di sincronizzazione
  - Multicast: server all'interno di LAN ad alta velocità invia in multicast il suo tempo agli altri che impostano il tempo ricevuto assumendo un certo ritardo di trasmissione
    - Non molto accurato
  - Procedure call: un server accetta richieste da altri computer (come algoritmo di Cristian)
    - Accuratezza alta
    - Utile se non è disponibile multicast supportato in hardware
  - Simmetrico: coppie di server scambiano messaggi contenenti informazioni sul timing
    - Accuratezza molto alta (per livelli alti della gerarchia)
- Tutti i modi di sincronizzazione usano UDP

NTP: sincronizzazione (2)

- Ogni messaggio contiene timestamp di eventi recenti
  - Tempi locali di *Send* e *Receive* del messaggio precedente *m*
  - Tempo locale di *Send* del messaggio corrente *m'*
- Il ricevente segna il tempo locale T_i in cui riceve il messaggio *m'*
- Nel modo simmetrico il ritardo tra l'arrivo di un messaggio e l'invio del successivo può essere non trascurabile



NTP: accuratezza

- Per ogni coppia di messaggi *m* ed *m'* scambiati tra due server, NTP stima un offset relativo o_i tra i 2 clock ed un ritardo d_i (tempo di trasmissione totale per *m* ed *m'*)
- Supponendo che il vero offset del clock di B rispetto ad A sia *o* e che i tempi di trasmissione di *m* ed *m'* siano rispettivamente *t* e *t'*

  $T_{i-2} = T_{i-3} + t + o$ e $T_i = T_{i-1} + t' - o$
- Il tempo totale di trasmissione dei messaggi è:

  $d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$
- Sottraendo le equazioni:

  $o = o_i + (t' - t)/2$ con $o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$
- Considerando che t, $t'>0$ si può dimostrare che

  $o_i - d_i/2 \le o \le o_i + d_i/2$
- o_i è una stima dell'offset e d_i è una misura dell'accuratezza della stima
- I server NTP filtrano le coppie $<o_i, d_i>$, stimano l'affidabilità dei dati dalla differenza con la stima e selezionano il peer che usano per sincronizzarsi scegliendo o_i corrispondente al minimo d_i
- Accuratezza di 10 ms su Internet (1 ms su LAN)

Tempo in un sistema asincrono

- Gli algoritmi per la sincronizzazione dei clock fisici si basano sulla stima dei tempi di trasmissione
- Nei sistemi reali non sempre i tempi di trasmissione sono predicibili
  - Viene a mancare l'accuratezza della sincronizzazione
- L'assenza di sincronizzazione implica l'impossibilità di ordinare gli eventi appartenenti a processi diversi usando il tempo
  - Il tempo di due eventi che accadono in processi diversi non può essere generalmente utilizzato per decidere quando un evento precede l'altro

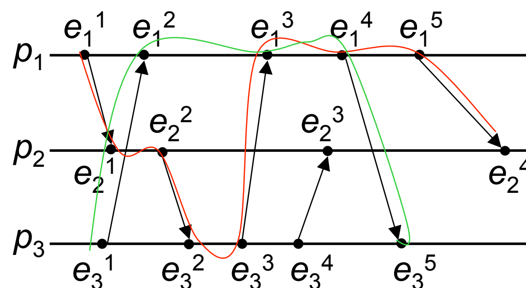
Tempo logico

- 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 (1978) introduce il concetto di relazione di *happened-before* (anche detta relazione di *precedenza* o *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

- Due eventi *e* ed *e'* sono in relazione di happened-before (indicata con $e \rightarrow e'$) se:
  1. $\exists\, p_i \mid e \rightarrow_i e'$
  2. $\forall$ messaggio m, $send(m) \rightarrow receive(m)$
     - $send(m)$ è l'evento di invio del messaggio m
     - $receive(m)$ è l'evento di ricezione del messaggio m
  3. $\exists\, e, e', e'' \mid (e \rightarrow e'') \wedge (e'' \rightarrow e')$
     - La relazione happened-before è transitiva
- Applicando le tre regole è possibile costruire una sequenza di eventi $e_1, e_2, \ldots, e_n$ causalmente ordinati
- *Osservazioni*
  - La relazione happened-before rappresenta un ordinamento parziale non riflessivo
  - Non è detto che la sequenza $e_1, e_2, \ldots, e_n$ sia unica
  - Data una coppia di eventi, questa non è sempre legata da una relazione happened-before; in questo caso si dice che gli eventi sono *concorrenti* (indicato da ||)

Relazione happened-before: esempio



- Sequenza $s_1 = e_1^1, e_2^1, e_2^2, e_3^2, e_3^3, e_1^3, e_1^4, e_1^5, e_2^4$
- Sequenza $s_2 = e_3^1, e_1^2, e_1^3, e_1^4, e_3^5$

- Gli eventi e_3^1 e e_2^1 sono concorrenti

Clock logico scalare

- Il clock logico 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*:
  - Si ottiene un ordinamento parziale: non è detto che guardando i timestamp di due eventi si riesca a capire in che relazione sono
  - Se $L(e) < L(e')$ non è detto che $e \rightarrow e'$

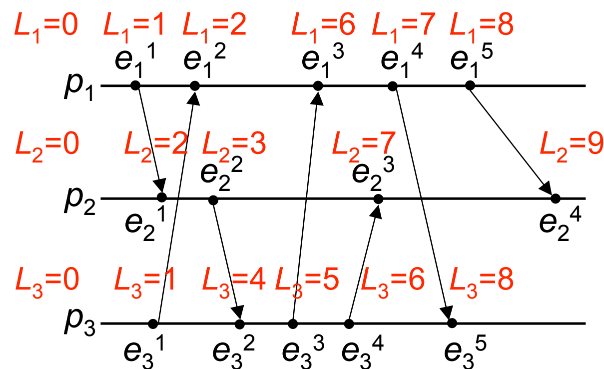
Clock logico scalare: implementazione

- Algoritmo di Lamport
- Ogni processo p_i inizializza il proprio clock logico L_i a 0 ($\forall\ i = 1, \ldots, N$)
- L_i è incrementato di 1 prima che il processo p_i esegua l'evento (evento interno o evento esterno sia *send* che *receive*)

  $$L_i = L_i + 1$$

- Quando p_i invia il messaggio m a p_j
  - Incrementa il valore di L_i
  - Allega al messaggio m il timestamp $t = L_i$
  - Esegue l'evento *send*(m)
- Quando p_j riceve il messaggio m con timestamp t
  - Aggiorna il proprio clock logico $L_j = max(t, L_j)$
  - Incrementa il valore di L_j
  - Esegue l'evento *receive*(m)

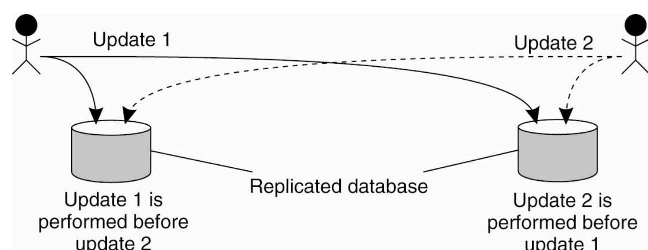
Clock logico scalare: esempio



- Osservazioni
  - $e_1^1 \rightarrow e_2^1$ ed i relativi timestamp riflettono questa proprietà (L_1=1 e L_2=2)
  - e_1^1 || e_3^1 ed i relativi timestamp sono uguali (L_1=1 e L_3=1)
  - e_2^1 || e_3^1 ed i relativi timestamp sono diversi (L_2=2 e L_3=1)

Esempio: multicasting totalmente ordinato

- Come garantire che aggiornamenti concorrenti su un database replicato siano visti nello stesso ordine da ogni replica?
  - p_1 aggiunge $100 ad un conto (valore iniziale: $1000)
  - p_2 incrementa il conto dell'1%
  - Ci sono due repliche
  - In assenza di un'opportuna sincronizzazione le due operazioni non vengono eseguite sulle due repliche nello stesso ordine
    - Replica #1 ← 1111 (prima p_1 e poi p_2)
    - Replica #2 ← 1110 (prima p_2 e poi p_1)



Esempio: multicasting totalmente ordinato (2)

- Applichiamo gli orologi logici di Lamport per risolvere il problema
  - Multicast totalmente ordinato: operazione di multicast con cui tutti i messaggi sono consegnati nello stesso ordine ad ogni destinatario
- p_i invia il messaggio msg_i (con timestamp basato sul clock logico scalare) a tutti gli altri, incluso se stesso
- Il messaggio ricevuto viene posto in una coda locale $queue_j$ ordinata in base al valore del timestamp
- Ogni messaggio in arrivo a p_j viene accodato in $queue_j$ e p_j invia in multicast un messaggio di conferma agli altri processi

Esempio: multicasting totalmente ordinato (3)

- p_j consegna msg_i alla sua applicazione se:
  1. msg_i è in testa a $queue_j$ e gli ack per esso inviati sono stati ricevuti
  2. Per ogni processo p_k, c'è un messaggio msg_k in $queue_j$ *con* un timestamp maggiore di quello di msg_i
  - Ovvero solo quando p_j sa che nessun altro processo può inviare in multicast un messaggio con timestamp minore o uguale a quello di msg_i

- Osservazione: si assume che la comunicazione sia affidabile e FIFO ordered
  - FIFO ordered: messaggi inviati da p_i a p_j sono ricevuti da p_j nello stesso ordine in cui p_i li ha inviati

Problema del clock logico scalare

- 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 Fidge (1991) propongono di ovviare al problema considerando anche l'identificativo del processo in cui l'evento occorre
  - Introduzione dei clock logici vettoriali

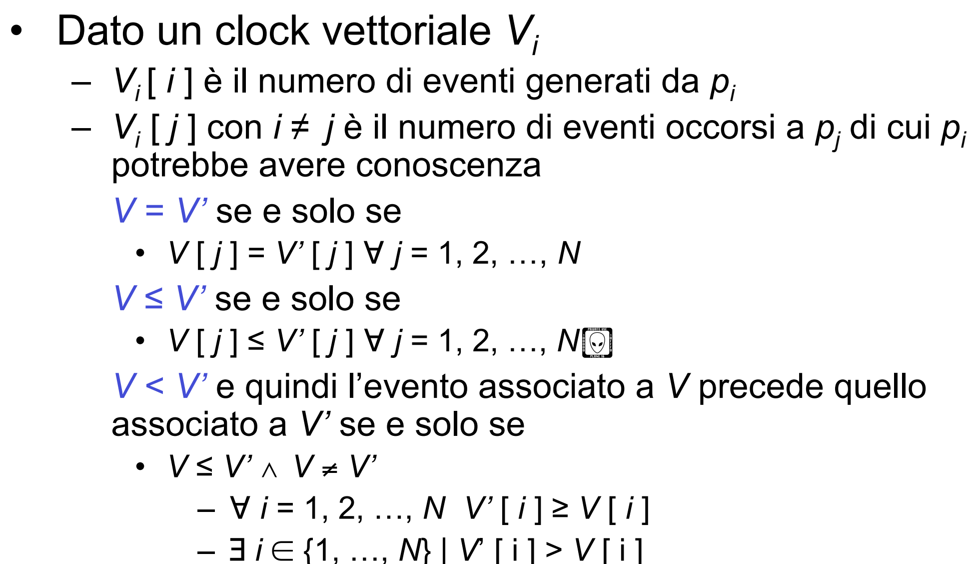
Clock logico vettoriale

- Il clock logico vettoriale per un sistema di N processi è dato da un vettore di N interi
- Ciascun processo p_i mantiene il proprio clock vettoriale V_i
- Per il processo p_i $V_i[\,i\,]$ è il clock logico locale
- Ciascun processo usa il suo clock vettoriale per assegnare il timestamp agli eventi
- Analogamente al clock scalare di Lamport, il clock vettoriale viene allegato al messaggio *m* ed il timestamp diviene vettoriale
- Con i clock vettoriale si catturano completamente le caratteristiche della relazione happended-before

Clock logico vettoriale: implementazione

- Ogni processo p_i inizializza il proprio clock vettoriale V_i
  $V_i[j]=0 \quad \forall j = 1, 2, \ldots, N$
- Prima di eseguire un evento (interno o esterno), p_i incrementa la sua componente del clock vettoriale
  $V_i[i] = V_i[i] + 1$
- Quando p_i invia il messaggio m a p_j
  - Incrementa il valore di V_i per la sua componente
  - Allega al messaggio m il timestamp $t = V_i$
  - Esegue l'evento *send*(m)
- Quando p_j riceve il messaggio m con timestamp t
  - Aggiorna il proprio clock logico $V_j[j] = max(t[j], V_j[j]) \; \forall j = 1, 2, \ldots, N$
  - Incrementa il valore di V_j per la sua componente
  - Esegue l'evento *receive*(m)

Clock logico vettoriale (2)



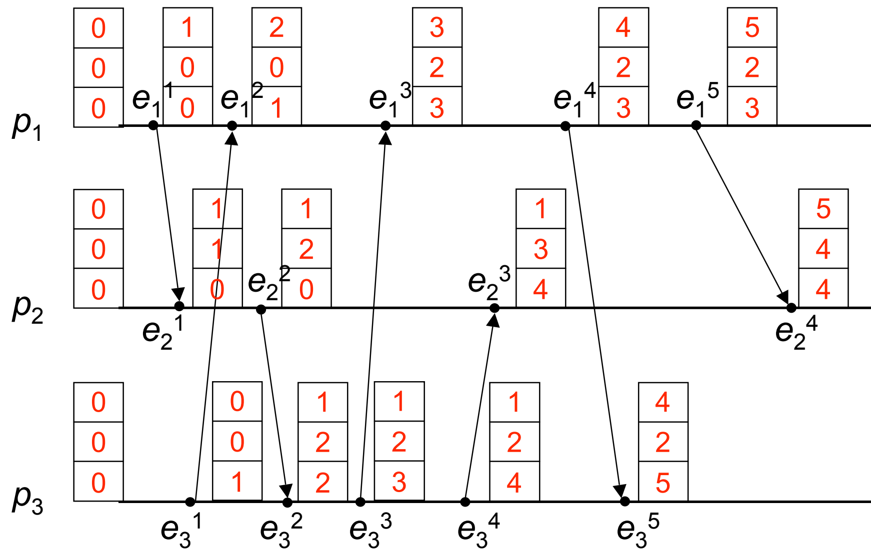
- Dato un clock vettoriale V_i
  - $V_i[i]$ è il numero di eventi generati da p_i
  - $V_i[j]$ con $i \neq j$ è il numero di eventi occorsi a p_j di cui p_i potrebbe avere conoscenza

    $V = V'$ se e solo se
    - $V[j] = V'[j] \; \forall j = 1, 2, \ldots, N$

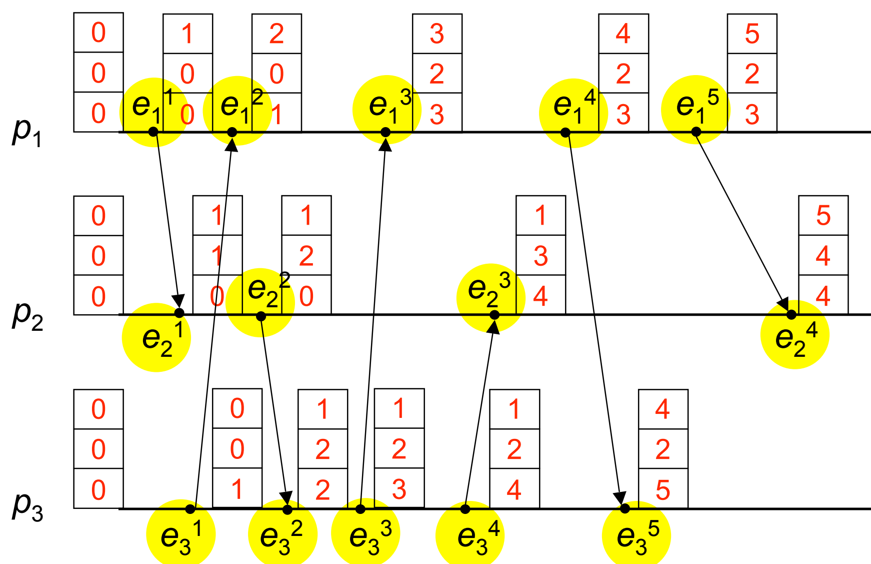
    $V \leq V'$ se e solo se
    - $V[j] \leq V'[j] \; \forall j = 1, 2, \ldots, N$

    $V < V'$ e quindi l'evento associato a V precede quello associato a V' se e solo se
    - $V \leq V' \wedge V \neq V'$
      - $\forall i = 1, 2, \ldots, N \quad V'[i] \geq V[i]$
      - $\exists i \in \{1, \ldots, N\} \mid V'[i] > V[i]$

Clock logico vettoriale: esempio



Clock logico vettoriale: esempio



Confronto di clock vettoriali

- Esaminando i timestamp si riesce a capire se due eventi sono concorrenti o se sono in relazione happened-before

V	V'
1	1
2	2
0	2

$V(e) < V'(e)$ e quindi $e \rightarrow e'$

V	V'
1	1
2	0
0	2

$V(e) \neq V'(e)$ e quindi $e \parallel e'$

Clock logico matriciale

- Passando da clock logico scalare a clock logico vettoriale aumenta la quantità di “conoscenza” che i processi si scambiano
- Ogni volta che un processo riceve un messaggio, aumenta il proprio livello di “conoscenza” integrando le sue informazioni locali con quelle ricevute da un altro processo
- Con l’introduzione dei clock logici matriciali ogni processo mantiene informazioni sulla conoscenza che ogni altro processo ha dello stato globale del sistema