LOGICAL TIME IN DISTRIBUTED SYSTEMS

In un sistema distribuito è impossibile avere un unico clock fisico condiviso da tutti i processo.
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 risalire a questo tempo è di vitale importanza poter stabilire quale evento è stato generato prima di un altro.

Soluzione 1:

- Si può decidere di sincronizzare con una certa approssimazione i clock fisici locali attraverso opportuni algoritmi.
- Ogni processo allega un timestamp contenente il valore del proprio clock fisico ad ogni messaggio che invia.
- I messaggi vengono così ordinati in funzione di un timestamp crescente.

Questo meccanismo funziona solo se l'errore di approssimazione dei clock fisici è limitato, altrimenti si rischia di alterare l'ordine reale degli eventi.
Ma l'approssimazione dei clock non sempre si può mantenere limitata: ad esempio in un modello asincrono NO (si dice che un sistema distribuito è asincrono quando non è possibile stabilire un upper bound al tempo di esecuzione di ciascun step di un processo, al tempo di propagazione di un messaggio di rete, alla velocità di deriva di un clock).
Così in un modello asincrono il timestamping non si può basare sul concetto di tempo fisico.

Soluzione 2:

- Il timestamping avviene etichettando gli eventi con il valore corrente di una variabile opportunamente aggiornata durante la computazione.
- Questa variabile, poiché completamente scollegata dal comportamento del clock fisico locale, è chiamata clock logico.

Questa soluzione, non basandosi sul concetto di tempo reale, è adatta ad un uso in sistemi asincroni.

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 software clock : $Ci(t)= \alpha Hi(t) + \beta$ che approssimativamente misura l'istante di tempo fisico t per il processo P_i.

Il cok hardware $H_i(t)$ è un componente fisico (oscillatore al quarzo) caratterizzato da un parametro detto DRIFT RATE:

- L'oscillatore è caratterizzato da una sua frequenza che fa divergere il clock hardware dal tempo reale.
- Il drift rate misura il disallineamento del clock hardware da un orologio ideale per unità di tempo.

Un clock hw $H_i(t)$ è corretto se il suo drift rate si mantiene all'interno di un limite $\rho > 0$ finito. (es. 10^{-6} secs/sec).

Se il clock $H_i(t)$ è corretto allora l'errore che si commette nel misurare un intervallo di istanti reali [t,t'] è limitato: $(1 - \rho) (t' - t) \leq H_i(t') – H_i(t) \leq (1 + \rho) (t' - t)$ supposto $(t<t')$

Per il clock software $C_i(t)$ spesso basta una condizione di monotonicità: t' > t implica $C_i(t') > C_i(t)$.
Si potrebbe garantire monotonicità con un clock hw non corretto scegliendo opportunamente i valori α e β.
La risoluzione del clock è il periodo che intercorre tra gli aggiornamenti del valore del clock.
Ci chiediamo quanto deve essere la risoluzione per distinguere due differenti eventi.
NB il tempo di risoluzione < intervallo di tempo che intercorre tra due eventi rilevanti.

Clock guasto: se non rispetta le condizioni di correttezza. NB clock corretto ≠ clock accurato.

L'UTC (Coordinated Universal Time) è uno standard internazionale per mantenere il tempo.
E' basato su orologi atomici ma occasionalmente aggiustato utilizzando il tempo astronomico.
L'output dell'orologio astronomico è inviato broadcast da stazioni radio su terra e satelliti.
Computer con ricevitori possono sincronizzare i loro clock con questi segnali.

SINCRONIZZAZIONE DEI CLOCK FISICI
Possiamo sincronizzare i clock di processi appartenenti a sistemi distribuiti attraverso due differenti strategie:

- Sincronizzazione esterna: i clock sono sincronizzati con una sorgente di tempo S, in modo che, dato un intervallo I di tempo reale: $|S(t) - Ci(t)| < D$ per i = 1,…,N e per tutti gli istanti t in I, cioè i clock Ci hanno un'accuratezza compresa nell'intervallo D.
- Sincronizzazione interna: i clock di due computer sono sincronizzati l'uno con l'altro in modo che $|S(t) - Ci(t)| < D$ per i = 1,…,N nell'intervallo I. In questo caso i due clock Ci e Cj si accordano all'interno dell'intervallo D.

I clock sincronizzati internamente non sono necessariamente esternamente sincronizzati. Tutti i clock possono deviare collettivamente da una sorgente esterna sebbene rimangano sincronizzati tra loro entro l'intervallo D.
Se l'insieme dei processi è sincronizzato esternamente entro un intervallo D allora segue che è anche internamente sincronizzato entro un intervallo 2D.

TIME SERVICES
Il gruppo di processi che deve sincronizzarsi fa uso di un Time Service. Il Time Service può essere a sua volta implementato da un solo processo (server) oppure può essere implementato in modo decentralizzato da più processi:
Time Service centralizzato:

- Request-driven (algoritmo di Cristian) – sync esterna
- Broadcast-based (Berkeley Unix algorithm) – sync interna

Time Service decentralizzato:

- Network Time Protocol – sync esterna

ALGORITMO DI CRISTIAN
Time server centralizzato e passivo:

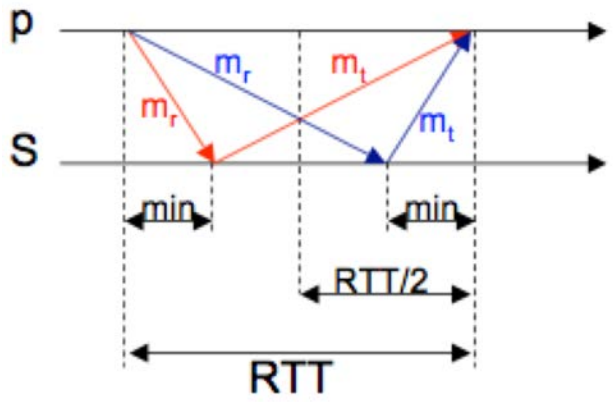
- Il time server S riceve il segnale da una sorgente UTC.
- Un processo P richiede il tempo con m_r e riceve t in m_t da S.
- P imposta il suo clock a $t+ RTT/2$.

NB RTT è il round trip time misurato tra P e S, si divide con il 2 perché a metà viaggio verrà inviato il valore di t, quindi il viaggio della richiesta viene ignorato.

Accuratezza nell'algoritmo di Cristian:

- Caso 1: il tempo impiegato dal messaggio di ritorno è maggiore rispetto alla sima fatta utilizzando RTT/2 ed in particolare è uguale a RTT-min (dove min è il tratto di andata in questo caso, cioè quello più breve).
  Δ = stima del tempo del messaggio di ritorno - tempo reale = (RTT/2) - (RTT - min) = -RTT/2 +min = - (RTT/2 - min)

- Caso 2: il tempo impiegato dal messaggio di ritorno è minore rispetto alla stima fatta utilizzando RTT/2 ed in particolare è uguale a min (essendo il ritorno in questo caso il tratto più breve).
  Δ = stima del tempo del messaggio di ritorno - tempo reale = (RTT/2) - min = + (RTT/2 - min)



Il tempo di S quando m_t arriva a P è compreso nell'intervallo [t+min, t+RTT-min].
L'ampiezza di tale intervallo è RTT – 2min.
Accuratezza ≤ ± (RTT/2 – min).

ALGORITMO DI BERKLEY
Algoritmo per la sincronizzazione interna di un gruppo di computer:

- Il time server è centralizzato e attivo (master).
- Il master invia a tutti gli altri processi (slaves) il proprio valore locale e richiede gli scostamenti dei loro clock da questo valore.
- Riceve tutti gli scostamenti e usa il RTT per stimare il valore del clock di ciascun slave.
- Calcola il valor medio.
- Invia a tutti gli slaves gli scostamenti necessari per sincronizzare i clock.

Esempio:
Supponiamo un master con il clock a 700. 4 slave con i clock rispettivamente a 690, 705, 695, 715.
Il master invia 700 a tutti.
Gli slaves rispondo con la differenza (slave – master): -10, +5, -5, +15.
Il master ne fa la somma (compreso il suo ritardo) e divide per il numero di processi(compreso lui): (0-10+5-5+15)/5 = 1.
Ora il master invia a ogni processo quanto deve sommare algebricamente al proprio clock per sincronizzarsi, ossia rispettivamente 11, -4, +6, -14, e lui si somma 1.
L'accuratezza dipende dalla variabilità del RTT dei vari canali che collegano slaves e master.

Guasti: se un master va in crash un'altra macchina viene eletta master, è tollerante a comportamenti arbitrari cioè se uno slave invia un valore di clock corrotto il master non lo considera nella somma poiché è impostato a considerare valori in un certo intervallo prefissato.

NB non si può pensare di imporre un valore di tempo passato ai processi slave con un valore di clock superiore a quello calcolato come valore di clock sincrono:

- Ciò provocherebbe un problema di ordinamento causa/effetto di eventi e verrebbe violata la condizione di monotonicità del tempo.
- La soluzione è quella di mascherare una serie di interrupt che fanno avanzare il clock locale in modo da rallentare l'avanzata del clock stesso.

NETWORK TIME PROTOCOL
Sincronizza client a UTC. Architettura: disponibile e scalabile poiché vengono usati server multipli e path ridondanti.
Le sorgenti di tempo sono autentiche.

La sottorete di sincronizzazione (tutti i livelli che includono i server) si riconfigura in caso di guasti:

- Un primary che perde la connessione alla sorgente UTC può diventare un server secondario.
- Un secondario che perde la connessione al suo primary (crash di questo) può usare un altro primary.

Modalità di sincronizzazione (sempre via UDP):

- Multicast: un server manda in multicast il suo tempo agli altri che impostano il tempo ricevuto assumendo un certo ritardo prefissato. Non molto accurato in mancanza di multicast hardware, poiché passa del tempo da un invio all'altro.
- Procedure call: un server accetta richieste da altri computer (come algoritmo di Cristian). Alta accuratezza. Utile se non si dispone di multicast hw.
- Simmetrico: coppie di server scambiano messaggi contenenti informazioni sul timing. Usata quando è necessaria un'accuratezza molto alta, ossia per gli alti livelli di gerarchia.

Con il metodo simmetrico bisogna tenere in considerazione i ritardi dei canali:
Per ogni coppia di messaggi scambiati tra i due server, NTP stima un offset O (o grande) tra i 2 clock ed un ritardo D (ritardo di trasmissione per i 2 msg).
Immaginiamo che o (o piccolo) sia il vero offset tra i due clock.
$t2 = t1 + d1 + o$ e $t4 = t3 + d2 – o$ dove o (o piccolo) è il vero offset, t1 il tempo di invio del primo messaggio, t3 quello del secondo, t2 il tempo di arrivo del primo messaggio, t4 del secondo, d1 e d2 i tempi di propagazione.
Da cui $D = d1 + d2 = (t2-t1) + (t4-t3)$

Se sottraggo $t2 – t4 = (t1 + d1 + o) – (t3+d2-o) = t1 + d1 – t3 –d2 + 2o$
Da cui $o=((t2-t1) + (t3-t4)) / 2 + (d2-d1) / 2$
Siccome il primo membro dell'addizione di o è proprio O posso dire che $o = O + (d2 – d1)/2$
L'offset stimato O ha quindi un errore massimo pari a $\pm (d2 – d1)/2$.

TEMPO LOGICO

In alcune applicazione potrebbe interessare solo la sequenza degli eventi e non l'istante preciso. Per cui si fa riferimento al tempo logico.

Nel caso in cui appunto non si può sincronizzare i clock dei processi possiamo ordinare i messaggi sull'assunzione di due fatti:

- Due eventi che accadono sullo stesso processo possono sempre essere ordinati
- Un evento di ricezione di un messaggio segue sempre l'evento di invio del messaggio stesso

Gli eventi possono essere quindi ordinati secondo una nozione di causa-effetto.

Lamport introdusse la relazione che cattura le dipendenze causali tra gli eventi:

- Denotiamo con $->_i$ la relazione di ordinamento tra eventi in un processo Pi
- Denotiamo con $->$ la relazione "avvenuto-prima" tra eventi pari

NB Due eventi avvenuti in un processo Pi accadono nello stesso ordine in cui il processo li osserva, quando Pi invia un messaggio a Pj l'evento send avviene prime dell'evento receive.

DEFINIZIONE DELLA RELAZIONE AVVENUTO-PRIMA

Due eventi e ed e' sono in relazione attraverso una relazione avvenuto-prima (e -> e') se:

- Esiste un processo Pi tale che e ->i e'
- Comunque scegli un messaggio m $e_{send(m)} -> e_{receive(m)}$ (send(m) è l'evento di invio del messaggio m e receive(m) è l'evento di ricezione dello stesso messaggio)
- Esiste e'', e''' tale che (e -> e'') Λ (e'' -> e''') (questa relazione è transitiva)

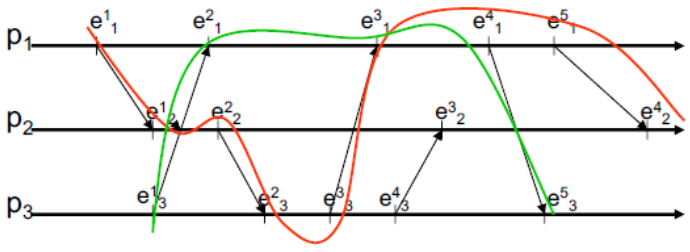
Usando queste tre regole è possibile definire una sequenza ordinata causalmente degli eventi e1, e2, …, en

NB

La sequenza e1, e2, …, en può non essere unica, potrebbe esistere una coppia <e1, e2> che non sono in relazione avvenuto-prima. Essi si dicono concomitanti (e1 || e2).

In un sistema distribuito per ogni coppia di eventi <e1, e2> deve poter essere vera una di queste condizioni:

1. e1 -> e2
2. e2 -> e1
3. e1 || e2



e^j_i is j-th event of process p_i

$S_1 = < e^1_1, e^1_2, e^2_2, e^2_3, e^3_3, e^3_1, e^4_1, e^5_1, e^4_2 >$

$S_2 = < e^1_3, e^2_1, e^3_1, e^4_1, e^5_3 >$

Note: e^1_3 and e^1_2 are concurrent

Questo esempio mostra come individuare una sequenza di relazioni happened-before.

LOGICAL CLOCK

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 Pi ha il proprio clock logico Li, usato per applicare i timestamp agli eventi.

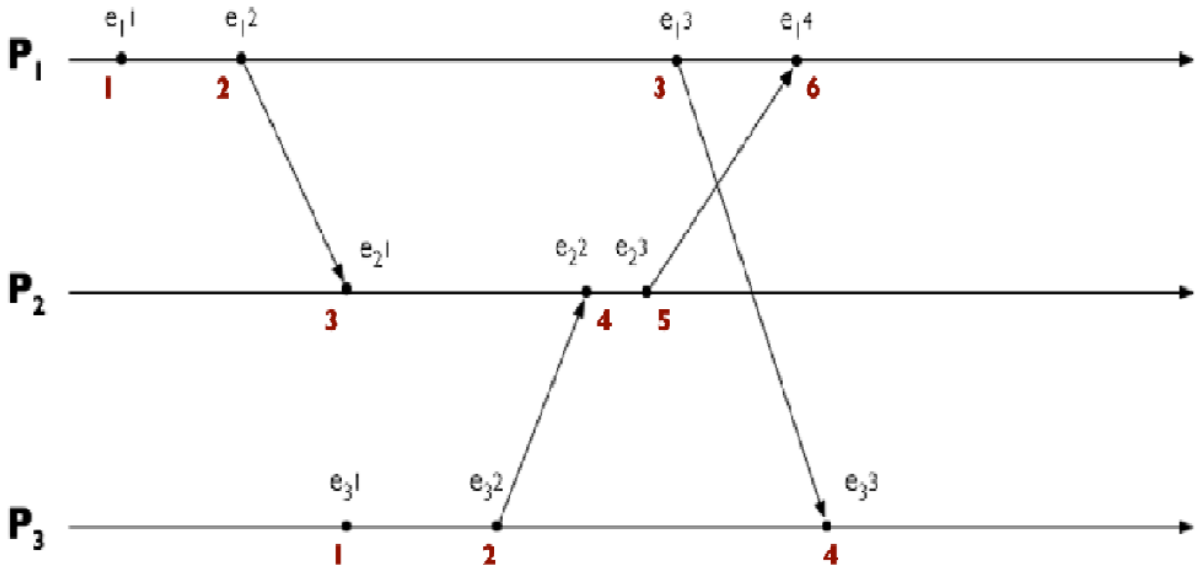
Denotiamo con Li(e) il timestamp, basato sul clock logico, applicato dal processo Pi all'evento e.

Proprietà: Se e -> 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, cioè se L(e) < L(e') non è detto che e -> e'.

Un'implementazione:

- Ogni processo Pi inizializza il proprio clock logico Li a 0.
- Li è incrementato di 1 prima che il processo Pi esegua l'evento(interno o esterno che sia). Li = Li + 1.
- Quando Pi invia il messaggio m a Pj:
    1. Incrementa il valore di Li
    2. Allega al messaggio m il timestamp t = Li
    3. Esegue l'evento send(m)
- Quando Pj riceve il messaggio m con timestamp t
    1. Aggiorna il proprio clock logico Lj = max(t,Lj)
    2. Incrementa il valore di Lj
    3. Esegue l'evento receive(m)



NB e_i^j significa evento j-esimo del processo i-esimo.

Possiamo notare come e_1^4 e e_3^3 non sono in relazione happened-before, sono quindi concomitanti e hanno valore di timestamp diverso.

Mentre e_1^1 e e_3^1 sono concomitanti e hanno timestamp uguale.

Limiti del clock logico scalare:

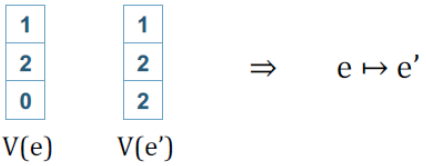
- Garantisce questa proprietà: se e -> e' allora L(e) < L(e')
- Ma non è possibile garantire questa: if L(e) < L(e') allora e -> e'
- Di conseguenza non è possibile determinare, analizzando sono il clock scalare, se due eventi sono concorrenti o correlati da una relazione happened-before.

Si passa quindi al clock vettoriale.

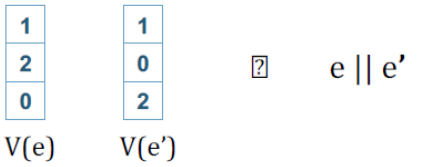
CLOCK VETTORIALE
Ad ogni evento e viene assegnato un vettore V(e) di dimensione pari al numero dei processi con la seguente proprietà: e -> e' ⇔ V(e) < V(e').

Che significato diamo al segno < tra due vettori?
V(e) < V(e') se e solo se:
comunque prendi x appartenente a {1, …, N} V(e')[x] >= V(e)[x] e esiste un x appartenente a {1,…,N} tale che V(e')[x] > V(e)[x]. (Cioè tutti gli elementi di V(e') devono essere maggiori o uguali ma comunque sia uno deve per forza maggiore!).

Comparare i valori di due clock vettoriali associati a due eventi distinti permette di capire la relazione che lega i due eventi:



$\Rightarrow \quad e \mapsto e'$



? $\quad e \,||\, e'$

Ogni processo Pi gestisce un vettore di interi Vi di n componenti Vi[1…n] ovvero una per ogni processo.
La componente Vi[x] indica la stima che il processo Pi fa sul numero di eventi eseguiti dal processo Px.
Il vettore è inizializzato a [-,-,…,0,…,-], 0 all'i-sema posizione, cioè quella di ogni processo.
Vi[i] rappresenta infatti il logical clock di Pi.

Il vettore viene aggiornato in base a queste regole:

- Quando Pi esegue un evento incrementa Vi[i] di un unità e poi associa un timestamp T all'evento il cui valore è pari al valore corrente di Vi.
- Quando Pi esegue un evento di invio messaggio, allega al messaggio il timestamp di quell'evento ottenuto dalla regola precedente.
- Quando arriva un messaggio a Pi con un timestamp T Pi esegue la seguente operazione: comunque preso x appartenente a [1…n] Vi[x] = max(vi[x],Tx]) ossia prendo il massimo tra le due componenti dei due vettori.
- Quindi esegue l'evento di consegna ed esegue ovviamente la prima regola incrementando il proprio logical clock.

RICART-AGRAWALA ALGORITHM (mutual exclusion in distributed setting)
I clock scalari possono suggerire una soluzione per la mutua esclusione in un setting distribuito:
Se semplicemente adattiamo l'algoritmo di Lamport potremmo ottenere una soluzione inefficiente, poiché i processi agiscono indipendentemente (senza coordinamento) cercando di accedere alla sezione critica.
Intuizione dietro l'algoritmo RA:

- Ogni processo entra nella doorway suggerendo un numero.
- Poi il processo invia a tutti gli altri processi il numero aspettando che concedano l'accesso alla critical section.

RA Algorithm

Assunzioni: processi non falliscono, messaggi non si perdono, latenza del canale finita.

Variabili locali:

- #replies (inizialmente 0)
- State che può assumere {Requesting, CS, NCS} (inizialmente NCS)
- Q coda delle richieste in attesa (inizialmente vuota)
- Last_Req (inizialmente MAX_INT)
- Num (inizialmente 0)

Repeat

1. State = Requesting
2. Num = Num + 1; Last_Req = Num
3. for i from 1 to N send REQUEST(Last_Req) to Pi
4. Wait until #replies == N – 1
5. State = CS
6. CS
7. Comunque prendo elem in coda Q send REPLY to elem
   Q = insieme vuoto
   State = NCS
   #replies = 0
   Last_Req = MAX_INT

NB la linea 2 è eseguita atomicamente.

Se si riceve una REQUEST(t) from Pj

8. Num = max(t, Num)
9. If State == CS or (State == Requesting and (Last_Req, i) < (t, j))
10. Then insert (t,j) into Q
11. Else send REPLY to Pj

Se si riceve una REPLY from Pj

12. #replies = #replies + 1

Il senso dell'algoritmo è: se ho bisogno di entrare in CS mi metto in Requesting, aumento il mio num (token) e invio a tutti gli altri processi tranne me la richiesta d'accesso. Se ricevo N-1 REPLY semplicemente vuol dire che o i processi non sono interessati o ho priorità maggiore io dettata da un valore di Last_Req minore (o nel caso uguale, ho un indice minore).
In quel caso metto in coda le richieste di altri processi in modo tale che si blocchino non ricevendo il mio REPLY e entro in CS. Una volta uscito completo l'invio di tutti i REPLY ai processi in coda che aspettano per entrare in CS e mi resetto le var locali tranne NUM, serve come numeretto che si prende dal panettiere quindi va sempre incrementato. Se invece sono interessato ad entrare in CS ma aspettando i REPLY dagli processi mi arriva una richiesta da un altro processo che ha priorità maggiore rispetto alla mia allora sono costretto a inviargli un REPLY e a sbattere la testa in busy waiting sul while #replies != N-1.