

Sincronizzazione nei Sistemi Distribuiti

Corso di Sistemi Distribuiti e Cloud Computing A.A. 2024/25

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Il tempo nei SD

- In un SD, i **processi**
 - sono eseguiti su nodi connessi in rete
 - cooperano per portare a termine una computazione
 - comunicano tramite scambio di messaggi
- Osservazioni:
 - Algoritmi richiedono **sincronizzazione**
 - I processi in esecuzione su diversi nodi del SD devono avere una **nozione comune di tempo** per poter effettuare azioni sincronizzate rispetto al tempo
 - Algoritmi richiedono **ordinamento degli eventi**, ovvero di stabilire quale evento è stato generato prima di un altro
- Nei SD **il tempo è importante ma problematico**

Il tempo nei SD

- In un **sistema centralizzato** è possibile stabilire l'ordine in cui gli eventi si sono verificati
 - Memoria comune e clock unico
- 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 in cui gli eventi sono stati generati
- Per numerosi algoritmi distribuiti è di vitale importanza risalire a questo tempo, o comunque è importante stabilire l'**ordinamento degli eventi**
- **Come?**

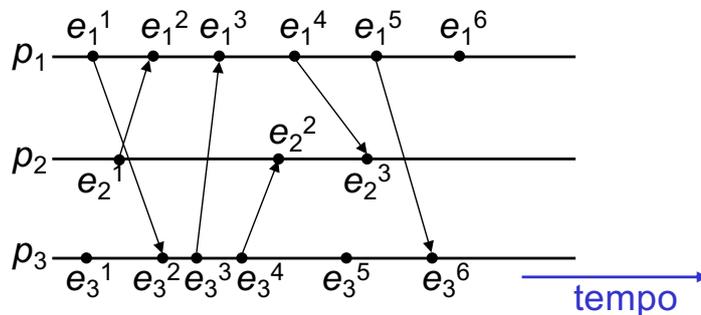
Il tempo nei SD: soluzioni

- Soluzione 1: **sincronizzare gli orologi fisici**
 - Orologio fisico: conta il numero di secondi
 - Ogni nodo del SD aggiusta il valore del suo clock fisico in modo coerente con quello degli altri nodi o con quello di un clock di riferimento
- Soluzione 2: **sincronizzare gli orologi logici**
 - Orologio logico: conta il numero di eventi
 - In un SD non è necessaria la sincronizzazione degli orologi fisici, ma solo l'ordinamento degli eventi (Leslie Lamport)

Modello della computazione distribuita

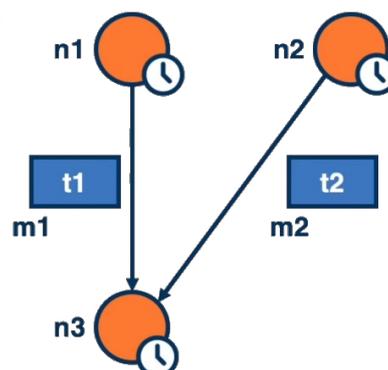
- Componenti del SD: N processi e canali di comunicazione
- Ogni processo p_i ($1 \leq i \leq N$) genera una sequenza di eventi
 - Eventi **interni** (cambiamento dello stato del processo) ed **esterni** (send/receive di messaggi)
 - e_i^k : k -esimo evento generato da p_i
- Evoluzione della computazione rappresentata con un **diagramma spazio-tempo**

→ _{i} : **relazione di ordinamento** tra due eventi in p_i
 $e \rightarrow_i e'$ se e solo se e è accaduto prima di e' in p_i



Timestamping

- Ogni processo etichetta gli eventi con un **timestamp**
- **Soluzione semplice**: ogni processo etichetta gli eventi in base al proprio clock fisico
- Funziona?
 - Possiamo ricostruire l'ordinamento di eventi avvenuti su uno stesso nodo
 - Ma l'ordinamento di eventi avvenuti su nodi diversi?
 - In un sistema distribuito è *impossibile* avere un unico clock fisico condiviso da tutti i processi



SD sincroni e asincroni

- Proprietà di un **SD sincrono**
 1. Esistono 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
 2. Ciascun messaggio trasmesso su un canale di comunicazione è ricevuto in un tempo limitato
 3. Ciascun processo ha un clock fisico con un tasso di scostamento (*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 i clock

- Prima soluzione
 - **Sincronizzare** con *una certa approssimazione* i **clock fisici** dei processi attraverso opportuni algoritmi
 - Ogni processo etichetta gli eventi con il valore del suo clock fisico (che risulta sincronizzato con gli altri clock con una certa approssimazione)
 - Timestamping basato su tempo fisico (**clock fisico**)
- E' sempre possibile mantenere limitata l'approssimazione dei clock fisici?
 - **No** in un **SD asincrono**
 - In un SD asincrono il timestamping non può basarsi sul tempo fisico, bensì sul 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 e 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 da un istante di riferimento fino all'istante t
 - In generale il clock non è completamente accurato: può essere diverso da t
 - Se C_i si comporta abbastanza bene, può essere usato per il timestamping degli eventi di p_i
- Quale deve essere la risoluzione del clock per poter distinguere due eventi?

$$T_{\text{risoluzione}} < \Delta T \text{ tra due eventi rilevanti}$$

Clock fisici in un SD

- In un SD **clock fisici diversi** con possibili valori diversi
- **Skew**: differenza istantanea fra il valore di due clock
- **Drift**: i clock contano il tempo con frequenze differenti (a causa di variazioni fisiche), quindi nel tempo **divergono** rispetto al tempo reale
- **Drift rate**: differenza per unità di tempo di un clock fisico rispetto ad un orologio ideale
 - Es.: drift rate di $2 \mu\text{sec}/\text{sec}$ significa che il clock incrementa il suo valore di $1 \text{ sec} + 2 \mu\text{sec}$ ogni secondo
 - Drift rate di normali orologi al quarzo: 10^{-6} s/s (circa 1 s in 11-12 giorni)
 - Drift rate di orologi al quarzo ad alta precisione: 10^{-7} o 10^{-8} s/s
 - A causa del drift rate, dopo un certo intervallo di tempo i clock di un SD saranno di nuovo disallineati → occorre eseguire una **sincronizzazione periodica** per riallineare i clock

UTC

- **Universal Coordinated Time (UTC)**: riferimento internazionale per il tempo
- Basato sul tempo atomico, occasionalmente corretto utilizzando il tempo astronomico
 - 1 sec = tempo impiegato dall'atomo di cesio 133 per compiere 9192631770 transizioni di stato
- Clock fisici che usano oscillatori atomici sono i più accurati (drift rate pari a 10^{-13} s/s)
- L'output dell'orologio atomico è inviato in broadcast da stazioni radio su terra e da satelliti (es. GPS)
 - In Italia: Istituto Galileo Ferraris
- Nodi con ricevitori possono sincronizzare i loro clock con questi segnali
 - Segnali da stazioni radio su terra: accuratezza nell'intervallo 1-10 ms
 - Segnali da satellite: accuratezza da 0.5 ms fino a 50 ns

Sincronizzazione di clock fisici

- Come sincronizzare i clock fisici con l'orologio atomico oppure 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:

$$|S(t) - C_i(t)| \leq \alpha \text{ per } 1 \leq i \leq N \text{ e per tutti gli istanti in } I$$
 - I clock C_i hanno **accuratezza** α , con $\alpha > 0$
- **Sincronizzazione interna**

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

$$|C_i(t) - C_j(t)| \leq \pi \text{ per } 1 \leq i, j \leq N \text{ nell'intervallo } I$$
 - I clock C_i e C_j hanno **precisione** π , con $\pi > 0$

Sincronizzazione di clock fisici



- 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 con accuratezza α , allora è anche sincronizzato internamente con precisione 2α

Correttezza di clock fisici

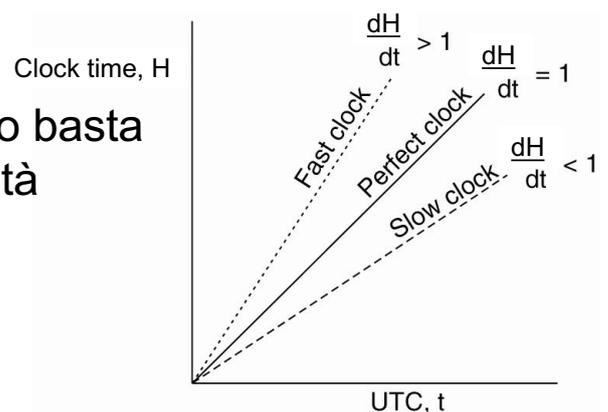
- Un clock hardware H è **corretto** se il suo drift rate è compreso tra $-\rho$ e $+\rho$ con $\rho > 0$
- Se il clock H è corretto, l'**errore** che si commette nel misurare un intervallo di istanti reali $[t, t']$ (con $t' > t$) è **limitato**:

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

- 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 i clock fisici?

- Consideriamo 2 clock aventi lo stesso tasso di scostamento massimo (*maximum clock drift rate*) da UTC pari a ρ
- Ipotizziamo che dopo la sincronizzazione i 2 clock si scostino da UTC in senso opposto
 - Dopo Δt dalla sincronizzazione, si saranno scostati al più di $2\rho \Delta t$
- Affinché i 2 clock non differiscano più di δ , occorre sincronizzarli almeno ogni $\delta/(2\rho)$

Sincronizzazione interna in un SD sincrono

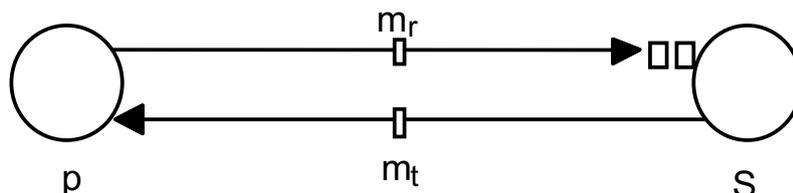
- Algoritmo di **sincronizzazione interna** tra 2 processi in un **SD sincrono**
 1. p_1 manda il suo clock locale t a p_2 tramite un messaggio m , con tempo di trasmissione pari a T_{trasm}
 2. p_2 riceve m e imposta il suo clock a $t + T_{trasm}$
 T_{trasm} non è noto ma, essendo il SD sincrono, $T_{min} \leq T_{trasm} \leq T_{max}$
Sia $u = (T_{max} - T_{min})$ l'incertezza sul tempo di trasmissione
 3. Se p_2 imposta il suo clock a $t + (T_{max} + T_{min})/2$, il lower bound ottimo sullo skew tra i due clock è pari a $u/2$
- L'algoritmo può essere generalizzato per sincronizzare N processi, con lower bound ottimo sullo skew pari a $u(1-1/N)$
- In un **SD asincrono** $T_{trasm} = T_{min} + x$, con $x \geq 0$ e non noto
 - Occorrono altri algoritmi di sincronizzazione dei clock fisici

Sincronizzazione fisica 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: centralizzato oppure distribuito
- Time service **centralizzato**
 - Request-driven: algoritmo di Cristian (1989)
 - Sincronizzazione esterna
 - Broadcast-based: algoritmo di Berkeley Unix - Gusella & Zatti (1989)
 - Sincronizzazione interna
- Time service **distribuito**
 - Network Time Protocol
 - Sincronizzazione esterna

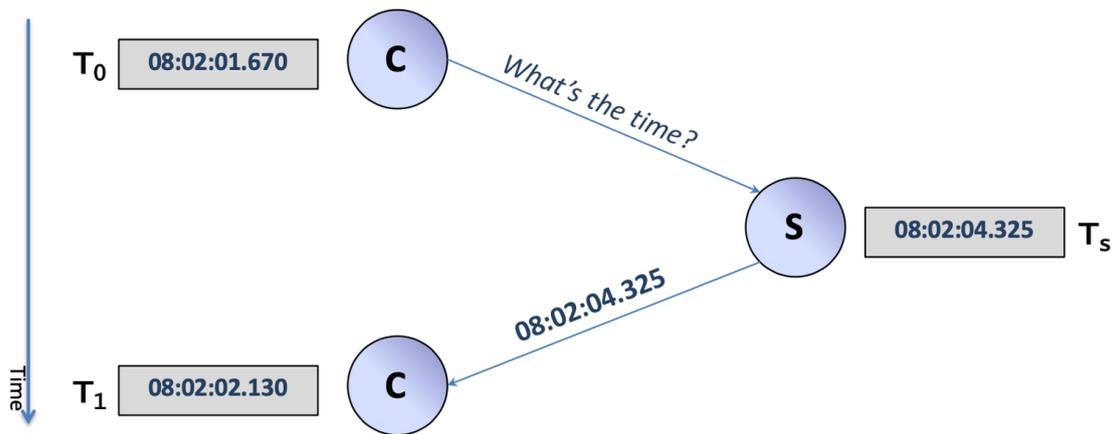
Algoritmo di Cristian

- 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 (RTT) misurato da p



- Osservazioni
 - Singolo time server potrebbe guastarsi
 - Soluzione: usare un gruppo di time server sincronizzati
 - Non gestiti time server maliziosi
 - Accuratezza ragionevole solo se T_{round} è breve → adatto per reti locali con bassa latenza

Algoritmo di Cristian: esempio



- RTT = 460 ms
- C imposta il suo clock a $08:02:04.325 + 230 \text{ ms} = 08:02:04.555$
- Il clock di C avanza di 2,425 s

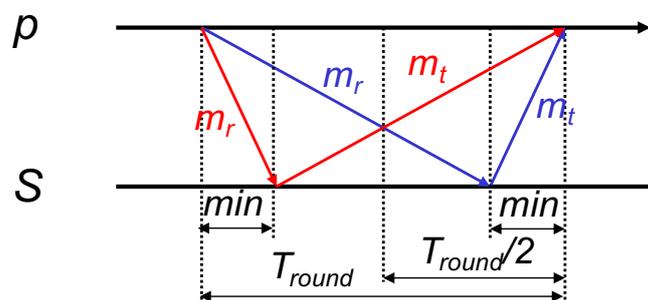
Algoritmo di Cristian: accuratezza

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

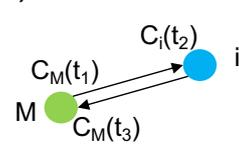
- min è il minimo tempo di trasmissione tra p e S

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

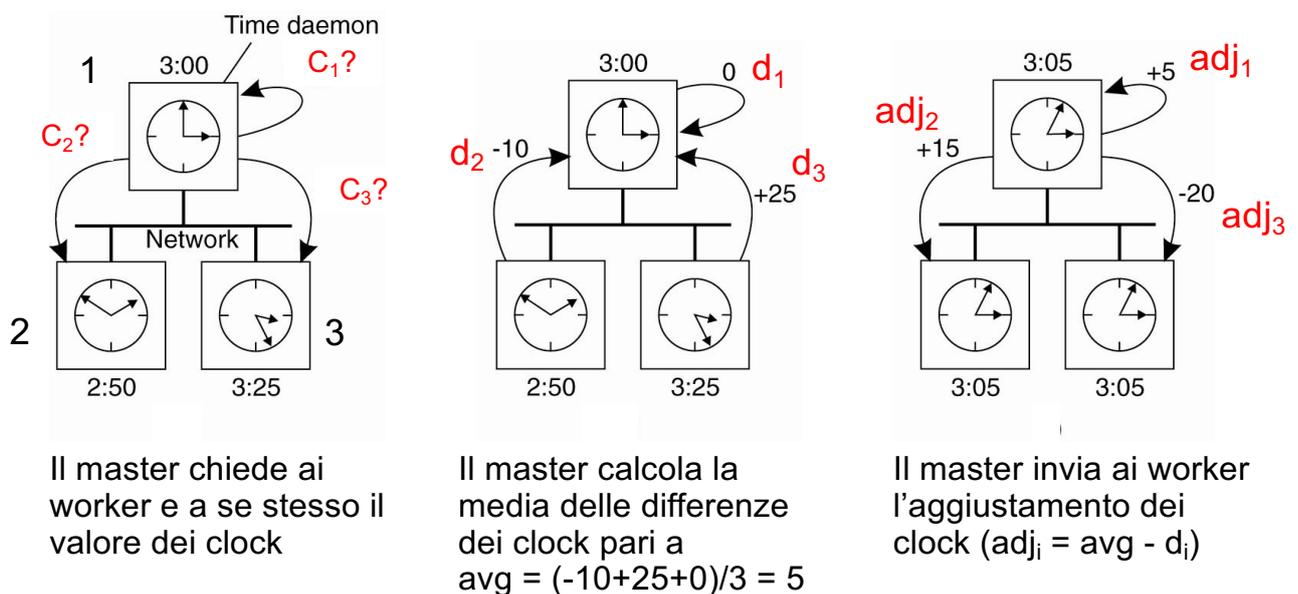
- Il tempo su S quando m_t arriva a p è compreso in $[t + min, t + T_{round} - min]$
 - Ampiezza dell'intervallo è $T_{round} - 2 min$
- Accuratezza $\alpha: \pm (T_{round}/2 - min)$



Algoritmo di Berkeley

- Algoritmo per la **sincronizzazione interna** di un gruppo di nodi
 - Il nodo **master** (time server *attivo*) richiede in broadcast il valore dei clock degli altri nodi (**worker**)
 - Il master usa i RTT per stimare i valori dei clock dei worker
 - δ_i : differenza tra clock del master M e clock del worker i (si calcola in modo simile all'algoritmo di Cristian)
- $$\delta_i = (C_M(t_1) + C_M(t_3))/2 - C_i(t_2)$$
- $C_M(t_1)$ e $C_M(t_3)$: clock su M all'invio e ricezione
 $C_i(t_2)$: clock su i
- 
- Il master calcola la media delle differenze dei clock
 - Il master invia un valore correttivo ai worker
 - Se il valore correttivo prevede un salto indietro nel tempo, il worker non imposta il nuovo valore ma **rallenta il clock**

Algoritmo di Berkeley: esempio



Algoritmo di Berkeley: caratteristiche

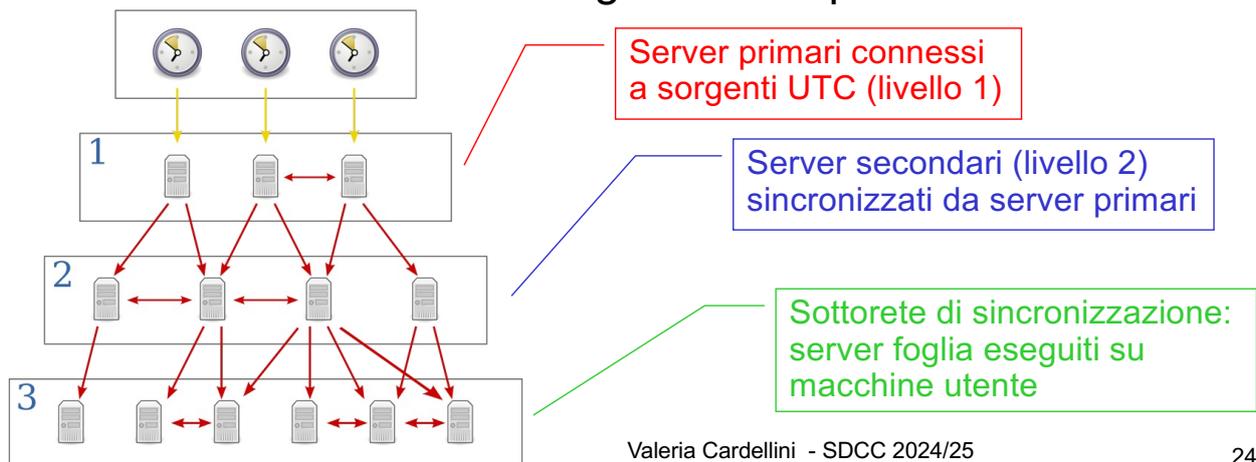
- Precisione dipende da RTT nominale massimo: il master non considera valori di clock associati a RTT superiori al massimo
 - Si scartano gli outlier
- Tolleranza ai guasti
 - Se il master cade, un altro nodo viene eletto come master
 - Tramite un algoritmo di elezione
 - Tollerante a comportamenti arbitrari (worker che inviano valori errati di clock)
 - Il master considera solo quei valori di clock che differiscono tra loro al più per una soglia specificata

Algoritmo di Berkeley: slowdown del clock

- Che cosa significa rallentare un clock?
- Non si può imporre un valore di tempo passato ai worker che hanno un valore di clock superiore a quello calcolato come clock comune
 - 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 rallentarne l'avanzata
 - Il numero di interrupt mascherati è pari al tempo di slowdown diviso il periodo di interrupt del processore

Network Time Protocol (NTP)

- Time service per Internet (standard in RFC 5905)
 - Sincronizzazione esterna accurata rispetto a UTC
 - Servizio configurabile su diversi SO. In GNU/Linux: demone `ntpd`, comando `ntpdate` per sincronizzare manualmente
- Architettura di time service disponibile e scalabile
 - Time server e path ridondanti
- Autenticazione delle sorgenti di tempo



NTP: sincronizzazione

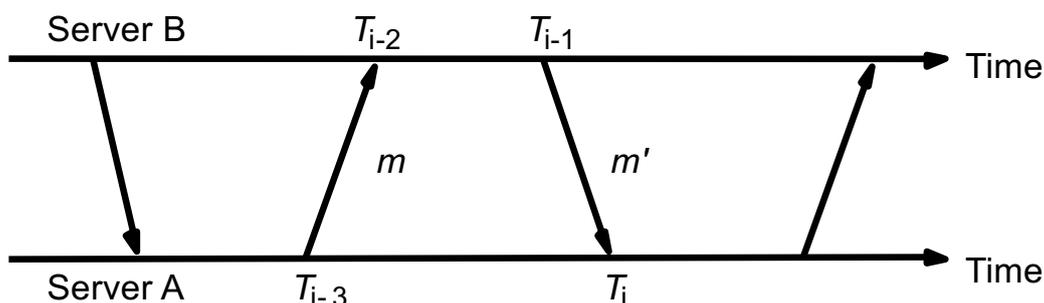
- La sottorete di sincronizzazione si riconfigura in caso di guasti
 - Server primario che perde la connessione alla sorgente UTC diventa server secondario
 - Server secondario che perde la connessione al suo primario (ad es. crash del primario) può usare un altro primario

NTP: modi di sincronizzazione

- Modi di sincronizzazione
 - **Multicast**: server all'interno di LAN ad alta velocità invia in multicast il suo clock agli altri server, che impostano il tempo ricevuto assumendo un certo ritardo di trasmissione
 - Accuratezza relativamente bassa
 - **Procedure call**: un server accetta richieste da altri (come algoritmo di Cristian)
 - Accuratezza maggiore rispetto a multicast
 - Utile se non è disponibile multicast
 - **Simmetrico**: coppie di server scambiano messaggi contenenti informazioni sul timing
 - Accuratezza molto alta (usato per livelli alti della gerarchia)
- Tutti i modi di sincronizzazione usano UDP

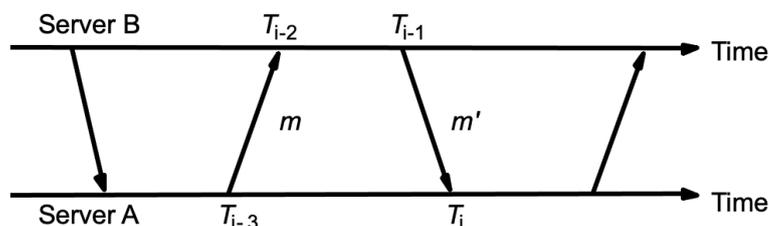
NTP: modo simmetrico

- I server A e B si scambiano coppie di messaggi (m, m') per migliorare l'accuratezza della loro sincronizzazione
- Ogni messaggio NTP contiene timestamp di eventi recenti, ad es. m' contiene
 - Tempo su B di *send* di m' (T_{i-1})
 - Tempo su A di *send* di m (T_{i-3}) e tempo su B di *receive* di m (T_{i-2})
- A registra il tempo di *receive* di m' (T_i)
- Il tempo tra l'arrivo di m e l'invio di m' ($T_{i-1} - T_{i-2}$) può essere non trascurabile



NTP: accuratezza

- Per ogni coppia di messaggi m ed m' scambiati tra i 2 server, NTP stima l'offset o_i tra i 2 clock e il ritardo d_i (pari al tempo totale di trasmissione di m ed m')



- Indicando con:

- o : offset reale del clock di B rispetto al clock di A ($o = \text{clock}_B - \text{clock}_A$)
- t e t' : tempi effettivi di trasmissione di m ed m' rispettivamente

- si ha

$$T_{i-2} = T_{i-3} + t + o \quad \text{e} \quad T_i = T_{i-1} + t' - o$$

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

- Sottraendo l'equazione per T_i da quella per T_{i-2} e risolvendo rispetto ad o :

$$o = [(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)]/2 + (t' - t)/2$$

ponendo $o_i = [(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)]/2$ si ha

$$o = o_i + (t' - t)/2$$

- Poiché $t, t' > 0$, allora $o_i - d_i/2 \leq o \leq o_i + d_i/2$
- Quindi: o_i è la **stima dell'offset** e $d_i/2$ la sua **accuratezza**

28

NTP: accuratezza

- I server NTP applicano un algoritmo di filtraggio statistico sulle 8 coppie $\langle o_i, d_i \rangle$ più recenti, scegliendo come stima di o il valore di o_i corrispondente al minimo d_i
- Applicano poi un algoritmo di selezione dei peer per modificare eventualmente il peer da usare per sincronizzarsi <https://www.eecis.udel.edu/~mills/ntp/html/warp.html>
- Accuratezza di NTP:
 - 10 ms su Internet
 - 1 ms su LAN

Perfect synchronization over networks
is actually impossible

Google's TrueTime (TT)

- Distributed synchronized clock with bounded non-zero error
 - Designed by Google for Spanner, a global-scale, multiversion, distributed NewSQL database
 - Relies on a well-engineered tight clock synchronization available on all Google servers thanks to GPS and atomic clocks
 - Enables applications to generate monotonically increasing timestamps
 - *Cons*: TT requires special hardware and custom-build tight clock synchronization protocol, which is infeasible for many systems
 - Google also relies on its very high throughput, global fiber optic network linking its data centers

Tempo nei SD asincroni

- Gli algoritmi per la sincronizzazione dei clock fisici si basano sulla stima dei tempi di trasmissione
 - Possiamo determinare l'accuratezza conoscendo upper e lower bound del tempo di trasmissione
- Ma in un SD asincrono, no vincoli sui tempi di trasmissione → non possiamo ordinare gli eventi che accadono in nodi diversi usando il tempo fisico
- Tuttavia, di solito ci interessa soltanto che i processi **concordino sull'ordine in cui si verificano gli eventi**, piuttosto che sul tempo in cui sono avvenuti
 - Il tempo fisico non è più importante, usiamo il tempo logico

Tempo logico

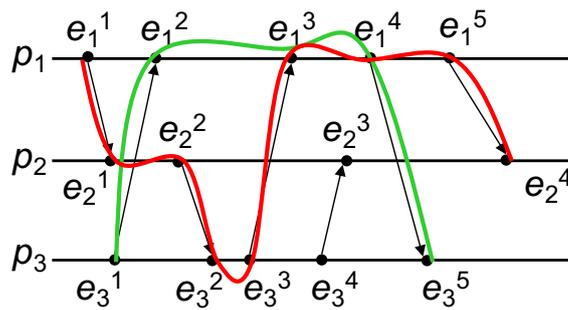
- Idea: ordinare gli eventi in base a 2 osservazioni intuitive:
 1. Due eventi occorsi sullo stesso processo p_i si sono verificati esattamente nell'ordine in cui p_i li ha osservati
 2. Quando un messaggio viene inviato da p_i a p_j , l'evento di *send* precede l'evento di *receive*
- Lamport (1978) introduce il concetto di relazione di **happened-before** (anche detta relazione di **precedenza** o **ordinamento causale**)
 - \rightarrow_i : relazione di ordinamento tra due eventi occorsi su p_i
 - \rightarrow : relazione di happened-before tra due eventi qualsiasi

L. Lamport, Time, Clocks and the Ordering of Events in Distributed Systems, 1978 <https://lamport.azurewebsites.net/pubs/time-clocks.pdf>

Relazione happened-before

- Due eventi e ed e' sono in relazione di **happened-before** (indicata con $e \rightarrow e'$) se è vero uno dei seguenti casi:
 1. $\exists p_i \mid e \rightarrow_i e'$
 2. $e = \text{send}(m) \wedge e' = \text{receive}(m)$
 e è l'evento di invio del messaggio m , e' è il corrispondente evento di ricezione
 3. $\exists e, e', e'' \mid (e \rightarrow e'') \wedge (e'' \rightarrow e')$
Ovvero la relazione happened-before è **transitiva**
- Applicando i tre casi è possibile costruire una sequenza di eventi e_1, e_2, \dots, e_n **causalmente ordinati**
- **Osservazioni**
 - La relazione happened-before rappresenta un **ordinamento parziale** (proprietà: *non riflessivo, antisimmetrico, transitivo*)
 - Non è detto che la sequenza e_1, e_2, \dots, 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 \parallel)

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 ed e_2^1 sono concorrenti
 $e_3^1 \not\rightarrow e_2^1$ ed $e_2^1 \not\rightarrow e_3^1$

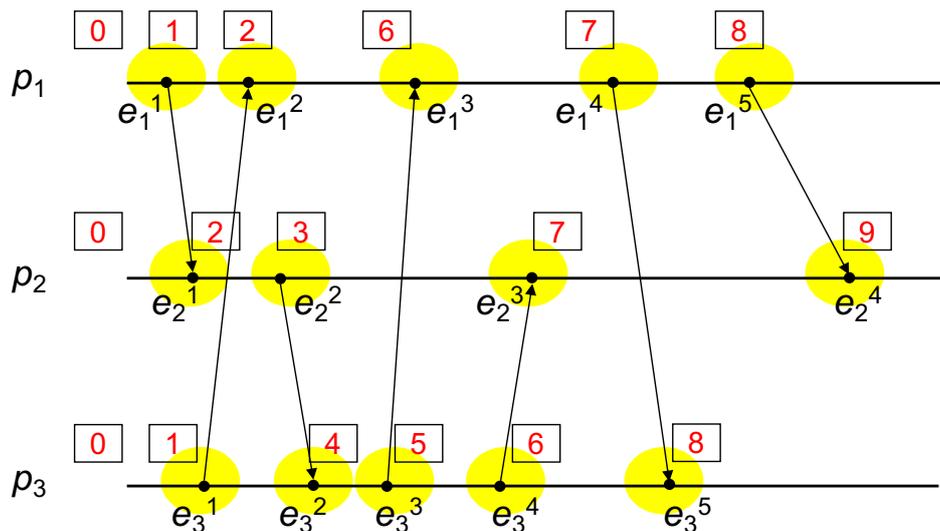
Clock logico scalare

- 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 e lo usa 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:
 - Se $L(e) < L(e')$ non è detto che $e \rightarrow e'$; tuttavia, $L(e) \not< L(e')$ implica che $e \not\rightarrow e'$
 - Happened-before introduce un *ordinamento parziale* degli eventi: nel caso di eventi concorrenti non è possibile stabilire quale evento avviene effettivamente prima

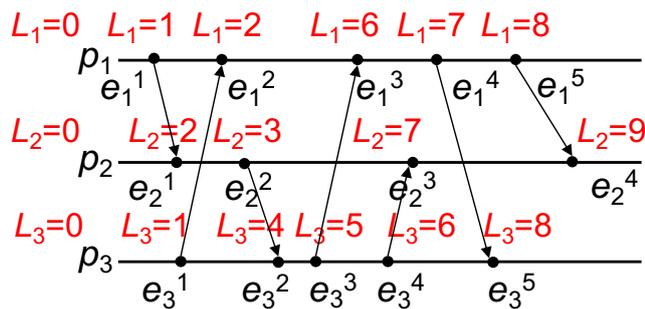
Clock logico scalare: implementazione

- **Algoritmo di Lamport**
- Ogni processo p_i inizializza il proprio clock logico L_i a 0 ($\forall i = 1, \dots, N$)
- Prima di eseguire un evento **interno**, p_i incrementa L_i di 1: $L_i = L_i + 1$
- Quando p_i **invia** il messaggio m a p_j
 - Incrementa il valore di L_i : $L_i = L_i + 1$
 - 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 : $L_j = L_j + 1$
 - Esegue l'evento $receive(m)$

Clock logico scalare: esempio



Clock logico scalare: esempio



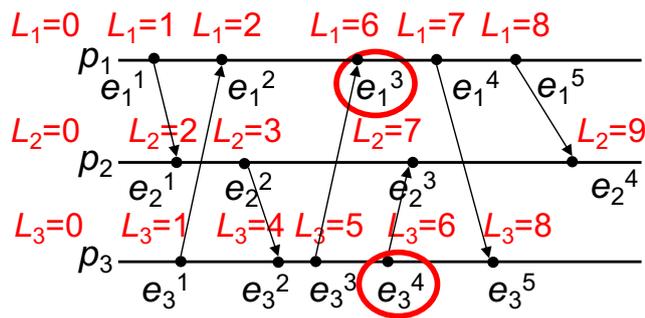
Osservazioni

- $e_1^1 \rightarrow e_2^1$ ed i relativi timestamp riflettono questa proprietà ($L_1=1$ e $L_2=2$); infatti *if* $e_1^1 \rightarrow e_2^1$ *then* $L(e_1^1) < L(e_2^1)$
- $e_1^1 \parallel e_3^1$ ed i relativi timestamp sono uguali ($L_1=1$ e $L_3=1$); infatti *if* $L(e_1^1) \geq L(e_3^1)$ *then* $e_1^1 \not\rightarrow e_3^1$
- $e_2^1 \parallel e_3^1$ ed i relativi timestamp sono diversi ($L_2=2$ e $L_3=1$); infatti *if* $L(e_2^1) \geq L(e_3^1)$ *then* $e_2^1 \not\rightarrow e_3^1$

Ordinamento totale

- Usando il clock logico scalare, due o più eventi possono avere stesso timestamp: come realizzare un ordinamento totale tra eventi, evitando così che due eventi accadano nello *stesso tempo logico*?
- Soluzione: oltre al clock logico, usiamo il numero del processo su cui è avvenuto l'evento
 - Si stabilisce un **ordinamento totale** ($<$) tra i processi
- **Relazione di ordine totale** tra eventi (indicata con $e \Rightarrow e'$): se e è un evento di p_i ed e' è un evento di p_j allora $e \Rightarrow e'$ se e solo:
 1. $L_i(e) < L_j(e')$ or
 2. $L_i(e) = L_j(e')$ and $p_i < p_j$
- Relazione applicata in alcuni algoritmi di mutua esclusione distribuita

Relazione di ordine totale: esempio



- e_1^3 e e_3^4 hanno lo stesso valore di clock logico: come ordinarli?
- $e_1^3 \Rightarrow e_3^4$ poiché $L_1(e_1^3) = L_3(e_3^4)$ e $p_1 < p_3$

Clock logico scalare: limitazione

- 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 $e \rightarrow e'$Esempio slide 36: $L(e_3^1) < L(e_2^1)$ ma $e_3^1 \parallel e_2^1$
- **Conseguenza:** non è possibile stabilire, solo guardando i clock logici scalari, se due eventi sono concorrenti o meno
- Come superare questa limitazione?
- Introducendo i clock logici vettoriali: ad opera di Mattern (1989) e Fidge (1991)

Clock logico vettoriale

- Il clock logico vettoriale per un sistema con N processi è 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 il clock vettoriale si catturano completamente le caratteristiche della relazione happened-before

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

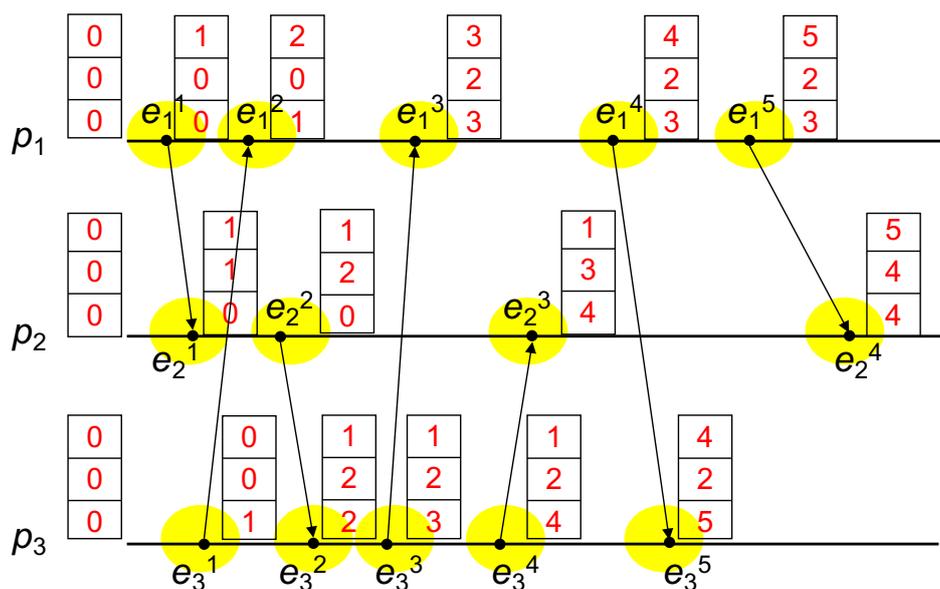
Clock logico vettoriale: significato e confronto

- Dato il 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 ha conoscenza
- Confronto di clock vettoriali
 - $V = V'$ se e solo se $\forall j: V[j] = V'[j]$
 - $V \leq V'$ se e solo se $\forall j: V[j] \leq V'[j]$
 - $V < V'$ (e quindi l'evento associato a V precede quello associato a V') se e solo se
 - $\forall i \in [1, \dots, N]: V[i] \leq V'[i]$
 - and
 - $\exists j \in [1, \dots, N]: V[j] < V'[j]$
 - $V \parallel V'$ (e quindi l'evento associato a V è concorrente a quello associato a V') se e solo se
 - $\text{not}(V < V') \text{ and } \text{not}(V' < V)$

Clock logico vettoriale: implementazione

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

Clock logico vettoriale: esempio



Confronto di clock vettoriali

- Confrontando i timestamp basati su clock vettoriale si può capire se gli eventi sono concorrenti o in relazione happened-before

1
2
0

V

1
2
2

V'

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

1
2
0

V

1
0
2

V'

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