

## 8. Sincronizzazione temporale

Il **tempo** è una caratteristica importantissima nei sistemi distribuiti, eppure i problemi di sincronizzazione tra nodi e/o processi sono molteplici e vanno risolti.

### Esempio di un problema di sincronizzazione

Supponiamo di voler prendere un autobus alle 18:05. Sulla base di come il nostro orologio sia tarato, potremmo trovarci di fronte a due scenari differenti:

1. Se il nostro orologio è 15 minuti indietro rispetto all'orario reale, **perderemo l'autobus**;
2. Se il nostro orologio è 15 minuti avanti rispetto all'orario reale, non perderemo l'autobus, ma aspetteremo di più prima di prenderlo.

La sincronizzazione del tempo è dunque fondamentale per **correttezza** (caso 1) ed **equità** (caso 2).

### Problema 2

Un altro scenario problematico in mancanza di sincronizzazione potrebbe essere legato a un sistema di prenotazione posti in aereo.

1. Il server A riceve una richiesta dal client per acquistare l'ultimo biglietto nel volo ABC 123;
2. Il server A annota come timestamp per l'acquisto del biglietto il suo orologio in quel dato istante, ad es. 9h:15m:32.45s;
3. Essendo questo l'ultimo posto prenotabile, il server A comunica al server B che l'aereo è pieno;
4. Il server B annota sul suo log che l'aereo è pieno, usando tuttavia come timestamp il suo orologio, che in tale istante è 9h:10m:10.11s;
5. Un server C, al momento dell'interrogazione dei log di A e B rimarrà confuso circa i due orari, poiché si ritroverebbe davanti una prenotazione effettuata ad un orario superiore a quello in cui viene notificato l'esaurimento dei posti in aereo.

Un'errata sincronizzazione del tempo comporterebbe dunque che C prenderebbe iniziative errate sulla base di quanto consultato precedentemente, cosa che non potrebbe succedere nel caso in cui A e B fossero sincronizzati correttamente.

### Sistema distribuito asincrono

A differenza dei processi sulla singola macchina, i quali condividono lo stesso clock di sistema (quello del processore), i processi nei sistemi distribuiti su Internet seguono un modello **asincrono**, in cui sono presenti:

- Ritardi nell'invio/ricezione di messaggi;
- Ritardi di elaborazione.

### Fondamenti di un sistema distribuito asincrono

- **Processi**: i protagonisti di un sistema distribuito
- **Stati**: i processi possono essere in stato di running, paused, stopped, ecc.
- **Decisioni**: prese da ogni processo per cambiare il suo stato
- **Eventi**: sono la conseguenza di un'azione

Ogni processo ha il suo **orologio locale** usato per assegnare agli eventi dei **timestamp**. Tale orologio **viene incrementato** dal clock di sistema di quel dato processo che è **different** dal clock di un altro processo (problema di sincronizzazione)

## Obiettivo finale

Il nostro obiettivo è quello di risolvere i problemi di sincronizzazione, facendo in modo che gli eventi vengano ordinati cronologicamente col timestamp corretto, a prescindere dagli orologi locali. Ad esempio, un evento legato al processo A, se avvenuto prima di un evento al processo B, deve essere ordinato cronologicamente in modo corretto.

# Clock Skew vs. Clock Drift

## Approfondimento ricevimento

### Differenza fra clock drift e maximum drift rate

Il clock drift è la differenza di frequenza che vi è tra due sistemi, e misura quanto questi divergono nel tempo. Il clock drift rappresenta la derivata del clock skew. L' MDR invece è la massima velocità di allontanamento che si può ritenere accettabile, quindi se riscontro una velocità di allontanamento troppo elevata, dovrà sincronizzare nuovamente i due sistemi

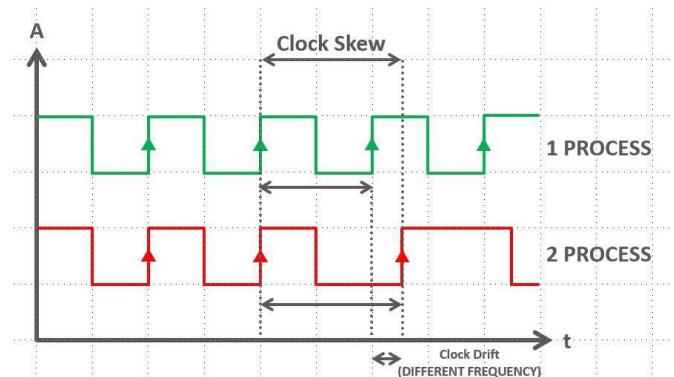
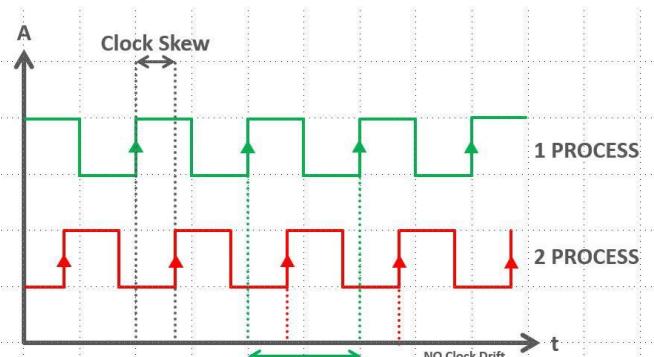
Nel momento in cui si comparano due clock di due processi, parleremo di:

- **Clock Skew:** differenza di tempo relativa ai **timestamp** di due processi;
- **Clock Drift:** differenza relativa delle **frequenze** di clock tra due processi.

Sulla base di questi due valori:

- Se il **clock skew** è diverso da zero, ciò implica che i due orologi non sono sincronizzati;
- Se il **clock drift** è diverso da zero, lo skew potrebbe eventualmente aumentare;
- Se il **clock drift** è uguale a zero, lo skew non subisce variazioni.

Si dice eventualmente, poiché qualora uno dei due processi avesse una frequenza di clock più bassa, ad un certo punto i timestamp combacerebbero, azzerando per un certo lasso di tempo lo skew. Ad un certo punto, se il clock drift rimane ancora non nullo, lo skew aumenta di nuovo.



Clock **drift** variazione del periodo di clock dovuto a fattori esterni ad es. la tempratura

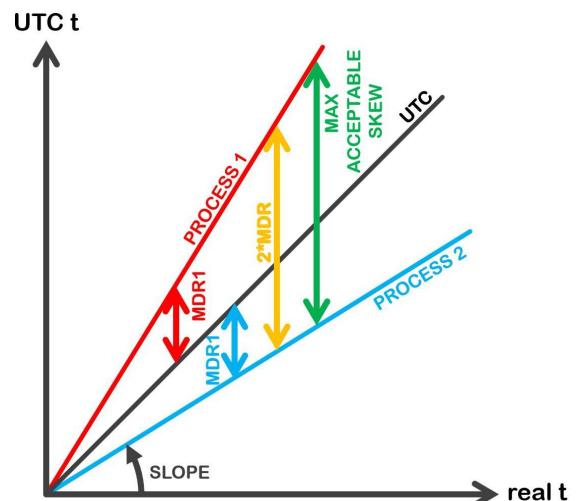
# Quanto spesso sincronizzare?

## MDR assoluto

L'**MDR assoluto** è definito rispetto al tempo universale coordinato **UTC** (che nell'esempio rappresentava l'orario reale nel quale passava l'autobus), mentre l'**MDR di un processo** dipende dall'ambiente di esecuzione.

## MDR tra due clock

Nel caso peggiore, uno dei due clock è **avanti** di MDR rispetto all'UTC, mentre l'altro è **indietro**, per cui si dice che il maximum drift rate tra due clock con MDR locale simile, quindi la loro differenza relativa, è data da  $2 * MDR$  (ipotizzando che  $MDR_1 = MDR_2$ ).



Osservando il grafico notiamo il processo 1 e il processo 2, lo slope indica la velocità di ciascun processo (maggiore è lo slope maggiore è la velocità). Bisogna sincronizzare una volta superato il rapporto fra Maximum Acceptable Skew (M)/ $2 * MDR$

# Sincronizzazione esterna vs. interna

Esistono due tipi di sincronizzazione:

- **Sincronizzazione esterna:** la sincronizzazione del tempo avviene rispetto a una sorgente esterna, quale può essere l'UTC. Ogni processo  $i$  avrà un suo clock  $C(i)$  il cui **timestamp** può avere al massimo un margine di errore pari a  $D$  secondi rispetto alla sorgente esterna  $S$ . A tal proposito, in **tutti** gli istanti di tempo si avrà sempre che  $|C(i) - S| < D$ .
- Esempi di algoritmi che usano questo tipo di sincronizzazione sono l'**algoritmo di Cristian** e il **Network Time Protocol (NTP)**.
- **Sincronizzazione interna:** i processi sono organizzati in **coppia**, in modo tale che i loro clock abbiano un margine di errore pari a  $D$  secondi rispetto alla loro differenza. In tutti gli istanti di tempo si avrà sempre che  $|C(i) - C(j)| < D$ .

Un esempio di algoritmo che usa questo tipo di sincronizzazione è l'**algoritmo di Berkeley**.

### Sincronizzazione interna ed esterna sono legate da una relazione:

- In un gruppo composto da coppie di processi, in cui viene usata la **sincronizzazione interna** è possibile fare uso anche della sincronizzazione esterna. In particolare se si ha una sincronizzazione esterna con margine pari a D, si può avere una sincronizzazione interna con un margine di errore massimo pari a  $2*D$ . Questo avviene poiché un processo può essere avanti di D rispetto a una sorgente esterna, mentre un altro indietro di D rispetto a tale sorgente, pertanto la loro distanza nel caso peggiore è pari a  $2*D$ .
- La presenza di una sincronizzazione interna **NON** implica la presenza di una **sincronizzazione esterna**. Per questi motivi, l'intero sistema può avere una notevole differenza di clock rispetto alla sorgente esterna **S**, e questo è uno degli svantaggi principali della sincronizzazione interna, quindi **non sarà possibile poter confrontare la cronologia degli eventi con sistemi esterni**.

## Sincronizzazione esterna

L'approccio più basilare possibile per sincronizzare un processo **P** con un server **S** potrebbe essere il seguente:

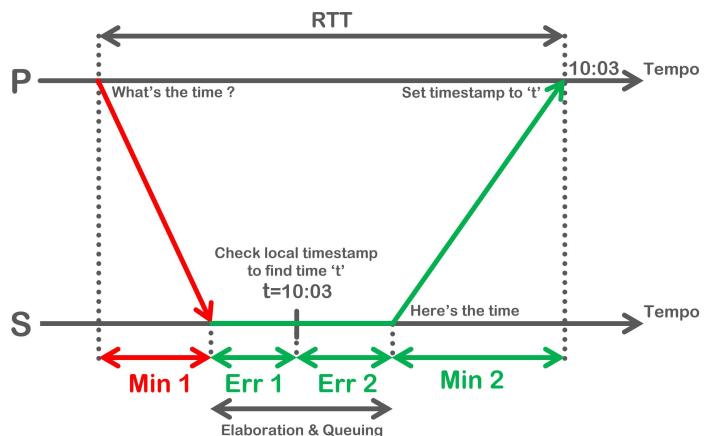
1. **P** manda un messaggio a **S** chiedendo "Che ore sono?";
2. **S** controlla l'orario locale e lo inserisce in un messaggio di risposta;
3. **P** riceve il messaggio di risposta e modifica il timestamp locale.

Tuttavia, cosa c'è di sbagliato in questo particolare protocollo? Nel momento in cui avviene uno scambio di messaggi, bisogna tenere conto dei ritardi di propagazione illimitati sui vari collegamenti, pertanto il processo **P** riceverà sempre un timestamp vecchio rispetto al tempo di **S**, quindi è opportuno implementare un algoritmo che, tenendo conto dei ritardi di propagazione, vada a correggere l'eventuale errore.

## Algoritmo di Cristian

Nell'**algoritmo di Cristian** si corregge l'inaccuratezza facendo sì che **P** misuri il **round trip time (RTT)** durante lo scambio dei messaggi, che assieme al timestamp fornito permetterà di ottenere quello corretto.

Supponiamo quindi di conoscere la latenza di trasmissione minima da **P** a **S**, che chiameremo **min1**, nonché la latenza minima da **S** a **P** che chiameremo **min2**. Entrambe dipendono dal sistema operativo, dal carico, ecc.

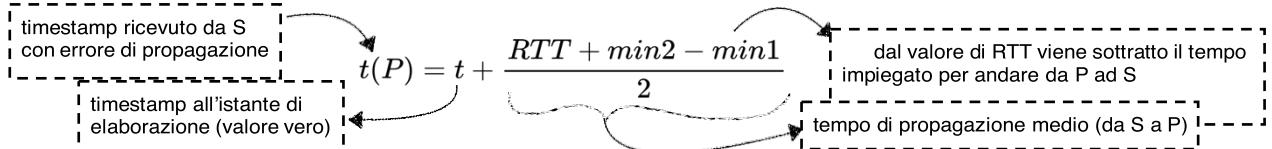


Il tempo attuale quando P riceve la risposta sarà **nell'intervallo**  $[t + min2, t + RTT - min1]$ . Si noti che l'RTT è **più grande** di  $min1 + min2$ , infatti bisogna considerare anche il tempo di elaborazione di S che sta svolgendo pure altri task.

- Nel caso in cui il messaggio di risposta impiegasse il tempo minimo, allora sarebbe ordine di grandezza di  $t + min2$ , cioè il timestamp + la latenza;
- Nel caso peggiore, sarebbe nell'ordine di  $t + RTT - min1$  dove RTT include il tempo di accodamento ed elaborazione da parte dell'sorgente

$$RTT = \text{ping}(P-S) + \text{Telaboraz} + \text{ping}(S-P)$$

Nell'algoritmo di Cristian, il tempo di P viene impostato per essere il valore medio tra il massimo e il minimo, ovvero:



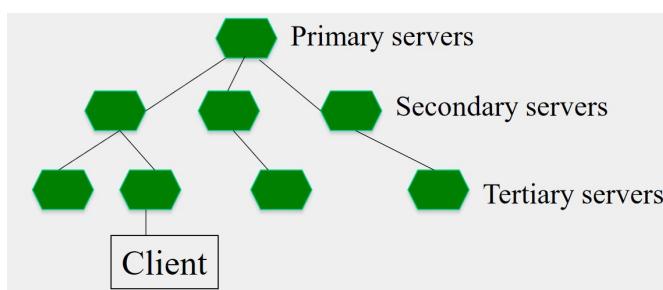
L'**errore** in questo nuovo valore sarà pari a  $(RTT - min2 - min1)/2$  e dipende dall'**RTT**, più è grande, maggiore è l'errore.

## Strategie

- Affinchè i timestamp degli eventi siano allineati correttamente, linearmente l'uno rispetto all'altro, il timestamp locale dovrebbe essere sempre aumentato, ma NON diminuito;
- La velocità del clock può essere, sia aumentata, che diminuita, a seconda di ciò che sta accadendo nell'algoritmo;
- In caso di eccessivi errori, si potrebbe pensare a letture multiple dalle quali calcolare un valore medio.

## Network Time Protocol (NTP)

Lo standard *de facto* per la sincronizzazione degli orologi dei vari processi è **NTP** (Network Time Protocol). La struttura prevede un'organizzazione **ad albero** con server primari, secondari, terziari, ecc.



Questi server si sincronizzano tra loro, in particolare: il nodo figlio si sincronizza col genitore.



Processo di sincronizzazione Fonte RFC 958 - Settembre 1985

<https://www.rfc-editor.org/rfc/rfc958>

The NTP protocol uses the following mechanism to **synchronize the timestamp of the client with the timestamp of a server**, where the distinction between client and server is significant only in the way they interact in the request/ response interchange.

1. The client constructs the initial UDP packet containing all necessary parameters for synchronization and also fills the Originate Timestamp (**T1**) field with the local timestamp of the instant when the packet is sent.
2. When the **server receives** the packet, it modifies the same packet by adding the timestamp (local server time) of the packet reception in the Receive Timestamp (**T2**) field.
3. The server also populates the Transmit Timestamp (**T3**) field with the timestamp (local server time) at the moment when the response packet is sent to the client.
4. The client receives the packet and immediately saves the time of packet reception (**T4**).
5. The **Roundtrip Delay (d)** or Roundtrip Time and the **clock offset (c)** are calculated.

The destination peer calculates the roundtrip delay and clock offset relative to the source peer as follows. Let t1, t2 and t3 represent the contents of the Originate Timestamp, Receive Timestamp and Transmit Timestamp fields and t4 the local time the NTP message is received. Then the roundtrip delay d and clock offset c is:

$c = (t_2 - t_1 + t_3 - t_4)/2$  and  $d = (t_4 - t_1) - (t_3 - t_2)$  where **t3 - t2 is the elaboration time which isn't included in RTT calculus**

L'offset **c** del timestamp inviato dal server viene calcolato a partire dal RTT anche detto RTD **d**, per far ciò risolviamo il seguente sistema

$$\begin{cases} t_2 = t_1 + c + \frac{d}{2} \\ t_4 = t_3 - c + \frac{d}{2} \end{cases}$$

Sottraiamo alla prima  
laseconda e  
ricaviamo il time  
offset **c**

$$\begin{aligned} t_2 - t_4 &= t_1 - t_3 + c + c + \frac{d}{2} - \frac{d}{2} \\ 2c &= t_2 - t_1 + t_3 - t_4 \\ c &= \frac{t_2 - t_1 + t_3 - t_4}{2} \end{aligned}$$

## Corso di Tecnologie per il cloud - Sincronizzazione temporale

Il valore **c** rappresenta l'offset ideale, ma per ottenere l'offset reale tra i due orologi ovvero  **$\sigma_{real}$**  bisogna tenere conto dell'errore che è definito dalla formula:

$$e = \frac{L2 - L1}{2} \quad \text{dove } L2 \text{ è uguale a } (t_2 - t_1) \text{ e } L1 \text{ è uguale a } (t_4 - t_3)$$

**quindi  $\sigma_{real}$  è pari a  $c + e$**

### Originate Timestamp (T1)

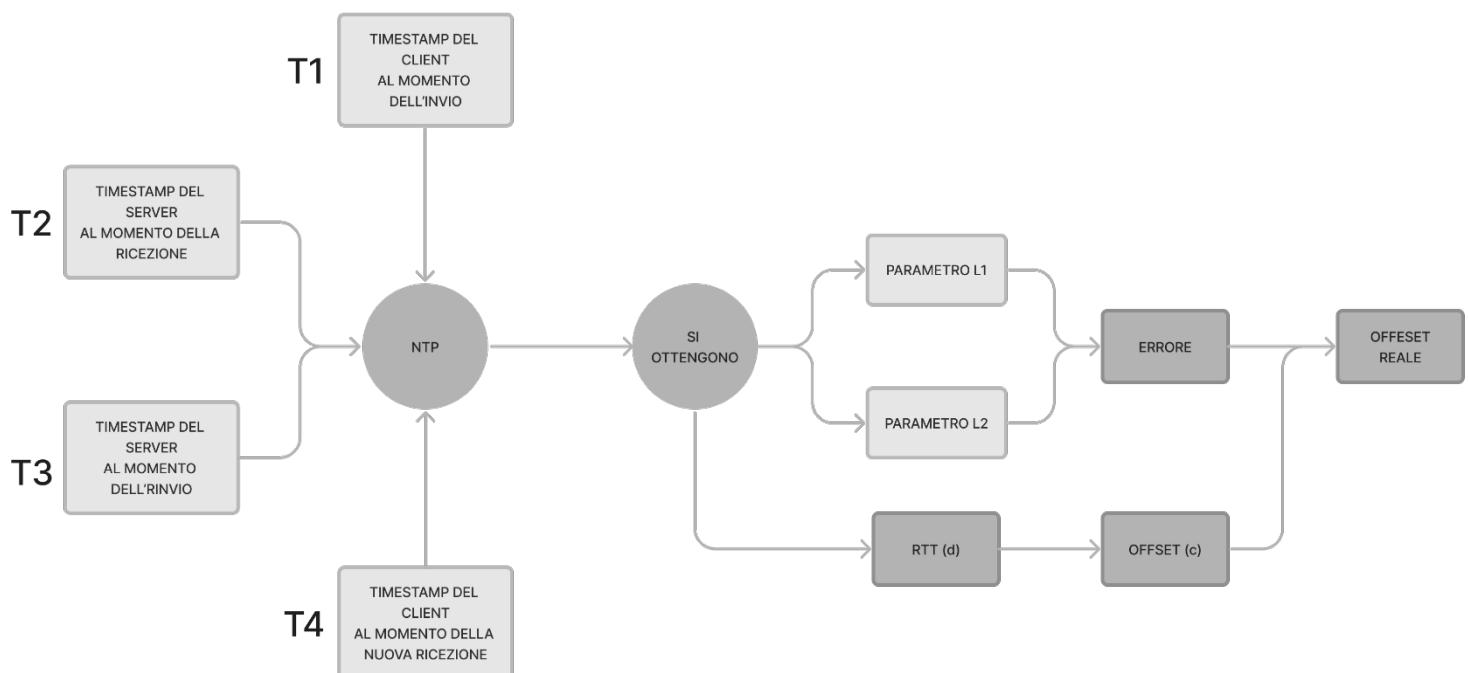
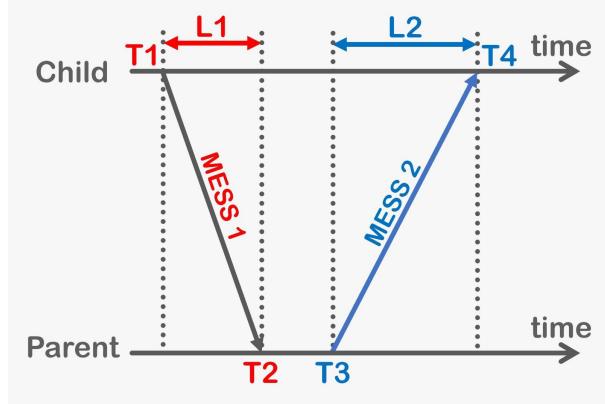
This is a 64-bit timestamp established by the client host and specifying the local time at which the request departed for the service host. It will always have a nonzero value.

### Receive Timestamp (T2)

This is a 64-bit timestamp established by the server host and specifying the local time at which the request arrived from the client host. If no request has ever arrived from the client the value is zero.

### Transmit Timestamp (T3)

This is a 64-bit timestamp established by the server host and specifying the local time at which the reply departed for the client host. If no request has ever arrived from the client the value is zero.



## Lamport timestamps

I Lamport timestamps sono un modo per assegnare timestamp logici ai vari eventi che avvengono all'interno di un sistema distribuito, in modo da poter determinare l'ordine parziale degli eventi che si verificano nei nodi del sistema.

Ogni evento che avviene in un nodo del sistema distribuito viene assegnato un valore numerico intero, detto timestamp, che rappresenta il momento in cui l'evento si è verificato. I timestamp non sono necessariamente rappresentativi del tempo reale, ma piuttosto del "tempo logico" dell'evento all'interno del sistema distribuito.

Per assegnare i Lamport timestamps, ogni nodo mantiene un **timestamp locale**, che rappresenta l'ultimo timestamp assegnato dal nodo stesso. Quando un evento si verifica nel nodo, il **timestamp locale viene incrementato di una unità** e il nuovo timestamp viene assegnato all'evento. In questo modo, ogni evento avrà un timestamp maggiore di tutti gli eventi precedenti in quel nodo.

Quando un nodo invia un messaggio a un altro nodo, il messaggio contiene il timestamp locale del nodo mittente. Quando il nodo destinatario riceve il messaggio, confronta il timestamp del messaggio con il proprio timestamp locale. Se il timestamp del messaggio è maggiore del timestamp locale del nodo destinatario, il nodo aggiorna il proprio timestamp locale al valore del timestamp del messaggio più uno. In questo modo, il nodo destinatario garantisce che il timestamp dei messaggi ricevuti sia sempre maggiore del timestamp degli eventi precedenti nel nodo.

La logica "happens-before" utilizza i timestamp di Lamport per stabilire l'ordine parziale tra gli eventi. In particolare, se un evento A ha un timestamp più piccolo di un evento B, allora A deve essere accaduto prima di B (o essere correlato causalmente a B). Questo ordine parziale viene utilizzato per garantire la corretta esecuzione delle operazioni in un sistema distribuito e per prevenire situazioni di inconsistenza dei dati.

In sintesi, la logica "happens-before" e i timestamp di Lamport sono strettamente correlati e vengono utilizzati insieme per garantire la corretta esecuzione delle operazioni in un sistema distribuito.

## Relazione logica

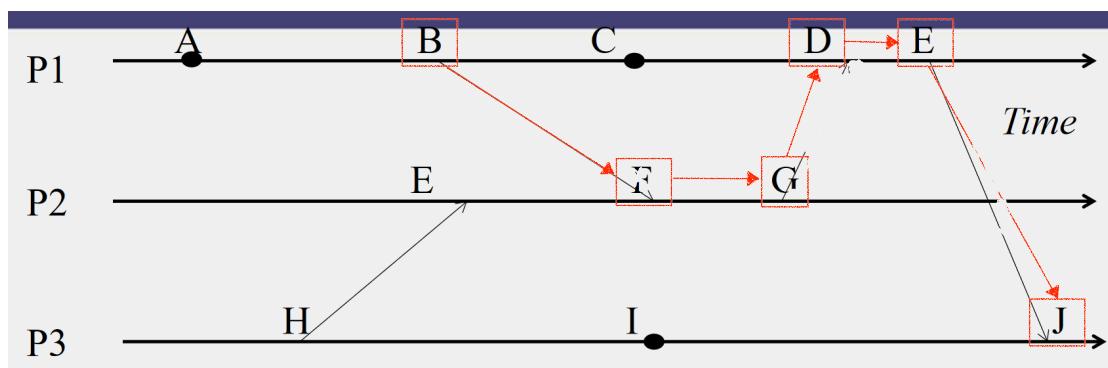
Per ogni coppia di eventi si definisce una relazione di tipo **Happens Before**, indicata con una *freccia in avanti orizzontale*. Dire  $a \rightarrow b$  significa che  $a$  è l'evento che porta a  $b$ .

Ci sono in particolare tre regole da rispettare, che permettono di determinare se due eventi sono **CAUSAlmente** correlati o meno:

1. Dato il **medesimo** processo,  $a \rightarrow b$  se l'evento  $a$  è accaduto a un tempo inferiore a  $b$  rispetto al timestamp locale;
2. Data una **coppia** di processi, se  $p_1$  invia un messaggio  $M$  a  $p_2$ , allora l'invio viene **prima** della ricezione;
3. Per la **proprietà transitiva**, se  $a \rightarrow b$  e  $b \rightarrow c$ , allora  $a \rightarrow c$ , permettendo così di relazionare eventi molto lontani temporalmente.

L'ordine tra gli eventi è **parziale**, e **solo** alcune coppie di eventi sono collegate tra loro.

## Logica Happens-Before



Siano **P1**, **P2** e **P3** tre processi inizialmente **non sincronizzati**, si faccia attenzione agli eventi **B, F, G, D, E, J** (**gli eventi NON SONO i timestamp**).

1. L'evento di invio del messaggio da **P1** a **P2** viene etichettato con **B**, mentre l'evento di ricezione viene etichettato con **F**;
2. L'evento di invio del messaggio da **P2** a **P1** viene etichettato con **G**, mentre l'evento di ricezione con **D**;
3. L'evento di invio da **P1** a **P3** viene etichettato con **E**, mentre l'evento di ricezione con **J**.

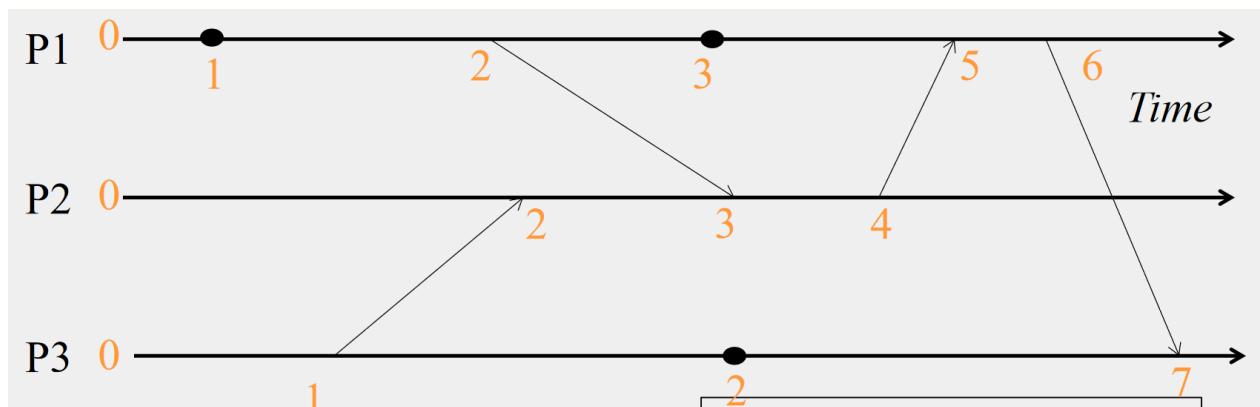
Diremo, per la proprietà **transitiva** che:

- $A \rightarrow B$
- $B \rightarrow F$
- $A \rightarrow F$
- $H \rightarrow G$
- $F \rightarrow J$
- $H \rightarrow J$
- $C \rightarrow J$

Affinché i timestamp vengano stabiliti secondo un ordine logico, ci sono alcune regole che Lamport ha stabilito:

1. Il **timestamp** è un valore intero;
2. Il **timestamp** è inizialmente impostato a **zero** e può essere aggiornato **SOLAMENTE** dal processo a cui appartiene;
3. Il **processo**, ogni volta che **invia o riceve** un messaggio, **incrementa** il contatore di **uno** e assegna il nuovo valore all'evento in corso;
4. Nel messaggio **ricevuto** da un processo, è compreso anche il valore di timestamp di invio del processo che l'ha inviato. Il processo ricevente **confronta** il timestamp del messaggio con quello locale e, se il messaggio ha un timestamp superiore, aggiorna il suo contatore con **tal valore + 1**;

### Esempio



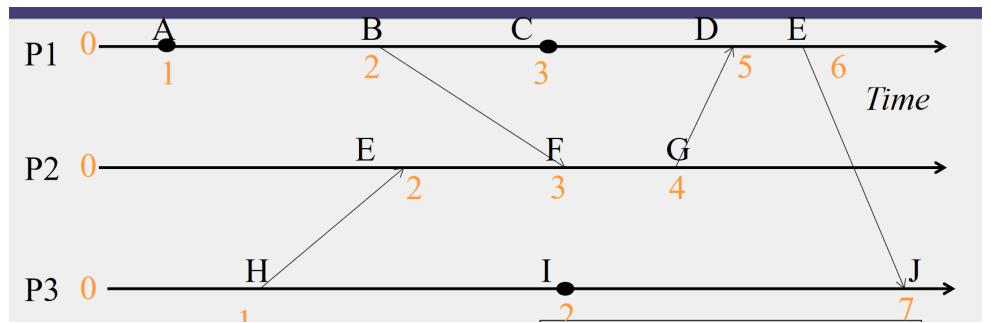
1. Il processo **P2** ha il contatore inizialmente nullo. Riceve un messaggio da **P3**, con timestamp pari a 1. Quindi imposta il suo timestamp a  $1+1=2$ ;
2. Sempre **P2** riceve un messaggio da **P1** con timestamp pari a 2, quindi aggiorna il suo contatore a  $2+1=3$ ;
3. **P1** riceve un messaggio da **P2** con timestamp pari a 4, essendo il suo contatore pari a 3, aggiorna il timestamp a 5.

## Eventi causalmente correlati

In questo caso, gli eventi:

- $A \rightarrow B$
- $B \rightarrow F$
- $A \rightarrow F$

Obbediscono alla **causalità**, poiché i loro timestamp sono a coppie ordinate ( $1 < 2, 2 < 3$ , ecc.).



Nonostante l'ordine dei timestamp, la causalità **non sempre** è verificata. Ad esempio:

- $C \rightarrow F$  non ha alcun percorso causale che porta da C a F, e viceversa.
- $H \rightarrow C$  non ha alcun percorso causale che da H porta a C e viceversa;

Per riassumere:

- $E(P2) \rightarrow E(P1)$  implica che  $timestamp E(P2) < timestamp E(P1)$ ;
- però se  $timestamp E(P2) < timestamp E(P1)$  **NON** implica che ci sia una correlazione, ovvero che  $E(P2) \rightarrow E(P1)$  oppure che  $E(P2)$  sia concorrente a  $E(P1)$ .  
questo perché i timestamp sono locali

Inoltre, i Lamport timestamp **NON** possono distinguere eventi concorrenti da eventi a causali.

## Vector timestamps

I **vector timestamps** vengono utilizzati nei sistemi in cui si associano delle coppie chiave-valore, come **Riak**.

**Inoltre i vector timestamps sono in grado di distinguere gli eventi causali da quelli concorrenti.**

Ciascun processo di sistema utilizza un **vettore** di timestamp e non un singolo numero intero.

- Siano **N** dei processi in un gruppo numerato da **1** a **N**, il vector timestamp di ciascun processo ha **N** elementi;
- Ciascun processo **i** può aggiornare arbitrariamente la **sua** componente:  $V_i [1 \dots N]$ , mentre aggiorna gli altri timestamp sulla base di quanto ricevuto dagli altri processi;
- Il **vector timestamp** associato al processo viene aggiornato quando: si riceve un messaggio da un altro processo oppure quando si verifica un evento all'interno dello stesso.

## Regole

Le regole sono simili a quelle dei timestamp di Lamport, con alcune differenze legate alla presenza del vettore:

1. Quando si esegue un'**istruzione** o l'**i**-esimo processo invia un messaggio, incrementa **SOLO** l'**i**-esimo elemento del suo vettore di **uno**.
2. Quando l'**i**-esimo processo riceve un messaggio:

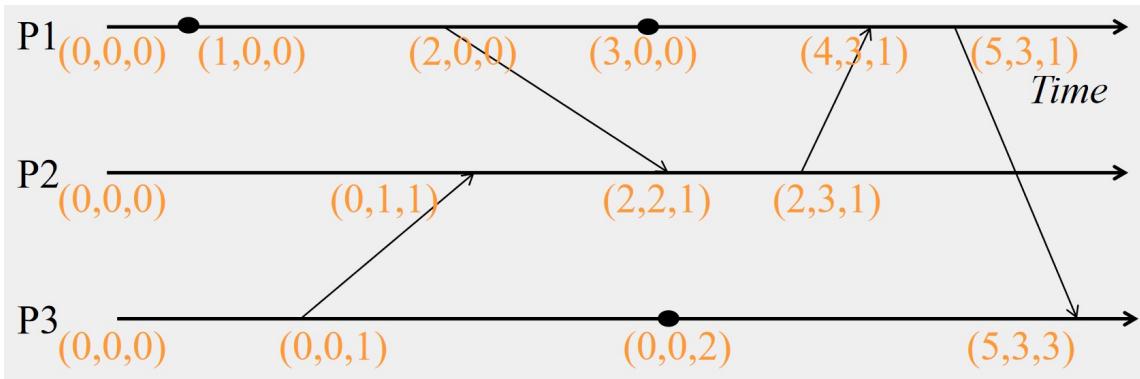
- a. La componente  $i$ -esima del vettore viene incrementata di **uno** rispetto al vettore locale;
- b. Delle altre componenti si confronta il valore massimo tra il vettore ricevuto nel messaggio e il vettore locale prima della ricezione del messaggio stesso.

In generale:

$$V_i[i] = V_i[i] + 1$$

$$V_i[j] = \max(V_{message}[j], V_i[j]) \text{ per } j \neq i$$

### Esempio



1. Inizialmente tutti i valori di timestamp sono impostati a **zero**;
2. Ad un certo punto, nel processo **P1** si verifica un evento che comporta l'incremento del suo contatore locale a 1;
3. Dopodiché, il processo **P1** aggiorna ancora una volta il suo contatore in occasione dell'invio del messaggio a **P2** e lo invia contestualmente al messaggio al processo **P2**;
4. Il processo **P2** ha, al momento della ricezione, il vettore locale su  $(0, 1, 1)$ . Non appena riceve il vettore di **P1**, porta l'attuale su  $(0, 2, 1)$ , aggiornando la seconda componente. Dopodiché, confronta  $(0, 2, 1)$  con  $(2, 0, 0)$ , prendendo i valori massimi per ogni componente. Il nuovo vettore sarà quindi  $(2, 2, 1)$ ;
5. Il processo **P2** procede con l'invio di un messaggio a **P1**, portando il suo vettore a  $(2, 3, 1)$ ;
6. **P1**, che ha come vettore attualmente  $(3, 0, 0)$ , poiché nel frattempo sono accaduti altri eventi, inseguito al confronto avrà  $(4, 3, 1)$ ;
7. Discorso analogo per tutti gli altri eventi.

## Corso di Tecnologie per il cloud - Sincronizzazione temporale **Causalità**

Dati due vettori  $VT_1$  e  $VT_2$  associati agli eventi **E1** e **E2**:

1. Sono **uguali** se l' $i$ -esimo elemento del primo vettore è uguale all' $i$ -esimo elemento del secondo vettore, per tutti gli  $i$ .

$$VT_1[i] = VT_2[i] \text{ per ogni } i = 1, \dots, N$$

2.  $VT_1 \leq VT_2$  se l' $i$ -esimo elemento del primo vettore è **minore** o **uguale** dell' $i$ -esimo elemento del secondo vettore, per tutti gli  $i$ .

$$VT_1[i] \leq VT_2[i] \text{ per ogni } i = 1, \dots, N$$

A questo punto, due eventi **E1** ed **E2** con timestamp vettoriali rispettivi  $VT_1$  e  $VT_2$  sono **causalmente correlati**, ovvero **E1** accade **prima** di **E2** se

- $VT_1$  è **strettamente inferiore** a  $VT_2$ :

$$VT_1[i] < VT_2[i] \text{ per ogni } i = 1, \dots, N$$

- $VT_1 \leq VT_2$ , ma **esiste** un  $j$ -esimo elemento per il quale nel primo vettore è strettamente inferiore al secondo:

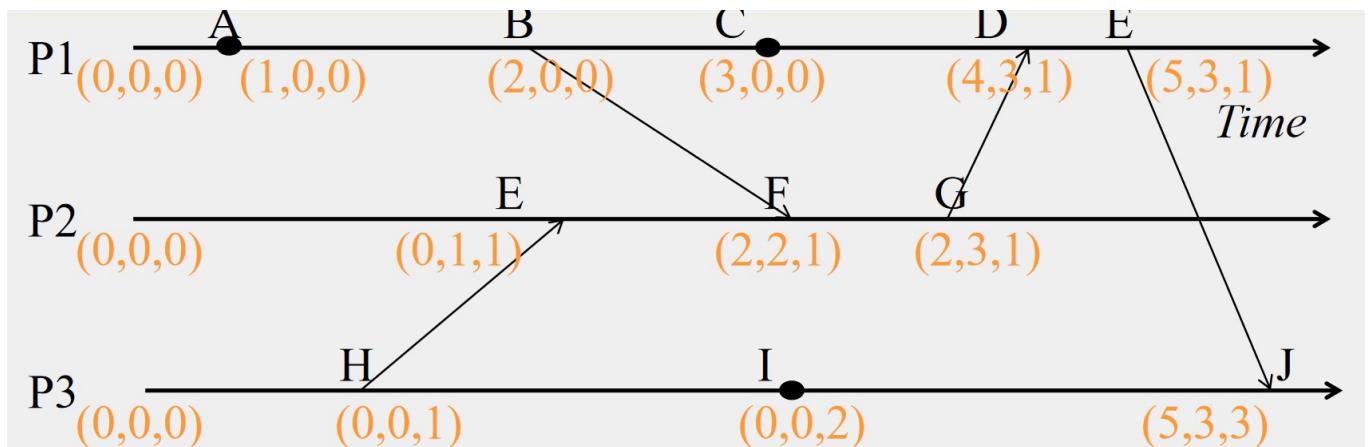
$$VT_1[j] < [VT_2]$$

Due eventi **E1** ed **E2** **NON** sono causalmente correlati se  $VT_1$  e  $VT_2$  sono **concorrenti**:

$$NOT(VT_1 \leq VT_2) \text{ AND } NOT(VT_2 \leq VT_1)$$

Ciò significa che  $VT_1$  e  $VT_2$  sono **simultanei** l'uno con l'altro, e si può indicare anche come:  $VT_2 \parallel VT_1$ .

## Esempio



### Eventi causalmente correlati

Si noti che per queste coppie di eventi, tutte le componenti del primo vettore sono strettamente inferiori a quelle del secondo vettore.

- $A \rightarrow B:: (1, 0, 0) < (2, 0, 0)$ ;
- $B \rightarrow F:: (2, 0, 0) < (2, 2, 1)$ ;
- $A \rightarrow F:: (1, 0, 0) < (2, 2, 1)$ ;
- $H \rightarrow G:: (0, 0, 1) < (2, 3, 1)$ ;
- $F \rightarrow J:: (2, 2, 1) < (5, 3, 3)$ ;
- $H \rightarrow J:: (0, 0, 1) < (5, 3, 3)$ ;
- $C \rightarrow J:: (3, 0, 0) < (5, 3, 3)$ ;

### Eventi non causalmente correlati

Le coppie  $(C, F)$  e  $(H, C)$  sono eventi **concorrenti**, poiché hanno almeno una componente del primo vettore rispetto al secondo **non** strettamente inferiore.

- $C \& F:: (3, 0, 0) ||| (2, 2, 1)$ ;
- $H \& C:: (0, 0, 1) ||| (3, 0, 0)$ ;

**Questo meccanismo permette ai vector timestamp di distinguere gli eventi causali da quelli concorrenti.**

# Mappa Capitolo

