Towards Logical Time

Alessio Bechini Dept. of Information Engineering, Univ. of Pisa

a.bechini@ing.unipi.it

Outline

Steps to uncover
the inner structure
of distributed computations,
and to reason about them

- Models of distributed computations
- On the Notion of time
- Timestamping

Modelling Distributed Computations

In un sistema distribuito non possiamo avere un orologio globale: è impossibile avere un singolo riferimento temporale poiché ogni nodo avrà il proprio riferimento temporale locale. Dovremo quindi implementare alcuni accorgimenti che ci consentiranno di basarci su alcune precedenze, e questi meccanismi sono noti come orologi logici.

Processes and Events

Our target system is made up of a (static) set of n *processes* $\{p_1, p_2, \ldots p_n\}$, each placed at a different node.
Nodes are connected by *channels*.
The execution of each process can be modeled by a sequence of **events**. Events can be:

- **Internal events** - significant actions, "state changes"
- **Communication events** - send/receive of a message (typically in an asynchronous way - but not necessarily)

Process History

Each single process is **sequential**, i.e. its events are executed one after the other, i.e. they are **totally ordered**.
Usually, the k-th event executed by process p_i is indicated as e^k_i

The sequence of all the events executed by a process is named as its **history**: $h_i = e^1_i, e^2_i, \ldots e^k_i \ldots$

A *prefix* of a process history corresponds to a progression point in the local execution.

Un "prefix" di una storia del processo è una porzione iniziale di quella storia. Ad esempio, se la storia completa è A, B, C, il prefisso potrebbe essere solo {A} oppure {A, B}.

Messages and Precedence

For any message m in the computation, handled in asynchronous communication, we can define an ordering on the related events:

$send(m)$ happens before $receive(m)$

(ovviamente non potranno mai avvenire al contrario)

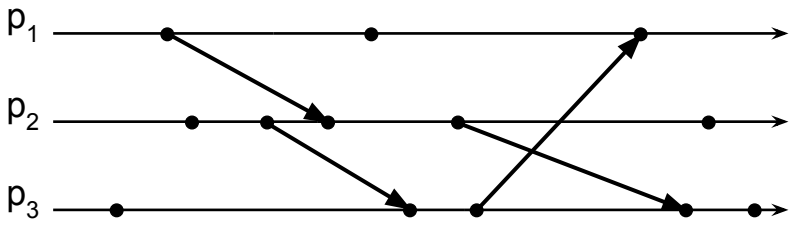
With other types of communication, e.g. with "synchronous" communications (rendez-vous), these two operations are considered to logically happen at the same time, so they cannot be totally ordered.

ovviamente questa è solo un'astrazione, in quanto nel codice avremo comunque un meccanismo di handshake

Le comunicazioni sincrone, o rendez-vous, sono un modello di comunicazione in cui due processi o agenti si incontrano in un punto specifico (rendez-vous) e si scambiano informazioni in modo sincronizzato. Questo significa che entrambi i partecipanti devono essere pronti e disponibili per la comunicazione al momento concordato.
In termini più pratici, immagina due processi A e B che desiderano scambiare dati. Con un'operazione di rendez-vous sincrona, A si ferma in attesa di B e viceversa. Quando entrambi sono pronti, avviene lo scambio di informazioni.

Space-Time Diagrams

A distributed execution can be graphically depicted with *space-time diagrams*:



Consistent Cuts (I)

A set of prefixes for all process histories is named **cut**.
The set of all the last events for each prefix is the cut *frontier*.
A **consistent cut** is a concept to formalize the notion of "possible progression point" of a whole distributed execution.

A cut is *consistent* if it satisfies the following property:

> for any message m such that $receive(m)$ is in the cut, $send(m)$ belong to the cut as well.

Ordering over (Consistent) Cuts

An ordering among cuts is plainly induced by the set inclusion relation!

chiaramente indotto

Given two cuts C_1, C_2 $\quad C_1 \rightarrow C_2 \ iff \ C_1 \subset C_2$

(precede) (è un sottoinsieme di)

Notably, some cuts could not be ordered...

Notion of Time

No Global Time/Clock

In a distributed system it is not possible to have clocks at all the nodes *in perfect synch*. Such a synchronization can be achieved within a certain *tolerance*.

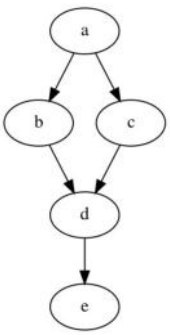
As a result, we cannot rely on a reference global time.

Without a global time, we need to understand **how to relate events executed at different progression points.**

Gli orologi logici sono un meccanismo per assegnare un ordine temporale coerente agli eventi in un sistema distribuito senza fare affidamento su un orologio fisico globale.
Ci sono diverse implementazioni di orologi logici, le più comuni sono gli orologi di Lamport e gli orologi vettoriali.

Towards Logical Time

The flow of control of a distributed execution is described by the **precedence relations** among its events.



Such precedence relations can substitute the notion of global time for the purpose of analyzing the structure of the overall execution.

The set of all the different types of precedences can be captured by a single relation to express the notion of *potential causality*.

La *causalità potenziale* è una relazione che cattura tutte le possibili relazioni di precedenza tra eventi. Non implica necessariamente che un evento abbia causato un altro, ma che esiste la possibilità che l'abbia fatto.

Happens Before (I)

Questa relazione è stata introdotta da Lamport per definire rigorosamente quando un evento può essere considerato come avvenuto prima di un altro.

One single relation can be defined to account for all precedences throughout the execution: $\xrightarrow{HB}$

durante

per tener conto di tutte

It can be constructed according to the following rules:

HB0 (transitivity) - for three events e, f, and g, if $e \xrightarrow{HB} f$ and $f \xrightarrow{HB} g$ then also $e \xrightarrow{HB} g$ holds

...

The "happens-before" relation is used to define the potential causal relationship between events in a distributed system.

NB: La relazione happens-before (→) permette solo un ordinamento parziale degli eventi.

Infatti, per molti eventi in un sistema distribuito, non c'è un modo per determinare l'ordine relativo. Ad esempio: Se l'evento A avviene in un processo e l'evento B avviene in un altro processo senza comunicazione tra i due, non possiamo dire se A→B o B→A. In questo caso, A e B sono considerati concorrenti.

Happens Before (II)

...

(ordine locale)

HB1 (in-process ordering) - for events e, f in the same process p_i, if $e \xrightarrow{i} f$ (avviene prima) then also $e \xrightarrow{HB} f$ holds

(passaggio di messaggi)

HB2 (asynch. comm.) - for any event $e = send(m)$ (non-blocking) and the corresponding event $f = receive(m)$ for the same m, $e \xrightarrow{HB} f$ holds

questa volta 'e' ed 'f' appartengono a processi diversi

HB and Concurrency

The definition of the “Happens Before” relation let us also formalize the notion of concurrency:

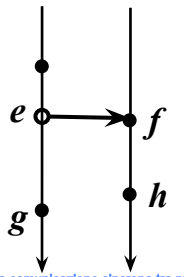
Two events e, f are said *concurrent*, denoted as $e \,||\, f$, iff

$$\underset{\text{NÉ}}{\neg}\left(e \xrightarrow{HB} f\right) \;\wedge\; \underset{\text{NÉ}}{\neg}\left(f \xrightarrow{HB} e\right)$$

Happens Before with Rendez-vous

In case synchronous communication is present as well, another rule can be added to the definition:

Remember: we had said that in the case of 'synchronous' communications (rendezvous), the send and receive are considered to logically happen at the same time.



Nella comunicazione sincrona tra processi, la send è tipicamente bloccante: il processo mittente si blocca fino a quando il msg non è stato ricevuto e processato dal processo destinatario.

HB3 (synch. comm.) - for any event $e = ssend(m)$ (blocking) and the corresponding event $f = receive(m)$ for the same m,
for any event g such that $e \xrightarrow{HB} g$ we have $f \xrightarrow{HB} g$ and
for any event h such that $f \xrightarrow{HB} h$ we have $e \xrightarrow{HB} h$

RIASSUNTO:
se g avviene prima di ssend(m), allora g avviene anche prima di receive(m),
se receive(m) avviene prima di h, allora ssend(m) avviene prima di h

HB Adding Shared Variables

With shared variables as well, HB can be extended in different ways, depending on the ordering assumed for the read/write operations on the same variable V.

Let $\xrightarrow{ob(V)}$ be the total ordering of reads/writes on sh. variable V.

HB4-strong - for two different events e, f for operations on V,
if at least one of them is a write event, then
if $e \xrightarrow{ob(V)} f$ then $e \xrightarrow{HB} f$

Weak HB with Shared Variables

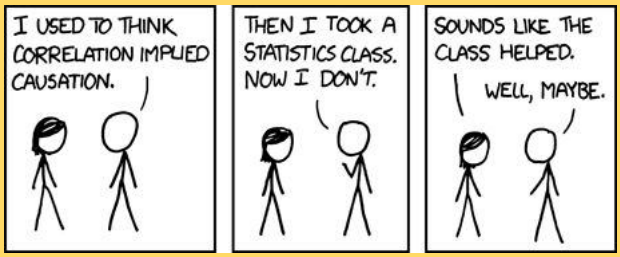
Another extension considers only the precedence of a read operation w.r.t. the previous write.
The version number of a variable V is incremented *at any write operation*.
$v(V,e)$ indicates the version number of V immediately after e.

HB4-weak - for a write event e and a read event f on the same V,
if $v(V,e) = v(V,f)$ then $e \xrightarrow{HB} f$

si può dire che e → f (scrittura precede lettura) solo se la lettura vede il valore scritto dall'evento di scrittura 'e'

Pause for Thought

Happens Before vs Causality

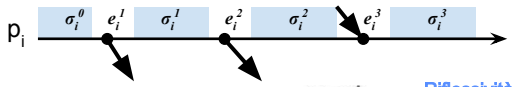


La potenziale causalità può essere usata per discutere in generale di dipendenze tra eventi, mentre la relazione "happens before" è specificamente utilizzata per ragionare sull'ordine temporale e le dipendenze nei sistemi distribuiti.

Events and Local States

- σ_i^0: Stato iniziale del processo p_i.
- $\sigma_i^1, \sigma_i^2, \sigma_i^3, \ldots$: Stati successivi del processo p_i, dove ogni transizione è causata da un evento $e_i^1, e_i^2, \ldots$.

A *local* computation corresponds to a sequence of *local states*, with transitions triggered by events.



Let define $a \xrightarrow{HB'} b$ as HB + reflexivity, i.e. $\forall\, a,\ a \xrightarrow{HB'} a$

estesa con la riflessività

Riflessività significa che "ogni evento precede se stesso".

We can state $\sigma_i{}^a \xrightarrow[\text{transizione}]{\sigma} \sigma_j{}^b \underset{\text{è equivalente}}{\equiv} e_i{}^{a+1} \xrightarrow{HB'} e_j{}^b$

La relazione $\sigma_i^a \xrightarrow{\sigma} \sigma_j^b$ viene utilizzata per indicare che c'è una transizione dallo stato σ_i^a allo stato σ_j^b causata dall'evento e_j^b, sotto la condizione che l'evento e_i^{a+1} (l'evento che causa la transizione da σ_i^a) precede e_j^b secondo la relazione HB'.

Questo evento e_j^b potrebbe essere il risultato di un messaggio ricevuto, un'azione interna, o qualsiasi altro tipo di evento.

From this definition, a distributed execution can be modeled as a partial order on the set of all states S: $(S, \xrightarrow{\sigma})$

Ipotizziamo che il processo p_i invia un messaggio, il suo stato cambia da (σ^a)i a (σ^a+1)i.
Il processo p_j, ricevendo questo messaggio, cambia il suo stato da da (σ^b)j a (σ^b+1)j.
La ricezione del messaggio è l'evento (e^b)j.

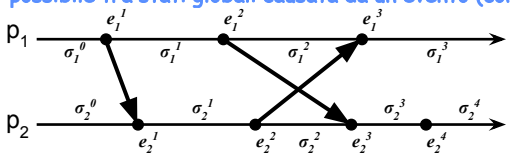
L'evento (e^a+1)i (invio del messaggio) avviene prima dell'evento (e^b)j (ricezione del messaggio).
Ciò significa che l'evento di invio è visto come precedente all'evento di ricezione, garantendo coerenza temporale e logica tra i due processi.

NON TORNA, NON CAPISCO. dovrei rivedere registrazione

raggiungibilità

Global States and Reachability Graph

Un Reachability Graph (grafo di raggiungibilità) è un grafo che rappresenta tutte le possibili transizioni di stato di un sistema. Ogni nodo nel grafo corrisponde a un possibile stato globale del sistema, e ogni arco rappresenta una transizione possibile tra stati globali causata da un evento (come l'invio o la ricezione di un messaggio).



A "Global State" is a collection of one local state per process:

$\Sigma = [\sigma_i, \sigma_i, \ldots \sigma_n]$

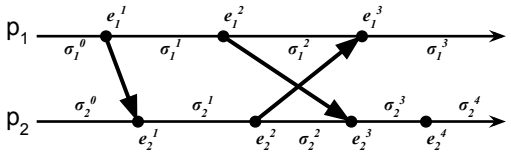
A GS Σ is *consistent* if $\sigma_i \,||\, \sigma_j \;\forall\; i, j, i \neq j$

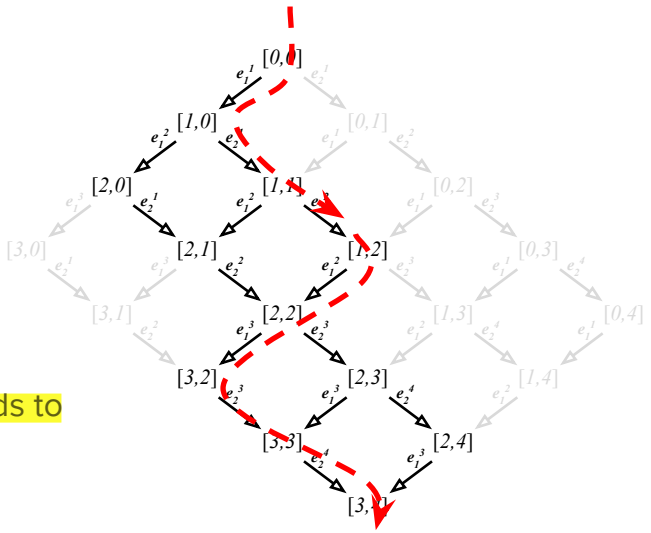
gli stati in grigio sono stati che non corrispondono a dei progression point, sono stati irraggiungibili che quindi non considero

Se σ1||σ2, significa che le azioni che hanno portato p1 in σ1 e p2 in σ2 sono state eseguite indipendentemente, senza alcuna comunicazione diretta tra i due processi che implichi che uno sia avvenuto prima dell'altro.

→ Se due processi hanno scambiato almeno un messaggio tra di loro, i loro stati locali successivi all'invio e alla ricezione del messaggio non possono essere concorrenti. Di conseguenza, un global state che include questi stati locali non può essere consistente secondo la definizione data, poiché esiste una relazione di "happens before" tra gli stati locali coinvolti.

Sequential Observations





A **sequential observation** is a *sequence* of all events/states, that complies with their partial order.

che rispetta

A sequential observation corresponds to a path in the reachability graph.

Clocks for Logical Time

HB and Logical Clocks

If we need to deal with the relative orderings of event occurrences, we can substitute the "global time" with an index that relates to the ordering of events.

A **logical clock** $C(\cdot)$ is a means to map events onto a partial order, so that $e \xrightarrow{HB} f$ implies $C(e) < C(f)$ (*clock consistency property*)

The simplest co-domain for $C(\cdot)$: sequence of increasing integers.
In this case, the corresponding time can be said *linear time*.

Lamport Timestamps

La lunghezza della catena di eventi che portano a un dato evento determina il suo timestamp.

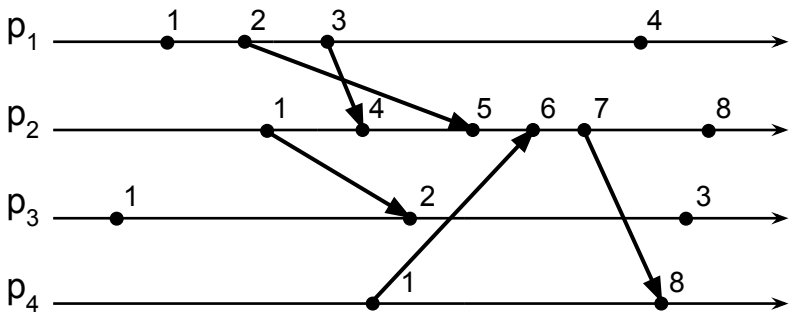
A classical logical clock T_{Lp}: an event e is associated with the length of the longest HB chain to reach e. How to assign such timestamps?

Each process p_i keeps a local counter $clock_i$;
It is initialized to 0, and updated according to the following rules:

1. for all events but *receive*s, $clock_i \leftarrow clock_i + 1$ and then $T_{Lp}(e) \leftarrow clock_i$ (eccetto)
2. On *send* events, its value is piggybacked in msg m - call it $ts(m)$
3. With *receive*s, $clock_i \leftarrow \mathbf{max}(clock_i, ts(m)) + 1$ and then $T_{Lp}(e) \leftarrow clock_i$

Dopo l'incremento, il timestamp logico dell'evento è impostato al valore corrente di clock_i.

Lamport Timestamps - Example



nel mettere i numerini devo solo fare attenzione alle receive, che devo aggiornare con valore = max(clock_i , ts(m)) +1

Lamport Timestamps - Properties

First, the clock consistency: $e \xrightarrow{HB} f$ implies $T_{Lp}(e) < T_{Lp}(f)$ by construction

T_Lp, 'L' sta per Lamport mentre 'p' per partial

! ma non il contrario

By contraposition, $T_{Lp}(e) \leq T_{Lp}(f)$ implies $\neg\left(f \xrightarrow{HB} e\right)$

I.e., if $T_{Lp}(e) \leq T_{Lp}(f)$, either e happened before f , or $e \,||\, f$.

Moreover, $T_{Lp}(e) = T_{Lp}(f)$ implies $e \,||\, f$

Problem: it is possible to have $(T_{Lp}(e) < T_{Lp}(f)) \wedge \neg\left(e \xrightarrow{HB} f\right)$ ☹

This means that T_{Lp} cannot be used to check precedence/causality!

Lamport Timestamps - Extensions

The algorithm for Lamport timestamps can be easily extended to deal with synchronous communication and use of shared variables.

Totally Ordered Lamport Timestamps

The linear time T_{Lp} can be used to obtain a total order T_{Lt} over all the events, such that it would be *consistent* with the HB relation.

Quello che voglio ottenere è un total order a partire dal grafico di pag 13. Decido quindi che tra tutti gli eventi con T_Lp pari ad 1, il primo in ordine temporale è quello del processo 1 e così via...

For process p_i, T_{Lt} can be defined as the pair (T_{Lp}, i) and the total order states that $(ts_a, i) < (ts_b, j) \text{ iff } (ts_a < ts_b) \vee ((ts_a = ts_b) \wedge (i<j))$ (OR, AND)

In practice, T_{Lt} can be an integer *conveniently* obtained as $T_{Lp} \ll B + i$, with $B = \lceil log_2 n \rceil$ and n number of processes ($\ll B$: shift left of B bits).

invece di confrontare una tupla, posso "codificare" le due informazioni (T_Lp , i) in un unico numero intero, così facendo l'ordine totale sarà più facile da leggere e confrontare.

NON FARTI FREGARE: i "Totally Ordered Lamport Timestamps" possiedono effettivamente la proprietà di "total order". Al contrario, il prossimo tipo di timestamp che vedremo, i "Vector Timestamps", non possiede la proprietà di "total order", ma ha la proprietà di "Strong Clock Consistency".

Towards Strong Clock Consistency

We'd like to have a clock able to indicate causal dependence,

i.e.: $C(e) < C(f) \Rightarrow e \xrightarrow{HB} f$

Possible solution: keep a ***vector*** counter V_i of n integers at each process p_i, so that $V_i[i]$ is the counter of events within p_i, and $V_i[j]$ corresponds to the most recent value of $V_j[j]$ as detected by p_i.

Whenever p_i sends a message, the most recent value of V_i is piggybacked in the message.

- V_i è il vettore di timestamp (vector clock) del processo pi.
- V_i[j] rappresenta il valore del contatore del processo pi relativo al processo pj.

Il meccanismo è quindi molto simile a quello proposto da Lamport, ma invece di mantenere un singolo counter per processo, ora ogni processo deve mantenere n counters organizzati in un array.

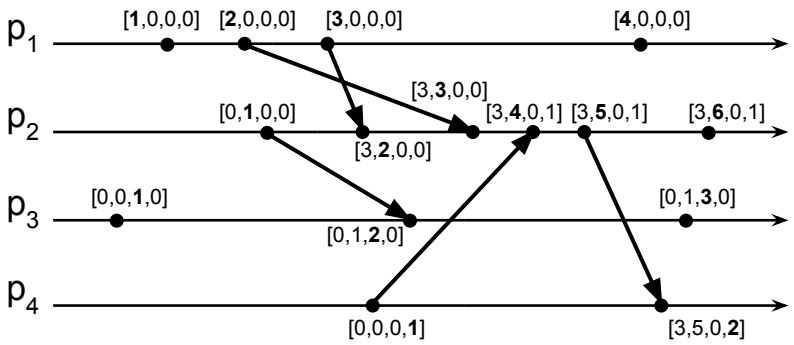
Algorithm for Vector Timestamps

At each event occurrence, the local vector counter V_i is updated, and its value is assigned to the event as its *vector* timestamp $T_V(e)$.

How to update local vector counters?

1. For all events, in the first place $V_i[i] \leftarrow V_i[i]+1$
2. On *send* events, $T_V(e)$ is piggybacked in msg m - call it $ts(m)$
3. On *receive* events, $V_i \leftarrow max_{compwise}(V_i, ts(m))$
4. Finally, $T_V(e) \leftarrow V_i$

Vector Timestamps - Example



Per le receive, il V lo calcolo come:
faccio la max_compwise() tra l'ultimo V del processo che riceve
ed il V piggybacked nel messaggio, infine aggiungo 1 al V_i[i]

How to Compare Vector Timestamps

Given two vector timestamps T_1 and T_2, we define:

$$T_1 \le T_2 \quad iff \quad T_1[k] \le T_2[k] \quad \forall k \in [1, \ldots n]$$

$$T_1 < T_2 \quad iff \quad T_1 \le T_2 \wedge T_1 \neq T_2$$

cioè almeno una componente deve essere diversa

$$T_1 \parallel T_2 \quad iff \quad T_1 \not< T_2 \wedge T_2 \not< T_1$$

In altre parole, dobbiamo verificare che nessuno dei seguenti sia vero:
- T1≤T2
- T2≤T1

The last two definitions relate to HB precedence and concurrency, as it will be shown. (sinceramente non ho capito dov'è che lo mostra)

Vector Timestamps - Properties

Clock consistency: $e \xrightarrow{HB} f$ implies $T_V(e) < T_V(f)$ by construction

Strong C.C. : $T_V(e) < T_V(f)$ implies $e \xrightarrow{HB} f$

! stavolta è vero anche il viceversa

Proof: by contraposition, let's show $\neg\left(e \xrightarrow{HB} f\right)$ implies $T_V(e) \not< T_V(f)$

$T_V(e) \not< T_V(f)$ means that, for at least one position i, $T_V(e)[i] > T_V(f)[i]$

Say e is in p_i, and f in a different p_j.

Just before e, p_i increases its $V_i[i]$, say to value t, thus $T_V(e)[i] = t$.

As $\neg\left(e \xrightarrow{HB} f\right)$, there is no way for value t to propagate to p_j,

thus $T_V(f)[i] < t$ □

DIMOSTRAZIONE NON CAPITA :-(

Cons. Cuts & Vector Timestamps

Vector Timestamps let us check the consistency of a cut.

Let $F(C) = \{f_1, ...f_n\}$ be the set of events in the frontier of a cut C.

Let the "cut timestamp" be $\boldsymbol{T}_V(C) = max_{compwise}(\boldsymbol{T}_V(f_1), \ ... \ \boldsymbol{T}_V(f_n))$

C is consistent if $\forall i \ , \ \boldsymbol{T}_V(C)[i] = \boldsymbol{T}_V(f_i)[i]$

In poche parole

In a nutshell, the maximum i-th value must always be on the i-th process/position: **no other process in the cut knows my future!**

cioè, mi basta controllare i ts V dell'ultimo evento in ogni processo per poter dire se un taglio è consistente oppure no

Cuts & Vector Timestamps - Check!