



# Towards Logical Time

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

a.bechini@ing.unipi.it

© A. Bechini 2023



## 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

© A. Bechini 2023

# Modelling Distributed Computations

## Processes and Events

Our target system is made up of a (static) set of  $n$  processes  $\{p_1, p_2, \dots, p_n\}$ , 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  $e_i^k$

The sequence of all the events executed by a process is named as its **history**:  $h_i = e_i^1, e_i^2, \dots e_i^k \dots$

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

# Messages and Precedence

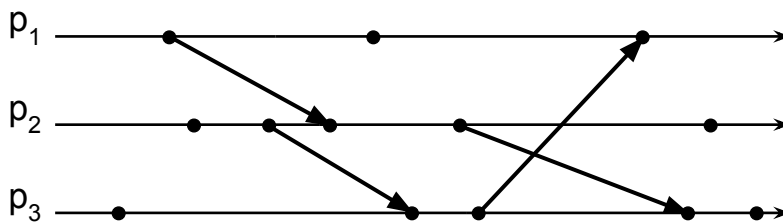
For any message  $m$  in the computation, handled in asynchronous communication, we can define an ordering on the related events:

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

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.

# Space-Time Diagrams

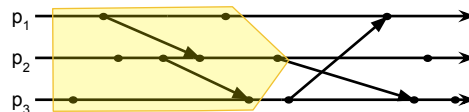
A distributed execution can be graphically depicted with *space-time diagrams*:



© A. Bechini 2023

A. Bechini - UniPi

## 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.

© A. Bechini 2023

A. Bechini - UniPi

# Ordering over (Consistent) Cuts

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

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



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.

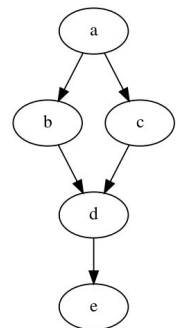


Without a global time, we need to understand **how to relate events executed at different progression points.**

## 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*.

# Happens Before (I)

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

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

...

# Happens Before (II)

...

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

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

# HB and Concurrency

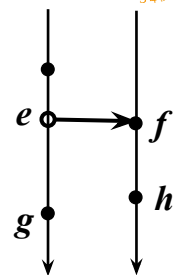
The definition of the “Happens Before” relation let us also formalize the notion of concurrency:

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

$$\neg(e \xrightarrow{HB} f) \wedge \neg(f \xrightarrow{HB} e)$$

## Happens Before with Rendez-vous

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



**HB3** (synch. comm.) - for any event  $e = \text{ssend}(m)$  (blocking)

and the corresponding event  $f = \text{receive}(m)$  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  $e \xrightarrow{HB} h$



# 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.

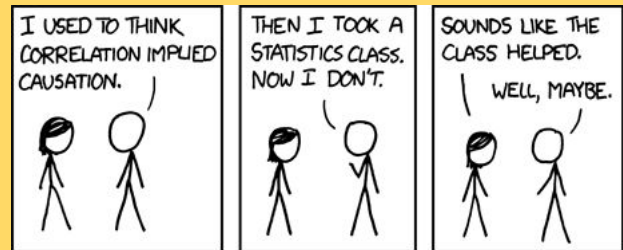
The version number of a variable  $V$  is incremented  
*at any write operation.*

$v(V, e)$  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  $e \xrightarrow{HB} f$

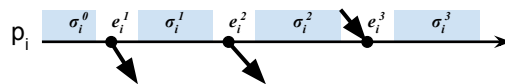
# Pause for Thought

## Happens Before vs Causality



## Events and Local States

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

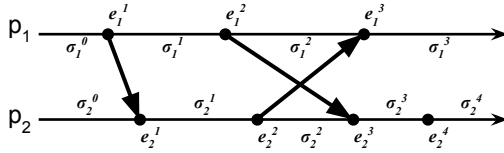


Let define  $a \xrightarrow{HB'} b$  as HB + reflexivity, i.e.  $\forall a, a \xrightarrow{HB'} a$

We can state  $\sigma_i^a \xrightarrow{\sigma} \sigma_j^b \equiv e_i^{a+1} \xrightarrow{HB'} e_j^b$

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

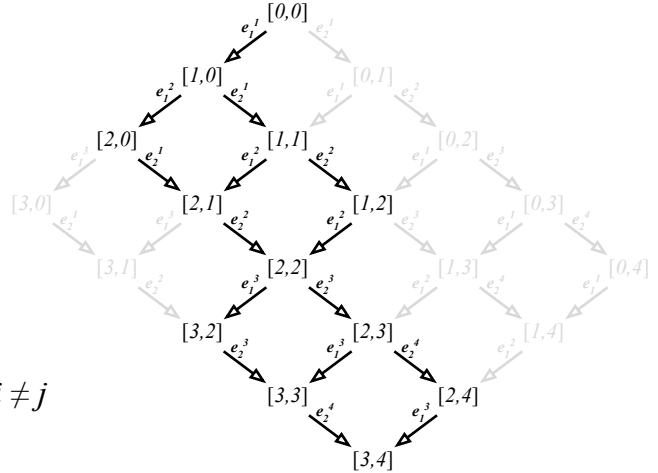
# Global States and Reachability Graph



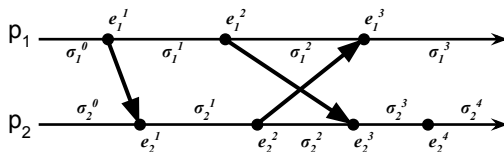
A “Global State” is a collection of one local state per process:

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

A GS  $\Sigma$  is *consistent* if  $\sigma_i \parallel \sigma_j \forall i, j, i \neq j$

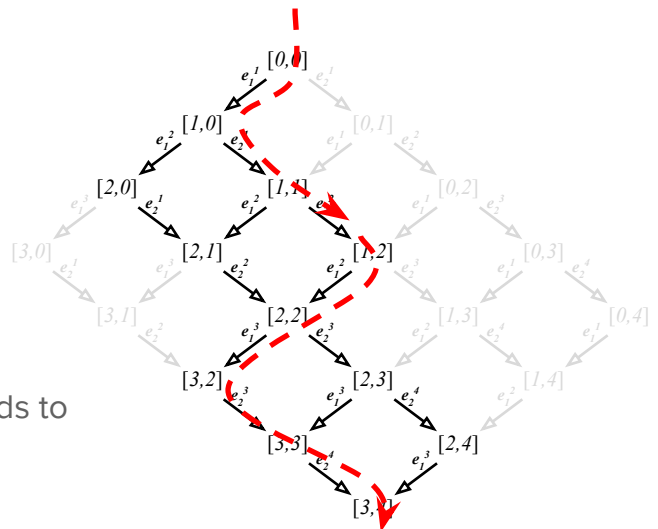


# Sequential Observations



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

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

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  $clock_i$ ;

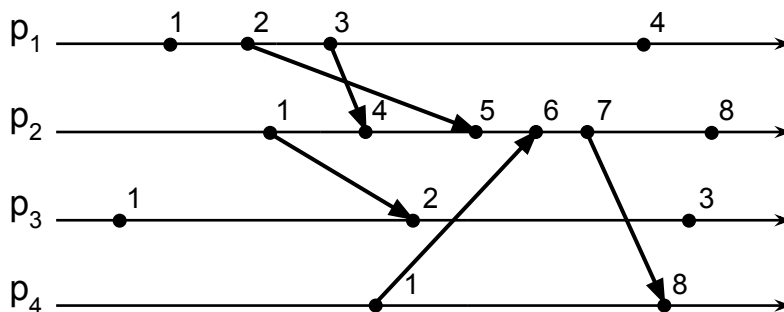
It is initialized to 0, and updated according to the following rules:

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

© A. Bechini 2023

A. Bechini - UniPi

## Lamport Timestamps - Example



© A. Bechini 2023

A. Bechini - UniPi

# Lamport Timestamps - Properties

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

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

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

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

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



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.

For process  $p_i$ ,  $T_{Lt}$  can be defined as the pair  $(T_{Lp}, i)$  and the total order states that  $(a, b) < (c, d)$  iff  $(a < c) \vee ((a = c) \wedge (b < d))$

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).

# 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.

# 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  $T_V(e)$ .

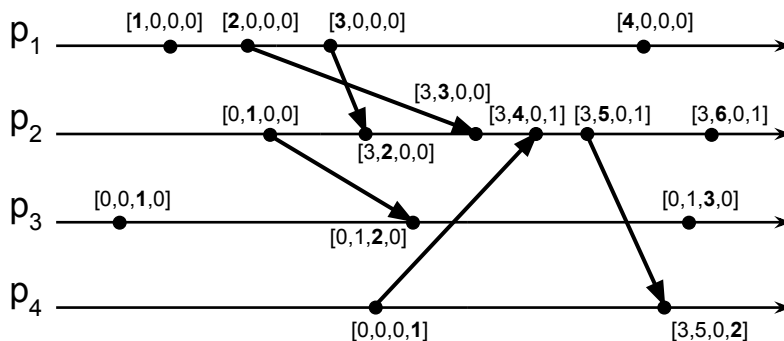
How to update local vector counters?

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,  $T_V(e) \leftarrow V_i$

© A. Bechini 2023

A. Bechini - UniPi

## Vector Timestamps - Example



© A. Bechini 2023

A. Bechini - UniPi



# How to Compare Vector Timestamps

Given two vector timestamps  $T_1$  and  $T_2$ , we define:

$$T_1 \leq T_2 \quad \text{iff} \quad T_1[k] \leq T_2[k] \quad \forall k \in [1, \dots, n]$$

$$T_1 < T_2 \quad \text{iff} \quad T_1 \leq T_2 \wedge T_1 \neq T_2$$

$$T_1 \parallel T_2 \quad \text{iff} \quad T_1 \not< T_2 \wedge T_2 \not< T_1$$

The last two definitions relate to HB precedence and concurrency, as it will be shown.

## 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  $e \xrightarrow{HB} f$

*Proof:* by contraposition, let's show  $\neg(e \xrightarrow{HB} f)$  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  $\neg(e \xrightarrow{HB} f)$ , there is no way for value  $t$  to propagate to  $p_j$ ,

thus  $T_V(f)[i] < t$

□

# Cons. Cuts & Vector Timestamps

Vector Timestamps let us check the consistency of a cut.

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

Let the “cut timestamp” be  $T_V(C) = \max_{\text{compwise}}(T_V(f_1), \dots, T_V(f_n))$

$C$  is consistent if  $\forall i, T_V(C)[i] = T_V(f_i)[i]$



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!**

## Cuts & Vector Timestamps - Check!

