Faculty of Information Engineering, Informatics and Statistics
Department of Computer Science

Distributed Systems

Author:
Simone Lidonnici

26 september 2025

Contents

1
Introduction

1.1 Defining a distributed system

We define a **distributed system** as a collection of processes $p_1, p_2, \ldots, p_n$ that run on different computers and cooperate to solve a problem. The processes comunicate using **channels** and we assume the system is fully connected, meaning that every pair of processes can excange messages between them. The channels are reliable, in the sense that te message arrive, but may delivered messages out of order.
The simple model for a distributed system is called **asynchronous**, that have no upper bound on the speed of processes and no upper bound for the delay of a message. If these upper bounds exists the system is called **synchronous**. The second one is stronger, in the sense that we do more assumpions and this means that every program that run on a synchronous system can run on an asynchronous system (the opposite can be possible but not sure).
A distributed system can have different properties:

- **Consistency**: every part of the system has the same information at every time
- **Avilability**: the information are available at every time
- **Partiotion tolerance**: if one part of the system goes offline the system can continue to run

Every type of distributed system can have up to 2 of this properties.

1.2 Computation

We describe the execution of a program on a distributed system as a collection of processes. Every process is defined as a sequence of **events**. The events can be internal or involve comunication like the events `send(m)` and `receive(m)`.
We label an event with the notation:

$$e_i^k$$

in which i represent the index of the process p_i and k the order of the event for that specific process.

Local and global history

The **local history** of a process p_i is a sequence of events $h_i = e_i^1 e_1^2 \dots$ that represent the sequencial execution of events in the process. We use h_i^k to represent the sequence of the first k events in the process p_i.
The **global history** of the computation is a set that contains all events:

$$H = h_1 \cup h_2 \cup \dots \cup h_n$$

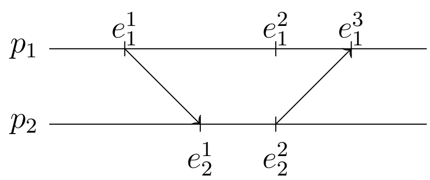
The global history gives us no information about the time in which these events are executed, because in an asynchronous system there is no global clock.

Relation of cause-effect

Events can be labeled based of the notion of **cause-effect**, defining a relation (with symbol $\rightarrow$) between two events such that:

- $\forall e_i^k, e_i^l \in h_i \wedge k < l \implies e_i^k \rightarrow e_i^l$
- $e_i = \texttt{send(m)} \wedge e_j = \texttt{receive(m)} \implies e_i \rightarrow e_j$
- $e \rightarrow e' \wedge e' \rightarrow e'' \implies e \rightarrow e''$ (Transitive)

This relation mean that $e \rightarrow e'$ if e causally precedes e', so the computation of e' is influenced by e. Two events can be unrelated, so neither $e \rightarrow e'$ nor $e' \rightarrow e$. We call this pair of events as **concurrent** and write them as $e||e'$.
We can graphically represent a computation with a space-time diagram like this:



An arrow from p_1 to p_2 means that p_1 sends a message to p_2.

Run

A **run** is a total order of all events in the global history, consistent with each local history, so the events in history h_i appear in the same order in R:

$$R = e_1^1 e_2^1 \dots$$

A single program can have many different runs because some events are unrelated.

1.3 Monitoring computations

Local and global state

We denote σ_i^k the **local state** of a process p_i after the event e_i^k.
The **global state** of the computation is an n-tuple of local states:

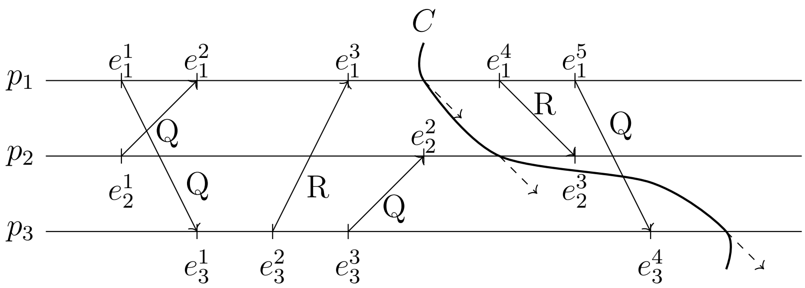
$$\Sigma = (\sigma_1, \sigma_2, \dots, \sigma_n)$$

Cut

A **cut** is a collection of local histories:

$$C =< h_1^{k_1}, h_2^{k_2}, \dots, h_n^{k_n} >$$

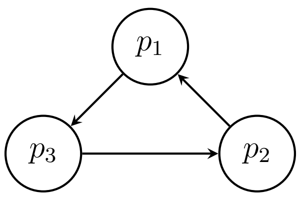
To monitor the computation or to compute a global problem (for example knowing if the system is in deadlock) we add a process p_0.
The first idea is to make p_0 send a message to every process to which a process p_i will respond with the current state σ_i. After all the response p_0 can construct a global state, that defines a cut.
Example:
Considering the follows space-time diagram with the cut C (the dashed arrow represent when the process sends the response to p_0):



If we use the responses to create a graph, based on the states, we can say that:

- p_1 is going to send a response to p_2
- p_2 is going to send a response to p_3
- p_3 is going to send a response to p_1

So the graph generated by p_0 will be:



It seems the system has a deadlock, but if we watch carefully we can see that this is not true. The problem is that the global state defined by the cut C could never happen during a computation.

Consistent cut

A cut is **consistent** if only if:

$$\forall e \rightarrow e' \wedge e' \in C \implies e \in C$$

So a cut is consistent if the global state generated by the cut could be possible during a computation.

If we use p_0 only to receive messages and we make every process notify the events to p_0 with a timestamp associated, p_0 can reorder the events to create a run. This is true only if there is a global clock, which is not true. To overcome this problem we have to create a local clock for every process and update it in a consisten way.

Clock condition

A run is consistent if only if follows the **clock condition**:

$$\forall e \rightarrow e' \implies TS(e) < TS(e')$$

If $TS(e) < TS(e')$ is possible but not garanteed that $e \rightarrow e'$.

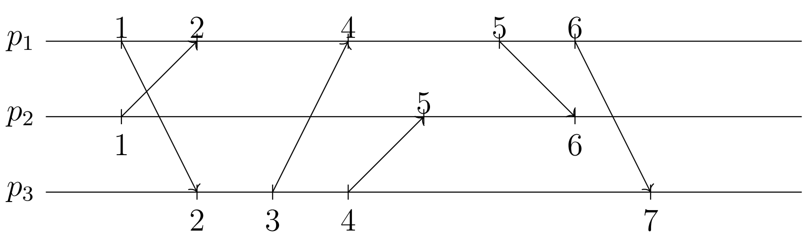
Local clock

We define the **local clock** of a process as follows:

$$LC(e_i) = \begin{cases} LC + 1 & e_i \text{ is internal or } \texttt{send} \\ \max(LC, TS(m)) + 1 & e_i \text{ is a } \texttt{receive(m)} \end{cases}$$

The local clocks satisfy the clock condition.

Example:
In the previous example the timestamps of the local clocks will be:



1.4 Vector clocks

We want to ensure that the process p_0 sends the messages from the network level to the upper level in order, so the communication between two processes has to be FIFO (First-In First-Out). To do that in a synchronous system with a global clock and an upper bound for a message Δt, we can follow a simple rule fro p_0:

- On a certain time t, deliver all the message with timestamp lower than $t - \Delta t$ in order based on the global clock.

In an asynchronous system we have to decide when to deliver a message, so we have to be sure that all the previous message are already delivered. We could wait for every other process to send to p_0 every notification with local timestamp lower than the one received before, but this could stuck the system if a process doesn't have enough events to notify.

Strong clock condition

We expand the definition of clock condition changing the implication with a if only if:

$$e \to e' \iff TS(e) < TS(e')$$

History of an event

We define the **history** of an event as a set:

$$H(e) = \{e' | e' \to e\} \cup \{e\}$$

This includes all the previous events that could have changed the result of event e.

We will identify an history of a process e with a vector in which every index i represent the last event of process p_{i+1} in $H(e)$. For example the history $H(e_1^4) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_2^1, e_3^1, e_3^2, e_3^3\}$ is identified by the vector $[4, 1, 3]$. This vector clocks are updated for a process p_i following the rule:

$$VC = \begin{cases} VC[i] = VC[i] + 1 & \text{always} \\ VC[j] = \max(VC[j], TS(m)[j]) \ \forall j \neq i & e_i \text{ is a } \texttt{recieve(m)} \end{cases}$$

These clocks respect the strong clock condition.

Example:

Using this new vector clocks in the previous example the result is: