Lecture notes: Studying distributed systems – Motivations and the ordering of events

M2 MOSIG: Distributed Systems

Thomas Ropars

2025

These notes present the motivations of the course. It describes basic concepts related to distributed systems and then, discusses the ordering of events in a distributed systems[1].

1 Introduction

1.1 Definitions

We start by defining the concept of distributed system. As a starting point we can quote L. Lamport:

Definition 1 (Distributed systems – by L. Lamport)**.** *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*

Our definition of a distributed system involves two main caracteristics:

Definition 2 (Distributed systems – for this course)**.** *A distributed system is:*

- *A system composed of multiple processes that seek to achieve some form of cooperation.*
- *A system in which some processes may stop working while the others might keep operating (notion of* partial failures*).*

In a distributed system, processes communicate by sending and receiving messages.

1.2 Motivations

This part is inspired from the Chapter 1 of the book of Cachin *et al.* [1]. The goal is to get an overview of the motivations for distributed systems.

The basic form of distributed computing that we are all used to is *client-server*. This is what happens when you are browsing the web for instance: your browser (the client) connects to a web server to retrieve some content. In this case, the *server* is a centralized process that provides a service to many remote clients. The interactions are in the form of request-reply. The clients issue requests to the server, that replies.

[1] Acknowledgments: Parts of these notes are strongly inspired by the lectures notes of Andre Schiper on *Distributed Algorithms*.

With this simple form of distributed computing, we can already anticipate some problems in the presence of partial failures: how to ensure that clients will keep receiving answers to their requests if the server stops working?

In practice, it is often the case that more than two processes need to cooperate to achieve a goal, which makes the problem even more difficult.

Inherent distribution Considering the case of web applications, the server that is contacted by a client through a request, might need the help of other servers to answer that request. Indeed, such applications are often divided into multiple services, each service being in charge of a simple task: authentication, data management, account management, payment, etc. This separation of concerns has a lot of advantages regarding the development of the application logic.

Other applications are distributed by nature. Here are a few examples:

- A communication system where the sources and destination of information are geographically distributed
- A colaborative document editing application
- Distributed databases/storage where the volume of data to store is so big that it requires several machines

Distribution as an artefact Some applications are not inherently distributed but rely on distributed computing to solve some engineering problems. *Fault tolerance* is a reason for building a distributed system. Having a single copy of an application running is a problem if the computing unit on which it runs fails. An alternative is to run multiples replicas of the application on different severs.

Performance can be another motivation for building a distributed system. If a single server becomes overloaded and takes time to answer requests from clients, replication can again be a solution.

2 Model of a distributed computation

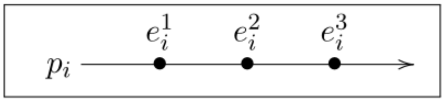
To be able to reason about distributed systems, we need to define a *model* that allows us capturing the relevant caracteristics of the systems we study.

A distributed computation is generated by some distributed algorithm. A distributed computation can be modelled as follows. The basic components of a distributed computation are *processes* and *channels*:

- *Processes* are denoted by $P = \{p_1, p_2, \ldots, p_n\}$. Processes communicate exclusively by message exchange.
- *Channels* (one-way channels) are denoted by c_{ij}: c_{ij} is the channel from $p_i \in P$ to $p_j \in P$. The set of channels is denoted by C. In this course, channels are assumed to be reliable (no message loss, no message corrupted, no message duplicated).

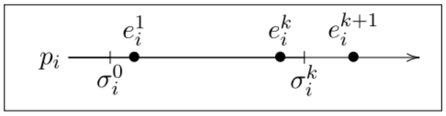
The activity of each process $p_i \in P$ is modelled as a sequence of events. Event #k of process p_i is denoted by e_i^k.

Event e_i^k can be:



1. send(m): sending of message m
2. receive(m): reception of message m
3. an internal event (all other events, e.g., $x \leftarrow x + 1$).

Events may modify the state of the process. The *state* of process p_i after the occurrence of e_i^k is denoted by σ_i^k. The initial state of p_i is denoted by σ_i^0.



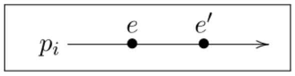
Definition 3 (Local history of process p_i)**.** *Defined as the (possibly infinite) sequence of events of p_i: $h_i \overset{def}{=} e_i^1 e_i^2 e_i^3 \ldots$.*
The prefix (of length k) of h_i is denoted by: $h_i^k \overset{def}{=} e_i^1 e_i^2 \ldots e_i^k$

Definition 4 (Global history:)**.** $H = h_1 \cup \ldots \cup h_n$

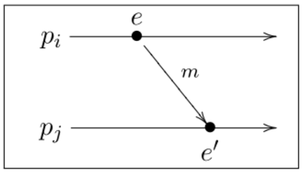
The basic relation among events is called the *happened before* relation. The relation was defined by Lamport in 1978 [3].

Definition *Happened before* relation (denoted by $\rightarrow$):
Let e, e' be two events. $e \rightarrow e'$ holds iff one of the following three conditions is true:

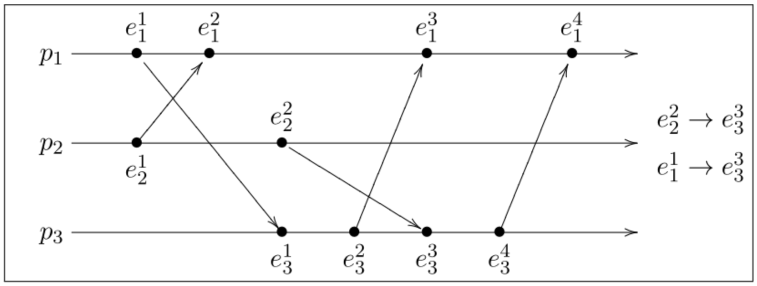
1. $e \equiv e_i^k, e' \equiv e_i^l, k < l$



2. $e \equiv \text{send}(m), e' \equiv \text{receive}(m)$



3. Transitive closure:
$\exists e''$ such that $e \rightarrow e''$ and $e'' \rightarrow e'$



The relation *happened before* is:

- *antisymmetric:* $e \to e' \Rightarrow e' \not\to e$
- *transitive:* $e \to e'$ and $e' \to e'' \Rightarrow e \to e''$

Therefore, the *happened before* relation defines a *partial order* on the events of a distributed computation. The relation does not define a *total order* (it would require, for all events e, e', to have either $e \to e'$ or $e' \to e$).

Definition *Concurrent* events: Let e, e' be two events such that neither $e \to e'$ nor $e' \to e$ holds. The events e, e' are said to be *concurrent* (notation: $e||e'$).

In the above figure, we have $e_2^2||e_3^1$.

Definition A *global state* of a distributed computation is defined by the tuple $(\sigma_1^{k_1}, \sigma_2^{k_2}, \ldots, \sigma_n^{k_n})$.

Note that the *global state*, as defined here, does not include the state of the communication channels.

3 Cuts

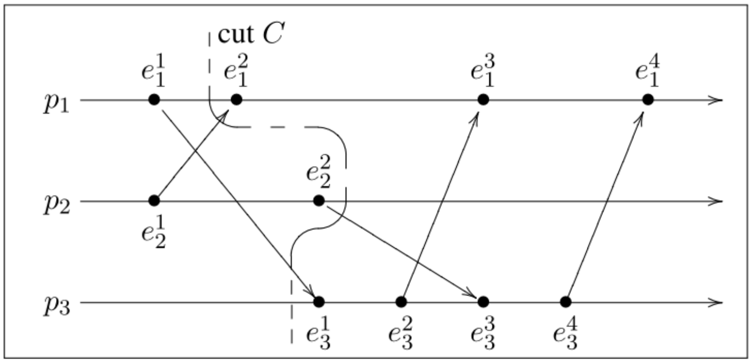
A *cut* C of a distributed computation is defined as a subset of the global history H that includes a prefix of each local history:

$$C \overset{def}{=} h_1^{ct_1} \cup h_2^{ct_2} \cup \ldots \cup h_n^{ct_n}.$$

A cut C

- is defined by a tuple $(ct_1, ct_2, \ldots, ct_n)$,
- defines the global state $(\sigma_1^{ct_1}, \sigma_2^{ct_2}, \ldots, \sigma_n^{ct_n})$

Example: cut C defined by the tuple $(1, 2, 0)$.



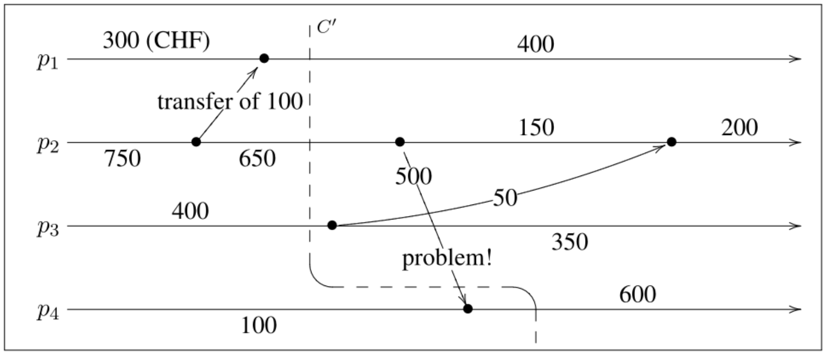
4 Consistent cut

Let us consider a scenario where we want to evaluate a global property on a distributed system. For instance, we would like to evaluate the total amount of money available on all bank accounts in a distributed banking application. To do such computation, we need a global state of the system.

We start by considering a *naive* strategy to compute a global state.

We start with the following naive strategy to compute a global state. Let a process p_0, outside of the system, ask each process p_i its local state σ_i. To do this, p_0 sends a request to each process p_i and waits a response containing p_i's local state. Once all responses are obtained, process p_0 builds the global state $GS = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ and evaluates the property on GS.

Let us consider the example below, where each process manages one bank account, and messages are used to transfer money between the bank accounts.



Consider the global state defined by the cut C': (400, 650, 400, 600). The evaluation of the global property on C' leads to 2050, which obviously is incorrect.[2]

What happened? *The cut is not consistent.*

Definition *Consistent cut:* a cut C is consistent iff for all events e, e' we have

$$e' \in C \textit{ and } (e \rightarrow e') \Longrightarrow e \in C. \tag{1}$$

Definition *Consistent global state:* a *consistent* global state is a global state defined by a consistent cut.

Note that if we evaluate a property on a consistent cut, we do not take into account messages in-transit. We will come back to this point soon.

Remark 1: Formula (1) is equivalent to

$$e \notin C \textit{ and } (e \rightarrow e') \Longrightarrow e' \notin C.$$

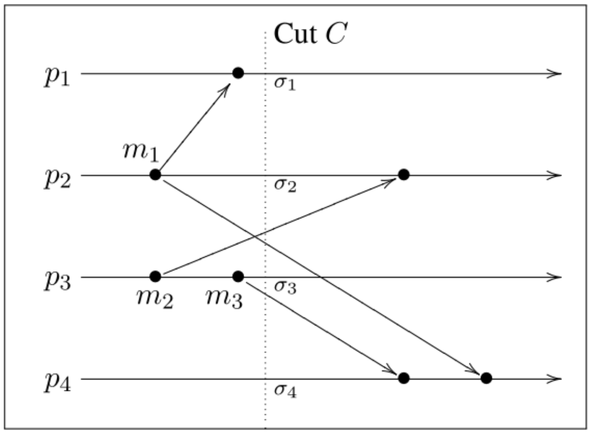
This follows from $a \Rightarrow b \equiv \neg a \textit{ or } b$.

Remark 2: Consider cut C that is not consistent. It can be shown that there exists message m such that *send*$(m) \notin C$ and *receive*$(m) \in C$.

[2] In the initial state the total amount of money is equal to 300 + 750 + 400 + 100 = 1550. Because of money in transit, the amount can be less than 1550, but never more.

5 Computing a consistent global state (snapshot)

We describe now an algorithm for computing a consistent global state (algorithm proposed by Chandy and Lamport [2]). The algorithm not only computes a consistent global state but also the state of the channels. Consider the following figure.



On the consistent cut C we have:

- state of the processes: $(\sigma_1, \sigma_2, \sigma_3, \sigma_4)$,
- state of channel c_{32}: contains message m_2,
- state of channel c_{24}: m_1,
- state of channel c_{34}: m_3,
- all other channels are empty.

Chandy-Lamport snapshot algorithm: The algorithm assumes FIFO channels. The snapshot algorithm is initiated by any of the n processes (e.g., p_1):

1. Process p_1 (the initiator of the snapshot) saves its state σ_1 and broadcasts the message SNAPSHOT to P (the set of processes).

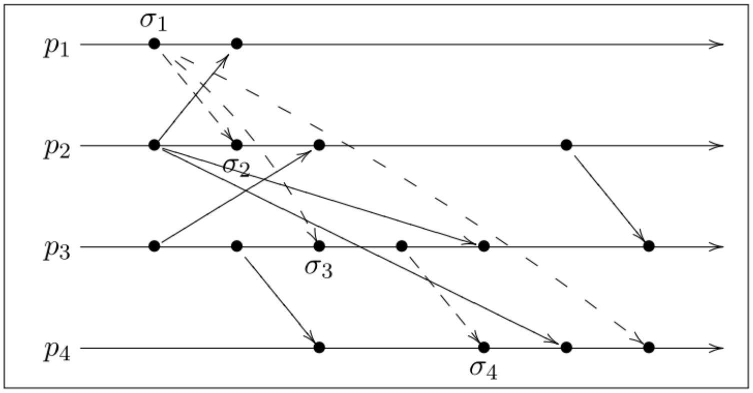
2. Let process p_i receive the SNAPSHOT message the first time from some process p_j (p_j can be different from p_1). At that time, p_i saves its state σ_i and forwards (broadcast) the SNAPSHOT message to P.

   No application event can take place on p_i between the reception of SNAPSHOT and the broadcast of SNAPSHOT. Moreover:

   - The state of the channel c_{ji} (from p_j to p_i) is set to empty.
   - For all $k \neq j$, the state of the channels c_{ki} is initialized to empty. Later, whenever a message is received from p_k, it is added to the state of channel c_{ki}.

3. When p_i receives SNAPSHOT from p_k, the computation of the state of c_{ki} is complete. As soon as p_i has received SNAPSHOT from all the processes in P, the computation of the snapshot is terminated.

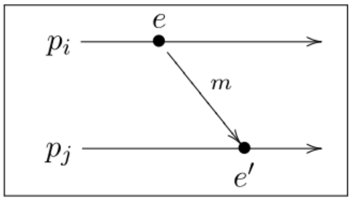
The algorithm is illustrated on the next figure. The SNAPSHOT message is represented by the dashed line. Only the SNAPSHOT message send by p_1, as well as the first SNAPSHOT message received by each process is drawn; moreover, only the computation of the global state is depicted. Observe that p_4 receives the first SNAPSHOT message from p_3, and not from p_1.



Proof of the Chandy-Lamport algorithm (sketch): We have to prove two things: (1) the global state is consistent, and (2) the state of the channels is correctly recorded.

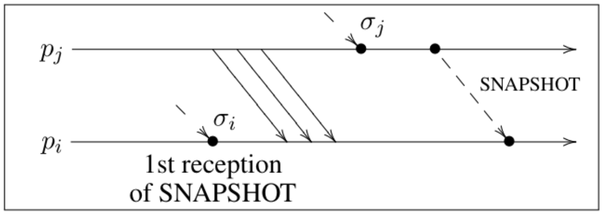
1. *The global state* $(\sigma_1, \sigma_2, \ldots, \sigma_n)$ *is consistent.* We prove the result by contradiction, using Remark 2 above.

   Let C be the cut defined by $(\sigma_1, \ldots, \sigma_n)$. Assume for a contradiction that C is not consistent. By Remark 2, there exists m such that $e = send_i(m) \notin C$ and $e' = receive_j(m) \in C$. (Statement A)

   If $e \notin C$, then $send_i(\text{SNAPSHOT}) \rightarrow send_i(m)$. Since channels are FIFO, we have $receive_j(\text{SNAPSHOT}) \rightarrow receive_j(m)$. It follows that $e' = receive_j(m) \notin C$. A contradiction with (A).

2. *The state of the channels is correctly recorded.*

   Consider the channel c_{ji}. The proof is in two steps: (a) a message m is recorded iff it is in transit on the cut, and (b) the messages are recorded in the sending order.

   We can reformulate (a) as follows.

(a) *m recorded by $p_i \Leftrightarrow send_j(m) \in C$ and $receive_i(m) \notin C$.*

- $\Rightarrow$:
  - m recorded by $p_i \Rightarrow send_j(m)$ (on p_j) is before the first reception of SNAPSHOT on p_j Therefore we have $send_j(m) \in C$.
    (if $send_j(m)$ is after the first reception of SNAPSHOT, since channels are FIFO, m is received by p_i after SNAPSHOT from p_j, i.e., once the computation of the channel c_{ji} is completed).
  - m recorded by $p_i \Rightarrow receive_i(m)$ (on p_i) is after the first reception of SNAPSHOT on p_i (computation of the channel starts after the first reception of SNAPSHOT). C corresponds to the first reception of SNAPSHOT; therefore we have $receive_i(m) \notin C$.
- $\Leftarrow$:
  (i) $send_j(m) \in C$ and FIFO channels $\Rightarrow p_i$ receives SNAPSHOT from p_j after m.
  (ii) $receive_i(m) \notin C \Rightarrow p_i$ starts the recording of messages before receiving m.

  (i) and (ii) $\Rightarrow m$ is recorded by p_i.

(b) *The order of the messages is correctly recorded.*
Follows directly from the FIFO property of the channels.

□

References

[1] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[3] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.