Sapienza
Università di Roma

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

Distributed Systems

Author:
Simone Lidonnici

16 october 2025

Contents

1

Introduction

1.1 Defining a distributed system

We define a **distributed system** as a collection of processes $p_1, p_2, \dots, p_n$ 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:

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

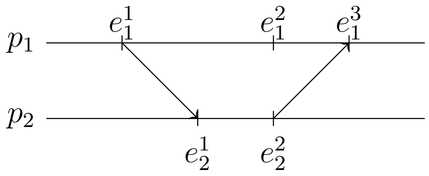
The global history gives us no information about the time in which these events are executed, because in an asynchronous system there is no global clock.

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:

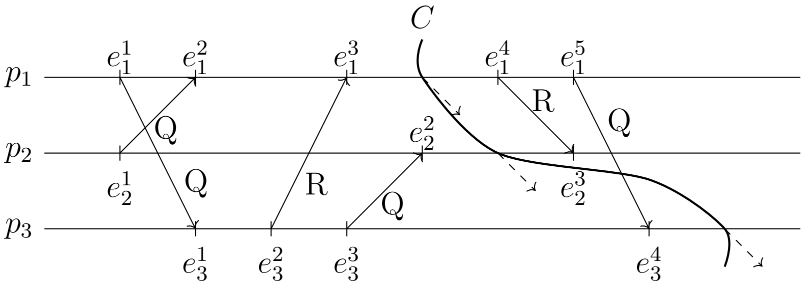
$$\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$$

Cut

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

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

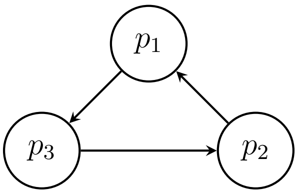
To monitor the computation or to compute a global problem (for example knowing if the system is in deadlock) we add a process p_0.
The first idea is to make p_0 send a message to every process to which a process p_i will respond with the current state σ_i. After all the response p_0 can construct a global 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:

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

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 $e \rightarrow e'$.

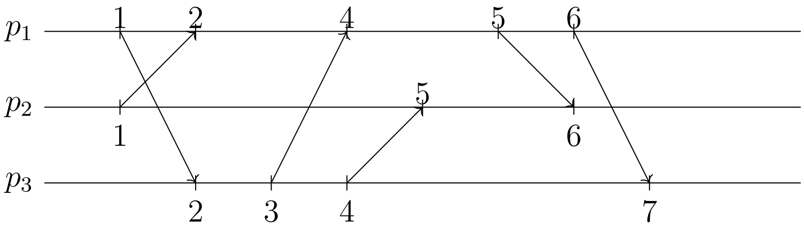
Local clock

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:

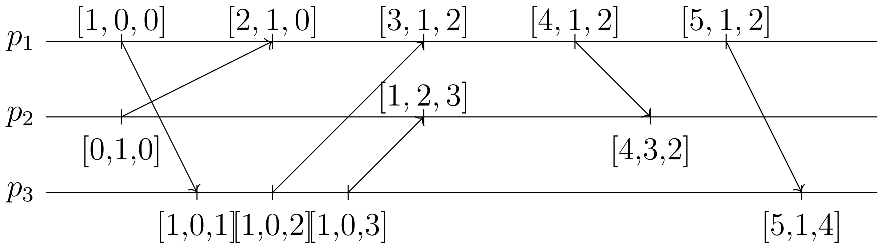
$$H(e) = \{e' | e' \to e\} \cup \{e\}$$

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:



To know when to deliver a notification p_0 has a counter D in which $D[i]$ contains the number of messages delivered from p_i. p_0 will deliver the notification of an event e_i if:

$$VC(e_i)[i] = D[i] + 1$$
$$VC(e_i)[j] \leq D[j] \ \forall j \neq i$$

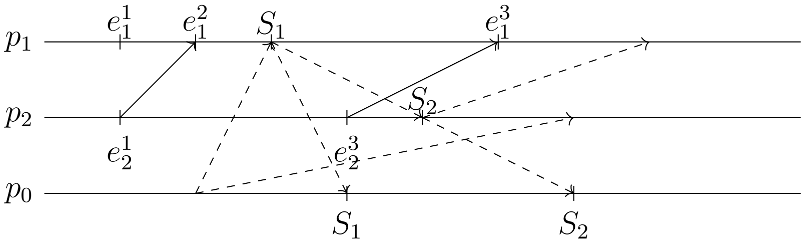
Gap detection

Taken two events e_i, e_j is possible to know if exists an event between the two:

$$\exists e_k \ e_k \rightarrow e_j \land e_k \not\rightarrow e_i \iff VC(e_j)[k] > VC(e_i)[k]$$

1.5 Distributed snapshots

If we reuse the idea proposed in Section 1.3 where p_0 sends a message to the other processes and they respond with their state, called **snapshot**. The cut generated by this idea was not surely consistent, so we apply a change.
When a process recieve the message ”take snapshot” from p_0, it broadcast his state to the other processes. If another process recieve the state of a process it will also broadcast his state. A process will send his state only one time during the protocol, so if he already sent his state to p_0 he will not send it again even if he recieves the state of another process. Every process knows the protocol ended when he receives the state of all the other processes. This protocol, called **Chandy-Lampart Protocol** generates a consistent cut if the channels are FIFO.



2

Atomic transaction

Atomic commit

An atomic commit is the problem that occurs when there is a transaction T that involves multiple sistes in a distibuted system. So all the sites do the transaction, or no one does.

We can explain an atomic commit as if every process involved in the transaction votes yes or no to committing the transaction. The system after the votes decide to commit or abort.
An atomic commit has different properties:

- If a process reach a decision, it must be the same.
- If a process reaches a decision, it cannot change that.
- Decision is to commit only if every process votes yes.
- If there sre not failures and all votes are yes the decision must be commit.
- If all failures ar fixed then the protocol should terminate.

2.1 2 Phase commit

One simple protocol to execute an atomic commit is the **2 Phase commit**, in which there is a coordinator process and there other processes are participants.
The protocol follows 4 phases:

1. The coordinator sends a vote request to every participants.
2. Each participant votes yes or no. If the vote is no the participant can already abort.
3. The coordinator controls the votes and if they are all yes sends a commit message to every participant. Else it sends an abort message.
4. The partecipants execute the decision recieved.

The main problems that can occur during this protocol are message not delivered or process that fail during the execution.
To resolve the message failure a timestamp can be used. At every phase the timeout is used in a different way:

1. The participant is waiting for the vote request and reaches the timeout, so it votes no and abort.
2. The coordinator is waiting for the votes and reaches the timeout, so it sends and abort message.
3. The participant is waiting for the decision and reaches the timeout, so it asks other processes for their decision. If they have a decision, it execute the same, if every process is waiting they have to continue waiting.

To resolve the sites failure the processes log during the protocol. Also in this case every phase has a different method to log:

1. The coordinator log the start of the transaction:

   $$\text{START2PC} \rightarrow \text{DTlog}$$

   Is better to log and then send the messages because in the case of a crash in between the two:

   - If the log is empty: nothing has been sent, so a transaction can start safely.
   - If the log is full: is safe to resend the request.

2. The participant logs its vote:

   $$\text{yes/no} \rightarrow \text{DTlog}$$

   Also in this case is better to log and then send the message because in the case of a crash in between the two:

   - If the log is empty: nothing has been sent, so is safe to abort.
   - If the log is full: is safe to resend the request.

3. The coordinator logs the votes received and ater it sends the decision.
4. The participant logs the decision before executing it.

In some distributed systems with a lot of servers with the same data, this protocol is not the best, because if even one server fails the transaction is aborted.

2.2 Paxos

The **Paxos** protocol has the goal to make a system work even in presence of failures. The servers are divided in three categories:

- **Proposer**: start the transaction and propose the value to vote.
- **Acceptors**: vote the value.
- **Learners**: keep track of the decision made by the Acceptors.

The protocol has two additional rules:

- All the servers need to choose the same value at the end of the protocol.
- The value chosen must be suggested from outside.

The protocol work with minimum one Proposer and one Learner. For the Acceptors the majority is needed so the maximum number of failures tollerated is with n Acceptors:

$$f = \left\lfloor \frac{n-1}{2} \right\rfloor$$

And the size of the quorum will be:

$$|Q| = n - f$$

The protocol works in **rounds** and every round is associated with only one transaction. A round work in this way:

1. Proposer sends a message to all the Acceptors:

   $$\text{P} \rightarrow \text{A} \qquad \text{prepare(curr-round)}$$

2. Acceptors respond to the Proposer:

   $$\text{A} \rightarrow \text{P} \qquad \text{promise(curr-round, last-round, last-value)}$$

   In which they promise to participate in the current round and to not participate to any round lower than that. In addition it also sends the last round in which it has voted and the last value voted.

3. Proposer waits for a quorum of promises and after sends to every Acceptor a message with the value proposed x, that is chosen in this way:

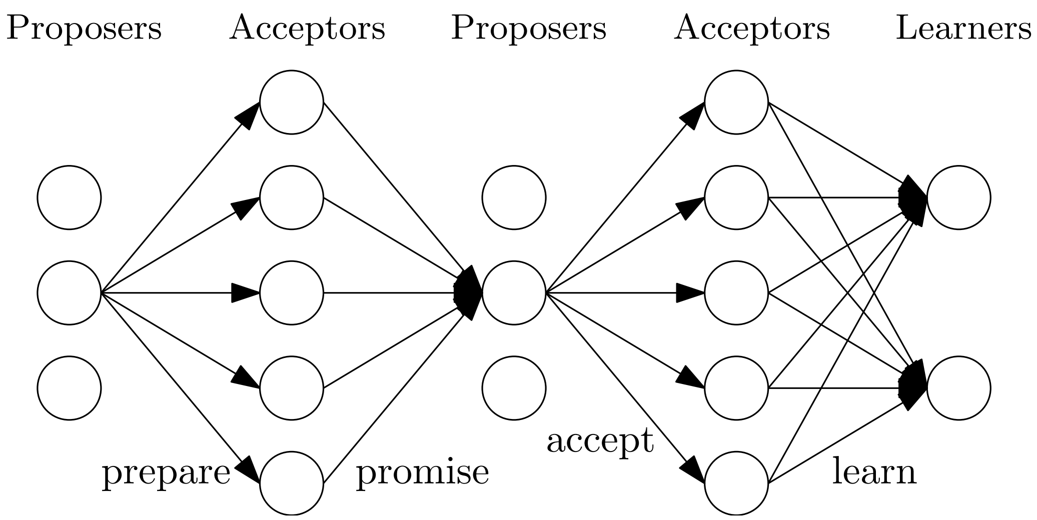
   - If no Acceptor has ever voted the value x can be chosen at random (value that comes from the client).
   - Else the value x is the last value voted associated with the largest value of last-round in the promises.

   $$\text{P} \rightarrow \text{A} \qquad \text{accept(curr-round}, x)$$

4. Acceptors sends their vote to the Learners:

   $$\text{A} \rightarrow \text{L} \qquad \text{learn(curr-round}, x)$$

5. Learners control the votes and if a quorum voted the same value in the same round the value is chosen. If there is no quorum after a timeout the Proposer will asks the Learners for a decision, if there is no decision there will be another round.



This protocol is **safe**, means that doesn't do anything wrong even in presence of more then f failures. Is not **live**, in the sense that is not always working.

There are some problems with transaction that could start in the same time. If the prepare message of a transaction with a lower round arrives after one with an higher round the first transaction will never be done. On the other hand if the prepare message of a transaction with an higher round arrives before the vote of a transaction with a lower round this transaction will be stopped after the acceptors promise with the higher round. If this prepare message arrive after the votes, the Acceptors will tell the Proposer that they just voted and the Proposer will be obliged to choose the same value as the transaction before.

Safety condition

For Paxos the safety condition is:
Is an acceptor A_k votes for a value x in round i, no value different from x can be chosen by learners in a previous round.

Demonstration:

The demonstration can be done by induction:

1. Base case: round 1 doesn't have previous rounds so the property is valid.
2. Induction ipothesis: for every round i, the value to vote is equal to the value x associated with the max last round j in the promises. If for j is valid the property, also is valid for i.
3. Induction step: the value voted in round i must be voted by some acceptors in round j, so in round j is impossible to have a quorum on a value different that x. All the acceptors that sent promises in round i don't vote in rounds between j and i so any of this rounds can have a quorum of votes.

2.2.1 Fast-Paxos

The **Fast-Paxos** protocol is a different version of Paxos, in which there is a proposer coordinator that send prepare(1) messages in advance, without having the value. After recieving the promises the coordinator sends an acceptany(1) message, that tells the acceptors to accept messages from any proposer. When a value x arrive to a proposer it istantly sends an accept$(1, x)$ message to the acceptors. This may cause that in the same round two proposer send different values to vote.

The other change that must be done is how to choose the value x to be voted in rounds where the promises have a max last round with different values associated. To overcome this problem the quorum is modified to be equal to $\frac{2}{3} + 1$. So, the maximum number of failures tolerated decrease to:

$$f = \left\lfloor \frac{n-1}{3} \right\rfloor$$

The value x to vote is choosen:

- If no Acceptor has ever voted the value x can be chosen at random (value that comes from the client).
- Else the value x is the most voted value in the max last round.

The protocol Fast-Paxosis safe.

Demonstration:

The demonstration can be done by induction:

1. Base case: round 1 doesn't have previous rounds so the property is valid.
2. Induction ipothesis: for every round i, the value to vote is equal to the value x most common in the max last round j in the promises. If for j is valid the property, also is valid for i.
3. Induction step: the value voted in round i is the most common value voted in round j, and is the only value that can reach a quorum in round j, because only 1 value every round has the possibility to have a quorum of votes. This is because with a quorum of $\frac{2}{3}$ only 1 value can have more than $\frac{1}{3}$ votes in the promises that could result in a quorum if the $\frac{1}{3}$ outside the quorum all voted for that value. All the acceptors that sent promises in round i don't vote in rounds between j and i so any of this rounds can have a quorum of votes.