"Sapienza" University of Rome
Faculty of Information Engineering,
Informatics and Statistics
Department of Computer Science

Distributed Systems

Author
Simone Bianco

November 14, 2025

Contents

Information and Contacts **1**

1 Note from the author (PLEASE READ) **2**

2 Introduction to distributed system (WIP) **3**

2.1 Computations in distributed systems 3

2.2 System monitoring 5

2.3 System snapshots 9

3 Solved exercises **10**

Information and Contacts

Personal notes and summaries collected as part of the *Distributed Systems* course offered by the degree in Computer Science of the University of Rome "La Sapienza".

Further information and notes can be found at the following link:
https://github.com/Exyss/university-notes. Anyone can feel free to report inaccuracies, improvements or requests through the Issue system provided by GitHub itself or by contacting the author privately:

- Email: **bianco.simone@outlook.it**
- LinkedIn: **Simone Bianco**

The notes are constantly being updated, so please check if the changes have already been made in the most recent version.

Suggested prerequisites:

Computer Networks

Licence:

These documents are distributed under the **GNU Free Documentation License**, a form of copyleft intended for use on a manual, textbook or other documents. Material licensed under the current version of the license can be used for any purpose, as long as the use meets certain conditions:

- All previous authors of the work must be **attributed**.
- All changes to the work must be **logged**.
- All derivative works must be **licensed under the same license**.
- The full text of the license, unmodified invariant sections as defined by the author if any, and any other added warranty disclaimers (such as a general disclaimer alerting readers that the document may not be accurate for example) and copyright notices from previous versions must be maintained.
- Technical measures such as DRM may not be used to control or obstruct distribution or editing of the document.

1

Note from the author (PLEASE READ)

These notes were quickly – very quickly – put together during my preparation for the first midterm of the Distributed System course of a.y. 2025-26 (I stopped writing them due to sickness, I couldn't waste time).

Therefore, it's pretty clear that these are very bad notes. Like really bad. Completely out of my standard. I think that they're still somewhat useful/understandable, but please keep this in mind.

On the other hand, the solutions proposed in the Solved Exercises chapter are well written (and well tought) since my sickness was gone by that time.

Final note: as announced on the readme page of the repository, the source code is soon going to be public. If anyone is interested in helping me complete these notes, you're welcome!

Good luck!

2

Introduction to distributed system (WIP)

2.1 Computations in distributed systems

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 assumed to be **reliable**, in the sense that te message arrive, but may delivered messages out of order.

We say that a distributed system is *asynchronous* when the computation between processes has no upper bounds for speed or message delay. If these upper bounds exists the, system is called *synchronous*. Clearly, synchronous systems are more strict compared to asynchronous ones. Nonetheless, it's easy to see that every computation executed on a synchronous system can also run on an asynchronous system, while the opposite isn't always true.

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

Each distributed system can have up to two of these properties (otherwise, they conflict with each other).

Distributed computations are defined by a collection of **events**, which can be *internal*, i.e. executed internally by the process, or *external*, i.e. involve comunication with another process. For our interests, we consider only two types of external events: sending a

message m to process j, written as $\text{send}_j(m)$, and receiving a message m from process j, written as $\text{receive}_j(m)$. For each process p_i, we denote its k-internal event with e_i^k.

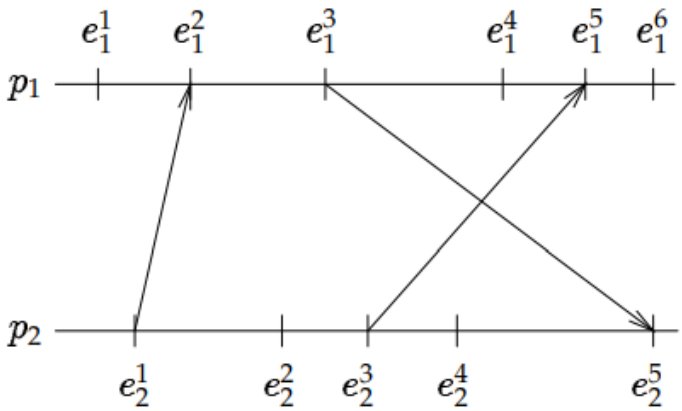


Figure 2.1: An example of distibuted computation. Arrows correspond to messages exchanged by processes.

Definition 2.1: Local and global history

We define the local history h_i of a process p_i as the sequence of its events, meaning that $h_i = e_i^1 e_i^2 \ldots$. We denote with h_i^k the restriction of h_i up the the k-th event, i.e. $h_i^k = e_i^1 e_i^2 \ldots e_i^k$. The global history H is defined as the union of the histories of all processes $H = h_1 \cup \ldots \cup h_n$.

We observe that the global history gives us no information about the time in which these events are executed: in an asynchronous system there is no global clock that governates the processes, thus we cannot associate a global timestamp to the events. The ordered sequence of all the events of the system is referred to as **run** and it must preserve the sequences of the histories.

Definition 2.2: Run

We define a run as an ordering $R = e_1^1 e_2^1 \ldots e_1^2 e_1^2 \ldots$ of H that preserves the orderings of the local histories $h_1, \ldots, h_n$.

To infer informations about which events e of the past influenced an event e' in the future, we define a **cause-effect relation**, where $e \rightarrow e'$ means that event e affects event e'.

Definition 2.3: Cause-effect relation

Given the global history H of a system, we define the cause-effect relation $\rightarrow \subset H \times H$ as follows:

- *Internal sequencing*: $\forall i \in [n]$ and $\forall e_i^k, e_i^{k'} \in h_i$ with $k < k'$ it holds that $e_i^k \rightarrow e_i^{k'}$
- *External sequencing*: $\forall i, j \in [n]$ and $\forall (e_i^k, e_j^{k'}) \in h_i \times h_j$ if $e_i^k = \text{send}_j(m)$ and $e_j^{k'} = \text{receive}_i(m)$ then $e_i^k \rightarrow e_j^{k'}$
- *Transitivity*: $\forall e, e', e'' \in H$, if $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$

Clearly, some events may never influence each other, meaning that $e \not\rightarrow e'$ and $e' \not\rightarrow e$. When this happens, the events are said to be **concurrent**.

Definition 2.4: Concurrent events

We say that two events $e, e' \in H$ are concurrent, written as $e \mid\mid e'$, when $e \not\rightarrow e'$ and $e' \not\rightarrow e$.

2.2 System monitoring

We denote the local state of process p_i after the execution of event e_i^k with σ_i^k. The n-uple of local states $\Sigma = (\sigma_1, \ldots, \sigma_n)$ represents the global state of the system.

Definition 2.5: Cut

We define a cut C as a collection of all the local histories restricted up to the events $e_1^{k_1}, \ldots, e_n^{k_n}$

$$C = \langle h_1^{k_1}, \ldots, h_n^{k_n} \rangle$$

Consider the following distributed computation.

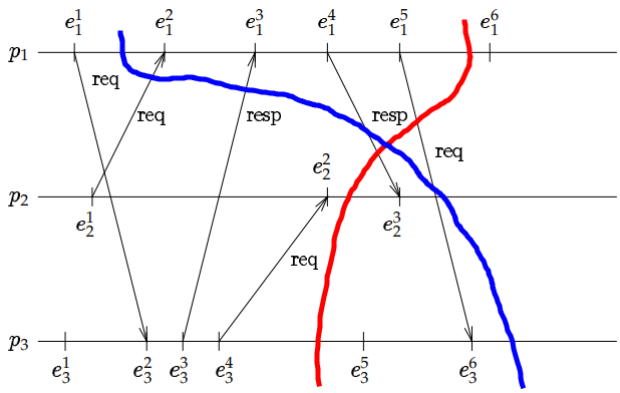


Figure 2.2: The blue an red lines represent the cuts $C = \langle h_1^1, h_2^3, h_3^6 \rangle$ and $C' = \langle h_1^6, h_2^2, h_3^4 \rangle$

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 called **monitoring process**. A simple idea is to make p_0 send a message to every process p_j to which a process p_i will respond with the current state σ_i. After all the responses, p_0 can construct a global state that defines a cut.

For instance, consider the following computation with a cut C.

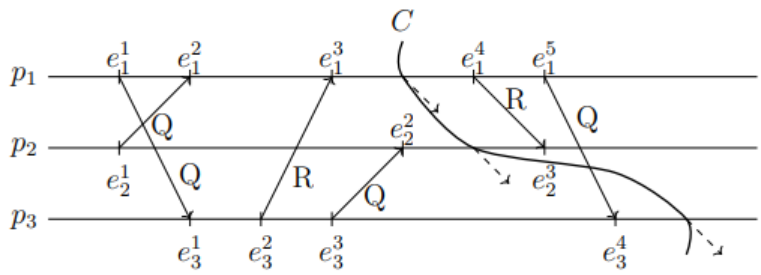


Figure 2.3: The dashed arrows represent message response messages sent to process p_0.

Based on the histories contain in C, we can expect that in the future process p_1 is going to send a response to p_2, process p_2 is going to send a response to p_3 and process p_3 is going to send a response to p_1. Does this imply that cut C found a deadlock? The answer is no because the global state defined by cut C can actually never happen since $e_1^5 = \text{send}_3(Q)$ happens after the cut and $e_3^4 = \text{receive}_1(Q)$ happens before it. In order to use cuts as proper monitoring tools, we have to ensure that they are *consistent*.

Definition 2.6: Consistent cuts and runs

A cut C is said to be consistent when $\forall e' \in C$ it holds that if $e \to e'$ then $e \in C$. Similarly, a run R is said to be consistent when forall $e \to e'$ it holds that e comes before e' in R.

Proposition 2.1

If a run is consistent then every cut is consistent.

Proof. Omitted. □

After defining consistency, a natural question arises: which tools can be used to certify it? The simplest idea is to consider a **timestamp** function T that associates an incremental timestamp to each new event.

Observation 2.1

In order to be algorithmically implemented, the timestamp function T requires that the system has a global clock and that the processes are synchronized.

By imposing that $\mathrm{T}(e) < \mathrm{T}(e')$ whenever $e \to e'$, we can ensure that the system is consistent. The generalization of the above property is called **clock condition**.

Definition 2.7: Weak and strong clock condition

Given a time function f, we say that f satisfies the (weak) clock condition if whenever $e \to e'$ it holds that $f(e) < f(e')$. Similarly, we say that f satisfies the strong clock condition when $e \to e'$ if and only if $f(e) < f(e')$.

We observe that the clock condition doesn't ensure that the opposite direction of the statement is true, i.e. that whenever $f(e) < f(e')$ it also holds that $e \to e'$. This is only true for the strong clock condition.

Proposition 2.2: Property of real time

A run is consistent if and only if the function T satisfies the clock condition.

Even thought the above proposition states that timestamps are a sufficient time metric to certify consistency, we recall that an asynchronous distributes system isn't provided with a global clock, making this metric useless. Therefore, we need a new metric that can be algorithmically implemented without requiring a global clock. The easiest way to solve this issue is to use a *local clock* instead of a global one, such as the **Lamport clock**.

Definition 2.8: Lamport clock

For each process p_i and each event $e_i^k \in h_i$, we define the Lamport clock of p_i as the value LC_i^k given by:

$$\mathrm{LC}_i^k = \begin{cases} 0 & \text{if } k = 0 \\ \mathrm{LC}_i^{k-1} + 1 & \text{if } e_i^k \text{ is an internal or send event} \\ \max(\mathrm{LC}_i^{k-1}, \mathrm{LC}(e_{i'}^{k'})) + 1 & \text{if } e_i^k \text{ receives from } e_{i'}^{k'} \end{cases}$$

where $\mathrm{LC}(e_h^t) = \mathrm{LC}_h^t$ for all $h \in [n]$ and $e_h^t \in h_h$.

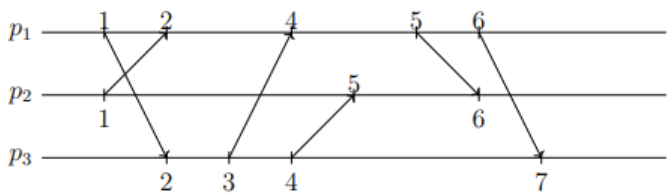


Figure 2.4: The Lamport clocks of the example in Figure 2.3.

Proposition 2.3

The Lamport clock satisfies the clock condition.

Proof. Omitted. □

We observe that, even tought they require to share clock information between processes, the Lamport clock can be algorithmically implemented by an asynchronous distributed system: all that we have to do is to send the clock information together with the message. However, this still requires that the processes are synchronized.

Observation 2.2

In order to be algorithmically implemented, the timestamp function T requires that the processes are synchronized.

To remove the synchronization assumption, the easiest idea is to remove the need to send the clock values altogether. To achieve this, each process keeps track of the local clocks of the other processes using a **vector clock**.

Definition 2.9: Vector clock

For each process p_i and each event $e_i^k \in h_i$, we define the vector clock of p_i as the vector VC_i^k where each entry $j \in [n]$ is given by:

$$\mathrm{VC}_i^k[j] = \begin{cases} 0 & \text{if } k = 0 \\ \mathrm{VC}_i^{k-1}[i] + 1 & \text{if } j = i \\ \mathrm{VC}_i^{k-1}[j] & \text{if } j \neq i \text{ and } e_i^k \text{ is internal or send} \\ \max(\mathrm{VC}_i^{k-1}[j], \mathrm{VC}(e_{i'}^{k'})[j]) + 1 & \text{if } j \neq i \text{ and } e_i^k \text{ receives from } e_{i'}^{k'} \end{cases}$$

where $\mathrm{VC}(e_h^t) = \mathrm{VC}_h^t$ for all $h \in [n]$ and $e_h^t \in h_h$.

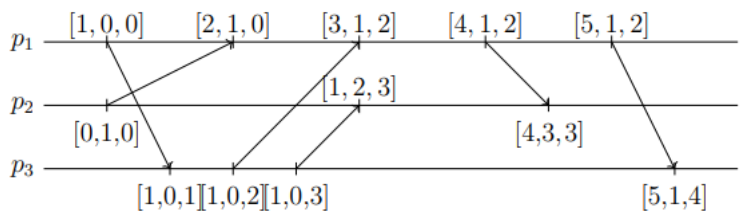


Figure 2.5: The vector clocks of the example in Figure 2.3.

Proposition 2.4

The vector clock satisfies the strong clock condition.

Proof. Omitted. □

To know when to deliver a notification, the process p_0 uses a counter vector D in which each entry $D[i]$, with $i \in [n]$, is the number of messages delivered to p_i. In particular, the monitor process knows that it can safely deliver a notification of event $e_i^k \in H$ if $\mathrm{VC}(e_i^k)[i] = D[i] + 1$ and $\mathrm{VC}(e_i^k)[j] \leq D[j]$ for each $j \in [n] - \{i\}$ (assuming FIFO is used).

Together with the strong clock condition, the vector clock also satisfies another important property called **weak gap detection**.

Proposition 2.5: Weak gap detection

Given an event $e_i^k \in h_i$ and an event $e_j^{k'} \in h_j$, if $\mathrm{VC}(e_i^k)[h] < \mathrm{VC}(e_j^{k'})[h]$ for some $k \neq j, i$ then $\exists e_h^{k''} \in h_h$ such that $e_h^{k''} \not\rightarrow e_i^k$ and $e_h^{k''} \rightarrow e_j^{k'}$.

2.3 System snapshots

A global snapshot consists of local states of each process in the system along with the in-transit messages on the communication channels. The system snapshot problem tries to determine the *current* global snapshot <u>without</u> stopping the system. There are two main issues in determining the global snapshot:

1. Process states cannot be catched all at the same time
2. Messages that are on the way cannot be seen

For each protocol pair p_i, p_j, we define the **channel state** frpm p_i to p_j, denoted with $\chi_{i,j}$, as the messages sent by p_i to p_j and that have not yet been reached.

Chandy-Lamport protocol:

1. The process p_0 starts the protocol by sending the message "take snapshot" to itself
2. For each process p_i, let p_{f_i} be the process from which p_i receives a "take snapshot" message for the first time. Upon receiving the message, p_i records its local state σ_i and relays the message on all of its outgoing channels (o intervening events on behalf of the underlying computation are executed between these steps). The channel state $\chi_{f_i,i}$ is set to empty and p_i starts recording messages recorded over each of its other incoming channels.
3. For each process p_i, let p_{j_i} be any process from which p_i receives a "take snapshot" message after the first time. Process p_i stops recording messages along the channel frm p_{j_i} and declares channel state $\chi_{j_i,i}$ with the messages that have been recorder. Since a "take snapshot" message is relayed only upon the first receipt and since the network is strongly connected, a "take snapshot" message traverses each channel exactly once. When p_i has received a "take snapshot" message from all of its incoming channels, its contribution to the global state is complete and its partecipation in the snapshot protocol ends.

3

Solved exercises

Problem 3.1

Let C_1 and C_2 be two consistent cuts. Show that the intersection of C_1 and C_2 is a consistent cut.

Proof. Fix $e' \in C_1 \cap C_2$ and suppose that $e \to e'$ for some event e. Since $e' \in C_1$ and $e' \in C_2$, by consistency of both cuts we have that $e \in C_1$ and $e \in C_2$, concluding that $e \in C_1 \cap C_2$. □

Problem 3.2

Let C_1 and C_2 be two consistent cuts. Show that the union of C_1 and C_2 is a consistent cut.

Proof. Fix $e' \in C_1 \cup C_2$ and suppose that $e \to e'$ for some event e. Without loss of generality, assume that $e' \in C_1 \cup C_2$ holds because $e' \in C_1$. By consistency of the cut, it must hold that $e \in C_1$, therefore $e \in C_1 \cup C_2$. □

Problem 3.3

Show that every consistent global state can be reached by some consistent run.

Proof. We proceed by induction on the number m of events in the consistent global state. If $m = 0$, the consistent run with no events reaches the state. Assume the inductive hypothesis holds for any consistent global state with m events and consider a consistent global state Σ with $m + 1$ events.

Let C be cut associated with Σ and consider an event $e \in C$ such that $e \notin H(e')$ for all $e' \in C - \{e\}$, i.e. an event that isn't the cause of any other event. We observe that

such an event must always exist, otherwise there would be at least one looping chain $e^{(1)} \to \ldots \to e^{(\ell)} \to e^{(1)}$ in C, which is impossible.

Consider now the cut $C - \{e\}$ and the global state Σ' associated with it. By choice of e, it trivially holds that $C - \{e\}$ is consistent, therefore by inductive hypothesis Σ' can be reached by a consistent run R'. Then, $R = R'e$ is a consistent run that reaches Σ. □

Problem 3.4

Let C_1 and C_2 be two consistent cuts. Prove that if C_1 is a subset of C_2, then C_2 is reachable from C_1. (there exists a consistent run that reaches C_1 and then reaches C_2.)

Proof. Let $|C_1| = m_1$ and let $|C_2| = m_2$. Let also Σ^1 be the consistent global states associated with cut C_1. Through the previous exercise, we know that there is a consistent run R that reaches Σ_1 (hence it reaches C_1.).

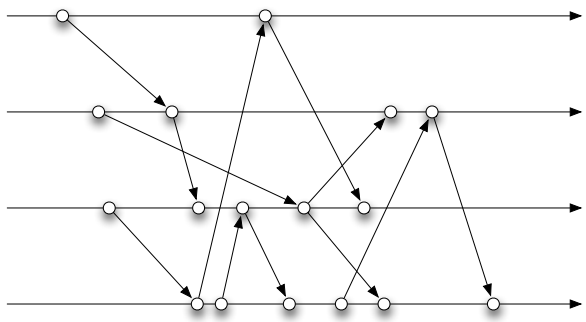
We define a series of cuts $C^{(0)}, \ldots, C^{(\ell)}$, where $\ell = m_2 - m_1 - 1$, defined through the following inductive construction:

- Set $C^{(0)} = C_1$
- For each i such that $1 \leq i \leq \ell$, set $C^{(i)} = C^{(i-1)} \cup \{e_i\}$ where e_i is any event $e_i \in C_2 - C^{(i-1)}$ such that $H(e_i) \subseteq C^{(i-1)} \cup \{e_i\}$, i.e. any event that isn't caused by events outside of $C^{(i-1)}$.

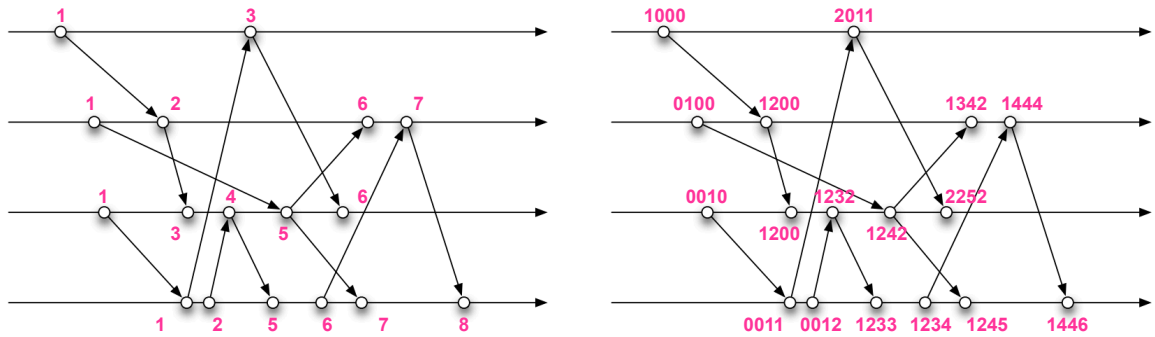
We observe that $C^{(0)} = C_1$ and $C^{(\ell)} = C_2$ by construction of the sequence. Moreover, it's easy to see that there is a consistent run that reaches $C^{(0)}, \ldots, C^{(\ell)}$ in that order. This can be proven by induction on the sequence index i of $C^{(i)}$. If $i = 0$, we know that $C^{(0)} = C_1$ and, by the previous example, that there is a consistent run that reaches $C^{(0)}$. Assume by inductive hypothesis that there is a consistent run R that reaches the cuts $C^{(0)}, \ldots, C^{(i)}$ in that order and consider index $i+1$. Then, $R' = Re_{i+1}$ is a consistent run that reaches the cuts $C^{(0)}, \ldots, C^{(i+1)}$ in that order. □

Problem 3.5

Label all the events of the distributed computation represented in the below image first by using the Lampert clock and then by using the vector clock (you can consider events that receive a message and immediately send it as single events.)



Solution:



Problem 3.6

Show that the Chandy-Lamport Snapshot Protocol builds a consistent global state.

Proof. We observe that the transitivity of the $\rightarrow$ relation makes proving this result tricky.

Let Σ be the global state derived through the protocol. For each process p_i, let e_i^* be the recording event (i.e. when the process receives its first "take snapshot" message). Let $C =< h_1^*, \ldots, h_n^* >$ be the cut associated with Σ. In order to show that C is consistent, we prove the following stronger claim.

Claim: For every chain $e^{(1)} \rightarrow \ldots \rightarrow e^{(\ell)}$ such that $e^{(\ell)} \in C$ it holds that $e^{(1)}, \ldots, e^{(\ell)} \in C$.

Proof of the claim. By way of contradiction, suppose that there is at least a chain for which the claim is false. Then, there must be a chain $\mathcal{C} = e^{(1)} \rightarrow \ldots \rightarrow e^{(\ell)}$ of minimal length for which the claim doesn't hold. Let p_i be the process of $e^{(\ell)}$. We may assume that at least one relation of the chain comes from a send-receive pair, otherwise each event $e^{(j)}$ of the chain is an event of p_i that occurred before $c^{(\ell)}$, concluding that $e^{(j)} \rightarrow e_i^*$, hence $e^{(j)} \in h_i^*$ and thus $e^{(j)} \in C$.

Let t be the minimal index of the chain such that $e^{(t)} \to e^{(t+1)}$ comes from the fact that $e^{(t)}$ is a send event in process p_j and $e^{(t+1)}$ is its corresponding receive event in process p_k.

Since $e^{(t+1)} \to \ldots \to e^{(\ell)}$ is a chain smaller than $\mathcal{C}$, it must satisfy the property of the claim, concluding that $e^{(t+1)}, \ldots, e^{(\ell)} \in C$. Since $e^{(t+1)} \in h_k^*$, it must hold that $e^{(t+1)} \to e_k^*$. We observe that $e^{(t)} \notin C$ must hold, otherwise $e^{(1)}, \ldots, e^{(t)} \in C$ by the same minimality argument, but this is impossible since at least one event of $\mathcal{C}$ must be outside of C. Since $e^{(t)} \notin C$, it must hold that $e_j^* \to e^{(t)}$. However, this implies that process p_j sent the "take snapshot" message to every process (p_k included) before sending the message of event $e^{(t)}$. Since the protocol uses the FIFO assumption, this "take snapshot" message must have been received by p_k before of $e^{(t+1)}$, meaning that $e_k^* \to e^{(t+1)}$ and thus raising a contradiction. Therefore, the only possibility is that the claim must hold for every chain. □

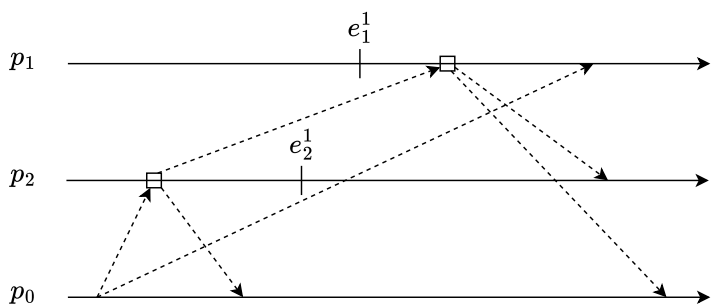
The claim directly concludes that the global state derived by the protocol is consistent. □

Problem 3.7

Show that the Chandy-Lamport Snapshot Protocol can build a global state that never happened.

Solution:

Consider the execution of the protocol represented by the picture below. Let R be the run obtained by ignoring the "take snapshot" messages, i.e. $R = e_2^1 e_1^1$ with $e_2^1 \mid\mid e_1^1$. It's easy to see that the cut $C = \langle e_1^1, \cdot \rangle$ produced by the protocol is associated with a global state that is never reached by R since it only reaches the states $\Sigma^0, \Sigma^1, \Sigma^2$ given by the cuts $C_0 = \langle \cdot, \cdot \rangle, C_1 = \langle \cdot, e_2^1 \rangle, C_2 = \langle e_1^1, e_2^1 \rangle$.



Problem 3.8

What good is a distributed snapshot when the system was never in the state represented by the distributed snapshot? Give an application of distributed snapshots.

Solution:

Even though a snapshot may not correspond to a global state that really happened, the global state retrieved by it is still logically consistent (meaning it still could have happened,

according to the causal order of events). This allows us to reason about the global properties of the system without requiring synchronization and detect stable properties (conditions that, once true, remain true thereafter), such as deadlocks, consistent recovery point, the total amount of money or messages in a system.

Problem 3.9

Consider a distributed system where every node has its physical clock and all physical clocks are perfectly synchronized. Give an algorithm to record global state assuming the communication network is reliable. (note that your algorithm should be simpler than the Chandy-Lamport algorithm.)

Solution:

Consider the following protocol:

1. The monitor process p_0 starts the protocol by broadcasting a "take snapshot at time $t+\Delta$" where t is the current time and Δ is the maximum time required by a message to reach every node.
2. Upon receiving the "take snapshot at time $t+\Delta$", process p_i waits until time $t+\Delta$ (ignoring all events) to then record and send its local state σ_i back to p_0

Problem 3.10

What modifications should be done to the Chandy-Lamport snapshot protocol so that it records a strongly consistent snapshot (i.e., all channel states are recorded empty).

Solution:

Consider the following protocol:

1. The monitor process p_0 starts the protocol by broadcasting a "STOP".
2. Upon receiving the "STOP" flag, process p_i stops sending messages and broadcasts an "ACK" message.
3. Upon receiving $n-1$ "ACK" messages, process p_i records and sends its local state σ_i back to p_0.
4. Every process resumes sending messages.

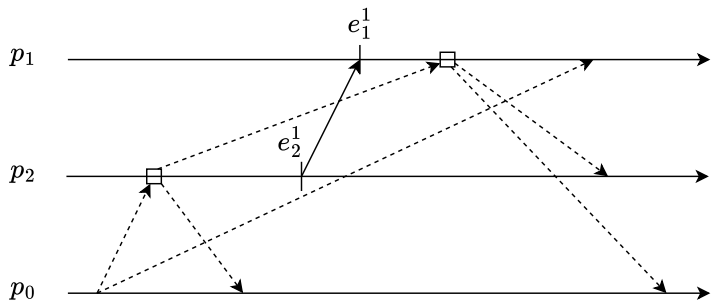
Under the FIFO assumption, the above protocol ensures that all the incoming channels and outgoing channels of each process are empty once every process receives $n-1$ "ACK"s: each "ACK" message will be the last incoming message on each incoming channel and the last outgoing message on every outgoing channel.

Problem 3.11

Show that, if channels are not FIFO, then Chandy-Lamport snapshot algorithm does not work.

Solution:

Consider the execution of the protocol represented by the picture below. It's easy to see that the output global state Σ associated with cut C is not consistent since $e_1^1 \to e_2^1$ and $e_2^1 \in C$ but $e_1^1 \notin C$.



Problem 3.12

Let Σ^0 be the global state when the Chandy-Lamport snapshot protocol starts, Σ be the global state built by the protocol, and Σ_1 be the global state when the protocol ends. Show that Σ is reachable from Σ^0 and that Σ^1 is reachable from Σ. Remember that Σ might not have happened.

Proof. Let C_0, C, C_1 be the consistent cuts associated, respectively, with $\Sigma^0, \Sigma, \Sigma^1$. We observe that, even thought Σ may not have happened, it still holds that $C^0 \subseteq C \subseteq C_1$. Thus, using the same idea of Problem 3.4 we can extend a consistent run for Σ^0 first to a consistent run for Σ and then to a consistent run for Σ^1. $\square$

Problem 3.13

Give an AC protocol that also satisfies the converse of condition AC3. That is, if all processes vote Yes, then the decision must be Commit. Why is it not a good idea to enforce this condition?

Solution:

We observe that the standard 2-Phase Commit protocol already has this converse property. Enforcing this property gives us a protocol for which the decision is a Commit if and only if every process votes Yes. Therefore, the protocol will enter a livelock if there is at least one failure on each round.

Problem 3.14

Consider 2-Phase Commit with the cooperative termination protocol. Describe a scenario (a particular execution) involving site failures only, which causes operational sites to become blocked.

Solution:

In a cooperative termination 2PC protocol, if a process p voted yes but hasn't received any decision from the coordinator, it may ask to another process q if they received the decision or not. Once q receives the request, it will always in one of two ways:

- It answers with the decision if they received it
- It answers with their vote if they also didn't receive the decision

In the first case, p copies the decision and executes it. In the second case, if q's vote was a No then p immediately Aborts since it knows that the decision couldn't have been a Commit. If q's vote was a Yes, p cannot infer any information, thus it keeps waiting. In the last case, p also cannot infer any information.

Therefore, if on every round of the protocol each process votes Yes but the coordinator always fails (after receiving the votes and before sending the decision), each process will be stuck waiting for the coordinator since they cannot infer any information from each other. This creates a livelock.

Problem 3.15

Show that Paxos is not live.

Proof. Suppose that proposer p_1 sends the message $\langle \texttt{PREPARE}\ i \rangle$ to all n acceptors and that everyone of them receives the message. Each acceptor a_j will answer with the message $\langle \texttt{PROMISE}\ i, \ell_j, v_j \rangle$, ignoring every message coming from rounds previous to round i. Then, proposer p_1 sends the message $\left\langle \texttt{ACCEPT}\ i, v^{(i)} \right\rangle$ to all acceptors.

Suppose now that another proposer p_2 sends the message $\langle \texttt{PREPARE}\ i+1 \rangle$ to all n acceptors and that everyone of them receives the message before receiving the last message sent by p_1. Now, each acceptor promises to ignore every message coming from rounds previous to round $i+1$, including the accept message of round i.

If this behaviour repeats alternating between the two proposers, the acceptors will never receive an accept message, reaching a livelock. □

Problem 3.16

Assume that acceptors do not change their vote. In other words, if they vote for value v in round i, they will not send learn messages with value different from v in larger rounds. Show that Paxos, with this modification, is safe. Unfortunately, the modification introduces a severe liveness problem (the protocol can reach a livelock).

Proof. To prove the safety of the protocol, we prove the usual C1, C2 and C3 properties:

C1. Only a proposed value may be chosen.

C2. Only a single value is chosen.

C3. Only a chosen value may be learned by a correct learner.

Properties C1 and C3 trivially follow from the definition of the protocol. Suppose now by way of contradiction that two different values v_i and v_j are chosen respectively on round i and j. Without loss of generality, assume that $i < j$. In order to be chosen, both values must have reached a quorum of learn messages. Let Q_i and Q_j be the respective quorums, both with $|Q_i|, |Q_j| \geq n - f$. Through the modification introduced, we know that each acceptor of quorum Q_i will keep voting the value v_i in following rounds. However, this implies that in round j we'll have at least $2n - 2f > n$ votes (at least $n - f$ votes for v_i and at least $n - f$ votes for v_j), which is absurd. This concludes that property C2 must also hold.

The livelock can be very easily reached by any round in which no learn quorum is reached but every acceptor as voted since they'll keep voting those values forever, thus never reaching a learn quorum again. □

Problem 3.17

How many messages are used in Paxos if no message is lost and in the best case? Is it possible to reduce the number of messages without losing tolerance to failures and without changing the number of proposers, acceptors, and learners?

Solution:

In the best case, each round of the Paxos protocol requires exactly $3n + \ell n$ messages, where ℓ is the number of learners: n proposes, n promises, n accepts and ℓn learns.

The number of messages can be reduced (assuming the best case) to $3(n - f) + n\ell$, the bare minimum in order to make the protocol chose a value: $n-f$ proposes, $n-f$ promises, $n - f$ accepts (sent to the same acceptors that received the $n - f$ proposes) and $(n - f)\ell$ learns.

Problem 3.18

Assume that you remove the property that every round is associate to a unique proposer. After collecting a quorum of $n-f$ promises (where n is the number of acceptors and f is such that $n = 2f + 1$), the proposer chooses one of the values voted in max round in the promises (of course it is not unique, the proposer chooses just one in an arbitrary way). Show that Paxos is not safe any more.

Proof. Assume that n is odd and suppose that two different proposers p_1 and p_2 send the message $\langle$`PREPARE` $1\rangle$ – where 1 is the first round of the protocol – to all n acceptors and that everyone of them receives the message. Each acceptor a_j will answer with the

message $\langle \texttt{PROMISE}\ 1, -1, -1 \rangle$. Then, each proposers will send, respectively, the message $\langle \texttt{ACCEPT}\ 1, x \rangle$ and $\langle \texttt{ACCEPT}\ 1, y \rangle$, for some newly generated values x and y such that $x \neq y$.

Suppose that a quorum of the acceptors receive the message $\langle \texttt{ACCEPT}\ 1, x \rangle$, while the others receive $\langle \texttt{ACCEPT}\ 1, y \rangle$, therefore implying that the value x will be chosen in round 1.

Suppose now that a new proposer sends $\langle \texttt{PREPARE}\ 2 \rangle$. Some acceptors will answer with the message $\langle \texttt{PROMISE}\ 2, 1, x \rangle$ while some others will answer with $\langle \texttt{PROMISE}\ 2, 1, y \rangle$. Through the max round rule, the proposer choses a random value between x and y. If y is selected and a learn quorum is reached, the learners will choose the value y. Hence, we conclude that two different values have been chosen, breaking safety.

□

Problem 3.19

Assume that all proposers are learners as well. Let even rounds be assigned to proposers with the rules that we know. Moreover, If round $2i$ is assigned to proposer p, then also round $2i + 1$ is assigned to proposer p. Odd rounds are "recovery" rounds. If round $2i$ is a fast round and if the proposer of round $2i$ sees a conflict (it is also a learner), then the proposer immediately sends an accept for round $2i+1$ with the value that has been most voted in round $2i$, without any prepare and any promise. Is safety violated? If yes, show an example. If not, demonstrate safety.

Solution: (TODO)

Problem 3.20

You are an optimization freak. You realize that in Fast Paxos, in some cases, it is not necessary that the proposer collects $n - f'$ (the Fast Paxos quorum) promises to take a decision. Which is the minimum quorum and under what hypothesis this minimum quorum is enough to take a decision?

Solution: (TODO)

Problem 3.21

Show that Raft is not live.

Proof. This can be easily proven in various ways:

1. The messages may always timeout, reaching a livelock.
2. The election may never reach a quorum, reaching a livelock.
3. The elected leader may crash after each election.

□

Problem 3.22

In Raft, it is sometimes possible that the elected leader for term t has not all the log entries that are stored in the followers. Show that, in that case, the log entries missing at the leader are actually not committed and so they can be overwritten by the new leader.

Proof. By way of contradiction, suppose that a leader ℓ elected on term i is missing a Commit entry that happened in term j, with $j < i$. In order to be committed, a quorum Q of processes must have agreed to such transaction, saving it as committed in their logs.

Then, all the processes in Q couldn't have voted for ℓ during the election since their log is more up-to-date. Since Q forms a quorum, this concludes that ℓ couldn't have been elected on round i. □

Problem 3.23

Show that the Ben-Or randomised consensus algorithm terminates with high probability (i.e. that the probability that it does not terminate goes to zero as the number of rounds goes to infinity).

Proof. First, we recall that:

$$\Pr[\text{exit on round } i] \geq \frac{1}{2^{n-f-1}}$$

Let X be the random variable whose value is the total number of rounds executed. Then, we have that:

$$\begin{aligned}
\Pr[\text{no termination}] &= \lim_{i \to +\infty} \Pr[X \geq i] \\
&= \lim_{i \to +\infty} \Pr[\text{no exit on rounds } 1, \ldots, i-1] \\
&\leq \lim_{i \to +\infty} \left(1 - \frac{1}{2^{n-f-1}}\right)^{i-1} \\
&= 0
\end{aligned}$$

Thus the protocol must almost surely terminate. □

Problem 3.24

Build a run of the Ben-Or randomised consensus algorithm that never terminates.

Solution:

Suppose that the number n of partecipating processes is even. For each round i, let $p^{(i)} = [p_1^{(i)}, \ldots, p_n^{(i)}]$ be the preference vector of round i, where $p_j^{(i)}$ is the preference of process j for round i. Consider now the two following properties:

1. The initial vector $p^{(1)}$ contains $\frac{n}{2}$ zeros and $\frac{n}{2}$ ones
2. For each $i > 1$ it holds that $p^{(i)} = p^{(i-1)} \oplus [1, \ldots, 1]$, where $\oplus$ is the bitwise XOR

By construction of the Ben-Or algorithm, a run with both properties will never terminate (when n is even): on each round every process will send a type 2 message containing "?", making every process flip a coin. Finally, we observe that such a run can indeed exist: if we start with a vector satisfying property (1), there is a non-zero chance for the coin tosses to always respect property (2).

Problem 3.25

Consider an asynchronous system of 5 processes that run the Ben-Or randomised consensus algorithm. The number of failures that the system allows is 2. Show that, if at most 2 failures occurs, than the probability that the protocol terminates after x rounds (or more) is smaller than α^x, for some α.

Proof. First, we recall that:

$$\Pr[\text{exit on round } i] \geq \frac{1}{2^{n-f-1}} = \frac{1}{2^{5-2-1}} = \frac{1}{4}$$

Let X be the random variable whose value is the total number of rounds executed. Then, we have that:

$$\begin{aligned}\Pr[X \geq x] &= \Pr[\text{no exit on rounds } 1, \ldots, x-1] \\ &\leq \left(1 - \frac{1}{4}\right)^{x-1} \\ &= \left(\frac{3}{4}\right)^{x-1}\end{aligned}$$

□