SAPIENZA
UNIVERSITÀ DI ROMA

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

Distributed Systems

Author:
Simone Lidonnici

29 december 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:

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

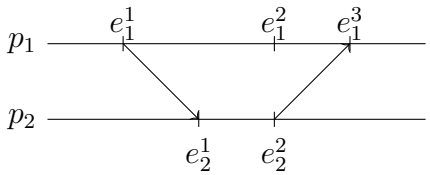
The global history gives us no information about the time in which these events are executed, because in an asynchronous system there is no global clock.

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:

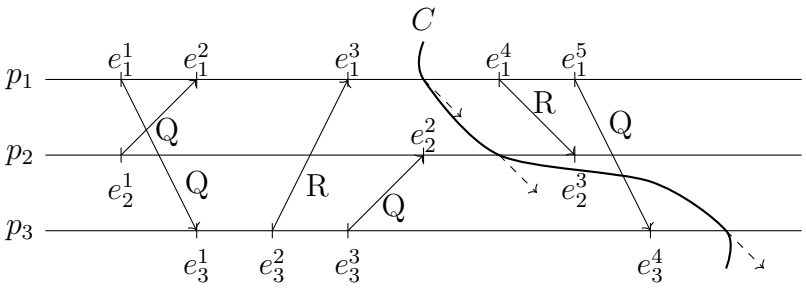
$$\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$$

Cut

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

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

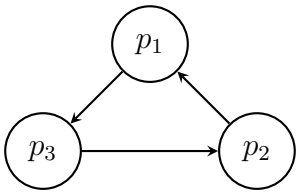
To monitor the computation or to compute a global problem (for example knowing if the system is in deadlock) we add a process p_0.
The first idea is to make p_0 send a message to every process to which a process p_i will respond with the current state σ_i. After all the response p_0 can construct a global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

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

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**:

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

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

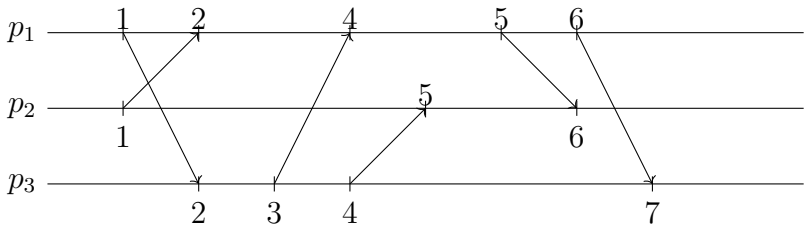
Local clock

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.

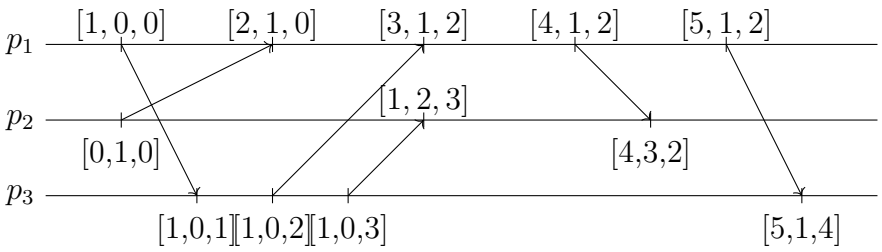
We will identify an history of a process e with a vector in which every index i represent the last event of process p_{i+1} in $H(e)$. For example the history $H(e_1^4) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_2^1, e_3^1, e_3^2, e_3^3\}$ is identified by the vector $[4, 1, 3]$. This vector clocks are updated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

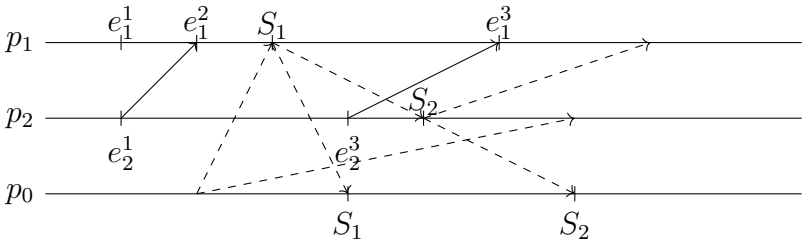
Taken two events e_i, e_j is possible to know if exists an event between the two:

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

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:

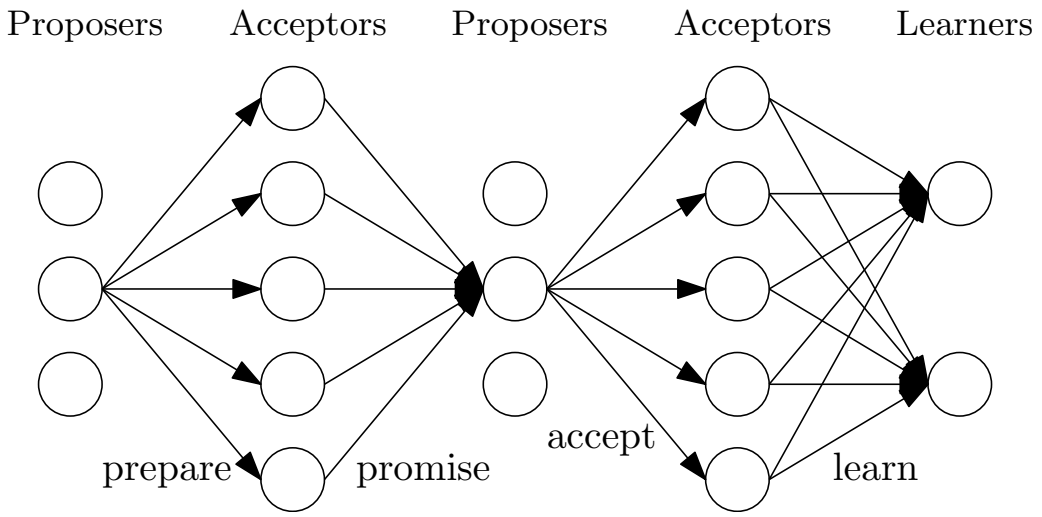
   - If no Acceptor has ever voted the value x can be chosen at random (value that comes from the client).
   - Else the value x is the last value voted associated with the largest value of last-round in the promises.

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

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.



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. This is the reason Paxos is not **live**, in the sense that if this continue to occur the protocol doesn't do nothing.

This protocol is **safe**, means that doesn't do anything wrong even in presence of more then f failures.

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.

Proof:

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 Coordinator in the Proposers that prepare the first round before the value to vote arrives. A round works like this:

1. Coordinator sends a message to all the Acceptors:
$$\text{C} \rightarrow \text{A} \qquad \text{prepare(curr-round)}$$
2. Acceptors respond to the Coordinator:
$$\text{A} \rightarrow \text{C} \qquad \text{promise(curr-round, last-round, last-value)}$$
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. Coordinator waits for a quorum of promises and after sends to every Acceptor a message that depends on the promises:
   - If no Acceptor has ever voted the Coordinator sends an acceptany message to tell the Acceptors to accept values from any Proposer.
   $$\text{C} \rightarrow \text{A} \qquad \text{acceptany(curr-round)}$$
   - Else the value x is the most voted value in the max last round.
   $$\text{P} \rightarrow \text{A} \qquad \text{accept(curr-round}, x)$$
4. When a value arrives to the Proposers they send an accept message to the Acceptors with the value x.
$$\text{P} \rightarrow \text{A} \qquad \text{accept(curr-round}, x)$$
5. Acceptors sends their vote to the Learners:
$$\text{A} \rightarrow \text{L} \qquad \text{learn(curr-round}, x)$$
6. 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 Coordinator will start another round.

In a round there could be different values voted, 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 protocol Fast-Paxos is safe. It also works with every simple protocol for leader election.

Proof:

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.

2.2.2 Multi-Paxos

Normally the problem is not only to agree on a single value, but on a series of values and on their order. This can be done with the protocol **Multi-Paxos** in which a lot of istances of Paxos (or Fast-Paxos) are run at the same time. Every istance agree on a value and the order in which the transactions are executed is the order of the istances.
The prepare message now contains also the istance of Paxos and multiple istances can be prepared at the same time, for example for n istances:

$$\text{C} \rightarrow \text{A} \qquad \text{prepare}(1\text{-}n, \text{curr-round})$$

Also Acceptors can promise to multiple istances:

$$\text{A} \rightarrow \text{C} \qquad \text{promise}(1\text{-}n, \text{curr-round}, \text{last-round}, \text{last-value})$$

And the acceptany message can be sent for multiple istances:

$$\text{C} \rightarrow \text{A} \qquad \text{acceptany}(1\text{-}n, \text{curr-round})$$

The accept messages have to be sent for every istance singularly and the istance to send a particul value is chosen by the Proposers. If the Prosopers are also Learners they know which istances have aleady chosen a value and can send the new values to other istances.

2.3 RAFT

This protocol uses leader election and is designed to make consensus on a sequence of values. Every server has a log with all the decisions taken over time.
The protocol works in **terms**, and a term is a period in which there is the same leader. In each term the leader election works as follows:

1. At the start everyone is a follower.
2. Someone becomes a candidate, so wants to become the leader, and send a message to the other servers asking for votes.
3. The other followers respond with a vote.
4. If a quorum of votes (50%+1) arrives the candidate become the leader for that term.

To choose how a server become a leader every server have a random timeout each term and after that it becomes a candidate. To resolve problems in case of more candidates another timeout is set (higher than the upper bound of the random one) and after this time if there is no leader the server goes to the successive term. This is the reason why this protocol is not live. In case a leader is chosen, the term is terminated when something bad happen to the leader, so in the ideal case there is only one term.
After chosing the leader, only it can write in the logs, so every transaction is sent to it. A transaction works in this way:

1. The leader write the transaction in its log with the term number.
2. The leader sends the transaction to every other server.

3. The followers write the transaction in their log and respond to the leader with an OK message.

4. If the leader recieve a quorum of OK, it marks the transaction as committed and can be executed.

When a server that is not the leader recieve a transaction, it marks all the previous as committed. If there is no next transaction the leader will send an empty one only to commit the last.
RAFT has some properties:

1. Election safety: there is at most one leader per term.

2. Committed entries are always in the log of the leader.

3. If to logs contain a value for the same index and term, it's the same value.

To ensure the property 2 there is a clause on the votes: a server can vote only for another server that has at least the same entries in the log.

2.4 Ben-Or

The **Ben-Or** protocol is a randomized protocol that permits to solve consensus and leader election. It's not used in practice but is important from a theoretical point of view.
The protocol has some properties:

- Agreement: all the process choose the same value (0 or 1)
- Termination: the protocol terminates at some point
- Validity: the value chosen is one of the input

The number of fault tolerated by the protocol is:

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

Each node follows this algorithm:

Algorithm: Ben-Or

```
def Ben-Or():
    preferences = input
    round = 1
    while true :
        send(1,round,preference)// broadcast of type 1
        wait for a quorum of messages of type 1
        if (messages with a specific value v)> n/2 :
            send(2,round,v,ratify)// broadcast of type 2
        else :
            send(2,round,?)
        wait for a quorum of messages of type 2
        if recieved a message with ratify v :
            preference = v
            if (messages with ratify v)> f :
                return v
        else :
            preference=random(0,1)
        round+=1
```

We assume retrasmission, so every messages will arrive at a certain point.
We are sure that:

- If exist two messages:
$$\left.\begin{matrix}(2, r, v, \text{ratify}) \\ (2, r, v', \text{ratify})\end{matrix}\right\} \implies v = v'$$
- If a process receive more that f messages with ratify v, then every process has recieved at least one of them. So, in the next round they are gonna agree and is safe to output the value v.
- In the worst case scenario (with exactly f failures) all process need to have the same value to have a quorum. The value is random between 0 and 1 so the probability to exit the protocol on a given round is:
$$P = \frac{1}{2^{n-f-1}}$$
So the expected number of rounds needed is 2^{n-f-1}. Theorically the protocol can't continue for infinite time and at some point it will terminate, making it live.

2.5 Failure detectors

Inside every process there is a failure detector D that the process can query to know if another process has failed.
Failures detectors are classified by 2 properties:

- **Completeness**: if a process crashes then D can see it
- **Accuracy**: if D says that a process is dead, is true

We define crash(σ) the set of process crushed during the run σ and Up(σ) the set of process not crushed during run σ. Also D_q represent the failure detector of process q.
The completeness can be of two types:

- Strong completeness:

$$\forall\sigma\ \forall p \in \text{crash}(\sigma)\ \forall q \in \text{Up}(\sigma)\ \exists t | \forall t' > t\ p \in D_q(t', \sigma)$$

- Weak completeness:

$$\forall\sigma\ \forall p \in \text{crash}(\sigma)\ \exists q \in \text{Up}(\sigma)\ \exists t | \forall t' > t\ p \in D_q(t', \sigma)$$

The accuracy can be of 4 types:

- Strong accuracy:

$$\forall\sigma\ \forall t\ \forall p, q \in \text{Up}(\sigma)\ p \notin D_q(t, \sigma)$$

- Weak accuracy:

$$\forall\sigma\ \exists p \in \text{Up}(\sigma) | \forall t\ \forall q \in \text{Up}(\sigma)\ p \notin D_q(t, \sigma)$$

- Eventual strong accuracy:

$$\forall\sigma\ \exists t | \forall t' > t\ \forall p, q \in \text{Up}(\sigma)\ p \notin D_q(t', \sigma)$$

- Eventual weak accuracy:

$$\forall\sigma\ \exists t | \forall t' > t\ \exists p \in \text{Up}(\sigma) | \forall q \in \text{Up}(\sigma)\ p \notin D_q(t', \sigma)$$

We can define a taxonomy of failure detectors based on this properties:

Accuracy → Completeness ↓	Strong (S)	Weak (W)	Eventual strong ($\diamond$S)	Eventual weak ($\diamond$W)
Strong (S)	Perfect (P)	Strong (S)	Eventual perfect ($\diamond$P)	Eventual strong ($\diamond$S)
Weak (W)	Theta (θ)	Weak (W)	Eventual theta ($\diamond\theta$)	Eventual weak ($\diamond$W)

With a P or $\diamond$P detector eventually there will only be a leader, so we could make Paxos live. This type of detectors don't exist in distributed systems.
Given a detector with weak completeness, we can create a strong complete one making the process q that knows that p is failed broadcast the information so that everyone will also know. So, detector with weak completeness don't really exist, making the only possible detectors being the Strong and Eventual Strong.

3

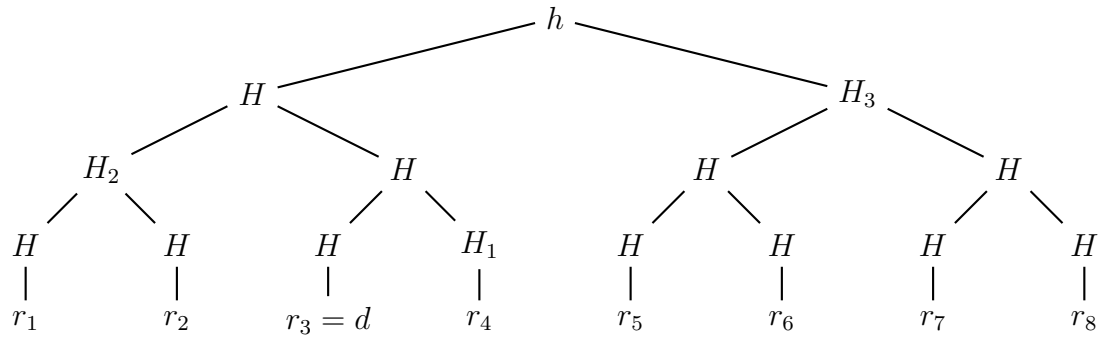
Uses of distributed systems

3.1 Cryptography

Cryptography is based on **hash functions**, a type of functions that takes in input data of variable length and gives in output a value of fixed length called **hash** (or digest).
An hash function $H(x) = h$ has some properties:

- If you know h it is very difficult to find the data x such that $H(x) = h$
- If you have x it is hard to find another data y such that $H(x) = H(y)$
- It is hard to find any x, y such that $H(x) = H(y)$

There are different families of hash functions like SHA and MD5 (is not used anymore because was broken).
A way in which hash functions are used is the **Public key Encryption (PKe)** that ensure confidentiality (there is no malicious entity that can read the messages). PKe uses asimmetric encryption, in which every person has a public key K_1 and a private key K_2. To send a message to someone I encrypt it with the public key of that person, in such a way that he can decrypt it using his private key. To be sure about a public key of someone I can use a certificate verified by an authority.
Another crypto tool is the **Merkle tree**, that is used in a case in which I have one record and I have to check if it's in a collection, only having the hash of the whole collection. For example, when you download a movie from a peer to peer system and you recieve a part of the movie from everyone and you need to check if that part is in the movie.
Let d be the record to check and $h = H(r_1, \ldots, r_n)$ the hash of the collection, the Merkle tree with $n = 8$ is constructed:



The Merkle proof that d is in the collection is (H_1, H_2, H_3). The verifier recieves d and the

Merkle proof and can verify if d is in the collection or not by doing:

$$d \xrightarrow{hash} d_1$$
$$d_1 H_1 \xrightarrow{hash} d_2$$
$$d_2 H_2 \xrightarrow{hash} d_3$$
$$d_3 H_3 \xrightarrow{hash} d_4$$

If d_4 is equal to h then d is in the collection. There is an H_i in the proof for every layer of the tree so the proof has logaritmic length.

3.2 Bitcoin

Bitcoin are represented by a file which contain the BTC, the public key as my name and the digital signature (file encrypted) using PKe. Let h be the hash of this file.

To transfer the property I create a new file which contain h, the new owner, the old owner and the signature of the old owner. Using a simple method like this creates a problem of double spending. To solve this problem a distributed system is used that gets consensus on which transaction commit and which not.

This system can't use Paxos because we don't know how many node are in the system, but more important:

- Is not safe if one node is malicious.
- Is made for small clusters, here there are too many messages.

The transaction are put togheter in collections called blocks and the system get consensus on all of them. The hash of a block is the first line in the successive block, creating a virtually connected **blockchain**.

A transaction has as input the previous transaction from which you have recieved the bitcoins that you want to spend, the signature of the previous owner and its name. As output the transaction has a list of new owners associated with the value that you want to transfer to that owner.

A block is made of an header, the hash of the previous block, the root of the Merkle tree of transactions, a Nuonce value (critical) and all the transactions in the block.

To have a valid block, the hash must have the K least significative bits equal to zero. The value used is $K \approx 30$ so the probability is very low.

Every server prepare a block and try to get a valid block, setting Nuance as 0, then 1, then 2 and so on. When it finds a value that creates a valid block it sends it to everyone and this become the new block. They do this because when a server finds a valid block, it recieves money as a transaction with no input. This is the only point where bitcoin are created and the amount is fixed, started as 50 BTC in 2009 and every two thousand blocks this amount is halved. This is why they are called miners.

If two miner find a valid block at the same time, half the miners recieve one block and half the other (fork). Some miners will try to extend the first one and some try to extend the second one. If again the two branches find blocks at the same time the blockchain is not resolved and committed. If you work on the smaller branch you will have to restart from the bigger. You

need to wait that 6 blocks are created on your branch to be sure is committed. On average, a block is created every 10 minutes but more than one hour is needed to commit a transaction.
This protocol is safe but it is not live if you are unlucky in every step of two branches (the probability is smaller every time), so is live with high probability. This protocol is not deterministic and is slow but this are not problems.

3.3 Dark Web and Tor

There are different methods to get privacy when we are using a browser and each method can give us privacy against different people:

- Incognito mode: privacy against other users of the same device.
- VPN: privacy against internet provider, but you need to trust the VPN provider. Doesn't give privacy against the government because they can ask the data from the VPN.
- Tor (The Onion Router): privacy against internet provider and government without giving trust to external devices.

Tor has a lot of servers (called relay), when using Tor you download the list of servers (IP and public key). You select some of them (tipically 5) and you send a packet to the first server (called guard), which sends it to the second one, and so on. The last server (called exit point) sends the packet to the website that you want to access. To ensure privacy each server must know only the last server from which the packet is sent and the next one where it has to send it.
To do that you use a secret key with each server the packet has to visit. Let $K_1, ..., K_5$ the secret key for the 5 servers, the packet will be encrypted starting from K_5 all the way to K_1 on the outside layer. Each relay will decrypt the outside layer and understand where it has to send the packet but without knowing anything else.
To share the secret key between the user and a server the protocol called **Diffie-Hellman** is used:

- The user choose a random number x and the server choose a random number y. The numbers are chosen in Z_p in which there is a generator g such that $Z = \{g^0, g^1, \dots, g^{p-1}\}$. Knowing g^x find x is a problem called distributed log and its NP-Hard.
- The user sends g^x encrypted with the public key of the server.
- The server decrypt the message and can calculate the secret key $K = g^{xy}$. The server sends to the user g^y and the hash of the key.
- The user calculate K and use the hash to control if the key is correct.

This protocol is used to share the secret key with the first relay, after that you share the secret key with the second server passing through the first and so on. Is important to not share the secret kay indipendently with every server because someone in the last relay could decrypt the message.
This system is slower than normal connection to a website and the latency is very high because packets has to travel the relays. The bandwidth is also the minimum of all the links.

3.3.1 Dark Web

Using Tor the users have privacy but not the website they connect to. The Dark Web is when also the website uses Tor and none knows where the site is.

To connect to a site in the dar web you connect to a Distributed Hash Table (DHT) in which are listed the introductory points to the site. You talk to the introductory point using Tor and decide a meeting server (called randevouz) where you connect with the site (both using Tor).

3.4 DNS and CDN

3.4.1 DNS

The **Domain Name System (DNS)** is a distributed system designed to translate memorable domain name strings into numerical IP addresses.

IP addresses are difficult to remember (four bytes as a number), so DNS provides human-readable names.

The structure of domains is hierarchical (top-level domains like `.it` have numerous subdomains). The resolution process is iterative, starting from a Root Server (there are 13 root servers in the world) and is directed down the hierarchy until the IP is obtained.

The DNS LookUp process works as follows:

- The user do a DNS Query with the name of the site they want to know the IP of.
- The local DNS resolver connects to the Root DNS server asking the top-level domain (TLD).
- The Root DNS server respond with the TLD nameserver names.
- The local DNS connects to the TLD nameserver and asks for the site requested by the user.
- The TLD nameserver respond with the Authoritative nameserver names.
- The local DNS connects to the Authoritative DNS resolver asking for the site.
- The Authoritative DNS respond with the IP of the site.

The DNS use a caching system where all retrieved IP information is cached by the DNS resolver to avoid repeating the entire resolution process. This break consistency if a domain changes IP address, to reduce this problem the IP changes are propagated to everybody (this takes time) or giving Time-To-Live (TTL) to the cached element. The caching system gives partition tolerance and aviability at expense of consistency.

The 13 logical root server have houndreds of replicas that share the same IP, so when a user queries a server they are routed to the closest replica. This IP is announced by a large number of Autonomous System (AS) using the BGP routing protocol.

The BGP (Border Gateway Protocol) is the routing protocol used for communication among Autonomous Systems (AS) on the internet. The internet is fundamentally composed of these Autonomous Systems, which, as their name suggests, operate autonomously. While routing within a single AS can use internal rules, communication between different Autonomous Systems follows the BGP protocol.

BGP routing works like that:

- Announcing IPs: Under the BGP protocol, an Autonomous System can announce which IP addresses are located within its system.
- Propagation and Path Building: These announcements are propagated across other AS. Every time an announcement is propagated, the receiving AS adds information indicating the path that must be followed to reach those IPs.
- Path Construction: If Autonomous System A announces IPs, saying traffic must go through A, and then this announcement is sent to Autonomous System B, B will propagate the information saying that to reach those IPs, you have to go through B, and then A. This process builds the paths necessary to route traffic across the internet.

This protocol is vital for reliability, as demonstrated by the use of Anycast for DNS root servers: the IP address of a root server is announced by all the many Autonomous Systems that host a replica of that server, routing users to the closest copy.

3.4.2 CDN

CDNs were developed to make access to internet content much faster and more reliable. HTML pages text are full of media content (pictures, videos) which require multiple HTTP connections to retrieve and take significant time to download. The Akamai's Solution was to build a globally distributed network of servers (CDNs) that hold copies of this media content.
The CDN mechanism consist in:

- HTML pages are modified ("Akamization") by changing the links to media files to include the CDN's domain (for example cnn.com.akamai.com/pictures/34.jpg).
- When a user's resolver queries the Akamai name, the Akamai DNS system intercepts the request.
- Akamai's DNS analyzes the location of the request and returns the IP address of the closest CDN server (Point of Presence) holding the content. The next HTTP connection is then made to this closer server, not the origin server.
- The CDN DNS system is also used to perform load balancing, routing users to a less-loaded server even if it is slightly farther away.

CDNs are distributed systems that prioritize availability and partition tolerance. For applications like news distribution or social media consistency is traded for availability. It is acceptable to retrieve a picture or post that is a few minutes old if it ensures the user can access the system. Conversely, applications like banking systems or plane ticket sales require high consistency, even if it means reduced availability.

3.5 Bit-Torrent

Bit-Torrent is a peer-to-peer system in which people can download a file from a server, but also download part of the file from other people in the system. After a while, we can download the

file even if the server is down, by downloading some parts from a person and other parts from other person simultaneusly (parallel download).
The components of the system are:

- Torrent file (.torrent): contains the url of the tracker, the file name, the piece length in which the file is divided (256 KiB), the protocol version, the length of the file, the root hash of the Merkle tree and the piece layers ($R_1, \ldots, R_N$).
- Tracker: server that coordinates people.

When downloading the pieces of the file from different sources the Merkle tree is used to know if all the pieces are correct. This is done by checking if the hash of the pieces in the order specified in the torrent file is equal to the root of the Merkle tree.
The order in which the pieces are downloaded can be:

- Random: balance the system and is the faster method.
- Rarest first: avoids possible rare pieces to miss but is the slower method.

Bit-torrent start from random to ensure that we have some pieces to upload and after that it goes for the rarest.
In a Bit-torrent system a user can have different roles:

- Free-rider: you only want to download without upload
- Tit for Tat: you can only download if you upload
- Leacher: you are downloading the pieces
- Seeder: you have all the pieces but remain in the system only to upload

The tit for tat is done by an algorithm called **Unchoking algorithm**, that consists in unchoking someone randomly every time t following the tit for tat rule (priority to the ones that you are downloading from or uploading to). This algorithm is optimistically unchoking.
When you are downloading and you are stuck with the last piece, you are allowed to obtain subpieces from different peers. This is called end game mode.
The system has:

- Partition tolerance (pretty good): is possible that a rare piece is cutted out or the cut doesn't contain a seeder.
- Consistency: no possible to lose, not defined.
- Availability: is not always guaranteed but is good (seeder, tracker and a subcut can work well).

The system can be broken in different ways:

- Be a seeder and lock who is uploading from you.
- Pollute the system with fake pieces (doesn't work weel because the hash is different).
- Pollute the system with fake entire files.

3.6 Distributed Hash Tables (DHT)

A distributed file system is like a disk, but implemented by several distributed server, the most famous one is the InterPlanetary File System (IPFS).
This system are content-address, which means that the name of the file is the hash of the file and to access them you need to know this content id (CID).
This systems have some properties:

- There is no replication.
- There is no cryptography, files are plain text and you need to trust the system.
- There is no access control, so anyone wit the CID can access the file.
- There is no modification of the files.

Every file is divided in chunks and distributed. To know how a file is stored and how to rebuild it you need to check the Distributed Hash Tables. This system is consistent because file can't change, but has low availability and low tolerance to network partition due to the fact that there is no replication.
There are some improvement you can do:

- Encrypt the file before saving it in IPFS to ensure confidentiality.
- Save a file multiple times to have replication.

When you save a file you can't guarantee integrity because a file could be ona server that at some point leaves the system, losing the file. To avoid it you can have a "pinned" file, that is guaranteed to not be deleated and be always reachable.
The system use DHT to retry your file from an hash value, to do this two protocols are used: Chord and Kademlia. Chord is a protocol usually used in universities because is better accademically and Kademlia is the one used in the real world because it has better performance when nodes enter and exit the system frequently.

3.6.1 Chord

Given a hash value with m bits, take a ring with the numbers that goes from 0 to $2^m - 1$ and project all the corresponding servers and files on the ring. The rule is that every file must be stored in the next server on the ring. For example if we have a server A with hash 150 and a server B with hash 200, all files with hash from 151 to 199 will be saved on server B. This protocol has load balancing because the server are generally uniformally distributed on the ring.
To read a file you start by asking a random server A where is the file, if you are lucky you find it, if not you use the finger table of A. The finger table of a server A contains all the server such that their IP is $H(A) + 2^x$ in which $x \in [0, m-1]$. You use this tale to ask the server with the bigger hash value lower than the CID you want. You contact this server because in the finger table of a server there aren't all the server in the system so in this way we are sure we don't go too far higher. You repeat this until you find the file, the max number of hop needed is $\log n$, in which n is the number of servers in the system.

This protocol is not efficient when new server joins because it needs to contact the next server to send all the file that are in you server now and you need to update all the finger table and create your own (same in the opposite way when a server leave). Theoretically this process has $\log^2 n$ but is more slower in reality.

E

Exercises

E.1 Exercises on cuts and global states

E.1.1 Exercise 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.

$$\begin{cases} C_1 \text{ consistent} \implies \forall e \in C_1 \wedge e' \to e \implies e' \in C_1 \\ C_2 \text{ consistent} \implies \forall e \in C_2 \wedge e' \to e \implies e' \in C_2 \end{cases}$$

$$\forall e \in (C_1 \cap C_2) \wedge e' \to e \implies e \in C_1 \wedge e \in C_2 \xrightarrow{\text{consistence}} e' \in C_1 \wedge e' \in C_2$$
$$\implies e' \in (C_1 \cap C_1) \implies (C_1 \cap C_2) \text{ consistent}$$

E.1.2 Exercise 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.

$$\begin{cases} C_1 \text{ consistent} \implies \forall e \in C_1 \wedge e' \to e \implies e' \in C_1 \\ C_2 \text{ consistent} \implies \forall e \in C_2 \wedge e' \to e \implies e' \in C_2 \end{cases}$$

$$\forall e \in (C_1 \cup C_2) \wedge e' \to e \implies e \in C_1 \vee e \in C_2 \xrightarrow{\text{consistence}} e' \in C_1 \vee e' \in C_2$$
$$\implies e' \in (C_1 \cup C_1) \implies (C_1 \cup C_2) \text{ consistent}$$

E.1.3 Exercise 3

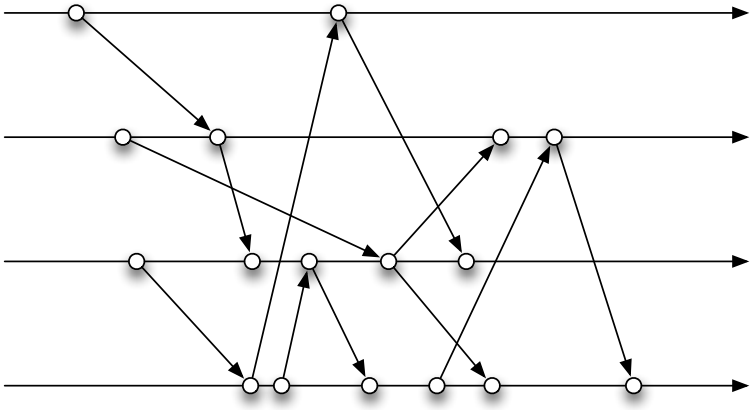
Show that every consistent global state can be reached by some consistent run.
Taken the run $R = e_1, \ldots, e_n$ where all the event $e \in \Sigma$ are at the start in the same order as they appear in Σ, R is consistent and reaches the global state Σ.

E.1.4 Exercise 4

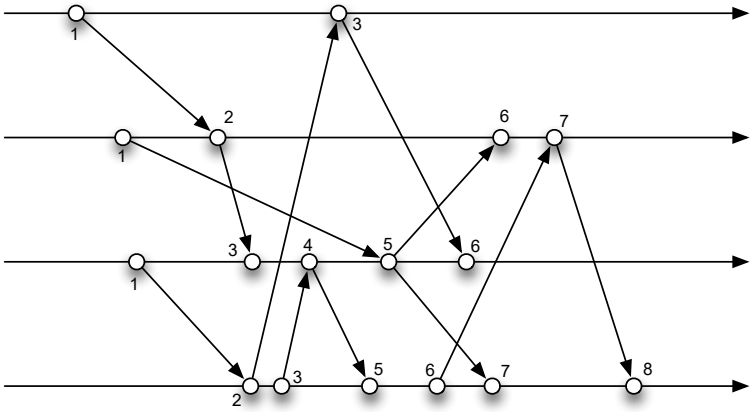
Let C_1 and C_2 be two consistent cuts. If C_1 is a subset of C_2, then C_2 is reachable from C_1. (There exists a consistent run that reaches C_1 and then reaches C_2).
Taken the run $R = e_1, \ldots, e_n$ where all the event $e \in C_1$ are at the start in the same order as they appear in C_1 and after them there are all the event $e \in C_2$ in the same order as they appear in C_2, R is consistent and reaches the cut C_2 passing through C_1.

E.1.5 Exercise 5

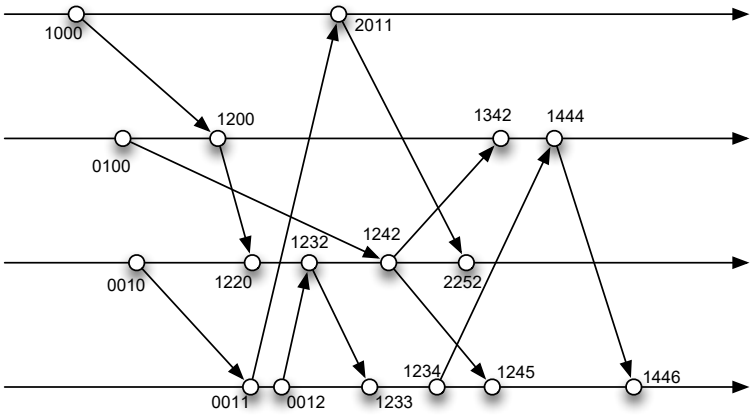
Label with proper logical clock all the events of the distributed computation in the image below. (You can consider events that receive a message and immediately send it as single events).



Logical clocks:



Vector clocks:



E.2 Exercises on distributed snapshots

E.2.1 Exercise 6

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

For every events $e'_i \rightarrow e_j$, if $e \in S$ it means that the process p_i must have taken the snapshot after e', because the channels are FIFO. So if the snapshot message from p_i to p_j is arrived after e i means it was sent after e' and $e' \in S$.

E.2.2 Exercise 7

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

An event e_i could happen after the snapshot taken by p_i but before an event e_j taken in the snapshot of p_j (event e_i and e_j must be concurrent).

E.2.3 Exercise 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.

Even if the system was never in the state S represented by the sanpshot, the only difference between S and the real state happened are concurrent events. So, every check on the system, like deadlock detection works the same in the two cases.

E.2.4 Exercise 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).

If there is a global clock the monitor can send at time t a "take snapshot" message with a timestamp $t + \Delta t$ where Δt is the maximum time required by a message to arrive. When a process recieves the message takes a snapshot at timestamp $t + \Delta t$ and sends it to the monitor.

E.2.5 Exercise 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).

When broadcasting its snapshot every process sends a flag to signal if its channels are empty. When a process recieves all the snapshot from other processes, if all signaled that they have empty channels te process sends its snapshot to the monitor.

E.2.6 Exercise 11

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

A possible problem is the following:

There are two events $e'_i \rightarrow e_j$. The process p_i takes a snapshot in which $e' \notin S_i$ and after the events e' occur sending a message. If the channels are not FIFO the message of event e' could arrive before the snapshot message, causing p_j to take a snapshot in which $e \in S_j$, so the snapshot is inconsistent.

E.2.7 Exercise 12

Let S_0 be the global state when the Chandy-Lamport snapshot protocol starts, S be the global state built by the protocol, and S_1 be the global state when the protocol ends. Show that S is reachable from S_0 and that S_1 is reachable from S. Remember that S might not have happened.

From the run R_0 that reaches S_0 can be added all the events $e \in (S - S_0)$ in the same order in which they appear in S_0, creating a consistent run R that reaches S from S_0. The same can be done with the events $e \in (S_1 - S)$, creating a consistent run that reaches S_1 from S.

E.3 Exercises on atomic commit

E.3.1 Exercise 13

Give an ACP 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?

An ACP that satisfies the converse of condition AC3, is the 2 Phase Commit protocol. Is not good to enforce the condition because to enforce that the fail tolerance must be 0. So, with every failure the transition is aborted.

E.3.2 Exercise 14

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

We have an execution in which the coordinator fails after receiving the votes, but before sending the decision to the other processes. If the votes were all Yes the cooperative termination procotol can't unlock the system because no process knows the decision yet. So, the system is blocked.

E.3.3 Exercise 15

Show that Paxos is not live.

In Paxos a round can be blocked by a successive round if the successive sends a prepare before acceptors voted in the previous round. The acceptors will respond with a promise to the successive round and will not vote in the previous round. This can happen infinite time in a row, so the protocol is not live.

E.3.4 Exercise 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).

If a value v is accepted in round i, this means that a quorum of acceptors will always vote v, so is the only value that can be proposed and accepted in successive rounds. The problem is that if the acceptors voted three different values and no value has a quorum of votes, the system is in livelock.

E.3.5 Exercise 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?

In the best case the number of messages sent is $(3 + l)a$, with a acceptors and l learners. The messages can be reduced sending the votes only to the learner that associated to the round (if learners and proposers are the same).

E.3.6 Exercise 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.

If the value is picked at random, a less voted value could be picked and this is unsafe because the other value could have been accepted in previous rounds.

E.3.7 Exercise 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.

If two value are found in promises, choosing the most voted for the fast round is safe because only the most voted could be accepted in previous round, works the same as Fast-Paxos.

E.3.8 Exercise 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?

To ensure safety only the value chosen by the proposer after receiving the promises could be accepted in previous round. So, the quorum could be set as 1 votes more than the maximum votes that the second most voted value could have received. If $|v|$ is the number of votes for

the second most voted value and d the number of promises not received yet, the quorum can be set as:

$$Q = |v| + d + 1$$

E.3.9 Exercise 21

Show that Raft is not live.

A term is skipped if a timeout is reached. This timeout can be reached infinite times in a row, so the protocol is not live.

E.3.10 Exercise 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.

Suppose the leader p doesn't have a log l hat is committed. To be committed l must be in at least the majority of logs, so this majority will not vote for p and p can't become leader, so is impossible that l is committed.

E.3.11 Exercise 23

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

For every round, the probability to terminate the protocol is $p < 1$. The probability to not end before a round n is:

$$P(n) = (1 - p)^n$$

So, as n grows the probability becomes 0:

$$\lim_{n \to +\infty} (1 - p)^n = 0$$

E.3.12 Exercise 24

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

A run that never terminates is a tun in which the majority doesn't choose the same value. An example is a system with 3 processes and one failed. So, to terminate the protocol each process must choose the same value. If we are unlucky the 2 processes could continue to choose different values.

E.3.13 Exercise 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, then the probability that the protocol terminates after x rounds (or more) is smaller than α^x, for some α.

The lower probability to terminate the protocol is when there are 2 fails and the probability to terminate every round is:

$$p = \frac{1}{2^{n-f-1}} = \frac{1}{2^{5-2-1}} = \frac{1}{4}$$

So, the probability to terminate after round x with 2 fails is:

$$P(x) = (1-p)^x = \left(\frac{3}{4}\right)^x$$

For any number of fails, the probability to terminate every round is $p' \geq \frac{1}{4}$ and so, to terminate after round x:

$$P(x) = (1-p')^x \leq \left(\frac{3}{4}\right)^x$$