“Distributed Systems” Notes

Prof. Alessandro Mei. Notes written by Alessio Lucciola during the a.y. 2022/2023.

1. Distributed Computation

1.1 Introduction

A **distributed system** is a system in which there are n **processes** distributed on a network. These processes cooperate to complete a single task. Each of these processes are running somewhere (for example, one may be running in some server in Rome, another one in New York) and they are connected by a **communication system** which is made of different **channels**.
Channels can be:

- **Reliable**: this type of channel assures that messages will arrive. Messages can be received out of order (a message sent first can arrive later than others) or with a FIFO structure (first in, first out). An example is the TCP/IP model.
- **Unreliable**: this type of channel doesn’t assure that a message will arrive in fact they can be lost. An example is UDP protocol.

In distributed systems, we assume that channels are **reliable**. If the message is lost then it is sent again and it will eventually arrive at its destination.
There are 2 types of systems:

- **Asynchronous**: there is no time bound for a massage to arrive.
- **Synchronous**: there may be a bound (for example, we can assume that in X seconds the message will arrive).

Most systems are asynchronous since it is not possible to assume a relative speed for processes. There are no bounds on the relative speeds of processes and there exist no bounds on message delays. What we assume is that **process speed is not predictable** (the system should work regardless of the speed) and **there is no global clock** so if there is a clock in each server then in a certain time there will drift (so they lose synchronisation).
Also, processes can:

- Correct: they properly work and they do what they have to;
- Fail: they can’t end their task properly. Failures can be:
    - Crash failures: Processes stop working;
    - Byzantine failures: They don’t actually stop working but they can do anything (apart from their given tasks) and this can bring security problems;

1.2 Processes timeline

A **distributed computation** describes the execution of a distributed program by a collection of processes. The activity of each sequential process is modelled as executing a sequence of events. An event may be either **internal** to a process and cause only a local state change, or it may involve communication with another process. Without loss of generality, we assume that communication is accomplished through the events *send(m)* and *receive(m)* and that match based on the message identifier m. In other words, even if several processes send the same data value to the same process, the messages themselves will be unique. Informally, the event enqueues messages on an outgoing channel for transmission to the destination process. The event , on the other hand,

corresponds to the act of dequeuing a message from an incoming channel at the destination process. Clearly, for an event to occur at process , message must have arrived at and must have declared its willingness to receive a message. Otherwise, either the message is delayed (because the process is not ready) or the process is delayed (because the message has not arrived).

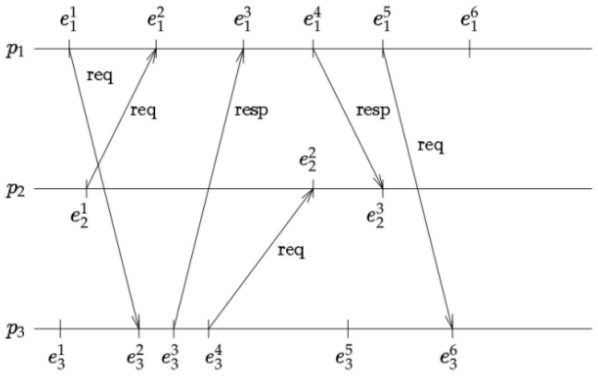
Now, we can represent the evolution in a distributed system with a **Space-Time diagram**.
Given a certain process we can represent the amount of time in which events occur. So, we'll have different events for a process in the timeline and also different processes with their own events.
Now, it is possible to define the local history of a process: The **local history** of process p_i during the computation is a (possibly infinite) sequence of events $h_i=e_i^1, e_i^2, ..$ where e_i^1 is the first event executed by process i and so on. This is a **canonical enumeration** and corresponds to the total order imposed by the sequential execution on the local events. The **global history** of the computation is a set $H=h_1,..,h_n$ containing all of its events.

In an asynchronous distributed system where no global time frame exists, events of a computation can be ordered only based on the notion of "**happen-before**": if an event e sends message m and e' receives message m, then e → e'. In general, the only conclusion that can be drawn from e → e' is that the mere occurrence of e' and its outcome may have been influenced by event e. It is possible that for some e and e', neither e → e' nor e' → e: we call such events **concurrent** and write **e||e'**.

So, for example:

- e^1_1 happens before e^2_1 since they are performed from the same process so the first one directly influences the latter;
- e^1_2 happens before e^2_1 since the first is a query and the latter is the answer (e.g. the first event asks the process p_1 some query, and it responds back so it generates another event)

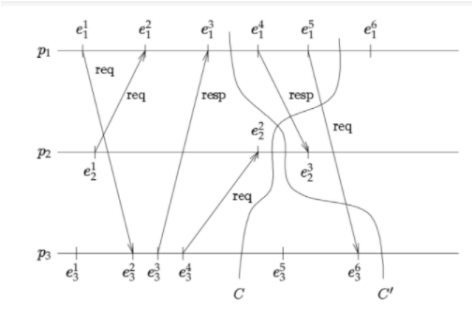
We can imagine that there is an event e at a very low level (for example a single instruction like x+=1). In real situations there are usually most important events which are not made of single instructions like the one in the example but a whole set of instructions (like the sending of a message).



These events are performed by different processes. For example process p_1 sends a message through the event e^1_1 to process p_3 which is received by the event e^2_3 and so on. The timeline shows when all the events occur in time. Every **run** of the system can have a different timeline, so if we run the same DS, these events can happen at different times (mainly because each process can perform at a different speed at each computation). The system must work in every possible run. There are also impossible runs, like the ones where an event that receives a message happens before an event that sends a message: **these runs are not consistent**.
For example:

- R: e^1_1 e^2_1 e^1_2 is not a possible run because e^1_2 must happen before e^2_1;

We need a definition of "happened before". In fact, we can notice that event e^1_1 has to happen before, for instance, event e^2_1, because it can change the context of the process. We can't say the same thing for events $e1_2$ and e^1_3 because they can't affect each other (they are concurrent), so it's not important for us the order in which they take place. Generally, given → = "happened before":



- Given e_i^k and $e_i^l \in h_i$ and $k < l$, then $e_i^k \rightarrow e_i^l$.
- If $e_i = \text{send(m)}$ and $e_j = \text{receiving(m)}$, then $e_i \rightarrow e_j$.
- $e^i \rightarrow e^{ii} \rightarrow e^{iii}$ then $e^i \rightarrow e^{iii}$ (transitivity).

→ is the smallest relation with these 3 properties.[1]

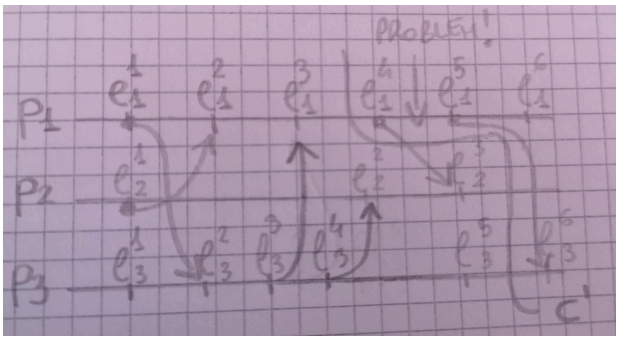
1.3 Consistency

If we have a run R and we stop it at some point, we can make a **cut** which is a snapshot of the system in a certain instance of time that is a subset of the history of the computation in which every process has stopped after a few events.

A cut is a run stopped at some point. A cut C is **consistent** if and only if:

$$e \rightarrow e^i \wedge e^i \in C \Rightarrow e \in C$$

It means that if a message is sent among a certain cut, it must be received in the same cut. It is not possible for an event to receive a message that it is sent in the next cut (when the system resumes).



EX. Notice that C is consistent since the property above is satisfied while C' is not because e^3_2 receives a message from event e^4_1 sent in the following cut. It's like if the message is sent in the future and received in the past and this obviously can't happen in real life.

Given two different consistent cuts C and C', it is possible to demonstrate that the intersection C ∩ C' is consistent too.

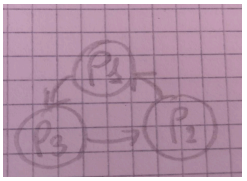
$$e \rightarrow e' \wedge e' \in C1 \Rightarrow e \in C1 \;\&\; e \rightarrow e' \wedge e' \in C2 \Rightarrow e \in C2$$

$$\text{Demonstration: } e \rightarrow e' \wedge e' \in C1 \cap C2 \Rightarrow e \in C1 \wedge e \in C2 \Rightarrow e \in C1 \cap C2$$

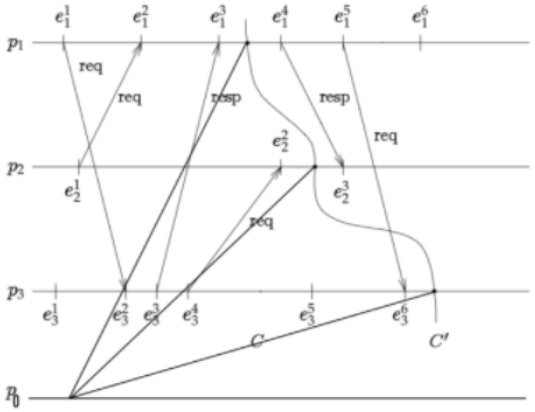
Note: it is also possible a demonstration for union.

1.4 Deadlock

In every system, when a process P1 sends a query (or a request) to a process P2, it usually waits for a response from the P2 process. A deadlock can occur in this situation: **deadlock** is any situation in which no member of some group of entities can proceed because each waits for another member, including itself, to take action, such as sending a message or, more commonly, releasing a lock.



We know that deadlock can happen in single systems, but there's no difference with distributed ones. We have *two* main approaches, **deadlock avoidance** and **deadlock removal**. The latter is the most practical and consists of checking periodically if the DS is in deadlock and removing it. A possibility is to add a process, let's say **P0** (also called **monitor**), that can ask the state of the system to every other process (for example, it can ask another process if it is waiting for some response to proceed). We have to remember that the system is asynchronous, so the P0 message can arrive in different moments for different processes. When P0 receives a message, it can build a graph that represents the "waiting situation" between processes. A **cycle** in the **waiting graph** is a deadlock.



Let's analyse this situation: P0 sends a message to all the processes which arrive at a certain time (just look at the spots in the diagram). Processes respond with a message that specifies their current situation. We can easily notice that P2 makes a query to P1 (event e^1_2) and *P2 is*

[1] Smallest relation: all three properties must be satisfied in order to say that an even

still waiting for a response from P1. Also P3 sends a message to *P2 and it is still waiting for a response from P2* (event e^4_3). Again P1 sends a message to P3 (event e^5_1). P3 can say to P0 that P1 is waiting for its response and eventually P2 will say that P3 is waiting for its response (so P1⬅P3⬅P2⬅P1, ⬅ stands for "waiting for"). Thus, we have a deadlock, but we know that in the system there isn't one. In fact, we can see how every query has its own response. So, why did this happen? Due to the asynchronous nature of the system, P0 can detect a deadlock where there isn't one. This happens when P0 takes a snapshot of the system that is an **inconsistent cut**. Hence, we have to make *snapshot protocols* that consider consistent cuts of the system state.

1.5 Clock

Now, we go back to the ghost deadlock problem and assume we want to find a way to solve this problem. Let's assume that we have a **Global Real Clock** (or simply RC) that every process can use to be perfectly aligned. Something like this doesn't exist in real life, but can be well simulated with protocols like NTP (Network Time Protocol). We can send a message to P0 after every event and assume they will arrive before a certain time t plus some delay Δ. At some moment in its timeline, P0 will receive these messages, that represents a run of our system. This is an inconsistent run of the system because we don't know in which order these messages will arrive since our system is asynchronous.
Thus, we can label these messages with the RC so that it is possible to **order these messages** when P0 receives them. This is a consistent run because we know that:

Clock Condition:

$$e \to e^i \Rightarrow RC(e) < RC(e^i)$$

So an event e happens before another event e' if the timestamp of e is smaller than the timestamp of e'.
Note: we don't have double implication, because e and e^i aren't necessarily related (they could be concurrent).

Instead of using the RC we can invent our clock, LC, that doesn't use the RC but it has the clock condition described above. We can assume that every process tags events with sequence numbers and these numbers represent LC. Informally, the rule that we use to build this diagram is:

- If we have a new internal or sending event, increase by one;
- If we have a receiving event, give it the max between preceding internal event and sending event, plus one;

$$LC(e_i) := \begin{cases} LC + 1 & \text{if } e_i \text{ is an internal or send event} \\ \max\{LC, \boldsymbol{LC}(m)\} + 1 & \text{if } e_i = receive(m) \end{cases}$$

By construction, OC preserved the RC rule. Thus, P0, by ordering labels, has now a *consistent run of the system*. This clock is named **Lamport's clock (LC)**.

EX. At the beginning we have LC=0. We start from process P1 and label the first event with LC+1=1, again we label the second internal event with LC+1=2. Now go move to process P3 where we have a starting internal process with LC+1=1 and a second event LC+1=2 (notice that this is a message sent by P1 so we have to choose the max between preceding internal event and sending event plus one but in both cases we have that the result is 2). Now, another internal event LC+1=3 that is a sending event to P1. P1 receives this message after the second one so this event will have max{LC=3, LC(2)}=3+1=4 as clock value. Keep repeating this process for all events.

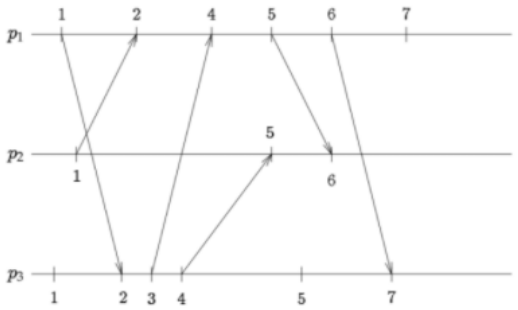


Figure 4: Our clock (actually, Lamport's clock 😅).

We still have a little problem, in fact given the possibility of identical labels, P0 can order these labels reconstructing a possible run, but not the exact run that's happened. Besides, we have another problem. We aren't sure if some messages with a lower timestamp than the last one we've received will arrive later (remember, we have asynchronous systems) (to make it simpler, the label of an event that occurred before any of the already received event labels).

We can define the **delivery rule** (DR1): At time t (now), *deliver* all messages in order whose timestamp is smaller than t − Δ. This means we now have a time bound, called Δ (delta), in which we are sure that every message will arrive.
Let's put ourselves in an asynchronous system and start using the previously defined Lamport Clock (LC), thus we label the event with sequence numbers as we know.
We have to find a way to know when a process must deliver a message. Let's introduce the definition of stable message:
Definition 1 A message m is **stable** if no future message with timestamp smaller than TS(m) can be received.

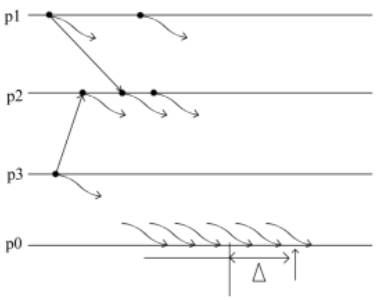
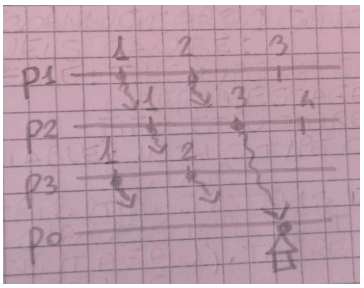


Figure 5: Delta time bound on synchronous system.

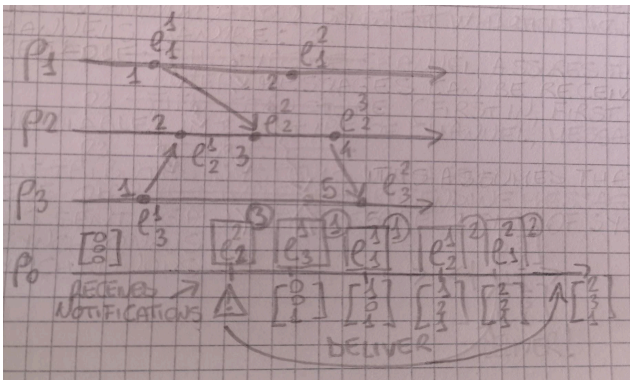
EX. We have a few events sent by different processes and each of them sends a notification to P0. The first notification received by P0 is the third one sent by process P2. Notice that the timestamp is equal to 3, but this message is not stable since other messages with a lower timestamp have to arrive.



Now we can define a new **delivery rule (DR2)**, using Lamport's clock: Deliver all received messages that are stable in timestamp order.

We can implement this rule using a FIFO structure. Thanks to the clock, it is possible to have a history for all processes. This history must be updated every time a new event is created from a process. This history can be represented through a **Vector Clock**. A vector clock is a vector owned by every event E in which every position shows the number of events happening in every process P before the event E. So P0 just has to remember the value of the highest TS received from process i inside of a vector of length n (where n is the number of processes).

EX. We start from timestamp "0" (VC = [000]) then we receive a notification from e^2_2 with timestamp TS(e^2_2)=3: we cannot deliver the response because we are still waiting for the messages with timestamp smaller than "3" from other processes. Then P0 receives a notification from e^1_3 with timestamp "1". P0 can deliver a response because the other processes have VC(P1)=VC(P2)=0 (VC = [001], update the vector so that the TS of the last event of P3 is 1). Same for the notification of e^1_1. Then e^1_2 sends a notification with timestamp TS(e^1_2)=2: it can be delivered since there are no messages with TS<2, so VC=[121]. For e^2_1, it can also be delivered since VC=[121] so VC becomes VC=[221]. Now the message from e^2_2 can be delivered so VC=[231].



So in order to say that a message with TS=6 has to check if it has received all messages with TS at least 5 from all the other processes but this solution is not so great since P0 could have to wait forever (think of the case if one of the processes fails). We can find a better solution.

Firstly, we want to enforce the clock condition defining the **strong clock condition** (SCC):

$$TS(e) < TS(e') \Leftrightarrow e \rightarrow e'$$

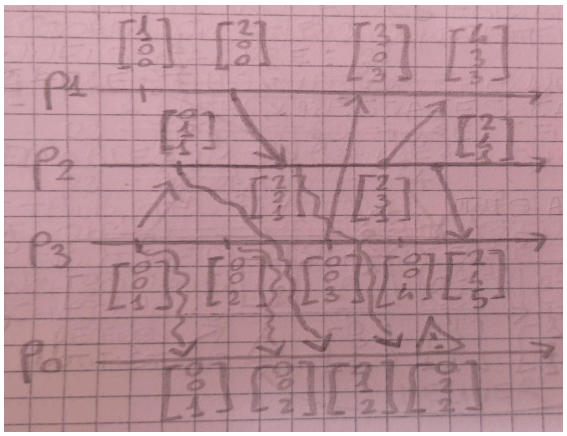
Remember that normal clock condition does not have the double implication in fact events could be concurrent.
Now we can define the **causal history of an event**:
Definition 2 The history of an event e_1 is: $\ominus$**(e) = {e'∈H : e' → e} U {e}**.
The history of an event is the set of events that happen before that event (so the list of events that influenced it).
EX. $\Theta(e^1_3) = \{e^1_3\}$
$\Theta(e^1_2) = \{e^1_2, e^1_3\}$
$\Theta(e^2_2) = \{e^2_2, e^1_2, e^1_3, e^1_1\}$

From this definition, we can get **e → e' ⇐⇒ Θ(e) ⊆ Θ(e')**:

Proof: If e → e' then by definition of causal history, it is immediate that Θ(e) ⊆ Θ(e').
If Θ(e) ⊆ Θ(e'), since we have included e in its history then e∈Θ(e') so, by definition, e happens before e'.

So, to store the history of an event Θ(e), it is enough to use a vector to remember what is the last (the biggest) timestamp for each process in the cut This is basically a vector clock which is represented as an n-dimensional array and it has the strong clock condition.
Returning to our first example, we can now tag the events with these vectors as their labels:

EX.



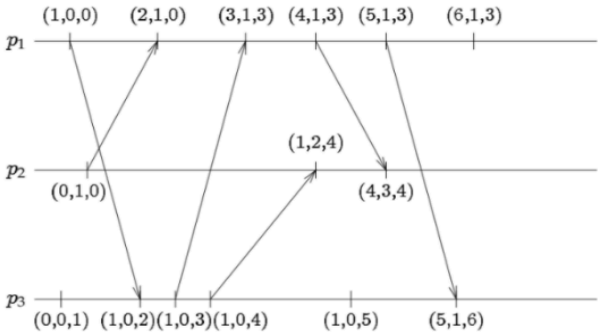


Figure 7: Vector clocks.

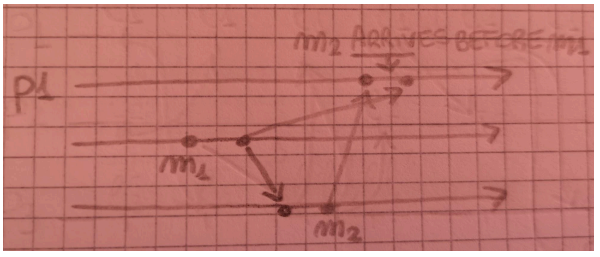
In order to build the vector clocks follow this algorithm:

1. Initially all clocks are zero.
2. Each time a process experiences an internal event, it increments its own logical clock in the vector by one. For instance, upon an event at process i, it updates:
$$VC_i[i] \leftarrow VC_i[i] + 1.$$
3. Each time a process sends a message, it increments its own logical clock in the vector by one (as in the bullet above, but not twice for the same event) and then the message piggybacks a copy of its own vector.
4. Each time a process receives a message, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element). For example, if process P_j receives a message m from P_i, it updates by setting:
$$VC_j \leftarrow max(VC_j[k] + 1, VC_i[k]), \forall k.$$

In general, A message can be delivered if all messages in the causal history have arrived. So this message is received by p0 but not delivered.

The comparison between timestamps lets us know if two events are related with the → relation or not (concurrent).
EX. e^2_2 = [2 2 1], e^3_3 = [0 0 3]
The history of event e^2_2 can't be included in the history of event e^3_3 since, for example, e^2_2 has 2 events in the first position while e^3_3 has 0 positions.

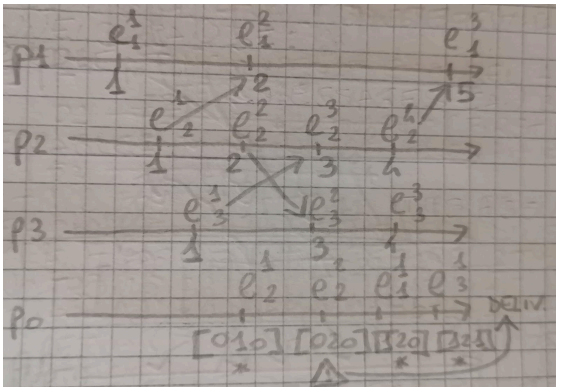
Now let's assume that the processes are monitored by a process p0, now we can use vector clocks to check if some notification will arrive in the future.
EX. [0 0 1] is the first message received by P0 and it is delivered since it's the first message sent by P3 and it is not influenced by other events. [0 0 2] is sent by the same process, the previous message has already arrived and is not influenced by events of other processes so it can be delivered. [0 1 1] is the first message sent by P2, notice that the message is influenced by the first message of P3 but P0 has already received 2 messages from P3 so it can be delivered. [2 2 1] cannot be delivered: P0 has received the first message sent by P2 and the first message sent by P2 but not the two messages sent by P1 so P0 has to wait for them before delivering the message. So wait for [1 0 0] and [2 0 0] to be delivered first.

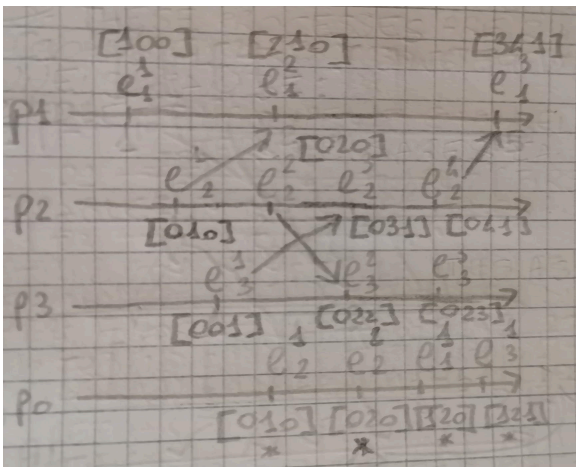


Thus, we can make our **third delivery rule (DR3)**:

$$\text{Deliver message m from } p_j \text{ as soon as } D[j] = TS(m)[j] - 1.$$
$$D[k] \geq TS(m)[k] \quad \forall k \neq j$$

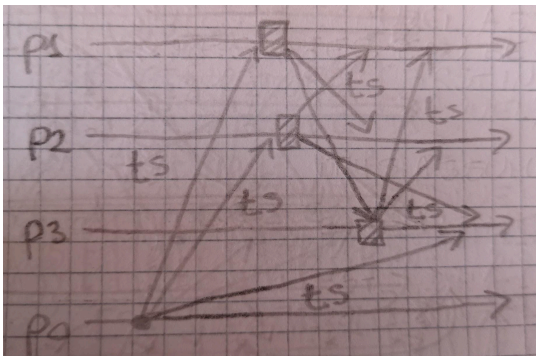
With this type of clock we can also solve limit situations. Let's see an example: P1 can understand that m_2 is caused (or preceded, at least) by m1, this can be done by looking at m2 timestamp. If P1 receives m_2 before m_1, it delays its arrival (as we can see in the screenshot below). This behaviour is called **causal delivery**.





RECAP: Let's imagine that we have n processes that represent a distributed computation. We also have a process P0 called monitor whose goal is to control the state of the system and avoid deadlocks. Only possible solution is to use a vector clock in which to store the last message (actually the Lamport's Clock) of each process.
So, look at the example on the right: Assume that we have a system like that and e^1_2 is the first event to send a notification to P0. The message can be delivered since it has a logical clock of 1 (we can update the vector to notify that P0 has received one event from P2). Then, P0 receives the notification of the event e^2_2. The message can't be delivered because its logical clock is 2 and the other processes have their logical clock equal to 0. This is a problem because it is possible that P0 will receive a notification of message with logical clock 1 in the future. Then, let's assume that e^1_1 arrives and it is delivered. Then e^3_1 arrives and it is delivered again. Notice that now the vector is equal to [1 1 1] so the message that couldn't be delivered, now it can be delivered since all messages with the logical clock equal to 1 have arrived.
This method is not so convenient because P0 would have to wait for the logical clock of other processes to reach the logical clock of the received event (it could wait infinitely, think of the case a process fails). So another solution would be using vector clocks to store the causal history of the events.
So, look at the second image on the right: P0 uses a vector to store which is the last message received from all events. Now, P0 receives e^1_2 and it can delivered, after that it receives e^2_2 and now it can be delivered because, since we have more information, the vector clock tells that we have received the second event from process P2 (it is the next one respect to the one already received) and it also depends on 0 events on process P1 (that's fine because P0 have received 0 events from process P1) and it also depends on 0 events on process P3 (as before). Then P0 receives the notification of e^1_1 and since it depends on 1 event on process P1 and P0 receives 0 events on P1 then it can be delivered.

1.6 Distributed Snapshot

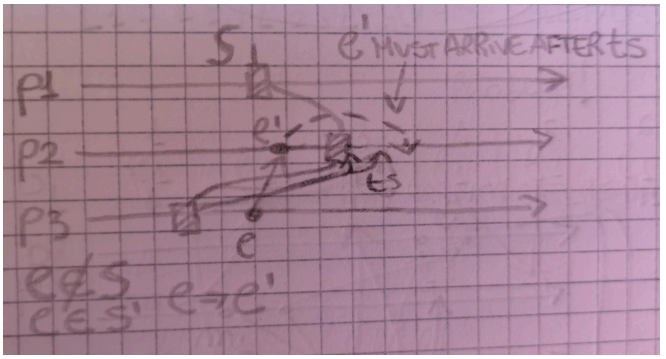


It is a protocol designed by Chandy and Lamport (and also takes the name of **Chandy-Lamport protocol**). This protocol makes use of FIFO channels and assumes there are no failures. The way P0 takes a snapshot is by sending a **TS** (take snapshot) message to every other process. The first time the process receives a TS message, takes a snapshot and resends a TS message to every other process (but P0). This applies to all the processes of the system.

How do we know if the protocol has ended (**protocol termination**)?
It's easy to tell when the protocol started because the processes receive messages. In this case, having n processes, it's clear that every process

will receive n TS messages, thus, we can say that we reach protocol termination when all the processes have received n TS messages.

How do we know that the **snapshot is consistent**?
We assume that this isn't true and that we can have an inconsistent snapshot. We take a cut S that is made from a snapshot and two events e ∉ S and e' ∈ S with e → e'. It is obvious that the cut is inconsistent because e' receives a message from the event e that happens later. This is impossible: we have a TS taken before e (in the image on the right, P3 sends a TS to P1 and P2 and, in the case of P2, it could be the TS that triggered the cut or another one arrived later) and since the channel is FIFO then e' must happen after that TS message (so e' ∉ S!).
To sum up, given the FIFO structure of the channel, the TS message triggered by P3 must arrive before the e' event, but this would mean e' is in the future and it implies that e' ∉ S.



2. Protocols

2.1 Atomic Commits

We have a number of the processes in our distributed systems, p_1, p_2, ..., p_n, we want that all of them are able to know whether a transaction can be done or not. Besides, we want that every process does the same thing: if a transaction can be committed, every process knows this and performs the operation (to maintain every copy of a database consistent, for example). One problem are **faults** such as:

- Crash failures (one process stops working so as all the system);
- Byzantine failure: byzantine processes can do every sort of thing, even malicious actions and thus make the system fail. This kind of failure is harder to cope with than a crash;

An atomic commit means that all processes commit or not commit atomically (all together). The problem of the agreement of these processes in an asynchronous system, even with one crash failure, is impossible to solve (FLP theorem). In broad terms, we can't distinguish a very slow process from a crashed one.
Two important properties of a distributed system are **safety** and **liveness** (does something that doesn't ensure safety). We have to try to at least reduce the problem if we can't solve it.
Let's focus on crash failures. Some properties:
AC1 All processes that reach a decision reach the same one.
AC2 A process cannot reverse its decision after it has reached one.
AC3 The Commit decision can only be reached if all processes voted YES.
AC4 If there are no failures and all processes voted YES, then the decision will be to COMMIT.
AC5 Consider any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

It's important to not implement *trivial protocols*, like the protocol that always aborts (it's crystal clear that it respects every property, but it's useless).

Our aim is to find a proper solution for asynchronous systems but first of all, let's try to see what we can do in the case of a synchronous one (so have a bound of time in which the message must arrive and it is lost then it must be sent again) in the case of:

- No failures: A possible protocol might be making a broadcast so that each process sends its vote to all the other processes. They'll eventually reach a decision since everybody knows the other's decision;
- One failure: This protocol can't work because if a process fails during the broadcast, it may happen that it sends its vote only to some of the nodes (the remaining ones don't receive it). This means that the first processes might commit while the latter might abort since they lack a vote (so AC1 is not satisfied).

A solution could be adding a second broadcast to let processes know each other's decisions: once they know that, they can compute a decision. Notice that a failure can also happen at the second broadcast but we assume that it is failure free;

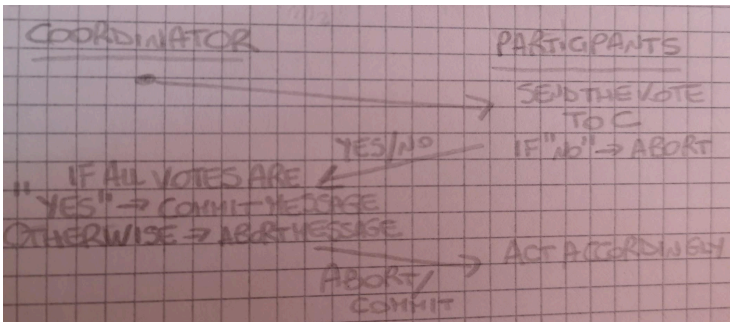
- Two failures: Same process as before but here we have to use 3 broadcasts (always assuming that the last one is failure free).

To sum up, in a synchronous system we can build a protocol that performs f+1 broadcast rounds where f is the number of failures.

If we assume the system is asynchronous then this problem can't be solved because we can't distinguish between failed processes and slow ones. One solution is the **two-phase commit** protocol.

2.2 Two-Phase Commit

Two-phase commit is the simplest atomic commit protocol. It's safe but not always live. In this protocol we have two actors, **coordinator** and **participants**. A coordinator (which can be one of the participants or a different process) sends a **vote request** message and participants send the vote to the coordinator. If all the votes are "yes", the coordinator sends a commit message, otherwise if there is a "no" it sends an abort message. The participants act according to the coordinator's message. Notice that if a process votes "no" during the vote process then it can abort immediately (it is safe to do so) while if it votes "yes" then it has to wait for the other's decisions. Of course, if the coordination decides to abort then the process which voted "no" can ignore the message since it has already aborted before. In case there are **no failures**, **all properties are satisfied**.



Let's introduce failures in this protocol: what happens is that some of the messages don't arrive. There are different cases:

- If one participant doesn't receive the vote request from the coordinator. The coordinator can't wait forever for the vote so we assume that there is a timeout after which the coordinator aborts and everything will be aborted according to the protocol (in this case the system is safe).
- If the coordinator doesn't receive all the votes of the participants, then the decision must be to abort (again it is safe to do so).
- Another problem is that the coordinator doesn't send the last decision to participants and they don't know if they have to commit or to abort (they get stuck in the last phase). If someone voted "no" then it is safe to abort but if everybody voted "yes" and the decision is "commit", if the coordinator sends only a few decisions and then it crashes then some participants could commit and others could abort (so it is not safe to abort). Some solutions are the following:
  - **Cooperative Termination Protocol**: Every participant sends a broadcast message to all the other one to understand what was the decision and to act accordingly.
  - **Recovery Protocol**: The idea behind this protocol is that, assuming that a process crashes, we can fix the problem by restarting it. To be able to do this, we make use of **logs** (saving decisions). If the vote is "yes" the process must log it before it sends it; if the vote is "no" we can do whatever we like (log the "no" and then send the message or vice versa). The important thing the coordinator has to do is to log the commit message before sending it. For the abort message the coordinator can do it either way (just like the "no" message).

We can use the **DTLog** (Distributed Transaction Log) that follows properties stated above. Given a coordinator C and a participant P:

1. When C sends a vote request writes Start2PC on DTLog. This message contains the ids of the participants and can be written before or after sending the vote request. The tricky case is, if you log before sending the request, you log, crash and in the log you can see the ids of the participant so you're

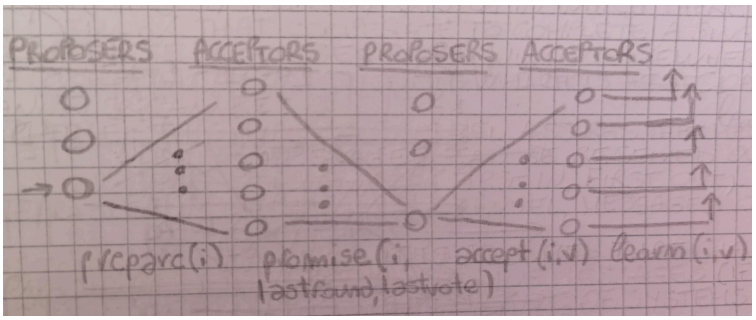
in a safe situation. If you send the message before logging and you crash after sending, you don't see anything in the log and you can just send an abort, so also in this case it is fine.

2. If P votes Yes, it has to write "yes" on DTLog before sending the response to C. Now it's important to log before, because if you log after sending, it might crash between the two operations. When the process restores its status and tries to recover, there's nothing written in the log and it doesn't know if it sent a "yes" or a "no", so it should abort (since nothing is written in the log) but it can't because if it sent a "yes" there might be a commit procedure running (already started by the coordinator C). If P votes "no", it can write the response in the log before or after sending the decision.
3. If C decides to "commit" it has to log it in the DTLog before sending the commit message to P (for the same reason as the second point). If C decides to "abort" then it can log the decision before or after sending it.
4. After receiving the decision, the participants must write it in the DTLog (so they are able to recover C's decision and use it to send a broadcast message if another participant crashes - see Cooperative Termination Protocol).

2.3 Paxos Protocol

Paxos is a protocol designed by Lamport for **distributed consensus**, works in an **asynchronous system** and it can deal with **crash failures**. A fundamental problem in distributed computing is to achieve overall system reliability in the presence of a number of faulty processes. This often requires coordinating processes to reach consensus, or agree on some data value that is needed during computation. The consensus problem requires agreement among a number of processes (or agents) for a single data value. Some of the processes (agents) may fail or be unreliable in other ways, so consensus protocols must be fault tolerant or resilient. The processes must somehow put forth their candidate values, communicate with one another, and **agree on a single consensus value**. With Paxos we *tolerate* process failures caring about *only* "correct" processes (the ones that don't crash). An example of Paxos usage can be found looking at shared files on cloud storage, like Google documents: if a server crashes, we don't perceive any difference since there are multiple copies of our file in other (still working) servers. Of course there is a limit on crash failures Paxos can tolerate: remember that, even with only one process, we can't be safe and live. So, we have to find a **safe** way to tolerate failures that can be as live as possible. Let's assume we have n processes, since processes have to agree on a value, then Paxos can't tolerate more than n/2 failed processes because, in this way, we have a *minority* (processes that stay up) that takes the decision. It's possible, indeed, that the majority picked a decision X and the minority a decision Y . If the majority crashes, making the minority do the decision can lead the system in a wrong state. The answer is that we can deal with f failures with $f = \lfloor (n-1)/2 \rfloor$.
We have three types of nodes:

- Proposers: there must be at least one;
- Acceptors: can be n nodes (also zero nodes);
- Learners: there must be at least one;



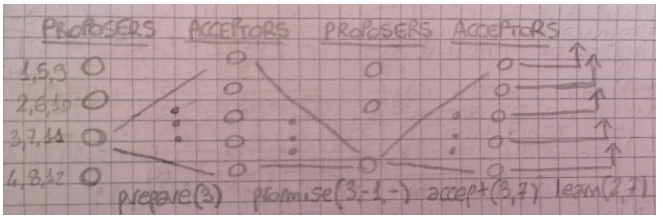
In the real system a node, at the same time, can be all three of them, but it's simpler to divide them. The acceptors are n, proposers and learners are at least 1. The protocol works in rounds and **every round is associated to a single proposer statically** (we know the associations a priori). If there are more than one proposers, we can for example associate one of them to even rounds and the other one to odd rounds (the association must be infinite that is, it is not possible to associate precise rounds to a proposer but there must be a sort of lineary). So in a certain round the proposer has a value and the other processes must agree on it. Any proposer P can start a round by sending a *prepare(i)* message to all the acceptors (where i is a round). All the acceptors reply to the proposer P with a *promise(i, lastround, lastvote)* message where i is the same as before, *lastround* is the last round in which the acceptor voted and *lastvote* is the last vote they gave (at the start of the protocol *lastround* is filled with a value that represents a void response, such as −1). Besides, within the promise message, every acceptor assures that they

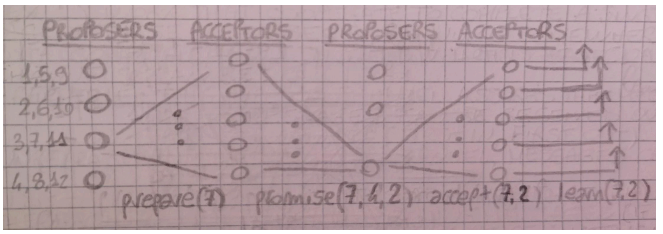
won't participate in rounds smaller than i (otherwise the protocol wouldn't work). When the proposer P gets the majority of votes (not necessarily all of the acceptors' votes), it sends them an *accept(i, v)* message where v is the value associated with the largest *lastround* in the promises. Once it gets the maximum value among *lastround* parameters, it takes the *lastvote v* associated with it (note that if all the promises have lastround = -1, the proposer can send whatever vote it wants).
The acceptors, if they have not promised otherwise, can vote with a learn(i, v) message sended to learners. When the learners get the majority of the same vote, that's the decision (and they log it locally).

EX. Let's see an example where P is a proposer and A the acceptors:

1. P sends prepare(3) (it can use that index since it "owns" the round);
2. All "A" send a promise (3, -1, -) (-1 because it's the first time they votes);
3. If P gets a promise message from majority, P sends accept(3, 7) (notice that the majority is at least 3 messages);
4. All "A" send learn(2, 7);

So P sends a "prepare" message with i = 2. All "A" reply with a "promise" message stating that they have never participated in a previous round (so they neither pass the last vote). Now, P gets the majority of votes so, since there is no last vote, P picks 8 as the value v and it sends the message to all the acceptors. They all send a "learn" message to the learner that they agree with the value 8 (that's the decided value since all (the majority) agree on that).

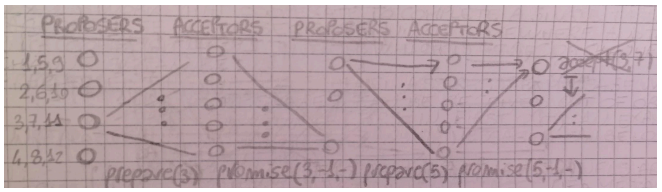




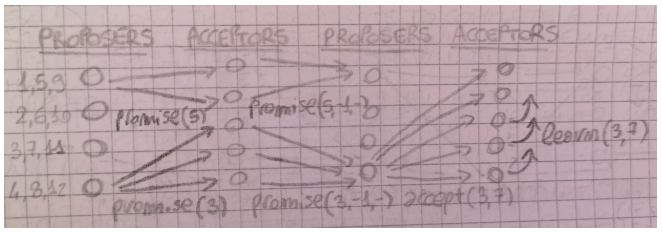
EX. Here the only difference is that we assume that all acceptors voted "2" in the last round "4". So the proposer gets the maximum value among lastround parameters (4) and its associated vote (2) and accepts it.

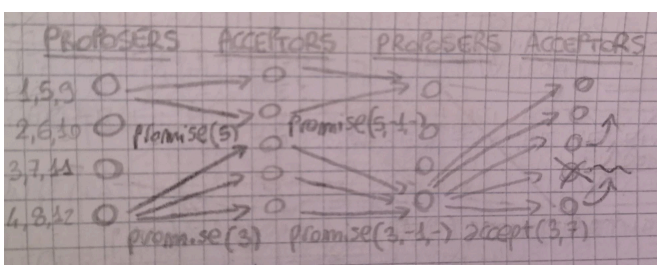
Of course it is not that easy, so let's start making some assumptions:

- Looking at the first example: if another proposer tries to send prepare(2) in the third step the acceptors reject the message because they already promised to participate in round 3 (so they won't participate in smaller rounds);
- Looking at the first example: if another proposer tries to send prepare(5) in the third step the acceptors make a promise(5, -1, -) to the that proposer, then when the one who started round 3 send the accept(3, 7) message it gets discarded since acceptors proposed not to vote for smaller rounds (notice that the second promise message has null lastround since acceptors sent the promise but they didn't vote). So Paxos isn't necessarily live.

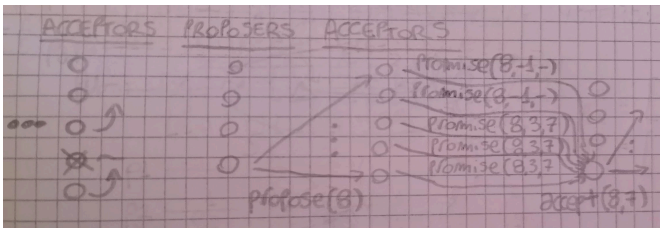


EX. A proposer P_3 sends a prepare(3) message but 2 messages are lost. Another proposer P_1 sends a prepare(5) but 3 messages are lost. The first round has no hope to be completed since it will never reach the majority (there are 5 acceptors, only 2 receive the message and the majority is at least 3) so P_1 won't send the accept message. For what concerns prepare(3), it reaches the acceptors and since their lastround is -1, they send promise(3, -1, -) and send it to P_3 that sends accept(3, 7) to the acceptors that send a learn(3, 7) and the value is logged since it has been decided by the majority.
Now, assume that one the acceptor crashes at the last step: since only two of them send learn(3, 7) then the value is not chosen since P_3 does not have the majority anymore. So





since both rounds 3 and 5 are stuck, assuming that there is a timeout, one proposer can start another round.
Now, let's start another round with propose(8) and get to the promise message. Each process will have a different message depending if they have voted before. Notice that the larger lastround is 8 so the proposer must send an accept(8, 7) (7 because it is the value associated with the lastround).



We have shown that Paxos is not live by giving an example.
Let's now show that **Paxos is safe** by induction.

DEF. If an acceptor a voted value v in round i, then no value $v^i \neq v$ can get a majority in any of the previous rounds j<i.
DIM. We must prove the above statement to say that paxos protocol is safe. We can prove it by induction:

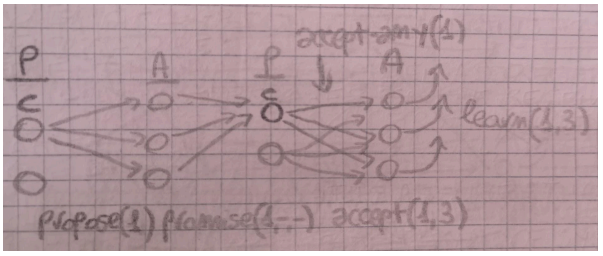
- Base case: let's image we are in round 1, if an acceptor a voted a value v then it is obvious that can't exist another value $v^i \neq v$ that got the majority in previous rounds since there are no previous ones (we are in round 1);
- Inductive case: We take 2 rounds, *lastround* and *round i* where *lastround* < *round i*. In *round i* a majority of acceptors Q (promises), voted for a value v. We are sure that in no round between i and lastround an acceptor a did a vote. This is because lastround was the maximum round found among the acceptors $a \in Q$ promises. It's then clear that, since Q is a majority, in no round between lastround and round i a majority voted for something. Moreover, during lastround the voted value must be v, because it is the value associated with the round. During lastround there's then an acceptor a^i that voted v, and for the same reason we know that no majority that voted v^i can exists before lastround. Accordingly, the property is verified.

2.4 Fast Paxos Protocol

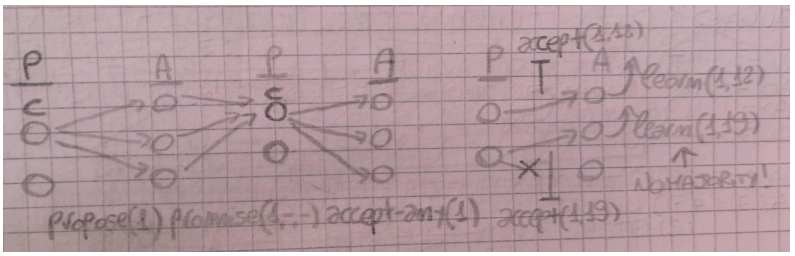
If we are able to elect one and only coordinator (which can be one of the proposers), then Paxos is live, in fact having only one proposer makes us sure that the round can't be messed up. The problem is that electing the coordinator isn't possible in a way that is safe and alive (think of the case we have a loop and the system chooses 2 coordinators for the same round). This is not a big deal because the system remains safe, we simply lose the live property. Choosing a coordinator need a consensus, the problem is called **leader election**.

In real life systems we don't have one value to choose, but a sequence of values. So we don't run a single instance of Paxos, but several of them (for example, the first value chosen is the first edit in a Google doc, the first block in a blockchain, the first transaction in a database, etc..). These instances run concurrently and often proposers are also learners. This enables them to know if one instance of Paxos is finished and the value chosen in it (as it is a learner), making it easier for them (as proposers) to choose a value for a new instance of Paxos. The messages we saw before are tagged also with the Paxos instance id. Actually, the coordinator can start all the X instances of Paxos, with a prepare([1...X], 1) message, removing a lot of messages (**more efficiently**). At the same time, the promise message can change like this: promise([1...X], 1, -1, -1). If these messages work, what's left to do is just accept and learn. Lamport made an assumption about the possibility of sending an **accept-any**([1...X], 1) message to make the third message a single one for X instances. To sum it up, the coordinator sends 3 messages: prepare, promise, accept-any. The proposer (that wants to propose a value) can now send only a simple accept(p, i, v) message, where p is a Paxos instance id, i the round in that Paxos instance and v the value to accept. Now all the acceptors can send their "learn" messages. Thus, we now have 3 single messages (the coordinator's ones) and only 2 multiple messages (the ones of the proposers and the final ones of the acceptors). This version of Paxos is called **Fast Paxos** and it makes sense if you have more than one instance of Paxos.

EX. The coordinator sends a propose(1) message to all the acceptors and they reply back with a promise message. If there is a majority of acceptors which didn't vote, then the coordinator can send an accept-any(1) and, as soon as there is a proposer who wants to propose a value, it sends immediately an accept message

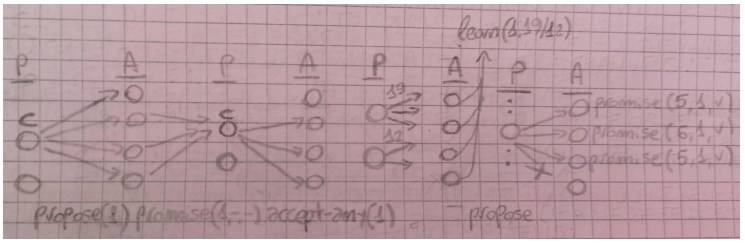


accept(1, 3) (where 3 is the proposed value) to the acceptors, and they can immediately send a learn(1, 3) message.

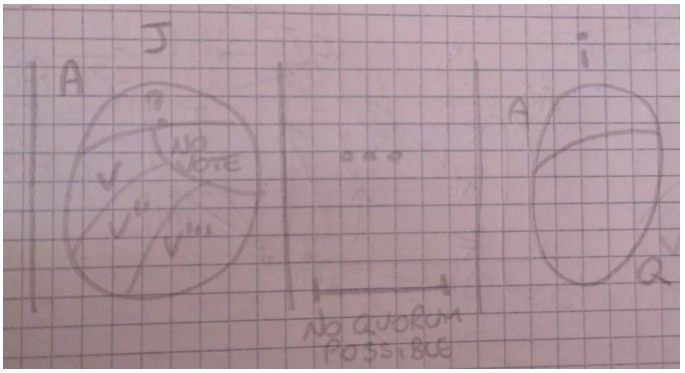


The problem with this version of Paxos is which value to choose. In Paxos, if two acceptors voted in the same lastround we have the same lastvote so a decision is made. With Fast Paxos it is possible to have more than one acceptor in the same lastround with a different lastvote in fact if the maximum lastround = −1 then this means that no one voted, so the coordinator can send accept-any and two proposers, triggered by the accept-any, send an accept message with 2 different values at the same time. Assuming that we have an error in the example above (one of the acceptor crashes) then we can't reach the majority.

So, while in Paxos we need (n/2)+1 processes to reach a decision (so we could tolerate up to f =⌊(n−1)/2⌋ failures), in Fast-Paxos we need a larger number of acceptors to form a majority which is n - f' where f'=⌊(n−1)/3⌋. So (2n/3)+1 acceptors.



EX. Same example as before but we have added another acceptor. Now, assume that we have an initial round in which two proposers send a different value (12 and 19) to different acceptors. We don't manage to get the majority so we have another round in which another proposer sends a propose(5) messages to all the acceptors. Notice that here we may have different cases. We assume in this example that one process crashes. If every process voted "12" before then the chosen value will be "12" (same for "19"). If two processes voted "12" and one voted "19", then "12" since we are sure it has the majority (if the processes that crashed resumes at some point and it voted "12" then it has the majority for sure, if it voted "19" that's still ok because in that case we would choose any value).
If each process has a different lastvote then we choose any value.



We now show that Fast Paxos is safe:
We know that to select a value v we have to take the one associated with the maximum lastround in promises. As seen before, Fast Paxos can have more than one value associated with lastround and thus we can let v be equal to the most frequent lastvalue in j = max(lastround). If j=−1 we don't select any v and we start a fast round (by sending an accept-any message). Let's prove that **Fast Paxos is safe**: We are in round i and we received a majority of votes Q (quorum), but not all of them (set A). Given j=max(lastround) we know that between round i and round j we have no quorum possible, just like Paxos' proof. In round j we have some people in the Q set that have voted for v, some for v', some for v'', ..., and some may haven't voted at all (their lastround was smaller than j). We said that the value voted in the round i is the most frequent in lastround and, since we voted in lastround, we are sure that any other value different from v couldn't reach the quorum. So if v actually reaches a quorum, then it's safe to vote v, but even if v don't make it, we know that every other value couldn't as well so it's still safe. Thus, there was someone in the round j that voted v and, by induction, we can say that was safe to vote v.

3. Consensus

How can we solve a consensus problem knowing the type of system and the type of crashes?

	No faults	Crash	Byzantine
Synchronous	1 round	f+1 rounds	$n \geq 3f + 1$
Asynchronous	1 round	FLP	FLP

If we have **no failures**, it doesn't matter if the system is synchronous or not, we know that every message will arrive at some point and we can safely send every one of them. So one round is enough.
If the system is synchronous and we have to deal with **crashes**, let's figure out what we need. If we know that we will have at most one crash, then we know that if we do 2 rounds, we have for sure one round that is totally correct. So if we have at most f crashes, we need f+1 rounds to make the protocol work. A simple protocol that works in this way is the one that, after the completion of a round, sends to every process a broadcast message to see if they all have the same value, and, if not, restarts the round.
If the system is asynchronous we know for the FLP theorem that we can't solve the consensus problem.
If an asynchronous system has to deal with byzantine behaviours, since this is a harder problem then simple crashes, FLP is still valid. One of the most important protocols that tries to deal with byzantine problems is **PBFT** (Practical Byzantine Fault Tolerance). If a synchronous system has to deal with byzantine behaviours, we can solve the problem. Actually, using PBFT in a synchronous way makes it safe and live. To be accurate, we need at least n rounds with $n \geq 3f + 1$. To prove it let's start with one failure (f = 1).
We have one *General* that has to give a command to two *Lieutenants* L1 and L2, either attack or retreat and the two Lieutenants must do the same thing.
If the General is byzantine he can send two different orders (he says to L1 to attack and to L2 to retreat), so there's no way in which the Lieutenants can know what is right to do. So, the two lieutenants could send a message to each other to tell the other one what's the message sent by the general (so they find out that they have different orders).
If the byzantine node is one Lieutenant, he can send to the other one false information or behave differently to the General command. So it's easy to see that with 3 people, you can't understand what it's the right thing to do (so 3 moves are not enough).

4. Bitcoin

Bitcoin was invented in 2009 by an anonymous group by the name of Satoshi Nakamoto. Some main characteristics are:

- No central authority;
- "Anonymous" (until you remain in the chain, but when you make transactions with this money you're not anonymous anymore);
- Only 21 million bitcoins (18 millions of them are already generated);

Just to make a comparison with normal cash, in that case we have a central authority and we have limited anonymity (if we use credit cards there is no anonymity at all). The actors in this ecosystem are people, authorities and miners, who mint bitcoin and approve transactions.

4.1 Easy Cripto

Before talking about Bitcoin, let's make a summary of some important concepts which are useful to understand how Bitcoin works.

4.1.1 Hash functions

An **hash functions** is a function $h : A \rightarrow B$ such that:

- h is easy to compute;
- A is infinite, B is finite;
- h^{-1} is hard to compute (it is difficult to get A back starting from B where B=h(A));
- Given x and h(x) it's hard to find y such that h(y) = h(x);

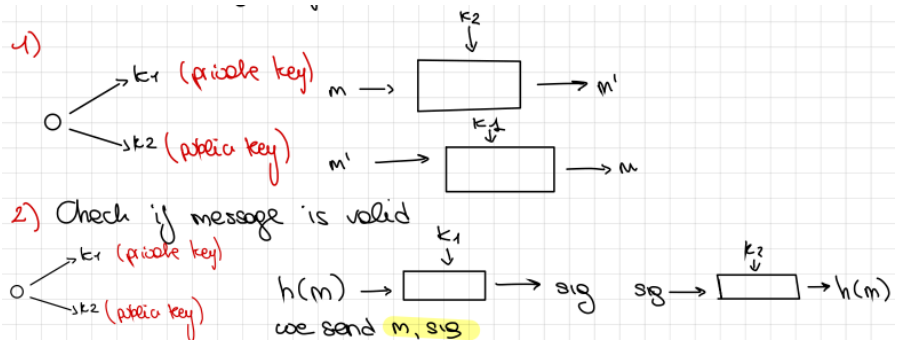
Few examples of hash functions are MD5 (it is broken in fact we can crypt any file with the hash we want), Sha-1, Sha-256 (it is pretty strong), etc..

4.1.2 Public Key Cryptography

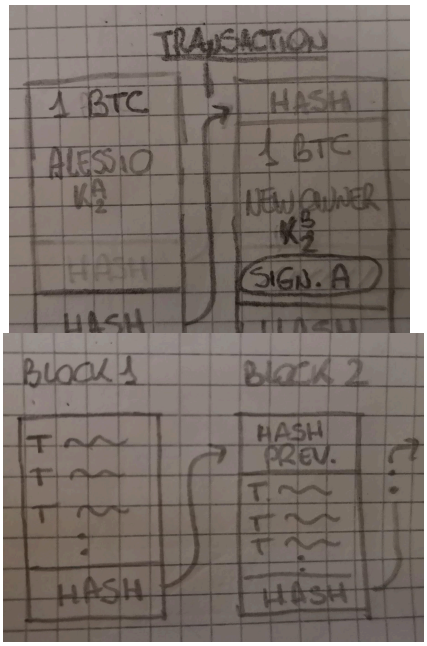
We can have a process that generates two keys k_1 (**private key**) and k_2 (**public key**) that are related in the sense that we can have a message m encrypted with k_1 resulting in m' and this message m', encrypted with k_2, gets us to m again. This means we can use these keys for:

- **Encrypting**: This allows us to send a message encrypted with the k_2 public key and make this message decryptable only by the one who has the private key k_1.
- **Signing messages**: We can use the private key k_1 to sign a message and everyone can decrypt it with the public key k_2 and be sure that the message could only be written by the k_1 owner.

Note: Typically signing is not done like this because this protocol it's actually not so lightweight. It is better to calculate the hash of the message we want to sign (using a particular hash function). We can encrypt the hash (which is smaller than the message) with k_1 and send the hash and the signature to the receiver so that he can decrypt the hash with the public key k_2 and see if it coincides with the received value.



4.2 Bitcoin and Consensus



A bitcoin is like a file in which there is written information like the **owner of a bitcoin** A is represented by a public key k_2^A. In order to **transfer bitcoins**, the owner must write on the file the new owner B and a signature A done with its private key k_1^A. In some way we are creating a new bitcoin file with the number of bitcoins and the public key k_2^B of B. So, if B wants to demonstrate he has received the money from A legally, it can use the public key k_2^A and get the file back again. This is what we call a **transaction**. Notice that if the owner loses his private key (he forgets it or someone steals it) then he can't access the coins anymore. The problem is that it is possible to take one bitcoin and make multiple copies of it and sign it to several people. This is called **double spending** and to avoid this we must have knowledge on which transaction has to be done and which not. This is done without a central authority and thus becomes a **consensus problem**.

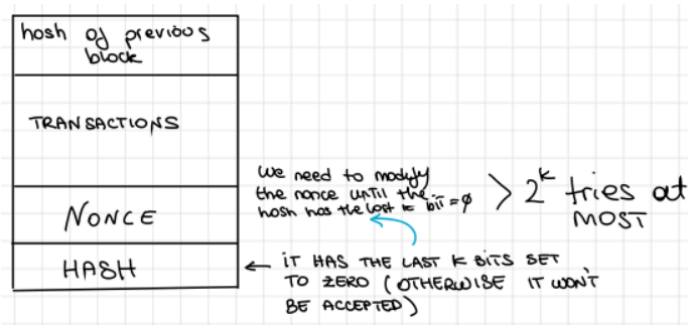
Every node in the system must have the same knowledge on what was approved and not approved. To achieve this we usually start a consensus protocol on a bunch of transactions (with fixed size), that is called a **block**. The result of the protocol is that everybody agrees that the transactions in the block were approved. The next block starts with the hash of the previous one, to link them, and this is why we talk about **blockchain**.
If a person tries to do double spending, the receiver can check in the blockchain if a transaction with the same coin exists (it should be done before spending the coin so we should wait for the transaction to be in the blockchain so that we are sure that we are not being scammed).
The key point here is: <u>who decides the next block?</u>
We can indeed see that we have to deal with *byzantine behaviours*, because there are actors in the system that can go against the rules. For this reason, protocols like Paxos are not enough (and also because we have broadcast messages). Also 2PC is not enough because it doesn't tolerate byzantine faults and crashes (if a node crashes, 2PC stops working).

4.2.1 Blocks and Miners

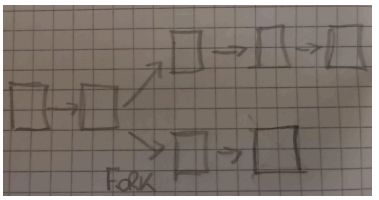
A block stores some transactions, the hash of the previous block, its hash and then reserves some space to **nonce** (which is an arbitrary number). The **hash of the block must have the last k bits set to zero** to be valid (requested by the protocol), but we have no guarantee that it'll be like this. Thus, the miner can change the nonce, redo the hash and hope that it'll match the property. The k number changes in such a way that the time to generate a new valid block in the world is 10 minutes. For example, with k = 32, on average the server has to make 2^{32} (almost 4 billions) hashes and actually k > 32. Once the right nonce is found, the block is sent in broadcast to all the servers and everybody accepts the new block in the chain: this trick is called **proof of work**. Another proof we'll see with Algorand is the **proof of stake** that works like a cryptographic protocol: a very small subset of miners is chosen to form a committee. A serial number is chosen probabilistically and the owner of that coin will be on the committee. For this reason, the probability of getting selected in the committee depends on the coin owned. This is a much faster protocol because it is easy to choose a committee, and once it is chosen then a new block gets accepted in a few seconds.

Why do we have to do this? At the time the price for finding a "good" block was 50 BTC and it's halved every 4 years (now it's 6.25 BTC, still a lot!).

The blockchain grows as miners find new blocks.
What happens when two miners find a good block at the same time?
Both miners send a message to everybody and maybe one block arrives to some miners and the other block to the other part. So we have created two branches (**fork**) of the blockchain that two parts of the miners are trying to extend at the same time.
What usually happens is that only one block "wins" and everybody stops working on the other branch, making it a dead branch and if a transaction is on a dead branch, it won't be approved.



How can we be sure that our transaction is on a good branch?
There's a "rule" that says that we have to wait at least 6 blocks after ours to be sure that our transaction will be approved. This is a probabilistic matter, it is really difficult that a 6 block long branch becomes dead.
Why is it so hard to find a new block?
If it were easy, a miner can extend the blockchain from wherever he wants and make this new branch grow so fast that it makes the other branches dead. The larger is k, the more difficult to find a valid block. Notice that usually k is changed after a certain amount of time (let's say 2 weeks).



This protocol allows us to choose the next block in the blockchain but **it is not safe** because there is a chance of a fork which is forever (if we are unlucky). At the same time, we can't be sure that this protocol is live, but it's very unlikely to have multiple forks that stops the protocol. So when we have a fork, one of the two branches must win, probabilistically speaking.

4.3 Frauds in the Cryptocurrency Ecosystem

To introduce this topic we can make an example of fraud with **Pump and Dump**. It's a market manipulation that works by pumping artificially the price of something, selling and then restoring the price to its original value: this action is illegal. This happens regularly with cryptocurrency mostly because the market is not regulated and also many crypto-coins are thinly traded so it's pretty easy to change the price.
There are groups that organise this scam and hence we have some actors involved: Administrators (always win, they know the coin and they already have it),

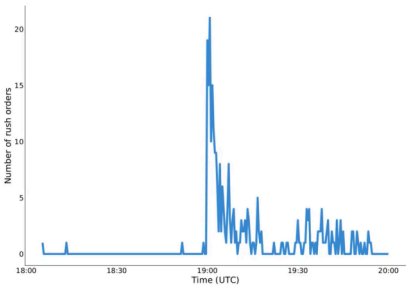
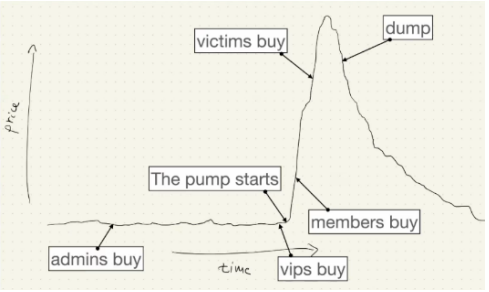


Figure 18: Abnormal amount of market orders.

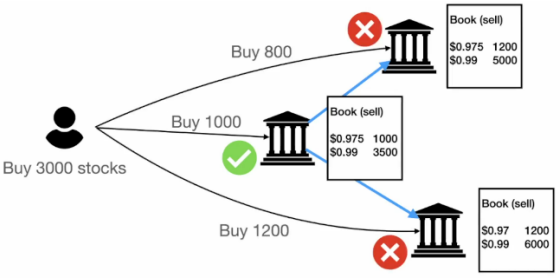
VIP group members (most win), group members (some win, some lose) and external investors (always lose, they don't know that a pump and dump will happen). One practical example regards RCA, an agency which sold gramophones back at the beginning of the 20th century: the price went from 10$ to the incredible price of $549 gaining a lot of shares, but a few weeks later the price sank again to $10.



The same idea can be applied to Cryptocurrency. There are a lot of famous groups that perform cryptocurrency frauds like Big Pump Signal, Trading Crypto Guide and Crypto Coin B. These groups organise to do the fraud on a certain time and on a certain coin. As said before, the VIP members know it some time before so that they can put market order when the coin value is already low, the members know it some seconds later so they may still be fast enough to do the same. The fraud works when external investors start buying the coin at a higher price: they will be the one to lose money. When the coin has reached a certain market value, the scammers start selling the coin making money.
Can we make a system that detects these scams? The idea is that we can observe that during a pump, most of the members of the group put market orders. There are some classifiers that use ML that allow you to detect frauds within a few seconds.

What are smart contracts? In some of the blocks in the blockchain, it is possible to put some code that works like a contract. This contract is given to all the miners, they start a consensus protocol and eventually a new contract is added to the blockchain. A client could call one of the methods in the contract and this call goes to miners that take the relative contract from the blockchain. They execute it and the new state is added to the blockchain. This is a computation that everybody can check and trust. Of course, since we have codes, we may have problems with bugs. One example of blockchain that uses this method is Ethereum.

Another type of fraud is **Frontrunning 2.0**: if someone is buying a large amount of coins on different servers and, depending on how the system works and how scammers are fast, they could notice the starting transactions (one goes through so they know the legit buyers started them) so they put market orders before the legit buyers so that, the latter eventually buy them with a larger price. Finally, scammers sell the coins again making money. This scam is also called the **Sandwich attack**. A way to prevent it is to build a system that sends the transactions so that they arrive at the servers at the same time (e.g. messages are sent in a certain order and each of them is sent after a certain delay).



5. Privacy

When we talk about internet privacy, one of the first examples that comes to mind is incognito/private browsing. This is a modality offered by browsers to assure local privacy (e.g. searches aren't saved in the history). The problem is that this doesn't allow anonymity towards the ISP and the government. The type of privacy we want to analyse is instead a **global privacy**, which means anonymity from internet providers, government and so on.

5.1 VPN

VPNs weren't build with privacy in mind, but to virtualize a local network to a not local one meaning we can be a node of a network even if you aren't connecting from there (e.g. a VPN can enable us to be a node of Sapienza even connecting from home). If we want to connect to a website through a VPN we build a connection to a remote server and then from this we connect to the website, hiding our IP. This means that the node in the middle knows everything about us so we have to trust it.

5.2 TOR Anonymity System

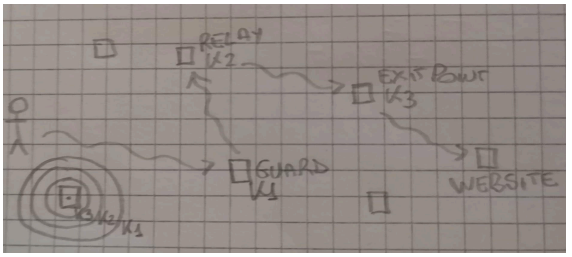
In the TOR (the Onion Router) system there are a number of servers in the world named **TOR relays** (about 7 thousands). If a person wants to connect to a website, he first connects to a relay called **Guard**, there are some relays in the middle and then an **Exit point**. To navigate in TOR we have to connect to 3 relays at least (in the real protocol they are usually 5) and every relay can be located in different parts of the world. The idea is:

- The guard knows the user but has no idea of the website he wants to reach;
- The relays in the middle know nothing;
- The exit point knows the website but not the user;

How can we do this?

Let's assume we have n nodes: We have n keys with n ≥ 3 and we encrypt our request first with k_n, then with k_{n-1} down to k_1, building a layered structure that resembles a half sliced onion. In this way the Guard will decrypt our packet using k_1, the middle relays using k_2, ..., k_{n-1} and the Exit point using k_n.

How can we build this type of connection?



Firstly, we get the list of TOR servers (we get it from every relay) and select at least 3 servers to build the connection. We want to make sure to be as anonymous as possible, so we can't share our public key (if we have one) to the Guard, instead we use a protocol that enables the guard and the user to make a key (a private one). This type of protocol makes sure of 2 things:

- After finishing, only us and the guard know the key;
- Even if there's a malicious process trying to see the "conversation" between the guard and us, the process can't get any knowledge from the stolen data;

One of the most famous protocols of this type is called **Diffie-Hellman** that works this way:
There are two people, g_1 and g_2, that want to share a secret so they need a key. Both of them think of a secret number, "a" and "b" respectively while g is known to everyone. g_1 sends to g_2 "g^a" and g_2 sends to g_1 "g^b". So g_1 can compute $(g^b)^a$ and g_2 can compute $(g^a)^b$ and it's easy to see that $(g^b)^a = (g^a)^b = g^{ab}$ thanks to the commutative property. So the chosen key will be $k = g^{ab}$ and it's known by both people and it will be used to encrypt messages. We can safely assume that it is extremely hard to reconstruct the number "a" starting from g^a (the same as "b" starting from g^b) due to the complexity that lays behind the g operation (it isn't possible to compute the discrete logarithm).

In TOR, since we have both private and public key, to choose the key between user and guard we don't exactly use Diffie-Hellman but an extension called **Authenticated Diffie-Hellman** that works this way: The user knows the public key of the guard. So the user can send an encrypted message with this key $E_{pk}(g^a)$ where E_{pk} is the public key of the guard. The guard decrypts it with its private key (so it gets to know g^a) and answers with g^b, $H(g^{ab})|$"Handshake" ($H(g^{ab})$ is a hash function of g^{ab}). When the user gets this message he can compute g^{ab} and hashing it he can check that the protocol worked correctly (so the $H(g^{ab})$ computed locally is the same as the one sent by the guard). g^{ab} will be the key used by the guard and the user to communicate.
This version is called authenticated because in this way the user can be sure that he's communicating with the guard thanks to the fact the hashes are the same. In the plain version we are vulnerable to **Man in the Middle attacks**.

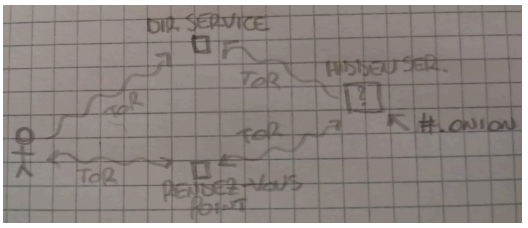
So the user establishes a secret key using Authenticated Diffie-Hellman and uses it to encrypt messages in the TOR circuit. Notice that in order to communicate with the different relays, the user has to pass through the previous one (e.g. to communicate with the second relay in the circuit, the user has to establish a connection through the guard).

5.1.1 Characteristics of TOR

Privacy
If a user enters the TOR environment, he can be selected as one of the relays. This means that, sometimes, he can be the Guard or the Exit point. Thus, we can know who is establishing a connection or where a connection is headed. This means that someone could do research to know, statistically, why TOR is used and who uses TOR.

TOR can also be used to **hide services**. We have seen an example where the user is hidden, but TOR can **assure privacy also to websites**. The idea is that we have two actors, a user and a **hidden service**. If the user wants to connect to a hidden service, first he has to connect to TOR relay called **directory service** using TOR (so the user establishes a connection through a series of relays as stated above) and the hidden service does the same. The result is that the directory service does not know where we are and who we are and does not know where the website is. The directory service now selects another relay called **rendez-vous point** that the user and the hidden service use to communicate, using TOR, with the rendez-vous point (used like a proxy). The directory service changes everyday. How can we address a hidden service? In the normal way, but with a *.onion* domain, only solvable by communicating with the directory service (DSN is not used). Websites that use this domain form the **dark web**.

Performance
Two of the most important metrics in internet navigation are latency and bandwidth. It's obvious that using this circuit makes the user lose some performance and this happens especially in latency because the packet has to pass through different relays that are possibly in different parts of the world. Generally, if all the relays have a good connection, the user won't lose bandwidth (but it's unlikely).



Attacks
If someone is the owner of both Guard and Exit point he can make a lot of attacks.
The most common is **traffic analysis**. You can check patterns like seeing a packet arrive at one of your nodes and after a second arrive at another one. If this happens frequently you can understand who is the user and which is the website. Obviously it is hard to have both the starting and end point of the system. There are systems that assure that Guard and Exit points are in two different parts of the world (not 100% safe, but most likely).
Another attack can be done by the website. It can **de-anonymize information** looking at the way the browser uses it to send messages. The browser, when connected to a website, sends a lot of information (OS, Browser version, size of display, etc..) and if you collect all this information you can identify a unique character among 200 thousand people. This means that a website can find two browsers interacting in the same way, they are most likely from the same person. This doesn't break anonymity, but **pseudo anonymity** (the website doesn't know you, but knows you are the same guy). To avoid this attack a user can use a browser that always sends the same information, and it is the **TOR browser**, based on Firefox.

6. Privacy in the Internet

When connecting to a Wi-Fi network, the classical protocol used by the phones was to register the mac address of each router they connected in the past. So, every time the user turns the Wi-Fi interface on, the phone listens to the channel and if there is one known router in the list then it tries to establish a connection. If the router replies then the connection is established otherwise the operation is repeated after a few seconds. This protocol is quite energy efficient because the phone only tries to establish a connection with a certain access point rather than sending messages to all the access points. The problem is that there were some issues regarding privacy: the list of known access points was sent in clear. Someone could sniff the list of access points and get to know information about the user just using a "TCP dump". He could get to know the user's visited place, where he was staying during a holiday, where he works and even where he lives. Also, statistics can be computed: it is possible

to understand if an access point is public or private (depending on the number of people which connected to it), understand if a person is someway related to another (by looking at the the common access points), understand if people belong to some organisation and even predict which political party will get the higher number of votes depending on various information extract the list of access points (e.g. where people in different cities come from).
Another way to sniff information on the internet is by using the browser. Each browser contains information and, if we take two different browsers, the probability they share the same fingerprint is really low making each configuration easily recognizable. Also notice that if one tries to set very secure and privacy-savvy settings, then it is easier to track the user. The tracked information is usually the version of the browser, the language, the font, the layout and so on and so forth. One possible way to hide information on the web may be using systems like TOR. It is also possible to find information on a user by looking at their profiles on social networks.

7. Failure detectors

A **failure detector** is a software structure that we can use to understand when another process is crashed. A process can use a detector to ask about the state of another process (e.g if some process crashed). Assume we have a process p and its detector D_P, then $q \in D_P(\sigma, t)$ which means that process q is believed to be crashed at time t at run σ by failure detector D_P (in this case p asks D_P if q is crashed). The D_P module is called **oracle**. A process can asks the oracle:

- What processes crashed at time t in run σ, Crashed(σ, t);
- What processes is up at time t in run σ, Up(σ, t);

There are some properties:

- **Completeness**: A failure detector is complete if when another process crashes, the FD realises it. To be only complete is pretty easy, if a FD always says that a process is crashed, is complete. There are 2 forms of completeness:
  - **Strong completeness**: $\forall\sigma\forall p \in Crashed(\sigma)$ and $\forall q \in Up(\sigma)$ then $\exists t\ \forall t^i > t \mid p \in D_q(\sigma, t^i)$.
    A down-to-earth explanation is if a process crash, the other processes knows it after some time.

  It means that for every possible run, and for every possible process P which has crashed and for every process q that is up, then there is a time after which the failure detector of q says that p crashed in run σ at some time t' (it will eventually say it at some time t', not necessarily immediately).

  - **Weak completeness**: $\forall\sigma\forall p \in Crashed(\sigma)\ \exists q \in Up(\sigma) \wedge \exists t\ \forall t^i > t \mid p \in D_q(\sigma, t^i)$.
    This means that not necessarily everybody realizes the crash of a process, but at least one.

  Same explanation but not necessarily all processes q (that are up) know about p (which is crashed).
- **Accuracy**: If a failure detector says that a process p has crashed, it should be right. To be only accurate is pretty easy, if a failure detector always says that a process is up, is accurate. There are few forms of accuracy:
  - **Strong accuracy**: $\forall\sigma\ \forall t\ \forall p, q \in Up(\sigma, t)$ then $p \notin D_q(\sigma, t)$.

  For every possible run for every time t, if we have 2 processes p,q that are up then the failure detector must say they are up.

  - **Weak accuracy**: $\forall\sigma\exists p \in Up(\sigma) \mid \forall t\forall q \in Up(\sigma, t)$ then $p \notin D_q(\sigma, t)$.

  For every possible run, there exists a process p that is up and for every possible process q at every time t which are up then p is not suspected by q to be crashed.

  - **Eventual strong accuracy**: $\forall\sigma\exists t \mid \forall t^i > t\ \forall p, q \in Up(\sigma)\ p \notin D_q(\sigma, t^i)$.

  For every possible run, there is a time t such that for every time t' > t and for every pair of processes p, q which are up then p is not suspected by q to be crashed (so it is like strong consistency but the property is not valid for all t but only for some t).

  - **Eventual weak accuracy**: $\forall\sigma\exists t \mid \forall t^i > t\ \exists p \in Up(\sigma)\forall q \in Up(\sigma, t^i)\ p \notin D_q(\sigma, t^i)$.

For every possible run, there is a time t such that for every time t' > t, for every processes q which are up and there exists a process p which is also up then p is not suspected by q to be crashed.

We have a **taxonomy**:

	S	W	Event. S	Event. W
S	P (perfect)	S (strong)	◇P	◇S
W	Θ	W (weak)	◇Θ	◇W

We can consider 8 different types of failure detectors depending on the combination of accuracy and completeness: P is the set of perfect FDs, ◇P (diamond P) the set of eventually perfect FDs, ◇S are eventually strong, W are the weak ones, ◇W are eventually weak and so on and so forth..

A FD which is both complete and accurate is difficult to achieve but with strong accuracy and strong completeness we have a perfect FD.
If we assume the **system is synchronous**, we could make **Paxos live with a perfect failure detector**. In order to do so we could elect a leader, so a single proposer that is correct. A simple way of electing a leader may be choosing the proposer p_1 with the lower id. If it crashes for some reason we may choose the second proposer p_2 and so on. The problem with this approach is that p_2 may think the first process crashed (even if it is up) so it may take the leadership and this would lead to having 2 leaders which is wrong. Assuming to have a perfect failure detector (so it has both strong completeness and, most of all, strong accuracy) we could use it to be sure that the process crashed: if it says the first process crashed that the second process can take the lead. Notice that, for the strong consistency property, p_2 get to know that p_1 crashed only at time t'>t so it may happen that for a certain amount of time there is no leader, but after time t' we get a leader (p_2 gets the lead) and it is impossible that there are 2 leaders.
If the system is **asynchronous** a perfect failure detector that works this way can't be made (in fact remember that for the FLP theorem, Paxos can't be both safe and live in asynchronous systems and with a perfect failure detector we would make it live, and we know it is safe by induction).

If we are asked to build an algorithm that controls a failure detector D_P. First of all let's assume we are in a **synchronous system** so we know messages arrive within a time Δt. It should implement these features:

1. Heartbeat: p sends a message (**ping**) to every process q;
2. If q doesn't reply within 2Δt and the timeout goes off, then $q \in D_P$;
3. Repeat every 2Δt;

So, we know the delay (the time needed for a message to go from a point A to a point B) so we are in a synchronous system. Note that the heartbeat process is **strongly complete** because if a process q crashes it doesn't reply to the ping, timeout goes off and then the process q is listed to the crashed processes. As it concerns accuracy, We have two cases:

- The timeout is greater (or equal) than the maximum delay Δ. In this case we have **strong accuracy** because:
    - When the system starts then everybody thinks the other processes are up (we have no information);
    - After the timeout (remember that the timeout period is greater than maximum delay), we actually have information on processes in fact if q doesn't reply to the ping then the process is suspected to be crashed;
- The timeout is smaller than the maximum delay Δ. In this case we could have false positives meaning that the reply from the process q doesn't arrive within the timeout (so it is put in D_P) but it arrives later. This means there is **no strong accuracy** since if the heartbeat is very slow then the other processes may think the process is dead while it is alive (same with weak accuracy). So we **eventual strong accuracy** because after some time t we will get the right timeout (every time we have a false positive, we can, for example, double the timeout so that we can eventually get the right timeout).

So again, if the system is synchronous (we know the delay) we can make a failure detector in P (so we can actually make Paxos alive as said before).

Now, let's assume we are in an asynchronous system so we don't know the delay. The system is still **strong complete** (for the same reason as before, if a process crashes then the other processes will know at a certain time independently on the timeout). We can't get strong accuracy because we don't know the delay and we basically fall into the case in which the timeout could be smaller than the maximum delay. What about the eventual strong accuracy? In this case **we don't have eventual strong accuracy** because it is not guaranteed that from a certain time t the system will be accurate. So, in an asynchronous system we can't have any form of accuracy.

Can we simulate a perfect FD with a weak one?
Yes. We can come up with a protocol that enables the process that knows who crashed (and it exists by definition of weak FDs) to broadcast a message with the crashes to all the other processes that are up.
Actually we can simulate every "strong" class with its equivalent "weak" class.

8. Peer-to-Peer Systems

Peer-to-Peer systems were common almost 20 years ago. They have no central authority and are made on a global scale. These systems are also very dynamic, people can connect and disconnect at any moment with their clients. They're not so efficient, so they are now outdated. Some examples of P2P systems are BitTorrent, Napster and Emule.

8.1. File distribution in P2P

Let's assume we want to distribute a file to millions of users stored in a server: having them connecting to the same server creates a single point of failure and a huge bottleneck problem. The solution that P2P systems propose is based on the idea that every person downloads a chunk of the file from the server and then starts downloading the other pieces from the other users (**peers**).
There are few actors in the system, one of them is the **tracker**. It has a list of the people that share a specific file. Each file to share has another associated file whose extension is *.torrent* and contains information such as name, size of the file and the chunks, tracker info (like its url). The file is typically divided into parts called blocks, so we also have their sizes and their hashes. We need hashes to make sure that we are downloading the correct part of the file and also to avoid an attack called **poisoning** (we have many examples of poisoning such as downloading a fake file even if we were promised another one).
Another actor is the **seeder** which is a node that has all the parts of the file. If a user wants to download a file, he asks the tracker to give him a node where to download the file. Initially the tracker addresses the seed, then starts to keep track of which part is in what node and redirects users to other peers. If a user holds every part of the file, his node becomes a seed. The other users are called **leechers** (the ones who are still downloading the file). In the system there are actors called **free riders** which are users that download but don't upload.

What part of the file should a user download first?
We have three possibilities:

- In order: many problems, we can't use it;
- **Rarest first** (the least distributed): This way we increase the probability that the rarest blocks are available at least at one node so that if a node which has that certain block stops uploading, there is another one that might have the block we are looking for. The problem is the overhead for the few users that have that part;
- Random;

Actually we combine rarest and random. Initially we use random and then rarest (it is done like that because random works pretty well with big numbers, then we use rarest to make sure that none of the parts of the file will be lost). It's good enough to make the first distribution randomly and then use rarest.

One problem is the one in which the user can't download some last blocks because those blocks have a low availability or maybe because the users that have it are really slow. There is a modality called "**endgame mode**" to avoid this problem. In this case the last slit of file is split again and the user can download every little piece from every peer in parallel.

How do a user select peers to share the file to?
The first approach is **random**: simply select randomly the node to which upload to. In BitTorrent there's a rule of **unchoking** (to permit the upload to another node), every peer decides to unchoke other 4 peers. This is done in order to avoid connection saturation and avoid using the bandwidth only for BitTorrent. A user selects the 4 nodes to unchoke from which he has a high probability to download. If a user is a free rider, there is a very low probability that someone unchokes him (decides to upload to him). There's a problem because if a user has just started it's obvious he has a low probability to be chosen. There are better algorithms, like **optimistic unchoking,** where there is a higher probability to make someone start downloading (even if he has nothing or he is not uploading chunks).

A way to disrupt a system like this might be to attack the trackers to make the system go down (noticing that it is a single point of failure). Another way might be tracking users who are trying to download a file and discover the identity of the user (e.g. his email address). For what concerns the files, someone could also poison the system by putting fake blocks (even if it is difficult to do due to the hash of the blocks). It is also possible to poison the whole file meaning that another content is downloading with respect to the one we were promised.

9. Algorand

Algorand is a blockchain with a currency called Algo. Algo is in the top 20 currency for market capitalization. This is an economic value computed in this way: numbers of the coin in the chain multiplied by their value. The max supply of Algo is 10 billion and there is about 6,3 billion circulating supply. Algorand was founded by Silvio Micali.

Algorand is a **programmable blockchain**. This means that it supports smart contracts, chunks of code that can be deployed on the chain and executed. The Algorand language is TEAL, wrapped also in Python with pyTEAL. Algorand is a **proof of stake blockchain** with cheap transaction fees. The proof of stake model enables the blockchain to be very fast (from seconds to less than a minute per block, remember Bitcoin takes about 10 minutes). Besides, in Algorand, if everything goes well, the chain can't have any fork and this reduces the problem of double spending.

Just a quick recap of what is a **proof of work** chain: To add a new block a user has to solve a puzzle game, this generates a system with slow transactions (∼10 min) and high energy usage, resulting in a need of expensive equipment.
For what concerns the **proof of stack**, **more money a user has in his wallet and more chance its node has to propose a new block**. So for example, assuming to have a total of 100 coins in the system, the probability of being picked is given by the ratio of the number of money owned by the user and the number of money in the system. Actually, in the real system we have some **validators** that collect a number of nodes so the money associated with the validator will be the sum of money of each node in it, and once a validator is chosen it will be the one to propose a new block. This results in a faster system, no need for expensive equipment. Besides, a system like this enables **stacking** that is using detained cryptocurrency to earn rewards.

As in Bitcoin we have a network of nodes that communicate in a P2P way. Every time that a relay node has a message, it shares it with every other node in broadcast. The P2P protocol used is called **Gossip**. The only time a node has to broadcast a message is when it sees it for the first time. The Algorand network is composed of two distinct types of nodes, **relay nodes**, and **non-relay nodes**. Relay nodes are primarily used for communication routing to a set of connected non-relay nodes. Relay nodes communicate with other relay nodes and route blocks to all connected non-relay nodes. Non-relay nodes only connect to relay nodes and can also participate in consensus. Non-relay nodes may connect to several relay nodes but never connect to another non-relay node. In addition to the two node types, nodes can be configured to be **archival**. Archival nodes store the entire ledger, as opposed to the last 1000 blocks for non-archival blocks. Relay nodes are necessarily archival. Non-relay archival nodes are often used to feed an indexer that allows more advanced queries on the history of the blockchain. Finally, a node may either **participate in consensus or not**. **Participation nodes do not need to be archival**. In addition, to reduce attack surfaces and outage risks, it is strongly recommended that participation nodes are only used for the purpose of participating in consensus. In particular, participation nodes should not be relay nodes.

There are 2 steps in the consensus:

1. **Proposal**: One token is randomly selected and its corresponding public key becomes known to all. The uses associated with the token can propose, sign and propagate the new block to all the other nodes;
2. **Agreement**: Since we could have more than one winner, a subset of nodes called "committee" (usually 1000 nodes) is chosen and their public keys become known to all. They must agree on the value of the consensus. Once they reach an agreement they sign the proposed block. Sufficiently signed blocks (about ⅔ of nodes that agree on it) are valid and propagated. Notice that this is a **Byzantine Agreement** meaning that **consensus is reached in presence of malicious actors**. Algorand's Byzantine Agreement is uniquely efficient thanks to the use of the **Verifiable Random Functions** (**VRF**) which is used to perform secret cryptographic sortition to select committees to run the consensus protocol.

We have also three type of nodes:

- **Observer**: Do nothing in the network and just listen;
- **Proposer**: The user's node that proposes a new block so the cast a vote;
- **Committee**: A node receives the votes and agrees on one of the blocks proposed;

Every node has a secret and a public key. At the beginning of the round no one knows who is who, but the richest nodes are most likely to be chosen to be proposers or a committee member. So we have the **sortition** between nodes in observers, proposers and committee members. Then a **proposer casts a vote** on a block x. The committee receives the votes and **agrees on two values**: the block x (just one block among the votes proposed) or the empty block. Finally consensus is reached on a block x. The block can be a **final block** (if it's the chosen block) or a **tentative block** (otherwise).

Before we said that to select the entities we use the **Verifiable Random Function** that allows us to solve some problems. The entities have to be selected randomly in order to avoid possible attackers to deliberately be, for example, one of the committee. The idea behind the VRF is that we have the need to generate a random number and we have to prove that the number is generated by the correct user and it's actually random. The function takes in input a seed and a role and gives in output a random hash number and the proof.

$$VRF_{pk}(seed||role) \rightarrow (Hash, proof)$$

$$VerifyVRF_{pk}(Hash, proof, seed||role) \rightarrow True\ or\ False$$

The computed proof is needed to prove that the hash is true. Everyone that has the hash, the proof, the seed (that is a shared parameter of the network) and the role of the user that computed the function, can check if the hash is true or false (so if the hash has been computed by the right user).

10. Content Delivery Network

It's an area of research about all the techniques that are used to get to the user content as fast as possible. One of the most famous CDN is Akamai which was developed by researchers from MIT.

10.1. DNS

DNS is a distributed system to translate addresses from a written form (e.g. www.cnn.it) to a numeric one (e.g. an IP address). There is a central server, called **DNS root**, which is responsible for storing the primary domains (.com, .gov, .it, etc..). These domains have their servers that address the other domains below them (for example .it is responsible for addressing the server uniroma1.it).
A user has to set his **DNS server** and use it to ask for address mapping. The server sends a query with the address to the DNS root: it responds with the server we're interested in. Then we make another query to that server and so on until we find the address of the wanted server (for example, for di.uniroma1.it, we first ask the DSN root that replies with the ".it" server. After that we ask the ".it" server that replies with the "uniroma1.it" server. Again, we ask the "uniroma1.it" that replies with "di.uniroma1.it". Finally we can send a query to the "di.uniroma1.it" and it will reply again with the IP address of the server). Usually all of this information is **cached**, so we don't have a lot of messages. Caching **raises the consistency problem**, because if someone changes the IP of a website in the cache, we might have to wait a day before it propagates on the system.

The problem is that the address translation does not depend on where the user is: the Akamai idea is to make it dependent on where the user is.

10.2. Akamai

Let's assume that the content of www.cnn.com is stored in several servers around the world (Rome, Paris, New York, etc..). A user in Rome that asks for the translation of the address, gets the IP address of the Rome server, a user in Australia will receive the Sidney one and so on. Having a local copy of the web browser makes the access a lot quicker.
How does this work?
The idea behind **Akamai** is to make the system work without changing the structure of the DNS protocol. This is because the user doesn't know the existence of the Akamai server a priori and so he will use the usual DNS system (so in the case of www.cnn.com, it will ask the DNS server which will make a query to the root and so on..). When a user asks for a webpage, he's actually asking for an HTML file, which will likely contain other file references (such as images). After getting the index file from the requested website, what the Akamai system does is to "akamaize" the file requested, which will be the only file obtained from the original server. This is done by putting akamai.com in front of the address (so for example: https://akamai.com/www/cnn/com/picture.jpg). To resolve this new address now we have to ask the Akamai server and it will redirect the user to the closest server that contains that file (it is easy to understand our position using the IP address).
The first problem that comes into mind is **consistency**, but if we have an x-hours old copy of the page, it's not a big deal. The majority of systems gives **availability** the priority over consistency, obviously this doesn't apply to every system (e.g. banks). Another problem to solve is the **overload** on the akamai.com server: if a server has many requests the system could slow down but a solution might be to replicate the server (or use multiple servers) and then use load balancing (e.g. if the closest server has too many requests, move to the second closest one).

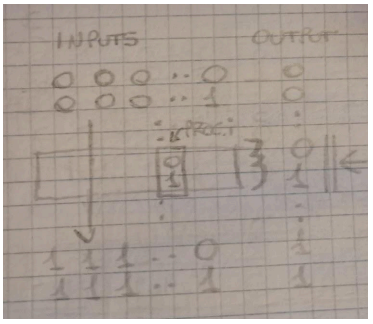
11. Lower Bounds in Consensus

How many processes do we need to tolerate f faults?
Let's define some **taxonomy**:

	No faults	Crash faults	Byzantine faults (without Digital Sign.)	Byzantine faults (with Digital Sign.)
Synchronous	1 round	f+1	3f+1	2f+1
Asynchronous	1 round	2f+1	3f+1	3f+1

If we have **no failures**, it doesn't matter if the system is synchronous or not, we know that every message will arrive at our destination at some point and we can safely send every one of them. A possible protocol might be making a broadcast so that each process sends its vote to all the other processes. They'll eventually reach a decision since everybody knows the other's decision. So **one round is enough**.

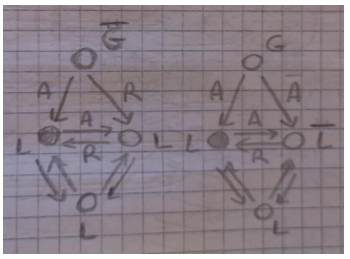
If the system is **synchronous** and we have to deal with **crashes**, let's figure out what we need. If we know that we will have at most one crash, then we know that if we do 2 rounds, we have for sure one round that is totally correct. If we have 2 crashes, then we need at least 3 rounds. So if we have at most f crashes, we need f+1 rounds to make the protocol work. In the case of a synchronous system we get both **safety and liveness**. A simple protocol that works in this way is the one that, after the completion of a round, sends to every process a broadcast message to see if they all have the same value, and, if not, restarts the round. If the system is **asynchronous** then we need 2f+1 processes but, for the FLP theorem, we know that we get **safety** but **not liveness**. The **simple proof of FLP** is the following: let's assume we have n processes and each of them have an initial value. We have some requirements:



- **Agreement**: All correct processes must agree on the same value;
- **Validity**: If all the processes agree on the same value, then the chosen value must be that one;

- **Termination**: Each correct process must eventually decide;

If all inputs are 0, then the chosen value must be 0 (for validity). The same applies to 1. Now, let's consider all possible configurations (like in the **Hamming Code**), so at each step we change one value. There will be a couple of configurations in which the first one has 0 as the output, while the second one has 1 as output. These 2 configurations are one next to the other in the Hamming Code and they **differ by just one bit** (this bit is process i). If the crash is process i then the other processes must agree either in 0 or 1. Since it is difficult to distinguish between a slow process and a crashed one, if the other processes commit 0 and the process i arrives later and its decision is 1, then the final result is wrong (the same for 1).



If the system is **synchronous** and we have to deal with **byzantine crashes**, then the system needs 3f+1 processes. Why 3f+1?
The proof is given by the **Byzantine Generals Problem (BGP)**: Let's assume we have 3 nodes, a general and two lieutenants L_1 and L_2. The general can give them commands that can be attack or retreat and they must do the same thing. If the General is byzantine (so we have one fault) he can send two different orders (he says to L1 to attack and to L2 to retreat), so there's no way in which the Lieutenants can know what is right to do. So, the two lieutenants could send a message to each other to tell the other one what's the message sent by the general (so they find out that they have different orders). If the byzantine node is one Lieutenant, he can send to the other one false information or behave differently to the General command, so the genuine lieutenant (the one with the full spot) has the same point of view as before. Assuming to have only 1 failure, with 3 lieutenants it is possible to understand what it's the right thing to do (so we need 3*1+1=4 processes).
In the case the system is synchronous, using these 3f+1 processes the system is both safe and live.
If the system is **asynchronous**, FLP is still valid so the number of needed processes is the same and we have safety but liveness is not guaranteed. So far **we assumed not to have digital signed processes**.
If we assume **we use digital signatures**, in the case of a synchronous system we can't forge commands so we fall in the case of the 2f+1 processes needed. If the system is asynchronous, messages are not guaranteed to arrive and this can be used for the faulty process not to show any proof of the decision, meaning that we need 3f+1 processes again even with digital signatures. The proof is called **PBFT (Practical Byzantine Fault Tolerance).**

12. GDPR

The **General Data Protection Regulation (GDPR)** is a **European Union regulation** on data protection and privacy in the EU. The GDPR's primary aim is to enhance individuals' control and rights over their personal data. Companies which must comply with GDPR must have a **Data Protection Officer (DPO)** that is the officer responsible for all personal data which must be stored properly in order to avoid data breaches. Citizens are some rights such as:

- Art. 15 **Right of access by the data subject**: In summary, the user has the right to ask a company (e.g. Google) all personal data concerning the user.
- Art. 20 **Right to data portability**: In summary, if the user has some personal data in a web server and he grants some other companies access to the data, then the company which has the data must be able to provide the data to the other ones using API or other ways in machine-readable format.

Every web site implements its procedure to start a subject access request. Usually, the procedure to exercise the right to access the data is described in the **privacy policy** page of the data controller. If no information is present on this page, it is necessary to contact the DPO via email.
Before transmitting the data, the DPO is supposed to **verify the identity of the requester**. As required by law, the DPO in this phase can ask for additional information if he has reasonable doubts concerning the identity of the requester. Some ways to obtain the identity of the user are: identity document, knowledgeable questions, email confirmation, phone call or the usage of cookie (open a page and read the cookie to the DPO). The problem is that most of these ways can be spoofed (e.g. email address).
The response format isn't standardised: A large number of controllers usually answer with a structured data format (JSON, CSV, XLS or XML), others use other formats or even photos of the data concerning the user. Notice also that companies should provide the list of third party trackers that possibly are used by the company itself.

Data transmission is usually done via email using plain text: this is a problem with sniffing or man in the middle attacks. Some other times TLS protocol is used but there are other problems such as the server that might be compromised or the recipient incorrect. The solution might be encrypt the data making attention not to use the same pattern to create passwords (e.g. *name_of_the_user*2022).
It is important to identify the requester because, starting from little personal data, attackers can build up a chain of requests and jeopardise the privacy of the user.

13. Cap Theorem

The **CAP theorem** (also named Brewer's theorem) states that any distributed data store can only provide two of the following three guarantees:

- Consistency: every read receives the most recent write or an error;
- Availability: every request receives a (non-error) response, without the guarantee that it contains the most recent write;
- Partition tolerance: the system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes;

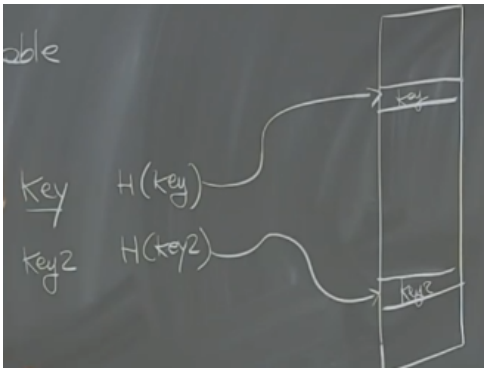
When a network partition failure happens, it must be decided whether to:

- Cancel the operation and thus decrease the availability but ensure consistency;
- Proceed with the operation and thus provide availability but risk inconsistency;

Thus, if there is a network partition, one has to choose between consistency and availability.

14. Consistent hashing

A **hash table** is a data structure used to store keys (that can represent any information) in a vector. Starting from a key we take its hash and this represents the location in which we store the information. After performing the hash of a key, it is possible to store it in the data structure just with one operation (just as reading). Most likely, two keys will end up in two different locations. We can have two or more keys that, after hashing, end up in the same place, so we have a **conflict**. If the position it's already taken we can deal with the situation in a few ways like:



- Storing the key in the next empty location;
- Build a list of keys;

Of course, in case of a conflict, storing won't be done in just one operation, especially if the hash table is overcrowded.

There is a problem for this data structure to be suitable for distributed systems. The location of the hash tables are computers that are part of the distributed systems. What if I want to add a new computer? One thing we can do it's to re-organize every file and thus we have to move them, and that's a huge problem. The same problem happens if someone wants to leave the system.
To adapt hash tables with the dynamic nature of distributed systems, we can come up with a different idea. Given the space of addresses, from 0 to 2^{m-1} (m is the space of the hash, not the number of nodes), we can visualise a ring on which we map every number in the space of addresses. If we now take a key, let's say k_1, and hash it (we obtain $H(k_1)$), we can map a dot on the circle. Another thing I can do is take a computer, let's say S_1, and hash it (obtaining H(S1)) and we represent it like a square on the circle. All the keys that go from one computer S_i to another computer S_j are stored on S_j.
So, for example, the key K_1 is stored in the computer S_1.

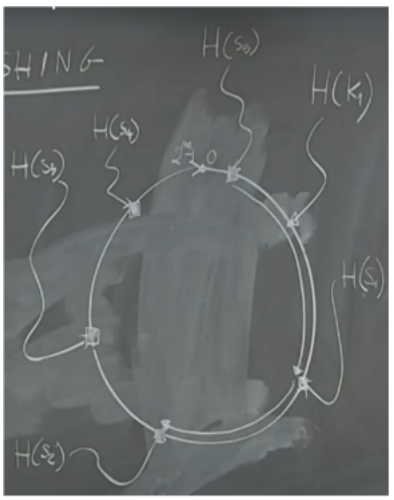


Figure 19: Consistent hashing.

Let's assume that we have a service that, given a certain point on the ring (where we want to store a key), tells us what server is responsible for that point. In this way we can contact the server and give it the value to store. If we want to search a file and we have the identifier, we hash the key and the service tells me what server is responsible for that key and we can ask it for the value.

What's Consistent Hashing?
We can build a routing table in order to get a system where every node knows the next one. How can we implement a service that tells us what server is responsible for a point? We can ask a server if it is responsible for a point and iterate all over the servers until we find the one we want. Obviously this system is a lot expensive and inefficient.
The idea is to build a **routing table** in a way that every server S_j knows who is responsible for the key with the property $H(S_j) + 2^i$ for i=0, 1, ..., m−1. If we are looking for a key we can ask any of the servers and it will redirect us to the nearest server responsible for that key. So we can get to the wanted server in a logarithmic time on the number of servers.
The steps, defined by the i parameter, are called **fingers**.

If we add one server S_x to the system, it will fall between two servers, let's say S_i, S_j (in this order respectively). The files that we have to move are just the files that now are under S_x responsibility, this to make S_j aware that it's not responsible anymore for them. Note that moving a file means that we are actually moving a part of the hash table linked to a server (so we're actually splitting it). Of course we have to change the finger to make everybody know that there's a new server. It's the same when leaving, S_x can simply send all its files to S_j (the next server).