Distributed Systems

Simone Zannini

December 2021

1 Distributed computation

1.1 Introduction

A distributed system is a system in which you have n processes distributed on a network. These processes cooperate to complete a single task. Each of these processes are running somewhere. Communication system is made of channels that connect every pair of processes. Channels can be reliable or unreliable. The first assure that if you send a message, it'll arrive; the latter instead tells us that the message can be lost. In the reliable one, messages can be received out of order or with a FIFO structure. An example of reliable channel is TCP/IP, of unreliable is UDP.
In distributed systems we assume that channels are reliable and these channels implement resending messages (if one doesn't arrive, simply resend it). We have 2 types of systems:

- Asynchronous (no time bound for message arrival). Mostly of them are asynchronous.
- Synchronous (may have a bound, e.g., we can assume in X seconds the message will arrive).

Besides, we can't assume a relative speed for processes, instead some of them are fast, some of them are slow.

1.2 Processes timeline

We can represent processes of a DS like events on a timeline.

We can imagine that an event "e" is, at very low level, a simple instruction (like x += 1). This isn't exactly true and we can say that with an event we are representing an "important"

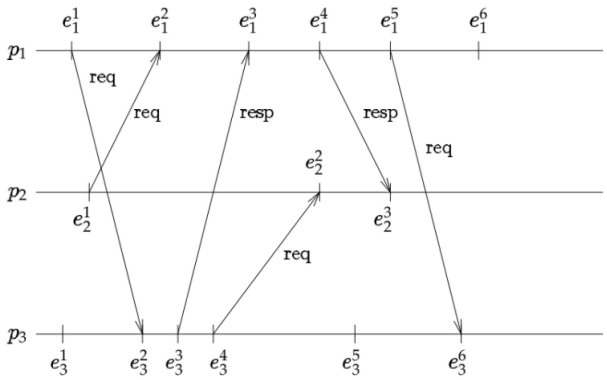

Figure 1: Processes timeline.

instruction (like the sending of a message). It's important to say that if we re-run the **same** DS, these events can happen on different times. So, this example is what we define as a *possible* run of the system. However, the system **must** work in every possible run. There are also *impossible* system runs, like the ones where an event that *receives* a message happens before the event that *sends* that message. 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_1^1, because it can change the context of the process. We can't say the same thing for events e_2^1 and e_3^1 because they can't affect each other, so it's not important for us the order in which they take place. We call this type of events concurrent. Generally, given $\rightarrow$ = "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 = send(m) and e_j = receiving(m), then $e_i \rightarrow e_j$.
- $e^i \rightarrow e^{ii} \rightarrow e^{iii}$ then $e^i \rightarrow e^{iii}$ (transitivity).

$\rightarrow$ is the smallest relation with these 3 properties, informally $\rightarrow$ is the simplest relationship that can be made with these 3 properties.
To make another example, seeing the picture above we can say that e_3^5 can happen before e_2^1.

1.3 Consistency

If we have a run R and we stop it at some point, we can make a **cut**. A cut representation:

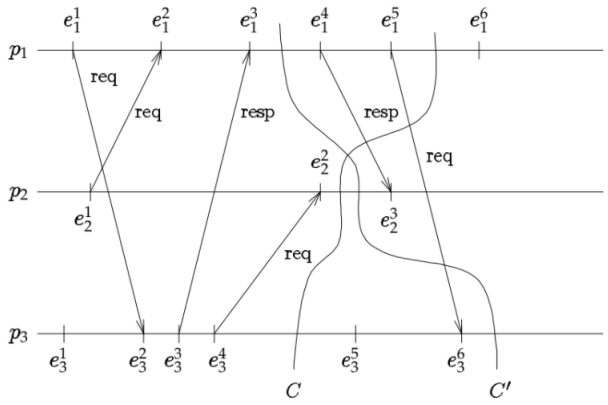

Figure 2: Run cuts.

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

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

1.4 Deadlock

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 in check periodically if the DS is in deadlock and remove it. A possibility is to add a process, let's say p_0, that can ask the state of the system to every other process.

We have to remember that the system is asynchronous, so the p_0 message can arrive in different moments for different processes. When p_0 receive every message can build a graph that represents the "waiting situation" between processes. A cycle in a waiting graph is a deadlock. Looking at this situation, we can easily notice how p_2 makes a query to p_1 and is still waiting for response, p_2 can say to p_0 that p_3 is waiting for its response and eventually p_3 will say that

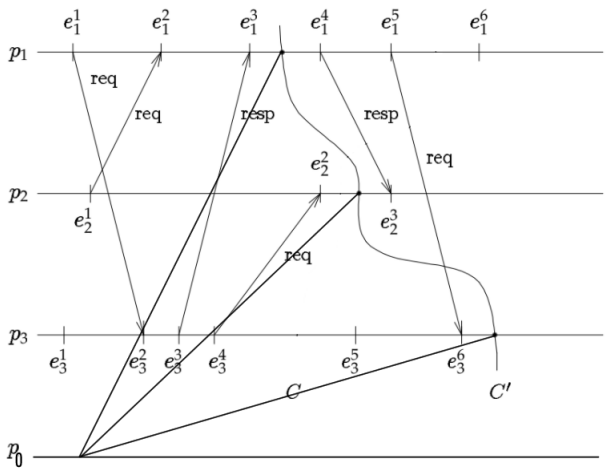

Figure 3: Messages to check system state.

p_1 is waiting from its response. Thus, we have a deadlock, but we know that in the system there isn't one. In fact, we can see how every query have its own response. So, why this happened? Due to the asynchronous nature of the system, p_0 can detect a deadlock where there isn't one. This happens when p_0 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 Clocks

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 to p_0 a message after every event. At some moment in its timeline, p_0 will receive these messages, that represents a run of our system. Is this a consistent run of the system? Unfortunately no, because we don't know in which order these messages will arrive. Thus, we label these messages with the RC and we order them when p_0 receive these. Is this a consistent run? Of course it is, because we know that:

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

Notice that we don't have double implication ($\iff$), because e and e^i aren't necessarily related.
Instead of using the RC we can invent *our clock*, OC, that doesn't use the RC but it has the right property described above. We can assume that every process tags events with sequence numbers and these numbers represent OC. Informally, the rule that we use to build this diagram is:

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

By construction, OC preserve the RC rule. Thus, p_0, by ordering labels, has now a consistent run of the system. This clock is named Lamport's clock.
We have still a little problem, in fact given the possibility of identical labels, p_0 can order these labels reconstructing a *possible* run, but not the exact run that's happened. Besides, we have

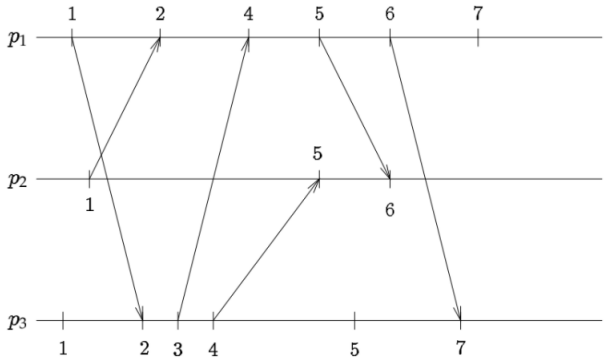

Figure 4: Our clock (actually, Lamport's clock 😅).

another problem. We aren't sure if some messages will arrive later (remember, we have asynchronous systems) and marked with a label smaller than the greatest one received (to make it simpler, the label of an event that occurred before of any of the already received event labels).

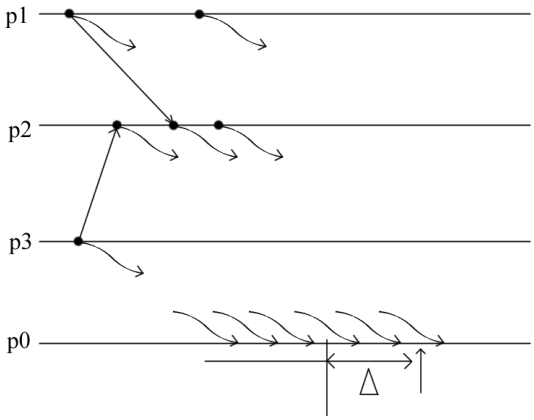

Figure 5: Delta time bound on synchronous system.

We can define the **delivery rule** (DR):

At time t, deliver all messages in order whose timestamp is smaller than $t - \Delta$.

Let's assume we can use the RC and we are in a synchronous system. This means we now have a time bound, called Δ, 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.

Definition 1 *A message m is **stable** if no future message with timestamp smaller than TS(m) can be received.*

Now we can define a new delivery rule (**DR2**), using Lamport clock:

Deliver all received messages that are stable in timestamp order.

Following the above example, if e_3^2 arrives to p0 ($TS(e_3^2) = 5$) we have to wait until the notification with timestamp 4 (e_2^3) arrives to deliver. Let's see that we can do better than this. Firstly, we want to enforce the clock condition defining the *strong clock condition* (SCC):

$$TS(e) < TS(e^i) \iff e \to e^i$$

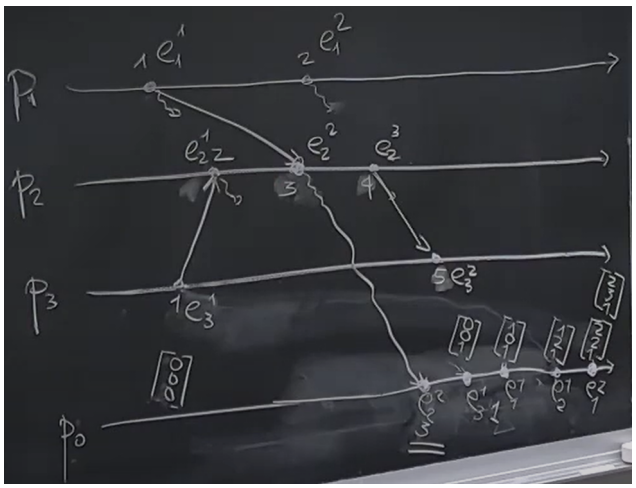

Figure 6: Example of the 2nd delivery rule.

Definition 2 *The **history** of an event* e_1 *is:* $\Theta(e1) = \{e \mid e \to e_1\} \cup \{e_1\}$.

From which we can get $e_i \to e_j \iff \Theta(e_i) \subseteq \Theta(e_j)$. We can now notice how the Θ set is nothing but our TS (timestamp) that follows the SCC (and that it is a consistent cut as well!). A way to encode this set is an n-dimensional array where n is the number of our processes and inside this array we have the last (the biggest) timestamp for each process in the Θ cut. These kind of arrays are named **vector clocks**.
Returning to our first example, we can now tag the events with these vectors as their labels:

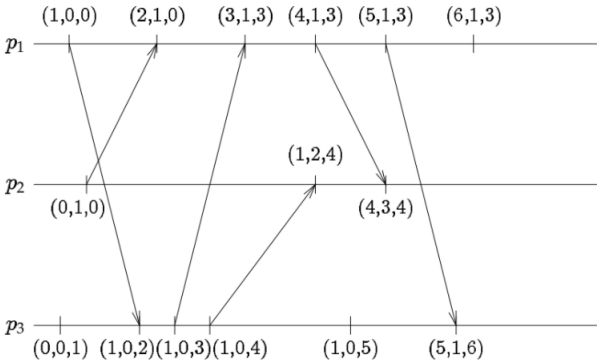

Figure 7: Vector clocks.

The comparison between timestamps lets us know if two events are related with the $\to$ relation or not (concurrent).
Thus, we can make our third delivery rule (**DR3**):

Deliver message m from p_j 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 solve also limit situation. Let's see an example: p_1 can understand that m_2 is caused (or preceded, at least) by m_1, this can be done by looking at m_2 timestamp. If p_1 receives m_2 before m_1, it delays its arrival (as we can see in the screenshot below). This behaviour is called **causal delivery**.

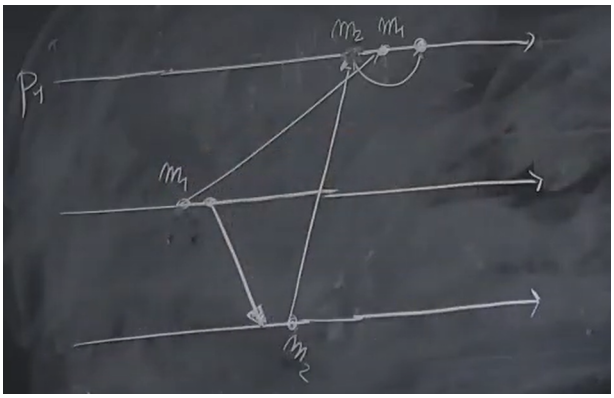

Figure 8: Causal delivery.

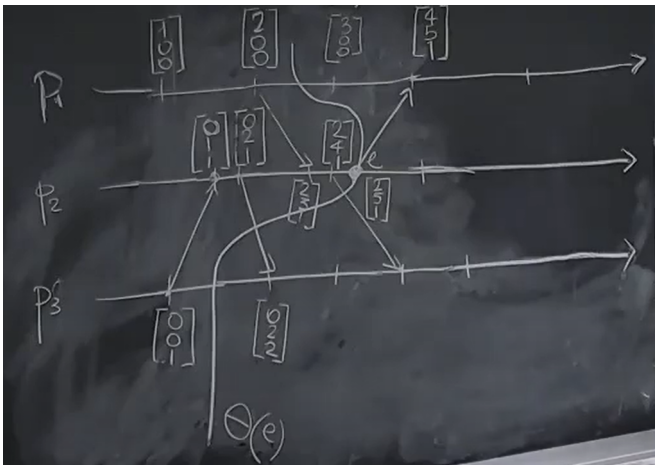

Figure 9: Vector clocks, another example.

1.6 Distributed Snapshot

It is a protocol designed by Chandy and Lamport (and takes also the name of Chandy-Lamport protocol). This protocol makes use of FIFO channels. The way p_0 takes 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 p_0). This applies to all the processes of the system.

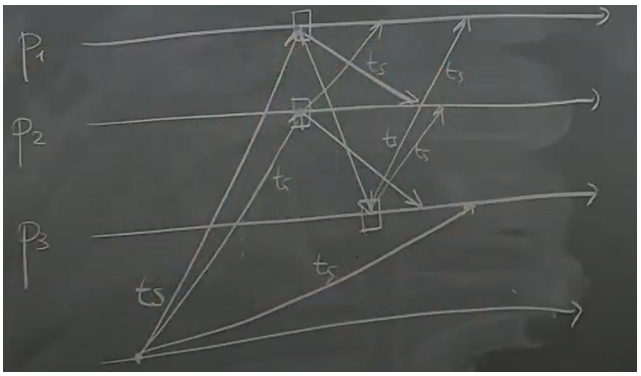

Figure 10: Chandy-Lamport protocol in action.

*How do we know if a protocol is ended (protocol **termination**)?*
It's easy to tell when the protocol started because the processes receive messages from it. 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, let's say S, that is made from a snapshot and an event $e \mid e \notin S$ and $e^i \in S$ with $e \rightarrow e^i$. Why isn't it possible?
We have a snapshot $s1$ taken before e because $e \notin S$ and we have a snapshot $s2$ after e^i because $e^i \in S$. $s1$ will trigger a ts message that will arrive after $s2$ (and e^i) and this is impossible because our channels are FIFO.
Another way to say it is that, given the FIFO structure of the channel, the ts message triggered by $s1$ **must** arrive before the e^i event, but this would mean e^i is in the future that implies $e^i \notin S$.

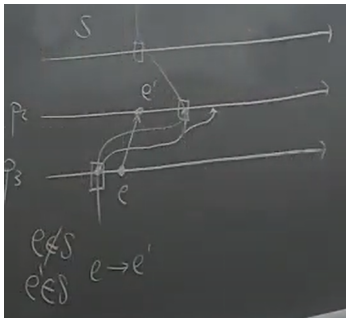

Figure 11: Demonstration of snapshot consistency.

2 Protocols

2.1 Atomic commit

We have a number of processes in our distributed systems, $p_1, p_2, ..., p_n$, we want that all of them are able to know whatever 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). Our biggest problem are *faults* and we're plenty of them:

- Crash failures (systems stops working).
- Byzantine failures. Byzantine processes can do every sort of thing, even malicious and thus make the system fail. These kind of failure is worse than a crash.

An atomic commit means that 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 with a crashed one.
Two important properties of a distributed system are *safety* (Italian public administration, very slow, maybe sometimes they don't do anything just to be safe 🤣, quoting Mei) and *liveness* (does something but doesn't ensure safety). We have to try to at least to reduce the problem if we can't solve it. Let's focus on crash failures.

2.1.1 Properties

AC 1 *All processes that reach a decision, must reach the same one.*

AC 2 *Processes cannot reverse their decisions.*

AC 3 *The commit decisions can only be reached if all processes voted "yes".*

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

AC 4 *If there are no failures and all processes voted "yes", decision must be to commit.*

This last property tells us that, even if all processes voted "yes", if there's a failure you have to abort.

2.2 Two-phase commit

It 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 sends the **vote request** message and participants sends the vote to the coordinator. If any of the votes is "no" then abort. If all the votes are "yes", send a commit message, otherwise send an abort message. The participants act accordingly to the coordinator message.
Let's introduce failures in this protocol. Basically what happens is that some of the messages don't arrive. For example, if one participant doesn't receive a vote request from the coordinator and the waiting time finishes, then it'll abort and everything will be aborted according to the protocol.
If the coordinator doesn't make it to send the decision, we are stuck in the last phase (participants don't know if the final decision is abort or commit). What we can do is send a broadcast message to all the participants to understand what was the decision, this can mitigate the problem. This approach is called **Cooperative Termination Protocol**. Another approach is named **Recovery Protocol** instead.

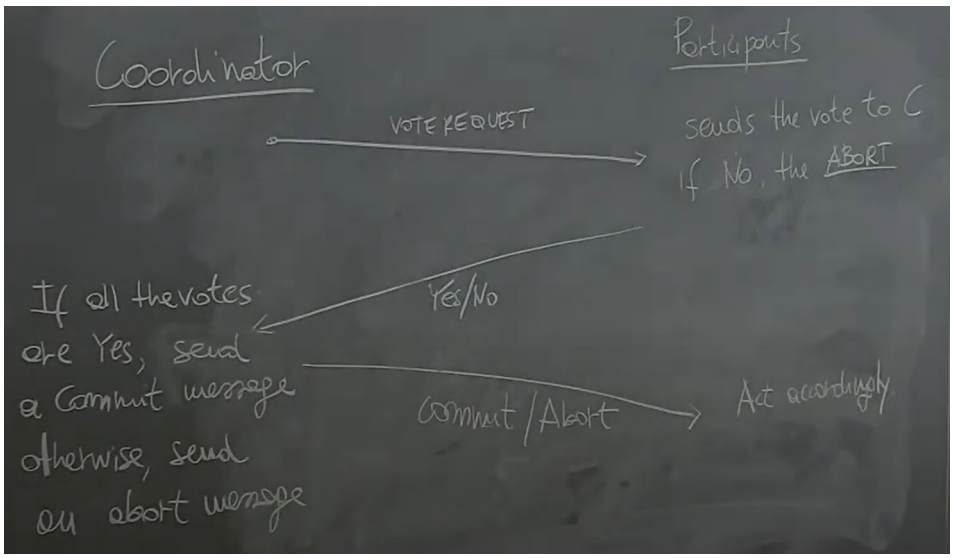

Figure 12: Two-phase commit.

2.2.1 Recovery Protocol

The idea behind this protocol is that, assume that a process crash, 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.
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 ways.

AC 5 *Consider any execution containing only failures that the algorithm is designed to tolerate. At any point of the execution, if all failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.*

2.2.2 Logging

We utilize a **DTLog** (Distributed Transaction Log) that follows these rules, given `C` as Coordinator and `P` as participant:

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 send 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 `Yes` to `C`. Now it's important to log before, because if you log after sending, you might crash between the 2 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 (nothing is written in the log) but it can't because if it sent a `Yes` there might be a commit procedure running.
   If P votes `No`, it has to write `No` on DTLog *before or after* sending `No` to `C`.

3. If `C` decides `Commit` it has to write `Commit` to DTLog `before` sending `Commit` to Ps.
   If `C` decides `Abort` it has to write `Abort` to DTLog `before or after` sending `Abort` to Ps.

4. After receiving the decision, the participants write it in the DTLog.

2.3 Paxos Protocol

It is a protocol designed by Lamport for distributed consensus, works in an asynchronous system and it can deal with crash failures. 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.
How many crash failures can Paxos 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. We should see that we can't tolerate more than $n/2$ because 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 $f = \lfloor \frac{n-1}{2} \rfloor$.
We have three types of nodes:

- Proposers.
- Acceptors.
- Learners.

In real system a node, at the same time, can be all the 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). Any proposer can start a round and it sends a `prepare(i)` message to all the acceptors where `i` is a round.
All the acceptors reply to the proposer 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 won't participate in round smaller than i. We can for now assume that this forces the progress of the protocol.
The proposer can now send an `accept(i, v)` message where `v` is the value associated with the largest round in the promises. We take the maximum value among `lastround` parameters and we take the `lastvote` associate with it. 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 local).

2.3.1 Example

Let's see an example where p is a proposer and a an acceptor:

1. `p` sends prepare(2).
2. `a`ll a send promise(2, -1, _).
3. p sends `accept(2, 8)`.
4. all a send `learn(2,8)`.

If another proposer tries to send `prepare(1)` the acceptors reject the message because $1 < 2$.
Let's see what could happen next straight from the blackboard:

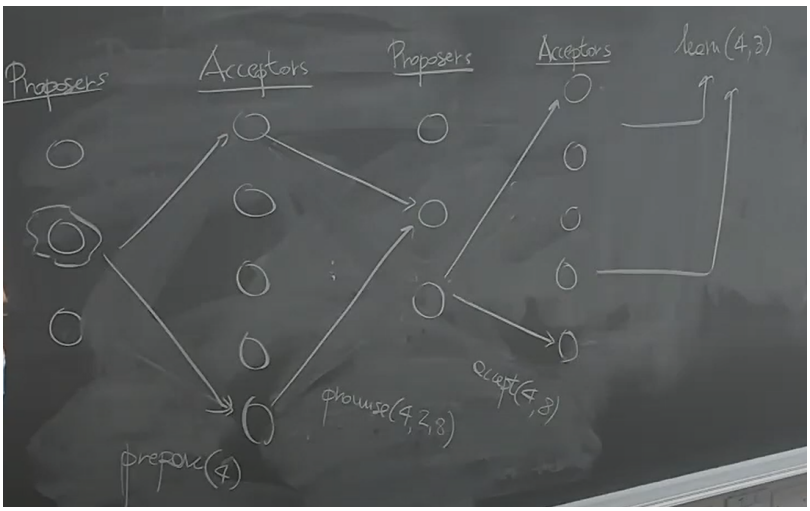

Figure 13: A Paxos example.

I recommend to see the lesson (2021-11-07) from timestamp 1h47'55" to see a complete and less trivial example (impossible to report by screenshots).

2.3.2 Safety

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. If we can prove it, then Paxos is safe. We can prove it by induction:
We take 2 rounds, *lastround* and i where *lastround* happened before i. In round i a majority, let's call it Q, 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 i a majority voted for something. During *lastround* the voted value must be v, because 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.

2.3.3 Liveness

If we are able to elect one and only coordinator, then Paxos is live. Why? 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 live. Choosing a coordinator need a consensus, the problem is called **leader election**. But we don't need a safe way to choose a coordinator, because Paxos is safe, so if for some problems we find more than one coordinators this is not a big deal (we lose liveness but not safety).
An unsafe way to choose the coordinator is take the one with the lowest id and make it sends to other proposers its *heartbeat* every x seconds, that is a message to simply say "I'm alive". If after x seconds the message doesn't arrive, then the proposer with the lowest id between the others takes the role of the coordinator. When a coordinator is chosen, it starts Paxos protocol. It's trivial to see that this idea isn't safe, if some messages are lost we may be in a situation where we have multiple coordinators. But as we said before, we don't care, Paxos is safe.
In real life systems we don't have one value to choose, but a sequence of value. 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, ... 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, making it easier for proposer

to choose a value for a new instance. 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 efficient). 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 also the third message a single one for X instances. To sum it up, the coordinator sends 3 messages, `prepare, promise, accept-any`. The proposers 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. This version of Paxos is called **Fast Paxos**.
The problem with this version of Fast Paxos is that it's possible that in the same round we accept more than a single value.

2.4 Fast Paxos

Fast Paxos makes sense if you have more than one instance of Paxos. Let's check a visual representation:

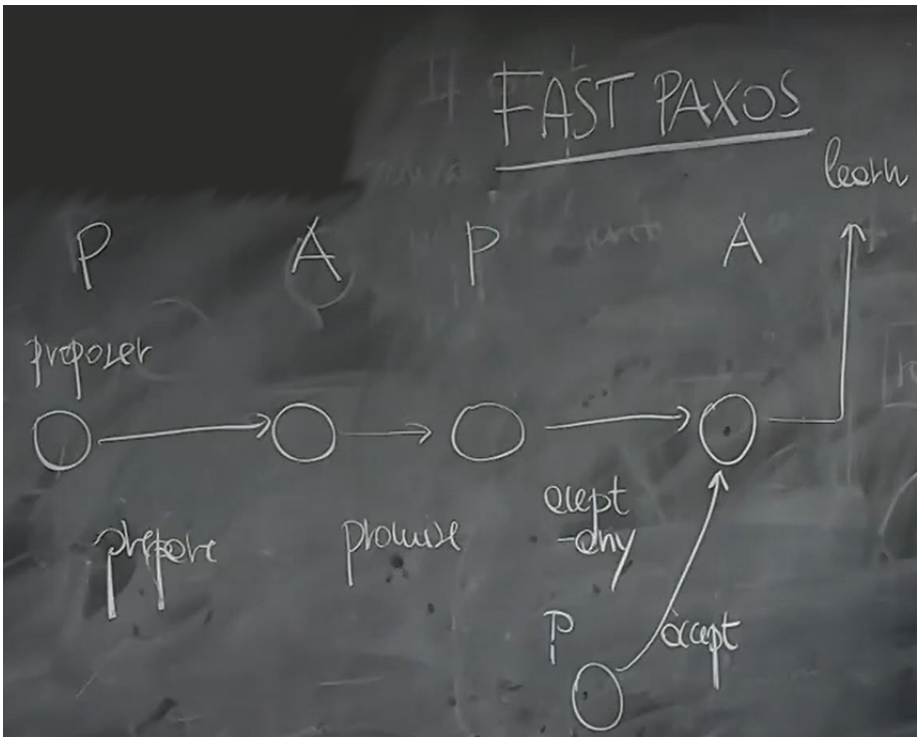

Figure 14: Fast Paxos.

As we said before, the problem with this version is the value's choice. In Paxos, if two proposer voted in the same *lastround* we have the same *lastvote*. With Fast Paxos is possible to have more than one acceptor in the same *lastround* with different *lastvote*. If the maximum $lastround = -1$ then this means that no one voted, so the proposer can send `accept-any`. This can happen, for example, if in the same round of a Paxos instance two proposers, triggered by an `accept-any`, send an `accept` message with 2 different values at the same time. Given this, it's simple to understand a limit case.
Given A the set of all acceptors and Q a Paxos majority ($\frac{n}{2} + 1$), if we split Q in two almost equal parts ($\frac{n}{2}$, $\frac{n}{2}+1$), P, R, we can be in the case where $\forall p \in P$, p voted a value v and $\forall r \in R$, r voted a value $v^i \neq v$. Every other acceptor $a \in A \setminus Q$ could vote either for v or for v^i or

for another value, or even didn't vote at all; but, for some reasons, we don't know the content of their messages (they're too slow or lost). It's simple to see that we can't know what is the majority, because if they all voted v then the protocol have to learn v and if they all voted v^i it's the other way round.
We can come to the conclusion that a Paxos-like majority is clearly not big enough.
The number of acceptors needed to form a majority in Fast Paxos is $\frac{2n}{3} + 1$ indeed. In this case if we split Q ($|Q| = \frac{2n}{3} + 1$) in two almost equal parts ($\frac{n}{3}$, $\frac{n}{3} + 1$) we can form two sets of acceptors, again P and R, with one of the two, say P, such that $|P| = \frac{n}{3} + 1$. We can notice that $|A \setminus Q| = \frac{n}{3} - 1$ and this implies that, given $A \setminus Q = X$, $|R + X| < \frac{2n}{3} + 1$, so we are sure that the value voted by acceptors in R can never be a majority.
Thus, we have find a way to select a vote that is safe in Fast Paxos.
The choice of this majority it's same as saying that Fast Paxos can tolerate f failures with $f = \lfloor \frac{n-1}{3} \rfloor$. In a system with common conflicts, Fast Paxos is slower than Paxos, this is why in real systems Paxos is a lot more used than Fast Paxos.

2.4.1 Safety

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^i, some for $v^i i$, ..., 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 reached a quorum, then it's safe to vote v, but even if v didn'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.

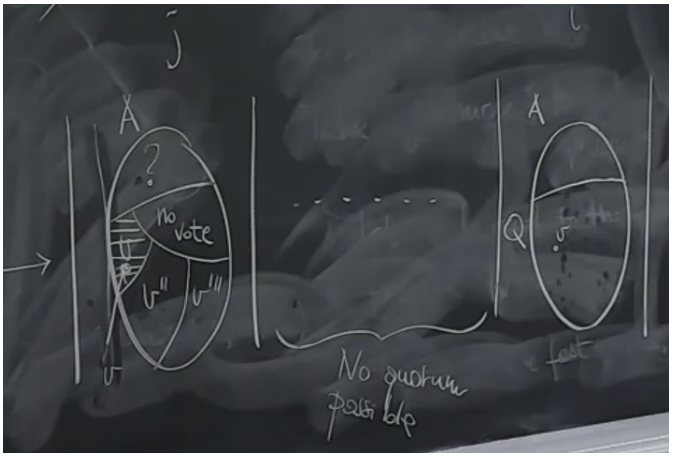

Figure 15: Fast Paxos safety proof.

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, 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 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 an harder problem then simple crashes, FLP is still valid. One of the most important protocol that tries to deal with byzantine problems is **PBFT**. If a synchronous system has to deal with byzantine behaviours, we can solve the problem. Actually, using PBFT in a synchronous 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*, either `attack` or `retreat` and the two Lieutenants must do the same thing. If the General is byzantine he can send two different order, so there's no way in which the Lieutenants can know what is right to do. If the the byzantine node is one Lieutenant, he can send to the other one a false information or behave differently to the General command. So it's easy to see that with 3 person, you can't understand what it's the right thing to do.

4 Bitcoin

It was invented in 2009 by Satoshi Nakamoto (actually it's a fake name). Its 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 and 18 millions of them are already generated.

The actors in this ecosystem are people, authorities and miners, who mint (coniano) bitcoin and approve transaction.

4.1 Easy crypto

4.1.1 Hash functions

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

- h is easy to compute.
- A is infinite, B is finite.
- $h^{-}1$ is hard to compute.
- Given x and $h(x)$ it's hard to find y such that h(y) = h(x).

Few examples of hash functions are MD5, Sha-1, Sha-256, ...

4.1.2 Public Key Crypto

We can have a process that generates two keys $k1$ and $k2$ that are related. We can have a message m encrypted with $k1$ resulting in m^i and this message m^i, encrypted with $k2$, gets us to m again. $k1$ is called private key, meanwhile $k2$ is public. This allows us to send a message encrypted with the $k2$ public key and make this message decryptable only by the one who has the private key $k1$.
Another usage is signing message. So we can use the private key $k1$ to sign a message and everyone can decrypt it with the public key $k2$ and be sure that the message could only be written by $k1$ owner. Typically signing is not done like this (2021-10-21, 23'05").

4.2 Bitcoin and consensus

A bitcoin is like a file in which there are written informations. The owner of a bitcoin is represented by a public key. If you want to transfer bitcoins you can write on the file the new owner B and a signature A (old owner). In some way we are creating a new bitcoin file with the number of bitcoins and owner B. This is what we call a **transaction**. 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, 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 way we talk about blockchain.

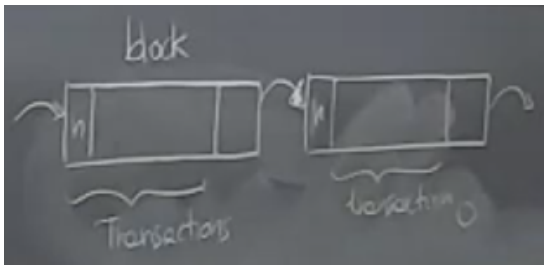

Figure 16: Blockchain example.

If a person tries to do double spending, the receiver can check in the blockchain if a transaction with the same coin exists.
The key point here is: who decides the next block?
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).

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**. This is an arbitrary number. The hash of the block must have the last k bits set to zero to be valid, but we have no warranty that it'll be like this. Thus, the miner can change the block 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 block in the world is 10 minutes. For example, with $k = 32$, in average you have to make 2^{32} hashes, and actually $k > 32$. Once you find the nonce you can send the block and everybody accept the new block in the chain.
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 message to everybody and maybe one block gets to a part of the miners and the other block to the other part. So we have created two branches of the blockchain that two parts of the miners are trying to extend. What usually happens is that only one block "wins" and everybody stop working on the other branch, make it a dead branch. If a transaction is on a dead branch, it won't be approved. How can 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.
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 makes the other branches dead.

4.2.2 Consensus protocol

We have seen how miners choose what block is the next one in the chain. It's easy to see that this protocol is nowhere near to safety, because of the chance of a fork. 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.
This is an example of a randomized consensus protocol, safe and live with high probability.
This protocol has several problems, the first it's the high power resources needed.

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. not writing searches in history. The type of privacy we want to analyze is instead a *global* privacy, which means anonymity from internet provider, government and so on.

5.1 TOR Anonymity System

TOR stands for The Onion Router. The idea of TOR is that we have a number of servers in the world named TOR relays (about 8 thousandths). If a person wants to connect to a website you first connect to a relay called *Guard*. the second one is just another one and the last is called *Exit point*. To navigate in TOR we have to connect to 3 relays at least. Every relay can be located in different parts of the world.
The idea is:

- The Guard knows you but has no idea of the website you want to reach.
- The relays in the middle know nothing.

- The Exit point knows the website but not you.

How we can do this?
Let's assume we have n keys with $n \geq 3$ and we encrypt our internet request first with k_n, then with k_{n-1} down to k_1, building a layered structure that resembles an 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?

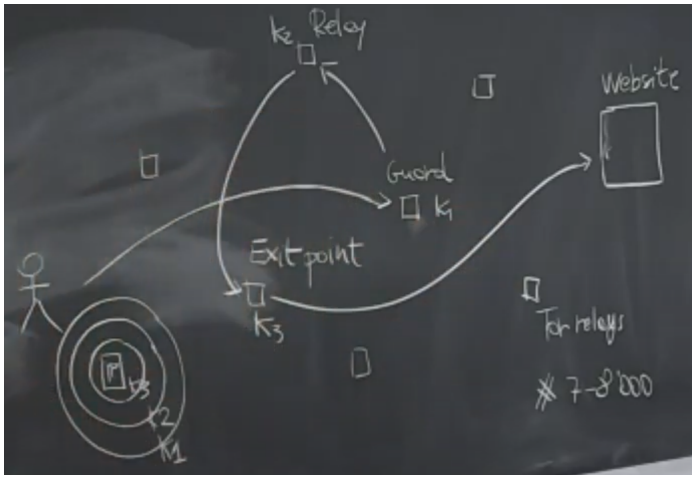

Figure 17: TOR.

Firstly, we want to make sure to be as anonymous as possible, so we can't share our private key (if we have one) to the Guard, instead we use a protocol that enables the guard and us to make a key. 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**.

5.1.1 Diffie-Hellman Protocol

There are two guys, g_1 and g_2, that want to share a secret. Both of them think of a number, a and b respectively. 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 $(g^b)^a = (g^a)^b = g^{ab}$. We can safely assume that is extremely hard to reconstruct a starting from g^a or g^b and so on, this is due to the complexity that lays behind the g operation.
To choose the key between user and Guard we don't exactly use Diffie-Hellman but an extention called **Authenticated Diffie-Hellman**.

5.1.2 Authenticated Diffie-Hellman

The two actors are User and Guard now. The user knows the public key of the guard. So the user can send an encrypted message with this key $E_{pk}(g^a)$. The guard answers with $g^b, H(g^{ab}) \mid$ "$Handshake$". When the user get this message we can compute g^{ab} and hashing it we can check that the protocol worked correctly. This version is called *authenticated* because in this way the user can be sure that he's communicating with the guard. In the plain version we are vulnerable to Man in the Middle attacks.
To extend our TOR circuit we use the previous relays to reach a new one. In the previous example (blackboard screenshot) we don't run this protocol directly with the second relay, but we communicate with it through the Guard.

5.1.3 Why TOR is used and who use TOR

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 we can do researches (and were made) to know, statistically, why TOR is used and who use TOR.

5.1.4 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, in theory, if all the relays have good connection, the user won't lose bandwidth, but it's unlikely.

5.1.5 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 to one of your nodes and after a second arrive to another one. If this happens frequently you can understand who is the user and which is the website. Obviously is hard to have both starting and end point of the system.
There are systems that assures that Guard and Exit point are in two different part of the world (not 100% safe, bust most likely).
Another attack can be done by the website. It can de-anonymize information looking at the way the browser use to send messages. The browser, when connects to a website, sends a lot of information (OS, Browser version, size of display, ...) and if you collect all this information you can identify a unique character among 200 thousandths 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 breaks anonymity, but pseudo anonymity (the website doesn't knows 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.

5.1.6 Service privacy

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 a TOR relay called *directory service* using TOR. 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 is the website. The directory service now selects another relay called *rendez-vous point* and then the user and the hidden service communicate, using TOR, with the rendez-vous point (used like a proxy). The directory service changes everyday.
How can we address an hidden service?
In the normal way, but with a `.onion` domain, only solvable by communicating with the directory service. Websites that use this domain form the **dark web**.

5.2 VPN

They weren't build with privacy in mind, but to virtualize a local network to a not local one, i.e. 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.

6 FLP (Fischer-Lynch-Paterson) Theorem

The statement that we want to prove is there isn't a solution to consensus in an asynchronous system with at least one failure. The specification is the essence of every problem. We are going to prove that there is a contradiction between the possibility of a solution and the definition of consensus problem itself.
Specification of consensus:

- **Validity:** if everyone propose a value, that's the value that should be decided.
- **Agreement:** if a correct node decides on a value, every correct node decide on that value.
- **Integrity:** the value that is decided upon must be a proposed value.
- **Termination:** participants eventually decide.

6.1 Intuition

In an asynchronous system, a process can't recognize if another project is too slow or dead. This leads us to two different situations: wait maybe forever or make a possible wrong decision. Before we dive into the proof, we want to come up with a model that can encapsulate any other consensus protocol.

6.2 The model

We have n processes and our network is a *message buffer* with infinite space. When we send a message we actually depositing it in the buffer. Periodically a process can take a message from the buffer and that message is either destined to the process or is a special λ message. We have the warranty that, pulling out messages forever, the process will eventually pull out its message. The λ messages are there to symbolize the waiting in an asynchronous system. Only p can fish its messages.
Every algorithm that we are going to build consists of two phases:

- **Receive phase**: some p removes from buffer a message or λ.
- **Send phase**: p changes its state and adds zero or more messages to the buffer.

We make two assumptions:

- **Liveness:** Every message sent will be eventually received if the receiver tries infinitely.
- **One-time:** p sends m to q at most once, i.e messages can have the same content but are unique.

We're gonna prove that, in a system like this, binary consensus is impossible and then that if we can't solve binary consensus, we can't solve neither the general one.

6.3 Configurations and schedules

6.3.1 Configurations

A configuration $C = (s, M)$ is supposed to capture the state of an algorithm A. s is a function that maps each process to its local state and M is the set of messages in the buffer. Thus C consists of a function that's able to produce the local state of each of the processes and a quantity that resembles the content of the buffer (not λ messages. they're infinite).
Every step of the algorithm that we take is a step towards a next configuration and if I'm in a configuration, not every step is possible. A step $e = (p, m, A)$ is applicable if and only if $m \in M \cup \{\lambda\}$.
We can then get a new configuration C^i applying the step to C, $C^i = e(C)$.

6.3.2 Schedules

A schedule S of an algorithm A is a finite or infinite sequence of steps of A. A schedule is applicable to a configuration C if I can apply the first step s_1 to C, then s_2 to $s(C)$ and so on. The empty schedule is applicable to every configuration.
If S is finite, $S(C)$ is the unique configuration obtained by applying S to C.

6.3.3 Correlation

A configuration C^I is accessible from a configuration C if there exist a schedule S such that $C^I = S(c)$. C^I is a configuration of $S(c)$ if $\exists S^I$ prefix of S such that $S^i(C) = C^i$.

6.4 Run

A run of an algorithm A is a pair $\langle I, S\rangle$ where I is an initial configuration and S is an infinite schedule of A applicable to I. A run is partial if S is a finite schedule of A. A run is admissible if every process, except possibly one (the one failure), takes infinitely many steps in S. An admissible run is unacceptable if every process, except possibly one, takes infinitely many steps in S without deciding.

6.5 Classifying configurations

We have three types of configurations:

- **0-valent**: a configuration C is 0-valent if some process has decided 0 in C or if all configurations accessible from C are 0-valent.
- **1-valent**: a configuration C is 1-valent if some process has decided 1 in C or if all configurations accessible from C are 1-valent.
- **Bivalent**: a configuration C is bivalent if some of the configurations accessible from it are 0-valent while others are 1-valent.

We're going to prove that exists some initial bivalent configuration.
Suppose A solves consensus with one crash failure and, let I_j be the initial configuration with I_0 0-valent and I_n 1-valent. We suppose, by contradiction, that there isn't a bivalent configuration.
We know that, to get to a 0-valent configuration (I_0) to a 1-valent one (I_n) we must have at least two neighbouring configurations where we change the decision (neighbouring = they are the same except from the one node that flips the decision).
We can then take two configurations, I_k I_l, where the only difference between them is that in I_l there's one node that has 1 as initial value. Before this can lead to any decision, we kill that

node. Now the two configurations are identical. We can claim that, given I_k is 0-valent, there's a finite schedule that leads me to a 0-valent state, but applying the same finite schedule to I_l we get a 1-valent configuration. This can't be possible because finite schedules are deterministic, if we have the same starting point we get to the same end point. This contradiction comes up because we suppose that there aren't bivalent configurations.
The next lemma is called commutativity lemma.
Let consider two schedules, S_1 $S2$, both applicable to a configuration C. Suppose that the set of processes taking steps in S_1 are disjoint from the set of processes taking steps in S_2. Then we can apply the two configurations in whatever order we want and we get to the same configuration.
The next lemma is called the procrastination lemma.
The intuition is that, no matter the step we want to take, we are always able to insert a schedule before that step (due to the asynchronous nature of the system) that leads us to a bivalent configuration. Thus we can make sure that the protocol will always go from a bivalent to a bivalent.
Let C be bivalent and let e be a step applicable to C. Then there is a (possible empty) schedule S not containing e such that $e(S(C))$ is bivalent.
We are going to assume that this lemma isn't true and that exists some step e that can lead me from a bivalent configuration to a monovalent one (0 or 1).
Thus we're going to assume that $e(C)$ is monovalent and for convenience 0-valent. We're going also to use a *mini lemma*, i.e. that exists a schedule S_0 applicable to C that does not contain e and such that $D = S_0(C)$. We can then do $e(D)$ and get to a schedule that is 1-valent. Let's prove this.
C is a bivalent configuration and we assumed that from this, we can for sure reach a monovalent configuration. If D is 1-valent and we reached it without using e we know we can apply e and get a 1-valent configuration, by definition of 1-valent.
There's a way to reach a 1-valent configuration without using e?
Suppose that I **must** go through e to get to a 1-valent state. We're going to call this configuration E, that is reached firstly applying an e-free schedule that we know will lead us to D, then from D we apply e and get to $e(D)$ and a few steps later we're going to get to E.
If E is 1-valent $e(D)$ can't be 0-valent, and it's either 1-valent or bivalent, but, if we look at it closely, $e(D)$ can't be bivalent neither, because this would mean that exists an empty schedule that, inserted before e, leads us from a monovalent state to a bivalent one, and we assumed this is impossible.

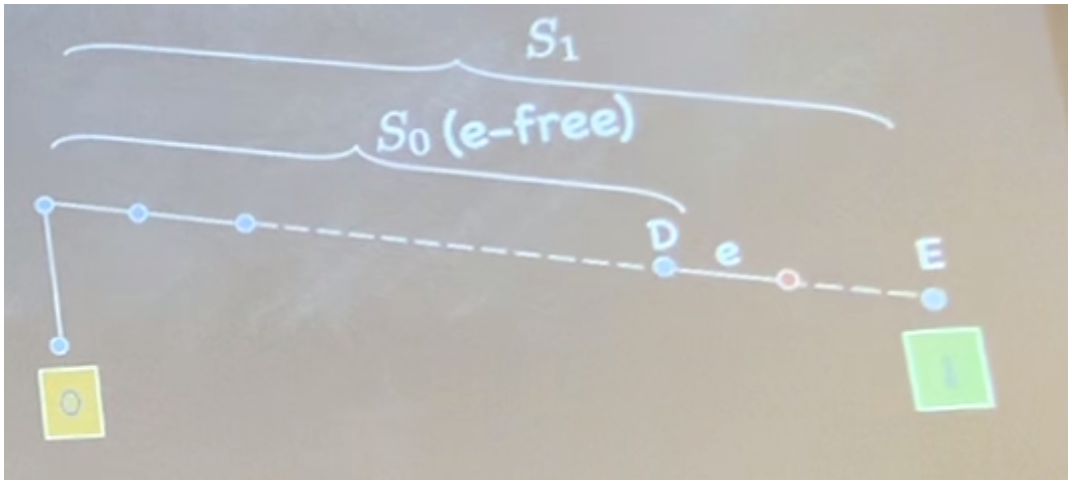

Figure 18: Mini lemma proof.

So $e(D)$ is necessarily 1-valent and thus we found a schedule $D = S_0(C)$ where $e(D)$ is 1-valent and hence we have proved the mini lemma.

Generally is easy to see that every time I apply e from C I have to get to a 1-valent configuration, unless we prove the procrastination lemma that we assumed is false.
We know that applying e straightforward to C lead us to a 0-valent state and that, after using S_0 we can get to a 1-valent state using e. This means that, applying e in different parts of the path from C to D will eventually lead us to get a 1-valent configuration. Thus we can imagine that, in the path defined by S_0, if we stop and apply e we get to a 0-valent configuration and there is a certain point where, applying e, leads us to a 1-valent configuration. Thus we can define A as the last configuration in the S_0 path where, if we apply e, we get a 0-valent configuration; and we can define B as the first configuration in the S_0 path where, if we apply e, we get a 1-valent configuration.

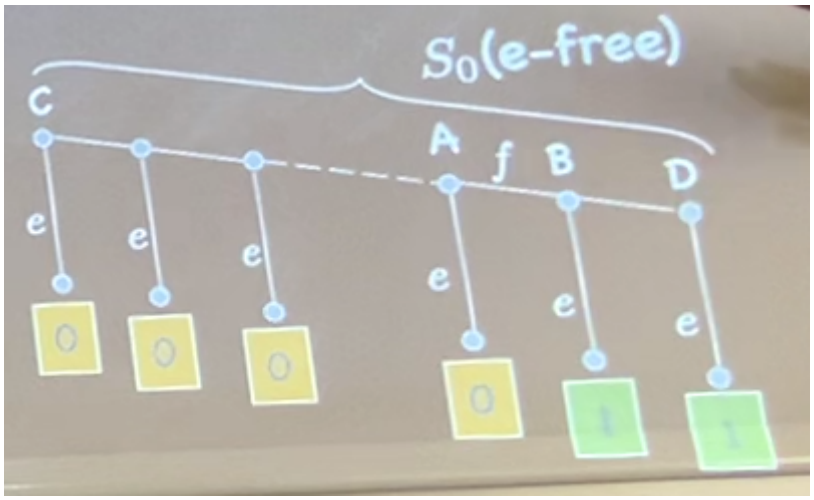

Figure 19: Procrastination lemma proof.

As we can see in the figure, we took a step f from A to B that flipped the behaviour of using e. We're going to state some that's gonna seem a bit weird, i.e. that e and f have to be taken by the same process. Why is that?
Suppose that two different processes take these steps. This means that we can use the commutativity lemma and thus do $f(A) = B$ and then $e(B)$ and move to a 1-valent state, but also apply e before due to commutativity and do $e(A)$ that is 0-valent and then $f(e(A))$ that is 1-valent (as we can see in the figure). But we know that it's impossible to go to a 1-valent state from a 0-valent one.

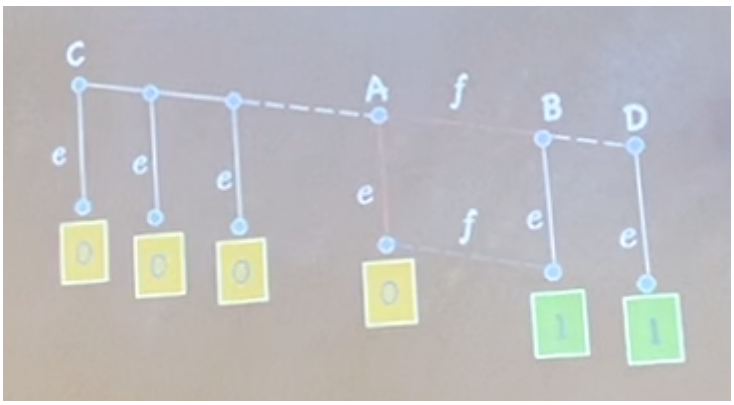

Figure 20: Procrastination lemma proof.

Thus it's mandatory that e and f are taken by the same process, let's say p. p crashes before take those steps. This is the one failure that we can tolerate and thus there is a schedule ρ applicable to A such that:

- $R = \rho(A)$.
- Some process decides in R.
- Process p doesn't take any steps in ρ.

Suddenly, p restores its state, but now I have a schedule that leads me to R, where we decide. We are going to show that, if p is not actually dead, we aren't able to do any choice in R. The decision cannot be 0:

- Consider $e(B) = e(f(A))$, we know this is 1 valent.
- We also know that, in schedule ρ p do no actions because we thought it was dead. So it's applicable to $e(B)$ and the resulting configuration must be 1-valent.
- In the same way we can firstly apply ρ to R and then e and f to get to $\rho(e(f(A)))$ that we know is valent. Thus, we know that R can't be 0-valent.

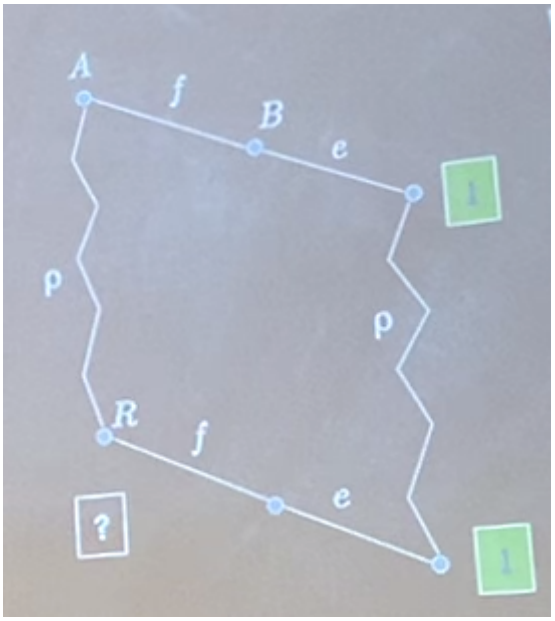

Figure 21: Procrastination lemma proof.

The decision cannot be 1:

- Consider $e(A)$, by construction it's 0-valent.
- Because e it's taken by p and ρ doesn't contain steps from p, ρ is applicable to $e(A)$ and the resulting configuration is 0-valent.
- By the commutativity lemma we can firstly apply ρ to A and then e to $\rho(A)$ getting to the same configuration as before, that it's 0-valent.
- Since $e(\rho(A)) = \rho(e(A)))$ is accessible from R, R can't be 1-valent.

Thus in R we can't make any decision and this is in contradiction with the hypothesis.

6.6 How can one get around FLP?

6.6.1 Weaken termination

We can use randomization to terminate with arbitrarily high probability or guarantee termination only during periods of synchrony.

6.6.2 Weaken agreement

Agreement on real values with a tolerance of ϵ.
k-set agreement:

- Agreement: in any execution there is a subset W of the set of input values, $|W| = k$, such that all decision values are in W.
- Validity: in any execution, any decision value for any process is the input value of some process.

7 Peer-to-peer Systems

Naming a few of them: E-mule, BitTorrent, Napster, ... They have no central authority and are made on a global scale. These systems are also very dynamic, people can connect and disconnect in every moment. They're not so efficient, so they are now outdated.

7.1 File distribution - The BitTorrent idea

7.1.1 Introduction

If you want to distribute a file from a server to million of users, 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**).

7.1.2 BitTorrent actors

There are few actors in the system, one of them is the **tracker**. He has the list of the people that share the file. The file extension is `.torrent` and contains information such as name, size, tracker info. The file is typically divided into parts, so we have also 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*.
Another actor is the seed 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 (and the user is called a seeder). The other ones are called *leechers* (sanguisughe).
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). The problem is the overhead for the few guys that have that part.
- Random.

Actually we combine rarest and random. Initially we use random and then rarest. This 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.
In the system there are actors called **free riders**, people that download but don't upload. How can we recognize them?

7.1.3 Choking algorithms

"To choke" means to stop the upload. In BitTorrent there's a rule of unchoking, every peer decides to unchoke other 4 peers (to not use the bandwidth only for BitTorrent). A user selects the 4 nodes to unchoke and with high probability they'll be peers from which he downloads. If a user is a free rider, there is very low probability that someone unchokes him (decide to upload to him). There's a problem because if a user has just started it's obvious he has low probability to be chosen. There are better algorithms, like *optimistic unchoking*, where there are higher probability to make someone start downloading.

7.1.4 Endgame mode

There are times in which a user can be stuck with the last part of the file and the download is very slow. In this case this last piece of file is split again and the user can download every little piece from every peer in parallel.

8 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. Assume we have a process P and its detector D_p, then $q \in D_p(\sigma, t) \Rightarrow$ process q is believed to be crashed at time t run σ by failure detector D_p. This guess may be wrong.
The D_p module is called **oracle**. We can ask:

- What process crashed in a run σ, $Crashed(\sigma)$.
- What process crashed at time t, $Crashed(\sigma, t)$.
- What process is up in run σ, $Up(\sigma)$.
- What process is up at time t, $Up(\sigma, t)$

We have some properties:

- **Completeness**: we say that a failure detector is complete if when another process crashes, the FD realizes it. To be only complete is pretty easy, if a FD always says that a process is crashed, is complete.
  - **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.
  - **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.
- **Accuracy**: if a FD says that a process is crashed, it should be right.
  - **Strong accuracy**: $\forall\sigma\ \forall t\ \forall p, q \in Up(\sigma, t)$ then $p \notin D_q(\sigma, t)$.
  - **Weak accuracy**: $\forall\sigma\exists p \in Up(\sigma) \mid \forall t\forall q \in Up(\sigma, t)$ then $p \notin D_q(\sigma, t)$.
  - **Eventual strong accuracy**: $\forall\sigma\exists t \mid \forall t^i > t\ \forall p, q \in Up(\sigma)\ p \notin D_q(\sigma, t^i)$.
  - **Eventual weak accuracy**: $\forall\sigma\exists t \mid \forall t^i > t\ \exists p \in Up(\sigma)\forall q \in Up(\sigma, t^i)\ p \notin D_q(\sigma, t^i)$.

If we have a FD with strong accuracy and strong completeness we have a *perfect* FD.
We could make Paxos live with a perfect FD, but this is against FLP, so it's not possible and then perfect FDs not exists in asynchronous systems.

8.1 Taxonomy

	S	W	event. S	event. W
S	P	S	$\diamond P$	$\diamond S$
W	W	Θ	$\diamond W$	$\diamond\Theta$

Where P is the set of perfect FDs, $\diamond P$ the set of eventually perfect FDs, $\diamond S$ are eventually strong, W are the weak ones, $\diamond W$ are eventually weak.

8.2 Simulation

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.

9 Cryptocurrency Market Manipulations

9.1 Libertarianism 2.0

Money should be anonymous, transferable in a peer to peer fashion and digital. There should be no central authority.

9.2 Challenges of digital cash

It's easy to model ownership (private and public key in the Bitcoin case), but it's hard to model transferability (blockchain, consensus protocols, ...). We have theorems like FTP and CAP (impossible to get consistency, availability and partition tolerance together) that sets limits to distributed consensus.

9.3 Pump and dump

It's a market manipulation that works pumping artificially the price of something, selling and then restoring the price to its original value. It's illegal. 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); VIP group members (most win); group members (some win, some lose); external investors (always lose, they don't know that a pump and dump will happen).

9.4 Detecting scam

Can we make a system that detects scams? The idea is that we can observe that during a pump, most of the members of the group put market orders. These are orders that set a limit on the prices of buying and selling. Sensible investors don't use market orders.

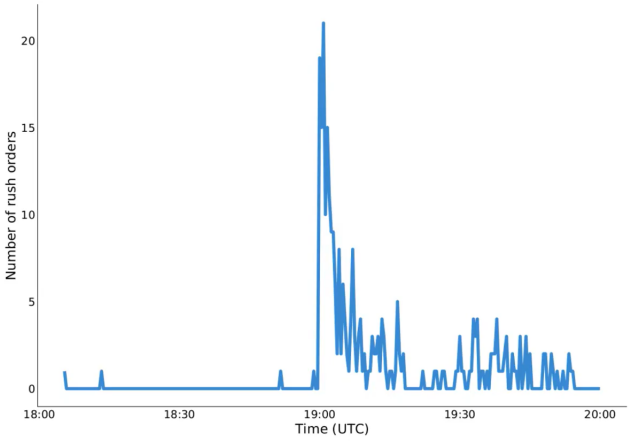

Figure 22: Abnormal amount of market orders.

10 Consistent hashing

10.1 Hash Table

It's a data structure used to store keys in a vector. Starting from a key we take its hash and this represents the location in which we store the information. 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 few ways, like storing the key in the next empty location. Another possibility it's to build a list of keys.

10.2 Distributed hash table

There is a problem for this data structure to be suitable for distributed systems. The location of hash tables are computers that are part of the distributed systems. What if we 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.

10.3 The idea

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 visualize a ring on which we map every number in the space of addresses. If now we take a key, let's say k_1, and we hash it $H(k_1)$, we can map a dot on the circle. Another thing I can do it's take a computer, let's say S_1, and hash it $H(S_1)$ and we represent it like a square on the circle. All the keys that goes from one computer S_i to another computer S_j are stored on S_j.

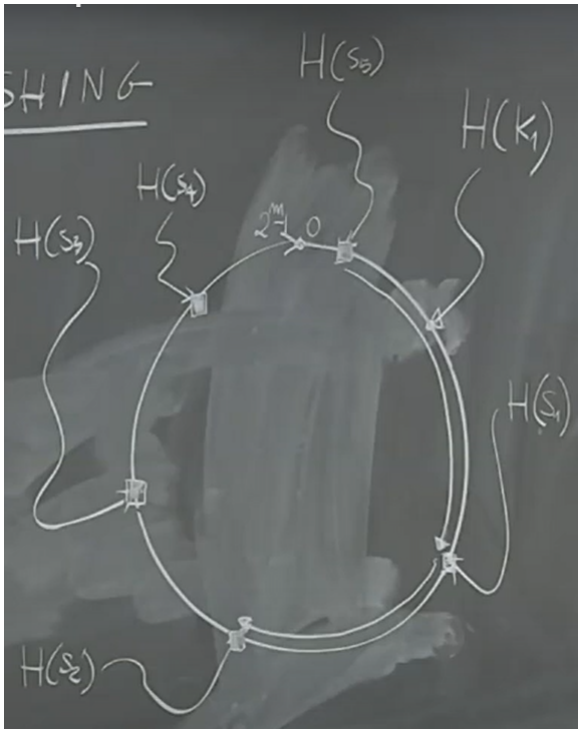

Figure 23: Consistent hashing.

Let's assume that we have a service that, given a point on the ring, says to us what server is

responsible for that point. In this way we can contact the server and give to 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 to it for the value.

10.4 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 to a server if it is responsible of 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 every key k with the property that $H(k) = H(S_j) + 2^i$ for $i = 0, 1, ..., m - 1$. If we are looking for a key we can ask to any of the servers and it will redirect us to the nearest server responsible to that key. The steps, defined by the i parameter, are called fingers.

10.4.1 Adding one server

If we add one server S_x to the system, it will fall between two server, 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. It's the same when leaving, S_x can simply send all its files to S_j (the next server).

11 Content Delivery Networks

It's an area of research about all the techniques that are used to get to the user content as fast as possible.

11.1 DNS

A distributed system to translate addresses from a written form to a numeric one. There is a central server, called **DNS root**, which is responsible to store the primary domains (`.com, .gov, .it, ...`). These domains have their servers that keep addresses of other domains below them.
A user has to set his DNS server and use it to ask addresses mapping. The server firstly ask to the DNS root, then, reading the answer, will ask to the server right below and so on. 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 have maybe to wait a day before it propagates on the system.
The address translation does not depend on where is the user. The Akamai idea is to make it dependant.

11.2 Akamai CDN

Let's assume that the content of `www.cnn.com` is stored in several servers of the world (Rome, Paris, New York, Tokyo, Sidney, ...). A user in Rome that asks 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 is to make the system work, but without changing it so much. This because the user doesn't know the existence of the Akamai server a priori and so he will use the usual DNS

system. When a user asks for a webpage, he's actually asking for an HTML file, which will likely contain other files references.
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. To resolve this new address now we have to ask to the Akamai server and it will redirect the user to the closest IP.
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, we can then replicate the server, use multiple servers, etc...

12 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. Algorand was founded by Silvio Micali.

12.1 The blockchain

Algorand is a programmable blockchain. This means that it supports smart contracts, chunks of code that can be executed on the chain. 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 reduce a lot the problem of double spending.

12.2 Proof of work

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 and high energy usage, resulting in a need of expensive equipment.

12.3 Proof of stake

More money a user has in his wallet and more chance its node has to propose a new block. This results in a faster system, no need of expensive equipment. Besides, a system like this enables **stacking** that is using detained cryptocurrency to earn rewards.

12.4 The protocol

As in Bitcoin we have a network of nodes that communicate in a P2P way. Every time that a node has a message, it shares it with every other nodes 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 protocol can be divided in 3 macro categories:

- **Round**: a complete run of the protocol, it ends with the creation of a new block.
- **Phase**: there are three of them and they divide a round of the protocol.
- **Step**: every time we broadcast a message.

We have also three type of nodes:

- Standard node, do nothing in the network and just listen.
- Proposer node, the user's node that proposes a new block.
- Committee node, a node that agrees on one of the block proposed.

Every node have 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.

12.4.1 Verifiable Random Function

Used in a lot of modern blockchains. The idea 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.

$$\mathrm{VRF}_{pk}(\mathrm{seed}||\mathrm{role}) \rightarrow (\mathrm{Hash}, \mathrm{proof})$$

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

Figure 24: Verifiable Random Function

From now on it's impossible to see the blackboard.

13 Consensus

We have different version of the problem.

13.1 Byzantine agreement

Informally, in this problem one process p has an initial value x, after the run of the protocol everybody agrees on a value. If p is correct, the agreed value must be x, otherwise can be another one. We have three properties:

- **Agreement**: all correct processes must agree on the same value.
- **Validity**: if p is correct, every process must agree on x.
- **Termination**: each correct process must eventually decide.

13.2 Interactive consistency

Every process has an initial value and all the processes must agree on an array of values. The 3 properties are:

- **Agreement**: all correct processes must agree on the same array A.

- **Validity**: if process p_i is correct, then $A[i]$ must be the initial value of p_i.
- **Termination**: each correct process must eventually decide.

13.3 Consensus problem

Every process has an initial value and we want to agree on a single value.

- **Agreement**: all correct processes must agree on the same value.
- **Validity**: if all the initial values are equal to x, then the chosen value must be x.
- **Termination**: each correct process must eventually decide.

What we know is that the problems are actually equivalent.

13.4 Reductions

We use reduction to prove that a problem is easier than another one. If we want to prove, for example, that Byzantine Agreement is easier than Interactive Consistency, we assume that we have a protocol that solves IC and we show that it works also with BA.

13.4.1 BA is easier than IC

We assume that we have a black box algorithm that solves IC. We have also to specify our fault tolerance model, if message are reliable or not, if we can deal with crashes and byzantine behaviours.
Let's then assume that messages are reliable and we have byzantine fault tolerance.
Let's consider that the process p_i has a value x and all the other processes has a special value s. We start an IC protocol and we come up with an array, hence we have to take the one at the i-th index. We can easily see that all the properties are verified:

- Agreement, because everybody has the same array and thus same i-th value.
- Validity, looking at IC validity we know that, if p_i is correct $A[i]$ is x. Thus, choosing $A[i]$ leads us to verify validity.
- Termination is the same, so it's verified.

13.4.2 IC is easier than BA

We now assume that we have a protocol for BA and we have an instance of the IC problem. We can thus run n instances of BA, with n equal to the number of the processes. In this way in every run we have a different initial process that proposes a value and we can build an IC array. We have to notice that runs are concurrent, but this doesn't change the functionality that much. We verify all 3 properties:

- Agreement, two arrays are the same if they have the same element at the same index, for all indexes. Running a working BA protocol makes the processes choose every time the same value, so in every array hold by every process we have the same element at the same index, for all indexes.
- Validity, in the i-th run, we choose p_i initial value, so of course we verify it.
- Termination is the same, so it's verified.

13.4.3 BA is easier than C

We now assume that we have a protocol working for C and an instance of the BA problem. Let's assume that the process p can successfully broadcast its value to every process. Now we can run the protocol for C and get the solution.
We have a problem with the first broadcast because we have to deal with crashes and byzantine behaviours. If p is correct, the message will arrive because we have reliable messages. The problem happens if p is crash faulty or byzantine faulty. However, we can notice that byzantine behaviours are not actually big deals. If p sends everybody a different value, for example, C protocol can run anyway and will choose a value. Be careful, we **DON'T** need to choose the value proposed by p, because p is not correct.
The real problem are crashes. If p crashes obviously it isn't correct, but its message may not arrive and in asynchronous system this is a problem. We'll leave this unanswered for now.

13.4.4 C is easier than BA

We now assume that we have a protocol working for BA and an instance of the C problem. We can run n instances of BA protocol. If there aren't faulty processes, all values are the initial values. If we have faulty processes we have some random values. We can think about choosing the value that has a majority, but faulty processes may invalidate this technique. We'll leave this unanswered for now.

14 Practical Byzantine Fault Tolerance - PBFT

It's the result of a 1999 work about a model called State Machine Replication. The idea is to look at a system like a Finite State Machine, implemented with a set of servers that change their state giving the input commands. The idea is to build a protocol to make this model deal with byzantine behaviour and it's used in systems like blockchains. Thus the usage of PBFT is when we have servers controlled by people of which we can't trust their behaviour.
In order to avoid the possibility that the same exploit works on every server of the system, we have the assumption that these servers runs different software, administrated by different people with different passwords.
In PBFT we are safe all time but no live.

14.1 The protocol

We assume that channels are **not** reliable. We use signatures for messages and represent them with the notation: $< m >_{\sigma_i}$ which means that the message m is signed with i. Signatures are very expensive, in general about one thousand more then hashes, we eliminate them were we can. We can tolerate f faults and we have n processes with $f \leq \lfloor \frac{n-1}{3} \rfloor$. The best number of nodes to tolerate f faulty nodes is then $n = 3f + 1$.
Our system is composed as follows:

- **Client**, C. It's an external actor and it's the one that gives commands to the system.
- **Primary replica**, P. It's the most important replica in the system, but it's not always the same.
- **Replica**, R. There are $n - 1$ of them and with the primary they make it to n nodes or processes.

The client can send a request to P, $\langle REQUEST, o, t, c\rangle_{\sigma_i}$, where o is the operation (what the client wants the system to do), t a timestamp (a way to give some order of the request coming from the **same** client, there's no sense to use it among clients because we don't have a global clock), c is the client. P can now send the request to all the Rs. These messages are called $\langle\langle PRE-PREPARE, v, n, d\rangle_{\sigma_p}\rangle$, where the signature p means that is signed by the primary. v is the view is the period of time in which P is the same, we can change the view if P stops working (there's a protocol for this, similar to PBFT); n is the sequence number of the operation within the view, d is the hash of m $H(m)$ and m is the message without signature. We can be sure that the message it's not changed because we signed the hash.
Generally when we say "I signed something" means that we signed the hash of the message and send it, this is done because the hash it's way smaller.
The replicas accept the message if 4 properties are verified:

- The signature and the hash are correct.
- We are in view v.
- The replica has not accepted any other message with same view number and sequence number.
- $h \leq n \leq H$. The sequence number is an integer and it can only grows. We can't let happen that a byzantine P uses an arbitrary number and so we establish an interval. There's a simple protocol to move this interval over time.

If a replica accept a pre prepare message, it can send a broadcast message $\langle PREPARE, v, n, d, i\rangle_{\sigma_i}$ where i is the replica number and σ_i indicates that it's signed by the i-th replica. We have a property called $prepared(m, v, n, i)$ that is true if and only if replica i has in its log request m, a pre prepare message from m in view v and sequence number n and $2f$ prepare messages from the other replicas. Why $2f$ prepare messages? Because, given two replicas i, j, if we consider the $2f + 1$ sets ($+1$ comes including the replica itself) of prepare messages received by this two it's obvious that we have an intersection (we have $3f + 1$ nodes). Thus we can say that *at least* we have $f + 1$ messages in the intersection. What can we deduce from this? That at least one server in the intersection is correct (we tolerate at most f failures) and it sent the same message to the two nodes that in this way can be sure of what is correct.
When a replica has checked the prepared predicate we can complete the protocol sending a commit message to everybody, $\langle COMMIT, v, n, d, i\rangle_{\sigma_i}$, d is the message digest (i.e its hash).
Now what everybody does is to check another predicate called $committed\_local(m, v, n, i)$ that is true if and only if $prepared(m, v, n, i)$ it's true and replica i has received $2f$ commits. Including the replica i itself, this predicate tells us that i has received *at least* $f + 1$ correct messages, so it's impossible to make another majority and can confirm the operation, sending $\langle REPLY, v, t, c, i, r\rangle_{\sigma_i}$, that is view v, timestamp t (the same timestamp from the request) , for client c, by replica i and with the operation's output r.
When the client receives $f + 1$ of this messages it is sure that it has at least one correct and can be sure that the protocol ended correctly.
This is a message with a very high number of messages, the order of magnitude is n^2. We say that this kind of protocols can't be used with an high number of nodes. For example Algorand chooses the committee group (a small group made by nodes of the blockchain) to run this protocol.

Exercises

Exercise 1

Let C1 and C2 be two consistent cuts. Show that the intersection of C1 and C2 (the set intersection) is consistent.

We have 4 cases:

1. If $C1 = C2$, trivially we have $C1 \cap C2 = C1 = C2$ and we already know both are consistent.
2. If the whole C1 is smaller than C2 (i.e. $\forall e \in C1 \Rightarrow e \in C2$), then trivially $C1 \cap C2 = C1$ and we already know C1 is consistent.
3. If the whole C1 is greater than C2 (i.e. $\forall e \in C2 \Rightarrow e \in C1$), then trivially $C1 \cap C2 = C2$ and we already know C2 is consistent.
4. Eventually, let's assume we have $C1 \cap C2 = C3$ with $C3 \neq C2 \neq C1$ and, ad absurdum, an event $e \in C3 \mid e \rightarrow e^i \wedge e^i \in C1 \wedge e^i \notin C2$. In this case $e^i \notin C3$ and this results in $C3$ being an inconsistent cut, but this is an absurdity because $e \in C3 \Rightarrow e \in C1 \wedge e \in C2$ and knowing that $C2$ is consistent, we know that $e^i \in C2$ that is in opposition with the hypothesis.
   We can thus make the same statement for $e^i \in C2$ simply reversing the hypothesis and demonstrate that C3 is consistent in every case.

Exercise 2

Let C1 and C2 be two consistent cuts. Show that the union of C1 and C2 (the set union) is consistent.

We know, by property, that $\forall e^i \in C1$, if $e \rightarrow e^i \Rightarrow e \in C1$. The same can be said for $C2$. This means that, given $C3 = C1 \cup C2$, $C3$ will still contain every e given $e \rightarrow e^i$ thanks to the structure of the union operation (we can't lose information by performing it). So, trivially, if $C1$ is consistent, the same elements will be in $C3$ and the same can be said for $C2$ and, for that, $C3$ is consistent.

Exercise 3

Show that every consistent global state (consistent cut) can be reached by a consistent run.

We start by making an observation, $\rightarrow$ is acyclic. We assume that we are building a run R and we want to extend it to reach a cut C in a way that R is still consistent. We take a random event in C
R and we start iterate to take the first event that happens in C
R and this event must exists because $\rightarrow$ is not cyclic. Taking this event we can extend R without making it inconsistent. Since we have a finite number of event in our system, at some point we have to meet the cut C.

Exercise 4

Tag with proper Lamport and vector clock this distributed computation.

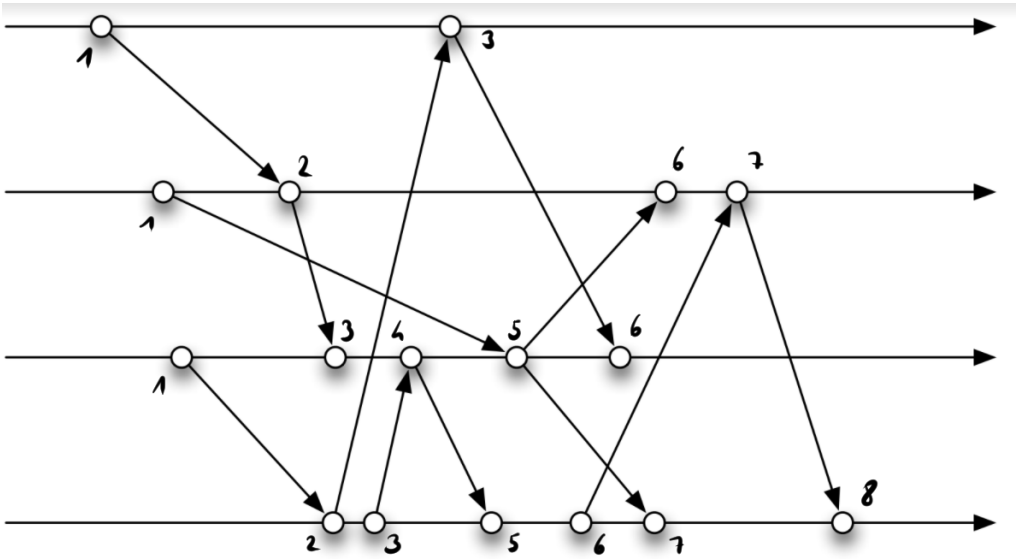

Figure 25: Exercise 4.1.

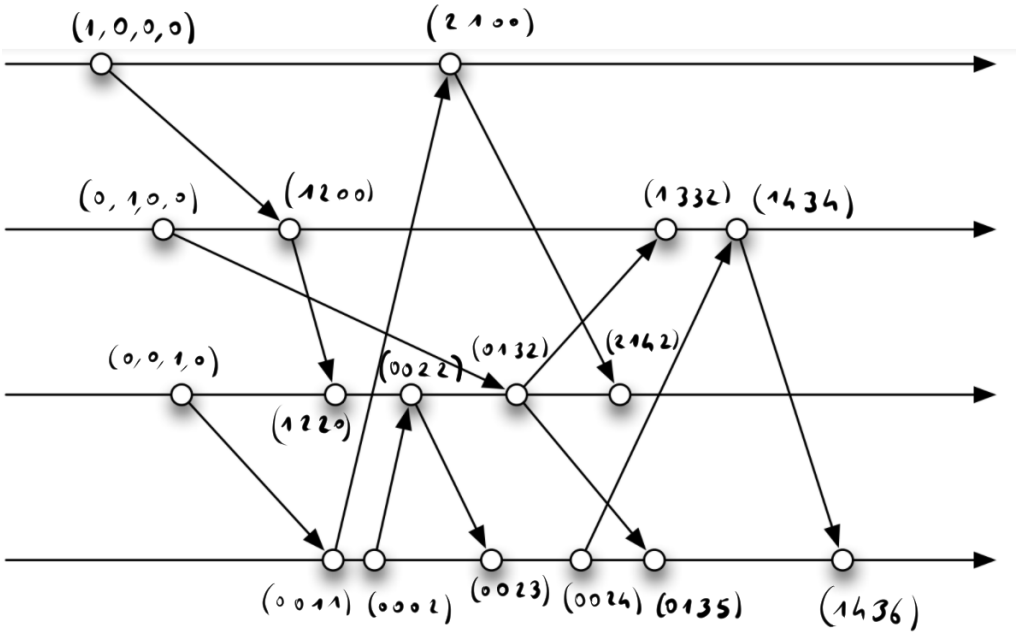

Figure 26: Exercise 4.2.

Exercise 5

Give an Atomic Commit Protocol that satisfies the converse of condition AC3. That is, if all processes vote Yes, then the decision must be commit. Why is it not a good idea?

The converse of AC3 condition is "If any process votes Yes, then the decision must be commit". This property can lead to an inconsistent state of the system, because we don't know if a process crashed (and didn't vote at all), thus committing with only one "yes" vote makes the system having different copies of data in different nodes.

Exercise 6

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

In 2PC with cooperative termination we know that participants can send a broadcast message when the coordinator stops working. This is done to mitigate the stops of the protocol. We have to remember that, being in an asynchronous system, we can't be sure of when the broadcast messages arrive. Thus, even if only one of them crashes, we can't know what voted (so the participants can't commit) and we can't distinguish the crash from a very slow process, hence we're blocked.

Exercise 7

Show that Paxos is not live.

By definition, the protocol can have more than one proposer active concurrently. Let's assume that a proposer p sends a prepare message to the acceptors, they correctly answer with a promise but immediately before receiving the accept, they receive another prepare message, from another proposer and with a round greater than the first one. They now have to promise not to vote for any other previous round. Looping through this kind of behaviour makes Paxos stuck forever.

Exercise 8

Show that in Paxos, if we remove the promise that the acceptor does not participate in previous rounds, Paxos is not safe any more.

We know that we can run Paxos on asynchronous systems with more instances of it running concurrently. This means that we can be in the situation where an acceptor A votes value x in the round j, but this message is yet to arrive to a learner $L1$. In round $j+1$ a proposer propose value $y \neq x$ and the acceptor A accepts it, sending this value to a learner $L2$. This means that we can have $2n+1$ acceptors with the same behaviour and thus we have a majority that votes x in the round j and a majority that votes y in the round $j+1$. But we know that Paxos is safe if and only if an acceptor a voted value v in round i and no value $v^i \neq v$ can get a majority in any previous round, and so we can't learn more than one value per Paxos instance.

Exercise 9

Show that Fast Paxos is not safe if we assume the same number of crash failures of Paxos.

We know that in Fast Paxos we are able to send an accept-any message if we have no value to propose. This means that, if X nodes are triggered at the same time, we can have X set of acceptors that votes for different values. It's simple to understand a limit case. Suppose we have a set A of $\frac{n}{2}+1$ acceptors (Paxos majority) divided in two set P Q that voted respectively for v and w. If all other processes crashed it's safe to say that we don't know what to vote, because we can't know if they voted v or w or didn't vote at all and so we don't get a majority. Thus we need $3f+1$ processes to tolerate f failures that is the same to say the quorum must be $\frac{2n}{3}+1$.

Exercise 10

Show the six reductions.

IC easier than C:
We can solve C, so we have n processes P with n initial values and one value is chosen, let's say this value belongs to process p. Every process logs the value in this way:
If process p has index i, we store the chosen value at the i-th index of an array replacing a placeholder value. No processes can change their initial value and then we restart C protocol pretending that p is crashed. So we run C between $P \setminus p$ and we choose another value, sending a message to all unused processes (only p so far). In this way every process can log the new value and we continue. At the end of the protocol we have an array of values stored in the log of every process where the process x_j has its initial value in the array at index j. Every correct process must have the same array by construction so we verify Agreement of IC. For the same reason, i.e. by construction, we verify validity. Termination is the same.

C easier than IC:
We can solve IC, so we have n processes P with n initial values and an array A of values is chosen. We can decide a number i and make every process choose $A[i]$. Every process knows A so every correct one will agree on $A[i]$ an we verify C agreement. If every process has x as initial value, $A[i]$ is for sure x, so we verify validity. Termination is the same.

C easier than BA:
We can prove that we can't solve C if we have a number of failures $f \geq \frac{n}{2}$ if n is the number of processes. Let's say we have 4 processes, p q r t. p and q are correct and r and t are byzantine. Every process has x as its initial value. We have an algorithm for BA so we run it with process p, that is correct. By properties p q r t will agree on x. The same happens after we run the algorithm for process q. Now we can run BA for process r, that is byzantine. It can broadcast a value $y \neq x$ in such a way that every other process can't distinguish its behaviour from the correct one. So BA will run accordingly and every process will agree on y. The same happens for process t.
Now we have an array of values $[x, x, y, y]$ and two problems: - We don't have a majority, so we can't choose neither of them. - If we eventually decide, by termination property, we may invalidate the validity of consensus, that we are trying to solve using BA. This because every process has x as its initial value but we're not going to choose it for sure. Given this assumption, we say that we can tolerate $\frac{n}{2}-1$ crashes or failures, applying the same protocol written before. We run n instances of BA and in every of them we will choose a value. Now we have an array and we can take the value that has a majority. Actually, if the initial values for every run were different we can choose whatever value we want and verify C agreement and C validity. If all initial values were the same, let's say x, we will for sure agree on x, because even if we have

$\frac{n}{2} - 1$ byzantine processes that send a different value we will have a majority of x in the final array.

BA easier than C:
We can prove that no byzantine agreement protocol works in asynchronous systems with just a single failure: Suppose that the protocol works and the following scenario: The process p with the initial value x is faulty and all other processes are correct. p doesn't send any messages during the protocol and, by the termination property, the other processes must agree on some value, let's say y at time t. The process p it's now correct and sends its messages, according to the protocol, but none of them arrive before t. Thus, all other processes agree and decide on y at time t, violating the validity (if p is correct we have to choose its initial value). This proof means that we can solve BA using the C algorithm only if we have no failures. Every process starts with an initial value and we can choose one of the process to let him broadcast its value, let us say x, to all other processes. Now every process has the same initial value and, by C validity, the processes will agree on x. Thus, we verify BA agreement because all correct processes agree on the same value; and validity, if p is correct every process must agree on x. Termination is the same so it's verified.