“Blockchain and Distributed Ledger Technologies” Notes

Prof. Claudio Di Ciccio. Notes written by Alessio Lucciola during the a.y. 2023/2024.
You are free to:

- Share: Copy and redistribute the material in any medium or format.
- Adapt: Remix, transform, and build upon the material.

Under the following terms:

- Attribution: You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- Non Commercial: You may not use the material for commercial purposes.

Notes may contain errors or typos. If you see one, you can contact me using the links in the Github page. If you find this helpful you might consider buying me a coffee 😄.

1. Introduction

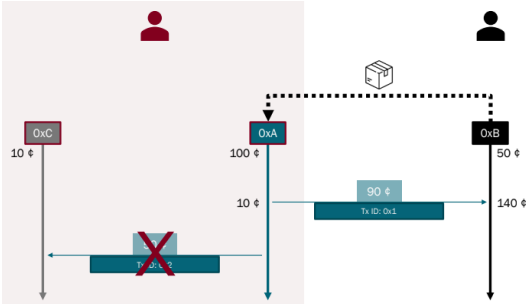
“**Blockchain** is an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way”. This definition was given in a paper of 2017 by M.Iansiti and K.R.Lakhani.
One could think that the **blockchain is a database but it is actually not** because it doesn't store data but records the transactions that lead the data from a point to another point, not the data itself. There are some building blocks that make up the blockchain:

- **Transactions are immutable** (there are some exceptions);
- A **copy** of the blockchain is accessible to every node on the network and it offers access to the history of all previous states;
- **Consensus** is achieved through dedicated algorithms;
- It also offers the possibility of executing user-defined scripts (the so-called **smart contracts**).

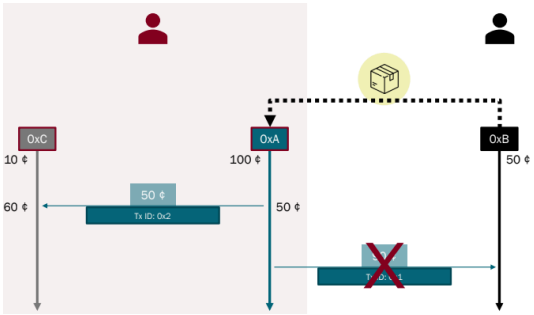
We’ll see all these concepts throughout the course.

Again, the **transactions** are the building block of Blockchain. Transactions are how data is added to the blockchain, and they typically involve the transfer of digital (crypto)assets, information, or the execution of smart contracts. A transaction usually involves two accounts A and B where an account is identified by a number. Inside of a transaction we can find some information such as the sender's address, recipient's address, the amount being transferred (in the case of cryptocurrencies), a digital signature and additional metadata. We need to be sure that not everyone can send transactions from a specific account and in order to do so we can use **digital signatures** thanks to which we can prove that the person who sends the transaction is authorized to do so. Notice that an account is not linked to a specific physical person (like a bank account) but a person is simply authorized to make a transaction with that specific account.
Note: Notice that the account is labeled with EOA. EOA (Externally Owned Account) means that the account it’s in the hands of a user that isn’t inside of the blockchain (it can be a person or a bot). There are other special types of accounts like smart contract accounts that don’t belong to anyone and are only controllable by smart contracts, and so they are not in the hands of a user.

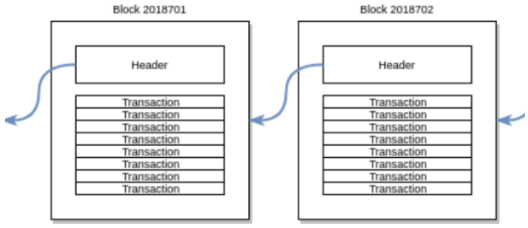
A **ledger** is a record-keeping system used to track transactions and maintain a chronological and organized history of those transactions. So, the ledger is **strictly totally ordered** and each transaction is unique and identified by its id.
These properties are mostly necessary in order to avoid the problem of **double spending**. It occurs when a person spends the same digital currency unit more than once, essentially creating a duplicate or counterfeit copy of the currency. Let’s assume we have two users R and B where R owns two accounts 0xC and 0xA and B owns 0xB. R uses account 0xA to send 90¢ to B and due to this transaction B



decides to send some stuff to R (a service is delivered to R). Notice that 0xA now has 10¢ while 0xB has 140¢. Now, R makes another transaction from 0xA to 0xC of 50¢ but this transaction will be rejected since R doesn't have enough coins in 0xA. A strange situation might happen: if R convinces the network that the second transaction was made before the first one then the situation changes because R would transfer the coins to himself and the transaction to B would be rejected. So R now has the same coins as before (even if in different accounts), while B has been robbed (the stuff that he sent already arrived to R!).



To be efficient, the ledger is divided into **blocks** that serve as a container for a set of transactions. The fact that transactions are processed in blocks is mostly because of efficiency: it is computationally heavy to compute each transaction one at the time, as well as all in the end. Each block has a **timestamp** that records the exact time at which the block was created or added to the blockchain. This timestamp is crucial for maintaining the chronological order of transactions. Each transaction in a block has to happen after each transaction in the previous block. Note that not each block has to have the same number of transactions and even an empty block is a block. One of the most important features of a block is its reference to the previous block in the blockchain. This reference is achieved through a **header** that is a unique identifier called the "**block hash**" or "previous block hash". It links the blocks together, creating a **chain**.



Blockchain is **different from a simple client-server** architecture because the blockchain is not an external service that the client invokes, but the node itself is the platform once it's part of the protocol. In a blockchain network, there are multiple participants or nodes, often referred to as peers. Each peer has the same role and can perform functions such as validating transactions, creating new blocks, and maintaining a copy of the entire blockchain. Peers communicate directly with each other, forming a network without a central server.
Another property is the **redundancy**: Each peer in a blockchain network maintains a copy of the entire blockchain and this ensures that if one node goes down or is compromised, the network can continue to function, and the data remains accessible.
Blockchains are typically **decentralized**, meaning there is no central authority or server controlling the entire network. Transactions are validated and recorded through a distributed consensus mechanism involving multiple nodes. This decentralization enhances security, resilience, and transparency.
Why not use centralized ledgers?
Some problems could happen:

- It could be lost or **destroyed**: A user must trust that the owner is properly backing up the system;
- It could containing **invalid transactions**: A user must trust that the owner is validating each received transaction;
- It could be **incomplete**: A user must trust that the owner is including all valid transactions that have
- been received;
- It could be **altered**: A user must trust that the owner is not altering past transactions.

So, is the decentralization problems-free?
No! Distributing the ledger makes for **permanence** but entails **no notion of a unique distributed clock**. Also:

- **Scalability**: One of the major challenges with decentralized systems is scalability. As the number of participants and transactions on a network grows, the system may struggle to handle the increased load efficiently. This is a common issue in many blockchain networks, resulting in slow transaction processing times and high fees.
- **Consensus Mechanisms**: Different consensus mechanisms have their own challenges. For example, Proof of Work (PoW) is **energy-intensive** in particular, checking that a solution is valid is easy but **solving the puzzle is difficult**. Proof of Stake (PoS) has potential issues related to centralization of stake and token distribution.

- **Lack of Reversibility**: The immutability of blockchain data means that errors or fraudulent transactions are not easily reversible. This can be a drawback in cases where dispute resolution is necessary.

Smart contracts are self-executing contracts with the terms of the agreement directly written into code (so it is just code, no more than that)and it is **deterministic** (there is no randomness). The **code executes only when called** so there is no listener. An entity that lives off the chain can invoke a function inside the chain, but a contract on the chain cannot invoke anything outside. They have direct control over their own balance and key/value storage.

Where are smart contracts executed?

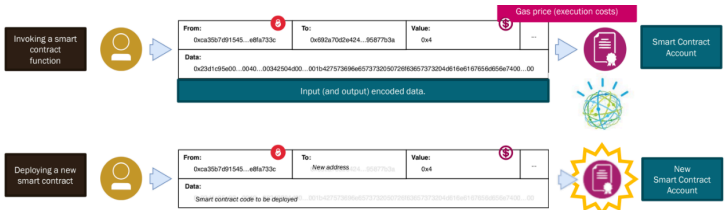
First they are executed on the mining nodes and then, since we have to take into account the distributed nature of blockchain, they are possibly executed on every node on the network. Note that only absolutely needed instructions should be in the code.

The **Ethereum Virtual Machine (EVM)** is a key component of the Ethereum blockchain platform. It is a decentralized, virtual machine that executes code written in smart contracts on the Ethereum network. So it can be thought of as an emulation of a single, global computer. The EVM is responsible for running and processing transactions, executing smart contracts, and maintaining the state of the Ethereum blockchain. There is a cost associated with the running of programs on the EVM and in general, the more the operation is computationally expensive, the more it costs. In particular, to prevent abuse and ensure the responsible use of computational resources, Ethereum uses a fee system known as "gas." **Gas** represents the cost of computation, and users must pay for the execution of smart contracts and transactions. Gas limits are imposed to prevent infinite loops and resource exhaustion. Notice that in order to do all these operations a **Ethereum wallet** is needed. This wallet will be used to pay for the deployment costs, including the gas fees associated with sending the contract bytecode to the network.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.

How is a smart contract deployed in the network?

If everything happens via transaction, also the deployment of a new smart contract happens via transaction. This transaction specifies the **contract bytecode** as data and includes the necessary gas and gas price for deployment. A new **smart contract account** (different from EOA because not actually used by a physical entity) is created, which has full control over the new smart contract. The smart account is deployed via a transaction to a non-existing **Ethereum address** (a random number that it has not picked yet). With the smart contract deployed,users (both the owner and others) can interact with it by sending transactions to its address. These transactions invoke the functions and logic defined in the contract, enabling it to carry out its intended actions.



Where are transactions sent when we interact with the blockchain infrastructure?

Since the infrastructure is a distributed system, there is a network of nodes. Transactions can be done by everyone, but in order to make it valuable we need to broadcast it to the entire network, in order for each node to validate it and broadcast it too. So a transaction is propagated to literally every node in the network.

How many transactions do I need in order for each node in the network to run the same method of a smart contract?

I need just a single transaction, because the protocol will propagate the transaction and so each node will execute the method. Since it's a distributed environment, what a node does, the entire network does too. Again, the blockchain is not a database and therefore it doesn't store the state of every single account but the transactions only.

Tokens are digital representations of value or assets that **are managed by smart contracts**. These digital tokens are typically created and managed on blockchain platforms and serve a wide range of purposes. It's important not to confuse the tokens with a cryptographic fuel: Tokens can be associated with real world entities, or have some economic value. A typical example of tokens before the blockchain are fidelity program points so the more I spend, the more points I get that I can spend on their shop.

Tokens are convertible to FIAT (real money) only if they are spread on a large enough network (es. in monopoly, you can spend the money inside the group, but outside they are useless). So the larger the network, the more the people that are willing to accept payments with that specific token.
Notice that there are **Private** and **Public blockchains**. In public everyone can become a member of the community, meanwhile in the private only a select few people (agents) can be part of the network. There are also **Permissionless** and **Permissioned consensus** within a certain type of blockchain. In the first, everyone can participate to consensus, in the latter only a selected group of nodes can (they need permission).

		Transactability / visibility	
		Private	Public
Consensus	Permissionless	**Selected** nodes can transact and view, **all** nodes can participate in consensus	**Every** node can transact and view, participate in consensus
	Permissioned	**Selected** nodes can transact and view, or participate in consensus	**Every** node can transact and view, **selected** nodes participate in consensus

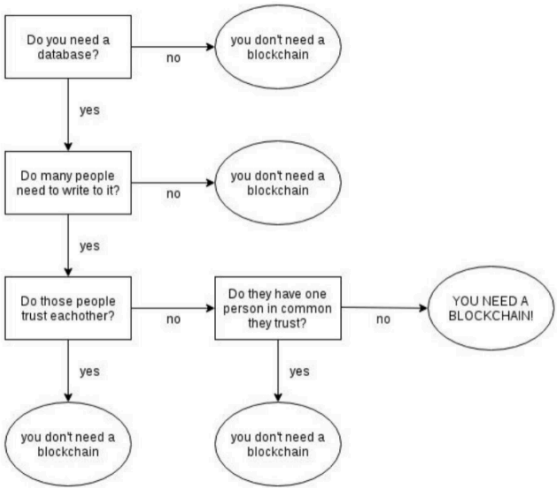
Note: Do not confuse private with permissioned. A network can be private and even permissionless and a public network could still have permissions as shown in the graph above.

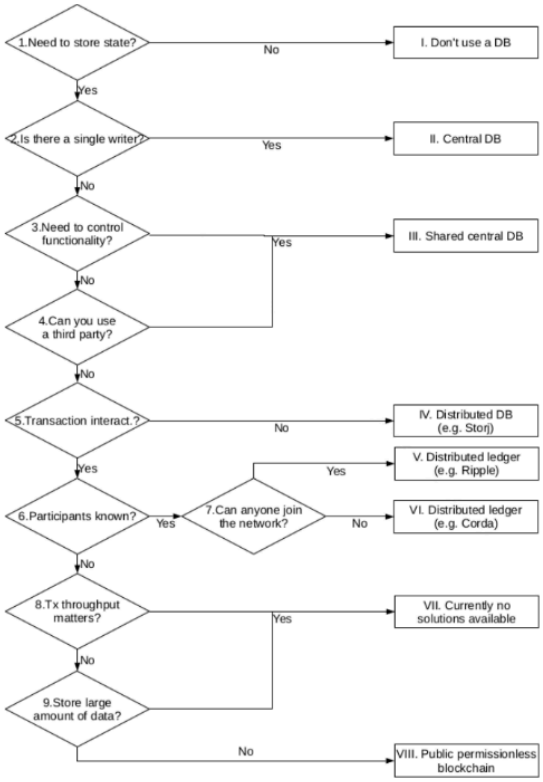
Another question that one could ask is: is the blockchain on the internet?
It is a worldwide distributed architecture that propagates information through the internet so the answer is yes. There are not the same things and the blockchain can't be considered "the new internet". The blockchain simply leverages the internet to propagate packets through the network.

The are actually different generations of the world wide web:

- In the Web 1 there were mostly thin clients, meaning all the heavy computation was done on the server;
- In the Web 2 the clients are more powerful and so some high computation is done locally, like the decoding of live videos;
- In the Web 3 the frontend is actually the same, but the backend can include smart contracts, and so code that runs on the blockchain. It's not an alternative to the web, it's just another paradigm which is decentralized contrarily to the client-server paradigm.

Do we necessarily need a blockchain to solve a certain task?
Blockchain at first went so viral that everyone wanted to use it for everything but blockchain is not always needed (just like the AI hype). AI on the other hand are more generalizable, while blockchain are way more specific and not suitable for certain purposes.

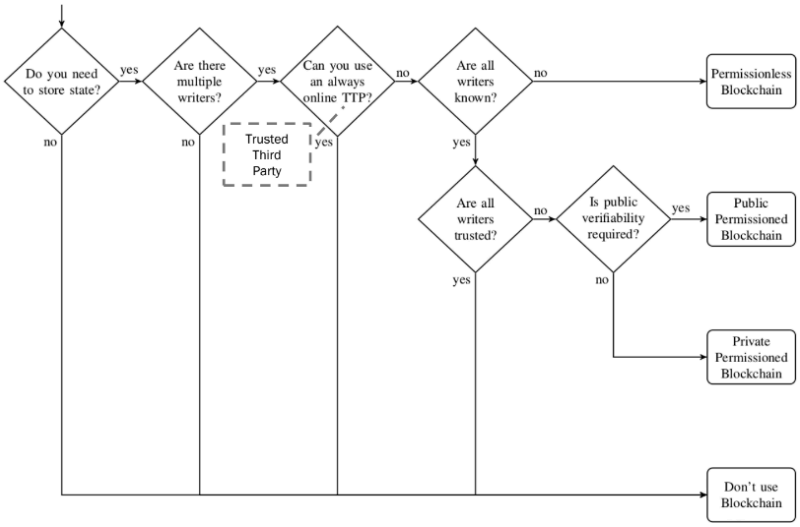




Examples of blockchain applications:

- Steem: Social media in which you can get tokens with likes;
- Open music initiative: Decentralized copyright instead of some centralized departments;
- Forestcoin: The more plants you get care of, the more coins you get;
- And many more..

There are various diagrams that can be used to know if a blockchain is required or not (see images on the right).



2. Preliminaries

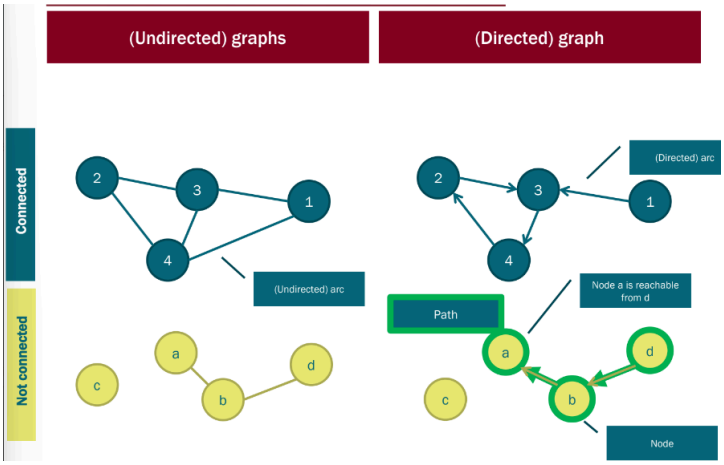
Let's make a summary of some things that will be necessary to understand the course.

Let's start by briefly explaining graphs and trees. **Graphs** are structures consisting of:

1. Nodes;
2. Arcs connecting nodes

If arcs have directions then we talk about **directed graphs** (otherwise **undirected graphs**). Two nodes are **adjacent** if at least an arc exists that is incident on them. A node is a **successor** of another node if an arc exists that is directed from the former to the latter. Now, some useful concepts:

- An undirected walk is an alternating sequence of nodes and arcs, such that every node is adjacent to the next;
- A walk is an undirected walk such that every node is a successor of the preceding one;
- A path is a walk such that arcs are never traversed twice;
- A cycle is a path such that at least an arc is traversed, and the first and the last node coincide.



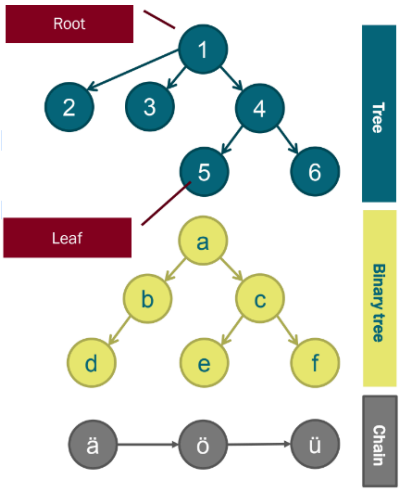
If a path exists that leads from a node to another, the latter is **reachable** from the former (note that a node is always reachable from itself). If, for every pair of nodes, one node is reachable from the other, the graph is **strongly connected**. If, for every pair of nodes, an undirected walk exists from one node to the other, the graph is **weakly connected** (or **network**). A graph that is neither strongly or weakly connected is **disconnected**.

A (directed rooted out-) **tree** is a connected graph such that:

- There exists **one and only one node** from which every node is reachable (**root**);
- There exists one and only one path from the root to every other node (no loops, no overlapping paths);
- Nodes from which no other node is reachable are named **leaves**.

A **binary tree** is a tree in which the root is connected to at most two nodes, and every other node is connected to
at most three other nodes.

A **chain** is a tree in which there exists **one and only one leaf**.

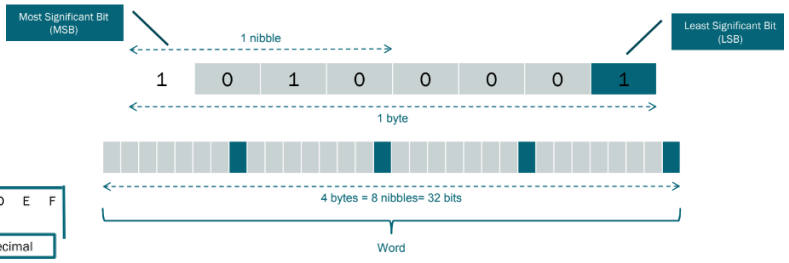


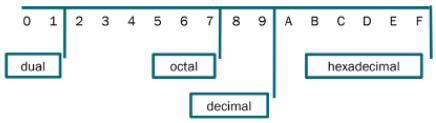
Let's also talk briefly about numeric representations. A **binary digit** (or bit), is an element in a binary set of symbols. We use 0 and 1 and symbols. A **nibble** is a sequence of 4 bits. A **byte** is a sequence of 8 bits.
Given a word W, the general formula to calculate the total value W:

$$W = \sum_{i=0}^{n-1} b_i \times B^i$$

Where B is the base, n is the number of digits, b_i is the value of the i-th digit (nominal value) and B^i is position weight.
Some example of notations are:

- Decimal system (base 10) -> 161;
- Dual system (base 2) -> 10100001;
- Octal system (base 8) -> 241;
- Hexadecimal system (base 16) -> A1 (or 0xA1).





What's the maximal value that can be expressed with a byte?

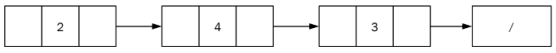
Word length	Min. value	Max. value
8 bits (1 byte)	0	$255 = 2^8 - 1$
16 bits (2 bytes)	0	$65{,}535 = 2^{16} - 1$
32 bits (4 bytes)	0	$4{,}294{,}967{,}295 = 2^{32} - 1$
64 bits (8 bytes)	0	$18{,}446{,}744{,}073{,}709{,}551{,}615 = 2^{64} - 1$
...	0	...
256 bits (32 bytes)	0	$2^{256} - 1$

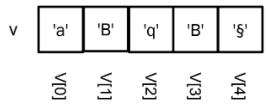
115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,913,129,639,935

Note: The programming language Solidity v0.8.9 supports unsigned integers from 8 bits (uint8) to 256 bits (uint256) at steps of 8 bits.

Considering **hashes**, let's introduce lists.

- **Chains** are often represented as (singly) linked lists, where every node consists of a content and a pointer to the next node. Notice that the length of the list is variable and new elements can be appended to the last one. One can navigate through the list starting from the first element and transitioning onto the next one.
- **Arrays** are positional data structures of fixed length where elements can be accessed by directly pointing at the indexed element.



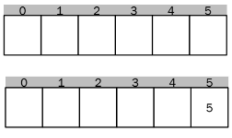


A **hash table** is realized as an array t[0], ..., t[m − 1] of size m and it can store m entries associated with m addresses, in the range [0, ... , m - 1]. By utilizing a hash function h(·), an element k can be associated to its (hash-)address in the table: t[h(k)] (where typically k stands for "key"). At best, a hash function should address elements in the range [0, ... , m - 1] in an **equally distributed manner**.
EX. A typical example is represented by the division remainder function: h(k) = k mod m.

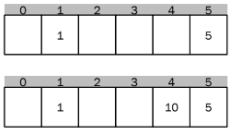
EX. Hashing example: Elements 5, 1, 10 and 11 should be inserted in a hash table of size 6.

$$m = 6$$
$$h(k) = k \bmod m$$

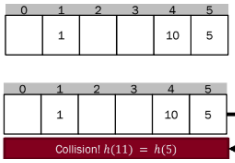
The address of 5 is $h(5) = 5 \; mod \; 6 = 5$



The address of 1 is $h(1) = 1 \; mod \; 6 = 1$



The address of 10 is $h(10) = 10 \; mod \; 6 = 4$



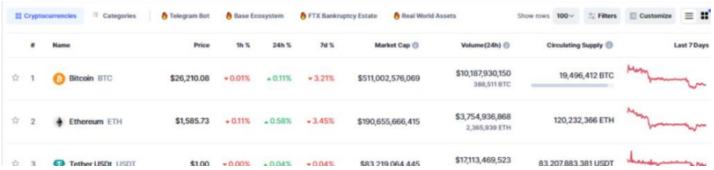
The address of 11 is $h(11) = 11 \; mod \; 6 = 5$

3. Overview Of Bitcoins

We can view **Blockchain as a protocol** where a **protocol** is a system of rules that describes how a computer can connect to, participate in and transmit information over a system or network and can also involve **hardware**, **software** and **plain-language instructions**. As a consequence, the blockchain is not a software, a computer, or a software system and this is why different blockchains exist. There could be different blockchains implementations of the same blockchain protocol and Ethereum is a type of blockchain protocol. Given a certain system, we call the systems in the network as **nodes**.

Let's now give some history. The idea of creating anonymous e-cash protocols was born in the 1980/90s. The reason to introduce such electronic cash was that the money only depended (actually even today) on a **third party** for its values. As many inflationary and hyperinflationary episodes during the 20th century demonstrated, this is not an ideal state of affairs. In 2005, N.Szabo had the idea to use a **one-way function** that was **prohibitively difficult to compute backward** (yet the output should be easy to compute) to compute a string of bits from a string of challenge bits. This was the first idea of proof of work.
The initial paper describing the Bitcoin electronic cash solution was called "Bitcoin: A Peer to Peer Electronic Cash System" published pseudonymously by **Satoshi Nakamoto** in 2008. This paper combined previous primitives such as managing ownership through **public key cryptography** and keeping track of who owns coins through a **consensus algorithm** based on "**proof of work**".

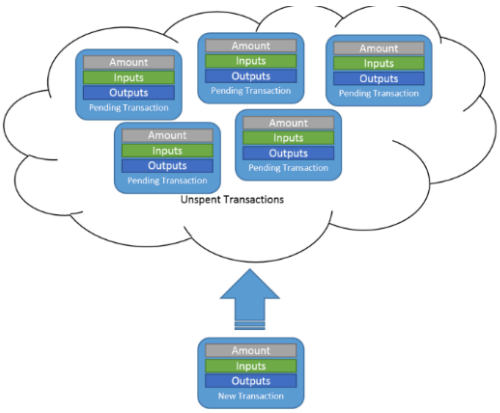
Starting from Bitcoin, the developer community proposed enhancements to the protocol and new types of coins (**alt-coins**) each one with its own blockchain network. Some examples are Ethereum, Tether USD, BNB, XRP, USDC, Cardano, Dogecoin, etc..



Now, let's go more into details with respect to what we saw during the introduction.

A **transaction** is a recording of a **transfer of assets** among parties and is identified by a **transaction ID**. In the blockchain, parties own accounts. Assets are transferred from one (or more, in Bitcoin) input account(s) to one (or more, in Bitcoin) output accounts(s). So notice that in Bitcoin a transaction we can include several inputs and outputs.

	Input	Output	Amount	Total
Transaction ID: 0xa1b2c3	Account A	Account B	0.0321	
		Account C	2.5000	
				2.5321



A **ledger** is a **collation of transactions**. A **collation** means that you keep the order by collecting them, it's different from collection (Like the "collate" in the print options). Users submit **candidate transactions** to the ledger by sending these transactions to some nodes in the blockchain. Submitted transactions are propagated to the other nodes in the network but this does not mean that the transaction is eventually included in the blockchain. The distributed transactions wait in a queue (or **transaction pool**) until they are added to the blockchain by a **mining node**. Mining nodes are the subset of nodes that maintain the blockchain by publishing new **blocks**. This means that transactions are added to the blockchain when a mining node publishes a block. As we said before, a block **collates** transactions that means that they are stored and sorted in blocks.
Validity is ensured by checking that

- The providers of funds (input accounts) have signed the transaction;
- The providers of funds (input accounts) have sufficient funds on their account;
- The block timestamp is greater than the median timestamp of previous 11 blocks, and less than the network-adjusted time + 2 hours.
- And other conditions..

The other mining nodes **will not accept a block** if it contains any **invalid transactions** and this is done in order to avoid injecting invalid (unworthy) transactions in the network.

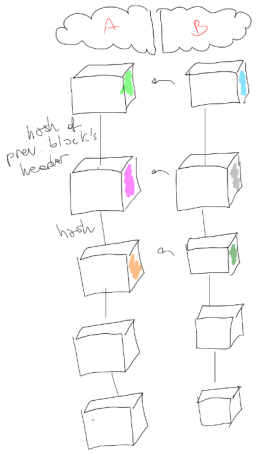
Remember that the problem with distributed systems is that **we do not have a global clock**. We could use the timestamp, but someone can always falsify it, so it's not reliable. In general, when a conflict happens it is not due to errors in the network but it is just a matter of **propagation of the transactions in the network**.
So, in order to understand how a block can be considered valid, we have to first introduce the so-called **Cap Theorem**. The CAP theorem states that out of the three following guarantees, **only two can be guaranteed** by a distributed system:

- **Consistency (C)**: It means that all nodes in a distributed system see the same data at the same time. When a new piece of data is written to the system, it is immediately available to all nodes, ensuring a consistent view of the data;
- **Availability (A)**: Every request to the system receives a response, without guaranteeing that it's the most up-to-date data. In a highly available system, even if some nodes are experiencing issues or network partitions, the system can continue to respond to requests;
- **Partition Tolerance (P)**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

The blockchain gives up on **consistency**. In blockchain systems, achieving strong consistency (ensuring all nodes see the same data at the same time) can be challenging, especially in highly decentralized networks. Since the **blockchain is eventually consistent**, meaning eventually there will be consistency, since after some blocks the other branches will not be valid anymore, and so we will find ourselves in the valid branch (In blockchain the "eventually" means 6 blocks, when the consensus is reached with a probability near 100%. In ethereum the number is about 12 blocks).
Prioritizing availability and partition tolerance is often a common choice to ensure the blockchain remains operational even during network partitions. The blockchain works in a way that you can destroy the whole network, but if there is still a node, this single node can restore the whole network, replicating the information to the other nodes in the network and the blockchain will still be operative (for this reason it's **partition tolerance**). Of course if we remove all the nodes, we will lose the information. **Availability** is also there, because if I want to know something about a transaction, I can ask myself because we store the replica of the current blockchain (according to us), and so we will always have an answer, even if it's not the most updated one.

Now, how can we deal with dropped transactions/blocks?
New transaction broadcasts **do not necessarily need to reach all nodes**. As long as they reach many nodes, they will get into a block before long. **Block broadcasts are also tolerant of dropped messages**. If a node does not receive a block, it will request it when it receives the next block and realizes it missed one.

We check that a block comes after another because it has **the hash of the previous block's header**. Maybe the network can disagree on the latest block, but it's very rare that the network disagrees on the blocks before that. When a node receives a block whose hash doesn't match the block before, it asks the network for the previous block, until it reaches a block whose hash is coherent with the block before, and they can realize that **until then the history is the same**. So, if consensus works (and it does) we backpropagate until we find the same history.
EX. In the example on the right, notice how the latest blocks (on the top) have different colors. One asks the one in the other branch what is its ancestor and finds that they are different (one is purple, one is gray). We continue like that until we notice that the forth ancestors are in common (it's a white block for both blocks).



The hash of a block is not just a fingerprint for that single block in isolation. It is rather a fingerprint of the whole blockchain till then.

So, **the local view is our version of the blockchain**, but the **global view gives us the blockchain tree**, which **contain some branches that will eventually be discarded**, and a **branch that eventually will become the main branch**. By default, the longest chain is the one that is accepted (not really the longest, but we will see more in the future). This works in the

case of proof of work. **If we're in the non valid branch, continuing mining has no sense** since we are just losing time, while the other miners are mining the blocks for the accepted branch.

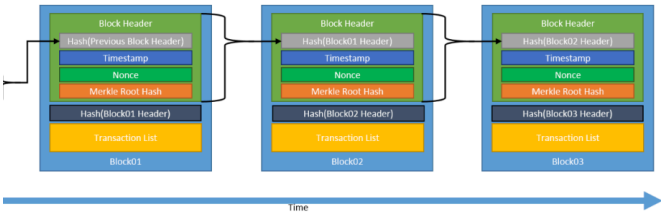
So, how can we sort blocks by hashing?
Remember that the **hashing** returns a fully random (numeric) code. It's deterministic so given a certain value to hash, if we give it again we'll receive the same digest. In practice, given a certain string of any length in input (a **message**) that can be any kind of file (e.g. a file, some text, an image, etc..), we get a fixed-length hash value in output (a **digest**). There is **no secret key involved** and **all operations are public**. Even the smallest change of input will result in a completely different output digest.

```
SHA3-256("Hi there!") =
0xe10f7b08a108024dcdd178e4
cf5a37a60afd2cc12fba6dd39f
d0bd9bf3190925
```

SHA3-256 is a hashing function whose output is 256 bit, which is a number that can represent decimal numbers from 0 to 2^{256}-1 which is a huge number. The higher the size of the digest (hash output), the harder it is to find a collision, meaning two seeds that have the same digest, because we will have less possible outputs. That's why with **SHA3-256 it is pretty much impossible to get a collision**.

So, again. the idea is

1. For every block, a hash is generated from itself;
2. Every block contains the hash code of the previous one.

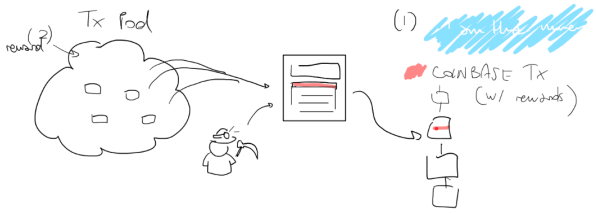


Which mining node to trust?
In blockchain networks, transactions are validated and added to the blockchain through a consensus mechanism. In the **Proof of Work (PoW)** model, the right to publish the next block is granted by solving a **computationally intensive puzzle**. **Solving the puzzle is difficult**, **checking that a solution is valid is easy** (this is also because each node has to verify the solution a lot of times):

- Any proposed block that did not satisfy the puzzle would be rejected;
- For each attempt, the mining node computes the hash for the entire block (incl. header).

Past work put into a puzzle does not influence one's likelihood of solving future puzzles. When a node receives a completed block from another one, they are incentivized to include the new block. Other mining nodes will include it and start building off it. If **they refuse to accept the new block**, they will be building off a shorter chain of blocks and remember that **by default, the longest** (again, it is not actually the longest, we'll eventually explain why) **valid chain is adopted** (this basically means that in the future they will be more likely to lose).

Mining nodes strain to solve puzzles and this costs electricity. For this reason, **winning mining nodes are awarded with coins**. This creates an economic system where you can only participate by incurring costs. This reward comes from nobody, but who writes that the transaction for the reward is for the miner? The miner itself! Why? Because since the miner is publishing a block, and this block will be accepted, then also the transaction to the miner will be in the accepted block, and whatever is in the accepted block, and so in the ledger, is reality for everyone. All the transactions in the block are taken from the pool, except one, which is the special transaction for the reward, which is put directly in the block.



The protocol dictates the rules for the coinbase transaction, in order for the miner to **not be able to give themselves a higher reward**. If a miner tries to **cheat, it will incur in losing money** (mostly because of the situation described in the previous paragraph). This simple **game-theoretical equilibrium** is the core of the Bitcoin decentralized consensus algorithm.

Also notice that the winning miner is not just the one who solves the puzzle, but their block also need to be accepted by the rest of the network. This is because if the block gets in the shorter branch of the bitcoin, in the future this branch won't be valid anymore and so it's like the block has never existed.

In Bitcoin, the time needed for a block to be published is **10 minutes**. If we don't receive transactions for about 10 minutes, we can also mine a block without transactions, and since this is a valid block, it can be accepted. This is

healthy for the network since even empty blocks are useful to consolidate the previous blocks (bury them down in the blockchain, in order for them to be more reliable), where without published blocks this wouldn't happen.

If a change has to be done to the protocol, the nodes that agree to this change, then they just change; the ones that don't agree they keep the protocol as it was, but the network will be separated in those two halves, which won't interact between each other.

Why does the puzzle require 10 minutes? Why not less so the blockchain could be faster? It's a tradeoff between being fast and secure (if the puzzle was too easy, malicious users could solve it easily; if it's too hard, the blockchain would be more secure but less efficient at producing transactions). This can be done because there is a "knob" to adjust the difficulty of the puzzle, which difficulty is increased or decreased depending on the average mining time for the last 2 weeks.

The **bitcoin reward is cut in half each 210.000 blocks** (each four years), so in a few years there won't be a reward anymore (when 21 millions BTC are released from the starting point in 2009), but the miner will only get the transaction fees (or changing the scheme of proof). This is done because mining causes inflation, since bitcoins are generated from thin air. The bitcoin circulating around is ever increasing, because even if we lose the private key and we cannot access the bitcoins anymore, they are still there.

Now, let's say something about consensus. **Consensus** is:

- **Fault Tolerant**: Decentralized systems are not likely to fail accidentally because they rely on many separate components;
- **Attack Resistant**: Decentralized systems are more expensive to attack and destroy or manipulate because they lack sensitive central points;
- **Collusion Resistant**: It is much harder for participants in decentralized systems to collude to act in ways that benefit them at the expense of other participants.

An example of why the order is important and why is important to play by the rules:

- Once step (1) has taken place, after a few minutes some miner will include the transaction in a block, say block number 270;
- After about one hour, five more blocks will have been added to the chain after that block, with each of those blocks indirectly pointing to the transaction and thus "confirming" it;
- At this point, the merchant will accept the payment as finalized and deliver the product; since we are assuming this is a digital good, delivery is instant;
- Now, the attacker creates another transaction sending the 100 BTC to himself. If the attacker simply releases it into the wild, the transaction will not be processed; miners will notice that the transaction consumes funds no longer available;
- So instead, the attacker creates an alternative version of the blockchain, starting by mining another version of block 270 pointing to the same block 269 as a parent but with the new transaction in place of the old one;
- Because the block data is different, this requires redoing the proof of work. Furthermore, the attacker's new version of block 270 has a different hash, so the original blocks 271 to 275 do not "point" to it; thus, the original chain and the attacker's new chain are completely separate;
- The rule is that [...] the longest blockchain is taken to be the truth, and so legitimate miners will work on the 275 chain while the **attacker alone** is working on the 270 chain;
- In order for the **attacker to make his blockchain the longest, he would need to have more computational power than the rest of the network combined** in order to catch up (hence, "51% attack").

Concerning the users in the Blockchain, they are **pseudonymous**, meaning that their **real-world identity is anonymous** but **their accounts are not** (notice that their address is always the same). This promotes complete **transparency**, which is a propeller for trust in its use. In the GDPR definition, anonymity means that you cannot ever single out an individual, even if you don't know their name, you can correlate the information in order to filter out more and more people. **In blockchain anonymity cannot work**, since if we cannot single out the account, then we cannot

follow the transaction the account made, and so we cannot compute the amount of coins that account has. Pseudonymity means that a person can have multiple accounts (notion of **wallets**), but also a single account can be owned by a multiplicity of people. **If someone connects the person with the wallet, then the pseudonymity is lost**. In the long run, with many transactions, one can analyze the data and **connect who you may be**.

How can we know that the owner of a transaction is a certain user without being able to copy their signature?
By using **public key cryptography** (a.k.a. **asymmetric key cryptography**). This kind of cryptography uses a pair of keys, the public one and the private one mathematically related one to another (notice that the private key cannot be derived from the public one though). The public key may be made **public** without reducing the security of the process while the private key must remain **secret**. The signature can then be verified using the corresponding **public key**. So, in Bitcoin **private keys are used to digitally sign transactions** while **public keys are used to derive addresses** (in other words **public keys are used to verify signatures generated with private keys**).

SUM. Blockchain is a decentralized protocol for a distributed ledger system eventually reaching consensus on transactions. It is:

- Politically decentralized: No entity controls the network;
- Architecturally decentralized: No infrastructural central point of failure;
- Logically centralized: There is one commonly agreed state* and the system behaves like a single computer;
- Distributed information: Every node* has access to the full history of transactions.

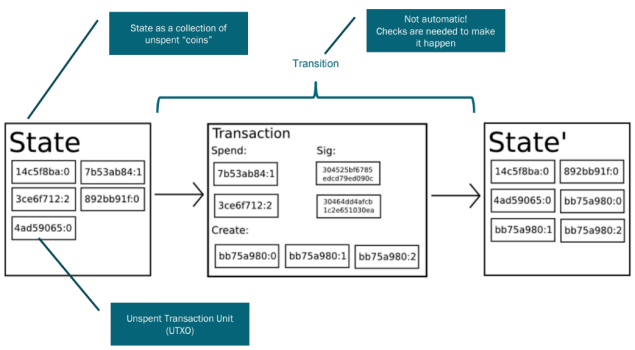
4. Bitcoin

Bitcoin is a decentralized digital cryptocurrency that was created by an anonymous person or group of people using the pseudonym "Satoshi Nakamoto." It was introduced in a whitepaper titled "Bitcoin: A Peer-to-Peer Electronic Cash System" published in 2008. Bitcoin is often referred to as the first and most well-known cryptocurrency.
Blockchain is a **decentralized system**, meaning the decision happens in a decentralized way (everybody can participate to consensus, but not everyone does, hence decentralized and not distributed). The whole network works in a **peer-to-peer** fashion meaning that **the whole network maintains collectively a public ledger.** The information is distributed, meaning that **potentially every node has a copy of the ledger**. If the information was decentralized and not distributed it was something like having multiple servers, since not all nodes have the copy of the ledger (Remember: If I need information about the ledger, I already have it).

Now, this is the behavior of every (full) node in the network:

1. **New transactions are broadcast** to all nodes;
2. **Each node collects new transactions into a block**;
3. Each node works on finding a difficult **proof-of-work** for its block;
4. When a node finds a proof-of-work, it **broadcasts the block** to all nodes;
5. Nodes **accept the block only if all transactions in it are valid** and **not already spent**;
6. Nodes express their **acceptance** of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash. When a node accepts a block, it doesn't tell anyone that it has accepted it, but it includes it in its own blockchain, and just broadcasts it to the neighborhood.

We can see the Bitcoin Blockchain as a **State Transition System**. In a standard banking system, for example, the state is a balance sheet, a transaction is a request to move $X from A to B, and the state transition function reduces the value in A's account by $X and increases the value in B's account by $X. If A's account has less than $X in the first place, the state transition function returns an error. Hence, one can formally define:
APPLY(S,TX) -> S' or ERROR
EX. APPLY({ Alice: $50, Bob: $50 },"send $20 from Alice to Bob") = { Alice: $30, Bob: $70 }.
APPLY({ Alice: $50, Bob: $50 },"send $70 from Alice to Bob") = ERROR.



Let's assume that we have a certain state s, then inside of the state s we have the collection of unspent "coins" that we call **Unspent Transaction Unit (UTXO)**. The "state" in Bitcoin is the collection of all coins (technically, "unspent transaction outputs" or UTXO) that have been mined and not yet spent, with each UTXO having a denomination and an owner. A transaction contains **one or more inputs**, with **each input containing a reference to an existing UTXO and a cryptographic signature produced by the private key associated with the owner's address**, and **one or more outputs**, with each output containing a **new UTXO to be added to the state**. Usually, the output number is two: We have **an output for the redeemer** and **one for the sender** (optionally, to return the change in case). The difference between inputs and outputs is called **transaction fees** and this value must always be a positive number (the inputs must always be higher than the outputs otherwise we would be creating money from nowhere and this is not allowed unless we are the miners). Amounts are expressed in **Satoshi (sat)** (also noted as ㋎) and 1 sat = 10^{-8} BTC = 0.00000001 BTC and represents the smallest amount we can send around. This value is used because it is preferable to use integers in transactions. Using decimals is not a good idea because they can lead to approximation errors.

The state transition function APPLY(S,TX) -> S' can be defined roughly as follows:

1. For each input in TX:
    - If the referenced UTXO is not in S, return an error.
    - If the provided signature does not match the owner of the UTXO, return an error.
2. If the sum of the denominations of all input UTXO is less than the sum of the denominations of all output UTXO, return an error.
3. Return S' with all input UTXO removed and all output UTXO added.

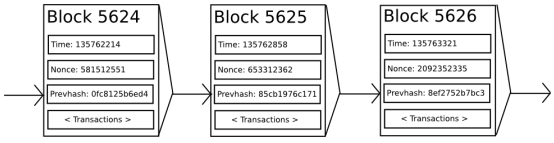
The first half of the first step prevents transaction senders from spending coins that do not exist, the second half of the first step prevents transaction senders from spending other people's coins, and the second step enforces conservation of value. In order to use this for payment, the protocol is as follows.

When we receive cryptocurrency in a transaction, it is received in the form of one or more UTXOs. These UTXOs represent ownership of specific amounts of cryptocurrency and are associated with your wallet's public key. UTXOs can be thought of as discrete coins or tokens. When you spend cryptocurrency, you need to reference and consume specific UTXOs. If the amount you want to send exceeds the value of a single UTXO, you can combine multiple UTXOs to reach the desired amount. Conversely, if you receive more cryptocurrency than you want to spend, the change is returned to you as a new UTXO. We can even merge together belongings from different multiple accounts (N accounts to 1 account).

EX. Suppose Alice wants to send 11.7 BTC to Bob. First, Alice will look for a set of available UTXO that she owns that totals up to at least 11.7 BTC. Realistically, Alice will not be able to get exactly 11.7 BTC; say that the smallest she can get is 6+4+2=12. She then creates a transaction with those three inputs and two outputs. The first output will be 11.7 BTC with Bob's address as its owner, and the second output will be the remaining 0.3 BTC "change", with the owner being Alice herself.

In Bitcoin, the algorithm for checking if a block is valid, expressed in this paradigm, is as follows:

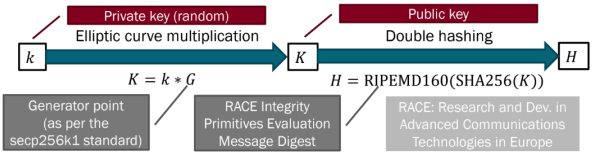
1. Check if the previous block referenced by the block exists and is valid;
2. Check that the timestamp of the block is greater than that of the previous blockfn. 2 and less than 2 hours into the future;
3. Check that the proof of work on the block is valid;
4. Let S[0] be the state at the end of the previous block;
5. Suppose TX is the block's transaction list with n transactions; For all i in 0...n-1, set S[i+1] = APPLY(S[i],TX[i]) If any application returns an error, exit and return false;
6. Return true, and register S[n] as the state at the end of this block.



Now let's go more into details about how **signing** works in Bitcoin. First of all remember that what happens in a transaction is that **each owner transfers the coin to the next by digitally signing it**. A specific digital signature

algorithm called **Elliptic Curve Digital Signature Algorithm (ECDSA)** is used. First of all a pair of public-private key is generate:

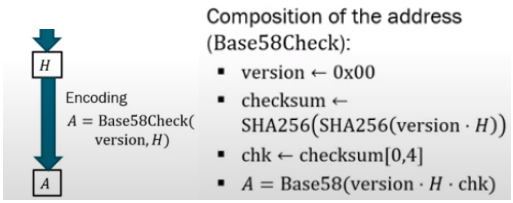
- **Private key**: A secret number, known only to the person that generated it (essentially, randomly generated). Notice that in this case **the account is the private key** (meaning that starting from it I can apply some operation to get the account). It is a 256 bit integer (32 bytes) generated randomly (not pseudorandom, but cryptographic random, so it doesn't take a seed) by the protocol. The private key is kept secret and is used to sign transactions.
- **Public key**: It is derived by the private key, but this is a one-way function, and we cannot derive the private key from the public key (so this is a **one way function**). It is a 33 bytes long (compressed) string. The public key is derived from the private key and is used as the recipient's or sender's address. Bitcoin uses the secp256k1 elliptic curve, which is defined by the equation $y^2 = x^3 + 7$. This curve has specific mathematical properties that make it suitable for cryptographic operations. Start with a known base point (G) on the elliptic curve. This base point is a predefined constant for the secp256k1 curve. To derive the public key (K) from the private key (k), you perform scalar multiplication on the base point (G) using the private key as the scalar (k * G). This operation involves adding the base point to itself k times and the result is the public key K.



Instead of using the public key directly as an address, Bitcoin typically uses the **RIPEMD160 hash H of the SHA-256 hash of the public key K** (so the public key hash H has a final length of 160 bits, because of the outer hashing algorithm). Again, this is a **one way function**. Moreover, to detect errors and prevent mistyped addresses (e.g. know if the account exists or not when doing a transaction), a **checksum is added to the public key**. This checksum is a result of a hash of the public key, and it is typically added as a final step. The public key, version prefix, and checksum are encoded using **Base58Check encoding**, which produces the **final Bitcoin address**. This allows us to decrease the number of digits required to represent a Bitcoin address (making it more user friendly) and the use of a checksum helps prevent situations where a mistyped address leads to a **nonexistent address**.

Going more into details of how this last step that generates the address works:

- Base64 uses 26 lowercase + 26 capital letters + 10 numerals and 2 more characters (+ and /). This enables you to encode any file into a subset of the ASCII format.
- Base58 is just Base64 without capital O, zero 0, lower L (l) capital i (I) and the symbols + and /. Which are all characters that can be confused.
- The checksum is generated taking in input a version number (0x03 for private key and 0x00 for public key - in this case we use the public one). And then the concatenation between the version number and the unencoded address is **double hashed with SHA**. We then take the first 4 bytes of the checksum and encode the concatenation between the version, H and the first 4 bytes of the checksum and encode it.



Note: You can also personalize the address, but this is dangerous, since the more personalized an address is, the less random it gets.

Note: Contrarily from the previous steps, in this last step it is possible to **retrieve the public key from the address** since the **base58 function is a two way function**. This is fine, the important thing is to not be able to retrieve the private key from anything. When computing H, we are encoding the concatenation between (version, H, chk). So we can decode it and obtain the concatenation in bits, and so you would be able to get the H from there (just remove the first bits of the version and get the 160 bits from there).

Some considerations:

- I cannot have one address stemming from two different public keys;
- I cannot have one public key stemming from two different private keys;
- I cannot produce signatures authenticated by a public key with different private keys;

- Therefore, I cannot forge my key as a user of a UTXO if I do not own the private key that ultimately leads to that address.

Finally, here is how all this components used to sign a Bitcoin transaction:

- Transaction Data: To create a transaction, you provide details about the transaction inputs (UTXOs being spent) and outputs (new destination addresses). The inputs reference previous UTXOs, and the outputs specify the amount and destination address for the funds.
- Hashing Transaction Data: The transaction data, including the inputs and outputs, is hashed to create a unique transaction identifier. This hash is part of the data that will be signed.
- Signature Creation: To sign the transaction, you use the private key (k) to create a digital signature for the hashed transaction data. The ECDSA algorithm is used to produce this signature. The signature is specific to the transaction and the private key.
- Signature Inclusion: The digital signature, along with the public key (or its hash), is included in the transaction's input script. It proves that the person signing the transaction has the private key associated with the public key.
- Transaction Broadcasting: The signed transaction is then broadcast to the Bitcoin network. Nodes and miners on the network will verify the transaction's signature to ensure that it is valid. If the signature is valid, the transaction will be added to a block.
- Confirmation and Inclusion in the Blockchain: Once the transaction is included in a block and the block is added to the blockchain, it is considered confirmed. The transaction is now part of the public ledger and cannot be altered.

How do we check the validity of a transaction?

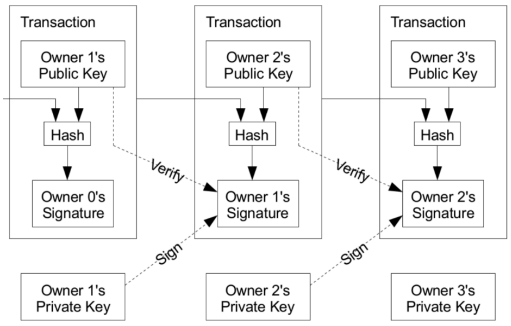
The **transition applies if and only if** all of the following apply:

- For each input in the current state:
  - The UTXO is in the current state;
  - The signature matches the owner of the UTXO*.
- The sum of denominations of all input UTXOs is higher than or equal to the sum of denominations in the output.

Otherwise, an error is returned.

The **next state** is such that:

- All UTXOs in the input are removed (implicitly marked as spent);
- All output UTXOs are added.



Note: About **privacy**, the public can see someone is sending an amount to someone else. Who those two people are, remains unknown (users are identified by their Bitcoin addresses and not to real world identities). Some linking is still unavoidable. For example, multi-input transactions are a type of Bitcoin transaction where a user combines multiple UTXOs (Unspent Transaction Outputs) as inputs to form a single transaction. This can lead to increased privacy risks because it may be easier to link the various UTXOs as belonging to the same user.

So far, we said that miners have to solve complex mathematical puzzles through computational work to validate and add new transactions to the blockchain. This is done with the **Proof of Work (PoW)** mechanism. The proof of work is based on finding the **nonce** (a 32 bit number) such that if we inject that nonce into the block header, the **hash of the block is less than a given number**. So, it is all about luck and computation power to find the right nonce before anyone else. Remember that this process is costly because is energy expensive so the main rewards for winning proof of work are:

- **Mining reward**: 6.25 BTC (as of May 2020). Halved every 210,000 blocks (roughly, 4 years: in 2009, it was 50 BTC);
- **Transaction fees**: If the output value of a transaction is less than its input value, the difference is a transaction fee.

$$\text{blockTxFee} = \sum_{tx \in \text{Block}} \left(\sum_{o \in tx} o.\text{value} - \sum_{i \in tx} i.\text{value} \right)$$

Note: If we make a transaction with 0 transaction fee, the chances that the transaction will be inside of a block are pretty much zero, since the miners are not incentivised.

Does every node need to download the full blockchain?
Only the **full nodes** have the entire blockchain stored on the disk, but most of the nodes are light nodes, which doesn't do that. **Light nodes** are also called **SPV nodes** (because of the Simplified Payment Verification protocol) which allows them to validate and issue new transactions by only downloading the block headers. Crypto wallets are light nodes in fact a user tracks only the UTXOs that belong(ed) to that user, rather than all transactions.
Note: Of course in a system like this light nodes have to trust full nodes from which they download the branches associated with transactions relevant to them. The whole system works only because there are a lot of full nodes. Otherwise, if only few full nodes remain it would be like a bank in which we have to trust a few entities (in the bank only one).

5. A Focus on Consensus

Proof of Work (PoW) is a **consensus mechanism (not a protocol!)** used in many blockchain networks, including Bitcoin and Ethereum, to agree on the state of the blockchain.It's a way for the network to determine which transactions are valid and should be added to the blockchain. So, it is just an instrument to make it more difficult to publish malicious blocks. In a PoW-based blockchain, **miners compete to solve a cryptographic puzzle**, known as the Proof of Work puzzle. The solution is hard to be found but easy to verify.
Bitcoin PoW is built upon Hashcash (1997). PoW is about **finding a nonce for which the hash of the header is less than a certain number**, which means finding a hash which has the **N most significant bits (the starting bits) equal to 0**. The **header is fixed**, except a numeric value (**nonce**, a word of 32 bits) which can be changed by the mining node to solve the puzzle.
EX. Let's see an example. Basically what we have to do is using the SHA-256 algorithm find the nonce n such that:

$\texttt{SHA256(concat("blockchain", } n\texttt{))} = 0x\mathit{N}$ such that N starts with **000000**

Let's start to try and guess n:

- `SHA256("blockchain3") =`
- `0xeb61c3724d6da33605084d2d232bba0563cb82f4ad82c101b42f23c2e86277ef`
  - Nope
- ... 10,730,892 attempts later ...
- `SHA256("blockchain10730895") =`
  `0x000000ca1415e0bec568f6f605fcc83d18cac7a4e6c219a957c10c6879d67587`
  - At last!

So it is all about trying a lot of numbers until we find the right one.

The **higher the N**, the **harder the problem** (so if we increase N to 7 in the last example we would probably need much more attempts before finding the right nonce). **Adding more leading zeros in the hash output effectively tightens the criteria for a valid solution**. To find a suitable nonce, miners need to perform more computational work and attempt many more hash calculations. As a result, it becomes harder to find a valid solution.
Could we save results gained with leading "000000" to spare cycles on "0000000" (maybe for the next block)?
No, because the block header changes every time according to the contents of the block (to stay with the metaphor, the prefix "blockchain" changes at every block!) so we have to compute everything from the beginning.
What we can use is a **divide et impera** approach by distributing the workload (and reward) to more than one node, each node checking in a range of nonces, since the nonce is a limited number. This is a **mining pool**.
EX. Node 1: check nonce between 0 and 536870911
Node 2: check nonce between 536870912 and 1073741823
Node 3: check nonce between 1073741824 and 1610612735
Node 4: check nonce between 1610612736 and 2147483647
Also note that the **nonce is not unique**, and more than one nonce can produce the correct hash (we just have to find a nonce that allows us to respect the properties stated above).

Changing the N makes the difficulty tuneable. Difficulty changes in Bitcoin every 2016 blocks (about 2 weeks) and the objective is, keep the average block time more or less stable. The operation of tuning is called **retarget**, and it's done in order to keep the rhythm of publishing blocks regular (one block each 10 minutes). How does this happen?

- During each difficulty adjustment period, the network calculates the actual time it took to mine the previous 2016 blocks. This is done by **recording the timestamps** of when each block is mined.

- The target block time in Bitcoin is 10 minutes. If the actual time taken to mine the previous 2016 blocks is less than two weeks (14 days), the difficulty is increased to slow down block generation. Conversely, if it took more than two weeks, the difficulty is decreased to speed up block generation.
- The difficulty is adjusted using the following formula:
  New Difficulty = Old Difficulty * (Actual Time / Target Time)
  If the actual time was greater than the target time, the new difficulty will be lower (easier to mine), reducing the work required. If the actual time was less than the target time, the new difficulty will be higher (harder to mine), increasing the work required.

The digest is computed in Bitcoin with **SHA-256** that is **demanding for processing, not for memory**.
ASICs (Application-specific Integrated Circuits) turn out to be very effective as well. It is a type of **hardware** designed and optimized for a specific computational task or application. In the context of blockchain technology, ASICs are specialized hardware devices created for the sole purpose of mining cryptocurrencies, particularly those that use Proof of Work (PoW) consensus mechanisms like Bitcoin. ASICs provide significant computational power, often surpassing the processing power of general-purpose hardware. Miners using ASICs have a competitive advantage in solving PoW puzzles and, therefore, a higher likelihood of being the first to mine a new block and earn the associated rewards.

What if two chains exist that are both valid?
If we have two branches, the one with the **highest cumulative work** is chosen (meaning the sum of all the work put in the entire branch of blocks), even if it's not the longest ones, in order to avoid attacks. This is all a matter of security because an attacker can produce longer chains by pushing all empty blocks, since there are no transactions to validate and so it'll be easier. This isn't quick, but may be faster than the competitors since we have to validate nothing and that's why we get the branch with the highest cumulative work instead of the longest one (even though they match most of the time).
Note: Choosing the branch with the highest cumulative work is a kind of consensus, even if there are no votes, but since each node in the network applies the same reasoning on deciding the valid branch, everyone will agree on the result
Work for a block refers to the **expected number of hashing attempts needed to find a valid proof-of-work for that block**:

$$\frac{2^{256}}{\text{target}+1}$$

Potential digests that meet the target

Talking about difficulty, we can give the following formula:

$$\frac{\text{target}_{MAX}}{\text{target}}$$

Where target_{MAX} is the maximum possible target or the lowest difficulty admitted (2^{16}-1*2^{208}).

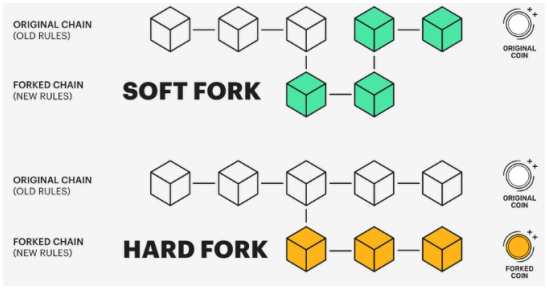
Again, the node who wins the puzzle gets a reward. Cryptos are dug out of nowhere and that's why nodes are called **miners**. In general, playing by the rules pays off.
What if two miners solve the puzzle with different blocks?
Stale blocks in bitcoins are valid blocks that are not included in the blockchain, and so the miner won't have any reward. In ethereum, even mining stale blocks will give a reward (less valuable though). In ethereum these are called **ommer** (or **uncle**) blocks.

What happens in Ethereum?
Ethereum uses the **Ethash algorithm**, which is a PoW similar to bitcoin's, but you also need to occupy the memory with a dataset of 1GB. In particular, key to this algorithm memory-hardness is its reliance on a directed acyclic graph (DAG) file, which is essentially a 1 GB dataset created every 125 hours, or 30,000 blocks also known as **epoch**. In this case the block time is way shorter (about 15 seconds) and this is better for computation but it happens that more miners guess the right nonce. Miners who find blocks that are valid, but not the winning block, are paid a reduced fee as **consolation** (more details when talking about Ethereum).

When does forking occur?

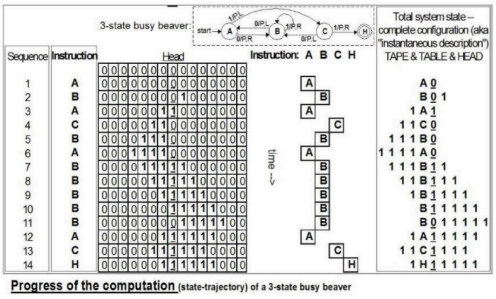
In Bitcoin, there are two types of fork:

- Soft Fork:
  - A soft fork occurs when there is a change to the protocol that is backward-compatible, meaning the new rules are compatible with the old rules. Existing nodes can still accept and validate transactions and blocks according to the new rules, even if they have not upgraded their software.
  - In a soft fork, there is no split in the blockchain because all nodes can continue to operate on the same chain. However, nodes that have not upgraded might not enforce the new rules, potentially leading to less secure network behavior.
  - Soft forks are typically used to make more minor protocol changes, such as tightening rules on block validity.



- Hard Fork:
  - A hard fork occurs when there is a change to the underlying protocol of the blockchain that is not backward-compatible. In other words, it introduces new rules that are incompatible with the old rules, leading to a split in the blockchain.
  - In a hard fork, two distinct chains emerge, with one following the old rules and the other following the new rules. Each chain continues to grow independently from the fork point, resulting in two separate cryptocurrencies.
  - Hard forks are often used to introduce significant protocol upgrades, changes to consensus rules, or changes to the block size limit.

6. A Focus on Ethereum



First of all, let's introduce the **Turing Machine**. It was invented by Alan Turing in 1937 and it is a theoretical, idealized model for mathematical computation. It consists of:

- A line of cells (tape) that can be moved back and forth;
- An active element (head) with a state that can change the property (color) of the active cell underneath it.

A set of **instructions** for how the head should **modify the active cell and move the tape** is presumed. At each step, the machine may modify the color of the active cell and change the state of the head. After this, it moves the tape one unit to the left or right.
The Turing Machine is important because it can **simulate ANY computer algorithms**, in particular it can be used to determine the computability of the decision problem. In general, given an algorithm to a machine, **this is not able to tell us if it terminates or not** (given that we can provide any possible input). This was proven with the Turing Machine: An algorithm solving the **halting problem** for all possible programs and inputs, does not exist.
Also, if a computing language is not Turing-complete, it can solve only a subset of problems that other Turing-complete languages can do. A programming language is said to be Turing-complete if it can simulate a Turing machine. In other words, a Turing-complete language can express any algorithm or computation that can be described in an algorithmic manner. There are many Turing-complete languages such as C, C++, C#, Python, Javascript, Solidity and others. On the other hand, notable examples of non-Turing-complete languages are SQL, HTML, CSS and **Script** where this last one is the **bitcoin blockchain programming language**.

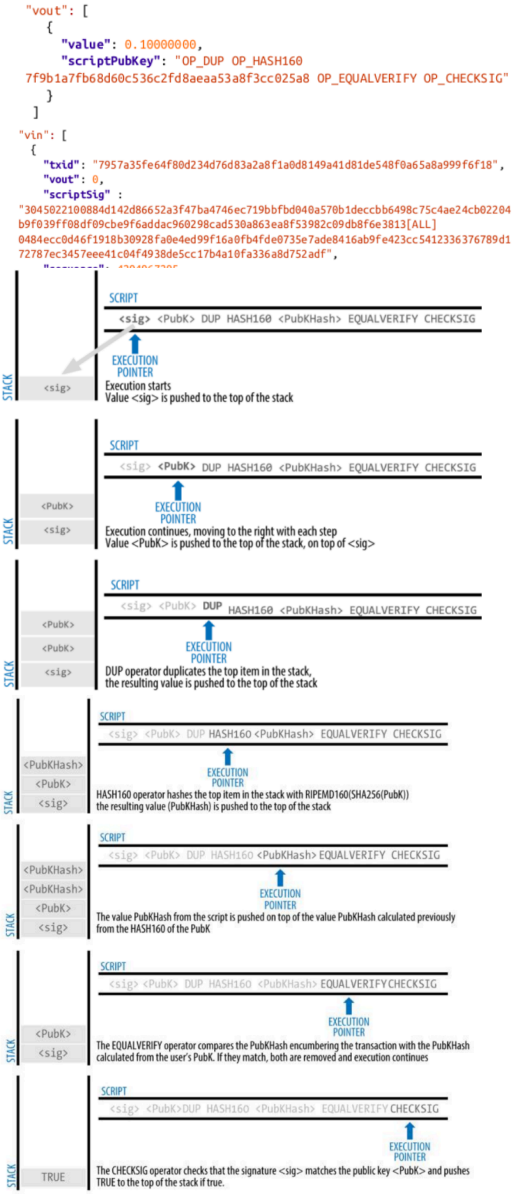
What isn't Script Turing-complete?

Script is not turing-complete on purpose, **in order for it to be more secure**, since it's less capable. This is because a Turing-complete language cannot be verified a priori (the halting problem). Script was indeed born not with the purpose of letting the blockchain be programmable. There are some reasons for which it is non Turing-Complete:

- Safety: Bitcoin's primary purpose is to facilitate secure peer-to-peer transactions and store value. Allowing a Turing-complete language could introduce a significant level of complexity and potential security risks. A

non-Turing-complete language ensures that scripts are more predictable and less prone to unintended or malicious behavior.

- Determinism: Bitcoin's scripting language relies on determinism. Determinism is a critical feature for ensuring that the outcome of a script is predictable and consistent across all nodes in the network. A Turing-complete language could introduce non-deterministic behavior, making it challenging to reach a consensus on the validity of transactions.
- Security: By limiting the complexity of the scripting language, Bitcoin reduces the risk of software bugs and vulnerabilities. A simpler and non-Turing-complete scripting language is less likely to have unexpected behaviors that could be exploited by attackers.
- Resource Consumption: A Turing-complete language could potentially lead to resource-intensive computations on the Bitcoin network. Limiting the computational capabilities of scripts helps control the computational burden placed on nodes and miners, ensuring that the network remains efficient and responsive.
- Block Validation: In Bitcoin, every full node validates all transactions in each block. A Turing-complete language could make block validation more computationally expensive, slowing down the network.
- Script Verifiability: Bitcoin's scripting language is designed to be simple and verifiable. By keeping the language non-Turing-complete, it is easier for nodes to quickly verify the correctness of scripts and transactions.



What is Script used for?

Script is used in **UTXOs**. They always have attached a **looking-script** (scriptPubKey) which **defines the conditions to spend the output**. It is usually put next to the fields value (the amount of the UTXO) and locking-script size.
EX. 3 OP-ADD 5 OP_EQUAL. This is an example of looking-script that can be translated into “Give me a value that summed to 3 gives me 5” (?+3 == 5).

Script is also used in transaction inputs. We use an **unlocking-script** (scriptSig) that verifies the conditions in the locking script meaning that it **defines the conditions that have to be satisfied in order to receive a specific UTXO** (the condition to unlock the transaction, meaning that the transaction is in the pool and can received only by the one that can provide an unlocking script). It is usually put next to the fields: Transaction hash (txid), output index (vout), unlocking-script size and locktime number (sequence). Notice that the locktime number specifies how many blocks we have to wait before allowing the transaction to be used to be spent again.
EX. An unlocking script for the previous locking script is 2, because 2 is the value that satisfies the condition 3 OP_ADD 5 OP_EQUAL.
Note: Of course this is a toy example, in the real world the unlocking script needs the signature and the public key in order to unlock the relative transaction, and so only the one with the private key that matches the public key can sign, and so only them can receive the transaction. A general example is given right below.

Another popular operation is the **unlocking script based on public key hash**. We have a locking-script (**scriptPubKey**) that does the follows:

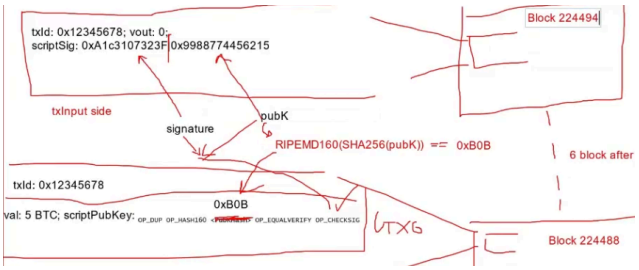
```
OP_DUP OP_HASH160 <PubKHash>
OP_EQUALVERIFY OP_CHECKSIG
```

Basically takes the public key hash (RIPEMD160 digest of the SHA256 hash of the recipient's public key, no BASE58CHECK involved here) and verify that is equal to the signature we are going to check (so when we send a transaction to whoever is capable of showing via signature that they are the ones that have the private key for the given public key). So notice that we don’t send the transaction to a specific account (we don’t use the Base58Check) but we send

it to the community and the one who has the key it will eventually receive it. Basically in the UTXO we write the public key hash that the next spender (the receiver) has to use to produce a signature (which means that they need the private key related to the public key). Now, this was locking-script, the unlocking-script (**scriptSig**) instead goes like this:

```
<sig> <pubK> OP_DUP OP_HASH160
<PubKHash> OP_EQUALVERIFY OP_CHECKSIG
```

The signature and the associated public key is given.
Note: We write conditions like incomplete expressions (there are some parts missing). It's up to the spender to fill the missing values to complete the expression and finally verify the condition. To sum up, to spend the funds, the spender must provide a valid signature and the corresponding public key. The script verifies that the provided public key hashes to the expected Bitcoin address (by computing the RIPEMD160(SHA256(pubK) of the public key provided) and that the signature is valid for that public key. If these conditions are met, the transaction is considered valid and the funds can be spent.



We can also use multiple signatures for a matter of security, so we may need at least M out of N signatures to unlock a UTXO. Multi-signature transactions require the authorization of multiple private keys to spend funds from a Bitcoin address. They are commonly used in situations where multiple parties or signers need to collectively approve a transaction and they are constructed using conditional opcodes like CHECKMULTISIG that defines the required number of signatures and the associated public keys:

```
M <Public Key 1> <Public Key 2> ... <Public Key N> N CHECKMULTISIG
```

EX. A multisig script might look like this:

```
2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG
```

Meaning that we need 2 out of 3 signatures to unlock the UTXO. Concerning the unlocking-script we have to provide both signatures (only 2 of them out of 3):

```
0 <Signature B> <Signature C> 2 <Public Key A> … 3 CHECKMULTISIG
```

Note: The 0 in the front is solely due to a workaround to an interpreter bug.

Some limitations of Script:

- Lack of Turing completeness: No loops (cycles);
- Value blindness: No way for an UTXO script to provide fine-granular control over the amount to be withdrawn because UTXOs are "all or nothing" (we can take it or not, it is not possible to break it somehow).
- Lack of state: UTXOs are either spent, or unspent (no half-spent, on-the-way, conditionally accepted..);
- Blockchain-blindness: Using Script, UTXOs are unaware of the block they are in, the timestamp, the previous block hash and so on.
- Notice that **withdrawal limits are not possible** with Script (again, because we can only accept a UTXO or not, nothing more fine-granular can be done).

Concerning the **accounts**, they can be owned by external users and that's why we talk about **Externally Owned Accounts (EOAs)** (we call them like this because we can't know who is the owner and that they do with their funds):

- They are controlled by the account's private keys (if we lose the private key, we can't use the account anymore!);
- They are capable of sending transactions;
- They contain a balance in ether (also denoted as ETH, or Đ);
- They have storage for data.

We also have another kind of account called **Contract Accounts (CAs)** that:

- Can be triggered by humans sending a **transaction** or other contracts sending a **message**;
- When executed, can perform arbitrarily* complex operations;

- Have their own persistent state and can call, or even create (!), other contracts;
- Have no owner*;
- Contain a balance in ether (also denoted as ETH, or Đ);
- Have storage for data.

Ethereum adopts a **balance model** (unlike Bitcoin where the number of coins are derived from the UTXO associated with the public key and it's based on a transaction model). An account is identified by an address. In order to **keep the local state info, a nonce is used** (here it's used as a counter so nothing to do with PoW), in particular:

- In **EOA**, the **nonce is used to express the number of transactions sent**. When an account initiates a transaction, it includes a nonce that corresponds to the account's sequential transaction count. For example, the first transaction from an account will have a nonce of 0, the second will have a nonce of 1, the third will have a nonce of 2, and so on.
- In **CA**, the **nonce is used to express the number of contract-creations made** (a contract can't issue a transaction). Remember that contracts are Turning-complete so that can possibly execute every piece of code. If we assume to use a language like Java, we may create some instances of an object. So the nonce is used to count the deployments of the contract.

Ethereum nodes process transactions in the order of their nonces. When a transaction is received by the network, the node checks the nonce to determine if it's in the correct sequence. Now:

- A **transaction with a nonce ahead of the current count** means that it has a nonce greater than the next expected nonce for that account. In this case, the transaction **is kept on hold** until the correct nonce is reached. It will be processed when the nonce matches the expected nonce.
  EX. If the current count is 6 and the nonce is >7, the transaction is kept on hold since the current count will be equal to the nonce in the future and at that moment it will be processed.
- A **transaction with a nonce beyond the current count** means that it has a nonce smaller than the next expected nonce for that account. In this case, the transaction **is discarded** because it is out of sequence (the current count will never arrive to the nonce). Ethereum nodes expect transactions to be executed sequentially based on their nonces.

Remember that Ethereum is a decentralized system so nodes may receive the transactions in either order. Now let's assume a situation in which a user Alice has 10 ETH on her account. Alice sends an important transaction worth 8 ETH first, then another one worth 3 ETH. Which of the two will be discarded? Without the nonce, it would be random! The nonce is used to establish an order for the transaction, and so discard the last transaction if the account doesn't have enough balance. **Without the nonce, there is no concept of order**. Moreover, if anyone wants to replay a transaction, the nonce will be the same and so the transaction will be discarded by the network. We also cannot modify the nonce since we have to sign the transaction again, and we cannot do it if we do not have the private key of the owner of the transaction.
Note: In Bitcoin there isn't this problem since transactions are transmitting UTXOs, and I have to prove that those UTXOs are mine, otherwise I cannot spend them. Moreover we can't replay a transaction because if the original one (the associated UTXOs) has been spent we can't spend it anymore (just look in the blockchain if it was spent).

In Bitcoin, the wealth is based on the sum of UTXO associated with an account (so we can't properly speak of balance). In Ethereum, the balance is given by the number of Wei's owned. In Ethereum, **"wei" is the smallest and indivisible unit of the cryptocurrency Ether (ETH)** where 1 Wei is equal to 10^{-18} Ether. Remember that Satoshi was 10^{-15} BTC, so Wei is more fine-grained. We use those currencies because we want to make operations with integers and not floating points. When we send or receive Ether, you might deal with various denominations like wei, gwei (gigawei), szabo, finney, and ether, depending on the context and the Ethereum wallet or platform you're using.

Unit	Wei equivalent
Wei	10^0
Kwei (babbage)	10^3
Mwei (lovelace)	10^6
Gwei (shannon)	10^9
Microether (szabo)	10^{12}
Milliether (finney)	10^{15}
Ether	10^{18}

In a contract account, every time the contract account receives a message, its code gets activated, allowing it to:

- Read and write to internal storage;
- Send other messages;
- Create contracts;

Also notice that a **smart contract is immutable**: if we deploy a contract with a bug in it then the bug is forever, it is not possible to solve it anymore. Why is the code of a smart contract immutable?
Because it's inside of a transaction that we deploy, and since a transaction, once in the ledger, cannot be changed, the code inside also is immutable. This is not a need, but simply a consequence.

Contracts in Ethereum live in the Ethereum environment have direct control over their own balance and key/value storage and execute a piece of code when called. The contract code runs (in a logical sense) on the **Ethereum Virtual Machine** that is an emulation of a single, global "computer" (so it's a **globally accessible virtual machine** (like a mainframe), with lots of smaller computers). Notice that a contract is in each full node, but it doesn't belong to anyone. It's also not bound to any underlying programming language. EVM reads **EVM bytecode** (just like Java Bytecode), in order to be independent of the platform and the language the smart contract is written in, the important thing is that the language compiles to EVM bytecode. So, **Smart Contracts can be written in the high-level language of choice** (EX. Solidity, Serpent, LLL..) but they **will be compiled in EVM bytecode anyway**.

All programmable computation in Ethereum is subject to fees. The execution of the running programs on the EVM is measured in **gas** (not to be confused with Ether!) which is a virtual currency to measure how much a running program costs to run. Every transaction has the maximum amount of gas that a sender is willing to consume for a transaction and it is called **gasLimit**. It is specified by the sender when creating the transaction. If the execution of a transaction consumes more gas than the gas limit, the transaction is reverted, and any changes made by the transaction are rolled back (this is done mainly in order to **avoid possible infinite loops** or other errors that consume all the sender's gas). When a transaction is executed, **gas is implicitly withdrawn from the sender's account balance** according to the gas limit and gas price specified. Any **gas that remains unused at the end of a transaction is refunded to the sender**. This means that if the gas limit is set higher than what is actually needed to execute the transaction, the excess gas is returned to the sender's account.
EX. To run a simple transaction you need 21.000 units of gas.
Again, **Gas is not Ether** and doesn't exist outside of the execution of a transaction. Why? The ether is traded publicly on cryptocurrency exchanges and, regardless of the ether price, the core cost of computation stays the same.
In summary, Ether (ETH) is the cryptocurrency used for financial transactions and value transfer on the Ethereum network. Gas, on the other hand, is a measure of computational work and represents the cost associated with executing actions on the Ethereum blockchain. You use Ether to pay for gas when interacting with the Ethereum network, and the two are closely related but distinct concepts.

Transactions will necessarily have a **trade-off between lowering gas price and maximizing chances that their transaction will be mined relatively soon**. Transactions with higher gas prices are typically processed by miners more quickly because they offer greater incentives.
Note that **gasLimit applies to the total gas consumed by that transaction and all sub-executions**.
EX. A sends a transaction to B with gasLimit=1000, B consumes 600 gas before sending a message to C, C consumes 300 gas for its execution, how much gas is left to B to spend before running out of gas?
Gas left for B = gasLimit - (Gas consumed by B + Gas consumed by C) = 1000 - (600 + 300) = 100

Gas makes Denial-of-Service attacks very expensive (each call to a contract would cost the attacker a lot so they are discouraged). **Gas circumvents the halting problem** in fact **if the contract does not stop**, **after a while it might eventually run out of gas**. Notice also that **Gas is paid also for data sent**.

Now let's see what's the differences between transactions and messages. **EOA can send transactions (account to contract) while contracts cannot** because **they can't sign them** since they don't have a private key. What they can do is send **messages (contract to contract)** to invoke some functions on other contracts (so contracts can communicate with each other by sending those messages - not transactions). Messages **are also not serialized** meaning that storage is in a persistent state and **messages are not stored in the ledger.** There is no need to save on the ledger the messages exchanged between the messages since the **message is deterministic**, meaning we can compute the output knowing the input. We just need to remember what was the input and the parameters invoked in order to get the same output.

Concerning the (account to contract) transactions, we can find the following fields:

- Recipient;
- Signature of the sender;
- Amount of Wei to be transferred;
- Optional data field (input data for calling contract methods, init code for contract creation, etc..).

Before the **London Upgrade** (**EIP 1559** - 2021) we had 2 other fields to specify:

- **STARTGAS (gas limit) value**: Limit of consumable gas the transaction execution can take that this defined the limit of computational steps;
- **GASPRICE value**: The fee the sender pays per computational step.

Ethereum primarily used a **first-price auction** mechanism for determining the transaction fees, also known as gas prices. In a first-price auction, participants submit bids, and the highest bid wins, but the price paid is the amount of the winning bid. It works like this:

- Transaction Senders Set Gas Prices: When users initiate transactions on the Ethereum network, they have the option to set the gas price, which represents the amount they are willing to pay per unit of gas for their transaction to be processed by miners.
- Miners Prioritize Transactions: Miners select transactions to include in the blocks they mine. They often prioritize transactions based on the gas price offered—the higher the gas price, the more likely a transaction is to be included in the next block.
- Winning Bid Pays the Price: In the case of a first-price auction, the transaction with the highest gas price wins and gets included in the block. However, the price paid is the actual gas price set by the winning bidder. This means that even if a user set a high gas price, they would pay the amount they bid, not the second-highest bid.

So, the amount of gas spent was equal to:
Total fee = Gas_units_limit * Gas_price_per_unit (before London Upgrade - EIP 1559).
EX. Alice creates a transaction and specifies a gas limit and a gas price. Let's say she sets a gas limit of 30,000 units of gas and a gas price of 20 Gwei (gigawei) per unit of gas. This means she's willing to pay 20 Gwei for each unit of gas used in the transaction. The total cost of the transaction is calculated as the product of the gas limit and the gas price. In this case, the total cost is:
Total Cost = Gas Limit x Gas Price = 30,000 gas x 20 Gwei/gas = 600,000 Gwei/10^9 (since 1 ETH = 1000000000 Gwei) = 0.0006 ETH.

After the London Upgrade, another fee system was implemented. The amount of gas spent now is equal to:
Total fee = Used_gas_units * (Base_fee + Priority_fee) (after London Upgrade - EIP 1559)
Where:

- **Base Fee**: Instead of a first-price auction system for gas fees, EIP-1559 introduced a mechanism where a **"base fee" is dynamically determined** by the network **based on demand**. If the demand is high, increase the base fee (and so the total fee). If the demand is low, decrease the base fee (and so the total fee). The **base fee is burned**, meaning it is removed from circulation, reducing ETH supply and potentially **leading to a deflationary effect**. Another way to say it is that burning the base_fee*gas_usage is meant to reduce the total ETH circulating and therefore, it adjusts the inflation/deflation mechanism as the price of ETH, not the gas, changes. Just like bitcoin halves the bitcoin price periodically, in ethereum they use the base fee in order to reduce the amount of eth in circulation. The higher the demand of ETH in the network, the higher the base fee.
- **Priority Fee**: Users can include a "priority fee" along with their transactions, indicating an additional amount they are willing to pay to have their transaction processed more quickly. This fee goes directly to miners.

EX. Eve pays Trent 1 ETH and in the transaction she specified a gas limit of 21,000 units, a base fee of 10 gwei (set by the protocol) and a tip of 2 gwei (to the miner/validator). So:
Total fee: 21 000 × (10 + 2) = 252 000 gwei = 0.000252 ETH
Where 1.000252 ETH are deducted from Eve's account (coins transferred + total fee), Trent will be credited 1.0000 ETH (coins transferred), the validator receives the tip of 2 gwei (0.000042 ETH) while the base fee of 10 gwei (0.00021 ETH) is burnt.
Note: We exchange ETH with FIAT currency. Gas is not directly exchanged with FIAT currency.

The **Max Block Size in Ethereum is an indicator of the network usage** so the gas usage in the block is used to determine the max block size. The maximum block size was increased to 30 million gas units, with a target size of 15

million gas units. In this context, gas is considered a notion of space within a block. The protocol adjusts the base fee dynamically based on network utilization. If the **network capacity goes below 50% utilization** (meaning there is low demand for block space), **the base fee decreases**. If the **network is more than 50% utilized**, **the base fee increases**. The goal is to maintain a balance between supply and demand for block space. There are limits on how much the base fee can increase or decrease. The **maximum increase or decrease is capped at 12.5%**. This introduces a level of predictability and prevents sudden, extreme fluctuations in fees.
So, again, more network activity entails more ETH burnt so rather than making miners wealthier, the ETH in the accounts of users becomes scarcer and more valuable (deflation).

Block Number	Included Gas	Fee Increase	Current Base Fee
1	15M	0%	100 gwei
2	30M	0%	100 gwei
3	30M	12.5%	112.5 gwei
4	30M	12.5%	126.6 gwei
5	30M	12.5%	142.4 gwei
6	30M	12.5%	160.2 gwei
7	30M	12.5%	180.2 gwei
8	30M	12.5%	202.7 gwei

Ethereum addresses are unique identifiers **derived from public keys or contracts** using the **Keccak-256 one-way hash function** (Not SHA256 like in Bitcoin) that takes the public key as input and produces a 256-bit hexadecimal (base-16) hash value. The whole process works like this:

- Generate a Public Key: If you have a private key, you can derive the corresponding public key using the elliptic curve algorithm (specifically, the secp256k1 curve) used in Ethereum. If you don't have a private key but want to create a new address, you can generate a random private key and compute the corresponding public key.
- Apply Keccak-256 Hash: Take the public key and apply the Keccak-256 (SHA-3) hashing algorithm to it. This step produces a 256-bit hash (64 hex characters / 32 bytes).
- Take the Last 20 Bytes: The Ethereum address is derived from the last 20 bytes (160 bits / 40 characters / 20 bytes) of the Keccak-256 hash. These 20 bytes represent the "digest" of the public key.
- Add "0x" Prefix: The 20-byte (40 characters) digest is converted to a hexadecimal string, and the "0x" prefix is added to it, forming the 42-character Ethereum address.

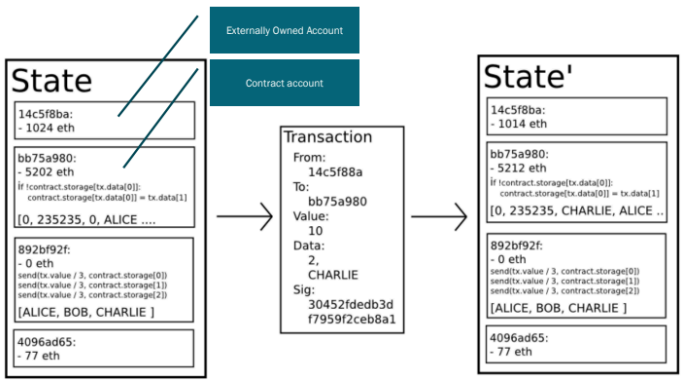
Ethereum addresses use a process that involves **checksums** and **mixed-capitals encoding** to **enhance human intelligibility** and **reduce the risk of address input errors**. Ethereum initially explored ICAP as a method for improving address intelligibility. This approach aimed to express addresses to resemble International Bank Account Numbers (IBANs) but using the non standard country code “XE”. This approach never took off.
Another approach to create checksum addresses is **Ethereum Improvement Proposal 55 (EIP-55)** that works like this:

- The first step is to hash the lowercase version of the Ethereum address (without the "0x" prefix) using the Keccak-256 hash function;
- Capitalize each alphabetic address character if the corresponding hex digit of the hash is greater than or equal to 8 (the uppercase digits in the output are the checksum).

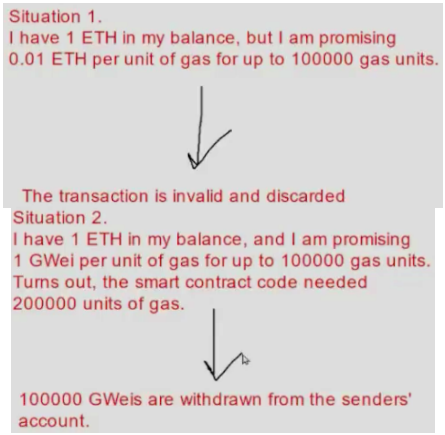
EX. Original Ethereum Address: 001d3f1ef827552ae1114027bd3ecf1f086ba0f9
Keccak-256 Hash (Partial): Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0f9") = 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9695d9a19d8f673ca991deae1
EIP-55 Address: 001d3F1ef827552Ae1114027BD3ECF1f086bA0F9 (to generate it, go through each digit e_i of the original address and if the corresponding hash digit h_i is greater than 8 then capitalize the final EIP-55 address digit o_i). For example e_6=f, h_5=c where c = 12 (in base 16) > 8 so f_6=F.

Ethereum's State Transition System is a model that captures how the blockchain evolves over time through a series of state transitions triggered by transactions.
Let’s see how the transaction execution is performed:

1. **Check if the transaction is well-formed**: The right number of values are there, the signature is valid, the nonce matches the one in the sender’s account (see page 20). Otherwise, return an error.
2. **Compute and apply transaction costs**: Calculate transaction fee, Subtract transaction fee from the sender's account balance and increment the sender’s nonce. If there is not enough balance to spend, return an error.



3. **Initialize gas**: Take off a certain quantity of gas per byte to pay for the bytes in the transaction.
4. **Transfer the transaction value from the sender's account to the receiving account**:
    a. If the receiving account does not yet exist, create it (Contract deployment).
    b. If the receiving account is a contract, run the contract's code (either to completion, or until there is gas left).
5. **Finalize transfer**:
    a. If it failed because the sender did not lock (promise) enough cryptos, or the code execution ran out of gas, then **revert all state changes except the payment of the fees**, add the **fees to the miner's account**.
    b. Otherwise, refund the fees for all remaining gas to the sender and send the fees paid for gas consumed to the miner.



So, **transactions represent the state transition functions** and the result of these functions can be stored (states) where the state is just a snapshot of what is happening in the system. Remember that Blockchain is not a database so we don't store data but just the transaction that leads the data from a state to another. A **"full/archival" node stores all transactions and, for each transaction, also saves the resulting state transitions locally** (the states of the blockchain at that moment) (e.g. think of Git that stores each commit). This is done so that when a client query the state of the blockchain, **it is not necessary to compute the state from the beginning**. Conceptually, blockchain can be separated into:

- Chain data: The list of blocks forming the chain. This is necessary to ensure the cryptographic chaining;
- State data: The result of each transaction's state transition.

Old state data can be discarded (pruning), since it would be useless.

Where are contracts executed?
First, in the mining nodes. If the mining node publishes the block including the transaction to execute the contract on every node.

How is a block validated?
The procedure works like this:

1. Check if the **previous block referenced exists** and is valid;
2. Check that the **timestamp of the block is greater than that of the referenced previous block, and less than a given amount of time into the future**;
3. Check that the block number, difficulty, transaction root, uncle root and gas limit (various low-level Ethereum-specific concepts) are valid;
4. Check that the **Proof of Work on the block is valid**;
5. Let S[0] be the state at the end of the previous block and let TX be the block's transaction list, with n transactions. For each i in [0, n-1], set S[i+1] = APPLY(S[i],TX[i]). If any application returns an error, or if the total gas consumed in the block up until this point exceeds the GASLIMIT, return an error;
6. Let S_FINAL be S[n], but adding the block reward paid to the miner;
7. Check if the hash* of the state S_FINAL is equal to the final state root provided in the block header;
8. If it is then the block is valid, otherwise it is not valid.

Concerning the rewards, the block reward in the Ethereum network was 2 ETH at the time of the information provided (before it was 5 ETH). They also get the contract execution gas (also if the contract runs out of gas) and the ommers (idle blocks - valid blocks that don't win - in Ethereum) even if it is smaller.

The Ethereum network had a mechanism known as the "difficulty bomb" or "ice age." This mechanism involved a gradual increase in the difficulty of mining every 100,000 blocks. As a result, the time between blocks increased, putting pressure on the network to transition to proof-of-stake. The difficulty bomb was intended to make mining progressively more difficult and less attractive, encouraging the move to Ethereum 2.0 (PoS). There were predictions that mining on Ethereum would become increasingly difficult and eventually impossible in 2021. The Merge refers to

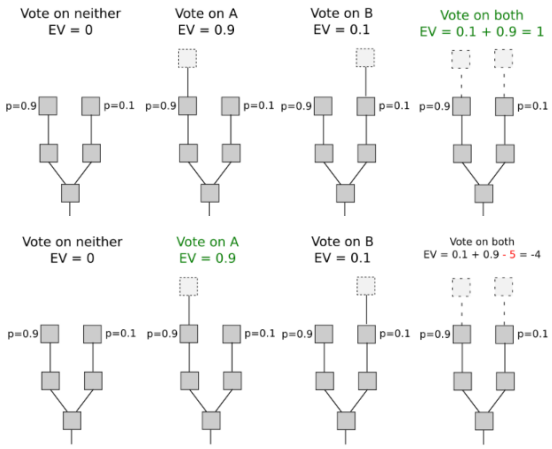
the transition from **Ethereum's proof-of-work consensus mechanism to proof-of-stake**. As of September 15, 2022, The Merge has occurred, and Ethereum has moved away from mining. The difficulty bomb has been removed.

7. A Focus on Consensus 2

Ethash is the **proof-of-work hashing algorithm used in Ethereum**. It is designed to be **memory-hard**, meaning it requires a significant amount of memory to compute, making it resistant to optimization by custom application-specific integrated circuits (ASICs). This is in contrast to Bitcoin, where the use of ASICs has become prevalent. Ethash relies on a **directed acyclic graph (DAG)** file, which is essentially a 1 GB dataset. This dataset is created anew every 125 hours or 30,000 blocks (where this periodic creation is called an **epoch**). This mechanism introduces a level of regularity and adaptability to the mining process. **Ethereum has a shorter block time compared to Bitcoin, with an approximate block time of 15 seconds** and this is beneficial for computation, allowing for quicker transaction confirmations and more frequent updates to the blockchain. Of course this means that also more miners guess the right nonce.

Ethereum stops the miner by increasing the difficulty exponentially inside of proof of work and, since mining pools started growing consistently, the difficulty was rapidly growing as well such that the Ethereum community planned an “end date” for PoW (More than a date, a Terminal Total Difficulty). For this reason Ethereum switched to **Proof-of-Stake**. **Proof-of-Stake (PoS)** is an alternative consensus mechanism to Proof-of-Work (PoW), which is commonly used in blockchain networks like Bitcoin. Instead of miners solving computationally intensive puzzles, PoS relies on participants, called validators, who hold a **stake (investment)** in the cryptocurrency. With proof of stake, we put on **stake some capital** in order to prove that we care about the network (so Stake refers to an amount of crypto-commodities that participants invest in the system). The capital is frozen, and the account can bet on the next block. The rationale behind PoS is that **participants with a significant stake in the network are less likely to want to subvert or compromise the blockchain** in fact the economic incentive is aligned with the security of the network. The more capital I put at stake, the less I want the platform to be destroyed (the more I care). If the block is appended (added to the Blockchain), then **the validator who had bet on that block, gets a reward proportional to the bet**.

One challenge in PoS is the "Nothing at Stake" problem, where validators might place their bets on all potential chains without the risk of losing anything. Basically I can always bet on each branch of the blockchain and never lose (This because I don’t have anything to lose, since I don’t actually lose the bet if the block isn’t approved). Let’s think of the case in which we have 2 branches (soft fork) A and B, why choose? I can put a bet on both. This is sometimes referred to as the "**Tragedy of the Commons**". To address this, PoS systems implement various mechanisms to penalize malicious or equivocating behavior. With proof of stake a punishing mechanism called **slashing** was introduced. In summary, when a validator is caught engaging in malicious behavior, they face penalties in the form of "slashing." The penalties often involve the confiscation of a portion or the entirety of the validator's stake.



Proof of stake has been introduced in **Ethereum 2.0**. The second update to Eth 2.0 are **Shard chains**, which are blockchains that are **side-chains which are maintained by smaller networks**, and from time to time they are synchronized with the master chain. The **beacon chain** serves as the proof-of-stake (PoS) chain in Ethereum 2.0 (so it is the master chain). **The beacon chain acts as a synchronizing backbone for 64 shard chains**, also known as side-chains. Because of the fact that the network is small, shard chains are way faster. For some time both proof of stake blockchain and proof of work were up, until The Merge of 2022, which is the date of the last block mined with proof of work. The final upgrade to Ethereum 2.0 happened in September 2022 with “**The Merge**”.

How does PoS work?

Contrary to Proof of Work, miners put capital at risk by expending energy, In **Proof of Stake validators explicitly stake capital in ETH**. **32 ETH** are put at stake which are deposited in a dedicated smart contract (Not quite a small amount → Approx. € 45,000). If a single account doesn't have the stake to participate in the proof of stake (32ETH), there are **staking pools** (just like mining pools) where multiple accounts combine their stake and split the reward.
Validators are responsible for:

- Checking the validity of propagated blocks on the blockchain. This involves ensuring that the contents of the blocks are correct and adhere to the consensus rules of the network. **Validators are not only responsible for validating the head** (most recent block) of the blockchain but also for **checking the validity of the entire block**. This ensures the integrity and correctness of the entire blockchain, not just the latest state. Validators express their agreement or "vote" for a particular block by sending **attestations** across the network which represents its declaration that a specific block is valid.
- Occasionally **creating and propagating new blocks**. When it is a validator's turn to propose a new block, they assemble a block with valid transactions and other necessary information, then propagate it across the network.

Casper the Friendly Finality Gadget (Casper-FFG) is a consensus mechanism designed to add a **finality layer** to the Ethereum 2.0 blockchain. Casper-FFG upgrades certain blocks to a "finalized" state meaning that they are considered **confirmed and irreversible**, providing a more secure foundation for the blockchain.

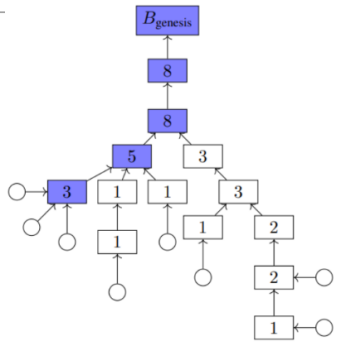
LMD-GHOST is a **choice algorithm** used in Ethereum 2.0, especially when validators have different views of the head of the blockchain. This can occur due to factors such as **network latency or cases where a block proposer has equivocated by proposing two blocks**. **LMD-GHOST helps determine which fork of the blockchain to favor** when there are disagreements among validators. To resolve forks and disagreements, LMD-GHOST identifies the fork that has the **greatest accumulated weight of attestations from validators**. Validators express their agreement with specific blocks through attestations, and the algorithm considers the cumulative weight of these attestations. If multiple messages are received from a validator, LMD-GHOST considers only the latest one. This algorithm is also called **Latest Message-Driven Greedy Heaviest Observed Subtree**.

Algorithm 3.1 LMD GHOST Fork Choice Rule.

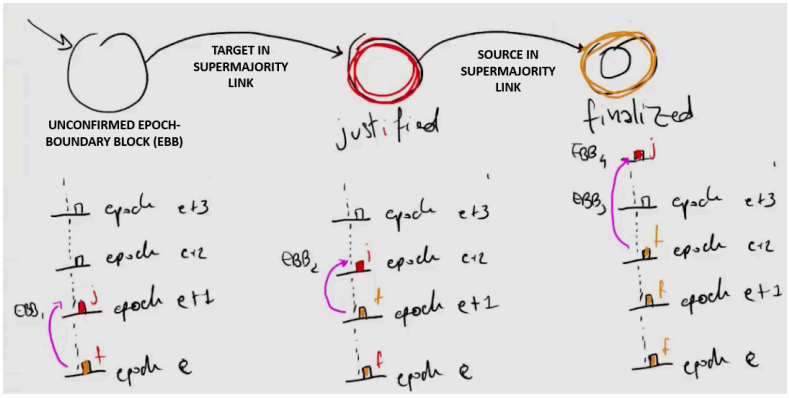
```
1: procedure LMD-GHOST(G)
2:     B ← B_genesis
3:     M ← the most recent attestations of the validators (one per validator)
4:     while B is not a leaf block in G do
5:         B ← arg max_{B' child of B} w(G, B', M)
6:         (ties are broken by hash of the block header)
7:     return B
```

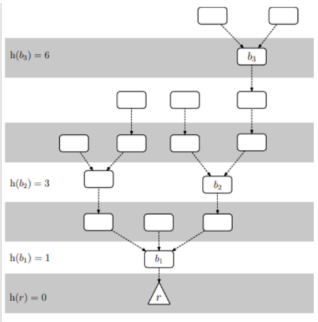
G: View

B: Block; M: set of latest attestations; w: weight

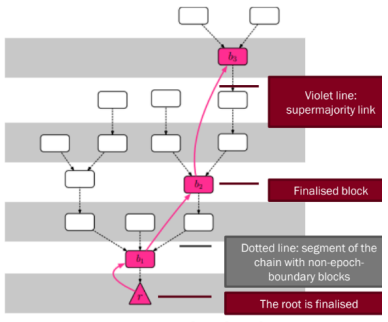


EX. The block on the left has 3 votes, the other ones have 1 and 1 votes. We select the one with 3 votes. Let's go to the following stage in which we have two blocks with 5 votes (notice that we sum all the parent votes so 3+1+1) and the other one has 3 votes so we select the one with 5 votes.





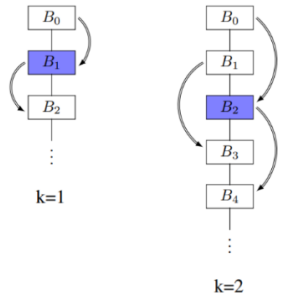
(b) The height function



(c) The justified chain $r \to b_1 \to b_2 \to b_3$

Figure 1: Illustrating a checkpoint tree, the height function, and a justified chain within the checkpoint tree.

Concerning the time and finality in PoS, unlike in Proof of Work (PoW), the system operates in **fixed time slots**, with each slot lasting **12 seconds**. For every slot, a committee of at least **128 validators** is randomly chosen and it plays a role in the block proposal and validation process. **One validator is randomly selected to be the block proposer** in each slot and it has 6 seconds to forge a new block. **Epochs consist of 32 slots, equivalent to 6 minutes and 24 seconds**. The first block in each epoch serves as a **checkpoint**.
Validators vote for **pairs of checkpoints**, with the most recent one (**target**) becoming **justified** if attested with votes from at least 2/3 of the total staked ETH. The least recent checkpoint (source) from the previous epoch is already justified. If it receives 2/3 majority votes, it becomes **finalized** and a finalized block is considered virtually undisputable. Only **epoch-boundary blocks** can be justified and finalized. This refers to the blocks at the beginning of each epoch. A "**supermajority link**" must exist between successive checkpoints A and B. This means that 2/3 of the total staked ETH must vote that checkpoint B is the correct descendant of checkpoint A. This condition is necessary to upgrade A to finalized and B to justified.
In most cases, we have a k-finalization with k=1 meaning that the target becomes finalized and the source for the next EBB. Still, we may have a different chain with, for example, k=2.
EX. In the example on the right, B_0 is the source for the target B_2 and B_1 is the source for target B_3. In this situation B2 is finalized but B_3 is not and neither is B_1. So, as long as I'm in the middle of two supermajority links then the node can be considered finalized.



Now, every 256 epochs (which is approximately every 27 hours) a **sync committee** of 512 validators is randomly assigned. The primary role of the sync committee is to sign block headers for each new slot within the assigned epochs. Block headers contain essential information about a block, such as its hash, timestamp, and other metadata. Light nodes can use the signed block headers from the sync committee for validation purposes. By relying on the signatures from the sync committee, **light nodes can consider the associated blocks as validated and part of the legitimate blockchain**.

To sum up, we have a total of 3 votes:

1. Validators provide attestations, which are votes indicating their agreement on the validity of a particular block. These attestations play a crucial role in the finalization and security of the blockchain.
2. Finalization of blocks, making them virtually undisputable, often requires a supermajority agreement among validators. This supermajority is typically set at 2/3 of the total staked ETH.
3. In addition to regular attestations, there is a sync committee that consists of 512 validators. This committee signs block headers, providing cryptographic evidence for the validity of blocks within specific epochs.

During each slot we have two votes, one for the LMD-GHOST (a single block) that determines what's the next head and one for Casper (a couple of blocks), and a new block is proposed at each round. Sync committees' outputs, represented by the signed block headers, are given continuously for each new slot within the epochs to which the sync committee is assigned. Again, the output is valuable for light nodes, which may not download the entire blockchain but still want to validate transactions. Light clients can use these signed block headers to verify the validity of blocks.

Where are attestations saved in Ethereum?
It's **ephemeral data**. While attestations themselves are not explicitly stored on the blockchain, the effects of attestations, such as finalization of blocks and the determination of validator rewards, are reflected in the state of the blockchain.

Let's now see briefly how the whole process works:

1. Sync Committee Signature Collection: Validators in the sync committee sign block headers for specific slots within the assigned epochs. Each validator contributes its signature to attest to the validity of the block for the given slot.
2. New Block Proposer Selection: A new block proposer is selected from the pool of validators. The proposer is responsible for creating and proposing a new block to be added to the blockchain.
3. Listening for Signatures: The newly selected **proposer listens for the sync committee signatures** related to the specific slots for which it needs to propose a block.

4. Signature Aggregation: The proposer collects all the individual sync committee signatures that match the chain. These signatures are related to the specific slots that the proposer is including in the proposed block.
5. Sync Committee Signature Aggregation: The collected signatures are **aggregated into a final sync committee signature**. Aggregation may involve a cryptographic process to combine multiple signatures into a single signature for efficiency.
6. Block Header Update: The block producer then updates the block header of the proposed block by adding the final sync committee signature. This block header, now including the sync committee signature, is an essential component of the proposed block.
7. Sync Committee Public Keys: Additionally, the block producer includes the public keys of the validators in the sync committee directly into the block header. This information is crucial for later verification of the sync committee signature.
8. Broadcasting the Block: The proposed block, now containing the sync committee signature and relevant information, is broadcasted to the network for validation and inclusion in the blockchain.
9. Network Validation and Finalization: Other validators in the network verify the sync committee signature and the overall validity of the proposed block. If the block is accepted by a sufficient number of validators, it contributes to the finality and security of the blockchain.

Now, let's go into an **Ethereum Block** and see what's inside of it:

- **Randao_reveal**: A pseudo-random value used to select the next block proposer. We cannot use full randomness because otherwise the network won't know who the validators are.
- **Eth1_data**: Information about the **(staking) deposit contract**. The Beacon Chain relies on this information to track and manage staking deposits.
- Graffiti: Arbitrary data used to tag blocks. Graffiti allows validators to include **additional information or messages in the block**, providing a way to add custom data to the blockchain.
- **Proposer_slashings**: A list of validators to be slashed. Proposer slashings specifically refer to penalties for **misbehavior by block proposers**.
- **Attester_slashings**: A list of validators to be slashed. Attester slashings refer to penalties for validators who attest to conflicting information, compromising the integrity of the network.
- **Attestations**: A list of attestations in favor of the current block. Inside of the attestation field we find:
    - Data:
        - Slot: The slot the attestation relates to;
        - Index: Indices for attesting validators;
        - Beacon_block_root: The root hash of the Beacon block containing this object;
        - Source: The last justified checkpoint;
        - Target: The latest epoch boundary block.
    - Signature: Aggregate signature of all attesting validators.
- **Deposits**: A list of new deposits to the deposit contract. Validators deposit a certain amount of cryptocurrency to participate in staking and become validators in the PoS system.
- **Voluntary_exits**: A list of validators exiting the network. Validators can voluntarily exit the network, and this process is initiated by submitting a voluntary exit request.
- Sync_aggregate: A subset of validators used to serve light clients. Sync aggregates allow light clients to efficiently synchronize with the blockchain by obtaining information from a specific subset of validators.

Block body

Field	Description
randao_reveal	a value used to select the next block proposer
eth1_data	information about the (staking) deposit contract
graffiti	arbitrary data used to tag blocks
proposer_slashings	list of validators to be slashed
attester_slashings	list of validators to be slashed
attestations	list of attestations in favor of the current block
deposits	list of new deposits to the deposit contract
voluntary_exits	list of validators exiting the network
sync_aggregate	subset of validators used to serve light clients
execution_payload	transactions passed from the execution client

Zooming in the attestation field

Field		
aggregation_bits	a list of which validators participated in this attestation	
data	slot	the slot the attestation relates to
	index	indices for attesting validators
	beacon_block_root	the root hash of the Beacon block containing this object
	source	the last justified checkpoint
	target	the latest epoch boundary block
signature	aggregate signature of all attesting validators	

Execution payload header

Field	Description
parent_hash	hash of the parent block
fee_recipient	account address for paying transaction fees to
state_root	root hash for the global state after applying changes in this block
receipts_root	hash of the transaction receipts trie
logs_bloom	data structure containing event logs
prev_randao	value used in random validator selection
block_number	the number of the current block
gas_limit	maximum gas allowed in this block
gas_used	the actual amount of gas used in this block
timestamp	the block time
extra_data	arbitrary additional data as raw bytes
base_fee_per_gas	the base fee value
block_hash	Hash of execution block
transactions_root	root hash of the transactions in the payload
withdrawal_root	root hash of the withdrawals in the payload

- **Execution_payload**: Transactions passed from the execution client. In Ethereum, transactions contain information about actions to be performed on the blockchain, and execution_payload refers to the data associated with these transactions.

The **Execution Payload Header** contains various fields that provide essential information about a block in Ethereum. Let's see what's inside of it:

- **Parent_hash**: Hash of the parent block. This links the current block to its parent in the blockchain, forming a chain of blocks.
- **Fee_recipient**: Account address for paying transaction fees. This specifies the recipient of fees collected from transactions in the block.
- **State_root**: Root hash for the global state after applying changes in this block. It represents the cryptographic hash of the state trie after the block's transactions have been processed.
- **Receipts_root**: Hash of the transaction receipts trie. This hash points to the root of the trie structure that stores transaction receipts, including information about the outcomes of transactions.
- Logs_bloom: Data structure containing event logs. The logs bloom is a compact data structure that helps efficiently check whether a particular event log is present in the block.
- Prev_randao: Value used in random validator selection. This is the previous value used in the random number generation process for selecting validators.
- **Block_number**: The number of the current block. It represents the sequential order of the block in the blockchain.
- **Gas_limit**: Maximum gas allowed in this block. Gas limit sets an upper bound on the total computational work that can be performed in the block.
- **Gas_used**: The actual amount of gas used in this block. It represents the total amount of gas consumed by the executed transactions in the block.
- Timestamp: The block time. It indicates the timestamp when the block was created or proposed.
- Extra_data: Arbitrary additional data as raw bytes. Extra data allows for the inclusion of custom information in the block.
- **Base_fee_per_gas**: The base fee value. It represents the minimum fee required per unit of gas. Remember that the **base fee per gas is measured according to how much gas is used in the network** (more gas, more computation, more demand).
- Block_hash: Hash of the execution block. This is the cryptographic hash of the entire execution block.
- **Transactions_root**: Root hash of the transactions in the payload. This hash points to the root of the trie structure that stores the transactions included in the block.
- **Withdrawal_root**: Root hash of the withdrawals in the payload. This hash points to the root of the trie structure that stores information about withdrawals.

Now, the Execution payload field is the same as the payload header, except for the last 2 fields:

- **Withdrawals**: List of withdrawal objects. This field likely contains information about withdrawals made during the execution of the block. Withdrawals typically involve users retrieving their staked funds or rewards from the staking system. Each withdrawal object in the list may include details such as the amount withdrawn, the destination address, and other relevant information.
- **Transactions**: List of transactions to be executed. Similar to the transactions_root field in the header, this field provides a list of transactions that are part of the block's payload. Transactions include actions or operations initiated by users, such as transferring funds or interacting with smart contracts. Each transaction object in the list contains details about the transaction, including the sender, recipient, amount, and transaction data.

Note: After the **Shanghai/Capella upgrade** (April 2023), you can withdraw the ethers that are beyond the 32ETH. The root of the withdrawal will go into the withdrawal_root field.

So, what are the total rewards a miner can obtain?

- **Attestation Rewards**: Validators are rewarded for providing attestations, which are votes indicating their agreement on the validity of a particular block. The timeliness of the attestation affects the reward. The faster a validator provides an attestation within the allocated slots, the higher the reward. In particular:

    - **Correct head attestation**: Validators receive rewards for providing correct attestations for the head of the blockchain. This reinforces the importance of accurately identifying the current state of the blockchain.
    - **Correct (checkpoint) source attestation**: Rewards are given for correct attestations regarding the source checkpoint. Validators are incentivized to provide accurate historical information about the blockchain.
    - **Correct (checkpoint) target attestation**: Validators are rewarded for correct attestations regarding the target checkpoint. This reinforces the importance of accurate predictions and agreement on the future state of the blockchain.
- **Sync Committee Reward**: Validators who participate in the sync committee, signing block headers within specific epochs, receive rewards. This encourages validators to contribute to the efficiency of light clients relying on the signed block headers.
- **Proposer Reward**: Validators selected as block proposers and successfully include attestations for the head, source, target, and sync committee output in the proposed block receive additional rewards.

Before, we said that dishonest behaviors are prone to **shashing**. Some possible malicious behaviors are:

- **Double proposal (equivocation)**: **A proposer signs two different beacon blocks for the same slot** (If the proposer does not propose a block, it misses the chance. If this case no slashing is applied);
- **FFG double vote**: An attester signs **two checkpoint attestations for the same target epoch**;
- **An attester signs a checkpoint attestation that "surrounds" another one**: The source epoch of the first block precedes the source epoch of the second, but the target epoch of the first block follows the target epoch of the second. This would cause a situation where the **attester is contradicting** what a **validator already said was finalized in a previous attestation**.

Validators who observe dishonest behavior can act as **whistleblowers**. They propagate a specific message containing details of the offense, allowing a proposer to include it in a block. Both the proposer and the whistleblower receive rewards for reporting and exposing the dishonest behavior. **Validators engaging in dishonest behavior are subject to slashing, and penalties continue until they eventually exit the beacon chain**.
Notice that we also talked about penalties. Some possible **penalties** (not slashing!) are:

- **Inactivity Penalty**: It is activated if the chain fails to finalize for more than 4 epochs. It subtracts staked ETH from validators voting against the majority, helping the majority regain a 2/3 majority to finalize the chain.
- **Sync Committee Penalty**: Validators can face penalties for **not participating in the sync committee** or **for signing the wrong head block**.

Some other considerations about the committee structure:

- Every epoch all active validators get a chance to attest once.
- The assumption is, **strictly less than 1⁄3 of validators are byzantine**.
- A validator can only be in **one committee per epoch**.
    - We have a 128+ members committee per slot.
    - Committees in an epoch do not have validators in common.
    - We should have at the very least 4096 validators.

  Why 128 validators?

  The committee size is chosen to be 128, which is **a power-of-2 round-up of 111**. This 111 value is likely determined based on mathematical considerations related to the desired probability of not picking 2/3 malicious validators in the committee when 1/3 of the validators are attackers:

  $$\sum_{k=\left\lfloor \frac{2}{3}n \right\rfloor}^{n} \binom{n}{k} \times \left(\frac{1}{3}\right)^{k} \times \left(\frac{2}{3}\right)^{n-k} \leq r$$

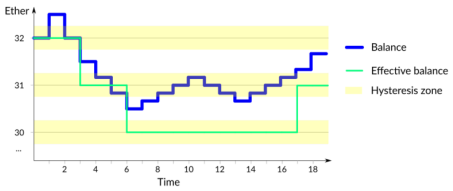
  With:

    - n Attesters: Represents the total number of validators.
    - k Selected Validators: The number of validators selected in the committee.
    - r Desired Probability: The desired probability not to pick 2/3 malicious validators when 1/3 of the validators are attackers.
- **Correlation Penalty**: The protocol imposes an additional penalty called the correlation penalty based on how many others have been slashed near the same time. The basic formula for the additional penalty is

validator_balance * 3 * fraction_of_validators_slashed. This penalty aims to discourage coordinated or correlated malicious activities among validators. If 1/3 of all validators commit a slashable offense in a similar period, they **could lose their entire balance (their deposits are burned)**. This creates a significant economic disincentive for validators to engage in malicious activities collectively. The cost of reverting a block with such coordinated malicious behavior could be substantial, approximately ~4 million ETH.

- **Voluntary Exit**: Validators can choose to **voluntarily exit** after 2048 epochs (approximately 9 days). In any voluntary or forced exit, there is a delay of four epochs before stakers can withdraw their stake. During this delay, a validator can still be caught and slashed if they have committed any slashable offenses.
- **Slashed Validator's Penalty Timeline**: A slashed validator incurs a delay of 8192 epochs (approximately 36 days) before being fully ejected from the network. Immediate penalty (up to 1 ETH) is applied on Day 1 Correlation penalty is applied on Day 18. The validator is ejected from the network on Day 36.

In Proof of Stake (PoS), validators explicitly stake a capital amount in ETH, initially set at 32 ETH. The **effective balance** is introduced as a concept to understand **the influence of a validator's balance on their decision power and rewards**. Here are key points regarding the effective balance:



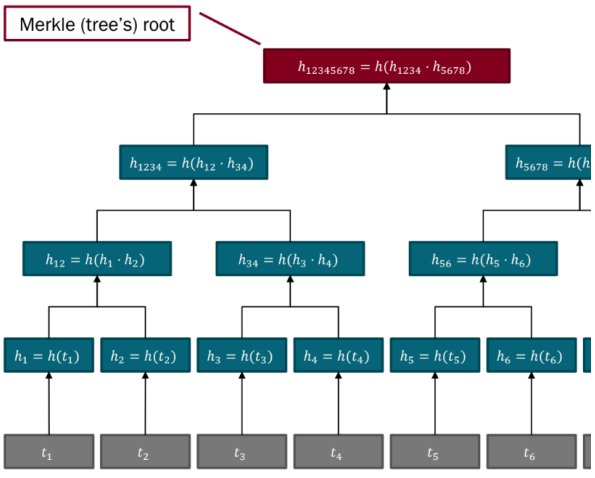
- Never More than 32 ETH: Regardless of the validator's actual balance, the effective balance is capped at 32 ETH.
- Always an Integer: The effective balance is always rounded down to the nearest integer. For example, a validator with a balance of 29.7 ETH would have an effective balance of 29 ETH.
- Increases with a Raise by 1.25 or More ETH: The effective balance increases if the validator raises their balance by 1.25 ETH or more compared to the current effective balance. This introduces a threshold for balance increases.
- Decreases with a Shrink by 0.25+ ETH: The effective balance decreases if the validator's balance is reduced by 0.25 ETH or more compared to the current effective balance. This introduces a threshold for balance decreases.

The effective balance concept ensures that **having more than 32 ETH does not provide extra decision power or additional rewards** beyond what is achieved with the capped effective balance.
EX. The effectiveness of a validator is calculated as the ratio of the effective balance to the validator's actual balance. For example, if a validator has an effective balance of 29 ETH and an actual balance of 29.7 ETH, the effectiveness would be approximately 97.54%.

8. Hashing

Hashing returns a fully random (numeric) code. It's deterministic so given a certain value to hash, if we give it again we'll receive the same digest. In practice, given a certain string of any length in input (a message) that can be any kind of file (e.g. a file, some text, an image, etc..), we get a fixed-length hash value in output (a digest). There is no secret key involved and all operations are public.

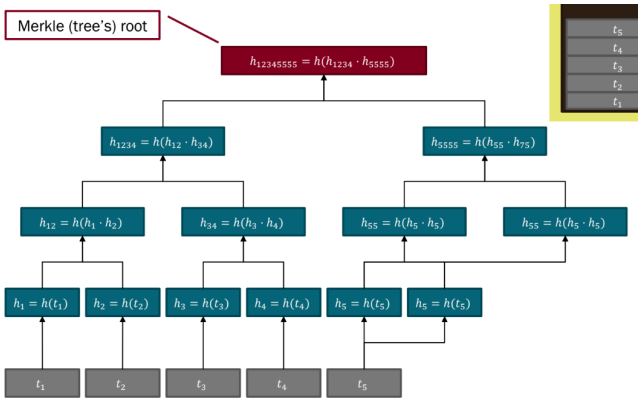
In Bitcoin and many other blockchain systems, Merkle trees and Merkle roots are used to efficiently represent and verify the integrity of a large set of data, such as transactions in a block. A **Merkle tree**, also known as a **binary hash tree**, is a tree structure in which every **leaf node is labeled with the hash of a data block**, and every **non-leaf node is labeled with the cryptographic hash of the labels of its child nodes**. It is constructed by recursively hashing pairs of nodes until a single root hash, known as the Merkle root, is obtained.



How does the Merkle proof work in Bitcoin?
Suppose a light node wants to verify the inclusion of a specific transaction t_2 in a block. Instead of downloading the entire Merkle tree, **the light node requests a Merkle proof from a full archival node**. The full archival node sends a Merkle proof to the light node.

This proof consists of the necessary hashes to construct the path from t_2 to the Merkle root. In this case, the hashes h_1, h_{34} and h_{5678} are provided. The light node uses the received Merkle proof to reconstruct the hash path from t_2 to the Merkle root. If the calculated Merkle root matches the one in the block header, the light node can be confident that t_2 is indeed included in the Merkle tree.

Note: Before we assumed to have 2^n transactions. In that case we can replicate the hashes to get to the same structure as before.



What about Ethereum?

Binary Merkle trees are perfectly suitable for list-like structures. In Ethereum, the structure for organizing and efficiently retrieving data is more complex than a straightforward binary Merkle tree. Ethereum uses three **Patricia radix trees** (or **tries**) to organize different types of data. These three tries are rooted at specific values in a block's header:

- State Root: The state trie (Patricia trie) stores the current state of Ethereum, including account balances and contract storage. It provides an efficient way to look up the current state of an account, including its balance and contract storage data.
- Receipts Root: The receipts trie contains transaction receipts for all transactions included in the block. Each transaction in Ethereum produces a receipt, which includes information such as logs (events) emitted during the transaction, the cumulative gas used, and the status of the transaction (success or failure).
- Transaction Root: The transactions trie stores information about the transactions included in the block. This includes details like sender and receiver addresses, gas limits, and the transaction data. The root of this trie is often referred to as the "transactions root".

Now, let's see the properties of a hashing function. A cryptographic hash function h must be able to withstand known types of cryptanalytic attack. The main properties are:

- **Preimage resistance**: Given a hash k it should be difficult to find any message m such that k = h(m). Functions that lack this property are vulnerable to preimage attacks.
- **Second-preimage resistance** (a.k.a. weak collision resistance): Given an input m_1 it should be difficult to find another input $m_2 \neq m_1$ such that $h(m_1) = h(m_2)$. Functions that lack this property are vulnerable to second-preimage attacks.
- **Collision resistance** (a.k.a. strong collision resistance): It should be difficult to find two different messages m_1 and m_2 such that $m_2 \neq m_1$ and $h(m_1) = h(m_2)$. The pair (m_1, m_2) is called **cryptographic hash collision**.

Bitcoin adopts SHA-256 by default for hashing operations and RIPEMD-160 for addresses. The digest size of RIPEMD-160 is 2^{160} which is about 1.46×10^{48}. The digest size of SHA-256 is 2^{256} which is about 1.16×10^{77}. In general, the higher the digest size, the better (the more difficult to find a collision).

The **SHA-1** is a cryptographic hash function that takes an input and produces a fixed-size (160-bit) hash value (or 40-character hexadecimal number). Let's see how it works:

- Padding: The input message is padded to a length that is a multiple of 512 bits. Padding involves adding a single '1' bit followed by a series of '0' bits, and finally appending the length of the original message (in bits).
- Initialization: Five 32-bit variables (A, B, C, D, E) are initialized to specific constant values: A = 0x67452301, B = 0xEFCDAB89, C = 0x98BADCFE, D = 0x10325476 and E = 0xC3D2E1F0.
- Processing Blocks: The padded message is divided into 512-bit blocks. Each block is further divided into sixteen 32-bit words.
- Message Schedule: A **message schedule array of 80 words is created**. For the first 16 words, they are directly obtained from the current block (they are taken as-is). For the remaining 64 words, they are generated using a specific formula involving bitwise operations (they are generated as a mega-mess-up of bits of the previous ones, in a chain).
- Main Loop: The main loop consists of 80 iterations. For each iteration, the five variables (A, B, C, D, E) are updated based on the message schedule and various logical and bitwise operations.

- Final Hash: The final hash value is the concatenation of the five variables (A, B, C, D, E) after processing all blocks. The hash value is typically represented as a 40-character hexadecimal number.

9. Introduction to Solidity

In the Blockchain, a token can be used in several applications:

- Currency: Used as a medium of exchange for goods and services.
- Commodity: Represents a basic good that is tradeable or exchangeable with other goods of the same type.
- Utility: Acts as a satisfaction quantifier for an economic good or service.
- Security: Functions as a financial instrument that guarantees ownership, credit, or decision power.

They can be:

- **Fungible**: Individual units can be mutually substituted. Examples include traditional currencies like dollars or cryptocurrencies like Bitcoin.
- **Non-fungible**: Units are unique and not interchangeable. Examples include non-fungible tokens (NFTs) that represent unique digital assets.
- **Semi-fungible**: Fungible until redeemed or expired. Semi-fungible tokens have characteristics of both fungible and non-fungible tokens. While each token is unique, certain aspects of the tokens may be interchangeable or have a standardized value. This category allows for fungibility within a certain context or until a specific condition is met.

Tokens and coins are related concepts in the context of blockchain and cryptocurrency, but they are not necessarily the same. Let's clarify the difference between tokens and coins:

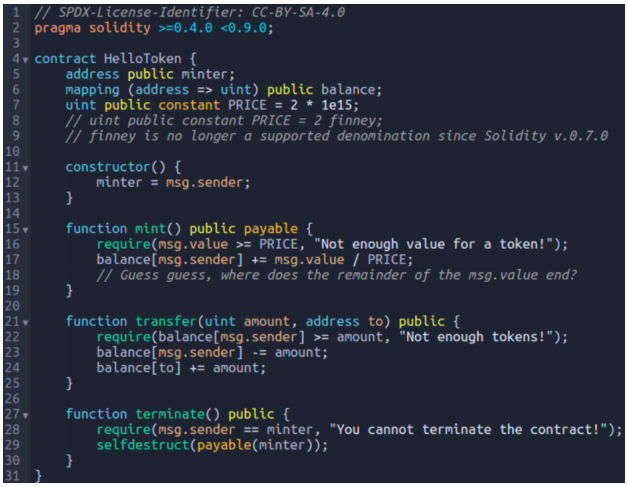
- Coins: Coins typically refer to native digital currencies that operate on their own blockchain. These coins have their own independent blockchains and are used as a medium of exchange within their respective networks.
- Tokens: They represent assets or value on existing blockchains. They are created through a process called tokenization and are often used to represent various assets, such as real-world assets, digital assets, or access to a particular application or service. While some tokens may serve as a form of digital currency, many tokens have specific use cases beyond being a medium of exchange.

A **DApp**, short for **decentralized application**, is a type of software application that operates on a decentralized network or blockchain. Unlike traditional applications that run on centralized servers, Dapps leverage the decentralized and distributed nature of blockchain technology. Key characteristics of Dapps include:

- Decentralization: Dapps operate on a decentralized network of computers, often using blockchain technology. This means there is no single point of control or failure, enhancing security and resilience.
- Blockchain Integration: Dapps leverage blockchain technology to store and manage data, ensuring transparency, immutability, and security. Smart contracts, which are self-executing contracts with the terms of the agreement directly written into code, often play a central role in Dapp functionality.
- Token Integration: Many Dapps have native tokens or integrate existing cryptocurrencies to facilitate transactions within the application or to incentivize users.
- Autonomous Operation: Dapps aim to operate autonomously, with code execution and data storage occurring on the decentralized network without the need for a central authority.

Now, let's finally see how to define a contract:

- **"pragma solidity"** is a statement in the Solidity programming language, which is commonly used for developing smart contracts on blockchain platforms. The **pragma statement** is used to specify the **version of the Solidity compiler** that should be used for compiling the smart contract;
- The **"contract"** keyword is used to define a contract;



```
// SPDX-License-Identifier: CC-BY-SA-4.0
pragma solidity >=0.4.0 <0.9.0;

contract HelloToken {
    address public minter;
    mapping (address => uint) public balance;
    uint public constant PRICE = 2 * 1e15;
    // uint public constant PRICE = 2 finney;
    // finney is no longer a supported denomination since Solidity v.0.7.0

    constructor() {
        minter = msg.sender;
    }

    function mint() public payable {
        require(msg.value >= PRICE, "Not enough value for a token!");
        balance[msg.sender] += msg.value / PRICE;
        // Guess guess, where does the remainder of the msg.value end?
    }

    function transfer(uint amount, address to) public {
        require(balance[msg.sender] >= amount, "Not enough tokens!");
        balance[msg.sender] -= amount;
        balance[to] += amount;
    }

    function terminate() public {
        require(msg.sender == minter, "You cannot terminate the contract!");
        selfdestruct(payable(minter));
    }
}
```

- Smart Contract accounts and EOA are stored using the "**account"** data type;
- Smart contract methods begin with the "**function**" keyword. Notice that in the function statement we can declare:
    - **Visibility** (**public, external, internal, or private**): These modifiers determine the visibility of the function. In particular:
        - **public**: Visible within and from outside the contract. It creates a getter function for storage/state variables.
        - **external**: (only for methods) visible only from outside the contract. To call an external method f from within a contract, use this.f(). It is not a great strategy as it translates to a CALL op (extra gas!).
        - **internal (default)**: Only this contract and contracts deriving from it can access.
        - **private**: can be accessed only from this contract.
    - **State Mutability** (**view, pure, payable**): These modifiers indicate the state mutability of the function. **view and pure functions do not modify the state**, while **payable functions can receive Ether**.
    - Function name: It has a shape **function()** where we can insert some **parameters**.
      EX. transfer(uint amount, address to) where transfer is the name of the function, amount is the integer amount of coins to transfer and to is the account to which to transfer the coins (notice the account data type).
    - Return type: It is also possible to specify a **return type**.

EX. In the example we basically define a HelloCoin (a new token). How does it work?

- Every HelloToken costs 2 finney (2/1000 ETH = 2×10^{15} wei) and the minimum threshold is 2 finney. If I want HelloTokens, I send ETH to the contract. If I send the contract 11 finney, I get 5 HelloTokens in my balance. If I send 1 finney, my request is refused.
- We define a new contract called Hello Token with a price (needed for the generation of a HelloToken) of 2^{15} (2 finney). In the constructor we define the minter that is the message sender (**msg.sender**) so it is the identity (address) of the one who calls the function. **msg.value** is the amount of wei (a fraction of ETH) attached.
- To mint a new HelloCoin, we have to check that the wei provided are higher than the ones required (variable PRICE). balance[msg.sender] += msg.value / PRICE calculates the number of tokens to mint based on the Ether sent and the specified price. It divides the amount of Ether (msg.value) by the price (PRICE) and adds the result to the balance of the sender (msg.sender). The division operation here calculates how many tokens the user should receive based on the Ether amount and the token price.
- To transfer the amount to tokens calculated, the function takes two parameters that are amount (the number of tokens to transfer) and to (the address to which the tokens are being transferred). require(balance[msg.sender] >= amount, "error") checks if the sender's (msg.sender) balance is greater than or equal to the amount being transferred (amount). If the condition is not met, the function will revert with the error message. This ensures that the sender has enough balance to perform the transfer. Then the balance is transferred from the sender account and added to the recipient.
- The last function is a simple self-destruction one so the contract sends its belongings to the account that deployed it.

Note: **"selfdestruct" was replaced with "deactivate"**. "selfdestruct" continues to behave almost the same as today, but instead of removing the account, it will leave most properties of the account intact, with the exception of two:

- Transfer all value and set balance to 0;
- Set "nonce" (of the smart contract) to 2^{64} (so that the execution fails - in EIP-2681 the account.nonce field can never exceed 2^{64}-1).

It's important to note that once a contract is self-destructed, its code and storage are removed from the Ethereum state, and the contract becomes inaccessible. Therefore, extreme caution should be exercised when using "selfdestruct", and it should only be used when it is absolutely certain that the contract is no longer needed.

The **owner of the Smart Contract is the creator** that writes the code and sends a transaction to the blockchain with that code attached. Once deployed, the Smart Contract now has **a unique address & balance**. When a smart contract is deployed, **a transaction is created with 0x0** (or 0x0000000000000000000000000000000000000000) as

the destination address (this signifies that the transaction is creating a new contract). The **bytecode of the new contract is included in the transaction data payload**. The **new account address is computed deterministically** based on the deployer's account address and nonce. The process involves taking the **Keccak-256 hash (SHA-3) of the RLP (Recursive Length Prefix) encoding** of a specific data structure. The data structure is formed by concatenating the **deployer's account address** with the **deployer's account nonce** (so we can know the contract's address in advance by knowing those two values). This is done in a specific way, and the result is then hashed. **EIP-55 encoding** (mixed-case encoding) is applied to the 20-byte suffix of the hash, resulting in the final Ethereum address of the deployed contract. Remember that **contracts can't sign transactions since they don't have a private key**!
Note: It is possible to attach some ETH while creating a smart contract and they will be burned but this is not the "official" way to burn coins since another dedicated address exists. Do not confuse the 0x0 address with the **burn address** (used to burn ETH) that is 0x000000000000000000000000000000000000dEaD.
Who gets the ETH of the contract?
The ETH are stored in the Smart Contract account. The creator can get the ETH upon the self-destruction of the contract

Some common value types is Solidity are:

- **Value Types Without Members**:
  - **Booleans (bool)**: Represents true or false.
  - **Signed and Unsigned Integers (int, uint)**:
    - int: Signed integer type.
    - uint: Unsigned integer type. The number of bits can be specified (e.g., uint8, uint16).
- **Value Types With Members**:
  - **Fixed-Size Byte-Arrays (bytes1 to bytes32)**: Represents a fixed-size sequence of bytes. Members include length to get the size of the byte array.
  - **Addresses (address)**: Represents a 20-byte Ethereum address. Notable members include:
    - balance: Query the balance of an address.
    - transfer(uint): Function to transfer cryptocurrency to the address from the calling contract.

Some reference types are:

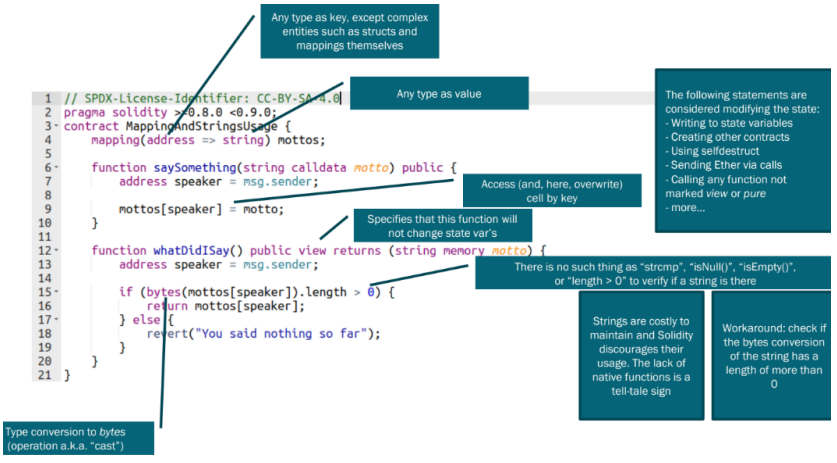
- **String literals (string)**: They can be written with double quotes or single quotes ("foo" or 'bar'). This is an **expensive data structure** as it lacks of native function so it is typically avoided. For example, if we want to check the length of a string, we can check if the bytes conversion of the string has a length of more than 0.
- **Arrays**: Given a type T, indicate the type as T[x] where x is the fixed length, or T[] in case of variable length. Access the k-th element of an array a as a[k]. Notice that bytes should always be preferred over byte[] because it is cheaper.
- **Mappings**: Given a type K (key), and another type V (value), the type is indicated as mapping(K => V).

Note: The following statements are considered modifying the state:

- Writing to state variables;
- Creating other contracts;
- Using selfdestruct;
- Sending Ether via calls;
- Calling any function not marked view or pure.

Let's see another example to go better understand how dictionaries, view and strings work:

- The first row represents the **license** (copyright) that tells users how they can use the code.
- The 4th line "mapping(address => string) mottos" creates a state variable mottos that can be used to associate



string values (mottos) with Ethereum addresses. For example, you might use this mapping to store personalized messages or mottos for each Ethereum address in a smart contract.

- We use the function in the 6th line to overwrite the given motto for the speaker that is the message sender.
- Notice that the function “whatDidISay()” has a **view state mutability meaning that it doesn’t change the state** and only limits itself to look at the current state. In that function we check it the length (in bytes) of the motto greater than 0 (notice that **there is no such thing as “strcmp”, “isNull()”, “isEmpty()”, or “length > 0” to verify if a string is there**) by converting it to bytes.
- “Revert” matches with “throw” in Java (so an error happens, the throw function is invoked).

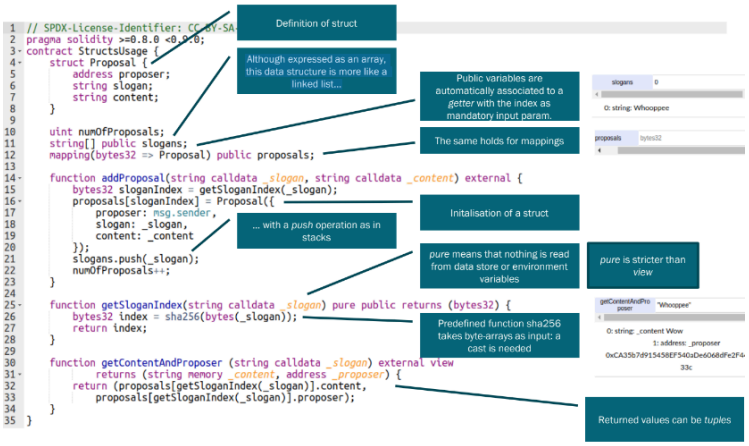
Before Solidity version 0.5.0, **the data location for variables**, especially reference types (arrays, structs, and mappings), **was implicitly determined**. Starting from version 0.5.0, specifying data locations became mandatory in function parameters. The different data types (from the cheapest to the most expensive) are:

- **Calldata**: Used for function arguments. It is **non-modifiable and non-persistent**. It is limited to the execution of the method and it usually represents the input data to a function.
- **Memory**: Used for variables that are temporary and local to a function. It is **modifiable within the function** but it is **non-persistent** (contents are not stored on the blockchain) so it is limited to the execution of the method.
- **Storage**: Used for state variables. It is **modifiable and persistent** (contents are stored on the blockchain). It exists throughout the lifetime of the smart contract. State variables retain their values between function calls and across the lifetime of the contract.


```
{
	"inputs": [
		{
			"internalType": "string",
			"name": "motto",
			"type": "string"
		}
	],
	"name": "saySomething",
	"outputs": [],
	"stateMutability": "nonpayable",
	"type": "function"
}
```


Note: When we compile a Solidity contract (for example using Remix) we are also offered an **ABI** which is a **descriptor of the methods publicly offered by the contract**. Moreover, in Remix we can easily retrieve the transaction and execution costs, and the bytecode for the input (also the payload of the contract). In general, **the payload of the transaction when deploying a contract is the bytecode of the contract**. Once we deploy a new contract and call a certain function (with the button “transact”), we are shown the “from” field with the address of the account that deployed the contract (from which a certain amount of gas will be deducted). **Function selector** is the process of knowing which function is the machine invoking on the contract. In ethereum it works by **hashing the name of the method and the input data types**, and **taking the first 4 bytes**.
EX. Hence, if we want to get the function selector for “whatDidISay()” function, we compute its Kekkac-256 and take the first 4 bytes of the hash. Notice that before we didn’t include any arguments. If we want to call a function with some arguments we have to include them in the hash (e.g. whatDidISay(string)) and always take the first 4 bytes. When you send a transaction to a smart contract that invokes a specific function, you include the function selector along with the encoded arguments in the transaction data.
So, the **payload** (input data) is composed of the **function selector** and the **encoded arguments**, following the **Ethereum ABI** (Application Binary Interface). In summary we find:

- Function Selector: The first 4 bytes (32 bits) of the payload represent the function selector. This is obtained by hashing the function name and its input parameter types (excluding the return types) using the Keccak-256 (SHA-3) algorithm and taking the first 4 bytes of the hash.
- Encoded Arguments: Following the function selector, the remaining bytes in the payload represent the encoded arguments. Each argument is encoded according to the Ethereum ABI, which defines a standard for encoding and decoding data to be sent between Ethereum contracts. Simple types (like integers, booleans, and fixed-size byte arrays) are encoded directly according to their type. Complex types (like dynamic arrays and structs) have more complex encoding rules.



In Solidity:

- A **struct** is a user-defined composite data type that allows you to group together different data types under a single name. Think of a struct as a record or a data structure that enables you to create more complex and organized data types. Although expressed as an array, this data structure is more like a linked list.
- An **enum** is another user-defined data type that allows you to create a set of named constant values. Enums are often used to represent a set of distinct options or states within a contract. Each value in the enum is associated with an integer index, starting from 0 for the first value.
- An **event** is a way for a contract to communicate with the external world or to notify external observers about specific occurrences within the contract. Events are declared in the contract and can be emitted using the emit keyword within functions. Events are declared using the event keyword, followed by the event name and parameters. Parameters define the information that will be logged when the event is emitted. Events are often used to log significant state changes, transactions, or other important activities in the contract.
- A **modifier** is a reusable piece of code that can be applied to one or more functions in a contract. Modifiers allow you to enforce conditions before and after the execution of a function. Modifiers are declared using the modifier keyword. Modifiers are often used for access control, validation, or other common conditions that need to be checked before executing certain functions.

10. Decentralized Applications

The terms Web1.0, Web2.0, and Web3.0 refer to different stages or phases in the development of the World Wide Web, each characterized by specific technological advancements and paradigms. In particular:

- Web 1.0: Web1 was the initial phase of the World Wide Web when content was primarily static and read-only. Websites were basic, providing information but lacking user interactivity. HTML was the main technology used, and there was a limited focus on user participation.
- Web2.0: Web2 marked a shift towards dynamic and interactive content. Social media platforms, blogs, and forums became prevalent. User-generated content and collaboration were key features. Technologies such as JavaScript, CSS, and AJAX played a significant role.
- Web3.0: Web3 represents the vision of a more decentralized, user-centric web. It is associated with technologies such as blockchain, decentralized protocols, and peer-to-peer networks. Web3 aims to empower users by giving them more control over their data and reducing reliance on centralized entities.

DApps are applications built on blockchain technology, typically leveraging smart contracts. They run on a **decentralized network** of computers rather than a central server. DApps aim to provide transparency, security, and trustlessness. They are a key component of the Web3 vision.

Ganache is a **re-implementation of the ethereum blockchain**, which can run entirely locally. It's used as a development environment to test the DApp. Of course the blockchain is emulated. Some features are:

- **Local Blockchain**: Ganache provides a local, in-memory blockchain that you can run on your machine. This allows you to interact with the blockchain without the need for a real network or the costs associated with deploying contracts on the Ethereum mainnet. Basically the blockchain is like running on a single node.
- **Instant Mining**: Blocks are instantly mined on Ganache, making development and testing faster. This is in contrast to the Ethereum mainnet, where mining can take some time.
- **Ether Faucet**: Ganache comes with a built-in Ether faucet, allowing you to easily send Ether to any account on your local blockchain. This makes it convenient for testing transactions and interactions within your DApp.
- **User Interface**: Ganache provides a user interface that displays information about the blocks being mined, transactions, and accounts. It allows you to inspect the state of the blockchain and see how your contracts interact with it.
- **Integrations**: It is possible to integrate it with other tools like Truffle or even Remix by inserting the host and port or where Ganasce is running.

Web3.js is a JavaScript library that provides a set of tools for interacting with the Ethereum blockchain from a web application. Basically it offers the possibility to access the blockchain using some APIs. Web3.js knows how to interact with the smart contracts thanks to the contract's ABI. In essence, it is a JS object (henceforth, web3):

- Offering handy methods such as:
    - V. 0.20: web3.fromAscii(string);
    - V. 1.8: web3.utils.asciiToHex(string);
- To access to the (an) Ethereum blockchain, so (e.g., accounts):
    - V. 0.20: web3.eth.accounts;
    - V. 1.8: web3.eth.getAccounts();
- Or contracts:
    - V. 0.20: web3.eth.contract(cntrAbi).at(cntraddress);
    - V. 1.8: new web3.eth.Contract(cntrAbi, cntrAddress).

Note: Remember that the **ABI** (Application Binary Interface) refers to the interface that defines how to interact with a smart contract. It specifies the methods (functions) that a smart contract exposes and the data types of their parameters and return values. The ABI is crucial for enabling communication between different software modules, such as between Ethereum smart contracts and applications. Remember that in ABI, **public variables are exposed as methods with no parameters**. As a consequence, when creating a public variable, it is automatically associated with a getter.
Note: Web3.js, just like other frameworks, avoids at all cost for the user to insert the private key by hand, and that is taken directly from ganache in development and from the wallet connected during production.
Note: In the frontend, when using JS, we may need to execute:

- .call() doesn't send a transaction to the ethereum network, but it's just a way of calling the method in the local machine. It's done in order to test the function locally.
- .send() is used to send the transaction to the blockchain. From this we receive the confirmation number and the receipt.

There are some differences between web accounts, Ethereum accounts, and nodes:

- **Web Account (Authenticated via username and password)**: This type of account is the standard user authentication mechanism for web applications. Users are identified by a username and authenticated using a password. This is suitable for typical web application users who access centralized services where authentication and authorization are managed by a centralized server.
  EX. In an online game with token purchases, game players who want to access their profiles, achievements, and possibly make in-game purchases.
- **Ethereum Account (Identified by an Ethereum account address)**: Ethereum accounts are addresses on the Ethereum blockchain associated with a private key. Users sign transactions with their private keys to interact with smart contracts or transfer Ether. Ideal for users who need to engage in decentralized applications (DApps) on the Ethereum blockchain, sign transactions, or make payments in cryptocurrency.
  EX. In an Online Game, users buying/selling in-game tokens directly using their Ethereum accounts. In Used Oil Consignments Tracing, this might not be the primary user type unless the tracing involves decentralized and transparent processes on the Ethereum blockchain.
- **Node (Running an Ethereum node)**: An Ethereum node is a software client that participates in the Ethereum network, maintaining a copy of the entire blockchain and validating transactions. Typically, this user type is more technical and involves individuals or organizations running nodes to support the Ethereum network's infrastructure. Regular DApp users do not need to run nodes.
  EX. Developers building and deploying smart contracts might run nodes to interact with the Ethereum network. Mining pools operate nodes to contribute to the consensus mechanism and mine new blocks.

Note: Running an ethereum node is not necessary for a DApp user. If the user has an ethereum wallet, it doesn't mean that they're running an ethereum node.

Events are a way for a contract to communicate that something significant has occurred within its execution. Events are part of the Ethereum Virtual Machine (EVM) and serve as a mechanism for emitting information that external applications or other smart contracts can capture and react to. Events are commonly used for various purposes in Ethereum development, such as:

- User Notifications: Events can be used to notify users or other contracts about specific actions or state changes within a smart contract.
- Decentralized Application (DApp) Updates: DApps can listen for events emitted by smart contracts to update their user interfaces in response to on-chain activities.

- Audit Trail: Events serve as an on-chain audit trail, providing a transparent and verifiable history of important contract actions.

So, Events in solidity are very important, since they allow a **Publisher-Subscriber scheme**, meaning that the frontend application can react to an event if it's subscribed to it.

Truffle is a development framework for Ethereum that simplifies the process of building, testing, and deploying decentralized applications (DApps) and smart contracts. It provides a suite of tools and utilities to streamline the development workflow, making it easier for developers to work with the Ethereum blockchain. If you restart the ganache, it will lose all the work. Truffle is an environment that does a lot of things like compiling and deploying smart contracts, but also keeping the ganache state persistent by using **workspaces**.

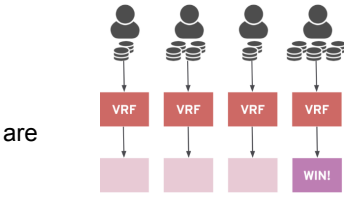
11. Algorand

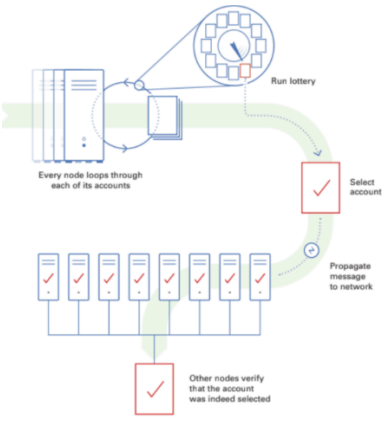
Many blockchains such as Bitcoin and Ethereum (before The Merge) used **Proof Of Work** as a consensus mechanism. PoW has some limitations like:

- Huge electrical consumption: Mining operations require powerful hardware to solve complex mathematical problems, and the competition among miners to solve these problems first and add a new block to the blockchain involves a considerable amount of computational work.
- Concentration of governance (in few mining farms): As the mining process becomes more competitive, mining operations tend to consolidate in areas with cheap electricity and access to specialized hardware. This concentration of mining power in a few entities or mining farms raises concerns about the decentralization and security of the blockchain network.
- Soft forking of the blockchain: Due to possibility of having different branches.

Another consensus mechanism is the **Proof of Stake**. In general:

- **Bonded proof of stake**: Validators bind their stake, to show their commitment in validating and appending a new block. Misbehaviors are punished. The problems are that:
  - Participating in the consensus protocol makes users' stakes illiquid;
  - Risk of economic barrier to entry (not enough money to be a validator).
- **Delegated proof of stake**: Users delegate the validation of new blocks to a fixed committee, through weighted voting based on their stakes. The problems are that:
  - Known delegate nodes, therefore exposed to DDoS attacks;
  - There might be a centralization of governance.

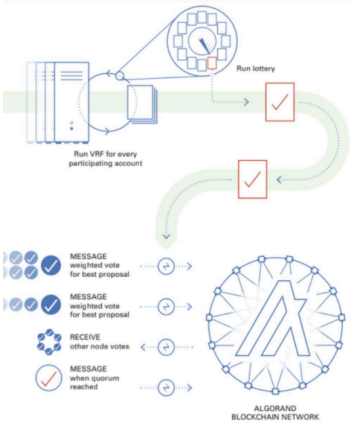




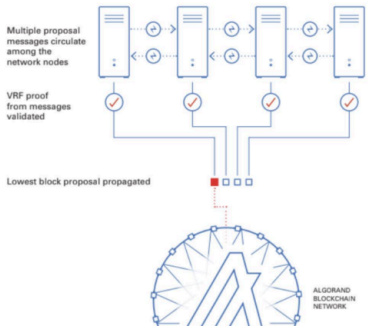
Algorand (2019) was introduced by Silvio Micali. The cryptocurrency is the **ALGO**.
Now, how does the consensus mechanism work in Algorand?
Let's think of a dice:

- Dices are perfectly balanced and equiprobable, nobody could tamper their result;
- Keeping observing dice rolls by no means increases the chance of guessing the next result;
- Each dice is uniquely signed by its owner, nobody can roll someone else's dice;
- Dices are publicly verifiable, everybody can read the results of a roll.

Each **ALGO can be considered a dice participating in a safe and cryptographic dice roll**. Hence, the more ALGOs we have, the more dice to roll. For each new block, dice rolls are performed in a distributed, parallel and secret manner. The winner is revealed in a safe and verifiable way only after winning the dice roll, proposing the next block. We can think of it as a **lottery ticket**. Hence each algo a user owns is a lottery ticket (so the more ALGO we have, the more tickets). A user runs a function for each ticket and **if the result of the function falls in a certain range, the user wins**. This function is called the **Verifiable Random Function (VRF)**. It works like this:



1. **Block proposal**: In the block proposal phase, accounts are selected to propose new blocks to the network. Once an account is selected by the VRF (the one who wins the lottery), the node propagates the proposed block along with the VRF output, which proves that the account is a valid proposer.
2. **Soft vote**: The purpose of this phase is to filter the number of proposals down to one, guaranteeing that only one block gets certified. Nodes will verify the signature of the message and then validate the selection using the VRF proof. Next, the node will compare the hash from each validated winner's VRF proof to determine which is the lowest and will only **propagate the block proposal with the lowest VRF hash**.
   A **committee** is elected and votes for the best proposal, so that only one block can be agreed upon. Once a quorum is reached for the soft vote the process moves to the certify vote step.
3. **Certify vote**: A **new committee** (different from the previous one) checks the block proposal that was voted on in the soft vote stage for overspending, double-spending, or any other problems. If valid, the new committee votes again to certify the block.

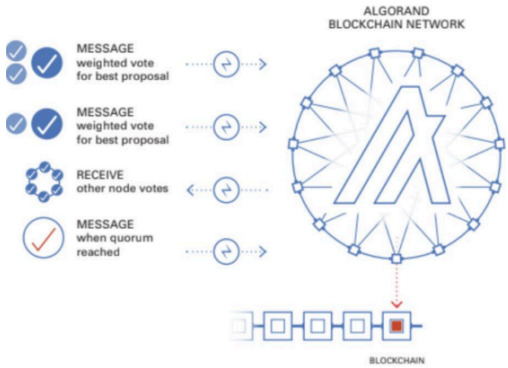


Let's see the general idea behind the consensus mechanism:

- A secret key (SK)/public verification key (PK) pair is associated with each account.
- For each new round r of the consensus protocol a new committee is sorted this way. Each account perform a VRF, using its own secret key (SK), to generate:
  - A pseudo-random number: hash;
  - The verifiable associated proof: π.

  $$< hash, \pi > \leftarrow VRF_{sk}(seed||role)$$

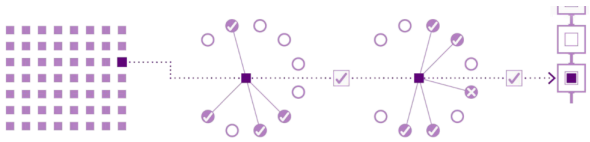
  Notice that for each step of the protocol it is necessary to add a seed and also the role of the account in that step of protocol.



- If **hash ∈ winner_range** , the account "wins the lottery" and virally propagates the proof of its **victory π** to other network's nodes, through a "gossiping" mechanism.
- Others node can use the **public verification key (PK) to verify that the hash was generated by that specific account** using the verifyVRF function:

  $$< 1/0 > \leftarrow \text{verify}VRF_{pk}(hash, \pi, seed||role)$$

  If the output is 1 the proof is verified, otherwise 0.



To sum up:

- An account is elected to propose the next block;
- A committee is elected to filter and vote on the block proposals;
- A new committee is elected to reach a quorum and certify the block;
- The new block is appended to the blockchain. This means that each round of the consensus protocol appends a new block in the blockchain.

Note: Algorand uses a modified version of the **Byzantine Agreement called BA***. This consensus protocol can tolerate **an arbitrary number of malicious users as long as honest users hold a super majority (> 2/3) of the total stake in the system**. In combination to using the VRF, with respect to the standard Byzantine Agreement Algorand can prevent DDoS attack on the committee because of:

- The adversary does not know which users he should corrupt because we run the lottery on his VM so the winner is revealed after the user wins the lottery. This means that the adversary realizes which users are selected too late to benefit from attacking them.
- Each new set of users will be privately and individually elected.

In Algorand we have two types of nodes:

- **Non-Relay Nodes:**

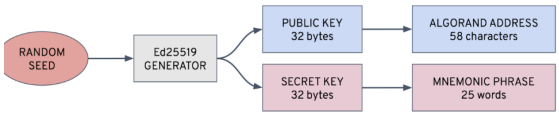
    - **Participate in the PPoS consensus** (if hosting participation keys);
    - Connect only to Relay Nodes;
    - They are two configurations:
        - Light Configuration: Store just the latest 1000 blocks (Fast Catch-Up) - good for memory;
        - Archival Configuration: Store all the chain since the genesis block;
- **Relay Nodes:**
    - Communication routing to a set of connected Non-Relay Nodes;
    - Connect both with Non-Relay Nodes and Relay Nodes. They kind of boost up network performance.
    - Route blocks to all connected Non-Relay Nodes;
    - Highly efficient communication paths, reducing communication hops.

Transactions are the core element of blocks, which define the evolution of distributed ledger state allowing to **change the Blockchain state**. In order to be approved, Algorand's transactions must comply with:

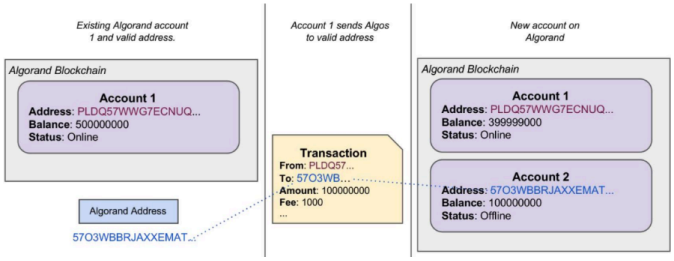
- **Signatures**: Transactions must be correctly signed by its sender, either a Single Signature, a Multi Signature or a Smart Signature/Smart Contract;
- **Fees**: In Algorand transactions fees are a way to protect the network from DDoS. In Algorand Pure PoS fees are not meant to pay "validation" (as it happens in PoW blockchains);
- **Round validity**: Transactions' have a validity of 1000 blocks (at most).

Concerning the keys, Algorand uses **elliptic curve cryptography** for **key pair generation**:

- **Private Key Generation (32 bytes)**: A random 256-bit integer is selected that serves as the private key.
- **Public Key Derivation (32 bytes)**: The public key is derived from the private key using elliptic curve cryptography. Algorand specifically uses the Ed25519 elliptic curve, which is based on the mathematical properties of the Edwards curve. The public key is a point on the elliptic curve computed by multiplying the private key by the curve's generator point.



- **Address (58 chars)**: An Algorand Address is the unique identifier for an Algorand Account. The public key is transformed into an Algorand Address, by adding a 4-byte checksum to the end of the public key and then encoding it in base32.
- **Mnemonic Phrase (25 words)**: The 25-word mnemonic is generated by converting the private key bytes into 11-bit integers and then mapping those integers to the bip-0039 English word list (2048 words).



Accounts are entities on the Algorand blockchain associated with **specific on-chain local state**. This means that an account can send a certain amount of coins to a (valid) Algorand address. This creates a new address on the state of the chain. There is a minimum balance required to perform some operation of Algorand (e.g. For running an account 0.1 ALGOs are required).

In Algorand we call tokens an **Asset**. They are backed in the protocol itself, therefore they do not require writing a smart contract. These are the field of ASAs:

- Total: The total number of base units of the asset to create;
- Decimal: The number of digits to use after the decimal point (if 0, the asset is not divisible);
- defaultFrozen: True to freeze holdings for this asset by default;
- UnitName: The name of a unit of this asset. (ex: USDT);
- AssetName: The name of the asset. (e.g. US dollar);
- URL: Specifies a URL where more information about the asset can be retrieved.
- MetaDataHash: 32-byte hash of some metadata.

```
sp = algod_client.suggested_params()
txn = transaction.AssetConfigTxn(

    sender=acct1.address, #acct1 is a valid account object
    sp=sp, #fields common to all txs types such as fee, validity's interval
    default_frozen=False,
    unit_name="DCC",
    asset_name="Di Ciccio Coin",
    manager=acct1.address,
    reserve=acct1.address,
    freeze=acct1.address,
    clawback=acct1.address,
    url="https://somesite.someTLD",
    total=1000,
    decimals=0,
)
```

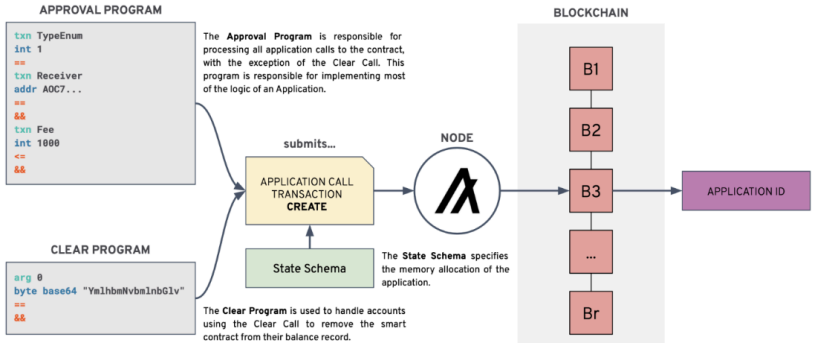
- ManagerAddress: The address of the account that can manage the configuration of the asset and
- destroy it.
- FreezeAddress: The address of the account used to freeze holdings of this asset.
- ClawbackAddress: The address of the account that can clawback holdings of this asset.

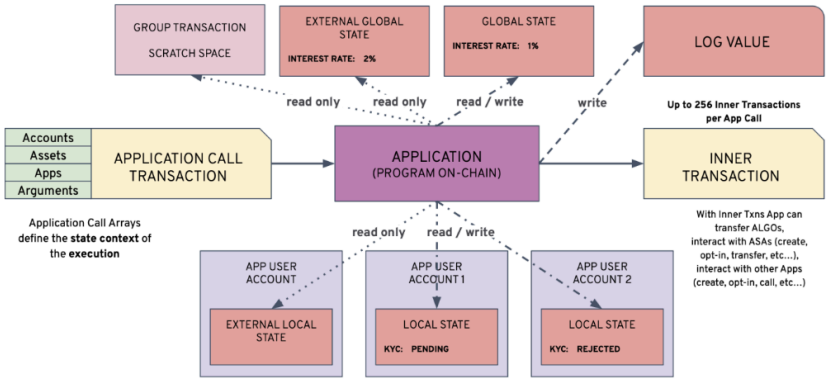
The **Algorand Virtual Machine** is a **Turing-complete secure execution environment** that runs on Algorand. Its purpose is approving or rejecting transactions' effects on the blockchain according to Smart Contracts' logic. **AVM approves transactions' effects if and only if there is a single non-zero value on top of AVM's stack**. AVM rejects transactions' effects if and only if:

- There is a single zero value on top of AVM's stack;
- There are multiple values on the AVM's stack;
- There is no value on the AVM's stack.

Concerning the **Smart Contract**, we can be written it in any language we want and they will be eventually compiled in the **AVM Bytecode** necessary to be executed by the Algorand Virtual Machine (AVM). In Algorand, smart contract works with two programs:

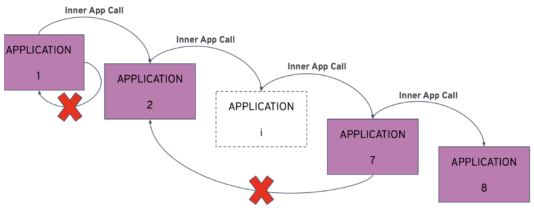
- **Approval program**: It is responsible for processing all application calls to the contract, with the exception of the Clear Call. When an account initiates a transaction that involves the smart contract, the Approval Program is executed to determine whether the transaction should be accepted or rejected based on the specified conditions and rules. This program is responsible for implementing most of the logic of an application.
- **Clear program**: It is used to handle accounts using the clear call to remove the smart contracts from their balance record. It essentially handles the process of clearing or closing out a smart contract from an account. Clearing a smart contract might involve releasing any resources held by the contract and updating the account state accordingly.





The **State Schema** specifies the memory allocation of the application that is a persistent storage of data associated with a smart contract. Notice that contrary to Ethereum, when a smart contract is deployed we are given an **Application ID** (not an address). Moreover, we can do some operations of a Smart Contract (even **updating** and **deleting** it). "**OPTIN**" refers to the process by which an account chooses to participate or "opt in" to holding a particular asset. The opt-in process involves a transaction where an account sends a 0 Algos transaction to itself, with the asset specified in the transaction. This transaction indicates the account's intention to participate in holding that particular asset. Once an account has opted in, it can receive and send units of the opted-in asset.

Smart contracts can read the local state of other applications and read and write the local state in its own local state. The global state is managed by the application and it corresponds to what Ethereum does on the Smart Contract and this can be read/written. The application can also read the external global state. **An application can also make inner transactions**.



Note: **An application may not call itself, even indirectly**. This is referred to as re-entrancy and is explicitly forbidden. An application may only call into other applications up to a stack depth of 8.

Some **differences between Ethereum and Algorand**:

- **Accounts**: Both Ethereum and Algorand are account-based blockchains supporting smart contracts. On Ethereum, smart contracts are actually represented by accounts, while on **Algorand smart contracts and accounts are two different objects**. **Ethereum's Externally-owned accounts (EOA) correspond to Algorand accounts**. They are both represented by an address. **Ethereum's contract accounts correspond to Algorand application ID**, which are 64-bit integers
- **FT and NFT**: Ethereum allows creating custom fungible and non-fungible tokens by deploying smart contracts following ERC-20, ERC-721, or ERC-1155 standards. Transacting with such tokens is very different from transacting the base cryptocurrency Ether. On Algorand, such custom tokens are all **Algorand Standard Asset (ASA)**. On Algorand, to interact with an application or ASA, an account must opt-in. Opting in to an ASA is done by making a 0 amount asset transfer of the ASA from the account opting in to itself. This helps reduce spam from unwanted ASAs. Tokens on Ethereum are defined by a contract address (+ an ID for ERC-1155 tokens). On Algorand they are just defined as a 64-bit unsigned integer ID.
- **Gas, Transaction Fees, and Minimum Balance**: On Ethereum, transaction fees are called gas fees. Gas fee is paid whether the transaction is successful or not. On Algorand, transaction fees are only paid if the transaction is included in the block. On Algorand all transactions can use the **minimum base fee of 0.001 Algo**. This minimum transaction fee is **independent of the transaction type**: application call, Algo transfers, and ASA transfers. In addition to transaction fees, Algorand also has a notion of **minimum balance** that acts like a deposit to rent space on the blockchain. If the space is liberated (e.g., opt out of the asset), the minimum balance requirement decreases. A basic account has a minimum balance requirement of 0.1 Algo.
- **Smart Contract Storage**: Ethereum smart contract storage is a huge array of 2^{256} uint256 elements. Algorand smart contracts have three different types of storage:
    - **Local Storage**:
        - Allocation: Is allocated when an account opts into an app (submits a transaction to opt-in to an app). Can include between 0 and 16 key/value pairs. Local storage is allocated in k/v units, and determined at contract creation. This cannot be edited later.
        - Reading: Can be read by any app call that has the app ID in its foreign apps array and a specific account in its foreign accounts array.
        - Writing: Is editable only by the app, but is delete-able by the app or the user x (using a ClearState call).
        - Deletion: Deleting an app does not affect its local storage. Accounts must be cleared out of the app to recover minimum balance.
          An account holder can clear their local state for an app at any time (deleting their data and freeing up their locked minimum balance). The purpose of the clear state program is to allow the app to handle the clearing of that local state gracefully.
    - **Global Storage**:
        - Allocation: Can include between 0 and 64 key/value pairs. The amount of global storage is allocated in k/v units, and determined at contract creation. This schema is immutable after creation.
        - Reading: Can be read by any app call that has specified app's ID in its foreign apps array.
        - Writing: Can only be written by the app itself.
        - Deletion: Is deleted when the app is deleted. Cannot otherwise be deallocated (though of course the contents can be cleared by the app itself).
    - **Boxes**:
        - Allocation: App can allocate as many boxes as it needs, when it needs them. Boxes can be any size (from 0 to 32K bytes). Box names must be at least 1 byte, at most 64 bytes, and must be unique within each app.
        - Reading: The App that creates the Boxes is the only app that can read the contents of its boxes on-chain. This on-chain privacy is unique to box storage.
        - Writing: The App that creates the Boxes is the only app that can write the contents of its boxes.

        - Deletion: Each App can delete its own boxes. If an app is deleted, its boxes are not deleted.
- **Multisig, Atomic Transfers**: On Ethereum, it is possible to write smart contracts to ensure that fund transfers require approval/signatures by multiple distinct users. On Algorand, multisig accounts are natively supported. Atomic transfers or group transactions are natively on Algorand, they allow the grouping of multiple transactions together so that they either all succeed or all fail, and they can be used to make one transaction pay the fee for the others (Pooled fee).

12. Frauds on Blockchain

CEX (Centralized Exchange) and **DEX (Decentralized Exchange)** refer to different types of cryptocurrency exchanges. In summary:

- In CEX a centralized organization has control. In DEX the architecture is backed by the Blockchain without third-party operators;
- CEX requires KYC verification. In DEX there is anonymity so no personal info is required;
- In CEX fiat currencies are allowed. In DEX Altcoins and Stablecoins transactions are available;
- In CEX data is exchanged internally using order books. In DEX peer-to-peer trading is used that simplify the connections between users and buyers.

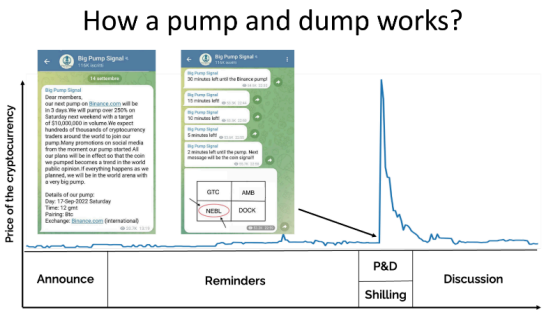
Let's see how **CEX** works. An **order book** is a real-time, continuously updated list of buy and sell orders in a particular market, including cryptocurrency markets. It provides information on the demand and supply for a specific asset or cryptocurrency at different price levels. There are two main types of orders in a cryptocurrency order book:

- **Limit Orders**:
    - **Buy Limit Order**: A buy limit order is an order to buy an asset at a specified price or lower. It will only be executed at the limit price or a better price.
    - **Sell Limit Order**: A sell limit order is an order to sell an asset at a specified price or higher. It will only be executed at the limit price or a better price.

  Limit orders are used by traders who want to set specific entry or exit points in the market. They are not executed immediately but are placed in the order book until a matching order is found.
  EX. I want to buy/sell 5 SAP for 10$.
- **Market Orders**:
    - **Buy Market Order**: A buy market order is an order to buy an asset at the current market price. It is executed immediately, as it is willing to pay the current asking price.
    - **Sell Market Order**: A sell market order is an order to sell an asset at the current market price. It is executed immediately, as it is willing to accept the current bid price.

  Market orders are used when a trader wants to execute a trade quickly without specifying a particular price. The order will be filled at the best available price in the order book.
  EX. I want to buy/sell 5 SAP (regardlessly of the price).

A **Pump and Dump** is a type of market manipulation scheme that occurs in financial markets, including the cryptocurrency market. It involves artificially inflating the price of a financial asset, followed by the rapid selling of that asset to take advantage of the inflated price. There are a lot of famous groups that perform cryptocurrency frauds like Big Pump Signal, Trading Crypto Guide and Crypto Coin B. It usually works like this: The organizers organize to do the fraud on a certain time and on a certain coin, usually using some communication channels such as Telegram. The group members know it some time before so that they can put market order when the coin value is already low. The fraud works when external investors start buying the coin at a higher price. When the coin has reached a certain market value, the scammers start selling the coin making money and the external investors will lose money. In summary:



- **Admins**: They buy low and sell high having a huge profit;

- **Members of the group**: A few of them buy fast enough to have a profit but the majority ends up losing money;
- **External investors**: They always lose money and are left with a worthless coin.

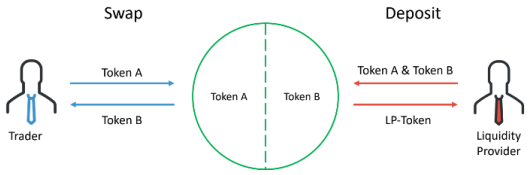
Concerning the members of the group, there is usually a priority according to which they receive the communication of the pump and dump. They usually can get a higher priority by joining an affiliation program. The ones with the higher priority are often called VIP members.

Can we make a system that detects these scams?

The idea is that we can observe that during a pump, most of the members of the group put market orders. There are some classifiers that use ML that allow you to detect frauds within a few seconds.

Let's now see what happens in **DEX** markets. DEX allows anyone to create a **market for a token** where the decentralized exchanges employ smart contracts. In a DEX, instead of relying on a traditional order book with buyers and sellers, **liquidity pools** are used to provide the necessary liquidity for trading. They work like this:

- We have a pool splitted in two parts with a token A and a token B. To determine the price of the liquidity pool it is possible to compute the ratio between the prices of the two tokens. For example, if we have 1 token A and 10 token B, if we want to buy token B we have to provide the liquidity pool 0.1 token A.
- Assuming to have money in the **liquidity pool**, a trader just makes a transaction so he gives a certain amount of token and he will get the respective amount of the other token back.
- The **liquidity provider** (investor) puts the tokens in the pool. He has both tokens that can be put inside of the liquidity pool and he can get back a token called **LP-token** that represents the share the liquidity provided has on the liquidity pool. Liquidity providers are compensated for the risks they take through trading fees. When users trade against the liquidity pool, they pay a fee, which is distributed among the liquidity providers based on their share of the pool.



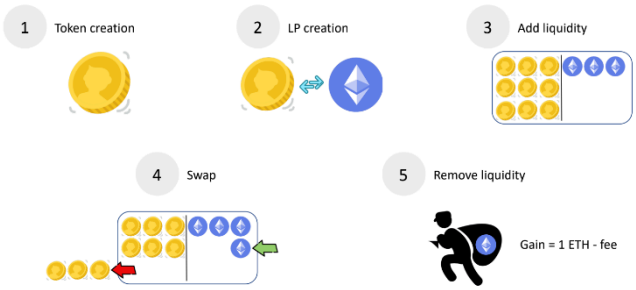
How much does a token last?

They live a lifetime (from its creation to the last transaction made) even though it was seen that this amount of time is usually less than one day (in this case we talk about 1-day tokens).

Who creates the token?

1% of addresses are responsible for creating over 20% of the tokens. Again, more than half of the tokens created by the token spammers are 1-day tokens.

Rug Pull is a kind of attack that can happen in the DEX market. In summary, a token is created and it's associated with another token (es. ETH) in a liquidity pool by creating the LP-token. How that the liquidity pool is created, some liquidity is added to it. Once a substantial amount of funds is locked in the liquidity pool, the fraudsters execute the rug pull. They withdraw the liquidity or funds from the project, often using smart contracts or other mechanisms to do so quickly and without giving investors a chance to react. With the sudden withdrawal of liquidity, the value of the project's token typically collapses rapidly. Investors who provided liquidity or held the token experience significant losses, as the promised returns and liquidity disappear. The individuals behind the rug pull often take steps to remain anonymous, making it challenging for affected investors to pursue legal action. They may exit the project and disappear from public view, leaving victims with little recourse.



Who buys the tokens?

Sniper bots, automated tools designed to buy tokens before anyone else, and as soon as possible. They scan the "mempool" for new liquidity pools.

13. Data Confidentiality with Public Blockchains

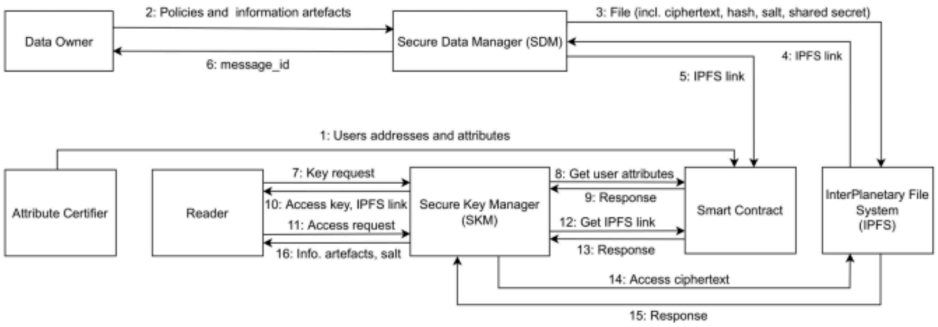
Let's firstly introduce the IPFS (InterPlanetary File System) that is a **distributed system for storage and access to files** with **no central authority**. When a file is uploaded in the IPFS, a unique identifier is produced and used to track the file in the system and it's **content-based** (if the content changes, also the location change).

Attribute-Based Encryption (ABE) is a type of public-key encryption. In CP-ABE, users are associated with attributes, typically represented as propositional literals. Messages, on the other hand, are associated with access policies, usually expressed as propositional formulae on attributes. The access policy defines the conditions under which a user is able to decrypt a given ciphertext. The phases are:

1. Pre-phase: Generation of two keys (public key (pk) and master public key (mpk));
2. Ciphering: Given a plain text, it is encrypted using a policy and the public key getting a ciphertext;
3. Key Generation: An **access key** (for decryption) is **created from a set of attributes** and the key pair;
4. Deciphering: Given a ciphertext, it is decrypted using the access key and the public key (pk).

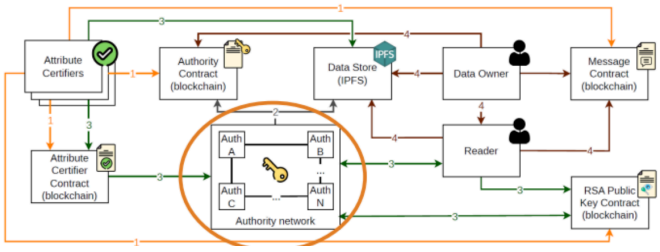
So, the attributes used to create the access key must satisfy the policy that was used for ciphering the plain text.

CAKE (Control Access via Key Encryption) is a mechanism to access data saved on the IPFS using the approach explained before.
Note: More details on slides.
In this kind of architecture there are two blocks called Secure Data Manager (SDM) and Secure Key Manager (SKM) necessary to store the information on keys, policies and information artifacts. The problem is that they are centralized (all information is stored on a server) and this leads to a certain amount of problems, especially concerning security. For this reason, a new architecture called **MARTSIA** (Multi-Authority appRoach to Transaction System for Interoperating Applications).



MA-ABE extends the concept of Attribute-Based Encryption (ABE) to involve multiple authorities. In traditional ABE, there is a single authority responsible for managing attributes. In MA-ABE, multiple authorities collaboratively manage attributes, increasing the flexibility and scalability of the system. In **CP-MA-ABE**, **users are associated with attributes** represented as propositional literals, similar to CP-ABE. However, in the multi-authority context, these attributes are distributed across different authorities. **Messages are associated with policies** expressed as propositional formulae on attributes. The policy specifies the conditions under which a user can decrypt a given ciphertext. The phases are:

- Pre-phase: We have different authorities each of one generate a set of public/private keys.
- Ciphering: Given a plain text, it is encrypted using a policy and (all) the public keys getting a ciphertext;
- Key Generation: An **access key** (for decryption) is **created from a set of attributes** and the private keys of the authorities;
- Deciphering: Given a ciphertext, it is decrypted using the access key and the public keys.



The architecture is the one described in the image on the right.
Note: More details on slides.
So, the MARTSIA approach has an authority network that manages the keys instead of a central authority.

14. Decentralized Oracles for the Ethereum Platform

An **oracle** is a third-party service or entity that provides smart contracts with external information. Smart contracts are generally **isolated from the external world** and **cannot directly access real-world data**. Oracles act as bridges between the blockchain and the outside world by fetching and delivering external data to smart contracts. The two main features of oracles are:

- **Reliability**: Oracles must provide accurate information to smart contracts. Inaccurate data could lead to incorrect execution of smart contract conditions.
- **Shared Usage Protocol (Interoperability)**: Having a shared protocol involves standardizing the way oracles interact with smart contracts and blockchain platforms. This can enhance interoperability across different blockchain networks and ensure that oracles can be seamlessly integrated into various decentralized applications (DApps).

There are several types of oracles based on the **source of information** such as:

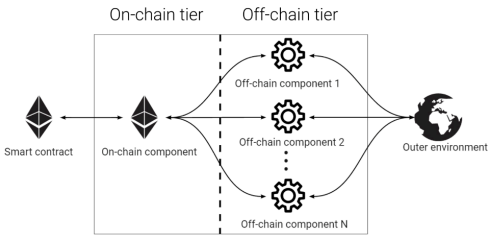
- **Software oracles**: Software oracles are programs or algorithms that automate the process of fetching and delivering external data to smart contracts. They use predefined rules and algorithms to gather information from various sources and the collected data is then transmitted to the smart contracts;
- **Hardware oracles**: Hardware oracles involve physical devices that provide information from the real world to smart contracts on the blockchain. These devices are often equipped with sensors or mechanisms to capture real-world data;
- **Human oracles**: Human oracles involve individuals who manually provide information to smart contracts based on real-world events or conditions.

Flow Direction \ Event Initiator	PULL	PUSH
IN	PULL-IN	PUSH-IN
OUT	PULL-OUT	PUSH-OUT

The **event initiator** refers to the entity or system that triggers the need for external data in a smart contract or decentralized application.
Based on the **event initiator** and **flow direction**, we can classify oracles in:

- **Pull**: In a pull-based model, the smart contract actively requests or pulls data from an external source or oracle when needed;
- **Push**: In a push-based model, the external data source or oracle actively pushes updates or information to the smart contract without the contract explicitly requesting it.
- **In (Inbound)**: Data flows into the smart contract from an external source or oracle. The smart contract consumes or receives the data to make decisions or execute actions.
- **Out (Outbound)**: Data flows out of the smart contract to an external system or oracle. The smart contract may send information or trigger actions in response to specific conditions.

Concerning the trust, we can classify between **centralized oracles** that are owned by a single entity and **decentralized oracles** owned by several entities. Notice that in the case we decide to use an oracles (a centralized communication mechanism) to make the smart contract communicate with the real world, we are **introducing a bottleneck in the decentralized ecosystem**. So we often use decentralized oracles to avoid this problem (es. If one fails, the others continue to operate). Decentralized oracles typically consist of two main tiers:



- **On-Chain Tier**: Oracle contracts, which are deployed on the Ethereum blockchain, serve as the interface between the smart contracts and the off-chain data sources.
- **Off-Chain Tier**: Oracle nodes that run off-chain and are responsible for interacting with external data sources. These nodes fetch real-world data and provide this information to the on-chain smart contracts when requested.

Note: See how the different patterns (push-in, push-out, pull-in, pull-out) works on slide.

CreditWorthinessOracle:On-chain component

Solidity code snippet

```
contract CreditWorthinessOracle{
. . .
function newVerification(int256 id,string memory taxId,address caller) public
    {
        bool[3] memory voted;
        bool[3] memory answers;
        OracleRequest memory req=OracleRequest(voted,answers,false,false);
        requestList.push(req);
        emit NewVerification(caller,id,taxId,requestCounter);
        requestCounter++;
    }
}
event NewVerification(address,int,string,int);
```

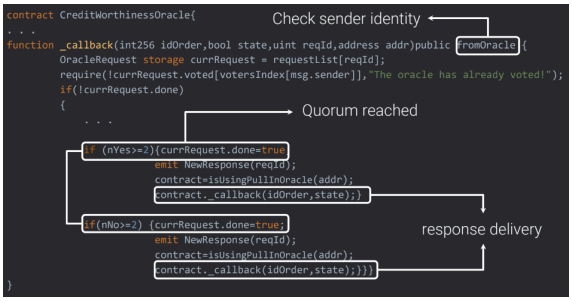
Requests storage

Event for the off-chain components

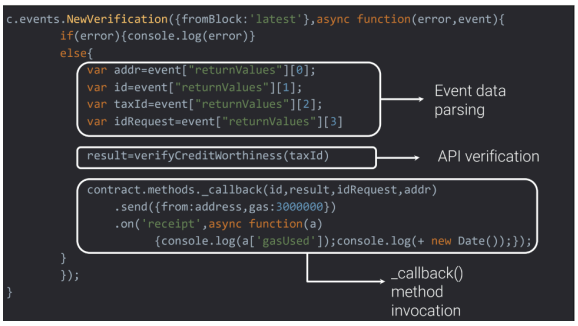
In order to implement an oracle we have to use **solidity to implement the on-chain component** and **Javascript (r.g. web3) to implement off-chain components**.

What are the pros and cons of decentralized oracles?

- Pros: The decentralization of blockchain is preserved. We may have multiple sources of information.
- Cons: Worse performances (with respect to centralized oracles) and higher communication overhead.



CreditWorthinessOracle:Off-chain component Javascript (Web3) code snippet



15. Resource Governance Framework

The **Resource Governance Framework** facilitates **usage control of digital assets in decentralized web environments** and can be instantiated by combining:

- Blockchain: Records policies expressing the usage conditions associated with resources & monitor their compliance;
- Trusted execution environments: Inside data consumers' devices (to enforce said policies).

In this framework we have:

- **Pod**: Decentralized data stores where people can store their data securely. Think of it like Folders in your file system. Each pod has its own ID, name, owner address, pod address, base url, etc..;
- **Resource**: A file belonging to one particular pod. Think of it like Files inside a folder in your file system. By default, the user who owns the pod owns its resources. A resource can have its own set of resource obligations or inherit from the parent pod.

The resource obligations are:

- Access Counter : The resource can be opened up to specific counts;
- Domain Obligation : The resource can be processed only for specific purposes;
- Temporal Obligation : The resource can be stored for up to specific days;
- Country Obligation : The resource can be loaded only in specified countries.

An owner can share their resource to any other user in the Decentralized Marketplace, provided they are an active subscriber agreeing to the usage policies. Whenever the resources are access / used , it is verified whether they are utilized as per the agreed policies and their usage logs are maintained. This enables the user to detect misconduct and unexpected or unpermitted usage. These logs keep track of details like resource id, monitoring id, timestamps, resource status, etc which can be exported to CSV files.

16. Blockchain based Resource Governance for Decentralized Web Environments

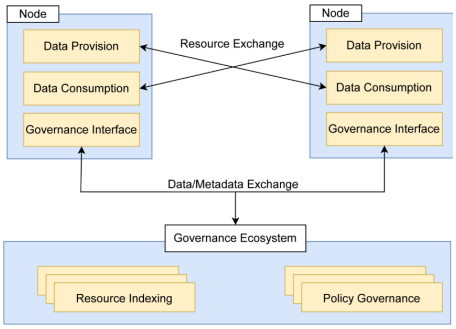
Data trading has emerged as one of the most significant markets in recent years. Data is usually owned by big tech companies (e.g. Google, Meta, X, Apple, etc..) but users don't know how this data is actually used.

A **Solid Personal Online Datastore (POD)** refers to a concept developed by the inventor of the World Wide Web, Sir Tim Berners-Lee, with the aim of **giving individuals more control over their own data on the internet**. Solid stands for "Social Linked Data" and the idea is to provide users with a way to **store their personal data in a secure and decentralized manner**.

What we could do is combine Solid with the **Usage control** that is a mechanism that performs checks during data access. Hence, the user who owns the data can set constraints that the data consumer must meet in order to be able to access and use the data. In the market, possible usage rule are:

- A maximum number of accesses to their data;
- A data limit for the use of their data;

- Geographical limitations;
- Purpose of use limitations.

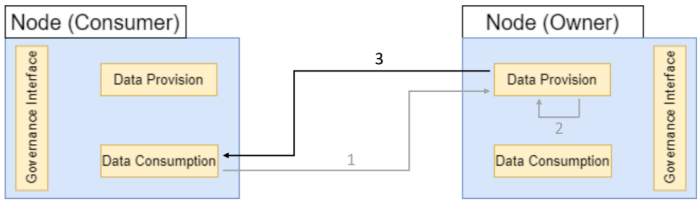


One possible architecture is ReGov. The ReGov Architecture is dìvided in two parts that are the Nodes and the Governance Ecosystem. Let's start from a **node** that is running on a user device and is responsible for **providing and consuming data**. It is composed of 3 parts:

- **Data provision**: It manages the requests of retrieving data (e.g. an image) from another node (e.g. the owner of a shop). The owner of the node provides a resource and the usage policy to retrieve it. Another user can make a request to get the data and the owner can start a monitoring how the data is used for;
- **Data consumption**: It is composed of several parts such as the resource retriever that is responsible for creating an authenticated request for a copy of a resource on a data market (e.g. the image published by the shop owner). The owner of the data (and the usage policy) sends the data through a gateway to the data manager that adds it to the protected data. After the data is retrieved, the user wants to use this data so, assuming we are running an application on a node, the user can make a call to the data manager that checks the enforcement mechanisms (the usage policy) to understand if the user can use the data or not and if the user can use it that it is given to him. Inside of the **protected data** that it is possible to find the resources retrieved from other nodes, the usage policies related to resources retrieved from other nodes and logs used for tracking activities into the Data Consumption component.
- **Governance Interface**: It handles all the ingoing messages of the node and the outgoing messages of the receiver.

The **Governance Ecosystem** is responsible for handling the **data/metadata exchange**. The owner of the data sends the resource and usage policy to the **data provision** that saves a copy of the usage policy in the **policy storage**. Moreover it allows the owner to start monitoring the data though the **monitoring manager**. In addition to the resource policy, the owner of the data also provides some metadata associated with the data itself that is saved to the **index component** though the **resource indexer**.

Data Retrivial Flow



Monitoring Flow