Architecture and scalability of Ethereum

Mirko Bez
mirko.bez@studenti.unipd.it

Giacomo Fornari
giacomo.fornari@studenti.unipd.it

September 1, 2018

Version 1.0.0

Contents

1. Introduction

The purpose of this report is to investigate the architecture and the scalability of Ethereum [1]. Ethereum is based on the Blockchain technology. Despite the first appearance of the blockchain in the actual form is due to Satoshi Nakamoto's groundbreaking paper, "Bitcoin: A peer-to-peer electronic cash system" (2008) [2], this technology has become one of the most active research fields in ICT only in the last couple of years.

The aim of the Blockchain technology is to provide a total order of the transactions in a distributed ledger without relying on a trusted third party (e.g. a bank [3]). Not relying on a trusted central authority may lead to practical issues like transaction repudiation and the infamous **double-spending problem**. The former is self-explanatory and can be solved by digital signatures, while the latter consists in using the same digital token to pay multiple entities. It found its first practical decentralized solution with the appearance of Bitcoin [4][1].

This technology has found, apart from merely financial applications, other applications such as auctions, supply chain and notary services.

Bitcoin Bitcoin is a state transition system, in which there is a transition from a valid state to another valid state through a valid *transaction*. The state consists in the balance of the addresses[2]. Each node in the network maintains a local copy of the state and updates its *own* copy of the state in a deterministic way according to the transactions. Therefore, to have an exact replica in each node, the order of transactions should be total and agreed by every member of the network. The mechanism through which this total order is provided and maintained is the blockchain, which is literally a chain of blocks. Each block of the chain contains an ordered list of transactions and is connected to the previous block by inserting the hash of the previous block in its header. Each node of the network has the faculty to create transactions and have to sign them to show that it owns the private key corresponding to a given address. The transactions are spread in the network through gossip protocols. Once a node receives a new transaction, it verifies that the transaction is well-formed. If it is the case, the node sends the transactions to the other known peers. Eventually, the transactions are received by a member of the network who groups some transactions in a block and tries to find a nonce such that the hash of the block is smaller than a given value. Since this task is computationally expensive, if the node finds this value, it adds at the beginning of the transaction list a transaction in which it assigns to a beneficiary address an amount of newly minted coins, according to the protocol's rule[3]. In addition to this reward, it receives also fees from the senders of the included transactions. The members of the network who try to create new blocks are called miners, because their action resembles the extraction of precious metals. The miners are incentivized to create valid blocks, that is, containing

[1] We refer to [4] for a complete survey about the history and ancestors of Bitcoin.

[2] The addresses correspond to a private/public key pair. Each peer of the network can have zero or more addresses.

[3] This value was initially 50 bitcoins. This reward halves every 210000 blocks. Currently it is 12.5 bitcoins. Around year 2140 no coins would be minted [5].

valid transactions and the correct solution to the puzzle, so the other peers of the network can accept *only* valid blocks. They can verify the correctness of the transactions (e.g. the balance of the addresses is always positive), because they have a local copy of the state, and the correctness of the nonce by computing a single hash. It is worth noticing that multiple parties try to create new blocks concurrently, therefore it is possible that multiple versions of the blockchain co-exist. Indeed, a mechanism to select the canonical blockchain is needed. In the case of Bitcoin it is simply the longest chain, because it corresponds to the one with more work invested on it. The co-existence of multiple blockchain can be very useful in case of a network partition, indeed, once the partition is over, the peers can agree on the blockchain. The drawback of this system is that there is no consensus finality [3], thus it is necessary to wait a certain number of blocks (confirmation blocks) to be sure that the transactions are really confirmed. The number of confirmation blocks in Bitcoin is six which correspond to approximately one hour [5].

Although this description of Bitcoin abstracts from various details, it is sufficient to show how the double spending problem is solved. The idea is to let each peer of the network know the current state and the transactions that are already spent. Moreover, after the confirmation time the blockchain can be considered immutable and tamper-proof, because to rewrite the sufficiently old transaction history, an enormous amount of work should be done. The immutability is an interesting property that can be used to emit certificates about the ownership of an asset such as a digital artwork or an intellectual property.

Permissioned vs Permissionless Blockchain Apart Bitcoin, a lot of cryptocurrencies, also known as *altcoins*[4] came out. They have different peculiarities, but the general idea is the same as Bitcoin. In the literature [3], it is common to distinct between permissionless and permissioned blockchains. The former are blockchains in which everyone can participate in, while the latter requires authentication and is commonly used by banks or consortium of companies. Prominent examples of permissionless blockchains are Bitcoin and Ethereum, while a representative permissioned blockchain is Hyperledger.

Overview In the remainder of the paper we will discuss exclusively Ethereum. Ethereum can be viewed as a generalization of Bitcoin. While in Bitcoin the execution environment (i.e. the script language) is stateless and is used only to express conditions to spend the money, e.g. demonstrate the possession of a given private key, in Ethereum the execution environment (i.e. the EVM) is stateful and Turing Complete. To avoid the misuse of the network resources, each opcode is associated with an amount of gas, so the termination of the computations is always guaranteed.

This paper is organized in two main chapters. The first chapter describes the architecture of Ethereum in a top-down fashion by individuating and describing the layers in which it can be split. The second chapter analyzes the scalability of the system, both theoretically, with the analysis of the literature (Section 3.2) and the description of the

[4]The contraction of "alternative coins".

cube of the scalability [6] applied to Ethereum (Section 3.3) and practically, by creating a private Ethereum Network and running some tests (Section 3.7).

2. Architecture

Although Ethereum clients with different implementations can agnostically participate in the system, there is no common agreement on the Ethereum architecture. Since "an implementation is not an architecture" [6], we propose a conceptual model which is inspired by the OSI reference model. The model is comprised of 1 vertical layer and 5 horizontal layers: the (1) network layer, the specification of the network topology, the (2) propagation layer, how the nodes communicate and which protocols are used, the (3) data layer, the data structures and data types, the (4) consensus layer, how the nodes reach consensus on the state which is represented in the data layer, hence the mining and the transactions execution process, and the (5) application layer, the smart contract as the business logic of the system.

We define the Ethereum Virtual Machine (EVM) as a vertical layer because it appears in the data, consensus and application layers.

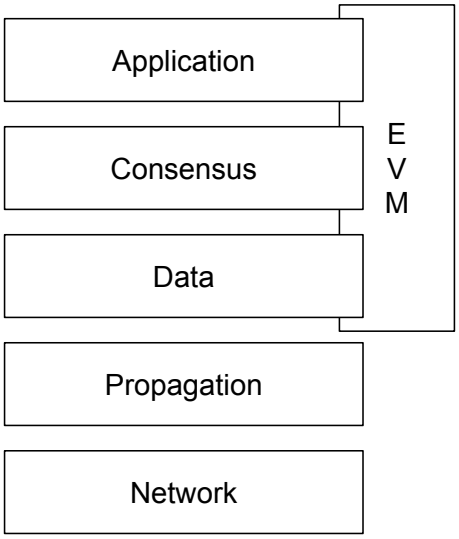

Figure 1: Layers of the Ethereum architecture

Sometimes we could refer to some implementation details along the description of the layers, because the official documentations are not always complete, especially those parts regarding the network and propagation layers. In this case, we take as reference implementation go-ethereum[5] (`geth`), which is directly maintained by the Ethereum Foundation.

[5] `https://geth.ethereum.org/`

2.1. Network layer

The aim of this layer is to build the peer-to-peer network, a decentralized architecture in which the nodes are logically equivalent and function as a *servent* (i.e. a node that acts as both client and server at the same time). In this type of architecture, the nodes are formed by processes and the links represent the possible communication channels, that is, a **structured overlay network** [7] in which the nodes of the network can propagate the information efficiently. Essentially, this layer is constituted by a slight modification of the Kademlia protocol [8]. In the Ethereum jargon, this protocol is known as *RLPx Node Discovery Protocol* [9]. In the remainder of this Section, we firstly describe Kademlia and afterward the Ethereum variant.

2.1.1. Kademlia protocol

The Kademlia protocol is an UDP based distributed hash table (DHT) system based on the XOR-metric for distance [8], that is the distance between two keys x and y is given by $x \otimes y$. The nodes have a unique m-bit (e.g. 160) identifier (ID) and are *logically* the leaves of a binary tree of size 2^m. The identifier of a node corresponds to the path from the root of this tree to the position of the node.

Each node in the network maintains m lists, which contain the contact information of the peers at a given XOR-distance from the node. In particular, the i-th list of a given node contains information regarding nodes at distance between 2^i and 2^{i+1} from it's ID. The maximal capacity of these lists, k, is chosen to minimize the probability that all the nodes in the lists fail at the same time. This parameter is known as *system-wide replication parameter*. Because of their maximal capacity the lists are usually denoted by the term *k-bucket*.

Each bucket is maintained sorted: at the head we find the least recently seen node and at the tail the most recently one. When a node receives a message from a sender, it uses the Algorithm 1 to update the contact table. One important feature of this algorithm is that, when the node discovers a new node, the latter is added only if one of the already known peers at the same distance is no more on-line. The rationale for this choice is due to the observation that the more a node has been on-line, the more likely it is that it remains up another hour [8].

The basic operation of this system is the *key lookup*. It is implemented by asking recursively for closer and closer nodes. A node selects the α peers closest to the searched key. This operation is efficient because it restricts the selection to the peers contained in the bucket in which the node would have inserted the searched key[6]. Afterward, the node sends *asynchronous* requests to these peers, that should reply with the contact information of the k closest nodes it knows. From the replied values the node takes only the closest α ones. The algorithm performs this step recursively. If in one of the phases no new closer nodes are discovered, the node retries with the k closest discovered nodes.

The lookup is fundamental to perform the task of storing a $\langle key, value \rangle$ pair in the

[6]It is possible that the k-bucket has less than α entries, in this case the node search also in other buckets.

Algorithm 1 Pseudocode algorithm to update a bucket upon receiving a message from a node. The sender and the receiver are denoted by the letters S and R, respectively.

```
distance ← S_ID ⊗ R_ID
bucket ← bucket containing nodes at the given distance
if S_ID ∈ bucket then
    move S_ID to the end of bucket
else
    if bucket not full then
        insert S_ID at the end of the list
    else
        H ← head(bucket)
        ping H
        if H replies then
            move H to the end of the list and discard S
        else
            evict H and put S_ID at the end of the list
        end if
    end if
end if
```

DHT, retrieving a resource from the DHT and get the contact information of a searched peer. To perform the lookup and manage the DHT, the Kademlia protocol relies on only four RPC functions:

- `PING` is used to check whether a node is still on-line or not
- `FIND_NODE` requests the replier to respond with the contact information of the k peers closest to the target
- `STORE` requests the receiver to store the given key-value pair
- `FIND_VALUE` requests the receiver to reply with the k nodes closest to the source. If the receiver has previously stored the key-value pair, it replies with the searched value.

Difference between RLPx node discovery and Kademlia The RLPx node discovery protocol is used only for discovery, so `STORE` and `FIND_VALUE` RPC functions are not needed. Each node is assigned to a 512 bit long ID and the XOR distance is calculated on the hash[7] of the IDs. Therefore, *conceptually* each node stores 256 k-buckets[8]. The replication parameter, k, is set to 16 and the concurrency parameter, α, is set to 3 [9].

[7] With "hash" we always intend the Keccak-256 hash.

[8] We provide only a general overview of the algorithm. For implementation optimizations we refer to [8].

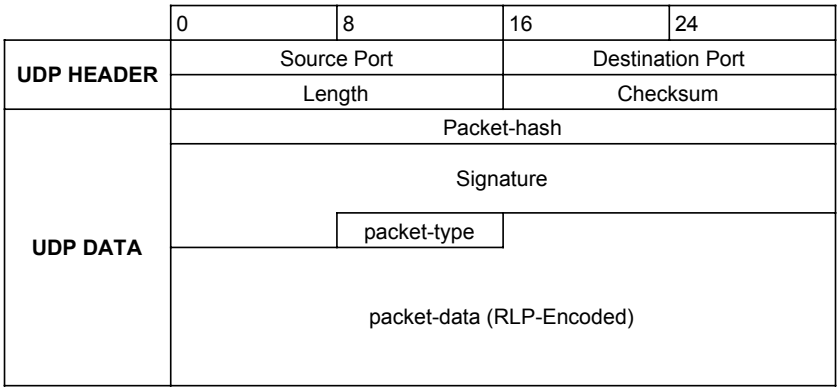

Figure 2: The structure of an RLPx Node Discovery package.

Packet Format The nodes communicate through UDP packets in which the payload is used to encode the messages of the Ethereum's network layer. Inside the UDP payload, the nodes should insert:

- the hash of the juxtaposition of signature, packet-type and packet-data, used to verify the integrity of the UDP datagram
- the signature, used to check the identity of the sender and can be verified through the node ID, which is the public key
- the packet-type, that is a byte that uniquely identify the packet type: `Ping` (0x01), `Pong` (0x02), `FindNeighbours` (0x03) and `Neighbours` (0x04) [9]
- the packet-data, that has a different format depending on the packet type and is encoded with the RLP algorithm (Section 2.2.1). We refer to [9] for the exact content exchanged with the different RLPx node discovery packets.

The packet format is illustrated in Figure 2.

Joining the network In order to join the network for the first time, a new node should generate a new public-private key pair[9] and know the contact of at least one participant. In Ethereum, this task is accomplished by hard-coding the contact information of some *bootstrap nodes* in the client's code. The aim of these nodes is to provide new nodes with contact information to other regular nodes that are already participating in the network.

RLPx uses its own URL scheme, the *enode*. In this scheme, the ID of the node encoded in hexadecimal format, the IP-Address and the TCP Port of the node are specified:

`enode://<hexadecimal-node-id>@<IP>:<TCP-Port>[?discport:<UDP-PORT>]`

The `discport` part is required only if the UDP port (discovery port) does not correspond to the TCP one. The default UDP discovery port is 30303.

[9]The public key is the ID and the private key is used to sign the packets.

2.2. Propagation Layer

The propagation layer's objective is to spread the information regarding the blockchain among all the peers. It exploits the contact information obtained in the network layer.

In this layer, we can identify three main components used to pursue this aim:

- *RLP*, the main encoding method used to serialize objects in Ethereum
- the RLPx protocol, intended as the transport protocol[10], which plays a role very similar to the OSI transport layer. For the sake of disambiguation, we refer to it as *RLPx Transport Protocol*
- the *Ethereum Wire Protocol*, useful to spread the blockchain information.

2.2.1. Serialization Algorithm

The Recursive Length Prefix (RLP) encoding algorithm is a fundamental building block in the Ethereum system. It is used both to serialize the content of the UDP and TCP packets sent as described in Section 2.1 and Section 2.2.2, and to reach a bit level consensus on the World State through the blockchain as described in Section 2.3.1.

This algorithm is *only* used to encode byte arrays of arbitrary length. It does neither try to deal with types nor considering signed integers and floating numbers [1]. The interpretation of the values is completely dependent on the message in the protocol, which should also specify the byte-size of the structures involved. According to the documentation [10], RLP has been chosen for its simplicity and its byte level consistency. For the formal specification of the algorithm, we refer to the Yellow Paper [1, Appendix B] and the RLP documentation [11].

2.2.2. RLPx Transport Protocol

The aim of this protocol is to provide a generalized mean to build arbitrary authenticated and encrypted protocols. The protocols built on top of this framework are known as subprotocols. Everyone can create a new subprotocol by simply selecting 3 ASCII characters to uniquely identify the protocol and by defining a list of packet types and the expected structure of their content, which will be encoded with the RLP algorithm.

The relationship between TCP, RLPx and the subprotocol packets is depicted in Figure 3[11]. The RLPx transport protocol packets are sent as payload in a TCP packet. If we do not consider the MAC codes used to encrypt the information, these packets consist of two fields:

- a header, which specifies information such as the size of the frame and the subprotocol that will be used

[10] `https://github.com/ethereum/devp2p/blob/master/rlpx.md#transport`

[11] For the sake of simplicity, here we report only single-framed RLPx packets, but we redirect the interested reader to the official documentation [12] to get details about multi-framed packets.

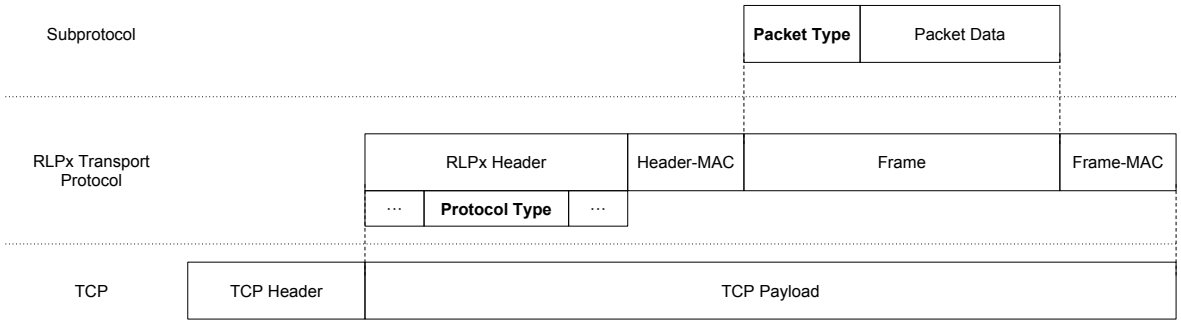

Figure 3: Relationship between TCP, RLPx and the subprotocols.

- the frame, which is the packet of the subprotocol.

In turn, the subprotocol packet contains a code that uniquely identify the message type (packet-type) and its content (packet-data), that is specific to the subprotocol. The content of this packet is serialized using the RLP algorithm.

When two peers establish a connection with the RLPx transport protocol, they perform a two-way handshake:

1. encoding handshake, used to exchange a cryptographic secret[12] that is used to encrypt and authenticate the subsequent RLPx messages between them

2. protocol handshake, in which the peers exchange and agree on the subprotocols and versions that both support (from now these pairs will be referred as *capabilities*).

To perform the protocol handshake, and in general to establish and keep the connection at this layer, the special subprotocol ĐΞVp2p Wire Protocol [13] is involved. This subprotocol does not have an identifier and reserves 16 message-types although only 4 are implemented. The `Hello` (handshake) message is used for the protocol handshake. This message specifies, among others, the protocol version, the capabilities, the port on which the client listens and the node ID. The `Disconnect` message notifies the receiver that the sender is going to disconnect itself optionally specifying an integer that encodes a reason[13]. The `Ping` and `Pong` messages are used to check whether the counterpart is still on-line or not.

After the protocol handshake, both peers know the shared capabilities. Since each protocol have a predefined amount of reserved message types, sorting them lexicographically makes it possible to build a data-structure to retrieve the protocol-type from the packet-type. We show how it works by means of an easy example shown in Table 1. Let's suppose that the peers share the capabilities (expressed as tuples ⟨subprotocol, version⟩) ⟨abc, 4⟩, ⟨abc, 5⟩ and ⟨zzz, 2⟩ and that the capabilities reserve 9, 11 and 5 message types

[12]It is beyond the scope of this report to describe the exact procedure. For further details, we refer to the official documentation [12] and to the Go Ethereum implementation `https://github.com/ethereum/go-ethereum/blob/master/p2p/rlpx.go`.

[13]We refer to the ĐΞVp2p specification [13] for a complete list of reason codes.

Capability	Reserved IDs	Effective Packet Types
-	[0x00, 0x10]	[0x00, 0x10]
⟨abc, 4⟩	[0x00, 0x08]	[0x11, 0x1A]
⟨abc, 5⟩	[0x00, 0x0A]	[0x1B, 0x26]
⟨zzz, 2⟩	[0x00, 0x04]	[0x27, 0x3C]

Table 1: Example of data-structure built after the protocol handshake.

respectively. The first row of the table represents the first 16 message types reserved by the RLPx transport protocol. Then, if one of the peers receives the message of type `0x1C`, it can determine that the message should be interpreted as a message of type `0x01` of the capability ⟨abc, 5⟩.

We notice that, although the RLPx Header (Figure 3) contains a field to specify the protocol type, it is not used by the implementations of Ethereum (at least `geth` and `ethereumj`[14]). If it would be the case, it would be sufficient to use the protocol-type in the RLPx header to identify uniquely a protocol. The reason probably lies in the fact that ĐΞVp2p protocol do not have an identifier.

2.2.3. Ethereum Wire Protocol

The Ethereum Wire Protocol (*eth*) [14] is an application level subprotocol of the RLPx transport protocol. It is used to spread the information about the blockchain and for the synchronization.

Currently, there are several versions of this protocol. Throughout this section we will consider only the versions 62 and 63 (which are compatible) that are currently supported by `geth` (v1.8.11).

The first message that should be exchanged between two peers is the `Status` message. This message type is used to exchange information such as the Ethereum Wire Protocol version, the network ID, the total difficulty of the heaviest chain known, the hash of the best known block and the genesis block's hash. This message should be sent only during the handshake phase. If the network ID or the genesis block's hash does not match or the supported *eth* protocol versions are not compatible, the peers should drop the connection since they are either on different chains or are not able to communicate with each other.

Version 62 - Model Syncing To spread the presence of one or more blocks to peers that are not aware of them, the `NewBlockHashes` message type is used. Moreover, the `Transactions` message type spreads transactions to peers who are not aware of them. It is specified that in the same session a peer should not send twice the same transaction to a recipient[15]. The `GetBlockHeaders` message type requests to the recipient a specified

[14]Since all the implementations should be interoperable, it means that all the implementations do not use this field or they deal with the case in which it is not used.

[15]To this extent, the `geth` implementation (in the file `eth/peer.go`) keeps track for each peer of the set of transactions hash (`knowTxs`) and the set of block hash (`knownBlocks`) known to be known by it.

amount of block headers descendant from the block with a given number or a given hash. The recipient of the message should respond with a `BlockHeaders` message in which it has the faculty to send a reply with less than the specified amount of headers. Clearly, if the recipient of a message is not aware of any descendant of the given block, it sends a valid empty reply. Furthermore, to request and receive the real content of the blocks, the peers have the `GetBlockBodies` and `Blockbodies` messages. The requester specifies the hashes of the blocks it wants and the recipient replies with the bodies (i.e. the transactions and the uncles) of the required blocks. Finally, the `NewBlock` message spreads a single new block.

Version 63 - Fast synchronization From version v1.3.1 of `geth`[16], it is possible to perform a fast synchronization. This synchronization type does not require that a node performs *all* the computations happened during the history (i.e. the whole EVM instructions Section 2.5.1). Indeed, the synchronizer downloads along the blockchain the transaction receipts which encapsulate useful information about the execution result of the transactions. This allows the synchronizer to deal only with the verification of the proof-of-work Section 2.4.1. At least in `geth`, this synchronization is possible only by the first synchronization for security reasons[17]. After the synchronizer reaches a *pivot point* (i.e. last block minus 1024), it retrieves the whole current state from the other peers and subsequently processes the blockchain normally. To perform this synchronization, the clients have at their disposal the `GetReceipts` and `Receipts` messages, which are the request for the receipts given the hash and the replies respectively. Besides these, there are also the `GetNodeData` and `NodeData` message types which provide the mean to query and send the required version of the state: the first one takes as input a variable number of hashes and the second one replies with the content.

2.3. Data layer

In the previous layers, we formed the network and we defined how the data is disseminated on it. Now, we need to define which data to propagate. The Data layer consists of the data structures and the data types that are present in the Ethereum system. With the term *data types*, we mainly mean the abstract data types which identify the common Ethereum objects, such as the state, the accounts, and the transactions.

2.3.1. State

The **World State**, referred as the *state*, in its simplest definition, is a mapping between account addresses and account states.

Figure 4 represents it. The block header contains 15 fields, among which the `stateRoot`, the `transactionsRoot` and the `receiptsRoot`. All these three fields are a hash of the root of a Merkle Patricia tree data structure:

[16] `https://github.com/ethereum/go-ethereum/releases/tag/v1.3.1`
[17] `https://github.com/ethereum/go-ethereum/pull/1889`

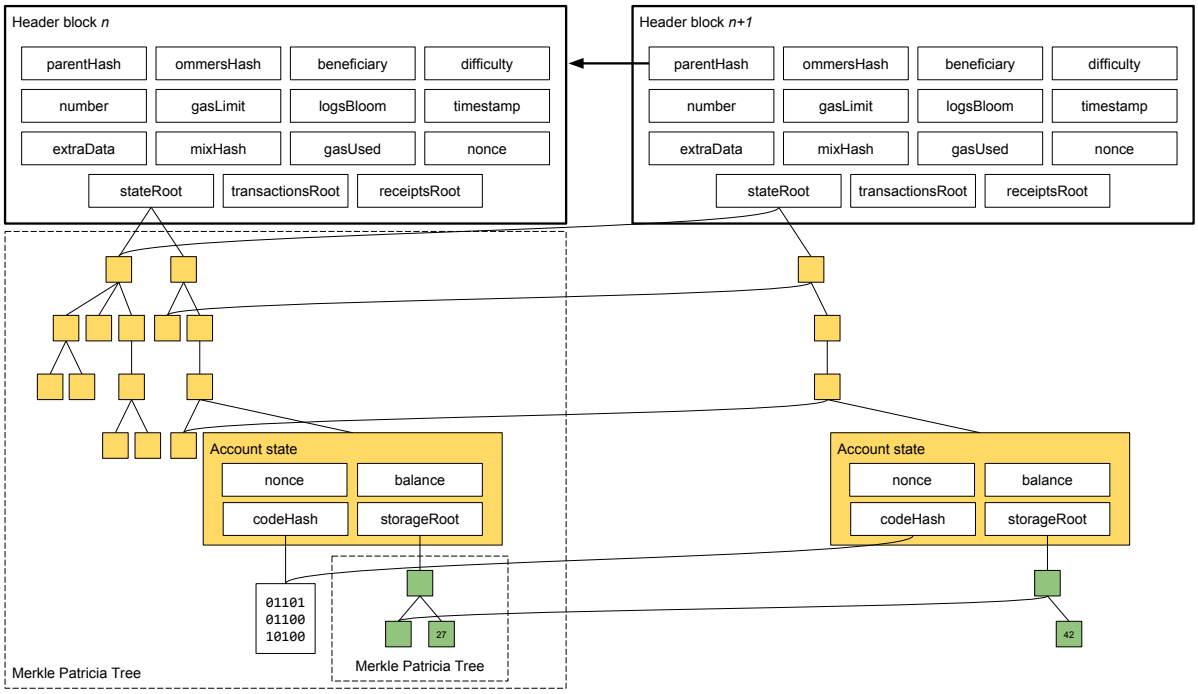

Figure 4: Representation of the World State.
Source: Adapted from `https://ethereum.stackexchange.com/a/757`

- the `stateRoot` represents the state tree, after that all the transactions are executed and the finalisations are applied (for a complete specification of the fields, refer to [1]), storing the mapping between account state and account address
- the `transactionsRoot` represents the list of the transactions included in the block
- the `receiptsRoot` represents the receipts list of the transactions included in the block, which shows the *effect* of each transaction. In [15], the Ethereum's founder Vitalik Buterin writes that with the receipt information someone can answer queries like "Tell me all instances of an event of type X (e.g. a crowdfunding contract reaching its goal) emitted by this address in the past 30 days".

Some other block header fields are described in Section 3.7. For a complete fields list and description, we refer to the Ethereum protocol specification [1].

Merkle Patricia tree A Merkle Patricia tree is a data structure which derives from a radix tree reducing the space complexity [16]. It has a single root, each node is the hash of its children and the leaves are the actual data, that is, for the case of the `stateRoot`, the accounts states, which in turn include the `storageRoot`, a hash of another Merkle Patricia tree representing the storage contents of the account state.

Since each node is the hash of its children, if any data in the tree changes, recursively and correspondingly all the ancestors nodes have to change from the changed node to the root node. This property allows to uniquely identify a tree having just the root node.

This is worth to notice because it allows the nodes of the network to verify that big data structures (like the World State) correspond to the one contained in the blocks' header by simply comparing a 256-bit long hash. Moreover, this feature can be exploited to create the light nodes as described in Appendix A.

2.3.2. Accounts

The accounts are also called the *state objects* and are essential for the user to interact with the Ethereum blockchain via transactions.

There are two types of accounts:

- **Externally Owned Account** (EOA) (also referred to as simply *account* or *non-contract* account)
- **contract account** (also referred to as *contract*), which has EVM code associated with it and is controlled by its contract code.

An *EOA* has no EVM code associated with it and is controlled by a private key. This type of account can send a message to another EOA, that is a value transfer, or to a contract account in order to trigger the execution of code. The state of an account is essentially its balance.

A *contract account* has EVM code associated with it and is controlled by it. This type of account cannot send messages or transactions on its own, but only as a response to a trigger. The state of a contract is its balance and its contract storage. A contract code is executed by the EVM, can manipulate its own persistent storage and can send internal transactions (i.e. message calls) to other contracts.

Both the types of account have an associated *nonce*. In the case of an externally owned account this value corresponds to the number of transactions sent by the account while in the case of a contract account it corresponds to the number of contract creations performed by the contract. Obviously, this value is always positive and can only be increased.

When creating a transaction, the EOA should specify its nonce. This guarantees that the order of transactions of a single account are processed in the order specified by the sender. Without this expedient something unforeseen can happen, for example, the balance of the account can get under a certain threshold and so other transactions cannot be performed. Since contracts cannot perform transactions but can still create other contracts once invoked, the nonce is used to guarantee that each different created account have a different contract address. Indeed, the contract address is obtained as function of the address of the creator and its nonce [1].

2.3.3. Messages and transactions

As we said in Section 2.3.2, the accounts can send messages or transactions to other accounts depending on the account's type.

A transaction is a single instruction constructed by an actor externally to the scope of Ethereum [1] and it is serialized and included in the blockchain. It can be used to transfer ether[18] or to trigger contract code execution. A transaction represents a message call or the creation of a new account with associated EVM code.

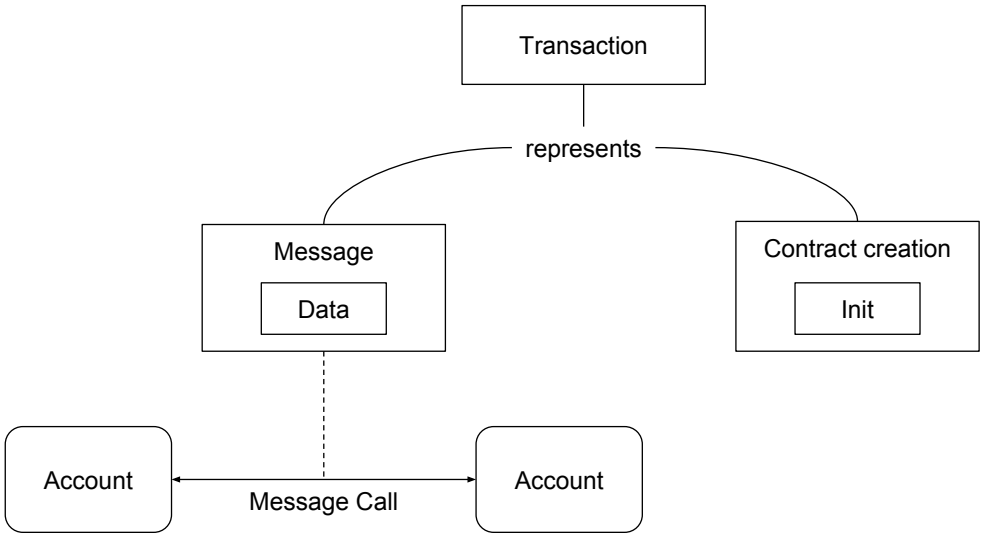

Figure 5: Representation of the relation between message and transaction.

A message is a virtual object that is not serialized and exists only in the Ethereum execution environment. A message between two contract accounts is also called *internal transaction*, because of its dual nature of being structured as a transaction and of being managed internally of the Ethereum execution environment.

The relation between transactions, messages and contract creation is represented in Figure 5. We notice that messages and contract creation have the same structure and both contain an unlimited size byte array. The interpretation of this value depends on the type of transaction: if the value is interpreted as `data`, it represents the input for the message call, else if it is interpreted as `init`, it represents a piece of EVM code used to initialize the storage of the smart contract and to return the EVM code that will persist on the world state. We can distinguish the two types of transaction depending on the receiver's address: if it is empty, it is a contract creation, otherwise it is a message call.

Figure 6 shows the message call with regard to a contract account. If the destination account is a contract, the message triggers the execution of its code (Figure 6a). Instead, if the message sender is a contract, the message is the result of the code execution (Figure 6b).

[18]The currency used within Ethereum.

(a) Message call to a contract account which trigger the execution of code.

(b) Message call from a contract account which represents the result of code execution.

Figure 6: Message call to and from a contract account.

2.4. Consensus layer

The ultimate aim of the blockchain technology is to provide a **total order to transactions** in a distributed ledger [3] without relying on a trusted third party. This permits to solve the double spending problem [2]. Moreover, in Ethereum the order of transactions can also affect the execution of smart contracts by altering the content of their storage.

In order to describe how the nodes reaches the consensus, we briefly describe the algorithm to agree on the transaction order (Section 2.4.1) and thereafter we describe the common state transition procedure that describe how to move to a new valid state given a transaction (Section 2.4.2), that can be either a contract creation (Section 2.4.3) or a message call (Section 2.4.4).

2.4.1. Consensus Algorithm

Finding an agreement on the order of transaction (i.e. the actual blockchain) and the world status is crucial, thus multiple consensus algorithms were proposed [3]. Ethereum follows an idea very close to the consensus algorithm of Bitcoin, which is also known in the literature as **Nakamoto consensus** [17].

The basic idea of this algorithm consists in: (1) accepting only valid blocks with regards to some validation criterion, (2) create new valid blocks by using a proof-of-work algorithm, (3) relying on a selection rule to choose between two different valid forks depending on the amount of work performed in each fork.

Validation The *validation criterion* used to determine whether a block is valid or not consists in:

- checking that the blocks and transactions are well-formed
- checking that the block header is valid
- re-performing all the transactions to verify whether the transaction receipts and the state root contained in the propagated block (Figure 4) are valid, i.e., corresponds to the values computed locally. This includes also re-executing *all* the EVM computations.

Proof of Work Ethereum uses an improved version of the Dagger-Hashimoto algorithm [18], known as Ethash [1, Appendix J] as PoW algorithm. The rationale to use of this memory intensive algorithm is its ASIC-resistance. ASIC are specialized hardware used massively in the Bitcoin ecosystem. This kind of hardware is a risk for centralization, because to begin mining new blocks and maintaining the infrastructure a big initial investment is needed and only few entities and definitely not small private parties can afford this cost.

Essentially, to create a valid block the miner should find a mixHash and a nonce (Figure 4) for the block. The PoW algorithm takes as input the block header without nonce, the candidate nonce and a big dataset (initially 1 GiB) known as **DAG**, and returns the mixHash and a number n. The puzzle is resolved if n is smaller than 2^{256} divided by the difficulty of the block. Clearly, the higher the difficulty the higher the number of tries to find a suitable nonce and, consequently, the consumption of resources.

The DAG can be pre-computed and is fixed for each epoch, i.e., 30000 blocks, that corresponds roughly to a number of hours in the range between 100 [19] and 141 [20]. To verify that the mixHash and the nonce are valid only a cache for the DAG is needed. This data-structure is required also to generate the DAG itself. At each epoch the DAG and the cache change and their size increase of 8 mebibytes and 128 kibibytes, respectively. We refer to the yellow paper [1, Appendix J] to get more details on how these data structures are computed.

Selection Rule The *selection rule* is required to avoid the infamous double spending problem. Indeed, in Bitcoin (and as well in Ethereum) the assumption is that the majority of computing power belongs to good players who will follow the rules. Therefore, in order to prevent bad agents to rewrite the transactions history with a high probability [2], the issuer of new blocks should prove that she invested resources in its creation by solving a computational heavy task. This mechanism is known as **Proof-of-Work** (PoW).

In Bitcoin the selection rule consists in accepting the longest chain that corresponds roughly to the one with more work invested on it. The Ethereum community *claims* that Ethereum implements a simplified version of the Greedy Heaviest-Observed Sub-Tree (GHOST) selection rule [1]: briefly, the stale blocks contributes to the difficulty of a fork. The aim is to allow an increase in the transaction throughput (by decreasing the block issue interval) while preserving the same security guarantees of the original Bitcoin consensus protocol. As noted in [21], Ethereum *does not* implement a simplified version of the GHOST selection rule. Indeed, currently the Ethereum's rule consists in choosing the fork with the highest accumulated difficulty [1]. Each block in the chain has an associated difficulty that determines how much effort is needed to mine a new block. This parameter depends *solely* on the difficulty of the previous block and the time that elapsed between the previous block's timestamp and the new block's timestamp, corrected with some bounds to avoid sudden decreases or increases in this value. The claims to use the GHOST rule are motivated by the fact that the headers of stale blocks (up to six blocks before, the *ommers*) can be included in the blockchain and rewarded, but they do neither contribute to the difficulty of the blocks nor the transactions are verified to be valid [21].

Thus, at the state of the art this rule resembles the Bitcoin one.

2.4.2. Transaction Execution

The transaction execution is the mechanism through which the world state is updated. It represents a transition from one valid state to another valid state.

A transaction specifies a *receiver* and the *value* that must be transferred from the sender to the receiver. Moreover, to avoid the abuse of the resources (CPU and storage) of the full nodes forming the network and to ensure that all executions terminate, the concept of *gas* is introduced. In this execution model each action that must be performed by the members of the network has an associated cost expressed in gas. In particular, each EVM byte-code instruction, increase in the storage space by a contract and the transaction itself have a fixed associated cost. A transaction specifies its *gas price* and its *gas limit*. The former is the price of a unit of gas and is bound to a particular execution. The higher this price the higher the possibility that the miner will include this transaction in the blockchain. Usually the miners advertise the minimum gas price they are willing to accept. The latter is the maximum amount of gas the executor is ready to consume for this particular execution.

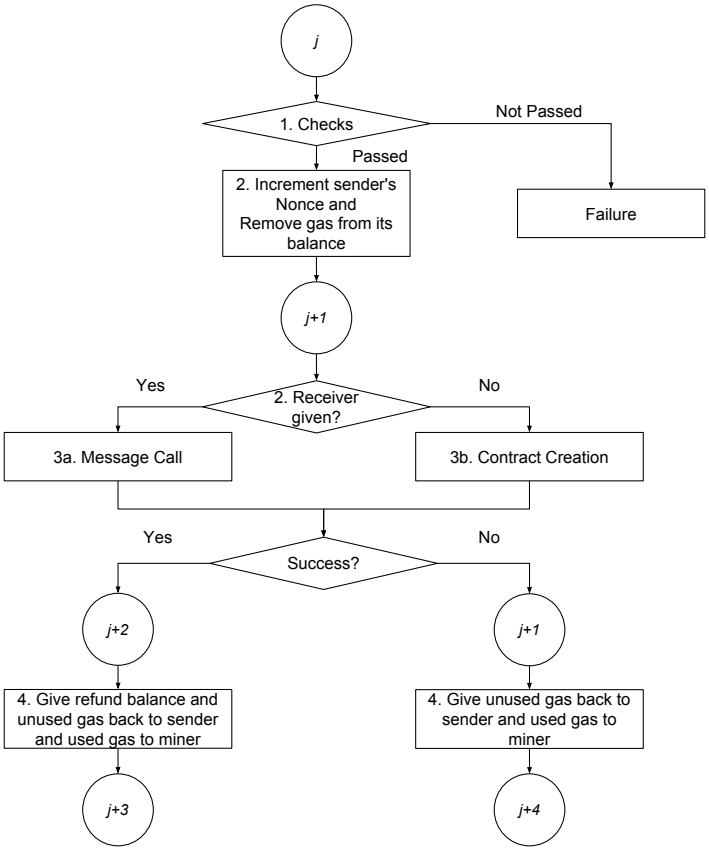

Figure 7: The steps of the transaction execution algorithm.

Figure 7 summarizes the steps of the algorithm for the transaction execution:

1. The transaction has to pass some simple validity checks, e.g. the transaction should be a well-formed RLP encoded string and the initiator of the transaction should have a balance big enough to afford the transaction. Moreover, since a transaction should be included in a block and the block has in turn its own gas limit, it should be also true that the sum of the accumulated gas used by the already included transactions and the gas limit of this transaction are smaller than the block's gas limit. If these checks fail, the transaction is simply not included in the block in case of miner or it indicates that the block is invalid in case of verifier.

2. The nonce of the initiator of the transaction is incremented by one and its balance is reduced by the product of the gas limit and the gas price. This modification to the state is irreversible.

3. Now, depending on whether the receiver address is given or not we should make a distinction between:

   a. Message Calls (Section 2.4.4)

   b. Contract Creation (Section 2.4.3).

   During the execution of the message call or contract creation the system keeps track of the *transaction substate*, i.e., some important information that are later used to complete the state transition. The transaction substate includes the *touched accounts*, the set of accounts that will be discarded following the completion (*self-destruct set*) and the *refund balance*, that is an amount of gas that is incremented by removing elements from the world state, e.g., by setting a non-zero value to a zero value in the storage or by removing a contract from the state.

4. Once the message call or the contract creation are concluded, the sum of the remaining gas and the refund balance are refunded to the initiator of the transaction at the transaction's gas price[19]. If the message call or contract creation did not successfully complete, the state is reverted and the refund balance is zeroed, since its modifications to the state are not considered. The gas used is given to the beneficiary address (i.e. the miner) who built and finalized the block. Finally, the self-destruct set and the touched accounts that became empty or dead after the transaction should be deleted from the world state.

2.4.3. Contract Creation

The contract creation is the act by which a new contract is deployed on the system. It can be triggered either by a transaction or during the execution of an existing contract. In the remainder of this section, we use the neutral term *sender* to refer to the entity who starts the contract creation. In the first case the sender corresponds to the transaction

[19]This sum cannot exceed the initial allocated price [1], in other words the refund balance can be used to mitigate the transaction cost, but not to profit.

initiator and in the second case it corresponds to the address of the contract that is executing.

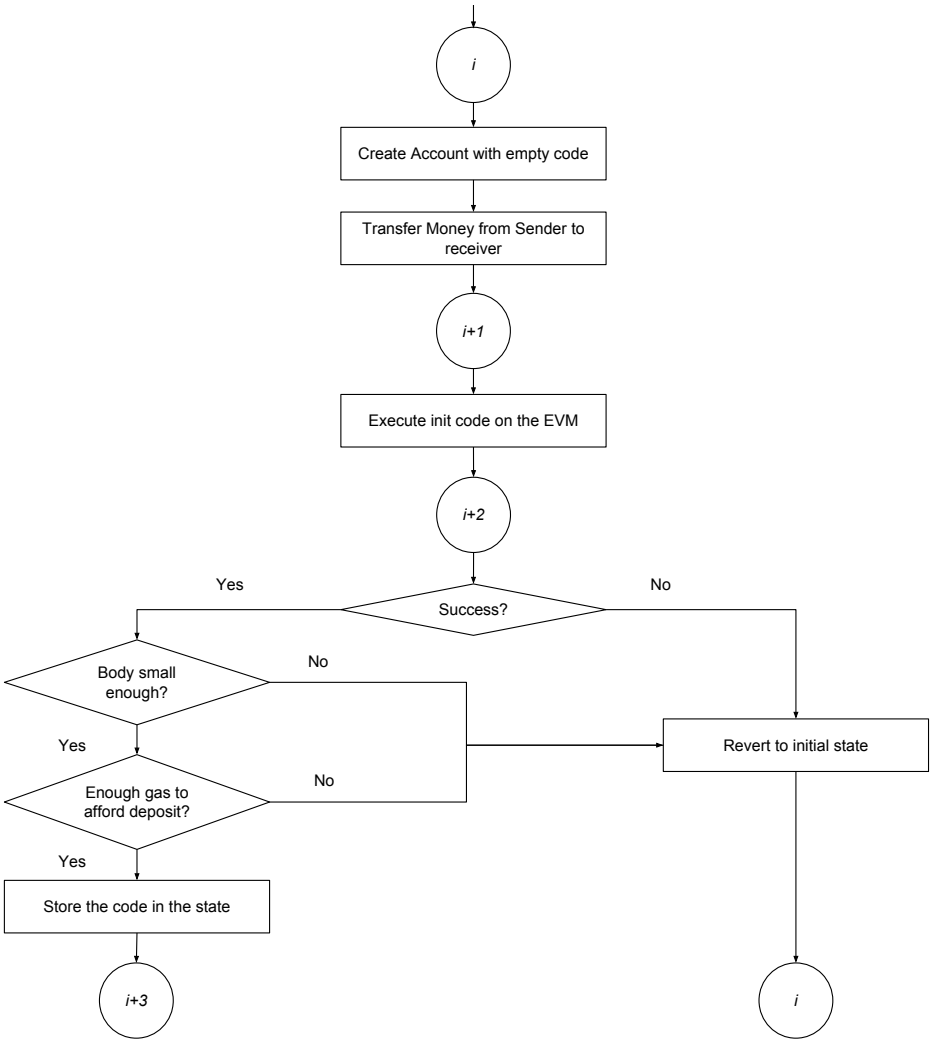

Figure 8: The steps of the contract creation algorithm.

The steps of the algorithm and the state transition are depicted in Figure 8. The circles represent the different states. Firstly, the 160-bit identifier for the contract account is determined as function of the sender's address and its nonce. The value specified in the contract creation is transferred from the sender to the brand new contract account. Afterward, the *init* code (Figure 5) is executed on the Ethereum Virtual Machine. This piece of code initiate the storage of the contract account and returns the *body*, i.e. the contract code that will persist on the account state. If during the execution an out-of-gas exception occurs, the state is reverted to the initial state as if the contract creation did not take place. If the execution of the init code completes successfully, the creator must pay an amount of gas proportional to the size of the body, because it must be stored by

all the full nodes. If the gas remained after the execution is not enough to afford this cost or the body of the contract code is too big, the state is reverted [1]. If all the whole procedure succeed, the hash of the body is saved on the contract account state. Usually, the implementations use this hash as a key for the contract code as showed in Figure 4.

2.4.4. Message Call

The message call is the mean to invoke a contract code and can be either triggered by a transaction or by the execution of a contract. A message call takes as input a state and the message itself (the data field) and returns an output, that is from the point of view of Ethereum only relevant if the message call is triggered by an execution.

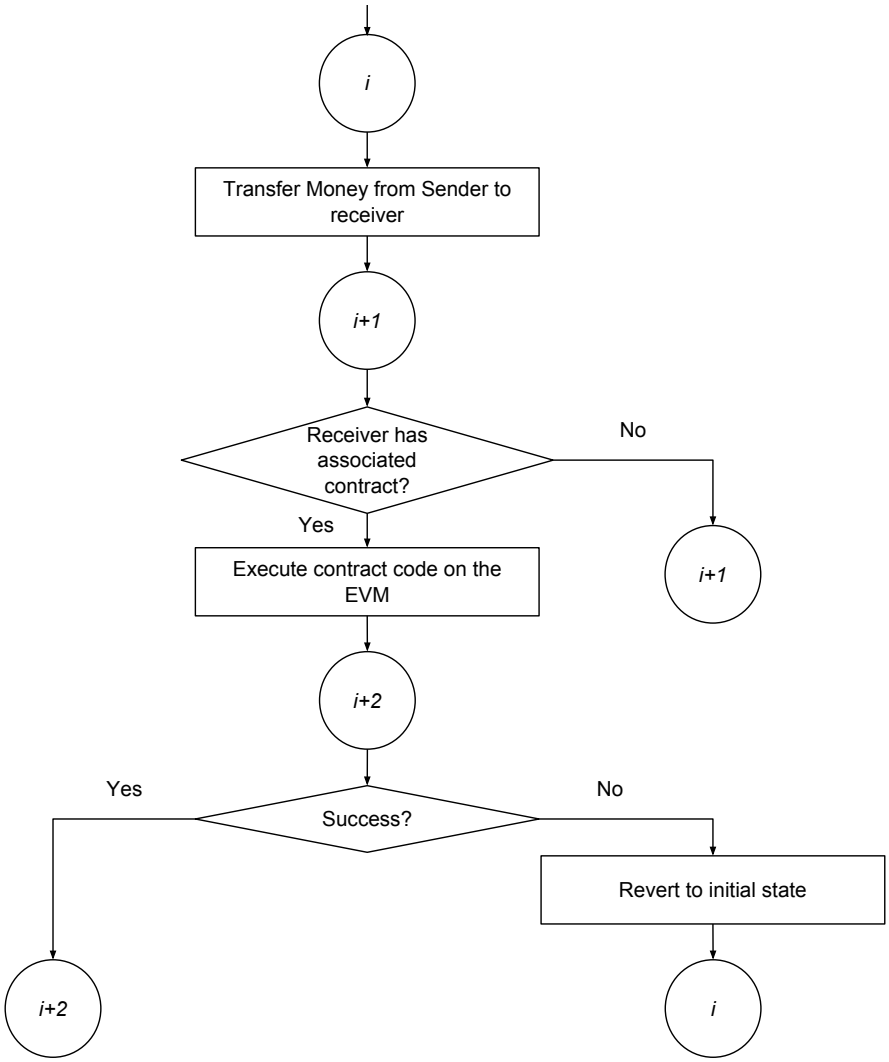

Figure 9: The steps of the message call execution algorithm.

The first step in the message call algorithm is the transfer of value from the sender

to the receiver account. If the recipient of the message is an externally owned account, the message call terminates, otherwise the contract of the account is executed on the EVM. If during this execution an error occurs (e.g. stack overflow/underflow, out of gas exception), the used gas is not refunded and the state is reverted to the state prior to the execution. This means that the sender looses the gas but recovers the sent value. If the execution succeed, this state persists. The steps of the algorithm are illustrated in Figure 9.

2.5. Application layer

So far we built the structure needed to communicate and agree on the state. Ethereum proposes an execution environment where code can be executed. This enables the implementation of arbitrary business logic opening possibilities towards a wide variety of applications.

The application layer comprises the smart contract, a collection of functions able to modify the state and executed on the Ethereum Virtual Machine (EVM).

2.5.1. Ethereum Virtual Machine

The EVM is an abstract computing machine that enables the nodes of the Ethereum network to execute smart-contract codes.

Figure 10 shows the components of the Ethereum Virtual Machine. The dashed lines represents the fact that there exists an EVM instruction that allows the transfer of data from one component to another. Like the JVM, the EVM is a stack-based machine. In this system the word size and the stack item size are both 256–bit[20], and the stack capacity is 1024. The EVM memory consists of a simple byte array and exists only during the execution. The size of this array is allocated using an on-demand logic. The storage is a key value hash table (stored as a merkle tree) that is persistent and is part of the state of the contract account. The program is stored on the blockchain and therefore cannot be modified[21]. In addition to the aforementioned structures, the EVM can read the information provided in the transaction/message that initiated the execution. In particular, the EVM can access the input field which contains the arguments for a peculiar execution. Moreover, the EVM can access the balance of all addresses and write to the logs that are stored in the blockchain. The logs are used both as a cheap output and also to notify external applications that something worth mentioning has happened. We provide more details about events in section 2.6 and Appendix B. For a full instruction list and specification of the EVM we refer to the official specification [1, Appendix H].

Unlike the Bitcoin's *Script* language [5] the Ethereum Virtual Machine can express and supports the execution of loops. To avoid the abuse of the resources (CPU and storage)

[20] The motivation for this choice is to facilitate the Keccak-256 hash scheme and elliptic curve computation that are pervasive in Ethereum.

[21] There exist techniques based on the strategy pattern to provide new smart-contract version updates as explained here: `https://ethereum.stackexchange.com/questions/2404/upgradeable-smart-contracts`

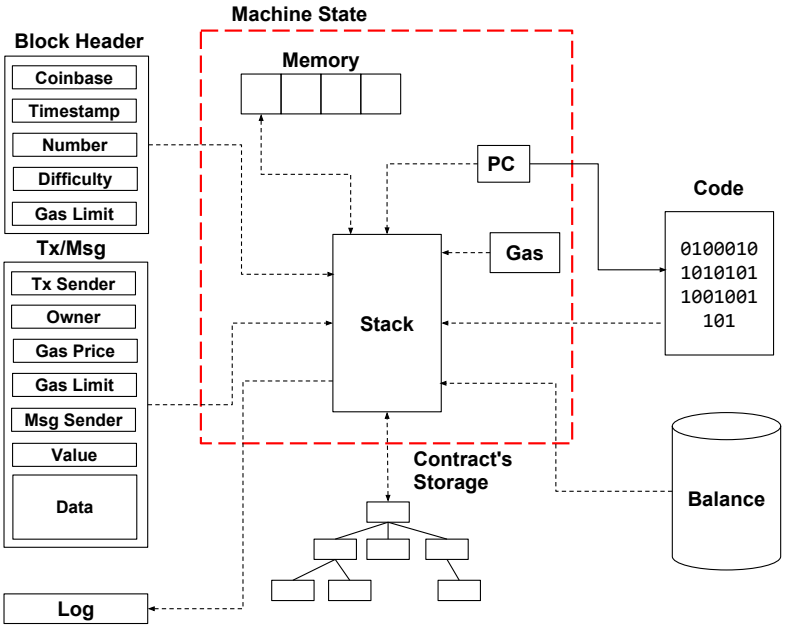

Figure 10: Overview of the Ethereum Virtual Machine

of the full nodes forming the network, Ethereum introduces the concept of *gas* [1], which was described in Section 2.4.2. In this execution model each instruction and each increase in the size of the memory or storage are bound to a cost expressed in gas.

Before executing a *smart contract*, i.e. a program on the EVM, the initiator of the transaction allocates a certain amount of gas that it is willing to spend for this execution. If during the execution all the gas is consumed, an *out of gas exception* is raised.

The *machine state* represents the current configuration of the Ethereum Virtual Machine interpreter. It consists of the remaining gas, the program counter, the memory and the stack.

As already explained in Section 2.4.2, a transaction can trigger the execution of contract creations and message calls. In turn, these can invoke the execution of other contract creations and other message calls. To deal with this eventuality, the EVM has at its disposal a *call stack*: when a new contract call or message call is executed, the current machine state is stored on the call stack and a new machine state with the invoked code is created. When the invoked code terminates, its activation record is popped from the call stack and a return status flag is put on the activation record of the invoker: 1 indicates that the code threw an exception and 0 indicates a successful termination. The call stack's depth is limited to 1024 [1]. Although the yellow paper does not define explicitly the call stack, more details can be found in the literature [22] and in the go-ethereum implementation[22].

[22] An example of use of the call stack is the function `opCallCode` in the file `core/vm/instructions.go`: it is clear that the implementation exploits the usual function call stack of the language to implement

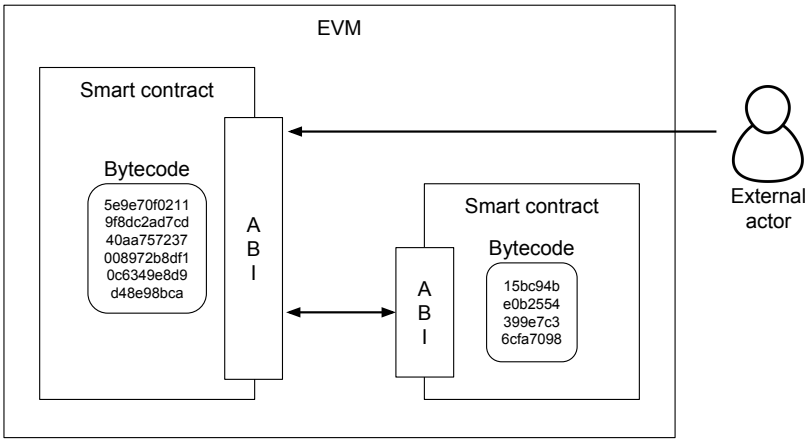

Figure 11: ABI representation.

2.5.2. Smart contract

The smart contracts in Ethereum are part of the application layer. They can be referred to as implementing the business logic of the applications that runs on the Ethereum system. A smart contract implements functions and has a state which can be persisted and accessed in a subsequent execution. They can implement *almost* any function which can be implemented in any Turing-complete machine. The difference is due to the gas needed for a transaction to be executed: since the gas can not be infinite, there will be always an upper bound on the possible total amount of computation [1].

Usually, a smart contract is written in a high-level language, e.g. Solidity (Appendix B) or Vyper[23], then compiled in EVM bytecode. The contracts communicate with each other and with external actors through their Application Binary Interface (ABI) represented in Figure 11. This interface serves as a declaration of the functions implemented by the contract and their arguments, thus another actor can trigger the execution of it and get its result. The execution takes place in the EVM.

Although there is a common agreement on the ABI format [20], this abstraction is not part of the core Ethereum protocol meaning that anyone can define and expose its own ABI, but the caller have to comply to the format used by the callee.

2.6. External Interaction

So far we described how Ethereum *internally* works, but we did not provide a description of how a user or an external application can *interact* with the system, e.g. how transactions are sent by users or how an application can read the balance of a given address.

To this extent, the Ethereum community has developed a JSON RPC API [23] compliant with the JSON RPC 2.0 specification [24]. JSON RPC is stateless and can

the EVM call stack.

[23] `https://github.com/ethereum/vyper`

be used on top of diverse protocols (e.g. HTTP). It allows external actors to invoke the exposed API methods by sending JSON encoded requests. These should specify the API version, the API method, the parameters encoded as a list and a nonce that binds a request to a reply. For the sake of clarity, we show an example that calls the method `eth_blockNumber`, which returns the number of blocks:

```
curl -X POST -H "Content-Type: application/json" --data \
'{"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":1}' \
http://localhost:8545
```

The server replies with a JSON string that contains the result and the same nonce as the request.

In addition to the JSON RPC API, a JavaScript API was developed. It is provided as a JavaScript library, `web3`[24], that allows JavaScript code to communicate with a running Ethereum client. It is simply a convenient JavaScript wrapper for the JSON RPC calls [25]. For a complete list of the methods supported by the two APIs, we refer to the respective documentations [23, 25].

The different Ethereum client implementations provide options to start JSON RPC server on top of different protocols: for example, `geth` allows to start the server on top of HTTP, WebSocket and IPC Socket, i.e. shared memory. Moreover, this client gives the possibility to the users to start or to attach to a given running instance a JavaScript Runtime Environment REPL console [26], which can execute JavaScript programs and access the JavaScript API methods.

ĐApp A so-called *decentralized Application* (ĐApp) is essentially the union of a convenient front-end (e.g., HTML, JavaScript) and a smart contract back-end. The front-end communicates with the back-end through the JSON RPC API or the JavaScript API (e.g. using web3).

3. Scalability

The aim of this section is to study the scalability of Ethereum. Although the scalability of permission-less blockchain, and of Ethereum in particular, is a well-known problem and a major concern in the respective communities as we present in Section 3.1, in the literature there are a limited number of scientific papers that address this problem. We give a little survey in Section 3.2. Next, we discuss the Ethereum scalability with regards to the three axis of the Scale Cube [6]. For each axis, first we give a brief description, then we provide a virtuous example of an architecture which scales on the axis, hence we conclude with an analysis of the current status of Ethereum and some of the proposals coming from the community to improve the scalability with respect to the inspected axis.

[24] `https://github.com/ethereum/web3.js`

3.1. To scale or not to scale

Before start discussing about the scalability in Ethereum, it is worth clarifying whether it is a real concern or not.

In [1], Gavin Wood defines Ethereum as "a project which attempts to build the generalised technology; technology on which all transaction-based state machine concepts may be built", suggesting that, even systems for which is expected a high number of transactions can be built with Ethereum. Following this aim, one typical example is the case of Visa. Visa is capable of handling 56000 transactions per second[25], while Ethereum can roughly process 15 transactions per second[26]. This value is confirmed also by our tests whose results are reported in Section 3.7. This comparison points out that Ethereum is far away from possibly implementing all such systems, showing that the scalability from this perspective seems crucial.

In January 2, 2018, the Ethereum Foundation announced two subsidy programs both intended to fund projects on the scalability research and development[27] recognizing that "scalability as perhaps the single most important key technical challenge that needs to be solved in order for blockchain applications to reach mass adoption", followed by the beneficiary announcement[28].

This shows how all the Ethereum community founders included believes that the scalability is a major concern and an actual bottleneck in the employment of the technology.

The search for the scalability is not only a concern of Ethereum [3]. Other blockchain proposals move towards this direction making the improved scalability with regards to the mainstream blockchains their forte[29].

3.2. Background

In this Section, we point out the usual concerns about the permissionless blockchain system, in particular Bitcoin, and the related works regarding Ethereum.

Bitcoin One major concern about Bitcoin and all permission-less cryptocurrencies in general is the limited **maximum transaction throughput** [27, 28, 29]. Thus, if the usage of this payment system augments, not all transactions emitted can be processed in a predictable and bounded time[30]. Moreover, the choice of the transactions to include in the blockchain depends on the will of the miners. Therefore, it is more likely that the more profitable transactions are included in the distributed ledger [27, 1].

[25] https://usa.visa.com/dam/VCOM/download/corporate/media/visa-fact-sheet-Jun2015.pdf

[26] https://medium.com/l4-media/making-sense-of-ethereums-layer-2-scaling-solutions-state-channels-plasma-and-truebit-22cb40dcc2f4

[27] https://blog.ethereum.org/2018/01/02/ethereum-scalability-research-development-subsidy-programs/

[28] https://blog.ethereum.org/2018/03/07/announcing-beneficiaries-ethereum-foundation-grants/

[29] https://eos.io/

[30] https://blockchain.info/charts/mempool-size

The maximum limited transaction throughput is due on the one hand to the 1 MB *maximal block size* and on the other hand to the 10 minutes *block interval* [30, 29]. Increasing the former would certainly augment the transaction throughput and decrease the transaction fees, but it would also slow down the propagation time, thus increasing the possibility of forks. Decreasing the latter would increase the transaction throughput and increase also the possibility of forks. The increase in the number of forks reduces the security of the whole system [30]. To address these bottlenecks multiple solutions were proposed. The Blockchain systems Bitcoin Cash and Litecoin try to augment the maximal transaction throughput by modifying the Bitcoin client to have a maximum block size of 8 MB and to have a block interval of 2.5 minutes respectively. The GHOST [30] protocol consider also the stale blocks to permit to diminish the block interval while preserving the same security guarantees of the original proof of work as we already described in Section 2.4.1.

In addition to these considerations, a number of confirmation blocks are needed to be sure that the transactions are really confirmed [28], i.e. it is unlikely that the block in which they are included can be discarded in favor of other blocks. This uncertainty in the acceptance of a block is also known as **lack of Consensus Finality** [3] and is due to the probabilistic nature of the PoW consensus algorithm. Currently, in Bitcoin the number of confirmation blocks is set to 6 [5]. This means that after the insertion of a transaction in a block at least an hour must be waited to be sure that it would be confirmed.

Another concern with this technology is the need to store an increasingly large amount of data to keep the desired security guarantees. Currently, the minimum requirement to run a Bitcoin full-node, that is a node that verifies all the transactions, is 145 GB[31].

Other major concerns are related to the *cost* of the system, especially the cost per confirmed transaction [29] which is due to the mining cost, the transaction validation, the bandwidth and the storage.

Finally, the bootstrap time, that is the time needed for a new node to download and process the whole transaction history, is another key factor that can contribute to the scalability (scaling out) [29] and the ability of new nodes to join the network.

Ethereum The previous considerations about the scalability of Bitcoin apply also to Ethereum. In addition, the introduction of smart-contracts in Ethereum makes the problem very similar to database replication and in particular *state machine replication* [3]. Also Wood [1] argues that it is very difficult to reach a high degree of scalability by parallelizing transactions in this system because it is essentially a state transaction machine. Indeed, the state in Ethereum influences the smart contract execution and therefore the majority of transactions are dependent from previous ones, thus making Ethereum **stateful**.

[31] https://bitcoin.org/en/bitcoin-core/features/requirements

3.3. The Scale Cube

The *Scale Cube*, as shown in Figure 12, uses the representation of a cube drawn on a 3-dimensional Cartesian space to define three different scaling directions an architecture can develop in order to growth and shrink along with the demand. Although in a Cartesian space we could measure the cube size, the Scale Cube does not provide actual metrics to quantify the scalability, but rather a way of think about scale; that is what we mean with *scaling directions*.

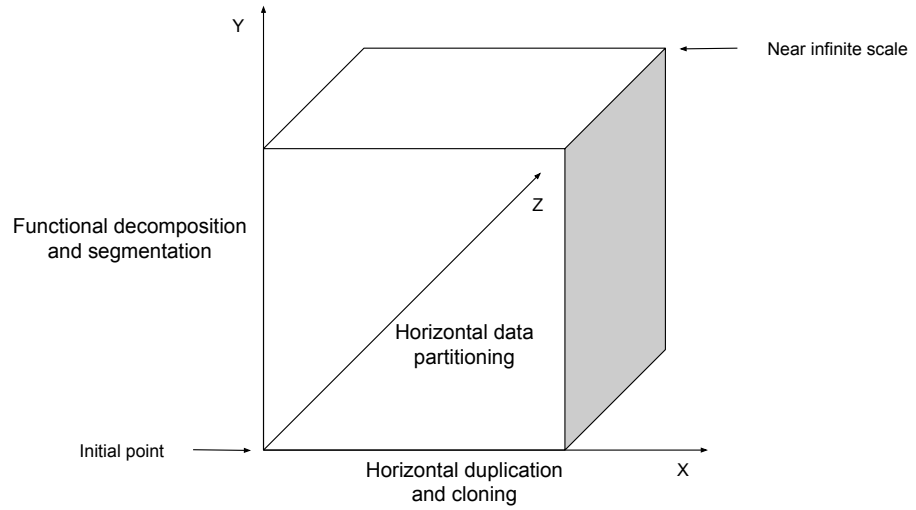

Figure 12: The Scale Cube.

The use of one or two axes does not preclude the possibility to scale on the third axis. The initial point with coordinates $(0, 0, 0)$ means least scalability. The prototypical system at the initial point consists of a single monolithic application and storage retrieval system likely running on a single physical machine [6]. Of course, it might scale up, that is it could run on a more powerful machine, but it won't scale out, hence it will not take advantage of a distributed architecture. All of the three axes scales well from a transaction perspective, that is, in our case, the transaction throughput.

In the Sections 3.4, 3.5 and 3.6 we describe the single axes with aid of examples, we argue where Ethereum is placed with respect to the specific axis and we describe some solutions proposed to augment the scalability of the system.

3.4. X-Axis: Horizontal Duplication

The x-axis of the cube of the scalability is concerned with the horizontal duplication and cloning of services and data with absolutely no bias, running each identical copy of the system on a different server. Usually, the work is distributed by a load balancer.

Reasoning on the x-axis is typically easy and the implementation can be fast, but the data sets have to be replicated in their entirety which increases operational costs.

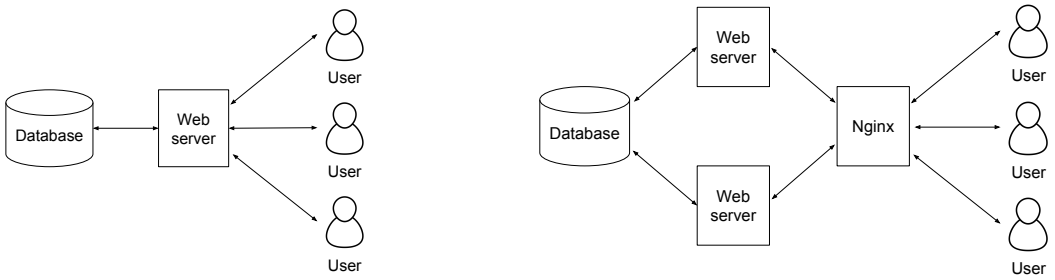

Figure 13: Web servers without (a) and with (b) load balancer.

3.4.1. An example: Web server replication

In order to better understand the concept, we bring an example of a common architecture which scales on this axis. Let's consider a small e-commerce business that runs its website on a single server on top of a single machine. This scenario is depicted in Figure 13a. This website is becoming increasingly popular and suddenly it has to face the explosive growth of HTTP requests. If this trend continues, the website cannot scale-up for a long time.

After a technical advice, the owner of the business decides to take the Web server codebase and deploy an identical copy of it. To make the website working on two different servers, she decides to use a load balancer and in particular `nginx`[32], an HTTP and reverse proxy server, mail proxy server and a generic TCP/UDP proxy server. Among the HTTP server features, it serves as load balancer. The load balancing methods supported by `nginx` are:

- round-robin, the requests are distributed in round-robin
- least-connected, the next request is assigned to the server with the least number of active connections
- ip-hash, the request is assigned to a server based on the client's IP address

The two Web servers now work in parallel and access the same database and, if they are *stateless*, the owner can choose any of the load balancing methods offered by the load balancer. The statelessness is an important property in this scenario, it avoids dependency between requests, that is the server can process a request without needing to access the information of another one. In the example, if the servers would not have been stateless (i.e. stateful), one request arrived at one of the two servers could be the ones on which another request being processing on the other server depends on, hence without permitting to successfully fulfill the latter.

[32] `http://nginx.org/`

3.4.2. State

Let's clarify the concept of *state* in order to better understand the importance of it for scaling on the x-axis. We said that, in other words, an application that uses state chooses the next action to be performed evaluating the current execution condition [6]. This definition holds for the protocols as well. A common example of stateless protocol is HTTP, since the receiver does not need to know anything about a previous request having all the information needed to fulfill the current one. On the contrary, an example of stateful application is a possible implementation of user session (which can be done also with a stateless approach), in which a user is authorized to request some resources only after an authentication request. In this setting, the result of the authentication could be stored in the server making it stateful, i.e. some requests are dependent on the authentication request. Referring at Figure 13b, in the stateful implementation, the user sessions could be stored at Web server level. In our example, the only load balancing method which supports session persistence is *ip-hash* thanks to its ability to always redirect requests from a client to the same server except when this server is unavailable or when the client's IP address changes. Indeed, with *round-robin* and *least-connected* each request is potentially distributed to a different server making it necessary to write stateless servers.

It should be clear now the role of the *state* scaling on the x-axis. Once we scale by cloning, we duplicate the data along with the service. If the state changes, these changes should be reflected on the server which accesses that same part of the state. This can be not a drawback at first, but it may become an issue increasing the data size.

3.4.3. Ethereum current state and proposals

In Section 2.4.1, we introduced the Ethereum consensus algorithm and the PoW algorithm which is run by the miners in order to create the next valid block. The more miners join the network (or improve the existents), thus incrementing the global hash rate, the more difficult a state transition become. In section 3.7.2, we measure the maximal transaction throughput in a private blockchain[33]. Our tests show that incrementing the number of miners in the network does not increment the transaction throughput, once the difficulty stabilizes. We notice that the results between different runs may vary probably as a consequence of the short running time of our tests (10 minutes), which limit the chance to reach a stable point. The turbulences in performance can be due to a variety of factors like network overhead, "unluckiness" of the miners which affects the mining difficulty, or perturbation in the performance of the nodes. Another significant test we present is described in section 3.7.2. In this test we set a gas limit high enough to not restrict the number of transactions which can be included in a block. The results show a little increment in the transaction throughput with regards to the previous test, but again the number of miners does not affect the overall performance. Hence, duplicating the miners in the network does not increment the transaction throughput making the scaling on the x-axis absent in the current implementation. When we are talking about

[33]See Section 3.7 for a detailed tests description.

duplicating, we are not assuming the nodes are running the same implementation. Indeed, there are multiple implementations (e.g., `geth`, `parity`[34], etc.), that adhere to the same protocols, therefore they are functionally equivalent.

In Section 3.2, we already pointed out that Ethereum is stateful, thus making it difficult to scale on the x-axis. Moreover, the implementation of the consensus layer requires that everyone validates each block propagated in the network executing all the transactions and the EVM computations which is in contrast with the concept of load distribution of the x-axis. Thus, the system is limited by the performance of the single computer duplicated in the network. Because of these issues, it is very difficult to individuate measures to scale on this axis without loosing other fundamental characteristics, like decentralization and trustlessness. Without these latter features, we would have a traditional system with a central authority on which we could implement some well know scaling solutions such as load balance among the different servers or simply scale up.

3.5. Y-Axis: Functional Decomposition

The y-axis scaling focuses on the separation based on the responsibility for an action which can be determined by the type of data or the type of work performed for a transaction. While scaling on the x-axis could means "everyone does everything", scaling on the y-axis specializes the workers splitting the whole service in small services and assigning each service to a single worker. If we combine the x-axis with the y-axis, we could have cluster of specialized workers such that each cluster covers a different service, but workers in the same cluster do the same job.

With regards to the data, differently from what happens in the x-axis, each service should have its own non-shared data, thus segmenting it based on what each service needs to have access to. This allows to size and optimize the resources based on the transactions demand for each service, and ultimately to reduce the operational costs.

As done before for the x-axis, we provide an example of an architectural approach which corresponds on the y-axis.

3.5.1. An example: Microservice architecture

Recalling the e-commerce example presented above, instead of duplicating the monolithic application and distribute the load equally between the clones, we can identify logical components, that is different functional areas of the application, such as account service, purchase service, product service.

The diagram in Figure 14 summarises the microservice architecture resulting from such division. The *user* entity condenses different client concepts (e.g. browser or mobile applications, bots, humans, or others), while the *gateway* represents what guides the requests to the right service. In the case of a browser application, the gateway could also exists only in it without needing a dedicated back end component, or it could not exists at all, for example advertising the location of each service and delegating to the

[34] `https://github.com/paritytech/parity`

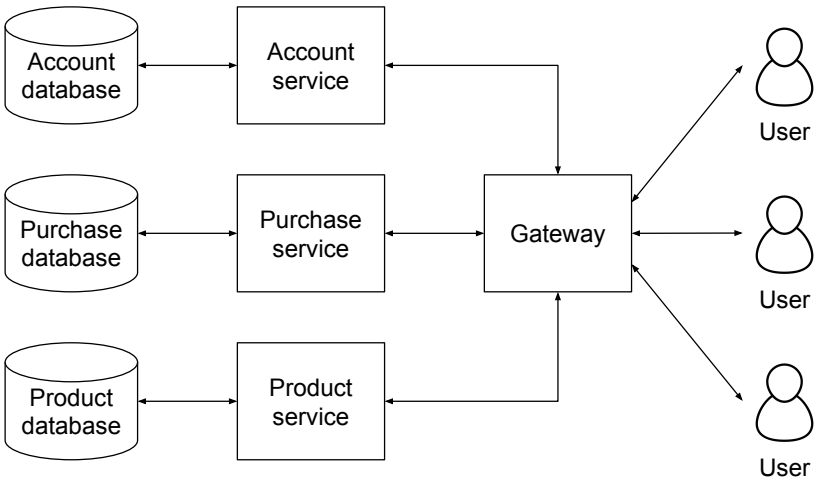

Figure 14: Example of a microservice architecture.

users the responsibility to choose the right service. Each service has its own non-shared database in order to be decoupled from other services, which represents a good solution in the case of perfect isolated services. More often, there could be some logical relations between data stored in different services making necessary to implement a mechanism to maintain data consistency. Such database separation opens the possibility to the services to choose the database type which fits better for the service they are providing. For example, the purchase service could use a SQL database taking advantage of the different isolation levels to concurrently and consistently handle the payments, while the product service could use a NoSQL database to tackle the different properties of the products. The services could be organized around the business capabilities (or "something that a business does in order to generate value" [31]), or decomposed by responsibility for particular actions or for all the operations on entities of a given type, but ideally each service should have only a small set of responsibilities.

The main drawbacks of the Microservice Architecture Pattern [32] are the increased complexity in development and deployment. On the other hand, it has a number of advantages, among which:

- **decoupling**. The functional decomposition of services decouples the components of the systems reflecting this behavior also in the database separation
- **fault isolation and design for failure**. Since the services implement different functionality of the system, they can be run on different processes leading to two consequences. The first one, as stated in [33], is that the application has to be designed thinking about a possible failure of a supplier, thus the client has to tolerate the failure of services responding as gracefully as possible. The second one is improved fault isolation, indeed, if a service has an unintended behavior, it less likely to affect the other components of the system

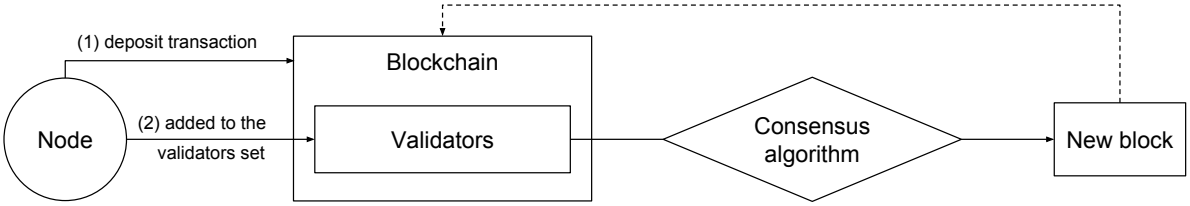

Figure 15: Overview of a block creation in the Proof of Stake.

- **scalability**. The services can be run on different more customized machines which better fit the resources requirements, hence making the vertical scaling more efficient wasting less resources. Moreover, if combined with the x-axis, it is possible to have a more fine-grained control on the horizontal scaling.

3.5.2. Ethereum current state and proposals

Currently Ethereum does not scale on this axis. A separation of responsibilities based on the type of work can be identified looking at the node type. A miner is responsible to collect pending transactions and build a valid block, while every node is responsible to verify the blocks generated by the miners, but, as we already discussed in Section 3.4.3, the transactions throughput does not follow the trend of the number of miners.

Among the proposals coming from the Ethereum community to improve the scalability, no one takes evidently this direction. The different node roles, although implementing different functions, do not define a functional decomposition since they operate on the global shared data. This is the case of Proof of Stake as well, whose we briefly introduce in Section 3.5.3, where we can identify different node functions as we do in PoW.

3.5.3. Proof of Stake

Proof of Stake (PoS) is a class of algorithms through which a cryptocurrency blockchain network achieves distributed consensus. As we have seen in Section 2.4.1, in PoW the truth is determined by heavy computation done by the miners. In PoS, there are validators instead of miners and the consensus depends on the *stake* confirmed by each validator, which consists in an economic deposit in the network's cryptocurrency (ether in this case).

Figure 15 shows a creation of a new block at high level. If a node owns an amount of the blockchain's base cryptocurrency, it can become a validator sending a deposit transaction which locks a given value. Once the transaction has successfully executed, the node becomes one of the validators. The consensus algorithm determines how a validator is chosen in the set of validators to *propose* the next block, and then how the validators agree on which block is canonical. Different type of PoS can be obtained by varying the consensus algorithm and how the rewards are assigned.

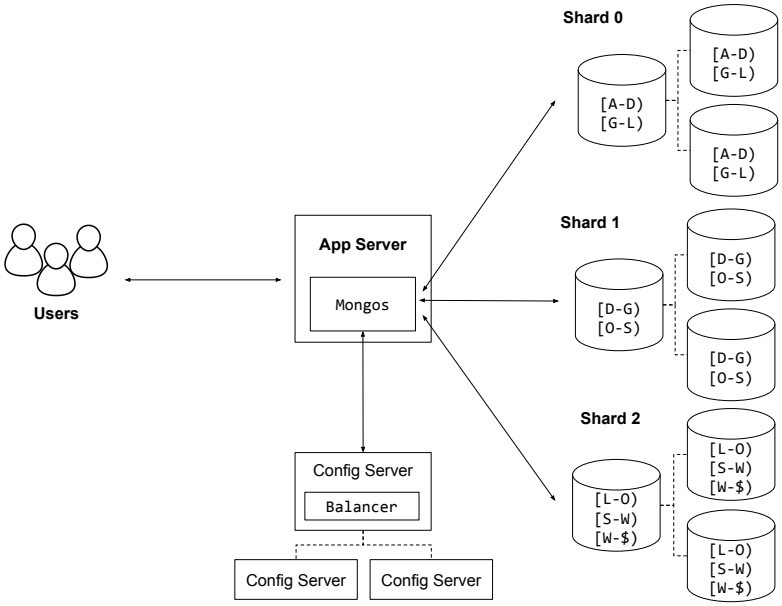

Figure 16: A typical architecture of an application server with a sharded MongoDB database.

3.6. Z-Axis: Horizontal Data Partitioning

The direction taken by scaling on the y-axis is to segment based on the service, i.e. based on *dissimilar* things. Scaling on the z-axis means segment on *similar* things, thus making the segmentation biased by the data or the actions that are unique to the sender or the receiver of the request. In particular, to enhance significantly the system a z-axis split should partition both transactions and the data necessary to perform the transactions. An example of such scaling could be, in a client-server architecture, the geographic distribution of the servers based on the clients requests, such that a request performed in Europe is undertaken by a server located in Europe instead of one located on the other side of the globe. One similar example of successful z-axis split is sharding.

3.6.1. An example: Sharding

Sharding is commonly used in distributed databases to scale *horizontally* by splitting the data of a single database in several servers. This way, it is possible to augment the transaction throughput by adding more machines, instead of requiring a more powerful machine, as in the case of *vertical* scaling. Indeed, when the number of requests grows, the resources of a single server could be insufficient to grant acceptable response times. The drawback of using multiple servers is the management overhead.

One emblematic example of implementation of (auto-)sharding is MongoDB[35] [34]. This kind of open-source NoSQL database stores documents without a fixed schema.

[35] `https://www.mongodb.com/`

MongoDB gives the possibility to split a collection of documents into different *shards* according to a selected *shard key* and a *sharding strategy*. Each shard consists of one or more replicated servers[36] and it is accountable to store the documents in a partition of the key space. These partitions are composed of one or more chunks. These are minimal piece of data (default 64MB) that contain the documents, whose shard key values are included within a contiguous range of the key space. To have a unique correspondence between documents and chunk, and transitively of documents and shard, no overlapping chunks are permitted. If after inserting new documents in a given chunk, it exceeds a configured size, the chunk is split. Furthermore, if a shard contains too many chunks, some of them may be migrated to other shards.

Since the users and the applications should be able to access the data *transparently*, that is without knowing where the documents are really stored, an entity called `mongos` is introduced. Essentially, it is a broker between the application and the database, that forwards the requests to the right shard(s), collects the responses and returns them to the requester. Usually each application has its own `mongos` instance, but other configurations are also possible [34].

In order to know where the data are stored, the `config servers` are used. They contain metadata and the configuration settings for the cluster, such as the list of chunks that are stored in each shard and the ranges covered by the chunks. Each time a chunk is split or migrated these settings should be updated.

To automatically balance the load between the different shards, a background process in the principal `config server` called *sharded cluster balancer* is employed[37]. It monitors constantly the number of chunks on each shard and whenever one shard contains more chunks than a given threshold, it tries to migrate chunks to balance the amount of chunks in each shards. In addition, it attempts to minimize the amount of data that should be migrated to reduce the performance impact due to the bandwidth and workload consumption caused by migration. For example, one shard cannot be involved in more than one migration at the same time. Moreover, if additional servers are available, the balancer may also create new shards or it may simply rearrange the partitions.

These concepts are summarized in Figure 16, in which the replicas are bounded with dashed lines whereas the solid lines represents which component communicates with which one. The chunks are shown with the ranges of the shard key space they cover. Whenever the application server must query the database to collect information, it demands the mongos instance to do so. If the query contains the shard-key, the broker can forward the request to the right shard(s). When this information is not available, `mongos` should broadcast the request to all the shards.

We clarify this by means of an example. Let's take into consideration Figure 16 and suppose that the sharded database contains the collections of users of the system of the application server, which needs the data related to a user to perform authentication. Furthermore, suppose that the shard key is the username and that the chunks are split

[36] In MongoDB jargon it is known as *replica set*.

[37] In version prior to 3.4 (the current version is 4.0) the role balancer was played by the different `mongos` instances on turn [35]

according to the initial letter of the username. Now, if the application server needs the data belonging to a certain user, let's say `JohnDoe`, it has to create a query and send it to the `mongos` instance. This searches in the `config server` which chunk *should contain* the searched key and discover that shard-0 contains the chunk ranging from G to L and therefore forwards the request to shard-0 and obtains the awaited response. `mongos` elaborates it and send the reply to the application server, that can now use the information. However, if the query does not contain the shard key, the broker has no way to know in which shard the information is stored and has to broadcast the request. It is worth to notice that the choice of the shard key is fundamental to grant a certain level of performance. If the server is sharded according to, let's say, the age of the users, the search by username would require a broadcast, because in the `config server` there would be no information to forward the request to the shard containing the required document.

3.6.2. Ethereum current state and proposals

We argue that currently Ethereum is not developed in the z-axis, because each node of the network should have information about the whole blockchain and the status of all accounts in the network (Section 2.3.1) to process the transactions. For this reason Ethereum cannot be more efficient than a single machine, as already pointed out by Vitalik Buterin in the muave paper [36].

Furthermore, a z-axis split similar in nature with the one of MongoDB, i.e. by letting different nodes store different part of the state would surely diminish the amount of data to store on each node, but (A) it would require a lot of data to be sent across the network to perform the computations, and (B) it would not augment the transaction throughput, but rather diminish it. To justify point (A) we consider that many computations require the ability to access a fair amount of addresses and their storage as the following example clarifies. Let's consider the process of a transaction that requires some computations on the EVM. To complete this action we need several information, such as the balance of the sender of the transaction and the code of the contract invoked. In turn, the contract invoked may call other contracts and so on. In addition, these may modify the balance of other accounts as a side effect, e.g. through the `SELFDESTRUCT`[38] opcode. Thus, to process a transaction we need the account states of the sender and the called contracts and all account state that are affected by the computations. The point (A) implies points (B), indeed the overhead due to the communication would surely slow down the transaction processing action.

With these considerations in mind and by recalling that to be significant a z-axis split should partition both the transactions and the data necessary to perform the transactions, more clever approaches were proposed.

To tackle a z-axis split, we can identify different proposals, among which Plasma and Sharding. The Plasma proposal is categorized as an *off-chain* solution because it executes some transactions outside the main chain. At the opposite, in which the sharding belongs,

[38] This opcode causes the deletion of the executing contract from the world state and sends the balance of the contract to a selected account [1].

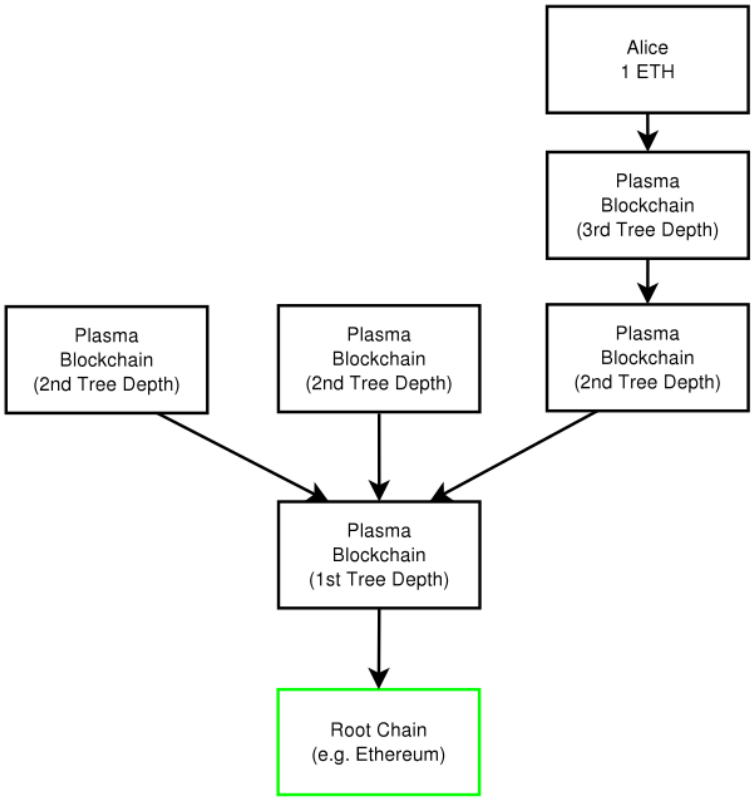

Figure 17: Plasma hierarchical tree structure representation.
Source: `https://plasma.io/plasma.pdf`

the *on-chain* solutions execute every transaction on the main chain. Typically, in order to apply an off-chain solution, one or more smart contracts are sufficient (for example, Vitalik proposed a specification for a minimal implementation called Minimum Viable Plasma[39]), while an on-chain solution requires an hard-fork, because it requires a radical change in the Ethereum protocol.

3.6.3. Plasma

The idea of Plasma is to create a tree hierarchy of blockchains. Each chain refers to a parent chain, except the root (i.e. the main chain, Ethereum in our case). Plasma consists of a series of smart contracts which allows for many blockchains within a root blockchain [37] as represented in Figure 17. The Plasma blockchains co-exist with their own business logic and all the computations are enforced at root level only in the event of proof of fraud, and only the block header hashes are submitted. During non-faulty states, only merkleized commitments are periodically broadcast to the root blockchain. The efficiency mainly comes from this last feature since multiple state updates can be condensed in a single state update in the root chain. This system design implies that

[39] `https://ethresear.ch/t/minimal-viable-plasma/426`

most of the computation can be done off-chain (i.e. outside the root chain) and the state is enforced on-chain (i.e. in the root chain). Significant scalability for the users is achieved through chain split: when a Plasma blockchain grows too large, it can be split in child chains allowing the users to observe only Plasma blockchains in which their funds resides.

3.6.4. Sharding in Ethereum

The Ethereum foundation [36, 38] and the scientific community [29] have proposed sharding combined with Proof-of-Stake (*shasper*[40]) as an effective measure to dramatically increase the transaction throughput. Although the details about this proposal are constantly updating[41] and there is no reference implementation[42], we will report here the basic ideas [36, 38] about sharding and explain why this approach could improve the scalability of the system.

The Ethereum sharding proposal consists in splitting the world state and transaction history into different partitions called shards. Each shard is a distinct universe, i.e. is itself a PoS chain, in which the transactions affect only the accounts in the same shard. The transactions affecting one shard are collected in so-called *collations* by members of the network known as *proposers*. Collations are the analogous of blocks at the shard level and like blocks (Figure 4) are chained and contain several fields, among which the list of transactions and the address of the proposer. Once per *epoch*, a block in the main PoS chain (main block) is created, where *cross-links* (e.g. the hash of the collations) to the accepted shard collations are inserted. The collations are accepted or refused by a committee of *attesters*. To become an attester or a proposer the members of the network should deposit an amount of ether. After this operation, they can be randomly assigned to different shards. This is done in order to grant a certain level of decentralization and security.

This sharding proposal allows to process the transactions of different shards in parallel and therefore significantly increase the transaction throughput. Another major benefit of this approach is that the nodes in one shard should verify only the transactions in their shard rather than verifying all the transactions.

This simplified description of sharding do not take into consideration the communication between different shards, but obviously it is a desirable characteristics. Thus, cross-sharding communication mechanisms were proposed [38].

Moreover, one important feature that is desirable and planned is the transparency of sharding to smart contract developers [38].

[40] This name is obtained by the contraction of the words *sharding* and *Casper*, the Ethereum's Proof of Stake proposal [39].

[41] `https://notes.ethereum.org/SCIg8AH5SA-O4C1G1LYZHQ`

[42] The Prysmatic Labs company is modifying the `geth` implementation to implement both PoS and sharding (`https://github.com/prysmaticlabs/prysm`), but it is far from being production ready.

3.7. Tests

To measure the scalability of Ethereum, we executed some tests to study the maximal throughput and the size of the blockchain with different configurations. To study the scalability of a permission-less blockchain system such as Ethereum, one should either rely on simulation [21] or run tests using thousand of nodes [21, 40]. We do not have neither a simulation of an Ethereum system nor so many resources. Therefore, we took inspiration from Blockbench [41], which compares the performance and scalability of Hyperledger[43] and Ethereum in a *private* (*permissioned*) scenario, that is, when we consider a limited number of authenticated nodes.

We tried to use the public available Blockbench repository[44] but we did not manage to configure it due to a lot of hard-coded configuration variables and the lack of a well-written documentation. Thus, we wrote our own test and benchmark system[45].

3.7.1. Test Configuration

To keep the configuration easy, we opted for a classic master-slave logic. The master, i.e. the initiator and coordinator of the tests, uses the ssh protocol to run commands on the remote machines. Similarly to [41], we distinguish between the *miner* and *client* roles. The nodes of the former type are accountable to generate new blocks while the nodes of the latter type create and propagate transactions, and both verify the blocks[46]. We can assign multiple roles to a single machine. In this case we run *one distinct* `geth` *instance* for each different role. The coordinator copies the right genesis file in the test machines.

In each run of test, we distinguish two main phases: 1. the setup and 2. the test itself.

Setup phase The setup consists in iterating on the test machines twice:

1. the required ethash data structures are generated
2. the genesis file is used to create the genesis block and initialize the ethereum World State.

Ethash data structures During the setup phase, the miners generate the DAG and the cache for the first two epochs, while the clients generate only the caches because they are required only to validate blocks. We generate the DAGs for the first two epochs, because ethash uses double buffer of DAGs to grant a smooth switch between epochs [18].

[43] `https://www.hyperledger.org/`
[44] `https://github.com/ooibc88/blockbench`
[45] `https://github.com/gfornari/ethereum-test/tree/benchmark`
[46] To reduce the number of test variables we consider only full nodes. For a list of node types we refer to Appendix A.

Genesis file The genesis file contains useful information to create (deterministically) the genesis block and the initial state. We report an extract of the genesis file we used in Listing 1. It is simply a JSON file, which specifies several parameters. Most of them directly specify the attributes of the block number 0, which are shown in Figure 4. Here, we describe only the fields that are not directly a parameter specification:

- `config`: it describes the network id, and the number and hash of blocks that marks the entry into force of the Ethereum Improvement Proposals (EIP), which indicates incompatible changes in the protocol or simply a new version of Ethereum
- `alloc`: it specifies an initial allocation of Ether for the accounts[47] we use in the tests.

```
{
    "config": {
        "chainId": 11691524842890,
        "homesteadBlock": 0,
        "eip155Block": 0,
        "eip158Block": 0
    },
    "coinbase": "0x00..00",
    "difficulty": "159268",
    "extraData": "",
    "gasLimit": "0x2fefd8",
    "nonce": "0x37a2f64534",
    "mixhash": "0x00...00",
    "parentHash": "0x00..00",
    "timestamp": "1528621597",
    "alloc": {
        "0xe505c82291141cea6c2d371e522caf2197740a78": {
            "balance": "100000000000000000000"
        },
        ...
        "0x1cd50bd930bd9d2474c671173e4ad283c0ac204f": {
            "balance": "100000000000000000000"
        }
    }
  }
```

Listing 1: An extract of the genesis file used in the tests.

Obviously, each node of the network should be initialized with the same genesis file, otherwise the hash of the genesis block differs and the peers cannot establish a connection with the Ethereum Wire Protocol as described in Section 2.2.3. Thus, the genesis file for the main network and the official Ethereum test networks are hard-coded. To create a new private Ethereum network it is sufficient to use a new genesis file in which some

[47] This possibility has been exploited for the so-called Initial Coin Offering (ICO) used by the Ethereum Foundation to obtain fiat currency to finance the project.

parameters are changed. Therefore, in our genesis file we use an arbitrary network id and nonce, so that packets of different networks are simply dropped.

Apart from the arbitrary values, that is the id and the nonce, and the data that perhaps are required by the system which do not have influence on the transaction throughput, the values of some parameters require some justification.

The **timestamp** of the genesis file, that corresponds to the one of the genesis block, influences the difficulty of the first blocks, and transitively of all the blocks. Since, for time constraints, we want to run each test for few minutes, we want to avoid sudden decreases in the difficulty due to a too old timestamp. Therefore, to prevent these unwanted changes in the difficulty value, during the second loop of the setup phase, the initiator reads its timestamp and gives it as parameter to the peers[48].

As already described in Section 2.4.1, the **difficulty** is an adaptive parameter that determines how much effort should be invested in the creation of a new block. In our tests we used the same hardware and same operating system in all nodes and for all miners (we deliberately used only one thread for mining). Therefore, to find a suitable starting value for the difficulty for the different configurations, we ran a simulation with one, two, four and eight miners for 24 hours. Figure 18 shows how the difficulty changed with the different number of miners. We can notice that in our homogeneous system, the final values of difficulties have a quasi linear dependency with the number of miners. For the test we took as initial value the median of the last 100 blocks. These values are represented in Table 2. We reported also the coefficient of variation to show that the values of the last 100 blocks are pretty stable.

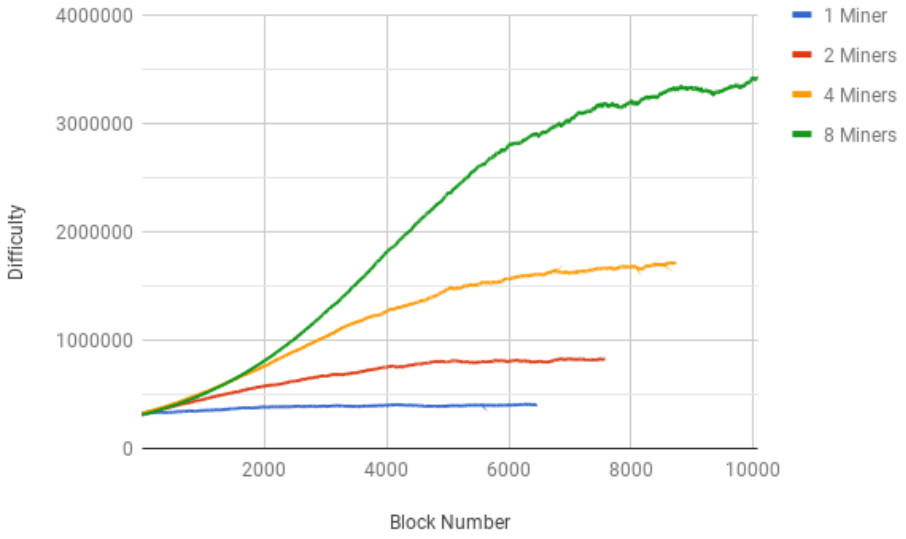

Figure 18: The growth of the difficulty in the 24 hour run.

[48]Before starting the test all the peers should be configured, therefore using the timestamp of the coordinator does not assume fine-grained coordinated clocks.

Number of Miners	Difficulty	Coefficient of variation
1	404559	0.0073%
2	821994	0.2330%
4	1711150	0.1458%
8	3409299	0.1742%

Table 2: The median of the last 100 blocks, value used in the tests as start difficulty, and the coefficient of variations.

The **gas limit** in the genesis block determines the gas limit of the first block. The gas limit of the subsequent blocks can be determined freely by the miners but must be contained in a range obtained by summing and subtracting to the gas limit a portion of itself [1]. Thus, in a relative brief simulation the initial value is fundamental.

Test Phase After the setup phase, the real test begins by starting the execution of the `geth` instances on the different machines. The clients emit transactions that cause simple transfers of values that do not require the intervention of the EVM, with a predefined release time (50 ms), while the miners starts collecting transactions and creating blocks. The tests are stopped after a configurable time (10 minutes). For each configuration the tests are repeated a configurable number of times (5) because the probabilistic nature of the Proof-of-Work algorithm and the interaction of many systems do not guarantee deterministic results.

Concrete Parameters We measure the throughput with different number of miners (1, 2, 4 and 8), fixing the number of clients at 16 with two different gas limits. The miner machine executes only one `geth` instance in miner mode, while the client machines execute two instances in client mode. For our tests we used the Scaleway platform[49]. We rent 17 (one master and 16 slaves) START1-S servers, which are provided with 2 X86 64 bit Cores and 2 GB of RAM, a 50 GB SSD and an internal bandwidth of 1 Gbit/s, so that the network would not be a bottleneck. Each server runs Ubuntu 16.04 LTS with `geth` version 1.8.11-stable. During the tests, the master acts also as bootstrap node. We described the role of bootstrap nodes in Section 2.1.

The two tests we conducted differs only for the gas limit in the genesis block, we wanted to confirm that this value influence the number of transaction processed as described in Section 3.2.

3.7.2. Results

In this section we report interesting results obtained from the two tests we conducted.

[49] `https://www.scaleway.com/`

Maximal throughput - Low Gas Limit For this test we used a gas limit of 3141592, which corresponds to approximately $150 \cdot 21000$. The results are reported in Table 3. We notice that in this case we could not exceed the 10 transactions per second and that the number of transaction in each block do not exceed 135, although there would be enough space to add other transactions.

	1 miner	2 miners	4 miners	8 miners
Avg throughput	5.37	7.17	6.73	6.65
Avg blocks	24.00	37.80	30.02	30.00
Avg txs per block	133.77	114.69	133.61	133.15
Max throughput	7.18	7.76	8.17	8.67
Min throughput	3.68	6.83	5.91	4.91

Table 3: Average throughput and number of mined blocks on 5 runs.

Maximal throughput - High gas limit For this test we use a very large gas limit 10500000 which corresponds to $500 \cdot 21000$. To give a term of reference, we can consider that currently (August 2018) the gas limit for the main Ethereum network is approximately 8 million, that corresponds roughly to $381 \cdot 21000$. The results are summarized in Table 4. In this case with two configuration we could exceed the 15 transactions per second.

	1 miner	2 miners	4 miners	8 miners
Avg throughput	8.17	11.55	9.02	11.07
Avg blocks	33.60	27.20	37.20	29.80
Avg txs per block	145.25	262.98	157.15	226.32
Max throughput	12.55	17.50	17.77	12.525
Min throughput	6.82	6.83	6.83	7.14

Table 4: Average throughput and number of mined blocks on 5 runs with high gas limit.

4. Conclusions

The Blockchain technology provides a way to find a total order of transactions in a distributed system without relying on a trusted third party. This permits to have an exact copy of the state in each node of the network, because each node starts from the same state and changes the state according to the transactions, whose execution is deterministic. The main advantage of this technology with respect to traditional transition systems is its decentralized nature.

In this report we took into consideration Ethereum, a representative of the permissionless blockchain technology. This system is particularly interesting because it supports a general-purpose execution environment, the EVM.

In the first part of this work we proposed a decomposition of the Ethereum's architecture in logical layers. The five stacked layers abstract from the implementation details and offer a conceptual organization useful to compare different blockchain proposals, although permitting flexibility as the case of the EVM which is a cross layer. Moreover, this decomposition allows a component-oriented development which helps reasoning on the system.

The focus of the second part is the analysis of the scalability of Ethereum. To do so, we analyzed the existing literature (Section 3.2) and confirmed empirically that the current version of Ethereum reaches a maximal transaction throughput of approximately 15 transactions per second, even if the transactions do not require computations. This value is not influenced by the number of miners, but rather by the block size and the block interval. Furthermore, we analyzed the motivation of this reduced transaction throughput with the aid of the cube of scalability [6]. We argue that the current version of Ethereum is not developed in any direction of the cube by analyzing the current specification. Thereafter, we categorize the scalability improvement proposals based on the axes they will affect. From this analysis, it is clear that the most efforts of the community are concentrated on the z-axis of the cube Section 3.6 with proposals like Plasma and sharding. Sharding is a scaling strategy already seen in the database systems, that is, systems which have to manage persistent data. Maybe, this common duty between Ethereum and the database systems leads the z-axis of the Scale Cube.

A. Node types

Currently, in Ethereum there are mainly three types of nodes: full nodes, archive nodes and light nodes. Hereafter, we describe each type of node and explain how to start nodes of this type with `geth` v1.8.11.

Full nodes The full nodes are the nodes we describe throughout our work. They store the whole blockchain (comprising of block headers and bodies), they have full copy of the most recent states, and they verify and process every transaction. A full node can be a miner.

When running a full node the user has the possibility to decide whether to use:

- the normal synchronization mechanism, i.e. the protocol version 62 section 2.2.3:

```
geth --syncmode full [other-options]
```

- the fast synchronization, i.e. the protocol version 63 section 2.2.3

```
geth --syncmode fast [other-options]
```

The default option is "fast", because it require significantly less time, as explained in Section 2.2.

Archive nodes Archive nodes are full nodes that store also the state tree for each block, hence they are the most storage-bound nodes. They are used by block explorers, like Etherscan[50] or enterprises that needs to have at their disposal historical information.

To run an archive node, it is sufficient to overwrite the garbage collection mode (gcmode) for the state, so that all the intermediate states are stored:

```
geth --syncmode full --gcmode archive [other-options]
```

Light nodes Light nodes store only the headers of the blocks of the blockchain. They do neither store the blocks' bodies nor the state. The idea behind light nodes, is to use the other peers in the network as a distributed hash table (DHT). In essence, they know the hash of the information and request the desired information from the peers they know with an on-demand logic.

For example, a light node can retrieve the state of an account at a desired block by recursively requesting the content of the World State, starting from its hash which is contained in the block's header (Figure 4). We refer to [42] for further details about how and what type of other information a light node can retrieve.

This type of clients assume that there are full-nodes in the network supporting the Light Ethereum Subprotocol (LES) [43] which have message types to retrieve data on-demand.

To run a light node it is sufficient to overwrite the default synchronization method:

```
geth --syncmode light [other-options]
```

[50] `https://etherscan.io/`

B. Solidity

Solidity [20] is the most popular high-level language used in the Ethereum ecosystem. It is a high-level statically typed language that is compiled to EVM bytecode. It takes inspiration from various languages, such as JavaScript, Python and C++.

In addition to the usual mathematical and logical operations, Solidity has several peculiar variables and functions that resides on the global scope and allow the programs to access the data shown in Figure 10. One of this specially crafted global variables is the `msg` variable, that allows the programs to access the message call data, such as the code invoker's address (`msg.sender`) and the amount of money sent along with the invocation (`msg.value`).

In order to provide an intuitive insight about Solidity, we provide a little example in Listing 2, that shows some key features of Solidity.

As it is easy to see, the contracts are the building blocks of the language and resemble objects. Indeed, in Solidity one contract can also inherit from other contracts. The fields of the contract are stored in the contract's storage. In the example there are two persistent variables `x` and `owner`. While the former is marked as public the latter is marked as private. Thus, the compiler creates a getter `x()` for variable `x` but not for variable `owner`. It should be noticed that the `owner` variable can still be read from outside the EVM[51], but it cannot be read from other contracts that run on the EVM.

After the definition of the field variables we can see the `constructor` method. It is run only once upon creation and returns the code of the contract as we described in Section 2.4.3. If it is not defined, an empty constructor is assumed.

The *events* are a high-level interface for the EVM's logging facilities. Events are emitted by smart contracts to notify external applications, that can listen to them to perform some application-specific operations. Logs are visible from the outside because they are stored in the blockchain as part of the receipt of the transaction which emitted them.

After the definition of the events, the *modifier* `onlyOwner` is defined. A modifier is a piece of code that is executed before or after the call of a function decorated with it. They can be used to implement final state machines [20] and pre and post-conditions checks for functions.

The *functions* are an abstraction of the language. They can be invoked either `internally`, that is inside the same call stack activation record with a `JUMP` or `JUMPI` instruction or `externally`, that is by creating a new message call. In the former case the machine state is the same of the invoker, while in the latter the code is executed in a new machine state, that is, by creating a new activation record on the call stack, as discussed in Section 2.5.1. The first 4 bytes of the hash of the signature of external function (function name and parameter types) is used as unique identifier (*selector*) of a function. When an actor wants to invoke a function externally it should provide the function's selector in the message data and follow the ABI format [20]. Essentially, the Solidity

[51] It is part of the contract's storage, which in turn is part of the world state that is replicated in each node.

```
pragma solidity ^0.4.24;

contract Example {
    uint8 public x; // Compiler creates getter
    address private owner; // Only this contract can see variable owner

    constructor() public {
        x = 0;
        owner = msg.sender; // The creator of the contract is the owner
    }

    event XHasChanged(uint8 oldValue, uint8 newValue);
    event FallBackCalled();

    modifier onlyOwner() {
        // Pre-conditions
        require(msg.sender == owner,
                "only␣the␣owner␣can␣modify␣x"
        );
        _; //Placeholder for the body of the function
        // Put additional post-conditions here
    }

    function modifyX(uint8 newValue)
    public onlyOwner returns (uint8 oldValue) {
        oldValue = x;
        x = newValue;
        emit XHasChanged(oldValue, newValue);
    }

    function () public{
        emit FallBackCalled();
    }


}
```

Listing 2: An example of a contract written in Solidity

compiler puts at the start of the compiled code a switch-statement that checks if the first four bytes of the message call data match a known function's selector. If it is the case, the program jumps to the definition of the matched function otherwise a so-called fallback function is invoked. The programmer may specify the fallback function by writing an unnamed function that does not take parameters. Since the signature depends on the number and the type of arguments, function overloading is easily implemented by using the new obtained selector.

Moreover, we distinguish also between `public` and `private` functions. The formers can be called either externally or internally, while the latters can be called only internally and are not visible to the contracts that are below in the inheritance hierarchy.

In addition to the visibility, the function header may contain a state mutability flag: we distinguish between `pure`, `view` and `payable`. The first indicates that the function will not access the state, the second promises to access the state in a read-only fashion, while the latter indicates that the function is capable of receiving money. `pure` and `view` functions can be invoked also from outside the EVM, i.e. without a transaction and without paying fees. After all, these functions read only the local state without the intervention of the other peers. Obviously, if these types of functions are called with a transaction or during the execution of another contract, the normal gas consumption applies, because all the full nodes should execute the code.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2018.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[3] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," in *International Workshop on Open Problems in Network Security*. Springer, 2015, pp. 112–125.

[4] V. Buterin. (2018-04-12) A next-generation smart contract and decentralized application platform. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[5] A. M. Antonopoulos, *Mastering Bitcoin: Programming the Open Blockchain*. O'Reilly Media, Inc., 2017.

[6] M. L. Abbott and M. T. Fisher, *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Pearson Education, 2009.

[7] M. van Steen and A. Tanenbaum, *Distributed Systems*. CreateSpace Independent Publishing Platform, 2017.

[8] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[9] Ethereum Foundation. (2018-05-15) Node discovery protocol. [Online]. Available: https://github.com/ethereum/devp2p/blob/master/discv4.md

[10] Ethereum Foundation. (2018-05-05) Design rationale. [Online]. Available: https://github.com/ethereum/wiki/wiki/Design-Rationale

[11] Ethereum Foundation. (2018-04-13) Recursive length prefix official documentation. [Online]. Available: https://github.com/ethereum/wiki/wiki/RLP

[12] Ethereum Foundation. (2018-03-27) Rlpx: Cryptographic network & transport protocol. [Online]. Available: https://github.com/ethereum/devp2p/blob/master/rlpx.md

[13] Ethereum Foundation. (2015-11-18) ÐΞVp2p wire protocol. [Online]. Available: https://github.com/ethereum/wiki/wiki/%C3%90%CE%9EVp2p-Wire-Protocol

[14] Ethereum Foundation. (2018-04-12) Official ethereum wire protocol specification. [Online]. Available: https://github.com/ethereum/wiki/wiki/Ethereum-Wire-Protocol

[15] V. Buterin. (2015-11-15) Merkling in ethereum. [Online]. Available: https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/

[16] Ethereum Foundation. (2018-04-19) Patricia tree. [Online]. Available: https://github.com/ethereum/wiki/wiki/Patricia-Tree

[17] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol." in *NSDI*, 2016, pp. 45–59.

[18] Ethereum Foundation. (2018-05-31) Dagger hashimoto. [Online]. Available: https://github.com/ethereum/wiki/wiki/Dagger-Hashimoto

[19] Ethereum Foundation. (2018-08-22) Mining. [Online]. Available: https://github.com/ethereum/wiki/wiki/Mining

[20] Ethereum Foundation. (2018) Solidity 0.4.25 documentation. [Online]. Available: https://media.readthedocs.org/pdf/solidity/latest/solidity.pdf

[21] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 3–16.

[22] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.

[23] Ethereum Foundation. (2018-05-25) JSON RPC API. [Online]. Available: https://github.com/ethereum/wiki/wiki/JSON-RPC#json-rpc-api

[24] JSON-RPC Working Group and others. (2012) JSON-RPC 2.0 specification. [Online]. Available: http://www.jsonrpc.org/specification

[25] Ethereum Foundation. (2018-06-01) JavaScript API. [Online]. Available: https://github.com/ethereum/wiki/wiki/JavaScript-API

[26] Ethereum Foundation. (2017-12-21) Javascript console. [Online]. Available: https://github.com/ethereum/go-ethereum/wiki/JavaScript-Console

[27] Z. Zheng, S. Xie, H.-N. Dai, and H. Wang, "Blockchain challenges and opportunities: A survey," *Work Pap.–2016*, 2016.

[28] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba, "A taxonomy of blockchain-based systems for architecture design," in *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 243–252.

[29] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer *et al.*, "On scaling decentralized blockchains," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 106–125.

[30] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *International Conference on Financial Cryptography and Data Security*. Springer, 2015, pp. 507–527.

[31] C. Richardson. (2018-08-17) Pattern: Decompose by business capability. [Online]. Available: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html

[32] C. Richardson. (2018-08-17) Pattern: Microservice architecture. [Online]. Available: https://microservices.io/patterns/microservices.html

[33] J. Lewis and M. Fowler. (2018-08-17) Microservices, a definition of this new architectural term. [Online]. Available: https://martinfowler.com/articles/microservices.html

[34] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media, Inc., 2013.

[35] MongoDB Inc. (2018-08-01) MongoDB Manual. [Online]. Available: https://docs.mongodb.com/manual/

[36] V. Buterin, "Ethereum 2.0 mauve paper," in *Ethereum Developer Conference*, vol. 2, 2016. [Online]. Available: https://cdn.hackaday.io/files/10879465447136/Mauve%20Paper%20Vitalik.pdf

[37] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," *White paper*, 2017.

[38] Ethereum Foundation. (2018-08-03) Sharding faq. [Online]. Available: https://github.com/ethereum/wiki/wiki/Sharding-FAQs

[39] Ethereum Foundation. (2018-08-09) Cbc casper faq. [Online]. Available: https://github.com/ethereum/cbc-casper/wiki/FAQ

[40] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 51–68. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132757

[41] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1085–1100.

[42] Ethereum Foundation. (2018-08-23) Light client protocol. [Online]. Available: https://github.com/ethereum/wiki/wiki/Light-client-protocol

[43] Ethereum Foundation. (2017-10-16) Light ethereum subprotocol (les). [Online]. Available: https://github.com/zsfelfoldi/go-ethereum/wiki/Light-Ethereum-Subprotocol-%28LES%29