ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Using Blockchain-based Smart Contracts for Identity Management Services

Mémoire présenté en vue de l'obtention du diplôme
d'Ingénieur Civil en informatique à finalité spécialisée (MA-IRIF)

Nazar Filipchuk

Directeur
Professeur Stijn Vansummeren

Superviseur
Martin Ugarte

Service
CoDE-WIT

Année académique
2016 - 2017

Table of Contents

List of Figures

List of Tables

Abstract

Identity management services need to store and manipulate digital information to identify individuals. This information, in practice, is stored by central governments. Some third party services (e.g. banks) need to *verify* the information of their clients in order with comply to local regulations. The fact that, local authorities are privately storing identity information, implies that, if a malicious party gets access to the identity management service (or the authority gets corrupted), they would be able to arbitrarily impersonate individuals. Moreover, any information loss (because of errors or hacks) in the identity management implementation would result in the loss of certain identities.

These problems have traditionally been thought as *unavoidable*, since it is natural to think that the storage and management of identity needs to be performed by some entity, which is naturally a central authority. However, with the introduction of the Bitcoin protocol and the concept of *smart contracts*, a new and decentralized form of managing data and transactions has emerged.

In this work, we embarked on the task of studying the requirements of identity management services and understanding how they can be decentralized by using the aforementioned technologies. To this end, we started by studying the way in which identity is managed by distributed systems nowadays. Then, we compare this with systems based on the Ethereum protocol, a protocol for smart contracts inspired by the Bitcoin.

Using the Ethereum protocol, we have designed a smart contract that fulfills the requirements of identity management services. The permissions of the different actors are encoded directly in the smart contract, and therefore they are enforced in a decentralized manner, without the need of trusting central authority. We also present a prototype of this smart contract developed in Solidity, a contract-oriented programming language. Additionally, we provide a web application to dynamically interact with the smart contract.

We have demonstrated that the widespread techniques to deliver web applications can be easily adapted to be used with smart contracts. Furthermore, its execution is verified by thousands of peers, delivering consistent answers to every user. Therefore, the identity management services can be released from the risks, the pressure of which is heavier with every client.

Keywords: identity management, decentralization, Ethereum, smart contracts

Résumé

Les services de gestion d'identité doivent stocker et manipuler des informations numériques pour identifier des individus. Ces informations, en pratique, sont stockées par des gouvernements centraux. Quelques services tiers (par exemple les banques) ont besoin de *vérifier* les informations concernant leurs clients pour se conformer aux règlements locaux. Le fait que les autorités locales stockent, à titre privé, des informations sur l'identité implique que si un parti malveillant obtient l'accès au service de gestion d'identité (ou dans le cas où l'autorité est corrompue), il pourrait arbitrairement interpréter le rôle de l'individu. De plus, n'importe quelle perte de l'information (à cause des erreurs ou des piratages) dans la mise en œuvre de la gestion d'identité aboutirait à la perte de certaines identités.

Ces problèmes ont toujours été pensés comme *inévitables*, puisqu'il est naturel de penser que le stockage et la gestion d'identité doivent être exécutés par une certaine entité, qui est naturellement une autorité centrale. Cependant, avec l'introduction du protocole de Bitcoin et le concept des *contrats intelligents*, une nouvelle forme décentralisée pour la gestion des données est apparue.

Dans ce travail, nous nous sommes embarqués dans la tâche d'étudier les exigences des services de gestion d'identité et comprendre comment ils peuvent être décentralisés en utilisant les technologies mentionnées ci-dessus. À cette fin, nous avons commencé en étudiant la façon dont l'identité est gérée par des systèmes distribués de nos jours. Ensuite, nous les comparons avec des systèmes basés sur le protocole Ethereum, un protocole pour des contrats intelligents inspirés du Bitcoin.

En utilisant le protocole Ethereum, nous avons conçu un contrat intelligent qui accomplit les exigences des services de gestion d'identité. Les permissions des différents acteurs sont codées directement dans le contrat intelligent et donc elles sont respectées de façon décentralisée, sans le besoin d'avoir confiance en une autorité centrale. Nous présentons aussi un prototype de ce contrat intelligent développé utilisant Solidity, un language de programmation axé sur les contrats. De plus, nous fournissons une application Web pour interagir dynamiquement avec le contrat intelligent.

Nous avons démontré, que les techniques répandues pour livrer des applications Web, peuvent être facilement adaptées pour etre utilisées avec des contrats intelligents. En outre, son exécution est vérifiée par des milliers de pairs, livrant des réponses cohérentes à chaque utilisateur. Donc, les services de gestion d'identité peuvent être mis hors de risques, dont la pression augmentant avec le nombre de clients.

Mots-clé : gestion d'identité, decentralisation, Ethereum, contrats intelligents

Chapitre 1

Introduction

Our social interactions become digital with every consecutive day. Our identities are represented as numbers, stored on machines which are owned by some authority. People are trusting the authorities to manage their identity, yet the trust is based on nothing but legal obligations. However, the centralization of the digital storage creates a unique point of failure for identity management systems.

The goal of this work is to research for a solution to the identity management problems using smart-contracts. It is probably known to the reader that a blockchain is widely used as a ledger for crypto-currencies, to keep track of transactions between users. However, blockchain systems recently found its application to create a decentralized execution of computer programs [12]. Hence, we will study how we can use it to create identity management services that are decentralized, but provide the same guarantees to identity authorities and auditing entities.

1.1 Problem Statement

An identity management service is usually centralized to some entity's database, upon which they have full reading and writing rights. As a running example, we will discuss identity ownership in the financial sector. In this example, all private information resides on servers owned by banks, which means they can execute transactions by impersonating a client's identity or might modify his information in any way.

Banks are required to make detailed correspondence between accounts and legal persons. In Belgium, for example, any entity providing financial services falls under the customer due diligence, defined by the National Bank of Belgium and the Financial Services and Markets Authority. This diligence aims to prevent money laundering and terrorist funding. The Belgian Financial Intelligence Processing Unit is the office targeting specifically investigations on suspicious activities, like a transaction exceeding 10000 euros or an unexplained cash deposit. To this end, the office provides regulations about the data that has to be gathered about clients (See Table 1.1). The information has to be supported by relevant documents, copies of which are kept either in paper or electronic form. The data has to remain private, and can be accessed uniquely by the relevant authorities in case of an investigation [13].

Natural Person	Other
Surname, First name	Corporate name
Date, place of birth	Registered office
Relevant address	Directors
	Provisions regarding power

Table 1.1: Information required to identify a client of a bank [10].

Identity management schemes, like the one presented above, involve at least three types of agents: identity management authorities, clients and auditors. Interactions between such different actors require a high level of trust. Nowadays, the trust relies on the law, where any unwanted activities has to be proved illegal in court. This implies that solving disagreements can require long times and large amounts of money.

Therefore, in the system we will implement, a *network* will be defined as a set of nodes running an identification service involving the following agents : *validators* using the service to identify clients, *clients* whose information is stored and *auditing authorities* who use the service to request information about clients. Hence, a protocol governing interactions on a network must provide at least the following functionalities:

- **Registration of a client:** on this first step, all necessary documents will be provided to verify his identity. If authentication was successful, the person will be registered accordingly to a membership service.
- **Identification of a client:** once registered, the service should provide functionalities to verify the client's identity and authenticate him.
- **Expenditure of the network:** the registered validators of the network should be able to add new validators and auditing authorities.
- **Auditing:** the correspondent auditing authorities should be able to register their requests to receive information about a client.

Based on these requirements we will continue with a description of the technical approach to develop a service which fulfills it in a decentralized manner.

1.2 Proposed solution

We sill develop a web service using a different design. Instead of the usual server based web application, we will use a decentralized replicated data storage. This implies that data updates will be governed by a protocol, and not the users implied. To develop such service, we will use an ensemble of computer systems that are referred to as *blockchain technology*. More precisely, we will seek for a system which supports the execution of smart-contracts. A **smart contract** is a computerized transaction protocol that executes the terms of a contract. The general objectives of a smart contract design are to satisfy common contractual conditions (such as payment terms, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries [14].

We will implement a smart-contract which will fit the requirements described in Section 1.1. Accordingly to the properties of a smart-contract, it will provide a trust-

less interaction between the parts. The smart-contract will reside on the Ethereum public blockchain system (see Section 3.5). The contract's integrity will be ensured by the blockchain properties. The Ethereum protocol will provide a decentralized execution of the contract and secure its modifications.

Under the assumption that a client can be identified with a unique identification number, the smart contract will provide sufficient information to ensure that anyone can verify the client's identity. To avoid exposing private information over a public network we will use a digital fingerprint instead. We suppose that the access to private information for registration and auditing will be done outside the system. Nevertheless, it will be possible to verify that the information is correct. Additionally, we will add the possibility to include clients to the service, by asking them to digitally sign modifications of their data.

To interact with the contract, we develop a web application interface. It will serve as a communication channel between the actors and the contract. Before the practical implementation, we will present the current state of the art in data distribution and blockchain technology. Afterwards, we will present a detailed study of the Ethereum protocol, as well as the Bitcoin protocol, which was used as the foundation.

1.3 Summary of Contributions

The contributions of this thesis are the following:

- A comparison between public and private blockchain systems.
- A study of the Bitcoin and the Ethereum protocols.
- A smart-contract to run on the Ethereum network that fulfills the requirements of an identification service.
- A web application to interact with the contract and providing the functionalities of an identification service.

Chapitre 2

Related Work

In this chapter, we will introduce the directions that are currently explored, to change identity management services. Besides, we will tackle technical implications related to identity management.

2.1 Directions in identity management

Unofficially, social identification takes place on the Internet. People identify themselves using the *reputation*, provided by social networks (e.g. Facebook, Google). People are getting increasingly attached to their Internet profiles, and most of web services accept it as a viable proof of identity [15]. Third-party services put their trust into big companies managing the profiles. However, the identity on its place is completely owned by related firms.

As we have explained in Section 1.1, there are is an enormous pressure on the financial institutions to perform customer due diligence. Consequently, announcements have been made about promising innovations in the identity management using blockchain systems [16] [17]. However, they focus on blockchain implementations using private networks (see Section 2.2). Such systems do not bring a different outcome for the user, and keep the data management to the authorities.

On the other hand, there is a potential in a project currently under development, called `uPort`. It aims to provide a self-sovereign identity management using the Ethereum blockchain. A peer will be identified using reputation points, delivered via smart-contracts by another peers. Consequently, the approach is similar to social network identification, but decentralized to a public blockchain instead of private data centers [18].

We have defined interactions to satisfy the needs for each part: the authorities, the clients and the related auditors (see Section 1.1). We think that decentralized management has to be introduced as a familiar service, yet with a goal to keep every actor engaged to his responsibilities.

2.2 Identity Management and Distributed Systems

The problem of identity management involves storing and manipulating sensitive data. Moreover, this data has to be synchronized between geographically distributed parties. As such, distributed systems are a natural approach for fulfilling the requirements of an identity management system (see Section 1.1). A distributed system is a set of physically distributed machines acting together. Overall, such systems aim for providing the following properties:

Availability: there are no downtimes, meaning that each request receives a response.

Consistency: the data provided in a response is the most recent.

Partition tolerance: even if some nodes are failing, and messages are lost, the system will continue to operate.

Security: unauthorized data operations are prevented.

These four properties are already achieved by distributed systems (e.g. distributed relational and key-value storages) used by institutions such as banks for managing their clients' identities [19]. Although it resolves technical requirements for a distributed data, the manipulations of data are in hands of the entity running the physical servers. Current implementations don't provide a way to include clients to the management in a trust-less way. Every interaction between a client and an authority is done in person, and is verified by the employees to be valid.

2.2.1 Byzantine fault tolerance in distributed systems

Distributed databases by themselves cannot decentralize updating power. They were not designed to look for an agreement between different actors updating the data. In our problem, we discuss a situation where two nodes of the same system might give different responses to the same request. Moreover, it is impossible to say which one is correct. Consequently, if one of the nodes is malicious, the response might be faulty. Therefore, correct functioning of the service will be interrupted. Such interruptions are referred as *byzantine faults* [20], a class of faults with no restrictions. Nevertheless, there are distributed systems designed to tolerate such faults using state machine replication.

State Machine Replication is an approach developed by Leslie Lamport [21] for fault tolerance in distributed systems. It defines a system by its state, and limits state transitions to a deterministic function. The state is replicated over every node of a network, and is updated in one possible way. This idea was extended by M. Castro and B. Liskov to develop a protocol for a practical byzantine fault tolerance (PBFT) [20]. This approach assigned a specific role of validators to a set of nodes on the network. Validators are required by the protocol to gather updates, verify their correctness, and settling up a consensus among themselves about the next state of the system [20]. Hence, validators are all running a state changing function, which can provide smart-contracts execution on a network. With popularity of blockchain systems, PBFT protocols and smart-contracts gained a particular interest from the financial and commercial sectors (e.g. Hyperledger [1]).

[1] The IBM's Hyperledger `https://www.hyperledger.org/`

The idea to assign to a set of nodes a different role requires a *private* network, also referred as *permissioned*. Such an approach implements a membership service on the basis of cryptographic certificates [2]. The certificates are used to establish authenticated communication between the nodes. Once the system can make a distinction between the nodes, the PBFT protocol implements a quorum based approach for consensus [20]. Validators are gathered in quorums, then votes are counted to agree if a new state is correct (a new block in case of a blockchain implementation). The use of PBFT protocols has its benefits and downsides. We summarize what a permissioned network approach provides as:

Cheaper and faster transaction validations: predefined validators usually are fewer then total number of nodes. Compared to public networks they only have to agree between themselves to achieve consensus (see Figure 2.1). Additionally, there is no need for economical motivation to validate blocks, it brings internal transactions cost to none, from the point of view of digital currencies.

Increased security and internal decentralization: validating nodes are known and have been preselected, it means that agents do trust them. It reduces risks for malicious transactions and attacks. It doesn't provide a complete decentralization as in public networks, but could have a good validation distribution if there are many different agents as validators.

Reading rights restriction: using a centralized authentication service, reading rights can be restricted to specific nodes. It is possible, using public key encryption, to create a protocol where only concerned parts will be able to read publicly accessible information.

Distribution security: we can mention that a quorum-based voting works under the assumption that at most $\frac{n-1}{3}$ out of total n nodes are malfunctioning [22]. Therefore, in practice, if we compare uniquely the percentage of the network to validate transactions, the private network approach gives in theory better security then the public networks [1]., based on Proof of Work consensus (see Section 3.3).

At the first sight, the properties of PBFT-based blockchain systems are very attractive for identity management. This is the case from the point of view of banks and the government, because it would not have much differences with systems being used at the moment. Therefore, it could, in theory, cut the expenses, but it would not change the approach for data management. The entity who defines validators, or who owns validating machines, would still be granted all power over the network. Consequently, it doesn't suit the purpose we have defined in Section 1.1, to distribute data ownership in a trustless manner.

Drawing the bottom line, we have described techniques used at the moment for identity management. They are able to deliver properties we seek to have in distributed systems. Moreover, a blockchain PBFT-based systems support smart-contracts execution, and implement a form of consensus achieved in an authenticated network. Still, none of them can achieve a publicly distributed ownership of data. On the other side, public network blockchain systems are designed for the environment where everyone is

[2] Cryptographic certificates are based on a public-key cryptography (see Section 3.1) enabling authentication.

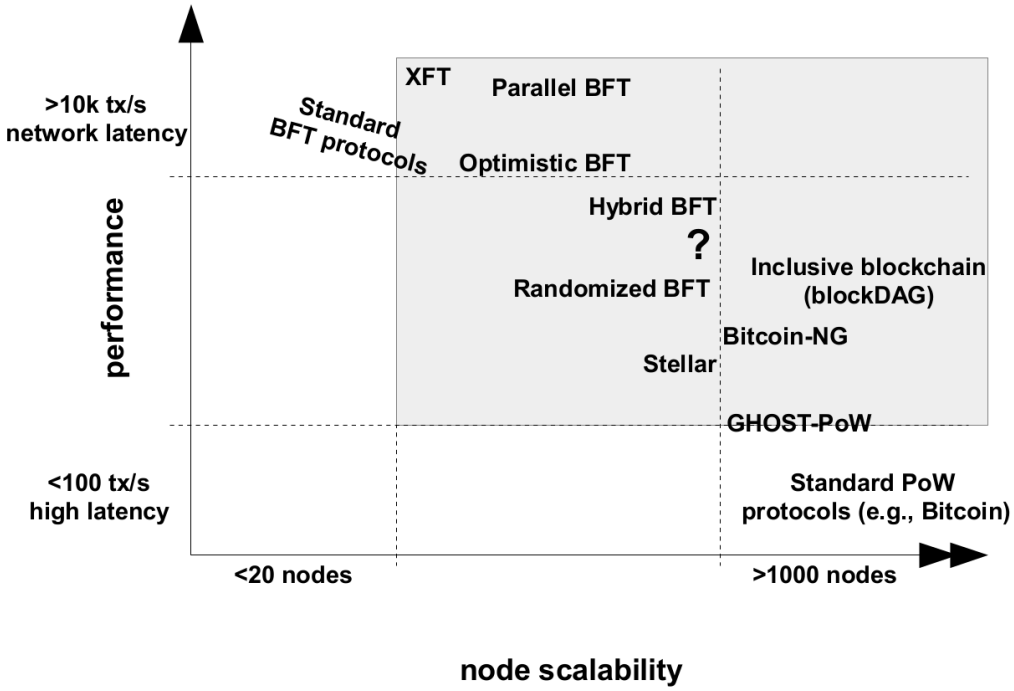


Figure 2.1: A performance to scalability comparison across consensus protocols for blockchain systems [1].

equal. Consequently, such systems are based on protocols and rules delivering a trustless environment. We will explain how it is achieved on the examples of the Bitcoin and the Ethereum protocols.

Chapitre 3

Blockchain systems

We decided to study public blockchain systems because of their main property: being able to establish a trust-less data updating for a distributed system. We are going to explain the pillars on which it is build, and the inherited properties. Cryptography is strongly present in the architecture, to the point to give a second name to digital assets stored on blockchains as cryptocurrency. It used in almost every part of the protocol, the reason why we will start with an introduction to cryptography.

3.1 Introduction to cryptographic functions

In blockchain systems cryptographic functions are used to secretly exchange information through a public channel, digitally sign a message or create a unique fingerprint of a message. *Cryptography* is defined as the computerized encoding and decoding of information [23]. Blockchain systems seek four main cryptographic properties:

Confidentiality - the prevention of unauthorized disclosure of information [24], in other words the message presented on a public channel can only be read by the sender and the receiver.

Integrity - the prevention of erroneous modification of information [24].

Non-repudiation provides a method to guarantee that the one who performed an action cannot falsely claim that he did not [24].

Authentication - the process of verifying that users are who they claim to be, when logging onto a system [24].

The properties are achieved with mathematical functions and protocols, we will not explain them in depth but describe some important aspects.

3.1.1 Cryptographic hash functions

One of the most used and important cryptographic functions in blockchain systems are hashing functions. When someone seeks for a property of data integrity, hashing is the answer. By definition, a **hash function** maps strings of arbitrary size to strings of fixed size, this property is called *compression* [2]. For data integrity the compression is not enough, additionally two more important properties have to be achieved:

Preimage-resistance: it has to be computationally infeasible to find an input given a specific output of the hash function. Also it has to be computationally infeasible to find the input knowing any other existent hash relations [2].

Collision resistance: it has to be computationally impossible to obtain two equivalent mappings for two distinct inputs [2].

For standardization, hashing functions are compared during open competitions. All of the main properties are tested together with the speed of hashing. The `SHA3` function is globally adapted as a winner [25] at the moment. The function chunks the input message into a constant-sized buffers and applies an internal sponge function (having all properties of a hash function) to each of them. Afterwards, all of the sub results are added to obtain a fixed sized output. A global schema of how it works is represented on the Figure 3.1. The `sha3-256` (giving an output of 256bits) function is often used in the protocols we will discuss after. We present in Table 3.1 a constant 256 bit output strings from a function applied to some random inputs.

In summary, if we encode a hash of some data, it is computationally impossible to provide the same hash with a distinct input. Therefore, noone can falsify the initially used information for hashing. Hence, hashes can be stored instead of complete data, it will significantly reduce the storage space, but still ensure the integrity of the information.

3.1.2 Public key cryptography

Public key cryptography, or *asymmetric* cryptography is essential to transfer a message through a public channel with properties of non-repudiation, authentication and confidentiality. Contrary to symmetric cryptosystems, where the encryption key is used for decryption, public key cryptosystems rely on a pair of keys: a public and a private one.

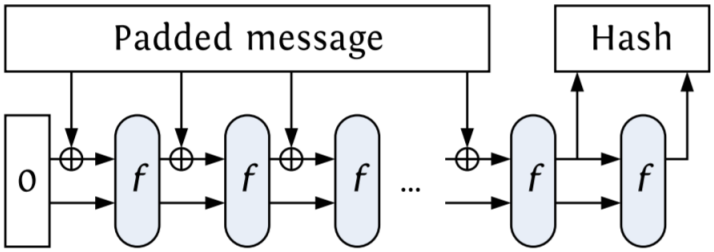


Figure 3.1: Keccak family hash function schema [2].

input	output
blockchain	45740502697d57cbc7e6522372d3247adf1ab8f1cdb0cda1f20a022bf3e153d0
cryptography	d182aed568b01fee105557a1d173791c798030db267cf94e17102b94dcbbda3c
asddfgsdg	39ea28a1dd6a26bc36fe81e7cfc25091df464785d33702fbe89b4191ca7607fd
a1	c1c2b6b28359b94ce47e960c647f0b1cad20e6fb24d5e83b2edb543c74dd3fb9

Table 3.1: Examples of the sha3-256 hash function outputs

Only the public key is publicly accessible, therefore if Alice wants to send a secret message to Bob, she will use Bob's public key for encryption, and Bob will use his private key to decrypt it. As a result, such systems are more likely to be used for large networks because they do not require exchange of private keys.

The security of asymmetric algorithms relies on a difficult, time-consuming task to solve. As example, for the RSA algorithm a multiplication of two large prime numbers is used. It is extremely hard to reverse the problem and find the prime numbers used for multiplication (the problem of factorization). Hence, the algorithm is considered secured if the used prime numbers are large enough, making factorization computationally hard [26]. The downside of typical asymmetric algorithms (e.g. RSA) is the key length, which is significantly larger compared to those used in symmetric algorithms. As a solution, Elliptic Curve Cryptography (EC) can be used. For example, to provide an equivalent level of security for a signature using the RSA algorithm with a 2048 bits key size, the ECDSA will use 224-255 bits keys [27]. Consequently, shorter keys are preferred for a frequent communication over the network to limit the bandwidth usage.

3.1.2.1 Elliptic curve cryptography

An elliptic curve is a solution to an equation of type $E = \{(x, y)|y^2 = x^3 + ax + b\}$. In blockchain systems ECC is widely used because of continuous peer-to-peer communication. Asymmetric algorithms rely on hard problems to solve, in ECC it is the Elliptic Curve Discrete Log (ECDL) problem. It states that a scalar multiplication on an elliptic curve is a one way function, therefore it is hard to reverse. In other words, a point on the curve, called the base point, is multiplied by a scalar number to obtain a new point on the curve. Knowing the coordinates, it is extremely hard to find which value were used to generate the new point. Consequently, the number is chosen privately and is referred

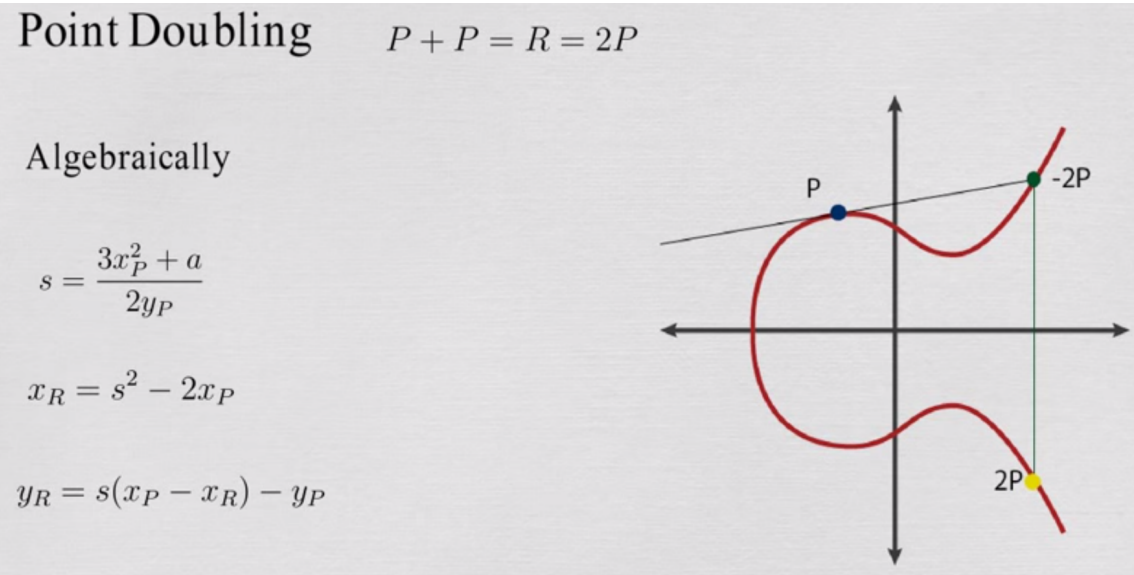


Figure 3.2: Point doubling on an elliptic curve defined on a field of rationals [3].

to as a *private key*. The calculated point is exchanged publicly and is referred to as a *public key* [28]. An example of scalar multiplication (point doubling) is represented on the Figure 3.2. Algebraically, the slope and the point coordinates are used to calculate the new coordinates.

3.1.2.2 Elliptic Curve Digital Signature Algorithm

Digital signatures are used for the purposes to achieve non-repudiation and authentication of the one signing a message. Cryptographic functions are used to derive a value linking a message and a key pair. In ECDSA the approach is the following [29] to sign a message m, assuming that we already have a key pair $\{a_{priv}, Q_{pub}\}$:

Generate point P: from a random a number (should be different for every signature), calculate point $P(x_P, y_P)$ using a scalar multiplication.

Hashing: using an arbitrary hashing function (e.g. `sha3-256`) produce the hash h of m, $h = sha3(m)$.

Signing: calculate $\{r, s\}$ pair representing the signature, where r is calculated from the x coordinate of the public key, and s is calculated using the private key, the hash of the message and r. Hence, s is binding the hash of the message with the private key.

To verify that the signature, a point V on the curve is calculated from the numbers r, s, the public key Q_{pub} and the hash of the message. Then, the r point is compared to the x coordinate of V. If the comparison is successful the signature is correct. Consequently, assuming that a signature is correct, a public key can be calculated solving the equality equation. However, as elliptic curves are symmetric, there are two points that satisfy the condition. One of those points is the public key.

To be sure which of the points was used as the public key, signatures in blockchain systems contain an additional v number to calculate exact coordinates and recover the public key. It is a one byte value, usually stored at the beginning of the signature, therefore called a header byte.

3.2 Cryptographic data structures

After introducing cryptographic function in the Section 3.1, we can present data structures used in block-chain systems to maintain integrity of information. In usual data structures, data is linked using memory pointers. Cryptographic data structure use hash function to link one piece of information to another.

3.2.1 Block-chain

The first data structure we are going to present is the blockchain. It is used to store information in containers of a limited size. A block consists of a header and a data parts (see Figure 3.3). The data can contain any information we are willing to store, respecting storage limitations. The header, in turn, has the requirement to record the hash of the header of the previous block. Moreover, a hash of data part of the block is added to the

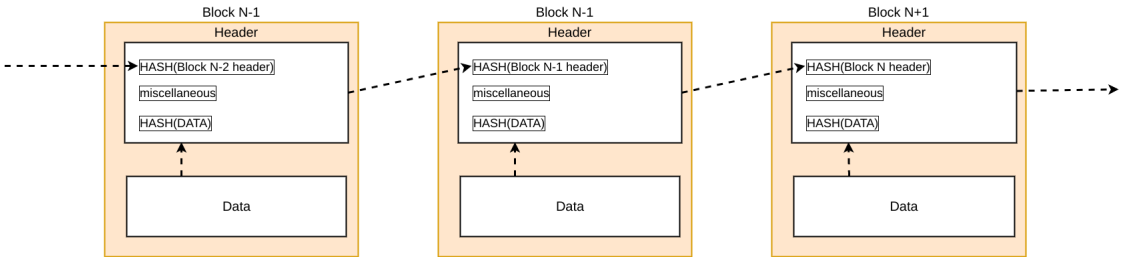


Figure 3.3: Schematic representation of a blockchain data structure.

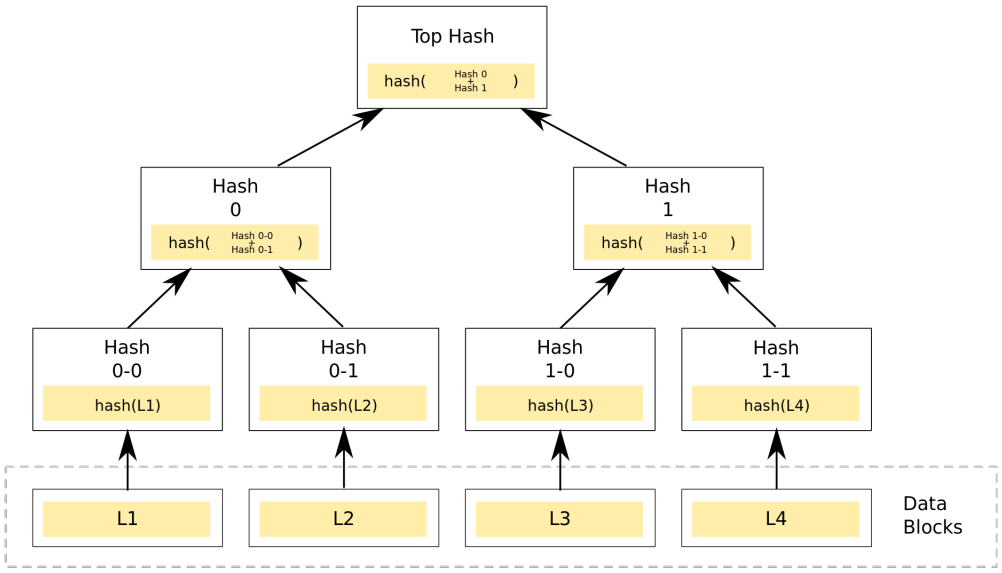


Figure 3.4: An example of a some binary hash tree [4].

header, to keep integrity of the whole block according to hash functions properties (see Section 3.1).

The goal of storing previous block's header is to provide integrity of sequenced information. It means that the data in each previous block was added before and the hash chain proves it. For cryptographic data structures the hashes, as represented on the Figure 3.3, are referred as hash pointers. The reason is that the hashes can be used as keys to look up for the data they were hashed from.

3.2.2 Merkle Tree

The next important data structure is the Merkle tree. It is a binary tree [30], where each internal node has two children. The bottom layer of children are called leafs, and store all actual information. The hash specification indicates that all internal nodes stored hashed information of its children. It results in a property that the root node is a cryptographic

fingerprint of the data stored in the leafs. In blokcchain protocols the leafs point to ordered lists of data.

The reason to use Merkle trees is to provide a proof that a specific piece of data is inside of the data list stored in the leafs. It is done using a Merkle proof: the data itself and the hashes needed to compute the root of the tree. For example on the Figure 3.4 the Merkle proof for `L2` is `[L2, hash(L1), Hash1]`. Having the proof, one can calculate `hash(L2)`, then `Hash0=hash(hash(L2)+hash(L1))` going up the tree to end up with `TopHash=hash(Hash0+Hash1)`. The calculated hash can be compared to the root of the tree to verify if the information block `L2` is in the leafs. In blockchain systems it is widely used to verify that a data was not altered or is present in a block.

3.3 Introduction to the Proof of Work protocol

In distributed computer systems a *consensus* is an agreement between agents of the network about some data. A trustless environment requires for this consensus to be decentralized from the agents of the network. Therefore, a protocol needs to be implemented to update the system independently of the peers. The Proof of Work approach is one of the possibilities.

Originally proof of work (PoW) was implemented to avoid email spamming [1]. Spamming is easy because sending en email was technically cheap, it didn't cost anything to abuse the service by sending a large amount of messages. To prevent spamming, email services added to the protocol a requirement to execute a resource-demanding task for the one sending the message, for it to be delivered. The task has to be relatively hard and at the same time it has to be easy to verify that it was done. The hashcash algorithm was invented for this purpose.

The Hashcash algorithm is implemented using hash functions and relies on its properties. Hash functions are designed to achieve collision resistance and preimage resistance (see Section 3.1). It makes hash functions good one-way functions, for which it is extremely hard to find the input knowing the output. This is the problem hashcash implements: reversing a hash. To reverse a hash one needs to find the input used to produce the hash. To do it, nothing else as brute-force can be used, a repeated hashing of consecutive numbers to fall onto a good input.

We schematically represent how hashcash algorithm is used for emails in the Figure 3.5. Whenever Alice was willing to send email to Bob, before sending she had to produce a specific hash of the email's header and a value called *nonce*. The produced hash has a requirement to start with 20 first bits as 0 in its binary form. It would require on average 2^{20} different hashes to find the one that suits. The header is specific and unique for each email, so the `nonce` number (see Figure 3.5) is incremented for each constructive hash. The receiver, for verification, can directly hash the header with the nonce and check that the output starts with 20 first bits as zeros.

The same approach was used to achieve a consensus for the first decentralized digital currency ledger [2] called the Bitcoin. The Bitcoin consists of a peer-to-peer network

[1] Sending large number of messages

[2] Historically a book keeping records of transactions, currently used to refer to a computer data structure

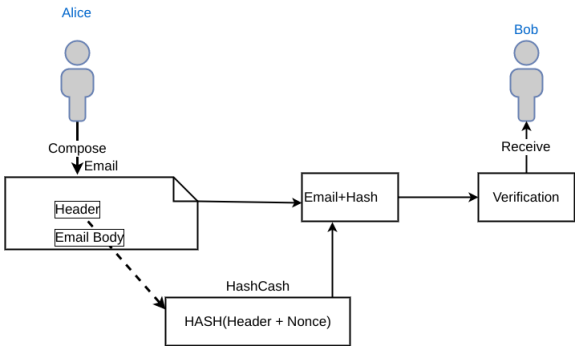


Figure 3.5: Hashcash proof used for email services.

connecting nodes, each of them stores the same ledger of transactions in a block-chain structure (see Section 3.2. Nodes will gather transactions form the network and periodically write them to a block structure. Then they will broadcast it other nodes. Proof of work is used to make it impossible to predict which node will have the rights to add the next block to the chain. A node willing to do so will have to spend some work to solve the PoW's task, and provide a proof to the rest of the network that he did it. Therefore, the updating power is randomly decentralized to the node who completed the task instead of a specific set of nodes. For a good decentralization, the network needs a lot of nodes competing between themselves to achieve a good security [3].

The PoW protocol in blockchain systems implements a variation of the hashcash algorithm. A node will have to hash some data in order to produce an output satisfying the requirements of the protocol. The complexity of the task has to vary in correlation with the global hashing power of the network, to ensure that one node cannot produce hashes faster then the network. Hashing requires some computation cycles of the hardware and consumes electrical power. To motivate nodes to compete for the right hash, nodes are rewarded with an amount of digital currency written to the network's replicated ledger.

On a blockchain network nodes store a replica of a database containing the blocks. In order to update its state a new block will be added in a following way [31]:

* A peer client sends a request with an operation (or a transaction) to be included to the database, the message will be broadcasted to the network and verified by corresponding nodes. Those who compete to find the hashcash hash.
* A peer who finds the hash, adds gathered transactions to the block and writes the hash to block's header. Then he adds the block to the local chain, by including the hash of the previous block to the current header. Afterwards the peer broadcasts block to the rest of the network.
* Other nodes accept the block if the transactions inside are correct and the hash produced satisfies the protocol. By accepting the block, nodes append it to their chain and continue mining on top of it.

Many efforts are contributed to researches in order to replace the proof of work with

having same role.

[3]Here network's security represents the probability to produce the next block for one peer, lower probability ensures higher security.

another, ecologically-friendly protocol. However the PoW for the moment is the only proven to be secure for global scale networks. Hence, we will focus on the Bitcoin and the Ethereum protocols.

3.4 Bitcoin protocol

The Ethereum protocol was planned as improved version of the Bitcoin protocol, so it has a lot of similarities and it will be easier to understand one after explaining the other. The Bitcoin system is the first to use a public replicated blockchain network, having as a goal to create a decentralized digital currency. It worked well by introducing an economical stimulation to the proof of work algorithm (see Section 3.3). The node who succeeded to find a right hash will add a reward transaction with an agreed amount to himself. Therefore, the currency issuance is not owned by the government (as in case of fiat currencies) but defined by a protocol. Digital currencies often take names of the protocol they run on, here it is **bitcoin** as a currency. We will explain each gear that makes the Bitcoin mechanism work, starting by describing how the peers of the network are represented.

3.4.1 Peers representation on the network

First of all, a peer to peer network is established between the clients. P2P protocols create a virtual network on top of the physical one, tagging each node with a special identifier.

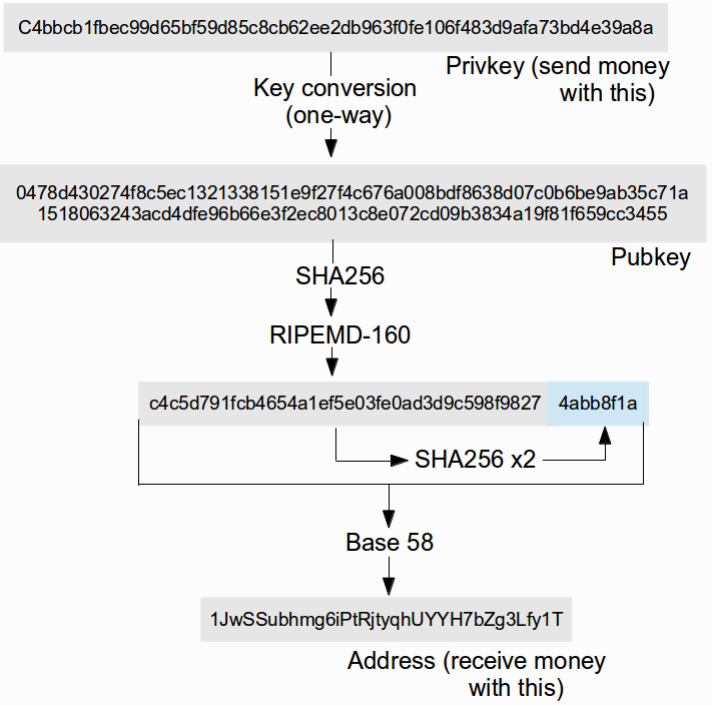


Figure 3.6: Public key conversion to a Bitcoin address format [5].

Global communication over the whole network is established by broadcasting a message from one hop to another. The datasets are public, meaning that anyone can download it from one peer that has it. However, to perform a transaction on the network, one needs a permission to spend bitcoins and a way to indicate the receiver.

For this purpose, each user is represented within the protocol by a unique *address*. The address is cryptographically derived from a public key. Which implies that a person on a network is solely represented by its cryptographic key pair. To prove one's identity, one has to prove the ownership of the private key. It is cryptographically impossible to impersonate someone, unless the private key was leaked.

To create an address, at first, a peer has to generate a key pair. Afterwards, a one way transformation using hash functions is applied to derive an address (see Figure 3.6). A conversion to a human readable format is performed afterwards. Peers can communicate their addresses off the network to receive bitcoins. Hence, addresses are used to indicate the destination of a transaction. A one way transformation serves to prove, anonymously, that you were the destination point of the transaction. Therefore, you have permission to spend it. This combination creates an `output-input` link between transactions.


```
{
    "hex" :
"0100000001344630cbff61fbc362f7e1ff2f11a344c29326e4ee96e787
0100f2052a010000001976a914dd40dedd8f7e37466624c4dacc6362d8e
    "txid" : "a9d4599e15b53f3eb531608ddb31f48c695c3d0b3538a
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "69d02fc05c4e0ddc87e796eee42693c244a31
            "vout" : 0,
            "scriptSig" : {
                "asm" : "3046022100ef89701f460e8660c80808a1
                "hex" : "493046022100ef89701f460e8660c80808
            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 50.00000000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 dd40dedd8f7e3746
                "hex" : "76a914dd40dedd8f7e37466624c4dacc63
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "n1gqLjZbRH1biT5o4qiVMiNig8wcCPQeB9"
                ]
            }
        }
    ]
}
```


Figure 3.7: A Bitcoin transaction in a readable format [6].

3.4.2 Transactions

Transactions are the actual data stored in blockchain data blocks. They serve to keep a track of bitcoin circulation on the network. Hence, the Bitcoin ledger is a trace of interactions between addresses. To create a transaction, one has to prove that he owns the amount of bitcoins he is willing to send. To do so, one indicates transactions in which he was the receiver of bitcoins. In other words, a *transaction* is a mapping of previous transactions to another addresses. For this reason the bitcoin ledger records are referred to as Unspent Transaction Outputs (`UTXO`). Every bitcoin is sent to an unspent output, in order to spend it one has to create a transaction proving that he owns a private key corresponding to the hashed public key (the address) in the output, and indicate the address of the new owner.

We present on the Figure 3.7 a transaction example, where 50 satoshis [4] are sent to a single output with the address `n1gq...QeB9`. The example has one transaction in the inputs array, where we can see its hash and the position of the output in the outputs array. A general transaction structure is summarized in the Table 3.2.

3.4.3 Transaction validation

At the moment we know that users of the Bitcoin network are represented by an address and they can write transactions. Transactions are broadcasted to other peers on the network. Consequently, they will need to verify if a transaction they received is correct, before adding it to a block. There are some simple tests that will be run about the structure of transactions. For example, verifying that the sum of *values* of outputs pointed by the inputs array is greater or equal to the sum of *values* in the outputs array of the transaction. Additionally, the most important part is to verify the scripts `pubKeyScript` and `scriptSig`. They will, eventually, let a client spend the unspent transaction. To run the scripts a bitcoin scripting language was developed. It is a stack language that works with 16-bit set of opcodes, meaning that every 16 bit of a transaction script will be decode to run a specific instruction. Hence, there are 256 possibles instructions.

Name	Desciption
MetaData	information to describe the transaction
Inputs	an array of inputs, where each input indicates a `txid` the identifier of the unspent transaction (its hash), a `vout` number pointing to the position of the relevant output in the outputs array. Moreover, a `scriptSig` script is added to prove that a person who is sending the transaction is the one owns the indicated output.
Outputs	an array of outputs, where each indicates the amount of Bitcoins to send, its position in the array and a `pubKeyScript` script, used to authenticate the receiver of the transaction.

Table 3.2: Bitcoin transaction structure

[4] *Satoshi* is the smallest subdivision of Bitcoin cryptocurrency, 1 satoshi = 10^{-8} BTC

OP_DUP	Duplicates the top item on the stack
OP_HASH160	Hashes twice: first using SHA-256 and then RIPEMD-160
OP_EQUALVERIFY	Returns true if the inputs are equal. Returns false and marks the transaction as invalid if they are unequal
OP_CHECKSIG	Checks that the input signature is a valid signature using the input public key for the hash of the current transaction
OP_CHECKMULTISIG	Checks that the *k* signatures on the transaction are valid signatures from *k* of the specified public keys.

Figure 3.8: Set of opcodes used for scripts to validate P2PH transactions [7].

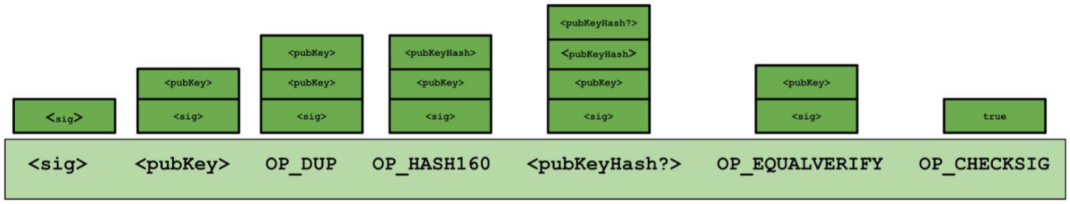


Figure 3.9: A validation process for a P2PH type transaction [7].

For validation there are two types of transactions, they differ in a way the input-output script pair is implemented. The first type, `Pay-to-PubkeyHash`, (or in other words pay to an address) is used for a simple client to client bitcoin transfer. For P2PH scripts use only the opcodes presented on the Figure 3.8. Hence, a `pubKeyScript` is defined as follows: [`OP_DUP`, `OP_HASH160`, `pubKeyHash`, `OP_EQUALVERIFY`, `OP_CHECKSIG`]; the `pubKeyHash` is the address of the receiver. And a `scriptSig` contains a signature from the correspondent public key: [`sig`, `pubKey`]. Both scripts are concatenated for the verification procedure [32] .

An example of an on-stack validation is represented on the Figure 3.9. At first, the signature and the public key from the `scriptSig` are pushed on top. Then by adding the rest of the script, the opcodes will be executed. As we see, the public key will be duplicated and then an address will be derived from it. The next opcode will compare the calculated address from the public key and the one mentioned in the output of the transaction. If both match, the execution continue by verifying the signature using the public key. If the signature is correct, the verification will return `true`. Once the verification has successfully ended, the transaction is correct and will be accepted by members of the network.

From the total possible 256 opcodes, 15 are disabled and 75 are reserved for special uses. For example there is no possibility to create loops (e.g. JUMP opcode), as it would create a risk for infinite loops (nodes would loop forever during transaction validation). The bitcoin scripting language is limited for security purposes, but another type of transactions is possible and is referred to as `Pay-to-Script-Hash` transactions. For this type

Name	Description
Header	a *block version*, a hash (*hashpointer*) to the previous block, a Merkle tree *root*, a *time-stamp* with the generation time, a *target* number, a *nonce*, a *blocksize* number and a *transaction counter* .
Data	an ordered *list of transactions*

Table 3.3: A description for a Bitcoin block components.

of transactions, in comparison with P2PH, the `pubkeyHash` is replaced by a `scriptHash` script. It can be any possible set of P2PH opcodes. Therefore, to spend the `UTXO` one has to provide a script that will hash to the script hash provided by the sender. For example P2SH can be used for *multi-signature* transactions script [32].

3.4.4 Blocks

The first block is created by a peer initiating the chain, all subsequent blocks are generated publicly using the Proof of Work. This process is more commonly known as *mining*.

We have presented that clients will send transactions to another peers, identified by their addresses, which are derived from public keys. Transactions will be broadcasted to other peers, who will execute the correspondent scripts for validation. The next step will explain how transactions are added to blocks. Bitcoin was the first protocol to use blockchain data structure, hence the protocol uses a structure very close to what we have defined in Section 3.2.

We present the Bitcoin block structure in the Table 3.3, it consists of a header and a data parts. The data part here holds the list of transactions. A Merkle tree is calculated from the list, and the root of the tree is added to the header, as we present on the Figure 3.10. Therefore, it is easy to verify if a transaction is inside of a block's list of

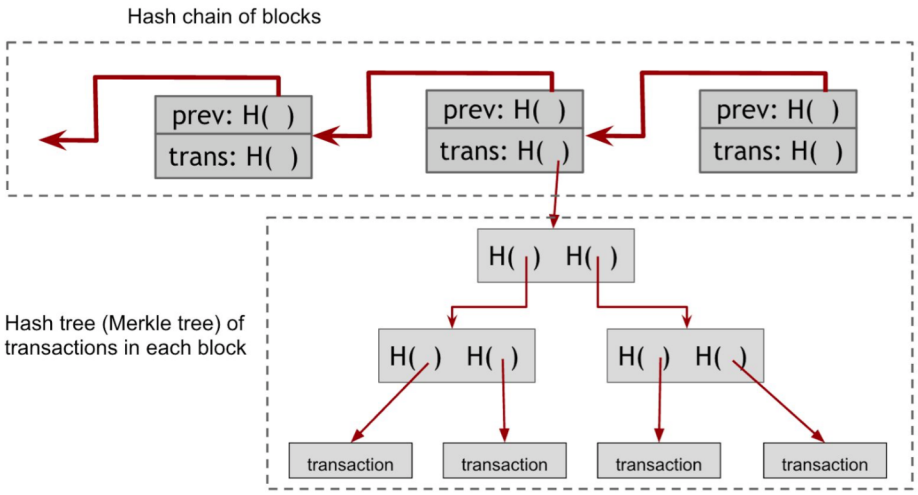
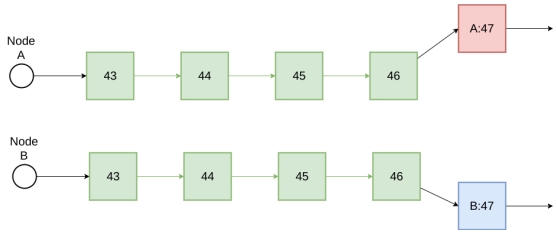
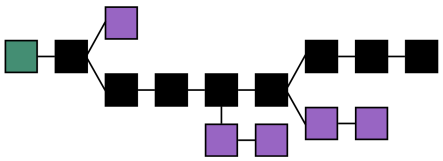


Figure 3.10: Bitcoin blockchain structure [7].



(a) Blocks discrepancy between nodes A and B, updating a correspondent chain to two different versions of the block 47.



(b) Blockchain forks, and agreement on the longest chain, where on each collision the black chain was adapted by the rule [33].

Figure 3.11: Schema for a chain fork.

transactions. The target number describes the difficulty used to generate a block. The transaction counter specifies the number of transactions. The block's size is limited to *1MB*. However, a node who adds a block is not obliged to include any transactions. To motivate him to do so, transaction fees are added to reward the miner.

3.4.5 PoW and consensus

We have presented in Section 3.3 the algorithm used to decentralize blocks generation. The bitcoin protocol implements the hashcash algorithm, and defines the mining difficulty (time to find the correspondent hash) in a way to maintain an average mining time of 10 minutes per block. Therefore, the difficulty is recalculated every 2016 blocks.

The proof of work ensures irreversibility of the chain. A chain is irreversible if noone can produce a longer chain faster than the actual network. It means that it is impossible for a node to have enough computing power to produce hashes faster then the growth of the main chain.

Multiple nodes might find correct but distinct hashes at approximately same time. They will all broadcast different blocks, in hope that other peers will accept the block and validate their reward. The network is distributed, so different set of nodes might accept different blocks. The global state of the chain will result into a *chain fork* [5] as shown on the Figure 3.11a. This divergence is solved with the rule of the *longest chain* adapted by the protocol. It states that if two chains have the same number of blocks they can exist at the same time, and miners will mine the next block each on his forked chain. However, if miners from one chain are faster to add subsequent blocks, the longest chain will be adapted by all other forks being slower (producing less work). Blocks that are not included to the main chain (*orphan* blocks) are lost, represented as purple blocks on the Figure 3.11b.

Using the PoW, the Bitcoin protocol implements a decentrlized ledger for bitcoin digital money, solving the double spending problem with the rule of the longest chain. However,

[5] A fork is as well possible if the source code of the protocol changes (hard-fork or soft-fork), but the same dataset will be used up to a forked block. The changes can be adapted by every peer or some part of them. Forks are used to update the system, to recover from a hack or to create an alternative cryptocurrency (*altcoins*)

as the Bitcoin scripting language uses a limited set of opcodes for security reasons, require to the Ethereum protocol.

3.5 Ethereum protocol

Inspired by the Bitcoin protocol, the idea of replicated transactions execution was taken further by the Ethereum protocol developers. Their goal was to create a distributed system to execute smart-contracts. To achieve this they implemented a PoW-based blockchain system, with a different validation approach. To execute transactions the Ethereum Virtual Machine (EVM) was created, which supports a pseudo-Turing-complete language (see Subsection 3.5.3).

Ethereum is based on the same concepts as Bitcoin, yet with modifications which we will explain in this section.

3.5.1 Clients and smart-contracts

The Bitcoin protocol defines only one type of actors on the chain - the clients, as bitcoins owners. In the Ethereum protocol, smart-contracts were introduced as a code, which is stored as an independent entity on the blockchain. The code, for example can create new contracts and send transactions. Smart-contracts and clients are both identified by a 160bit address (derived as in the Bitcoin), yet to indicate the state of each *accounts* were introduced. An account will represent the state of an address. The Ethereum blockchain will keep a track of state changes for every account in the system.

As the result, Ethereum addresses are mapped to account states, this mapping is called a *world state*. Accounts keeping information about clients are referred as *externally owned* accounts. Such accounts can send transactions and are controlled by cryptographic keys generated by clients. The accounts used to store smart-contract's state are called *contract accounts*. The contract addresses are not owned by anyone. All accounts are written to a trie which is then serialized using an RPL encoding, which then written to blocks data part [35].

Name	Description
Nonce	a value representing the number of transactions performed from this account, or in case of a contract account indicates the number of contract-creations.
Balance	a value representing the number of Wei [6] owned by this account.
StorageRoot	hash of the root node of a trie that encodes the storage content of the account, encoded into the trie as a mapping from 256-bit hashed integer keys to the RLP [7] encoded 256-bit integer values.
CodeHash	the hash of the EVM code, executed when the address receives a message call. In case of a non-contract account the `codeHash` is the hash of an empty string.

Table 3.4: Ethereum account fields definition [11].

3.5.2 Transactions

Unlike in the Bitcoin protocol, transactions in the Ethereum serve to modify the world state. Therefore, there are two types of transactions: those who result in a *message call* and those who result in a new *contract account* creation. A message call is used to modify the world state by adding a transfer of specific amount of Wei (the *value* field) towards an address (the *to* field). It can be done in a simple way or under specific conditions defined by a contract, defined by the *data* field of the transaction (see Appendix A.1). In the last scenario the transfer of Wei will be addressed to a contract and will fall under conditions defined by the code.

For a contract creation transaction, the world state will be modified by mapping a newly generated address to a new contract. The contract will be initialized on the blockchain by adding a code from the (*init* field) of the transaction.

A transaction is directly signed using ECDSA, and the signature will be parsed to the *v,r,s* fields. We have introduced public key recovery from a signature in the Section 3.1. This procedure is applied to recover the public key of the sender, then the address will be derived from. In a case of a digital money transfer, the value being sent will be subtracted from the signer and added to the target account.

3.5.3 Transactions validation

We have presented that a transaction has specific field to hold a byte-code *init* or *data*. Transactions will be executed by running the byte-code on the Ethereum Virtual Machine. EVM assembly supports uses, as well, 16-bit set of opcodes, but they are not restricted as in the Bitcoin scripting. Therefore, a `JUMP` operation is possible to create loops.

To ensure that every program executed in the EVM halts, a value called **gas** is added to transaction execution (it is indicated inside of the transaction as *gasLimit* (Appendix A.1). Each opcode in the EVM has a gas cost, predefined by the protocol. During the execution, the gas sent in the contract will be consumed to execute the code. The execution of the program will halt if run out of gas or the program itself ended [11]. This is the reason why it is *pseudo* Turing completeness, the program as it is might not halt, yet the overall system will. If a transaction is executed well, it updates might update the world state. The changes will be registered to the state trie data of a new block. Transactions are added to their own trie as its data, additionally logs feedback of transactions will be added to the receipt trie data. A transaction execution can end with a failure, for example because of insufficient funds, a wrong code or a lack of gas.

3.5.4 Blockchain data in Ethereum

Merkle trees are used in the Bitcoin protocol to store the root in the header, but Ethereum protocol stores more information to describe its state. Therefore, the developers have used a modified version of a Merkle Patricia Tree and referred to it as a *trie*. We will explain the use of it and its difference.

3.5.4.1 Merkle Patricia Tree

From the original definition: a trie is defined in terms of a mutable data structure to map between 256-bit binary fragments and arbitrary-length binary data, typically implemented as a database. The core of the trie, and its sole requirement in terms of the protocol specification, is to provide a single value that identifies a given set of key-value pairs, which may be either a 32 byte sequence or the empty byte sequence [11].

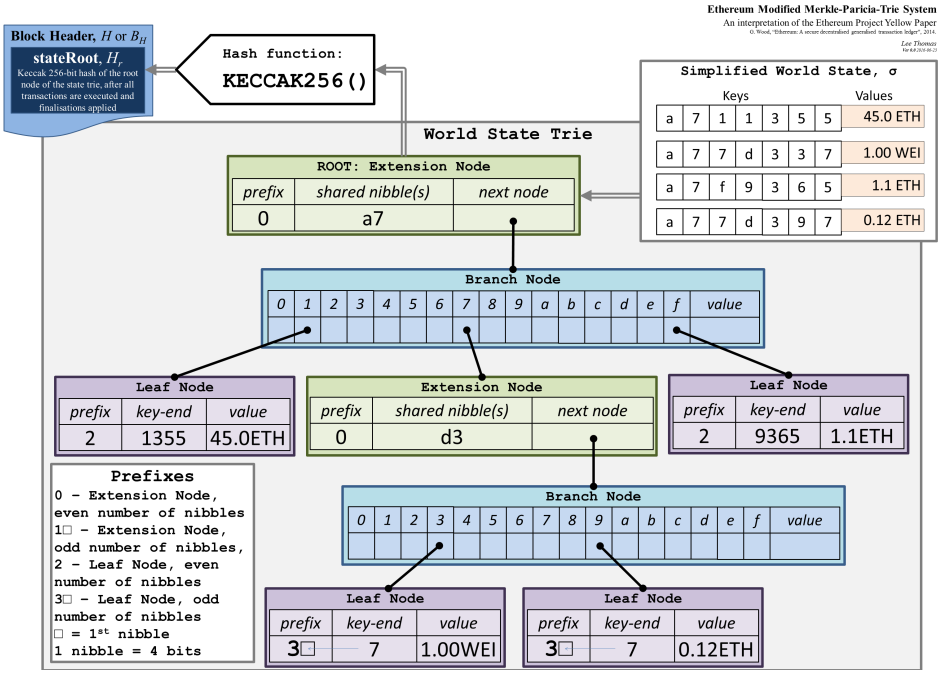


Figure 3.12: Schematic representation of a modified Merkle Patricia tree, used in the Ethereum protocol with a simplified world state [8]

Merkle trees have as a property, that the root is a cryptographic fingerprint of the data stored in the leafs. A trie combines it with a prefix lookup. We can underline dissimilarities with a Merkle tree: four types of nodes are introduced. The first type are *blank nodes* with no data (not shown on the schema), then *leaf nodes* containing a list of `key-value` pairs, where the value is the actual data stored. Then, there are *extension nodes*, they contain a list of `key-value` pairs, where the value is a hash of some other node. Finally *branch nodes* are lists of 17 items, 16 of which represent all possible hex values and the last one is used to store a `key-value` pair pointing to a branch node [36].This shows as well why a recursive prefix length encoding is used to encode the tries data.

A schema on the Figure 3.12 shows how a key look-up works. For example, we can lookup the key `a711355`, we start with a root extension node containing `a7`, then we go to the branch nodes it points to look for the value `1` in its list and follow to corresponding data. It points to a leaf node having the rest of the key `1355` pointing to the actual data representing amount of 45.05 ethers.

3.5.4.2 The blocks

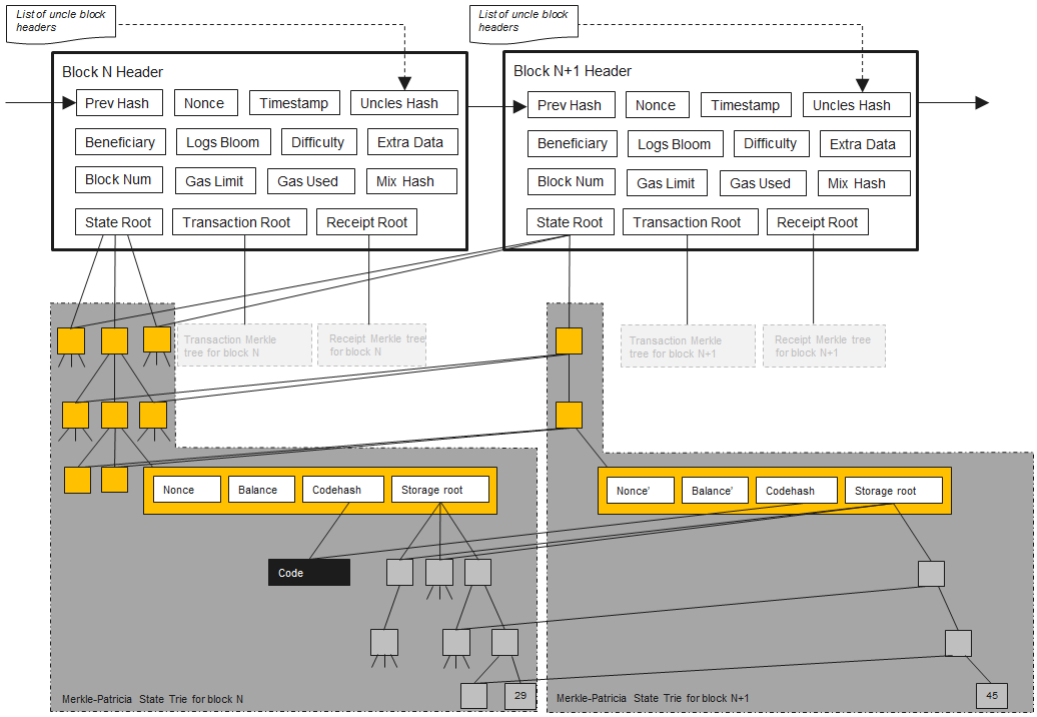


Figure 3.13: Schematic representation of Ethereum blockchain [9].

In the Ethereum protocol a blockchain structure is used to keep track of the changes in the world state. A representation of the block structure is on the Figure 3.13. Compared to what we have seen for the Bitcoin protocol (Section 3.4), additional tries are added to the data block, there are: a *state trie*, a *transaction trie* and a *receipt trie*. The state trie represents the state of accounts in the current block. The transaction trie is formed form the list of transactions in the block, and the receipt trie records all the outputs (events, explained in Section 4.1) of transactions. The receipts help to get the output of a transaction without calling it. They are stored using a bloom structure, mapping hash functions to byte arrays [8]. This was done to give light clients [9] more functionalities.

The data in tries is encoded using recursive length prefix. It omits the type of objects, making the encoding space efficient [37]. The roots of the tries are added to the header. Also, the sum of the gas needed to execute all of the transactions in the block can be found in the header (official description in Appendix A.1). We notice the `ommerHash` and the `logsBloom` fields. The `logsBloom` is a bloom filter for the receipts trie. The `ommerHash` (or the uncle) comes from the consensus algorithm, which includes orphan blocks to decide on the main chain [38]. Hence, the uncle represents hashes of blocks that did find a required hash but were not selected to continue the chain.

[8] For more information see `https://en.wikipedia.org/wiki/Bloom_filter`

[9] The peers storing only the headers of the blockchain to reduce storage space

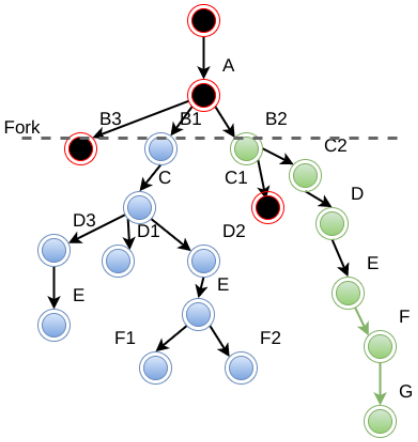


Figure 3.14: The blockchain on the current block level A for the main chain, forking into B1,B2 B3. The longest chain rule would select the block B2, as it results in a chain of 7 hops. The GHOST rule selects the block B1, as it results into a sub-tree of weight 9, compared to 8 for the block B2.

3.5.5 Block generation and consensus

In the Ethereum protocol a PoW algorithm is used to generate new blocks. The hashing algorithm itself was modified from the one used in the Bitcoin protocol. For the reason to increase the mining time to 14 seconds. Moreover, hashing simplicity in Bitcoin is exploited by hardware developers to build specifically designed machines. They have an advantage over an everyday user and promote centralization of mining power. To avoid ASIC [10] mining the algorithm variation (`ethash`) used in the Ethereum protocol was designed to be memory intensive, adding a memory transfer bottleneck to the algorithm [39].

Shorter mining time results into more forks. Consequently, it is more difficult to reach consensus on which chain is correct. The rule of the longest chain is more vulnerable to the 50% attack [38]. A node having significantly higher hashing power than the surrounding peers, can produce the longest chain faster and gain control over the block generation. As the result, another rule was used in the Ethereum protocol to agree on the main chain, called Greedy-Heaviest Observed Sub-tree (GHOST). The algorithm was originally developed to improve transaction speed over the Bitcoin protocol, but it was never implemented.

The GHOST algorithm counts all the forks for subsequent children to evaluate the value for the block to chose. Hence, blocks that are not included (stale blocks) to the main chain are used by the algorithm to chose the main chain. We present on the Figure 3.14 the GHOST selection compared to the rule of the longest chain. On the example we see that it is possible that the green chain (generated by a node with more hashing power) is longer then the others, but the blue sub-tree of nodes is proven to have made more work together. As the result the blue node will be chain will continue the main.

Some modifications were added to the Ethereum-GHOST implementation. The biggest

[10] Application-Specific integrated circuit, the hardware designed to be efficient for a specific computing task.

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"web3_sha3","params":["0
    x68656c6c6f20776f726c64"],"id":64}' localhost:8545

// Result
{
 "id":64,
 "jsonrpc": "2.0",
 "result": "0
     x47173285a8d7341e5e972fc677286384f802f8ef42a5ec5f03bbfa254cb01fad"
}
```

Listing 3.1: An example of a web request to the Ethereum client to hash some data

difference, compared to the original paper, is the *uncle* reward [11]. If a descendant includes a stale (orphan) block as his uncle to the `ommerHash` field, both of them will be rewarded. The stale block (the uncle) will receive 87.5% of the base reward (without the transaction fee) and the block which included it (the nephew) will receive the rest. Economically it promotes a decentralized proof of work.

For the sub-tree comparison, the protocol will go 7 levels deep, and evaluate each sub-tree weight. It implies for a transaction being officially included in the block one has to wait until 7 generations be mined. Afterwards the the block with the transaction is said to be irreversible on the main chain [12].

3.5.6 The Etehereum Client for Web Services

The Ethereum protocol is implemented as a program with the Ethereum client. There are four official implementations, but the most popular is the one implemented in Go language [12]. The client has all the technical details we have explained: address generation, mining, tries hashing, etc [13]. Moreover, the client manages network protocols for connections and communication between peers.

We will use the HTTP-RPC [40] applciation interface provided by the Ethereum client. The reason is that JSON encoding and HTTP requests are easily implemented in JavaScript [41] (the language used to develop the web application). An example [40] of a `POST` [14] request to run a hash function within a client is presented in Listing 3.1.

3.6 Summary

The Ethereum client provides a peer-to-peer network connection between the nodes. Additionally, an application interface is implemented to send transactions to the client or query the dataset of the peer. The network of clients provides a decentralized storage of replicated dataset. It is referred to as *a world state*, which is stored using a blockchain.

[11] In the original paper only the main block is rewarded [38].
[12] `https://www.ethernodes.org/network/1`
[13] Client specifications `https://github.com/ethereum/go-ethereum`
[14] `https://www.w3schools.com/tags/ref_httpmethods.asp`

It represents the history of accounts, their states and state transitions. Accounts might store either the balance of a user or a code referred to as *a smart-contract*. The code is executed by each node in the same way using the Ethereum Virtual Machine.

A smart contract can modify its own state or the state of an account. The modifications are executed by sending transactions to the system. Transactions, states and logs are stored as tries. They are verified by every peer of the network, and gathered into blocks by the mining nodes. Miners compete between themselves to get a correspondent hash, which proves that they spent enough work to find it. The one who did find the hash can publish a block to the blockchain and broadcast the changes to others. The GHOST algorithm is applied as the rule to select the main chain, in case several blocks where mined in parallel.

The actual information is stored in the local storage of each node to verify the correctness of subsequent blocks. Whenever a block is confirmed by enough work spent on the chain, the block is said to be irreversible. The data, then, can be moved to a disk storage and only the header remains in the main memory.

A trustless environment is therefore achieved by the Ethereum protocol based on four pillars: a decentralized block generation with the PoW protocol, a chain selection using the GHOST algorithm, an irreversible blockchain record of state transitions, and a unique state transition function executed by the Ethereum Virtual Machine. Therefore, we will implement a smart contract to reside on the blockchain and ensure a decentralized implementation of the identity management service. We will explain the architecture of the contract, then how to deploy it on the network. Once on the chain, we will provide a web application to call available methods.

Chapitre 4

Identity Management Service

We have studied distributed systems with the goal of finding a suitable implementation of a decentralized identity management application. As the result, the Ethereum public block-chain network was chosen to run the back-end service. We will describe in this chapter the development of the smart-contract to fit the requirements of a decentralized identity management.

4.1 Identity management service smart-contract

Contract-oriented programming languages are new, there are Serpent, LLL and Solidty supported by the Ethereum compilers [1]. We will use Solidity, the official Ethereum programming language for smart-contracts [42].

We have stated in the Section 1.1 the rules under which agents will interact on the network. We need to distinguish three types of peers: clients, authorities and auditors. In the Ethereum network, an entity is uniquely described by its address (see Section 3.5). Hence, we will need to implement a registration procedure inside the application itself.

4.1.1 Registration, data structures

To begin with, we will define data structures to store information about each part. We have presented information needed to identify a client in the Table 1.1. This information will be used outside the system and verified by the authorities. On the other hand, we will store a hash of the information and a description on the blockchain. The authorities using the service and verifying the private information outside the network will be named *validators*. On Figure 4.1 we represent code samples with the information we will provide to the contract about each part. Whenever a node is subscribing to the network, the system needs to memorize its address and to add the required information. We will use mapping [2] data structure to create a key-value relation link between addresses and their information, see Listing 4.4. Mappings have every key initialized, therefore we will need (See Figure 4.1) to add a boolean variable `isRegistered` to mark an address as registered.

[1] How to find each `https://github.com/ethereum/wiki/wiki/FAQ#contracts`

[2] Mappings are hash-table-like data structures, initializing every possible key to a byte array of zeros. See `http://solidity.readthedocs.io/en/develop/types.html#mappings`

```
1
2 struct idCLient{
3  bool isRegistered;
4     bytes32 hash;
5  string description;
6 }
```

Listing (4.1) Data structure for client information.

```
1
2 struct idAuditor{
3   bool isResgistered;
4   string name;
5   string entity;
6   address current_audit;
7 }
```

Listing (4.2) Data structure for auditor information.

```
1
2 struct idValidator{
3   bool isRegistered;
4   uint branchID;
5   string name;
6 }
```

Listing (4.3) Data structure for validator information.

Figure 4.1: Data structures to record information for the registration

```
1   address[] public registeredAddresses;
2   mapping (address => idCLient) public clients;
3   mapping (address => idAuditor) public auditors;
4   mapping (address => idValidator) public validators;
```

Listing 4.4: Public maps to register each part.

Additionally, to have a call access to registered addresses and for monitoring purposes, we will gather all registered addresses into a public array and consequently create a getter for it.

4.1.2 Registration, contract creator

To populate the registration service we will start with the node that creates the contract. We will use the contract's constructor, which is the method with the same name as the

```
1 pragma solidity ^0.4.6; // used to indicate the version of the compiler
2  contract IMS {
3  string public contractName;
4  function IMS(){
5    contractName = "Identity Management Tool";
6    validators[msg.sender] = idValidator(true, 8, "first");
7    registeredAddresses.push(msg.sender);
8  }}
```

Listing 4.5: Contract constructor definition.

```
1  function addClient(address input, string info_hash, string description)
       payable{
2   if( validators[msg.sender].isRegistered == true){
3     if( clients[input].isRegistered ){ feedBack(msg.sender, input,      "
          Already registered");
4       throw;
5     }else{
6       registeredAddresses.push(input);
7       clients[input] = idCLient(true, info_hash, description);
8       feedBack(msg.sender, input, "Cleint was added");
9   }
10   }else {throw;}
11  }
12 }
```

Listing 4.6: The method to add a new client.

contract. The constructor will be executed upon contract initialization as an account. We present the constructor on the Listing 4.5. It will initiate contract's name, as well as add the node who sent the contract creation transaction as a validator. All other nodes will be granted registration if their address and information was communicated by a transaction sent from a client with an address registered as a validator.

4.1.3 Registration of subsequent members

To add a subsequent nodes as a validator, an auditor or a client, we will have to update the state of the contract. The state will change by adding new members to the contract's memory. Such transaction changes world's state, therefore is defined `payable`, as some gas will be burnt for the execution.

We present an example of registering a new *client* to the system on Listing 4.6. First, a verification is made to check the address of the sender, if he has been registered as a *validator* because only an authority can register a new client. Then, the function will proceed by verifying if the requested address has already been registered as a client. If it is the case, the function will throw its execution, otherwise add the address to the `clients` mapping with the correspondent hash of his information. To add a validator or an auditor similar functions are added to the smart contract.

We have to mention that there are two options to execute a function when a boolean condition is not satisfied. A `throw` call can be used to stop the execution in the EVM. In this case transaction will not be added to a block and the transaction reward will not be spent (however gas is paid upfront as we said in Section 3.5). On the other hand, we might proceed without a throw, but with an event call, explaining the problem. This is only necessary for logging during development.

```
event feedBack( address indexed sender, address indexed client, string
    information);
```

Listing 4.7: Event information log to create feedback when a client is added.

```
function compare(string _a, string _b) returns (int) {
function verifyClient(address toVerify, string hashed_info) returns(bool){
  if( clients[toVerify].isRegistered){
    //imported [compare(string a,string b) returns(bool)] function
    if (compare(hashed_info, clients[toVerify].info_hash) == 0){
      return true;
    }
  }
}
```

Listing 4.8: Verifying provided hash of client's information.

4.1.4 Contract events

Events are used to keep an information log easily accessible for external calls (see Section 3.5). At maximum three parameters [3] can be marked as indexed for the filter. The log is stored in the blockchain and inherits its integrity properties. Meaning that it cannot be altered by any contract call. For example we added an event called `feedBack` (see Listing 4.6) to give a feedback about the execution `addCLient` function. In the event we store an indexed address of the transaction sender, an address to add as a client and a feedback.

4.1.5 Verifying identity of the client

The next part of the identity management service is to verify the identity of a client. We proceed with the next public function `verifyClient` shown on Listing 4.8. The input contains client's information hash and his address. The hash will be compared [4] with the one in the mapping.

4.1.6 Auditor Investigations

In case of an investigation, the auditor will create an investigation request. He is only aware of the client's address on the network. Therefore, we added an address field to Auditor's data structure `current_audit`. This address will be initialized to his own address, representing the case in which there is no investigations. The address will point to a client's address if he needs to get his information. Consequently we provide two functions to add and delete clients address (see Listing 4.9).

[3] See for more detailed description `http://solidity.readthedocs.io/en/develop/contracts.html#events`

[4] String comparison function from Ethereum libraries `https://github.com/ethereum/dapp-bin/blob/master/library/stringUtils.sol`

```
1  function addInvestigation(address to_investigate) payable{
2    if(auditors[msg.sender].isRegistered && clients[to_investigate].
         isRegistered){
3      auditors[msg.sender].current_audit = to_investigate;
4      auditFeedback(msg.sender, to_investigate, "Added investigation" );
5    }else
6      throw;
7    }
8  function endInvestigation(address to_delete) payable{
9    if(auditors[msg.sender].isRegistered && (msg.sender ==to_delete)){
10     auditors[msg.sender].current_audit = msg.sender;
11     auditFeedback(msg.sender, to_delete, "Deleted request" );
12   }else
13     throw;
14   }
15   function getInvestigatedAdr(address aud_adr) returns(address){
16     return auditors[aud_adr].current_audit;
17   }
```

Listing 4.9: Methods to manage auditors requests.

We work under the assumption that an auditor will communicate his address to the authority who has information about the client outside the network. Therefore, authorities will query the chain to get the relevant address and provide the information about their client. The procedure will be also done outside of the protocol. We assume, the authorities are obliged to provide the information and auditors work with one investigation at once [5].

We provide to the auditor the possibility to verify if the information is correct using the client verification function. We presented it in Listing 4.8. An auditor will need to hash the information he received and then to verify if it matches the hash stored on the blockchain.

4.1.7 Client signature verification

Once basic functionalities are implemented, we are going to add the client to the protocol by letting him sign the changes of his information. We will implement a method to verify the signature, and use it in correspondent method. To verify that the signature is correct, we will use Solidity utilities `ecrecover` [6] method [7] (see Listing 4.10 line 16). From the documentation we notice that the method requires from the string input, as shown on the Listing 4.10.

[5] Solidity has limitations to work with arrays, and the operations are heavy in gas cost, we have run out of gas often during testings, so we have replaced the array implementation to a unique request.

[6] Doesn't work in TestRPC, the EVM compiler adds a prefix to the signature verification for security reasons, threfore `ecrecover` method works as if the prefix is added.

[7] Method defined on `http://solidity.readthedocs.io/en/develop/units-and-global-variables.html`

```
function verifySignatureStr( address to_compare, string hash_str, uint8 v,
    string r_str, string s_str) returns(bool){
  bytes32 r;
  bytes32 s;
  bytes32 hash_b32;
  //data type conversion
  assembly{
    r:= mload(add(r_str, 32))
    s:= mload(add(s_str, 32))
    hash_b32:= mload(add(hash_str, 32))
    }

  bytes memory prefix = "\x19Ethereum Signed Message:\n32";
  bytes32 prefixedHash = sha3(prefix, hash_b32);

  address addr_sig = ecrecover(prefixedHash, v, r, s);
  return addr_sig == to_compare;
```

Listing 4.10: Client's signature recovery.

4.1.8 Contract price estimation

We can calculate the contract ether price that we need to pay to send the transaction. Default gas price for the Ethereum client is `gasPrice=`$20 * 10^9$ `Wei` [43]. We run the contract through local JavaScript EVM to estimate the gas usage, it is around $2 * 10^6 gas$. We will use $2.7 * 10^6 gas$ for the transaction. Consequently the ethereum price for the contract is $price : \frac{20*10^9*2*10^6}{10^{18}} = 0.054 ETH$, that, at the moment of writing, is around 16 euros [44].

4.2 Defining the Ethereum network

The smart-contract presented in the Section 4.1 has to be tested and deployed on a computer network running Ethereum clients. A network is defined by its protocol specification, written to the genesis block (see Listing 4.11). Additionally, a network has a network identification number. Hence, clients can use the same dataset, but reside on distinct networks.

4.2.0.1 Development Network

We will use two networks during the project. First one is will be used for testing and demonstrations, the JavaScript client named TestRPC [8]. It has no mining time, every transaction is mined almost isntantly. Moreover, there is no need to have ethereums to execute code on the TestRPC EVM. We will run a TestRPC client instance in a Docker [9] container, binded to the local machine(see Figure 4.2) to make `JSON-RPC` requests.

[8] Official distribution for TestRPC `https://github.com/ethereumjs/testrpc`

[9] Software container platform, creating separate environements without virtualization `https://www.docker.com/`

```
{
 "config": {
 "chainId": 2104,
 "homesteadBlock":0,
 "eip155Block": 0,
 "eip158Block": 0
 },
 "difficulty": "0x3a40000", //61079552
 "gasLimit": "0xb2d05e00", //3000000000
 "alloc": // allocate balance for some accounts
 {}
}
```

Listing 4.11: Example of the Genesis block definition used with a Go client.

4.2.0.2 Deployment network

The TestRPC client has slightly different EVM that the one used in the official client. Therefore, we will test the contract on the real Ethereum network. As we have explained networks differ by its protocol definition and network id. Based on the official genesis block, there are the main network under `id=1`, and the test network `id=3` [10]. Both require synchronization of the database that have already been mined. However, the test network has lower difficulty, making mining faster.

To avoid synchronization time and to adapt mining to the local machine, we have decided to run a private network initialized with the genesis file (see Listing 4.11). We have written commands to install Ubuntu and the `go-ethereum` client to a Dockerfile. The commands to run the client will be executed manually, because accounts have to be initialized with a password. At least one node has to start the client with the mining option to execute transactions.

4.3 Identity management service contract deployment and interactions

We will compile and test the contract we developed in Section 4.1 on the networks we presented presented in Section 4.2.

4.3.1 Using the TestRPC client

The TestRPC client is friendly with users by providing feedbacks and error codes during EVM compilations. Moreover, we can automatically unlock specific amount of accounts, which facilitates the testing. We used `Web3` JS [11] library to encode and send RPC calls to the client.

On Figure 4.2, we present the sequence of actions to take to publish the contract. First

[10] Is named as Ropsten network, is officially agreed as a common test-network between Ethereum clients.
[11] https://github.com/ethereum/wiki/wiki/JavaScript-API

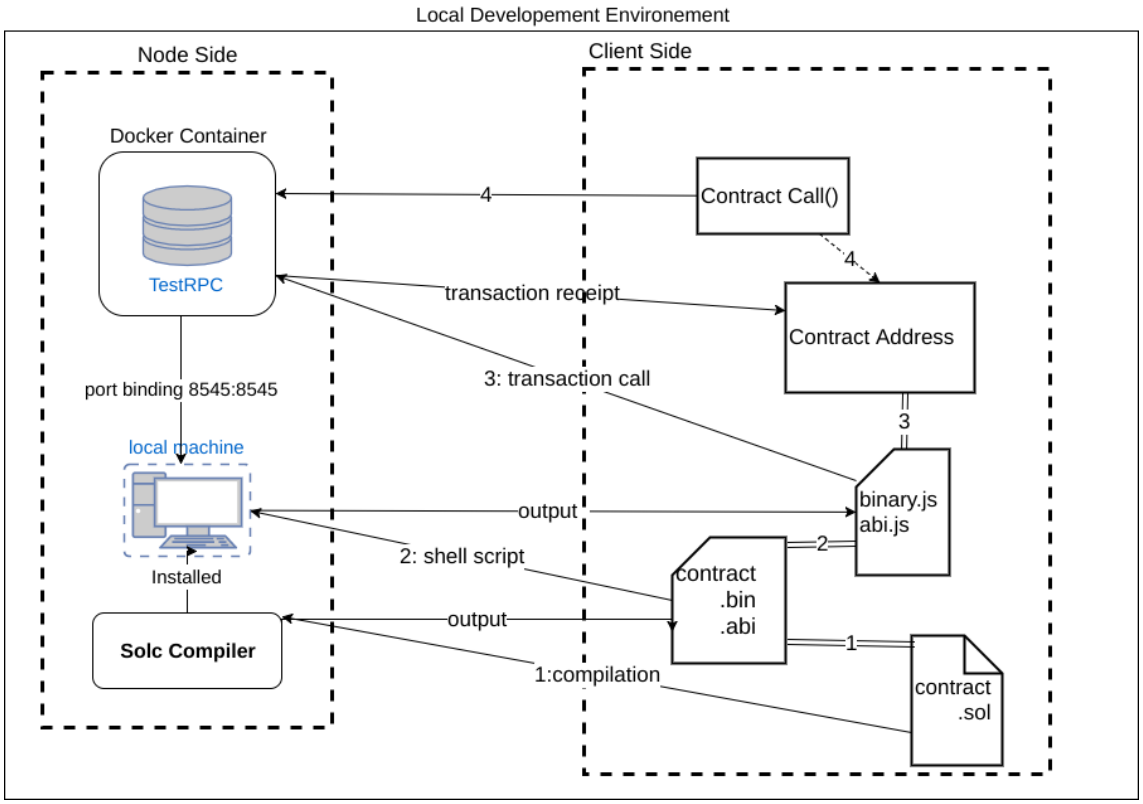


Figure 4.2: Schema of contract deployment using a TestRPC client.

of all, on the *Step 1* the contract code is compiled to its *bytecode* and *ABI* [12] definition. Then we write the output of the compiler to correspondent JavaScript(JS) files, *Step 2*. In each file we define JS variables to represent the bytecode and the ABI definition. They will be directly included to the node application source folder, and consequently to the application using Webpack [13] dependencies manager. The bytecode variable will be used to perform the transaction, creating a new account with the contract code,*Step 3*.

For the web application we need to provide a connection to the client. We will indicate to the `web3.HttpProvider` object the IP address and the correspondent port. Once a connection is established, we have to create or import accounts to work with. For testing we work with a list of 10 unlocked accounts. With an account we can send transactions to the chain (see Listing 4.12). Transaction is hashed, and the hash is used to query the blockchain for a transaction receipt. The receipt indicates contains the address of the contract. Using the address we define a `web3` contract object for the contract. Additionally the method requires the ABI definition (from contract compilation). The object will let us use the methods defined in the ABI, and wrap them into direct JSON-RPC calls to the client, represented as the *Step 4*.

We can see the output of the contract creation transaction on the Figure 4.5b, we

[12]Application Binary Interface defines the functions of the application and how to interact with it.
[13]Dependencies manager, including all requried file to one JS file, `https://webpack.js.org/`

```
web3.eth.sendTransaction(
 {from: web3.eth.accounts[0],
  data: binaryContract,
  gas: '2700000'}, function(err, transactionHash) {
   if (!err)
   var receipt = web3.eth.getTransactionReceipt(transactionHash);
 });
```

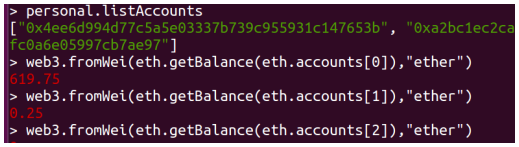
Listing 4.12: Contract creation transaction using web3 library.

Figure 4.3: Trnasaction receipt, as the output of the query to the blockchain using the relevant transaction hash.

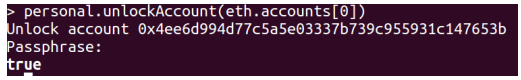
```
// var vm = this; is a label for the controller object
  var abi = JSON.parse(vm.minedContract.abi);
  var addr = vm.minedContract.address
  vm.contract = web3.eth.contract(abi).at(addr);
  vm.events = vm.contract.allEvents();
  // watch for changes
  vm.events.watch(function(error, event){
  if (!error)
    console.log(event);
  });
```

Listing 4.13: Creating a Web3 contract object.

represent directly a contract receipt query based on the transaction hash. At first, we monitor the blockchain state, and notice that it was incremented by adding a new block `0x5a..b5`. The transaction we queried for is indexed under the number 0. We can also notice the amount of gas that was used for to run the transaction, as well as the total used in the block. The amounts are equal as there is only 1 transaction. The receipt indicates the address of the created contract, it can now be used to create a web3 contract object (see Listing 4.13). We additionally add an event watcher to query for new updates in the contract.

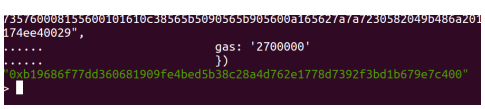


(a) Accounts stored on the node and their balances.

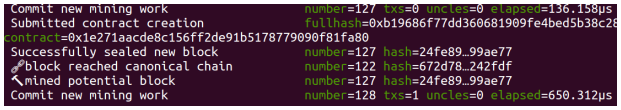


(b) Unlocking the coinbase account to deploy the contract.

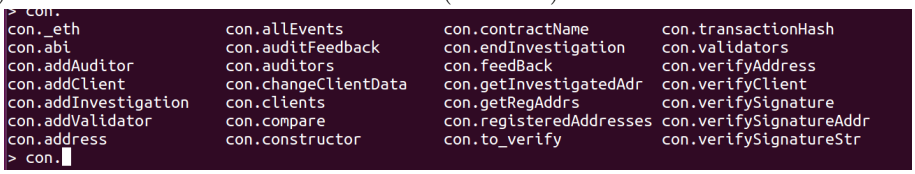
Figure 4.4: Accounts managing in the Ethereum client.



(a) Contract creation transaction. (Sender)



(b) The transaction was added to the mined block 127.(Receiver)



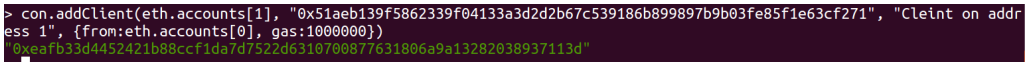
(c) Contract object methods.

Figure 4.5: Contract creation transaction processing.

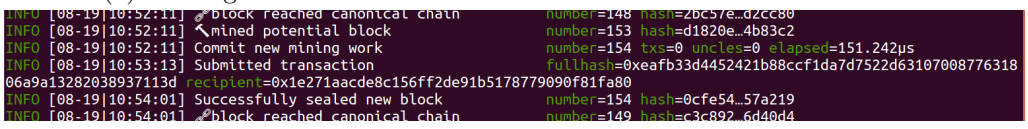
4.3.2 Using the Go client

We run a private `go-ethereum` client as a miner to test functionalities of the contract. At first, we create few accounts and mine some ethers (see Figure 4.4a), then unlock the first account (see Figure 4.4b). Having an account, we send a transaction with contract's bytecode and the gas amount to execute it 4.5a. We can see from the miner's output (see Figure 4.5b) that he has received the transaction, and added it to the freshly mined block 127. Afterwards, as well as in the TestRPC client, we create a contract object to call the functions (see Figure 4.5c).

We can now use the functionalities of the contract. As example, we add one of the accounts on the node as a client (see Figure 4.6a). The transaction is sent using the method call, and added to the next mined block 154 (see Figure 4.6b).



(a) Sending the transaction to call `addCLient` method of the contract.



(b) The transaction to add a new client was added to a mined block 154.

Figure 4.6: Client registration transaction processing.

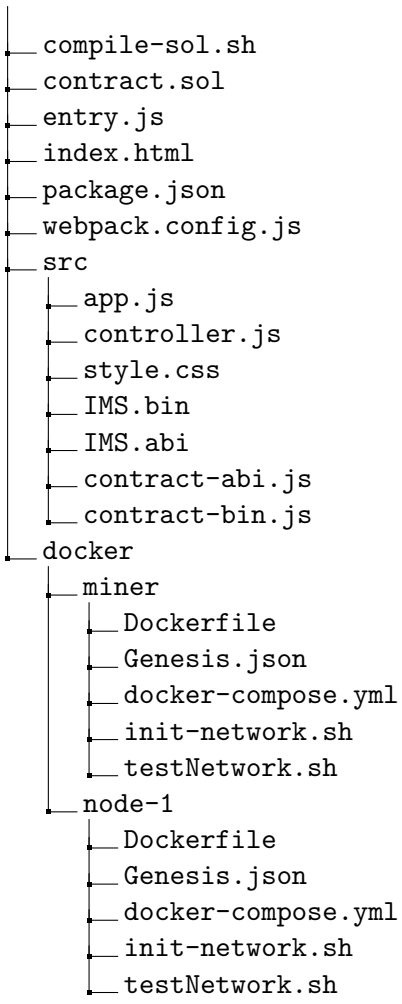


Figure 4.7: Project structure.

4.3.3 Web application to interact with the contract

Contract methods are binded to a web application interaction interface. The code of the application can be found via `https://github.com/nazarfil/IMS`, with the corresponding requirements and installation guidances. The application is organized under the structure presented on the Figure 4.7.

To compile and parse Solidity contract code `compile-sol.sh` script is provided. The contract itself is in the `contract.sol` file. The javascript files `entry.js`, `app.js`, `controller.js` are used to define the application functionalities and indicate the required modules. One can use the service from the `index.html` file with the help of a web-browser. Additionally, folders with docker definitions are available in the project to start a network or to connect to one.

Chapitre 5

Conclusion

The identity management services, as we know them, are all implemented in a similar way. A centralized storage is used to keep the digital representation of individuals. We have explored a new direction, moving the management to computer programs known as smart-contracts.

5.1 Summary

We have provided a detailed explanation of protocols that achieve a trustless environment for distributed computer systems. The explanations were made on concrete examples of the Bitcoin and the Ethereum protocols. We have studied the adoption of smart contracts in the Ethereum protocol, and have designed one to manage identities. The architecture provides functionalities required by the services we know today.

The contract resides on a publicly replicated dataset known as the Ethereum blockchain. The Ethereum protocol decentralizes the writing rights and ensures the program is executed independently from the actors taking part in it. The history of contract execution is unique and immutable, by the properties of the blockchain structure. The changes to the client's data have to be approved using their digital signature. We have, also, developed a web application for the service. It provides an interface for the users to interact with the contract.

Overall we have demonstrated an alternative approach to implement web services for data management. Smart-contracts, residing on the blockchains, can provide better integrity, security and consistency of private information.

5.1.1 Limitation of the approach

The strong reliance on cryptography in blockchain systems provide many essential properties. However, it creates a danger from the user side. If the user loses his key pair, and is unable to recover it, he loses the means to claim his identity. Moreover, if one has stolen the keys and impersonates the user, he cannot prove that it were not his actions.

The implementations of blockchain systems that support smart contract are not mature enough to be called secure. The attacks on the opcodes pricing already took place in the

Ethereum network. More attacks will, probably, take place in the future, and change the system we know at the moment.

Additionally, the method calls to the contract have to be approved by miners, therefore a delay is introduced to the service.

Appendix A

Ethereum

A.1 Ethereum transaction definition

Name	Desciption
nonce	a value to represent the number of transactions sent by the sender. Having for purpose to mark a transaction as unique.
gasPrice	indicates the exchange rate between Wei to gas (an amount assigned to each `op_code`, represents the price of execution by Ethereum Virtual Machine).
gasLimit	indicates the maximum amount of gas needed to execute the transaction. The amount is paid up-front and cannot be increased later.
to	is a 160-bit address of the message call's recipient or is empty in case of contract creation.
value	represents the number of Wei to be transfered to the recipient. In case of a contract is a value of endowment to a new contract.
init	exclusively for a contract transaction, an unlimited size byte array that contains the contract code for account initialization procedure.
v,r,s	values used for the signature of the transaction. The address will be recovered from these values.
data	exclusively for a message call transaction, is an unlimited size byte array with the data of the message call. Is present only for message calls.

Table A.1: Ethereum transaction structure definition [11]

A.2 Ethereum block definition

- **parentHash**: is the hash of the previous block's header.
- **ommersHash**: is the hash of a list of blocks called uncles, they were mined at the same time as this block, however are not used as a parent block to continue the chain. Is as well refereed as **Uncle** or **Uncle hash**.
- **beneficiary**: represents the address collecting transaction and mining fees.
- **stateRoot**: the root of state trie
- **trasnactionRoot**: the root of transaction trie
- **receiptRoot**: the root of receipts trie
- **logsbloom**: the Bloom filter composed from indexable information contained in each log entry from the receipt of each transaction on the transaction list.
- **difficulty**: difficulty under which the block was mined.
- **number**: total number of previous blocks.
- **gasLimit**:represents the amount of gas expenditure per block.
- **gasUsed**: the sum of gas needed to run every transaction from the block.
- **timestamp**: time when block was conceived
- **extraData**: additional data relevant to the block.
- **mixHash**: a 256bit hash used together with *nonce* to prove that the amount of computations was sufficient.
- **nonce**: a 64bit hash used together with *mixHash* to prove that the amount of computations was sufficient.

Information resumed from the yellow paper [11].

Bibliography

[1] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," 2015.

[2] O. Markovitch, "Computer security: Integrity," 2016.

[3] R. Pierce, "Elliptic curve diffie hellman." https://www.youtube.com/watch?v=F3zzNa42-tQ, 2014. [Online, Accessed on 06-March-2017].

[4] "Hash tree image." https://en.wikipedia.org/wiki/Merkle_tree#/media/File:Hash_Tree.svg. [Online, Acessed 23-May-2017].

[5] "Bitcoin address generation schema." https://bitcoin.stackexchange.com/questions/32353/how-do-i-check-the-checksum-of-a-bitcoin-address. [Online, Acessed 23-May-2017].

[6] "Bitcoin address generation schema." https://people.xiph.org/~greg/signdemo.txt. [Online, Acessed 23-June-2017].

[7] "Btc images for stack and blocks source." https://leuchine.github.io/2016/07/19/Blockchain-and-Bitcoin-(3).html. [Online, Acessed 06-July-2017].

[8] "Merkle patricia tree image." https://i.stack.imgur.com/YZGxe.png. [Online, Acessed 23-May-2017].

[9] "Schema of ethereum blockchain structure and states transition between two consecutive blocks." https://i.stack.imgur.com/eOwjD.png. [Online, Acessed 02-August-2017].

[10] FATF, "Guidance for a risk-based approach - the banking sector." http://www.ctif-cfi.be/website/images/EN/rba/rbabankingsector.pdf, 2014. [Online; accessed 2-August-2017].

[11] G. WOOD, "Ethereum: A secure decentralised generalised transaction ledger," 2014.

[12] E. Foundation, "Ethereum white paper." https://github.com/ethereum/wiki/wiki/White-Paper#modified-ghost-implementation, 2014.

[13] R. Hoskens, "Know your customer quick reference guide." https://www.pwc.com/gx/en/financial-services/publications/assets/pwc-anti-money-laundering-2016.pdf, 2016. [Online; accessed 2-August-2017].

[14] N. Szabo, "Smart contracts." http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html, 1994. [Online; accessed 2-August-2017].

[15] G. Ralph, A. Alessandro, and H. J. H., *Information revelation and privacy in online social networks.* 2005.

[16] "Canade banking system identity management using blockchain.." http://business.financialpost.com/news/fp-street/canadas-big-banks-testing-toronto-based-digital-identity-network-powered-by-blockchain/ [Online, Accessed March-30-2017].

[17] "French arkea insurance system identity management using blockchain.." https://www-935.ibm.com/industries/fr-fr/banking/case-studies/blockchain_credit_mutuel_arkea.html. [Online, Accessed February-12-2017].

[18] "uport, an ethereum based identity management with reputation.." https://www.uport.me/. [Online, Accessed March-03-2017].

[19] D. A. Bell and J. B. Grimson, *Distributed Database Systems.* 1992.

[20] M. Castro and B. Liskov, "Practical byzantine fault tolerance," 1999.

[21] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," 1978.

[22] D. Malkhi and M. Reiter, "Byzantine quorum systems," 1996.

[23] "Definition of cryptography." https://www.merriam-webster.com/dictionary/cryptography. [Online, accessed 9-August-2017].

[24] W. Jiang, "Cryptography: What is secure?," 2003.

[25] NIST, "Sha-3 selection announcement." http://csrc.nist.gov/groups/ST/hash/sha-3/sha-3_selection_announcement.pdf, 2012. [Online, Accessed 14-August-2017].

[26] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," 1977.

[27] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Nist recommendation for key management – part 1: General," 2012.

[28] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," 2001.

[29] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm," 1999.

[30] M. Ramkumar, *Symmetric Cryptographic Protocols.* 2014.

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

[32] "Bitcoin transactions types explained." https://bitcoin.org/en/developer-guide#standard-transactions. [Online, Acessed 17-July-2017].

[33] Unkown, "Image for longest chain rule." https://cdn-images-1.medium.com/max/1200/1*xgpVRSJA707FH3m_x4nHWw.png, 2017. [Online, Accessed 12-August-2017].

[34] "Rlp encoding source." https://github.com/ethereum/wiki/wiki/RLP. [Online, Acessed 10-August-2017].

[35] "Ethereum trasnaction types." https://github.com/ethereum/wiki/wiki/White-Paper#ethereum-accounts. [Online, Acessed 04-May-2017].

[36] "Ethereum trie explained." https://easythereentropy.wordpress.com/2014/06/04/understanding-the-ethereum-trie/. [Online, Acessed 23-May-2017].

[37] "Recursive length prefix." https://github.com/ethereum/wiki/wiki/RLP. [Online, Accessed 13-July-2017].

[38] Y. Sompolinsky and A. Zohar, "Accelerating bitcoin's transaction processing fast money grows on trees, not chains," 2013.

[39] https://www.vijaypradeep.com/blog/2017-04-28-ethereums-memory-hardness-explained/, 2017. [Online, Accessed 09-August-2017].

[40] "Ethereum clients and the apis they support." https://github.com/ethereum/wiki/wiki/JSON-RPC. [Online, Accessed 16-August-2017].

[41] "Javascript json documentation." https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/JSON. [Online, Accessed 18-August-2017].

[42] "Solidity, official contract language.." [Online, Accessed 18-August-2017].

[43] "Ethereum client cli." https://github.com/ethereum/go-ethereum/wiki/Command-Line-Options.

[44] "Cryptocurrencies market price and market capitalization, averaged.." https://coinmarketcap.com/. [Online, Accessed 08-July-2017].