



Blockchain Ethereum Private Network

IOANNIS MICHAEL

Professor Costas Lambrinoudakis
Piraeus 2018

Table of Contents

1. Introduction	4
2. Trust in modern computer networks	5
2.1 Reputation Systems	5
2.2 Public Key Infrastructure	6
2.3 Trust Classes	7
3. Software Architectures	8
4. Distributed Systems.....	10
4.1 CAP Theorem.....	11
4.2 Byzantine Generals	11
4.3 Consensus.....	12
5. Blockchain Technology	14
5.1 Block Structure	14
5.2 Block Header	14
5.3 The Genesis Block	17
5.4 Mining	17
5.5 Bitcoin Proof of Work.....	17
5.6 Ethereum Proof of Work (Homestead).....	18
5.7 Proof of Storage.....	18
5.8 Ethereum Proof of Stake (Serenity).....	19
5.9 Intel SawtoothLake Proof of Elapsed Time	19
6. Smart Contracts.....	21
6.1 Oracles	23
7. Decentralized Applications DApps.....	24
7.1 Classification of DApps.....	25
7.2 Advantages and Disadvantages of DApps	25
7.3 User identity	26
7.4 User Accounts.....	27
7.5 Popular DApps	27
7.5.1 Bitcoin	27
7.5.2 Namecoin	28
7.5.3 Ethereum.....	28
7.5.4 The Hyperledger project	28
7.5.5 Litecoin	29
7.5.6 Primecoin	29
7.5.7 Zcash	29
7.5.8 Sia.....	29
8 Ethereum	30
8.1 Ethereum Clients	30
8.2 Ethereum Accounts.....	31
8.3 Ethereum Currency.....	32
8.4 Ethereum Transactions	33
8.5 Gas	34
8.6 Ethereum Consensus	34
8.7 Ethereum Virtual Machine EVM.....	35

8.8 Ethereum Block	36
8.9 Ethereum Network – Peer discovery.....	38
9. Private Network Setup.....	40
9.1 Geth Install	40
9.2 Private Data Directory.....	40
9.3 Genesis File.....	41
9.4 Initialize Network.....	41
9.5 Account Creation	42
9.6 Peering	44
9.7 Sending Ether	46
10. DApp Development	49
10.1 Development Workflow	49
10.2 Application Workflow	50
10.3 Setting the requirements for the nodes	51
10.3.1 Supervisor – Running Script on Boot.....	51
10.3.2 Script for the requirements.....	52
10.4 Launching the DApp.....	53
10.5 Authentication	58
10.5.1 Smart Contract.....	58
10.5.2 Login Form.....	60
11. Conclusion.....	66
Bibliography	67

1. Introduction

Throughout the history of mankind, trusted relationships have played a vital part in every transaction humans have made. Those transactions belong to a spectrum that starts from everyday life decisions and acts, to a more complex, sensitive and wide area that even nations are involved.

Before the era of globalization of telecommunications that we live in, achieving trust was more related to human relations. Even though that the meaning of trust is known to all, it is hard to find a definition that strictly describes it.

Trust is multidimensional, multidisciplinary and multifaceted concept. Many definitions can be found in literature and are related to notions as goodness, strength, reliability, integrity, ability or character of a person or thing. A trust relationship involves two parties, a trustor and a trustee. The trustor is the person that holds confidence, belief on the reliability of another person or thing which is the other party, the trustee. (Zheng & Valtteri Niemi, Towards User Driven Trust Modeling and Management, 2009)

How though trust is established in modern computer networks, where the notions of the trustor and trustee are not represented by strictly humans, but from entities that might never have had a relationship upon the trust can be build.

In this project we will study the achievement of trust in traditional kinds of networks such as ad-hoc, mobile and wireless and we will examine the ability to elevate the trust level in a computer network using the under development and mostly promising blockchain network.

The network is going to be setup as a private blockchain network, where all the nodes that consist it, will be pre-set from an administrative team. The computers that will participate will have all the requirements in order to connect to the private network running as services on boot.

The application will run on each node and on starting the application the very first check will be to start the node and connect to the network. Only if the network has been found and the node is connected to it, the application proceeds with checking the presence of web3js and only after successfully checking the communication of the web3js with the network, the user is prompt with the login page.

The authorization of the user is checked upon a smart contract on the blockchain network and after a successfully prompt from the smart contract, the credentials are checked, in our case, on a fake backend where a JWT token is issued to the user in order to use the application depending on the role that he has.

2. Trust in modern computer networks

Before we see how trust is being established in modern computer networks we should see the differences between the real world and the world that is defined by computer networks.

In real life, establishing and extending a trust relationship is defined by trusting a colleague and trusting a colleague's trust about another colleague. In computer networks, the colleague is replaced by the node entity. Thus, trust is established by trusting a node, and extended by trusting a node's trust about another node.

The flow of how trust is established in the real world is constructed in the following example. For the sake of the example we have colleague A, B and C. Colleague A trusts colleague B. Colleague B trusts colleague C. In order to establish trust between colleague A and C, colleague C contacts A with a reference to colleague B. At the end colleague A calls colleague B to confirm his trust. (Eschenauer, Gligor, & Baras, 2002)

Of course, the role of the participants in a relation of trust can be played either by persons, organizations, government authorities and states. The physical interaction is crucial to the initial establishment of the trust relation.

In the computer network, for the sake of the example we will use nodes A and B. A certification authority certifies node A. The certificate is stored in a server. Node B has to contact the server to fetch that certificate in order to establish trust on node A.

We can observe that in the real world no trust infrastructure is required to achieve trust. In the computer network the presence of a trust authority is vital and that leads us to a single point of trust, to a single point of failure, that of the trust towards to the certification authority.

Additionally, due to the fact that communications in the computing network rely not only on human beings and their relationships, but also in digital components, the establishment of trust involves more aspects. The visual trust impression is missing and the accuracy of remote sent information can easily be distorted or even faked identities can be created. So in order to mitigate a possible security or privacy risk, it is crucial to understand the trust relationship between two digital entities. (Zheng & Holtmanns, Trust Modeling and Management: from Social Trust to Digital Trust, 2007)

2.1 Reputation Systems

The evolution that internet has brought into the electronic transactions has led towards the discovery of a new trust model. This is reputation based. The idea that is based on, is that entities after the completion of a transaction can rate each other and the aggregated ratings about a given party are used to achieve a trust based on the reputation score. That score can assist parties that want to complete future transactions in order to decide whether or not to transact with that specific party.

The natural side effect of this system is in order the given party to maintain a high reputation score and as result to be chosen for future transactions, is to have a good behavior that results to an overall positive effect on market quality. (Audun, Roslan, & Colin, 2007)

We see that trust is not only based on the reliability of an entity but also can be defined by the decision that will be made based upon it.

2.2 Public Key Infrastructure

Public Key Infrastructures (PKIs) are facilities to manage trust relationships. In order the relying parties to determine whether or not to trust another party, are depending on chains of PKI provided certificates. Except the main hierarchical model type of trust used in PKI there are several different models such as

- Subordinated hierarchy
- Cross-certified mesh
- Hybrid
- Bridge CA
- Trust Lists

(Linn, 2000)

In general, a PKI consists of hardware, software policies and standards in order to manage the creation, administration distribution and revocation of keys and digital certificates. The digital certificates affirm the identity of the certificate subject and bind that identity to the certificate's public key.

- The elements that are included in the PKI include a trusted party, that is called certificate authority (CA). This party acts as the root of trust and is the one that provides services that authenticate the identity of entities.
- Following the PKI chain, a subordinate CA called registration authority is certified by the root CA in order to issue certificates that are permitted by the root CA.
- The certificates requests and issues are stored in a certificate database. The database also revokes certificates.
- The certificate store which resides on a local computer in order to store issued certificated and private keys.

A CA in order to verify the identity of an entity, issues a certificate for that entity which signs with its private key. Its public key is available to all interested parties in a self-signed CA certificate. (Rousse, n.d.)

2.3 Trust Classes

The establishment of trust that we mentioned above, through the reputation or PKI systems is related to the Identity trust. In the computer network world, there are also other trust classes that we need to consider.

According to the Grandison & Sloman's classification (2000), trust classes are consist of the following:

- Provision trust: the relying party's trust in a service or resource provider.
- Access trust: trust in principals for the purpose of accessing resources owned by the relying party.
- Delegation trust: trust in an agent that acts and makes decision on behalf of the relying party.
- Identity trust: trust that an agent is the one that claims.
- Context trust: trust that the relying party believes that all the necessary systems are in place in order to support the given transaction and provide a safety net in the case something goes wrong.

(Audun, Roslan, & Colin, 2007)

In order to be able to develop an application that stands as trusted, we have to look into all the for mentioned trust classes and find ways to secure each one of them.

3. Software Architectures

In the for mentioned trust in computer networks, all the implementations are base on a architectural approach of a centralized system. In that centralized system the relying parties are all connected to a central component that provides the service. That implies that if for a reason that central component ceases to provide the service (trust) the whole system becomes unstable and unable to operate.

The decentralized system has several central components and the relying parties are connected to one of them.

On the contrary, the concept of distributed systems, don't rely on a central component but instead all the relying parties are connected to each other. None of the relying parties is connected to all others directly, but instead at least indirectly.

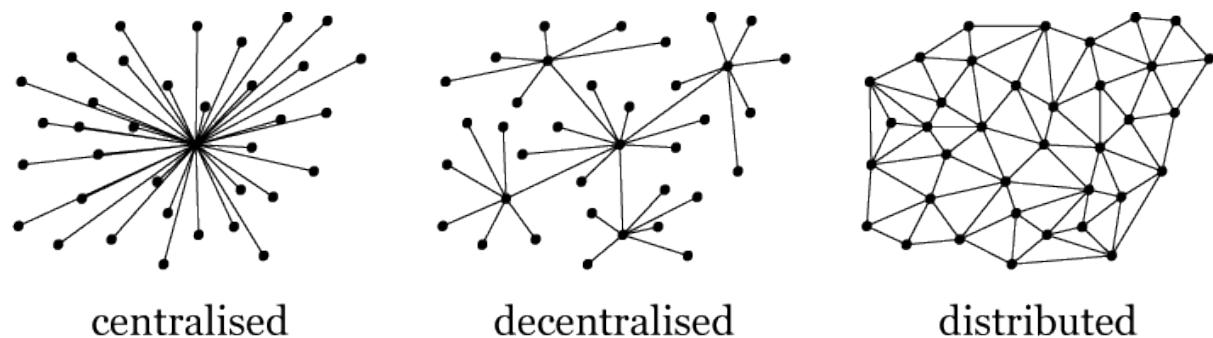


Figure 1: Types of System Software Architectures

There are several pros and cons to the different software architectures but we need to highlight the following

- **Points of failure / Maintenance:** Centralized systems have a single point of failure so its easy to maintain. Decentralized have more that one point of failure but still the number is finite. Distributed systems since have no central point of failure are the most difficult to maintain.
- **Fault tolerance / Stability:** Since centralized systems have one point of failure, a given issue with that point, makes the whole system unstable. In decentralized systems, a failure on a central entity will transform the system to multiple centralized ones. On the contrary, distributed systems are very stable and a single failure doesn't interfere much with the stability of the whole system.
- **Scalability / Max Population:** The centralized systems have low scalability, decentralized moderate and distributed infinite.
- **Ease of development / Creation:** Centralized systems can be created easily, but decentralized and distributed have to first examine details in the way resource sharing will be done and how communications between the entities will be set up.

- **Evolution / Diversity:** Centralized systems, since are based on a single model framework are difficult to evolve and have no diversity. Decentralized and distributed systems despite the difficulty on primary infrastructure, the evolution is tremendous. (Goyal, n.d.)

Decentralization has been used in various fields of our society, like strategy, management and governance. The idea is to distribute control and authorities over various nodes and peripheries, avoiding having one central authority that will have full control. The concept of decentralization, has many similar characteristics with blockchain, as both have no signal central authority to control the whole organization.

Since the Blockchain in its core is a distributed system we will go thoroughly through the concept of the distributed system and then analyze how blockchain can create a decentralized system via consensus mechanisms.

4. Distributed Systems

A distributed system is a collection of independent computers, that appears to its users as a single coherent system. (Tanenbaum & Van Steen, 2007, p. 2)

The points of interest that come out of that definition are that the nodes, or computers of the distributed system are autonomous and the people that interact with the system they have the impression that they are dealing with a single system. In order that perception to be achieved, the nodes need to collaborate with each other. An important remark is that no assumption is being made regarding the type of nodes. That means that the participating nodes can vary from large mainframes to small sensors distributed on the network.

Some of the important characteristics of the distributed networks are, as they formulate by the very definition of the network, the ability to hide from users the differences between the nodes, the consistent and uniform way the users interact with the system no matter where and when that interaction takes place, and the relatively easy way to expand and scale.

The ability of the distributed system to scale derives from the fact that participating nodes can be different computers, working as part of the system without their presence to be shown to the holistic view of the network.

Another characteristic is that is available continuously, even if some parts of the system may be out of order. As result, the modification, replacement or addition of new nodes are not noticed by the end users.

In order to achieve the seamlessly collaboration of the different nodes within the system, and at the same time achieving a single system view, most of the times the distributed systems rely on a software layer that is placed between a higher application layer and a layer that consists of the various operating systems.

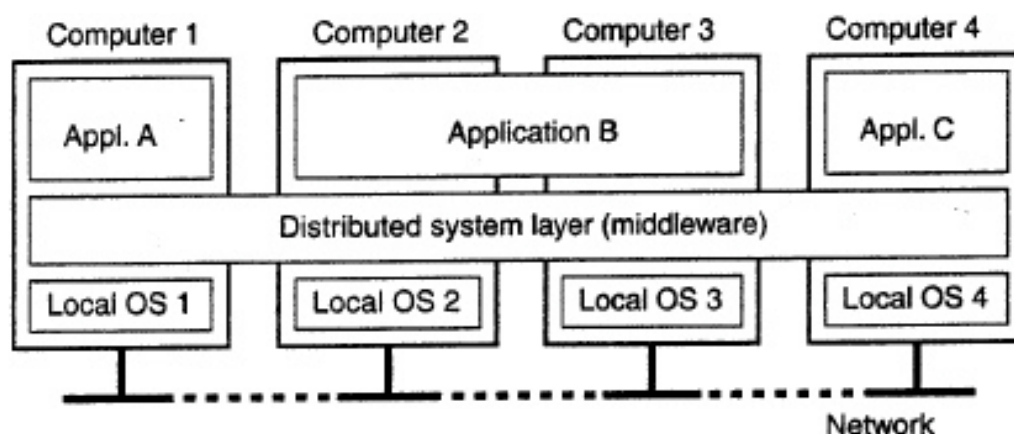


Figure 2: Distributed System Unification of Operating Systems and Computers

At a more compact definition we can describe a distributed system as a computing paradigm, where two or more nodes coordinate and work with each other in order to achieve a common outcome. The distributed system is modeled in a way that the end users perceive it as a single logical platform. (Bashir, 2017, p. 64)

As the different nodes in the system actually act as an individual player and all nodes can send and receive messages to and from each other, that doesn't mean that the behavior of the nodes can be always honest. The nodes, can have honest, faulty or even malicious behavior.

The main challenge in a distributed system is no matter the behavior of the nodes, the system should be able to tolerate this and continue to work flawlessly.

4.1 CAP Theorem

The complexity of designing a distributed system is very challenging. The CAP theorem, proves and states that any distributed system cannot guarantee at the same time Consistency, Availability, Partition Tolerance. There must be made trade-offs in order to achieve the desired level of performance and availability required for a specific task.

- **Consistency:** is the property that ensures that all nodes within a distributed system have a single latest copy of data. In order to achieve consistency consensus algorithms are used.
- **Availability:** is the property that indicates that the system is functioning and is accessible for use, receiving and sending data as and when required
- **Partition Tolerance:** is the property that ensures that if a failure occurs in a distributed system and part of the nodes are not functioning, the system will continue to work correctly. In order to achieve fault tolerance, the most common and widely used method is replication.

If Consistency and availability are important, the partition of the data is not possible. If Consistency is not important, we can partition the data to achieve high availability. If the data need to be partitioned and also Consistency is important then there is no high availability. (Veen, 2015)

4.2 Byzantine Generals

Since the participating nodes act on their own and the distributed system has no hierarchical structure where a leading node instructs the rest, and the behavior of the node can vary from being honest, the distributed systems face with the Byzantine Generals problem.

The Byzantine Generals problem, is defined by the following scenario. A number of generals command different parts of the Byzantine army and want to attack an enemy city. In order to achieve that, and to be successful, they must have a common plan, agree on that and execute it at the same time. The communication between the generals is made solely with messengers.

The problems in that scenario are that on more generals can be a traitor and can communicate a misleading message. That message can cause confusion to the army. In order to solve that, a viable mechanism has to be found, to allow agreement between the generals so that the attack will occur at the same time. The mechanism will have to be able to achieve the desired result even in the presence of a malicious General.

In a real world comparison the Generals could be processors, the traitors faulty processors or system components including system software, and the messengers the communication/system data bus. Compared to the distributed system, Generals could be nodes behaving honestly, traitors could be malicious nodes, and the messenger as a channel of communication between the nodes.

The problem with the generals, was solved in 1999 by presenting the Practical Byzantine Fault Tolerance Algorithm (Castro & Liskov, 1999)

The algorithm states that in order to achieve agreement between the loyal generals in the presence of m traitors there must be:

- At least **$3m+1$** generals to deal with m faults
- Each general must be connected to each other with at least **$2m+1$** communication paths
- **$m + 1$** rounds of messages must be exchanged
- the nodes must be synchronized within a known skew of each other

(University of Houston, n.d.)

The algorithm was used in practice in 2009 as part of the Proof of Work algorithm in Bitcoin, in order to achieve consensus. Other practical implementations of the algorithm have been used by aircraft manufacturers, in flight control systems.

4.3 Consensus

Looking into the for mentioned Byzantine Generals problem, it is easy to understand the importance to achieve consensus in distributed systems. By consensus we mean that the distributed system will reach to a final state of data where all the nodes have agreed upon.

The Consensus is a problem that occurs in distributed systems and for that the use of consensus algorithms has to be applied. The agreement between two nodes is relatively easy to be achieved, like in a server-client model, but when multiple nodes are participating the use of the algorithm is crucial to make sure, that once the value that we need to agree upon is committed, that decision is final and will not be overwritten by future commits.

There are several requirements that must be met so we can reach the desired result of the consensus. Briefly are the following:

- **Agreement:** The same value must be decided by all the honest nodes
- **Termination:** The consensus process is terminated by all honest nodes, and eventually those nodes reach to a decision.
- **Validity:** The value that was agreed by all honest nodes must be the same as the initial value that was proposed by at least one honest node.
- **Fault Tolerant:** The presence of malicious nodes shouldn't interfere with the ability of the consensus algorithm to run.
- **Integrity:** Every node should make the decision only one time in a single consensus cycle.

(Bashir, 2017, p. 72)

The main types of Consensus regarding the mechanism that is being used are the Byzantine fault tolerance-based and the Leader-based.

The Byzantine fault tolerance-based rely on nodes publishing signed messages. The consensus is reached when a specific number of messages have been received.

The leader-based mechanism, rely on the process of electing a leader between the nodes that proposes the final value upon the agreement will be made. The election is achieved after the nodes competing with each other.

Two of the most popular consensus algorithms are Raft and Paxos. Raft is designed to be easy to understand, works by electing a strong leader and having it coordinate with a number of followers. The algorithm assigns any of the three states, Follower, Candidate, Leader to all nodes. When a node receives enough votes becomes a Leader, and all the decisions have to go through him.

Paxos is an older algorithm than Raft and was the first algorithm that had a formal proof of correctness. The nodes in a Paxos system have types as Proposer, Acceptor, Learner. The nodes in Paxos are assigned various roles and the algorithm decides on a single value by a cluster of nodes. Proposers propose values that should be chosen by the consensus. Acceptors form the actual consensus and accept values. Learners learn which value was chosen by each acceptor and therefore the consensus. (Bakhoff)

5. Blockchain Technology

All the DApps we described before rely on the deployment on their very core of Blockchain. Before trying to give the definition of Blockchain we have to understand what a distributed ledger is and how it differs from the traditional notion of the database.

A distributed ledger on its simplest form is a form of database that is held and updated not by a central authority but from each participant (node) in a network. The records of the ledger are independently constructed and held by every node. Every single node processes every transaction, reaching to its own conclusions and then through the consensus process, when the agreement is reached, the ledger has been updated and all nodes keep the identical copy of the ledger. (Bauerle, 2017)

The main difference between a database and a ledger is that in a ledger we can only append new transactions while in a database we have the ability to append, modify and delete records. Although, a database may be used to implement a ledger. (Prusty, 2017, p. 55)

Blockchain is a data structure, used to create a decentralized ledger. The structure of the blockchain consists of blocks connected to each other in a serialized manner. Within a block, a set of transactions is contained, also with a hash of a previous block, a timestamp that indicated the time that the block was created, and more information that will look into further. (Prusty, 2017, p. 56)

In another definition, more compact one, a Blockchain is a timestamped, ordered and immutable list of all transactions. (Bashir, 2017, p. 231)

5.1 Block Structure

Since the block is a container data structure, that contains transactions in order to be included to the public ledger, there are several fields that consist it. The block structure, contains the following fields

- **Block size:** a 4 bytes field that contains the size of the block
- **Block Header:** an 80 bytes field that contains all the header fields from below
- **Transaction Counter:** 1 to 9 bytes field that contains the total number of transactions in the block
- **Transactions:** Variable length, contains the transactions recorded in this block

(oreilly, n.d.)

5.2 Block Header

The header block consists of three sets of block metadata. The first set contains a reference to a previous block hash, and in total all those connection to the previous block form the blockchain. The second set is related to the mining competition and the third set is the merkle tree root. The merkle tree is a data structure to summarize all the transactions in the block.

More in depth, the block header consists of the following fields

- **Version:** a 4 byte field which indicated the block version number to track software / protocol upgrades
- **Previous Block Hash:** a 32 bytes field, a reference to the hash of the previous block in the blockchain
- **Merkle Root:** a 32 bytes field, containing the hash of the merkle tree of all the transactions included in the specific block.
- **Timestamp:** a 4 byte field, that shows the approximate creation time of this block in the Unix epoch time format.
- **Difficulty Target:** a 4 byte field, indicating the difficulty set for this block in the proof of work algorithm.
- **Nonce:** a 4 bytes field that is a number that is changed from miners in order to produce a hash of the block that fulfills the difficulty target that is set.

(oreilly, n.d.)

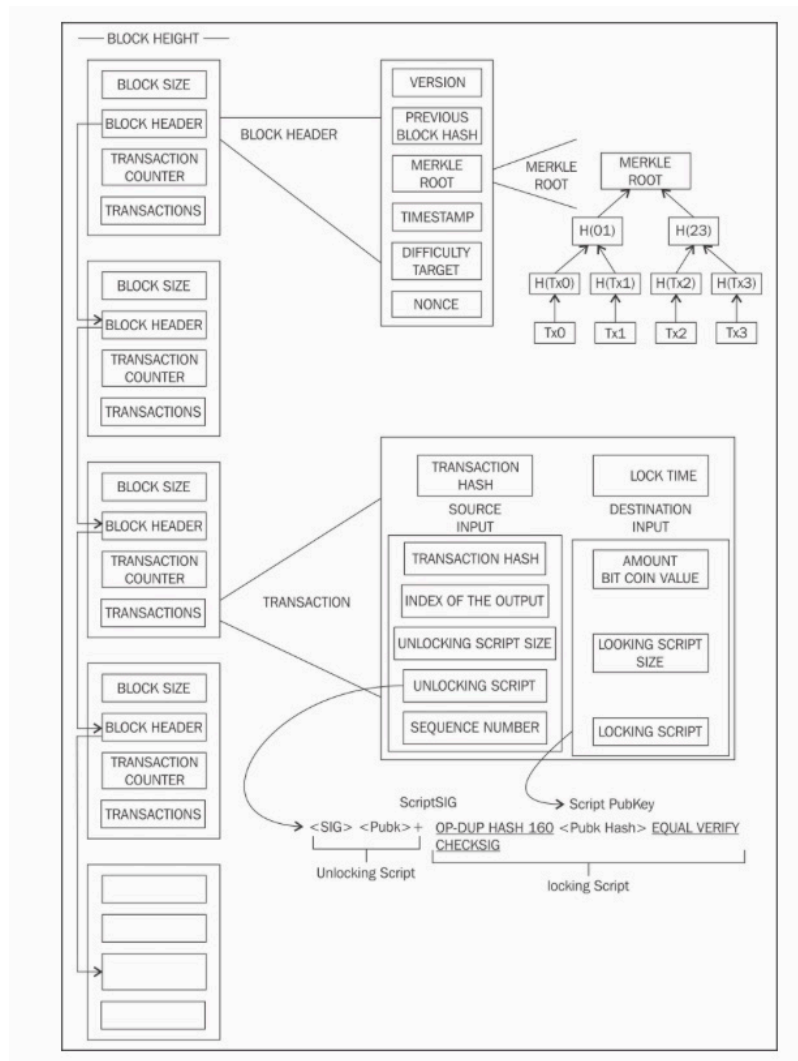


Figure 3: Visualization of Blockchain Structure

In order to understand the complete structure of the blocks that form the blockchain it is better to visualize it. At the figure 5, we see that every block is connected to each other with a reference to the hash of the previous block, that is stored within the block header. That chain connection of one block to its ancestor, forms the chain.

Instead of keeping all the transactions within the block header, in order to be more sufficient, each block contains the merkle root hash of its transactions. A merkle tree is a data structure, that is used for efficiently summarizing and verifying the integrity of large sets of data. In reality a merkle tree is a binary tree that contains cryptographic hashes. The leaves of the tree get hashed, and the parent of the leaves actually is a hash of the hash of the two leaves. This process repeats till we reach the root of the merkle tree, where it's a hash of all the nodes that are contained in the tree. In Blockchain, each node of the tree represents the hash of a transaction.

(oreilly, n.d.)

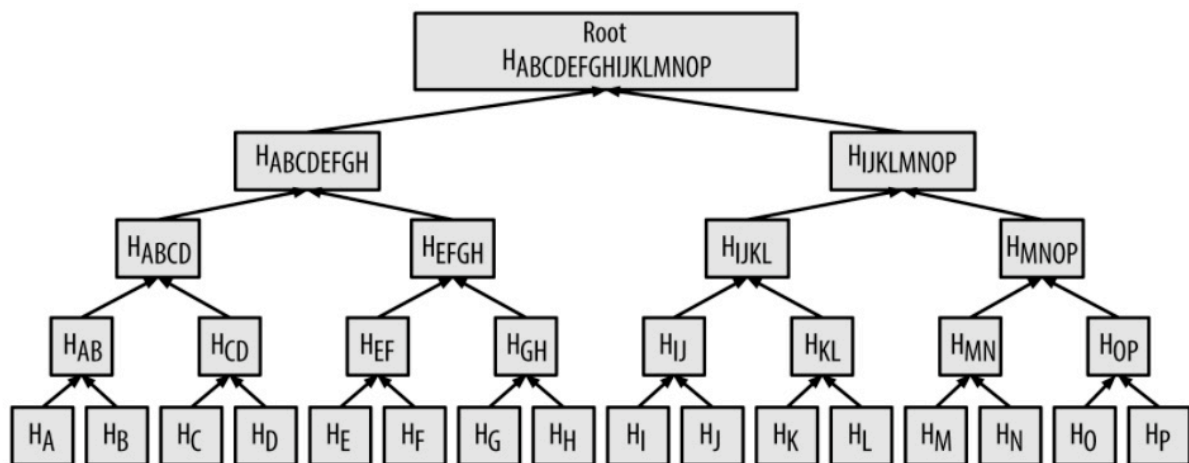


Figure 4: Merkle tree

5.3 The Genesis Block

Since the Blockchain consists of blocks referencing to the previous one, the genesis block is the first block of the blockchain. The assigned number to it is block number 0. It is the only block within the blockchain that doesn't reference to a previous one, because no previous exist. For various networks, the genesis block is hardcoded in the client. Two nodes in the same network will only pair with each other as long as they have the same genesis block, otherwise they will reject each other.

```
{
  "hash" : "0000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
  "confirmations" : 308321,
  "size" : 285,
  "height" : 0,
  "version" : 1,
  "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
  "tx" : [
    "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
  ],
  "time" : 1231006505,
  "nonce" : 2083236893,
  "bits" : "1d00ffff",
  "difficulty" : 1.00000000,
  "nextblockhash" : "00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"
}
```

Figure 5: Bitcoin Genesis Block

5.4 Mining

In order to add a new block to the Blockchain, and create distributed trustless consensus and at the same time solve the double spend problem, the solution of mining was introduced.

Mining is a resource-intensive process by which new blocks are added to the blockchain. The transactions that are contained within the blocks, get validated via the mining process by the mining nodes. In order to make sure that the required resources have been spent by the miners to add a block to the blockchain, the whole procedure is very resource-intensive. One block is created every 10 minutes, and the miners are rewarded with new coins (Bitcoin) if and when they add a new block to the chain, as well as transaction fees so they will include the transaction in the blocks. (Bashir, 2017, p. 237)

5.5 Bitcoin Proof of Work

In Bitcoin, to add block to the blockchain, each node has to show that has performed an amount of work. The node has to find a hash value that is less than a certain number, that is the difficulty target that is set by the network. The difficulty is dynamically set by Bitcoin in order to stabilize the production of one block in ten minutes.

The first node to find a winning hash, adds its proposed block to the Blockchain and collects the reward. Sometimes, it happens that more than one nodes reach to the winning hash at the same time and each node adds its block and broadcasts it to the network. In such an

occasion a fork has been created and the chain continues to build by adding blocks to these branches by nodes that are closer to the winning node. The protocol though ensures that the branch with the maximum PoW will be included in the blockchain and the rest will be discarded. (Baliga, 2017)

The mining algorithm in Bitcoin Proof of Work consists of the following steps:

1. The previous hash block is retrieved from the bitcoin network.
2. Contain a set of transactions that are broadcasted on the network into a block.
3. Pick a nonce and compute the double hash of the header block and the previous hash using SHA256.
4. If the hash is lower than the difficulty target stop the process.
5. If the hash is greater, repeat the process by incrementing the nonce.

Due to the fact, that the mining difficulty has increased and using CPU is no longer fit, miners started using other devices that produce more cryptographic power. Such devices include GPUs, FPGAs that is field programmable gate array, an integrated circuit programmed to perform specific operations and ASICs, application specific integrated circuits, to perform SHA256 operation. Due to the increased difficulty, even the ASICs are not profitable any more. That led to the creation of mining pools, thus pools that miners contribute their cryptographic power and split the awards depending on the agreement they had made in order to join the pool.

5.6 Ethereum Proof of Work (Homestead)

Ethereum in the current version that is called Homestead, it uses its own proof of work model, that is called EthHash. The reason that EthHash was designed was to counter mining centralization. In Bitcoin proof of work, organizations and companies that have the power, were able to build enormous pools to mine bitcoin, using dedicated equipment like the ones we mentioned. Especially using ASICs.

Ethereum utilizes two different techniques to counter that. The first is called memory hardness and is defined as the ability of a computer to move data around memory instead of performing calculations. This kind of process it performing well within the computer hardware, but is not working using ASICs. In that way the algorithm is ASIC resistant. The second technique is called GHOST and it is related to the recently orphaned blocks called uncles. So the nodes that produced an orphaned block, that was included in a temporary fork, still get a smaller reward even though that the block wasn't included in the blockchain as an encouragement to continue mining. (Baliga, 2017, p. 7)

5.7 Proof of Storage

As an alternative to proof of work, Microsoft Research introduced a model called proof of storage. In this model the miners are required to store a pseudo randomly selected subset of large data in order to perform mining.

5.8 Ethereum Proof of Stake (Serenity)

In proof of stake the main idea is that the miners are required to demonstrate possession of a certain amount of currency. In that way they prove that have a stake in the coin. In that way, mining is easier for users that have a large amount of currency.

With proof of stake, we overcome the high demand in electricity that PoW requires. So, for example a user that if was applying a PoW model would have to spend an amount of \$3000 to buy specific mining equipment, now by applying PoS he simply would have to buy \$3000 value of cryptocurrency and use that as stake to buy proportional block creation chances in the blockchain and become a validator.

Due to the fact that the selection of the validators is pseudo-random, no validator knows that he is going to be selected prior to his selection. The most common problem in PoS, is the Nothing-at-Stake problem. In PoW, a node in order to vote on multiple forks, would have to split resources and mine towards different forks. In PoS since that doesn't apply, nothing stops a node to vote at multiple forks in order to increase its chances to win the reward. Thus, the malicious node would try to mine on multiple forks using the same coin.

There are some different forms of stake such as proof of coinage, proof of deposit, proof of burn and proof of activity. In each one forms of PoS, different characteristics of PoS are taken into advantage. In proof of coinage, the miner is rewarding for holding and not spending the coins. In proof of deposit, the coins are somewhat locked and unable to be spent for a certain period of time. Thus the miners perform mining at the cost of freezing some coins for some time. In proof of burn, the miners get rewarded for destroying a number of bitcoins in order to get altcoins. In proof of activity, there is a hybrid state between PoW and PoS. The initial mining is based on PoW, and then the mining of the block is moved to PoS, where randomly three stakeholders are assigned to the block in order to digitally sign it. (Prusty, 2017, p. 289)

Ethereum version Serenity, will adapt the PoS algorithm and more precisely, Proof of deposit. It is called Casper and is considered to be the most advanced PoS algorithm. Casper addresses the Nothing-at-Stake problem in the following way. If a validator's block is included in the chain, the block reward for the validator is equal to the total ether within the active validator set. If the block is not included, then the validator loses a security deposit equal to the block reward. In this way a malicious validator, will not try to vote in multiple forks because he will lose that security deposit from the every fork he participated. (Baliga, 2017, p. 8)

5.9 Intel SawtoothLake Proof of Elapsed Time

Intel SawtoothLake is a blockchain platform that is developed by Intel, and its consensus algorithm is designed to work on a trusted execution environment (TEE) such an Intel's Software Guard Extensions (SGX).

The protocol randomly selects a leader in order to finalize the block. The trusted execution environment is responsible to guarantee the procedure of randomly electing a leader, from

within all participating nodes and at the same time to secure the way for other nodes to verify that the leader was elected in a non-manipulative way.

The way the election works is that every node has to run TEE using Intel GSX. Then each validator request a wait time from the code within the GSX. The leader is the validator that has the smallest wait time of all. (Baliga, 2017, p. 8)

There are more algorithms based on the Byzantine Fault Tolerance and its variations that are used by different blockchain platforms such as the HyperLedger Fabric, mainly focused on consortiums where participants are known and centrally registered and verified, and Ripple and Stellar, platforms that are targeted to financial use cases and payments.

6. Smart Contracts

The concept of smart contracts is not new, but the high impact that have on their deployment on the blockchain in means of reducing cost of transactions and simplifying complex contracts has lead many organizations and financial and academic institutions to research on them. (Bashir, 2017, p. 333)

In 1994 Nick Szabo, realized that smart contracts could be used in a decentralized ledger. Smart contracts are also called digital contracts, blockchain contracts, self-executing contracts. Smart contract are converted in computer code, and then are stored and replicated on the system while the network of nodes that participated in the blockchain supervises them. (Buterin, 2017)

The smart contracts were included also in Bitcoin but in very limited way.

Trying to give a definition to a smart contract, we could say that is a secure and unstoppable program representing an agreement that is automatically executed and enforceable. (Bashir, 2017, p. 335)

In another definition, smart contracts are computer programs that can be consistently executed by a network of mutually distrusting nodes, without the arbitration of a trusted authority. (Bartoletty & Pompianu, 2017, p. 1)

From the definitions of smart contract their main characteristics are derived

- Written in a language that a computer can understand
- Includes agreements between parties in the form of business logic
- Automatically executed when conditions are met
- Enforceable, all contractual terms are executed even if adversaries are present
- Secure and unstoppable, thus designed for fault tolerance and execution in reasonable amount of time.
- Deterministic, thus the smart contract can be run in any node on the network and reach at the same result.

(Bashir, 2017, pp. 336-339)

At the following example of a smart contract, an option contract between two parties is written into the blockchain. The triggering event for a smart contract to be executed could be the reach of an expiration date or a strike price that is hit. At that point where the specified criteria are met and are coded within blockchain, the smart contract executes. The smart contract is distributed into the blockchain network, where the actual event can be shown that took place but at the same time the privacy of the actor participated is maintained.

Smart contract not only define the rules and the penalties that are set by an agreement, but at the same time enforce the obligations that the participating parties have. (Buterin, 2017)

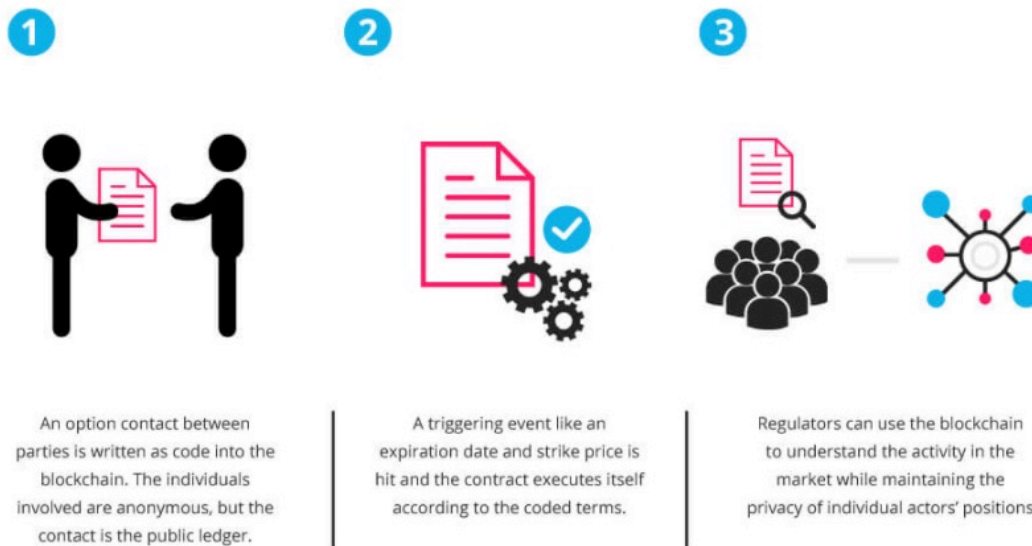


Figure 6: Visual presentation of a Smart Contract

```

/* Allow another contract to spend some tokens in your behalf */
function approve(address _spender, uint256 _value)
    returns (bool success) {
    allowance[msg.sender][_spender] = _value;
    return true;
}

/* Approve and then communicate the approved contract in a single tx */
function approveAndCall(address _spender, uint256 _value, bytes _extraData)
    returns (bool success) {
    tokenRecipient spender = tokenRecipient(_spender);
    if (approve(_spender, _value)) {
        spender.receiveApproval(msg.sender, _value, this, _extraData);
        return true;
    }
}

/* A contract attempts to get the coins */
function transferFrom(address _from, address _to, uint256 _value) returns (bool success) {
    if (balanceOf[_from] < _value) throw; // Check if the sender has enough
    if (balanceOf[_to] + _value < balanceOf[_to]) throw; // Check for overflows
    if (_value > allowance[_from][msg.sender]) throw; // Check allowance
    balanceOf[_from] -= _value; // Subtract from the sender
    balanceOf[_to] += _value; // Add the same to the recipient
    allowance[_from][msg.sender] -= _value;
    Transfer(_from, _to, _value);
    return true;
}

/* This unnamed function is called whenever someone tries to send ether to it */
function () {
    throw; // Prevents accidental sending of ether
}

```

Figure 7: Ethereum Smart Contract example

6.1 Oracles

The main limitation of smart contracts is their lack of communication with the external world in a sense that internal world is specified by the borders of the blockchain. In that way an oracle works as a bridge between the real world (external) and the blockchain (internal) by providing the requested data to the smart contracts. (blockgeeks, 2017)

Oracles is an important component to the smart contract entity. It's the way for smart contracts to have the ability to access external data that might be required to fulfill the business logic that the smart contract describes.

Depending on the type of industry that the oracle is deployed, the data that can be supplied to the smart contracts vary from stock markets to sports data to weather data, even to data coming from Internet of Things devices.

The oracles are digitally signing the data and in that way prove that the data are authentic. The smart contract then, can pull the data from the oracles or the oracles can push the data to the smart contracts. In order though to be able to be sure that the data an oracle is providing is not manipulated, it would be a good practice for the developers of the smart contracts to rely on data that come from an oracle that is provided from a large trusted party. (Bashir, 2017, p. 348)

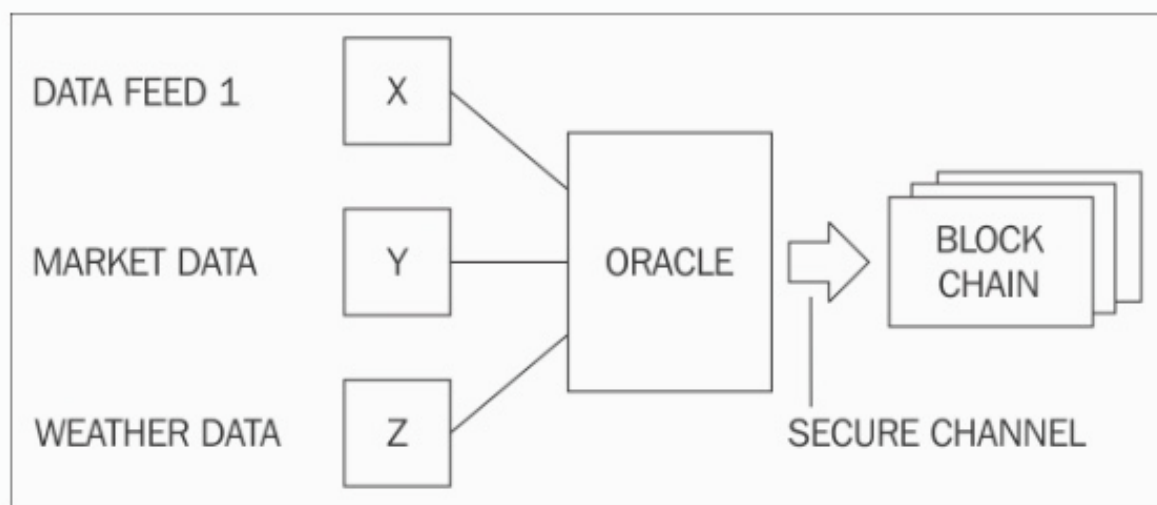


Figure 8: Smart Contract - Oracle

7. Decentralized Applications DApps

The majority of the applications that are Internet based, rely on a centralized architecture. The main relation of the application and the end users, is that the application is hosted to a server which is owned by a company or person and the end users are accessing that server in order to use the application.

This inherits the common issues met with centralized architecture such as less transparency, single point of failure etc. to the application. Due to this concerns, a decentralized point of view is being brought into the designing and implementation of applications. The application that is based on a decentralized system is called a DApp.

The DApp's backend runs on a decentralized peer-to-peer network. Because of that, no node has ever complete control over the DApp. There are many different data structures in order to store the application data depending the DApp and the way it functions.

The DApps also inherit the characteristics of the distributed network that are based on, and thus the developers have to solve the issue, that not every computer – node that is connected to the network will behave honestly. The misuse of the system in terms of altering the application data, or transmitting and sharing wrong information to others, has to be dealt with the for mentioned consensus mechanism. (Prusty, 2017)

In order for the application to be considered as a DApp the following criteria have to been met:

- The application has to be open-source.
- The application has to operate autonomously, no entity controlling the majority of its tokens.
- The application may adopt its protocol in response to proposed improvements and all changed must be decided by consensus of its users.
- The application's data and records of operation must be cryptographically stored in a public, decentralized blockchain so no central points of failure exist.
- The application must use a cryptographic token which is necessary for access to the application and through the token any contribution will be made to the miners.
- The application must generate tokens according to a standard cryptographic algorithm, acting as proof of the values that nodes contribute to the application.

(Johnston, et al., 2015)

7.1 Classification of DApps

The following classification of the DApps, is based on to the fact whether the DApp uses its own blockchain or they use another DApp's blockchain.

- **TYPE I:** DApps that have their own blockchain. The most famous type I DApps are Bitcoin and Litecoin.
- **TYPE II:** DApps that use the blockchain of a type I DApp. Type II DApps are protocols and have tokens that are necessary for their functionality. An example is the Omni Protocol that was formerly known as Mastercoin. At the Omni protocol the claim is that the existing bitcoin network can be used as a protocol layer and on top of it a new layer will be built with new rules to apply but without changing the foundation. (Willett, Hidskes, Johnston, Gross, & Schneider, 2017)
- **TYPE III:** DApps that use the protocol of a type II DApp. At the same philoshop type III DApps are protocols and have tokens that are necessary for their functionality. A type III example is the SAFE Network that uses the Omni protocol in order to issue safecoins that can be then used to acquire distributed file storage. (Lambert, Ma, & Irvine, 2015)

(Johnston, et al., 2015)

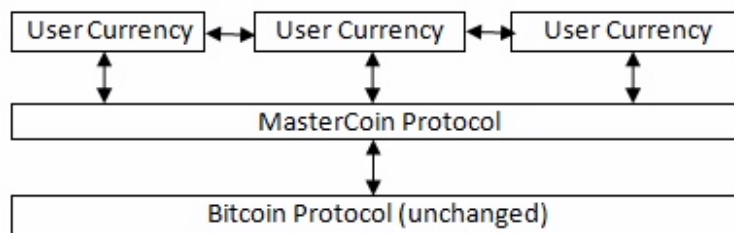


Figure 9: Type II DApp visualization

7.2 Advantages and Disadvantages of DApps

As any kind of applications, the decentralized applications have some pros and cons that we have to be aware of. Some of the advantages are:

- **Fault tolerance:** There is no single point of failure, as they are distributed by default.
- **Prevention of net censorship:** Since there is on central authority on which a government can apply pressure to remove content, net censorship cannot be achieved. Not even the block of the app's domain or IP address can happen, as the DApps are not accessed via a traditional IP address. The shutdown of the nodes, tracked by the node IP address can happen, but since the number of nodes is huge and the nodes are distributed in different countries, shutting down the whole application is very difficult.
- **Elevated Trust:** Since no single authority exists, the users' trust is elevated

Some disadvantages are:

- **Updating / Fixing:** Since every peer in the network has to update/fix the node software, this task is very difficult on DApps.
- **Verification of user identity:** If a verification of the user is required, since there is no central authority to issue that verification, it's difficult to address this issue.
- **Build difficulty:** Due to the complexity of the protocols and the achievement of the desired consensus, building is challenging.
- **Dependencies:** DApps shouldn't depend on a centralized application API, but can be dependent on other DApps.

(Prusty, 2017, pp. 50-52)

7.3 User identity

As mentioned at the disadvantages, the major disadvantage is the difficulty to manage the user identity / verification. In the centralized world the identity of a user can be verified made by submitting a piece of information and the business in charge decides. The term is also known as KYC, know your customer. When moving to the decentralized world, it's critical to address this issue without adding extra complexity. All the participating parties will have to come to an agreement on the accountability of the identification verification criteria, in a trustless way. There is no way yet to accomplish that, most of the directives on KYC are issued by governments.

Such an approach is the e-Resident authentication system that Estonia uses, where a digital ID card created securely digital signatures and proves the identity of the person / holder. Those features are very useful to the deployment of smart contracts that we will see further down. (Beregszaszi, 2016)

The most popular form of a digital identity is the digital certificate, an electronic document used to prove ownership of a public key.

Due to the limitations and difficulties that present with the use of a digital certificate as in form of a digital identity, such as to include the public keys of all the authorities and to be able to update or add new ones, the verification is included in the client side.

The only viable way is to verify the user identity manually by an authorized person of the company that provides the client. (Prusty, 2017, p. 59)

Of course, efforts are being made towards an identity management system as a DApp named , KYC-Chain. At this DApp, Ethereum is employed and works via the use of trusted gatekeepers, who can be any individual or legal entity permitted by law to authenticate KYC documents. The trusted gatekeeper, will perform a check on the user's ID through the KYC-chain platform and authenticate them. The files, will be stored on a distributed database system, where can later be retrieved by the trusted gatekeeper, or the user, in order to demonstrate that his ID is genuine. (coindesk, 2017)

Users own the keys to their personal data and identity certificates, so they are the ones that choose what information will be shared, with whom and under what terms. The identity management does not only apply to a natural person but also to legal entities. (KEY-chain, n.d.)

7.4 User Accounts

The most popular way in a DApp to implement a user account is by the use a private-public key pair. The unique identifier of the account is the hash of the public key. In order a user to make a change to the account's data the user needs to sign the change with his/her private key. A critical point to this implementation is the safe storage of the users private keys. In order to store safely the user keys, a software called wallet is used for that purpose. The wallet can be downloaded and installed (software wallet) and the user then can create and store multiple accounts. Various types of wallets have been developed such as software wallets, online wallets, hardware wallets and mobile ones.

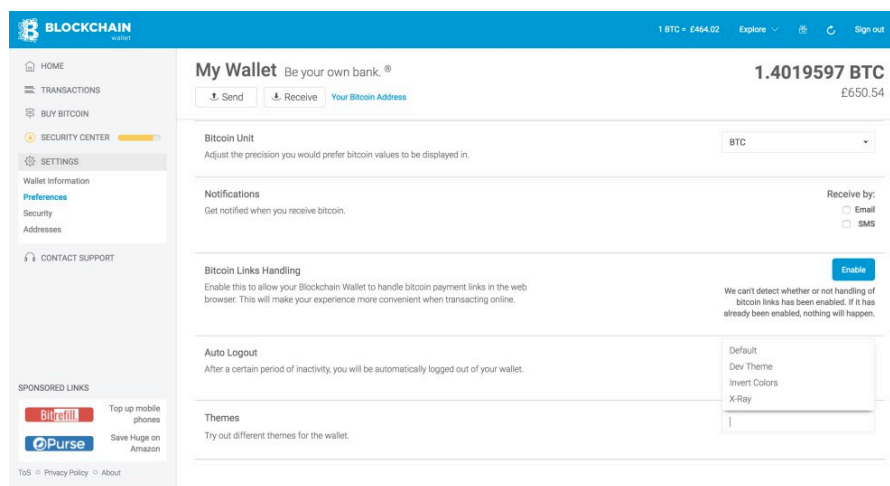


Figure 10: Software Wallet

7.5 Popular DApps

The world of decentralized applications is evolving day by day, as not only individual developers tend to participate, but also big organizations and governments that have the vision to acknowledge the benefits of deploying DApps. Some of the most popular DApps are the following.

7.5.1 Bitcoin

Bitcoin is not just a decentralized currency. It is a protocol, a digital currency and a platform. The combination of peer-to-peer network, protocols and software in order to create and use

the digital currency named bitcoin. To distinguish between the protocol and the currency, the referral to the protocol is with a capital B, while the referral to the currency is with a small b. (Prusty, 2017, p. 285)

7.5.2 Namecoin

Namecoin was the first fork of Bitcoin and was the first to implement merged mining and a decentralized DNS. It solves the Zooko's Triangle problem thus, creating a naming system that is at the same time secure, decentralized and human-meaningful. It is based on same technology with Bitcoin but with its own blockchain and wallet software. The idea is not to provide an altcoin but to provide improved decentralization, censorship resistance, privacy, security and faster naming.

The Name coin securely records and transfer arbitrary names (keys), can attach a value (data) to the names (up to 520 bytes), transacts the digital currency named namecoins (NMC). (namecoin, 2017)

7.5.3 Ethereum

Ethereum is a decentralized platform, that allows to run DApps on top of it. The main advantage of it, is the ability to run smart contracts, thus applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third party interference. We will study Ethereum thoroughly later on. (ethereum, 2017)

7.5.4 The Hyperledger project

Hyperledger is a project to bring blockchain to businesses and to built permissioned DApps. It provides a place for various Blockchain frameworks to be hosted such as:

- **Burrow:** modular blockchain client with a permissioned smart contract interpreter
- **Fabric:** foundation for developing blockchain applications or solutions
- **Iroha:** project to incorporate into infrastructure projects requiring distributed ledger technology
- **Sawtooth:** modular blockchain suite designed for versatility and scalability
- **Indy:** provides, tools, libraries and reusable components for digital identities rooted on blockchains.

(hyperledger, 2017)

7.5.5 Litecoin

Litecoin is a peer-to-peer internet currency that enables instant, near-zero cost payments to anyone. It is a fork of Bitcoin released in 2011, and due to the faster block generation time of 2.5 minutes, it allows faster transactions compared to bitcoin as well as improved storage efficiency. (Litecoin, 2017)

7.5.6 Primecoin

Primecoin is derived from Bitcoin but uses a different proof of Work algorithm, based on the search of prime numbers. This method, not only provides security and minting to the network but also generates a special form of prime number chains that are of high interest in the mathematical research.
(primecoin, 2017)

7.5.7 Zcash

Zcash is a new crypto-currency released in 2016, based on the zero-knowledge proofs known as zero-knowledge Succinct Non-interactive Arguments of knowledge in order to provide complete privacy to the user. In contrary with the bitcoin and the rest crypto-currencies, that expose the payment history to the public, Zcash can fully protect the privacy of transactions. Even though it is based on the Bitcoin core, the addition of the advanced cryptographic techniques, guarantees the validity of the transactions without revealing additional information about them. It actually encrypts the contents of shielded transaction and in order to verify their validity, since they are encrypted, it uses a novel cryptographic method. (z.cash, 2017)

Although the implementation of the Zcash requires that the initial set up will be made by the Zcash team and that means that the organization or person that wants to deploy it, will genuine trust the Zcash team. This has brought controversies to the blockchain community.

7.5.8 Sia

The referral to Sia, is not mostly based on its popularity, but mainly to show the different implementations of blockchain in the everyday tasks that we perform on the Internet. Sia was created to split apart, encrypt and distribute the files of a user across a decentralized network. The user, owns the keys thus owns the data. No outside company has access to the users files, unlike traditional cloud storage providers.

The way it works, is simple, user A has storage that wants to rent to the Sia network, and user B wants to have access to storage to store his/her files. For that, user B pays siacoins to user A as rent for the storage service he provides. (sia, 2017)

8 Ethereum

Ethereum is a decentralized platform that gives us the ability to deploy DApps on top of it, and at the same time allows smart contracts to be run, making the execution to run as programmed without downtime, censorship, fraud or any other type of interference. (ethereum.org, 2017)

Ethereum was created by Vitalik Buterin in 2013, and the key idea behind it was the development of a Turing-complete language, being able to develop smart contracts for blockchain and decentralized applications. (Bashir, 2017, p. 393)

8.1 Ethereum Clients

Using the term client, we are just referring to a deployment of a node that is able to parse, verify the blockchain, its smart contracts and all the information that is related to that. A Client gives the ability to create transactions and also mine blocks.

There are various clients that have been developed for Ethereum using different languages but the most popular between them is the go-Ethereum, known as geth, and Parity.

Client	Language	Developers	Latest release
go-ethereum	Go	Ethereum Foundation	go-ethereum-v1.4.18
Parity	Rust	Ethcore	Parity-v1.4.0
cpp-ethereum	C++	Ethereum Foundation	cpp-ethereum-v1.3.0
pyethapp	Python	Ethereum Foundation	pyethapp-v1.5.0
ethereumjs-lib	Javascript	Ethereum Foundation	ethereumjs-lib-v3.0.0
Ethereum(J)	Java	<ether.camp>	ethereumJ-v1.3.1
ruby-ethereum	Ruby	Jan Xie	ruby-ethereum-v0.9.6
ethereumH	Haskell	BlockApps	no Homestead release yet

Figure 3: List of Ethereum Clients

The Ethereum client actually runs on the nodes and connects to the peer-to-peer Ethereum network and downloads the blockchain and stores it locally. The local copy of the blockchain is regularly synchronized with the network. Web3.js is an Ethereum compatible Javascript API which implements the Generic JSON RPC spec. Thus, it communicates with the local node through RPC calls.

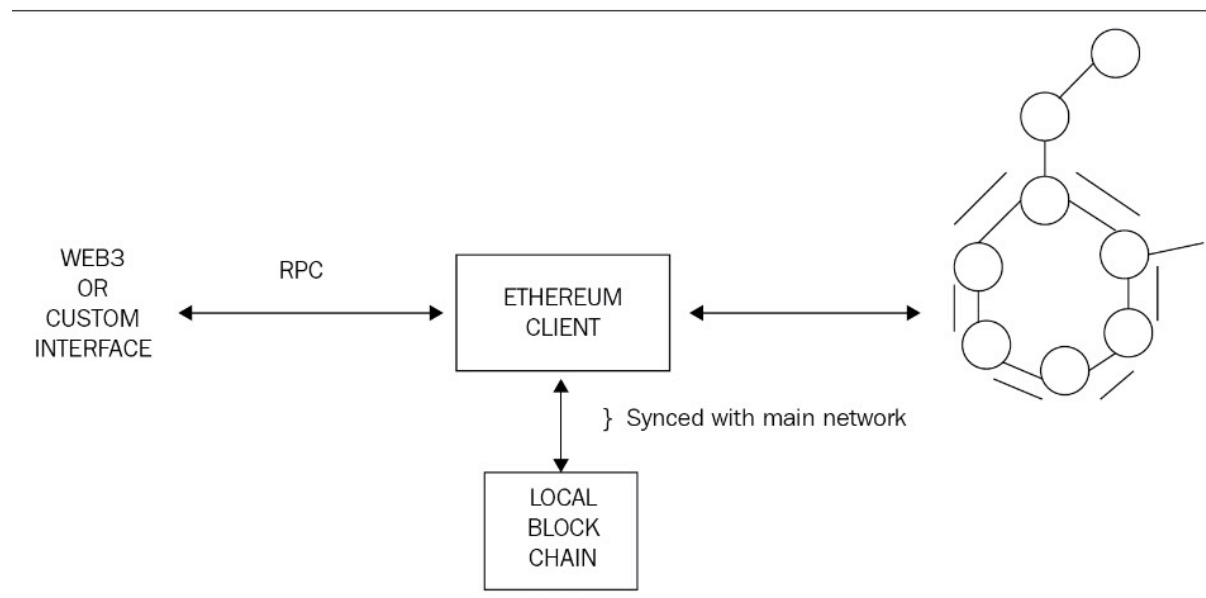


Figure 4: Visual Representation of Ethereum Client Interconnections

8.2 Ethereum Accounts

An Ethereum account, is a simple public/private keypair that is used to sign transactions. Ethereum uses elliptic curve cryptography ECC, specifically using the secp256k1 parameter, thus 256-bit encryption. The public/private key is 256-bit long but in order to be represented by processors, is encoded as a 64-bit HEX string.

A usual misunderstanding is that the actual public key represents the address of the account but this is not the case. Once the public/private keypair has been produced the following procedure takes place so the address can be generated.

1. Generation of the keccak-256 hash of the public key which is a 256-bit number.
2. Drop of the first 96 bits, so that remain 160-bit that is 20 bytes of data.
3. Encode of the address as HEX string. That is the account address

(Prusty, 2017, p. 79)

- 1 Ether = 1000000000000000000 Wei
- 1 Ether = 1000000000000000 Kwei
- 1 Ether = 1000000000000 Mwei
- 1 Ether = 1000000000 Gwei
- 1 Ether = 1000000 Szabo
- 1 Ether = 1000 Finney
- 1 Ether = 0.001 Kether
- 1 Ether = 0.000001 Mether
- 1 Ether = 0.000000001 Gether
- 1 Ether = 0.000000000001 Tether

Figure 6: Denominations of Ether Currency

8.4 Ethereum Transactions

As a transaction in Ethereum is defined the signed data package that transfers ether from one account to another or to a smart contract, in order either to invoke methods of a contract or to deploy a new one. (Prusty, 2017, p. 80)

Depending on the output that is being produced the transactions may be divided in two types, the message call transactions, where a simple message call is produced to pass messages from one account to another, or to contract creation transactions that result to the creation of a new contract. (Bashir, 2017, p. 404)

The message call transaction, contains the following information

- The recipient of the message
- A signature that identifies the sender
- The amount of ether to be transferred
- The gas limit
- The gas price

In case the transaction is a contract creation one, it contains also the initialization code for the contract, or if it invokes a method of a contract, it contains input data. (Prusty, 2017, p. 80)

The following image shows the differences between the two types of transactions

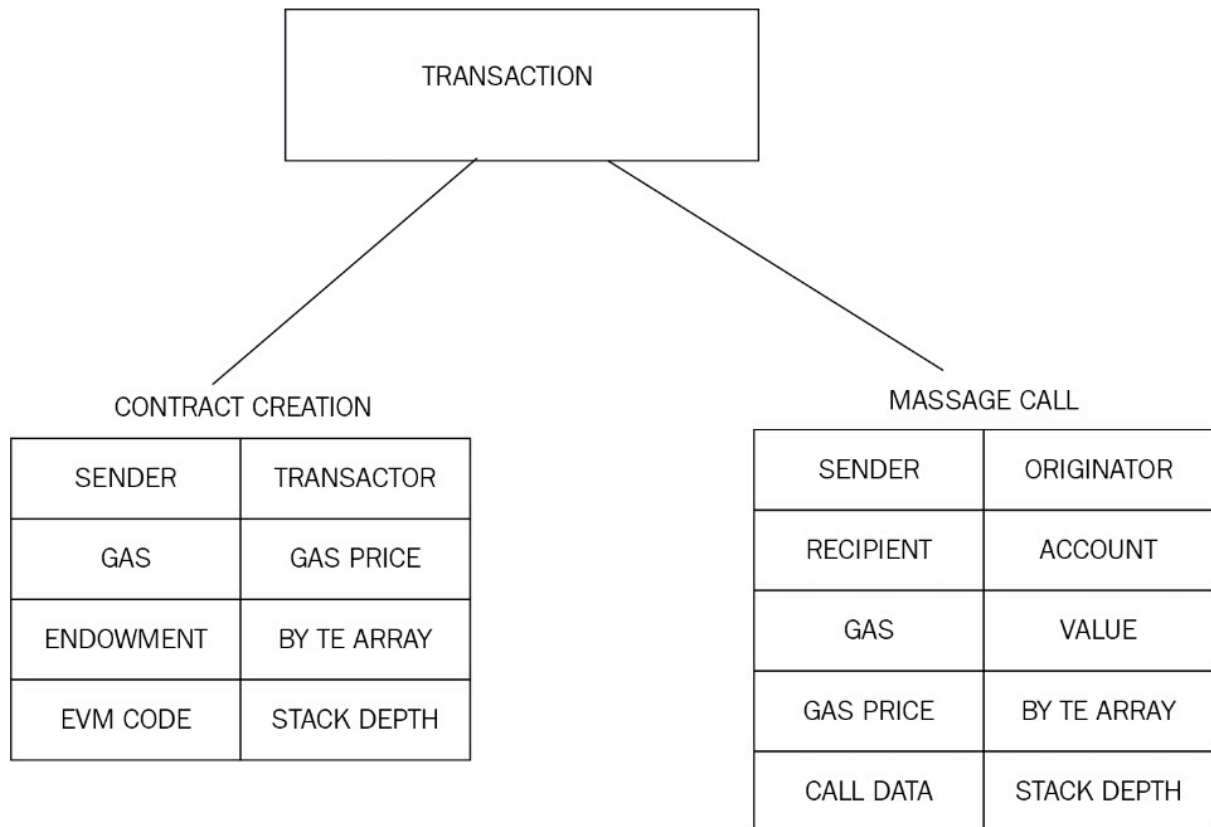


Figure 7: Differences in Fields between Transaction types

8.5 Gas

In Ethereum, the concept of Gas exists in order to define the computational cost that each transaction that takes place on the blockchain requires, so it can be executed.

In the real world imagine that a vehicle wants to move from city A to city B. in order to do that, the owner would have to fill it up with gas and burn the required gas to travel from the one city to another. In addition, in order the road to be in fair condition there are people that work to maintain the road network. So driver has to pay specific tolls for travelling from city A to B.

In a similar way, in the blockchain network, the node must have an Ether balance, and pay the gas price that is required in order to execute the transaction. Gas is considered to be a unit for computational steps. The gas limit is the maximum amount of gas that can be consumed in order for the transaction to be executed. The miners in order to include that transaction they collect the fee.

8.6 Ethereum Consensus

As for mentioned, Ethereum in its current version uses proof-of-work mechanism in order to achieve consensus within the network. The consensus mechanism is based on the Greedy

Heaviest Observed Subtree GHOST protocol, utilizing a simpler version of it. So in contrary with the longest chain in Bitcoin, Ethereum utilizes the heaviest chain.

In Ethereum everybody can become a miner, try to solve the puzzle and as result may be awarded with 5 Ether and the amount of the transaction gas fees that are included in the specific block.

In order to understand better the way that consensus is achieved, we will follow the procedure that takes place by the miners.

A miner collects the new unmined transactions broadcasted to it and determines which of the transactions are valid and which are not. This is done by checking if the transaction is properly signed with the private key, the account the wants to execute the transaction to have sufficient funds (enough balance). With the collected valid transactions the miner creates a block, which is comprised of a header an the content. As content is the list of the collected valid transactions. We have for mentioned the fields that the header contains but in order to keep reading consistency, the header has the hash of the previous block, the block number, the nonce, target, timestamp, difficulty, the address of the miner etc.

The miner changes the nonce in order to find the solution to the puzzle. The key to winning the puzzle is to find the proper nonce that when the block is hashed, the result hash is less or equal to the target has been set. The lower the target the more time is needed to find the proper nonce, the higher the target the less time is needed.

When a miner finds the proper nonce, broadcasts the block across the network so each node will validate it and finally add their own copy to the ledger. If two miners A and B have been on the same block, and miner B finds the hash, then miner A stops working on the block and repeats the process for the next block repeating the whole procedure.

Every block gets find by a miner approximately in every 12-15 seconds. If there is a deviation from this time, meaning the miners either solve the puzzles faster or slower then the algorithm automatically adjust the difficulty of the problem in order to bring back the miners to a 12 second time frame.

Sometimes more than one miners reach to the mining of the block at the same time. There is no issue whether the blocks are valid or not, but the network cannot accept two blocks with the same number, nor can reward two miners for mining the same block. As solution, in the end the blockchain that has the higher difficulty will be accepted by the network. The valid blocks that are finally left out, are called stale blocks.

8.7 Ethereum Virtual Machine EVM

Every node on the Ethereum network runs the Ethereum Virtual Machine EVM. EVM is a simple stack based execution machine that runs bytecode instructions to change state. Stack size is limited to 1024 elements, the word size is 256-bits and is based on LIFO queue.

It is a Turing-complete machine and due to the fact that is limited by the amount of gas that is required to execute a transaction, attacks such as DoS are not possible. It is fully isolated and sandboxed and the code that runs in EVM has no connection to the filesystem or the network.

The program code is stored in a virtual read only memory VROM, and is accessible using the CODECOPY instruction. That instruction is used to copy the program code to the main memory which is read by the EVM and then starts to execute the instructions step by step. After each instruction execution the program counter and the EVM stack are updated (Bashir, 2017, p. 414)

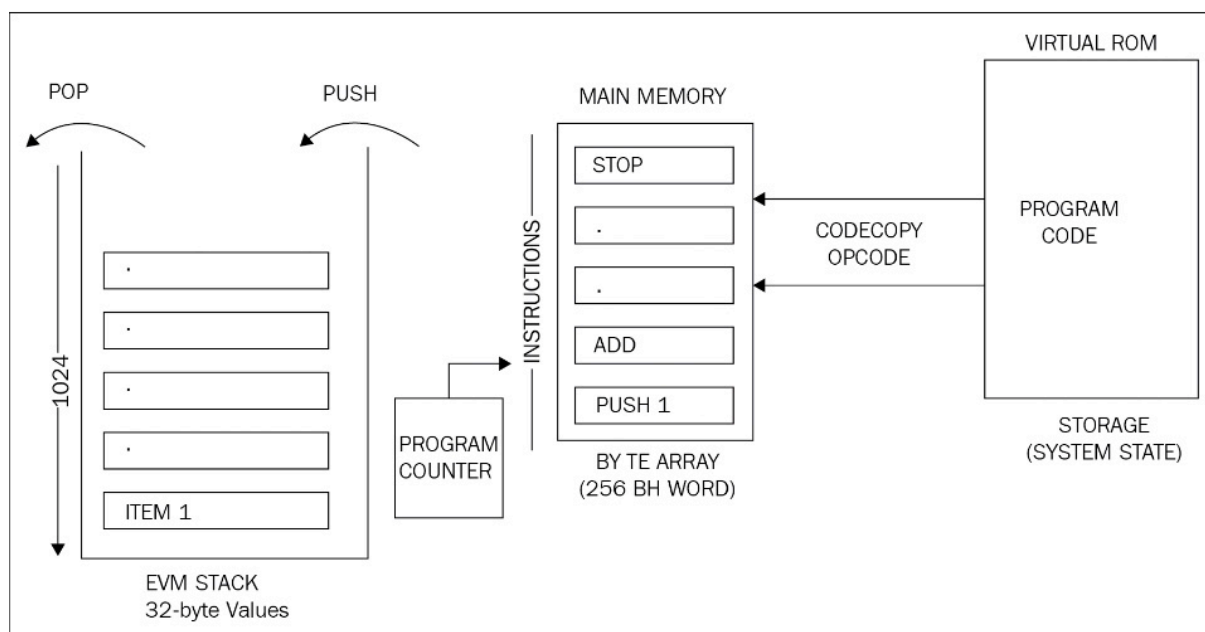


Figure 8: EVM operation

The future development of EVM is headed to the ability to run the EVM instruction set codes natively in the CPUs making it way faster and efficient. At the same time, the research on utilizing web assembly is ongoing, thus gives the ability to run the machine code in the browser and result in a better speed benchmarks.

8.8 Ethereum Block

In Ethereum the block consists of the transactions list, the list of the headers of Ommers or Uncles and the block header (stale blocks)

The Ethereum block header consists of the following:

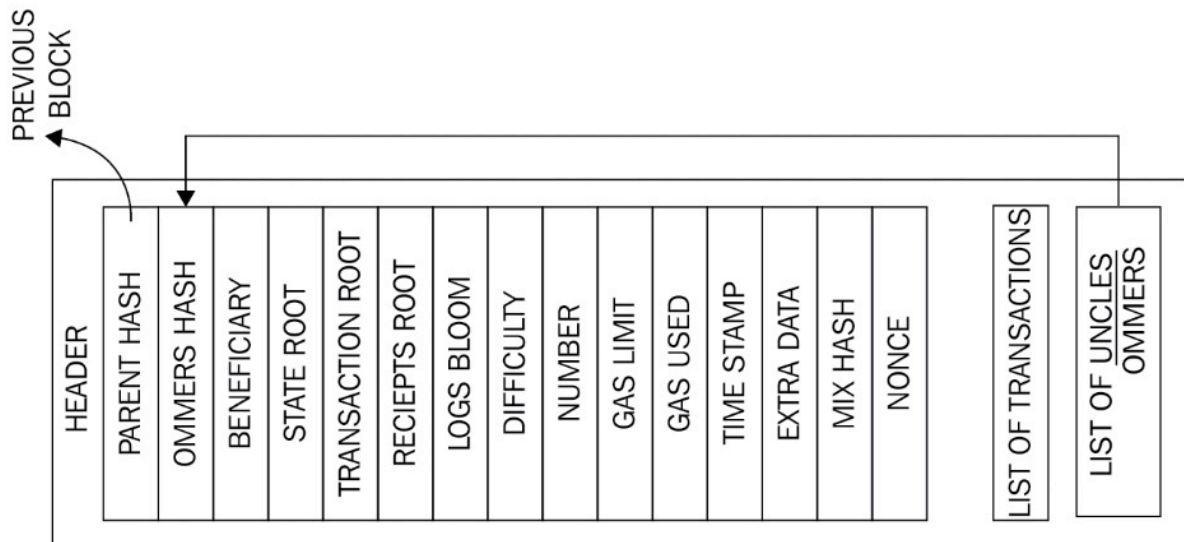


Figure 9: Ethereum Block Fields

- Parent hash is the hash value (Keccak-sha3-256-bit) of the previous block header.
- Ommers hash is the hash value (Keccak-sha3-256-bit) of the list of the stale blocks that are included in the block.
- Beneficiary is the address of the recipient that will receive the mining reward (miner).
- State root is the Keccak-sha3-256-bit hash of the root node of the state trie.
- Transaction root is the Keccak-sha3-256-bit of the root node of the transaction trie, thus the list of all the transactions that are included in the block.
- Receipt root the Keccak-sha3-256-bit of the root node of the receipt trie, thus the list of all the receipts of the transactions in this block.
- Logs Bloom is a bloom filter that contains the logger address and the log topics of the log entry of each transaction receipt in the block.
- Difficulty is the difficulty set on the block.
- Number is the total number of the previous blocks.
- Gas limit is the value on the gas consumption per block.
- Gas used is the value that represents the total amount of gas used by the transactions on this block.
- Timestamp is the epoch Unix time of the time of the block initialization.
- Extra data is a field that can be used to store data related to the block.
- Mixhash is a 256-bit hash that when is combined with the nonce is used as proof of the computational effort that has been spent to create the block.
- Nonce is the 64-bit hash that used with the mixhash is used as a proof of the computational effort that has been spent to create the block.

(Bashir, 2017, p. 449)

The transaction receipts that are mentioned is a mechanism that is used to store the state of a transaction after the execution of it. The receipts are stored in a index-keyed trie and the hash of the root of the trie is placed in the block header.

In order to validate a Block certain checks have to be met such as:

- Check if all Ommers are indeed Uncles and also if proof-of-work for uncles is valid
- Check if the previous block exists and it is valid
- Check that the timestamp of the block is valid. Meaning that is higher than the parent's timestamp

If any of these conditions fail then the block is considered not to be valid.

The final step is to finalize the block. The process is being made by the miners in order to validate the contents of the block and thus to apply the rewards. There are four steps that need to be executed and these are:

- Validation of the Ommers.
- Validation of the transactions.
- Reward application.
- Validation of state and nonce.

8.9 Ethereum Network – Peer discovery

In the Ethereum network, all nodes are connected to each other but no every node is connected to all. A node is connected to few other nodes, which in turn connect to few other nodes leading to a network where eventually all nodes are connected.

Depending on the usage and the purpose of the creation of the network and also on the characteristics of it, we are able to divide it in three different types.

Those types are, the main network, which is the current live network of Ethereum and its public, meaning that there are no restrictions as to whom is able to join.

The test net, which as described by its name is a test network of the Ethereum blockchain. It is used to deploy and test smart contracts and DApps before these are deployed to the main network.

There are some cases though, that for specific reasons, a group of entities want to have a controlled network environment and for that there is the private nets.

Ethereum has its own discovery protocol. The nodes are gossiping with each other to find about other nodes on the network. There are special nodes, called Bootstrap nodes that can be set in the source code that maintain a list of all nodes connected to them. When a new node connects, it first communicates with the Bootstrap nodes in order to connect and synchronize. (Github-Ethereum, 2017)

An example in geth how to set up on start up which nodes will be Bootstrap nodes is the following:

```
geth --bootnodes  
enode://pubkey1@ip1:port1,enode://pubkey2@ip2:port2,enode://pubkey3@ip3:port3
```

We can see that each node is described by its public key and the actual ip and port on the network. In order to start the network without the default discovery protocol, geth must be run with the `-nodiscover` parameter. In this way, when on a private network, the nodes that are connected are only the nodes that explicitly get connected manually to a specific node. For example, if the private network consists of nodes A, B, C, D and node B is set as Bootstrap node we can add a new node by manually telling it to connect to node B. As soon as this will be done, the node list that node B holds will be shared with new node E and it will get synched with the network.

9. Private Network Setup

In this section we will try to setup a private network that consists of 3 nodes, Testnode1, Testnode2, Testnode3, we will see the communication between them and also we will send Ethereum from one node to the other. Geth client is going to be used as an implementation of the Ethereum nodes.

The setup of the private network is going to be on Ubuntu 16.04 VMs. In order to set up the private network, there are 3 components that are needed.

- The private network ID
- The Genesis file
- The data directory that will be used to store blockchain data.

9.1 Geth Install

In order to install geth on Ubuntu we have to run the following commands

```
angular@blockchain3:~$ sudo apt-get install software-properties-common
```

```
angular@blockchain3:~$ sudo add-apt-repository -y ppa:ethereum/ethereum
```

```
angular@blockchain3:~$ sudo apt-get update
```

```
angular@blockchain3:~$ sudo apt-get install ethereum
```

In order to verify the installation of geth we issue the following command and see the outcome.

```
angular@blockchain3:~$ which geth
/usr/bin/geth
angular@blockchain3:~$
```

We give the command `geth account new` to create an account and give the password for it.

9.2 Private Data Directory

The Ethereum is going to save the blockchain data in a specific directory that we are going to issue this as a parameter later at the initialization of the network. We create a directory as `~/ethereum/privatenet`

9.3 Genesis File

In the Genesis file, all the necessary information regarding the setup of the private network is included. We will use the following genesis file which will be placed in a directory `~/private_ether/`

The Genesis file we are going to use is the following

```
{
  "config": {
    "chainId": 15,
    "homesteadBlock": 0,
    "eip155Block": 0,
    "eip158Block": 0
  },
  "nonce": "0x0000000000000042",
  "mixhash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
  "difficulty": "0x200",
  "alloc": {},
  "coinbase": "0x0000000000000000000000000000000000000000",
  "timestamp": "0x00",
  "parentHash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
  "gasLimit": "0xffffffff",
  "alloc": {

}
}
```

9.4 Initialize Network

In order to initialize the private network we issue the following command

```
angular@blockchain3:~$ geth --datadir ~/.ethereum/privatenet/ init ~/.private_ether/privategenesis.json
```

and we expect the following outcome

```
WARN [11-14|09:40:16] No etherbase set and no accounts found as default
INFO [11-14|09:40:16] Allocated cache and file handles      database=/home/angular/.ethereum/privatenet/
geth/chaindata cache=16 handles=16
INFO [11-14|09:40:16] Writing custom genesis block
INFO [11-14|09:40:16] Successfully wrote genesis state      database=chaindata
hash=8da729...3e3e9f
INFO [11-14|09:40:16] Allocated cache and file handles      database=/home/angular/.ethereum/privatenet/
geth/lightchaindata cache=16 handles=16
INFO [11-14|09:40:16] Writing custom genesis block
INFO [11-14|09:40:16] Successfully wrote genesis state      database=lightchaindata
hash=8da729...3e3e9f
```

Next step is to launch geth, having 2 screens running in parallel in order to be able to have both the console and the log.

```
angular@blockchain3:~$ geth --datadir ~/.ethereum/privatenet/ --nodiscover console 2>> eth.log
```

At a new terminal window we issue the command: tail -F ~/.ethereum/privatenet/ eth.log

```
angular@blockchain3:~$ geth --datadir ~/.ethereum/privatenet/ --nodiscover console 2>> eth.log
Welcome to the Geth JavaScript console!

instance: Geth/v1.7.2-stable-1db4ecdc/linux-amd64/go1.9
modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

> █
```

```
angular@blockchain3:~$ tail -F ~/.ethereum/privatenet/ eth.log
==> /home/angular/.ethereum/privatenet/ <==
tail: error reading '/home/angular/.ethereum/privatenet/': Is a directory
tail: /home/angular/.ethereum/privatenet/: cannot follow end of this type of file; giving up on this name

==> eth.log <==
INFO [11-14|09:47:29] Disk storage enabled for ethash DAGs      dir=/home/angular/.ethash          count=2
INFO [11-14|09:47:29] Initialising Ethereum protocol           versions="[63 62]" network=1
INFO [11-14|09:47:29] Loaded most recent local header           number=0 hash=8da729...3e3e9f td=512
INFO [11-14|09:47:29] Loaded most recent local full block       number=0 hash=8da729...3e3e9f td=512
INFO [11-14|09:47:29] Loaded most recent local fast block       number=0 hash=8da729...3e3e9f td=512
INFO [11-14|09:47:29] Regenerated local transaction journal     transactions=0 accounts=0
INFO [11-14|09:47:29] Database deduplication successful         deduped=0
INFO [11-14|09:47:29] Starting P2P networking
INFO [11-14|09:47:29] RLPx listener up                         self="enode://d6f301f18c1c5f60dd0c0491e90e568f7c685e769906c048a8e703542dd127a54a7003d9c
e4672a54aa41e7681912d02fb7e96b6618abb8a72d4f9a072e55b4f@[::]:30303?discport=0"
INFO [11-14|09:47:29] IPC endpoint opened: /home/angular/.ethereum/privatenet/geth.ipc
█
```

We see from the log that the IPC endpoint has been successfully opened.

9.5 Account Creation

Next step is to create an account in the current node. We issue the following commands

```
angular@blockchain3:~$ geth --datadir ~/.ethereum/privatenet/ --nodiscover console 2>> eth.log
Welcome to the Geth JavaScript console!

instance: Geth/v1.7.2-stable-1db4ecdc/linux-amd64/go1.9
modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

> personal.newAccount() █
```

It asks for a passphrase and the account is created.

```
angular@blockchain3:~$ geth --datadir ~/.ethereum/privatenet/ --nodiscover console 2>> eth.log
Welcome to the Geth JavaScript console!

instance: Geth/v1.7.2-stable-1db4ecdc/linux-amd64/go1.9
modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

> personal.newAccount()
Passphrase:
Repeat passphrase:
0xccc75e763a05d70b402ae881eda1f4e19571bc"
> █
```

Since we didn't allocate ether to the node in the genesis file, checking the balance of the account will show 0

```
> eth.getBalance(eth.accounts[0])
0
> █
```

Next step is to set this account as Etherbase/coinbase in order to get the mining reward

```
> miner.setEtherbase(personal.listAccounts[0])
true
>
```

We start the miner in order to get Ether into the account

```
> miner.start()
null
>
```

And we see at the eth.log the generation of the DAG which is a directed acyclic graph for the proof of work algorithm. The DAG is generated for every epoch, thus every 30000 blocks (100 hours). (Github Ethereum, 2017)

```
INFO [11-14|10:27:21] Starting mining operation
INFO [11-14|10:27:21] Commit new mining work
INFO [11-14|10:27:25] Generating DAG in progress
INFO [11-14|10:27:27] Generating DAG in progress
INFO [11-14|10:27:30] Generating DAG in progress
INFO [11-14|10:27:33] Generating DAG in progress
INFO [11-14|10:27:36] Generating DAG in progress
INFO [11-14|10:27:39] Generating DAG in progress
INFO [11-14|10:27:42] Generating DAG in progress
INFO [11-14|10:27:45] Generating DAG in progress
INFO [11-14|10:27:48] Generating DAG in progress
INFO [11-14|10:27:50] Generating DAG in progress
INFO [11-14|10:27:53] Generating DAG in progress
INFO [11-14|10:27:56] Generating DAG in progress
INFO [11-14|10:27:58] Generating DAG in progress
INFO [11-14|10:28:01] Generating DAG in progress
INFO [11-14|10:28:04] Generating DAG in progress
INFO [11-14|10:28:07] Generating DAG in progress
INFO [11-14|10:28:10] Generating DAG in progress
INFO [11-14|10:28:13] Generating DAG in progress
INFO [11-14|10:28:16] Generating DAG in progress
INFO [11-14|10:28:19] Generating DAG in progress
number=1 txs=0 uncles=0 elapsed=834.422µs
epoch=0 percentage=0 elapsed=2.921s
epoch=0 percentage=1 elapsed=5.805s
epoch=0 percentage=2 elapsed=8.770s
epoch=0 percentage=3 elapsed=11.679s
epoch=0 percentage=4 elapsed=14.583s
epoch=0 percentage=5 elapsed=17.483s
epoch=0 percentage=6 elapsed=20.421s
epoch=0 percentage=7 elapsed=23.368s
epoch=0 percentage=8 elapsed=26.067s
epoch=0 percentage=9 elapsed=28.729s
epoch=0 percentage=10 elapsed=31.282s
epoch=0 percentage=11 elapsed=33.952s
epoch=0 percentage=12 elapsed=36.444s
epoch=0 percentage=13 elapsed=39.312s
epoch=0 percentage=14 elapsed=42.428s
epoch=0 percentage=15 elapsed=45.402s
epoch=0 percentage=16 elapsed=48.433s
epoch=0 percentage=17 elapsed=51.448s
epoch=0 percentage=18 elapsed=54.710s
epoch=0 percentage=19 elapsed=57.755s
```

Once the DAG is generated then the mining of the blocks starts

```
INFO [11-14|10:36:01] Successfully sealed new block
INFO [11-14|10:36:01] block reached canonical chain
INFO [11-14|10:36:01] mined potential block
INFO [11-14|10:36:01] Commit new mining work
INFO [11-14|10:36:05] Successfully sealed new block
INFO [11-14|10:36:05] block reached canonical chain
INFO [11-14|10:36:05] mined potential block
INFO [11-14|10:36:05] Commit new mining work
INFO [11-14|10:36:08] Generating DAG in progress
INFO [11-14|10:36:14] Generating DAG in progress
INFO [11-14|10:36:15] Successfully sealed new block
INFO [11-14|10:36:15] block reached canonical chain
INFO [11-14|10:36:15] mined potential block
INFO [11-14|10:36:15] Commit new mining work
INFO [11-14|10:36:16] Successfully sealed new block
number=30 hash=ebdd13...ea1362
number=25 hash=d4fa06...c1a572
number=30 hash=ebdd13...ea1362
number=31 txs=0 uncles=0 elapsed=131.004µs
number=31 hash=31b5b5...25699d
number=26 hash=853cdd...0ccc1e
number=31 hash=31b5b5...25699d
number=32 txs=0 uncles=0 elapsed=177.383µs
epoch=1 percentage=36 elapsed=3m49.394s
epoch=1 percentage=37 elapsed=3m55.702s
number=32 hash=034ed4...a4e904
number=27 hash=f60433...15150e
number=32 hash=034ed4...a4e904
number=33 txs=0 uncles=0 elapsed=99.813µs
number=33 hash=fcddc1...04a9b5
```

Now we stop the miner and check again to see the balance on the account

```
> miner.stop()
true
>
```

We see that the account has mined Ether

```
> eth.getBalance(eth.accounts[0])
1.745e+21
>
```

The procedure so far has to be repeated in every node that we wish to participate in the network.

9.6 Peering

The created node is not connected to any other node since we issued the `--nodiscover` parameter at the Geth start. We can see that by issuing the following command.

```
> admin.peers  
[]  
>
```

If we repeat the procedure so far to every node that we want as part of the private blockchain, we will see that none has member at the `admin.peers`.

In order to connect a node to another we have to get information of the node that will connect to. We have repeated the process and have created two other nodes, node 1 and node 2. In total our network has 3 nodes. We consider node 1 to be our core bootstrap node, so we need to connect node 3 to node 1.

In node 1 we issue the following command

```
> admin.nodeInfo.enode  
"enode://3bfe3f4d6fd98155ef1672bb8a49b1c85940f83cee73df180057edae7383805af3f12579381a25087d9c82d415e3f24aa71b9cf35b2407218f076bfc109542fc@[::]:30303?discport=0"  
>
```

In our network the three nodes have the following IP addresses

- Node 1 : 192.168.235.129
- Node 2: 192.168.235.133
- Node 3: 192.168.235.132

In node 3 we have to issue the following command and in the brackets we have to specify the IP and port that node 1 listens.

```
> admin.addPeer("enode://3bfe3f4d6fd98155ef1672bb8a49b1c85940f83cee73df180057edae7383805af3f12579381a25087d9c82d415e3f24aa71b9cf35b2407218f076bfc109542fc@192.168.235.129:30303?discport=0")  
true  
>
```

We see at `eth.log` the procedure of synchronizing with the network nodes taking place.

```

INFO [11-14|11:08:19] Block synchronisation started
INFO [11-14|11:08:19] Imported new state entries count=4 elapsed=89.947µs processed=4 pending=0 retry=0 duplicate=0 unexpected=0
INFO [11-14|11:08:20] Imported new block headers count=192 elapsed=1.121s number=192 hash=2603cf...07d4c2 ignored=0
INFO [11-14|11:08:20] Imported new block receipts count=2 elapsed=114.156µs bytes=8 number=2 hash=133a98...fb8c38 ignored=0
INFO [11-14|11:08:20] Imported new block headers count=113 elapsed=21.259ms bytes=1466 number=115 hash=85f99d...68fc34 ignored=0
INFO [11-14|11:08:20] Imported new block receipts count=192 elapsed=57.728ms number=384 hash=f76865...6a53ff ignored=0
INFO [11-14|11:08:20] Imported new block headers count=77 elapsed=13.292ms bytes=814 number=192 hash=2603cf...07d4c2 ignored=0
INFO [11-14|11:08:20] Imported new block receipts count=39 elapsed=366.362µs bytes=156 number=231 hash=0c0c2c...d6be7d ignored=0
INFO [11-14|11:08:21] Imported new block receipts count=153 elapsed=1.672ms bytes=1629 number=384 hash=f76865...6a53ff ignored=0
INFO [11-14|11:08:21] Imported new block headers count=192 elapsed=761.664ms number=576 hash=3dafcb...bed42c ignored=0
INFO [11-14|11:08:21] Imported new block receipts count=156 elapsed=2.068ms bytes=624 number=540 hash=55a763...1cf8cc ignored=0
INFO [11-14|11:08:21] Imported new state entries count=3 elapsed=55.351µs processed=7 pending=0 retry=0 duplicate=0 unexpected=0
INFO [11-14|11:08:21] Imported new block receipts count=1 elapsed=54.931µs bytes=4 number=541 hash=835222...8c9567 ignored=0
INFO [11-14|11:08:21] Committed new head block number=541 hash=835222...8c9567
INFO [11-14|11:08:21] Imported new chain segment blocks=35 txs=0 mgas=0.000 elapsed=27.742ms mgasps=0.000 number=576 hash=3dafcb...bed42c
WARN [11-14|11:08:21] Skipping deep transaction reorg depth=227
INFO [11-14|11:08:22] Imported new block headers count=70 elapsed=376.803ms number=646 hash=ad05b8...80e753 ignored=0
INFO [11-14|11:08:22] Imported new chain segment blocks=19 txs=0 mgas=0.000 elapsed=3.646ms mgasps=0.000 number=595 hash=45bffe...650ac
INFO [11-14|11:08:22] Imported new chain segment blocks=51 txs=2 mgas=0.042 elapsed=36.050ms mgasps=1.165 number=646 hash=ad05b8...80e753
INFO [11-14|11:08:22] Fast sync complete, auto disabling

```

Now if we give the command to check the nodes in node 3 we see the following.

```

> admin.peers
[[
  {
    caps: ["eth/62", "eth/63"],
    id: "3bfe3f4d6fd98155ef1672bb8a49b1c85940f83cee73df180057edae7383805af3f12579381a25087d9c82d415e3f24aa71b9cf35b2407218f076bfc109542fc",
    name: "Geth/v1.7.0-stable-6c6c7b2a/linux-amd64/go1.9",
    network: {
      localAddress: "192.168.235.132:46478",
      remoteAddress: "192.168.235.129:30303"
    },
    protocols: {
      eth: {
        difficulty: 90643859,
        head: "0xad05b8a6b69e809351d9ab3da42231269fd1c6d77973427d41e828750880e753",
        version: 63
      }
    }
  }
]]

```

In node 1 we see that admin.peers result in showing both node 2 and node 3

```

> admin.peers
[[
  {
    caps: ["eth/62", "eth/63"],
    id: "cc73c9261087c5d78ee761e2706ffeeae2ba2d9315bf2c95e5c7328309d7cce9013255efa1fc599761a9c366b3ac51d04390241e2a84a9ce93c7b3b663c5d7c1",
    name: "Geth/v1.7.0-stable-6c6c7b2a/linux-amd64/go1.9",
    network: {
      localAddress: "192.168.235.129:30303",
      remoteAddress: "192.168.235.130:49958"
    },
    protocols: {
      eth: {
        difficulty: 90643059,
        head: "0xad05b8a6b69e809351d9ab3da42231269fd1c6d77973427d41e828750880e753",
        version: 63
      }
    }
  }, {
    caps: ["eth/63"],
    id: "d6f301f18c1c5f60dd0c0491e90e568f7c685e769906c048a8e703542dd127a54a7003d9ce4672a54aa41e7681912d02fb7e96b6618abb8a72d4f9a072e55b4f",
    name: "Geth/v1.7.2-stable-1db4ecdc/linux-amd64/go1.9",
    network: {
      localAddress: "192.168.235.129:30303",
      remoteAddress: "192.168.235.132:46478"
    },
    protocols: {
      eth: {
        difficulty: 49379705,
        head: "0x6475e561bb7d8f867d4abf222ad90346dda0699820396d6b2369e1573065c349",
        version: 63
      }
    }
  }
]]
>

```

9.7 Sending Ether

In the following example we are going to send 10 ethers from node 1 to node 3 and from node 3 to node 2.

In node 3 terminal we issue the following command to retrieve the address of the account that we are going to send ether from node 1

```

> eth.accounts
["0xcec75e783a05d70b402ae881edafb1f4e19571bc"]
>

```

and in node 3 we first unlock the account

```

> personal.unlockAccount(eth.coinbase)
Unlock account 0x2ff5990db2abd2eb6d107c717bdb896e9342ea91
Passphrase:
true
>

```

and then we give the command to send ether to the address of node 3

```

> web3.fromWei(web3.eth.getBalance(web3.eth.coinbase), "ether")
3182
> eth.sendTransaction({from: eth.coinbase, to: "0xcec75e783a05d70b402ae881edafb1f4e19571bc",
value: web3.toWei(10, "ether")})

```

The transaction is not yet executed and we can see that is pending

```
> eth.pendingTransactions
[[
  {
    blockHash: null,
    blockNumber: null,
    from: "0x2ff5990db2abd2eb6d107c717bdb896e9342ea91",
    gas: 90000,
    gasPrice: 18000000000,
    hash: "0x204ffb4f279cc1c986f56d2c79173a7d9579606dfe0fe8b9d43bc427842389cb",
    input: "0x",
    nonce: 7,
    r: "0x92ab069704ee69ce1757c83691212448fe193f3a3be92ff3ab09a9729106791c",
    s: "0x1f0bbe53db4f09f790f1e924d288ccddb20e2114d98e16785651b5d42e63aa5",
    to: "0xcec75e783a05d70b402ae881edafb1f4e19571bc",
    transactionIndex: 0,
    v: "0x42",
    value: 1000000000000000000
  }
]]
>
```

It will be executed once we start mining in node 1 and we see the eth.log output

```
INFO [11-15|13:57:54] Submitted transaction                fullhash=0x204ffb4f279cc1c986
f56d2c79173a7d9579606dfe0fe8b9d43bc427842389cb recipient=0xCEC75e783a05d70B402ae881eDafb1F4e
19571bc
INFO [11-15|14:00:23] Updated mining threads                                threads=0
INFO [11-15|14:00:23] Transaction pool price threshold updated price=18000000000
INFO [11-15|14:00:23] Starting mining operation
INFO [11-15|14:00:23] Commit new mining work                                number=647 txs=1 uncles=0 ela
psed=5.402ms
INFO [11-15|14:00:57] Successfully sealed new block                          number=647 hash=ecdae4...3168d0
```

We stop the miner and we check again for pending transactions.

```
> miner.stop()
true
> eth.pendingTransactions
[]
>
```

and we check in node 3 to check the balance. Note that previous balance was 40

```
> web3.fromWei(web3.eth.getBalance(web3.eth.coinbase), "ether")
40
> web3.fromWei(web3.eth.getBalance(web3.eth.coinbase), "ether")
70
>
```

We repeat the process to send ether from node 3 to node 2 and to node 1

```

> personal.unlockAccount(eth.coinbase)
Unlock account 0xcec75e783a05d70b402ae881edafb1f4e19571bc
Passphrase:
true
> eth.sendTransaction({from: eth.coinbase, to: "0xcec75e783a05d70b402ae881edafb1f4e19571bc", value: web3.toWei(30, "ether")})
0x85c1dcbab25b94930c6729d44df9bbc47be7f3c2b779833e1c8327776c6323a2"
> eth.sendTransaction({from: eth.coinbase, to: "0xf8548fbd530cf256aa6f170e9102afa2f1d8c7e1", value: web3.toWei(30, "ether")})
0xb736964745c0c9a44721301dbd782a24b2e97e8588f75ac4df3ad41c0122c3c4"
> eth.pendingTransactions
[[
  {
    blockHash: null,
    blockNumber: null,
    from: "0xcec75e783a05d70b402ae881edafb1f4e19571bc",
    gas: 90000,
    gasPrice: 18000000000,
    hash: "0x85c1dcbab25b94930c6729d44df9bbc47be7f3c2b779833e1c8327776c6323a2",
    input: "0x",
    nonce: 0,
    r: "0x623ba777f3800ff6f64452466cc8c5525e123f7566968184484d9ec60ab10c73",
    s: "0x2317e23751734afa7e41247417053a2edd3990614ef00fd020c86cc7415112f",
    to: "0xcec75e783a05d70b402ae881edafb1f4e19571bc",
    transactionIndex: 0,
    v: "0x42",
    value: 30000000000000000000
  }, {
    blockHash: null,
    blockNumber: null,
    from: "0xcec75e783a05d70b402ae881edafb1f4e19571bc",
    gas: 90000,
    gasPrice: 18000000000,
    hash: "0xb736964745c0c9a44721301dbd782a24b2e97e8588f75ac4df3ad41c0122c3c4",
    input: "0x",
    nonce: 1,
    r: "0x343915487b4085c7025b45153f85ef1753c0874af8c4c7263f836f6c4acd1540",
    s: "0x24f366e9c9383ee3fcc9b32eb35ce10603dd133f365c354bec7f6c55b883759",
    to: "0xf8548fbd530cf256aa6f170e9102afa2f1d8c7e1",
    transactionIndex: 0,
    v: "0x41",
    value: 30000000000000000000
  }
]]
> miner.start()
null
>

```

and we see the difference in ether in node 2.

```

> web3.fromWei(web3.eth.getBalance(web3.eth.coinbase), "ether")
96
> web3.fromWei(web3.eth.getBalance(web3.eth.coinbase), "ether")
126
>

```


10. DApp Development

In order to get the benefits both of the blockchain and the private network tight security, we can develop a model under a DApp can be developed.

The case scenario is the following:

- The application must be accessible only from the organization's private network.
- The application must be installed only verified by the administrators' devices.
- In case the authorized device is moved outside the organization's network, no access to the application should be possible.
- In case another device sits on the organization's network no access should be gained over the network and conclude to successfully launching the application.

10.1 Development Workflow

For the requirements to be achieved we have to rely onto a trusted person, administrator, that will be responsible to initially set up the blockchain nodes to the devices and create the accounts.

The administrator on top of securing the device, has to repeat the described process to successfully install a blockchain node and the actual DApp installation. The administrator is responsible for creating the account on the blockchain node and thus providing the node passphrase to the user. After installing and testing the Blockchain node, the administrator must install the application on the device and test it in conjunction with the network. After the test is successful, the administrator must provide the credentials needed to the end user.

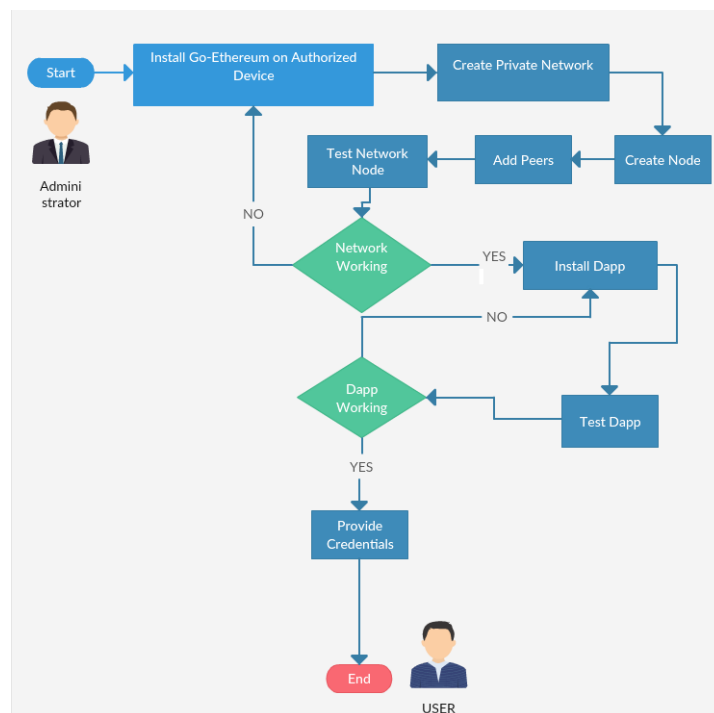


Figure 10: Workflow Procedure

10.2 Application Workflow

Since we need the application to be based on the security that blockchain provides, no access to the application should be achieved if prior we haven't met the network.

The DApp is developed in Angular 2, so the user has a front end UI in order to be able to interact with the network. In order to save resources the blockchain node is not working on booting the device. An automated procedure is done on booting the device in order all the requirements are met to start the node.

On visiting the web application page the user only has the option to press a button and connect to the network. Only if the node is connected to the organization's private blockchain network the application redirects to the actual login page where the user can enter his/her's credentials.

By entering the credentials, the application invokes a trigger to a smart contract in order to check if the user is authorized or not. Upon success the user is able to login and enter the main features of the application.

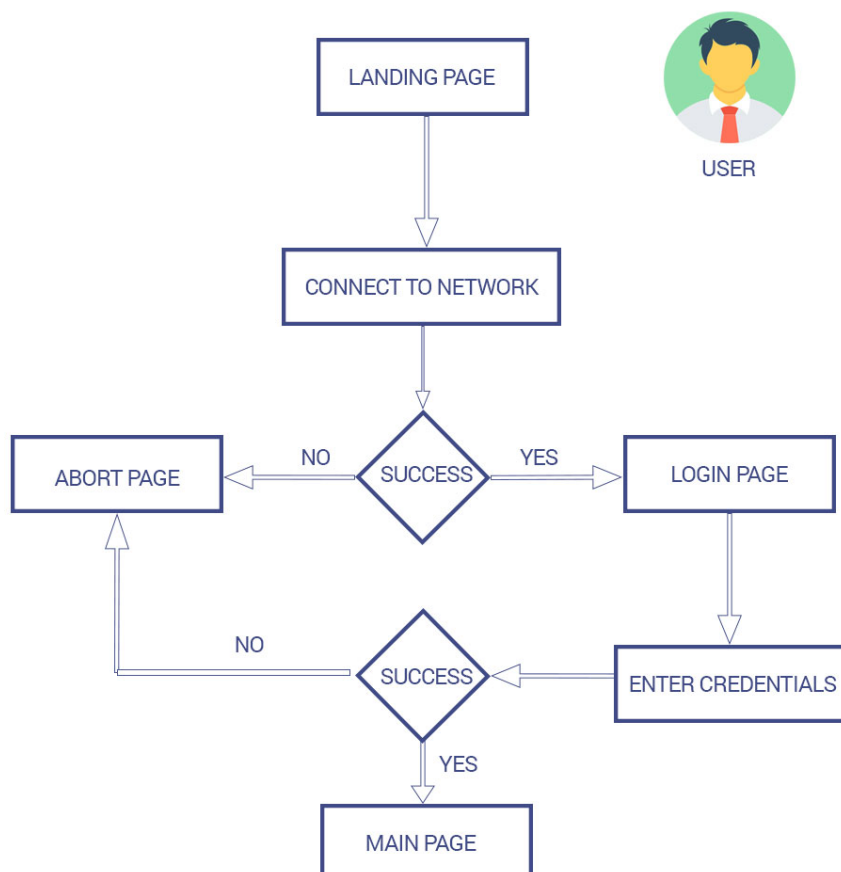


Figure 11: Application Workflow

10.3 Setting the requirements for the nodes

10.3.1 Supervisor – Running Script on Boot

The concept behind the setup of the application is that the systems that participate on the blockchain network and running the application, will have all the requirements set up and running on boot. For this purpose we install at the Ubuntu Virtual Machines, the supervisor.

Supervisor is a client/server system that allows its users to control a number of processes on Unix-like operating systems. (Supervisor, 2017)

In order to install the supervisor we give the following command

```
blockchain@blockchain-virtual-machine:~/Desktop/dapp_project$ sudo apt-get install supervisor
```

After the supervisor is installed we need to specify in the configuration file, the script that we want to run on boot, that will prepare and set all requirements for the node to get up and running when we desire.

The configuration file for the service has to be located at `/etc/supervisor/conf.d`. At the file we specify the path of the script to be executed, and some parameters that can be found at supervisor documentation.

```
blockchain@blockchain-virtual-machine:/etc/supervisor/conf.d$ cat geth-flask.conf
[program:geth-flask]
command=/home/blockchain/Desktop/dapp_project/startup.sh
autostart=true
autorestart=true
user=blockchain
stderr_logfile=/var/log/supervisor/geth-flask.err.log
stdout_logfile=/var/log/supervisor/geth-flask.out.log
blockchain@blockchain-virtual-machine:/etc/supervisor/conf.d$
```

10.3.2 Script for the requirements

Supervisor will execute the script we specified on the configuration file. The start-up script, is responsible for installing and setting up the requirements for the Flask framework.

The Flask framework is a micro-framework for Python. It is called a micro-framework because it does not require any particular tools or libraries. It is very simple, with no database abstraction layer, or form validation but can be extendable through various extensions. (Ronacher, 2010)

```
blockchain@blockchain-virtual-machine:~/Desktop/dapp_project$ cat startup.sh
#!/bin/sh

### Script to run python flask framework on start up
echo "User is:" whoami
echo "Moving to dapp folder..."
cd /home/blockchain/Desktop/dapp_project/
echo "Current folder is:" pwd
echo "Executing virtualenv..."
virtualenv venv
echo "Executing activation of virtualenv..."
. venv/bin/activate
if [ $? -eq 0 ]; then
    echo Venv Activated
else
    echo Venv FAIL
fi
echo "Installing Flask..."
pip install Flask
if [ $? -eq 0 ]; then
    echo Flask Installed
else
    echo Flask FAIL
fi
echo "Executing run.py..."
python run.py
```

As we can see the script loads all the requirements for the flask framework to be up and running and once those are executed it runs the run.py file. The Flask framework, enables a Python web server running at localhost:5000

This is the url that the Application calls in order to execute the script to enable the blockchain node. In order this to work, since the application runs on different port, CORS has to be enabled or specified on the Flask configuration loading the appropriate flask extension.

Once the application connects through the Flask's websocket to the blockchain network and receives the proper response that the blockchain network is connected and the node is running, only then the application proceeds to the authentication step of the user.

By following this workflow, we ensure that only the system that is properly setup to connect to the blockchain network, connects to it, and after this, the application proceeds to the next step which is the authentication of the user. At this point since we are connected on the blockchain, the authentication process will be done by triggering a smart contract on the blockchain.

10.4 Launching the DApp

Since all the requirements have been met with the network setup, we launch the application on localhost and on port 4200. The first page of the application will just show a button to connect to the network. At this point by pressing the button, the application will trigger the web socket at localhost and port 5000 and will connect the node to the network.



Figure 12: Launching the DApp

From the console we see that before pressing the button the web socket is waiting the trigger at localhost and port 5000.

```
(from Jinja2>=2.4->Flask)
Flask Installed
Installing flask-cors
Requirement already satisfied: flask-cors in ./lib/python3.6/site-packages
Requirement already satisfied: Six in ./lib/python3.6/site-packages (from flask-cors)
Requirement already satisfied: Flask>=0.9 in ./lib/python3.6/site-packages (from flask-cors)
Requirement already satisfied: Werkzeug>=0.7 in ./lib/python3.6/site-packages (from Flask>=0.9->flask-cors)
Requirement already satisfied: click>=2.0 in ./lib/python3.6/site-packages (from Flask>=0.9->flask-cors)
Requirement already satisfied: itsdangerous>=0.21 in ./lib/python3.6/site-packages (from Flask>=0.9->flask-cors)
Requirement already satisfied: Jinja2>=2.4 in ./lib/python3.6/site-packages (from Flask>=0.9->flask-cors)
Requirement already satisfied: MarkupSafe>=0.23 in ./lib/python3.6/site-packages (from Jinja2>=2.4->Flask>=0.9->flask-cors)
Flask - CORS Installed
Executing run.py...
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 114-388-159
```

Figure 13: Web Socket at standby

By pressing the button, the trigger to the web socket is activated and the node is started and connected to the network. The application shows a message upon successfully connected to the network as well as a button to disconnect from it by actually killing the process geth (the client that we use to run the Ethereum node).

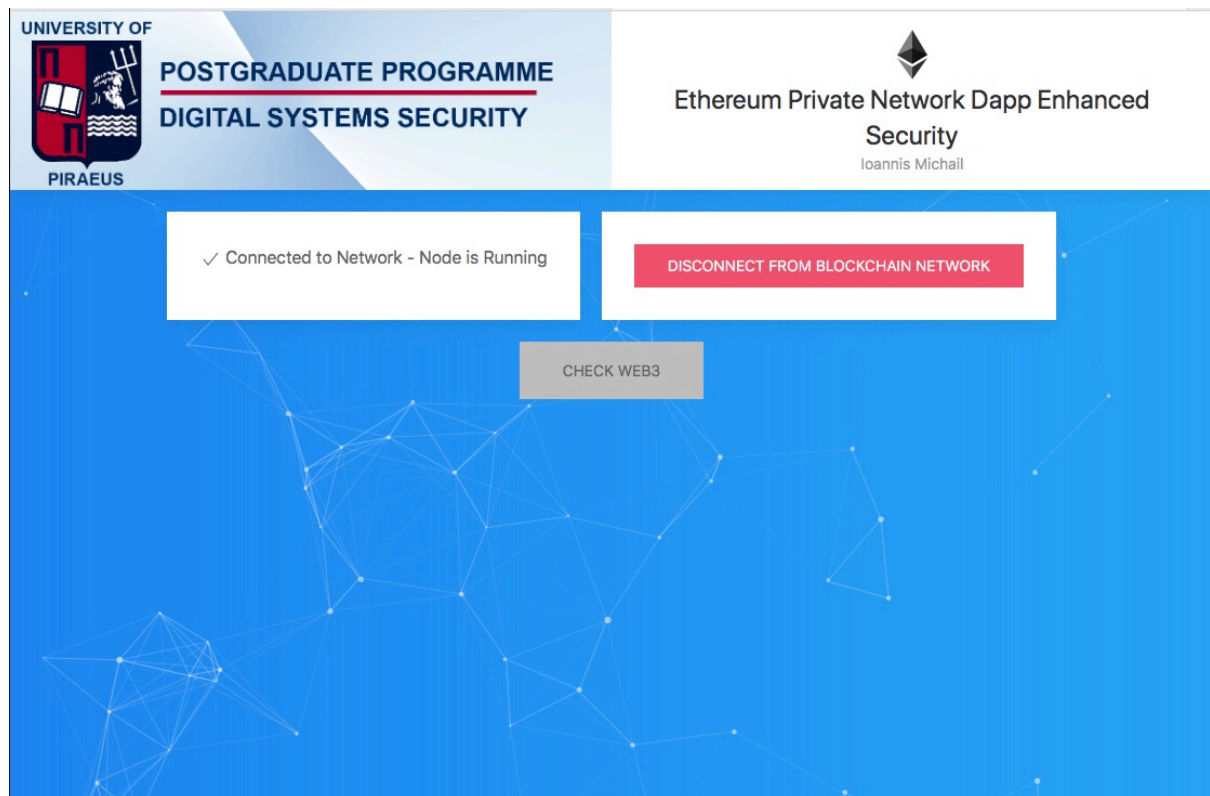
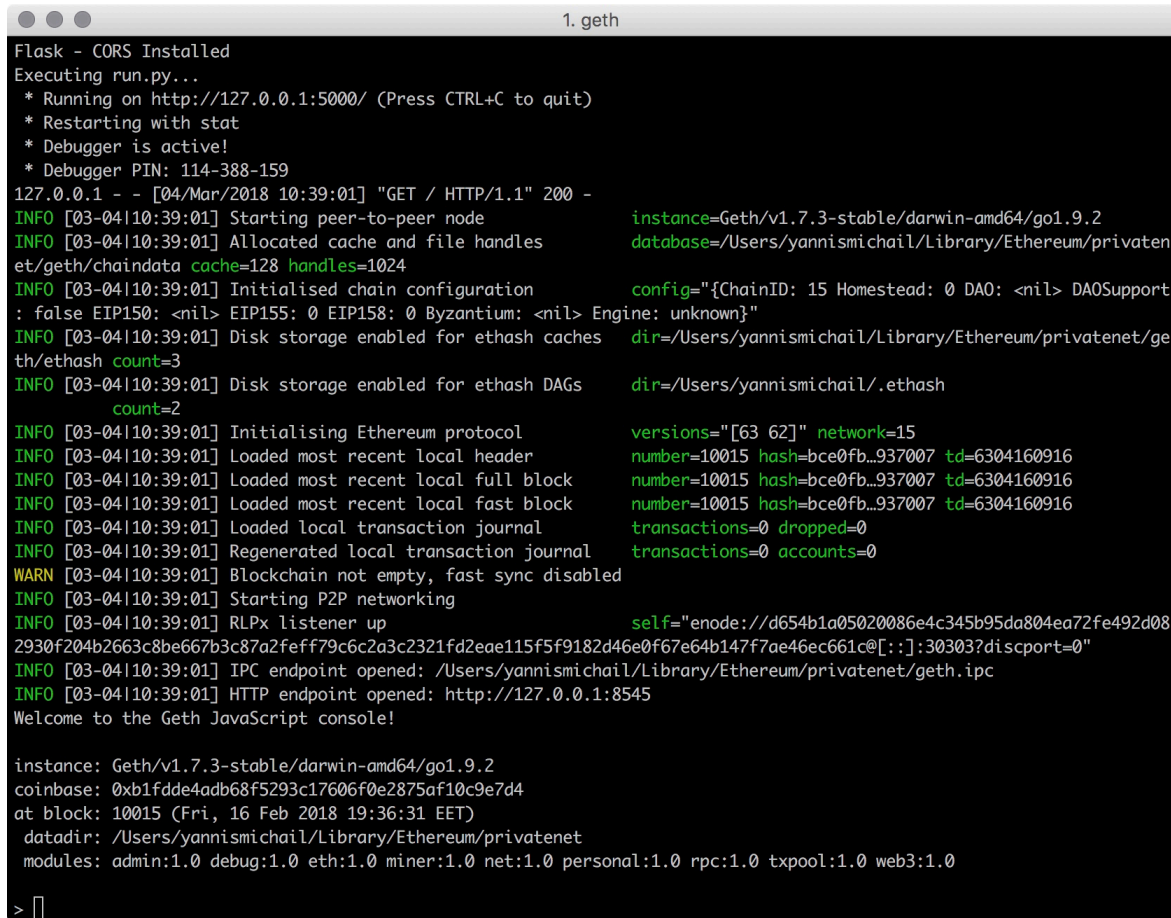


Figure 14: Connected to the Network - DApp

From the console we see that the node has successfully been started and connected to the network and the RPC endpoint that it listens to is localhost and on port 8545.



```
Flask - CORS Installed
Executing run.py...
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 114-388-159
127.0.0.1 - - [04/Mar/2018 10:39:01] "GET / HTTP/1.1" 200 -
INFO [03-04|10:39:01] Starting peer-to-peer node           instance=Geth/v1.7.3-stable/darwin-amd64/go1.9.2
INFO [03-04|10:39:01] Allocated cache and file handles                   database=/Users/yannismichail/Library/Ethereum/privatenet/
geth/chaindata cache=128 handles=1024
INFO [03-04|10:39:01] Initialised chain configuration                     config="{ChainID: 15 Homestead: 0 DAO: <nil> DAOSupport
: false EIP150: <nil> EIP155: 0 EIP158: 0 Byzantium: <nil> Engine: unknown}"
INFO [03-04|10:39:01] Disk storage enabled for ethash caches              dir=/Users/yannismichail/Library/Ethereum/privatenet/ge
th/ethash count=3
INFO [03-04|10:39:01] Disk storage enabled for ethash DAGs                dir=/Users/yannismichail/.ethash
count=2
INFO [03-04|10:39:01] Initialising Ethereum protocol                     versions="[63 62]" network=15
INFO [03-04|10:39:01] Loaded most recent local header                    number=10015 hash=bce0fb...937007 td=6304160916
INFO [03-04|10:39:01] Loaded most recent local full block                number=10015 hash=bce0fb...937007 td=6304160916
INFO [03-04|10:39:01] Loaded most recent local fast block                 number=10015 hash=bce0fb...937007 td=6304160916
INFO [03-04|10:39:01] Loaded local transaction journal                   transactions=0 dropped=0
INFO [03-04|10:39:01] Regenerated local transaction journal               transactions=0 accounts=0
WARN [03-04|10:39:01] Blockchain not empty, fast sync disabled
INFO [03-04|10:39:01] Starting P2P networking
INFO [03-04|10:39:01] RLPx listener up                                   self="enode://d654b1a05020086e4c345b95da804ea72fe492d08
2930f204b2663c8be667b3c87a2feff79c6c2a3c2321fd2eae115f5f9182d46e0f67e64b147f7ae46ec661c@[::]:30303?discport=0"
INFO [03-04|10:39:01] IPC endpoint opened: /Users/yannismichail/Library/Ethereum/privatenet/geth.ipc
INFO [03-04|10:39:01] HTTP endpoint opened: http://127.0.0.1:8545
Welcome to the Geth JavaScript console!

instance: Geth/v1.7.3-stable/darwin-amd64/go1.9.2
coinbase: 0xb1fdde4adb68f5293c17606f0e2875af10c9e7d4
at block: 10015 (Fri, 16 Feb 2018 19:36:31 EET)
datadir: /Users/yannismichail/Library/Ethereum/privatenet
modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0
> []
```

Figure 15: Console Geth connected to network

From the console window and the initialization information we get, we can see that the private network ID is indeed 15, the one we set up on the genesis file, as well as the location of the local private database for the blockchain network. Since the HTTP endpoint is opened at localhost on port 8545, the next step is to check that the DApp is able to connect to the HTTP endpoint through the WEB3 Javascript API.

The WEB3 Javascript API is responsible to communicate to a local node, through RPC Calls.

By pressing the check WEB3 button, the application checks the presence of web3.js, and upon successfully communicating with the local node, as proof of concept it returns a message verifying that the WEB3 is present, the node address and the latest block number that is found on the blockchain.

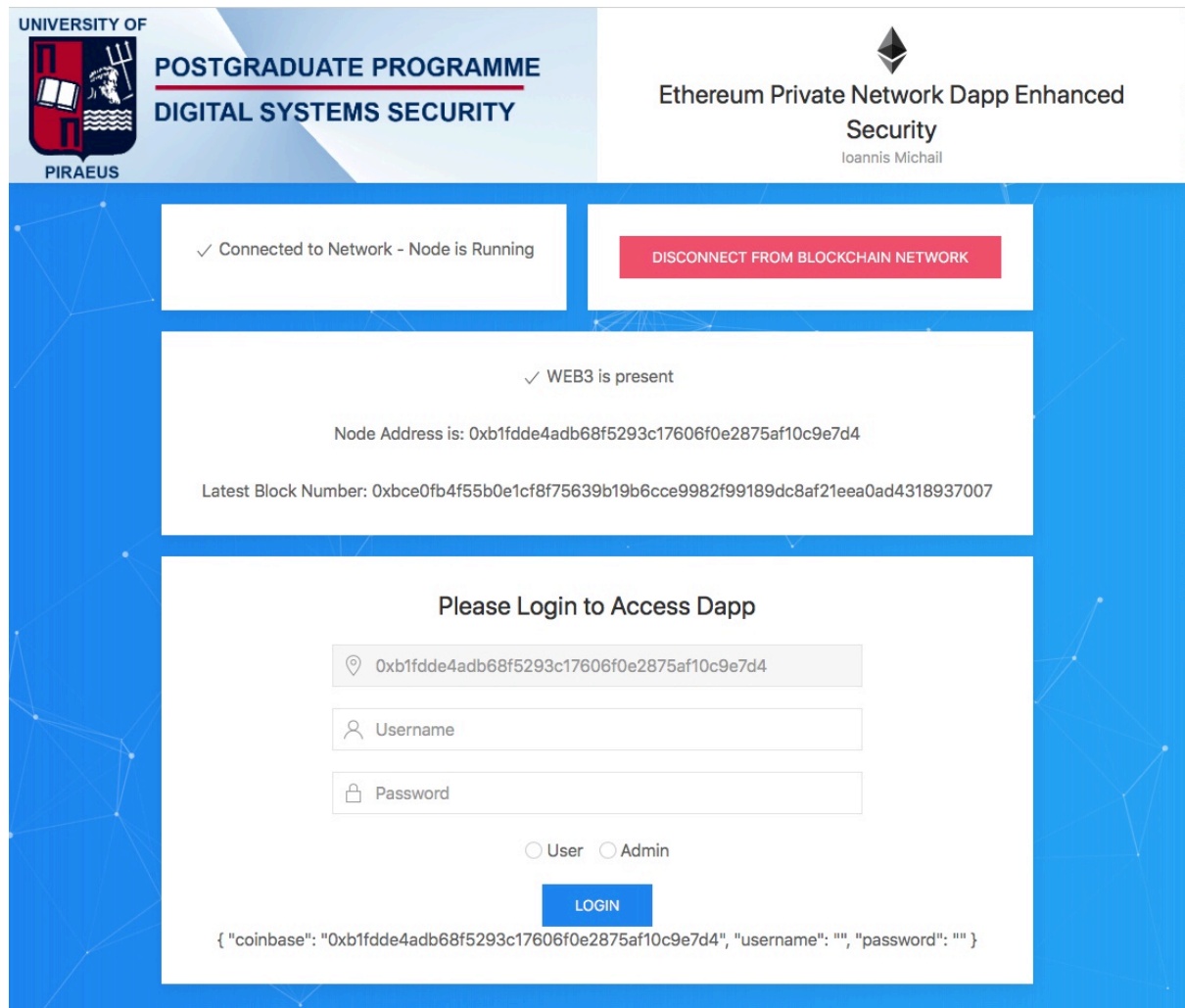


Figure 16: DApp checking WEB3 and login form

Since the WEB3 is successfully found and communication between the DApp and the local node is established, the DApp shows the login form to the user.

The login form consists of 3 fields, the first one is the coinbase address of the node, which is taken from a web3.js call to the local node and is not accessible by the user, the second is the username and the third is the password of the user.

These data are set from the administrator on the initial set up of the node and are given to the user. The final field of the login form is a radio button, indicating whether the person that wants to login is an user or an administrator of the DApp.

For the purpose of the project under the login button we have left an indication message fields to see the actual data that are going to be sent to the application.

10.5 Authentication

10.5.1 Smart Contract

For the purpose of the authentication a smart contract has been created that sits upon the blockchain network. We are going to go through the smart contract so to be able to understand better the authentication procedure.

```
pragma solidity ^0.4.17;

contract LoginAddresses {
    struct AllowedAddress{
        string username;
        string hashed;
        string role;
    }

    mapping(address => AllowedAddress) public approvedAddresses;
    address[] public addresses;
    address public admin;

    function LoginAddresses() public {
        admin = msg.sender;
    }

    modifier restricted(){
        require(msg.sender == admin);
        _;
    }

    function createAllowedAddress(address _address, string _username, string _hash, string _role) public restricted {
        var approvedAddress = approvedAddresses[_address];

        approvedAddress.username = _username;
        approvedAddress.hashed = _hash;
        approvedAddress.role = _role;

        addresses.push(_address) -1;
    }

    function getAllowedAddresses() public view returns(address[]){
        return addresses;
    }

    function getAllowedAddressesCount() public view returns(uint) {
        return addresses.length;
    }

    function getAllowedAddress( address _address, string _hash) public view returns(bool){
        if (keccak256(approvedAddresses[_address].hashed) == keccak256(_hash)){
            return true;
        }else{
            return false;
        }
    }

    function getHash(address _address) public view returns (string){
        return approvedAddresses[_address].hashed;
    }
}
```

Figure 17: Smart Contract

The smart contract is written on Solidity. The first line is to declare which version we use, at this point is 0.4.17

The name of the contract is LoginAddresses.

We declare a struct named AllowedAddress, and it consists of three strings, the username, the hashed and the role. This is the struct where we are saving the data of the allowed users to login to the DApp.

Next we declare a mapping named approvedAddresses, that gets an address and maps it to the struct AllowedAddress.

We also declare an array of addresses named addresses and an address admin.

The constructor of the smart contract has to have the same name as the contract itself, and there we set the admin address that we declared to be equal to the msg.sender address, thus the address of the person that is calling the smart contract.

We construct a function modifier to restrict that the message sender address is the admin and no other address.

Next we construct a function named createAllowedAddress that takes the address, the username, the hash and the role strings.

The address is needed to make a mapping to that address and the rest of the data are inserted to the struck object that this specific address maps to.

So with this function we can add a username, a hash and a role on the smart contract that are mapped to a specific address. This function is restricted, meaning that the person that calls it has to be an administrator.

Following we have helper functions named getAllowedAddresses and getAllowedAddressesCount that the first returns as array with the addresses and the second the length of that array.

The actual check of the authentication takes place on the next function named getAllowedAddress. This function takes an address and a hash and checks on the mapping if the specific address has the same hash value and returns either true or false.

So when from the DApp, a user fills in the form, with the procedure we are going to explain next, the form calls the getAllowedAddress from the smart contract and gets a Boolean value of true or false in order to proceed with the authentication. If a false value is returned, meaning that there is no allowed user to use the DApp with that address and hash, the login procedure is never going to proceed to the backend API to verify the credentials. Thus the whole procedure is controlled by the smart contract response.

10.5.2 Login Form

As previously described the login form gets from the local node the coinbase address, and from the user the username, password and role. The login form next calls the function `signIn` that takes those credentials and proceeds as follows.

Using the CryptoJS we encrypt the password that the user inputs, using AES. Next we concat to a variable the coinbase the username the role and the encrypted password and we hash the variable using `keccak256`.

The values that are being sent to the smart contract are the coinbase address and the total hashed value.

```
async signIn2(credentials) {
  console.log('credentials before auth service', credentials);
  console.log('address:', credentials.coinbase, 'username:', credentials.username, 'password:', credentials.password, 'role:', credentials.role);
  // Encrypt the Password with Base64
  const key = CryptoJS.enc.Base64.parse( encodedMessage: ' ');
  const iv = CryptoJS.enc.Base64.parse( encodedMessage: ' ');
  const coinbase = credentials.coinbase;
  // Implementing the Key and IV and encrypt the password
  const encrypted = CryptoJS.AES.encrypt(credentials.password, key, {iv: iv});
  console.log('encrypted password:', encrypted.toString());
  console.log('concat:', credentials.coinbase + credentials.username + credentials.role + encrypted );
  const toHash = credentials.coinbase + credentials.username + credentials.role + encrypted;
  const hash = keccak256(toHash);
  console.log('Hash Value:', hash);
  console.log('Send to auth service:', credentials.coinbase, hash);
  const login = await this.authService.testLogin3(coinbase, hash);
  console.log('returned login:', login);
  if (login){
    this.signIn(credentials);
  }
}
```

Figure 18: Sign In function

The `signIn2` function then calls the authentication service `login` function that is responsible to call and get the reply from the smart contract.

```
async testLogin3(_address, _hash){
  var deployed = await this.contractService.LoginAddresses.deployed();
  const result = await deployed.getAllowedAddress(_address, _hash);
  console.log('login3result:', result);
  return result;
}
```

Figure 19: Authentication Service Login Function

Once the true Boolean value is returned from the smart contract, meaning that indeed the user is allowed to proceed with the login, then the `signIn` function is called. For the purpose of the project a fake backend API provider has been set. In production environment this backend would be the actual backend service of the DApp.

This fake backend is based on the `MockBackend` feature of Angular CLI. (Angular, 2018)

The `signIn` function makes a call to a fake http url `/api/authenticate` and if the username and the password set there are the ones that the user has entered, then the fake backend gives to the user a JWT. The JWT is responsible with the role that contains if the user is an admin or not and accordingly gives access to the allowed pages that are guarded by it.

The JWT, thus the JSON Web Token is an open industry standard RFC 7519 method for representing claims securely between two parties. (JWT, 2018)

For the purpose of this project we have set a JWT on the backend, with the coinbase field, the username set to correctusername and the role to admin.

We fill the login form with the following credentials

UNIVERSITY OF PIRAEUS
POSTGRADUATE PROGRAMME
DIGITAL SYSTEMS SECURITY

Ethereum Private Network Dapp Enhanced Security
Ioannis Michail

✓ Connected to Network - Node is Running

DISCONNECT FROM BLOCKCHAIN NETWORK

✓ WEB3 is present

Node Address is: 0xb1fdde4adb68f5293c17606f0e2875af10c9e7d4

Latest Block Number: 0xbce0fb4f55b0e1cf8f75639b19b6cce9982f99189dc8af21eea0ad4318937007

Please Login to Access Dapp

0xb1fdde4adb68f5293c17606f0e2875af10c9e7d4

yannis

••••

User Admin

LOGIN

```
{ "coinbase": "0xb1fdde4adb68f5293c17606f0e2875af10c9e7d4", "username": "yannis", "password": "1234", "role": "2" }
```

Figure 20: Filling the Login Form

By pressing login, the form gets a true Boolean value from the smart contract, that there is a mapping with the specific address and the encrypted and hashed value sent, calls the fake backend and a JWT is set for the user to login.

The user since the JWT has a role of admin, is verified as administrator, thus has access to the admin area of the DApp.

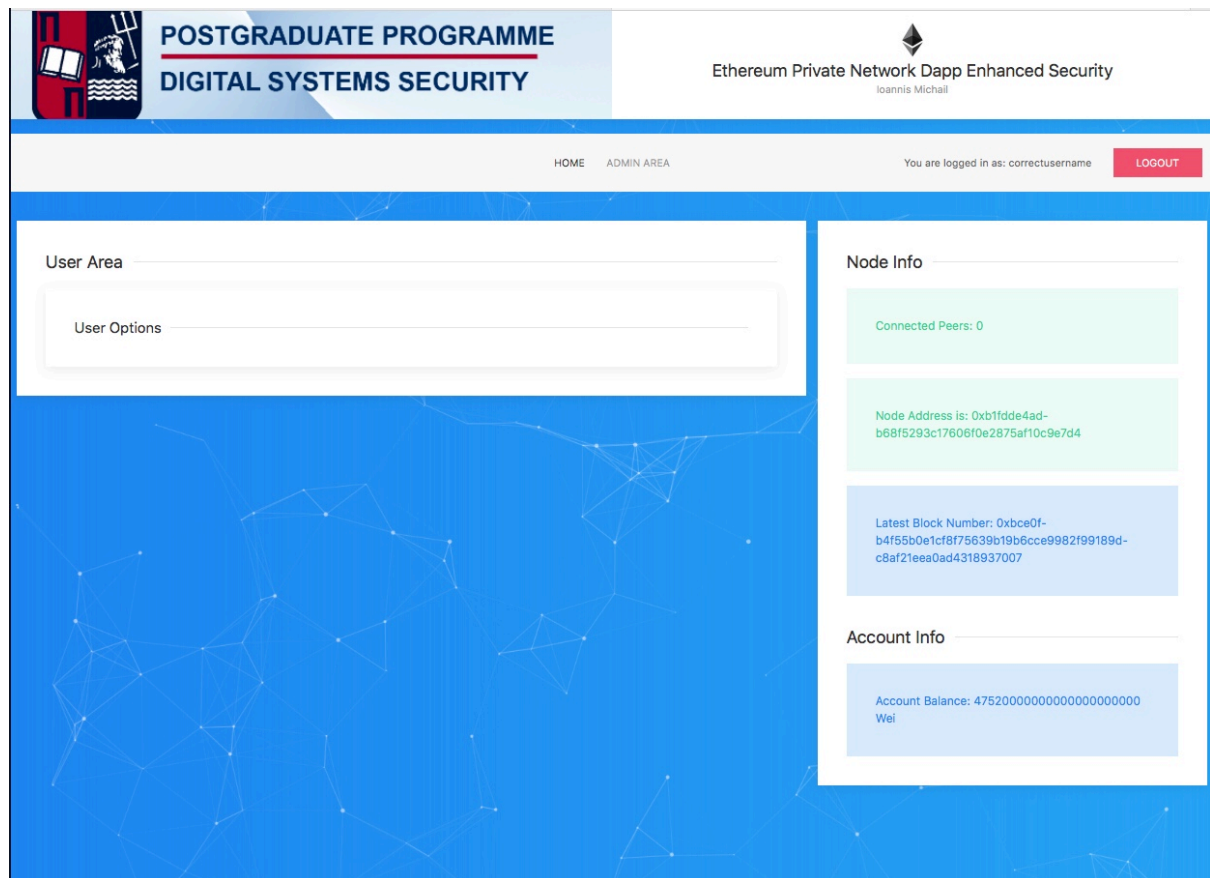


Figure 21: Main Page of DApp after login

At the main page of the DApp, and for proof of concept, we show how many connected nodes (peers) are on the blockchain network, the node address, the latest block number of the blockchain and the account balance that the user account has in Wei.

At the top right corner we see the message that the user is logged in as correctusername, which is the value we set on the JWT.

Since the specific user is verified as admin, there is an admin area that is accessible. The admin area is separated into two section, one that the admin user can see the address that the smart contract is deployed and by whom (which admin address. Following there is a table that consists of all the allowed nodes addresses.

POSTGRADUATE PROGRAMME
DIGITAL SYSTEMS SECURITY

Ethereum Private Network Dapp Enhanced Security
Ioannis Michail

HOME ADMIN AREA You are logged in as: correctusername LOGOUT

Administrator Area

LOGIN ADDRESSES CONTRACT INFO MANAGEMENT OPTIONS

Contract deployed at address:
0x09f7f4dce4864c996d387-fa6a466f1705e182cc6

Contract deployed by Address:
0xb1fdde4ad-b68f5293c17606f0e2875af10c9e7d4

Allowed Nodes Information

NODE
0xb1fdde4adb68f5293c17606f0e2875af10c9e7d4

Node Info

Connected Peers: 0

Node Address is: 0xb1fdde4ad-b68f5293c17606f0e2875af10c9e7d4

Latest Block Number: 0xbce0f-b4f55b0e1cf8f75639b19b6c-ce9982f99189d-c8af21eea0ad4318937007

Account Info

Account Balance: 47520000000000000000 Wei

Figure 22: Admin Area Information Page

At the management options page besides the general information regarding the contract and the number of the addresses that are deployed on the contract, the administrator can add an allowed node and credentials for the user through a form. In this form the coinbase of the user must be added, as well as the username and the password and the role of the user.

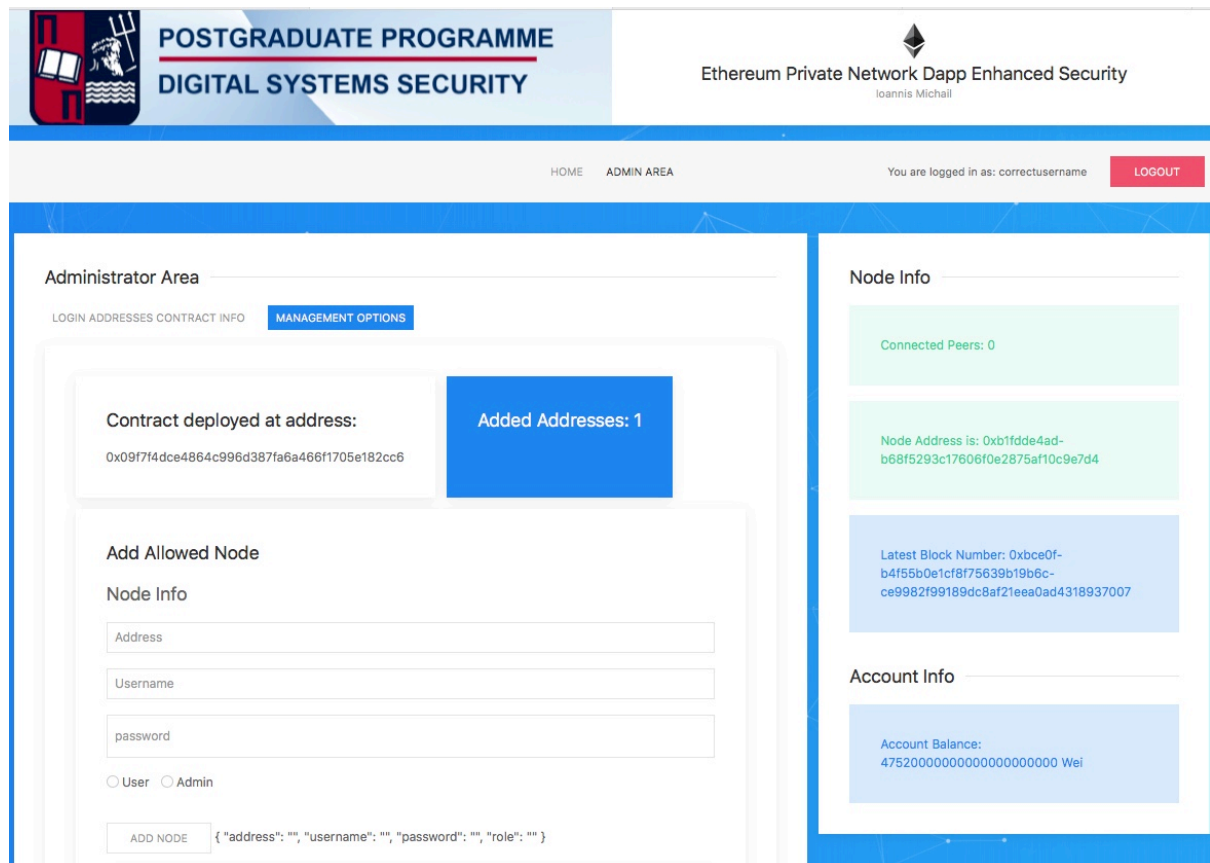


Figure 23: Administrator Management Area

Upon the form submission the DApp calls the smart contract function createAllowedAddress and maps the given address with the hashed value that the login form later will be called to compare.

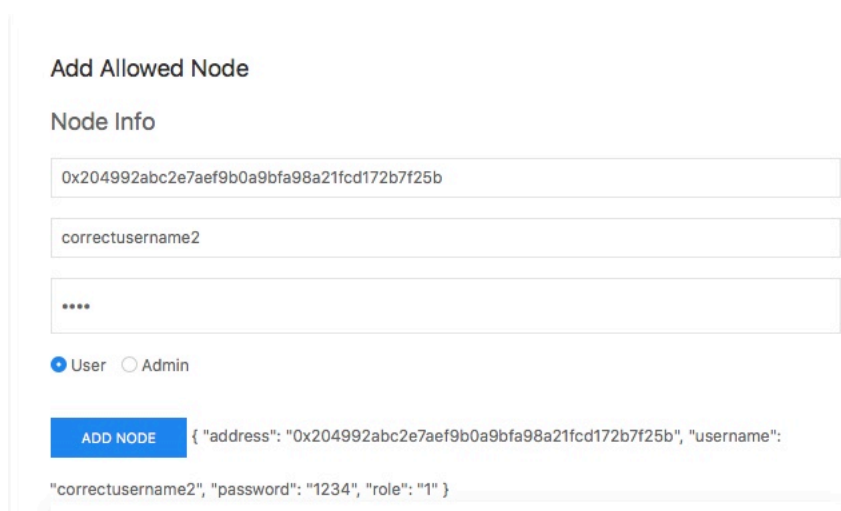


Figure 24: Entering Second Address

Prior to submitting the form, since the form has the intention to alter the smart contract, thus means that needed gas has to be set in the function and also the coinbase account has to be unlocked and the miner has to be started.

By submitting the form we see the following at the console.

```
INFO [03-04|14:03:28] Commit new mining work      number=10038 txs=0 uncles=0 elapsed=275.69us
INFO [03-04|14:03:30] Submitted transaction    fullhash=0xaf70c4748d2dbcfedcd041a202a8d548b6e85bbb95b3454d97659adb23c118 recipient=0x09f7f4Dce4864c996d387fa6e466f1705e182cc6
INFO [03-04|14:03:44] Successfully sealed new block  number=10038 hash=f18089..f6efa8
```

We are able to see the hash of the transaction sent, and also the address of the smart contract that is the recipient of the transaction.

At the DApp we can see that the new address has been added to the smart contract.

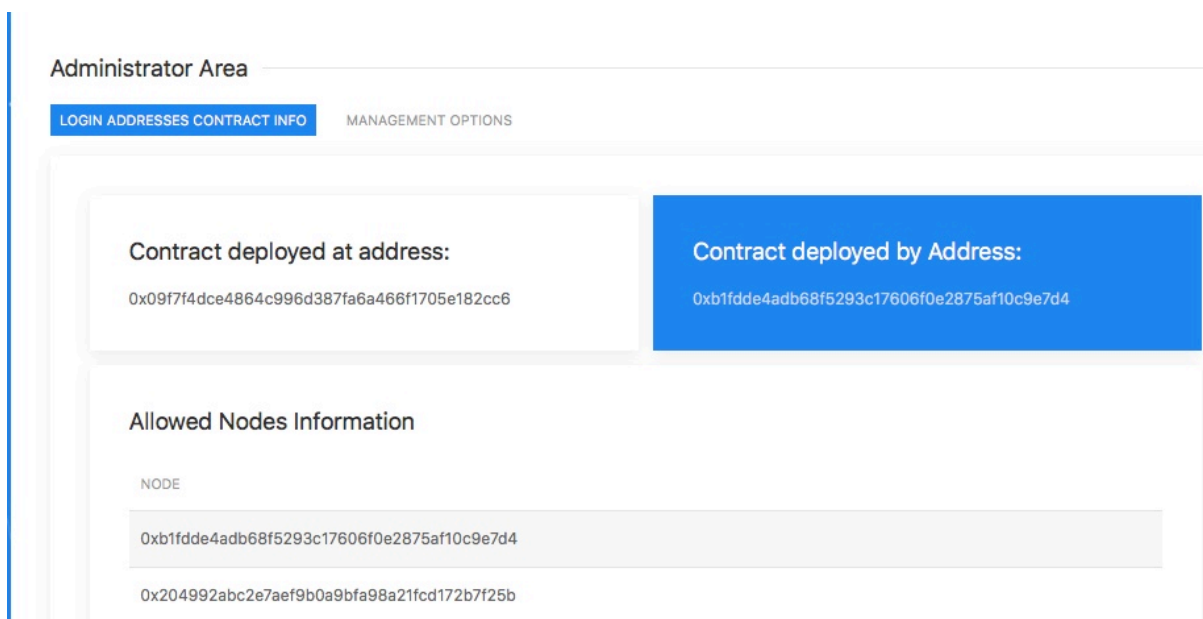


Figure 25: Added Address in Smart Contract

11. Conclusion

Being able to build a private blockchain network, where only pre-set computers from specific administrators will be able to join, both on the network section and on the application section we are able to increase the security of our deployed DApp.

Instead though implementing the authentication security on the smart contract, only by checking if a mapping is present on the smart contract, thus making the node suitable, when solidity will be mature enough we could implement the set of the JWT on the smart contract. In that way there would be no need to check afterwards on a traditional backend service whether the user is authenticated and to provide him/her the JWT in order to login and navigate to the pages of the DApp that are set depending on his/her's access level.

The Blockchain world is a constant developing and changing environment and only by being able to build private networks and program and control the blockchain through smart contracts, shows how many things have been accomplished by the developers in such a short time on such a new technology.

In the years to come and when the technology will be mature enough, and more significantly understood by the majority of the end users, the things that can be achieved could change the way we interpret the internet and the technologies that come with it and really lead to revolution in designing and building applications, services and secure networks.

Bibliography

- Angular. (2018). *MockBackend*. Retrieved from angular.io: <https://angular.io/api/http/testing/MockBackend>
- Audun, J., Roslan, I., & Colin, B. (2007). *A Survey of Trust and Reputation Systems for Online Service Provision*.
- Bakhoff, M. (n.d.). *Consensus algorithms for distributed systems*.
- Baliga, A. (2017). *Understanding Blockchain Consensus Models*.
- Bartoletty, M., & Pompianu, L. (2017). *An empirical analysis of smart contracts: platforms, applications, and design patterns*.
- Bashir, I. (2017). *Mastering Blockchain*. Packt Publishing.
- Bauerle, N. (2017). *What is a Distributed Ledger?* Retrieved from coindesk.com: <https://www.coindesk.com/information/what-is-a-distributed-ledger/>
- Beregszaszi, A. (2016, 4 4). *Smart contracts and ID cards*. Retrieved from ledger.press: <https://ledger.press/smart-contracts-and-id-cards-11bc16155737>
- blockgeeks. (2017). *Blockchain Glossary: From A-Z*. Retrieved from blockgeeks: <https://blockgeeks.com/guides/blockchain-glossary-from-a-z/>
- Buterin, V. (2017). Retrieved from Blockgeeks: <https://blockgeeks.com/guides/smart-contracts/>
- Castro, M., & Liskov, B. (1999). *Practical Byzantine Fault Tolerance*. Massachusetts: Laboratory for Computer Science, Massachusetts Institute of Technology.
- coindesk. (2017). *7 Cool Decentralized Apps Being Built on Ethereum*. Retrieved from Coindesk: <https://www.coindesk.com/7-cool-decentralized-apps-built-ethereum/>
- Eschenauer, L., Gligor, V. D., & Baras, J. (2002). On Trust Establishment in Mobile Ad-Hoc Networks. *Proceedings of the Security Protocols Workshop*. ECE Department, University of Maryland.
- ethereum. (2017). *Ethereum*. Retrieved from Ethereum: <https://ethereum.org/>
- ethereum.org. (2017). *Ethereum*. Retrieved from ethereum.org: <https://ethereum.org/>
- Github Ethereum. (2017). *Mining*. Retrieved from Github Ethereum: <https://github.com/ethereum/wiki/wiki/Mining>
- Github-Ethereum. (2017, 09 21). *Connecting-to-the-network*. Retrieved from github.com: <https://github.com/ethereum/go-ethereum/wiki/Connecting-to-the-network>
- Goyal, S. (n.d.). *Centralized vs Decentralized vs Distributed*. Retrieved from Medium: <https://medium.com/@bbc4468/centralized-vs-decentralized-vs-distributed-41d92d463868>
- hyperledger. (2017). *hyperledger*. Retrieved from hyperledger.org: <https://hyperledger.org/>
- Johnston, D., Yilmaz, S. O., Kandah, J., Bentenitis, N., Hashemi, F., Gross, R., . . . Mason, S. (2015, 2 2). *The General Theory of Decentralized Applications, DApps*. Retrieved from github: <https://github.com/DavidJohnstonCEO/DecentralizeDApplications>
- JWT. (2018). *JWT*. Retrieved from jwt.io: www.jwt.io
- KEy-chain. (n.d.). Retrieved from key-chain: <https://kyc-chain.com/>
- Lambert, N., Ma, Q., & Irvine, D. (2015). *Safecoin: The Decentralised Network Token*.
- Linn, J. (2000). *Trust Models and Management in Public-Key Infrastructures*. RSA laboratories.
- Litecoin. (2017). *Litecoin*. Retrieved from Litecoin: <https://litecoin.org/el/>

namecoin. (2017). *namecoin*. Retrieved from namecoin.org: <https://namecoin.org/>

oreilly. (n.d.). *The Blockchain*. Retrieved from oreilly.com:
http://chimera.labs.oreilly.com/books/1234000001802/ch07.html#_structure_of_a_block

primecoin. (2017). *Primecoin*. Retrieved from Primecoin: <http://primecoin.io/>

Prusty, N. (2017). *Building Blockchain Projects*. Packt Publishing.

Ronacher, A. (2010). Retrieved from Flask: <http://flask.pocoo.org>

Rousse, M. (n.d.). *PKI (public key infrastructure)*. Retrieved from Searchsecurity Techtarget:
<http://searchsecurity.techtarget.com/definition/PKI>

sia. (2017). *Sia*. Retrieved from sia.tech: <http://sia.tech/>

Supervisor. (2017). Retrieved from Supervisor: <http://supervisord.org/introduction.html>

Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed Systems_ Principles and Paradigms*. Pearson.

University of Houston. (n.d.). Retrieved from University of Houston:
<http://sce.uhcl.edu/goodwin/Ceng5334/downLoads/byzantine.pdf>

Veen, S. (2015, 12 25). *CAP theorem for distributed systems explained*. Retrieved from saipraveenblog: <https://saipraveenblog.wordpress.com/2015/12/25/cap-theorem-for-distributed-systems-explained/>

Willett, J., Hidskes, M., Johnston, D., Gross, R., & Schneider, M. (2017, 1 23). *Omni Protocol Specification (formerly Mastercoin)*. Retrieved from github:
<https://github.com/OmniLayer/spec>

z.cash. (2017). *z.cash technology*. Retrieved from z.cash:
<https://z.cash/technology/index.html>

Zheng , Y., & Holtmanns, S. (2007). *Trust Modeling and Management: from Social Trust to Digital Trust*. Idea Group Inc.

Zheng, Y., & Valtteri Niemi. (2009). *Towards User Driven Trust Modeling and Management*.