Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Laurea Magistrale in Informatica

Blockchain and Smartcontracts: Fundamentals and a Decentralized Application Case-Study

Relatore:
Chiar.mo Prof.
Stefano Ferretti

Presentata da:
Maxim Gaina

Sessione III
a.a. 2016-2017

"Set your house in order
before you criticise the world" ...
Jordan B Peterson.

Sommario

Questo lavoro di tesi tratta l'implementazione di un'applicazione decentralizzata in grado di certificare eventi accaduti nel mondo reale, compiuti da parte di un'organizzazione. Tali certificati devono essere (i) verificabili da chiunque, in un qualsiasi momento della storia e senza la necessità di terze parti, (ii) avere un costo di rilascio ragionevole per l'autore e (iii), avere tempi di conferma accettabili. Per poter definire meglio (a) le modalità in cui è possibile farlo, (b) l'esito del lavoro svolto ed eventuali problemi emersi, e (c) i futuri sviluppi, la tesi si prefissa i seguenti obiettivi. Verrà discusso in che modo un generico protocollo Blockchain si inserisce nell'ambito dei sistemi distribuiti. L'avvento di Bitcoin infatti, permette per la prima volta di implementare un sistema distribuito e decentralizzato, in cui è possibile avere fiducia tanto quanto necessario per eseguire transazioni finanziarie. Verranno individuate le proprietà che un protocollo deve avere per garantire sicurezza, e se possono essere raggiunte contemporaneamente. Nello stesso modo in cui i protocolli blockchain esistono "sopra" le reti peer-to-peer, verrà descritto il modo in cui gli Smartcontracts possono essere introdotti nelle blockchain. Verrà visto come questi costrutti rendono possibile l'implementazione di Applicazioni Decentralizzate (dapp). Si vedrà che tali registri distribuiti e pubblici, sono comunque in una fase di sviluppo piuttosto immatura e presentano alcuni problemi. Fra questi, quello della scalabilità all'aumentare degli utenti. Verrà visto il motivo, e il modo in cui questo problema interferisce con l'operatività della dapp realizzata.

Introduction

Reaching global consensus about facts within a distributed system, with no trusted authority has always been a problem. The advent of Bitcoin [2] and its consensus algorithm, is the result of various attempts to solve the digital currency double-spending problem, in a completely *decentralized* system for payments. For the first time indeed, the participants of a system are able to agree about a ledger of totally ordered events, with no intermediary. Such systems that require no *trust* into other participants however, are not only useful for payments. The underlying Blockchain technology started to be used in a variety of use cases, and new business models with no *middlemen* involved are being tested. Blockchain technology gave an impulse to research for new consensus algorithms, aimed to both public and private systems. It allowed then to integrate *Smartcontracts* [7, 8] into its architecture, giving life to decentralized programmable networks [15, 17] capable of running *decentralized applications*. The innovating impact of this new decentralization paradigm is broadly recognized. In this relatively immature stage of their life, blockchain and related technologies are facing a series technical, social and political challenges. Among all of them however, one main problem is how to reach global-scale adoption by solving its scalability issues. It is known that a blockchain network congestion almost certainly results into higher transaction fees and confirmation time delays. While it can be a temporary problem for normal users willing to transfer funds, it may be prohibitive to realize decentralized applications for real-world businesses. Often, they have a transaction cost upper bound, and should always guarantee operativity.

Aims of this Thesis

A blockchain is a public, immutable and ordered distributed ledger. This features allow to realize a wide range of applications. A specific case-study decentralized application will be realized. Given an information, and an event related to it which must be unique in time, the problem is to encode them and certify their authenticity on-demand on the blockchain. Such a decentralized application, must satisfy several requirements related to trust, transaction cost and confirmation delays. At any point in the history indeed, anyone must be able to verify a certificate authenticity, and no trusted party should be needed in order to do it. For real-world businesses, another problem is given by transactional cost upper bound and time after which a certificate is confirmed by the network consensus. In order to understand (i) in which degree such requirements can be achieved and how to realize such a decentralized application, (ii) how to interpret the outcomes and (iii) what is the likeliness that emerged problems will be solved, this thesis is structured as follows.

Bitcoin is the first blockchain protocol that with its consensus algorithm, reaches a relaxed form of *consistency* on pure peer-to-peer networks. It will be seen what kind of improvements it introduces as a distributed system. Prior *Byzantine Fault Tolerant* agreement algorithms instead, often relied on synchrony assumptions in order to achieve desired properties. The Bitcoin Protocol contribution to the problem of reaching fully asynchronous distributed consensus will be discussed in Chapter 1.

Since distributed ledgers are in some degree a modular technology, a blockchain protocol generalization will be viewed. It is then required to understand what kind of properties these protocols should have in order to be secure. Then, how these properties can be achieved and if simultaneously. Decentralized distributed ledgers introduce indeed new challenges that Chapter 2 will discuss. Particularly, the above mentioned scalability problem will be reviewed.

Just like blockchains exist on top of peer-to-peer networks, a blockchain

protocol allowed in turn to enforce other technologies on top of it. Smartcontracts are used to define higher level concepts like Decentralized Applications and Decentralized Autonomous Organizations. Chapter 3 will introduce the fundamentals of the Decentralization Paradigm, and the specific Ethereum implementation of this vision.

The above principles allow to implement decentralized applications like the one that will be described in this thesis. Today, multiple platforms providing these functionalities exist. Some of them are in a too early stage of their development, while others are suitable only for some problem types and not for others. The problem of issuing blockchain certificates will be defined, then we will try to understand what is the best platform to solve it, currently. Chapter 4 will provide a decentralized business logic definition, and a way to actually interact with it. The goal is to analyze in which degree our requirements of trust, operational cost and confirmation time are satisfied simultaneously. Then, what current architectural challenges are limiting our possibilities and why.

Contents

List of Figures

List of Tables

Chapter 1

Asynchronous Distributed Systems

A *ledger* is a book in which accounts or events are recorded. Historically, ledgers are used to reach consensus about facts such as ownership, identities, authority and status. A common agreement about facts, how and when they change should be reached. Most types of services available nowadays require to interact with an entity that must be *trusted*, which is often an intermediary actor that keeps its own ledger about users and their related data. It is well known today that sets of data generated by this *middlemen* are highly valuable, which of course led to an increase of data driven business models. To trust such a centralized entity implies to rely on its integrity, on its ability to handle attacks and to have confidence in the certainty of its future operativity. Last but not least, implies a position of vulnerability for the user. Alternately, building a system that behaves in a trustworthy manner without knowing any of its participants, has always been a problem, and a particular instance of this problem is given by digital currencies. As for physical mediums of exchange, digital currencies have to face the problem of being counterfeited. But while a physical object cannot be in multiple places at once, digital coins must additionally solve the *double-spending* problem,

where an actor is able to make multiple transactions using the same digital currency unit. For a digital currency that is backed by a national currency, the problem is solved by a centralized Financial Institution. Also known as Clearing House, it stands between participants and validates transactions according to its own rules, beside making sure that the system works properly.

The idea of a digital currency that reaches decentralized global consensus about the order of transactions, was around since decades but without any concrete solution. As we know, this was until 2009, when in an attempt to create such an electronic payment system, the first so called *crypto-currency* was born, known as *Bitcoin* [2]. The pseudo-anonymous owner of a Bitcoin is able to transfer it to the next, by signing previous output transactions which belongs to him, through asymmetric cryptography and trusting no one. After being validated by the network, the receiver is then able to unlock the content of the incoming transaction. When the receiver wants, he spends it by performing the same operations the sender did. New coins are produced through computational efforts made by the *peer-to-peer* network nodes. But these concepts had already been considered before:

- electronic cash problems such as accountability and anonymity have already been discussed previously [3], resulting into new cryptographic tools proposals;
- the concept of money creation through computationally challenging problem solving was proposed before Bitcoin too [5, 6] (using *hashcash* mechanism, initially designed to limit email spam and denial of service attacks [4]).

All this ideas however were still based on the principle that transactions must be timestamped by trusted servers to keep them ordered. What makes Bitcoin an applicable in practice innovation, allowing to interact with its system in a trustless manner, is the introduction of a computing power based writing permission. This ability to write refers to a distributed ledger of ordered *blocks* of transactions, instead of a list of ordered transactions. Net-

work participants are allowed to disagree about the most recent blocks of transactions, but they will always agree on the same unique *chain* structure. Since it can be viewed as an append-only chain of blocks, this data structure is called *blockchain*. The blockchain is continuously expanding according to its most computationally expensive version, which is the basis the Bitcoin's *decentralized consensus protocol*. Without any centralized component indeed, the protocol allows all the nodes in the network to agree on a unique event order. The main goal of this chapter is to understand the principles of the blockchain technology, and how it places itself as a distributed system. Blockchains are mainly a combination of the following science fields:

1. *Distributed Systems*;
2. *Cryptography*;
3. *Game Theory*.

Primary role of cryptographic primitives is to secure transaction ownership and ledger safety. Elements of game theory and behavioral economics assumptions are crucial when designing consensus algorithms. Decentralization technologies became also objects of study from a juridical, economic and political points of view. While some of these aspects will be covered, none of them are the argument of this work. Decentralized ledger technologies will be mainly discussed in this chapter, as well as in the entire thesis, as distributed and peer-to-peer systems.

1.1 Distributed Systems

This section provides basic concepts about distributed systems, to better understand how the blockchain introduces itself as a technology. A distributed system is a network consisting of autonomous components, that process *local* knowledge to achieve *global* goals. Different resources and capabilities are shared among nodes, to provide each user with a coherent network that behaves, as much as possible, as a single system.

Adopting the terminology seen in [9], when considering a distributed system, assumptions are to be made about several crucial factors:

1. *timing model*, which is formed by the timing events in a distributed system, can be distinguished as:
   - *synchronous*, meaning that components perform simultaneous, well organized computational steps, there is a precise time lower and upper bound for events to occur;
   - *asynchronous*, where distinct components perform computational steps in an arbitrary order, with a message based progress and no timing guarantees, with high levels of uncertainty;
   - *partially synchronous*, the lower and the upper bounds exist like in the asynchronous model, but they are unknown, therefore unusable as parameters.
2. *inter-process communication*, specifies the communication type between nodes, that can be message-based or by accessing a shared memory;
3. *failure model*, which is the expected class of unintentional errors or malicious behavior conducted by the nodes;
4. *addressed problem*, the class of problems which the system or the protocol tries to solve.

The asynchronous model has no global clock, and its nodes are expected to make local decisions, based on external messages and internal computations. It captures indeed environments like public Internet, where an event may occur with an arbitrary delay or never. Partially synchronous models are often used to implement real world applications, but their latency assumptions might be violated. Synchronous models on the other side, are mainly useful for integrated circuits and theoretical environments, where reasoning on technical feasibility algorithms is required. The most sever type of failure model is given by asynchronous public networks, where a node or

a group of nodes may behave arbitrarily (1.1.2), enter or leave the system whenever they want. The failure model is also defined by the software and hardware specifics the nodes have in common. The problems addressed by a distributed system might be communication, resource allocation systems, service providers, consensus problems and so on.

1.1.1 Consistency, Availability and Partition Tolerance

When designing web services for asynchronous distributed systems, the following properties are often required:

- **consistency**, (also referred as *safety* or *atomicity*) guarantees that all operations made in the system are totally ordered, or, in other words, the distributed shared memory of the network is asked to respond one request at a time, as if it were a single local machine memory;
- **availability**, (also referred as *liveness*) ensures that every request made to the system must terminate, it means that if received by a healthy node, must result in a response to the client;
- **partition tolerance**, the network is allowed to lose an arbitrary number of messages from one node to another.

The described properties however, cannot be achieved all three in the same asynchronous network. It has been firstly stated as a conjecture by Eric Brewer, in 1998, which resulted later into the CAP Theorem 1.

Theorem 1 (CAP Theorem)**.** *A distributed system cannot have Consistency, Availability and Partition Tolerance simultaneously.*

A formal proof has been later provided [1], which shows that it is only possible to achieve two out of these three properties. While the partition tolerance is a binary property, meaning that it can be only allowed to lose messages or not, consistency and availability are measurable in a spectrum. Therefore, assuming that the network has no partitions, it is possible to

provide full consistency and availability. Otherwise, it is only possible to have a trade-off between them.

1.1.1.1 Distributed Consensus Algorithms

The previously discussed consistency property is achieved by *consensus algorithms*. Consensus algorithms allow network nodes to share the same memory state, even if a subset of its participants are faulty processes. An ideal consensus mechanism possess the following characteristics:

1. *termination*, correctly working nodes always terminate their execution of the consensus protocol;
2. *agreement*, correctly working nodes make decision based on the same values;
3. *fault toleration*, the algorithm continues to operate even in presence of malicious or broken nodes;
4. *validity*, values agreed by nodes are the same as the ones initially proposed by at least one correct node;
5. *integrity*, correctly working nodes take a decision once, in a single decision cycle.

The way these challenges have been addressed will be discussed below.

1.1.2 Byzantine Generals Problem

In the field of agreement protocols, a well known problem is the *Byzantine Generals Problem* (BGP). To better explain it, the author tells a scenario where distinct divisions of the Byzantine Army surround an enemy city [10]. Each general in charge of its own division, after observing the city, communicates to other generals his own decision to attack or retreat. The problem is that some of these generals might be traitors, sending then wrong or even incoherent messages to each of other generals. In order to have a system

with reliable outputs, the following goals must be achieved: the group of loyal generals must agree on the same plan, attack or retreat; and the group of traitors shouldn't be able to induce loyalists to make bad decisions. If $v(i)$ is the information communicated by the ith general, these two conditions have to be verified in order to achieve the goals:

1. every loyal general must have the same information $\{v(1), ..., v(n)\}$;
2. if the ith general is loyal, the information he sends must be used by every loyal general as $v(i)$.

Since both conditions are focused on a single event $v(i)$, the problem can be reduced as follows. Instead of having n generals, it can be viewed as a single commanding general and its $n-1$ lieutenants.

Problem 1 (Byzantine Generals Problem)**.** *A commanding general must send an order to his* $n-1$ *lieutenant generals such that:*

1. *All loyal lieutenants obey the same order;*
2. *If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.*

Withing this metaphor, loyal generals and lieutenants are *correct processes*, the orders they send and receive are *messages*, traitors are *faulty processes* (whether malicious or unintentional). In a reliable system, in order to make a set of processes generate the same valid output, first of all it is necessary to provide all of them the same input. So the commander general is the input generator, while lieutenants are the processes that should elaborate the same input. Hopefully, none of them are faulty. Every voting system should first of all resolve the BGP.

Oral messages By assuming that generals and lieutenants communicate through oral messages, the following conditions are implied:

(A1) every sent message is guaranteed to be delivered correctly;

(A2) a message receiver always knows who sent it (meaning that the nodes form a fully connected network with unbreakable links);

(A3) the absence of a message is always detectable (the system is synchronous).

The initial problem can be solved by recursively assigning each general the commander role, while treating the others as lieutenants. Under these conditions, it has been shown that a solution exists only if $n > 3m + 1$, where n is the number of nodes and m of them are faulty. Obviously, the above assumptions are too strong for a realistic distributed system.

Signed messages Oral messages allow a traitor to lie about the decision he was told previously. But let's take in consideration an additional assumption, by introducing signed messages:

(A4)
1. a loyal general signature cannot be falsified, and any alteration of his message can be detected;
2. anyone can verify the authenticity of the general's signature.

If signed messages are introduced, a traitor can be easily detected, because his signature can be found on signed contradictory statements about $v(i)$. It has been shown that the solution in the case of signed messages, solves the problem for any number of traitors up to m, where m should be already known. Both solutions can be modified to have more relaxed assumptions about node connectivity, but they still operate in synchronous environments.

Asynchronous model When considering real world networks, the following considerations have to be made about the previous constraints:

(A1) correct message delivery guarantee is not achievable, but a missing message can also be viewed as one of the m traitor processes;

(A2) knowing the sender is already achieved with (A4);

(A3) detecting the absence of a message isn't achievable, it can be simulated via time-out, which in turn requires the following assumptions:

1. there is a minimum time for message generation and transmission, which is doable;
2. sender and receiver have synchronized clocks with a fixed maximum error, which is a problem of the same degree of difficulty as BGP [11];

(A4)
1. complete unforgeability of the general's message is not achievable, but the probability of a signature to be falsified can be reduced as much as needed;
2. message authenticity verification is completely achievable.

The above solution clearly doesn't fit asynchronous models. The most severe scenario is an asynchronous public network with arbitrary message delays, with no global clock and no preliminary knowledge about participants. It has been proved that no fully asynchronous consensus protocol can tolerate even a single unannounced process fault [12]. Even worse, the conclusion is made upon the assumption that the message system is completely reliable, which means that a message is always delivered and exactly once (with no time-out or order guarantee). It is clearly stated however, that the result doesn't show a *practical* solution impossibility. The asynchronous model rather requires more realistic assumptions and more relaxed requirements, resulting into new distributed computing models. Data consistency in distributed databases are affected by this conclusions, too. For example, a set of processes that participated in some transaction processing, should agree whether to commit the transaction result or not. All involved processes must agree on the final decision, in a fully asynchronous environment with no central component.

1.1.3 Practical Byzantine Fault Tolerance

It has been seen that a Byzantine Fault (BF) can be caused by unintentional malfunctions, or by a malevolent intelligence which tries to subvert the system. Byzantine Faults induce a subset of nodes to produce inconsistent outputs with arbitrarily long delays up to the infinite. Such events may harm the consistency requirement of a distributed system. BFs are expected to become more and more frequent with time, since software becomes more complex and malicious attacks bring higher reward if successful. A process is *stateful* if it is able to remember previous events, the information actually stored in its memory defines its *state*. State replication algorithms are used to implement resilient to BFs distributed systems. The property of a system to correctly survive a certain amount of BFs is referred as *Byzantine Fault Tolerance* (BFT).

The first attempt to introduce a state-machine replication protocol in a real-world like asynchronous model, has been proposed as *Practical* BFT (PBFT) [13]. The protocol allows to implement a deterministic service with a state and its operations, which allow to perform arbitrarily complex computations. Components within this model are connected by a network that may fail to deliver messages, delay them, duplicate or change their order. It is also assumed that nodes have distinct implementations of the same software and different administrators. Collision resistant hash functions are used to prevent message corruption, and each message digest is signed through public-key cryptography and attached to plain text. It is required that all replicas are aware of others' public-key. The adversary is assumed to be powerful enough to manipulate groups of faulty nodes, delay messages and correct nodes, but not forever.

The PBFT algorithm provides consistency in a fully asynchronous model. All client requests are totally ordered. The algorithm requires the same start state for all replicas, and a maximum number of allowed faulty nodes f. It operates according to the following principles:

- there is a set of replicas $R = \{r_0, ..., r_{n-1}\}$ in the network;

- each replica $r_i \in R$ is part of a *view*, such that:
  - each view has a *primary* replica, the others are *backups*;
  - replicas in the view have a index v, the primary replica has index $p = v \mod n$;
  - if the primary replica appears to be faulty, the view changes.
- the state of r_i is modified according to the following protocol:
  1. the client c make an operation request to primary;
  2. the primary assigns a sequential number to the request and broadcasts it (*pre-prepare* messages);
  3. if r_i accepts this number, it multicasts the event (*prepare* messages), offering to replicas the possibility to agree on total ordering;
  4. if r_i receives $2f$ prepare-messages, it multicasts the *commit* message, meaning that replicas agreed on total ordering;
  5. if r_i receives $2f + 1$ commit-messages, request is placed in queue to be executed;
  6. each non-faulty r_i sends the result directly to c.
- c accepts the result when at least $f + 1$ replies are identical.

It is proved that such an approach guarantees asynchronous consistency, usable in real-world applications. However, it happens if less than $f = \frac{n-1}{3}$ exhibit BF. The requirement to have $3f + 1$ replicas is indeed expensive. Synchrony assumptions are still required to achieve availability. In the case that the primary is faulty, c sets a time-out when making a request. If the time-out expires and there is no response, c will broadcast the same request to all replicas. After that, a change-view operation will be required as described in the protocol. Other important problems are related to message overhead and system *scalability*. A series of PBFT-like algorithms have been proposed through time, but they still require various forms of *weak* synchrony.

1.2 The Bitcoin Protocol

In public networks, an arbitrary number of participants might be spawned by an attacker to induce the system to behave incorrectly. Bitcoin behaves correctly as long as the majority of network computing power is offered by honest nodes. Next, in order to show why Bitcoin is reliable *enough* for financial transactions in such a severe environment, its consensus principles are provided. However, for a deeper view of the Bitcoin implementation, and how to actually use it, which is not the goal of this thesis, the reader can refer to [16]. In order to reach global ledger consensus, Bitcoin nodes perform the following independent, local steps:

1. transaction verification, performed by every node according to a well defined set of rules;
2. transaction aggregation into a new block and *Proof of Work* computation, performed by *miner nodes*;
3. candidate block verification, and their concatenation to the blockchain;
4. block selection, based on the highest cumulative Proof of Work chain version.

Each mentioned local step will be described, in order to see if and how termination; consensus agreement; fault tolerance; validity and integrity have been achieved.

1.2.1 Transaction Verification

The process of changing a fact in a ledger is called *transaction*. A common agreement about facts, how and when they change, should be reached. In *double-entry bookkeeping*, transactions are recorded in the ledger with *input* and *output* parameters. A basic Bitcoin transaction is a record that encode value transfer from a user to another. Each transaction is a new entry in this public, double-entry ledger which underlies Bitcoin. Transaction outputs are

amounts of BTC[1], recorded on the ledger and confirmed by the network. An unspent transaction output is called UTXO, and it is defined by:

- currency amount in its smallest unit, called *satoshi*;
- a *locking script*, or cryptographic puzzle, that a user must satisfy in order to spend the amount.

The script is written in the Bitcoin's stateless, stack-based scripting language called *Script*. A transaction input instead is made up of:

- one ore more UTXOs that the user is spending;
- a *unlocking script* which satisfies the above mentioned locking script.

Transaction inputs, are defined by which UTXO will be used and the user ownership proof. Every transaction is a Bitcoin state-machine transition. The Bitcoin state is defined by the *UTXO set*, which is the current amount of the UTXOs on the ledger. Suppose that Bob has 10 BTC and he sends 2 of them to Alice. This action will generate two transaction outputs: the amount sent to Alice (2 BTC) and the remaining amount sent back to himself. A wallet contains a collection of UTXOs that the user is able to unlock and spend. If a user makes a transaction through his wallet, it is then broadcasted on the network. Other nodes receive the transaction. But before propagating it again to its neighbors, they will first verify if the transaction is valid. The verification proceeds according to a strict set of rules which include, but is not limited to:

- transaction syntax and data structure correctness;
- transaction semantic properties, like input and output non emptiness, sum of inputs shouldn't be lower than output value, etc...;
- size in bytes smaller or equal to a parametric value (currently, 100byte);

[1] BTC will be used when referring to Bitcoin as a currency unit, Bitcoin will be used when referring to the protocol itself.

- locking and unlocking scripts related constraints;
- transaction fee verification.

If a transaction doesn't match all the requirements, it will not be propagated on the network. Otherwise, every node will receive it asynchronously. At this point however, a transaction is still *unconfirmed*. Each node maintains a set of received unconfirmed transactions called *transaction pool* or *mempool*.

1.2.2 Block Creation and Mining

Bitcoin network is a collection of peer-to peer nodes running the protocol on top of the internet. There are no hierarchies and each node runs several modular functions. One of these functions is called *mining*, and it is what allows to reach network consensus without a central authority. Valid transactions from local mempool are selected by miner nodes. This limited amount of transactions are then timestamped, by being putt in the same *block*. Each miner wants his block to be the next one attached to the *chain* of blocks that they have locally: the *blockchain*. This is what differentiates miners from other nodes, they propose *candidate blocks* to be appended next to the ledger.

Roughly, a block creation can be described as follows. After an empty block is created, a special *coinbase* transaction is added to it (which will be discussed in 1.2.4), this transaction doesn't require an `UTXO`. After that, a selected set of mempool transactions are added. On the blockchain, each block will contain the summary of all the transaction contained in, in order to efficiently prove that a given transaction tx_i has been included. For this purpose, *binary hash trees* are used (or merkle trees). A merkle tree is built by hashing each leaf tx_i, and by recursively hashing pairs of nodes $(h_{\llcorner n/2 \lrcorner}, h_{\llcorner n/2+1 \lrcorner})$, until a *merkle root hash* is obtained. Anyone will be able to verify if tx_i is included in the block in $O(\log n)$.

In order to have a winner candidate block, miners start a computationally expensive race between them, which consists into solving a mathematical

problem through iterative input trials. The solution to this problem is called *Proof of Work* (PoW), which is a hashcash based [4] mechanism. It is difficult to produce but easy for others to verify that the miner spent resources for. This is a round-like competition and each round length should be constant over years and decades. Therefore, PoW difficulty is dynamically adjustable by the system, in order to face the constantly improving hardware used by miners. The difficulty is given by the probability to obtain a solution, which is calculable prior to a range of consecutive rounds. The *block header* is filled by a miner with the following information:

1. protocol version that the node is running;
2. previous block hash (again, according to the local version of the ledger);
3. block creation timestamp;
4. merkle root hash;
5. target, which encodes the difficulty to find a solution;
6. nonce.

Roughly, the nonce is the input variation which allowed the miner to find the solution to the problem. Once the miner obtains the PoW after a solving session, he attaches it to the block and broadcasts it.

1.2.3 Block Verification and Consensus

If a miner is still solving the given problem, but receives from the network someone else's block that satisfies the difficulty target, he immediately stops. The miner knows he lost at this round. Beside starting to prepare transactions for the next round, a local validation of the received candidate block occurs. Similarly to a transaction verification, a node propagates it only if the candidate block satisfies several conditions. Blocks must have valid syntax, they must satisfy the difficulty target all included transactions

must be valid, and so on. Dishonest miners proposals are rejected, and they lose both the possibility to modify the ledger and their computational efforts.

It is still possible that several nodes may obtain a PoW a the same time, and all of them will be broadcasted. At this point, more nodes will surely start to disagree about the most recent blockchain history. Ledger copies aren't consistent because newly mined blocks arrive at different moments to each peer. Different versions of the blockchain are called *forks*. However, inconsistencies are temporary: each node will always prefer the blockchain version with the highest cumulative Proof of Work. Even if more than one node mined a block in a given round, some solutions are always harder to find than others. This is also know as *the longest chain*. In other words, given a round R, the winner block will always be the one with most expensive PoW.

1.2.4 Incentive and Network Attacks

It has not yet been said why someone should spend resources, trying to have the right to append a block to the chain. Mining is also a process of creation of new satoshi units. Miners receive reward in terms of `BTC` in two ways, they are indeed rewarded with:

1. new coins created with each new block, the amount produced will decrease over time down to 0, due to the deflationary nature of Bitcoin;
2. fees from all the transactions included in that block, approximately in year 2140 this will be the only miner reward system.

The first type of reward is given to the miner through the previously mentioned coinbase transaction. A coinbase transaction is always the first to be included. It doesn't consume any `UTXO` and creates `BTC` from *nothing*. Coinbase transaction output is the miners' address.

1.2.4.1 Attacking Bitcoin

The above summarized reward system stimulate nodes to participate, but its purpose isn't limited to that. The network is public with high levels of

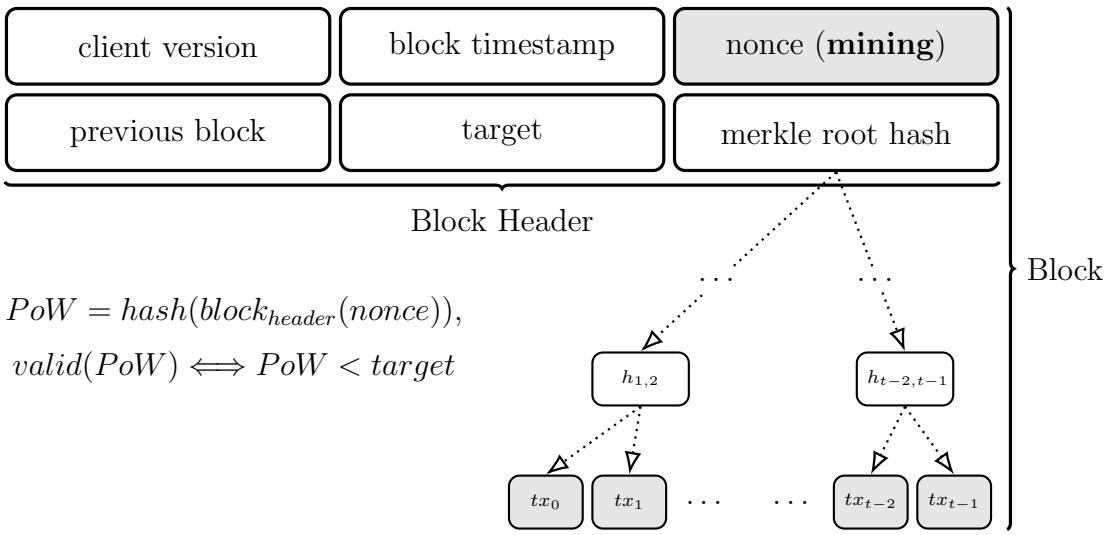


Figure 1.1: *Block structure and Proof of Work correctness, miners actively search a nonce such that the block header satisfies shown conditions.*

uncertainty, and consensus participants could be unknown entities. The PoW algorithm makes it more convenient for them to work according to the rules, instead of working *against* the protocol. Bitcoin and Bitcoin-like theoretical models are secure with adversary hash power strictly lower than 1/2 (in a byzantine-style notation, correct nodes should be more than $2f + 1$, where f is the computational power held by bad actors). Hash power requirements have been shown to be lower only in case that the adversary has the ability to delay messages for an arbitrarily long amount of time [30] (even infinite). With unbounded network delays consistency and *chain-quality* (blockchain protocol security properties 2.1.1) cannot be achieved.

An emerged problem is that a minority pool of miners could be incentivized to delay their own messages. Bitcoin is designed to reward miners in proportion of the overall computing power they control. A theoretically feasible strategy has been described that could help a mining minority group to obtain more reward than its power share [32]. This *selfish* miners, in other words, could keep private their version of chain if it is longer than the publicly shared one. After several rounds selfish miners will publish the whole

chain they kept privately, forcing honest miners to adopt it. Selfish miners will be able to claim all the coinbase and fees rewards, while others will have wasted their efforts. The proposed solution however, ensures that groups of miners with a smaller power share than 1/4 cannot engage this strategy. To easily detect selfish pools then, at least 2/3 of the network mining power must be owned by honest nodes.

1.3 Achievements in Asynchronous Systems

Upon a public asynchronous environment like Internet, Bitcoin Protocol [2, 16] prevents double-spending in a decentralized manner. All the nodes are aware that a transaction occurred and a coin is spent, being part of a confirmed block. All the following transactions of the *same coin*, made by the same user, will be rejected. Bitcoin doesn't contradict CAP Theorem 1, but manages to achieve all three properties in practical and probabilistic manner. A relaxed consistency assumption which fits asynchronous environments is introduced. Consistency isn't obtained immediately. There is a strong probabilistic guarantee that nodes will reach consensus in near future instead, while currently having both partition tolerance and availability. Academic efforts have shown that Bitcoin do achieve these properties both in formal synchronous and asynchronous environments [14, 30, 31]. These achievements rely on *practically hard to break* assumptions related to adversary computing power; arbitrary message delay possibilities or malicious alliances by groups of miners.

On the top of the Internet multiple protocols with different blockchains may exist, where the definition of a transaction and the architecture of the blockchain itself may vary. Therefore, each blockchain may have a different use case and digital currencies such as Bitcoin are only one of them. The previously introduced trust problem is solved by decentralized consensus, introducing the so called *trustless trust* architecture. Some might argue that this is a misname, and that it is actually a *distributed trust among nodes*,

or even a *system trust*. However what is important is that a blockchain user is now allowed not to know any of other participants, expecting the right outputs or the wanted service, with no intermediaries. Other promising features of this new paradigm are:

- configurable transaction transparency;
- ledger immutability;
- service availability;
- security;
- decentralization.

Alternative Consensus Models Bitcoin is based on the consensus algorithm called Proof of Work. After PoW however, other consensus models have been proposed and implemented in order to deal with public asynchronous networks. For example, in the Proof of Stake consensus model, the idea is that the rights to write on the ledger are given not by the author's computing power proportion, but in terms of his *stake* in the system. The principle is that the more an actor has invested in the network, the less he will want to break the rules. Some versions require that in case of bad behavior, the actor loses at least a part of its stake. One main advantage of PoS is that it doesn't consume energy. In Proof of Work the barriers to be enabled as a miner is given by the mining difficulty, while in PoS the barriers are given by the price of stake units. However, less reseach have been made in terms of PoS. Other consensus models are implemented in blockchain public systems, however, their full coverage is out of scope of this work.

Chapter 2

Blockchain and Distributed Ledgers

Bitcoin is the first specific implementation of a blockchain protocol. As a paradigm, this type of decentralization is already proving to be useful in different kinds of real world applications, not only financial transactions. When designing a blockchain, different use cases would require to enforce some paradigm properties, while not focusing so much on others. The advent of blockchain on public asynchronous networks however, has stimulated additional research efforts in *permissioned* consensus protocols, too. While the term blockchain refers indeed to *permissionless* distributed systems, where anyone can join and leave, *Distributed Ledger Technologies* (DLT) refers to permissioned systems. In a distributed system with a permissioned consensus algorithm, the transaction validator is an already trusted entity or a group of. In this thesis the main focus is on permissionless ledgers.

First, it will be seen how blockchains have been formalized as a concept, in order to understand its properties and possible architectural trade-offs. This chapter is going to be a summary of the main features and challenges that a blockchain architecture introduces.

2.1 Blockchain

Blockchain protocols work upon a relaxed form of consensus mechanism, called *T-consistency*. It requires that correct nodes agree on the chain structure, except for a potentially small amount T of the latest added blocks. Usually, these T blocks are called *unconfirmed*. T-consistency is the most appropriate form of consistency for blockchain protocols, with arbitrary message delays. However, when a block becomes *confirmed*, its placement on the chain will be permanent for all the nodes. Only the confirmed part of a blockchain is actually a chain, but from a global point of view, the unconfirmed part is a tree-like structure. In this section an already existing blockchain protocol definition [30] will be shown briefly, to understand how it would look like more formally. The minimal set of properties of a secure blockchain will be then provided. Each block can be defined as a triple (h_{-1}, η, m), where:

- h_{-1} represents a pointer to the previous block;
- m is the record component of the block;
- η a Proof of Work is derived both from m and h_{-1}.

This is in fact an abstract view of the block header described in 1.2, and the blockchain protocol itself is defined as in 1.

Definition 1 (Blockchain Protocol)**.** *A blockchain protocol is a pair of algorithms* (Π^V, C)*, such that:*

- Π *is a stateful algorithm that maintains a local state, receives a security parameter* κ *as input;*
- $C(\kappa, \texttt{state})$ *outputs* $\overrightarrow{m}$*, an ordered sequence of records* m*, that is the record chain;*
- $V(\overrightarrow{m}) = 1 \iff \overrightarrow{m}$ *is valid, where* V *is a validity predicate that has a semantic definition;*

The Bitcoin specific protocol is parametrized by an adjustable mining hardness parameter p, which defines the target $D_p = p(\kappa) * 2^{\kappa}$. A block can be proposed as candidate if, for a given η, $H(h_{-1}, \eta, m) < D_p$. H is a hash function and, as we have seen, its output should be *less than the target*. The record chain $\overrightarrow{m}$ is another way to say blockchain, and the validity predicate V might be the semantic definition of *no double spending*. (Π^V, C) blockchain protocol execution is determined by the joint view of all parties, denoted by $EXEC^{(\Pi^V, C)}(A, Z, k)$. Z is an execution environment which activates a number of participants as either correct or corrupted. A is the attacker that controls corrupted parties and sets the messages that they send.

2.1.1 Security Properties

It has been already said that PoW based protocols allow to achieve a certain level of consistency. However, T-consistency is insufficient to provide a *secure* blockchain [30]. A blockchain, in order to be *secure*, must have the following properties:

1. the already mentioned T-consistency, which means that extremely high probability, at any point, the chains of two correct nodes can differ only within the most recent T blocks;

2. *future self-consistence*, at any two time points r, s, with an extremely high probability, the chains of two correct nodes can differ only within the most recent T blocks;

3. *g-chain-growth*, at any point of the execution and with extremely high probability, correct node chains grow by at least T blocks in the last $\frac{T}{g}$ rounds, where g is the chain-growth rate;

4. *μ-chain-quality*, at any point of the execution, for any T consecutive blocks and for any correct node chain, with an extremely high probability, the fraction of blocks attached by honest players is at least μ.

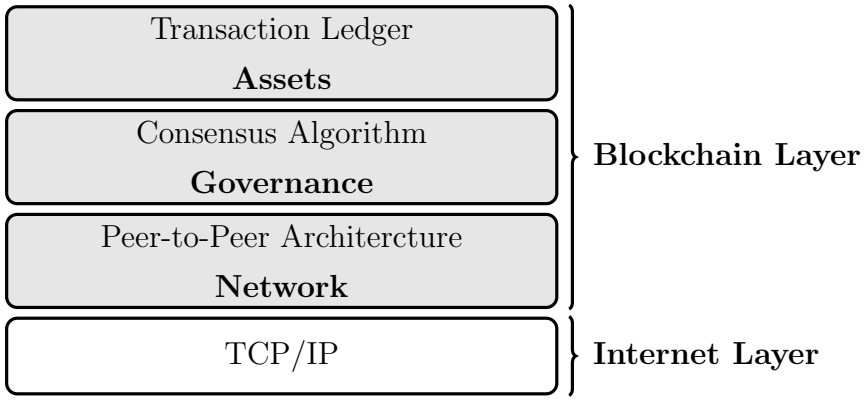


Figure 2.1: *Blockchain Technology Stack.*

The chain quality property ensures that the number of blocks attachable by A is proportional to its computational power. Chain growth guarantees that the chain grows proportionally with the number of rounds in a protocol. Future self-consistency is required in order to exclude such protocols that, for example, oscillates between two chains $\overrightarrow{m}_1$ and $\overrightarrow{m}_2$ on odd and even rounds. Authors show that with Bitcoin protocol all these properties have been achieved, under some value of mining hardness p, if message delays are Δ-bounded. Higher is Δ, higher is the probability of chain forks. Meaning that if messages circulate less, than it is more likely that nodes will have their own version of facts for a greater amount of time. Malicious behavior like selfish mining is possible with unbounded message delays 1.2.4. It has been shown too that with unbounded delays it is not required to own more than 1/2 computing power to attack the network. All of these four properties have been proved necessary to achieve consensus which simultaneously satisfies both *persistency* and *availability*. Persistency is the ability of a ledger to never suffer from an already added record deletion. While the already mentioned availability property guarantees that a network response is always provided.

2.2 Decentralized Systems

A system where its components process local data in order to receive global goals, has been defined as *distributed* in Section 1.1. The term *decentralized* instead, has grown in popularity since the advent of blockchain. It is often used to indicate that blockchain protocols are able to disjoint monolithic organizations into multiple independent entities that perform the same task. Both decentralization and distribution are often used as interchangeable concepts, but this habit is incompatible with arguments treated in this thesis. This section provides an attempt to make a distinction between them, and to define *decentralization* as for how it will be further used.

The first known time that terms *decentralized* and *distributed* have been used to express distinct concepts, while discussing network survivability, was in [34]. Roughly, a decentralized network has been defined as a hierarchical structure that is a mixture of a centralized network and a distributed system (Figure 2.2). By attacking the small number of nodes on top of the hierarchy all the system might result unavailable. The distributed version of the network is viewed as an ideal state of information and communication redundancy, with an optimal survivability in case of physical attacks.

In today's blockchain paradigm however, a distributed system can still be described as *centralized*. It is necessary to distinguish the centralization-decentralization spectrum upon multiple dimensions. A system can be decentralized[1]:

- *architecturally*, depending both by:
  a. the number of physical subsystems which is made up of;
  b. the degree to which the system is able to tolerate a subset of its components being down;
- *logically*, if the system does not require a single-state definition, and

[1]The Meaning of Decentralization (`https://medium.com/@VitalikButerin/the-meaning-of-decentralization-a0c92b76a274`).

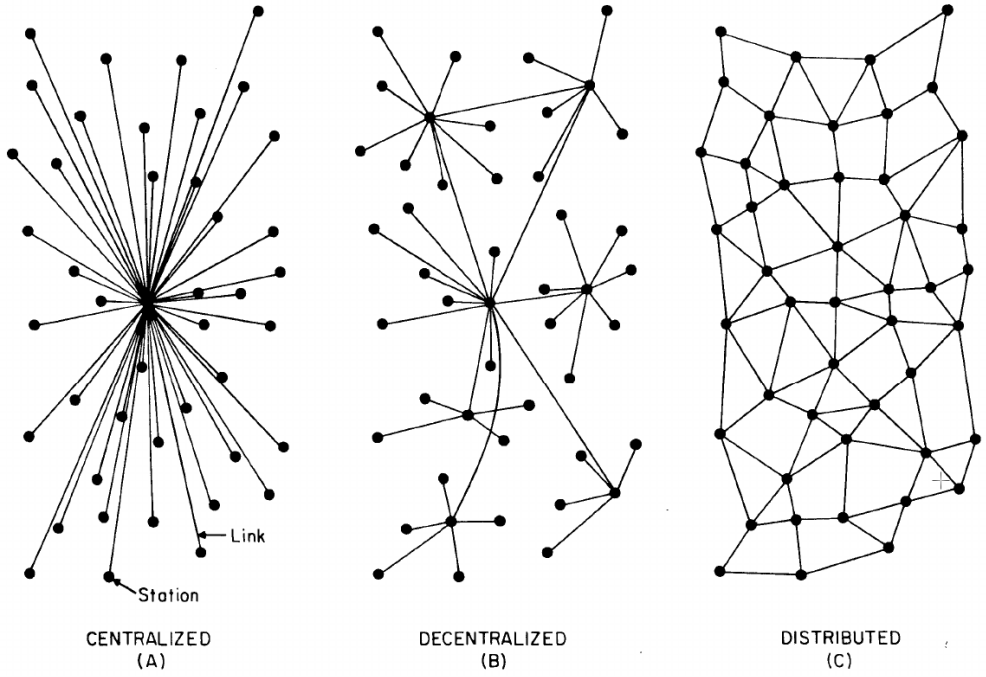


Figure 2.2: *Centralized, Decentralized and Distributed according to [34].*

acts independently, in the same manner, if divided in N parts;

- *politically*, depending on how many organizations are able exert a form of control over its components.

Bitcoin is architecturally decentralized, because it relies on multiple peers and there is no software or hardware single point of failure. Distributed systems are in fact *architecturally* decentralized. Bitcoin is also politically decentralized, since no single entity controls it, and anyone can join the network as a node. However, Bitcoin is logically centralized, because all the nodes have to agree on a unique state of the data structures.

A system can be more or less (de)centralized upon the above mentioned dimensions. An objective scale to measure the decentralization parameter of a system, would allow to compare multiple systems and to observe how much a system modification impacts on it. In an attempt to quantify decentralization the *Nakamoto Coefficient* [25] has been proposed. It is a Social

Network Analysis style measurement that could potentially give a pretty accurate view of a system centralization status. Given a system S, the method proposed in [25] requires to:

1. decompose S into a set of its relevant subsystems $S = \{sub_0, ..., sub_{n-1}\}$;
2. for each sub_j, determine $p_1 > p_2 > ... > p_{|sub_j|}$ such that $\sum_{i=1}^{|sub_j|} p_i = 1$, p_i is the power proportion of an actor i in the given subsystem;
3. for each sub_j, determine a power threshold th_j that no single entity should control in order to compromise sub_j;
4. for each sub_j, calculate the minimum number of its entities whose power proportions reach th_j;

$$N_{sub_j} := min\{k \in [1, ..., |sub_j|] : \sum_{i=1}^{k} p_i \geq threshold_j\}$$

5. the Nakamoto Coefficient of S will be

$$N_{min} := min\{N_{sub_0}, ..., N_{sub_{n-1}}\}$$

Nakamoto coefficient is the minimum number of actors necessary to corrupt in order to compromise a subsystem, and consequently the entire system. What nakamoto coefficient is saying is that a system is centralized as much as its most centralized subsystem sub_j. The main challenge introduced by this proposal however, consists into finding the right subsystems of S to analyze and its power thresholds. Especially when political decentralization is considered. For example, in a public blockchain the following subsystems might be detected:

- mining subsystem, governed by miners;
- repository committing power, induced by developers;
- client software subsystem;
- geopolitical subsystem, defined by geographic location of nodes;

- ownership, which describes the distribution of currency units among users;

- exchanges, or how many of them have relevant trade volume.

It is already clear why mining must be decentralized, where $th_{mining} = 0.51$. Distribution of commits is also important: as [26] points out, a blockchain project survivability is correlated with the amount of active committing developers. Developers shouldn't also belong to a single organization. Geographic location of nodes should be distributed, too. For example, a large amount of Bitcoin mining power is located in China. Some political decisions made by a country could harm system stability, if geographically centralized on its territory. Too much currency units (or any type of asset) controlled by a single entity could harm its decentralization. Despite not being part of the protocol, it is known that too much trading volume control exerted by a single exchange could harm the system, too[2]. As an example, some divergences are possible when talking about client software decentralization. Bitcoin's observed dominating mindset is to use a single main client (Bitcoin Core), upon which an independent number of developers work. On the other side, Ethereum (Section 3.2) actively promotes multiple client implementations in different languages (`C++`, `Python`, `Go`, `Rust`, etc...)[3]. This feature reminds the assumption seen in the PBFT proposal [13], which states that nodes should have different software implementations in order to avoid the same simultaneous fault. It can be said that political decentralization has a big influence, and quantifying it is a highly empirical work, based on Social Network Analysis tools.

[2]In 2014 Mt. Gox Bitcoin exchange was handling 70% of worldwide transactions, before finding out that approximately `850000 BTC` have been stolen over time from users. Currently the volume share is fairly distributed among many competitors (`https://en.wikipedia.org/wiki/Mt._Gox`). Today it is also possible to trade using Decentralized Applications (Chapter 3), with no intermediary.

[3]A list of Ethereum clients, proposed by distinct developer teams.`http://ethdocs.org/en/latest/ethereum-clients/choosing-a-client.html`

2.3 Decentralization, Consensus and Scale

At the time of writing this document, the Blockchain and Distributed Ledgers are relatively immature technologies. Currently, even the most advanced protocols are still under development and have a series of issues to be solved. The main challenges these decentralized protocols must face, are going to be described in this section.

Usually in a blockchain protocol all the nodes share the same state, and do process all system state transitions. The problem that emerges with this approach however, is that while providing *security*, it limits *scalability*. A blockchain protocol like Bitcoin cannot process more transactions per round more than a single node can. In order to increase the system throughput (transactions per second), one of the most common arguments would be to increase the block size, which is the number of transactions that fits in. But increasing the block size has a drawback in terms the above discussed decentralization. In this manner indeed, a network would be kept up only by computationally powerful players. These players would become also politically influential within the system. Other practical approaches to solve the problem bring us to the same *centralization* issue. The process of tuning a blockchain protocol parameter in order to make it operate better is called *reparametrization*. Intuitively, turns out that while designing a blockchain protocol, the following trilemma has to be considered: *a blockchain can have at most two out of the three properties of decentralization, scale and security*[4].

A more formal way to analyze the problem have been proposed in [33], with the above stated trilemma resulting into a theorem. In other words, a generic peer-to-peer system S that executes an abstract blockchain protocol, can have the following properties:

1. **decentralization**, guarantees that the system has no architectural single point of failure nor political single point of decision (architectural

[4]Ethereum trilemma introduction (`https://github.com/ethereum/wiki/wiki/Sharding-FAQ`).

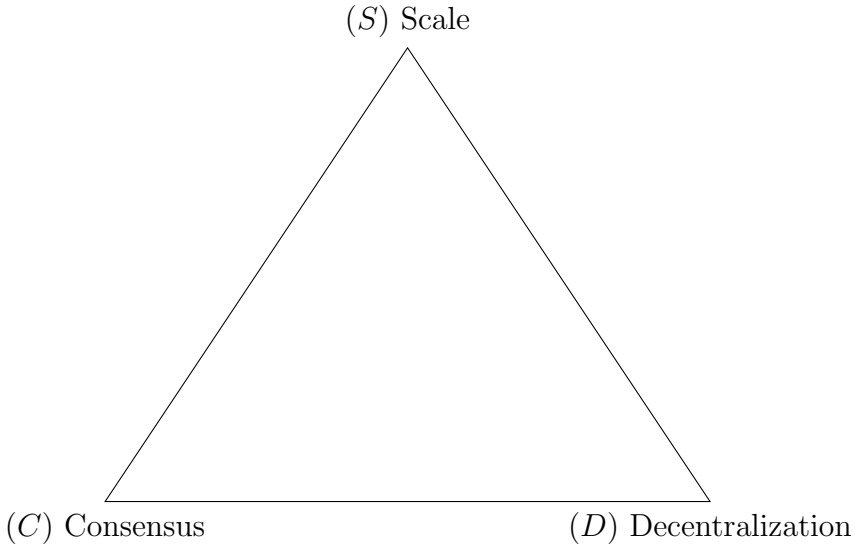


Figure 2.3: *The DCS Triangle.*

and political decentralization, Section 2.2);

2. **consensus**, network participants share the same system state, reached by consensus algorithms (T-consistency is implied, too);

3. **scale**, defines the capability of the system to handle a growing workload demand.

Scaling property of a system also implies that it is capable to satisfy the same transaction demand of any competing system, which provides the same service to the same set of users. The proposed theorem in [33] is intuitively represented by the DCS Triangle shown in Figure 2.3. Only two out of the three mentioned properties can be achieved.

Theorem 2 (DCS Theorem)**.** *Decentralized consensus systems centralize at scale when consensus participants maintain full consensus over the entire state of the system.*

The work [33] shows why the probability of such an outcome is extremely high. Its proof is built upon the assumption that in any sufficiently large pop-

ulation, individual access to computational power is distributed unequally. Another axiom is that the majority of participants do not have the required amount of computational power and storage capacity, to elaborate messages generated by a global scale user-base. These assumptions are empirically true. *Consensus participants* are defined as independent entities who maintain a complete copy of the system's state, and are part of a voting process to update this state. Let S be a decentralized consensus system whose consensus participants maintain full consensus of the system state. The same notation as in [33] will be used to define a system operational throughput.

Definition 2 (Computational Throughput)**.** *The computational throughput T(S) of a consensus system S refers to the rate at which the system updates its state by processing all input messages.*

A system throughput is determined mainly by the following factors:

- the size of quorum required in order to consider consensus reached;
- the computational power of each consensus participant;
- the time-out message period which has been set-up by the mechanism.

In a PoW consensus mechanism, the quorum is defined by the most computationally powerful users. For example, Bitcoin has a time-out period of 10 minutes, and slow participants are likely to hit it more often then others. $T(S)$ is fast as much as the slowest participant within the consensus quorum. As mentioned before, a typical way in Bitcoin to increase $T(S)$ would be to increase the block size (or other reparametrization solutions). But there would be less nodes able to include more transaction in a block within the same 10 minutes time-out.

The *coordination cost* $C(S)$, is defined as the difficulty for a participant to engage others and share the same goal. The higher is the coordination cost, the better is for the system S. It ensures that the probability of coordinated malicious behavior is very low. The author then, puts coordination cost

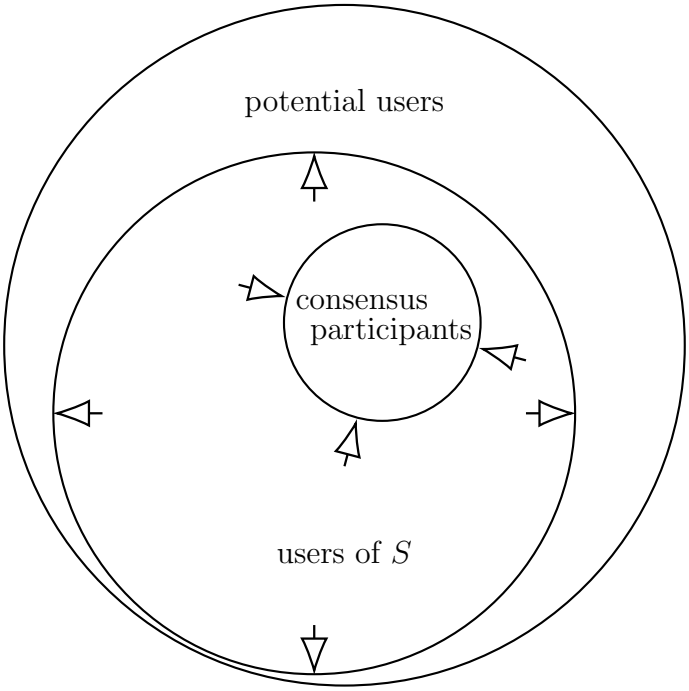


Figure 2.4: *Scaling user base brings transaction demand, but increasing throughput $T(S)$ implies that less consensus participants are able to (1) afford system state maintenance and (2) enter the deciding quorum.*

into a directly proportional relation with the number of system's consensus participants: $C(S) = |S|$.

The unequal computational power distribution among population, and the fact that most of users do not have a sufficient amount of it to maintain a global scale consensus, leads to the following conclusions. Let c refer to the average computational power of all participants that ever participated in a sufficiently significant series of consensus rounds. Then $T(S)$ exceeds c at scale proportionally to the new users of S. The subsequent phenomena is that at scale, the coordination cost $C(S)$ for consensus participants decreases. Low $C(S)$ leads to the risk of coalition based behavior from consensus participants. Unfortunately, single point of failure or political single point of decision is highly probable here. It is clear that an initial decentralized state of a system becomes centralized at scale. When maintaining the full system state, a system centralize at scale.

2.4 Scalability

It is clear that the main problem blockchain protocols must face is *scalability*, which is a factor that can make the difference between wide global adoption or a more limited use of it. The scalability challenge consists into finding a solution to increase the transaction throughput $T(S)$ at scale, without making it over-expensive for consensus participants and users. It has been seen that solutions such as block size increase and block interval reduction can be only taken as eventual preliminary steps in order to achieve a scalable system.

2.4.1 Blockchain Sharding

A way to get around the DCS triangle and increase $T(S)$ can be again, to rethink the consistency property. Using again the same terminology seen in [33], a system is DC if it focuses mainly on Decentralization and Consistency. A global scale system can be achieved by combining multiple DC systems

called *shards*. Each shard has its own group of consensus participants which trust other groups. Again, a inter-group trust mechanism can be achieved at system level, through various forms of transparency techniques. Obviously, a combination of DC systems has an overall weak form of consistency. Various challenges are introduced when high levels of inter-shard communication and consistency is required.

There are, however, distinguishable types of sharding. The first type can be referred as *state sharding*, where each shard maintains a portion of the global state (storage). Multiple sharding-enabled systems are in development phase, at time of writing. Ethereum (which will be seen in Chapter 3) will eventually implement state-sharding in the future.

Another type is the *computational sharding*, and an example of such a system will be briefly shown. Zilliqa platform [45] introduces the idea of dividing the network into smaller consensus groups that process transactions in parallel, with no overlaps. Each group is called shard. Zilliqa protocol uses Proof of Work only for sybil-attack prevention. Consensus is instead achieved through a PBFT-style improved algorithm (Subsection 1.1.3) with communication complexity $O(n)$ in the normal case, and a constant $O(1)$ signature size. It has indeed a two-layer blockchain structure that is out of scope of this work.

2.4.2 State channels

A system is DS if it is designed in order to maintain Decentralization and Scalability. Let *main-chain* be a DC system such as Bitcoin. A *side-chain* is a DS system that allows the main-chain to increase $T(S)$ at scale, letting it to preserve its T-consistency. The idea is that the side-chain is used for updating and processing transactions off the main-chain [46]. But when some desired state is finalized, it is written to the main-chain. Both systems work together in such a way that consensus is required less often, thus unloading time and power-consuming operations from the main-chain. A transaction on the main blockchain can encode a state-channel birth between two or more parties and

another one its end, while a smartcontract (Chapter 3) can be used to enforce the rules of participation. When a state-channel is closed and side-chain communication ceases, the state modification induced by this interaction is written on the main blockchain. This approach can be very useful because (i) a few transactions are able to encode multiple transactions and (ii) it results in a fee reduction when performing micro-payments. However, the reduction does not occur when the fee is also a function of (a) the amount bytes written on the blockchain and (b) the computational complexity of a transaction. As we will deduce after reading Chapter 3 and Chapter 4, since in Ethereum there is a *block gas limit* that measures (a) and (b), multiple smartcontract deployments couldn't be encoded within state-channels, even if it makes sense.

2.5 Anonymity and Zero Knowledge Proofs

When designing a secure public blockchain, confidentiality questions are also necessary to answer. It is often required to be able to chose what information to keep private or public. A basic blockchain protocol which relies on asymmetric cryptography provides pseudo-anonymity. Meaning that for each transaction, the identity of the user that generated it, is tied to a public key or *address*. For technical or personal reasons, the user (or its wallet) can potentially generate as much addresses as necessary, in order to avoid traceability from third parties. In the Bitcoin Network, however, it has been shown that:

- it is possible to associate identities to blockchain addresses [35, 36];
- to classify both identified and unknown user's behavior [36, 37];
- even to perform Social Network Analysis [38].

These tools might be also used as an attack vector to real identities and miners. Attacks like network partitioning and message delaying (Subsection

1.2.4) can be performed if coordination cost $C(S)$ of a system S isn't high (Section 2.3), and especially if transactions are accessible as plain data [39, 40].

While being able to verify transaction correctness and reach system consensus, it is often required the ability to hide its recipient, sender and content. Such advantages are offered by a recently proposed cryptographic tool called *zero-knowledge Succinct Non-interactive Arguments of Knowledge* (zk-SNARKs) [41], that will be briefly explained below. Let L denote a NP language and P a program (circuit or universal function) in L. Let n be the dimension of input instances acceptable by P. Given the input instance x such that $|x| = n$, a peer $s_1 \in S$ broadcasts that $x \in L$. Another peer s_2 receives the statement $x \in L$ and he wants to know if the statement is true. Given this peer-to-peer system within which no one knows each other, zk-SNARK is a *non-interactive* and *succinct* proof π which proves that $x \in L$. Non interactive because s_1 and s_2 never interact and they don't know each other, and succinct because it should be very easy to verify the statement:

- s_2 is protected because he is able to verify if $x \in L$ through π;
- s_1 is protected because s_2 is able to verify only $x \in L$, but not the source s_1 or other information.

In order to achieve such a system, it must be built as follows:

1. a trusted peer s_0 takes the program P and performs an initial system setup;
2. the system setup produces two public keys, proving key pk and verification key vk;
3. then, an untrusted peer s_i uses pk to create a proof π, that proves $x \in L$;
4. other untrusted peer will use vk to verify π, preserving full anonymity as stated above.

Succinctness property requires $|\pi|$ to be constant, and π verification cost is $O(n)$. zk-SNARKs have been used in a blockchain protocol proposal known as *Zerocash* [42], and later implemented as *Zcash*[5]. More practically, zk-SNARKs are proofs that s_j performed some computation over an input, but without revealing the input. Miners then, will verify the transaction as for Bitcoin, but without any type of knowledge about the transferred amount and addresses. In Zcash, each user is provided with two addresses. `z-addr` is a fully anonymous address and it is used in transactions as described above. `t-addr` is a bitcoin-like pseudonym. Each user is able to chose what transaction to keep fully anonymous.

The problem afflicting zk-SNARKs is the requirement to have a trusted peer s_0. Even assuming that s_0 is honest, an attacker could possibly corrupt the initial system setup. Therefore, generated keys might be corrupted too. There is no way to mathematically verify that s_0 is actually honest, and that the sensible setup information have been destroyed after the process. For example, a malicious party could use this information to *issue* new units of currency in the system. *Zero-knowledge Scalable Transparent Arguments of Knowledge* (zk-STARKs) instead [43] have been proposed as proofs that could potentially replace zk-SNARKs, eliminating the necessity for s_0. zk-STARKs allow to have a system with at least the same features zk-SNARKs provide, with no initial setup and in a trustless manner. Currently however, the main drawback is that it a zk-STARK proof π is approximately 1000 times larger in size than a zk-SNARK.

2.5.1 Scalability with Proof Systems

As authors point out, zk-STARKs and other proof systems could be also used to reduce scalability issues by exponentially decreasing transaction verification time. Some nodes could integrate a *prover function*, that will produce proofs in quasi-linear time. These proofs can convince the network to accept a current ledger state as valid, avoiding several expensive factors:

[5] `https://coinmarketcap.com/currencies/zcash/`

- to re-execute identical computations that already occurred elsewhere, in order to achieve the same state;
- to store the entire blockchain state on each node.

2.6 Distributed Ledger Technologies

Having said that the work in this thesis covers permissionless protocols in asynchronous networks, a brief introduction will be given about alternative forms of distributed ledgers. In a permissionless setting like Bitcoin, anyone is enabled to start a public node by downloading the publicly available code. Anyone capable of, is able to validate transactions and participate into consensus processes. Transaction are transparent but with potentially hidden content. The main advantage of public ledgers is the ability to eliminate intermediary entities, to lower or eliminate infrastructural costs through decentralized applications.

Several ideas behind blockchains could however be used with restricted access to a limited amount of known entities. Since these entities trust each other and are co-interested to run the network and its protocol, there is no need to have incentive mechanisms. These systems type are indeed *coinless*, and are referred with a generic term *Distributed Ledger Technologies* (DLTs). A type of DLT are the *Federated Ledgers*, consensus is maintained by a group of pre-selected set of nodes, and the access to read can be restricted. *Private Ledgers* instead, have its write permissions assigned to the entity which created it, for internal business logic purposes.

Open source distributed ledger projects[6] can also be offered *as a service*[7]. Hyperledger Fabric is a distributed ledger framework implementation, which that allows consensus models and internal services to be modular. Related projects such as Hyperledger Composer instead, can be viewed as a language to specify ledger actors, assets and logic. Since decentralization is heavily

[6] `https://www.hyperledger.org/`

[7] `https://developer.ibm.com/blockchain/sandbox/`

sacrificed in these types of system, the main advantage is that they can easily scale.

Chapter 3

Decentralized Computing Paradigm

The advent of blockchain allowed to experiment more sophisticated versions of such protocols. It is becoming more clear that multiple protocols could coexist on the top of Internet, bringing new possibilities when interacting with the *Web*. When it was born, the Web was mainly composed by read-only architectures, where the user had a one-way communication with servers, and the ability to make a request and receive some sort of information viewable in a browser. Today it is often referred as *Web 2.0*, because new features have been added to it: search engines, social networks, user-generated and uploadable content, the ability to use tags or extensions, and so on. Especially, Web 2.0 is today an application platform. Such web applications are kept up by centralized intermediary entities, operating between multiple parties involved a process. The term *Web 3.0* instead, is often used in contexts like Semantic Web, Artificial Intelligence, Internet of Things and even Virtual Reality. In the Blockchain Paradigm however, which by the way merges very well with the previously mentioned disciplines, Web 3.0 refers to a next generation decentralized web. It allows to run *Decentralized Applications* on *Decentralized Computing Platforms*, and to govern *Decentralized*

Autonomous Organizations. Direct connection between involved parties may exist in a business process, allowing to skip intermediary entities in a lot of case scenarios. This property helps to eliminate single points of failure, expensive commissions, total data control by third parties, and breaks new ground to emergent business models. This chapter aims to provide the basics for decentralization as a paradigm, and how it could be used to define new application types and organizations.

3.1 Smartcontracts

The most common way to formalize voluntary relationships between two or more parties, whether enforced by a government or otherwise, is through a *contract*. Centuries of cultural evolution and experience have given birth to contracts and principles affiliated to it, which are the basis of a market economy. The digital revolution however, introduced new types of relationships between parties around the world. Digital media are indeed dynamic, it means that beside vehiculating multi-sensory information, they are able to make different kind of decisions. This idea that a transition is needed from the traditional static paper-based solution, to a model where contractual clauses can be embedded into software, called *smartcontracts*, was for the first time discussed in [7] and rewritten in [8]. A crucial requirement for smartcontracts is to be prohibitively expensive to violate for possible attackers, and this is the exact feature that a blockchain provides. The main concept behind the term smartcontract has slightly changed over time however, which is now often used to indicate a generic portion of code that resides on a blockchain network. A portion of code which is identifiable by a unique address, defined as a set of state variables and functions, and executable when transactions to its address are made. Alternative terms such as *chain-code* might be also used for. The source code is viewable by anyone involved, so *code is law*: many use cases doesn't require a third party to enforce it.

In [44] and abstract definition of a smartcontract has been given, which

includes both *smart contract code* and *smart legal contract* concepts. In this thesis however, despite using similar terms, a smartcontract will be referred as a smart contract code which resides on the blockchain. Such a smart contract has the following ideal characteristics.

- *enforceable*, which means that the smartcontract is always executed as defined, with deterministic behavior and no mediation;
- *automatically executable* when required conditions are met;
- *tamper-proof*, meaning that once started, its execution is unstoppable and secure (against BFs in its environment).

A smartcontract is enforced by the underlying blockchain protocol. It is automatically executed when triggering network transactions occur. Tamper-proof property on the other side, isn't a constant and may vary to certain degrees among implementations. Another interesting property which is not treated here, is the ability of a smartcontract to be understood by all humans, too. If humans and machines can both understand a smartcontract, it is more likely that a smartcontract will be valued in legal and social contexts.

Definition 3 (Smartcontract)**.** *A smartcontract is a tamper-proof computer program which implements an automatically executable agreement, enforced by the underlying decentralized consensus protocol.*

Stateless and Stateful A smartcontract can be *stateless*, meaning that it has no internal state. A form of stateless contract are the Bitcoin's scripts: they keep no data during execution. A *stateful* contract instead, can be expressed through more powerful constructs like loops, which allow to maintain an internal state.

Vulnerabilities Smartcontracts could also have exploitable vulnerabilities, like any other software. Attack vectors are more dangerous on stateful smartcontracts where definition language provide complex high level constructs.

Oracles and Contract Incompleteness A *complete* contract is an ideal type of contract which encodes what is going to happen in every possible scenario. On the other side, an *incomplete* contract depends on circumstances and even real-time variables, which often makes them renegotiable in case of unexpected events. These type of contracts has always been expensive to execute. However, smartcontracts lower the transaction and information cost related to incomplete contracts, making it accessible in more use-cases and for a larger set of organizations. An *oracle* is an entity, or a group of entities, that converts real-world information into data that can be processed by a smartcontract. The main goal of an oracle is to provide sufficient contract completeness, in order to make them definable in an algorithmic way, which takes the form of a smartcontract. In other words, an oracle is a connection between real world data and blockchain decentralized logic.

3.2 The Ethereum Implementation

As a result of various attempts to integrate the above principles into the blockchain, the concept of a *decentralized, programmable and transaction-based state machine* has emerged, resulting into the Ethereum proposal [15], and into its further formalization [17]. In other words, Ethereum is a decentralized computing platform and smartcontracts are its programs, currently writable in a turing-complete language, and runnable by the Ethereum Virtual Machine on each node. Ethereum smartcontracts are indeed stateful. Bitcoin functionalities are implementable as an Ethereum application: with its expressive power indeed, it contains the Bitcoin stack-based scripting language. This section describes how Ethereum works and how its Smartcontracts can be defined, deployed and executed.

3.2.1 Ethereum

With *Script*, Bitcoin allows to implement very weak forms of smartcontracts. For example, an `UTXO` (Subsection 1.2) can be owned by multiple

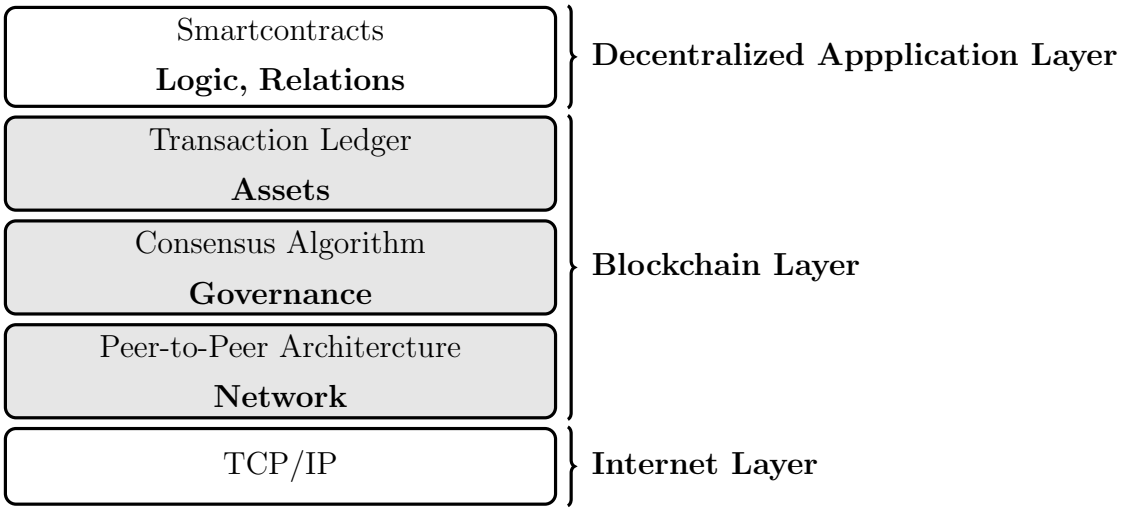


Figure 3.1: *Blockchain and Smartcontract Technology Stack.*

parties and unlocked only if all them agree to do it [16]. It is a stateless language with limited expressive power, which has its advantages: no consensus participant has to remember something and send to others; less attack vectors; fast execution with guaranteed termination, and it is sufficient to perform payments. However, beside lacking higher level constructs like loops, Bitcoin doesn't allow the user to access blockchain variables, to interact with its state, neither interact with some form of real-world data.

In order to create a programmable blockchain, Ethereum [15] provides a built-in turing-complete programming language that is used to write smartcontracts. Smartcontracts can be defined and deployed by anyone, in order to govern a process according to their own rules.

Account types Ethereum has two account types:

1. Externally Owned Accounts (EOA), which is created and controlled by a user through public and private key-pair, it is defined by its address, the account keys and the `ETH` balance;

2. Contract Accounts (CA), where smartcontracts reside, are defined by

their address, the contract code and `ETH` balance, too;

Any user can create an EOA. When the user starts a transaction in order to deploy a smartcontract on the blockchain, a new CA is created. In order to trigger the same smartcontract, a transaction (with eventual parameters) to its CA address has to be made. The right to trigger a smartcontract code is programmable, too.

3.2.2 Smartcontracts in Solidity

Several languages can be used to define an Ethereum contract, and every definition can be compiled into *bytecode*. Bytecode is executable by *Ethereum Virtual Machines* (EVM), that each full node runs. Initially available only as a proof-of-concept, the first one was the *Low-level Lisp-like Language* (LLL). *Serpent*[1] is a Python-like language that can be used for the same purpose. But the main language which is used to defined smartcontracts is *Solidity*. With its *contract-oriented* paradigm in a `C++` style, a `Solidity` smartcontract reminds an object-oriented class definition. A smartcontract is composed by:

1. state variables, contract storage variables used to maintain the its state;
2. events, which can be used to track a contract execution, even from outside Ethereum;
3. modifiers, used to change function behavior, they often check conditions (like contract ownership) before executing a function;
4. functions, which include the code executed when transaction to CA are sent.

Every smartcontract has also a constructor. Contracts can be compiled and deployed as we will see in chapter 4.

Ethereum transactions modify the network state. Since Ethereum has a built-in turing-complete language, several precautions have to be made in

[1] `Serpent` (`https://github.com/ethereum/wiki/wiki/Serpent`).

Unit	wei	babbage	lovelace	shannon	szabo	finney	ether
Value (`wei`)	1	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}

Table 3.1: *Denomination of Ether subunits and value.*

order to avoid infinite loops on the network. A smartcontract computational complexity is measurable in `gas` units. In fact, each atomic operation consumes a certain amount of `gas`. A contract function is a combination of operations. Each execution has a `gasLimit` in order to avoid a variety of undefined behaviors. Beside specifying `gasLimit`, a user also tells how many `wei` (the minimum sub-unit of an ether (`ETH`), Table 3.1) he is willing to pay for a `gas` unit. The final transaction fee will be $tx_{fee} = gas_{consumed} * gasPrice$. In other words, the amount of `gas` specifies the transaction complexity, while `gasPrice` is used to determine how fast the computation will happen. A transaction has also a `data` field, which contains the transaction parameters. A transaction `data` field contains a smartcontract bytecode for example, when a contract is being deployed on the blockchain. Transactions to EOA are fund transfers, a transaction to a CA are a method invocation.

Calls can be used for read-only operations on the network and don't require fees. Calls don't modify the network state. All smartcontract functions which do not modify its state are callable for free.

3.2.2.1 Other Smartcontract Concepts

Roughly, Solidity is an object-oriented language, where an object always belongs to a contract class. Upon decentralized environments however, other paradigms could be used in order to define a smartcontract. An example will be given. The Zilliqa platform [45], that has already been said to implement the so called *computational sharding* (Subsection 2.4.1), uses *dataflow programming* for its smartcontract layer. A smartcontract is represented by a directed graph, where a graph node indicates an independent execution in the program (functions and operations). When computing the smartcontract,

each graph node can be assigned to a different shard. This approach could guarantee a powerful decentralized parallel computing platform. Again, at time of writing, this and other tools are still in early development phase.

3.2.3 Oracles

Oracles have been presented as entities which provide a link between smartcontracts and real-world data (Section 3.1). This link is often required because of a contract incompleteness property. This missing data is provided through APIs. An oracle can be *centralized*, too, which means that a single organization provides content for smartcontracts residing on blockchains[2]. Some might argue that a single organization could provide arbitrary data to smartcontracts. In this case, oracles can be also decentralized. A *decentralized oracle* is an ecosystem of potentially competing data sources[3][4] which feed smartcontracts. Each decentralized oracle node is incentivized to do so in a Bitcoin-like manner. Also, different mechanisms are introduced to heavily discourage faulty behavior, like bringing false information.

3.3 Decentralized Logic, Storage and Messaging

A global scale decentralized computer, as a vision, not only relies on a *decentralized logic*, which the smartcontracts are. Communication protocols as *decentralized messaging* [22], and peer-to-peer, self-sufficient *decentralized storage* networks [18, 19, 20, 21], both with a built-in incentive system, are also required. Once combined, these complementary technologies allow the

[2] Oraclize is an example of organization providing data to decentralized applications (`http://www.oraclize.it/`).

[3] *Chainlink* is a decentralized oracle which aims to feed Ethereum smartcontracts with requested information (`https://www.smartcontract.com/`).

[4] *Mobius* is a decentralized application marketplace which integrates decentralized oracle principles (`https://mobius.network/`).

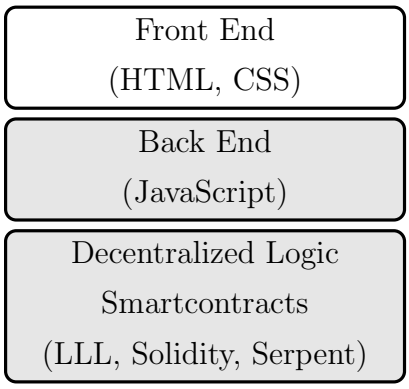


Figure 3.2: *A Decentralized Application Architecture.*

full operativity of ecosystems of *Decentralized Applications* (dapp), as in Figure 3.2. Decentralized applications existed before the blockchain advent too, defined as applications which runs on top of a peer-to-peer network. A dapp, as implied further in this thesis, has its own logic defined by a set of smartcontracts that reside on the blockchain (with some variations), that use decentralized messaging protocols to interact with each other, and retrieve data from underlying decentralized storage. While eventually having a presentation layer deployed on a central server, which is the only concept in common with an app, to *deploy a dapp* will usually indicate running its parts on a decentralized platform. For completeness, it can be said that dapps are deployable whether including a proper *token economy* [24] or not. In this way, in now avoidable requiring a large number of separate blockchains for each dapp, too expensive for small organizations.

Decentralized applications can be developed to operate in a wide range of domains. Some examples are:

- internet of things, which refers to devices that are connected to internet and have unique identities. IoT devices can exchange or collect data using blockchain based applications, with a smartcontract regulated behavior. Some concrete examples are connected vehicles, smart home devices, locks (which can be used to rent real-world physical objects)

and parking;

- industry and manufacturing, decentralized applications can be used for on-demand manufacturing (like 3D-printing), product traceability and shipment, supply-chain tracking (for example, a pharmacological product can be traced, to certify that it has not been exposed to dangerous temperatures), product certification and so on;

- financial technology, decentralized applications could be used for payment platforms [5], insurance platforms, trading, investments and banking.

- records and identity management, dapp can be used to certify document authenticity and their existence at a specific time point in history, marriage and birth certificates, copyright protection and land registry.

3.3.1 Decentralized Autonoums Organizations

Dapps can be used to run *Decentralized Organizations* or *Decentralized Autonomous Organizations* (DAO). Both of them can be viewed as organizations whose governance model is implemented as dapps, but unlike a DO, a DAO is fully automated and does not require any input to execute the logic. Successful public DAOs have been shown to be highly correlated with *community* support, where none of multiple parties involved produce the vast majority of commits [26]. Bitcoin is, actually, the first form of DAO ever seen. Company-owned blockchain projects also started to proliferate, especially through *Initial Coin Offerings* (ICO) [27], which is a process of generating tokens and distributing them to parties, in return for the platform's native token or other currencies. Users are willing to buy these tokens, if they *value* a product that will exist on a decentralized network, or will be tied to it. While the necessity of an ICO might be questionable in some cases, as well as the blockchain integration, new business models are now

[5] An example of such a platform is Request Network (`https://request.network/#/`)

definable upon token economies, otherwise prohibitive to realize [28]. Later, the idea of a *Decentralized Autonomous ICO* (DAICO) to be developed, has also been proposed [29].

Chapter 4

Decentralized Certificate Issuer

This chapter will define the specific problem of how to write blockchain data in such a way that (a), it is enabled to certify a generic occurred event and (b) to be verifiable by anyone, anytime, in a trustless manner. Implementation cases will be provided for public ledger platforms. Since permissioned and federated ledgers require all involved parties to trust each other and to actively participate in the network, these type of systems aren't suitable for the problem given here.

4.1 Problem Definition

The problem requires to realize a decentralized application that is able to issue *certificates* on the blockchain. At any point in the history, no trusted third party should be needed to verify an issued certificate authenticity. It is already known that a blockchain is a public, immutable and ordered distributed ledger. It provides all the necessary features to realize such applications.

Let's assume that organization I is the certificate issuer. Let $C = \{c_0, ..., c_{n-1}\}$ be a set of entities authorized to interact with I. Given a specific information $info_{c_i}$ that c_i has, c_i should be able to ask I to perform an event $E(info_{c_i}, attach_j)$. $E(info_{c_i}, attach_j)$ might be some service that

I offers, where $attach_j$ is a proprietary information that initially belongs to I. The content of $info_{c_i}$ is irrelevant in this scope. At any point in the history, I and even a subset of C might disappear. However, any entity even outside $C \cup I$, should be able to verify that $E(info_{c_i}, attach_j)$ actually happened somewhere in the past. c_i will ask I to perform E multiple times, so there will be a sequence $(info^0)...(info^{T-1})$ for each c_i. For every request instance, I must perform the event and issue a blockchain certificate $cert(E(info^t_{c_i}, attach_j))$, such that:

(R1) with no trusted parties and whenever, anyone should be able to verify that for any i and t, $cert(E(info^t_{c_i}, attach_j))$ exists and $attach_j$ is unique.

The constraint (R1) ensures that I will not use its proprietary information $attach_j$ for more than a single certificate. Such a mechanism has multiple use-cases. For example, a University could use it to certify someone's academic achievements. C could be its students, $E(info^t_{c_i}, attach_j)$ could be its graduation or intermediate courses, and some company could be interested to check authenticity of $cert(E(info^t_{c_i}, attach_j))$. (R1) becomes more important in other cases, where let's say, $attach_j$ represent the identifier of a very expensive product.

There is another factor to take in consideration. Since nodes must be kept incentivized in order to run a protocol, blockchain services have a transaction-based cost in terms of the platform's native coin. These coins must be bought by I on pairs like `<COIN>/USD`. DLT services on the other side, might be additionally offered *as a service*, with time-based payments. A solution must therefore be considered in terms of a second constraint:

(R2) each certificate deployment process performed by I should have a *reasonable* operational cost.

Additionally, not too slow deployment processes are wanted.

(R3) each certificate deployment process performed by I should be confirmed by the ledger after a *reasonable* amount of time.

The practical work described in this chapter, aims to answer whether a dapp that solves this problem is already implementable today or not. A dapp that provides (R1), (R2) and (R3). And if not, what is the likeliness that it will be achievable soon.

4.2 Dapp on Stellar

Stellar is a micro-payment oriented platform. With its federated internet-level consensus model [47] it is relatively fast, and satisfies both following constraints:

(R2) is satisfied, transaction are a combination of at most a few operations, which have an extremely low cost of `USD 0.000004`[1];

(R3) is satisfied, because Stellar network usually reach consensus in less than 5 seconds.

Constructs with full expressive power like turing-complete smartcontracts are not currently offered by this platform, and likely will never be. Stellar relies on simplicity and speed given by light-weight transactions, which in turn are made of simple declarative operations. This approach requires the issuer organization I to create a Stellar account for each customer c_i, or an account ownership by c_i. Then, each deployed certificate to c_i is represented by a transaction to its account, with incorporated information that encodes $cert(E(info^t_{c_i}, attach_j))$.

JavaScript code in Figure 4.1 is given by an attempt to realize such a dapp. As an architectural choice, in order to guarantee high throughput, Stellar allows to only attach some very small data to its transactions. There are two attachable data types we can be interested in[2]:

[1] At the beggining of year 2018, Stellar basic operation fee is `XLM 0.00001` = `USD 0.000004` (`https://coinmarketcap.com/calculator/`).

[2] Stellar Documentation (`https://www.stellar.org/developers/guides/concepts/transactions.html`).

Figure 4.1: *Part of the JavaScript code that allows to perform a payment with attached data, using `StellarSDK`.*

```
var server = new StellarSdk.Server(horizonTestnet);
StellarSdk.Network.useTestNetwork();

window.app = {
  generateCertificate: function () {
    var cert = {'info': 'Rafaello', 'attached': 'id'}
    var hash = crypto.createHash('sha256');
    hash.update(JSON.stringify(cert));
    var memoHashHex = hash.digest('hex');

    return memoHashHex;
  },

  issueCertificate: function () {
    var certHash = this.generateCertificate();
    var issuerPair = StellarSdk.
                        Keypair.fromSecret(issuerPriv);

    server.loadAccount(destPub)
      .catch(StellarSdk.NotFoundError, function (error) {
        throw new Error('Dest. account does not exist!');
      })
      .then(function() {
        return server.loadAccount(issuerPub); })
      .then(function(sourceAccount) {
        var transaction = new StellarSdk
            .TransactionBuilder(sourceAccount)
            .addOperation(StellarSdk.Operation.payment({
                destination: destPub,
                asset: StellarSdk.Asset.native(),
                amount: "1"
            }))
            .addMemo(StellarSdk.Memo.hash(certHash))
            .build();
        transaction.sign(issuerPair);

        return server.submitTransaction(transaction);
      })
      .then(function(result) { /* result handling */ });
      .catch(function(error) { /* error handling */ });

    return;
  }
}
```

- `MEMO_TEXT`: a string encoded using either `ASCII` or `UTF-8`, up to `28 bytes` long;

- `MEMO_HASH`: a `32 byte` hash.

`MEMO_TEXT` is too short to allow the dapp to attach useful information in most applications. `MEMO_HASH` could be an elegant solution to link arbitrary big data to a transaction. The function defined at line `5` in Figure 4.1, converts a test `JSON` to a hash string. When building a transaction (line `26`), the data hash can be added to it (line `33`). Unfortunately, this is insufficient to fully guarantee (R1). An external entity that wants to verify a certificate authenticity, should retrieve first the certificate plain data from a trusted party (I itself or some other trusted party). After data retrieval it would be able to hash the certificate plain data, with the same algorithm. Only then, the external entity would be able verify if the hash it has, matches with the hash on the ledger. Transaction time confirmation and its cost satisfies very well other constraints. However, Stellar is a good choice only if involved parties are willing to sacrifice (R1).

4.3 Dapp on Ethereum

Ethereum allows to implement fully on-chain dapps, without third party support in a wide imaginable use-cases. Explicit information can be stored directly on the platform. In this way, given its immutability property, certificates would have blockchain-lifetime. Even in the worst case scenario with all the Ethereum nodes going down, with C and I disappearing, the ledger will continue to exist and computation might restart from a given commonly agreed block. The dapp implementation and the possibility of (R1) will be firstly discussed. Then, an analysis of deployment cost (R2) and time (R3), and how are they linked, will be provided.

4.3.1 Logic Definition - Fabric Pattern

Let's suppose that I have already deployed a certain number of certificates as smartcontracts. It would be very convenient to just *query* the blockchain, and return a set of smartcontracts that match certain conditions (like in a classic database). It would require to have the smartcontract template we are in search for, as certificates are just a set of its instances with different data. Alternatively, blockchain search engines for public data could be used. As for now however, such functionalities are partially or not implemented at all. To workaround this, the dapp that is going to be described uses the so called *fabric pattern*. The fabric pattern in this case consists of two smartcontract definitions, which interact as described below. When I receives a request $info^t_{c_i}$ from c_i, it triggers an already deployed on the blockchain smartcontract called `Issuer`. `Issuer` starts then an *internal* blockchain transaction that creates `Certificate` with all its necessary data, as a result.

4.3.1.1 Certificate

$cert(E(info^t_{c_i}, attach_j))$ could be encoded by the `Certificate` smartcontract shown in Figure 4.2. The field `issuer` is the address of the contract that created *this* instance, and `issuerOwner` is the addresses of I, which triggered deployment process (line `4-5`). `ownerId` and `ownerInfo` (line `7-8`) refer respectively to data which describe c_i and $info^t_{c_i}$. At lines `10-11` `productId` encodes $attach_j$ and `date` is the deployment timestamp. `Certificate` constructor (line `13`) access global blockchain and transaction variables in order to initialize some of these fields (`msg`, `tx` and `block`). Once created, `Certificate` instances are read-only smartcontracts.

4.3.1.2 Issuer

First of all `Issuer` imports `Certificate`, it must contain its bytecode in order to be able to deploy it. Once created at the very beginning, `Issuer` can be used only by its creator I. This is ensured by the function modifier

Figure 4.2: *Solidity certificate definition.*

```
pragma solidity ^0.4.18;

contract Certificate {
    address public issuer;
    address public issuerOwner;

    uint256 public ownerId;
    string public ownerInfo;

    uint public productId;
    uint date;

    function Certificate(
        uint id,
        string info,
        uint product
    )
        public
    {
        issuer = msg.sender;
        issuerOwner = tx.origin;

        ownerId = id;
        ownerInfo = info;

        productId = product;
        date = block.timestamp;
    }
}
```

`onlyOwner` at line `11` in Figure 4.3, and `owner` is determined at construction time (line `18`). `issueCertificate(args...)` function launches an internal transaction to deploy a new certificate with its parameters, at line `28`. At line `29`, messaging system is used to launch a certificate creation event (declared at `6`) that should be eventually captured. The newly created certificate address is saved on the blockchain too, in the `certificates[]` array. In this way, anyone can retrieve deployed certificate addresses with `getCertificates()` method (at line `37`), for inspection.

4.3.2 Interacting with the Blockchain - Back End

Once the logic has been defined through `Issuer` and `Contract`, the remaining part of the dapp have to be implemented. First of all, the dapp back-end must implement the following operations:

1. *smartcontract deployment*, which allow to deploy `Issuer` on the blockchain, this will be required only once or very rarely;

2. *issuance triggering*, which allow to trigger the capability to deploy a new `Certificate`, will be used very often;

3. *transaction tracking*, which means to capture and handle a `Certificate` deployment progress;

4. *information retrieval*, allowing to get deployment history and to insepect issued `Certificate`s.

Listed operations define indeed our *interaction* with the Ethereum Blockchain Network. In order to be able implement this interaction, a JavaScript API is offered: Web3. Web3 provides a `web3`[3] object which in turn has various sub-objects responsible for different tasks:

[3]The described dapp uses `web3 1.0`, which at the time of writing is in beta (`https://web3js.readthedocs.io/en/1.0/`). Beta version usage is motivated by the introduction of more high-level functionalities such as the `PromiEvent` construct.

Figure 4.3: *Solidity certificate issuer definition, imports certificate definition seen in Figure 4.2.*

```
1  pragma solidity ^0.4.11;
2
3  import "./Certificate.sol";
4
5  contract Issuer {
6      event LogNewCertificate(address certificate);
7
8      address public owner;
9      address[] public certificates;
10
11     modifier onlyOwner {
12         if (msg.sender != owner)
13             revert();
14         _;
15     }
16
17     function Issuer() public {
18         owner = msg.sender;
19     }
20
21     function issueCertificate(
22         uint id,
23         string info,
24         uint product
25     )
26         public onlyOwner returns (address certificate)
27     {
28         Certificate newCert = new Certificate(id,
29             info,
30             product);
31         LogNewCertificate(newCert);
32         certificates.push(newCert);
33
34         return newCert;
35     }
36
37     function getCertificates()
38         public constant returns (address[])
39     {
40         return certificates;
41     }
42
43     function getTotalCertificates()
44         public constant returns (uint)
45     {
46         return certificates.length;
47     }
48 }
```

- `web3.eth`, blockchain and smartcontract related functionalities;
- `web3.shh`, whisper protocol for peer-to-peer communication;
- `web3.bzz`, swarm protocol for decentralized file storage;
- `web3.utils`, different helper functionalities.

Whisper and Swarm will not be used however during the realization of this dapp, since they are not required. The package manager used is `npm`[4] and the JavaScript module bundler is `webpack`[5].

```
var Web3 = require('web3');
```

For learning purposes `Truffle`[6] framework has been initially used. However, at time of developing `Truffle` isn't compatible with `web3 1.0`. Therefore, the choice is to directly work with `web3`. Usually, when interacting with dapps in a browser, dapp transactions require the user to pay some fee. Another note regarding the learning curve is that a developer can (at least temporarily) avoid wallet management when developing a dapp, by simply using browser extensions such as `MetaMask`[7]. It provides an interface with integrated wallet management. When the dapp viewed in the browser starts an Ethereum transaction, with user permission `MetaMask` automatically injects `web3` code which signs it cryptographically. Again, for account management, direct work with `web3` has been performed in order to achieve fully automatic transactions. Another way to learn more easily dapp principles, which has been used, is to initially interact with simulated on a local machine full Ethereum clients. The tool used for this purpose is `ganache-cli`[8].

[4] `https://www.npmjs.com/`

[5] `https://webpack.github.io/`

[6] `Truffle` is a full pipeline development environment with built-in capabilities such as smartcontract compilation, linking and deployment on top of `web3` (`http://truffleframework.com/docs/`).

[7] `https://metamask.io/`

[8] `https://github.com/trufflesuite/ganache-cli`

Ethereum runs on its *main-net*, where users spend real amounts of `ETH` in order make programmable transactions occur. Once the dapp is more mature, but it is still too early to deploy it on the main-net, the developer can use a *test-net*. Unlike a simulated environment offered by `ganache-cli`, the test-net is a real network but still with fake `ETH`. It is useful to test how the dapp behaves on public Internet with varying time delays. The test-net used in this work is `Ropsten`. Both main and test-net require to run an Ethereum client[9] in order to interact with it. It is however avoidable, since `RPC` access to Ethereum clients are also offered as a service. During this dapp development `infura`[10] access point has been used.

```
var testnet_provider = 'https://ropsten.infura.io/<user_id>';
var mainnet_provider = 'https://mainnet.infura.io/<user_id>';
```

In order to be able to perform and sign tasks that we are going to describe, an Ethereum user account is needed. It can be created and managed through `web3` methods, or on interface sites like [11]. A test-net account can be easily filled with fake `ETH`[12], while exchanges need to be used to buy `ETH` on the main-net.

```
var account = '0x18ee6f47ab374776a7e42c4ff7bf8f8b7e319a98';
web3.eth.accounts.wallet.add('0x' + privateKey);
```

Below, the above mentioned operations and their implementation will be described: smartcontract deployment, issuance triggering, transaction tracking and information retrieval.

Figure 4.4: *JavaScript Web3* `Issuer` *compiling and instance creation.*

```
1  getIssuerInstance: function () {
2    var contract = web3.eth.compile.solidity(issuerSource);
3
4    // extract ABI and Bytecode from compiled solidity code
5    var issuerAbi = contract.abiDefinition;
6    var issuerBytecode = contract.code;
7
8    // new contract instance
9    return new web3.eth.Contract(issuerAbi,
10                                {data: issuerBytecode});
11 }
```

4.3.2.1 Smartcontract Deployment

If *I* hasn't yet an `Issuer` contract address to use, the first thing the dapp will be used for is to deploy it. In order to be deployable, `Issuer` has to be compiled. `Issuer.sol` can be compiled manually with solidity command line compiler `solc`. However, the dapp should be able to compile it automatically before each deployment. After being loaded from `Issuer.sol`, the `Issuer` source code (Figure 4.3) is compiled through `web3.eth.compile`[13] like in Figure 4.4 at line `2`. The returned datatype contains the *abstract binary interface* (`abi`) and the executable *bytecode*, which are required to create a new contract instance (line `9`). The returned instance however, still owns no address because it has not been yet deployed on the blockchain. A deployment process is a transaction because it modifies the state of the blockchain. Deployment is performed by calling `issuerInstance.deploy(args...)`. The related transaction is tracked in the same way as later will be seen in Figure

[9] `http://ethdocs.org/en/latest/ethereum-clients/choosing-a-client.html`

[10] `https://infura.io/`

[11] `https://www.myetherwallet.com/`

[12] Enter the account address in order to receive test-net `ETH` (`http://faucet.ropsten.be:3001/`), or, through APIs (`https://github.com/sponnet/locals-faucetserver#api`).

[13] `web3.eth.compile` can also compile contracts defined in other languages like `LLL` and `Serpent`.

4.5, for a `Certificate` creation.

4.3.2.2 Issuance Triggering

Once the `Issuer` is on the blockchain we can trigger its methods and deploy `Certificate`s. Figure 4.5 shows the `newCertificate(args)` method which instantiate a new `Certificate`, with its parameter `args` that contains both $info^t_{c_i}$ and $attach_j$. At lines `2-3` a new `web3 Issuer` object is created, and its address field is set with the address value at which `Issuer` is already deployed. Variables declared at line `5-6` will be used to update the front-end which will be discussed later. Before instantiating however, it is necessary to retrieve approximated transaction parameters. When calling `estimateGas()` upon the `issueCertificate(args)` method (line `11-12`), a transaction is simulated by an Ethereum client but with no final state change. The amount of consumed `gas` is returned as the simulated `gasLimit`. `web3.eth.getGasPrice()` instead, returns the median of last few blocks, of the amount of `wei` that users paid per each `gas` unit (line `9-10`). It is reminded that `wei` is an Ethereum sub-unit. As seen in Chapter 3, the final transaction fee will be `gasLimit * wei`. This information could be used by I to understand the network state and to decide whether to accept or lower the transaction fee. After that, at line `15-16`, the `web3 Issuer` instance method `issueCertificate(args)` is invoked as a transaction. It is done through `.send()` where sender `account`, `wei` and `gasLimit` are specified.

4.3.2.3 Transaction Tracking

A `PromiEvent` is a JavaScript promise combined with an event emitter. Promises are combined with event management constructs like `on`, `once` and `off`. Each of these is executed when the Ethereum network reaches various stages of an action, and we are looking for its results. For example, we would like to capture events like a transaction creation, its block confirmation or to eventually handle Ethereum Virtual Machine thrown errors.

A certificate deploying has been triggered as a transaction at lines `15-16`

Figure 4.5: *JavaScript Web3* `Certificate` *deployment handling.*

```
1  newCertificate: async function(args) {
2    var issuerInstance = this.getIssuerInstance();
3    issuerInstance.options.address = issuerAddr;
4
5    var txTable = document.getElementById('tableCerts');
6    var row;
7
8    var wei, gasLimit, tx;
9    await web3.eth.getGasPrice()
10     .then(function(res) { wei = res });
11   await issuerInstance.methods
12     .issueCertificate(args).estimateGas({from: account})
13     .then(function(res) { gasLimit = res });
14
15   issuerInstance.methods.issueCertificate(args)
16     .send({ from: account, gas: gasLimit, gasPrice: wei })
17     .once('transactionHash', function(txHash) {
18       row = txTable.insertRow(1);
19       row.insertCell(0); row.insertCell(0);
20       var cell = row.insertCell(0);
21       row.cells[0].innerHTML = txHash;
22       this.sendBack(txHash);
23     })
24     .on('confirmation', function(confNumber, receipt) {
25       row.cells[2].innerHTML = confNumber;
26       if (confNumber >= MIN_CONFIRMS)
27         row.cells[2].innerHTML = 'Full';
28     })
29     .on('error', function(error) { this.handleError(tx) })
30     .then(function(receipt) {
31       issuerInstance.methods.getTotalCertificates().call()
32       .then(function(res) {
33         var nrCerts = document.getElementById('nrCerts');
34         nrCerts.innerHTML = res;
35       });
36
37       row.cells[1].innerHTML = receipt.
38         events.LogNewCertificate.returnValues.certificate;
39       this.HTMLupdateBalance();
40
41       return newCertAddr;
42     });
43
44   return;
45 }
```

in Figure 4.5. Once transaction hash creation event occurs on the network (line `17`), the promise is executed and the web page of I is updated with a new hashed transaction (lines `18-21`). The `txHash` could be also sent back to c_i enabling him to track the transaction by its own (line `22`).

Each time a new block is appended after the block which contains our transaction, a new confirmation occurs. A confirmation event is handled at line `31`. A transaction here in considered fully confirmed if it reaches `MIN_CONFIRMATIONS` confirmations, which is `24`. Otherwise, if after a certain number of blocks appended to the ledger the transaction is still unconfirmed, an error event is thrown (line `29`). Currently, in `web3 1.0 beta` this limit is `50` blocks. An EVM could throw other errors, like insufficient `gas` or lack of execution rights, resulting into a machine state reversal. These an other exceptions can be handled by `handleError(tx)`. In case that I wants to rise the gas price in order to speed up confirmation, a transaction with the same nonce should be rebuilt and propagated again on the network, with a slightly higher `wei` parameter.

What has been triggered however, is a chain of two transactions. The first transaction has been launched explicitly by the `web3` code. The second internal to the blockchain transaction is started by the `Issuer` contract itself (Figure 4.3, line 28). The received through promises receipt only provides information about the first transaction. The second transaction, creates the smartcontract that encodes $cert(E(info^t_{c_i}, attach_j))$. `Certificate` address is communicated through a *solidity* event instead (Figure 4.3, line 31), that is captured on the `web3` back-end side in Figure 4.5 at lines `37-38`. An intuitive representation of the transaction deployment and tracking process can be observed in Figure 4.8.

4.3.2.4 Information Retrieval

Methods to retrieve information from `Issuer` are going to be described. First of all, it can be seen in Figure 4.9 that the number of total deployed certificates is displayed. This is achieved by performing an explicit `call()`

on the `Issuer` method seen in Figure 4.3 at line `43`, which is a read-only function:

```
issuerInstance.methods.getTotalCertificates().call().then(...)
```

Calls do not modify anything, no fee is required. Therefore, $I \cup C$ and other entities can read $cert(E(info^t_{c_i}, attach_j))$ whenever they want. Let's say now that all certificates ever deployed are wanted. Read-only contract methods like `getCertificates()` in Figure 4.3 at line `37` can be defined, which returns an array of deployed certificate addresses or a chunk of:

```
...
7: "0x2fFed016976aA1d37247d45c6e54F601F5a347d8"
8: "0x57c3B7954b8138E2AE2677E192432D1946Dc9d4e"
9: "0x1d843847f5f73842Dc0402803c5C9269f38aA113"
10: "0xae70a3993565278896d2cadF0A87D1B0CE97e1b0"
11: "0xbD2d50B8f921A911184b9EF2bC16C62F773dC1Cb"
12: "0xf7CD9Ee298D8C4Fa17898190559914B9D823f93b"
13: "0x86EBB57605F73a2d8F7d047c5296D63df1cD5e6E"
...
```

`web3` code in Figure 4.6 shows how, for example, it is possible to retrieve deployed certificates and view their owners. At line `2` a `web3 Certificate` object is built. When `getCertificates()` is called and it returns the array at lines `4-5`, a loop can be performed where at each step we (1), change the `web3 Certificate` object address and (2), access the desired smartcontract information through another call (like `ownerId`, line `9`).

4.3.3 Solution analysis

Requirement (R1) is fully satisfied, because:

(R1) with no trusted parties and whenever, any entity knowing `Issuer`'s address on the blockchain is able to verify for any i and t, if

Figure 4.6: *JavaScript Web3* `Issuer` *compiling and instance creation.*

```
getDeployedCerts: function() {
  var cert = self.getCertificateInstance();

  issuerInstance.methods.getCertificates().call()
  .then(function(res) {
    for (i = 0; i < res.length; i++) {
      cert.options.address = res[i];

      cert.methods.ownerId().call().then(function(ret) {
        console.log(cert.options.address, 'owner is', ret)
      });
    }
  });

  return;
}
```

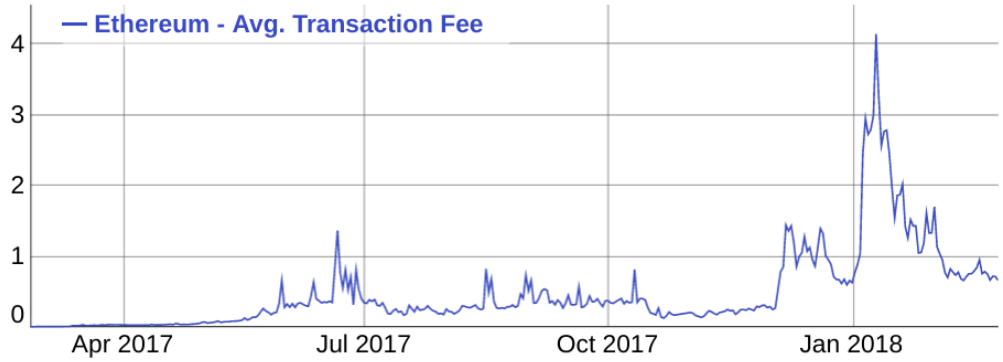
$$cert(E(info^{t}_{c_i}, attach_j))$$

exists and $attach_j$ is unique.

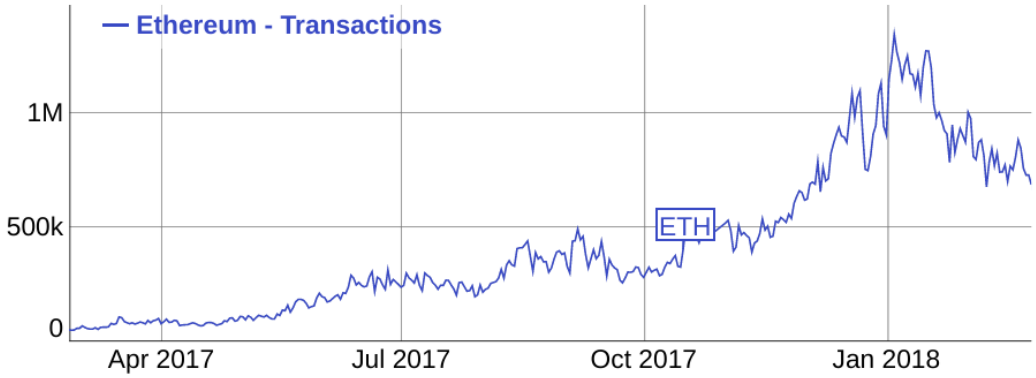
Before analyzing the remaining two requirements some note is necessary to take about the network state. At time of testing the deployed dapp on the main-net, Ethereum experienced one of the greatest network loads ever, as it is observable in Figure 4.7. (R2) and (R3) requirements will be shown in a worst case scenario. Deployments occurred at approximately 1 million `tx/day`, which is close enough to all time high values at time of writing this thesis.

It has been shown in Section 2.3, that at the current state of development, decentralized and consistent systems S like Ethereum should centralize in order to satisfy enormous transaction demand $T(S)$. Since Ethereum still maintained a certain degree of decentralization and full global T-consensus, the amount of pending transactions increased. In this scenario, in a congested system where users want their transaction to be confirmed sooner, they will be willing to rise transaction fees in order to be selected first by the

[14] `https://bitinfocharts.com/comparison/ethereum-transactionfees.html#1y`



(a) *Ethereum average transaction fee expressed in* `USD`.



(b) *Ethereum transactions per day processed.*

Figure 4.7: *Charts describing the Ethereum Network workload in January 2018, when the Issuer dapp has been realized (source*[14]*).*

gas price	test-net (ropsten)		main-net	
	`txFee`	confirmation	`txFee`	confirmation
$4Gwei$	`EUR` 1.21 `ETH` 0.00134	$45s$	`EUR` 1.26 `ETH` 0.00140	$480s$
$9Gwei$	`EUR` 2.71 `ETH` 0.00302	$39s$	`EUR` 2.71 `ETH` 0.00302	$1080s$
$20Gwei$	`EUR` 6.64 `ETH` 0.00738	$33s$	`EUR` 6.039 `ETH` 0.00671	$90s$
$40Gwei$	`EUR` 12.09 `ETH` 0.01343	$42s$	`EUR` 12.09 `ETH` 0.01343	$17s$

Table 4.1: *Transaction fees and required confirmation time at different gas price values.* `gasLimit` *is fixed because determined by computational complexity which is static, and* `1 ETH` *exchange rate is fixed at* `900 EUR`*.*

miners (consensus participants). This behavior however, leads to an overall increase of commissions (Figure 4.7a), and transactions with previously acceptable fees will be more likely be put in the wait queue. This is where the Decentralized Certificate Issuer has been tested, and the results can be observed in Table 4.1. The test-net is irrelevant to see any relation between confirmation time and fees, it has been reported for completeness. On the main-net however, it is sufficiently clear that rising the fee resulted into a faster transaction confirmation time.

(R2) states that deployment operations should occur at a *reasonable* cost. However, it is up to I to decide what is reasonable. In a micro-transaction environment, where transactions occur very often in order to certify frequent events, the presented solution may still not be acceptable for a real-world business. Otherwise, if events to certify are more rare and real world costs related to $E(-)$ are high for I, a blockchain certificate deployment $cert(E(-))$ usefulness could exceed by far its fee.

(R3) requires that a deployment network confirmation should occur after a reasonable amount of time. On public ledgers, a deployment process is

asynchronous. Meaning that even if it is fast, I shouldn't block its execution in order to wait deployment to occur. I should evaluate how much urgency the deployment has: can it happen even after hours, or at most a few seconds are allowed? As Table 4.1 shows, the answer to this question is closely related to (R2), too.

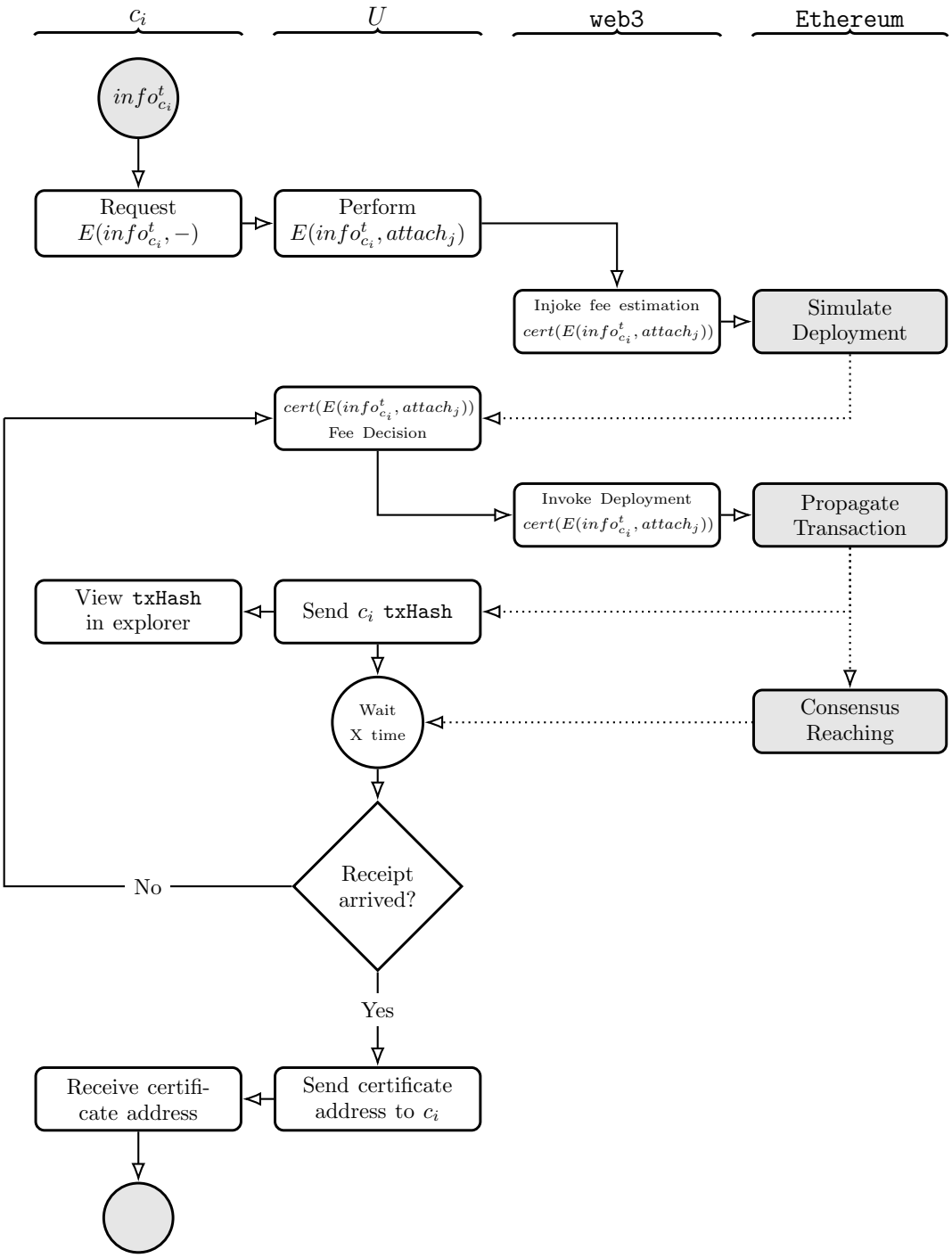


Figure 4.8: *A diagram that intuitively shows how a deployment process occurs with* `web3`*.*

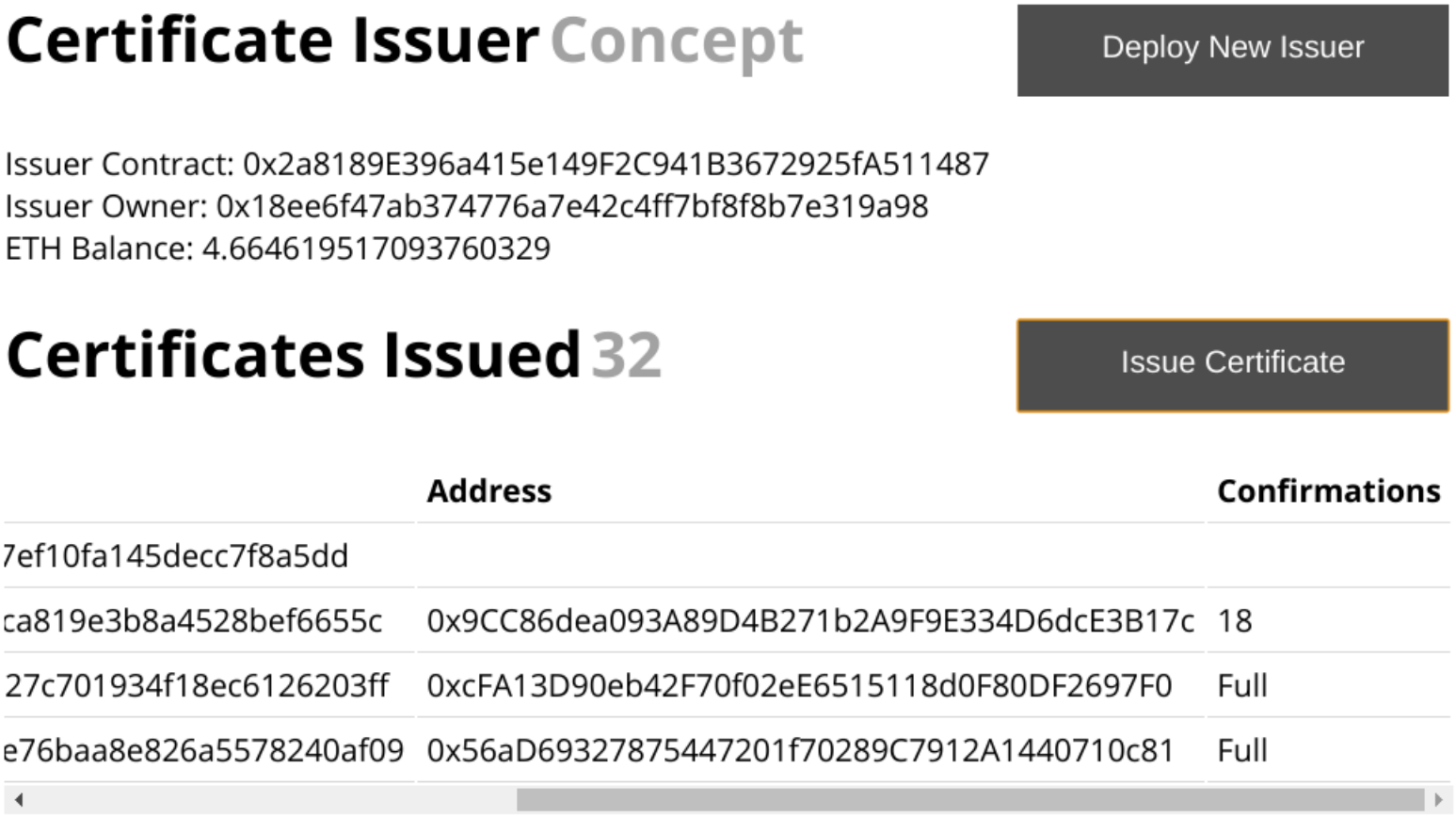


Figure 4.9: *Decentralized Issuer web page for I. Has the ability to deploy new* `Issuers`*; account information; the ability to deploy certificates (only for test purpose from here), their total number and its transaction stages.*

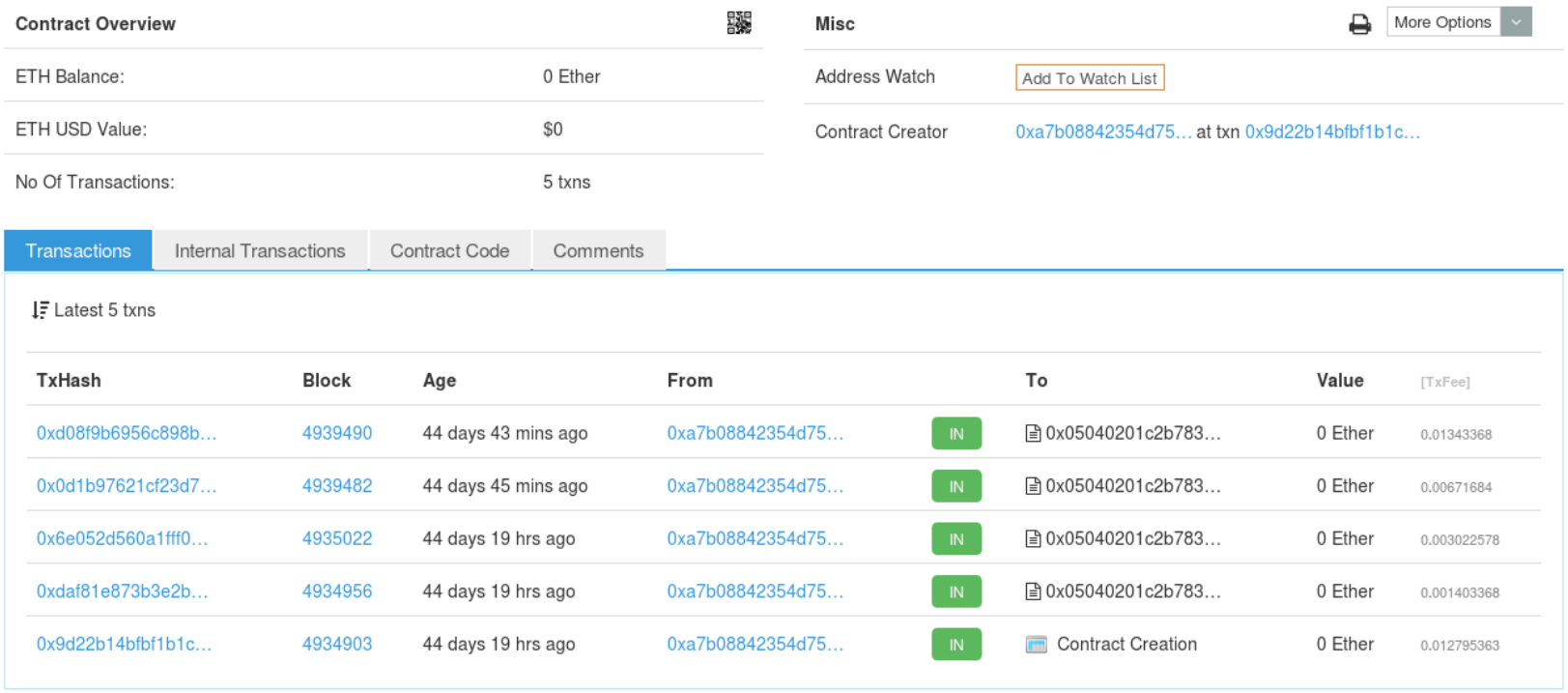


Figure 4.10: *Block explorer which shows 5 transactions: the first transaction created* `Issuer` *itself, the remaining four have deployed certificates using* `Issuer`*'s methods. Internal* `Issuer` *transactions can be seen at* `https://etherscan.io/address/0x05040201c2b783402c1258b65b9178e0de456a69`.

Conclusions

It has been seen that Bitcoin simultaneously provides system availability, partition tolerance and a relaxed form of consistency. This consistency type guarantees in a probabilistic manner that consensus will be reached in the near future instead. It is also known now, that Bitcoin consensus resiliency in fully asynchronous environments like Internet could be only lowered by time unbounded message delays. A *secure*, Proof of Work based blockchain system, should guarantee that the number of ledger writings is proportional to the author's computational power. It has been discussed that a distributed system isn't automatically decentralized, and that a system is decentralized as much as it is its most centralized subsystem. The main desired properties in a blockchain architecture are (i) political and architectural decentralization, (ii) scalability to transactional demand and (iii) consistency. However, it is now clear that in order to maintain consistency at scale, blockchain systems tend to centralize. This is, indeed, the main challenge that public distributed ledgers are currently facing. Beside reparametrization, multiple research and development directions have been taken in order to solve it: alternative consensus algorithms, multiple chain aggregation techniques, logic and cryptographic enhancements.

Blockchain's immutability feature gives a great opportunity to write information that must persist over time. The first requirement (R1) that such a dapp should satisfy, states that certificate content and authenticity must be verifiable by everyone, whenever, with no trusted party involved (certificate author included). For a real-world business however, a solution should

furthermore provide (R2) an appropriate certificate deployment fee and (R3) acceptable temporal confirmation delays. Ethereum has been used to realize a decentralized application with certificate deployment functionalities. In order to be able to certify a given event on the blockchain, the fabric pattern has been adopted. It consists into having a main Issuer smartcontract which always resides on the blockchain. Each time its methods are triggered by the owner, it creates a new Certificate smartcontract with the given parameters. The dapp has been tested in the middle of January 2018, while Ethereum was experiencing a nearly all-time-high workload of 1 million tx/day. The following conclusions have been made about the previously discussed requirements:

(R1) is fully satisfied, because any entity, with no trusted party and whenever, is able to retrieve all certificate addresses and their content by only having the address which issuing organization used;

(R2) an appropriate transaction cost should be defined by the organization controlling the dapp. However, during the dapp testing, various transaction fees have been set, from the lowest possible (at the time) `1.26 EUR` up to `12.06 EUR`, which can be definitely prohibitive for frequent micro-transaction based environments;

(R3) of course, confirmation time delay tends to be inversely proportional to the fee, at a fixed network workload. At the highest tested `12.06 EUR` fee confirmation time was `17s`, and `90s` at `6.04 EUR`. While for some organizations might be acceptable a confirmation time of 8 minutes for a fee of `1.26 EUR`, for other ones it might be unacceptable.

Observed (R2) and (R3) drawbacks are explainable by the DCS Theorem. Ethereum maintained the same level of full consistency and decentralization, therefore it was unable to satisfy a growing transactional demand. Since pending transaction increased, users started to offer higher fees in order to be preferred by miners. It caused an overall average fee increase. It can be said that currently, high fees and longer confirmation time are closely

related to scalability problems. Currently, the main approach to overcome the scalability problem is to sacrifice (R1).

A method of how to relax the (R1) requirement, to lower fees and confirmation delays, has been seen on the Stellar platform. There are no smartcontracts, and a certificate can be encoded by a transaction with an appended certificate hash, from the issuer to the requester account. However, in order be able to verify the hash (meaning the certificate content), the verifying actor must still retrieve plain data from a trusted party. The required off-chain support is clearly in contrast with (R1). On the other side, confirmation time is less than `5s` and a transaction fee is usually much less than a single cent, which makes Stellar to fit into micro-transaction environments.

Private, permissioned ledgers aren't the focus of this thesis, some notes however have to be made. This ledger type, by totally rejecting decentralization provides high throughput, full consistency and comparatively low costs. Ledgers as a Service such as solutions based on Hyperledger Fabric, can be used if *trustless* systems aren't required. Such solutions however, require all the actors (that trust each other) to be active participants within the network. Certificated data could potentially lose any validity if its actors cease their activity.

Future Works

The conclusions made in this work are to be taken as a state snapshot of public distributed ledger technologies. Blockchains can be still considered in their early stage development. Scalability is currently considered the most urgent problem to solve. Smartcontract enabled decentralized computing platforms, are not only rapidly evolving into more sophisticated version of themselves, but also new solutions are emerging. Sharding-enabled platforms, side-chains, alternative data-structures and consensus algorithms, scalability-friendly cryptographic primitives, are still to be widely experimented. Therefore, a wide range of future works can be considered:

- in the fabric pattern each deployment consists of two transactions: the triggering one and the internal transaction which creates a certificate. By avoiding this pattern, the dapp could use a single cheaper transaction. In order to be able to *query* a blockchain, a search-engine-like mechanism is required. Given a certificate definition, such a tool would allow to find its instances on the blockchain. A future work consists into finding such tools that are currently in development phase, evaluate them and implement the related dapp;

- Ethereum, in its "2.0" version, will implement an hybrid consensus model based on Proof of Work and Proof of Stake, features such as state-sharding, and possibly zero knowledge proofs. After a sufficiently relevant evolution, a future work could consist of retesting the scalability of this platform;

- feasibility study and implementation on currently immature platforms should be made. Conceptually different public distributed ledgers are now being developed, too. Some examples follow. The discussed dapp feasibility could be studied on platforms like the already cited Zilliqa [45], which scheduled its main-net launch in year 2018. IOTA [48] on the other side, is a currently under development "blockchain-less" protocol based on a direct-acyclic-graph data structure. IOTA's most interesting feature is the absence of fees, where for each transaction, its author should validate two already existing on the network transactions.

Bibliography

[1] Seth Gilbert and Nancy Lynch. *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.* 2002.

[2] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008.

[3] David Chaum. *Blind Signatures For Untraceable Payments.* 1983.

[4] Adam Back. *Hashcash - A Denial of Service Counter-Measure.* 2002.

[5] Wei Dai. *b-money.* 1998.

[6] Nick Szabo. *Bit Gold.* 2005.

[7] Nick Szabo. *Smart Contracts: Building Blocks for Digital Markets.* 1996.

[8] Nick Szabo. *Formalizing and Securing Relationships on Public Networks.* 1997.

[9] Nancy A. Lynch. *Distributed Algorithms.* Elsevier, April 16, 1996.

[10] Leslie Lamport, Robert Shostak, Marshall Pease. *The Byzantine Generals Problem.* ACM Transactions on Programming Languages and Systems, 4(3): 382-401, 1982.

[11] Leslie Lamport and P. M. Melliar-Smith. *Synchronizing Clocks in the Presence of Faults.* Journal of the Association for Computing Machinery, Vol. 32, No. 1, pp. 52-78, January 1985.

[12] Michael J. Fischer, Nancy A. Lynch and Michael S. Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* Journal of the Asscktion for Computing Machinery, Vol. 32, No. 2, pp. 374-382, April 1985.

[13] Miguel Castro and Barbara Liskov. *Practical Byzantine Fault Tolerance.* Proceedings of the Third Symposium on Operating Systems Design and Implementation, February 1999.

[14] Andrew Miller, Joseph J. LaViola. *Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin.* University of Central Florida, 2014.

[15] Vitalik Buterin. *Ethereum: A Next-Generation Cryptocurrency and Decentralized Application Platform.* 2014.

[16] Andreas M. Antonopoulos. *Mastering Bitcoin - Programming the Open Blockchain.* O'Reilly, 2nd Edition, 2017.

[17] Gavin Wood. *Ethereum: A Secure Decentralized Transaction Ledger.* 2014.

[18] Viktor Trón, Aron Fischer, Nick Johnson. *Smash-proof: Auditable storage for swarm, secured by masked audit secret hash.* May 2016.

[19] David Vorick, Luke Champine. *Sia: Simple Decentralized Storage.* November 29, 2014.

[20] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, James Prestwich, Gordon Hall, Patrick Gerbes, Philip Hutchins, Chris Pollard. *Storj: A Peer-to-Peer Cloud Storage Network.* December 15, 2016.

[21] Protocol Labs. *Filecoin: A Decentralized Storage Network.* August 14, 2017.

[22] *Whisper PoC 2 Protocol Spec.* November 15, 2017. `https://github.com/ethereum/wiki/wiki/Whisper-PoC-2-Protocol-Spec`

[23] Arshdeep Bahga, Vijay Madisetti. *Blockchain Applications - A Hands-On Approach.* 2017.

[24] John P. Conley. *Blockchain and the Economics of Crypto-tokens and Initial Coin Offerings.* June 06, 2017.

[25] Balaji S. Srinivasan. *Quantifying Decentralization.* July 28, 2017. `https://news.earn.com/quantifying-decentralization-e39db233c28e`

[26] Jesus Leal Trujillo, Stephen Fromhart, Val Srinivas. *Evolution of Blockchain Technology: Insights from the GitHub Platform.* 2017, Deloitte Center for Financial Services.

[27] Avtar Sehra, Philip Smith, Phil Gomes. *Economics of Initial Coin Offerings.* August 1 2017.

[28] Rob Massey, Darshini Dalal, Asha Dakshinamoorthy. *Initial Coin Offering: A new paradigm.* 2017, Deloitte.

[29] Vitalik Buterin. *Explanation of DAICOs.* January 6 2018. `https://ethresear.ch/t/explanation-of-daicos/465`

[30] Rafael Pass, Lior Seeman, Abhi Shelat. *Analysis of the Blockchain Protocol in Asynchronous Networks.* September 13, 2016.

[31] Juan A. Garay, Aggelos Kiayias?, Nikos Leonardos. *The Bitcoin Backbone Protocol: Analysis and Applications.* June 23, 2017.

[32] Ittay Eyal and Emin Gun Sirer. *Majority is not Enough: Bitcoin Mining is Vulnerable.*

[33] Greg Slepak, Anya Petrova. *The DCS Theorem.* October 4, 2017.

[34] Paul Baran. *On Distributed Communications: I. Introduction to Distributed Communications Networks.* August 1964.

[35] Martin Harrigan, Fergal Reid. *An Analysis of Anonymity in the Bitcoin System.* May 7, 2012.

[36] Sudeep Pillai, Michael Fleder, Michael S. Kester. *Bitcoin transaction graph analysis.* January 3, 2014.

[37] Dorit Ron and Adi Shamir. *Quantitative analysis of the full bitcoin transaction graph.* 2012.

[38] Matthias Lischke and Benjamin Fabian. *Analyzing the bitcoin network: The first four years.* March 7, 2016.

[39] Laurent Vanbever, Maria Apostolaki, Aviv Zohar. *Hijacking bitcoin: Large-scale network attacks on cryptocurrencies.*

[40] Hannes Hartenstein, Till Neudecker, Philipp Andelfinger. *A simulation model for analysis of attacks on the bitcoin peer-to-peer network.*

[41] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, Madars Virza. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture.* May 19, 2015.

[42] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, Madars Virza. *Zerocash: Decentralized Anonymous Payments from Bitcoin.* May 18, 2014.

[43] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, Michael Riabzev. *Scalable, transparent, and post-quantum secure computational integrity.* February 19, 2018.

[44] Christopher D. Clack, Vikram A. Bakshi, Lee Braine. *Smart Contract Templates: foundations, design landscape and research directions.* August 4, 2016.

[45] The ZILLIQA Team. *The ZILLIQA Technical Whitepaper.* August 10, 2017.

[46] Joseph Poon, Thaddeus Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments.* January 14, 2016.

[47] David Mazières. *The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus.* Stellar Development Foundation. February 25, 2016.

[48] Serguei Popov. *The Tangle.* October 1, 2017.

Ringraziamenti

Ringrazio mio Padre e mia Madre per essere stati la mia base di sostegno lungo questo cammino, senza la quale il tutto sarebbe stato enormemente più difficile. Rimarranno impressi i professori, i cui insegnamenti non solo hanno contribuito a percepire meglio varie discipline, ma anche alla creazione di un *forma mentis* più adatto al mondo circostante. Si ringraziano il relatore accademico e aziendale per la loro disponibilità. Parte importante di questo percorso personale invece, sono stati compagni di studio e vecchi amici tra i quali ci si aiutava a vicenda; come anche i compagni di viaggio in treno, e i volti noti che si incontravano spesso.