Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

Proof of Location through a Blockchain Agnostic Smart Contract Language: Design and Evaluation over Algorand and Ethereum

Relatore:
Chiar.mo Prof.
STEFANO FERRETTI

Presentata da:
MICHELE BONINI

Correlatore:
Chiar.mo Dott.
MIRKO ZICHICHI

II Sessione
Anno Accademico 2021/2022

The greatest danger for most of us is not that our aim is too high and we miss it, but that is it too low and we reach it.

Michelangelo Buonarroti

Sommario

Le applicazioni che offrono servizi sulla base della posizione degli utenti sono sempre più utilizzate, a partire dal navigatore fino ad arrivare ai sistemi di trasporto intelligenti (ITS) i quali permetteranno ai veicoli di comunicare tra loro. Alcune di questi servizi permettono perfino di ottenere qualche incentivo se l'utente visita o passa per determinate zone. Per esempio un negozio potrebbe offrire dei coupon alle persone che si trovano nei paraggi. Tuttavia, la posizione degli utenti è facilmente falsificabile, ed in quest'ultima tipologia di servizi, essi potrebbero ottenere gli incentivi in modo illecito, raggirando il sistema. Diviene quindi necessario implementare un'architettura in grado di impedire alle persone di falsificare la loro posizione. A tal fine, numerosi lavori sono stati proposti, i quali delegherebbero la realizzazione di "prove di luogo" a dei server centralizzati oppure collocherebbero degli access point in grado di rilasciare prove o certificati a quegli utenti che si trovano vicino. In questo lavoro di tesi abbiamo ideato un'architettura diversa da quelle dei lavori correlati, cercando di utilizzare le funzionalità offerte dalla tecnologia blockchain e dalla memorizzazione distribuita. In questo modo è stato possibile progettare una soluzione che fosse decentralizzata e trasparente, assicurando l'immutabilità dei dati mediante l'utilizzo della blockchain. Inoltre, verrà dettagliato un'idea di caso d'uso da realizzare utilizzando l'architettura da noi proposta, andando ad evidenziare i vantaggi che, potenzialmente, si potrebbero trarre da essa. Infine, abbiamo implementato parte del sistema in questione, misurando i tempi ed i costi richiesti dalle transazioni su alcune delle blockchain disponibili al giorno d'oggi, utilizzando le infrastrutture messe a disposizione da Ethereum, Polygon e Algorand.

Abstract

Applications that offer services leveraging the user's location are increasingly used, starting with a navigation system and continuing with Intelligent Transportation Systems, which would allow communication between vehicles and road infrastructures. Some of these services also enable incentives if the user visits specific areas. For example, a shop could reward the users placed nearby with coupons. However, users' location could be spoofed, i.e., trick a system into thinking that the user is in one location while he is in another. In the latter category, they could be incentivized to cheat by deceiving the system and, for this reason, designing a system architecture is necessary to prevent a user from submitting a fake location. For this purpose, many works have been proposed, delegating the creation process of location proof to a centralized entity or placing access points empowered to build proofs or certificates and send them to the nearby users, attesting their position. In this thesis work, an architecture different from the related research has been proposed, leveraging blockchain features and distributed stored mechanism. In this way, we were able to implement a decentralized and transparent solution, ensuring a tamper-proof ledger through the use of blockchain. In addition, a use case will be detailed, showing how the architecture introduced by us could be used and highlighting the advantages. Finally, we have implemented some components of the proposed system, measuring the latencies and costs required by the blockchain transactions during the interaction of the users. These tests were conducted on different blockchains: Ethereum, Polygon, and Algorand.

Contents

List of Figures

List of Tables

Introduction

Nowadays, many users' activities are supported by a different number of mobile applications, leveraging their position to offer specific location-based services. However, it is also possible to spoof the user location, tricking the system and, sometimes, even causing damage to the platform itself.

In this work, we identified an application use case that we would take as an example, proposing a solution to the above problem, to which the related works refer as *Proof of Location*. The use case proposed by us will allow the users, located in a specific location, to make reports about the context in which they are, through a collaborative approach and using their mobile phone. The main category of the reports will focus on our environment; for example, it will be possible to indicate if a river presents oily spots or if there are large quantities of illegally abandoned wastes nearby. However, to ensure that users are where they claim to be, we decided to focus on the design of the *Proof of Location* system. In particular, we opted to build a *decentralized architecture*, using tools such as *blockchain* and leveraging on a *distributed storage* mechanism. More precisely, users can generate *location-proof* in a private, decentralized, and distributed way, using nearby users as *witnesses*.

These choices allow us to inherit the features of the *Distributed Ledger Technology* and to propose a system different from the related works which rely on a trusted third party, for example, the use of the *blockchain*s will guarantee that the reports will be created without spoofing the location and data will be readable by a broad audience, certifying

that a user's mobile device is in a specific location. We will also need to check the user's identity because they may deliberately impersonate other people who are not really.
We decided to test our decentralized application on different *blockchain*s since many exist, performing a performance analysis and evaluating speed and cost transaction metrics. Specifically, we chose two of the leading *blockchains*, that means *Ethereum* and *Algorand*, also using a *layer-2* solution proposed by *Polygon* chain. To this purpose, we used *Reach*, a *blockchain* agnostic language, which allows us to build *smart contracts* starting from a single source code and generating the code for each of the *blockchain*s used by us.
Currently, many distributed tools are emerging, such as IPFS and DLTs. We decided to build a project that is entirely decentralized and subsequently show what the trade-off of adopting this choice is. We proposed a sustainable and environment-friendly use case that removes the single point of failure and intermediary. Indeed, we decided to use tools such as *blockchain*, *DHT*, *decentralized identity*, and others to help ourselves to realize something truly decentralized, focusing on transparency, data availability, and integrity.
Every design choice that we made has a sustainable perspective. For example, we did not test our application only on Ethereum, the most famous and adopted blockchain; instead, we also decided to use Algorand since it is considered a *green* blockchain to identify which could be suitable for this type of use case. We want to show that the applications that could leverage the blockchain are not only finance-centered, but they can range from different fields introducing the advantages of using this type of technology as fast transactions, low fees, interoperability, and removing the intermediary.
In summary, the contribution of this work concerns the creation of a *Proof of Location* system which will prevent fake-location submission and where it is provided the opportunity to store the data in a decentralized architecture and verify data integrity through the use of distributed ledger and *smart contract*.
This thesis is organized into five chapters, and it is structured as follows: chapter 1 is the State of the Art and will also present the related works, chapter 2 will introduce

our architecture, highlighting the main design choices, chapter 3 is a description of our use case, chapter 4 will detail how we built part of the designed system, while chapter 5 will show the performance and evaluation analysis of the tests applied to the different *blockchains*.

Chapter 1

State of the Art

In this chapter, we will introduce the main features and technologies of our trusted location-proof architecture, which will allow users to acquire and issue location-proof in a secure manner. We will also explore the related works available in this field. We will start by talking about Location Based Services that rely on user location to offer services such as navigation apps, Intelligent Transport Systems [8], where vehicles broadcast their current locations along with other information to avoid issues (such as collisions and congestions), or ride-sharing.

1.1 Location Based Services

Location Based Service (LBS) is software that performs functions using the geo-localization of the user or *context-aware* information. The majority of Location Based Services rely on the user's current location offering services, for example, a navigation map, traffic updates, weather forecasts, and proximity-based marketing. The position of the user, computed for example through GPS coordinates [1], is shared with a server and,

[1]GPS (or Global Positioning System) is a free service with planetary coverage which uses up to 72 satellites, owned by the United States and launched in 1978. Some of its drawbacks consist in being a

in turn, the server sends data to the users or enables some service [9].

The position that a user transmits to an LBS is computed by his own device and, for this reason, a malicious user can lie about his position by having his device transmit a location of his choice. Moreover, most of these types of *location-aware* applications offer some rewards or benefit to users who prove to have been in a given place at a given time, and, for this reason, some problems can arise when a malicious user try to submit a fake location to illegally obtain the reward [10]. From this emerges that these types of applications cannot trust solely the users' devices' location transmitted because they have the incentive to cheat.

For example, Zhang et al. [11] found that 75% of location check-ins on Foursquare are fake. Foursquare [12] is a location-based social network that, until 2014, allowed users to obtain virtual rewards such as a *badge* when they sent their current location through *check-in* operation.

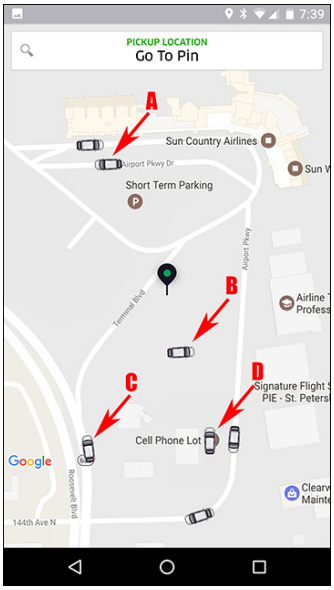


Figure 1.1: Fake location spoofed by Ubers' drivers.

Regarding the manipulation of locations in navigation systems, attackers can divert valuable vehicles or target persons to unsafe areas [13]. For example, in 2012, Australian police rescued tourists directed by an erroneous Apple Maps navigation application to a life-threatening desert with no water supply and extremely high temperatures [14].

A similar issue, where location has been spoofed, happened with Uber in 2017 [15].

In particular, Uber drivers spoof their location in line at the airport to enter its FIFO

single point of failure, don't provide proof of origin, don't provide indoor localization, and being unreliable because of missing authentication.

Queue. However, the driver wasn't actually there, instead, he took trips and made money with other consumers. In figure 1.1 is shown the app spoofs the location: for example, user C is parked on the exit ramp of the airport and user B appears to be placed in the employee parking lot within a gated fence, so neither of them is actually there.
Again a *customer-loyalty reward system* could be designed, in which some discounts may be offered to users who frequently visit the shop.
An enormous number of fields can benefit from this type of application, such as **collaborative crowd-sensing** or smart cities.
Therefore, what becomes necessary is a new strategy that allows the user to prove their current or past locations, which means proving that he was where he claimed it and when he claimed it. Again, a clear definition of this task can be found in [16], where the challenge is defined as "*proving of being present at a certain location, at a certain time with a certain situation awareness to perform certain actions, while having the incentive to participate*".

1.2 Proof of Location

The generation of proof became necessary to increase location assurance and prevent fake-location submissions by malicious users, that cheat the system by obtaining incentives. We can define the **proof-of-location** as a *digital certificate* that certifies the position of a user at a specific time [17] and can be used as *evidence*. Saroiu et al. defined a location proof as "*a small piece of meta-data issued by a component of the wireless infrastructure (e.g., a Wi-Fi access point or a cell tower) in coordination with a mobile device*" [18].
However, this is not the only solution because **infrastructure-independent** or **peer-to-peer** approach can be designed to generate proof-of-location. In APPLAUS project [5], the proofs are mutually generated by nearby users, called *witnesses*, instead of relying

on the access points or other types of infrastructure.
According to Saroiu et al., [18] four main location-proof properties can be identified:

- **Integrity**: nobody can modify a location-proof issued from someone: Location Proofs (LPs) must be personal and mapped to one single identity;
- **Non-transferability**: once a location-proof is issued, it cannot be transferred from one user to another;
- **Trustworthiness**: self-reported location-proof should not be trusted and it has to be validated by someone, i.e., a witness;
- **Non-repudiation**: once a location-proof has been generated, it can not be denied.

An example of Location Proof (LP) is shown in the code listing below 1.1, where the issuer's Private Key is used to sign the whole proof.

```
<locationproof>
    <issuer> Public Key issuer </issuer>
    <recipient> Public Key recipient </recipient>
    <timestamp>Timestamp</timestamp>
    <location>
        <latitude>...</latitude>
        <longitude>...</longitude>
    </location>
    <signature>Issuer Signature with his Private Key</signature>
</locationproof>
```

Listing 1.1: Example of a proof-of-location.

1.3 DHT and Hypercube

Regarding the storage of the data, different strategies could be designed. Since we want to guarantee a great decentralization, avoiding issues such as the single point of

failure, we opted to integrate a **Distributed Hash Table** in our system.

A Distributed Hash Table (DHT) is a **structured peer-to-peer system**, responsible for storing **key-value** pairs, formed by nodes organized with different topologies: a ring, a binary tree, a grid, etc [19].

As the authors claim, one of the characteristics of a **structured peer-to-peer system** is that each node is uniquely associated with an identifier, the key, and its **hash** is used as an index to retrieve the contents of the node itself. That means: $key = hash(identifier)$. The topology that characterizes the DHT, is used to look-up the data efficiently: any node can be asked to look up a given key and route the lookup request to another node if he is not responsible for the given key.

In this project, we decided to use a DHT with a **hypercube** topology, where each node is responsible for a specific *keyword set* and the related content, it speeds up the look-up operations by reducing the number of hops needed to locate contents [20] compared to a classical DHT.

A **hypercube** is an *n-dimensional* cube formed by a *fixed* number of logical nodes, i.e., 2^r, where r is the number of dimensions of the **hypercube**. The ID associated with the **nodes** of the **hypercube** is an *r-bit* string obtained from a *one-way hash function*, applied to the *keyword* that the node is responsible for, with r digits, i.e, the key for an *r-bit* string equal to 1010, with $r = 4$, is 10 [2].

The hypercube structure speeds up the look-up operation because the difference between the nearby nodes is solely of one bit, and it also allows complex query optimizing the routing by specifying the maximum number of hops permitted to locate a specific node and its content. For this reason, during the look-up operations, the node will forward the request to the closer nodes if he is not responsible for the key search. An example of the hypercube is represented with the image 1.2.

[2] If we convert the binary number 1010 to decimal we obtain the key 10.

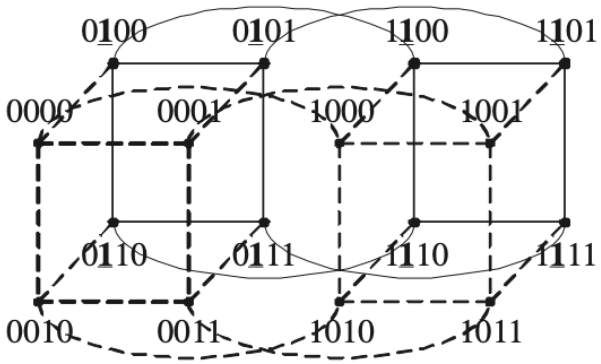


Figure 1.2: A graphic example of a hypercube with r=4, 4 dimensions [1].

1.3.1 Location Encoding Systems

The use of *latitude* and *longitude* by GPS and Location Based Service, are fundamental to identify and represents a specific location. However, they are challenging to use, they are too long, and, if swapped between them, they represent a completely different position [21]. A Location Encoding System can solve the problems that arise with the use of the *latitude* and *longitude* coordinates, associating an alphanumeric, but short, string to a geographic location. Some of the technologies available are:

- **Geohash**: proposed in 2008, allows encoding *latitude* and *longitude* into an alphanumeric string. The Geohash code represents an area where its size depends on the number of digits in the Geohash string which can be composed using 32 characters (*0-9/A-Z* excluding "A", "I", "L" and "O"). One of its disadvantages is that a single location can be associated with more than one Geohash string, for example, "c216ne4" and "c216new"[22] decode to the same coordinates "(45.37, 121.70)";

- **Open Location Code** [23]: is a technology developed by Google and released to public in 2014 that allows to generate *tiles*, partitioning the Earth's surface and then associating each of these *tiles* to a specific and *unique* code. OLC is a string from 2 to 15 characters long [3], used for identifying an area on the Earth and its

[3]The default OLC length is 10 digits and its precision is 13.9 meters.

size will depend on the number of digits in the strings and is built encoding *latitude* and *longitude*. The set, to build a string, is made up of 20 characters which are: "23456789CFGHJMPQRVWX".

In this project, we will design a hypercube where the *keyword set* is associated with the user's location using the *Open Location Code* technology and the reason can be found in its simplicity, it guarantees privacy, and because it is in the public domain.
Going into the specifics, we will use a *dual encoding*, converting the location of the user, with a *latitude-longitude* format, firstly to the respective *Open Location Code* and then to the *r-bit* string. The latter consist of a particular type of logical transformation that allows the OLC to be converted into the *r-bit* string which will be used to represent the node's ID. This conversion was presented in the work proposed by Zichichi et al. [24] and shown in figure 1.3. As it is possible to see from the image, we tried to convert the

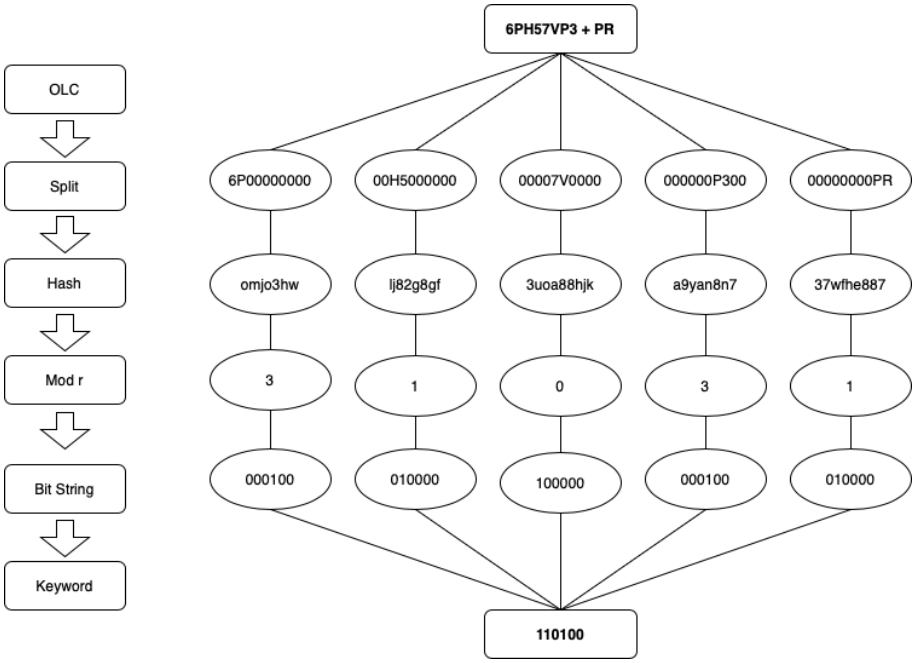


Figure 1.3: Encoding process from OLC to r-bit string, with r=6.

OLC "6PH57VP3+PR" to the corresponding *r-bit* string with $r = 6$, and the first step

of this algorithm is to split the OLC into a specific number of pieces. To do this, we followed the guidelines located inside the Open Location Code repository that suggest using zeros as *padding symbol*s [4]. In our case, the generated segments are: 6P00000000, 00H5000000, 00007V0000, 000000P300, 00000000PR. Then will be necessary to compute a *hash* function on every split piece and execute the *modulo* operation where its result will highlight which *bit* "turn on" in the *r-bit* string. Finally, the final result will be obtained by computing the XOR ($\oplus$) between every bit string. In our case we will obtain: $(000100 \oplus 010000 \oplus 100000 \oplus 000100 \oplus 010000) = 110100$.

1.4 Blockchain layer

As we shall see in the related works section, some Proof Location systems adopt a Blockchain-based approach, i.e., in the project proposed by Brambilla et al. [6] or FOAM. But what is a blockchain?

Blockchain is an implementation of Distributed Ledger Technology (DLT) where a shared ledger, a database that contains all the transactions, is kept by the nodes/participants of the network. Moreover, there is no centralized node or authoritative manager in the blockchain network and it is immutable.

According to Yang Lu [25], blockchain consists of immutable decentralized shared ledgers that can group data blocks into specific data structures in chronological order, and provide secure, transparent, anonymous, decentralized, low-cost, and reliable data or asset transactions in a decentralized network.

One of the purposes of the blockchain removes the intermediary which is empowered to manage the transactions between two users, using a protocol that allows the two entities to transact directly [8] ensuring immutability of data, authenticity, confidentiality, and transparency.

[4] According to [21], "zeros in Open Location Codes must not be followed by any other digits".

Nowadays, many blockchains exist and they are different in purpose, mission, and vision. An example consists of the use of blockchain to keep track of the transactions of specific cryptocurrencies ensuring security, tamper-proof, and transparency, i.e., Bitcoin tries to build a decentralized currency system. In particular, the problem that Bitcoin wanted to solve is the **double-spending** problem: a user that only has €1 can not send €1 to two different people.

A blockchain is composed of blocks, which contain transactions, connected to each other through the use of the hash function. In particular, a specific transaction is definitely added to the blockchain solely if it has been inserted in a block and this block has been connected to the last block of the chain. After a block is added to the ledger, it is computationally infeasible to tamper with its transactions. Therefore, it is not a good strategies store sensitive data on the blockchain and it is not even suitable for recording heavyweight data.

One of the main challenges that an efficient blockchain needs to face up [26] concerns the **blockchain trilemma**, proposed by Vitalik Buterin, and which it claims that we can have, at the same time, only two of the following properties [5]:

- Security: protection against Sybil attacks (51% takeovers) or DDos attacks, etc.;
- Scalability: blockchain should be able to process thousands of transactions in a second;
- Decentralization: the network has to be composed of many nodes and without a central authority.

Another challenge consists in identifying a way and a user that will be responsible to add the next block to the blockchain. Specifically, some users, called *miners*, are empowered to add a new block through a distributed **consensus algorithm** that allows them to reach an agreement on which block to add. The consensus algorithm will select the user

[5] One of the three properties is compromised for the sake of the other two.

that will add the next block.

The main consensus algorithms are **proof-of-work** and **proof-of-stake**.

The most famous algorithm, the proof-of-work, is used by Bitcoin blockchain, where the miner empowered to add the next block is the one that has solved *complex mathematical puzzles* that require a lot of computational power. Moreover, regarding proof-of-work, since two miners can add two blocks at the same time, we could have a *fork* of the chain. In this case, the longest of the two chains is considered authoritative. Specifically for Bitcoin, "*a transaction should not be considered as confirmed until it is a certain number of blocks deep*" [27], so, users have to wait a lot of time and check if their transaction remains on the authoritative chain or not [6].

The proof-of-stake, introduced for the first time on *Bitcointalk* [28], allows the creation of new blocks only if the user stake/holds a **minimum amount of coins** where the stake is the number of network tokens that a user is willing to set aside [29]. There is no need for specialized hardware or expensive cost for electricity. In proof-of-stake, we don't have *miners* but *validators* [7] which deposit their network's currency to participate in the creation of new blocks. If the *validator* will select wrong and fraudulent transactions, a little bit of its stake will be confiscated as a form of *penalty*. We can have different PoS (proof-of-stake) versions which are:

- **Bonded PoS**: users put their money in a table (bond), and nobody can touch them. They will select block and if they misbehave, their money will be confiscated;
- **Delegate PoS**: Empower special users to choose the next block;
- **Pure PoS**: A cryptographic self-selection algorithm decides which node became

[6] On Bitcoin is recommended wait that at least 6 blocks have been confirmed. The average Bitcoin confirmation time is **10 minutes** for block and, for this reason, a transaction may require *more than an hour* to be definitely confirmed.

[7] In addition to storing the ledger, *validator*s have to update it reaching an agreement on which transactions must be added to the next block.

"miners".

Blockchain can be divided in:

- **Permissionless**: they are public and everybody could participate in the network by writing new transactions, creating their own node, or reading the ledger. There is a lack of a central authority, i.e., Bitcoin or Ethereum;
- **Permissioned**: only some well-defined users can interact with the ledger. An example is the blockchain Hyperledger Fabric [30].

We decided to realize our Decentralized Application on two public blockchains at the same time: Ethereum and Algorand. The choice fell back on this technology because of its characteristics such as reliability, security, transparency, fast transactions, and the existence of **smart contracts**.
Smart contract concept was first proposed in the 1990s by Nick Szabo as a computer protocol designed to propagate, verify, and execute contracts [31]. Considering the blockchain context, a smart contract is an executable program stored in the blockchain that defines a specific set of rules that must be followed. If we want to run a smart contract, it has to be called through the use of blockchain transactions that can trigger and execute the program in a distributed way. Again, since smart contracts are stored on the blockchain, their execution and outputs can be traced by everyone.
Finally, Algorand and Ethereum are public blockchains and for this reason can have many advantages such as complete decentralization, total transparency, more security (more users can verify the transactions), and self-sustainability.

1.4.1 Ethereum

Ethereum, introduced by Vitalik Buterin in 2013 and launched in 2015, is a technology for building apps and organizations, holding assets, transacting, and communicating

without being controlled by a central authority [32]. It follows a permissionless approach and has its own cryptocurrency [8] named Ether (ETH) which can be used to pay services or network fees.

Although either Bitcoin or Ethereum allows digital payments without third parties, the latter is a "*fully fledged Turing-complete programming language*" [33] that enables developers to build and deploy decentralized applications on its network leveraging on **smart contracts** [9]. On Ethereum, these programs are run by a virtual computing environment called **Ethereum Virtual Machine** (**EVM**). At the writing time, Ethereum network is composed of over 8,000 running nodes [34] that synchronize their state and act as a single machine and, each of them runs an instance of **EVM**.

1.4.1.1 Ethereum Virtual Machine

Ethereum Virtual Machine is the virtual environment in which all Ethereum accounts and smart contracts live, and which specifies changing state rules. It hosts the piece of software empowered to verify the validity of transactions and manages network security. When we talk about the Ethereum *node*, we are referring to the running version of that software. **EVM** can manage two types of Ethereum transactions:

- **Contract creation**: when a contract is created and deployed [10] on the network;
- **Message call**: if addressed to a contract, allow to execute its bytecode.

Users have to pay for the amount of gas that is consumed by their transactions because each operation has a cost, the most complex is the smart contract, most expensive is its cost.

[8] Currency exchanges can be carried out without a *third party*, i.e., an intermediary/mediator.

[9] Examples of smart contracts could be lending apps, decentralized exchanges (DEX), insurance, crowdfunding apps, etc.

[10] Deploying a smart contract is a process of pushing the code to the blockchain. A specific and unique address will be assigned to the on-chain contract and the code will be immutable.

The most high-level language to write Ethereum smart contracts is *Solidity*, however many other languages exist.

1.4.1.2 Consensus algorithm

On September 15th, 2022, Ethereum's consensus algorithm switched from proof-of-work to **proof-of-stake** (PoS) because it is more secure, less energy-intensive, and allows new and better-scaling solutions compared to the previous proof-of-work architecture. The PoS algorithm allows nodes to stake Ethers into an Ethereum smart contract and act as validators [11], that means checking that new blocks propagated over the network are valid or they can create and propagate new blocks. The validators' funds located inside the smart contract will be destroyed if the validators misbehave.
Specifically, a validator is randomly selected to be a block proposer every 12 seconds, a slot, and will be responsible for creating a new block and propagating it on the network. Then a random **committee** of validators is chosen, and they will be empowered to vote to determine the validity of the proposed block.

1.4.1.3 Gas fees

To avoid malicious users from spamming the network, creating infinite loops, or useless code operations, Ethereum is gas fees based which means each *op-code* is associated with a specific amount of gas cost *ensuring the termination* of the computation. Gas prices are denoted in **gwei**, *giga-wei*, which are 10^{-9} ETH, each gwei is equal to 0.000000001 ETH. The gas fees, paid in Ether (ETH), are the cost that users pay for each transaction (call to smart contracts included). According to Ethereum.org, "*gas refers to the unit that measures the amount of computational effort required to execute specific operations on*

[11] Validators nodes have to stake at least 32 Ethers. If new blocks from peers on the consensus network are received, a validator has to re-execute the transaction inside the block ensuring its validity and propagating its vote (an attestation on the truth of the block).

the Ethereum network" [35]. Since the initial version of the consensus algorithm was the proof-of-work, the gas fees were introduced to compensate the miners for their work done (such as the use of expensive hardware or mining farm). In the Ethereum proof-of-stake version, the fees will be distributed to the users who stake Ethers.
The Ethereum fees can be calculated as:

$$gasFee = (base\_fee + priority\_fee) * units\_of\_gas\_used \tag{1.1}$$

The *base fee*, determined by the network itself, is indicated by each block and is the amount of **gwei** that will be removed from the circulation supply ("*burned*"). In particular, the *base fee* depends on the *amount of gas used for all the transactions* in the previous block and can increase by a maximum of 12.5% per block.
The *priority fee* is optional and determined by the user. The higher the *priority fee*, also called "*miner tip*", the faster will be the transaction goes through because it incentives miners to elaborate your transactions instead of others.
It is possible to add a **gas limit**, very useful during the smart contract development, which is the maximum number of units of gas that users are willing to pay. In particular, Ethereum requires setting a maximum number of computational steps that each transaction is allowed to take, and if execution takes longer computation is reverted but fees are still paid.
If the network is very busy, the resulting transaction costs will be higher, which means, the same blockchain will have **variable fees** depending on the congestion of the network. This feature leads the blockchain to have very high gas fees as it happened in May 2022 [12] when gas prices rose to unprecedented levels, hitting 1261 gwei of the *base fee* and making users pay even 2.6 ETH, or $6500, to 5 ETH, or $14000 as gas fees. In the image,1.4 we can see the amount of gas required for some specific operations. For example, suppose that the *base fee* is equal to 10 and the *priority fee* is 2, making a transaction can cost

[12]ETH gas price surges as Yuga Labs cashes in $300m selling otherside NTFs: https://cointelegraph.com/news/eth-gas-price-surges-as-yuga-labs-cashes-in-300m-selling-otherside-nfts.

Name	Value	Description
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{jumpdest}	1	Amount of gas to pay for a JUMPDEST operation.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
G_{verylow}	3	Amount of gas to pay for operations of the set W_{verylow}.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{\text{warmaccess}}$	100	Cost of a warm account or storage access.
$G_{\text{accesslistaddress}}$	2400	Cost of warming up an account with the access list.
$G_{\text{accessliststorage}}$	1900	Cost of warming up a storage with the access list.
$G_{\text{coldaccountaccess}}$	2600	Cost of a cold account access.
$G_{\text{coldsload}}$	2100	Cost of a cold storage access.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	2900	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{\text{selfdestruct}}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{\text{selfdestruct}}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{\text{codedeposit}}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
$G_{\text{callvalue}}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{\text{callstipend}}$	2300	A stipend for the called contract subtracted from $G_{\text{callvalue}}$ for a non-zero value transfer.
$G_{\text{newaccount}}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
G_{expbyte}	50	Partial payment when multiplied by the number of bytes in the exponent for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
G_{txcreate}	32000	Paid by all contract-creating transactions after the *Homestead* transition.
$G_{\text{txdatazero}}$	4	Paid for every zero byte of data or code for a transaction.
$G_{\text{txdatanonzero}}$	16	Paid for every non-zero byte of data or code for a transaction.
$G_{\text{transaction}}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
G_{logdata}	8	Paid for each byte in a LOG operation's data.
G_{logtopic}	375	Paid for each topic of a LOG operation.
$G_{\text{keccak256}}$	30	Paid for each KECCAK256 operation.
$G_{\text{keccak256word}}$	6	Paid for each word (rounded up) for input data to a KECCAK256 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{\text{blockhash}}$	20	Payment for each BLOCKHASH operation.

Figure 1.4: Ethereum gas costs [2].

$(10 + 2) * 21,000 gwei = 252,000 gwei$ or 0.000252 ETH.

1.4.1.4 Scaling

Smart contracts allow Ethereum, and all the other blockchains that support them, to host services that are typically offered by applications like banks or trading platforms, but, to do that, they have to be scalable to millions of users. However, Ethereum has some scalability issues because, as shown by Bez et al. [36], it respects the blockchain Trilemma renouncing to scalability and opting for security and decentralization. This

type of issue can lead the consensus network to have a limited *transaction throughput*.
To overcome the blockchain Trilemma and ensure all three properties, Ethereum can adopt strategies such as *sharding* [37], *rollups*, *sidechain* and *layer-2* [38] which allow a significant increment in the scalability [13].
For example, **Polygon** [39] [14] blockchain is one of the many *layer-2* solutions, that is an overlay network that improves some aspects of the Ethereum blockchain. One of the features of *layer-2* is that they can be considered an *off-chain* solution, that is they derive some properties such as security from the Ethereum mainnet but, at the same time, implements a new blockchain introducing improvements. Compared to Ethereum, **Polygon** allows low fees, and high transactions per second [15] without sacrificing decentralization and security.

1.4.2 Algorand

Algorand, [40] 1.5, is an open-source project and a distributed public ledger introduced by Silvio Micali in 2017 and deployed in 2019. Micali et al. decided to build Algorand because they supposed that blockchain was inefficient to be managed, i.e., Bitcoin required a lot of computational effort, concentrating power in few hands [16], it had scalability problems and it was slow because each block is generated every 10 minutes. As we shall see, Algorand guarantees all the three properties of the blockchain Trilemma without compromising or without adopting *layer 2* solutions [17].

Figure 1.5: Algorand logo.

[13] The switch from proof-of-work to proof-of-stake allows Ethereum to be more scalable.

[14] Its first name was *Matic* network but in 2021 it was rebranded.

[15] Polygon can execute up to 65,000 transactions per second.

[16] A large amount of energy is needed to create a new block and this is the reason why Bitcoin is not entirely decentralized. There are few mining pools that have enough energy, although Bitcoin supposed that malicious users do not control the majority of computational power.

[17] At the writing time, Algorand is a *layer 1* solution.

According to Micali's work, Algorand uses a cryptographic function to automatically determine, leveraging on the previous block, the *leader*, a user that will propose the next block, and the *verifier set*, in charge to reach an agreement on the block submitted by the *leader*. Both *leader* and *verifier set*s are randomly chosen among the set of all users, reducing attacks from malicious users.
Since malicious users could corrupt *leader* and *verifiers* by suggesting a block to propose, Algorand ensures that both *leader* and *verifier set* **secretly** learn of their role but they are also able to prove their role to everyone using a **credential**. When the *leader* proposes the block and propagates his choice, it will be too late for a malicious user to influence the selection of the new block. As we shall see, Algorand assumes that the amount of money held by *honest* people, users that run bug-free software, is above 2/3 of the total Algorand's monetary value.
Algorand reaches consensus on a new block with low latency thanks to its agreement protocol [18] and the probability of having a fork is minimal [19] because each block is safely final as soon as it enters the blockchain (as opposed to Bitcoin).
According to [41], Algorand has three challenges that have to face up:

- Avoid Sybil attack: creation process of many pseudonyms aimed at influencing the agreement protocol. It is addressed by selecting users considering their amount of stake as weight;

- The Algorand agreement protocol must scale to millions of users: it is obtained through the use of a randomly chosen committee;

- Algorand has to continue to operate even if an adversary disconnects some of the nodes.

[18] Algorand uses a new Byzantine Agreement protocol to reach a consensus that has been modified by Micali et al. [40]. One of the advantages of this protocol is that it can scale to millions of users.
[19] According to [41], an attacker has to control less than 1/3 of the Algorand's monetary value.

1.4.2.1 Consensus Algorithm: Pure Proof of Stake

Algorand is a *layer 1* and **pure proof-of-stake** (PPoS) blockchain, that means, instead of relying on solving complex crypto-puzzle [20] and consumes a lot of energy, it improves security and power efficiency across the network by limiting miners to validating transactions proportional to an ownership share [41]. Using proof-of-stake instead proof-of-work reduces the block confirmation time, which means reducing the time needed for a transaction to be confirmed. This can also be applied with **pure proof-of-stake**, but Algorand's network is not monopolized by stakers with high amounts of tokens [29] as instead happens for the blockchains that use the proof-of-stake (not pure) because in **PPoS** the protocol selects users randomly, irrespective of their stake. That means, **pure proof-of-stake** does not require a minimum amount of tokens during the staking phase and, in addition, there is no penalty if a user misbehave.

The new proposed block will be validated only if the majority of stakes are agreed on it. Since specific and expensive hardware is not required, proof-of-stake blockchains can be composed of many nodes, which can improve the network's decentralization.

The users that act as validators will receive a reward proportional to their stake. The reward will be distributed using Algorand's cryptocurrency that is **Algo**.

The block validation phases are shown in figure 1.6. Specifically, the **pure proof-of-**

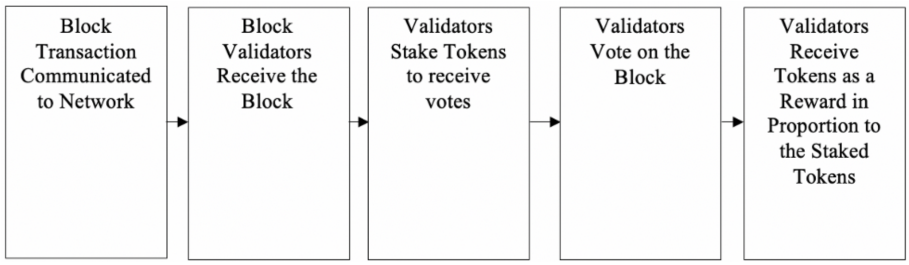


Figure 1.6: Algorand block validation [3].

[20] The success probabilities to solve the *puzzle* depend on **computational power** only. This leads to strong competition between miners.

stake mechanism is based on computing a *Verifiable Random Function* [41] [42], it is executed for each block and can be summarized as follow:

1. A user, the *leader*, is randomly chosen and learns secretly its role executing a secret *crypto-graphic sortition* [21] (users are selected at random and weighted using their stake). In addition, since an account can own a big amount of money and, for this reason, can be chosen frequently, the sortition algorithm returns a parameter j that will indicate how many times a user was chosen;

2. A committee, the *verifier set*, is randomly chosen and learns secretly their role [22];

3. The *leader* chooses its block to propose and propagate its choice on the network along with his credential;

4. The committee certified the chosen block reaching an agreement.

Thanks to the **PPoS** consensus protocol, Algorand can satisfy all three properties of the blockchain Trilemma.

1.4.2.2 Algorand Virtual Machine

As we have seen for Ethereum, also Algorand has its own **Algorand Virtual Machine (AVM)** that can be run on each node [43] [23].

[21] *Sortition* is implemented using Verifiable Random Functions which will return a hash and a proof (the **credential**).

[22] The function will indicate if a user is chosen, returning a short string that proves this user's committee membership to other users.

[23] At the time of writing, Algorand has over 1600 running nodes [44], and the minimum requirement to run an Algorand full-node are shown here [45].

AVM contains a stack engine that evaluates smart contracts that, once deployed, anyone can call through an "application call" to the contract and which will be evaluated by the **AVM**. In addition, to evaluate the smart contracts, the **AVM** is empowered to check the *smart signatures*. The difference between smart contracts and smart signatures is that the former is *stateful* [24], the changes will be recorded to the blockchain if the call is successful, while the latter is *stateless*, they are used to approve spending or asset transfer and its logic is submitted with the transaction. The **AVM** interprets an *assembler-like language* called **TEAL** (Transaction Execution Approval Language).

```
int 0
txn ApplicationID
==

// if not creation skip this section
bz not_creation

// save the creator address to the global state
byte "Creator"
txn Sender
app_global_put

// verify that 5 arguments are passed
txn NumAppArgs
int 5
==
bz failed

//store the start date
byte "StartDate"
txna ApplicationArgs 0
btoi
app_global_put
```

Figure 1.7: Teal example.

However, the Algorand ecosystem offers other high-level languages that can help the developers. One of them is PyTeal, a Python library that allows facilities for the development, and Reach, another high-level language, similar to Javascript which, when compiled, can produce the Algorand and Ethereum smart contract source code which are respectively TEAL and Solidity.

1.5 Interplanetary File System

Interplanetary File System (IPFS) is a protocol and implementation of Distributed File Storage, launched in 2015, that allows peer-to-peer *file sharing* using a distributed system. Everyone is given the opportunity to become a node of the network and start uploading/downloading files. The IPFS protocol assigns each object to a *unique* address called **Content IDentifier** (CID) built hashing [25] the file content.

[24] Stateful contracts are applications that run on the chain. They are used when we want to store values on the chain globally or locally after that a user option to the contract.

[25] The hash used is the *SHA-256*, source: https://developers.cloudflare.com/web3/ipfs-gateway/concepts/ipfs/.

The IPFS is built through the use of a DHT which is used to map each **Content IDentifier** to the *IP address* of the owner. One of the disadvantages of IPFS is that it does not offers incentives to users who decide to participate in the project hosting some file and, for this reason, a specific object could disappear from the network if nobody decides to host it.

1.6 Decentralized IDentifier

Today, **identification** and **trust** are a big issue on the internet. The majority of online applications such as social media or finance services [26] need users to prove who they are to gain access to the resources, using a *username* or *e-mail* as a unique identifier on that specific platform. An example can be found in the Federated Identity where the user gives his trust to a specific identity provider [27] such as Google with the benefits of login into many different apps using Google as a primary source of identity. This can lead to very potential issues: if the identity provider disables a user's account, the latter might not be able to access the other online services which he previously accessed using that specific identity provider. Moreover, in Federated Identity, the identity provider is at the center, so we cannot use our Google credentials to log in to every web application, i.e., our bank account; that means credentials are not portable outside. For this reason, another important task of this work is to verify the user's identity without relying on a centralized database or central authority. The new model of *Decentralized Identity*, also called *SSI* (Self-Sovereign Identity), follows a **user-centric** [28] scheme, allowing users to became the owner of their data and is based on **Decentralized IDentifier** and **Verifiable Credentials** [46], leveraging Blockchain and distributed ledger technology (DLT) infrastructure.

[26] They can require a KYC (Know Your Customer).

[27] They guarantee identity for the user.

[28] User maintain control of his data.

Decentralized IDentifiers (DIDs) [47] are a new type of digital and *globally unique identifier*, standardized by *W3C* (*World Wide Web Consortium*), where the DID uniqueness is ensured by *DID methods* [48] [29] and by the DID design itself, i.e., the *DID document* [30], shown in figure 1.8, which contains information for authenticating the DID owner. Specifically, through the DID *resolution* it is possible to reach the *DID document*, stored in a verifiable data registry such as a blockchain, starting from the DID itself [49] [31]. So, the DID *resolution* is the process that allows us to find *DID documents* using DID as a parameter. According to [50], instead of using a username or an email address as primary identifier, it is used a DID, an alphanumeric string that represents who the user is, in a given context, without giving trust to the specific identity provider. DIDs rely on cryptography, specifically, users can use a private key to prove control over a DID (as a password does for the username). As we shall see in the next chapter, inside the *General behavior* section, the **authentication** is the process that allows someone to prove its control on a DID.

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/suites/ed25519-2020/v1"
  ]
  "id": "did:example:123456789abcdefghi",
  "authentication": [{
    // used to authenticate as did:...fghi
    "id": "did:example:123456789abcdefghi#keys-1",
    "type": "Ed25519VerificationKey2020",
    "controller": "did:example:123456789abcdefghi",
    "publicKeyMultibase": "zH3C2AVvLMv6gmMNam3uVAjZpfkcJCwDwnZn6z3wXmqPV"
  }]
}
```

Figure 1.8: DID document example. The "id" is the entity that the document describes.

[29] The *DID method* identifies where the DID *resolution* happens,i.e., it happens on bitcoin blockchain it will be *did:btcr*.

[30] The *DID document* contains detailed information about the DID. It can specify the *DID controller* which is the entity that has the authority to modify the document.

[31] More information about DID and its features at https://w3c-ccg.github.io/did-primer/.

1.7 Related works

Since many different schemes could be used for location proofs, we will talk about related works and their implementations/ideas.

For example, Zichichi et al. [51] proposed a use case leveraging on a decentralized architecture aimed at the development of novel services for Intelligent Transportation Systems using Distributed Ledger Technology and Distributed File Storage to *store* and *certify* crowdsensed information coming from vehicles on the road. They used IOTA ledger to store the data while Ethereum was utilized to execute smart contracts.

The majority of related works, that involves the proof-of-location, focus on security and privacy challenges, and only a few try to realize something truly decentralized. Generally, the location-proof systems are divided into two categories considering their system architecture: **centralized** and **decentralized** verification approach. In the **centralized** systems there are databases, servers, or Location Based Services that can check the proximity of a prover to a witness or are used to store the location proofs. Given their architecture, this type of system could suffer from a single point of failure issue, making the application insecure or allowing users to spoof their location easily. Nosouhi et al. [10] assume that the LPs generation process could be slow in the case that a database was adopted, above all when the user uses a long private key "*since a prover device must respond to m challenge messages that a witness sends to it, where m is the size of the prover's private key*".

In the **decentralized** applications, the system could solve the issues above designing a blockchain-based infrastructure. However, most proof-of-location systems are centralized, i.e., they rely on servers to store location proofs, as said before.

Related works subdivide these categories into another two which are: **infrastructure dependent** and **infrastructure independent**. For the generation and validation process of LPs, the former could use a *bipartite gathering approach* while the latter a *collaborative gathering approach* [17].

1.7.1 Infrastructure dependent

In this type of system, usually, a trusted fixed access point (e.g Wi-Fi) or specific hardware is employed to check users' location and issues location proof. An introduction of location proof, using access points and which contains six potential applications, published by Microsoft research can be found at [18]. However, they do not consider the privacy issue. In the solution proposed by Saroiu et al. users can communicate with access points or cell towers, requesting location proof.

Javali et al. [4] propose a *centralized* solution for generating LPs that manage the verification process, leveraging unique Wi-Fi signals. In this solution, the LPs are generated and provided to mobile users through access points, which are considered trusted by default. Subsequently, a Location Based Service will verify the proof and grant specific services if the LPs are valid or not 1.10. One of the drawbacks of this solution is the expensive costs that should be incurred while dealing with many access points.

Figure 1.9: FOAM logo.

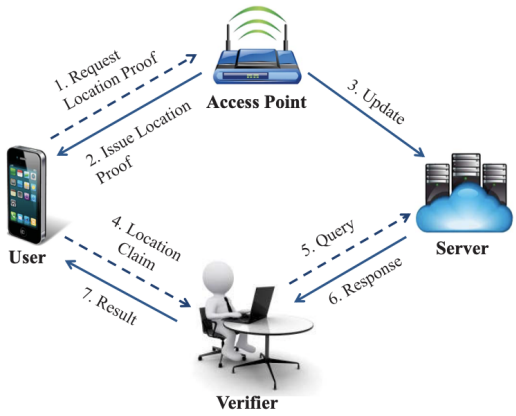


Figure 1.10: Location-proof system with infrastructure-dependent approach [4].

Another *infrastructure-dependent* project is FOAM [52] which is a **decentralized open infrastructure** that is trying to use the concept of *zone* and *zone anchor* (a radio beacon), placed outdoors, creating a **trust zone** used for location tracking. FOAM is also a project that uses Ethereum blockchain which allows it to become more decentralized allowing users to contribute, and control when and with whom they share their personal data. One of the core points of this project consists of the use of *Crypto-Spatial Coordinates*: a technology that uses the Geohash standard which ensures that any physical location has a corresponding smart contract address that is accessible for decentralized applications. Their mission is to provide tools that enable a crowdsourced map and decentralized location services such as replacing GPS by offering a new location tracking mechanism. The former considers the users as a *cartographer* that can contribute by adding a new point of interest to the map, and the latter use *zone anchors*, a set of at least 4 radio beacons, to send signals to discover and connect with others nearby.

Figure 1.11: Use of CSC with FOAM.

Another Ethereum-based project, that tries to solve the issue of spoofing the GPS position, proposed by Victor et al., [53] relies on the existing infrastructure of a **mobile network operator** leveraging a vast network of **cell towers**. To apply this strategy, the user must be equipped with a *terminal* or an IoT device that must be locatable in terms of the network cells. In this way, leveraging the geographical coverage area of the

mobile network operator, it is possible to indicate where the terminal must be placed. Specifically, the authors of the article propose an approach to represent **geofences** using Ethereum smart contracts, checking if the user's position (using the *terminal*) is located inside the *virtual borders* and then and trigger specific actions accordingly.
According to the authors, the smart contract cost for each grid cell is equal to **20,000 gas** and geofence consisting of 100 grid cells leads to a total gas cost of 2,088,102 gas (also considering other base and overhead transaction costs).
On September 28th, 2022, this would translate to **0.187 ETH** [32] that means €250.69 [33] or **$240.23** [34] to store a single geofence.
The authors compute the same analysis on August 16th, 2018 [35] founding that the gas cost for storing a geofence was equal to **$1.89** on the main Ethereum blockchain.
In conclusion, storing a geofence with 100 grid cells on Ethereum today is no longer a viable solution cause its expensive costs.

1.7.2 Infrastructure independent

According to E. Pournaras [16], *decentralized systems* can be used to design a more informed and participatory collective decision-making utilizing the concept of *witness presence*, and remain *independent* from the use of specific hardware/infrastructure. This application is characterized by the presence of **witnesses** that usually use a short-range technology of wireless radios, e.g. *Bluetooth*, that ensures the physical proximity of mobile users nearby. The absence of access points allows the infrastructure to be *cheaper* than the infrastructure dependent. Specifically, in an *infrastructure-independent* location-proof system, the issues process of the location-proof is delegated to mobile users (*witnesses*)

[32] The ETH price is approximately €1334 for a unit.
[33] The base fee is equal to 8 Gwei and the priority fee is 1 Gwei.
[34] Exchange rate: €0.96 is equal to $1.
[35] Retrieving the historical data of August 16th, 2018 and using the pre **London Upgrade** [54] mechanism: the ETH price was $298.48.

who are empowered to validate the prover's location.

Since the witness is not trusted, a **verifier** user is required to verify the LPs generated.

APPLAUS [5] (A Privacy-Preserving LocAtion proof Updating System) is one of the pioneer *infrastructure-independent* projects that proposed a *centralized* scheme where, through a short-range communication method, users mutually generate location proofs and report them to a server 1.12. The proof requested by the prover is generated by a witness using a random number, pseudonym [36], computing a hash and signing using the witness Private Key 1.13. Then the proof will be sent to the central

Figure 1.12: Architecture of APPLAUS [5].

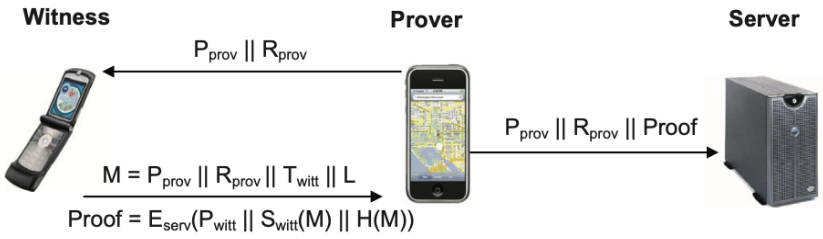


Figure 1.13: Proof generation process in APPLAUS [5].

server by the prover which leads this project to be **centralized** cause it uses an untrusted central server to store the historical records and other data, as said before. One of the participants in this architecture is the **Central Authority** who knows the mapping between the Public Key and the real identity of the provers.

As shown in figure 1.12, the **Central Authority** will be queried by the **verifier**, passing

[36] To preserve users' location privacy, the user's pseudonym change periodically.

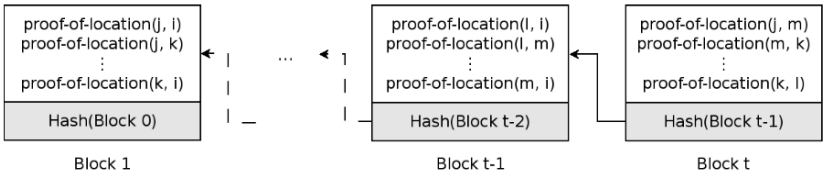


Figure 1.14: Proofs of Location recorded on Blockchain [6].

the real identity of the subject which will need to be verified, and then the **CA**, after authenticating the **verifier**, will convert the real identity to its corresponding pseudonyms and retrieve the location proof from the server.

In the blockchain-based architecture proposed by Brambilla et al. [6], valid proofs of location are recorded into blocks, which are then added to the end of the chain, and, once confirmed by consensus, they cannot be changed 1.14. However, this solution is vulnerable to collusion attacks because the protocol allows direct communication between provers that could cheat the system. Since the location proof will be stored and verified

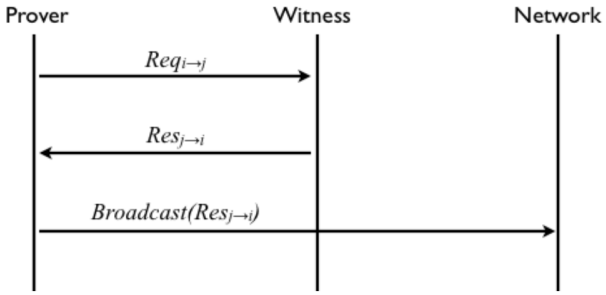


Figure 1.15: The generation process of LPs and diffusion [6].

using the blockchain 1.15, a central server is no more required.

The proof-of-location **request** 1.16a and **response** 1.16b of this architecture are shown in the images. As we can see, every peer puts all known valid unacknowledged proofs of location into a block [37] of the blockchain. As authors Brambilla et al. [6] wrote: "*if most peers add the block to the blockchain, then consensus is achieved, therefore proofs*

[37]The block also contains the ID of the user that generated it and is signed with his Private Key.

of location are made persistent. Otherwise, the block is discarded and not attached to the blockchain." [38]. Other checks will be made to ensure that nobody will cheat the system, for example verifying that the proof-of-location inserted in a new block is not already present in previous blocks of the blockchain, not broadcasting and discarding it. The performances and results are focused on peers that try to cheat the system.

Another *infrastructure-independent* and blockchain-based solution has been proposed by

$$Req_{i \to j} : \left\{ \begin{array}{c} K_i^{pu} \\ \langle latitude, longitude \rangle_i \\ h(Block_{t-1}) \\ timestamp \end{array} \right\}_{K_i^{pr}}$$

(a) Location proof request sent by Prover [6].

$$Res_{j \to i} : \left\{ \begin{array}{c} Req_{i \to j} \\ K_j^{pu} \\ \langle latitude, longitude \rangle_j \\ timestamp \end{array} \right\}_{K_j^{pr}}$$

(b) PoL response sent by Witness [6].

Nosouhi et al. with the PASPORT architecture [55], where the main actors are the prover, witness, and verifier empowered to assign the witness to the prover for the location-proof generation. The authors claim that their system was Prover-Prover and Prover-Witness collusion resistant, however, the verifier could not act in "good-faith" and misbehave.
Gambs et al. proposed PROPS architecture [17] which, although it follows a *collaborative* approach, uses a single Location Based Service for the verification phase.

[38] The consensus algorithm is Proof of Stake using a pseudo-random to decide who will add the next block.

Chapter 2

Architecture Design

We tried to design the architecture of our Decentralized Application where its purpose concerns the construction of a system that allows users to report specific areas, through a **collaborative** approach. The nature of the report will be detailed along with the use-case vision in the next chapter.

In the following sections, we will describe the design choices for this work, where the main challenges concern the realization of a **Proof of Location System** and the **Decentralized Application** (DApp). Through interactions with an **off-chain structure**, the hypercube, our DApp will be able to retrieve values, the report information, and show them to the users without authoritative servers, ensuring the truthfulness of the data in the DHT. Indeed, one of the primary roles of the proposed Proof of Location architecture is to realize a sort of "**garbage in**" empower to skim the input data before they are entered in the hypercube. Without it, users will be able to insert all the data, truthful or not, that they want without limits, overloading the system and taking up many resources such as memory.

This led us to identify two main challenges of our Location Proof system: the former consist of computing the **Proof of Location** in a **decentralized** manner, the latter is to **verify** that data inserted by provers inside Smart Contracts are truthful Proof of Lo-

cation allowing them to be inserted inside the hypercube. Although the first point has been built to maintain its decentralization, the second is not completely decentralized for two reasons: there is a Certification Authority and not everybody can act as a verifier.
So, an advantage of our decentralized architecture is that there is no need to employ trusted access points to generate new location proofs. However, we did not focus on the *Prover-Prover* [1] or *Prover-Witness* [2] collusions which have been considered by the related works [3].
As we saw in the *State of the Art* chapter, there are some related works concerning this argument designed with different characteristics; in this project we have oriented ourselves towards these features:

- **Decentralized System**: using smart contracts, DHT, and IPFS for the storage of data;
- **Blockchain based**: ensure more *interoperability*, *security*, *transparency*, and *availability* removing the single point of failure through the use of smart contracts instead a single Location Based Service or server/database. The Blockchain-based strategy will allow for building a reward mechanism aimed at users who participate in the system;
- **Infrastructure independent**: we will use Bluetooth to communicate between the prover and witness. We will not use sensors, Wi-Fi access points, or specific hardware to generate proof of locations;
- **Proof of Identity**: the prover should prove his identity to the witness, and should do this by avoiding central identity providers;

[1] Two distant Provers cooperate with each other in order to create a location-proof.
[2] This is one of the major challenges: a Witness is able to generate a location-proof for the Prover even if one of them (or both) is not located in the location that he claimed.
[3] According to Nosouhi et al. [55], a reliable solution has not yet been proposed, even if some researches have been carried out.

- **Permissioned verification**: the verifiers are well known and not everyone can become one of them.

In this work, we leverage the usage of a *smart contract* in order to temporarily store the data that still have to be verified and which are sent by provers. Specifically, as we shall see in the following sections, we decided to associate a different location for every smart contract that could be created, storing the *contract id* inside the hypercube.

2.1 System actors

In the figure 2.1 we can see the general architecture of the system where the majority of complex aspects are hidden, such as the specific information and proof details sent by the prover to the smart contract, and the main actors are:

1. **Prover**: a user, with a mobile device, who needs to validate his or her location by obtaining proof that can be verified;

2. **Witness**: the primary role is to compute and issue location proof (LP), sending it back to the provers. We assume they are untrusted. Indeed, to ensure that LP cannot be *forged*, the LP itself is signed by the private key of the witness that generates the proof.

3. **Verifier**: these users have the task of confirming and verify the locations and other information stored in the blockchain. They will also check that the signature of the prover and witness are valid.

Moreover, another important actor is the **Certification Authority**; it is not represented in the image to make it more readable. The **Certification Authority** is in charge of indicating the verifiers. In a new version of this project, they will issue Verifiable Credentials to the users that have a DID and are required for them. So, they know the

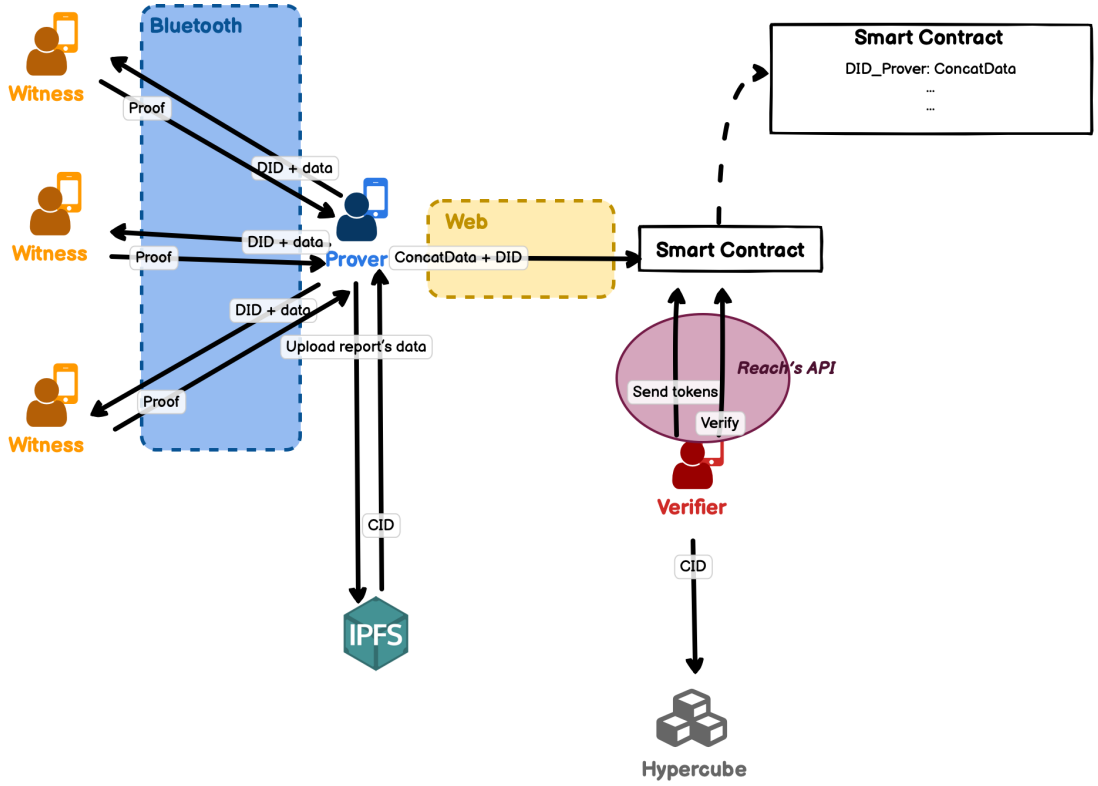


Figure 2.1: The general architecture of the system.

mapping between the real identity and the pseudonyms (public keys) of users. Every time someone wants to become a witness he will have to communicate his *public key* to the Certification Authority. In this way, the witness *public key* will be added to the **witnesses list** delivered to the verifier [4], necessary for the verification process . In addition, also the verifier needs to be trusted by the Certification Authority. The use case diagram 2.2 shows the possible interactions between users and the system. The authority actor is not shown because not modeled here.

As we can see, prover and witness have very similar behavior. Depict them separately it was merely a design choice to better understand the system.

In particular, the **prover** will:

[4]We were inspired by the PASPORT architecture, proposed by Nosouhi et al. [55].

Figure 2.2: Use case diagram: Proof of Location.

- Request for a DID generation: this was one of the first steps that had to be done and allowed to obtain a DID. Every user, including the prover, should interact with the smart contract empowered to generate and return a new DID [5];

- Insert Proof/Location/DID inside the smart contract: once obtaining the proof, the prover will insert the data inside a specific smart contract;

- Search for location inside the hypercube: this is the first action that users do if they want to start the verification process of their location. In particular, he will have to

[5]In a long-term vision of this project, this DID could be used to obtain **Verifiable Credentials** and used them in this application

search for his position inside the hypercube and get the associated contract address. If not found, he will have to deploy a new smart contract and insert the ID and location inside the hypercube;

- Request proof from the witnesses: if a prover wants to initiate the verification process, he will have to ask witnesses to compute the proof for him. For this purpose, the prover will broadcast his location to the users nearby;

- View users nearby: the application will show, using the **Bluetooth**, the users nearby [6].

In addition, **witness** is responsible for:

- Compute Proof: it consists of computations and cryptographic techniques that will produce the Proof;

- Send Proof to the prover: witnesses, and provers that act as witnesses for other provers, will send the proof to requesting users.

Finally, the **verifier** is tasked with:

- Insert funds inside the Smart Contract: the funds will be used to reward the provers and witnesses. This is a design choice and does not exclude that this role will be assigned to the authority in a new version of this project;

- Search a user and his Location/Proof: to verify someone, the verifier will query the smart contract filtering by the user's DID and obtain the data associated with it;

- Validate Proof and user's location: this is a computation process that will be executed by the verifier and allow him to insert the verified data inside the Hypercube.

[6] This feature could be implemented in future work.

2.2 General behavior

In this section, we will show the general behavior of the system. Recall that to interact with this application, the user must create a new blockchain wallet and generate a DID. The first interaction comes from the need of the prover that wants his position confirmed by neighbors. However, the prover must check if a smart contract associated with his position already exists and will do this **querying the hypercube** 2.3 which will return a **contract address** in case of a positive outcome, otherwise, the user will have to deploy a new contract and insert the new **contract address** into the hypercube.

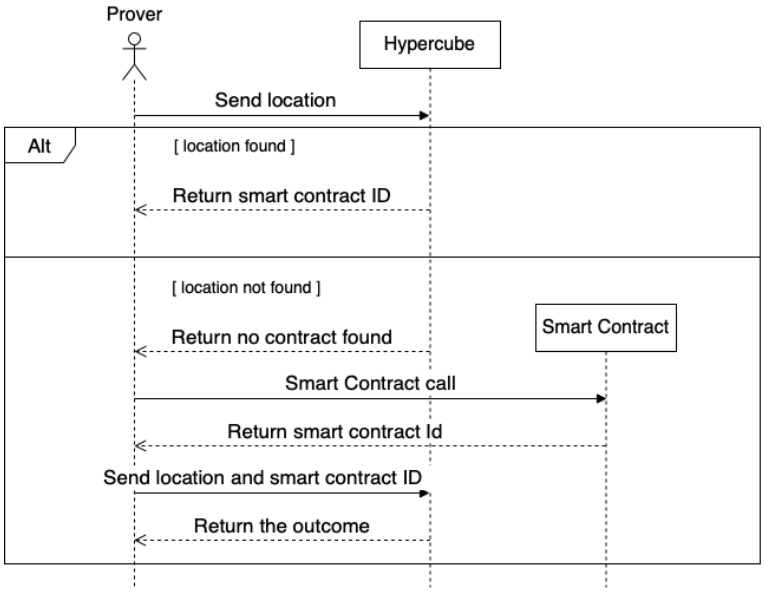


Figure 2.3: Sequence diagram of the initial phase.

Subsequently, as shown in the image 2.1, the prover will have to broadcast his credentials and location to the users nearby using **Bluetooth** technology which will ensure that nobody will be able to **spoof the position**. Indeed, if someone wants to use a particular service that requires a user's location, i.e., the one described in our use case, a witness will be empowered to validate the position. Firstly, the witness will compute

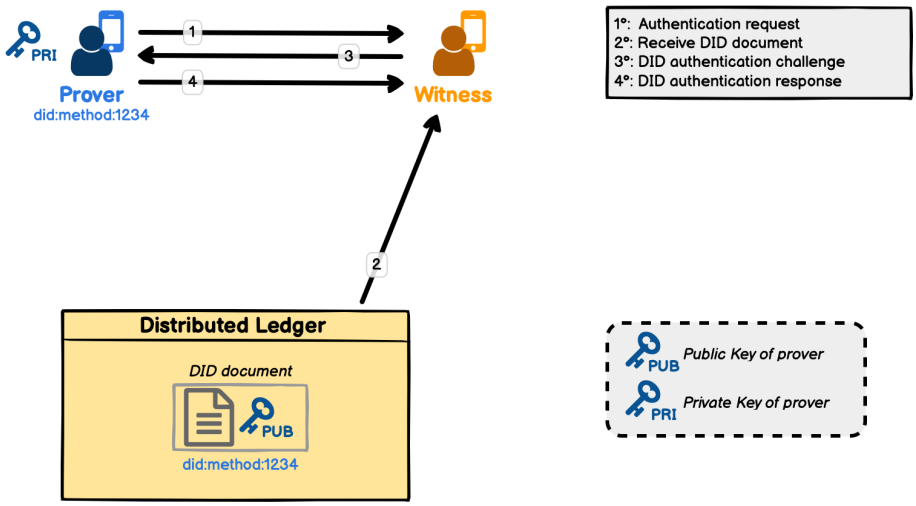


Figure 2.4: DID authentication.

specific operations that will check if their position is equal to the one received through Bluetooth. Moreover, they will also control that the credential obtained is truthful (in this case we will use the Decentralized IDentifier for simplicity). The DID is unique by design and will be used by the witness to authenticate the prover, using a *challenge-response authentication*. Without a DID, the prover will be able to use false identities.
The authentication process that allows the prover to prove the control of the DID, is shown in the figure 2.4 and works as follows:

- The witness obtains the *DID document* through the *resolution* of the prover's DID;
- The witness generates and encrypts a random value using the public key (located inside the *DID Document*) and sends the *challenge* to the prover;
- The DID owner decrypts the challenge by using its private key, which is associated with the public key. Then the prover will send the response, containing the decoded challenge, to the witness.

If everything goes well, the witness will compute the **Proof**, the *signed certificate*, and send it back to the prover (we will enter it into detail in the next section). Once the prover

receives the proof, he will insert the system's required data inside the smart contract. In this case, the data are composed of *proofs*,, *DID*, *wallet address*, the *nonce* used during the creation of the proof, and the *CID* which will be used to retrieve the report's data.
After that, the verifier will search for proof inside the smart contract and verify which ones are truthful and with valid data. In particular, he will filter for a specific DID getting the concatenation of associated values.
When the verifier validates a user and his proof, it will be up to the verifier itself to insert the data inside the hypercube [7], ensuring the **garbage-in**.

2.3 Proof of Location

As we said before in the *State of the Art* chapter, two pillars of this project consist in **computing the Proof of Location** for a prover and **verify** that the PoL and data inserted inside the smart contract are valid and legitimate.
We recall that every user is described by a unique global identifier (Decentralized IDentifier) and owns a Public and Private Key. The last one will be used to sign the proof by witnesses digitally.

2.3.1 Compute and verify a Proof

When we talk about computing the proof we are referring to the moment when the witness uses his Private Key applied to a *hash function* on prover proof. We saw that, if we use an infrastructure-independent approach, the proof could be built in different ways such as using pseudonyms and random numbers [5], so we will not stray too far from this definition.

[7] To retrieve the data, the verifier will have to use the CID and download them from the IPFS.

2.3.1.1 Build a location proof

In our work, the prover will broadcast, to a nearby witness, a request which contains the prover's *current location*, his *DID*, a random number, i.e. *nonce*, and the *CID* [8], as shown in the figure 2.5 [9]. The data located inside the request will be used by the witness

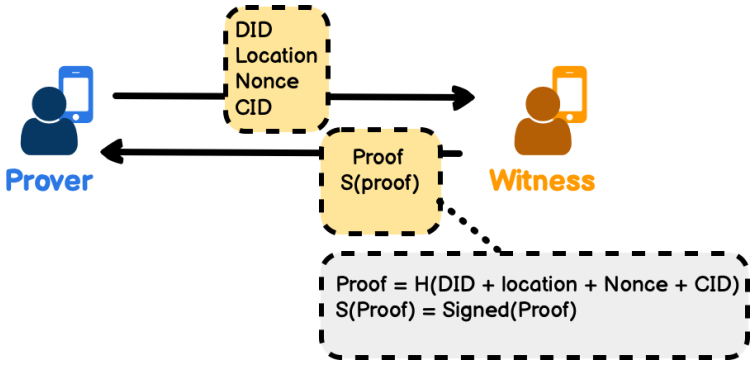


Figure 2.5: Location Proof generation.

to build a location proof and then send it back to the prover [10].

All of the parameters contained inside the request will allow the verifier to check that the proof/certificate is associated with a well-identified prover in a specific location, in a range of some witnesses. This is why we decided to hash the DID but also the location: if we hadn't put this, the verifier wouldn't be able to attest that the prover was in the position that he claimed.

For example, the prover Alice will request a location-proof certificate from the witness Bob sending her solely DID that will be used to generate the proof. Although both users are located in Bologna, Alice could enter her proof in a smart contract associated with a different location (Milan, Rome, etc.). The verifier may not be able to attest that the

[8] We suppose that the prover has already uploaded his report's data on IPFS and obtain the resulting CID.

[9] The signed proof, *S(Proof)*, is obtained applying the private key of the witness to the proof.

[10] Recall that before starting to build the location proof, the witness must authenticate the prover using the DID resolution, as described in the previous section.

location of Alice is Bologna rather than Milan or Rome.
The same thing happens for the CID and the data associated with it: if we hadn't put and subsequently hashed the CID inside the request, the prover could modify the report's data using a new CID and submit them to the smart contract, tricking the system. In this way, his proof would be invalid because associated with a CID different from the original.
As you can notice, along with the DID, location, and CID, we also hash a random number, the *nonce*, to avoid **replay attacks** seen in [18]. This type of attack consists of re-broadcasting an outdated location proof/certificate to nearby witnesses, and *nonce* is a number generated by the witness and sent to himself by the prover to avoid this type of attack. Usually, it is used in the *infrastructure-dependent* approach and it is associated with the access point as a sequence number broadcast to the nearby provers.

2.3.1.2 Verify a location proof

After computing the proof, the data inserted by provers inside the smart contract have to be verified by specific users, **verifiers**, that will subsequently insert the data inside the hypercube, ensuring **garbage in**.
One of the main aspects is that the verifier owns a **list** of *public keys of witnesses* of the system. This list will be delivered to verifiers by the Certification Authority every time a new witness is added and will be used to check which witness has signed the proof of the prover.
Generally, the verification process executed by the verifier is composed of two main parts:

1. Check that the proof is valid and has been signed by a known witness;

2. Check that the hash inside the contract, signed by the witness, is equal to the hash of the concatenation of DID, location, nonce, and CID. If their equality were confirmed, both the location and the CID would be the original declared by the prover and attested by the witness through the generation of the proof.

The figure 2.6 represents the architecture with more details regarding the verification process.

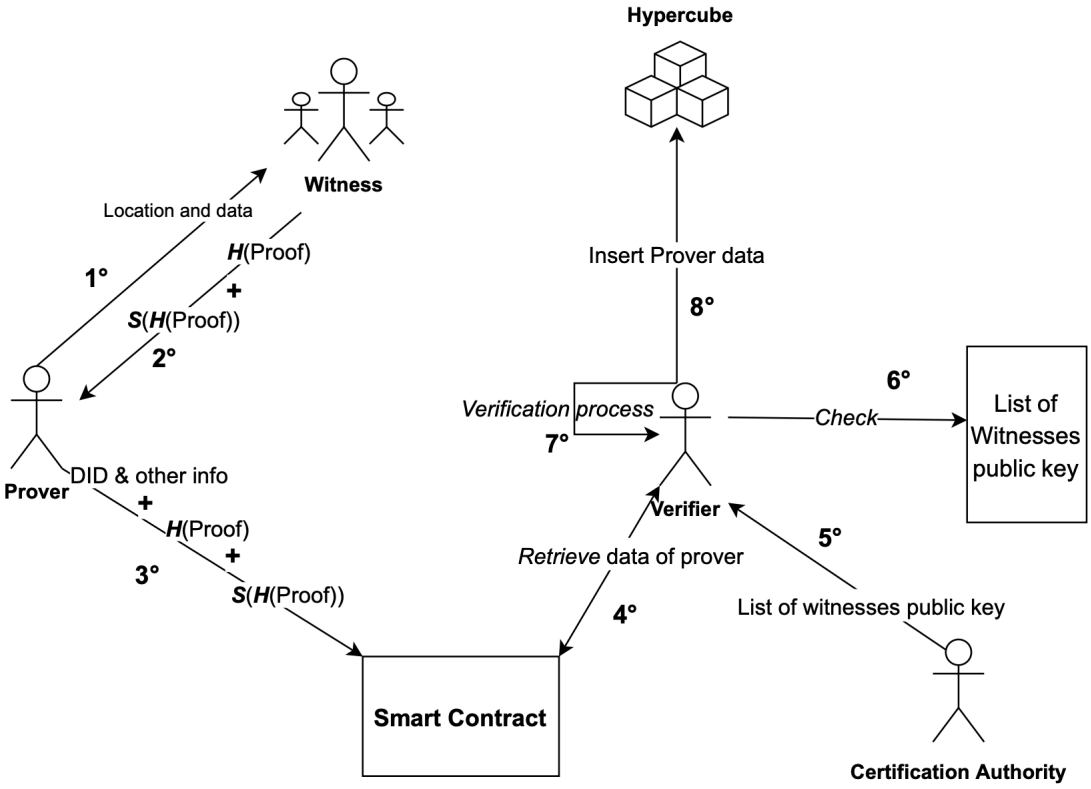


Figure 2.6: Verification architecture.

In the figure 2.6, the verifier retrieves the **signed proof**, $Sign(Hash(proof_{Prov}))$, and the **hash of proof**, $Hash(proof_{Prov})$ from the Smart Contract, *step 4°*, in addition to retrieve the data such as the blockchain *wallet* of the prover [11] and the CID containing the information that will be added to the hypercube (the title of the report, description, etc.). The reason for this is motivated in *step 7°* which is the verification process where the verifier checks if the *public key*, used to **sign** the hash of the proof, is valid or not (reading in the list of witnesses public keys) [12].

The verifier will check that the hash located inside the smart contract, $Hash(proof_{Prov})$,

[11] It is located inside the proof as we have seen previously.

[12] As the prover's Public Key may be in the witness list keys, the verifier will also check that the

is equal to the hash result obtained by applying the public key of the witness on the signed proof.
In particular, the signature of the proof is produced by the witness according to the formula 2.1:

$$SignedProof_{Prov} = PrivateKey_{Wit}(Hash(proof_{Prov})) \tag{2.1}$$

The formula below 2.2 is the verification process computed by the verifier, given that he knows both the $PublicKey_{Wit}$ (the *Public Key* of the witness) and the $Hash(proof_{Prov})$.

$$Hash(proof_{Prov}) = PublicKey_{Wit}(SignedProof_{Prov}) \tag{2.2}$$

In this way, the verifier can attest to the integrity and validity of the signature, avoiding possible fraudulent attempts by the prover to sign his own proof or cheat the system.
As shown in one of the architectural images 2.1, during the proof verification, the verifier will retrieve data from the smart contract filtering by the prover's DID.

2.4 Smart Contracts

One of the first smart contracts could be designed with the aim of producing DIDs for users that required it, while the second category of smart contracts, the most important, has the purpose of storing the prover's data. In particular, some data that has to be stored are the following:

- DID of the prover;
- Hash of the Proof;
- Signed Hash of the proof;
- Prover wallet address: used to return possible rewards;

Public Key of the prover, located inside the smart contract is different from the one of witness, and next computing the hash using both.

- The nonce used by the witness for generating the proof;
- CID (Content IDentifier) of the data.

In order to store data inside the contract, Reach's **Maps** have been used. They allow the storage of information using a *key-value* approach. However, the language is still under development, so issues have been encountered while using this data structure. The image below 2.7 represents the base content of the Map. The *key* of the Map is represented by

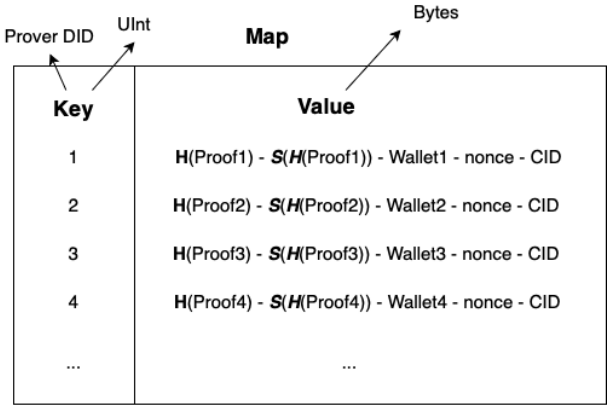


Figure 2.7: Content of the Map inside the Smart Contract.

a UInt [13], the DID of the prover [14], while the *value* is a concatenation of Bytes that will be retrieved by the verifier and used during the verification process. Every key-value row will be eliminated after the verification process.
The main Reach features designed inside the smart contract are:

- Participants: users that can interact with the program;

[13] At the writing time is not possible to use Bytes as a key type for the Map, especially if we want to use our DApp on Algorand consensus network.

[14] We are aware that the UInt format does not represent a correct DID. However, we do this only for testing purposes. Future work would include integrating the valid format of the DID inside the smart contract.

- APIs: functions that can be called by the frontend and which allow more than one user to interact with our DApp asynchronously. They can modify the Smart Contract's state, so APIs have a cost;

- Views: functions that can send values from the backend to the frontend, showing the current state of the smart contract. Since they only read the contract's state, their use does not cause any cost.

The main attributes and methods are shown in the image below 2.8: Since the smart

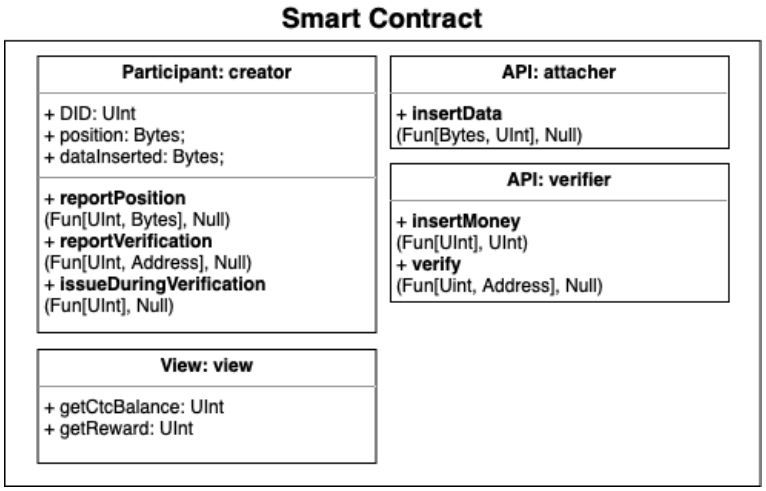


Figure 2.8: Main attributes and methods of smart contract.

contract's memory is limited, we have designed a solution where every smart contract corresponds to a specific location and it is possible to attach to them retrieving by the hypercube.

For this purpose the **factory pattern** [15] comes to our aid as well as being safer for users: its use allows the users to trust a single smart contract, the factory, and the *source code* that will be used to deploy new contracts. As many smart contracts will have to be

[15]More information about the factory pattern with smart contracts: https://betterprogramming.pub/learn-solidity-the-factory-pattern-75d11c3e7d29.

deployed, the use of *factory pattern* avoids the risk that the code being changed, during the time, causing damage to users.

2.4.1 The factory pattern

The idea of the factory pattern is to have a contract (the factory) that will carry the mission of creating other contracts, which means spawning instances using a single template. The use of this pattern be made considering the following reasons:

- Make the smart contracts management easy and allow them to be tracked and monitored;
- Save *gas fees* on Ethereum consensus network;
- Improve the contract security.

This pattern will allow associating many locations to different smart contracts, guaranteeing more scalability considering; every area will have a specific smart contract.
The association between *contract ID* and location will be stored inside the hypercube (more in the next sections).

2.5 Hypercube and IPFS

The hypercube was used to guarantee a more decentralization of the project, without relying on a central server and compromising fast queries of big amounts of data. Since storing the information on the blockchain could be expensive and time-consuming, some strategies [8] suggest storing data inside Decentralized File Storage systems, such as IPFS, and then referencing them using a Distributed Ledger Technology. However in this case will be difficult to query the blockchain when looking for specific data.
For this reason, we decided to use a hypercube which will contain only the data that

verifiers have validated. So, our application will be able to interact with the hypercube and retrieve the information with the minimum time cost. Thanks to the topology of the hypercube, objects described by similar keywords are managed by nodes that are close together. The process described in the previous sections, the garbage-in, will ensure that all the data inside the hypercube have been validated.
As you can notice from the figure 2.9, the *keyword set* of the hypercube, mapped to an *r-bit* string, is represented by the Open Location Code and the content of the nodes is a JSON that contains the following information:

- ContractID (or ApplicationID);
- Open Location Code in which the contract has been deployed;
- Array of CIDs.

When a user wants to insert information in a contract he must have to check inside the hypercube if a contract associated with the user's location exists. If not exists, the user must deploy a new contract and insert its ID inside the hypercube along with its location.
The CIDs array will be filled by the **verifier** after validates the proof and, since the data stored inside the hypercube can require a lot of space, an option could consist of the usage of IPFS. The CIDs will be used to retrieve report data such as *title*, *description*, or *images* which are stored on IPFS. In particular, when the verification process is completed, will be up to the verifier to insert the data inside the hypercube, acting as follows:

1. The verifier retrieves the data from the IPFS using the CID;
2. The verifier applies the encoding algorithm to the Open Location Code value [16], generating the binary ID of the node responsible for that position;
3. The verifier contacts the node and inserts the prover's data.

[16]The OLC value is retrieved by the smart contract.

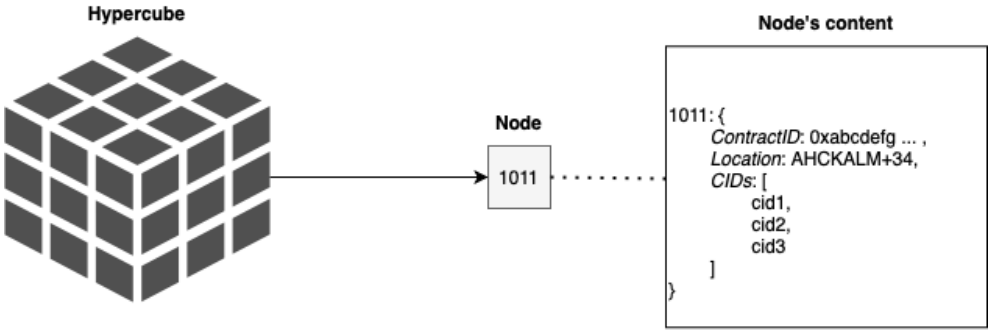


Figure 2.9: hypercube: the content of a node.

2.6 Open Location Code

The Open Location Code was used to reduce the usage of the hypercube's memory and improve the user's privacy.
In particular, before broadcasting the location to witnesses, the prover will have to retrieve its position using the GPS and convert it to the Open Location Code. As we said before, although the GPS can be spoofed, Bluetooth will guarantee that two or more users are nearby.
The substitution of GPS coordinates with an OLC will prevent the system from knowing the exact position of the user, because, as we saw in the *State of the Art* chapter, the OLC identifies an area and not a specific place. The size of the area will depend on the precision of the OLC and, for this project, was used the default precision which corresponds to **10 digits** of OLC that guarantees an area precision of *10.5m* x *13.9m* [23].

2.7 User privacy

During the design of the architecture of this project we treat personal data ensuring the user's privacy according to GDPR [56]. This is a difficult challenge to which bring a practical solution in our specific context since the majority of DLTs must be transparent,

our hypercube structure can be readable by everyone and also the IPFS's data are readable if the CID is known. In some cases, for example, it is possible to trace sensitive personal information, and track user location over a range of time and for this reason some projects, i.e., APPLAUS [5] use a pseudonym mechanism.
In our project, we don't use sensitive information, except the DID and the OLC, that are used to identify a user and an area where he is located. In addition, the DID and the wallet address are not directly connected to the user identity and both could be changed periodically. This mechanism is referred by GDPR as "*pseudonymous*" and it is not considered *fully anonymous* and, for this reason, needs to be treated with caution. Moreover, as described in the previous section, to guarantee better data protection, we didn't use the specific location of the user, but the area in which he was located.

2.8 Users rewards

This section is part of the strategy that will benefit users to participate in the project and behave correctly. The amount of reward used will be in *Algo tokens* or *ETH*, respectively the Algorand and Ethereum blockchain currency. Regarding Algorand, in the future will be possible to create a new token and transfer it, using the *Algorand Standard Assets* (ASAs), instead of using the native cryptocurrency.
Rewards will be transferred by the smart contract, which is funded by the verifier, to prover in an automatic way when specific conditions happen. Indeed, in order to validate new proof of locations, the verifier will have to insert a specific amount of Algos inside one or more contracts and a prover will receive his rewards solely if the verifier will verify his data, such as the proof, and insert them inside the hypercube. The purpose of this type of incentive is to guarantee continuous participation in the project through the insertion of new reports. The reward feature has been implemented and will be described in the implementation chapter.

We did not develop the code empowered to reward also the witnesses because they could act as a prover and report a specific situation, receiving their reward. This design choice was made to improve the number of reports associated with a location. Specifically, if a prover asks for location proof from a witness, the witness itself will have the opportunity to behave as a prover and ask, in turn, for a location-proof.
However, a new strategy could consist in send the reward to the witness after that verifier has to check his signature placed on the proof.

2.9 Tools and languages used

The languages used to develop this project are *Python*, *Javascript*, and *Reach*. We use technologies such as Docker, IPFS, and the hypercube implementation [57] and some Python libraries, i.e, Matplotlib to display charts, Web3 to interact with Ethereum blockchain, and AlgorandSDK.

2.9.1 Python

Python is one of the most popular high-level programming languages, released in 1991, that can be used by a different set of people such as software engineers, web developers, data engineers, etc. Python is object-oriented, dynamic, and interpreted language. In particular, the executed code produces a bytecode that will be used by the Python virtual machine.

2.9.2 Javascript

Javascript is a high-level language with a syntax similar to Java and interpreted means it is directly executed. Usually, it is used to write dynamic web pages and add scripts and interactive effects, but, in our case, we use the *Javascript Modules* through the use of

the *.mjs* files extension, which allows us to display our frontend with the command line interface.

2.9.3 Reach

Reach [58] is a high-level language [17], similar to Javascript, which allows the creation of DApps on a specific blockchain chosen by the programmers. Reach is **blockchain agnostic**: it is possible to run a Decentralized Application in different blockchains without code change.

At the moment the available blockchains are Ethereum, Algorand, and Conflux, but other blockchains are scheduled. In particular, the *TEAL source code* of Algorand's smart contract and the *bytecode* for Ethereum blockchain are located inside the *index.main.mjs* file, generated by Reach compiler [18].

Figure 2.10: Reach logo.

One of the big advantages of Reach is the verification process on the written code, figure 2.11. Indeed, the validity of some theorems will be checked by Reach itself to guarantee a safe and efficient program. An example is the verification of *token linearity property* which requires an empty balance when the smart contract terminates. When we compile our

```
Verifying knowledge assertions
Verifying for generic connector
  Verifying when ALL participants are honest
  Verifying when NO participants are honest
Checked 42 theorems; No failures!
```

Figure 2.11: Reach verification process.

DApp, Reach is empowered to produce the middleware and the smart contract defined

[17] Reach was released in 2020.

[18] Through Reach is it possible to obtain TEAL and Solidity source code respectively for Algorand and Ethereum.

inside the *index.rsh* file, managing the connection between the middleware itself and the smart contract. All the complexity is hidden, and for this reason, the developers don't have to specify the details about how a smart contract works at a low level, but only its rules. Indeed, in this project, users will interact with a frontend defined by *index.mjs*, which interacts with a middleware that will have a specific interface defined by our Reach program in the *index.rsh file*. The middleware is connected to the smart contract in the way shown in the image 2.12.

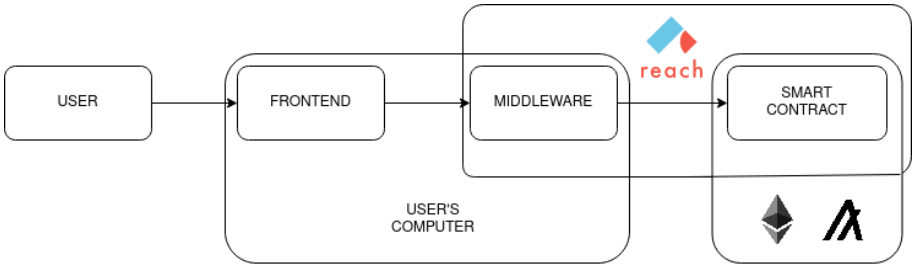


Figure 2.12: Reach model.

2.9.4 Node providers

To deploy and interact with our smart contract, and consequently interact with the blockchain, on the testnet, it was necessary to manually run a node for every blockchain that we would think to use. However, this can require specific hardware and a lot of time to sync with the network. The alternative would consist of the use of API online services, also named *node providers*, which are: Purestake to connect to the Algorand network [19], Infura to the Ropsten and Goerli networks [20], and Quicknode for Polygon [21]. To use all three services, we have to register to their online platforms and obtain the API key [22].

[19]Link to the Purestake website: https://www.purestake.com/.
[20]Link to the Infura website https://infura.io/.
[21]Link to the Quicknode website https://www.quicknode.com/.
[22]We used the free plan.

Chapter 3

Use case: environment issues reports

Everything revolves around the environment and the social responsibility of people that live on Earth.
The use case we propose is a Decentralized Application that tries to have a social impact and help our environment and, for this reason, is *sustainable* and *environment-friendly*. Initially, this DApp was designed to flag unusual behaviors towards nature and extended to many other cases. Users can report a specific situation with different typologies, such as a hole in the road, contaminated ground, waste on the street, a crowded place, cheats in tows, etc., through their smartphone and a collaborative mechanism. Moreover, users could receive a reward if their report is valid and truthful.

3.1 The Application

3.1.1 Crowdsensing

The data inserted inside the report are supposed to be textual data or images. Specifically, users have to insert a *title* and *description* of the report, and there will be opportunities to take a picture of the situation. However, to correctly submit the report, the user must use Bluetooth and ask for a location-proof generation from the nearby witness.

After receiving the proof, the system will be empowered to store all the required data inside an intelligent contract.
As we saw, this project uses many different technologies, from the Decentralized Identifier to Decentralized File Storage and Blockchain. This application is thought to be executed on the Algorand blockchain since it is considered *carbon-negative* [59], and committed to the environment. However, other blockchains are compatible with the platform since, for research reasons, we have chosen a *blockchain-agnostic* language. Thanks to this strategy, we can use incentives for users to participate in the project with a token that can be distributed as a reward. The token is *ALGO* currency if the project is run on the Algorand network, otherwise *ETH* for Ethereum or *MATIC* for Polygon.

3.1.2 Application features

Recall that the application retrieves the report from the hypercube and the algorithm that will lead to the insertion is composed of the following steps:

1. Prover creates a request for LP to insert his data such as its DID and report's data;
2. Prover sends the request for LP to the nearby witness using Bluetooth;
3. Witness check if the Prover is a neighbor using Bluetooth, computes the LP, and sends it back to the prover;
4. If a smart contract for the prover's location exists, then the prover inserts the LP inside the smart contract. Otherwise, he deploys a new smart contract, inserts its ID inside the hypercube, and finally inserts the information inside the smart contract;
5. The verifier interacts with smart contracts and checks if the LPs are valid;
6. If valid, the verifier will insert the report's data inside the hypercube and the smart contract will reward the prover.

Summarizing, the application is composed of two main tasks:

- Allow users to *insert a new report* for a specific location;
- *Display the valid reports* associated with a location.

Concerning the first point, the prover will have to interact with the smart contract to insert the report. Figure 3.1 [1] shows the contract's lifecycle of our Decentralized Application through the blockchain explorer EtherScan. This exploration allows everybody to look up the history of a specific wallet or contract address, also knowing important information such as the current balance of the contract and the type of coins/NFTs held.
The image must be read from the bottom to the top, so firstly, the creator deploys the contract, and then he inserts the information. Subsequently, some new users can attach to the contract and publish their data, and then a verifier can insert funds and verify some of the provers attached to the contract. The used testnet in the image is Goerli, and the reward token is ETH. Indeed, as you can notice, the verifier funds the contract with 0.4 ETH, a portion of which will be equally distributed to the provers during the verification process.

Concerning the hypercube, objects described by similar keywords are managed by nodes that are close together. In particular, the keyword set of the nodes is represented by the locations, so two reports in two different, but close, locations will be stored by two neighboring nodes.
The figure 3.2 shows how the application interacts with the hypercube and IPFS in order to retrieve data and display them to the user. A user will query the hypercube filtering by a specific location, and an array of CIDs, if exist, will be returned. The CIDs will be used to retrieve the real reports from the IPFS.

[1] Deciding to not verify the creator of the contract was a design choice to highlight that it is not mandatory to verify all the users.

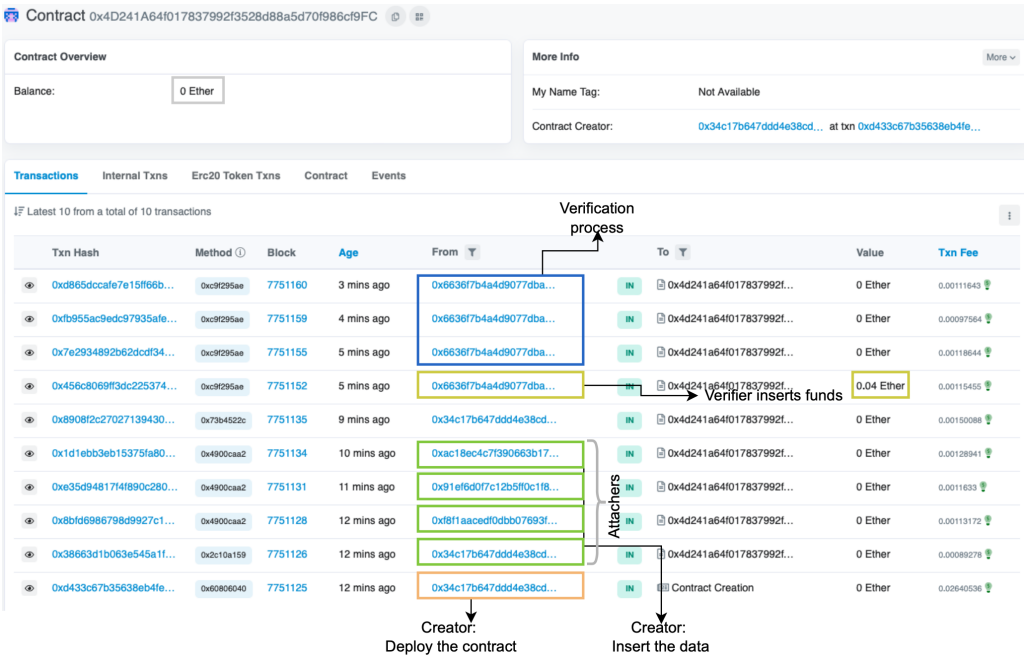


Figure 3.1: Goerli: creation, interaction and verification of a smart contract [7].

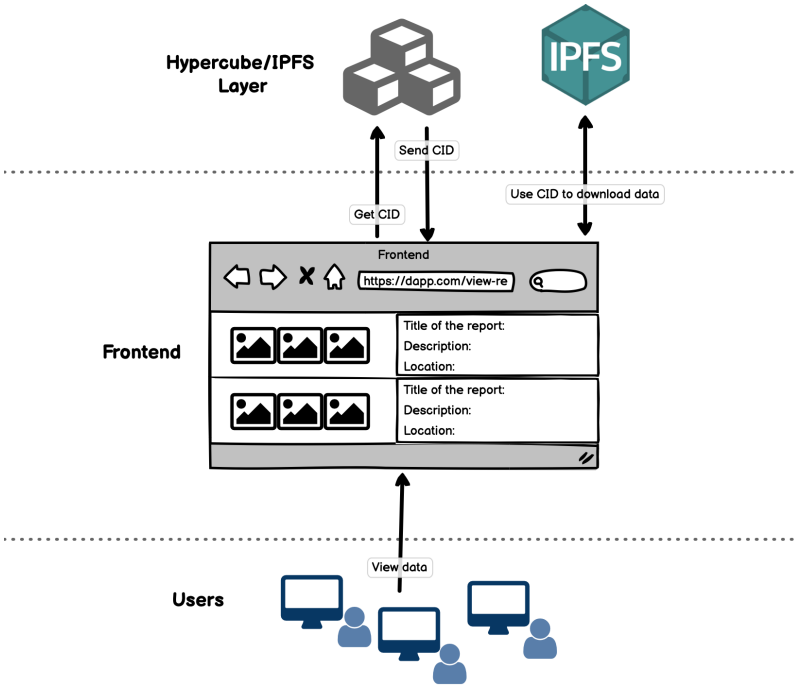


Figure 3.2: Display the data on the application.

Chapter 4

Implementation

As emerged from the previous sections, we have developed part of the presented architecture [60]. In particular, after designing the interaction between prover and witness, we decided to implement the interaction between prover and smart contract. Developing the proof generation, or interaction between the users and the hypercube, could be considered as future works. However, in our case, we focused on more than just the realization of the smart contract since we decided to test our work in three blockchains. For this reason, it was necessary to write different "frontend" code to interact with the smart contract.
Specifically, we have two folders: one is dedicated to the complete execution of the prover-contract interaction while the second, named *test-suite*, is used for evaluation and tests analysis on the three consensus networks.
In this chapter, we will see the implemented details regarding how a smart contract works, how the prover interacts with the contract, and how the test phases were built through the use of simulation scripts.

4.1 Smart contract

The smart contract has been developed using the **Reach** language and the code is located inside the *index.rsh* file which is responsible for defining the participants and the rules of the contract.
In our project, we used the following Reach features: *Participants*, *Views*, *APIs* and *ParallelReduce*.

4.1.1 Participants

Participants are defined with an interface, figure 4.1, and we can refer to them as the users that can interact with the program. A participant is defined in the backend: *Participant(participantName,participantInterface)* and the "*participantInterface*" is implemented by the frontend.

```
const Creator = Participant('Creator',{
    ...hasConsoleLogger,
    position: Bytes(128),
    did: UInt,
    data_inserted: Bytes(512),
    reportData: Fun([UInt, Maybe(Bytes(128))], Null),
    reportVerification: Fun([UInt, Address], Null),
    issueDuringVerification: Fun([UInt], Null),
});
```

Listing 4.1: Participants interface.

The snippet of code shows that we have solely one participant which is the first prover that arrives in a specific location and acts as the *creator* of the contract. The other provers will act as attachers and, although they can be considered participants, their interactions are managed using the Reach *APIs*.
Variables and functions are defined inside the interface and implemented by the frontend.

Our variables are: *position*, required to deploy the smart contract together with the DID of the creator *did* and his data, *data_inserted*. In a real use case, the DID should be of a *Bytes* type, but at the actual state, Algorand does not support indexing of Map with key type differs from *UInt* [1]. The function $reportData()$ is used to log when some users insert the data inside the contract; $reportVerification()$ is used to report when a verification success and $issueDuringVerification()$ is used when a verification fails.

4.1.2 APIs and Views

APIs are functions that can be called by the frontend and allow users to interact with the contract asynchronously; this type of behavior would not be possible if we represent attachers as participants. We developed two types of *APIs*: one to allow the attachers to interact with the contract and the other dedicated to the verification process executed by verifiers.

```
const attacherAPI = API('attacherAPI',{
    insert_data: Fun([Bytes(512),UInt], UInt),
});
```

Listing 4.2: Attachers API.

The code listed in the figure 4.2 defines the method $insert\_data()$ that could be called by an attacher in order to insert data inside the contract. In particular, this function takes as input a concatenation of values and the DID, returning a number that represents how many provers can attach to the contract.
The verifier could have two different types of interactions with the smart contract: they can insert funds and they can also verify some provers. For this reason, as the figure 4.3 shows, there are two functions: $insert\_money()$ used to insert tokens inside the contract,

[1]Link to the GitHub discussion on Algorand BoxStorage: https://github.com/reach-sh/reach-lang/discussions/1211.

and *verify*() used to verify someone. While the former takes as input the amount of money, the latter needs the DID of the prover to verify and his wallet address [2].

```
const verifierAPI = API('verifierAPI',{
    insert_money: Fun([UInt], UInt),
    verify: Fun([UInt,Address], Address),
});
```

Listing 4.3: Verifier APIs.

To call an API involves paying the network fees, for this reason, we used *Views* which are functions that can be called by the frontend and retrieve backends' values to subsequently display them without costs. An example could be displaying the contract balance or the reward that could be received. The definition of Views is represented in the figure 4.4

```
  const views = View('views', {
    getCtcBalance: UInt,
    getReward: UInt,
  });
```

Listing 4.4: Reach views.

4.1.3 Creator: create and insert data

In Reach, we can have *local steps* or *consensus steps*. The former are executed independently by participants, and the latter requires all participants to come together and agree on the computation. A Reach *local step* can occur in the body of *only* or *each* statement. An example is one of the first steps executed by the *creator* of the contract, shown in figure 4.5, where the first prover who interacts with the contract has to initialize the *position* variable. Indeed, as we saw in the project's design, every smart contract is associated with a specific location. The *interact* keyword is used to retrieve value from the frontend, defined inside *index.mjs* file. By default, every value retrieved from the

[2]It can be easily retrieved by the concatenation of values associated with the contract.

frontend is kept secret to avoid actions from malicious users. We used the *declassify*() function to make them visible.

```
Creator.only(() => {
    const pos = declassify(interact.position);
    const get_did = declassify(interact.did);
    const data_ins = declassify(interact.data_inserted);
});
```

Listing 4.5: Reach local step.

After that, the creator will have to *publish* his information on the network through the use of the Reach *publish*() function, in order to make the data transparent to other participants. In our case the used code is *Creator.publish(pos, get_did, data_ins)*.
The data inserted by the creator and subsequently by attachers will be stored inside a Reach *Map*, a data structure with a key-value approach where the key is the prover's DID and the value are all of the other data concatenated.
The *commit*() expression will allow the ending of the *consensus step* allowing more *local steps*.

4.1.4 Attacher: insert data

After the creation and the insertion operation executed by the first prover, attachers will be able to interact with the contract and insert their data. In order to do this, we used Reach's *ParallelReduce* feature: it allows multiple attachers to interact at the same time with the contract and execute the operations defined by API. Again, they are an "advanced" expression of the simple Reach's *while* condition because they allow starting a *while* with a *fork* inside, which executes code when the APIs are called. Our code will look like the one in the figure 4.6.

```
const availableSits =
  parallelReduce(SMART_CONTRACT_MAX_USER)
```

```
3     .invariant(balance() == balance())
4     .while(availableSits > 0)
5     .api(attacherAPI.insert_data,
6       (data, did, returnedSits) => {
7         returnedSits(availableSits-1);
8         easy_map[did] = fromSome(easy_map[did], data);
9         return availableSits - 1;
10    })
```

Listing 4.6: Insert prover data.

This type of structure allows many attachers to interact with our contract following the rules define inside the *ParallelReduce*. In particular, there can be a maximum number of users, defined by $SMART\_CONTRACT\_MAX\_USER$ variable, attached to the contract and the *invariant* define a condition that must be true for all time that the *while* goes on.
For the whole execution of the *while* [3], provers will be able to call the API *insert_data* that takes as input the *data* and the DID, returning how many users can still attach to the contract.

4.1.5 Verification of a prover

The verification phase is managed by a second *ParallelReduce* and only the verifiers can interact with it; it will terminate when all the users will be verified or the timeout will be triggered. In this case, the *ParallelReduce* will manage two APIs, one for the insert of funds/tokens by the verifier and the other for the verification process. Although we will not show again how the *ParallelReduce* is formed, we decided to illustrate how these two APIs have been built, both located inside the *ParallelReduce* itself.

[3] To make the DApp more similar to the use case described in the previous chapter, will be useful to change the termination condition and add a timeout that closes the contract at the end of the day or at a specific time.

There can be up to four parameters that can be passed to the Reach *api*() function, and generally structured as in the figure below 4.7.

```
.api(apiExpression,
    assumeExpression,
    payExpression,
    consensusExpression)
```

Listing 4.7: General structure of a Reach api.

The four parameter of an *api*() can be:

- *apiExpression*: the name of the API that is called;
- *assumeExpression*: the assumption that has to be true to continue the execution of the API;
- *payExpression*: the payment that the users have to do when calling the API;
- *consensusExpression*: the code to execute when the API is called.

In our case, the insertion of new funds/tokens can be defined as in the figure 4.8.

```
.api(verifierAPI.insert_money,
        (money) => {
          assume(money > 0);},
        (money) => money,
        (money, moneyInserted) => {
          moneyInserted(money);
          return keepGoing2_counter;
        })
```

Listing 4.8: Verifier inserts funds/tokens into the contract.

The API *insert_money* is responsible for fill the contract with tokens and this action is executed by the *payExpression* while in the *consensusExpression*, a mandatory field, we simply return how many tokens have been successfully inserted.

The second API is empowered to validate new users and it is shown in the figure below 4.9.

```
.api(verifierAPI.verify,
        (did, walletAddress, ret) => {
          if (balance()>=REWARD_FOR_PROVER){
            transfer(REWARD_FOR_PROVER).to(walletAddress);
            Creator.only(() => interact.reportVerification(did, this));
            delete easy_map[did];
            ret(walletAddress);
            return keepGoing2_counter -1;
          }else{
            Creator.only(() => interact.issueDuringVerification(did));
            ret(walletAddress);
            return keepGoing2_counter;
          }
        })
```

Listing 4.9: Verifier validate provers.

This API is named "*verify*", it takes as input the DID of the user to verify and his wallet address. This API will be called by a verifier after he completes the verification procedure and one of its first steps is to check if the balance of tokens is greater than the reward dedicated to the user. In our case, we used the *balance*() function to retrieve the balance of **native currency** inside the contract. In a future version of this project, this line will have to be modified if other types of tokens such as Algorand Standard Assets will be used.

If the balance isn't greater than the reward, will be called the $issueDuringVerification$ [4] function and a log error will be triggered. Otherwise, a number of tokens, representing the reward, will be sent to the wallet address of the prover through the $transfer()$ method. Then the DID and the value associated with it will be deleted from the map and the

[4]This function is implemented inside the frontend.

number of attachers to the contract will decrease by one unit.
A timeout function will be called in order to close the contract after a specific amount of time and the number of tokens that remains in the contract will be sent to the creator.

4.2 Frontend

The frontend is located inside the *index.mjs* and is a *command-line interface* implemented with JavaScript language which, in its most general sense, is responsible for the acquisition of the input data and the implementation of backend interface and methods. Moreover to interact with the smart contract we used the Reach JavaScript standard library. Through the frontend, users will be able to deploy a contract and attach to it, passing the required data.
One of its first steps consists of asking the user to enter the passphrase used to retrieve their blockchain **account** using $newAccountFromMnemonic(passphrase)$ Reach function.
To build a console interface that manages the interactions between provers and verifiers, we used the *ask* function of the Reach JavaScript standard library. This function allowed us to manage different types of inputs, for example, $Ask.yesno$ parses "Yes"/"No" answers. An example of the use of this library can be found at the beginning of the program, figure 4.10, when the user has to communicate if he already knows a contract *id*. If he does not know an *id* maybe, in the real use case, he didn't find a contract address associated with its location inside the hypercube, so he will empower to deploy a new contract.

```
1 const user_know_id = await ask.ask(‘Do you already have a contract id?‘,
      ask.yesno);
```

Listing 4.10: Ask to the user if he already knows a contract id.

Different types of actions will be executed based on the *role* of the user.
For example, the creator of the contract will have to insert the data required by the smart

contract such as the DID, figure 4.11.

```
var did = await ask.ask(
    `What is your DID?`,
    (did => did));
```

Listing 4.11: Ask DID to the creator.

The frontend has the purpose of implementing the methods defined inside the backend and passing the variables. The figure below 4.12 shows the implementation of the three creator methods (*reportData*, *reportVerification*, and *issueDuringVerification*), that will be used inside the contract.

```
const creatorInteract = {
  reportData: (did,  data) => console.log(`New data inserted \n DID: "${
   did}" \n data: "${data}"`),
  reportVerification: (did, verifier) => console.log(`DID "${did}" has
   been verified by Verifier "${verifier}"`),
  issueDuringVerification: (did) => console.log(`DID "${did}" has NOT
   been verified.`),
};
```

Listing 4.12: Implement the methods of the backend.

In the code snippet below 4.13 we pass the parameter to the backend. In particular, the creator must pass three values: his DID, the current location and the concatenation of values that will be associated with the DID inside the Reach *Map* of backend.

```
const concatData = ((params) => {
  const {proofHashed, proofSigned, walletAddress, nonce, cid} = params;
  return `${proofHashed}-${proofSigned}-${walletAddress}-${nonce}-${cid
   }`;
});

const addrCreator = stdlib.formatAddress(acc.getAddress());
var data_concat = concatData({
```

```
    proofHashed: String(hasProof),
    proofSigned: String(proof_creator_signed),
    walletAddress: addrCreator,
    nonce: nonce_inserted,
    cid: String(cid_declared)
});
creatorInteract.did = did_inserted;
creatorInteract.position = location_creator;
creatorInteract.data_inserted = data_concat;
```

Listing 4.13: Ask DID to the creator.

The creation of the contract implies its deployment which means moving the smart contract from the local machine to the blockchain network (then it could be run by everybody); this is done through the following code:

```
ctc = acc.contract(backend);
ctc.getInfo().then((info) => {
    console.log(`The contract is deployed as=${JSON.stringify(info)}`);
});
```

Listing 4.14: Deploy the contract and printing its id.

The *acc.contract*(*backend*) is a function that returns a *handle* of the Reach contract based on the backend, provided with access to the *acc* account while the $getInfo()$ method is used to print the contract *id*. Since the *backend* variable is located inside the *index.main.mjs* file, in a complete, decentralized environment without a database or server, this file could be bundled with the web app, including it inside the source directory of the project.

The contract *id*, also named contract address, will be used by other provers and verifiers to attach to the contract and interact with it4.15.

```
const info = await ask.ask(
    `Please paste the contract information:`,
```

```
    JSON.parse
);
ctc = acc.contract(backend, info);
```

Listing 4.15: Attach to a contract using his contract id.

Subsequently, if the user is a prover, he will have to call the backend API *insert_data* as follows 4.16:

```
await call(() => attacher_api.insert_data(
    String(data_concat),
    String(did)));
```

Listing 4.16: Prover call API for insert.

where, as for the deployer, the *data_concat* contains all the five parameters that have to be inserted inside the contract.
Otherwise, if the user is a verifier after attached to the contract as shown in the code 4.15, he will have to decide if inserts funds or validate some users. The verifier will have to call the *insert_money* API in order to insert funds into the contract 4.17:

```
const money_sent = await call(() => verifierAPI.insert_money(amountChose
   ));
```

Listing 4.17: Verifier call the API to inserts funds.

or the *verify* API to verify and send to someone the reward, shown in the code snippet below 4.18:

```
await call(() => verifierAPI.verify(
    parseInt(didProver),
    walletProver));
```

Listing 4.18: Verifier call the API verify.

4.3 Simulation scripts

Nowadays, many blockchains exist and are being used by developers to execute their applications. The question that we asked ourselves is which of these blockchains to choose. Logically, it would be infeasible to test our smart contract in all the consensus networks, and, for this reason, we opted to use Ethereum and Algorand. While the former, and his scaling solution such as Polygon, are really used by developers, the latter is growing up quickly and presents a lot of interesting features.

Since we would like to get the delay time, plot charts, and produce some metrics, we decided to write the simulation scripts with Python. Specifically, we re-write our frontend eliminating the interaction of the user with the *command-line-interface*, substituting it with a **complete automatize process** that creates provers and makes them interact with the smart contract.

Our *test-suite* is composed of three main files: the one that manages the execution of the simulation, *startSimulation.py* [5], the interaction with the smart contract, *index.py*, and the *index.rsh* file related to the contract itself.

To serve us Python, we have to use **Reach RPC Server** which provides access to compiled JavaScript backends via an JSON-based RPC protocol [61], based on HTTPS. This protocol allows the developer to specify the RPC method to invoke as follows: $/stdlib/METHOD$, where $METHOD$ is a function of *Reach JavaScript standard library*.

To communicate with the RPC server we used $rpc()$ and $rpc\_callbacks()$ functions: the former is used to invoke a synchronous RPC method, and the latter is useful when we have an interactive RPC method with our backend.

First of all, the script starts with the generation of N provers [6] which leads to creating

[5] The simulation will start when all the N provers are generated.

[6] We do this ensuring that the generation process of the prover will not affect the delay times.

and assigning new random *DID* [7] to every prover, locating them in a specific location, creating for each of them a new blockchain account (or retrieve it if already set). We did not focus on the creation of report details and generation of CID because this script will have to measure the blockchain performances, so their presence would not have relevance to the results.

The prover is represented by a *prover object* which has a constructor that will be called by the *generateProvers()* function, before starting the simulation; takes in input N which is the number of provers that will be generated.

A *prover object* is an instance of the *Prover's class* that implements the following methods:

1. *find_neighbours()*: Will return the *list of neighbors* in the same area of the prover that is calling the method;
2. *createAccount()*: Used to generate N *testAccount* or retrieve the ones already created;
3. *deploySmartContract()*: Used to deploy a new contract through the interaction with the *index.py* file;
4. *attachToSmartContract()*: Used to attach to a specific contract.

Concerning the 2° method, if we use a real testnet, we have to create in advance the users that will be used by the simulation script and we did this thanks to other scripts that will be presented in the next section. The public [8] and private keys, different for each blockchain, have been saved in a dictionary and will be used by *startSimulation* script and they are generated by the *support scripts*.

Subsequently, a *for loop* will be executed on every prover as follows:

1. For every prover, check if he has some neighbors. In a real case, this corresponds to the moment when users use Bluetooth to check who is nearby;

[7] The generation process of random *DID* is done ensuring that everybody has a unique identifier.

[8] Also referred as the *wallet address*.

2. We have simulated the interactions with the hypercube. In particular, the user will check inside the hypercube if a contract ID associated with his position already exists: if not exist user will deploy a new contract with *deploySmartContract*() function, otherwise will retrieve the contract ID and then call *attachToSmartContract*().

Since Reach requires that the steps inside the smart contract are executed in a certain order, we used the *Thread* library. The snippet of code below represents the deployment of the contract 4.19:

```
def deploySmartContract(self, proverObject):
    ctc_creator = rpc("/acc/contract", proverObject.account)
    creatorThread = Thread(target=play_Creator,
                           args=(ctc_creator,
                           proverObject.location,
                           proverObject.did,
                           proverObject.data))
    return creatorThread, ctc_creator
```

Listing 4.19: Call to the deploySmartContract() function.

The *play_Creator*(), called when the *Thread* starts, is imported from the *index.py* file and is empowered to execute the deployment through the interaction with the backend 4.20. Specifically, *play_Creator*() sends the creator's contract *RPC handle* [9] and his implemented participants interface to the server using the *rpc_callbacks* function.

```
def play_Creator(contract_creator, position_ins, did_ins, data_ins):
    rpc_callbacks(
        '/backend/Creator',
        contract_creator,
        dict(position=position_ins,
            did=did_ins,
            data_inserted=data_ins,
```

[9] An *RPC handle* is a string that represents the corresponding resource, i.e., an account representation, and is included in the RPC response.

```
8             **player('Creator')),)
```

Listing 4.20: Creator deploy the contract.

In our case, the *rpc_callbacks* will interact with the backend, passing the values inside the *dictionary*, where the *player*() function allows us to implement the methods defined in the *participant* contract interface. In our case this is represented by the snippet of code below 4.21:

```
1  def player(who):
2      def reportPosition(did,  data):
3          ...
4      def reportVerification(did, verifier):
5          ...
6      def issueDuringVerification(did):
7          ...
8      return {'stdlib.hasConsoleLogger': True,
9              'reportPosition': reportPosition,
10             'reportVerification':reportVerification,
11             'issueDuringVerification':issueDuringVerification}
```

Listing 4.21: Implementation of the participant interface.

As we said before, we have simulated the interaction with the hypercube: if the smart contract is not associated with a location in the dictionary, the deployment will be executed. Otherwise, it will be up to the attach method called by *attachToSmartContract*() as in the figure below 4.22:

```
1  def attachToSmartContract(self, provObj, ctc_creator):
2      attacherThread = Thread(target=play_bob, args=(ctc_creator, provObj.
     account, provObj.data, provObj.did,))
3      return attacherThread
```

Listing 4.22: startSimulation.py: call the attachToSmartContract().

In this case, the attach will be performed by the *play_bob*() function defined inside the *index.py*; it will be empowered to *opt-in* to the contract and insert the information of the prover 4.23:

```
def play_bob(ctc_user_creator, accc, data_concat, did):
    ctc_bob = rpc("/acc/contract", accc, rpc("/ctc/getInfo",
   ctc_user_creator))
    result_counter = rpc('/ctc/apis/attacherAPI/insert_data', ctc_bob,
   data_concat, did)
```

Listing 4.23: index.py: perform the attach.

Regarding the verifier APIs, we will not enter into details since they are very similar to the attach operation, the only difference consists of specifying the name of API to call, for example, the snippet of code 4.24 represents the API used to insert a specific amount of tokens (*native currency*) inside the contract [10].

```
def verifier_pay(ctc_user_creator,accc):
        ctc_verifier = rpc("/acc/contract", accc, rpc("/ctc/getInfo",
   ctc_user_creator))
        money_payed = rpc('/ctc/apis/verifierAPI/insert_money',
   ctc_verifier, SMART_CONTRAT_PAYMENT)
```

Listing 4.24: Verifier inserts funds inside the contract.

4.4 Support scripts

Usually, in order to interact with the blockchain, a minimum balance inside the wallet is required. For this reason, we have to create, and subsequently fund, the wallets used on each of our three consensus networks. For example, for what concerns

[10] SMART_CONTRACT_PAYMENT is a constant defined as *rpc*("/*stdlib*/*parseCurrency*", 0.5). The *parseCurrency* Reach's method, is used to format a specific amount of money that will be sent to the backend.

Algorand, we wrote and used *NewAcc_algo.py* file to create N accounts using the *account.generate_account*() of Algorand SDK function, storing the public and private keys generated. Every wallet was manually funded using the **Algorand online dispenser**, that act as a *faucet of Algos* [11].

Concerning Ethereum, we created a new script that interacts with *node providers* to create new wallets through the use of the *Web3* library. However, in this case, we have to make an additional component empowered to send funds to the wallet, since the Ethereum *faucet*, usually, can send a minimum amount of tokens [12] only to one account for a day. The quantity of the tokens distributed by Ethereum and Polygon *faucets* was not enough, therefore, we decided to contact the testnet administrators and ask them for a large number of tokens. Subsequently, inside the *eth_new_account.py* file, we created a method empowered to distribute the received tokens from the one communicated to the admin to the other created accounts.

To send these transactions, we first have to define the endpoint of the *node provider* such as "*https://goerli.infura.io/API_APIKEY*" used for the Goerli network. The next steps include creating a transaction specifying the number of tokens to send and other mandatory parameters. Subsequently, the sender will sign the transaction using his private key and propagate it as the code snippet below shows 4.25.

```
tx = {
    'nonce': nonce,
    'to': to_address,
    'value': web3.toWei(0.3, 'ether'),
    'gas': 200000,
    'gasPrice': gasPrice
}
signed_tx = web3.eth.account.sign_transaction(tx, private_Key)
```

[11] Every round of fund, send 10 Algos from the faucet to the receiver address. Link to the *faucet*: https://bank.testnet.algorand.network/

[12] Between 0.1 and 0.5 ETH for Goerli testnet, and between 0.5 and 5 MATIC for Mumbai Polygon.

```
9 tx_hash = web3.eth.send_raw_transaction(signed_tx.rawTransaction)
```

Listing 4.25: Fund a wallet.

The *send* method will be called from within a *for loop* which will be executed for every wallet.

As you can notice, both Algorand and Ethereum present some limits if we want to test with many accounts and this is due to the minimum balance and the procedure to fund them. Moreover, Ethereum gas fees depend on the congestion of the network and create many transactions with the purpose of sending tokens might cause high transaction costs.

4.5 Execution of the project

In this section, we will show some behaviors of our smart contract when *command-line interface* is used as *frontend*.

Firstly, to execute the project we have to specify the consensus network, for example, we can use "*REACH_CONNECTOR_MODE=ETH ./reach run*" and run on Ethereum Reach *devnet*. We can also set the *connector* to *ALGO* if we prefer the Algorand blockchain.

To make the test more understandable, figure 4.1, we set the maximum number of attachers, defined inside the contract, to 2. Then we created and deployed the contract and, in two other terminals, we respectively attached one user, verifying both the two provers with a verifier account. [13]. As you can notice, in the image is also shown the log of the verification process of our two prover DIDs, which are 9999 and 12.

To execute the *test-suite* the command is "*REACH_CONNECTOR_MODE=ETH ./reach rpc-run python3 -u startSimulation.py*" and to test it on the real testnet the connector must switch to *ETH-live* or *ALGO-live*. Then you will have to apply some changes inside the *startSimulation* script, setting the *ETH_NODE_URI* for Ethereum be-

[13]The signed proof, hashed proof, and CID information were purely invented.

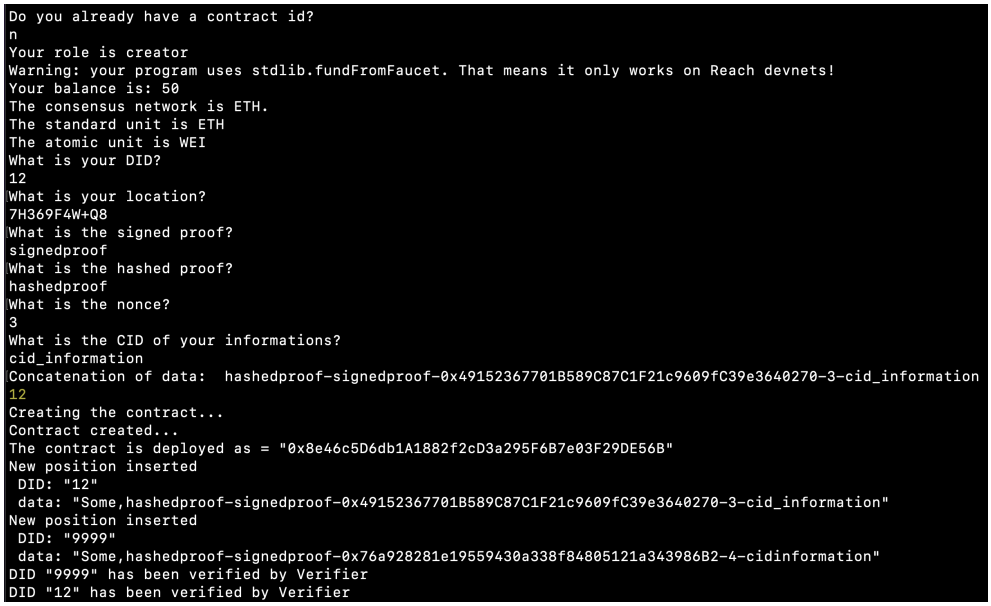


Figure 4.1: Execution of a contract with two attachers on the Ethereum blockchain.

tween Goerli and Polygon Mumbai. You also have to set to a new RPC method when creating/retrieving a blockchain account, switching from "*/stdlib/newTestAccount*" to "*/stdlib/newAccountFromSecret*" specifying the passphrase as the parameter. The image below represents part of the initial execution of our *test-suite* on the Goerli testnet 4.2.

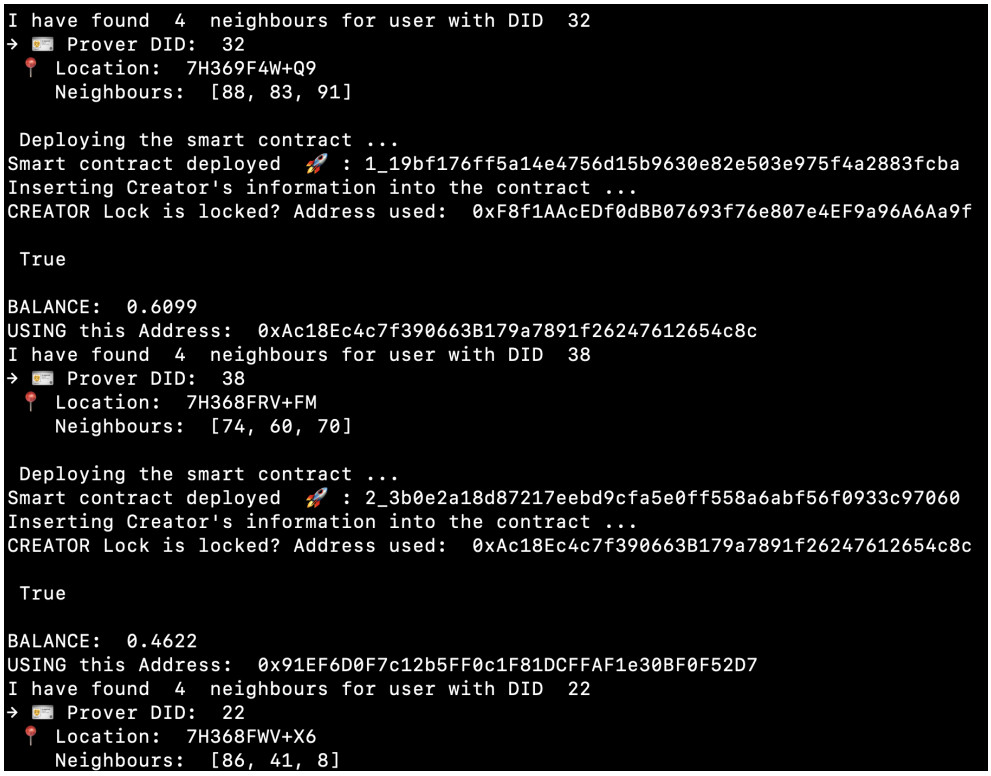


Figure 4.2: Execution on the Ethererum Goerli testnet.

Chapter 5

Performance evaluation

The Algorand and Ethereum performances' systems have been measured focusing on **transactions latency** and **transaction costs** both for the deployment and for the attach operations. The deploy function integrates an insert function of the creator's data, so it isn't a merely deploy.
Precisely, we measure the performances using the Ropsten [1], Goerli and Polygon [39] Mumbai testnet [2] concerning the Ethereum blockchain. At the same time, we use Algorand testnet for Algorand itself.

5.1 Results

We decided to measure only the deploy and attach phases of the insert operation into the contract, excluding the verification process. This was made because the verify operation is similar to the attachment since it is a basic API call to the contract.
We tested the smart contract architecture with different numbers of users: 8, 16, 24, and 32, and we created the corresponding numbers of smart contracts: 2, 4, 6, and 8. Recall

[1] Only a few tests have been completed causing the deprecation of the testnet. For this reason, we decided to replicate the tests on a new Ethereum testnet, Goerli.

[2] The base fee was between 40 and 70 gwei with 1.5 gwei for the priority fee.

that every smart contract must have four users attached to it (contract creator included). We didn't have the opportunity to test our DApp with more users since every one of them needs a minimum amount of tokens as balance, and this is not easy to achieve with many users.

5.1.1 General analysis

Before entering into the details of our two consensus networks, we executed a general analysis using the Reach tool; the output is shown in the figure 5.1.

Figure 5.1: Conservative analysis of the smart contract.

As we can see, the output shows some information such as the amount of memory used, the steps of our smart contract and fees. Concerning the fees, they are blockchain agnostic, so they do not represent the exact amount of ALGOs or gas fees, but they can be easily derived using the amount of fees displayed.

In addition, both Goerli and Polygon, have a deployment process that used **1,440,385 gas** while the amount of gas used for the attach is **82,437**.

5.1.2 Ethereum performances

We measured the Ethereum transactions speed on the **Ropsten testnet**, using *eight* provers where *two* of them are creators and *six* are the neighbors. As the chart in figure 5.2 shows, the interaction time between users and smart contracts is unstable and can be very high.

Specifically, the **deploy** phases (the first and the fifth bar) are the ones that require more time while the **attach** need less time due to fewer transactions for each user.

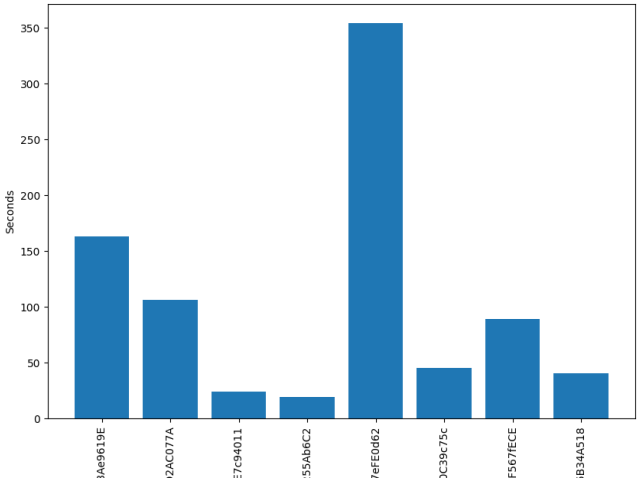


Figure 5.2: Ethereum Ropsten Testnet: performance of 8 transactions.

The reasons for this big instability and executions transactions time can be caused by the Ethereum gas fees system and Ropsten testnet itself. Since miners validate transactions with higher fees first, this may have impacted the performances.

Moreover, the Ropsten testnet has been deprecated, so we have decided to make more tests on different networks such as Goerli. We have made four different tests in order to

measure the transactions speed:

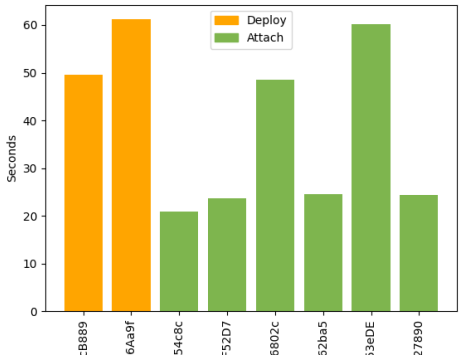
- **1° Test**: using 8 provers where 2 of them are creators and 6 are the neighbors;
- **2° Test**: using 16 provers where 4 are creators and 12 are the neighbors;
- **3° Test**: using 24 provers where 6 are creators and 18 are the neighbors;
- **4° Test**: using 32 provers where 8 are creators and 24 are the neighbors.

The smart contracts have been deployed for up to 8 different positions which are: 7H369F4W+Q8, 7H369F4W+Q9, 7H368FRV+FM, 7H368FWV+X6, 7H367FWH+9J, 7H368F5R+4V, 7H369FXP+FH and 7H369F2W+3R.
The chart does not display the time for each transaction, but the total interaction time between one user and the smart contract, i.e., if the smart contract is already deployed only two transactions will be needed to interact with the contract realizing the **attach** operation.
The performance on Goerli testnet is shown in the figure 5.3. Regarding the first image of the figure 5.3a, the first two users are the ones that deploy the two contracts; in the second images, 5.3b, the first four users are the deployers; in the last images, 5.3c, the first six users are the deployers while the others are the attachers, and so on.

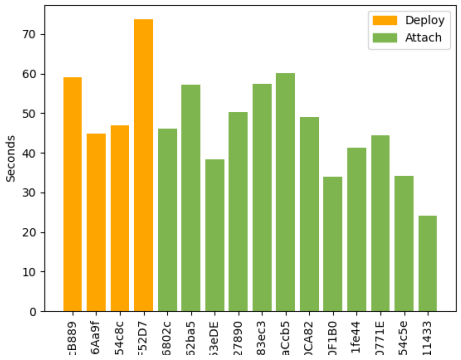
In general, the attacher time requires less time than the deploy phase. However, sometimes, an attach operation could require more time than a deployment and this can be seen in the figure 5.3d. We can also notice that the required time is only sometimes stable and this may be due to the congestion of the network.

5.1.3 Polygon performances

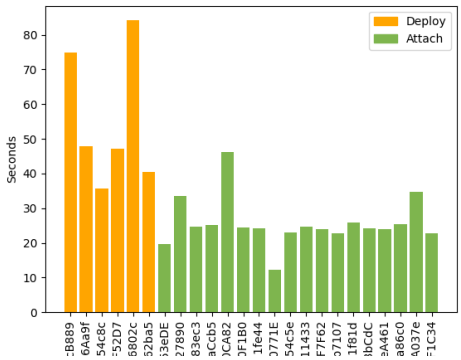
Regarding Polygon, the results are shown in figure 5.4. As you can notice, and we shall see in a more detailed analysis, neither Polygon has a stable transaction time.
Indeed, some users may take longer, and others may take less time.



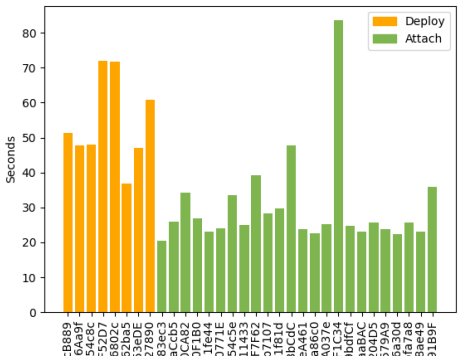
(a) Goerli: performances with 8 users.



(b) Goerli: performances with 16 users.



(c) Goerli: performances with 24 users.



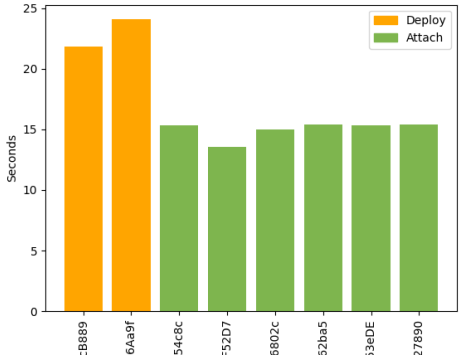
(d) Goerli: performances with 32 users.

Figure 5.3: Goerli performances.

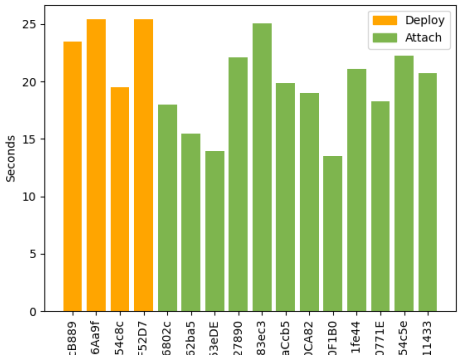
However, the fact that it is a *layer-2* and *off-chain* solution leads to processing many transactions per second and allows it to be faster than the Ethereum Goerli testnet, taking less than half the time.

However, as we shall see in the next section, it is only sometimes convenient to leverage Polygon to deploy smart contracts.

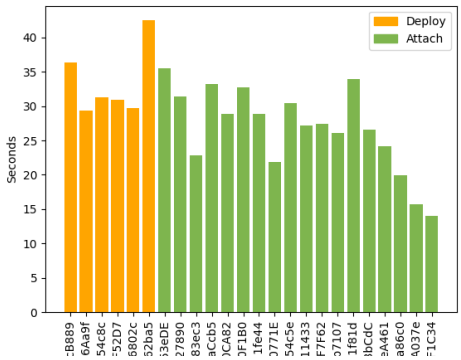
This is because the cost of its transactions, as happens for Ethereum, is influenced by the congestion of the network. That means if the network is very busy, we could pay a lot for the transaction fees.



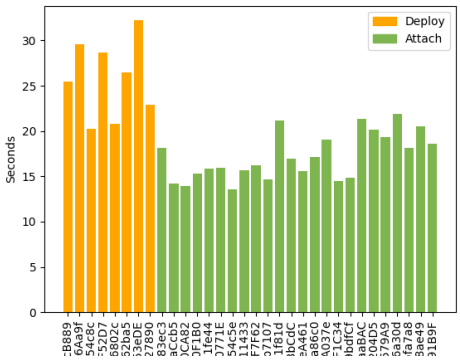
(a) Polygon: performances with 8 users.



(b) Polygon: performances with 16 users [62]



(c) Polygon: performances with 24 users.

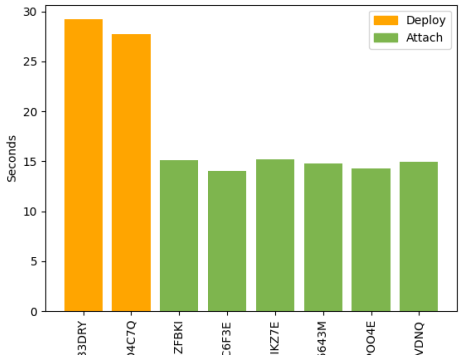


(d) Polygon: performances with 32 users.

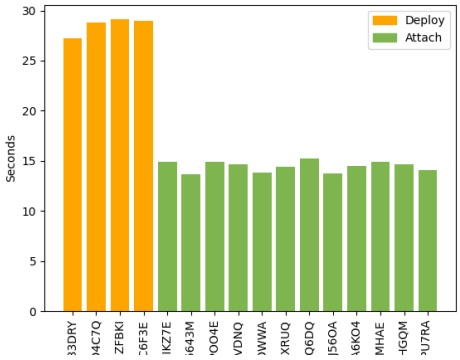
Figure 5.4: Polygon performances.

5.1.4 Algorand performances

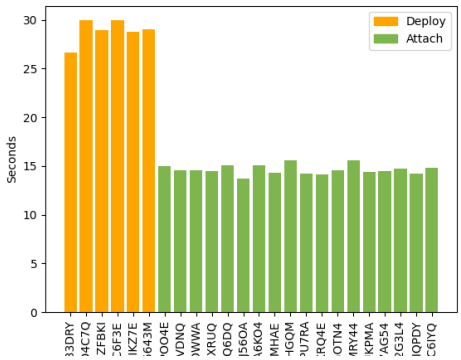
Concerning Algorand, the results of our analysis, shown in the figure 5.5, highlighted that Algorand has a low and stable total transaction times compared to Ethereum.
Indeed, although its latencies are very similar to those of the Polygon chain, there is little dispersion of the required time for each user interacting with the blockchain.
That means the deployment time is always the same for each user, and the same happens for the attach time.
This can be identified as a sign of the stability and efficiency of the network.



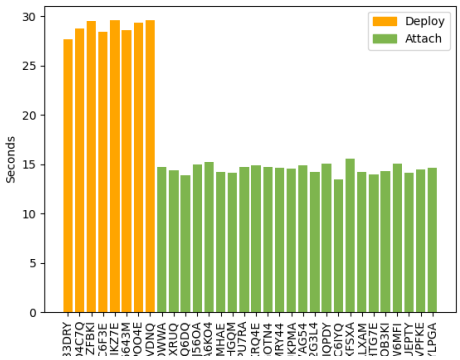
(a) Algorand: performances with 8 users.



(b) Algorand: performances with 16 users.



(c) Algorand: performances with 24 users.



(d) Algorand: performances with 32 users.

Figure 5.5: Algorand performances.

5.1.5 Results comparison

Finally, we decided to compare some of our results. In particular, we compared the two operations, deploy and attach, for the test with 16 and 32 users [3].

The following tables show some metrics to identify better which could be the most suitable blockchain for our use case. Due to the expensive cost of Goerli and Polygon, it was impossible to compute the metrics on the same day and, for this reason, the results were

[3]The price are computed on November 17th, 2022: an Ether costs €1156, an ALGO €0.26, and one MATIC costs €0.85.

calculated on different days. In particular, the results of Algorand and Goerli are in the table 5.2 and 5.4 have been computed the day after calculating the metrics for Polygon.

Deploy | 16 users						
Testnet	Mean	Max	Min	Dev Std	Fees	Euro
Goerli	56.15s	73.7s	44.85s	11.52s	0.06 ETH	€ 69
Polygon	23.44s	25.41s	19.52s	2.4s	0.002 Matic	€ 0.002
Algorand	28.53s	29.12s	27.23s	0.76s	0.005 Algo	€ 0.001

Table 5.1: Performances of the deployment operation, with 16 users.

Deploy | 32 users						
Testnet	Mean	Max	Min	Dev Std	Fees	Euro
Goerli	54.4s	71.86s	36.84s	11.74s	0.019 ETH	€ 21.96
Polygon	25.78	32.22	20.27	4.02	0.002 Matic	€ 0.002
Algorand	28.93s	29.57s	27.66s	0.64s	0.005 Algo	€ 0.001

Table 5.2: Performances of the deployment operation, with 32 users.

We can notice from the first table, 5.1, that Algorand's deployment requires, on average, more time than a deployment on Polygon (28.53 seconds instead 23.44) considering 16 users test. However, the Algorand *standard deviation* seems to be nice below the other two blockchains, which means there is little dispersion, both for the deployment and attach. This is a very important measure since it shows us that Algorand is more stable, requiring the same time for every deploy operation and attach operation respectively. It also seems that Algorand's cost is greater than the Polygon blockchain, however, Polygon can reach very expensive transaction cost, increased by more than 100%, as we can see from our

test on a different day where there was some network congestion [4], while on Algorand this doesn't happen. In addition, the same thing also happens for Goerli which can reach a cost equal to 0.1401 ETH [5] which corresponded to €171.18 on October 21 2022.
If we compare the table containing the test with 16 users 5.1 and the one with 32 users 5.2, we can notice that Algorand maintains the same performance while the other two blockchains do not. The required time for Goerli and Polygon is greater than Algorand. In addition, Algorand executed more transactions than the other two blockchains, in the deployment phase, due to the design of the network. For this reason, it may take longer. In the tables below, we compare the attach results for both 16 and 32 users. Also, in this case, using a different number of users led to a different amount of time required by Goerli and Polygon, while not on Algorand, which has approximately the same time between the test with 16 and 32 users. The attach operation for Algorand is faster than the other two

Attach seconds 16 users						
Testnet	Mean	Max	Min	Dev Std	Fees	Euro
Goerli	35.95s	44.47s	24.03s	7.84s	0.0137 ETH	€ 15.83
Polygon	20.6s	22.23s	18.3s	1.44s	0.00053 Matic	€ 0.0004
Algorand	14.54s	14.93s	14.08s	0.31s	0.009 Algo	€ 0.002

Table 5.3: Performances of the attach operation, with 16 users.

blockchains and this can be due to the fact that Algorand can support many transactions per second, thanks to the recent upgrade that happened on September 2022 which allows Algorand to manage until 6,000 transactions per second. Finally, the attach operation generally requires few fees, except for the Goerli testnet which has reached 0.0137 ETH

[4]Expensive fees on Polygon Mumbai: Contract 0x9e978434d0334ff305c4d99402c0bb141eb0504f.
[5]Goerli fees: Contract 0x15d19f65f81d9e7c5a095dc6bdac5ab36bb6c16dcf1a14045749b0c60489ce4b.

Attach | 32 users						
Testnet	Mean	Max	Min	Dev Std	Fees	Euro
Goerli	25.56s	83.53s	22.47s	4.06s	0.003 ETH	€ 3.46
Polygon	19.35s	21.87s	14.82s	2.09s	0.00053 Matic	€ 0.0004
Algorand	14.54s	15.59s	13.96s	0.5s	0.009 Algo	€ 0.002

Table 5.4: Performances of the attach operation, with 32 users.

that is €15.83 [6].

[6]The transaction fees of the attach operation is computed making the sum of the cost for each user that attach to the contract.

Conclusion

In this work, we presented an architecture focused on the Proof of Location problem, and we proposed a use case where users could collect data through a collaborative approach that consists of an environment-friendly application, making reports considering the context where they are, reporting pollution cases, natural disasters, vandalism, etc. Specifically, we start by highlighting the possibility of a malicious user that uses this kind of location-based application that offers some incentive to spoof his position and unfairly obtain the reward.

After working attentively researching amongst the related works, we decided to realize our architecture using an infrastructure-independent approach, which means we will use the witnesses' presence near the prover to attest his location instead of relying on access points which would involve a high economic cost. In particular, witnesses will be empowered to generate the proof/certificate that will be sent to the requesting prover and stored inside our system.

The architecture was designed using DLT and DFS, which means blockchains, DHT, and IPFS. These tools allow our infrastructure to inherit their properties, such as true decentralization, data integrity, immutability, security, and transparency. Again, one of the most important features of our use case concerns its complete decentralization because, thanks to the blockchain and DHT, we could remove the single point of failure of a centralized server/database.

We implemented part of the proposed architecture, focusing on the interaction between

provers, verifiers, users empowered to validate the location proofs, and smart contracts choosing Ethereum, Polygon, and Algorand blockchains because of their performances and the possibility to execute code over them. Our smart contract is written thanks to the use of Reach, a blockchain-agnostic language that is able to generate the source code for different consensus networks from the starting code.

Finally, we evaluate the performances of the interactions between users and the blockchains, measuring the time required to deploy a new contract and attach to it, quantifying the amount of fees needed from these operations. There, we found that Algorand, a layer-1 solution that solves the blockchain Trilemma, performs better than the other two blockchains in every test that we produced, considering both the aspect of transaction time and their cost.

Future works will be focused on the enhancement of our architecture, building a mobile app, which would be used to generate the proof, and the respective software needed by the verifier to validate the smart contract content, and also implementing the features that allow the users to interact with the DHT, used to store the reports. Moreover, it will be useful to modify the architecture proposed by us to solve the issues of the collusion attacks.

Bibliography

[1] Yuh-jzer Joung, Li-wei Yang, and Chien-tse Fang. Keyword search in dht-based peer-to-peer networks. *IEEE Journal on Selected Areas in Communications*, 25(1):46–61, 2007.

[2] Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf.

[3] Brian Haney and Archie Chaudhury. Algorand autonomous. *Brian Haney and Archie Chaudhury, Algorand Autonomous*, 2021.

[4] Chitra Javali, Girish Revadigar, Kasper B Rasmussen, Wen Hu, and Sanjay Jha. I am alice, i was in wonderland: secure location proof generation and verification protocol. In *2016 IEEE 41st conference on local computer networks (LCN)*, pages 477–485. IEEE, 2016.

[5] Zhichao Zhu and Guohong Cao. Applaus: A privacy-preserving location proof updating system for location-based services. In *2011 Proceedings IEEE INFOCOM*, pages 1889–1897, 2011.

[6] Michele Amoretti, Giacomo Brambilla, Francesco Medioli, and Francesco Zanichelli. Blockchain-based proof of location. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 146–153, 2018.

[7] Goerli etherscan: example of a smart contract interactions. https://goerli.etherscan.io/address/0x4d241a64f017837992f3528d88a5d70f986cf9fc.

[8] Mirko Zichichi, Stefano Ferretti, and Gabriele D'Angelo. A distributed ledger based infrastructure for smart transportation system and social good. In *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE, 2020.

[9] Xinlei Wang, Amit Pande, Jindan Zhu, and Prasant Mohapatra. Stamp: Enabling privacy-preserving location proofs for mobile users. *IEEE/ACM Transactions on Networking*, 24(6):3276–3289, 2016.

[10] Mohammad Reza Nosouhi, Shui Yu, Wanlei Zhou, Marthie Grobler, and Habiba Keshtiar. Blockchain for secure location verification. *Journal of Parallel and Distributed Computing*, 136:40–51, 2020.

[11] Zengbin Zhang, Lin Zhou, Xiaohan Zhao, Gang Wang, Yu Su, Miriam Metzger, Haitao Zheng, and Ben Y Zhao. On the validity of geosocial mobility traces. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013.

[12] Foursquare website. https://foursquare.com/.

[13] Kexiong Curtis Zeng, Yuanchao Shu, Shinan Liu, Yanzhi Dou, and Yaling Yang. A practical gps location spoofing attack in road navigation scenario. In *Proceedings of the 18th international workshop on mobile computing systems and applications*, pages 85–90, 2017.

[14] Apple maps gets drivers lost in australian out back, police warn, 2012. https://www.cnet.com/tech/mobile/apple-maps-gets-drivers-lost-in-australian-outback-police-warn/.

[15] Gps spoofing with uber. https://soliddriver.com/index.php?a=GPS-Spoofing-A-Growing-Problem-for-Uber.

[16] Evangelos Pournaras. Proof of witness presence: blockchain consensus for augmented democracy in smart cities. *Journal of Parallel and Distributed Computing*, 145:160–175, 2020.

[17] Sébastien Gambs, Marc-Olivier Killijian, Matthieu Roy, and Moussa Traoré. Props: A privacy-preserving location proof system. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 1–10. IEEE, 2014.

[18] Stefan Saroiu and Alec Wolman. Enabling new mobile applications with location proofs. In *Proceedings of the 10th workshop on Mobile Computing Systems and Applications*, pages 1–6, 2009.

[19] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[20] Mirko Zichichi, Luca Serena, Stefano Ferretti, and Gabriele D'Angelo. Governing decentralized complex queries through a dao. In *Proceedings of the Conference on Information Technology for Social Good*, pages 121–126, 2021.

[21] Introduction to open location code. https://github.com/google/open-location-code/blob/main/docs/olc_definition.adoc#farvacque.

[22] Geohash website: example with "c216new" string. http://geohash.org/c216new.

[23] Precision of open location code. https://github.com/google/open-location-code/blob/main/docs/olc_definition.adoc#open-location-code.

[24] Mirko Zichichi, Luca Serena, Stefano Ferretti, and Gabriele D'Angelo. Complex queries over decentralised systems for geodata retrieval. *IET Networks*, 2022.

[25] Yang Lu. Blockchain and the related issues: a review of current research topics. *Journal of Management Analytics*, 5(4):231–255, 2018.

[26] Algorand's core technology (in a nutshell). https://medium.com/algorand/algorands-core-technology-in-a-nutshell-e2b824e03c77.

[27] Block confirmation. https://en.bitcoin.it/wiki/Confirmation.

[28] Bitcointalk: introduction to the proof-of-stake. https://bitcointalk.org/index.php?topic=27787.0.

[29] Proof of stake article. https://www.algorand.com/resources/blog/proof-of-stake-vs-pure-proof-of-stake-consensus.

[30] Hyperledger fabric. https://www.hyperledger.org/use/fabric.

[31] Szabo, n. the idea of smart contracts. https://nakamotoinstitute.org/the-idea-of-smart-contracts/.

[32] What is ethereum? https://ethereum.org/en/what-is-ethereum/.

[33] Ethereum: A next-generation smart contract and decentralized application platform. by vitalik buterin (2014). https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[34] Ethereum nodes tracker. https://etherscan.io/nodetracker.

[35] Ethereum documentation. https://ethereum.org/en/developers/docs/gas/.

[36] Mirko Bez, Giacomo Fornari, and Tullio Vardanega. The scalability challenge of ethereum: An initial quantitative analysis. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 167–176. IEEE, 2019.

[37] Ethereum sharding. https://ethereum.org/en/upgrades/sharding/#what-is-sharding.

[38] Ethereum layer 2. https://ethereum.org/en/layer-2/#what-is-layer-2.

[39] Polygon. https://polygon.technology/.

[40] Jing Chen and Silvio Micali. Algorand. *arXiv preprint arXiv:1607.01341*, 2016.

[41] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

[42] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.

[43] Algorand virtual machine. https://developer.algorand.org/docs/get-details/dapps/avm/.

[44] Algorand metrics: Number of running nodes. https://metrics.algorand.org/#/decentralization/.

[45] How big is algorand. https://developer.algoscan.app.

[46] Oscar Avellaneda, Alan Bachmann, Abbie Barbir, Joni Brenan, Pamela Dingle, Kim Hamilton Duffy, Eve Maler, Drummond Reed, and Manu Sporny. Decentralized identity: Where did it come from and where is it going? *IEEE Communications Standards Magazine*, 3(4):10–13, 2019.

[47] Decentralized identifiers did: W3c reccomendation. https://www.w3.org/TR/did-core/.

[48] W3c: Did methods. https://www.w3.org/TR/did-core/#dfn-did-methods.

[49] Min-Hyung Rhie, Kyung-Hoon Kim, DongYeop Hwang, and Ki-Hyung Kim. Vulnerability analysis of did document's updating process in the decentralized identifier systems. In *2021 International Conference on Information Networking (ICOIN)*, pages 517–520, 2021.

[50] Intro to dids for people. https://medium.com/mattr-global/intro-to-dids-for-people-5ebe620b5a15.

[51] Mirko Zichichi, Stefano Ferretti, and Gabriele D'angelo. A framework based on distributed ledger technologies for data management and services in intelligent transportation systems. *IEEE Access*, 8:100384–100402, 2020.

[52] Foam whitepaper. https://foam.space/publicAssets/FOAM_Whitepaper.pdf.

[53] Friedhelm Victor and Sebastian Zickau. Geofences on the blockchain: Enabling decentralized location-based services. In *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 97–104. IEEE, 2018.

[54] Ethereum london upgrade. https://ethereum.org/en/developers/docs/gas/#pre-london.

[55] Mohammad Reza Nosouhi, Keshav Sood, Shui Yu, Marthie Grobler, and Jingwen Zhang. Pasport: A secure and private location proof generation and verification framework. *IEEE Transactions on Computational Social Systems*, 7(2):293–307, 2020.

[56] General data protection regulation: data protection law. https://gdpr.eu/.

[57] daduz11, hypercube on ipfs: hypfs repository. https://github.com/AnaNSi-research/hypfs.

[58] Reach language. https://reach.sh.

[59] Algorand pledges to be the greenest blockchain with a carbon-negative network now and in the future. https://www.algorand.com/resources/algorand-announcements/carbon_negative_announcement.

[60] Michele971, code repository. https://github.com/Michele971/hypfs-upgrade.

[61] Reach rpc. https://docs.reach.sh/rpc/.

[62] Polygon 16 users contract example. https://mumbai.polygonscan.com/address/0xa8319052b577490fee7a0ac92837c3d947288585.

Ringraziamenti

Con il concludersi di questi due anni universitari vorrei approfittarne per ringraziare alcune delle persone che sono state presenti.
Innanzitutto, ringrazio il Prof. Ferretti ed il Dott. Zichichi per avermi accompagnato durante questo progetto di Tesi e per aver dedicato parte del loro tempo ad aiutarmi sia dal punto di vista tecnico che emotivo.
Ringrazio i miei genitori, i quali hanno saputo supportarmi per tutta la durata dei miei studi, consigliandomi nel momento del bisogno. Grazie anche alle mie due sorelle Alice e Valentina, ai nonni, ai miei zii Davide e Angela, ed al resto della famiglia.
Uno degli amici incontrati durante il periodo universitario che merita un immenso grazie è Emanuele. In questi anni, abbiamo condiviso esperienze e realizzato progetti che credevamo più grandi di noi.
Un grande grazie va al mio amico Riccardo per il supporto a me dato in questi due anni.
Un doveroso grazie va Martino per essere sempre stato disponibile ad accompagnarmi in nuove sfide e per essersi confrontato con me, condividendo idee e soluzioni.
Infine, un immenso grazie va a Clara per esserci sempre stata e per avermi saputo ascoltare e consigliare.