Università degli Studi di Padova

Dipartimento di Ingegneria dell' Informazione

Tesi Magistrale in Ingegneria Informatica

A distributed CMS for small enterprises aggregation

Supervisor
Mauro Migliardi
Università di Padova

Master Candidate
Davide Martini

Anno Accademico 2018/2019

8 Luglio 2019

Dedicato alla mia famiglia,
per il vostro inestimabile affetto e supporto.

Abstract

As online platforms are increasingly part of everyday life, e-commerce and digital marketing campaigns are common practices for big companies that aim to build a solid brand awareness and engage new customers. Small and medium-sized enterprises are individually adopting these strategies, especially by publishing their products and services on popular social networks. On the other hand, groups with a common line of business could also benefit from the exposition as a single, organized structure. This scenario is relevant especially for communities of micro firms that belong to rural contexts and produce geographically branded goods. However, these small enterprises lack the resources and the expertise to implement and maintain a traditional centralized system. Moreover, competition makes entrepreneurs suspicious and less disposed to delegate content management to a potentially biased moderator.

Thus, the goal of this work is to design and implement a distributed content management system (CMS) for small enterprises aggregation and the online promotion of their products.

The proposed solution overcomes several issues combining two fundamental technologies. The first one regards Ethereum smart contracts and its blockchain to record interactions and the state of the system in a verifiable and permanent way. The second one is the so called InterPlanetary File System or IPFS, whose role is to allow the distribution of published contents in a peer-to-peer fashion. Both are described in deep in order to provide a comprehensive and detailed explanation of the proposed system.

Sommario

Le piattaforme online sono sempre più presenti nella nostra vita quotidiana, così il commercio in rete e le campagne di marketing digitale sono diventate pratiche comuni per le grandi aziende che mirano a costruire marchi riconosciuti e ad attrarre nuovi potenziali clienti. Analogamente, le piccole e medie imprese adottano le medesime strategie pubblicizzando i prodotti e servizi offerti nei più popolari social network. D'altro canto, è possibile affiancare ad una strategia di promozione individuale la presenza in piattaforme che coinvolgono aziende operanti nel medesimo settore. Questo scenario è particolarmente importante se si considerano comunità di micro imprese artigiane che creano prodotti di alta qualità e legati al territorio. Tuttavia, queste piccole realtà non possiedono le risorse nè le conoscenze per organizzarsi e sostenere i costi di un sistema centralizzato. Inoltre, la competizione rende gli imprenditori diffidenti nel collaborare e nel delegare la gestione dei propri contenuti ad un manager potenzialmente parziale.

Da ciò nasce l'obiettivo di questo lavoro che consiste nel progettare ed implementare un sistema distribuito di gestione dei contenuti (CMS) per l'aggregazione di piccole imprese e la promozione online dei loro prodotti.

La soluzione proposta risolve e supera diversi problemi combinando principalmente due tecnologie. La prima riguarda gli smart contracts di Ethereum e la relativa blockchain che permette di registrare le interazioni e lo stato del sistema in modo verificabile e permanente. La seconda è il cosiddetto InterPlanetary File System o IPFS, il cui ruolo è di permettere la distribuzione dei contenuti pubblicati attraverso un protocollo peer-to-peer. Entrambe queste tecnologie sono descritte approfonditamente così da fornire una presentazione dettagliata il sistema proposto.

Contents

Listing of figures

1 Introduction

Small and medium-sized enterprises (SMEs) are considered the backbone of the European economy and this fact is clearly stated in the annual report for the European Commission on SMEs [1]. Indeed, in 2017, there are more than 24.48 million SMEs which is the 99.8% of all enterprises in the EU-28 accounted for non-financial business sector. They also employ over 94, 76 million workers, which is the 66.4% of the total and they are responsible for the 56.8% of the value added which corresponds to 4, 160.7 trillion Euros per year. The SMEs category includes also the so called micro enterprises. They are characterized by a small amount of employees, in general lower than 10 units, and a total balance sheet lower than 2 million Euros. These micro firms are extremely common in the EU-28, since they account for the 93.1% of all companies and the 93.3% of all SMEs. In spite of their wide diffusion, they employ only 41, 98 million workers, which is 29.4% of the total for the non-financial business sector. These numbers are unexpectedly low in comparison to those provided by small and medium-sized enterprises which are accounted for 20.0% and 17.0% respectively of total employment.

While sectors such IT, software development, green technologies and fin-tech are well disposed to cooperation and internationalization, consolidated SMEs in the EU-28 are mainly concentrated in industries which export relatively little. Indeed, a big number of enterprises operate in wholesale and retail trade 26.3%, manufacturing 8.8%, business services 19.2%, constructions 14.2%, accommodation and food services 8.0%. This explains why, in 2017, slightly more than 55% of SMEs were active in industries which export less than 5% of their

turnover while only 7.3% of SMEs accounts foreign sales for 10% or more of the total.

This trend is particularly true for micro enterprises in the retail trade and manufacturing industry. An interesting case of study is Italy [2], where SMEs cover the 99.9% of all enterprises and 28.6% are micro firms. These ones provide 45.9% of overall employment and they are mainly focused in manufacturing sector (31% of the total). Micro enterprises in this industry provide 31.1% of all SMEs value added which is higher proportion than the EU-28 average. However, their target is mainly domestic clients and only 3.8% of them were selling their products or services beyond the EU in 2017. Given the fact that many micro firms are located in rural contexts or produce geographically branded goods, their business can benefit from e-commerce and coordinated initiatives of digital marketing which represent untapped opportunities for Italian and European SMEs.

These considerations prove the need of aggregation and cooperation in online presence for communities of micro, small and medium-sized enterprises that share a common line of business. However, SMEs (especially micro firms) may lack the economical resources and the expertise to build and maintain a traditional centralized web-site where group members could promote or sell their goods. Moreover, entrepreneurs want to decide independently their way of presenting, advertising and selling the products. This is an understandable behaviour, since competition mines their trust and sense of commitment to each other as a community. As consequence, they are less disposed to delegate their promotion policies to a potentially biased moderator of the central web-site. These barriers make SMEs prone to individual digital marketing initiatives, which are often limited to the publication of contents in popular social networks.

However, as described before, in terms of brand awareness and both domestic and international customers engagement, the online exposition as a single and organized structure is an additional business opportunity that SMEs should exploit. Thus, in this document I propose a distributed content management system (CMS) that allows small and medium-sized enterprises to advertise their products overcoming the previously discussed issues. The core of my work is not only a design study, but also a precise implementation that highlights strengths and weaknesses of the adopted strategy. As it will be discussed in the following chapters, my solution makes each member of the community responsible for the content presented, which is authenticated and can be modified only by the rightful owner. In addition, in order to reduce entrepreneurs' suspects, the management policies of both members and contents should be unbiased. In my solution, this is achieved through a majority vote that is triggered by a participant that wants to perform moderation operations like expelling

a member or removing a content from the platform. Also the registration of a new enterprise in the community has to be approved with a vote from the verified users.

After the definition of the problem and provided a justification for its importance, my workflow followed the organization of this document:

- Chapter two gives an overview on the state of the art and describes eventual solutions to this problem in the literature. Then, I describe my proposal that heavily relies on two main technologies: Ethereum [3] and the InterPlanetary File System [4].
- Chapter three describes Ethereum in deep. Topics like blockchain state, accounts, smart contracts, transactions, blocks, consensus protocol and the Solidity programming language are explained.
- Chapter four regards the InterPlanetary File System, which is commonly referred as IPFS. In particular, it is presented as a content-addressable, peer-to-peer protocol for storing and sharing data that combines consolidated technologies like BitTorrent and Git.
- Chapter five dives into the implementation of the proposed architecture. Particular attention is given to the pros, cons and trade-offs that I faced during the software development.
- Chapter six concludes the document describing the results achieved and future works.

2

CMS architecture design

The first step of my work was to search eventual strategies in the literature that deal with or can be applied to the problem of online aggregation of SMEs. Thus, the first section of this chapter reports my study of the state of the art and provides some insights in interesting solutions adopted. Once my research was completed and I had a clear understanding of the current technology, I designed my architecture. The resulting distributed CMS is described in the second section of this chapter. Here, the goal is to provide an overview of the system in order to show novelties and differences in comparison to other solutions proposed in the literature. As explained in the introduction of this document, the system is explained in its details in chapter five.

2.1 The state of the art

In the literature, there is no research whose goal is explicitly to design a distributed content management system for SMEs aggregation. However, my scenario can be split in two sub-problem that have been studied in the literature. The first one is to establish trust on the state of the common platform where the community of SMEs publish their products. The second one regards the moderation of contents and group members, but its implementation heavily depends on the solution adopted to address the previous issue. As explained in the introduction, these are crucial requirements in my scenario where each entrepreneur in the community shows a skeptical behaviour towards other members and a potentially biased

third party moderator.

In the literature, the first problem refers to the topic of consensus in distributed systems, which has been studied since 1970s alongside the rise of distributed databases and transactions. Once Bitcoin [5] was born in 2008, the blockchain technology gained interest in the scientific community. It has been employed in a wide range of distributed architectures where a trusted record or a consensus mechanism has to be implemented. Its application can be found also in the most recent and advanced distributed marketplaces and content delivery networks. These topics are particularly interesting due to their similarities to the problem of online SMEs aggregation.

Thus, in order to provide a brief description of the papers and systems that inspired the design of my architecture, it is important to present the principles of blockchain technology. Its study was an important aspect of my work also because the proposed CMS heavily relies on it.

2.1.1 Principles of blockchain technology

Currently, there is no standard regarding blockchain technology and each implementation has its peculiar features. However, they all combine peer-to-peer networking, asymmetric cryptography, cryptographic hashing and a consensus protocol. Thus, general concepts are presented in order to understand blockchain's role in more complex scenarios. A detailed description will be dedicated only to the Ethereum implementation in chapter three, since it is relevant for the designed CMS.

A permissionless blockchain can be defined as a public, decentralized, replicated and potentially immutable ledger. It records all transactions denominated in the system's cryptocurrency and its correctness can be verified by anyone. A blockchain is typically implemented as a linked list of blocks which are composed of an header and a set of transactions. In order to realize the chain, each header contains a pointer which is the cryptographic hash of the previous block and other information that depend on the implementation. This pattern is important since it allows the verification of the chain integrity. Indeed, if someone modifies the content of a block, its new hash is different from the one in its successor and so on. On the other hand, transactions specify some transformation on the state of the blockchain and they are usually organized in structures like the Merkle hash tree [6] in order to allow their verification in an efficient and secure way. In order to provide a graphical representation, a simplified image of the Bitcoin blockchain is shown in Fig. 2.1. When a new transaction is performed, it is broadcast in the network and it remains pending until a node adds it to a new

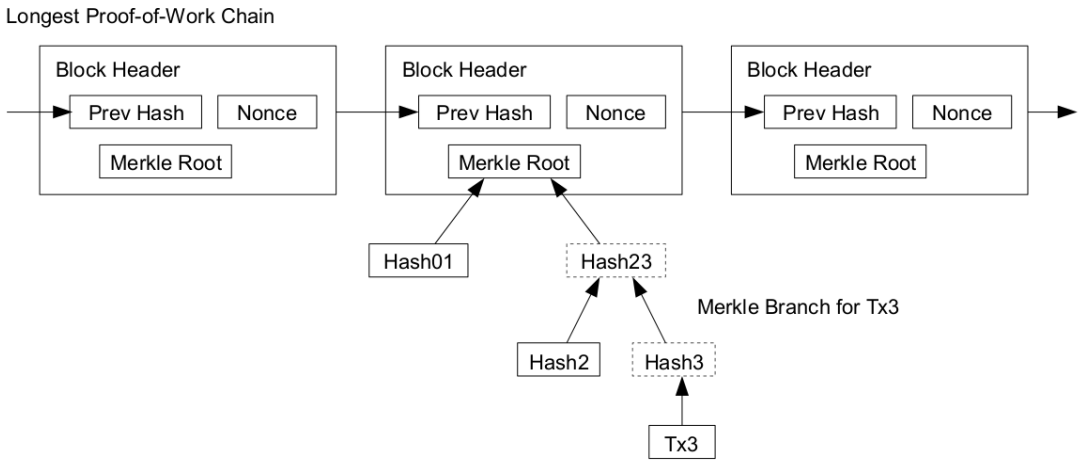

Figure 2.1: Bitcoin blockchain model.

block. The insertion of a block has to be agreed by all the peers of the network that have a copy of the blockchain. This is performed through a consensus protocol and there are many of them, as it is well explained in [7].

For instance, Bitcoin uses the famous proof-of-work (PoW) which requires a node (also called miner) to collect pending transactions and solve a mathematical puzzle that is computationally expensive. It consists in finding a *nonce* to insert in the block header, so that the resulting block hash presents d consecutive zero bits. Using, for instance SHA-2-256, the required work is exponential in d but its correctness can be verified by executing a single hash. In a sense, d represents the difficulty of the puzzle and it is calibrated so that a new block is found, on average, once every ten minutes. Once a miner finds a proof-of-work, it broadcasts the new block to all nodes in the network. Then, they check its validity and express their acceptance of it by mining the next one in the chain which contains the hash of the accepted block as the previous hash. Nodes earn an amount of coins for each block mined as an incentive to be honest and as a reward for the CPU time spent in resolving the puzzle. In order to receive the prize, a miner inserts as first transaction a special one that starts a new coin owned by the creator of the block. This also provides a way to initially distribute coins into circulation, since there is no central authority to issue them.

A crucial aspect in blockchain is represented by forks. They may occur if two miners find two different blocks built on the same previous one. Each consensus protocol has to resolve this situation and, for instance, Bitcoin implements the acceptance of the longest chain as the correct one. In this case, an attacker that wants to modify a transaction in a block must

have the majority of the computing power on the network in order to create a fork of the blockchain that is longer than the official one.

Finally, it is important to mention that Bitcoin transactions can include reference to deterministic scripts that operate on the transaction inputs and produce some outputs. This shows the intention to make the blockchain as the basis of general purpose platforms and not only related to payment systems. This happened especially with Ethereum, where scripts are replaced with smart contracts, which can be considered as self-executing code that enforce a digital contract.

2.1.2 Blockchain in complex systems

As stated before, the blockchain is the standard technology applied in modern systems and literature proposals to solve the problem of trust on the status of a distributed system. However, it is usually one of many components employed in complex scenarios. Now, I briefly describe some works that I studied in order to design my distributed content management system for SMEs aggregation.

The first research that inspired my system is described in [8]. This paper presents a solution for moderating a generic peer-to-peer content delivery network. In particular, it provides a proof-of-concept implementation that extends Twister [9] architecture. Twister is a microblogging platform that relies on Bitcoin to authenticate users, which are stored on the blockchain as usernames and public key pairs. On the other hand, contents posted by authenticated members are distributed through a BitTorrent network. The paper proposes an authoritative control over the system using cryptographic accumulators that contain the network administrators. The accumulator containing the founders is stored on the blockchain since its genesis block. In this way, once a user receives a command, it is able to decide whether to accept or to ignore it by inspecting the blockchain. For instance, if a delete request comes from the author (verified through its public key) or it is sent by a member registered in the accumulator, the operation is allowed and thus performed by the user. The paper also describes a way to add new administrators introducing a new type of transaction that can be performed by group members. In particular, this procedure is implemented through the scripting system of Bitcoin.

More recent studies face these problems in different scenarios by applying the so called distributed web technologies. An interesting research that caught my attention is [10]. In this paper, the authors propose a decentralized marketplace, which is obviously a similar topic to the case in study. The goal of their work is to overcome the limitations that a centralized plat-

form may impose on sellers. In particular, they focus on the fees paid both when listing and selling products and the lack of privacy of users' data. This research describes an architecture which is based on Ethereum, MongoDB and the InterPlanetary File System. In particular, the role of the Ethereum blockchain is to store authentication information, while IPFS is responsible to keep and distribute item data in a decentralized fashion. In this system, MongoDB is used to optimize the blockchain query process but this is a potential central point of failure. This issue is also pointed out by the authors that suggest to replace MongoDB with a decentralized service like BigChainDB [11].

This combination of Ethereum and IPFS is also presented in [12]. Here the goal is to build a decentralized social network addressing the problems of security and privacy that emerged from the recent leakage scandal from Facebook to Cambridge Analytica [13]. In the proposed architecture, the Ethereum blockchain and IPFS hold the same role as in the previous system. The business logic of the platform (e.g. the follow-unfollow mechanism) is addressed via smart contracts while IPFS performs the actual storage and delivery of the posts.

The study of these systems proposed in the literature lead me to the following conclusions. Considering a distributed scenario where untrusted parties publish content on a common platform, the suggested architecture relies on two components:

- A blockchain to provide a trusted system state and a secure business logic that allows user authentication.
- A distributed system to deal with the actual content distribution and availability.

Thus, this structure is consolidated, even though the involved technologies change with the years. On the other hand, while the problem to authenticate the owner of a content has been addressed, none designed a system that allows a truly unbiased moderation. Indeed, also in [8] the control is performed by the group of administrators in an authoritative way. Therefore, my architecture is similar to the proposed ones, but the novelty is the presence of a trusted and democratic moderation mechanism. Its actual implementation is described in detail in chapter five, while the next section provides an overview of the system.

2.2 The designed architecture

In order to provide an organized presentation, this section is divided in two parts. The first one describes the process that I followed to choose the technologies that best fit in my ap-

plication. The second one provides an overview of the system and how these components interact.

2.2.1 Technologic components

Considering my conclusions on the analysis of the state of the art, I started my design by selecting the two elements that best suit the problem needs. At first, the blockchain technology has to satisfy these requirements:

- It must be public and permissionless, since SMEs may not have the expertise to run a full blockchain node and commit new blocks. Moreover, the group is dynamic because SMEs can freely join and leave the application.
- It must be programmable in a flexible and easy-to-use way. This is a crucial aspect due to the complexity of the application's business logic.
- It must be able to sustain the potential growth of the community. Scalability is not a major issue since the platform aims to aggregate SMEs that share a line of business and a common geographic location. However, transaction speed is to keep in consideration in order to provide a responsive user experience.
- It must provide side libraries that allow interoperability, especially in order to access the blockchain from the browser. This is important because the presence of a client software to be installed is a big downside that limits the popularity of distributed applications. This clearly explains the requirement since the platform is meant to advertise SMEs' products and engage both domestic and international clients.
- The technology is still considered to be in its infancy, so the presence of documentation and a community of developers is appreciated.

Currently there are many blockchain platforms, but I chose Ethereum since it respects all the crucial requirements. In particular, it provides a virtual machine that runs smart contracts written in Solidity. Moreover, the block time in Ethereum is more or less fifteen seconds, which is considerably lower than the ten minutes of Bitcoin. However, the most important fact is that in order to talk to a node from inside a JavaScript program, the Ethereum Foundation provides a library called Web3.js [14] that exposes an interface for JSON-RPC methods.

The second choice I had to make regards the distributed system for content storage and delivery. Here the requirements are fewer but important as well:

- It must implement an addressing mechanism that provides a content identifier suitable to be stored on Ethereum. This is crucial since the trusted state of the common platform is guaranteed by the blockchain.
- It must be accessible via JavaScript from the user browser. The installation of a client software could be tolerated by SMEs that post on the platform, while this is impractical for casual visitors. As stated for the blockchain, this clashes with the goal of the application.

Here, I had to choose from two possibilities that satisfy these issues. Swarm [15] and the InterPlanetary File System are very similar in the goal since they claim to offer an efficient decentralized storage layer. In particular, they both address data on the network through the content hash rather than its location and this is extremely important for one big reason: consistency. Indeed, if the file location is stored in the blockchain and the related IPFS node leaves the network or removes it, the system can no longer see the content even though another peer may have a copy of it. A worse situation may happen if the node with the desired data is malicious and swaps the original file with another one. Thus, the usage of content hash as identifiers to keep in the blockchain is critical since it prevents these scenarios. As regards the differences between the two technologies, they can be seen in the development status and in implementation technicalities. As a developer, I decided to rely on IPFS because it has an official JavaScript implementation, a wider adoption and a dedicated community, even though Swarm has a stronger relationship with Ethereum peer-to-peer library.

One of the biggest problems that all decentralized storage systems have to face is data persistence. It is not reasonable that a node stores and delivers other users' content without an incentive. This is a known issue and solutions like Storj [16] and Sia [17] rely on the blockchain technology. Also the founders of the IPFS started an interesting project called FileCoin [18] which is based on the same principle. In FileCoin, miners earn the native protocol token by cryptographically proving continuous data storage, while clients pay them to store or distribute files and to retrieve data. Thus, IPFS and Filecoin are complementary protocols: the first one allows peers to store, request, and transfer verifiable content with each other, while the second provides the missing incentive structure. As stated before, in my design I use only IPFS because FileCoin presents two aspects that compromise its applicability. The first and the most important one is the current impossibility to perform cross-chain operation between Ethereum and FileCoin. Apart from the development difficulties, this may lead to inconsistencies and security issues on the platform. The second one regards the fact that FileCoin requires users to pay a fee in order to retrieve a content from a

miner. FileCoin developers justify this choice saying that the fee is meant to repay the miner for the bandwidth used during the data transfer. This obviously makes FileCoin unsuitable for my advertisement application because new customers are casual visitors of the platform that must be able to freely see the posted products.

The problem of data persistence is also taken in consideration by Swarm. As stated before, Swarm is still in a proof-of-concept stage with a Go implementation. However, the team working on Swarm have announced that they are planning on adding an incentive storage layer based on Ethereum. For sure, this will make Swarm the preferred persistent data storage technology for distributed applications that rely on Ethereum due to their deep connection and interoperability. Therefore, a future work to improve my design could be the switch from IPFS to Swarm once the technology is ready for production. By now, the issue of data persistence remains unresolved in the current architecture even though I mitigated this problem in the implementation of the system. Since this aspect is not relevant to describe the components of the architecture and their interaction, further information are given in chapter five.

2.2.2 Architecture overview

In conclusion, the overall designed system is shown in Figure 2.2. The arrows represent the main interactions between the technological components that I chose to implement the architecture. The enumeration in the picture refers to the process of loading the single-page application and show the home content. I considered this action because it is the most complex one that involves all the technologies previously presented. Obviously, if we consider other operations, the order may change or some interactions could not be performed. Following the enumeration in the image, I provide a brief description of their meaning.

1. In order to access the advertisement platform, a user can simply use its own favourite browser. As stated before, this is extremely important because a mandatory dedicated software installation is a big downside for the popularity of the application. However, it is possible for an user to be part of the Ethereum and/or the IPFS networks by running a local node. In particular, having a 24h online IPFS node could interest SMEs that want to be sure about the availability of their posted products.

2. Once the user has opened the browser, it has to retrieve the front-end assets of the application from IPFS. The URL that the user has to enter is typically formatted like *https://<gateway_address>/ipfs/<content_hash>*, where *<content_hash>* is the hash that identifies the front-end assets on IPFS. If the user is running a local peer, then

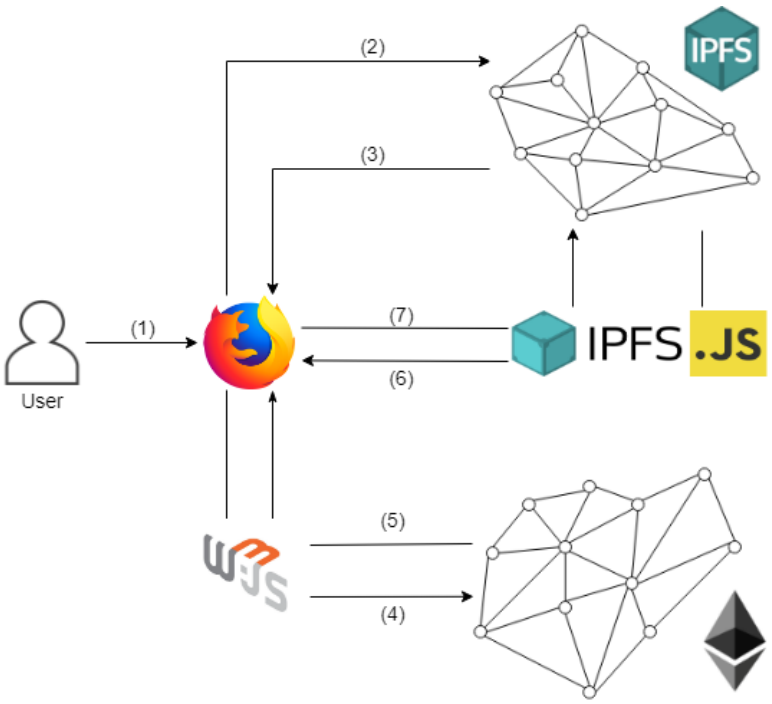

Figure 2.2: Model of the designed advertisement platform for SMEs' aggregation.

it can request the files from the daemon set on *localhost:8080*. Otherwise, it can access the network through the so called public gateways and retrieve them just like in a traditional client-server architecture. IPFS gateways are third-party nodes that fetch content from the IPFS network and serve it to the user over HTTPS. Currently, there are 39 gateways available and the user can select the preferred and online one [19]. During the development of my work, I used the one provided by CloudFlare since it appeared to be the most reliable. However, in order to upload the front-end assets of the application I had to run a local node. The persistence of these files on IPFS is crucial, otherwise the advertisement platform can not be accessed. Therefore, since IPFS does not guarantee data persistence, this responsibility should be in charge of the one that committed the system. Hosting an IPFS node instead of a traditional web server is meant to make the overall architecture suitable for Swarm.

3. The user receives HTML, CSS, JavaScript and JSON files from IPFS through the gateway or the local node and the browser renders the application. In particular, it gets the compiled smart contract which exposes its application binary interface (ABI) in JSON format and other JavaScript libraries required in the system. Points two and three are performed only once in order to access the platform which is designed as single-page application.

4. In order to perform business logic operations and inspect the status of the system, the user has to trigger a function from the application's smart contract through a call or a transaction to the Ethereum blockchain. More information are given in chapter five, but, as an example, this procedure happens when retrieving the IPFS hash of products published by the community of SMEs or when posting a content on the platform. The interaction with Ethereum in the browser is possible thanks to the Web3.js library. However, the user has to be part of Ethereum peer-to-peer network since Web3.js only exposes an interface for JSON-RPC methods. Unfortunately, there is no work-around and a piece of software has to be installed. In order to limit this downside and ease the access to the platform, it is enough to download MetaMask [20] instead of an Ethereum full node. MetaMask is a popular browser extension that allows to easily manage an Ethereum wallet and automatically injects Web3.js in the window object. Under the hood, it relies on Infura [21] with a zero-client approach and uses its APIs to connect to different Ethereum networks. Thus, this add-on is mandatory for common browsers and I used it for FireFox and Chrome in order to test the system. Currently, there are some projects that aim at building browsers integrated with Web3.js like Brave [22]. Also the Ethereum Foundation proposed its own browser called Mist [23], but the project has been recently deprecated.

5. If the function triggered on the smart contract has a return value, it is usually exposed by Web3.js as a JavaScript array or object, depending on type conversion. Then, it is possible to perform further computations using these data as parameters. In the case in focus, the returned value could be the array of strings representing the IPFS hash of valid contents published on the platform. The actual implementation is a bit different and is described in chapter five.

6. Once the IPFS hash of all contents are available, it is possible to download them from the peer-to-peer network. However, instead of retrieving the file from a public gateway as explained in the first step, the system relies on the JavaScript implementation of IPFS. This library is received by the user once the point two is performed. The motivation behind this choice is that the upload operation is not granted by IPFS public gateways. Once the content is received it is possible to show it to the user through the presentation logic of the application.

7. The upload operation of a file is the reason behind the usage of IPFS.js [24] library. In particular, once the front-end assets are collected, the platform is implemented so that each user is actually running an IPFS node in its browser. This allows to upload contents form a local node, but this fact is also used to deal with the problem of data persistence on IPFS. As stated before, a product is available in if there is at least a peer that provides it. Thus, a possible way to mitigate the problem is to have file replicas on the network. This is exactly the motivation behind the presence of this step in the

operation of showing the home content. Indeed, each product downloaded for presentation purposes is then added and pinned by the local node that makes the content available to the network. In order to better explain this mechanism, it is described in chapter five. Obviously, this process introduces inefficiencies, especially in terms of user experience, but these problems are limited with an accurate implementation and a reasoned presentation logic. As stated before, this is a work around to the problem of application's content persistence, which remains unresolved in general. However, there are two positive side effects. The first one regards the fact that each member in the SMEs' community is the first responsible for the availability of its own content, which can easily be granted through IPFS.js by staying on the application's web page. Thus, members' selfish behaviour and lack of commitment to each other have a positive impact on content availability due to this replicas procedure. In addition, also casual visitors of the platform are contributing to the system. In this scenario, replicas are a valid backup strategy to limit the problem of data persistence on IPFS. The second one regards the fact that once a product is published, its IPFS hash is permanently and securely stored on Ethereum blockchain. Thus, even though there is no peer in the network that provides that content, it is enough for the owner to upload again the same file in IPFS with no costs and the system is consistent.

3

Ethereum and Solidity

This chapter is a comprehensive description of Ethereum. A deep study of this blockchain tecnology was an important component of my work since Ethereum is fundamental in the designed content management system. The exposition is divided in two sections. The first one regards accounts, transactions, blocks and their validation in Ethereum. The second one is dedicated to Solidity, the main programming language for smart contracts.

3.1 Ethereum fundamentals

Vitalik Buterin was impressed by the blockchain technology introduced by Bitcoin and its scripting mechanism. His idea was to extend its applicability beyond the payment system, so in late 2013 he announced Ethereum [25] [26]. His intention was to provide a blockchain with a built-in, fully fledged, Turing-complete programming language that could be used to create smart contracts. These programs are the core of Buterin's idea because they allow developers to encode arbitrary state transition functions and implement a secure business logic for any distributed application in few lines of code.

The concept of smart contracts is not introduced by Ethereum, but it was coined in 1990s by Szabo that defined it as "a set of promises, specified in digital form, including protocols within which the parties perform on these promises" [27]. Szabo also expected that smart contracts could be extremely functional, but his idea did not see the light till the emergence of blockchain technology.

As explained in the previous chapter, a permissionless blockchain is a decentralized, verifiable and potentially immutable ledger that records transformations of the state. In Bitcoin payment context, the state is a representation of the ownership of all coins mined and not yet spent (UTXOs). Thus, the concept of transaction expresses the evolution of the system from a state to another. Therefore, in order to understand the structure of Ethereum it is important to define its state and how transactions are composed, performed and validated.

3.1.1 Ethereum state and accounts

The (world) state in Ethereum is a mapping that defines unique pairs *(address, account)*. Addresses are 160-bit identifiers, while accounts have a common structure even though they can be distinguished in two types. There are the so called externally owned accounts (EOAs) controlled via public-private key, which are usually associated to humans or machines. Then, there are the contract accounts, also known as smart contracts, which are controlled by their code. Both are composed by four fields:

- The *nonce* is a value that represents the number of transactions sent from this address or, in the case of contract accounts, the number of creations performed.
- The *balance* of the account in Wei, which is the smallest denomination of Ether, the cryptocurrency provided by Ethereum. The conversion rate is $1 \text{ Ether} = 10^{18}$ Wei.
- The *storage* of the account, which is organized as a modified Merkle Patricia tree ([26] Appendix D) to encode contents of the account. This persistent data structure is also called Trie and it provides a mapping between arbitrary-length byte arrays. The core of the Trie is to expose a single value that identifies a given set of key-value pairs.
- The hash of the *code* if the account is of contract type, otherwise it contains the Keccak-256 hash of the empty string. This is the code that gets executed once this address receive a message call. Obviously, it can read from and/or write to the account's *storage*. Unlike the other fields, it is immutable and it can not be modified after account construction.

As stated in the previous chapter, each node in the Ethereum network stores (at least) a copy of the data in the current (world) state. This is done with an off-chain database that maintains a mapping of byte arrays to byte arrays which can be accessed by users through an external application, most likely an Ethereum client. On the other hand, the blockchain maintains only the Trie's root in order to minimize the on-chain storage needs, thus its overall size. As consequence, it is possible to retrieve information about the current Ethereum

(world) state by querying this database which is charge of its persistence. This procedure is usually referred as a call in contrast to a transaction.

3.1.2 Transactions, Gas and Blocks

A transaction is a single cryptographically-signed data package that allows the evolution of Ethereum (world) state. There are two types of transactions with two different purposes: contract creation and message call transactions. The first ones can be committed by an EOA that deploys a new smart contract in Ethereum state. The second ones can be created either by an external entity or a contract and they target a contract account. Message call transactions are performed in order to transfer Ether to the recipient and/or trigger the execution of a function in the target contract that modifies the (world) state. Both transaction types provide a common structure and two peculiar fields:

- A *nonce* that represents the number of transactions sent by the sender.
- A scalar value called *gasPrice* equal to the amount of Wei to be paid per unit of Gas for all computation incurred as a result of the execution of this transaction.
- A number *gasLimit* equal to the maximum amount of Gas that should be used in executing this transaction. This is paid a-priori, before any computation is done and may not be increased later.
- The target *to* of the transaction. This field contains the 160-bit address of the recipient in case of a message call transaction. Otherwise, the value is set to 0 in case of a contract creation.
- The *value* which is the amount of Wei to be transferred to the message call's recipient. In the case of a contract creation transaction, this number represents the initial balance of the newly created account.
- Values corresponding to the SECP-256k1 signature of the transaction and used to determine the sender ([26] Appendix F). They are called as v, r and s.
- A byte array of unlimited size that specifies the input data in the message call transactions. This field is referred as *data* and it is used to specify the contract function to execute and eventually to pass input parameters.
- A byte array of unlimited size that specifies the code for the procedure of contract account initialization. This *init* field is peculiar of contract creation transaction.

In this listing that describes the structure of transactions, I introduced a fundamental concept in Ethereum which is Gas. Any programmable computation has its own universally agreed fee, which is expressed in Gas and not in the internal cryptocurrency. In a sense, the *gasLimit* field implicitly states the maximum number of computational steps that the execution of this transaction is allowed to take. On the other hand, the *gasPrice* states the amount of Ether required to purchase a Gas unit. It is important to remark that the concept of Gas is relegated to the scope of a transaction, while Ether represents the actual coin in the system. Therefore, in order to send a transaction, an account has to pay in advance the miner using its balance, because it is the one that performs the actual computation and the transition from a (world) state to another one. Here it is important to remark that if a contract function only performs at most read operations on the current (world) state, it does not use any Gas. This happens because miners are not involved since there is no evolution in the (world) state and thus no transaction. In general, miners are allowed to select the transactions they want to include in the next block. As consequence, transactions with a higher *gasPrice* will more likely be chosen because the miner will receive a higher amount of Ether. Thus, the sender has a trade-off to make between paying less Ether and the chance that its transaction will be mined in a shorter period of time.

All the presented data structures are organized into blocks which are the core elements of the blockchain. As stated in the previous chapter each block has a header and a body. The latter keeps information about the comprised transactions, but also a set of headers whose blocks are sibling of the current block's parent (such blocks are known as *ommers*). The presence of this field can be understood once Ethereum's consensus protocol is described. As regards the header, an Ethereum block has many fields some of which refer to structures already mentioned:

- The Keccak 256-bit hash of the entire parent block's header. This piece of information is called *parentHash*.
- The Keccak 256-bit hash of the ommers list in the body of the current block. This piece of information is called *ommersHash*.
- The 160-bit address of the EOA that mined this block. It represents the *beneficiary* to which all fees are transferred.
- The Keccak 256-bit hash of the new (world) state Trie's root. This piece of information is referred as *stateRoot*.

- The Keccak 256-bit hash of the Trie's root populated with transactions listed in the body of the current block. This field is referred as *transactionsRoot*.
- The Keccak 256-bit hash of the Trie's root that contains the receipts of each transaction listed in the body of the block. This field is called *receiptsRoot*.
- The Bloom filter which is a probabilistic data structure that states if an element is part of a set without false positives. It is used to efficiently address log entries created from the receipt of each transaction listed in the body. This field in the header is *logsBloom*.
- A number that represents the *difficulty* of this block. It can be calculated from the previous block's value and the *timestamp*.
- A scalar value equal to the *number* of previous blocks. The enumeration starts from zero which is assigned to the genesis block.
- The current *gasLimit* expenditure per block.
- The total amount of *gasUsed* in all transactions included by this block.
- A *timestamp* expressed as Unix's time format set at this block's inception.
- A byte array containing data relevant to this block. This must be 32 bytes or fewer and it is referred as the *extraData* field.
- A 256-bit hash called *mixHash*. Combined with the *nonce* of the block, it proves that a sufficient amount of computation has been performed in order to mine this block.
- A 64-bit hash which is referred as the *nonce* of the block. As stated in the previous chapter, it can be considered as the proof-of-work in such a consensus protocol.

The overall block structure is shown in Figure 3.1.

3.1.3 Transition function and the EVM

The execution of a transaction implies the evolution of the system from a state to another one. In particular, the progress is achieved with the addition of a new block. This procedure is defined in the Ethereum protocol with a transition function that, given the current (world) state and a transaction as inputs, it returns the new state which is broadcast to Ethereum peers. A nice example is provided in [25], where it is easy to see some validity tests and simple operations performed by this function.

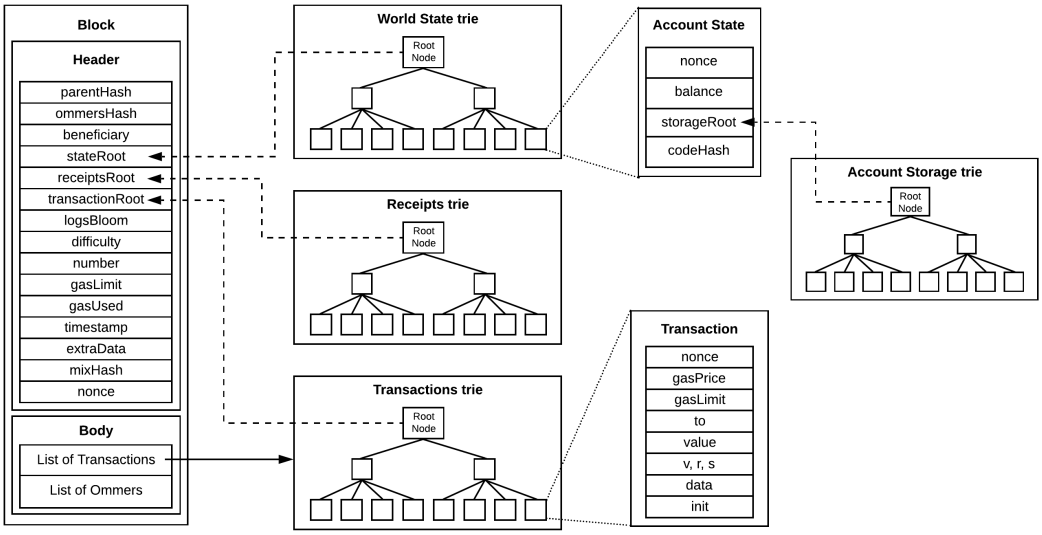

Figure 3.1: Ethereum block model.

- It checks if the transaction is well-formed. In particular, it verifies the signature using *v*, *r* and *s* and if the *nonce* matches the one in the sender's account.

- It checks if the sender's balance has an amount of Ether which is enough to purchase the required Gas. In this case, the transaction cost is subtracted from the sender's account balance and its *nonce* is increased. Otherwise, an error is thrown and the transition is stopped.

- It sets the Gas of the current transaction to *gasLimit* and subtract an amount that depends on the bytes of the transaction.

- It checks if there is an amount of Ether to be transferred to the recipient by inspecting the *value* field. Then, in case the transaction has data in *init*, a new smart contract is created with a balance equal to *value*. Otherwise, the transaction is of type message call and the specified amount of Ether is transferred to the recipient. Moreover, if the target is a contract account, the related code specified in *data* is executed using the provided Gas.

- It reverts state changes in case of a failure in Ether transfer (e.g. the sender does not have such an amount in its balance) or in code execution (e.g. the *gasLimit* is lower than the value required for the computation which results in an out of Gas exception). In particular, the payment of Gas fees are not reverted and they are added to the miner's balance anyway.

- It refunds the fees to the sender depending on the remaining Gas and transfers the Ether paid for Gas consumed to the miner.

However, this description of the transition function does not answer to two important questions. The first one regards where the code is actually run since Ethereum has a distributed architecture. The second one regards the actual code execution once a contract address receives a message call. The former is easily answered: since each transaction is added into a block, the called contract code will be executed by the miner and then by all nodes that download and validate the blockchain. This is important in order to keep consistency in the copies of the (world) state stored by each peer. On the other hand, in order to describe how it is run by nodes, I have to introduce the Ethereum virtual machine (EVM).

The official programming language used to develop smart contracts is Solidity, but the code is compiled into bytecode or EVM code for the actual execution. At a low level, it defines operations that allow to access three types of memory spaces:

- The *stack* with 32-byte elements.
- The *memory*, an infinitely expandable byte array.
- The contract's *storage* previously described when introducing Ethereum accounts.

It is important to remark that unlike the *storage*, data in the *stack* or in the *memory* is not persistent since their scope is respectively a transaction and a contract function. The knowledge of these concepts is fundamental for developers that design smart contracts for Ethereum. As regards the EVM, the code execution is an iterative procedure that starts by fetching the current instruction corresponding to the EVM's program counter. Each instruction has its own definition in terms of how it affects the computational state of the EVM. In particular, Ethereum specifies their Gas consumption and the way each operation interacts with the *stack*, the *memory* and the *storage*. Once the selected operation is performed, the program counter is incremented and this cyclic routine continues until the end of the code is reached or an error (e.g. the out of Gas exception) occurs.

3.1.4 Ethereum consensus protocol

Ethereum implemented a proof-of-work (PoW) consensus protocol which has been briefly described in the previous chapter. Since the mining process is responsible to distribute new coins in the system, the execution of the PoW function should be as accessible as possible and

not relegated to people with specialized hardware like ASICs. For instance, this is a problem for Bitcoin because the only task that ASICs have to solve is SHA-2-256 which is clearly CPU-intensive. Thus, an accessible ASIC-resistant mining procedure is found by designing a sequential memory-hard PoW function such that it is not possible to find multiple *nonces* at the same time by running the procedure in parallel.

The resulting algorithm proposed since the first version of Ethereum is called Ethash. The proof-of-work function takes as input the new block's header without *nonce* and *mixHash*, the candidate *nonce* for the block and a large dataset which is required to compute *mixHash*. The output is an array with the *mixHash* of the new block as first item and as second element a pseudo-random number n which is cryptographically dependent on the hash function used and the dataset. The input *nonce* is valid if n is less or equal to 2^{256} divided by the new block's *difficulty* value. Further details on Ethash are provided in [26] Appendix J.

As regards the double-spending problem, Ethereum provides an alternative solution to Bitcoin's which consists in automatically choose the longest chain in the network. Indeed, Ethereum developed a strategy called Greedy Heaviest Object subTree or GHOST protocol. It requires that miners focus on whichever chain the most other miners are working on. Due to differences in data propagation over the network, this procedure tends to create more uncles. This explains the need to store the block's *ommers*. Nevertheless, the blockchain itself is equally secure and this method allows for higher throughput of transactions than Satoshi's solution.

In conclusion, it is important to say that the Ethereum Foundation is extremely active and innovative in terms of consensus protocols. For instance, Ethash has changed many times and it is said to be gradually replaced once Ethereum 2.0 is released. Indeed, important projects like Casper FFG [28], Casper CBC [29] and Plasma [30] are showing an increasing interest towards the proof-of-stake (PoS) consensus protocol. The assumption behind PoS is that who holds a stake (i.e. deposit, currency) in a network is prone to act correctly because it is in its interests. In other words, the more stake a node has, the higher should be its will in preserving the system. Following this principle, in PoS-based blockchains, a set of nodes called validators take turns proposing and voting on the next block. The weight of each vote depends on the size of the validator's stake. Anyone who holds Ether can request to be promoted as validator by sending a special transaction that locks up their coins into a deposit. The process of creating and agreeing to new blocks is then done through a consensus algorithm. Thus, there are many different implementations, but they all can be split in two categories:

- chain-based PoS selects a validator in each time interval, and assigns that node the right to create a single block. The new block must point to the one at the end of the previously longest chain.
- Bizantine Fault Tolerant (BFT) proof of stake chooses randomly the validators with the right to propose blocks, but consensus is established through a multi-round process. In this procedure, every validator votes for some specific block during each round and once this process ends, all (honest and online) validators permanently agree on whether or not include a block in the chain. The core difference is that the consensus on a block does not depend on the length or size of the chain after it.

The motivation behind this interest in proof-of-stake is that, in comparison to PoW, it provides significant advantages.

- There is no need to consume large quantities of electricity in order to run the consensus protocol and secure a blockchain, while energy efficiency has become an issue for PoW systems.
- Due to energy savings, there is not as much need to issue as many new coins in order to motivate participants and repay the electricity involved in the mining process.
- Proof-of-stake allows to design many mechanisms to discourage the creation of centralized cartels or prevent them from acting in a harmful way to the network.
- It reduces centralization risks, as economies of scale are much less of an issue. In PoS the amount of coins of a validator directly represents its vote weight, on the other hand having mass-production equipment is an advantage for proof-of-work.
- It has the ability to make various forms of 51% attacks vastly more expensive to carry out than proof-of-work.

An interesting protocol similar to PoS is proof-of-authority (PoA) where instead of selecting validators through stake, they are chosen by identity. In this context, identity means the correspondence between a validator's personal identification on the platform with official documentation for the same person. The idea of "staking identity" means voluntarily disclosing who you are in exchange for the right to validate the blocks. Thus, since users' identity and actions are public, they are prone to behave correctly and preserve the network in order to maintain their good reputation. One problem that PoA tries to overcome is the fact that stake in PoS is not a good measure for a node's commitment to the network. For instance, PoS may consider as equal validators both a newbie and an early adopter user with

the same stake. The proof-of-authority protocol uses identity as a equalized stake which is understood and valued the same by all actors. Proof-of-authority is important for Ethereum since there are services like Parity [31] as far as Rinkeby and Kovan test networks that are based on PoA. As it will be described in chapter five, the proposed CMS is deployed on Ropsten test network which is based on proof-of-work protocol, just like Ethereum's main network.

3.2 Solidity

The goal of this section is to provide an overview of the official programming language used to build smart contracts on Ethereum. This is not a Solidity documentation, rather a presentation of its most important constructs and their correspondence to what I described in the previous section about Ethereum. Everything reported in this section regards Solidity version 0.5.10.

3.2.1 Contracts

Solidity is a statically-typed programming language that allows to define smart contracts. Since they manage money and, as stated in the previous section, their *code* field is immutable, a contract with bugs may lead to serious problems. The DAO [32] is a perfect example, because this crowd-funding platform has been attacked in 2016 and around 60 million US Dollars have been stolen. This attack has been nullified by performing a hard fork on Ethereum main network, but since that accident, many researchers and developers listed some good practices in designing and coding smart contracts [33].

From a developer point of view, a *contract* is similar to the concept of class in the object-oriented programming paradigm. Indeed, they may contain state variables, functions, function modifiers, events, struct types, and enum types. Each contract may have only one *constructor* which is executed before it is actually deployed on the blockchain. This procedure can be considered as performing a contract creation transaction and it can be split in two steps. The first one is to compile the Solidity code into bytecode understandable by the EVM. The easiest way to do this is in JavaScript by using Node.js and Solc package. From the resulting object it is possible to inspect the actual bytecode and the contract's application binary interface (ABI). The second step is to actually deploy the contract on Ethereum via Web3.js. In particular, the only mandatory input is the ABI but other options can be provided as fallbacks for calls and transactions that involve the newly created contract account.

As explained in chapter five, in the implementation of my CMS I relied on Truffle framework and its built-in functions to easily compile and deploy the contract.

3.2.2 Variables

In Solidity, the type of each variable needs to be specified. Moreover, there is no undefined or null value and each newly declared variable has always a default value. In general, Solidity groups variables in two categories:

- value type variables are always copied when they are used as function arguments or in assignments. For instance, booleans and integers are primitive types in this class. Another important data type of this category is *address*. An *address* variable can store the 160-bit address of an Ethereum account. Given an account address, it is possible to retrieve its balance in Wei (*address.balance*) and send a given amount of coins to it with *address.transfer(amount)* (reverts on failure) or with *address.send(amount)* (returns false on failure and the current contract will not stop with an exception).
- reference type variables can be modified through multiple different names. Arrays, structs and mappings belong to this category.

As stated in the previous section, each code operation can access and modify three types of memory spaces, but only the *storage* of the contract account guarantees data persistence. This is reflected in Solidity once the programmer creates a new variable. Indeed, it is mandatory for reference type variables to explicitly provide also their data location which can be *calldata*, *memory* or *storage*. In particular, *calldata* can be used only for function arguments since it is a non-modifiable, non-persistent area. On the other hand, value type variables are stored in the *memory* by default. An exception from this is provided by *state variables* which are maintained in the *storage* field of the contract account independently from their type. Another difference between data locations is their costs in terms of Gas for read, write and re-size operations.

Solidity also provides three special variables whose scope is the global namespace:

- The *msg* variable allows to inspect the current message call transaction. In particular, it is possible to retrieve the sender address (*msg.sender*), the amount of Wei transferred to the contract in the current transaction (*msg.value*), the *data* field (*msg.data*) or only the first four bytes of *data* that identify the triggered contract function (*msg.sig*). This variable is updated any time a new message call transaction is addressed to the contract i.e. each time an external function call happens.

- The *block* variable allows to inspect the current block. In particular, it is possible to retrieve the *beneficiary* i.e. the address of its miner (*block.coinbase*), the *number* of the block (*block.number*), its *gasLimit* (*block.gaslimit*), the *difficulty* parameter used in the mining process (*block.difficulty*) and its *timestamp* (*block.timestamp*). Since miners choose the transactions to include in a block, the *block* variable refers to the one the transaction is being included in. Thus, these information depend on when the miner executes the current transaction.
- The *tx* variable allows to inspect the current transaction. In particular, it is possible to retrieve its Gas price (*tx.gasprice*) and the original sender (*tx.origin*). The difference between *tx.origin* and *msg.sender* is easily explained by recalling the previous section. Message call transactions can be performed by EOA and contract accounts to communicate with each other. However, a transaction is always originally triggered by an externally owned account and eventually it may lead to an interaction between contracts. Therefore, it is clear that *tx.origin* always provides the address of the EOA that started the transaction, while *msg.sender* may or may not be equal to *tx.origin* depending on the implementation. In particular, they refer to the same address if there is no external function call in the code executed with the original transaction.

In the implementation of the CMS, I used a lot the *msg.sender* variable in order to provide authentication of the users.

3.2.3 Visibility, functions and modifiers

As stated before, a contract is similar to a class and in this analogy, its functions can be referred to methods. As in other programming languages, Solidity defines the visibility that developers can assign to functions and state variables.

- Functions labelled as *external* are part of the contract interface, so they can be called via message call transactions from other contracts and EOA. This functions can not be called internally from the contract itself unless *this* keyword is used.
- Functions or state variables declared as *public* are part of the contract interface and they can be either called internally or via message call transactions. In particular, Solidity automatically generates getter functions for public state variables.
- Functions or state variables labelled as *internal* can only be accessed from within the current contract or contracts deriving from it.
- Functions or state variables declared as *private* are only visible for the contract they are defined in and not in derived contracts. It is important to recall that making something private only prevents other Ethereum accounts from calling it or modifying its value, but it will still be readable to the whole world outside of the blockchain.

By default, each function in Solidity has the right to modify the contract's state variables. However, it is possible to specify two types of functions that are not allowed to write new values on state variables nor to send Ether from the contract balance. In particular, *view* functions promise not to modify the state, so they are perfect as getters. Moreover, *pure* functions do not allow neither to read state variables and also the access to *block*, *tx* and *msg* is limited. Therefore, *view* and *pure* functions do not use any Gas and they are executed directly by the Ethereum node the caller is connected to. As stated before, these operations are referred as calls as opposite to actual transactions. The complete list of restrictions for these functions can be consulted in Solidity documentation.

Lastly, I want to mention modifiers, payable and fallback functions because I used them in the project. A *modifier* can be used to easily change the behaviour of functions. It is commonly applied to automatically check a condition prior to executing the function. The combination of modifiers, state-reverting exceptions (like *require(bool)*) and special variables is important to perform an authentication control over the account that called a function. As regards payable and fallback functions, they address the problem of receiving the Ether sent in the message call transaction. All coins collected by these function are in the contract *balance* which can be inspected with *address(this).balance*. In particular, a contract can have exactly one unnamed fallback function. This *external* function has no arguments and no return values. It is executed on a call to the contract if none of the other functions match the given function identifier or if no *data* was supplied at all. It is important to have this function because it allows the contract to receive coin without executing any code.

4

The InterPlanetary File System

As explained in chapter two, the InterPlanetary File System (IPFS) is the technological component that I chose for the storage and the delivery of contents published by SMEs on the platform. Therefore, in order to integrate IPFS in my system, I had to study this content-addressable, peer-to-peer protocol. This chapter is split in two sections: the first one provides a brief description of consolidated technologies that inspired IPFS, then the second one explains how IPFS is actually designed and particular focus is given to BitSwap protocol.

4.1 IPFS background

IPFS [34] aims at creating a global, versioned, peer-to-peer file system that replaces HTTP and provides a more open and decentralized web. In order to achieve this goal, IPFS reinvents and relies on consolidated technologies like distributed hash tables, the BitTorrent protocol and Git.

4.1.1 Distributed hash tables and Kademlia

Distributed Hash Tables (DHTs) are widely used to coordinate and maintain metadata about peer-to-peer systems. The most fundamental aspects of a DHT is the existence of a common key space and the definition of a *lookup(x)* operation, which returns data associated with a key x. In a file sharing context, keys represent both nodes' identifiers and files addressed by the system. A peer with key x is responsible for the storage of information about files whose keys

are closer to x than any other node identifier according to some distance function. Moreover, all peers maintain information about a subset of other participants in the network, usually *O(logN)* where N is the total number of peers in the system. Thus, a DHT provide a sort of scalable routing mechanism that peers have to follow in order to retrieve the desired key. Several peer-to-peer systems have implemented these concepts each with their peculiar features. The one that mostly inspired IPFS is Kademlia [35] and its modified version S/Kademlia [36].

In Kademlia, the keys are 160-bit values (or 128-bit depending on the implementation) and the distance measure is defined with the bitwise XOR operation. Each node in Kademlia keeps a list of peers with a distance between 2^i and 2^{i+1} from itself for each $0 \leq i < 160$. In other words, the *i-th* list can contain nodes whose identifiers have a common prefix of $160 - i$ bits. Therefore, for small values of i the lists will generally be empty, but for bigger i they can grow up to size k. The parameter k is chosen so that any given k peers are very unlikely to have a failure within an hour to each other. These lists are named k-buckets after this value k. The elements in each list are triplets *(IP, UPD port, NodeID)* and an entry is added after a direct interaction with the related peer which proves its availability in the network. It is important to notice that since the number of nodes in each list decreases with smaller distances, k-buckets with smaller i provide a much more detailed description of the network in the neighborhood of the node in focus. This is a general feature of DHTs: each peer has better knowledge of the close keys, but it has information to reach the opposite side of the key space anyway. Moreover, in Kademlia, the storage information about a file referenced by the key x is addressed by the k closest nodes to x. It is also possible to implement a policy to replicate files distributed by the system in order to improve their availability. For instance, consider a peer that publishes a file with key x. Instead of just informing the k closest nodes to x about the new file it can provide to others, it could also transfer them the actual data. In this way, the k closest nodes to x are the ones responsible to serve it in the network. However, this replicas policy introduces a big overhead in the system and it forces peers to download and distribute unknown files also against their will while they are participating in the network. Therefore, depending on the purposes of the system that relies on Kademlia DHT, this replicas procedure may or may not be implemented even though the original protocol includes it by default.

Kademlia provides four protocol messages:

- *ping* is called by a node to see if the recipient of the message is alive.

- *store* is used to instruct a peer to store a *(key, value)* pair in the DHT.
- *find_node* is a RPC that passes a 160-bit node identifier to the recipient. Then it returns the *(IP, UPD port, NodeID)* triplets for the closest k peers to the target it is aware of. Usually, these values come from the same k-bucket if it has enough entries.
- *find_value* is very similar to *find_node* but in case the recipient actually has the required key, it returns directly the associated file.

Therefore, the general *lookup(x)* operation is actually addressed by Kademlia through the *find_node* and *find_value* functions. They are both performed by a node which at first, selects n peers from the closest k-bucket to the target key x. Then, it asynchronously forwards the request for their k closest known nodes to x. Once it receives the responses, the initiator updates its k-buckets and repeats the procedure by selecting n of the newly retrieved nodes. This process continues until no peers closer to the target are found or the requested value is retrieved.

One of the problems of Kademlia is given by nodes' identifiers. Indeed, they are random values of 160-bits and they may not be equally distributed over the key space. Moreover, it should be hard for an attacker to generate a large number of node IDs or choose a node key freely. This should be done to prevent the Eclipse attack which tries to place malicious nodes in the network in order to cut off from it one or few target nodes. Another reason to do this is to impede the Sybil attack that consists in creating a big fraction of nodes to control the moderation/reputation mechanism that drives all peers' behaviour. S/Kademlia addresses these security issues introducing a procedure for the generation of node identifiers. In particular it provides a PKI that ensures peers' authentication and the messages' integrity. Then it impedes the Sybil and Eclipse attacks with a sort of proof-of-work puzzle. Another problem regards the *lookup(x)* procedure because it fails as soon as a single malicious peer is queried. S/Kademlia extended the process by making lookups independent and ensuring that the routing paths are actually disjoint. More information about this can be found in [36].

4.1.2 BitTorrent

BitTorrent [37] is a very popular peer-to-peer file sharing protocol and its data exchange procedure heavily inspired the design of IPFS.

In BitTorrent, the nodes in the network do not transfer entire files with each other since they are split into *pieces*. The information about published contents are provided by their

authors via *torrent* files. A torrent contains several metadata like the name, the size, the hash of all pieces and the length of each segment. Thus, in order to retrieve and participate in the distribution of a content, a node has to have its torrent. Then, it is able to join a *swarm* of peers that share the same torrent in order to download and upload simultaneously each segment of data. Indeed, each torrent has a field that expresses the URL of a *tracker* which is a server that eases the communication between peers by keeping a log of nodes in the related swarm. The tracker is not directly involved in the transfer of data and does not have a copy of the file. However, it is a potential central point of failure and newer versions of the protocol introduced tracker-less torrents, where the process of finding peers that provide a file is replaced by a Kademlia-like DHT called Mainline [38].

If we consider a specific torrent and its swarm, the BitTorrent protocol defines three types of peers:

- A *downloader* is simply a peer that does not have all file's pieces and it is currently downloading them.
- A *seed* is a node that uploads the already collected pieces for other peers in the swarm. Obviously, it may be also a downloader in case it does not have the whole file.
- A *leecher* is a downloader that does not contribute to the upload of data segments. Therefore, a leecher has a negative effect on the swarm.

Obviously, the fact that someone has the torrent of a file does not imply its actual availability in the network. Indeed, this is only guaranteed if all its pieces are provided by online peers. BitTorrent mitigates this issue by studying carefully the sequence followed by a downloader to request data. The goal is to replicate different pieces on different peers as soon as possible. This is important for file persistence, especially for unpopular contents. Indeed, it should be avoided a situation in which every seed has all the pieces that are currently available and none of the missing ones. Moreover, a carefully designed policy improves the performance of the protocol. For instance, if many downloaders request the same piece at the same time, the seeds that provide it may not have the required bandwidth to satisfy the demand.

The algorithm implemented by BitTorrent is called *rarest first* [39] and selects the next piece to download as the one that is provided by the smallest number of seeds. In particular, each downloader retrieves these information from the tracker or using Mainline protocol messages. The idea behind this algorithm is obviously to reduce the risk of missing one or more pieces once a seed leaves the network. However, the *rarest first* procedure is modified by combining three additional policies:

- The *random first* states that the first four pieces to download are chosen randomly, then the *rarest first* can be applied. Here, the aim is to permit a downloader to retrieve quickly its first data segments. This is also important for the *choke* algorithm.
- The *strict priority* is applied to the so called *blocks* or *sub-pieces*. They are the actual transmission unit in which all pieces are split. This policy states that once a block has been downloaded, the others of the same piece are requested with the highest priority. The goal of the *strict priority* is to complete the download of a piece as fast as possible.
- The *end game mode* is the consequence of the fact that the protocol requests the same block to different nodes. This pipelining principle is implemented to retrieve pieces as quickly as possible. Thus, each time a block is received, the *end game mode* sends a remove message for the received block to all the peers that have the corresponding pending request.

Another issue that affects the file availability and distribution is provided by leechers. Indeed, their selfish behaviour certified by a small share ratio should be penalized due to their lack of commitment in the swarm. This problem is addressed by BitTorrent with the *choking* algorithm. A node can temporarily refuse to upload to another one, but still accept data from it. This limitation in the cooperation between peers in the same swarm is called *choke*. Thus, the basic idea of the *choking* algorithm is that a node keeps uploading to peers that have recently shared pieces with it, otherwise it chokes the connection. This policy is a sort of tit-for-tat strategy, but there is the problem to determine which peers to choke and which to unchoke. A downloader unchokes every 10 seconds the three peers with highest share ratio with itself and every 30 seconds it unchokes a random peer. Therefore, each downloader can have at most four remote peers as unchoked at the same time. The randomly chosen node is called the *optimistic unchoking* peer and this strategy is mainly implemented to give new peers their first piece to share. With the introduction of Mainline DHT, this algorithm has been slightly modified especially for seeds that can provide all pieces. However, its description goes beyond the purposes of this chapter and further information can be found in [39].

4.1.3 Git's directed acyclic graph

Version control systems (VCSs) record changes to a file or set of files over time in order to be able to recall specific versions in the future. They are fundamental tools to manage software development workflows, especially when more people are involved in the project and conflicts may occur. Git is a content-addressable filesystem and undoubtedly it is the most

popular distributed VCS. Git represents a repository's history with a directed acyclic graph (DAG) where nodes are linked through hashes. This data structure is extremely important in IPFS since files are split in content blocks and a Merkle DAG describes how they fit together.

Git defines basically four main types of nodes for its DAG. They are called *objects* and, given a repository, they can be found in the *.git/objects* folder.

- The *blob* (binary large object) is the object type used to store the contents of each file. Each blob is associated with a key that allows to retrieve it. A blob is created for each file passed to the command *git add*.
- The *tree* object allows to store a group of files together, so it is similar to a directory. A single tree contains one or more entries, each of which is the SHA-1 hash of a blob or sub-tree.
- The *commit* object specifies the top-level tree for the snapshot of the project at a specific point. In particular, it stores the information about the author, the committer, the eventual parent commit and the provided message.
- The annotated *tag* is an object used to emphasize specific points in a repository's history as being important. They contain the tagger name, email, date and a message.

These objects are content addressed, so each one has its own identifier and it is possible to inspect their content with the command *git cat-file*. This identifier is obtained with SHA-1 and the first two characters are used to name the directory in *.git/objects* that contains the object, while the remaining ones identify the object file. As an example, the DAG resulting from Listing 4.1 is shown in Figure 4.1 where keys are truncated to the first seven characters.

Listing 4.1 Git DAG example

```
1: $ git init
2: $ echo "This is the README" >> README
3: $ git add README
4: $ git commit -m "first commit"
5: $ echo "Modify the README" >> README
6: $ git add README
7: $ mkdir dir
8: $ echo "Hello" >> dir/hello.txt
9: $ git add dir
10: $ git commit -m "second commit"
11: $ git tag -a -m "annotated tag" v0.1 3030a8d
```

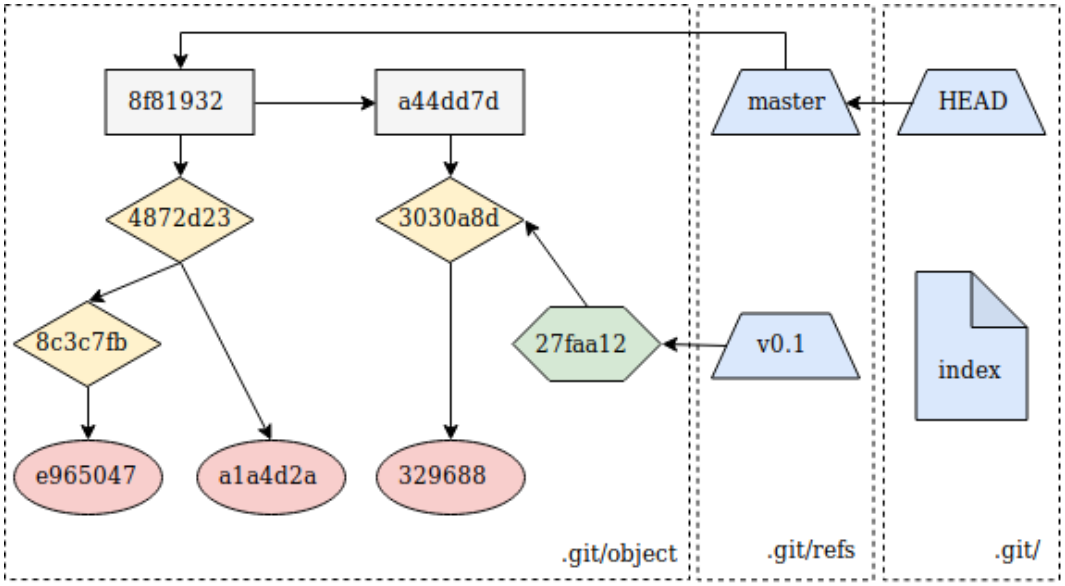

Figure 4.1: Git DAG example. Nodes in red are blobs, those in yellow are trees, commits are in light gray and the one in light green is the annotated tag.

In particular, line 2 creates a blob with key 329688 and line 4 creates the commit $a44dd7d$ and the tree $3030a8d$. Then, I modified the README file and I created the blob $a1a4d2a$ containing the entire README. At line 9 I added the tree $8c3c7fb$ that contains the *hello.txt* file $e965047$. The commit $8f81932$ resulting from line 9 created the tree $4872d23$ that includes the sub-directory *dir* with its content and the modified README. Finally, the annotated tag with identifier $27faa12$ that points to the object $3030a8d$ is created.

It is important to notice that the parent of the second commit is exactly the hash of the first one. This shows how it is possible to inspect the history of the repository by following the hash links between nodes. Moreover, this implies that versioning metadata like branches and tags accessible from *.git/refs* are simply pointers that contain the SHA-1 identifier of the related object. Using the command *git log –graph* it is possible to inspect the graph created by commit nodes.

4.2 IPFS design

The InterPlanetary File System is a peer-to-peer system designed as a stack of protocols that implement different functionalities combined in three main components:

- *libp2p* [40] addresses the problem of nodes' identity generation and verification. Then, it provides the network protocols that manage peers' connection and the routing pro-

cedures to locate specific nodes and objects. It also introduces a protocol for data exchange between peers called *BitSwap*. With this protocol, nodes only store and/or distribute content they explicitly want to store and/or distribute. Therefore, there is no content replication policy implemented by default.

- The *InterPlanetary Linked Data* (IPLD) [41] organizes objects into a DAG that makes IPFS a content-addressed and versioned file system. This implies an interesting feature of IPFS which is the fact that published data are immutable. That's because changing an object would change its hash and thus its identifier, making it a different object altogether.
- The *InterPlanetary Name System* (IPNS) [42] is a naming layer that manages the creation of mutable pointers to permanent objects and human-readable names.

This section covers all these aspects, but it is important to remark that IPFS is constantly updated, so it is possible that implementation details may change in the future.

4.2.1 Peer identities

The first problem to address in a peer-to-peer system is to define a procedure that assigns node identifiers in a secure way. In particular, it should be able to prevent the Sybil and Eclipse attack, but also to provide a *nodeId* that actually authenticates a peer (i.e. none should be able to steal or fake a *nodeId*).

The solution adopted in *libp2p* is the one designed by S/Kademlia with its static crypto-puzzle. In order to create its own *nodeId*, each peer has to generate a private and public key pair. If x is the public key, the node has to compute a value P defined as

$$P = H(H(x)) \tag{4.1}$$

where H a cryptographically secure hash function. Then, the procedure introduces an integer parameter d which represents the overall difficulty. Finally, if P has at least d most significant bits set to zero, the *nodeId* is $H(x)$. Otherwise, the procedure is repeated from the generation of the keys until a valid P is found.

The complexity of this crypto-puzzle is exponential in d and, as stated before, it is very similar to the proof-of-work consensus protocol. Thus, it is particularly effective against Eclipse attack because a node can not choose its identifier freely. Therefore, it is almost impossible to control a part of the network or cut off one or more peers. Moreover, since peers

exchange their public keys once they first connect in the network, this solution allows to perform nodes' authentication. Indeed, once a node receives a signed message, it can validate the signature and then checks if the sender solved the crypto-puzzle. This can be done simply by verifying if $H(sender.pubKey)$ equals its *nodeId*. In case the two values do not match, the connection with the other peer is terminated.

Finally, a note on the hash function and a little clarification about *nodeId*. In order to update the system once newer and more secure hash functions will be defined, IPFS allows to select the desired H. The consistency of the system is achieved by storing digest values in a *multihash* format [43]. In particular, it specifies in the prefix a code to identify the hash function H used and the number of bytes in the digest. For instance, consider a peer with public key "hello world" that uses SHA-2-256. At first, it computes the hash of the key which is $0xb94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9$. This is not the proper *nodeId* as stated before, because it is not in the *multihash* format. Thus, it requires a prefix of two bytes to be appended: one that identifies SHA-2-256 and the other that references a 32 bytes long digest. Currently, their respective value is $0x12$ and $0x20$. Finally, the concatenated result is encoded in base58 and the resulting *nodeId* for this peer is $QmaozNR7DZHQK1ZcU9p7QdrshMvXqWK6gpu5rmrkPdT3L4$. As consequence, a peer that receives a message from that node still has all the information to check the validity of the crypto-puzzle and the signature of the message. As explained later, this *multihash* format is also used to identify objects provided by the network.

4.2.2 Network and Routing

The *libp2p* library is designed to allow IPFS to run over any transport protocols. Moreover, IPFS will be able to operate on top of architectures like NDN [44], since there is no assumptions made about the network layer in the protocol. In order to achieve this goal, IPFS uses addresses in *multiaddr* format [45]. They allow each node to define the details of the protocols it runs and the corresponding information. For instance, */ip4/192.0.2.42/tcp/443* is a valid *multiaddr* because it is composed as a chain of *(protocol name, value)* pairs. Since *multiaddr* specifies how peers communicate with each other, usually a complete address ends with */ipfs/nodeId* in order to identify nodes in the swarm. Finally, *libp2p* provides an ICE-like protocol [46] to accomplish NAT traversal and it also implements HMAC to perform message authentication.

Moreover, *libp2p* is responsible for the routing procedures that allow to find a specific node and the peers that provide a particular object. The solution adopted by IPFS is a DHT

based on S/Kademlia and Coral [47] which can be considered as extentions of Kademlia. In particular, IPFS provides an interface with five main protocol messages:

- *find_peer* is called to get the *multiaddr* addresses of a specific node.
- *set_value* is used to instruct a node to store small metadata in its DHT. The format is as always a *(key, value)* pair.
- *get_value* is used to retrieve the data corresponding to a specific *key* in a DHT.
- *provide_value* announces that the current node can serve an object with a given identifier expressed in *multihash* format.
- *find_value_peers* is used to collect a minimum number of *nodeIds* that provide a specified object.

An alternative to this Kademlia-like routing is provided by multicast DNS. In this strategy, when a node starts or detects a change in the local network, it sends a query for all peers. As responses arrive, the peer updates its information about the others in the local database. This proves the flexibility of IPFS routing system which can be changed to fit users' needs as long as the described interface is met.

As an example, Listing 4.2 shows how these routing procedures can be accessed with IPFS command line interface. In particular, the first line starts a local node and it is possible to inspect known peers with the command in line 3. It provides not only their *nodeIds*, but also their *multiaddr*. Then, all DHT operations are listed from line 4. As you can see, all the original protocol messages described before are provided (even though with different names) and the *query* command is added, which is used to find the closest *nodeIds* to a given *nodeId*. Check the documentation [48] for a more detailed description.

Listing 4.2 IPFS CLI commands that maps routing protocol

```
1: $ ipfs daemon
2: ## next commands in another terminal
3: $ ipfs swarm addrs
4: $ ipfs dht findpeer <nodeId>
5: $ ipfs dht findprovs <key>
6: $ ipfs dht get <key>
7: $ ipfs dht provide <key>
8: $ ipfs dht put <key> <value>
9: $ ipfs dht query <nodeId>
```

4.2.3 BitSwap protocol

In IPFS, BitSwap is responsible for data distribution between peers but it does not guarantee their persistence nor it creates content replicas. Like BitTorrent, in this protocol implemented by *libp2p*, files are split in *blocks* which is the smallest unit of data transferable. Another similarity is provided by the principles inspiring the exchange strategy. However, a substantial difference is that in BitSwap there is no concept of swarm of peers that provide pieces of a specific torrent. Indeed, each node can acquire the blocks it needs regardless of what files those blocks are part of.

In order to explain how BitSwap works, consider, for instance, a peer x that sends a request message for a set of desired blocks to some known nodes. Now, suppose that after some steps of the routing procedure, at least a peer y that serves those pieces is found. BitSwap allows y to decide whether to begin the data transfer or not depending on the commitment to each other, which is measured in terms of bytes exchanged in the past. The share ratio r (also called debt ratio) is a concept considered also by BitTorrent's choking algorithm, but in BitSwap it is the core parameter of a function that states the probability of sending a block to x given the current share ratio.

$$P(send|r) = 1 - \frac{1}{1 + exp(6 - 3r)} \text{ where } r = \frac{bytes\_sent}{bytes\_received + 1} \tag{4.2}$$

Now, if r is between 0 and 1 it means that y is in debt with x, so it will likely accept the transfer. On the other hand, as the bytes credit (i.e. the ratio) grows, the probability drops. The reason is obviously to deal with leechers that never share blocks to others and to choke deteriorated relationships between nodes until a fairer balance is re-established. Moreover, this function does not penalize too much seed nodes that want to serve blocks without anything in exchange.

It is interesting to notice that having a slightly positive or equal share ratio is a good investment for peers and this fact leads to a side effect regarding pieces' distribution. Indeed, if x is an honest node in debt with y, it is in its interest to re-balance the share ratio because y has proven to be a reliable seed. On the other hand, y can exploit its credit by requesting blocks to x who will transfer them with high probability. The easiest strategy that leads to this balance condition regards rare blocks. For instance, y is one of the few other peers that serve some rare pieces of potentially popular contents. On the other hand, x is someone that had to request those data in order to complete a download. The lack of providers makes x prone

to accept the transfer from *y* even though this leads to the creation of a debt with it. Therefore, BitSwap implicitly incentives nodes in the network to cache rare pieces, even if they are not interested in them directly because those blocks potentially guarantee high share ratio with other peers. In a sense, this remembers the *rarest first* policy implemented by BitTorrent since the result is the dissemination of rare pieces which improves content availability and persistence.

In order to perform block exchange and this transfer decision, the BitSwap protocol defines a set of state information that each peer has to maintain.

- A set of *ledgers*, one for each of the nodes it is connected to. Each ledger contains the *nodeId* of the partner and the information regarding bytes sent and received to/from it.
- A set of *peers* that represents currently open connections.
- A *need_list* that contains the *multihash* values of blocks needed by the current node.
- A *have_list* that contains the *multihash* values of blocks the current node can transfer to others.
- A *want_list* that contains the *multihash* values of blocks wanted by the current node. As opposed to *need_list*, it also contains blocks that the current node downloads to re-balance the share ratio with some of its partners.

Moreover, BitSwap defines an interface that all peers have to satisfy. In particular, it consists in four main methods that manage nodes' interaction. Obviously, they integrate and rely on the routing procedures described before.

- The *open* function aims at creating a new connection with a specified node. The caller sends a message to the recipient that contains a ledger. In particular, if the two peers have exchanged blocks in the past, the ledger is a copy of the one stored by the sender, otherwise its fields are set to zero. The recipient of an *open* message compares the provided ledger with the one in its possession and if they match, it can decide to accept the connection. In case it is refused, the receiver may ignore further requests by default for the duration of a timeout.
- The *send_want_list* function can be called once a peer has at least one open connection in order to advertise its *want_list*. Currently, a node sends the *want_list* once a connection is opened and then after a randomized timeout. In addition, a new message is sent in case the *want_list* is modified. The receiver, checks if it has any of the requested blocks in its *have_list* and then it decides whether to transfer the data or not depending on Function 4.1.

- The *send_block* function simply transfers a block in the *want_list* from a peer to the other using the protocol specified with its *multiaddr*. Once the data is received, the node verifies the integrity of the data with the same procedure as the one described for peer identities. Then, if the test is passed, the block is removed from the recipient's *need_list*. Finally both the peers involved update their respective ledgers to keep consistence in the amount of bytes exchanged and the receiver removes the block from its *want_list*. If the data fails the test, the receiver is free to refuse future requests.

- The *close* function is called to inform the partner peer that the connection is closed. The message provides a boolean parameter to the recipient that states if the intention to tear down the connection is the sender's or not. The second case is often due to internal errors or to a lack of messages received in an interval of time. At this point the partner's *want_list* can be removed and eventually, a new *open* procedure can be called to re-establish a new connection.

Finally, it is important to remark that while a connection is established between two peers *x* and *y*, both know the *want_list* of each other. Therefore, if *x* requests a block provided by *y* but it currently has no block to exchange, *x* could update its *want_list* and seek also pieces wanted by *y* but with lower priority. This can be done in order to re-balance the share ratio during the current connection and the consequence is a wider dissemination of blocks in the network.

Listing 4.3 IPFS CLI BitSwap example

```
1: $ ipfs daemon
2: ## next command in another terminal
3: $ ipfs get Qmdsrpg2oXZTWGjat98VgpFQb5u1Vdw5Gun2rgQ2Xhxa2t
4: ## while download is running, in another terminal
5: $ ipfs bitswap wantlist
6: $ ipfs bitswap ledger <nodeId>
7: Ledger for <peer.ID Qm*r5QBU3>
8: Debt ratio: 0.000000
9: Exchanges:  11
10: Bytes sent: 0
11: Bytes received: 2883738
```

As an example, consider Listing 4.3 where in line 3 we want to download from the network a large file. It is a video of a lesson taken by Juan Benet, the author of IPFS, at Stanford University in 2015. Under the hood, the *get* command selects one of the peers returned

by *find_value_peers* routing procedure. While the connection is open and the download is running, it is possible to inspect the *want_list* of the other node with the command at line 5. Finally, once the download is over, we can inspect the ledger and thank the seed. Note that the share ratio is zero because we did not send anything back in exchange.

4.2.4 Objects and Files

As stated before, *libp2p* is responsible to make IPFS a peer-to-peer system where nodes exchange blocks of data with a defined policy. On top of this layer, IPLD builds a Merkle DAG where objects (i.e. DAG's nodes or vertices) are referenced with identifiers created from their content's hash. Moreover, the edges in the graph are embedded in the source objects as identifiers of the destinations. This idea is clearly similar to Git's DAG described before and it is one of the most important feature that make IPFS suitable for my project. Indeed, as stated in chapter two, this content representation is perfect to be stored in the blockchain.

Therefore, the first issue that IPLD has to address is obviously to provide a specification for object identifiers (CIDs). How to address contents is a problem that interests also underlying layers, thus the first version of CIDs has the same *multihash* format of *nodeIds*, where the first byte represents the hash function used and the second one is length of the digest in bytes. Moreover, all CIDs of the first version use SHA-2-256 and they are encoded in base58 by default. On the other hand, newer CID format (version 1) has four fields but retro-compatibility is guaranteed.

- *mbase* is a multi-base prefix describing the base that encodes this CID.
- *version* is the version number of the CID i.e. 0 or 1.
- *mcodec* is a multi-codec identifier used to serialize and deserialize the block once it is sent from peer to peer. Currently supported formats are JSON, YAML, XML and others.
- The usual *multihash* determined from the hash of the data.

These CIDs are important because they make IPFS links permanent. In other words, it means that each object and also each modified version of the same content have unique identifiers. Therefore, even if an object published in the past is no longer available in the network, its identifier is still valid and maintained in the DAG by available parents. Once it is clear how to address objects, IPLD defines four main types of nodes in the DAG in order to model a versioned file system:

- The *blob* object represents a file and contains an addressable and transferable unit of data. Therefore, there is a match between each blob and the related block provided by peers. Moreover, these objects can only have incident edges in the DAG. Here, it is important to notice the similarity with Git's blob.
- The *list* object represents a large file that is split into an ordered sequence of blobs because it does not fit in a single block. It is also possible for a list to contain other list objects. Therefore, this type of vertex has many outgoing edges pointing to the related blobs and lists in the DAG. As stated before, these relationships are maintained in the parent node that stores all the CIDs of its children in a list. This type of node is introduced in order to keep consistency with underlying protocols.
- The *tree* object represents a directory and it is basically the same concept as Git's tree. Like list objects, trees are meant to maintain many outgoing edges pointing to blobs, lists and other sub-trees.
- The *commit* object represents a snapshot in the version history of any object represented in the DAG. Therefore, it can be linked to any blob, list, tree or commit. As consequence it is possible to reveal differences between two versions by comparing the commit objects in focus and all their children.

The publication of a new object can be performed by any peer in the network by simply adding its CID in the DHT. Then, anyone that knows the key can download the content and eventually add it to their *have_list*. In order to ensure the persistence of particular objects in the local storage and save it from garbage collection operations, a node can *pin* these contents. However, this procedure does not guarantee at all the data persistence in the network, which can be provided if and only if there is at least a peer serving all content's blocks.

A good example of these concepts can be found in IPFS white paper [34].

4.2.5 Naming

At this point of the protocol stack, IPFS is a peer-to-peer system for exchanging blocks of data representing a content-addressed and version-aware DAG of objects. However, it is convenient to have mutable pointers to permanent objects like Git's branches and tags. For instance, this can be useful in order to automatically redirect people to the last version of a content even though they do not have the newer CID. As stated before, IPNS is the component that provides this functionality taking inspiration from self-certifying file system [49].

IPNS is a simple service that assigns every user a mutable namespace where it can publish an object signed with its private key. This namespace is accessible from */ipns/<nodeId>* and

the most important fact to remember is that *nodeIds* do not change. Therefore, if the author of a content knows in advance that newer versions will be released, it should publish the content under a commit object with IPNS and share the related address. Then, once an user retrieves the stored object, it can verify that the signature matches the public key and *nodeId* of the peer.

As consequence, in IPFS there are two main types of paths to address contents in the DAG:

- *mutable* references begin with */ipns/* and they are pointers to permanent objects. In the *set_value* message of the routing procedure, the actual metadata added in the DHT is the pair *(nodeId, ns_object_hash)*.
- *immutable* references begin with */ipfs/* and represent the actual CID of permanent objects.

Finally, since these addresses are difficult to remember due to hashes, IPFS improves user friendliness and readability. Indeed, it allows users to link others' objects directly into their own ones, creating a web of trust. Moreover, it allows IPFS to look up */ipns/* addresses in its DNS TXT records.

5

CMS implementation

Once I designed the CMS architecture, and after having studied in deep its two main components, I moved to the actual implementation of my solution for SMEs online aggregation and product advertisement. The organization of this chapter follows my workflow as a developer. At first, I listed all the functional and non-functional requirements that the proposed CMS has to satisfy. Some of those have already been pointed out previously and guided me through the architecture design. However, their clear and complete definition was important in order to implement the software functionalities. In particular, I focused on the problem of unbiased moderation of both contents and group members. The solution I adopted is a majority vote triggered each time a controversial decision has to be taken. Then, I searched for useful frameworks and tools capable of easing the implementation of the software. Once the project organization had been set up, I begun with the actual coding.

Although the architecture of the proposed CMS is not the standard structure of a web application, it is possible to identify all the features of the Model-View-Controller (MVC) design pattern. Indeed, the smart contract represents the controller component that implements the business logic of the CMS. As regards the model, the responsibility of application's data can be split between the Ethereum blockchain and IPFS. As stated before, the identifiers of published contents provided by IPFS (CIDs), the group members etc. are stored in a persistent way in the *storage* state field of the contract account. On the other hand, the actual upload and download of both product images and enterprises' logos are performed via IPFS.js. Therefore, the software development begun with a precise implementation of the

smart contract. Then, I designed the view component starting from the navigation process that a standard user has to follow and the mockups to describe the content disposition. In particular, the presentation logic of the application exposes all the functionalities provided by the contract.

5.1 Requirements analysis

This section discusses the requirements that the system has to satisfy. In order to define them properly, it is important to recall the main goal of the software. It consists in a distributed CMS that has to provide a common platform where SMEs can advertise their products. Moreover, moderation policies of both contents and group members have to be unbiased in order to overcome entrepreneurs' skepticism.

5.1.1 Functional requirements

Functional requirements define what the system has to do. The following list describes them all and provides a brief motivation for each one.

- The application has to model enterprises which provide general information like name and e-mail. Moreover they can have a profile picture to show their logo.
- The application has to keep track of registered enterprises in a permanent and secure way. This is mandatory in order to identify different users of the platform which can be either casual visitors (i.e. potential customers the platform is meant to attract) or group members.
- The application has to allow a group member (i.e. a registered enterprise) to update its personal information. In particular, it has to be possible for a registered member to update the logo of its enterprise.
- The application has to allow the registration of new members in the community of enterprises. Moreover, it has to check not to insert two times the same user. This is important in order to pursue the aggregation of SMEs.
- The application has to model a product and provide some information like a brief description and its price. Moreover, each post can have a picture of the related product.
- The application has to allow registered enterprises to publish products on the platform. Each user has to be able to retrieve and inspect all posts.

- The application has to maintain a list of all products published by each of the registered enterprise in a persistent way. This is mandatory in order to show authenticated posts and identify their author.

- The application has to model a poll which is used to evaluate the consensus of the community on a moderation decision. This procedure is mandatory in order to ensure entrepreneurs about the impartiality of the platform.

- The poll has to provide a subject, a brief description and an expiration time. Moreover, it has to keep track of voters in order to avoid double voting.

- The application has to allow registered users to inspect the details of an active poll. Moreover, it has to allow only registered enterprises to vote.

- There can be at most one active poll at a time and the number of voters can not change while a poll is running. This is mandatory in order to keep consistency during the voting procedure.

- The poll has to record the number of actual voters and how votes are distributed between pros and cons. The decision is approved and the moderation consequence is performed if and only if a quorum is passed (i.e. the actual voters are more than $50\% + 1$ of the total) and the number of yes votes is greater than no votes.

- The application has to trigger a poll once a registered enterprise notifies a moderation issue. There can be three situation that require a poll. The first one regards a potentially inappropriate content to be removed. The second one is to decide whether to ban a member of the community which has been signaled as misbehaving. The third and last one regards the acceptance of a new enterprise in the group.

- The application has to allow registered enterprises to remove their own posts. This procedure has to be performed correctly without any community decision if there is no active poll on a post in focus. Moreover, any member can leave the community freely. However, this can be done when there is no poll running.

- The application has to provide a web interface that allows both casual visitors and registered enterprises to interact with the system. The motivation behind this requirement was already mentioned in chapter two and it regards the popularity of the platform. Indeed, it is meant to engage new customers and a dedicated software installation reduces drastically the amount of reachable people.

- The web interface has to provide a home page where all posted products can be seen by the visitors of the platform. Moreover, the home page has to contain a menu to ease the navigation process.

- The web interface has to provide a profile page which is accessible only if the current user is registered in the community. This page has to list all information of the current member and the products it posted. From this page, a user has to be able to update its information, its profile picture and most importantly to post a new product in the platform.
- The web interface has to provide a poll page where registered enterprises can see if there is an active poll. This page has to report the details of the poll and it has to allow the current user to vote if it has the right.
- The web interface has to allow the registration of a new member in the community of SMEs.

The next sections of this chapter, especially the ones dedicated to the implementation of the smart contract and the web interface will emphasize how these requirements are satisfied.

5.1.2 Non-functional requirements

Non-functional requirements define the features and qualities of the system and they are listed in the following.

- The application state has to be verifiable by any user, especially by entrepreneurs of the community. The motivation behind this feature is the lack of commitment and trust to each other. Therefore, as pointed out in chapter two, the application relies on blockchain technology and in particular on Ethereum.
- The responsibility of actual content availability shall be at first in charge of their authors. The reason behind this feature is again the skeptical behaviour of entrepreneurs. Therefore, the application has to rely on a peer-to-peer protocol that allows the storage and the delivery of all images. As explained in chapter two, the application uses IPFS and it implements a simple replica policy which is not granted by IPFS.
- The content disposition in the home page of the application shall be randomly shuffled. This is important because in this way, each content could appear at the beginning of the home page. Therefore, the visibility that the platform provides to posted products is unbiased. This policy is required in order to overcome entrepreneurs' suspects of favoritism.

As already said, some of these requirements have already been identified before and during the design of the application. Indeed, the issues regarding state verification and content authentication, storage and distribution lead me to rely on Ethereum and IPFS.

5.2 Project set up

The very first step in the actual implementation of the CMS was to search for a framework and tools for developing Ethereum smart contracts. One of the most popular is Truffle [50] which allows to easily compile, deploy and test contracts. It can be installed with Node Package Manager (NPM) so, at first I downloaded Node.js [51] on my machine running Ubuntu 18.04. Then, I was able to install Truffle which currently comes with version 5.0.5. It also provides a local blockchain server with a GUI that displays the transaction history and chain state. This tool is called Ganache [52] and it is very useful to develop and test contracts without actually deploying them in public blockchains like Ethereum main network, Ropsten, Rinkeby or Kovan.

At this point, I created a Git repository and my Truffle project called EtherAd, whose final, but also simplified structure is shown in Figure 5.1. As you can see, it presents many folders and files whose role is summed up in the following list.

- The *build/contracts* folder contains the smart contracts compiled with the command *truffle compile*. These files are in JSON format, so it is possible to read them and inspect the ABI object. The contract compilation is implicitly called by *truffle migrate* command if not manually performed before.

- The *contracts* folder contains all the contracts of the project written in Solidity. The default empty project comes with a *Migrations.sol* file and its role will be explained in a following point. As you can see, the *EtherAd.sol* contract which is the core of EtherAd is placed in this folder.

- The *dist* folder and all its content is meant to be published on IPFS. Recalling Figure 2.2, it is the IPFS object requested and obtained through public gateways or a local node in order to interact with the web interface of the application. In particular, its CID is $QmQvz7QC6u1ojFncmm3cWCCCPG1ZtLXH2maC9diBt4kf4R$. It contains the whole *src* folder and all the compiled contracts. In order to upload it to IPFS, I actually had to install IPFS v0.4.19 on my machine.

- The *migration* folder is meant to contain JavaScript files required for the Truffle migration process which is triggered by *truffle migrate* command and called in order to deploy one or more contracts. The default Truffle project comes with a file called *1_initial_migration.js*. This file is parsed the first time the *truffle migrate* command is called and it specifies that the contract in focus is *Migrations.sol*. Then the procedure compiles it and generates a contract creation transaction which actually deploys it on a network. This *Migrations.sol* essentially maintains a counter that identifies a file in the *migration* folder and it is updated each time a migration is performed. Therefore,

once I created my *EtherAd* contract and I wanted to deploy it, I also had to create a *2_deploy_contracts.js* file that references *EtherAd.sol*. All this process is useful especially in case the project involves many contracts interconnected with each other and the developer wants to add or modify one. Indeed, instead of re-deploying all contracts, this migration procedure allows the developer to specify only the affected ones, thus saving contract creation transactions and Ether.

- The *src* folder contains all the assets used to build the web interface of the application. It has three sub-folders and an HTML file. As their name suggests, in *css* there are some stylesheets written in CSS, in *img* there are some images used in the front-end design and in *js* there are some JavaScript files. They will be discussed in the next sections but, it is important to mention two important libraries contained in *js* folder. The first one is the JavaScript implementation of IPFS, which is called IPFS.js (I used version 0.34.4) and it has been already introduced in chapter two. The second one is the truffle-contract [53] package. In particular, it is required in order to expose an abstraction of compiled contracts and easily perform transactions and/or calls to trigger the execution of contracts' functions. The current version available is 3.0.6.

- The *test* folder is meant to contain files with JavaScript or Solidity code which are used by Truffle to run tests on the contracts. This procedure can be executed with *truffle test* command and I used this feature especially with Ganache before the implementation of the web interface and the poll procedure. As explained later in this chapter, the voting functionality requires a scheduling mechanism whose integration in the project makes this testing unfeasible. The ones I run were written in JavaScript with the Mocha [54] testing framework provided by Truffle.

- *bs-config.js* is a configuration file for lite-server [55], which is a Node package that I used to ease front-end development. It is useful because it serves a web application in a browser tab and it automatically refreshes the page whenever a HTML, CSS or JavaScript file changes.

- The *package.json* file is an important configuration file for NPM and the whole project. Indeed, it specifies the required dependencies on Node packages to be installed with the command *npm install*. Once the *npm install* procedure ends, it generates the *node_modules* folder with all installed packages and the *package-lock.json* file which provides some information on those modules like version etc. In *package.json* file I specified two main dependencies: the already mentioned lite-server (version 2.3.0) and truffle-hdwallet-provider [56] which currently comes in version 1.0.5. This last package is a Web3.js provider used to sign transactions for an Ethereum address given its mnemonic phrase and an Ethereum network access. Its goal in the project is to allow Truffle to deploy the contracts or test them on public networks spending Ether

from my personal balance. Indeed, Truffle has to pay the Gas used during the contract creation or message call transactions and someone has to fund this operations. Finally, I also added two other packages mainly because the project is added on Git. The first one is Truffle itself so that whoever clones the repository can install the exact same set up that I used. The second one is dotenv [57] version 7.0.0 which allowed me to create and manage the content of *.env* file.

- The *truffle-box.json* file comes by default with each Truffle project. It is a configuration file that allows to create *boxes* which are essentially boilerplates for other users available from Truffle website. Since I was not interested in this feature, I ignored it in my project.

- The *truffle-config.json* file is very important because it defines relevant configuration parameters. At first it specifies the Solidity compiler to use. I relied on Solc [58] version 0.5.1 provided by the Ethereum Foundation and bundled with Truffle. Then it allows to specify the networks on which the contracts can be deployed and tested. Therefore, I created a *development* network that points to Ganache private blockchain set at 127.0.0.1 and port 7545. Then I defined the object that references Ropsten which is the test network where I actually deployed application's contract. As stated before, Ropsten is a public blockchain with the same PoW consensus protocol as Ethereum main network. This is the reason why many software products are tested on Ropsten before actually deploying them on the main network where real money are involved. Moreover, blocks, accounts and transactions can be easily inspected on Ropsten through Etherscan [59]. However, since I do not run a local Ethereum node, I had to specify a provider that allows the connection with Ropsten network. In order to solve this issue, I relied on Infura decentralized infrastructure, which requires the creation of a project on their platform in order to access their Ethereum endpoints. Then, I set the Ropsten provider with truffle-hdwallet-provider specifying the mnemonic phrase of my personal Ethereum account and the access point given by Infura at the URL *https://ropsten.infura.io/<infura_project_id>*. Finally, other information can be provided like the maximum amount of Gas that Truffle can use for each transaction on Ropsten.

- The *.env* file is meant to store a variable that addresses the secret mnemonic phrase of my Ethereum account. This is used by truffle-hdwallet-provider to sign transactions performed by Truffle to deploy contracts and test them. Moreover, it stores a variable that specifies the project identifier I created on Infura.

At this point, the project set up is almost complete, but we still miss an important component. As pointed out in chapter two, both SMEs and casual visitors of the platform could not reasonably install and run an Ethereum node in order to interact with the CMS. The easiest

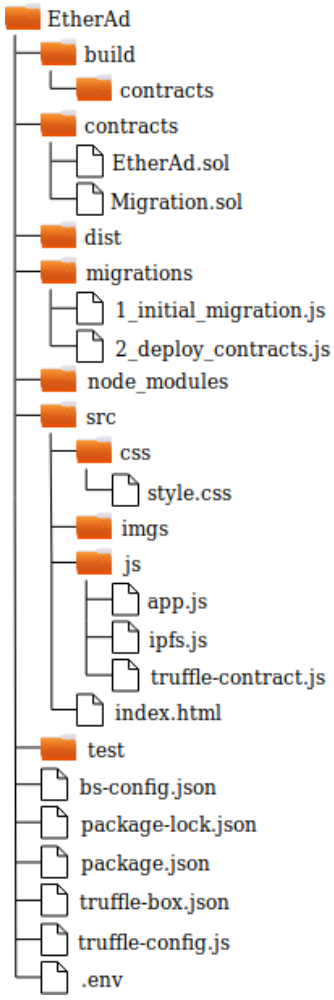

Figure 5.1: Project files structure. This diagram is simplified since the content of some folders is not represented in order to save space.

solution is MetaMask extension that allows the communication between any contract deployed on any Ethereum network (both public or private, so Ganache included) and a web interface through Web3.js. In order to connect with Ropsten, also MetaMask relies on Infura infrastructure. In this way, as pointed out in chapter two, MetaMask is the only piece of software whose installation is required by any user that visits the CMS. In particular, MetaMask allows to create an Ethereum (externally owned) account from a mnemonic seed phrase or to import other ones via private key.

At this point we need some Ether to spend for contracts deployment and testing. This can be easily performed via Metamask faucet for Ropsten [60] where we can ask for free

Ether. Obviously, these coins do not exist in the Ethereum main network, because the transaction that transfered Ether in the balance is on a different blockchain. As regards Ganache, it provides 10 accounts with 100 coins each and they can be imported via private key. It is important to say that in case Ganache is the target network for the deployment of a contract, the amount of Gas used is taken from the first account in the list.

Finally, I want to mention that in order to develop the software, I used Atom editor since it provides useful packages for the Solidity language.

5.3 Smart contract implementation

Once all project components are set up properly, I moved to the actual implementation of *EtherAd* contract which is defined in *EtherAd.sol* and it is the one that coordinates the business logic of the application. This section does not comment all the code, but it focuses on its most interesting aspects, how the requirements are satisfied and the design choices I had to make.

5.3.1 Data model

The very first issue I had to face was how to model enterprises. The first option I had, was to define them as stand alone contracts. In order to understand the consequences of this choice, consider for instance, a user with x as Ethereum address that wants to be registered in the community. In this scenario, it should call a specific function of *EtherAd* contract that creates a new enterprise instance (i.e. contract account with address y). This practice is possible thanks to the *new* keyword provided by Solidity and the actual contract creation transaction is performed by the EOA referenced by x. Then, *EtherAd* has to keep a mapping that associates x with y. This is not important only to know and maintain the registered enterprises, but also to perform easy operations. Consider, for instance, that x now wants to update its profile picture because the enterprise logo has changed. In order to do so, *EtherAd* should perform a message call transaction to the correct contract account. Notice that the function to update this information is in the enterprise contract and not in *EtherAd* because it has no right to modify directly the *storage* field of y account. Otherwise, x can also call directly this (public or external) function, for instance via Web3.js. Anyway, in this scenario, y is literally exposed to any account on the network. Therefore a control on the accessibility of this function has to be implemented in order to allow only x to update its logo eventually through *EtherAd*. As pointed out by [33], there are some good practices to follow in order

to code situations like this in a secure way. Another minor issue regards the efficiency in retrieving information about registered enterprises. Indeed, these data are spread in Ethereum (world) state and not in a specific account.

Therefore, using contracts to model enterprises seems an overkill of the problem which can be solved more easily with structs. Indeed, the definition of an enterprise struct within *EtherAd* allows to create and maintain a variable for each member of the community directly in the contract meant to control the business logic of the application. The same discussion can be extended also to the requirement of modelling both products and the poll. If we consider that the application has to store all products posted by each of the registered application, it is clear that relying on structs is definitely a better design choice. Moreover, since the insertion of a new enterprise in the group has to be approved with a voting procedure and given the fact that registered members and posted products can be banned, it is not reasonable to create stand alone contracts that exist independently on the state of the platform.

In addition to the structs for products, enterprises, and the poll, I also created one to model a circular buffer of Ethereum addresses. Its role is described in the next subsection and in the one dedicated to the moderation policies. The definition of all these data structures is reported in Listing 5.1, Listing 5.2, Listing 5.3 and Listing 5.4. Notice that the *Enterprise*, the *Product* and the *Poll* structs satisfy the requirements about the information to be maintained by the application. In particular, the *Enterprise* and the *Product* have a field that represents the CID provided by IPFS respectively for their profile image and product picture. Here it is possible to notice how well IPFS permanent identifiers can be integrated with Ethereum blockchain. Moreover, the presence of arrays, mappings and so many integers is explained in the next section because it is linked to state variables of *EtherAd* contract. Finally, I want to mention the *PollStatus* and *PollSubject* fields which are *enum* data types important in the moderation procedures.

Listing 5.1 Product struct

```
1: struct Product {
2:     bytes32 pID;
3:     string productImageIPFSHash;
4:     string productDescription;
5:     uint productPriceInWei;
6:     uint ePostListPointer;
7: }
```

Listing 5.2 Enterprise struct

```
1: struct Enterprise {
2:     string eName;
3:     string eMail;
4:     address eAddress;
5:     string profileImageIPFSHash;
6:     uint eListPointer;
7:     uint inListPointer;
8:     bytes32[] ePostList;
9:     mapping(bytes32 => Product) ePostStructs;
10: }
```

Listing 5.3 Poll struct

```
1: struct Poll {
2:     PollStatus status;
3:     PollSubject subject;
4:     address eAddress; // Enterprise in focus
5:     bytes32 pID; // Eventual product in focus
6:     uint yesVotes;
7:     uint noVotes;
8:     string pollDescription;
9:     uint expirationTime;
10:    address[] eBallotStatus; // Enterprises that voted
11: }
```

Listing 5.4 CircularBuffer struct

```
1: struct CircularBuffer {
2:     address[10] list;
3:     uint start;
4:     uint end;
5:     uint size;
6: }
```

5.3.2 EtherAd state variables

At this point, it is possible to identify the state variables of *EtherAd* contract which are maintained permanently in the account's *storage*. Obviously, there has to be a data structure that contains the enterprises currently registered in the platform. The usual strategy in Solidity is to have a mapping between *address* and *Enterprise* variables. However, since mappings are not iterable, there must be also an array of addresses that contains all the keys. In the code I named these data structures respectively *eStructs* and *eList*. Here, each *address* that references an *Enterprise* both in the mapping and in the array is the one of the user that registered in the CMS. An important aspect to consider is that any modification of these data structure is a contract call transaction that needs to be funded. Therefore, in order to save Gas, insertions and removals must be performed efficiently. I addressed this issue simply without sorting the elements in the array and maintaining in each *Enterprise* an integer value that represents its index in *eList*. In this way, the search of a specific *Enterprise* can be done in *O(1)* through the mapping. Moreover, a new member can be added both at the end of *eList* and in *eStructs* in *O(1)*. Finally, the removal of an *Enterprise* can be done in two steps. At first, it is retrieved from the mapping, then its index in the array is available and the removal can be performed in *O(1)* by swapping it with the last element and reducing *eList*'s length. These operations are constant in the number of elements also in case a the data structures need to be resized because they are in the *storage* of the contract. This solution is also adopted to store all the products posted by each enterprise. Indeed, each *Enterprise* contains a mapping between keys of 32 bytes and *Product* variables called *ePostStructs* together with the array *ePostList* containing all the identifiers. In particular, the key of a *Product* is generated with the Keccak 256-bit hash of the string that concatenates the Ethereum address of the related *Enterprise* with its description. Finally each *Product* has the integer that specifies its index in the array of the related *Enterprise*. All these details can be found in Listing 5.1 and Listing 5.2.

Together with the mapping and the array that maintain registered enterprises, other state variables are required. At first, an instance of *Poll* to manage the voting procedure. Then a *CircularBuffer* variable named *inList* and another mapping called *inStructs* between *address* and *Enterprise*. These last variables are mandatory due to the fact that an enterprise is accepted in the community if and only if it is approved with a poll. Therefore, they represent the structures to temporary maintain candidates that want to enter the group. Finally, there are three state variables whose meaning is explained in the following of this section dedicated to the contract implementation. The first one is the *address* of the owner of *EtherAd*, which

references the EOA that deployed the contract. This *address* is defined as *payable*, so this means that *EtherAd* contract can transfer coins from its own balance to this account. The other two are the instance of *Aion* contract and an integer value representing a fee. Their role will be cleared in the section dedicated to the moderation policies.

In conclusion, *EtherAd* state variables are summed up in Listing 5.5. For sake of clarity, the following discussion will reference these variables directly.

Listing 5.5 EtherAd state variables

```
1: address[] public eList;
2: mapping(address => Enterprise) public eStructs;
3: CircularBuffer public inList;
4: mapping(address => Enterprise) public inStructs;
5: Poll public ePoll;
6: address payable public contractOwner;
7: Aion public aion;
8: uint public aionAmount;
```

5.3.3 The contract owner and access controls

As explained in chapter three, each contract has a constructor and in *EtherAd* it is responsible to initialize state variables. In particular, it specifies the address of the contract owner which is the one used by Truffle to deploy *EtherAd*. Moreover, the constructor creates an *Enterprise* associated to this address and inserts it directly in the group of registered members. In order to justify this choice, it is important to figure out who could actually commission this CMS for SMEs online aggregation.

The most reasonable promoter of such an initiative is the public administration that wants to improve and create growth opportunities for the local economy. Therefore, it may want to have peculiar and exclusive privileges on the platform even though the moderation policies are delegated to the community. For instance, only the owner of the contract should be allowed to withdraw coins from *EtherAd* balance. The presence of some coins can be due to donations performed by accounts to the contract or to the moderation policies (see next subsection). This money can be used by the public administration to fund this project of SMEs aggregation. Indeed, even though fundamental information are publicly available and permanently stored on Ethereum blockchain, the front-end assets of the application has to be always available in order to let users access the web interface of the platform and ease

its visibility. In the architecture shown in Figure 2.2, those files are provided via IPFS but, as said many times, it does not guarantee data persistence. Therefore, the only solution to comply with this issue is to host an always online IPFS node that serves those contents in the network. This responsibility should be in charge of the contract owner which can (partially) fund these expenditures with *EtherAd* balance. Moreover, the public administration may want the exclusive right to deactivate *EtherAd* contract. This can be performed implementing a function in *EtherAd* that calls *selfdestruct(address)* operation provided by Solidity. In particular, it transfers all Ether stored in the contract balance to a designated target and then the account's *storage* and *code* is removed from the (world) state.

Due to the importance of the contract owner, in *EtherAd* I defined not two, but three levels of accessibility.

- There are two functions whose execution can be triggered only by the contract owner. One is called *withdrawEth()* which transfers all the coins in *EtherAd* balance to the owner. This explains why the state variable that references the *address* of the public administration is defined as *payable*. The other is meant to call *selfdestruct(address)* and transfer all the coins to the owner itself.

- There are many functions whose execution can be triggered only by registered members of the community. For instance, those called by an enterprise to update its name, e-mail or the CID of the logo, but also the function that allows the publication of a new product on the platform. Other relevant ones in this category are those involved in the moderation policy and in the voting procedure. Notice that the access control over these functionalities was a clear requirement of the application. Obviously, the contract owner has access to all the mentioned contract functions since it is a member of the group.

- There are functions whose execution can be triggered by any Ethereum account. They are simple getters designed to retrieve information and show them in the web interface. To this category also belongs the unnamed fallback function of *EtherAd* contract which can be used to receive donations.

As already mentioned in chapter three, these access controls are performed with a combination of *modifiers* and *require(bool)* clauses. In particular, they rely on *msg.sender* in order to check the exact *address* of the account that called the function. Indeed, using *tx.origin* may cause vulnerabilities to *EtherAd*. For instance, consider an EOA registered in the group that calls a function on a malicious contract whose code triggers in turn the *EtherAd* function to update the logo of its enterprise. If the access control is based on *tx.origin*, the contract in the

middle can specify the new profile image of the enterprise. Here, it is important to remark that the group member may not be aware of what the contract code does, even though everything on Ethereum is public and visible in the (world) state. This is the reason why *msg.sender* is almost always the better choice. Listing 5.6 shows how this kind of access control is actually performed.

Listing 5.6 EtherAd access control example

```
1: function isRegistered(address a) public view returns(bool){
2:   if(eList.length == 0) return false;
3:   return (eList[eStructs[_a].eListPointer] == a);
4: }
5:
6: modifier onlyRegisteredEnterprises() {
7:   require(isRegistered(msg.sender));
8:   _; // Continue executing rest of function code
9: }
10:
11: function updateEProfilePicture(string memory _newIpfsHash)
12:     public onlyRegisteredEnterprises {
13:   eStructs[msg.sender].profileImageIPFSHash = _newIpfsHash;
14: }
```

These access conditions are verified by the miner that receives the message call transaction and decides whether to insert it in the next block or to drop it and revert its own state.

5.3.4 The voting procedure and moderation policies

The moderation of both contents and registered enterprises is probably the most important and challenging feature of the designed system. Indeed, while the integrity of the application's state is publicly guaranteed by Ethereum blockchain, the moderation still remains one of the main sources of entrepreneurs' skepticism. As already mentioned in the requirement analysis, the solution I designed relies on a majority vote procedure instead of a potentially biased authority. This is a simple idea but it leads to a win-win scenario. Indeed, the public administration is in favour of delegating the moderation of the platform because it is an additional cost to bear. At first, it should pay an operator responsible for these controls. Then, it has to fund these moderation procedures because they are message call transaction that use

Gas to modify the contract *storage*. Therefore, the voting strategy is convenient for the public administration, even though special rights should be reserved as last resort like the possibility to deactivate *EtherAd*. On the other hand, this solution assigns to each member of the community the responsibility to control what is published and who can post in the platform. This is exactly what entrepreneurs want: an unbiased procedure to measure consensus on a moderation decision. Obviously, it is in entrepreneurs' interest that the community respects some minimum standards, otherwise their brands will be damaged. Therefore, a voting procedure directly managed by the members of the group is a more effective and responsive solution to define and maintain those standards rather than an authoritative moderation.

As clearly stated in the requirement analysis, the operations that has to involve a poll are the acceptance of a candidate in the community of SMEs, the decision about the ban of a group member and the removal of a posted product. As consequence, I had to implement this voting procedure in *EtherAd* before the actual moderation functionalities. At first, I had to design the *Poll* struct, which is shown in Listing 5.3. An obvious feature of a poll is that it identifies a well defined period of time in which voters can express their opinion. Once the expiration time is reached, it is possible to count the votes and claim the winner. This was the biggest issue I had to face in this project because of the distributed nature of the blockchain.

In order to provide a more detailed description of the problem and a clear explanation on how I solved it, let me recall two fundamental fields of the *Poll*. The first one is the *status*, which identifies the poll as *EXPIRED* or *IN_PROGRESS*. Then, there is the *expirationTime* which is an integer value that specifies when it actually expires. In particular, once a group member triggers a voting procedure with a message call transaction, it implicitly sets the *expirationTime* of the poll as *block.timestamp* plus the duration of the poll expressed in seconds. The *block.timestamp* depends on the block in which the transaction is added by a miner, while the duration is hard coded in *EtherAd*. I set this value as 600 seconds (i.e. 10 minutes) for testing purposes, but in practice it should be bigger to provide a reasonable amount of time to vote. This point together with the quorum value should be studied more in the future. Anyway, this definition of *expirationTime* is reasonable since the timestamp of a block must be strictly larger than the timestamp of the previous one. Moreover, it should not be possible for any account to set the duration of a poll for two security reasons. The first one regards the fact that a malicious registered member may set expiration time far in the future. Since *EtherAd* has at most one active poll at a time, this attack prevents the possibility to perform other moderation procedures. The second reason is that the distribution of votes

is always visible in Ethereum (world) state, even though the contract may not expose them directly with a getter function. Therefore, it could be possible for the malicious member to change the expiration time of the poll as soon as it is satisfied of the result.

However, this acceptable definition of the *expirationTime* is not enough to solve all the issues of a poll. For instance, it identifies clearly a point in the future after which all transactions triggered to express a vote should be dropped by miners. On the other hand, it is still possible that a transaction meant to start a new poll is executed before the one meant to count the votes. This situation should be prevented, because it overwrites *ePoll* which is the unique *Poll* state variable (see Listing 5.5), thus nullifying the just expired poll before any moderation consequence is performed. Therefore, *EtherAd* has to implement access controls on functions that manipulate *ePoll* considering both the *msg.sender* and when the respective transactions can be included in a block. Before diving into this discussion, it is important to clearly identify the functions in focus.

- *vote(bool)* allows a registered member to vote. The resulting transaction increases *yesVotes* or *noVotes* in *ePoll* depending on the vote expressed by the user.
- *startPoll(args)* allows a registered member to start a poll. In particular, it provides some information like the *address* of the *Enterprise* in focus, the identifier of the *Product* in case it is the subject of the moderation procedure etc. As you can see in the *Poll* struct definition shown by Listing 5.3, all these details are maintained by *ePoll*. Moreover, this function is responsible to set the *status* of *ePoll* as *IN_PROGRESS* and the *expirationTime* of the new poll.
- *countVotes()* is triggered to count the votes and eventually update the state of the application depending on the result. Moreover, this function has to set the *status* of *ePoll* as *EXPIRED* and reset all its fields for a future poll. More details about this function will be explained later.

Now, *ePoll* can be in three different situations which depend on the combination of two factors. The first one is the value of its *status* field. The second factor is when a transaction is mined with respect to *expirationTime*. The overall access control over the previously described functions is implemented through *require(bool)* clauses and it is summed in the following list.

- *ePoll* is *in progress* if its *status* value is *IN_PROGRESS* and the transaction is meant to be in a block whose timestamp is lower than *expirationTime*. If the transaction references the *vote(bool)* function and it is generated by a registered member of the

group, it is executed correctly. On the other hand, all transactions meant to close the poll or start a new one are dropped by miners because the poll is running.

- *ePoll* is *active* if its *status* value is *IN_PROGRESS* and the transaction is meant to be in a block whose timestamp is bigger than *expirationTime*. If the transaction references the *vote(bool)* or *startPoll(args)* functions, it is dropped by miners independently on the sender. This is reasonable because *EtherAd* is waiting for a transaction that counts the votes, performs the moderation consequences and closes the poll.

- *ePoll* is *expired* if its *status* value is *EXPIRED* and the transaction is meant to be in a block whose timestamp is bigger than *expirationTime*. If the transaction references the *startPoll(bool)* function and it is generated by a registered member of the community, it is executed correctly. On the other hand, *EtherAd* blocks the access to both *vote(bool)* and *countVotes()* functions, because the poll is expired and already invalidated.

The hard coded definition of *expirationTime* and this access control based on the *ePoll* status prevents anyone to misbehave or any error due to the order in which transactions are mined. Indeed, there is no possibility to start a new poll while another one is *in progress* or *active*. Moreover, there is no chance to vote once the poll is *active* or *expired*. Finally, it is not possible for anyone to end the poll while it is *in progress* or *expired*. However, this is not enough because there is another way to attack the platform. Indeed, if none calls the *countVotes()* function, *ePoll* will remain in the *active* status and further moderation decision can not be performed. Therefore, there must be someone that performs the related transaction.

A good intuition is to define *countVotes()* as a *public* function. In this way, literally any Ethereum account can call it, but reasonably it is more likely to be executed by a member of the community since it is in their interest to moderate the platform and protect their brands. On the other hand, we have to consider the selfish behaviour of entrepreneurs and the fact that the account that calls *countVotes()* has to fund the transaction cost. This may lead to a situation where each registered member wants to close the moderation issue but it waits for another one to do it. In order to overcome this potential deadlock, a simple idea is to attribute the responsibility to call *countVotes()* to the registered member that started the poll. In fact, this is not a solution because none and nothing will ever guarantee for the reliability of this account. Therefore, in order to make this last solution feasible, there must be a mechanism to force the member to end the poll. For sake of simplicity, let me call *x* this registered account that started the poll.

The first idea is to leverage on the balance of the member to make it behave correctly. Consider, for instance, the case in which *x* had to transfer a big amount of Ether to *EtherAd*'s balance to start the poll, but with the promise of a complete restitution coded in *countVotes()*. This scenario is interesting because not performing the transaction to close the poll will cost a lot to a malicious account. Moreover, also in case *x* is disposed to lose money to attack the platform, it is still possible for another member to call *countVotes()*. As consequence, the obtained result is a big donation to the platform accessible by the contract owner. Even though this situation seems to be ideal, it presents some cons that convinced me to discard its implementation. At first, the amount of Ether to commit has to be big enough to discourage *x* to misbehave. However, not all members may have such an amount of money and this prevents some registered members to initialize a moderation issue which will be delegated to the ones that have more Ether. Moreover, entrepreneurs may be scared about this procedure, not mainly for the money exchange but for the risk that the contract owner itself steals the money while poll is *in progress*. The other issue regards a scenario in which *x* is the one under attack by an account in the community. Since the order in which transactions are mined depends on their *GasPrice*, an attacker could try to call *countVotes()* before *x*. In case it succeeds in this procedure, *x* can not have its money back otherwise the risk in case of a malicious *x* is nullified.

The second idea is to programmatically schedule the transaction to end the poll directly once *x* calls *startPoll(args)*. In this way, *countVotes()* is funded by *x* itself and it will be executed for sure. Therefore, this solution prevents any misbehaviour from *x* but also nullifies the risk of an attack to *x* from another group member. Indeed, the additional amount of money required by *x* is limited to afford the cost of the Gas used by the transaction for *countVotes()*. Therefore, if the attacker calls this function before *x*, simply it will pay the transaction instead of *x*. In fact, in this scenario it is not mandatory for *countVotes()* function to be declared as *public*, since it is *EtherAd* itself to call it. The problem with this solution is that it is not possible to be implemented due to the nature of smart contracts. As already mentioned in chapter two, they are programs stored in Ethereum (world) state whose code is executed only when miners receive the related request. Therefore, in order to call itself in the future, a smart contract should be run at that specific time. In this way, it can generate the transaction that triggers the execution of the scheduled function. However, this is impossible because there will be none supposed to run that contract at that specific time. This loop is the reason why every transaction is originated by an EOA, also in case of message call type which may involve multiple contracts in sequence. Therefore, the only way to schedule a transaction in

the future is to have an EOA that performs it at the desired time. So this idea does not solve anything since it brings back the original problem of who will call *countVotes()*.

However, since the problem of scheduled transactions is a known issue, there are services that provide this facility in a secure way. The one that I chose is Aion [61] because it requires a small fee and it is available for Ropsten test network. In my scenario, this scheduling mechanism performs a sequence of actions described in the following list.

1. *EtherAd* has to call the *scheduleCall(args)* function of Aion smart contract. Therefore, it creates an instance of such contract which is maintained as state variable (see Listing 5.5) and initialized in *EtherAd* constructor with the address of Aion contract deployed on Ropsten. The *scheduleCall(args)* function requires four main arguments. The first one is the time when the scheduled transaction should be created. In my scenario, it is specified by the value of *expirationTime*. The second one is the target of the transaction which is *EtherAd* itself. The third one specifies the *GasPrice* for the scheduled transaction. The final one is the function to be triggered which is *countVotes()*. Finally, with the transaction related to *scheduleCall(args)* I have to transfer some Ether to Aion contract in order to pay the fee for the service and the cost of the Gas used by the scheduled transaction.

2. If it is the first time for *EtherAd* to request a scheduled transaction with Aion, the *scheduleCall(args)* function creates a dedicated contract to deposit the funds transferred by *EtherAd* to schedule the transaction. Further requests will always involve this exact contract. The idea of using a contract as a deposit is a well designed security mechanism because it proves and allows anyone to verify that *EtherAd* money can not be touched by Aion for any other purpose rather than scheduling the desired transaction.

3. An off-chain process inspects the blockchain until a verified block whose timestamp is greater than *expirationTime* arrives. At this point, it automatically creates a transaction to Aion smart contract which in turn triggers the contract described in 2 to call the *countVotes()* function. If the amount of Ether transferred by *EtherAd* to schedule the transaction is not enough to fund the Gas used, the transaction is dropped by the miner and an out of Gas exception is raised. On the other hand, all the exceeding Ether are transferred back to *EtherAd* balance within the same transaction. The estimation of the Gas used for the scheduled transaction is performed in advance by Aion contract.

Relying on Aion makes the implementation of this second solution not only feasible, but also convenient. Indeed, *x* is the first responsible of the poll it started. It can not misbehave because *countVotes()* is scheduled and funded in advance. This last aspect can be easily

hard coded as an access control for the moderation function. Indeed, it is enough to inspect *msg.value* and if it is zero or less than the required amount for Aion procedure, the transaction to start a poll is dropped. Moreover, the risks taken by *x* are very limited because the amount of money involved is little, especially with respect to the solution that leverages *x* balance to make it behave correctly. The worst case scenarios of this implementation are mainly two. The first one regards a malicious member of the community that succeeds in calling *countVotes()* before the scheduled transaction is mined. This could possibly happen since the *GasPrice* of the scheduled transaction is hard coded in *EtherAd* and thus it is visible from Ethereum (world) state. The consequence is that the Ether allocated by *x* are blocked in the contract created by Aion. However, in order to keep them there, the attacker has to anticipate Aion for every future *countVotes()* call, otherwise all the money will be transferred to *EtherAd*. This makes the attack expensive in the long period, especially if the community continues to perform poll procedures. The second one regards the situation in which Aion does not perform the scheduled transaction due to any internal problem. Exactly like the previous case, all the money is still safe in the dedicated contract, so they can be refunded in the future and it is still possible for any Ethereum account to call *countVotes()* autonomously. This is due to the fact that it is mandatory to declare *countVotes()* as *public* in this implementation, otherwise Aion could not call it.

The overall voting functions are summed up in Listing 5.7, Listing 5.8 and Listing 5.9.

Listing 5.7 EtherAd pseudo-code for vote(bool) function

```
1: function vote(bool _vote) public
2:     onlyRegisteredEnterprises {
3:   require(isInProgress(ePoll));
4:   require(!hasVoted(msg.sender));
5:
6:   if(_vote){
7:     ePoll.yesVotes = ePoll.yesVotes + 1;
8:   } else{
9:     ePoll.noVotes = ePoll.noVotes + 1;
10:  }
11:
12:  // Keep track of voters
13:  ePoll.eBallotStatus.push(msg.sender);
14: }
```

Listing 5.8 EtherAd pseudo-code for startPoll(args) function

```
function startPoll(args) private {
  require(isExpired(ePoll));
  require(eList.length > 2); // At least three voters

  // Set ePoll fields passed as args to specify its subject
  ePoll.status = IN_PROGRESS;
  ePoll.expirationTime = block.timestamp + 600 * seconds;
  aion.scheduleCall.value(aionAmount)(ePoll.expirationTime,
    address(this), gasPrice, encode("countVotes()"));
}
```

Listing 5.9 EtherAd pseudo-code for countVotes() function

```
function countVotes() public {
  require(isActive(ePoll));

  uint tot = ePoll.yesVotes + ePoll.noVotes;
  bool approved = (tot >= (eList.length/2 + 1)) &&
    (ePoll.yesVotes >= (tot/2 + 1));

  if(approved) {
    if(ePoll.subject == PollSubject.BAN_POST) {
      removeProduct(ePoll.eAddress, ePoll.pID);
    } else if(ePoll.subject == PollSubject.BAN_ENTERPRISE){
      removeEnterprise(ePoll.eAddress);
    } else if(ePoll.subject == PollSubject.ADD_ENTERPRISE){
      addEnterprise(ePoll.eAddress);
    }
  }
  // Reset all ePoll fields
  ePoll.status = EXPIRED;
  if(inList.size > 0) {
    startPoll(args); // New poll to admit new member
  }
}
```

I actually did not include all the details and technicalities regarding their exact implementation in *EtherAd*. Indeed, I decided to structure this discussion so that it follows the way I approached the problem of the majority vote. In particular, I focused on the issues I faced, the solutions I designed and a precise evaluation of pros and cons for each one. However, I still have to explain how the actual moderation procedures are integrated with the described poll strategy.

As regards the decisions about whether to ban a registered member and remove a posted product, their implementation in *EtherAd* is very similar. Therefore, let me describe only the procedure that a registered user *x* has to follow in order to request a moderation decision on a content published on the platform. At first, it has to call *removeProductRequest(address, bytes)* and specify the *address* of the author and the identifier of the *Product* in focus. In particular, it is a *payable* function which performs two access controls. The first one is that the caller has to be a group member. The other one checks with *msg.value* that the transaction created by *x* sends to *EtherAd* balance at least *aionAmount* Ether. However, since I have to provide registered members the possibility to remove their own products without involving a poll, this transfer is not required in case the *Product* in focus is published by *x* itself. Notice that it is mandatory for *removeProductRequest(address, bytes)* to be defined as *payable* in any case. As regards the *aionAmount* parameter, it is set as state variable (see Listing 5.5) and defines the sum of Aion fee plus the cost to pay the Gas for *countVotes()* function. A part from the fixed *GasPrice* of the transaction which is hard coded in *startPoll(args)*, the value of *aionAmount* should depend on the involved operation because they consume different amounts of Gas. Moreover, if the poll does not approve the moderation issue, *countVotes()* may not perform any consequence (see Listing 5.9). Therefore, it is possible that some Gas is saved and *x* should be refunded for the exceeding Ether sent to *EtherAd* balance. However, given the fact that the public administration (i.e. the contract owner) has to fund the hosting of an IPFS node to make the web interface always accessible, I decided differently. Indeed, *aionAmount* is a fixed value which is enough to fund the most expensive moderation consequence. Moreover, in case there are some savings from *countVotes()*, they are not transferred back to *x*. In this way, the exceeding Ether can be used by contract owner to fund the hosting service since it is the only one allowed to withdraw from *EtherAd* balance. This procedure is safe since Aion refunds all unused Ether and defines a certain source of incomes in addition to casual donations to *EtherAd*. Moreover, the community of SMEs is also in favour of this tiny contribution because it is meant to sustain the visibility of the advertising platform. All these considerations about access controls can be extended also to

removeEnterpriseRequest(address) function and partially to *addEnterpriseRequest(args)*.

At this point, *removeProductRequest(address, bytes)* checks if there are at least three potential voters to start an eventual poll (see access control in Listing 5.8). In case there are only two elements in *eList*, the removal of a posted product is performed only if the *Product* in focus is published by *x* itself or if *x* is the contract owner. This is done internally by *removeProductRequest(address, bytes)* with the *private* function *removeProduct(address, bytes)*, whose role is to remove the *Product* instance from the *storage* field of *EtherAd* contract. Now, consider a scenario in which the minimum amount of voters to start a poll is satisfied. In case the *Product* to be banned is published by *x* itself, *removeProductRequest(address, bytes)* allows the direct removal as explained before only if *ePoll* is not *in progress* or if its subject is not the *Product* in focus. The last possibility is that *x* specifies a content posted by another registered member. Obviously, if the *ePoll* is not *expired*, the request is dropped as shown in Listing 5.8. Otherwise, *removeProductRequest(address, bytes)* calls internally *startPoll(args)* specifying the information regarding the goal of the poll and the product in focus. This is the reason why *startPoll()* is defined as *private* in Listing 5.8. Then, *countVotes()* is triggered by Aion and in case the moderation decision is approved, *removeProduct(address, bytes)* is finally called. As stated before, the function called to remove a registered member of the community follows the analog procedure.

On the other hand, the *addEnterpriseRequest(args)* is a little bit different. At first, in case there are less than three members in the group, the responsibility to insert a new member is directly given to the contract owner. A part from this specific situation, an enterprise that wants to enter the community has to be approved with a poll. However, if it performs the request while *ePoll* is not *expired*, its transaction has to be dropped. Therefore, in order to reduce the occurrences of this issue, I relied on *inList* and *inStructs* (see Listing 5.5). Once a candidate calls *addEnterpriseRequest(args)*, the function checks that the caller is not already member of the group, then it controls that the user transferred *aionAmount* Ether to the contract balance and inspects the size of *inList*. In case the buffer is completely filled, the transaction is dropped and the candidate has to repeat the request in the future. Otherwise, the function creates an *Enterprise* instance with all the information provided by the candidate. Then, this variable is added in *inStructs* and its *address* is inserted in *inList*. Now, in case the *ePoll* is *expired* and the candidate is the only element in *inList*, *addEnterpriseRequest(args)* behaves just like the other moderation functions. Indeed, it internally calls *startPoll()* and once it expires, *countVotes()* may call *addEnterprise(address)* depending on the voting result (see Listing 5.9). This *private* function moves the candidate from *inList* and *inStructs* to

the group of registered members which is maintained by *eList* and *eStructs*. On the other hand, if the poll is not *expired* or there are previous candidates to be processed, *addEnterpriseRequest(args)* stops its execution. In this last situation, someone has to trigger the admission poll for the first candidate in the *CircularBuffer*. As you can see from Listing 5.9, the responsibility of this issue is in charge of the Ethereum account that calls *countVotes()* function. The presence of Aion ensures that all requests arrived to *EtherAd* will be processed.

One of the potential risks of this implementation regards the fact that an attacker may continue to fill *inList* in order to monopolize all the polls and prevent other moderation decisions. However, it is important to notice that such attack is extremely expensive in the long period because every time the malicious account calls *addEnterpriseRequest(args)*, it has to fund *EtherAd* with *aionAmount* Ether. A major issue that could affect the application is the Sybil attack. Indeed, if a user controls the $50\% + 1$ of all registered members in the community, it can actually decide the results of all the polls for any moderation decision. This is an issue that *EtherAd* can not deal with because it can only guarantee not to have duplicate *address* in *eList*. In fact, the correspondence between each Ethereum account and a different person is an open problem. However, I designed *EtherAd* so that it is not possible to ban or modify the contract owner. In this way, it is always possible for the public administration to access *EtherAd* balance and eventually deactivate the contract.

5.4 Front-end implementation

Once the functionalities of *EtherAd* contract have been implemented in order to satisfy the requirements, I moved to design the web interface of the application. At first, I had to keep in mind that all front-end assets are meant to be published and distributed with IPFS as shown in Figure 2.2. This made me decide to design the advertising platform as a single-page application (SPA) in which the navigation of the user is actually performed by changing the content presented by the unique web page. In particular, this implies that all required code and libraries for the SPA has to be contained in one IPFS *tree* object. As already mentioned in the description of the project structure, the */dist* folder serves this exact purpose. However, its availability in the IPFS network has to be guaranteed, otherwise the visibility and the accessibility of the platform are compromised. As said in the previous section and in chapter two, I supposed that the one who committed the CMS addresses this responsibility by hosting an always online IPFS node with that specific object pinned in its *have_list*. Anyway, the choice of a SPA is almost mandatory due to the features of IPFS and its role in the designed

architecture. Consider, for instance a usual web application where the navigation process requires to load different pages from a web server. In my scenario, each of these pages should be a different IPFS object. Now, the simple navigation from the home page to the profile of the current user and backwards is impossible to solve with IPFS. Indeed, in the home page I can not specify the *href* attribute of any *anchor* element for the profile page and vice versa because both need the CID of the target which depends on its content. This problem is the equivalent of creating a loop in IPFS object DAG, which is acyclic because edges are stored in source nodes as the cryptographic hash (i.e. the CID) of the target object. On the other hand, in case the navigation of the user flows from the home page to the profile and it can not move backwards, it is possible to design the application with different IPFS objects because there are no loops. However, designing the platform as a SPA is clearly a better choice that gives me more freedom as a developer.

At this point, the previous considerations and the functional requirements gave me all the premises to design the web interface of the application. At first, I started with creating the mockups for the different contents to show in the SPA in order to have a visual representation of the final result. In particular, I decided to create a fixed *navbar* in the top and a menu on the left side in order to ease the navigation flow and the accessibility to stand alone functionalities provided by *EtherAd*. This basic structure is common to all mockups and it is implemented in the *index.html* file under the *src* folder of the project. More in detail, the navbar shows the application logo and the Ethereum address of the current visitor. On the other hand, the menu contains two toggle forms and several buttons. The first ones allows the user respectively to request its admission in the community by providing all required information and to donate Ether to the platform. Then, there is a button used by the contract owner to withdraw all the money from *EtherAd* balance. The others are used to trigger the related JavaScript functions in the *app.js* file that set the content of the main *div* element in the SPA. I designed four main mockups, one for each of the contents to show.

- The *home* content is meant to show the products posted by registered enterprises, which it is the main goal of the advertisement platform. However, since the number of published products will be reasonably bigger than the members of the community, I decided to present at first the enterprises with their information. Then, an user can inspect all the products posted by each of them in a second step with a toggle *div*. This is done in order to limit the loading time of the *home* content due to the procedure of image replicas implemented with IPFS.js. From this page it is possible to request the removal of a registered member and/or of a posted product. Moreover, the order in which the enterprises are presented in the *home* page is randomized. This satisfies the

non-functional requirement meant to provide unbiased visibility to registered members.

- The *profile* content is only visible if the current user is registered in *EtherAd*. At first, it is meant to show the information of the community member and it provides the functionalities to update them. From this page, it is possible to post a new product on the platform specifying its image, a brief description and its price. Then, the *profile* content lists all the products posted by the current enterprise. Here, it is possible to directly remove one of them without involving the poll procedure. Finally, in case the user notices that one or more of its images is no longer available on IPFS, it can re-upload the exact same files from this *profile* page without any cost since *EtherAd* will not be modified.

- The *poll* content is visible if the user is registered in *EtherAd* and if *ePoll* is *in progress*. In particular, it is meant to show the details of the current voting procedure like its goal and the subject of the moderation decision. From this page it is also possible to vote for the running poll.

- The *info* content is meant to provide a brief description of the software and a helper section to guide users. For instance, it explains that MetaMask extension is required and how to get Ether from its faucet.

The final result for the home content is shown in Figure 5.2. In particular, I linked Bootstrap v4.3.1 and other CSS files in the header of my *index.html* document in order to customize the style of the presentation logic.

During the architecture design in chapter two, I introduced IPFS.js since it solves and/or mitigates some issues. At first, it allows any visitor of the platform to run an IPFS node directly in its browser. In this way, each user is autonomous, since it can download the pictures of published products and the logos of registered companies through its own peer and without relying on a hard coded public gateway. This is important because if the selected one is no longer available, it is not possible to change it without modifying the CID of */dist* folder. Therefore, the visibility and the accessibility of the application is affected because the URL of the SPA will be different since it is */ipfs/<CID_of_dist_object>*. A possible work around to this issue is to rely on IPNS. In this way, the web application is accessible at */ipns/<PA_node_id>*, where this *mutable* address references the IPFS node hosted by the public administration which points to the last version of */dist* object. This can also be a good idea for future upgrades in the presentation logic of the application. However, the CMS does not rely on IPFS.js for casual visitors, but it is mainly meant to ease the entrepreneurs

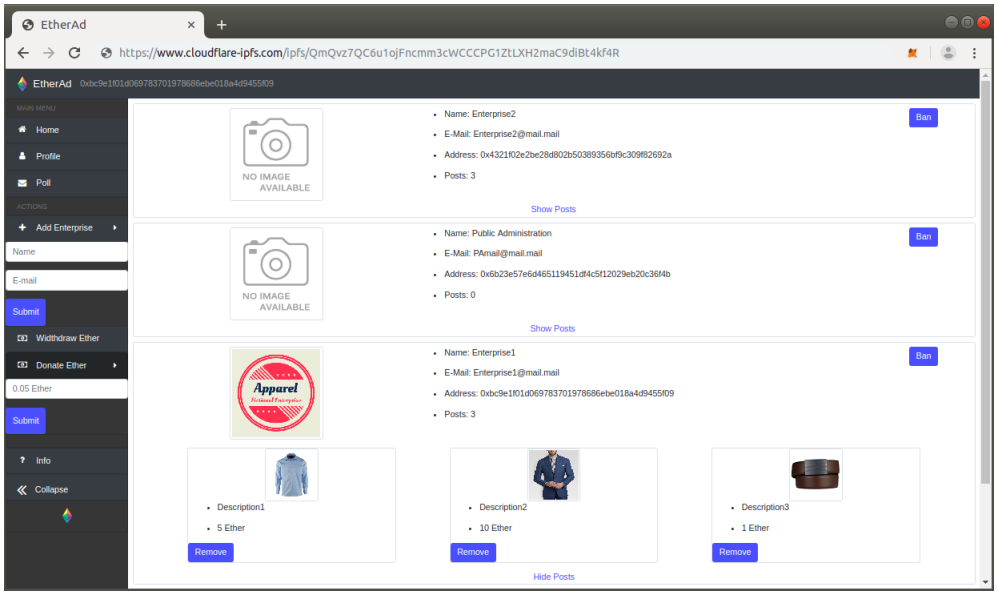

Figure 5.2: Home content of the SPA. This screenshot shows the disposition of registered enterprises and their products. Notice the URL of the page which shows that I relied on CloudFlare IPFS gateway to retrieve front-end assets.

registered on the platform. Indeed, as already mentioned in chapter two, it is not possible to upload contents on the IPFS peer-to-peer network through public gateways. Therefore, in order to update a picture or to publish a new one, each of the registered members in the community has to run an IPFS node. This is even more crucial given the fact that the only way for entrepreneurs to guarantee the availability of their contents is to have an always online peer that serves them on the network. In this scenario, IPFS.js is handy because it allows to avoid the installation of a node from scratch, which could be complicated for inexperienced entrepreneurs.

Since IPFS.js allows each visitor of the platform to run a peer, I can exploit this to mitigate the problem of data persistence with a replicas procedure not integrated in IPFS itself. This is important also because entrepreneurs may not have the expertise nor they want to afford the costs for hosting an IPFS node. The replicas policy I implemented in JavaScript is actually very simple. Indeed, once the user retrieves any CID from *EtherAd* contract via Web3.js, at first it has to download the image with the peer provided by IPFS.js. At this point, I simply hard coded the procedure to add and pin the content to the local node's *have_list*. This mechanism introduces some obvious delays, but it achieves its goal and it has some advantages. At first, it allows to actually spread the contents in the IPFS network, thus making the discovery and file transfer more efficient through the DHT. Finally, it provides more possibilities for

the local node to balance its share ratio with other peers. This is particularly important for casual visitors of the platform because they usually behave as leechers.

Since the interaction with *EtherAd* contract is performed via JavaScript with Web3.js, let me show how this is done in *app.js* which is included as a script in *index.html* after *truffle-contract.js* and *ipfs.js*. At first, I designed a JavaScript object called *App* which contains all the properties and methods required to manage the SPA. Once the page is loaded, I set up three main components:

- *App.web3Provider* is created from the *web3* object injected directly by MetaMask in the *window*. In particular, it specifies the provider used to connect with the selected Ethereum network. As already explained, the one referenced by MetaMask is Infura, but it is possible to configure it in order to use Ganache.

- *App.account* specifies the current Ethreum account logged in MetaMask. It is retrieved with *web3* as *web3.eth.accounts[0]* and its value is also shown in the top navbar of the SPA.

- *App.etherad* is the JavaScript object representing an instance of *EtherAd* contract. It is obtained using *truffle-contract.js* which has to parse the compiled contract. That's why *EtherAd.json* has also to be included in */dist* folder together with the files in */src*. The interaction between *App.etherad* and the Ropsten network where I actually deployed the contract is possible through *App.web3Provider*.

- *App.ipfs* represents the instance of an IPFS node created with IPFS.js constructor. At this point, the user has an online peer whose identifier can be inspected with the *App.ipfs.id()* asynchronous function.

In order to show how the procedure for image replicas and the interaction between web interface and *EtherAd* contract are performed, let me describe two operations.

The first one regards the presentation of the registered enterprises in the *home* content whose result is shown in Figure 5.2. It is performed by calling a method of the *App* object named *renderHomeContent()* which builds the HTML to be inserted in the SPA. Since this function has to download the profile images of the registered enterprises, at first it creates and shows a loading animation by calling the auxiliary function *setLoading(true)*. At this point, it performs a call to *EtherAd* contract to retrieve all the members of the community. Since *eStructs* is not iterable, at first I had to retrieve *eList* and then search each element in the mapping (see Listing 5.5). Notice that these operations do not modify the Ethereum (world) state, thus they are not transactions to be funded and their return values come directly by the

inspection of the blockchain stored by the Ethereum node the user is currently connected via Infura. At this point, considering the i-th *Enterprise* instance, it is possible to inspect the IPFS identifier of the *blob* object that represents its logo and download it with the local peer. If this operation succeeds, it is possible to add the file and make it available for other nodes in the network. On the other hand, a default image from the */imgs* folder is shown. Finally, *renderHomeContent()* builds an HTML template for the current enterprise and it is appended to the main *div* of the SPA. This procedure is shown in Listing 5.10 and the final result is the one in Figure 5.2.

Listing 5.10 Pseudo-code for App.renderHomeContent() function

```
renderHomeContent: async function() {
  App.setLoading(true);
  var contentDiv = $("#content");
  contentDiv.html("");

  const rEnterprises = await App.etherad.getEList();
  // Create a shuffled array to randomize the presentation
  for (var i = 0; i < shuffledArray.length; i++){
    const enterprise = await
    App.etherad.eStructs(rEnterprises[shuffledArray[i]]);
    var pp = "<img src=\"../imgs/No-Image-Icon.png\">";
    const ipfsCID = enterprise[3];
    if(ipfsCID != "" && ipfsCID != undefined) {
      const files = await App.ipfs.get(ipfsCID);
      if(files.length == 1 ) {
        await App.ipfs.add(files[0].content);
        pp = "<img src=\"" +
        files[0].content.toString("base64") + "\">";
      }
    }
    // Build the enterprise template
    contentDiv.append(enterpriseTemplate)
  }

  App.setLoading(false)
}
```

Listing 5.11 Pseudo-code for App.removePost(eIndex, pIndex) function

```
1: removePost: async function(eIndex, pIndex) {
2:   App.setLoading(true);
3:
4:   const isREnterprise = await
5:       App.etherad.isRegistered(App.account);
6:   if(!isREnterprise) {
7:     App.setLoading(false);
8:     return;
9:   }
10:
11:  // Controls omitted on input params
12:  const eAddress = await App.etherad.eList(eIndex);
13:  const pID = await
14:  App.etherad.getEPostID(eAddress, pIndex);
15:  const fee = await App.etherad.getAionamount();
16:  const isPollExpired = await App.etherad.isPollExpired();
17:  if(isPollExpired){
18:    await App.etherad.removeProductRequest(eAddress, pID,
19:    {from:App.account, to:App.etherad.address, value:fee});
20:  }
21:  else {
22:    console.log("Poll not expired, retry later");
23:    App.setLoading(false);
24:    return;
25:  }
26:
27:  App.setLoading(false)
28:  App.renderHomeContent();
29: }
```

As an example of the interaction between web interface and *EtherAd* contract, consider a registered member of the community that wants to remove a product published by another enterprise. This action can be done directly from the *home* content (see Figure 5.2) and the consequences from the contract point of view have been described in the previous section. Once the user clicks on the remove button, it triggers the *App.removePost(eIndex, pIndex)*

asynchronous function. At first, it performs a call to *EtherAd* contract to check if the current user is registered in the community and another one to verify whether *ePoll* is *expired*. In this case, the transaction that triggers the *removeProductRequest(address, bytes)* function can be performed. Let me point out a couple of things from Listing 5.11. At first, *eIndex* and *pIndex* represent respectively the index of the enterprise in *eList* and the one of the product in the *ePostList* field of the related *Enterprise* (see Listing 5.2). The actual product identifier is retrieved with the contract call at line 13. The second thing to notice is that I decided to implement again the same access controls on the poll. This is not mandatory since it is *EtherAd* that has to save the application state securely, but in this way I prevent users to experience errors thrown by MetaMask due to a dropped transaction by the miner. Finally, observe the format of the transaction to remove the product from *EtherAd* state. A part from the function arguments, I added an object that specifies the sender, the recipient and the amount of Ether involved in the transaction. The first two parameters are not mandatory, since each transaction is signed automatically by MetaMask using the private key of the current account and the default recipient is specified by the definition of *App.etherad*. The only mandatory field is the *value* because as stated in the previous section, the *removeProductRequest(address, bytes)* function has to be funded to afford the voting procedure. Indeed, MetaMask by default subtracts from the account's balance only the amount required to pay the Gas for the transaction. The code shown in Listing 5.11 is actually simplified from all the particular cases described in the previous section.

6

Conclusions and future works

In this work, I proposed to address the issues that prevent SMEs and micro firms from exploiting the opportunity of online aggregation for product advertisement. These obstacles can be summed up in three aspects. The first one regards the entrepreneurs' commitment as a community which is mined by the competition and their selfish behaviour. The second one is their skepticism to delegate the moderation of the community to a potentially biased authority. Finally, the lack of resources and expertise to run and maintain a centralized system which are relevant in case the initiator of the project is a micro firm or the public administration.

The CMS I designed and implemented presents a fully distributed architecture which is shown in Figure 5.1. Indeed, each enterprise in the community is the first responsible for the availability of their published contents in the InterPlanetary File System (IPFS) peer-to-peer network. Moreover, the business logic of the application is implemented in a decentralized fashion with the Ethereum blockchain. However, the current impossibility to guarantee data persistence in IPFS for the front-end assets of the web interface imposes the initiator of system to bear some hosting costs. The decision to rely on IPFS instead of a traditional web server is meant to show and implement exactly the same architecture in case it is available a technology that provides interoperability with Ethereum and includes an incentive mechanism to make peers store others' data in a secure and persistent way.

Then, I focused on mitigating the major sources of skepticism that mines entrepreneurs' trust to each other and to the common advertisement platform. In order to overcome this

situation, the software I designed heavily relies on the Ethereum blockchain which allows to maintain the state of the CMS in a verifiable and secure way. This guarantees entrepreneurs that all community members can not misbehave since all allowed operations are hard coded in the smart contract that manages the business logic of the system.

Finally, I faced the issue about the presence of an authoritative moderator that makes entrepreneurs suspicious about any decision to be biased. As a consequence, I addressed this problem by implementing a majority vote in the smart contract of the application. Indeed, each moderation decision triggers a poll that defines the consensus of the community on a specific subject. In this way, the entrepreneurs themselves are the ones that define the quality standards of the published contents and the initiator of the CMS (e.g. the public administration that wishes to support the aggregation of entrepreneurs) delegates these moderation costs to registered members.

In conclusion, my work provides a solid basis for a content management system that suits the needs of small aggregations of SMEs as it is fully distributed and without a central authority for content moderation, but there is still room for improvements.

The first one I envision would be to switch from IPFS to Swarm once its development status reaches a more stable version. With Swarm it should be possible not to force the initiator of the CMS to host an always online node, since the storage of all HTML, CSS and JavaScript files for the web interface can be funded with the exceeding of the moderation procedures plus an eventual fee payed by each community member. Moreover, entrepreneurs may decide to fund the persistent storage of their contents or maintain the current implementation where they have to run a peer to guarantee the availability of their contents. At the time of writing, Swarm seems the most interesting technology since its incentive mechanism will be based on Ethereum, thus its interoperability with the business logic of the application should be guaranteed.

Then, besides refining the quorum value and the duration of the poll, another issue that I did not tackle in chapter five regards the anonymity in the voting strategy. Indeed, since all transaction can be inspected in the blocks of the Ethereum blockchain, it is actually possible in the current implementation to retrieve what a target account has voted in a specific poll. This privacy concern is relevant for entrepreneurs due to possible rivalries between members of the community. This problem is an active topic in the literature and a very interesting study is proposed in [62]. In this paper, the authors describe their solution tested on Ethereum blockchain and the many technical difficulties they encountered. The most critical one is that the number of voters can be up to sixty due to the *gasLimit* value for a

block, which is set by the miners at 4.7 million Gas.

Finally, this platform can be extended not only to advertise products, but directly to sell them just like in online markets. The interaction between buyers, couriers and registered enterprises should be carefully managed via smart contracts to prevent frauds and malicious behaviours of one or more of the involved actors.

References

[1] K. L. R. A. R. D. F. M. M. R. S. Muller, Mattes, "Annual report on european smes 2017/2018," 2018.

[2] ——, "Italian sba fact sheet 2017/2018," 2018.

[3] [Online]. Available: https://github.com/ethereum/go-ethereum

[4] [Online]. Available: https://github.com/ipfs/ipfs

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

[6] Merkle, "Secrecy, authentication and public key systems: A certified digital signature." *Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University.*, 1979.

[7] A. B. A. M. M. D. Bano, Sonnino, "Sok: Consensus in the age of blockchains," *arXiv preprint arXiv:1711.03936*, 2017.

[8] P. Migliardi, Merlo, "On the feasibility of moderating a p2p cdn system: a proof-of-concept implementation," *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 2015.

[9] Freitas, "Twister: the development of a peer-to-peer microblogging platform," *International Journal of Parallel Emergent and Distributed Systems*, 2015.

[10] P. M. M. Prasad, Dantu, "A decentralized marketplace application on the ethereum blockchain," *IEEE 4th International Conference on Collaboration and Internet Computing*, 2018.

[11] [Online]. Available: https://github.com/bigchaindb/bigchaindb

[12] G. L. Xu, Song, "Building an ethereum and ipfs-based decentralized social network system," *IEEE 24th International Conference on Parallel and Distributed Systems*, 2018.

[13] H. Isaak, "User data privacy: Facebook, cambridge analytica, and privacy protection," *2018 IEEE Computer Society*, 2018.

[14] [Online]. Available: https://github.com/ethereum/web3.js

[15] [Online]. Available: https://github.com/ethersphere/swarm

[16] [Online]. Available: https://github.com/storj/storj

[17] [Online]. Available: https://gitlab.com/NebulousLabs/Sia

[18] [Online]. Available: https://github.com/filecoin-project/go-filecoin

[19] [Online]. Available: https://ipfs.github.io/public-gateway-checker

[20] [Online]. Available: https://github.com/MetaMask/metamask-extension

[21] [Online]. Available: https://github.com/INFURA

[22] [Online]. Available: https://github.com/brave/brave-browser

[23] [Online]. Available: https://github.com/ethereum/mist

[24] [Online]. Available: https://github.com/ipfs/js-ipfs

[25] Buterin, "A next generation smart contract and decentralized application platform," *Ethereum foundation*, 2014.

[26] Wood, "Ethereum: a secure decentralized generalized transaction ledger," *Ethereum foundation*, 2014.

[27] Szabo, "Smart contracts: Building blocks for digital markets," 1996.

[28] G. Buterin, "Casper the friendly finality gadget," *Ethereum foundation*, 2018.

[29] [Online]. Available: https://github.com/ethereum/cbc-casper

[30] B. Poon, "Plasma: Scalable autonomous smart contracts," *Ethereum foundation*, 2017.

[31] [Online]. Available: https://github.com/paritytech

[32] [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[33] Z. Wöhrer, "Smart contracts: Security patterns in the ethereum ecosystem and solidity," *2018 International Workshop on Blockchain Oriented Software Engineering*, 2018.

[34] Benet, "Ipfs - content addressed, versioned, p2p file system (draft 3)," *arXiv preprint arXiv:1407.3561*, 2014.

[35] M. Maymounkov, "Kademlia: A peer-to-peer information system based on the xor metric," *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, 2002.

[36] M. Baumgart, "S/kademlia: A practicable approach towards secure key-based routing," *2007 International Conference on Parallel and Distributed Systems*, 2007.

[37] Cohen, "Incentives build robustness in bittorrent," *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, 2003.

[38] K. Wang, "Measuring large-scale distributed systems: Case of bittorrent mainline dht," *IEEE International Conference on Peer-to-Peer Computing*, 2013.

[39] M. Legout, Urvoy-Keller, "Rarest first and choke algorithms are enough," *Internet Measurement Conference, ACM SIGCOMM conference*, 2006.

[40] [Online]. Available: https://github.com/libp2p/

[41] [Online]. Available: https://github.com/ipld/ipld

[42] [Online]. Available: https://github.com/ipfs/go-ipns

[43] [Online]. Available: https://github.com/multiformats/multihash

[44] Z. Y. L. Li, Xu, "Packet forwarding in named data networking: Requirements and survey of solutions," *IEEE Communications surveys and tutorials*, 2018.

[45] [Online]. Available: https://github.com/multiformats/multiaddr

[46] [Online]. Available: https://tools.ietf.org/html/rfc8445

[47] M. Freedman, Freudenthal, "Democratizing content publication with coral," *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, 2004.

[48] [Online]. Available: https://docs.ipfs.io/reference/api/cli/

[49] K. Mazieres, "Self-certifying file system," *PhD thesis*, 2000.

[50] [Online]. Available: https://github.com/trufflesuite/truffle

[51] [Online]. Available: https://github.com/nodejs/node

[52] [Online]. Available: https://github.com/trufflesuite/ganache

[53] [Online]. Available: https://github.com/trufflesuite/truffle/tree/master/packages/truffle-contract

[54] [Online]. Available: https://github.com/mochajs/mocha

[55] [Online]. Available: https://github.com/johnpapa/lite-server

[56] [Online]. Available: https://github.com/trufflesuite/truffle/tree/master/packages/truffle-hdwallet-provider

[57] [Online]. Available: https://github.com/motdotla/dotenv

[58] [Online]. Available: https://github.com/ethereum/solidity

[59] [Online]. Available: https://ropsten.etherscan.io/

[60] [Online]. Available: https://faucet.metamask.io/

[61] [Online]. Available: https://github.com/ETH-Pantheon/Aion/

[62] H. McCorry, Shahandashti, "A smart contract for boardroom voting with maximum voter privacy," *IACR Cryptology ePrint Archive*, 2017.

Acknowledgments

Scrivere quest'ultima pagina è veramente un insieme di emozioni indescrivibili: c'è soddisfazione per aver terminato un percorso di studi importante a cui ho dedicato tutto il mio impegno, c'è la consapevolezza che oggi si chiude un capitolo importante della mia vita ed infine l'entusiasmo per le nuove esperienze che mi aspettano. Desidero quindi ringraziare tutti coloro che mi sono stati vicino, mi hanno supportato e sopportato in questi anni sia all'università che al di fuori.

Ovviamente non posso che iniziare dai miei genitori, mamma Alba e papà Valter. Siete sempre stati presenti e mi avete incoraggiato in ogni mia scelta anche se vi sono costate più sacrifici di quanto mostrate. Oggi non avrei mai raggiunto questo risultato senza di voi, il vostro sostegno e il vostro affetto. Spero di avervi reso orgogliosi di me, vi voglio bene.

Voglio dedicare un pensiero anche a mio fratello Francesco. La tua allegria mi ha aiutato molto a distrarmi dallo stress e dalle difficoltà che ho incontrato in questi anni di ingegneria. Ora che hai intrapreso questo mio stesso percorso, spero di farti da spalla almeno tanto quanto tu lo sei stato per me.

Un grande grazie lo devo anche a tutti gli zii, zie, cugini e cugine che mi hanno pensato soprattutto durante queste ultime fatiche. In particolare, ai miei nonni Carlo, Luciana e Giovanna che hanno sempre creduto in me, esame dopo esame. Forse è anche grazie a tutte le vostre candele se ho raggiunto questo importante traguardo.

Poi ci sono tutti gli amici con cui ho affrontato questo percorso. A partire da Marco, il mio collega e compagno di corso con cui ho condiviso gioie, dolori, ma soprattutto tanti progetti e chiamate su Skype. Poi ringrazio Ludovica ed Andrea, i bioingegneri più simpatici del DEI con cui ho passato molte avventure e pause pranzo. Infine a Giovanni, Stefano e Sebastiano, gli amici dei tempi del liceo con cui trascorro molti sabati sera tra film discutibili e chiacchere in libertà.

Un ringraziamento finale lo dedico al prof. Migliardi per la sua disponibilità e la sua pazienza nel seguirmi in questo lavoro. Quando ho scelto questa tesi, l'ho vista come una sfida e un'opportunità per dimostrare a me stesso di essere in grado di gestire un progetto complesso in cui avrei dovuto imparare da solo molti argomenti. La consapevolezza che mi ha lasciato questa tesi è una spinta notevole verso il mondo del lavoro che mi aspetta.