SAPIENZA
UNIVERSITÀ DI ROMA

# An Ethereum-based system for resource ownership in data markets

Faculty of Information Engineering, Computer Science and Statistics
Master's Degree in Computer Science

**Davide Basile**
ID number 1810355

Advisor
Prof. Claudio Di Ciccio

Co-Advisor
Prof. Sabrina Kirrane

Academic Year 2021/2022

**An Ethereum-based system for resource ownership in data markets**
Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: davide.basile.9898@gmail.com

# Contents

# Chapter 1

# Introduction

Blockchain technologies are distributed systems which safely store information in sequential data structures called "blocks" [45]. Numerous studies show the effectiveness of such technologies in the fields of global supply chains, healthcare systems, finance, insurance, and digital rights management [43, 75, 76]. The high scalability and distributed nature of blockchains make these technologies highly suitable for big data contexts, in which, requirements for information integrity, availability and traceability are manifested [29]. The diffusion of social network applications and Internet of Things-based smart systems has exponentially increased volumes and values of personal data [62, 19]. As reported in [40], the big data field's revenue will reach $ 273.4 billion by 2026. Such a scenario highlights the need for safe data trading environments that guarantee the fair execution of the involved exchange mechanisms. Among the most popular blockchains, Ethereum has opened up a new phase for such technologies [9]. Ethereum supports the execution of distributed applications, called smart contracts, which are run by the nodes of the blockchain [8]. Smart contracts enable blockchain adoption in different areas.
The central theme of the thesis is the realisation of a decentralized data market for web resources based on the Ethereum platform. The infrastructure underlying the system is inspired by decentralized web initiatives such as Solid [58] and Digi.me[71], which aim to give data owners more control over their data. The system architecture is based on several enabler technologies: personal online datastores (pod), Trusted Execution Environments (TEE) and the Ethereum blockchain. Pods are technologies accessed via web services, controlled by data owners, that store and provide on-demand data. Pod managers are the software components of pods that are capable of processing resource requests and communicating with the on-chain infrastructure of the market. The Ethereum platform hosts the smart contracts implementing the logic of the market distributed application. Finally, TEEs allow data consumers to

get and utilize data. The architecture is built upon the HTTP protocol [1], which enables data delivery processes between data owners and data consumers, and the communication with the on-chain logic of the market for requests validation, resources initialization and subscriptions payment.

After the design stage, the functional requirements of the implementation have been defined. Among the different components to be implemented, this thesis focuses on the data owner perspective of the proposed architecture. The study and implementation of the data consuming perspective and the TEE components are covered by a separate study. The goal is to realize the technologies enabling users to share and sell personal resources in the market, while maintaining a high degree of control over their data. The infrastructure components considered by the implementation work of the thesis are pods and Ethereum smart contracts.

The first part of the implementation concerns the Ethereum ecosystem. This phase of the study addresses the on-chain components of the market, which are grouped into several modules. The tokenization module deals with the exchange mechanisms necessary to join the market. The indexing module contains the logic necessary for resource initialization. Information needed to index pods and market resources (e.g. web URLs, IDs, addresses) are stored and managed by the smart contracts of this module. Finally, the obligation module collects the smart contracts aimed at managing the usage limitations of the initialized resources of the market.

The second part of the implementation involves pod managers. According to the functional requirements, pods must guarantee the proper handling of HTTP requests, communication with the on-chain smart contracts, and the managing of the resources in the local filesystem. Initialized resources of pods are physically stored in a specific location of their filesystem. By interacting with pods, users can initialize or deactivate resources, add obligation rules, and monitor their compliance. Regarding data provision, pods deliver resources to data consumers through HTTP responses. However, an authentication mechanism is necessary to identify data consumers and determine their rights to the requested resource. The encryption methodologies provided by the Ethereum platform have been used in this regard.

The last part of the thesis focuses on qualitative and quantitative evaluations of the implementation work. Through the functional testing, it is verified that the pod manager prototype and the smart contracts fulfil the specification of the requirements defined during the design phase. The quantitative evaluation involves costs and code analysis. The execution of on-chain code collected in smart contracts requires the payment of a tax measured in Gas, whose amount is calculated according to the computational effort of the code to be executed. The state of the art provides several

[1]https://www.w3.org/Protocols/

tools for the automated cost evaluation of smart contracts [4, 26, 65]. The purpose of this assessment is to delineate the expenditures for infrastructure maintenance and resource control procedures related to data owners. The identification of critical issues in this regard highlights opportunities for future code improvement and optimization.

## 1.1 Structure of the document

Chapter 2 of the document focuses on the thesis background and defines its core concepts such as privacy in big data topics, data trading, blockchain technologies and decentralized web initiatives. Subsequently, chapter 3 proposes a motivating scenario for the DecentralTrading application, which contextualises the market's main functionalities. Chapter 4 exposes the design choices and architectural components involved in the market, by making clear the distinction between data owners and data consumers. The implementation stage is addressed by chapter 5, which describes code details related to the on-chain infrastructure and pod managers. The resulting implementations are evaluated in terms of costs in chapter 6. Finally, chapter 7 concludes the thesis and outlines the future work.

# Chapter 2

# Background and state of the art

## 2.1 Big Data and privacy preservation

Big data refers to huge, diverse, and complex data collections that are challenging to store, analyze, and visualize [52]. The big data phenomenon can be expressed in terms of three main key-descriptors. Variety indicates the type of data sources involved in the phenomenon (e.g., structured, semi-structured, or unstructured data). Velocity describes the frequency with which information are generated (e.g., streams, real time, nearly real time, and batch). Volume expresses the range and scale of the phenomenon (terabyte, petabyte, exabyte and zettabyte). The big data industry is continuously growing. According to estimates [72], $2^{18}$ around bytes of data are generated each day across all industries. One of the fundamental big data challenges involve privacy [74]. In the era of big data, the readily available current technologies and techniques, such as search engines, social networks and hacking tools constitute a significant threat to individual privacy. More specifically, the main danger to today's privacy protection comes from data mining and machine learning algorithms, which are able to extract knowledge from large datasets. Such techniques extremely ease profiling, clustering and non-obvious relationship awareness findings [38, 68]. Studies examining privacy in big data contexts are numerous. The most common approach for protecting privacy is still encryption. Asymmetric encryption [56] can be extremely useful to restrict data access to users provided with proper public and private keys. Attribute based encryption proposes a methodology that generates secret keys based on descriptive attributes (e.g. IDs, dates, locations) [25]. Homomorphic encryption is the research branch aimed at operating on encrypted data by producing consistent and meaningful outputs [51]. Systems dealing with big data contexts must guarantee the fair management of confidential user data. Proper solutions to this end are typically integrated into the system design. Access control and usage control are two of the most popular approaches to managing data confidentiality. The following paragraphs set out their principles.

### 2.1.1 Access control

Access control is a security approach that limits who or what can access resources in a computing environment [55]. In system infrastructures, access control is dependent upon and coexists alongside other security services. Such technologies require the presence of a trusted reference entity that mediates any attempted access to confidential resources. In order to decide who has rights on specific resources, access control frameworks make use of authorization rules, typically stored inside the system. A set of rules constitutes a policy. The literature distinguishes three main types of access control policies. Discretionary policies control access to information according to a user's identity and authorizations [36]. In this case, rules define the access modes (such as read, write, or execute) that each user is permitted to have on each object in the system. Mandatory policies control data access by classifying users and protected resources according to their security level [37]. If the user's level matches the security level of the resource, access is granted. Finally, role-based policies limit users' access depending on the actions they take in the system. A role may be described as a set of duties and obligations connected to certain working activities [54]. A popular approach to implementing access policies is through Access Control Lists (ACLs). Each protected resource has an associated ACL file, which lists the rights each subject in the system is allowed to use to access objects. Access control is a topic that has generated much interest in the scientific community. Tran et al. [66] propose a solution to integrate access control into an untrusted Peer-to-Peer (P2P) environment. The work of Ouaddah et al. [49] leverage the distributed nature of blockchain technologies to implement an access control framework for Internet of Things contexts. Koshutanski et al. focused on business processes by designing an access control architecture for Web Services [32].

### 2.1.2 Usage Control

Modern privacy challenges require solutions that can not be properly offered by traditional access control. Novel security requirements are brought by the progress of computer systems, necessitating the use of new safety protocols [1]. Usage control is an extension of access control whose policies take into account obligations and conditions besides authorizations [34]. Authorization predicates define limitations that consider user's and resource's attributes. Obligations are constraints that must be fulfilled by users before, during, or after resource usage. Conditions are environmental rules to be satisfied before or during usage. Mutability of attributes and access continuity are two innovative ways that usage control improves on conventional access control models. Usage control frameworks must guarantee the proper handling of attribute alteration. Similarly, because of the rules' components that must be fulfilled after resource access, continuity is a key feature of usage control technologies. The most notorious usage control model is UCON$_{ABC}$ [50]. The model represents rules by defining specific rights (e.g. operations to be executed) related to sets of subjects (e.g. users who want to perform an operation), objects (e.g. the resource to operate), authorizations, obligations and conditions. UCON$_{ABC}$ supports the definition of mutable subject and object attributes, which increases the descriptive depth of the model. A usage control framework should require a proper infrastructure to guarantee policies' enforcement and monitoring ,in order to detect misconducts and execute compensation actions (e.g. penalties and right revocations). The state of the art offers interesting cases of study that approach the technique. UCIoT [33] is a framework to enforce usage control policies in IoT environments. UCIoT provides the features of a U-XACML-based usage control framework on a distributed, peer-to-peer (P2P), decentralised architecture. Similarly, the study carried out by La Marra et al. is a research concerning usage control in industrial IoT scenarios [39]. The work interconnects a set of usage control systems through distributed hash tables. Carniani et al. shift their focus to cloud computing environments, designing a framework for usage policy enforcement in the cloud platform OpenNebula [11].

## 2.2 Data trading: context and opportunities

The exponential growth in the amount of big data combined with improvements in analytical techniques based on machine learning and AI methodologies has led to great interest for large firms in daily user information [12]. Business organizations and companies realized the prominence of data resources in today's ultra competitive environment and initiated major investments in information systems and data warehouses aimed at improving the strategic, tactical and operational sphere of a firm [28]. According to [20] the 92% of the considered firms agree that they need to increase the use of outside data. The large amount of information required by businesses is fundamental to implement decision making processes by means of business intelligence techniques [46]. Due to the large and growing demand, data become assets endowed with considerable value [60]. Such a scenario opens up the opportunity to monetize information through exchange procedures that allow individuals and companies to sell and buy data. To this end, the need for proper structures to ensure data integrity, coordinate exchange processes and enforce data owners and consumers' rights is manifested [61].

Data Marketplaces are platforms which offer the right environment to enable the selling and purchasing of data sets and data streams [59]. The most popular taxonomy regarding data trading systems provides a categorization based on the information origin [16]:

- Personal data marketplace
- Business to Business (B2B) data marketplace
- Sensor/IoT data marketplace

Personal data marketplaces allow individuals to sell their own daily information (e.g. location, shopping preferences, everyday habits). B2B data marketplaces involve the information exchange between different companies. Finally, IoT data marketplaces allow IoT devices to automate the selling of collected data. The purchase of data in a marketplace can take place according to different models. According to the one-off purchase model, data consumers pay for specific data resources at a fixed price. The subscription model allows buyers to access the data for a specific period of time based on the subscription duration. In the on-demand model, data consumers pay for the data as and when they need it.

The world of data marketplaces is extremely composite and the innovative initiatives are numerous. Datarade is a B2B data trading system, which counts more than 1,000 commercial providers [17]. The commerce platform on which the system is based is called Data Commerce Cloud. It enables companies to buy and sell data across the globe. The Snowflake platform is a B2B cloud computing solution whose architecture is composed of three layers: the cloud services layer, the compute layer, and the data storage layer [57]. Similarly, Amazon Web Services (AWS) Data Exchange is a service which allows AWS customers to subscribe to and use third-party data in the Amazon Web Service cloud infrastructure [3].

Recent developments in the blockchain field have had important implications for the data marketplace scene. Several features of such technologies are perfectly matched to the needs of data trading scenarios [5]. Blockchain technologies enable, by definition, the immutable recording of transactions generated in a data marketplace. Moreover, the majority of available blockchains allow distributed applications implementing the logic of the market to be run by the blockchain infrastructure. The distributed nature and the cryptographic methodologies of such technologies increase the robustness and reliability that a marketplace application must guarantee [48]. The literature offers several studies in which blockchains are used for data trading purposes. Chuen et al. [13] propose a blockchain based architecture for vehicle data selling . Hu et al. [30] employ the computational power of Smart Contracts to store big data resources and regulate exchange processes, which are endorsed in combination with an off-chain key management entity. Similarly, Xiong et al. [73] designed a challenge response mechanism for a data market scenario, based on the conjunction of smart contracts technologies and Machine Learning algorithms. The Datum initiative proposes a decentralized and distributed NOSQL database backed by a blockchain ledger, which permit users to safely store, share and sell personal data from social networks, smart homes and IoT devices [27]. Its architecture involves the physical storage of data resources in the underlying blockchain. Another interesting solution is proposed by the IOTA Data Marketplace [31], which leverages an architecture based on the combination of a centralized cloud backend for data purchase and consumption, and a blockchain infrastructure for data storage and submission. Ultimately, despite the heterogeneity of the examined solutions, they all involve no direct communication between data owners and data consumers, with the blockchain infrastructure serving as middleware between the two actors.

## 2.3 Blockchains and Ethereum

Blockchains are distributed data structures through which huge amounts of data are safely stored and shared between users [45]. The key distinguishing features of such technologies are the distribution and immutability of information inside the system. The information sharing takes place via a distributed ledger in which data is grouped into sequential structures called "blocks". The structure's sequentiality is achieved as each block contains a reference to the previous one. Due to their distributed nature, blockchains must guarantee information consistency between the nodes. This issue, which can be extended to the entire family of distributed systems, is addressed by the use of shared protocols based on consensus algorithms [42]. Despite the multitude of consensus algorithms, consistency in distributed environments is still a subject of study, due to the lack of efficient solutions in terms of performance and energy consumption [35].

Undoubtedly, the notoriety of these technologies lies in their application to financial economics. The global scale spread of BitCoin, ideated by the pseudonym Satoshi Nakamoto, established the standards for such technologies, which were later used in the development of numerous other blockchains [45]. Nakamoto sensed that the distribution and immutability of information would provide suitable technological foundations for the development of a currency exchange system. In such a scenario, a blockchain is used as a shared immutable ledger, through which users can send and receive values. Transactions are the sending of value from a sender user to a recipient user. When a new transaction is submitted to the system, it is placed, after a validation procedure, along with other transactions, in a new block that is appended to the end of the data structure. Each blockchain account is provided with a unique public key (also called address) and a private key that are exploited for identification purposes and to sign transactions via asymmetric key encryption. Due to the cryptographic methodologies involved in the exchange process, blockchains' currencies are named cryptocurrencies.

Since the Bitcoin explosion, numerous research projects and initiatives regarding this area have given shape to several technologically diversified blockchain platforms. Access control, consensus, network type and throughput are some of the main taxonomy criteria [63]. During the last decade, blockchains gained recognition among wider audiences, and applications based on these technologies have strongly increased. The study proposed by Vadgama et al. [67] counts 271 new blockchain-based projects born between 2010 and 2020. The use case contexts in which blockchains are involved are numerous. Global supply chains management, healthcare systems, finance, insurance and digital right management are some of them [43, 75, 76].

### 2.3.1 Ethereum and smart contracts

Ethereum is the decentralized platform that has opened a new era in the blockchain realm. Its large-scale adoption ushered in the beginning of second-generation blockchains. The declared aim stated by its founder Vitalik Buterin was to combine the existing coin exchange functionality , introduced by Bitcoin, with the opportunity to build Distributed Applications (Dapps) that run on the same blockchain network [64]. Although the initial idea was to provide tools to automate and facilitate payment processing, the freedom given to programmers makes the possibilities for development virtually endless [44].

The enabler technology that allows Dapps to be executed is the Ethereum Virtual Machine (EVM) [9]. It is a software distributed across all the blockchain nodes, which, through virtualization and emulation procedures, is able to provide a computing environment. Smart contracts are the most innovative and differentiating element of Ethereum [41]. They collect immutable and deterministic code that is executed by the EVM. Smart contracts appear as classes, equipped with state variables and methods. Ethereum users can write their own smart contracts and deploy them into the network. Once a smart contract is deployed, it is related to a unique Etherum address, thanks to which users can invoke its execution. The most commonly used programming language for smart contracts is Solidity [1], a Turing-complete language, capable of performing all kinds of computations.

[1]https://docs.soliditylang.org/en/v0.8.17/

The cryptocurrency underlying Ethereum is called Ether (ETH). The sending and receiving of ETH is performed through transactions. Each transaction involves the payment of a tax that is charged to the sender. Transaction fees are measured in Gas, whose price expresses the equivalent of a single unit in ETH [10]. Transactions are extremely relevant for smart contract interactions too. When a method invocation causes the state of a smart contract to change (e.g. the change in the value of a variable), a new transaction is generated. The Gas amount of such a transaction is proportional with respect to the computational effort of the code to be executed. The higher the computational burden, the greater the amount of Gas. Differently, the invocation of a method that does not cause the smart contract state to change (e.g. the reading of a variable) does not generate any transaction. Therefore, the invocation does not involve any fee. Regarding consensus, the first algorithm used in the Etherum network is Proof of Work (PoW). PoW assumes that each node, called miners, votes with his "computing power" by solving hard computational problems through brute force approaches. However, consumption, scalability and performance issues with the algorithm have directed the Ethereum foundation to move to the more sustainable Proof of Stake (PoS) [23]. In PoS, participant nodes with higher coin ages have higher chances of being selected for the validation procedure [47]. The transition process of the Ethereum main-net to the PoS begun in December 2020 and will be completed in 2023 [22].

## 2.4 Decentralized web

Since its development, the Web has steadily evolved into an ecosystem of large dimensions, in which the vast preponderance of personal information is controlled by few mega-platforms that coordinate the majority of online activities. In July 2022, 4.70 billion people were using social media globally, making up 59.0 percent of the world's population, according to Data Reportal [24]. Web centralization and the development of large platforms have undoubtedly brought several benefits. By making web interactions more usable and friendly, it facilitated access for the general public to important research and information tools. However, a small number of stakeholders wind up having disproportionate influence on the content that the public can produce and consume. The scale of the phenomenon emphasises the need for initiatives aimed at safeguarding users' rights in such a scenario. In order to decentralize the Web and take personal users' information out of the hands of a select few firms, proponents of decentralized systems suggest related technology. Nowadays, the state of the art proposes numerous solutions for web decentralization.

### 2.4.1 Solid

Solid[2], led by the inventor of the World Wide Web, Tim Barnes-Lee, aims to significantly change how web applications operate today, bringing actual data ownership and increased privacy. The Solid protocol is build upon W3C[3], Semantic Web and RDF standards [7]. The central idea of the project is to let users store their data in personal servers called pods, rather than entrusting their data to centralized applications. Pods are web-accessible storage systems that can be set up on private servers owned and operated by users, as well as on public servers owned and operated by pod providers [53]. By making Solid applications read and write data directly from pods, the need to store confidential data in application infrastructures is thus avoided. To work properly, applications can aggregate data from multiple pods spread across the world. Solid applications operate data via REST-ful HTTP methods. Authentication methods are necessary to identify individuals and applications which want to access specific user information. Solid proposes an efficient and secure method of decentralised authentication on the Web called WebID-TLS protocol [70]. A WebID is a unique identifier for web agents(e.g. individuals, coorporate ,applications) [69]. It takes the form of an HTTP URI pointing to a document that contains the necessary information for authentication. Pod services enabling the generation and storage of such documents are called identity providers, and they are queried during authentication processes. The Solid protocol preserves data confidentiality through access control methodologies. Pods decide whether to grant data operations via Access Control List (ACL) files. These documents specify the permitted actions on specific data resources for different web agents (identified by their WebID).

### 2.4.2 Digi.me

Digi.me [71] is a new ethical and sustainable approach for users to take ownership of their data and share it discreetly with data-driven apps and services. The distributed user-centric design of the project transfers power to users by facilitating the private sharing of personal data with apps and businesses that adhere to the protocol. The Digi.me architecture is composed of three main components. The client app allows users to control and safely share information. The cloud data storage holds and groups data spread over user's online services (e.g. social media, finance, health). Dropbox, Google Drive and Microsoft OneDrive are the cloud data storages supported by digi.me. Finally the cloud platform coordinates the processes between the client application, the cloud data storage, and third-party

[2]https://solid.mit.edu
[3]https://www.w3.org

online services. Through the cloud platform, the client application can add data into the cloud data storage from the web services and then safely share information, maintaining a high degree of control.

### 2.4.3 InterPlanetary File System

The InterPlanetary File System (IPFS) is a distributed peer-to-peer filesystem that aims to link all computing devices with a shared filesystem [6]. IPFS makes an important contribution by streamlining, developing, and integrating verified methods into a single, cohesive system. The distributed structure of the platform is composed of numerous nodes, which, as a whole, constitute a P2P network [15]. Nodes connect and transmit to each other objects representing files and other shared data structures. Despite each node stores specific files, all the nodes in the network can access other nodes' files. Nodes are identified by a node id, which is a public-key cryptographic hash. The hundreds of nodes in the IPFS network regularly exchange messages with other nodes over the internet. A routing system is necessary in order to find other peers' network addresses and peers related to particular resource objects. To this end, a Distributed Hash Table (DHT) is employed [14]. By swapping blocks with peers via the BitSwap protocol, IPFS distributes files [18]. In BitSwap, peers have a list of blocks they want to buy (a want list) and a list of blocks they are willing to trade for (a have list). A credit-like mechanism encourages nodes to seed even when they have no specific need for it. Indeed, they could have the blocks that others are looking for. According to this, BitSwap nodes optimistically broadcast blocks to their peers in the hopes that the debt will be paid. Peers keep track of the amount of confirmed bytes they have with other nodes. As a result, they transmit blocks to peers who are debtors probabilistically, through a function that falls as debt increases.

# Chapter 3

# Use Case Scenario

The goal of this chapter is to provide an overview of the use case scenario that motivates our work. The system, named DecentralTrading market, is a distributed and decentralized application that enables users to sell and consume web data resources. The market is backed by the Ethereum blockchain, which hosts its logic via smart contracts. The main functions of the system are described below without going into technical details, that will be addressed in the next chapters.

## 3.1 DecentralTrading Market

A new decentralized data market called DecentralTrading aims to facilitate data access across decentralized web data stores. Bob and Alice sign up for the DecentralTrading market. They both have their own Ethereum credentials (public and private key), which are enough to join the application. By spending Ether(ETH), Alice and Bob buy a DTsubscription, uniquely related to their Ethereum accounts, which allows them to enter the market. Bob is interested in sharing his resources in exchange for remuneration. Differently, Alice would like to access specific resources in the market. In order to share his resources Bob sets up a personal online datastore (pod) service on a machine of his choosing and initialize the data that he would like to trade into the market. The market enables the employment of rules on resources to set usage restrictions on shared data (e.g., medical datasets may only be used for medical purposes, internet-browsing datasets must be deleted after one week). Indeed, Bob decides to make its data available only for medical purposes and sends metadata with respect to the resources that he would like to trade and the associated rules to the DecentralTrading market. Alice is interested in Bob's medical dataset. She asks the DecentralTrading market for a reference to the data and generates a certificate that proves her identity and that she has paid the subscription. Alice uses the reference to contact Bob's personal data store, which checks if the certificate is valid. Thereafter, it returns Bob's medical dataset and the associated rules. Alice can only use the data obtained from the market on their trusted devices (which is part of the terms and conditions stipulated by the DecentralTrading market), which ensures that Alice only uses the data for medical purposes. At any point in time, Bob can ask the DecentralTrading market to check that his resource rules are being adhered to.

## 3.2 Joining the market

The only currency considered by the DecentralTrading market is the DTtoken, a digital asset created for the Ethereum platform. Users who want to enter the market spend ETH to buy DTtokens. The exchange rate ETH/DTtoken is publicly accessible on the Ethereum network. DTtokens are accumulated in Ethereum accounts and used to buy rights to access the market. Such rights are represented by means of DTsubscriptions. A DTsubscription collects several information, such as an identifier, the coverage period, and the owner's address of the subscription. By purchasing DTsubscriptions, users have the right to use the market's resources and get rewards for sharing their data, for the specified period of time. Data owners earn DTokens according to the amount of access to their resources. The greater the numbers of accesses, the greater their remuneration will be. In this way, data providers are expected to be able to recover the initial expense of market subscription.

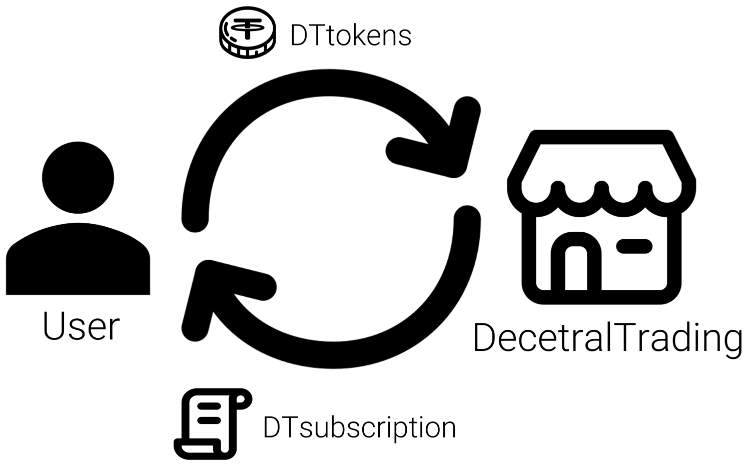


**Figure 3.1.** Users spend DTtokens to purchase DTsubscriptions

## 3.3 Resource sharing

DecentralTrading provides distinct functionalities for data owners and data consumers. Resource sharing is the core function of the system. In order to sell their resources, data owners use the Pod manager software implementing a web service that is able to provide on-demand data according to the market logic. Through such software, users are allowed to initialize data sources and data into the market. Once a resource is initialized, it can be requested by data consumers having an active DTsubscription. All the initialized resources can be removed at any time. Pod managers enable the definition of rules called obligations, regarding the proper use of the accessed resource. Three different types of obligations have been instantiated. Temporal obligations express the maximum time a resource can be held in data consumers' devices. Access counter obligations put a limit to the number of accesses to the resource in the utilizers' devices. Domain obligations define which kinds of applications can use the resource. Finally country obligations restrict resource usage to specific countries.

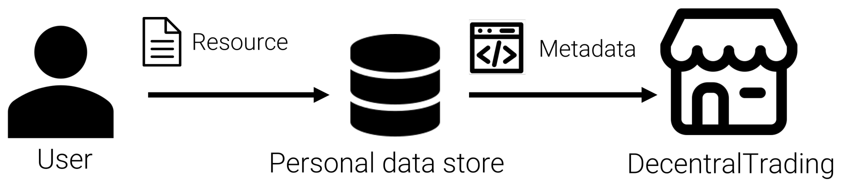


**Figure 3.2.** Resources sharing

## 3.4 Resource usage

The functionality related to resource utilization aims at providing data consumers tools to properly retrieve data from the market, ensuring providers fairness on resource usage. To this end, trusted client software is employed. The software allows data consumers to request data by providing evidence of their identity and subscriptions, according to the market protocols. Once data is retrieved, data consumers must interact with the trusted client, which manages the resource utilization. Trusted clients are in charge of the log-keeping functionality. Indeed, data consumers must be able to provide data owners with records attesting to the fulfilment of obligation rules.

# Chapter 4

# Architecture and design of the market

In order to contextualise the previously proposed use case scenario, the following chapter offers the architecture underlying the market. The goal of this part of the study is to clarify all the involved technologies and their roles in the architecture. After an overall description of the big picture of the market, the focus is brought to services related to on-chain market logic. The last two sections of the chapter will cover the components for data ownership and data consumption.

## 4.1 The DecentralTrading infrastructure

The DecentralTrading market's architecture is built upon Web-related standards whose principles are combined and adapted to the purposes of the application. Two key objectives were pursued during the design and implementation of the system:

**(a)** to produce a data trading system designed for resources (e.g., files, datasets, pages) shared and retrieved through Web protocols.

**(b)** to instantiate a shared resource control protocol that would protect data providers over the use of their data.

The choice of the components characterizing the architecture was carefully selected according to the two above requirements. The architecture component's of DecentralTrading are categorized in three groups of technologies.

- Market logic technologies
- Data owner technologies
- Data consumer technologies

Market logic technologies deals with the infrastructure of the market. The Ethereum blockchain is employed in this regard. Data owner technologies enable the safe sharing of resources in the market. To this end, pod services are used. Finally, data consumer technologies are custom client software, through which users access and use market's resources. We propose a Trusted Execution Environment solution in this regard. In the following subsections, the principles and functionalities of the three components are presented.

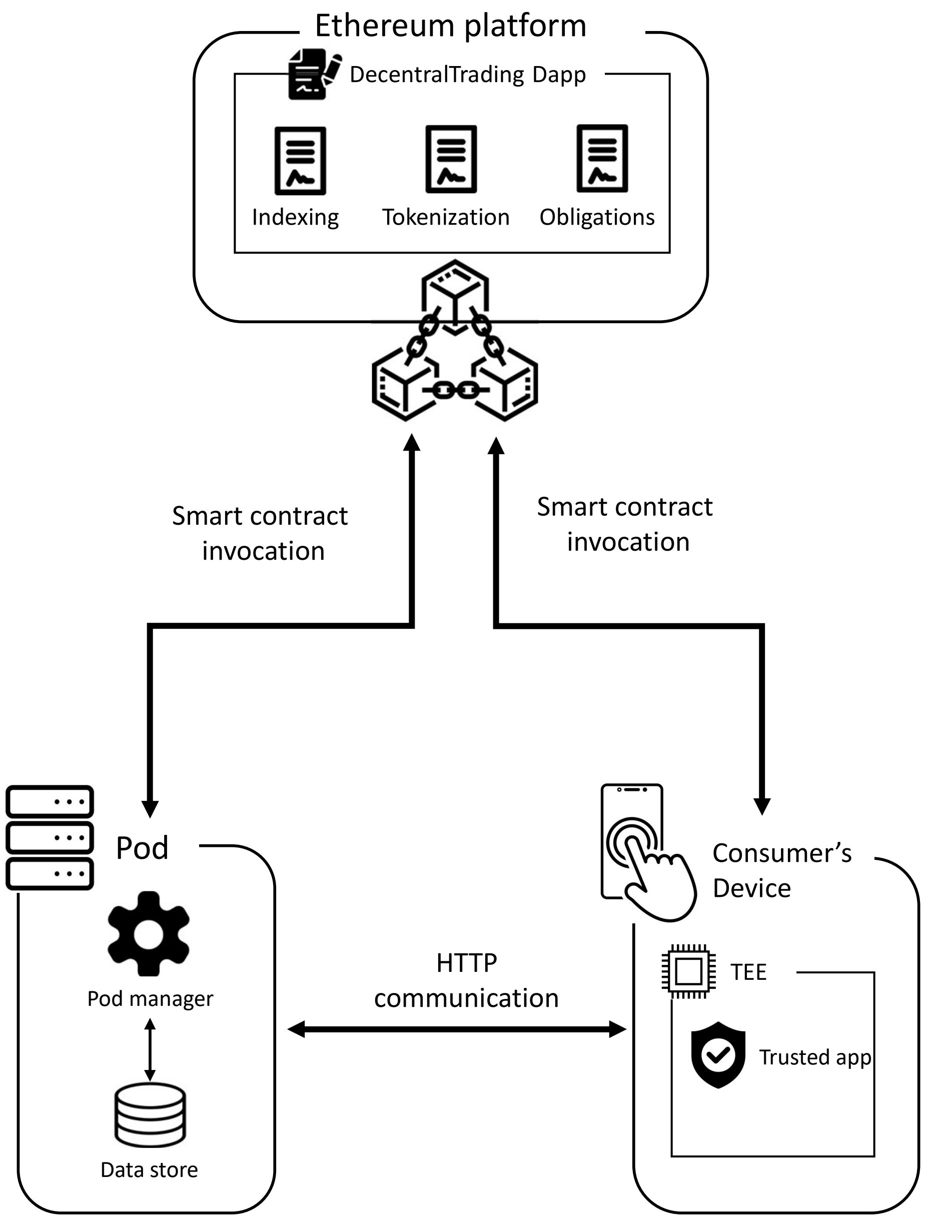


**Figure 4.1.** Architecture diagram of the DecentralTrading application.

### 4.1.1 Ethereum and the on-chain infrastructure

The Ethereum blockchain platform is the core technology of DecentralTrading. Its adoption provides a trusted and fully decentralized infrastructure capable of validating and supervising all operations occurring in the market. Ethereum hosts the market functionalities through multiple interconnected smart contracts , whose methods are directly invoked by users. A dedicated smart contract has been designed for each functionality. The adoption of the blockchain requires all the users of the market to have Etherum credentials (public address and private key), in order to authenticate and sign transactions.
The on-chain infrastructure must guarantee the proper handling of the DTtoken currency, i.e., a custom digital asset that is used for payment purposes. The ledger functionalities of Ethereum are extremely useful for managing DTtoken transactions. The Ethereum platform is involved in the generation and storage of subscriptions (named DTsubscriptions) that prove users' rights to access the market.
Another fundamental task addressed by smart contracts is indexing. Resources that are shared in the market must be properly registered and equipped with unique identifiers. The Web URI of resources is related to initialized data, which, alongside additional metadata coming from initialization, is managed through a specific smart contract.
Resources' utilization rules specified by data providers are managed by autonomous smart contracts. The system architecture requires each pod service to be related to a specific smart contract, implementing the logic for loading and changing the utilization rules of stored resources. Through Ethereum authentication, it is ensured that functions of such a smart contract are invoked only by the uniquely related pod service.

### 4.1.2 Pods and data owners

Undoubtedly, the inspiration from decentralized web initiatives has been central in the market design process. Among the several decentralized web projects considered, Solid proposes features and principles that are highly suitable for DecentralTrading's goals. The main idea behind Solid is to move users' data from few and centralized Web application servers to personal data store named pods. Pods are accessed via web services and are directly controlled by users, which provide on-demand resources via HTTP. Solid applications request data directly from pod services, leading to a complete decentralization of the Web.
Pods' principles can be extremely useful in a data trading scenario, being adopted for the fulfilment of (a). Such technologies offer data providers controlled environments to store and share their resources with the market. Pods are able to communicate

with DecentralTrading's smart contracts in order to accomplish its functionalities. To this end, Ethereum credentials are associated with each pod, to enable such components to authenticate and submit transactions for smart contract interaction. Pod technologies physically control market resources in data providers' filesystems. This includes the implementation of protection mechanisms over initialized data. Indeed, resources can be directly manipulated only through pod technologies, installed on the data owner's machine. Encryption and firewalling can be extremely useful in this regard.

By interacting with pods, data owners share their resources through initialization procedure. To this end, pods communicate with the on-chain component of the market and submit resource metadata (e.g. the resource Web URI). Once a resource is initialized by the pod, it can be accessed by other users having market rights. Pods allow data owners to submit and modify utilization rules on their resources, which are stored in Ethereum smart contracts.

Data consumers retrieve resources by contacting pods through HTTP requests. Pods are able to authenticate such requests with the use of the blockchain infrastructure. The validation procedure covers two key aspects. The first is related to the identity of the sender. In addition, it is verified that the submitted subscription is valid and covers the necessary rights to access the resource. Through Ethereum authentication, it is ensured that functions of such a smart contract are invoked only by the uniquely related pod service.

### 4.1.3 Trusted Execution Environments and data consumers

In order to guarantee (b), dedicated client technologies are involved in the market architecture. The objective is to instantiate architecture components to enable adoption of a usage control-inspired solution. Users who want to retrieve data from the market make use of trusted applications running in Trusted Execution Environments. TEEs are employed to guarantee the fulfilment of the rules imposed by data owners over their resources. Trusted applications running in TEEs communicate with the on-chain infrastructure of the market to retrieve resources' metadata (e.g., the web URI, references, rules). In order to physically obtain a resource, a trusted client must send an HTTP request to the owner's pod service. After proper verification involving the Ethereum infrastructure, the pod can decide to send the resource in the HTTP response. Once a resource is accessed, the TEE enables data fair usage according to the rules specified by the owner. Details related to TEEs are not addressed by this thesis and are covered by a separate study.

## 4.2 On-chain and off-chain communication

A crucial aspect to be addressed during the design phase regards the communication between the on-chain infrastructure of the market and the off-chain components. Pod managers and trusted applications are coordinated by Ethereum-based services and need to continuously exchange messages with on-chain smart contracts that constitute the DecentralTrading application. The possible approaches that can be adopted to the communication between off-chain and on-chain components strongly affect the architecture of the framework. The following subsection will show three different solutions and their effects in the DecentralTrading application.

### 4.2.1 Direct communication model

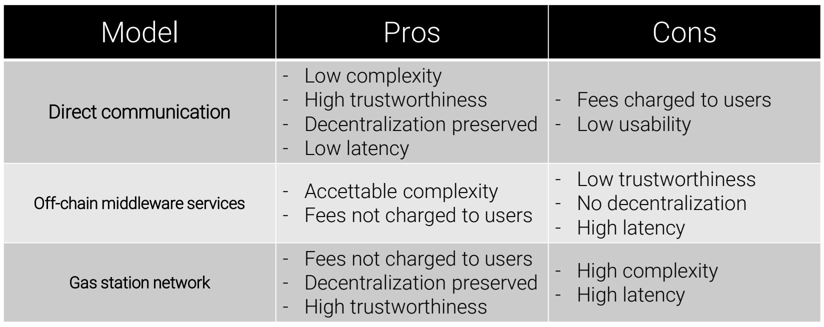
The simpler solution assumes off-chain and on-chain entities communicate directly, without intermediary services. Trusted applications and pods are equipped with Ethereum credentials to sign transactions and generate the execution of on-chain code. Once a transaction is signed, it is submitted to Ethereum by third party providers (e.g., Infura, Alchemy and Moralis) or by the off-chain component itself. The last case requires pods and trusted applications to locally run Ethereum nodes. Such a scenario leads to a complete decentralization of the architecture. Indeed, decentralized client devices interact with the distributed infrastructure of the blockchain. The solution involves Ethereum accounts related to off-chain components to pay for the costs of transactions they generate. According to this, fees related to smart contract execution are charged to the users. The biggest advantage this architecture brings is high trustworthiness. Indeed, off-chain components are in close contact with the on-chain infrastructure, and they are therefore responsible for the events that take place in it. A complete decentralization of the architecture decreases the risks of scalability and availability issues since all possible bottlenecks have been addressed. Moreover, the approach proposes the simplest possible solution, reducing the overall complexity of the architecture. Unfortunately, the model is not free of critical issues. The biggest criticality concerns usability. Paying user fees severely diminishes the attractiveness of the project. Payback mechanisms are needed in order to mitigate the issue.

### 4.2.2 Middleware services model

The adoption of off-chain middleware services can represent a solution to the user's fees issue. This option involves off-chain servers that run Ethereum nodes that are equipped with reserved credentials. The goal of such web services is to intercept off-chain component requests and provide backend logic for pods and trusted applications. Once a request is received by off-chain servers, backend logic is eventually executed and a new transaction is generated to communicate with the on-chain smart contracts of the market. By using their own credentials, web services pay for users' transactions, and usage costs are attributed to the infrastructure of the market. Moreover, the solution provides a better separation between front-end and back-end functionalities for the off-chain elements of the architecture, keeping the complexity of the architecture acceptable. However, the employment of centralized web services running Ethereum nodes can have negative effects on availability. Such services can represent a vulnerability in this regard, and the bottleneck issue may occur. Moreover, while indirect communication with the blockchain may have positive effects on costs, it can negatively impact the trustworthiness of the application. Indeed, such a solution hides the blockchain infrastructure from pod technologies and trusted applications, which have no way to directly interact with the market ecosystem.

### 4.2.3 Gas station network model

Through a Gas station network solution, it is possible to introduce off-chain intermediate services between pods/trusted applications and Ethereum without running into the centralization issue and charging fees to users. An Ethereum transaction can be signed and sent without the original sender (the end-users) having to pay for gas, using a decentralized network of relayers called Gas station network. The off-chain components of the architecture sign a message containing details about a transaction they would like to execute and transmit it to the decentralized network of relayer servers. Such messages are called meta transactions. Relay servers will validate a transaction after receiving a request to relay it from the off-chain components. Then, the relayer submits a native Ethereum transaction to the mempool, signs it, and sends it back to the client for verification. The client can simply select a different relay server and attempt to send a transaction if something goes wrong. A Gas station network involves on-chain entities as well. A RelayHub contract is the on-chain reference point for relay servers to submit transactions. A Paymaster contract keeps an ETH balance in the RelayHub and has the capability to execute any business logic to approve or disapprove the code execution (and the related expense). This feature can be employed to verify if a DTsubscription is related to the off-chain component that wants to interact with the on-chain infrastructure of the market. Finally, a forwarder contract is in charge of generating the code execution of the market smart contracts, once all checks are successful. Market's smart contracts must implement specific interfaces so that their execution can only be invoked by the forwarder contract. The decentralized structure of the relayers network ensures decentralization. The Gas station network approach has considerable effects on usability by providing a mechanism for the on-chain infrastructure interaction without the need for immediate payment of execution fees. Although they do not interact directly with market's smart contracts, pods and trusted applications are fully aware of the code executed on Ethereum. This ensures a high degree of trustworthiness. The major issue related to such a solution involves the increased complexity of the architecture. The inclusion of a larger number of components has negative performance effects on throughput, latency and costs.



| Model | Pros | Cons |
|---|---|---|
| Direct communication | - Low complexity<br>- High trustworthiness<br>- Decentralization preserved<br>- Low latency | - Fees charged to users<br>- Low usability |
| Off-chain middleware services | - Accettable complexity<br>- Fees not charged to users | - Low trustworthiness<br>- No decentralization<br>- High latency |
| Gas station network | - Fees not charged to users<br>- Decentralization preserved<br>- High trustworthiness | - High complexity<br>- High latency |

**Figure 4.2.** Pros and cons for the proposed communication models.

## 4.3 Components coordination

Several basic interaction scenarios have been identified and grouped in order to describe how the components of the architecture cooperate. In the following subsections, the main functionalities of the system are addressed, focusing on the coordination between pods, trusted applications and the on-chain infrastructure of the market.

### 4.3.1 Pod initiation

The process starts when data owners make a request to their pod manager to initialise a new DecentralTrading pod. Users must provide their Ethereum credentials to the pod manager. The pod manager sets up the pod in the local filesystem, by defining a new address and default obligation rules. Subsequently, a transaction is signed by the pod manager using user's credentials and information about the pod's metadata(e.g., the Web URI and the default policy) is sent to the Ethereum blockchain. The on-chain infrastructure of the market registers the new pod and deploys the smart contract for the obligation rules. Finally, the pod manager receives back the pod's id and its obligations smart contract address.

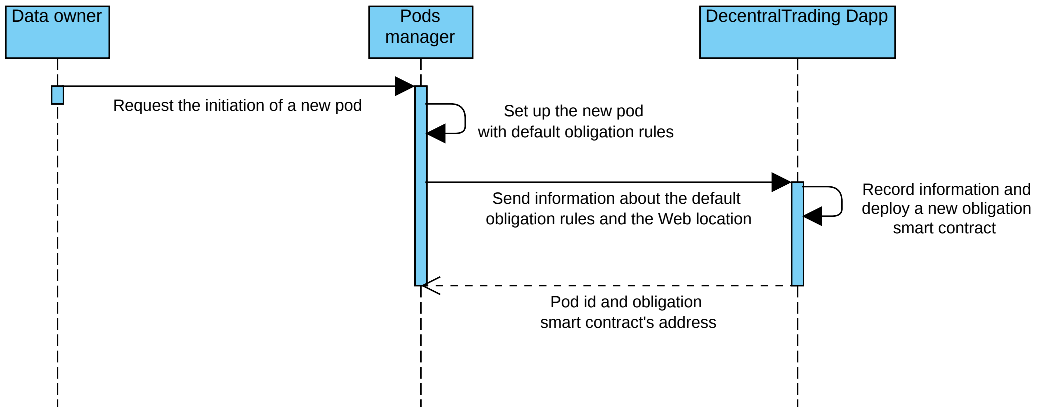

Data owner
Pods manager
DecentralTrading Dapp
Request the initiation of a new pod
Set up the new pod with default obligation rules
Send information about the default obligation rules and the Web location
Record information and deploy a new obligation smart contract
Pod id and obligation smart contract's address

**Figure 4.3.** UML sequence diagram for the pod initiation process.

### 4.3.2 Resource initiation

Data owners add data to their pods by interacting with the pod manager. The resource initiation process enables the adding of a new resource to the Dencentra-Trading app. The process is initiated when the data provider asks the pod manager to add a resource to his pod. The pod manager first adds a copy of the resource in the pod's location of the filesystem. Subsequently, the pod manager uses the Ethereum credentials to forward, via transaction, the necessary metadata (i.e., a reference to the resource) to the DecentralTrading app, which adds the resource to the index. Once the resource is indexed, it can be accessed by the data consumers of the market.

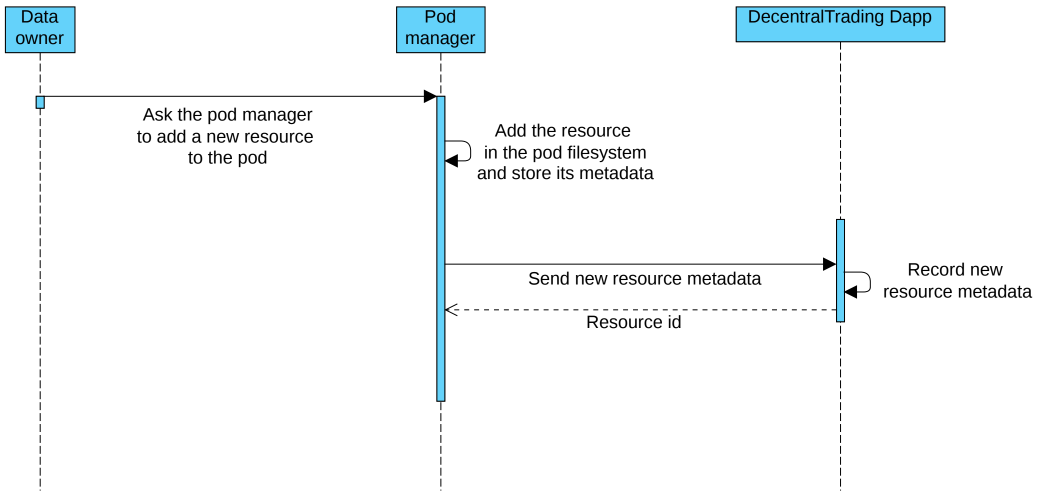

Data owner
Pod manager
DecentralTrading Dapp
Ask the pod manager to add a new resource to the pod
Add the resource in the pod filesystem and store its metadata
Send new resource metadata
Record new resource metadata
Resource id

**Figure 4.4.** UML sequence diagram for the resource initiation process.

### 4.3.3 Resource indexing

The resource indexing process involves data consumers, and allows them to retrieve a link to a resource that is indexed in the DecentralTrading app. Given that data consumers may not know the exact location of a resource they are interested in, they asks the on-chain infrastructure for a Web URI. The process is initiated when a data consumer requests information about one or more resources initialised in the market. The trusted application from which the data owner makes the request, runs in a trusted execution environment, and read the resource information directly from the DecentralTrading on-chain infrastructure. This information is subsequently stored in the data owner's trusted execution environment.

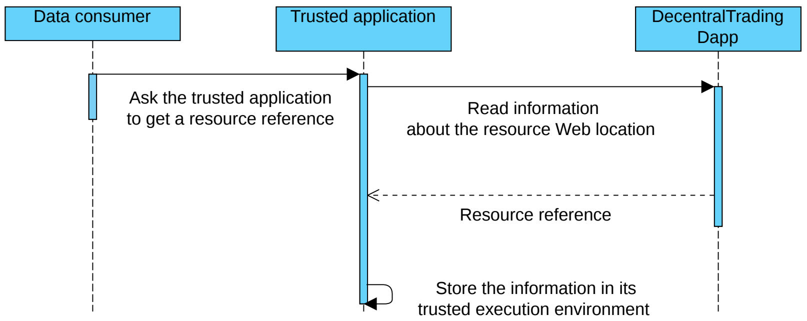

Data consumer
Trusted application
DecentralTrading Dapp
Ask the trusted application to get a resource reference
Read information about the resource Web location
Resource reference
Store the information in its trusted execution environment

**Figure 4.5.** UML sequence diagram for the resource indexing process.

### 4.3.4 Resource access

The resource access process is used to retrieve a resource from a DecentralTrading pod. In order to access a resource, data consumers' trusted applications, running in a trusted execution environment, make a request to the pod manager, which includes a certificate that proves users have paid the market fee. The trusted application contacts the pod manager using a reference, previously obtained via resource indexing process. The pod manager first checks the identity of the data consumer through the authentication mechanism. Once the identity is verified, the pod manager contacts the on-chain infrastructure to check the validity of the certificate. If so, the pod manager sends back the resource to the trusted application, which stores it in its trusted data storage.

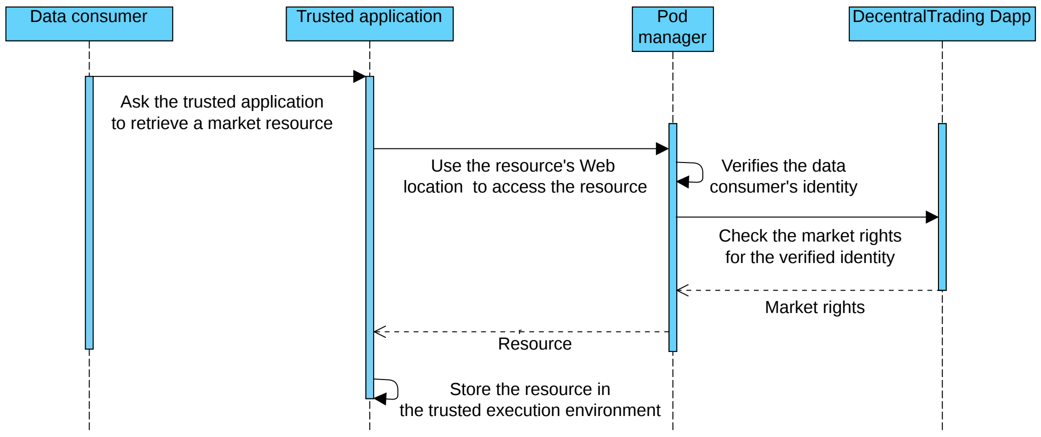

Data consumer
Trusted application
Pod manager
DecentralTrading Dapp
Ask the trusted application to retrieve a market resource
Use the resource's Web location to access the resource
Verifies the data consumer's identity
Check the market rights for the verified identity
Market rights
Resource
Store the resource in the trusted execution environment

**Figure 4.6.** UML sequence diagram for the resource access process.

## 4.3.5 Obligation modification

The policy modification process updates obligation rules in the DecentralTrading app. For instance, data owners may decide to change the purpose for which a particular resource or pods may be used. Here we assume that such updates are permitted according to the general rules of the market. The data owner makes a request to his pod manager to change an obligation rule. The pod manager changes the rule locally and uses the pods's credentials to send the modification to the on-chain infrastructure of the market. The old rule is overwritten. The DecentalTrading app notifies every data consumer that has a copy of the involved resources that the rule has been updated. Trusted applications update their local rule, check if any action needs to be executed locally, and if so, execute the required action.

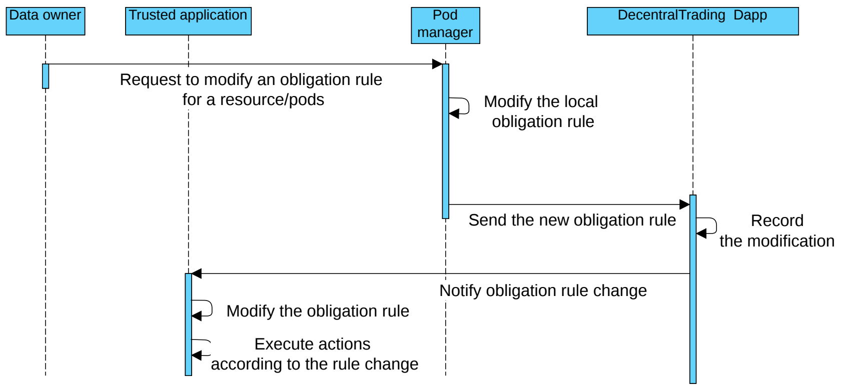

Data owner
Trusted application
Pod manager
DecentralTrading Dapp
Request to modify an obligation rule for a resource/pods
Modify the local obligation rule
Send the new obligation rule
Record the modification
Notify obligation rule change
Modify the obligation rule
Execute actions according to the rule change

**Figure 4.7.** UML sequence diagram for the obligation modification process.

### 4.3.6 Obligations monitoring

The policy monitoring process is used in order to continuously check if obligations rules are being adhered to. The pod manager initiate the monitoring (for instance via a scheduled job) and forwards the request to the on-chain infrastructure. The DecentralTrading app in turn communicates with all devices that have a copy of the resource in their trusted execution environment, and requests evidence that the usage policies are being adhered to. The trusted applications running in a trusted execution environment send back the evidence to the on-chain infrastructure, which performs verifications and communicates the result to the pod manager that initiated the monitoring process.

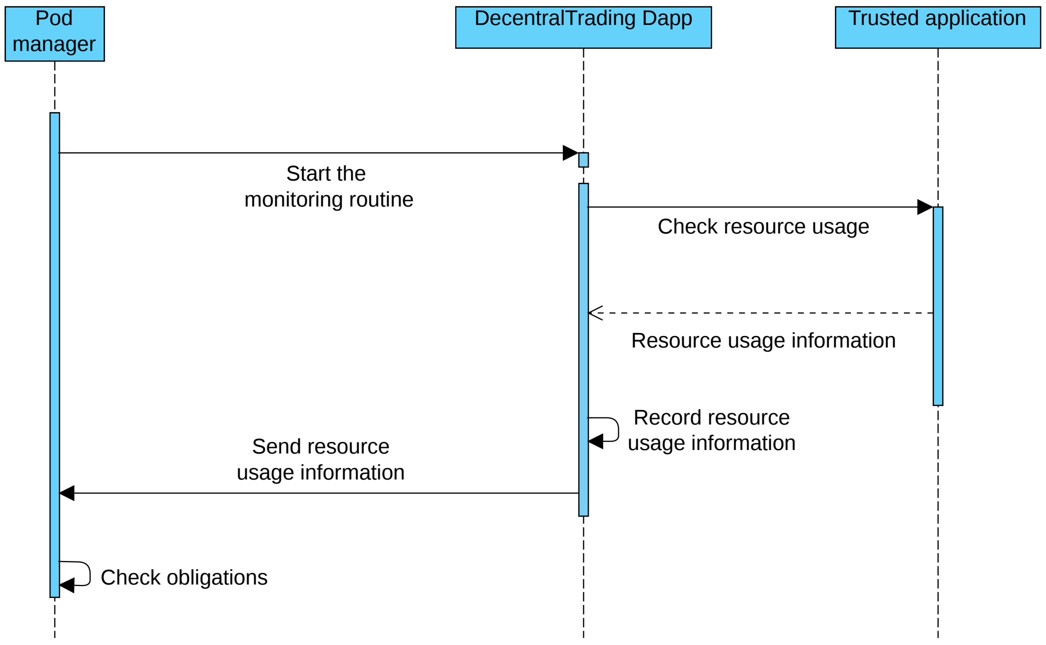

Pod manager
DecentralTrading Dapp
Trusted application
Start the monitoring routine
Check resource usage
Resource usage information
Record resource usage information
Send resource usage information
Check obligations

**Figure 4.8.** UML sequence diagram for the obligations monitoring process.

# Chapter 5

# Implementation

Once the design of DecentralTrading has been defined, we moved to the implementation of the architecture's components. The thesis addresses the implementation, focusing on the data ownership perspective and the on-chain infrastructure. The goal of this part of the study is to produce smart contract prototypes implementing the market logic and a proof of concept for the pod manager. The design and the implementation of data consumption technologies involving Trusted Execution Environments and trusted applications are covered by a separate study.
The first part of the implementation concerns the realization of the Ethereum application's components, which are grouped into three key modules: the tokenization, the indexing and the obligations modules. Utilities, implementation details and code structure are clarified for each of these. After that, the discussion broadens to the pod manager's proof of concept. The key subjects that are addressed by this part concern details on the physical structure of pods, how market resources and obligation rules are managed by the pod manager application and the implementation of data provision procedures.
The goal of the chapter is to discuss in a concrete way what was designed in the architecture chapter and how the components considered were implemented. Among all the possible design solutions concerning the on-chain/off-chain communication presented in Section 4.2, the model chosen is the direct communication model (Section 4.2.1). The design assumes the off-chain components run the backend logic locally and directly send transactions to the blockchain by using their Ethereum credentials. The full code of the implementation is available at the following url:

`https://github.com/dave0909/DecentralTrading`

In order to provide an implementation overview that references the actual code produced, mentions to filenames and functions in the previously proposed repository are used.

## 5.1 The on-chain smart contracts of the market

The components of the application running on the Ethereum ecosystem represent the core functionalities of DecentralTrading. Through the on-chain infrastructure, a distributed application is implemented in order to provide users with a third-party entity to validate and coordinate all the exchanges that take place in the market. Smart contracts are the enabler technologies to this end. Indeed, the DecentralTrading distributed application is composed of several communicating smart contracts, which are deployed by one or more authority accounts related to the owners of the infrastructure. The role of owner refers to service providers and is necessary for performing maintenance, moderation and supervision procedures of the infrastructure. The market smart contracts are grouped into three main modules:

- **Tokenization module**: manages subscriptions and user's right to enter the market.
- **Indexing module**: represents the logic for controlling metadata related to pods and resources initialized in the market.
- **Obligations module**: handle the storage and modification of obligation rules related to pods and resources.

Smart contracts contained in such modules fulfil several functions and functional dependencies that can be instantiated among them to enable communication between different components. Each smart contract is characterized by a public address that enables the interaction and the function invocations. It is assumed that smart contract addresses are freely disclosed as well as their code in order to increase trustworthiness with users.

All the smart contracts in DecentralTrading are written in Solidity, the native Ethereum programming language. The development environment adopted for the implementation is Remix IDE[1], an open source Ethereum Integrated Development Environment for writing, compiling and debugging Solidity code. Remix enables the integration of different Ethereum networks. During the implementation phase, Ganache[2] has been fundamental in simulating the blockchain environment. The software enables the execution of the Ethereum blockchain protocol, which is uniquely run by the node of the local machine. Remix IDE is able to communicate with the local blockchain hosted by Ganache. After the implementation phase, the smart contracts have been tested on the Ropsten[3] network. Ropsten is the most famous

[1]https://remix-project.org
[2]https://trufflesuite.com/ganache/
[3]https://ropsten.etherscan.io

Ethereum testnet, which offers a distributed environment that runs the same protocol as the Ethereum mainnet without the need to pay fees for code execution. Ropesten allowed us to test the on-chain infrastructure in a realistic blockchain environment.

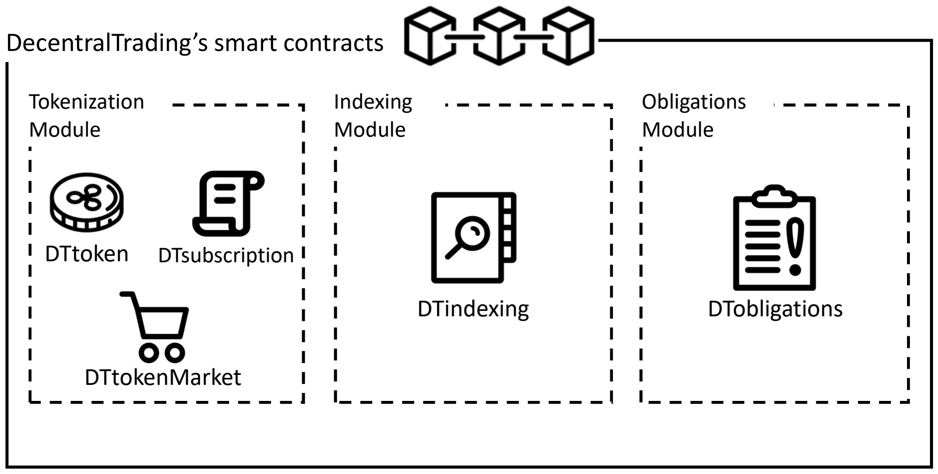


**Figure 5.1.** Smart contracts of the on-chain infrastructure.

### 5.1.1 Tokenization

The Tokenization module collects smart contracts implementing the exchange logic to enter the market and earn a profit. In order to acquire the rights to be part of the DecentralTrading application, users safely interact with these smart contracts using their Ethereum personal credentials in order to be uniquely identified. Digital fungible and non fungible tokens (NFTs) are the main technologies handled by the smart contracts of the tokenization module, and their use is employed to model the market currency and subscriptions. The immutability of information and ledger functions of Ethereum offer the perfect environment to store transactions of the digital assets involved in DecentralTrading.

The smart contracts which characterize the module are:

- DTtoken.sol
- DTtokenMarket.sol
- DTsubscription.sol

In the following paragraph code details and their functionalities are addressed.

#### 5.1.1.1 DTtoken.sol

The `DTtoken` smart contract is used to model and represent the custom currency employed for data trading purposes in DecentralTrading. The digital asset is named

DTtoken. The smart contract aims at keeping track of the accounts' balances and providing users the functionality to manage their amounts of DTtokens. Moreover, several functions, designed for the infrastructure owners, are implemented in order to control the flow of currency minted. DTtoken inherits the `Ownable` and `ERC20` abstract smart contracts. The purpose of these abstract entities is to provide standard functions to model the behaviour of the inheriting smart contract.
Ownable implements basic access control mechanisms to preserve functionalities solely accessible to an owner account. In the case of the DTtoken, the owner role is embodied by the service provider of the market. The abstract smart contract implements ownership by using a private address field named `_owner` and the `onlyOwner()` modifier usable in the header of functions that can only be invoked by the owner. Ethereum Request for Comment 20 (ERC20) is an Ethereum standard for Solidity smart contracts aimed at the implementation of fungible tokens. A fungible token can be compared to any kind of interchangeable asset, right, or access. By inheriting the implementation of ERC20 offered by the library OpenZeppelin, DTtoken receive the necessary code to execute token transactions between parties. Several fields of the `DTtoken` smart contract are used to provide descriptive information about the DTtoken. These include the private variables `_name` , `_sybmbol` and `_totalSupply`. `DTtoken` offers different standard functions, inherited by `ERC20`, publicly accessible to users through transactions or calls. The `totalSupply()` costant function returns to users the amount of minted DTtokens. The `balanceOf()` function deliver the balance of DTtokens of a given address. The transactional utilities of the smart contract are implemented through `transfer()` and `transferFrom()`. The `transfer()` function allows the invoking user to send a given amount of DTtokens to an input address. Differently, by using the `transferFrom()` function, a smart contract may automate the transfer procedure and transmit a certain quantity of the token on the owner's behalf. To this end, the `approve()` is necessary. This function takes as input a spending address and an amount. Once a user invokes the `approve()` function, the specified spender is enabled to move an approved input amount (or less) to a specified account via `transferFrom()`. The functions `decreaseAllowance()` and `increaseAllowance()` are both related to the `approve()` function. Indeed, they are able to increase or decrease the allowance previously granted through the `approve()` function. Two functions `mint()` and `burn()`, uniquely accessible by the user owner of the smart contract, have been added to the functions inherited from ERC20. To both is placed the `onlyOwner()` modifier in the header. The method `mint()` generates and adds a given amount of DTtokens in the balance of an input account. Oppositely, the `burn()` method decreases the balance of an input account by a given amount. The functions have been implemented to deliver service providers

an increased level of control over the on-chain infrastructure aimed at performing moderation procedures.

```
contract DTtoken is ERC20,Ownable {

    constructor() ERC20("DTtoken","DTt") {}

    function mint(address account, uint256 amount) public onlyOwner returns (bool)
    {
        _mint(account, amount);
    }

    function burn(address account,uint256 amount) public onlyOwner virtual
    {
        _burn(account, amount);
    }
```

**Listing 5.1.** Solidity code for the DTtoken smart contract.

#### 5.1.1.2 DTtokenMarket.sol

The main purpose of DTtokens is to offer users a currency with which DecentralTrading's subscriptions can be purchased. However, a dedicated distribution mechanism for DTtokens is necessary, to permit users to safely acquire amounts of currency by spending ETH. Again, Ethereum offers a suitable environment for such needs. The `DTtokenMarket` smart contract represents a reference point for users who want to buy or sell their DTtoken, by exchanging them with ETH. The smart contract is characterized by its own DTtoken balance and maintains as a field a reference to the `DTtoken` smart contract, in order to directly manage its transactions. `DTtokenMarket` is charged with tokens by the service providers, through earlier mentioned `mint()` function of the DTtoken smart contract. Once DTtokenMarket has accumulated enough tokens, it is ready to sell and exchange them for ETH. The `buyTokens()` function is the method users refer, to buy an input amount of DTtokens from the smart contract. This method uses two different modifiers. The `areTokensAvailable` modifier checks if the smart contract balance has the requested amount of DTtokens to be sold. The `isEtherEnough` modifier verifies if the transaction generated by the user contains the right amount of ETH to be exchanged for DTtokens. If both the checks performed by the modifiers are successful, the transfer of tokens from the DTtokenMarket balance to the user's balance will take place. The price of a single unit of DTtoken in ETH is expressed as a public field of the smart contract.

```
contract DTtokenMarket
{
    DToken dToken;
    uint public weiValue;
    constructor(address tokenAddress){
      dToken=DToken(tokenAddress);
      weiValue=500000000000000;
    }
    modifier areTokensAvailable(uint requiredTokens)
    {
        require(dToken.balanceOf(address(this))>=requiredTokens,
        "The requested amount is not available");
        _;
    }

    modifier isEtherEnough(uint etherAmount, uint neededEther)
    {
        require(etherAmount>=neededEther,
        "Insufficient amount of ether");
        _;
    }

    function buyTokens(uint amount) areTokensAvailable(amount)
       ↪ isEtherEnough(msg.value,weiValue*amount) payable public returns (bool
       ↪ success){
        dToken.transfer(msg.sender,amount);
        return true;
    }
}
```

**Listing 5.2.** Solidity code for the DTtokenMarket smart contract.

#### 5.1.1.3 DTsubscription.sol

The DecentralTrading market membership is currently managed by means of temporal subscriptions. Users who have accumulated DTokens in their accounts can spend them in order to acquire the rights to enter the market. Users with active subscriptions are allowed to perform operations involving the on-chain infrastructure of the market. The subscription to DecentralTrading is named DTsubscription and it is modelled by the homonymous smart contract `DTsubscription`. Similarly to `DTtoken`, the smart contract inherits `Ownable` to define functions restricted to the service provider's account. DecentralTrading subscriptions are implemented in the form of non fungible tokens (NFT). A NFT is a certificate which represents the ownership and authenticity of digital and non-digital goods. The main difference with classic fungible tokens is that NFTs contain specific information (e.g., identifiers, serial numbers, names) which makes them unique, non-interchangeable, and identifiable. Ethereum Request for Comments 721 (ERC721) is the most famous NFT standard for Ethereum smart contracts. It defines the main functionalities an NFT smart contract should implement in order to properly represent non-fungible tokens and manage their lifecycle. The `DTsubscription` smart contract extends the ERC721 implementation proposed by the OpenZeppelin library, in the form of an abstract smart contract. Several state variables are used in order to add more information to nature of the NFT. The fields `name_`,`symbol_` and `duration` are some of them. The `price` field expresses the amount of DTtokens necessary to purchase a DTsubscription. A reference to the DTtoken smart contract is stored as a state variable. The integer variable `mintedTokens` is crucial to keeping track of all the generated DTsubscription and assigning them a unique identifier. The distinctive information that characterize each token are represented by means of the struct `SubscriptionInfo`. The object contains three main attributes:

- **registeredOn**: an integer field representing the unix epoch in which the DTsubscription has been generated.
- **expiresOn**: an integer field representing the unix epoch after which the DTsubscription is no more. valid.
- **subscriptionType**: an enum field that distinguish the type of DTsubscription.

Ownership and identification information are added to the struct `SubscriptionInfo` via mapping variables. The mapping `tokenOwner` relates an integer NFT identifier to the Ethereum address of the owner. Differently, the `idToSubscriptionInfo` mapping uses an integer identifier as a key and a `SubscriptionInfo` object as a value. Functions such as `transferFrom()`, `approve()`, `ownerOf()` and `safeTransferFrom()` work similarly to the ERC20 version, offering users mechanisms to control their NFTs. Subscriptions are generated through the `purchaseSubscription()` function. This method allows user to buy DTsubscriptions, by spending DTtokens. The function requires the buyer user to have invoked the `approve()` method of the `DTtoken` smart contract. In this way, the `DTsubscription` smart contract can use the `tranferFrom()` method to retrieve the necessary DTtokens from the buyer. If the procedure is successful, a new `DTsubscription` is generated and assigned to the sender user. Different functions have been implemented to check the validity of a user's subscriptions. The method `isSubscriptionActive()` checks if a given integer identifier is related to a subscription that has not expired yet and returns a boolean value. The function `verify_subscription()` determines if a given subscription identifier is owned by an input address. Through `get_subscriptions()` users obtain the identifier of all the subscriptions owned by the input address.

```
. . .

function purchaseSubscription() public returns(uint)
{
  dToken.transferFrom(msg.sender,address(this),price);
  mintedTokens+=1;
  _mint(msg.sender,mintedTokens);
  tokenOwner[mintedTokens]=msg.sender;
  idToSubscriptionInfo[mintedTokens]=SubscriptionInfo(block.timestamp,block.timestamp
      ↪ + subscriptionDuration,SubscriptionType.FULL_SUBSCRIPTION);
  return mintedTokens;
}

function isSubscriptionActive(uint256 _tokenId) public view returns (bool state){
  SubscriptionInfo memory token = idToSubscriptionInfo[_tokenId];
  if(token.expiresOn<block.timestamp){return false;}
  else{return true;}
}

function verify_subscription(uint256 _tokenId,address claim_owner)public view
    ↪ returns(bool){
  address real_owner=tokenOwner[_tokenId];
  return isSubscriptionActive(_tokenId) && real_owner==claim_owner;
}

. . .
```

**Listing 5.3.** DTsubscription smart contract's fragment

### 5.1.2 Indexing

The indexing module offers several functionalities for data owners and data consumers regarding the resources initialised for the DecentralTrading market. The main goal of this component is to keep track of market's data. Data owners' technologies interact with this module in order to index their pods and resources. Users' register resources and share valuable metadata which can be used for data retrieval. Similarly, data consumers make use of the smart contract to find references for registered resources through search functionalities. The core smart contract of the module is `DTindexing`. Its code details are addressed in the next paragraph.

#### 5.1.2.1 DTindexing.sol

The `DTindexing` smart contract groups all the functionalities of the market regarding pods and resources. This smart contract allows data owners to register their pods and data. Pods and resources' identification is performed by means of serial identifiers stored as integer fields. These correspond to the private state variables `podsCounter` and `resourceCounter`. The two fields are incremented and assigned whenever new pods and resources are initialized. Pods' descriptive information are modelled through the `Pod` struct. The variables of such a struct are:

- **id**: integer identifier of the pod.
- **podType**: Enum type expressing the nature of the pod.
- **owner**: address of the user owner of the pods.
- **podAddress**: address with which pod managers sign transactions on behalf of the pod.
- **baseUrl**: bytes field containing the web reference of the pod service.
- **isActive**: boolean value defining if the pod can be contacted.

Similarly, the `Resource` struct contains information for initialized resources:

- **id**: integer identifier of the resource.
- **owner**: address of the user owner of the resource.
- **podId**: integer identifier of the pod storing the resource.
- **url**: bytes field containing the web reference to retrieve the resource.
- **isActive**: boolean value defining if the resource can be retrieved.

`Pod` and Resource structs are stored into array variables named `podList` and `resourceList`. The `registerPod()` allows pod managers to initialize new pods in DecentralTrading. It takes as input a bytes web reference for the pod service, the address of the owner and the pods type. The function creates a new `Pod` struct and stores it in the `podList`. Since every pod is related to an `Obligation` smart contract (see Section 5.1.3), the method performs the deployment of such an entity. Finally, the function emits a `NewPod` event containing the id and the address of the `Obligation` smart contract for the new pods. The method `registerResource()` works similarly, generating a new `Resource` object and storing it in the `resourceList` state variable. Oppositely, `deactivateResource()` and deactivatePod() make pods and resources no more accessible. The smart contract offers different search functions that can be extremely useful for data consumers. The methods getSocialPods(),getFinancialPods(),getMedicalPods() returns to market users the pod information according to their type. The three functions make use of the private method searchByType(), which takes as input a pod type to be searched. Through getPodResources() users obtain a list of Resource structs which are stored in the pods having the input id. The function makes use of the validPodId modifier to check if the given id is related to an active pod. The getResource() method accepts as input an integer identifier and returns the Resource struct having the given id. In this case, the validResourceId modifier verifies the validity of the input resource identifier.

```
. . .
modifier validResourceId(uint id)
{
    require (id<resourceList.length,"The given id is unknown");
    Resource memory resource= resourceList[id];
    require(resource.isActive==true,"The resource is not active");
    _;
}
modifier validPodId(uint id,uint idSubscription,address owner)
{
    require (id<podList.length,"The given id is unknown");
    Pod memory pod= podList[id];
    require(pod.isActive==true,"The pod is not active");
    require (pod.podAddress==owner,"The sender is not the pod");
    _;
}
function registerPod(bytes memory newReference,PodType podType,address podAddress)
    ↪ public returns(int)
{
   int idPod=podsCounter;
   podList.push(Pod(podsCounter,podType,msg.sender,podAddress,newReference,true));
   DTObligations obligation = new DTObligations(address(this),podAddress);
   emit NewPod(podsCounter,address(obligation));
   podsCounter+=1;
   return idPod;
}
function registerResource(int podId,bytes memory newReference) public
    ↪ validPodId(uint(podId),idSubscription,msg.sender) returns(int)
{
   int idResource=resourceCounter;
   resourceList.push(Resource(resourceCounter,msg.sender,newReference,podId,true));
   emit NewResource(resourceCounter);
   resourceCounter+=1;
```

```
  return idResource;
}
. . .
```

**Listing 5.4.** DTindexing smart contract's fragment.

### 5.1.3 Obligations

One of the main purposes of DecentralTrading is to ensure data owners have a high degree of control over shared resources. Most data markets do not propose any mechanism for controlling data once it has been retrieved. DecentralTrading offers usage control inspired procedures to allow data providers to define restrictions on resource utilization. To this end, obligation rules have been designed. Obligations are stored in the Ethereum blockchain and their access and modification take place through smart contract interaction. The design of DecentralTrading involves the deployment of several `DTobligation` smart contracts, as many as there are pods in the market. Indeed, each pod manager is owner of a specific `DTobligation` smart contract. The next paragraph addresses the `DTobligation` implementation.

#### 5.1.3.1 DTobligation.sol

As with many smart contracts of the on-chain infrastructure of the market, `DTobligation` extends the `Ownable` abstract smart contract in order to establish the ownership of the related pod technology's credentials. Through `Ownable`, the functionality of writing and modifying obligation rules is reserved only for transactions signed through the pod's credentials. The actual implementation includes four types of obligations. The `AccessCounterObligation` struct models a restriction on the resource accesses in the client device and, it is composed by an integer field `accessCounter` and a boolean variable `exists` which is set on `true`, when the rule is defined. The `CountryObligation` struct represents limitations on the country in which a resource can be used. Its main fields are `countryCode`, which represents countries' identifier, and the `exist` variable. `DomainObligation` struct expresses for which purposes resources can be used. Finally, `TemporalObligation` imposes a maximum duration for resource usage. Obligation rules are grouped via `ObligationRules` struct. An `ObligationRules` struct can refer to a specific pod resource or more generally to the pod. In the latter case, all the resources of the pod that do not have a specific `ObligationRules` object inherit the struct related to the pod. An `ObligationRules` object is composed of the following fields:

- **idResource**: integer variable containing the identifier of the resources to which the rules refer. If the rules refer to the pods, a default value is set.

- **acObligation**: `AccessCounterObligation` struct variable.
- **countryObligation**: `CountryObligation` struct variable.
- **temporalObligation**: `TemporalObligation` struct variable.
- **domainObligation**: `DomainObligation` struct variable.
- **exists**: boolean variable which expresses if the object have been instantiated.

The smart contract stores obligation rules thanks to the fields `defaultPodObligation` and `resourcesObligation`. The first of them stores the `ObligationRules` object associated with the pod, while `resourcesObligation` is a mapping variable which links integer resources identifiers to `ObligationRules` structs. Among the fields of the contract we find the `dtIndexing` variable, which keeps a reference to `DTindexing` smart contract of the market. The `addDefaultAccessCounterObligation()`, `addDefaultTemporalObligation()`, `addDefaultDomainObligation()` and `addDefaultCountryObligation()` functions are invoked by pod managers in order to set the default rules associated with the pod. All of them make use of the `onlyOwner` modifier to ensure that the functions are invoked only by the owner pod. The `isValidTemporal` modifier is employed by addDefaultTemporalObligation() to check if the input time duration is a consistent value. The methods `addAccessCounterObligation()`, `addTemporalObligation()`, `addDomainObligation()` and `addCountryObligation()` offer the same functionalities as the previous functions, but referencing specific resources. Indeed, they all require an input resource identifier in order to record a new rule. In this case, the `isTheResourceCovered` modifier verifies that the given id belongs to a resource of the pods. Default and specific resource rules can be deleted thanks to dedicated methods such as `removeDefaultAccessCounterObligation()` and `removeAccessCounterObligation()`. Finally, through reading functions, market users can read the stored obligation rules. The method `getDefaultObligationRules()` returns the `ObligationsRules` object stored in the `defaultPodObligation` field. Similarly, `getObligationRules()` requires an input identifier to retrieve the `ObligationsRules` struct belonging to the resource from the `resourcesObligation` mapping variable.

```
. . .
struct ObligationRules{
    int idResource;
    AccessCounterObligation acObligation;
    CountryObligation countryObligation;
    TemporalObligation temporalObligation;
    DomainObligation domainObligation;
    bool exists;
}
function addDefaultTemporalObligation(uint temporalObligation)public
    ↪ isValidTemporal(temporalObligation) onlyOwner()
{
    uint d=1 days;
    require(temporalObligation>d,"The temporal obligation must be at least 1 day");
    defaultPodObligation.temporalObligation.exists=true;
    defaultPodObligation.temporalObligation.usageDuration=temporalObligation;
}
function removeDefaultTemporalObligation() onlyOwner() public
{
    defaultPodObligation.temporalObligation.exists=false;
    defaultPodObligation.temporalObligation.usageDuration=0;
}
function addTemporalObligation(int idResource,uint deadline)
    ↪ isTheResourceCovered(idResource) public isValidTemporal(deadline)
    ↪ returns(ObligationRules memory) onlyOwner()
{
    if (resourcesObligation[idResource].exists){
        resourcesObligation[idResource].temporalObligation=TemporalObligation(deadline,true);
    }
    else{
            resourcesObligation[idResource].exists=true;
            resourcesObligation[idResource].idResource=idResource;
            resourcesObligation[idResource].temporalObligation=TemporalObligation(deadline,true);
    }
    return resourcesObligation[idResource];
}
function removeTemporalObligation(int idResource) isTheResourceCovered(idResource)
    ↪ public hasSpecificRules(idResource) onlyOwner()
{
    resourcesObligation[idResource].temporalObligation.exists=false;
    resourcesObligation[idResource].temporalObligation.usageDuration=0;
}
. . .
```

**Listing 5.5.** DTobligations smart contract's fragment.

## 5.2 The pod manager

Once the on-chain infrastructure of DecentralTrading is addressed, the focus of the discussion is brought to the data owner's technologies. The thesis proposes a prototype implementation of the pod manager software. The goal of the prototype is to provide data owners with the functionality to properly manage their resources in the data market. The software is able to physically control the user's resources in specific locations of the filesystem. Pod managers communicate with on-chain smart contracts of the infrastructure to initialize and deactivate resources. Moreover, the software implements web services to deliver resources to data consumers through the HTTP protocol. The implementation details inherent in these issues will be addressed in detail in the following subsections.

### 5.2.1 The pod manager application

A pod manager software is the core technology to enable the sharing and the control of the resources in DecentralTrading. The application has been designed to be installed on users' machines and to interact with local filesystems. The pod manager prototype is developed in the Python language, which guarantees elasticity and dynamism during the implementation phase. Python supports the Web3.py module, allowing the creation of communication protocols with Ethereum and on-chain smart contracts of DecentralTrading. Through the use of Tkinter, the prototype offers a Graphical User Interface aimed at simplifying the interaction with the pod manager's functionalities. The core structure of the user interface is in the `app.py` file. The controller entity of the application is the `App` class, extending `Tk`. The class contains the necessary logic to control window settings and to coordinate the view transitions. The views of the application are implemented by means of `Frame` classes extensions. Each class is characterized by methods and variables required for the execution of the view's functionalities.
The first view that the user interacts with is modelled through the `WelcomePage` class. It manages the physical generation of pods in the local filesystem and their initiation into DecentralTrading. The page offers a form aimed at creating a new pod by asking the user for descriptive information and its Ethereum credential to sign the initiation transaction (see Section 5.2.3). The function of `WelcomePage` which starts the initiation operation is `submit_validation()`. The function inspects the form fields and, in the case of a positive outcome, initiates the procedure by invoking `register_pod()`. The method communicates with the on-chain infrastructure and sets up the filesystem location of the pod generating configuration files through `generate_config_files()` (see Section 5.2.2). Once a pod and the associated

metadata are generated, the view allows users to select the location in order to perform control operations.
The selection of a pod filesystem causes the transition to the pod management view, implemented through the `PodManagement` class. It reads the pod information from the configuration files in the location and stores them into class variables such as `self.pod_address`,`self.pod_owner` and `self.pod_id`. The view collects the core functionalities to manage pods and their resources. The class functions `start_server()` and `stop_server()` deals with the pod web service for data provision (see Section 5.2.4). In order to initialize new resources, users must select an existing file from the filesystem, that will be copied into the pod location. The functions `register_resource()` and `add_resource_to_config()` use the input file information to initiate the resource both in Ethereum's smart contracts and in the local configuration file. All the Ethereum transactions involved in pod management operations are signed using the pod credentials. By interacting with the `PodManagement` view, users visualize the list of the pod's initialized resources. Users can perform the dectivation operation on each resource to remove it from the market. The method `remove_resource()` is crucial for such a purpose, by starting the procedure to physically remove the resource from the filesystem, invoking the involved smart contract function and synchronising the local configuration files. `PodManagement` collects the functionalities to manage the default pod's obligation rules. Indeed, the view shows the actual state of the rules and offers users visual tools to submit transactions and modify them. The resource list of the view permits data owners to get market details for each initialized file on a dedicated page. Such details are represented and shown through the `ResourceManagement` class.
Once the class is initialized through the `initialize_layout()` method, it stores relevant information in state variables such as `self.resource_information`. Such data is displayed in dedicated sections of the view. The page is designed to enable the modification of the obligation rules specifically associated with the resource. Indeed, users can specify access counter, temporal, country and domain obligations for the given resource. The submission of the new rule generates a new Ethereum transaction for the smart contract invocation. Moreover, the function changes the local configuration files containing the resource rules information. The resource page is consequently reloaded after the operation. A similar functionality is provided by `remove_obligation()` that allows users to deactivate a selected rule for the specified resource.

### 5.2.2 Physical organization of pods

One of the main functionalities of a pod manager is to organize and control the resources in the local filesystem. A pod is presented as a folder located in a specific location of the filesystem. The aim of the pod manager action is access to the location and execute physical changes to the resources stored therein. The pod manager software offers users intermediary operations to indirectly interact with the pod's location. Indeed, besides market resources, a pod's location contains sensitive metadata that are crucial for the proper running of the pod manager and to correctly interact with DecentralTrading's entities. To this end, pod's resources are protected through encryption and firewalling methodologies, aimed at isolating the location from inconsistencies and confidentiality issues. In this way, users are forced to manage their initialized resources solely through the pod manager that holds the rights to overcome the designed preventive measures. Pod's resources can be stored in the root location or in nested folders, placed inside the root. `DTconfig.json` and `DTobligations.json` are the two meta-resources containing descriptive and confidential information about the pods. Both the JSON files locally store data that is replicated in on-chain smart contracts. Therefore, synchronization procedures aimed at preserving the consistency of replicated information are necessary.
`DTconfig.json` is characterized by the following attributes:

- **id**
- **owner**
- **address**
- **private_key**
- **URI**
- **resources**

The `id` variable is the pod's integer identifier, generated during the pod registration procedure. It is used to uniquely identify the pod inside the market. The `owner` field stores the public key of the market user that generated the transaction for the pod initiation. The file maintains the pod's Ethereum credentials in `address` and `privatekey`. The two variables are critical to allowing the pod manager to sign authenticated transactions on behalf of the pod. They are both generated during the pod creation and are uniquely assigned to it. `URI` keep in memory the web reference of the pod's web service for data retrieval purposes. Finally, `resources` is a json list object, aimed at storing the initialized resources of the pod. Each resource object is characterized by a resource `id` and the relative `location` in the pod.

**DTobligations.json** collects information regarding the obligation rules associated with the pod and its resources. The attributes of the json file follow the structure:

- **"default"**
- **"id**$_{resource1}$**"**
- . . .
- **"id**$_{resourceN}$**"**

The `default` key attribute defines the general obligations related to the pod, inherited by all the resources not having specific rules. Differently, rules belonging to specific resources are associated with the market id of the resource. Obligations are modelled as json objects characterized by attribute fields, each of which defines the value of a different rule (access counter, domain, country, temporal).

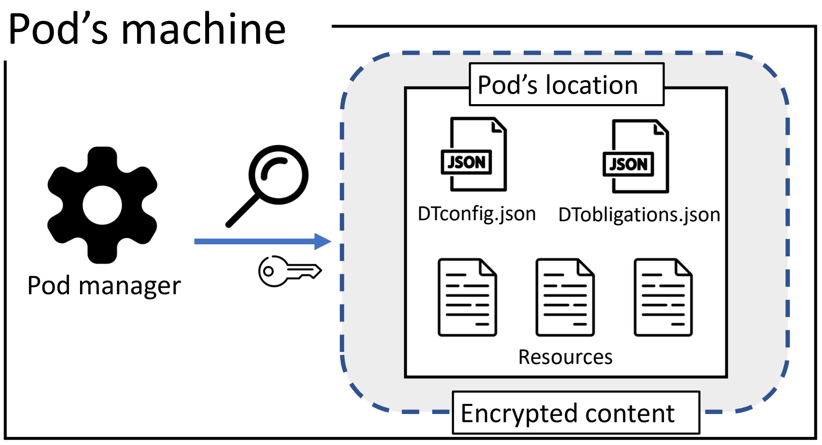


**Figure 5.2.** Structure of pods.

### 5.2.3 Blockchain interaction

The pod manager prototype makes use of dedicated code components aimed at providing communication mechanisms with the on-chain smart contracts of Decentral-Trading. Indeed, the Ethereum infrastructure safely stores crucial resource metadata that guarantees the proper functioning of the market. The `DTaddresses.py` file contains the addresses of the deployed `DTindexing` and `DTsubscription` smart contracts. The two references are stored in the variables `DTINDEXING` and `DTSUBSCRIPTION`. The files containing the logic for the blockchain interaction are:

- **DTsubscription_oracle.py**

- **DTindexing_oracle.py**
- **DTobligation_oracle.py**

The term “oracle’ refers to communication mechanisms used in blockchain-based processes to enable the communication between on-chain smart contracts and off-chain entities. In that case, the use of the term is figural, having chosen the design philosophy of the direct communication model between off-chain entities and smart contracts (see Section 4.2.1).
`DTsubscription_oracle.py` is in charge of implementing the interaction with the `DTsubscription` smart contract and performing subscription verifications. To this end, the `DTsubscription_oracle` class has been designed. The initialization of the class requires as input the address of the smart contract and pod private key that are stored in the `self.indexing_address` and `self.private_key` variables. The class field `self.contract_abi` memorizes the Application Binary Interface string for the function invocations. The main method of the class is `pull_subscription_verification()`, which takes as input a subscription id and a public address. The method queries the blockchain, invoking the `verify_subscription()` function of `DTsubscription` and determines if the subscription id is active and related to the given address.

```python
class DTsubscription_oracle:

    def __init__(self, *args, **kw):
        self.indexing_address=args[0]
        self.private_key=args[1]
        self.contract_abi=CONTRACT_ABI
        self.account= Account.from_key(self.private_key)
        self.provider=Web3(Web3.HTTPProvider('HTTP://127.0.0.1:7545'))
        self.contract_instance =
            ↪ self.provider.eth.contract(address=self.indexing_address,
            ↪ abi=self.contract_abi)

    def pull_subscription_verification(self,id_subscription,claim):
        result=self.contract_instance.functions.verify_subscription(
            ↪ id_subscription,Web3.toChecksumAddress(claim)).call()
        print("is subscription verified: ",result)
        return result
```

**Listing 5.6.** DTsubscription_oracle Python class.

`DTindexing_oracle.py` contains the `DTindexing_oracle` class that allows the proper invocation of the `DTindexing` smart contract’s functions. The instantiation process follows the principles of the previous class, by storing in state variables the necessary information to execute contract calls. The `register_pod()` method is used by the pod manager during the generation of a new pod. The function accepts as input the pod metadata obtained from the form filling and generates new Ethereum credentials to be associated with the pod. Subsequently, the `private_key_owner` variable is used to sign the transaction for the invocation of the `registerPod()` method. The method is executed by the blockchain environment, which initialize the new pod and, in the case of a positive outcome, emits an `NewPod` event containing the id. The operation of the other functions of the classes is quite similar to the latter. The `add_resource()` method is used to initialize new resources in the pod and invokes the on-chain `registerResource()` function. Oppositely, `deactivate_resource()` allows users to delete a resource from the pod by generating a transaction for the invocation of the `deactivateResource()` smart contract method. Finally, `get_resource_information()` queries the on-chain infrastructure about an initialized resource, in order to retrieve its metadata by calling `getPodResources()`.

```python
def get_resource_information(self,resourece_id):
    contract_instance = self.provider.eth.contract(address=self.indexing_address,
        ↪ abi=self.contract_abi)
    return contract_instance.functions.getPodResources(resourece_id,0).call()

def register_Pod(self,pod_reference,pod_type,public_key_owner,private_key_owner):
    public_key_pod,private_key_pod=DTaccount_generator().generate_account()
    tx=self.contract_instance.functions.registerPod(pod_reference,pod_type,
        ↪ Web3.toChecksumAddress(public_key_pod)).buildTransaction({'gasPrice':
        ↪ Web3.toWei(21, 'gwei'),'nonce':
        ↪ self.provider.eth.getTransactionCount(public_key_owner)})

    signed_txn = self.provider.eth.account.sign_transaction(tx,
        ↪ private_key=private_key_owner)
    tx=Web3.toHex(self.provider.eth.sendRawTransaction(signed_txn.rawTransaction))
    retVal = self.provider.eth.waitForTransactionReceipt(tx)
    processed_receipt=self.contract_instance.events.NewPod().processReceipt(retVal)
    id=processed_receipt[0]['args']['idPod']
    obligation_address=processed_receipt[0]['args']['obligationAddress']
    return id,public_key_pod,private_key_pod,obligation_address
```

**Listing 5.7.** Fragment of the DTindexing_oracle class.

`DTobligation_oracle.py` implements the logic to correctly interact with the `DTobligation` smart contract. The `DTobligation_oracle` class offers the functionality to modify the obligation rules stored in the blockchain. Obligations can be generally associated with the whole pod or with specific resources. The functions `set_default_access_counter_obligation()`, `set_default_ temporal_obligation()`, `set_default_country_obligation()`,`set_default_domain_obligation()` are employed to set new values for the default rules of the pod. Oppositely, `deactivate_default_ access_counter_obligation()`, `deactivate_default_ temporal_obligation()`, `deactivate_default_country_obligation()` and `deactivate_default_domain _obligation()` delete the limitations previously introduced by the pod regarding the default rules. In a similar manner, functions to modify or remove rules related to specific resources such as `set_domain_obligation()` and `deactivate_domain_obligation()` are implemented. Such functions require an input parameter that defines the identifier of the resource whose rules are to be changed.

```
def set_default_temporal_obligation(self,temporalObligation):
    tx=self.contract_instance.functions.addDefaultTemporalObligation(
        ↪ temporalObligation ).buildTransaction({'gasPrice': Web3.toWei(21,
        ↪ 'gwei'),'nonce':
        ↪ self.provider.eth.getTransactionCount(self.account.address)})
    signed_txn = self.provider.eth.account.sign_transaction(tx,
        ↪ private_key=self.private_key)
    tx=Web3.toHex(self.provider.eth.sendRawTransaction(signed_txn.rawTransaction))
    print(self.provider.eth.waitForTransactionReceipt(tx))
```

**Listing 5.8.** Fragment of the DTobligation_oracle class.

### 5.2.4 Data provision and requests authentication

The architecture of DecentralTrading assumes data consumers retrieve resources by communicating with pods through the existing HTTP standard. Indeed, pods share the market web references that are used by consumers to access specific resources. Such a scenario requires pod managers to be able to handle RESTful interactions to deliver on-demand initialized data. In order to properly manage data provision, the pod manager prototype implements a web service which listens to HTTP requests, authenticates senders, verifies their market rights over the claimed resource, and correctly delivers data through HTTP responses.
`DTpod_service` contains the core functionalities to this end and represents them through the `DTpod_service` class. It extends `BaseHTTPRequestHandler`, which provides a number of classes, instance variables, and methods to handle GET and POST requests. The `serve_forever()` method activates the web service and keeps the pod manager waiting for new http requests. Oppositely, `force_stop()` ends the request handling and the web service. The `do_POST()` function is the main method of the class. Indeed, due to confidentiality reasons, the web service only responds

to POST requests and ignores GET ones. Indeed, POST enables the inclusion of parameters inside the body of the request, while the GET method specifies the parameters in the visible URL. Such parameters are crucial for the authentication procedure. In order to correctly demand a resource, requests must specify an URL composed of the web domain name of the service followed by the relative path of the requested resource inside the pod's filesystem.
Through the authentication mechanism, pod managers clarify the identity of the request user, and decide if the delivery of the resource is granted. Authentication is based on the Ethereum's elliptic-curve cryptography algorithm. The main idea behind the mechanism is to make senders sign a message, which is sent alongside the request. The string message is characterized by the following structure:

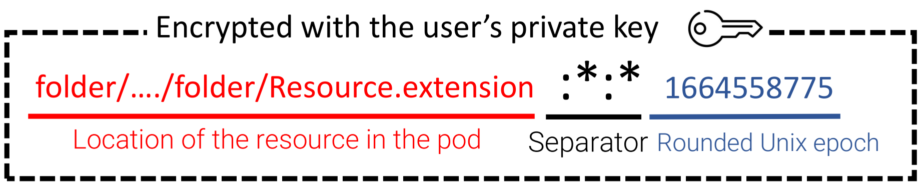

Encrypted with the user's private key
folder/..../folder/Resource.extension
:*:*
1664558775
Location of the resource in the pod
Separator
Rounded Unix epoch

**Figure 5.3.** Structure of the authentication message.

The first part of the message refers to the relative path of the requested resource, which is concatenated with the ":*:*' separator. The last part of the message specifies a Unix epoch rounded by 5 minutes. This last part is employed to create a limited validity temporal interval for the message. Once the message is built, it is signed by the sender through its private key. To this end, the `Web3` library offers the `sign_message()` function that takes the hashed message and the private key as input. The function produces a string containing the signature of the given message. In order to be processed by the pod manager web service, a POST request must contain in its body the following attributes:

- **auth_token**: the signature of the message.
- **claim**: the public address related to the claimed identity.
- **subscription_id**: the identifier of a DTsubscription owned by the claimed identity.

Once the request is received, the doPOST() method checks that the previously mentioned fields are present in its body. If they are set, the method proceeds with the authentication and rights verification operations. The authentication is managed by doPOST() through the `DTautenthicator` class, contained in `DTauthenticator.py`. The `authenticate` function is the core method of the class. The function uses the url of the requested resource and the rounded unix epoch to build the unsigned message. After that, the received `auth_token` field is used in combination with the unsigned message to extract a public key by means of the `Web3 recover_message()` function. Finally, the `claim` field is compared with the extracted public key and a boolean value is returned to `doPOST()`. If `claim` and the extracted public key are equal, the claimed identity is verified and the execution continues. Otherwise, an HTTP error is sent as a response to the sender. The following verification step checks if the verified Etherum user is part of the DecentralTrading market. To this end, the `subscription_id` field of the request is exploited. Indeed, the Ethereum on-chain infrastructure is queried to determine if `subscription_id` is associated to an active DTsubscription, owned by the verified identity. The interaction with the DTsubscription smart contract is managed through the `DTsubscription_oracle` class (see Section 5.2.3). The `pull_subscription_verification()` method returns to `doPOST()` the boolean result of the verification. If the verification succeeds, data provision procedures are adopted. The method in charge of retrieving the requested resource to be sent is `send_head()`. The function uses the relative path of the resource contained in the request URL to get the file from the pod's filesystem. If an `IOError` exception is raised and the resource is not found, the function sends a 404 message error. Otherwise, the header of the response is set according to the resource type(e.g. text, html, image) identified via `gues_type()` and the file content is returned to `doPOST()`. Finally the resource is written in the response body, and it is forwarded to the sender.

```python
class DTauthenticator():
    def __init__(self, *args, **kw):
        self.w3=Web3(Web3.HTTPProvider('HTTP://127.0.0.1:7545'))
    def rounded_to_the_last_30th_minute_epoch(self,unix_time):
        date_time = datetime.fromtimestamp(unix_time)
        now = date_time
        rounded = now - (now - datetime.min) % timedelta(minutes=5)
        return rounded.timestamp()
    def get_time_in_rome(self):
        rome_tz= pytz.timezone("Europe/Rome")
        time_in_rome = datetime.now(rome_tz)
        return time_in_rome
    def encode_unsigned(self,resource,time):
        msg_to_hash=resource+":*:*"+time
        msghash = encode_defunct(text=msg_to_hash)
        return msghash
    def authenticate_signature(self,signature,msg_hash,claim):
        return claim==self.w3.eth.account.recover_message(msg_hash,
            ↪ signature=signature)
    def authenticate(self,resource,signature,claim):
        time_in_rome=self.get_time_in_rome()
        rounded=self.rounded_to_the_last_30th_minute_epoch(int(time_in_rome.timestamp()))
        msg_hash=self.encode_unsigned(resource,str(rounded))
        bytes_signature=HexBytes(signature)
        return self.authenticate_signature(signature,msg_hash,claim)
```

**Listing 5.9.** DTauthenticator Python class.

```
def send_head(self):
    path = self.translate_path(self.path)
    f = None
    if os.path.isdir(path):
        if not self.path.endswith('/'):
            self.send_response(301)
            self.send_header("Location", self.path + "/")
            self.end_headers()
            return None
        else:
            self.send_error(404,"Pod resource not found")
    ctype = self.guess_type(path)
    try:
        f = open(path, 'rb')
    except IOError:
        self.send_error(404, "Pod resource not found")
        return None
    self.send_response(200)
    self.send_header("Content-type", ctype)
    fs = os.fstat(f.fileno())
    self.end_headers()
    return f
```

**Listing 5.10.** Function send_head() of the DTpod_service class.

# Chapter 6

# Evaluation

In order to discuss the implementation of the addressed DecentralTrading components, a quantitative assessment is offered. The main goal of this chapter is to distinguish the on-chain functionalities of DecentralTrading's smart contracts according to the users for whom they are intended and evaluate their individual costs. The users considered are data owners, data consumers, and service providers. The purpose of the evaluation is to highlight the most onerous functionalities and provide a guideline for future code improvements. The functionality grouping points out which users spend the most in the current version of the infrastructure. Runs performed for quantitative evaluation are also employed in order to verify the proper operation of on-chain components and to conduct a functional validation of the implemented code.
The first section describes the object of evaluation and the methodology by which it is assessed. After that, an objective depiction of the results is offered. Finally, the presented findings are analyzed and interpreted.

## 6.1 Methodology

Costs are the major factor considered by the evaluation procedure. The DecentralTrading data market collects several smart contracts on the Ethereum platform designed to coordinate and validate application parties. Smart Contracts' code execution is performed in Ethereum through the Ethereum Virtual Machine (EVM), a distributed execution environment. Each node of the network executes the EVM and simultaneously runs the instructions resulting from smart contract invocations. Smart contracts collect Turing complete code and they can potentially execute forever, consuming huge amounts of electricity and locking up every single node on the blockchain. To safeguard the operation of the platform, the Ethereum protocol associates the execution of smart contracts with a fee charged to the invoking user, according to the complexity of the code to be executed. The higher the computational burden, the higher the fee. Such a fee is measured in Gas. Numerous automated tools enable the cost analysis of Solidity smart contracts. Because the implementation of DecentralTrading smart contracts has been conducted using Remix IDE, an integrated solution to the development environment has been adopted to perform the cost analysis. Specifically, Gas Profiler and Static Analysis Tool are the plugins used for the purpose of the assessment. The evaluation procedure starts with the identification of the target users (data owners, data consumers, and service providers) for each function produced. Subsequently, Remix IDE is connected to the Ethereum environment offered by Ganache. Ganache enables the execution of a local Ethereum blockchain that replicates the Ethereum protocol and supports the generation of transactions for testing purposes. Once the testing environment is set up, the smart contracts to be evaluated are properly deployed to the local Ganache blockchain, respecting the functional dependencies between them. In order to test smart contract functions, valid input parameters are collected in order to generate transactions, resulting in the successful execution of the code. In addition to cost evaluation, the said transactions are used to validate the functions implemented. The outputs and post-conditions resulting from code execution are collected and verified in order to build a functional validation of the smart contracts. The Gas Profiler and Static Analysis Tool plugins are used to determine the Gas amount related to code runs. The obtained results are collected in several tables, each of which is related to a specific smart contract. Evaluation tables associate each public function with the target user and the related gas cost.

## 6.2 Results

Result tables are characterized by three main fields. The “function” field expresses the evaluated method’s name. The “cost” field contains the amount of gas spent by the method. Finally, “charged to” clarifies the target user that is supposed to use and pay for the method execution. The Gas cost of a function is calculated by the EVM according to the machine instructions to be executed. Each instruction requires a specific cost, and the final result is given by the sum of all the involved costs. The final cost has a variable range due to the specific execution context (e.g., number of loops, the complexity of input parameters). For this reason, the values reported are approximations, useful in providing an order of magnitude for each of the assessed methods. The following subsections show the final results for each smart contract.

### 6.2.1 DTtoken

DTtoken provides functionality to control the homonymous ERC20 token. The deployment operation, thanks to which DTtoken is published by service providers, costs 1623406 units of Gas. The method `increaseAllowance()` is the most onerous function of the smart contract with 46000 units of Gas. Oppositely, `decreaseAllowance()` spends only 15828 units of Gas. The smart contract offers two `view` functions that do not generate transactions, resulting in no Gas expenditure. The majority of the smart contract's methods involve all the user typologies. However,`mint()`, `burn()` and deployment are solely targeted at the market's service providers.

| Function | Cost (Gas) | Target User |
|---|---|---|
| deployment | 1623406 | Service Providers |
| mint() | 37640 | Service Providers |
| burn() | 36730 | Service Providers |
| transfer() | 36811 | Service Providers, Data Owners, Data Consumers |
| transferFrom() | 45752 | Service Providers, Data Owners, Data Consumers |
| increaseAllowance() | 46000 | Service Providers, Data Owners, Data Consumers |
| decreaseAllowance() | 15828 | Service Providers, Data Owners, Data Consumers |
| allowance() | - | Service Providers, Data Owners, Data Consumers |
| balanceOf() | - | Service Providers, Data Owners, Data Consumers |

**Table 6.1.** Gas analysis results for the DTtoken smart contract.

### 6.2.2 DTtokenMarket

DTtokenMarket enables the distribution of DTtokens in exchange for ETH. The only costs associated with smart contracts are those associated with deployment and the texttttbuyTokens() function. The deployment of the smart contract by service providers requires 442560 Gas. Data owners and data consumers buy tokens from the smart contracts, spending 45516 units of Gas.

| Function | Cost (Gas) | Target User |
|---|---|---|
| deployment | 1623406 | Service Providers |
| buyTokens() | 37640 | Data Owners, Data Consumers |

**Table 6.2.** Gas analysis results for the DTtokenMarket smart contract.

### 6.2.3 DTsubscription

DTsubscription offers functionalities aimed at controlling and modelling market subscriptions. The DTsubscription deployment procedure costs 2502600 Gas units. The smart contract is characterized by several Gas-free functions such as `balanceOf()`, `get_Subscription()`, `isSubscriptionActive()` and `verify_Subscription()`. Oppositely, `purchaseSubscription()` and `burn()` involve Gas expenditures. The `purchaseSubscription()` method is employed by data owners and data consumers to generate subscriptions by exchanging their DTtokens. The operation of the function costs 180792 units of Gas. The `burn()` function, used by service providers, implements the revocation of a subscription by spending 204592 Gas units.

| **Function** | **Cost (Gas)** | **Target User** |
|---|---|---|
| deployment | 2502600 | Service Providers |
| purchaseSubscription() | 180792 | Data Owners, Data Consumers |
| burn() | 204592 | Service Providers |
| balanceOf() | - | Data Owners, Data Consumers |
| get_subscriptions() | - | Service Providers, Data Owners, Data Consumers |
| isSubscriptionActive() | - | Service Providers, Data Owners, Data Consumers |
| verify_subscription() | - | Service Providers, Data Owners, Data Consumers |

**Table 6.3.** Gas analysis results for the DTsubscription smart contract.

### 6.2.4 DTindexing

Through DTindexing, pods and resources' metadata are initialized and stored in the Ethereum ecosystem. In order to publish the smart contract on the network, service providers spend 4824135 Gas units. Data owners and data providers are the user typology to which most of the smart contract functionalities refer. The `registerPod()` method is the most expensive DTindexing function. Indeed, such a function costs data owners 2703393 units of Gas. The Gas consumption of `registerResource()` turns out to be significantly lower, with a value of 143004. The less expensive function of the smart contract is `deactivateResource()`, which costs to data owners 21465 Gas units. Data consumer-oriented functions do not involve the generation of transactions since they provide read-only functionalities. Gas expenditure in those cases is zero.

| **Function** | **Cost (Gas)** | **Target User** |
|---|---|---|
| deployment | 4824135 | Service Providers |
| registerPod() | 2703393 | Data Owners |
| registerResource() | 143004 | Data Owners |
| deactivateResource() | 21465 | Data Owners |
| getFinancialPods() | - | Data Consumers |
| getSocialPods() | - | Data Consumers |
| getMedicalPods() | - | Data Consumers |
| getPodResources() | - | Data Consumers |
| getResource() | - | Data Consumers |

**Table 6.4.** Gas analysis results for the DTindexing smart contract.

### 6.2.5 DTobligation

DTobligation contains functionalities solely directed at data owner users. The smart contract is deployed during the indexing of a new pod at a cost of 2650030 Gas. DTobligation offers methods and functions to modify the obligation rules related to the pod or to a specific resource contained therein. Among the functions for adding rules, the most expensive one turns out to be `addAccessCounterObligation()` with a value of 138768 Gas units. Differently, the adding of a domain restriction through `addDefaultDomainObligation()` costs data owners 44219 Gas units per invocation. Methods implemented for rules deactivation determine a lower expense than the previous ones. Among them, `removeDomainObligation()` requires 38111 Gas units per execution, while `removeDefaultTemporalObligation()` turns out to be the cheapest (16079 units of Gas). Finally, the methods `getDefaultObligationRules()` and `getObligationRules()`, which are also aimed at data consumers, are free of charge.

| **Function** | **Cost (Gas)** | **Target User** |
|---|---|---|
| deployment | 2650030 | Data Owners |
| addDefaultAccessCounterObligation() | 62627 | Data Owners |
| addDefaultTemporalObligation() | 62638 | Data Owners |
| addDefaultDomainObligation() | 44219 | Data Owners |
| addDefaultCountryObligation() | 62561 | Data Owners |
| addAccessCounterObligation() | 138768 | Data Owners |
| addTemporalObligation() | 97737 | Data Owners |
| addCountryObligation() | 97728 | Data Owners |
| addDomainObligation() | 79452 | Data Owners |
| removeDefaultAccessCounterObligation() | 23780 | Data Owners |
| removeDefaultTemporalObligation() | 16079 | Data Owners |
| removeDefaultDomainObligation() | 24747 | Data Owners |
| removeDefaultCountryObligation() | 23758 | Data Owners |
| removeAccessCounterObligation() | 28184 | Data Owners |
| removeTemporalObligation() | 28151 | Data Owners |
| removeCountryObligation() | 28173 | Data Owners |
| removeDomainObligation() | 38111 | Data Owners |
| getDefaultObligationRules() | - | Data Owners, Data Consumers |
| getObligationRules() | - | Data Owners, Data Consumers |

**Table 6.5.** Gas analysis results for the DTobligation smart contract.

## 6.3 Discussion

A Gas to EUR conversion is necessary in order to better contextualize the obtained results. The Gas price considered for the conversion is the average value of 10.96 GWei (0.00000001096 ETH) [2]. The current ETH exchange rate is about 1440.00 EUR [21].

The first considerations concern infrastructure maintenance costs, which are borne by service providers. The approximate value of publishing DecentralTrading smart contracts on the Ethereum network is given by the sum of the deployment operations. Considering the previously mentioned exchange rates, the publication cost of all smart contracts is around 148.239 EUR (9392701 Gas units). Deployment operations prove to be the most onerous to perform because of the large number of instructions involved. This point highlights the need for careful and strict code analysis procedures aimed at identifying bugs and critical issues before smart contracts are published. However, publication fees are one-off costs, and smart contract availability is constantly guaranteed by the network after the deployment. The remaining service providers' functionalities result in significantly lower costs. Indeed, the mean value per function invocation for service providers is about 1.46756 EUR (92987.33333 Gas units).

Data owners' interaction costs are quantified according to the target operation to be performed. A pod initialization procedure requires about 42.6660 EUR (2703393 Gas units). Such a high cost is due to the deployment of the DTobligation smart contract, which is charged to data owners. Resource management functionalities turns out to determine the highest average per invocation result, with a mean value of 1.297857773 EUR (82234.5 Gas units). Differently, subscription management interactions require lower per invocation costs, with an average result of 0.97508665 EUR (61783.16667 Gas units). Obligation management operations are the least expensive for data owners, costing an average value of 0.845061703 EUR (53544.5625 Gas units) per invocation. Globally, considering all the functionalities, data owners spend an average value of 0.915300946 EUR (57995.04167 Gas units) for each smart contract invocation.

Since the thesis addresses the data ownership perspective of the application, the number of functionalities intended for data consumers is limited, and they mainly consist of Gas-free read-only functions that are not considered in the average calculation. The average expenditure per invocation in this case turns out to be slightly higher than data consumers, with an average value of 0.97508665 EUR.

In conclusion, the cost assessment pointed out rather high user fees. The infrastructure costs can be sustained by service providers thanks to revenues coming from DTsubscriptions. However, it is necessary to delineate a business model in which data owners generate a profit that overcomes interaction costs. A solution based on the number of accesses to shared resources may prove suitable for such purposes. However, cost reduction practices are necessary to increase market usability. These include changes to the overall architecture and on-chain code optimization operations.

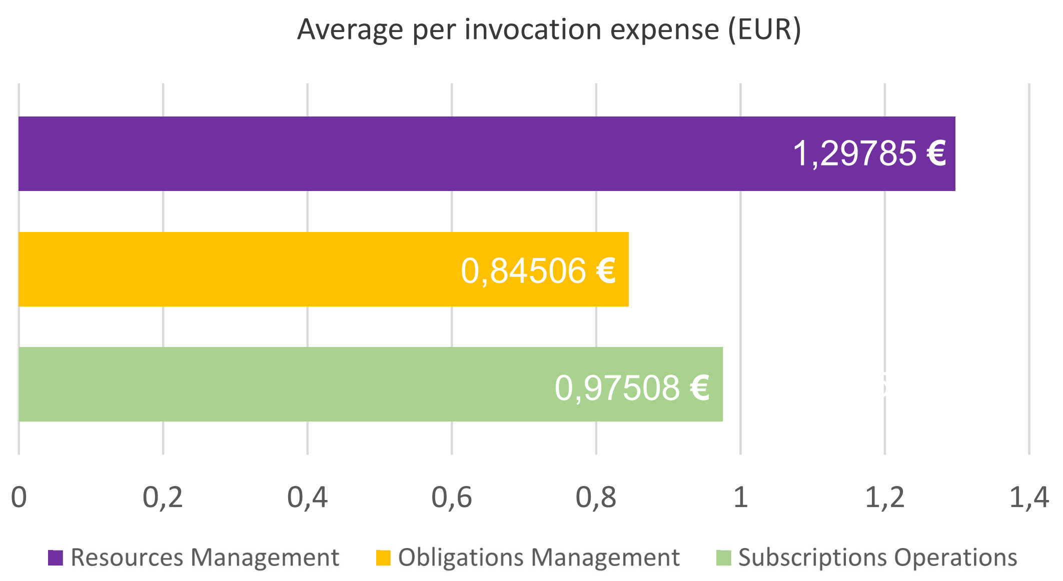


**Figure 6.1.** EUR per invocation expenditure comparison for data owners' routine operations.

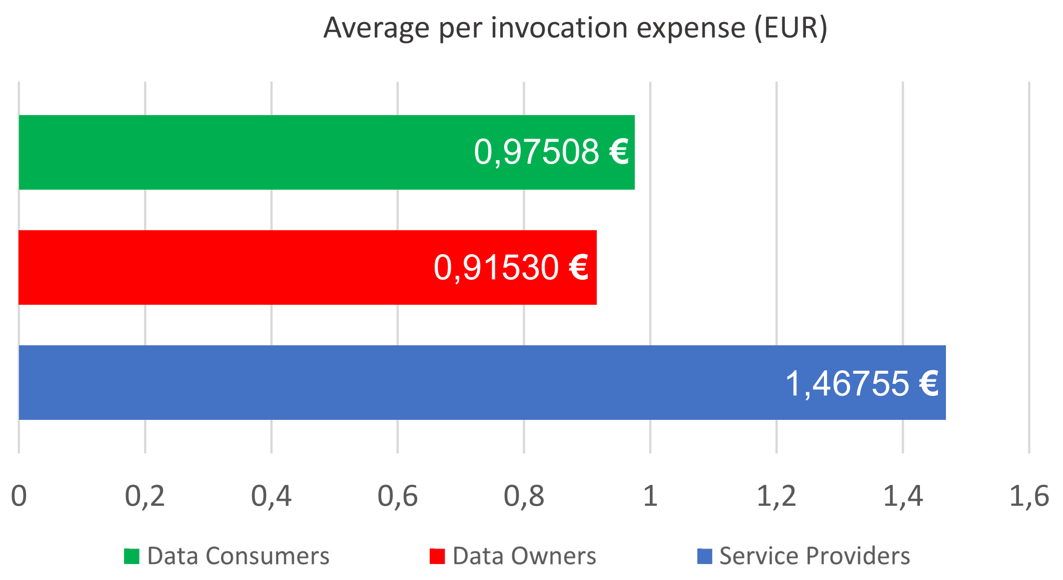


**Figure 6.2.** EUR per invocation expenditure comparison for data owners, data consumers and service providers.

# Chapter 7

# Conclusion and future work

The primary objective of the study conducted for the thesis was to draw attention to a question that the scientific community finds intriguing by offering a cutting-edge technology-based solution. Nowadays, blockchain technology is being widely adopted and finding application in many computer science fields. These technologies find a natural contextualization in the topic of digital markets, where the need for integrity and traceability is crucial.
The preliminary study of the state of the art was critical to achieving a full understanding of the theoretical representation of the problem. This phase of the study has been crucial for the in-depth study of issues such as big data, privacy preservation, and the decentralized web.
The design phase allowed us to establish the conceptual foundations upon which the DecentralTrading market was built. During this step, different alternatives concerning the market architecture have been theorised by distinguishing between the data owner and data consumer perspectives of the application. This resulted in the integration of the blockchain, pods and trusted execution environments.
The theoretical concepts defined by the DecentralTrading design were actualized in the implementation of functioning prototypes concerning the data owners' viewpoint. The implementation phase required the coordination of various technologies such as the Solidity and Python programming languages. For implementation purposes, the use of the Ganache and Remix IDE environments was essential. This phase led to the implementation of pod technologies to control market data and to interect with the on-chain infrastructure.
In order to discuss the prototype realized, the study included a quantitative evaluation whose main focus was costs. Indeed, such a theme represented the major critical issue for the system's usability. The evaluation involved the generation of more than 300 transactions, grouped according to their target user. Transaction costs were collected thanks to the Remix IDE Gas Analyzer tool, and their values were

carefully discussed. Through the assessment, the need for cost reduction practices and code optimization procedures were manifested.

The alternatives considered during the design stage allow us to outline a solid path for the DecentralTrading market's future improvements. Considerable changes can be introduced to the market architecture in order to improve its usability. The communication between on-chain and off-chain components will be changed, and the direct communication model will be turned into the Gas Station model. The decentralized off-chain network of relayers will permit users to interact with on-chain smart contracts without directly spending Gas/ETH. The transition to the Gas Station communication model will enable a more sustainable cost policy for data owners and data consumers and will improve the overall usability of the system.

The world of blockchain is constantly evolving, and the innovations being made in these areas may open up new opportunities for the market. The most significant change in the short term is the release of Ethereum version 2.0, which will result in the transition from the Proof of Work to the Proof of Stake consensus algorithm. The consensus mechanism change could significantly improve the performance and cost of the components running on the blockchain. However, the complete replacement of the blockchain environment cannot be excluded if the right conditions are met. So far, Ethereum has been the most appropriate choice for its smart contract programming versatility and support, although it does highlight limitations. Numerous blockchain environments, such as Algorand[1] and Hyperledger[2] are making significant progress in the field of smart contracts by adopting more sustainable fee policies. Comparative environmental procedures will be required in order to determine which the best solution for DecentralTrading is.

A key focus of future work will also be on improving the pod prototype implementation. Although the issues of off-chain authentication and data provision have already been extensively addressed, there remain other questions that need to be covered more specifically. First among these is undoubtedly the integration with the study inherent in data consumers and trusted execution environments, conducted in parallel with this thesis. Moreover, encryption mechanisms to protect pods' data within users' filesystem will be added to the implementation.

Finally, by including the results of the study inherent in data consumers, the design and implementation of different mechanisms for remunerating data owners based on accesses will be possible. This will also result in the implementation of new smart contracts that can model alternative business models supporting different subscription mechanisms.

[1]https://www.algorand.com
[2]https://www.hyperledger.org

# Bibliography

[1] Ines Akaichi and Sabrina Kirrane. “Usage Control Specification, Enforcement, and Robustness: A Survey”. In: *arXiv preprint arXiv:2203.04800* (2022).

[2] *Avarage Gas price estimation.* https://ycharts.com/indicators/ethereum_average_gas_price.. Accessed: 2022-09-25.

[3] *AWS Data Exchange.* https://aws.amazon.com/it/data-exchange/. Accessed: 2022-09-02.

[4] Shaun Azzopardi, Joshua Ellul, and Gordon J Pace. “Monitoring smart contracts: Contractlarva and open challenges beyond”. In: *International Conference on Runtime Verification.* Springer. 2018, pp. 113–137.

[5] Prabal Banerjee and Sushmita Ruj. “Blockchain Enabled Data Marketplace–Design and Challenges”. In: *arXiv preprint arXiv:1811.11462* (2018).

[6] Juan Benet. “Ipfs-content addressed, versioned, p2p file system”. In: *arXiv preprint arXiv:1407.3561* (2014).

[7] Tim Berners-Lee, James Hendler, and Ora Lassila. “The semantic web”. In: *Scientific american* 284.5 (2001), pp. 34–43.

[8] Vitalik Buterin et al. “A next-generation smart contract and decentralized application platform”. In: *white paper* 3.37 (2014), pp. 2–1.

[9] Vitalik Buterin et al. “Ethereum white paper”. In: *GitHub repository* 1 (2013), pp. 22–23.

[10] David Carl and Christian Ewerhart. “Ethereum gas price statistics”. In: *University of Zurich, Department of Economics, Working Paper* 373 (2020).

[11] Enrico Carniani et al. “Usage control on cloud systems”. In: *Future Generation Computer Systems* 63 (2016), pp. 37–55.

[12] Rubén Casado and Muhammad Younas. “Emerging trends and technologies in big data processing”. In: *Concurrency and Computation: Practice and Experience* 27.8 (2015), pp. 2078–2091.

[13] Chuan Chen et al. “A secure and efficient blockchain-based data trading approach for internet of vehicles”. In: *IEEE Transactions on Vehicular Technology* 68.9 (2019), pp. 9110–9121.

[14] Frank Frank Edward Dabek. “A distributed hash table”. PhD thesis. Massachusetts Institute of Technology, 2005.

[15] Erik Daniel and Florian Tschorsch. “IPFS and friends: A qualitative comparison of next generation peer-to-peer data networks”. In: *IEEE Communications Surveys & Tutorials* 24.1 (2022), pp. 31–52.

[16] *Data Marketplaces: What, Why, How, Types, Benefits, Vendors.* https://research.aimultiple.com/data-marketplace/. Accessed: 2022-09-02.

[17] *Datarade.ai.* https://datarade.ai. Accessed: 2022-09-02.

[18] Alfonso De la Rocha, David Dias, and Yiannis Psaras. *Accelerating Content Routing with Bitswap: A multi-path file transfer protocol in IPFS and Filecoin.* 2021.

[19] Natarajan Deepa et al. “A survey on blockchain for big data: approaches, opportunities, and future directions”. In: *Future Generation Computer Systems* (2022).

[20] *Digital Is Driving The Next Generation Of Data Marketplaces.* https://www.pitneybowes.com/content/dam/pitneybowes/us/en/white-papers/pitney-bowes-forrester-data-us.pdf. Accessed: 2022-09-02.

[21] *ETH/EUR exchange rate.* https://www.google.com/finance/quote/ETH-EUR.. Accessed: 2022-09-27.

[22] *Ethereum 2.0 updates.* https://ethereum.org/it/upgrades/.. Accessed: 2022-09-02.

[23] Arthur Gervais et al. “On the security and performance of proof of work blockchains”. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.* 2016, pp. 3–16.

[24] *Global Social Media Statistics.* https://datareportal.com/social-media-users.. Accessed: 2022-09-02.

[25] Vipul Goyal et al. “Attribute-based encryption for fine-grained access control of encrypted data”. In: *Proceedings of the 13th ACM conference on Computer and communications security.* 2006, pp. 89–98.

[26] Neville Grech et al. “Madmax: Surviving out-of-gas conditions in ethereum smart contracts”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–27.

[27] Roger Haenni. “White paper v15”. In: *Source: https://datum. org/assets/Datum-WhitePaper.pdf* (2019).

[28] Andrei Hagiu and Julian Wright. “When data creates competitive advantage”. In: *Harvard business review* 98.1 (2020), pp. 94–101.

[29] Ibrahim Abaker Targio Hashem et al. “The rise of “big data” on cloud computing: Review and open research issues”. In: *Information systems* 47 (2015), pp. 98–115.

[30] Donghui Hu et al. “A blockchain-based trading system for big data”. In: *Computer Networks* 191 (2021), p. 107994.

[31] *Iota Data Marketplace*. https://wiki.iota.org/blueprints/data-marketplace/overview.. Accessed: 2022-09-02.

[32] Hristo Koshutanski and Fabio Massacci. “An access control framework for business processes for web services”. In: *Proceedings of the 2003 ACM workshop on XML security*. 2003, pp. 15–24.

[33] Antonio La Marra et al. “Implementing usage control in internet of things: a smart home use case”. In: *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE. 2017, pp. 1056–1063.

[34] Aliaksandr Lazouski, Fabio Martinelli, and Paolo Mori. “Usage control in computer security: A survey”. In: *Computer Science Review* 4.2 (2010), pp. 81–99.

[35] Jingming Li et al. “Energy consumption of cryptocurrency mining: A study of electricity consumption in mining cryptocurrencies”. In: *Energy* 168 (2019), pp. 160–168.

[36] Ninghui Li and Mahesh V Tripunitara. “On safety in discretionary access control”. In: *2005 IEEE Symposium on Security and Privacy (S&P’05)*. IEEE. 2005, pp. 96–109.

[37] Hakan Lindqvist. “Mandatory access control”. In: *Master’s thesis in computing science, Umea University, Department of Computing Science, SE-901* 87 (2006).

[38] T Soni Madhulatha. “An overview on clustering methods”. In: *arXiv preprint arXiv:1205.1117* (2012).

[39] Antonio La Marra et al. “A Distributed Usage Control Framework for Industrial Internet of Things”. In: *Security and Privacy Trends in the Industrial Internet of Things*. Springer, 2019, pp. 115–135.

[40] Aashish Mehra. *Big Data Market worth $273.4 billion by 2026, 2020.* https://www.marketsandmarkets.com/PressReleases/big-data.asp. [Online; accessed 2-August-2022].

[41] William Metcalfe et al. "Ethereum, smart contracts, DApps". In: *Blockchain and Crypt Currency* (2020), p. 77.

[42] Du Mingxiao et al. "A review on consensus algorithm of blockchain". In: *2017 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE. 2017, pp. 2567–2572.

[43] Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. "An overview of smart contract and use cases in blockchain technology". In: *2018 9th international conference on computing, communication and networking technologies (ICCCNT)*. IEEE. 2018, pp. 1–4.

[44] Debajani Mohanty. "Ethereum Use Cases". In: *Ethereum for Architects and Developers*. Springer, 2018, pp. 203–243.

[45] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: *Decentralized Business Review* (2008), p. 21260.

[46] Solomon Negash and Paul Gray. "Business intelligence". In: *Handbook on decision support systems 2*. Springer, 2008, pp. 175–193.

[47] Cong T Nguyen et al. "Proof-of-stake consensus mechanisms for future blockchain networks: fundamentals, applications and opportunities". In: *IEEE Access* 7 (2019), pp. 85727–85745.

[48] Silas Nzuva. "Smart contracts implementation, applications, benefits, and limitations". In: *School of Computing and Information Technology, Jomo Kenyatta University of Agriculture and Technology, Nairobi, Kenya* (2019).

[49] Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah Ait Ouahman. "FairAccess: a new Blockchain-based access control framework for the Internet of Things". In: *Security and communication networks* 9.18 (2016), pp. 5943–5964.

[50] Jaehong Park and Ravi Sandhu. "The UCONABC usage control model". In: *ACM transactions on information and system security (TISSEC)* 7.1 (2004), pp. 128–174.

[51] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. "On data banks and privacy homomorphisms". In: *Foundations of secure computation* 4.11 (1978), pp. 169–180.

[52] Seref Sagiroglu and Duygu Sinanc. "Big data: A review". In: *2013 international conference on collaboration technologies and systems (CTS)*. IEEE. 2013, pp. 42–47.

[53] Andrei Vlad Sambra et al. "Solid: a platform for decentralized social applications based on linked data". In: *MIT CSAIL & Qatar Computing Research Institute, Tech. Rep.* (2016).

[54] Ravi S Sandhu. "Role-based access control". In: *Advances in computers*. Vol. 46. Elsevier, 1998, pp. 237–286.

[55] Ravi S Sandhu and Pierangela Samarati. "Access control: principle and practice". In: *IEEE communications magazine* 32.9 (1994), pp. 40–48.

[56] Gustavus J Simmons. "Symmetric and asymmetric encryption". In: *ACM Computing Surveys (CSUR)* 11.4 (1979), pp. 305–330.

[57] *Snowflake Marketplace*. https://www.snowflake.com/en/data-cloud/marketplace/. Accessed: 2022-09-02.

[58] *Solid*. https://solidproject.org/about. Accessed: 2022-03-30.

[59] Markus Spiekermann. "Data marketplaces: Trends and monetisation of data goods". In: *Intereconomics* 54.4 (2019), pp. 208–216.

[60] Sarah Spiekermann and Jana Korunovska. "Towards a value theory for personal data". In: *Journal of Information Technology* 32.1 (2017), pp. 62–84.

[61] Florian Stahl, Fabian Schomm, and Gottfried Vossen. *The data marketplace survey revisited*. Tech. rep. ERCIS Working Paper, 2014.

[62] Wei Tan et al. "Social-network-sourced big data analytics". In: *IEEE Internet Computing* 17.5 (2013), pp. 62–69.

[63] Paolo Tasca and Claudio J Tessone. "Taxonomy of blockchain technologies. Principles of identification and classification". In: *arXiv preprint arXiv:1708.04872* (2017).

[64] Sergei Tikhomirov. "Ethereum: state of knowledge and research perspectives". In: *International Symposium on Foundations and Practice of Security*. Springer. 2017, pp. 206–221.

[65] Sergei Tikhomirov et al. "Smartcheck: Static analysis of ethereum smart contracts". In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 2018, pp. 9–16.

[66] Huu Tran et al. "A trust based access control framework for P2P file-sharing systems". In: *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. IEEE. 2005, pp. 302c–302c.

[67] Nikhil Vadgama and Paolo Tasca. "An analysis of blockchain adoption in supply chains between 2010 and 2020". In: *Frontiers in Blockchain* 4 (2021), p. 610476.

[68] Martijn Van Otterlo. “A machine learning view on profiling”. In: *Privacy, Due Process and the Computational Turn-Philosophers of Law Meet Philosophers of Technology. Abingdon: Routledge* (2013), pp. 41–64.

[69] *WebID Specification*. http://www.w3.org/2005/Incubator/webid/spec/identity/.. Accessed: 2022-09-02.

[70] *WebID-TLS Specification*. http://www.w3.org/2005/Incubator/webid/spec/tls/.. Accessed: 2022-09-02.

[71] *What is digi.me?* https://digi.me/what-is-digime/. Accessed: 2022-03-30.

[72] *Who is Using Big Data in Business?* https://itchronicles.com/big-data/who-is-using-big-data-in-business/#:~:text=Despite%20the%20fact%20that%20only,at%20%2477%20billion%20by%202023.. Accessed: 2022-09-02.

[73] Wei Xiong and Li Xiong. “Smart contract based data trading mode using blockchain and machine learning”. In: *IEEE Access* 7 (2019), pp. 102331–102344.

[74] Shui Yu. “Big privacy: Challenges and opportunities of privacy study in the age of big data”. In: *IEEE access* 4 (2016), pp. 2751–2763.

[75] Peng Zhang et al. “Blockchain technology use cases in healthcare”. In: *Advances in computers*. Vol. 111. Elsevier, 2018, pp. 1–41.

[76] Kaspars Zıle and Renāte Strazdiņa. “Blockchain use cases and their feasibility”. In: *Applied Computer Systems* 23.1 (2018), pp. 12–20.