



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

MASTER THESIS IN COMPUTER ENGINEERING

A Decentralized Content Management System

MASTER CANDIDATE

Tommaso Baldo

Student ID 2007728

SUPERVISOR

Prof. Mauro Migliardi

University of Padova

ACADEMIC YEAR 2021/2022
DECEMBER 5, 2022

A mio nonno Argeo

Abstract

Nowadays online presence is a must for every business. Nevertheless, many small and medium enterprises struggle to be online due to a lack of expertise and economic investments. A solution is creating a single and organized access point where similar enterprises can aggregate. On top of that, it can also offer better brand awareness and exposure on a national and international scale. Thanks to decentralization, it is possible to remove any middle-person, such as a webmaster, and avoid an expensive cloud solution. At the same time, responsibilities have to be split among users: they are in charge of maintaining and sustaining a common platform.

Finally, the goal of my work is to design and create a decentralized application where similar SMEs can: aggregate, share their products, and get rewarded for having an active role in the community. These core functionalities are achieved by the combination of Ethereum and Swarm. Thanks to smart contracts, it is possible to create an automated set of rules and a tokenomics to reward the best users. Ethereum also handles user authentication. Instead, Swarm hosts the website and serves as distributed storage system to save product images and enterprise logos.

Sommario

Al giorno d'oggi la presenza online è fondamentale in ogni business. Nonostante ciò, molte piccole e medie imprese sono scoraggiate da diversi ostacoli, come la mancanza di conoscenze o dei fondi necessari per poter essere online. Una possibile soluzione è creare una singola piattaforma dove unire imprese simili, per tipo di prodotto o per target di clientela. Così facendo, ogni azienda partecipa e gode della visibilità nazionale e internazionale della piattaforma. Grazie alla decentralizzazione, è possibile rimuovere figure terze, come ad esempio un webmaster, o evitare di ricorrere a costose soluzioni in cloud. Allo stesso tempo però gli utenti sono chiamati ad essere attivi nella gestione della piattaforma per garantirne il mantenimento. Concludendo, il mio lavoro mira a progettare e realizzare un'applicazione decentralizzata dove le imprese possano aggregarsi, condividere i propri prodotti e ottenere ricompense se attive nella gestione della piattaforma stessa. Queste funzionalità sono ottenute dalla combinazione di due tecnologie: Ethereum e Swarm. Grazie alla prima, è possibile realizzare un insieme di regole automatizzate e un sistema di ricompense per gli utenti più attivi. Swarm invece funge da web host e allo stesso da sistema di storage distribuito dove salvare le immagini dei prodotti e i loghi delle aziende.

Contents

List of Figures	xi
List of Code Snippets	xiii
List of Acronyms	xv
1 Introduction	1
2 State of the Art	5
2.1 An Overview	5
2.2 Open Issues	8
2.3 Data Persistence	9
2.4 URL Mapping	10
2.5 Content Visibility and Social Incentives	12
2.5.1 Steemit	13
2.5.2 Curation Markets	13
2.5.3 Ethernas	14
2.5.4 D.tube	14
3 Ethereum	17
3.1 The World Computer	17
3.2 Smart Contracts	19
3.3 Consensus Mechanism	21
3.4 Ethereum Name Service	24
4 Ethereum Swarm	27
4.1 Overlay network	27
4.1.1 Kademlia Connectivity	28
4.1.2 Storage Model	29

CONTENTS

4.2	Incentive System	32
4.2.1	Bandwidth Incentives	32
4.2.2	SWAP	34
4.2.3	Storage Incentives	36
5	CMS Design and Implementation	39
5.1	Requirements	39
5.2	Stack Design	42
5.2.1	Back-end Development	43
5.2.2	Enter the blockchain	45
5.2.3	Accessing the Swarm	46
5.2.4	Front-end interface	49
5.2.5	Entering the website	50
5.3	Test Environment	54
5.4	Tokenomics	56
5.4.1	Token Design	57
5.5	Implementation	59
5.5.1	Back-end code	60
5.5.2	Front-end code	75
5.5.3	A Meaningful Example	78
6	Conclusions and Future Works	81
	References	83
	Acknowledgments	89
A	Smart Contracts	91
A.1	Migrations.sol	91
A.2	SwarmAd.sol	92
A.3	SwarmAdGovernor	101
A.4	SwarmAdRewarder	108
A.5	SwarmAdReputationPoints	110
B	Testing scripts	113
B.1	SwarmAdTest.js	113
B.2	GovernorTest.js	115

C Front-end	119
C.1 Home	119
C.2 Register	120
C.3 CreateItem	123
C.4 Showcase	127
C.5 EnterpriseShowcase	128
C.6 SwarmAd.js	129
C.7 SwarmClient.js	131

List of Figures

3.1	Example of GHOST fork choice rule [43]	24
3.2	ENS resolving name mechanism [6]	25
4.1	Comparison between IPFS and Swarm retrieval system [51]	29
4.2	A Binary Tree Chunk [51]	31
4.3	Feed structure [51]	32
4.4	Downloading of a chunk	33
4.5	Uploading of a chunk [51]	34
4.6	Basic functioning of SWAP protocol [51]	35
5.1	dApp architecture	44
5.2	How a user will connect to the app	53
5.3	How a node owner will connect to the app	53
5.4	Bee-factory containers	55

List of Code Snippets

4.1	"Console interaction with stamps endpoint"	36
5.1	"A simple example of Bee-js functionalities"	50
5.2	"Updating the web interface via a Swarm Feed"	51
5.3	"Pre-computing addresses"	61
5.4	"Access control in SwarmAd"	62
5.5	"Deploying smart contracts"	63
5.6	"Defining Enterprise"	64
5.7	"Querying Enterprise struct"	64
5.8	"Defining Product"	65
5.9	"Create a new product"	65
5.10	"Superlike function"	66
5.11	"Defining Poll"	68
5.12	"Calling Governor to create a poll"	69
5.13	"Moving an Enterprise from waiting list"	70
5.14	"Procedure to close the poll"	71
5.15	"Making ERC20 token non-transferable"	72
5.16	"Rewarder computes interests"	74
5.17	"Minting and burning RPs"	74
5.18	"Redeeming Community Token"	75
5.19	"Setting up the Home"	76
5.20	"Querying SwarmAd"	77
5.21	"Upload an image via web interface"	78
5.22	"Create an enterprise via web interface"	78
A.1	Migrations.sol	91
A.2	SwarmAd.sol	92
A.3	SwarmAdGovernor.sol	101
A.4	SwarmAdRewarder.sol	108

LIST OF CODE SNIPPETS

A.5	SwarmAdReputationPoints.sol	110
B.1	SwarmAdTest.js	113
B.2	GovernorTest.js	115
C.1	Home.jsx	119
C.2	Register.jsx	120
C.3	CreateItem.jsx	123
C.4	Showcase.jsx	127
C.5	EnterpriseShowcase.jsx	128
C.6	SwarmAd.js	129
C.7	SwarmClient.js	131

List of Acronyms

ABI	Application Binary Interface
BMT	Binary Merkle Tree
BLS	Boneh-Lynn-Shacham
BNS	Blockchain Naming System
CAC	Content Addressed Chunk
CID	Content Identifier
CLI	Command Line Interface
CMS	Content Management System
CT	Community Token
CRUD	Create Read Update Delete
DAO	Decentralized Autonomous Organization
dApp	Decentralized App
DeFi	Decentralized Finance
DHT	Distributed Hash Table
DISC	Distributed Immutable Store of Chunks
DOSN	Decentralized Online Social Network
EIP	Ethereum Improvement Proposal
ENS	Ethereum Name Service

LIST OF CODE SNIPPETS

EOA Externally Owned Account

ERC Ethereum Request for Comments

EVM Ethereum Virtual Machine

GHOST Greediest Heaviest Observed SubTree

IPFS InterPlanetary File System

IPNS InterPlanetary Name System

NFT Non Fungible Token

PO Proximity Order

PoS Proof-of-Stake

PoW Proof-of-Work

SME Small Medium Enterprise

SOC Single Owner Chunk

SPA Single Page Application

SWAP Settle With Automated Payments

SWCT SwarmAd Community Token

SWRP SwarmAd Reputation Point

RBAC Role-Based Access Control

TLV Total Locked Value

TTL Time To Live

UI User Interface

VP Voting Power



Introduction

Web3 is gaining more and more interest among Internet users. The idea of building a decentralized web is catching because it means moving away from single central authorities. Decentralization cannot be achieved by a single and atomic task but we have to reach several different goals, such as giving data ownership back to the users or creating a censorship-free web [1]. Obviously, it is not a trivial mission: developers need tools such as blockchains or distributed storage systems onto which they can build their Decentralized App (dApp)s. On top of that, these applications must reach certain popularity among users. This is why as a direct consequence of Web3 hype, the technology stack to build dApps has evolved quickly and the market demand has grown as well. This is highlighted by the Total Locked Value (TLV) of Web3-related smart contracts, which reached a peak of 193.14 billion USD in December 2021 [2]. Even if a standard in Web3 development hasn't settled yet, many technologies succeeded to reach a "production-ready" state or at least a more mature build in comparison with previous years.

Under these considerations, it is clear that the view on Web3 has changed. Starting from the decentralized Content Management System (CMS) designed and developed by Martini [3], the goal of my thesis is to improve it by adopting solutions that were not feasible in 2019 because of the lack of maturity of some technologies.

After a deep and accurate analysis, I found the following three main issues:

1. Data Persistence: InterPlanetary File System (IPFS) doesn't ensure the long-term persistence of data we upload on our platform

2. Naming: because of the decentralized structure, it is difficult to provide a secure and human-readable point of access
3. Content visibility: lacking a central authority, we cannot suggest content as in Web 2.0, where an algorithm is in charge of what we will see. We need an incentive system to push quality content in a fair and rewarding way for both creators and users.

The proposed solutions for issues (1) and (3) will share the same line of the reasoning we can see in Martini's work: members of the community must be responsible for maintaining and moderating it, creating a loop where seeking personal advantage leads to generating an advantage for the whole community. In other words, we will build an incentive system to reward good behavior. Long-term persistence is fundamental to making our social platform fair. Indeed IPFS, like any other distributed storage which doesn't provide a storage incentive, saves data like in a cache [4]. As a consequence, popular data are more likely to be stored while unpopular one could be very rare or even lost. Without storing incentives, the threat of building an amnesic and fragmented Web is real.

As we will see in further chapters, the solution I have chosen for data persistence is adopting Ethereum Swarm as a storage layer [5]. It is a decentralized storage system with built-in incentives for forwarding and storing data. As a consequence, it has a self-sustainable economy where nodes are rewarded for being active and it is also able to ensure long-term persistence.

The solution to issue (2) is straightforward and it relies on the interaction between Swarm Feeds, a special type of memory chunk we will describe later, and Ethereum Name Service [6]. Thanks to this combination, we are able to link what we store in Swarm to a human-readable domain. This could seem like a little change but it is an important step forward in accessibility and user experience.

As far as the last issue is concerned, the idea is to create a *tokenomics*. By adding a utility token, we can measure how much a user is contributing to community wellness. Accumulating this token, users will be able to obtain exclusive services or rewards. A sort of social incentive to keep users involved in community moderation, content filtering and selection.

As the final result of my work, I designed and implemented a dApp based on Swarm as storage layer and Ethereum blockchain for user authentication. It is a social portal to aggregate Small Medium Enterprise (SME)s and make them cooperate to maintain and sustain a common platform where they can advertise

their products. Consequently, this dApp removes the entry barrier made by economic investments and technological skills required by building and maintaining a traditional website. On top of that, it offers better brand awareness and better exposure on a national and international scale by creating a single and organized point of access for end users. This is particularly helpful for micro firms located in rural contexts [3].

My work is organized in the following way:

- Chapter 2 is to describe the actual state of the art, i.e. what has been implemented and especially what has been inspirational to my work.
- Chapter 3 and Chapter 4 present the fundamental blocks to understand my work: Ethereum, Swarm and their related technologies such as ENS, ERC-20 Token, and Swarm Feeds.
- Chapter 5 describes my design process and how I managed to implement all the desired functionalities.
- Chapter 6 finally sums up my work and it shows in which direction we could move on to improve the project.



State of the Art

A blockchain is not efficient when dealing with large files like images or videos because of writing time and transaction costs. Developers have been encouraged to explore off-chain solutions to upload files. The mixture of on-chain and off-chain storage established as a common design pattern for dApp development in the recent years [7]. To support this claim, in this chapter, we are going to explore some relevant examples of dApps based on a blockchain and a distributed storage. In the first section, we will examine two decentralized content management systems similar to my project. Then in the further sections, we will explore several examples that could be useful to solve the problems underlined in Chapter 1.

2.1 AN OVERVIEW

As first example, we have HiDe. It is an offline first decentralized CMS with a sustainable and permission-less reward distribution protocol called HiDe Protocol. Starting from HiDe open-source codebase, it is possible to build social media dApps where users showcase their posts or articles and get rewarded for activities and contributions.

Hide social media app works as follows. Users' content is stored in Ceramic, a distributed storage built atop of IPFS. Whenever a user creates a new post, it is uploaded to Ceramic, and the reference is saved on Polygon. Polygon is a layer 2 blockchain built on top of Ethereum, so it provides more scalability and

2.1. AN OVERVIEW

fewer transaction costs. Meanwhile, it keeps Ethereum advantages in terms of security. As a result, each user has a list of posts or articles associated with his account which is saved on blockchain. Besides, HiDe provides integration with the most famous blogging platform such as WordPress or Medium, so a user can post or export articles to and from these platforms.

HiDe protocol implements a complex token economy to make this entire system self-sustainable. It is based on five different entities:

- **Treasury:** it is a smart contract that generates the reward pools from deposits staked by users who are supporting HiDe.
- **Voting Power (VP):** staking produces VP tokens. VPs can be spent to vote for budget allocation or other relevant decisions about the organization.
- **Community Token (CT):** it is based on ERC-20 standard but it also decreases automatically block by block. Whenever a certain budget is allocated to a community, it gets converted into a certain amount of CTs. CTs can be burned to reward an NFT or can be converted into community share.
- **community share:** users who own community shares get a dividend when budget is allocated to their community
- **kudos:** they can be accumulated by making contributions in a community. They can be used in community voting.

Finally, HiDe is mainly focusing on becoming a community-building tool where social media features help users to bond and reward system motivates content creators to produce quality content [8].

As a second example, we have Helios. It is a project founded by the European Commission to provide a platform to develop decentralized social networks apps for Android. Its primary focus is on privacy, ownership and protection of users' personal data and content.

It is a modular platform composed of three layers: Helios core, extension modules and applications [9]. Helios core has different components. Each one provides a specific basic functionality. For example, there is the communication manager which provides the access to p2p network and it includes basic messaging functionalities. There is also a security manager which handles access policy and privacy settings.

Then we have extension modules. They can be developed also by third-parties. Their scope is to provide additional functionalities over Helios core. For example, implementing a media streaming player or graph mining algorithm for

content recommendation. Among several extension modules, we can find the reward system. It aims at incentivising users to be active on the platform. The whole tokenomics is built around a single utility token, called Helios Token or HLO. Users can collect HLOs by performing simple actions like creating and sharing content, upvoting and downvoting. Then HLOs can be spent to access premium content, unlocking premium features or purchase a third-party service. As a last thing, it is interesting to notice how the token distribution is organized in Helios. Every day, an algorithm determines the size of the daily reward pool, i.e. the predetermined number of HLO to distribute among active users. Then the relevance of each user is calculated. Every action, as a like or a comment, has a specific weighted score s . User contribution is the sum of all his action weighted scores. The user relevance is computed as the ratio between a user's contribution and the contribution of all users. The daily reward pool is then split between users according to their relevance [10].

Finally, we have the application layer. Developers can take advantage of the functionalities provided by Helios core and extension modules to create a social network service working on Android. As an example, in Helios ecosystem we can find `helios.talk`, an app that allows users to communicate in a secure and decentralized way. Another one is `helios.CJReporter`. It is an app to share videos anonymously. The videos are stored on IPFS and the access control is managed by an Ethereum smart contract.

If we move to more specific use cases, we can find Opus [11] and Audius [12]. Both two are decentralized music sharing platforms with the goal of ensuring a fair compensation for artists. Their structures follow this design pattern: they rely on a blockchain for user authentication while they use a distributed storage system, IPFS, for storing songs and other data relevant to platform workflow. On top of that, both have a community token to motivate users' participation, as artists and also as simple end-users. For example, Opus rewards users who create popular playlists. As we will see in a further section, content curation is particularly relevant in a dApp since it is more difficult to build a content recommendation system than in traditional web2 applications because of decentralization.

2.2 OPEN ISSUES

The previous examples confirm the effectiveness of the *hash-on-chain* pattern. Indeed, it allows for reducing the expenses due to gas fees and the time in writing and reading the blockchain. Recalling HiDe and Helios, we can see their main focus is on social functionalities, like commenting or sharing a post. As a direct consequence, content creation is the beating heart of both platforms because it feeds a whole set of social functionalities. This claim is confirmed in the respective reward systems. As an example, Helios awards 10 points for creating a post while the reward for sharing is 2.

If we now consider our use case, i.e. a platform where enterprises can post and advertise their products. It is clear how Helios and Hide reward systems are not a good fit. As the first thing, we are in a competitive scenario where it is unlikely that users are willing to share others' content. Indeed, they are competitors so sharing a post is like gifting free advertising to a market rival. Secondly, content creation in social networks is far away from the one we will see on my platform. To be clear, an enterprise has to post according to its production rate which depends on the availability of employees, machinery, and raw materials. Following this reasoning, it is important to not push for content creation. Another difference is that HiDe and Helios do not face community moderation. This claim is confirmed by their reward systems. The lack of a moderation mechanism can be justified because in social media, especially if it is decentralized, it is important to ensure a censorship-free system where every user is free to share his opinions. Nevertheless, in my project, community moderation is crucial to keep a clean and working platform. Indeed, it is oriented to serve the needs of all the possible niche markets and it is important to remove off-topic content or inappropriate users. Obviously, since the platform is meant to be a showcase for each enterprise, users are motivated to keep the highest reputation possible. Indeed, if the platform has a bad reputation, the latter will affect their enterprises as well. Concerning user moderation, there is another big difference between my dApp and social media apps. Generally speaking, these last provide a free registration mechanism where everyone can sign up. HiDe and Helios too. In my platform, every user has to meet a set of requirements to be accepted in the community. The decision about accepting or rejecting a newcomer is in the hands of the registered members.

For what concerns the technical side, Helios is more focused on mobile develop-

ment. In our case, we think a progressive web app is a better solution because it eases accessibility and hence it maximizes the number of visitors. Since we are speaking about an advertising platform, it is a relevant aspect without any doubt. Other technicalities, such as the choice of a certain distributed storage system or the issue of content visibility, will be delved into later sections. To sum up, I think I am proposing a platform that has different social dynamics from the traditional applications, hence the moderation mechanism is way more relevant. In comparison with Martini, I think that the most efficient way to create an effective moderation mechanism is to build a tokenomics to reward users.

2.3 DATA PERSISTENCE

In Martini's design, the storage layer relies on IPFS [3]. It is a distributed system for storing and accessing files and data [4]. It is also the first protocol belonging to the so-called "next generation" of P2P data networks [13].

As stated before, IPFS doesn't ensure data long-term persistence. After this issue was identified, several way around tried to patch it. For example, the *pinning* service. By pinning a file, a node marks it as important and it keeps that file in its local storage. Consequently, the pinned file is surely available when the node is online. Martini himself implemented this service to mitigate the problem. *Pinning* may not be sufficient to ensure long-term persistence, especially if we rely on a small number of nodes as in our use case [14]. This is the reason why many *pinning* services on top of IPFS have been developed. These services provide a host node on a cloud service to store and pin data. However, they are not free and they are somehow in conflict with the concept of decentralization since users have to trust and rely on a third-party actor.

If we want to remain in the neighborhood of IPFS, we have to consider FileCoin. It is a protocol for storing and sharing data in peer-to-peer networks [15]. Basically, it works as a storage marketplace where nodes can rent their storage in change of money. All the payments are performed in form of FIL, which is FileCoin cryptocurrency. The economic incentive in form of micropayments ensures long-term persistence.

It is important to notice that FileCoin and IPFS are fully complementary and they can also be used together. A service of this kind is provided by Powergate, an API-driven solution built by Textile [16]. Regarding this approach, IPFS is

2.4. URL MAPPING

in charge of data storage and chunk retrieval while FileCoin is for ensuring a long-term backup. If we keep in mind our use case, Powergate is the only feasible solution where we can implement FileCoin because the latter alone would not be able to perform efficiently due to its large retrieval time. Indeed storing, verifying and unsealing data are time-expensive operations that need a certain computational force [15].

The other option is Ethereum Swarm, a decentralized storage system. It is similar to IPFS since both rely on *libp2p* and *Kademlia*. It offers two features that could be useful to our platform: storage incentives and mutable content address. Back in 2019 when Martini developed the first version of EtherAd [3], it was a promising but still premature protocol. After three years the Swarm team is still developing new features and improving the existing ones but the status of the Swarm network is mature: it counts an average of 2000 active nodes all around the world [17]. The increasing number of dApp built on Swarm proves the last claim. In Chapter 4 we are going to have a closer look at Swarm technology stack.

2.4 URL MAPPING

Surfing the centralized web is straightforward. It has a location-based addressing system hence we just need a valid URL to get the desired web page. When we rely on a distributed storage like IPFS or Swarm, we move to content-based addressing because files can be accessed by their corresponding hash identifier. In such systems, the identifier is generated on the file or folder's hash. Consequently, a change in a file leads to a change in the content address. As a result, it is not possible to rely on an identifier as an access point to our website since every update will lead to a change in the hash value.

The naming issue is well-known in decentralized web literature and it is often referred to as "Zooko Triangle" [18]. In 2001 the American computer security specialist Zooko Wilcox-O'Hearn claimed it was not possible to create a naming system able to ensure at the same time the following three attributes: decentralized, secure and human-readable.

Before checking if this claim still holds, we have to consider the current options we have:

- **Mutable address:** data structures like InterPlanetary Name System (IPNS) or Swarm Feeds allow creating address pointing to mutable data [19]. For

example, in IPNS a name is the hash of a public key. Once we create a name, we can make that name point to the Content Identifier (CID) of the latest version of your website. However, IPNS doesn't provide a human-readable name. As we will see in Chapter 4, Swarm Feeds work in a similar way.

- **DNSLink:** it is another solution proposed by IPFS Foundation to overcome the readability issue of IPNS. It uses TXT records to map a DNS name [20]. A DNS record is editable at any time so we can make it pointing always to the latest version. However, it is important to notice it is relying on DNS servers so this solution is not fully decentralized and censorship-free. Indeed central authorities, like ICANN, manage DNS root [21].
- **ENS (Ethereum Name Service)** is a distributed naming system based on Ethereum blockchain [6]. It works by translating human-readable names into Ethereum addresses. The latest can point to a specific CID or Swarm hash.
- **Handshake:** it is a naming protocol to manage the registration, renewal and transfer of DNS top-level domains (TLDs) [22]. It is a blockchain-based since it relies on a utility coin system for name registration.

As it is easy to notice, "Zooko Triangle" applies to IPNS and DNSLink, which are not human-readable and decentralized, respectively. As stated by Aaron Swartz, a Blockchain Naming System (BNS) can solve this trilemma [23]. Indeed, once a register is saved on the blockchain is impossible to change the previous records because of the amount of computational power required to perform the attack. Starting from Swartz's idea, it has been developed Namecoin, the first BNS able to solve the triangle.

Going back to our problem, ENS is the best option because it is built on Ethereum and it is also well integrated with IPFS and Swarm. Nevertheless, we can achieve an even better solution. Indeed, if we are using ENS alone, we have to pay gas fees for every update because we have to make the domain point to the latest website identifier. This could be seen as a minor drawback but in the long run, it could lead to significant expenses.

The suggested solution is to adopt a mutable address, such as IPNS or Swarm Feeds, and then the latter will link to an ENS domain. Thanks to this combination, we are able to obtain a human-readable link and avoid blockchain interaction at every content update. In Chapter 4 we are going to see Feeds in detail.

2.5 CONTENT VISIBILITY AND SOCIAL INCENTIVES

When we are using a social network relying on a centralized architecture such as Facebook, Reddit or Youtube, the content we are going to see in our homepage is meticulously selected by an algorithm. It usually collects users' data during the time spent on the platform itself and then it processes it to propose the most relevant and interesting content. On top of that, an algorithm of kind is most likely to favor viral and attention-grabbing content in order to gain more and more interaction from the users [24].

One of the principles of the decentralized web is to give ownership back to the users so things get very different. Indeed there is no central entity capable of having control of all users' data, using data for commercial purposes or changing unilaterally the existing terms of service.

If well designed, decentralization is able to ensure data privacy and security but at the same time, it introduces new challenges in data availability and content management. This issue is crucial in Decentralized Online Social Network (DOSN), where it gets tough to manage and propagate the data users create, update, and exchange [25]. This is the main reason why I analyzed several DOSN and micro-blogging platforms to understand how they managed to overcome the problem.

In this kind of platforms, it is widely spread the implementation of micropayments to users. As a consequence, content creators are encouraged to publish high quality content while users can profit from producing reasonable and valuable comments which add extra value to the article or post itself. The idea is to shift from the advertising-centric model, typical of web 2.0 where centralized systems are sustained by the revenue produced by ads, in favor of a user-centric model.

Generally speaking, the reward is a utility token. It is an asset intended to provide access to a service, a benefit, or a reward [26]. For example, LikeCoin is a protocol based on IPFS and Ethereum blockchain. Its functioning is pretty straightforward: whenever a user likes a post, the creator gets a certain amount of LikeCoin. In opposition to traditional social networks, it is possible to monetize LikeCoin.

As another example, Peepeth is a micro blogging platform based on Ethereum and IPFS. Its exclusive feature is that each user has only one like per day so its meaning is way bigger than in traditional social networks, where we are used to

liking several posts or photos daily. As a result, it generates a sense of exclusivity.

2.5.1 STEEMIT

Steemit is another famous example and it is the first blockchain-based blogging platform implementing an economic reward via cryptocurrency [27]. Its token business is designed around the content posted by users. By creating quality post or taking part in community moderation, a user is rewarded in STEEM, i.e. Steemit cryptocurrency. On top of that, Steemit recognizes a special weight to users who have invested a significant value in a long-term commitment. Basically, if a user commits his STEEM in a thirteen week vesting schedule, he will convert them into Steem Power (SP). As it is logical to think, users who invested more, have more interest in making Steemit grow healthily. As a matter of fact, Steemit decides to empowers users according to their economic investment [27]: each user's influence over reward distribution is directly proportional to the amount of SP he owns [27]. This business model gives users an incentive to create traffic and contribute to the community, ensuring the platform's sustainability at the same time [28]. However, it opens an ethical question. It is important to balance the influence of each user fairly to avoid creating a sort of aristocracy where a few users can directly influence the whole community, as Steemit itself is risking [29]. For example, since our project is about an advertising platform, it is crucial to give both old and new users their opportunity to shine.

2.5.2 CURATION MARKETS

Curation Markets is a smart contract built on Ethereum to reward curating and moderating in blogs and forums. The idea is to create a token for each topic. Each user who is creating or contributing to quality content gets rewarded by the token related to the topic. As it is logical to think, the better the content the more the platform will be able to attract users [30]. In our use case, it could work for rewarding topic filtering, moderation and, content ranking.

Innerlight is a decentralized discussion platform about mental health. It is built on IPFS and Ethereum and it has its own implementation of Curation Markets [31]. Its token is *LightCoin*. For example, it can be used to upvote a comment.

2.5. CONTENT VISIBILITY AND SOCIAL INCENTIVES

Once an algorithm picked the best answer, the user who wrote it and the users who upvoted it get rewarded. In this way, the Curation Markets system rewards the one who created the content but also the other users who took part in the discussion. As a result, a certain quality standard should be ensured.

2.5.3 ETHERNA

Etherna is a decentralized video platform based on Ethereum and Swarm [32]. In its architecture, we can find two interesting implementations of Swarm functionalities. The first one is the *Etherna Index*. An index is a data structure that contains references for each content that has been uploaded to the platform. The aim is to create and maintain an index for each topic or category, then assign a weight to each video accordingly to quality parameters. As a result, an index is a weighted list of videos. Following this approach, the content is internally organized and it is easier to select and show the best content the platform can propose. Thinking about our use case, it would be interesting to add indexes to collect products or enterprises belonging to a common theme or category. In this way, the website content could be better arranged and easier to explore.

On top of that, Etherna introduces also the concept of *Frames*. By definition, a frame is linked to a topic and it has a self-managed community in charge of moderating and managing the content. Following the same concept of *Indexes*, *Frames* aims to arrange videos and split them into smaller groups to make moderation activities easier. It is important to notice that every user can create his own frame and also that a video can be inserted in multiple frames.

The last relevant implementation is about Swarm Feeds. Each creator has his own Feed and any new content will be uploaded to the Feed since it is all managed at the network level. In this way, Etherna offers an easy way to access a specific creator. It is a sort of Youtube channel page.

2.5.4 D.TUBE

As the last example, we have another video streaming platform: D.tube. It is built atop of Avalon blockchain and it relies on IPFS as distributed storage. The interesting thing about D.tube is its tokenomics. It has a basic value, called D.tube Coin (DTC). Staking DTC produces VP. VPs then can be spent to vote a video. Since the content exposure is based solely on upvotes, downvotes and

tags, VPs have a direct influence on content exposure. On top of that, D.tube proposes a gamification scheme for curation activities. When a user votes a content using VPs, it is like he gets a percentage of video revenues because he is endorsing it and hence he is contributing to its popularity. As a consequence, if the video goes viral and earns popularity among the community, i.e. it gets voted by other users and it earns VPs, voting users will earn a certain amount of DTC proportional to the VPs spent on their votes. In other words, it is like users can bet on the content they think will become popular. If the guess is right, they will earn. Otherwise, they will lose their VPs. At the same time, creators are rewarded in DTC according to the amount of VPs a video gets. Finally, D.tube has a self-sustainable economy where creators are motivated to produce popular content and users are also rewarded for taking part in community moderation [33].



Ethereum

Ethereum is a cornerstone of my dApp since it handles all the business logic. Also, the naming system and the distributed storage that I have chosen to implement in my project are built on top of Ethereum. Consequently, it is important to revise some basic concepts about blockchain and Ethereum to better understand the whole project architecture.

3.1 THE WORLD COMPUTER

Decentralized digital currency has been a research topic from early 80s, but it became a real thing only several decades later when Satoshi Nakamoto created Bitcoin in 2009. Bitcoin is a peer-to-peer electronic cash system based on its homonymous cryptocurrency, *bitcoin*.

Nakamoto's mission was allowing digital payments between two parties without involving any financial institutions. The innovative idea was relying on a distributed ledger, called blockchain, to keep track of the collection of all bitcoin, also referred as *state*.

Blockchain is an ever-growing sequence of blocks where each block has a list of transactions and it is linked to the previous one. Transactions are simply transfer of bitcoin from a user to another. In other words, we can say a transaction updates the network state since it changes the coin distribution. On top of that, Bitcoin also allows to attach a script in every transaction. A script is a list

3.1. THE WORLD COMPUTER

of instructions where the sender can specify how the receiver can gain access [34]. It enforces an agreement between two parties by adding predefined and automated clauses.

However, Bitcoin scripting language is not Turing-complete. This is a barrier against writing complex programming functions. For example, it is not possible to write a loop cycle. Moreover, the Bitcoin state can hold only one information: if a coin is spent or unspent. Another internal state cannot be kept [35].

In 2013, Vitalik Buterin created Ethereum with the mission of overcoming these two problems and hence allowing developers to create their own rules for ownership, transaction formats and state transition functions. As Bitcoin, Ethereum is based on blockchain technology, but the key difference is that Ethereum is a distributed state machine rather than a simple distributed ledger. Indeed, it acts like a single computer where it is possible to run software programs able to define how the network state changes from block to block. Ethereum distributed computer is called Ethereum Virtual Machine (EVM) and it is also referred as "The World Computer".

It is important to define what is the state in the Ethereum network and understand how it is different from Bitcoin one. Ethereum state is a combination of accounts and state transitions to transfer an economic value or information between accounts.

An Ethereum account is identified by a 20-bytes address and it is made of four values: nonce, ether balance, contract code (optional) and storage. *Ether*, or ETH, is Ethereum native cryptocurrency. On top of that, we can distinguish two different types of accounts: externally owned and contract. The first one is controlled by a set of private keys and it usually belongs to a human being. The second one instead is automated because it is controlled by contract code, as we are going to delve into the next section.

If a Externally Owned Account (EOA) wants to change the network state, it has to initiate a transaction, which is a signed data package. For example, the simplest transaction is a transfer of ether from an account to another. A transaction has always a related cost, called *fee* or *gas fee*, which quantifies the computational effort required to execute the related operation on the Ethereum network. The fee price increases according to the congestion level of the network, but it is always possible to predict it. Indeed, a fee is made of a *base* and *priority*. The first value depends on previous blocks and it is burnt after the block is minted. The second one instead is an optional parameter to speed up the minting operation.

In other word, an account can pay an additional amount of ether to incentive nodes to mint that block. As last thing, this fee mechanism serves in preventing spam and to avoiding computational wastage. Indeed, each transaction has a parameter, *startGas*, which specifies the limit of computational steps. In this way, it prevents infinite loops due to bad programming or a bug in the code [35].

3.2 SMART CONTRACTS

A smart contract is a self-executed digital agreement. This concept is way older than Bitcoin and Ethereum. It was formulated in 1994 by Nick Szabo, a legal scholar and a computer scientist.

Szabo underlined the need of a new way to formalize agreements and relationships and he proposed to enforce written agreements using programs running on a computer network [36]. In order to better explain the concept of smart contract, we can mention the parallelism between a smart contract and a vending machine made by Szabo. Between vending company and customer there is the following implicit agreement: once the payment is received, the desired item will be dropped. In a similar way, a smart contract executes its predefined clauses when the conditions are met.

In the Ethereum network, we refer to smart contract as a code script running on the EVM. It is made of a set of functions and a state, which is stored on the blockchain.

As Buterin himself said, smart contracts are treated as "first-class citizens". They are a type of Ethereum account, hence they have a balance and they can submit or receive a transaction.

Generally speaking, smart contracts serve as backend to handle the business logic of decentralized applications and Decentralized Autonomous Organization (DAO). DAOs are a perfect example to understand how powerful a set of smart contracts can be. Indeed, a DAO is a collectively owned organization without a centralized leadership. All the rules are defined by smart contracts. Crucial decisions such as budget allocation, investments strategy or proposal approval are taken by voting. As a result, since smart contracts are tamper-proof, it is impossible for any member to cheat and act on behalf of the entire governance. Complex organizations like DAOs are possible thanks to EVM Turing-completeness, which allows developers to write rules without any limitations. Moreover, Ethereum has several high level programming languages

3.2. SMART CONTRACTS

such as Solidity and Vyper that make writing smart contract easier.

The Ethereum community has grown prominently from its foundation and the interest and the support in writing smart contracts have grown as well. As a consequence, many Ethereum Request for Comments (ERC)s have been proposed to solve common problems and standardize smart contract developing. Among them, the most famous is the ERC-20. It is the technical standard for implementing a fungible token. It is built atop Ethereum network to provide a secondary feature such as representing a financial asset, voting rights or reputation points [37]. According to the technical documentation [38], an ERC-20 token is defined by six mandatory methods:

- `totalSupply`: returns the total number of token created
- `balanceOf`: returns the number of token in the wallet of specified address
- `transfer`: moves tokens from total supply to a user's balance
- `transferFrom`: allows token transfers between users
- `approve`: checks if it is possible to perform the transaction, i.e. that the `totalSupply` is not overcome
- `allowance`: the remaining number of tokens the spender can spend on behalf of the caller

Then there are three optional methods to increase usability:

- `name`: returns the token name
- `symbol`: returns the token symbol
- `decimals`: returns the token divisibility

As the last thing, there are two events that the smart contract must trigger:

- `Transfer`: a log of a successful call from `transfer` or `transferFrom`
- `Approval`: a log of a successful call from `approve` method

We will see later in chapter 5 how ERC-20 has been implemented in my work to realize a tokenomics, granting rewards to active users.

As the last step, it is useful to examine a smart contract life-cycle [39]:

- `creation`: it includes design, implementation, and validation. It is a bit different from traditional software development because a smart contract is not patchable or upgradable.

- compilation: a compiler turns the contract in bytecode to make it readable for EVM. It also produces the Application Binary Interface (ABI), a JSON file to describe the functions in the smart contract and the contract itself.
- deployment: contract account is created and stored on blockchain so it is accessible from any party.
- execution: the code will be automatically executed whenever a condition is met.
- completion: after the execution, the related transactions are stored in the blockchain and the state is updated accordingly.

3.3 CONSENSUS MECHANISM

As we have seen earlier, transactions can change the network state. A transaction request must be broadcast to the whole network and then, a node called validator will execute the transaction and transmit the updated network state to the other nodes. Since there is no central authority checking the state, there is not a single source of truth able to arbitrarily select the right block to attach to the blockchain. Consequently, there is a need for a consensus mechanism. It is a stack of protocols, algorithms and incentives to ensure a distributed set of nodes is able to meet a general agreement about the current network state. Decentralized p2p networks such as Ethereum can be severely affected by a Sybil attack where a single malicious entity tries to present multiple identities to control a fraction of the system [40]. As a consequence, a consensus protocol must implement a sybil-resistant component to protect the network. The two most common are Proof-of-Work (PoW) and Proof-of-Stake (PoS). The main difference is who are the users in charge of performing state transitions. In PoW, the set of users is made of the owners of computational power, also called *miners*. They have to solve a cryptographic puzzle to attach a new block to the blockchain and hence get a reward. As a result, the dominant strategy for a miner is to always direct all its computational power on only one block to beat the competition. Meanwhile, PoS empowers stakeholders. In order to join the protocol and become a validator, a node must stake a specific amount of money. For what concerns Ethereum, the stake is a deposit of at least 32 ETH. Initially, Ethereum relied on PoW but it recently switched to PoS after the Ethereum Mainnet joined the Beacon Chain [41]. The switch to PoS ensures

3.3. CONSENSUS MECHANISM

more security and more decentralization [42].

Currently, the Ethereum consensus mechanism is Gasper, a combination of two different protocols: Casper and GHOST [43].

First thing, Casper is a block finalizing mechanism based on round robin voting. [44]. It is based on PoS and it provides an incentive system to motivate validators to be honest and punish those which don't fulfill their tasks or act against the rules. A node gets a reward whenever it proposes a new block or it votes for a new one. At the same time, its deposit can be slashed if it breaks one of the following Casper rules:

- a validator cannot vote for multiple block proposals
- a validator cannot produce contradictory attestations

The incentive design is vital to ensure a properly working consensus mechanism because voting procedure in PoS is free. In comparison with PoW, it is not required computational power and energy consumption, so without severe punishment, the dominant strategy for a validator would be to vote for every block to be sure to guess the right one [45]. Casper rules counterbalance this strategy by allowing each validator to place only one vote. In this way, a vote is a sort of maximum odd bet and validators have every reason to play it right because attesting to an invalid or contradicting block leads to a penalty.

Before having a look at Casper voting procedure, it is worth describing how randomness is achieved because it is a key aspect to ensure fairness and security. Indeed, a fully predictable protocol provides more opportunities for attackers. When we talk about blockchains, randomness is not truly achievable because each node produces a different value, or better, nodes cannot agree on the same random value. Hence, it is not possible to reach a consensus. Instead, it is possible to achieve pseudo-randomness. In this regard, the chain stores a *bytes32* value called *RANDAO*. After every block, the last block proposer mixes its digital signature with the existing *RANDAO* value. In detail, Ethereum validators have a Boneh-Lynn-Shacham (BLS) signature to sign blocks and attestations. Since these signatures are generated to be uniformly distributed and secrets, it is not possible to predict in advance the next *RANDAO* value. As a result, the combination of *RANDAO* with the proposer's signature leads to incrementally gather randomness [46]. During each epoch, about 6.4 minutes, a new seed is computed by mixing the *RANDAO* value and the round number. Then, the list of validators is shuffled via a *swap-or-not* algorithm which uses the seed to ensure pseudo-randomness. After the shuffle, validators are divided into subsets

of at least 128 nodes, called *committees*. On top of that, it is selected the block proposer. In detail, also this selection relies on the seed value to ensure pseudo-randomness. On top of that, the probability is directly proportional to the node balance [47]. Committees are disjoint sets so a validator can be assigned only to one subset. Technically speaking, committee creation is not essential in the voting procedure but it is extremely functional because creating smaller p2p sub-nets avoids overwhelming the network or the nodes. After committees are created, each validator has to make its *attestation* where it shares its view on the blockchain. In particular, it claims that block A is the parent of the new block B , or using Ethereum's slang, A is linked to B .

In an ideal case, a blockchain is similar to a linked list of blocks where each parent block has only one child. Nevertheless, a *fork* can happen because of latency or a malicious attack. As a result, the blockchain can have sub-trees. In these cases, validators need a rule to recognize the canonical head of the chain and attest to it. In the Ethereum blockchain, the fork choice rule is provided by a greedy algorithm called Greediest Heaviest Observed SubTree (GHOST). According to GHOST, anytime there is a fork in the blockchain tree, a validator has to attest as the head of the chain the sub-tree with the heaviest weight. The *weight* w of a block B is the sum of the stake of the validators whose last attestation is to B or descendants of B . Since a validator is rewarded when its attestation is right, the best strategy could be to wait for others' attestations and vote after being sure to attest properly [46]. To be precise, the above strategy was actually the best one until the Altair update in October 2021 [48]. Altair changed the penalty and reward values to motivate timeliness over carefulness. In other words, now a validator is more likely to be fast than prudent. Consider the example from Fig. 3.1 and imagine every validator has a stake equal to one. We can see that three validators voted for the blue block so its weight is equal to three. As a result, a validator running GHOST has to attest that the blue block is the next head of the chain.

Attestations are then gathered together in a single message by a set of special nodes called *aggregators*. Each committee has a certain number of aggregators, which are defined by the seed value at the start of the epoch. The message will be later sent to the block proposer. At the end of the voting round, if a link has been voted by $\frac{2}{3}$ of the validators is said to be a *supermajority* link. As a result, block A goes from *justified* to *finalized* while B is *justified*. It is easy to see that each proposed block can be finalized in two rounds [49].

3.4. ETHEREUM NAME SERVICE

A *finalized* block cannot be changed, or reverted, without burning a significant amount of ETH. Indeed, in the case of a contradictory claim, from $\frac{1}{3}$ to $\frac{1}{2}$ of the validators would lose their deposit. On top of that, a special mechanism called *inactivity leak* is triggered if the chain does not finalize a block in a certain period. The idea is to gradually reduce the stakes of inactive or malicious validators until the active ones control again $\frac{2}{3}$ of stake and hence they are able to finalize a block [46].

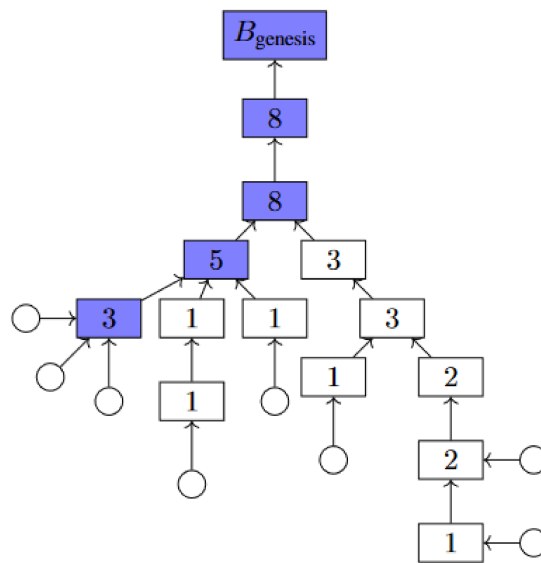


Figure 3.1: Example of GHOST fork choice rule [43]

Finally, an honest validator which runs GHOST to solve block conflict and acts according to Casper rules, will never see its deposit slashed [43].

3.4 ETHEREUM NAME SERVICE

Ethereum Name Service (ENS) is a distributed and extensible naming system based on Ethereum. Its task is to map a human-readable name to a machine-readable identifier such as an Ethereum address or a content hash [6]. With regard to our use case, we want to map the Swarm hash of our website to a human-readable URL.

ENS functionalities are provided by three smart contracts:

- Registry: it has to maintain a list of all domains and subdomains where, for each record, it stores the owner, the resolver and the time to live. Indeed, as

it happens in centralized web, a user owns the domain name for a limited amount of time, which depends on how much money the user spent.

- Resolver: it translates names into addresses. It is pointed at the register.
- Registrar: it is responsible for subdomain allocation. For example, top-level domains such .eth or .xyz are handled by a registrar.

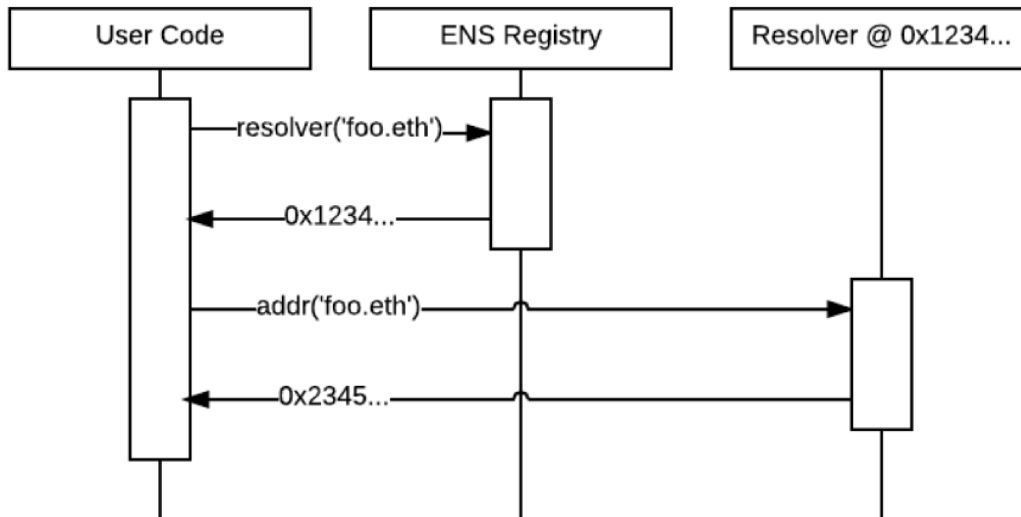


Figure 3.2: ENS resolving name mechanism [6]

As we can see from Fig. 3.2, the name resolution is straightforward: a user asks the registry a human-readable string, which is "foo.eth" in this example. The registry then returns the resolver in charge of it. As the last step, the user asks the resolver, the latter returns the address.

Finally, it is important to notice that ENS registry doesn't store the actual name, as "foo.eth" in the example, but it generates a unique hash value thanks to a recursive process, called Namehash. As a consequence, there is no need for string manipulation and hence it allows a easier and faster lookup [50]. Besides, this expedient allows to save gas. The EVM has 256-bits memory slots so any value stored in smaller data type requires the EVM to fill up the missing bits with zeros. Since this operation costs gas, saving a 256-bits value instead of a string is more efficient.



Ethereum Swarm

Swarm is a peer-to-peer network of nodes to provide a decentralized storage system and communication service. Its mission is creating a “distributed hard disk for Web3” [51].

It is organized in four different layers:

1. Underlay p2p network
2. Overlay network
3. High-level access via API
4. Application

Layer (1) is based on *lib p2p* protocol. It provides basic networking functionalities such as addressing, dialing, and listening. On top of that, it ensures security in the communication channel, protocol multiplexing, delivering, and serialization.

In this chapter, our focus will be on layer (2) and layer (3) because they are the core of Swarm.

4.1 OVERLAY NETWORK

The overlay network is built upon Kademlia. It is a peer-to-peer protocol exploiting the Distributed Hash Table (DHT) technology to enable communication between two nodes.

4.1. OVERLAY NETWORK

4.1.1 KADEMLIA CONNECTIVITY

Swarm is a Kademlia topology: a network where each node is connected to the closest nodes, its neighbors, and also it is connected to at least one peer from different clusters [51]. To better understand this definition we have to comprehend how addressing works in a Swarm network.

In the overlay network, each node has a unique 256-bit address. It is computed as the Keccak hash of an Ethereum address and the ID of Swarm Network, also called BZZ NetId. This pair of identifiers ensures the address uniqueness over all Swarm networks. On top of that, the overlay address is crucial for computing the Proximity Order (PO). This value quantifies how much two nodes are related. Taken two addresses, PO is computed by counting the number of matching bits in their related binary representations from the most significant one up to the first one that differs. As an example we can take two addresses x and y . Since each matching bit counts as one, if we have $PO(x, y) = 256$, then there is the max proximity order degree possible between two nodes. As a result, we have $x = y$.

Now we can create a Kademlia table for each node. It is the indexing of all peers based on proximity order. By definition, a Kademlia table is saturated if all nodes as near as an arbitrary proximity degree d_x are peers of x and at least one peer for each PO equivalent class is a peer of x .

Finally, since each node in the overlay network has a saturated Kademlia table, Swarm is a Kademlia topology.

The latter claim is fundamental to understanding how routing works in Swarm. In opposition to other peer-to-peer implementations, the Swarm team decided to rely on forwarding as the routing flavor [51]. It works as follows: when a node has to relay a message, it has to send it at least one step closer to the destination. In a Kademlia topology like Swarm, a message is always routable. It is easy to understand why: since a node is connected to at least one peer from each equivalence class, it can always forward the message one step closer to the destination. Obviously, it is crucial to maintain a Kademlia topology to ensure the routability of all messages. As a consequence, a node must keep in several connected peers for each PO class to be sure that node dropout doesn't distress the saturation of its Kademlia table.

Forwarding routing deletes unpredictability in message delivery because nodes relay messages only to nodes that are surely online and connected.

4.1.2 STORAGE MODEL

The second Swarm cornerstone is Distributed Immutable Store of Chunks (DISC). It is a storage model: it provides a set of rules to make nodes collaborating in storing and serving data.

A DHT is a key-value mapping distributed over a network. Generally speaking, DHT associates an address, the key, to a set of nodes, the value. For example, in IPFS the content retrieval works as follows: a Downloader D asks for a key, which is the content identifier, and then it gets as value a list of nodes that can serve the content. As last step, D requests the content to one node in the list.

With DISC, Swarm introduces a new interpretation of DHT for storage over Kademlia routing. The main difference is what value Swarm associates to each key. In opposition to IPFS, values are not content locations but they are chunks of the content itself. Swarm approach is more straightforward because DISC deletes one step from retrieval procedure, as we can see clearly see from Figure 4.1.

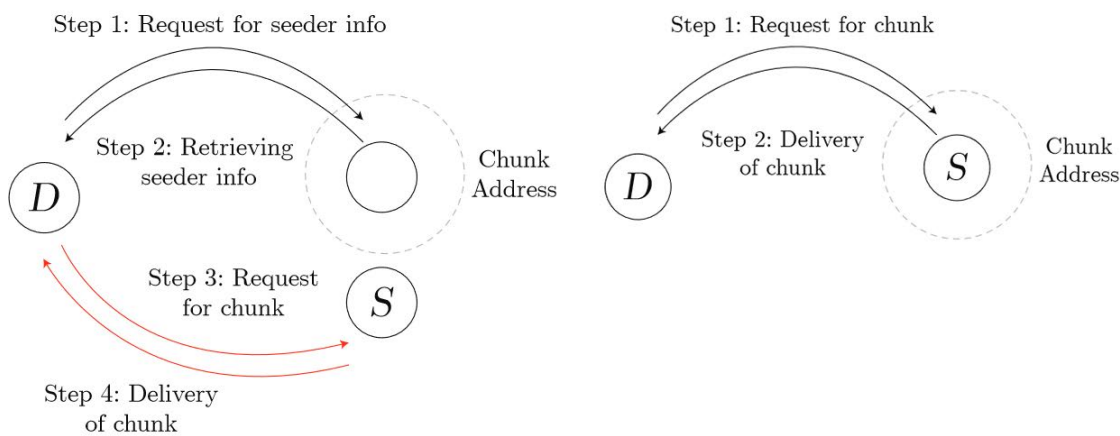


Figure 4.1: Comparison between IPFS and Swarm retrieval system [51]

TYPE OF CHUNKS

A *chunk* is the basic storage unit in the DISC model. We can distinguish two types: Content Addressed Chunk (CAC) and Single Owner Chunk (SOC).

A CAC is a Binary Merkle Tree (BMT) chunk [51]. It is a data structure composed of leafs, 32-byte data segments, and branches. Each leaf is labeled using the Keccak hash of the data segment while a branch label is computed as the Keccak

4.1. OVERLAY NETWORK

hash of its children labels. In figure 4.2 we can see the whole structure of a binary tree chunk: D_i^j are the leafs, H_i^j the branches.

Each CAC is composed of an 8-byte span plus a payload of a max of 4096 bytes. Its address is the Keccak hash of the BMT root plus the span. As a consequence, Swarm addressing is collision-free because each chunk has a unique address and it is also deterministic because it allows the downloader to check the correctness of the content. In other words, it enables local validation and integrity guarantee. While in CAC the address ensures integrity, in SOC a user can assign arbitrary data to an address. As a consequence, it is the user himself who has to attest to integrity via his digital signature. A SOC is made of

- identifier: 32 bytes
- signature: 65 bytes
- span: 8 bytes
- payload: 4096 bytes

The address is then computed as the Keccak hash of the identifier plus the owner Ethereum address.

SWARM FEEDS

Feeds are the primary use case of SOC. A feed is a sequence of SOC with a predictable address, also called *feed chunks*. It is the perfect solution to use a static address for mutable content [52].

Like every chunk, a feed one is made of a 8-byte span and the payload of max 4096 bytes. On top of that, each feed chunk has a unique identifier that is composed of *Topic* and *Index*. The first one is the hash of an arbitrary string, usually human-readable, that is attached to specify the topic. The second one depends on the type of feed we are considering. In the simplest case, it is an integer value.

The whole structure is well represented by figure 4.3.

The owner of a feed can post an update following this procedure:

1. constructing the feed identifier using the topic and the correct index
2. combining identifier and owner Ethereum address to sign the chunk

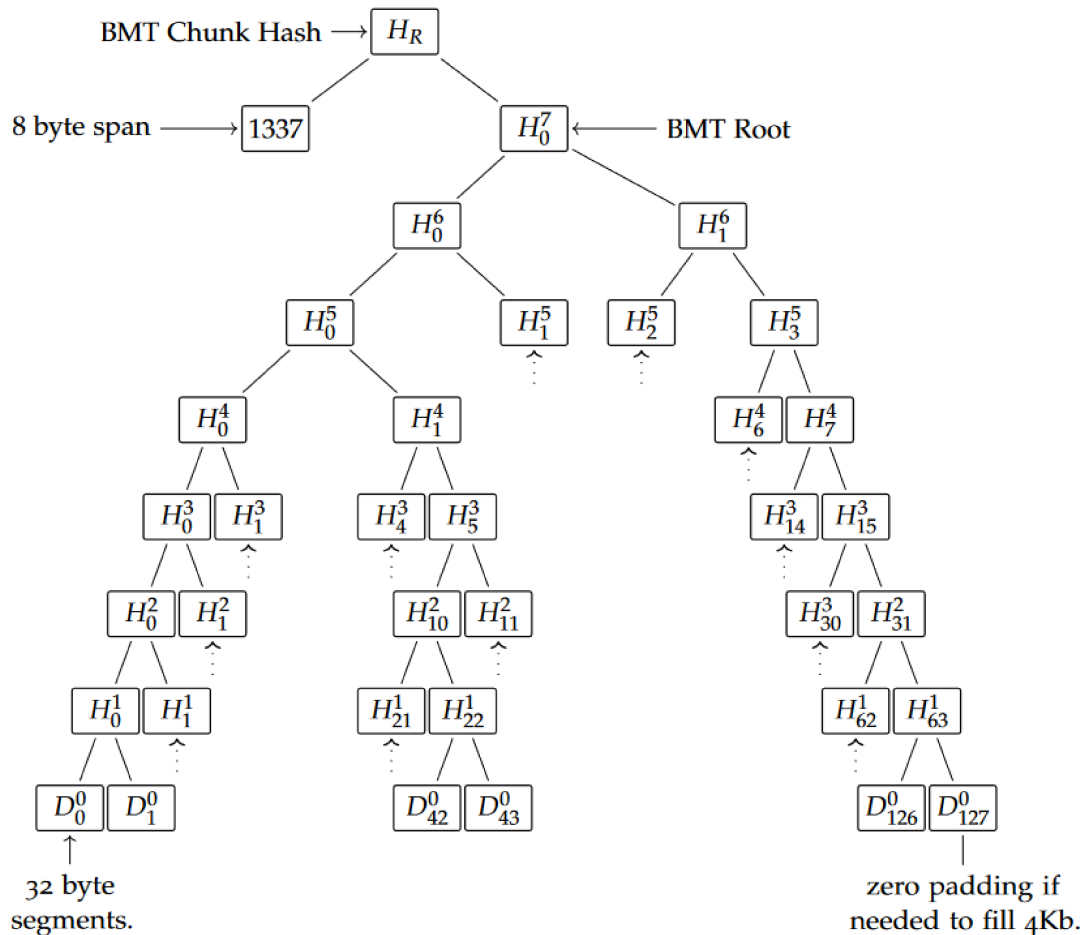


Figure 4.2: A Binary Tree Chunk [51]

On the other hand, a user can get the feed by retrieving the chunk by its address. The user has to compute the address using the owner's public key and feed identifier.

As the last thing, it is important to distinguish the three distinct update semantics and the related feed sub-types:

- **substitutive:** feed content is replaced with the newer version because users are interested only in the last update. This sub-type is useful for domain resolution in combination with ENS where the content reference is linked in a feed and then the feed is solved by reference using ENS.
- **alternative:** updates are independent of each other and each one must be maintained. It is useful for representing a series of content aggregated by a common topic or theme such as social media or blog posts.
- **accumulative:** updates are meant to be added to previous ones, following the temporal order. For example, it is useful to represent a video stream.

4.2. INCENTIVE SYSTEM

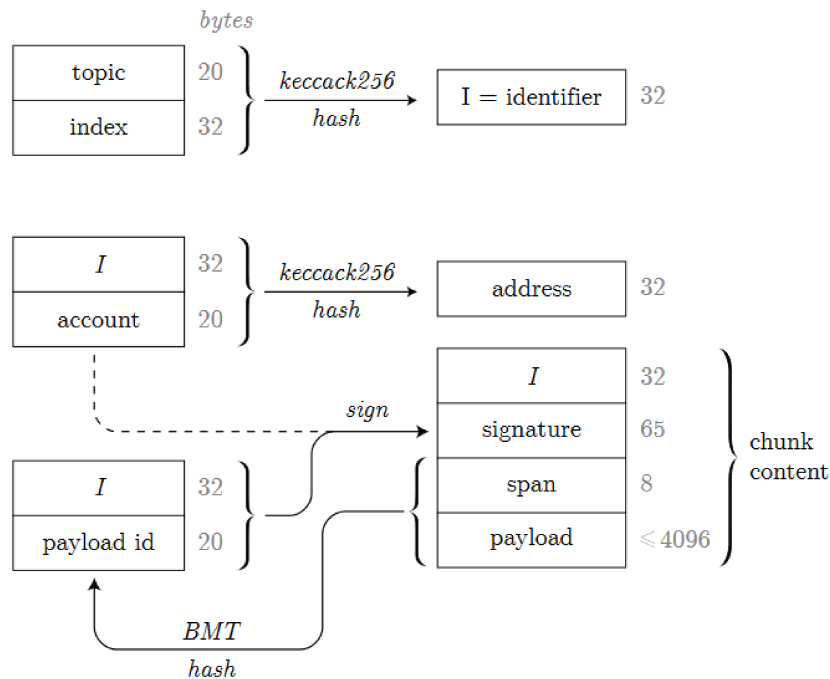


Figure 4.3: Feed structure [51]

4.2 INCENTIVE SYSTEM

Swarm network has a built-in incentive system that aims to create a self-sustainable ecosystem. It is unquestionably a killer feature because no other distributed storage system is implementing a similar thing natively. From a user point of view, it is catching because running a node makes upload and download free if it provides service to the network [53]. It is based on BZZ, an ERC-20 token that enables net users to pay net providers.

4.2.1 BANDWIDTH INCENTIVES

As we have seen in the previous section, the routing is based on Kademlia forwarding. Maintaining a working Swarm network requires that each peer is cooperative and it shares its bandwidth to help another node download or upload a file. Nevertheless, a node is a selfish agent and its actions are aimed at maximizing profit and minimizing resource usage. This is the reason why Swarm implements rewards on chunk retrieval.

In very few words, a requester D pays a certain amount of BZZ token to down-

load a chunk. The accounting event is triggered only when the chunk is delivered to D and if the delivery happens within a certain time.

According to forwarding routing, the chunk request gets bounced from one node to another until it gets to the storer node S . Except for the requester D , each node F_i has an income from F_{i-1} and an outflow to pay node F_{i+1} . The difference between income and outflow is what each node can earn from forwarding. Since retrieval price is a decreasing function of proximity, this difference is always positive and hence it is a profit. On top of that, prices in equivalent proximity classes are uniform due to price auto-balancing. To be clear, a price can vary because of bandwidth cost and bandwidth demand. If a node F_i has a lower price in comparison with its neighbors, the demand will increase because nodes F_{i-1} will have more economic convenience to pass the request through it. As a result, increasing demand will lead to increasing prices until the price will not be anymore the lowest in the neighborhood. Since local delivery is not rewarding and non-local delivery is a monotonically increasing function, a node is motivated to get the chunk as forward as possible to maximize its earnings. From a user point of view, it means saving hops and hence reducing latency.

On the other hand, Swarm has to ensure that each chunk is properly uploaded

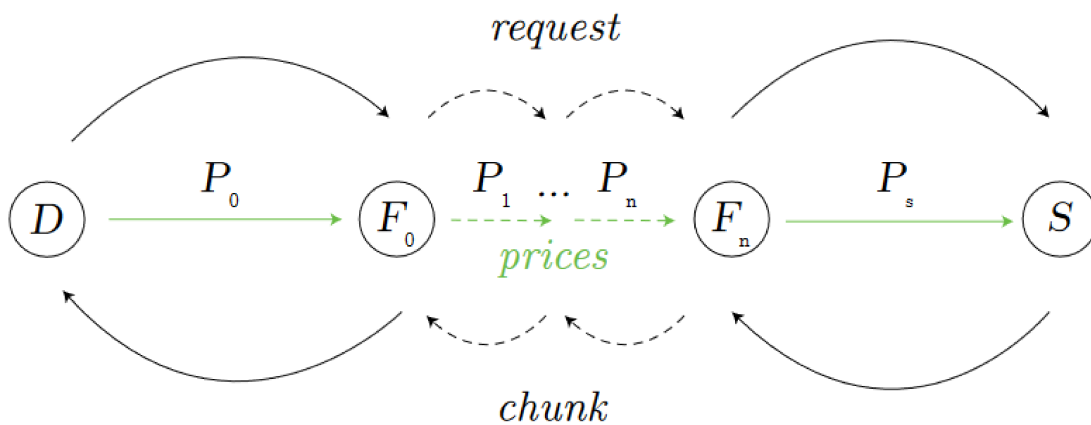


Figure 4.4: Downloading of a chunk

to the network. The uploading procedure is handled by a push&sync protocol. Nodes must forward the chunk from the uploader U to a node storer S , which is the closest one to the chunk address. As we will see in a further section, storing a chunk has an economic value. Consequently, a node F_i could be tempted to act selfishly and hold a chunk before it gets to its destination. Swarm solves

4.2. INCENTIVE SYSTEM

this issue by introducing a statement of custody receipt. It is a paid message [51] sent by storer S once the latter gets the chunk. The statement receipt is sent back to the uploader U following the exact route of the chunk. Each forwarding node F_i gets a certain value when it passes the statement. If uploader U doesn't receive the receipt within a certain time, the upload attempt will be marked as failed and uploader U will try an alternative route. As a consequence, if a node holds the chunk, then several other nodes will be willing to forward the chunk to be rewarded. As a result, holding a chunk is not profitable. This threat is enough to ensure that each node will behave properly.

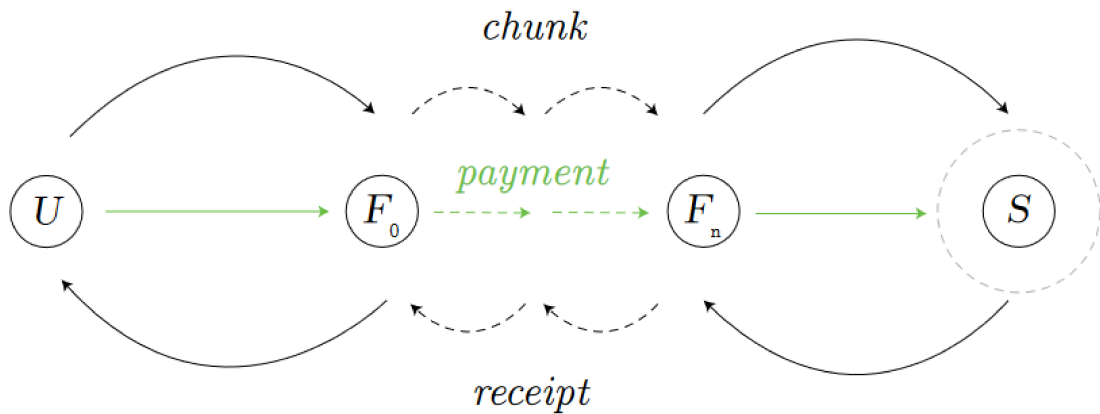


Figure 4.5: Uploading of a chunk [51]

4.2.2 SWAP

Settle With Automated Payments (SWAP) is a protocol for peer-to-peer accounting to implement bandwidth incentives. It is based on a tit-for-tat scheme: a popular game theory strategy where two agents, in our case two nodes, can choose to cooperate or defect. If they both decide to cooperate, both do well. Otherwise, if one defects then the opponent will be motivated to defect too [54]. The idea behind SWAP is to transpose this concept in the context of the Swarm network. First thing, the SWAP protocol is in charge of keeping track of data traffic between two peers. Until there is an equal balance between consumption and provision, no payment is required. In the opposite case, if a node generated an imbalance, an automatic payment is issued to restore the balance. To be precise, the *payment threshold* and the *disconnect threshold* are the two values

capable to trigger an automated event. The first one is the limit on self-balance, when it is reached it triggers a transfer of found via a cheque to the remote peer's address. While the second one is the limit that triggers the disconnection from the remote peer once reached [55]. As we can see from figure 4.6

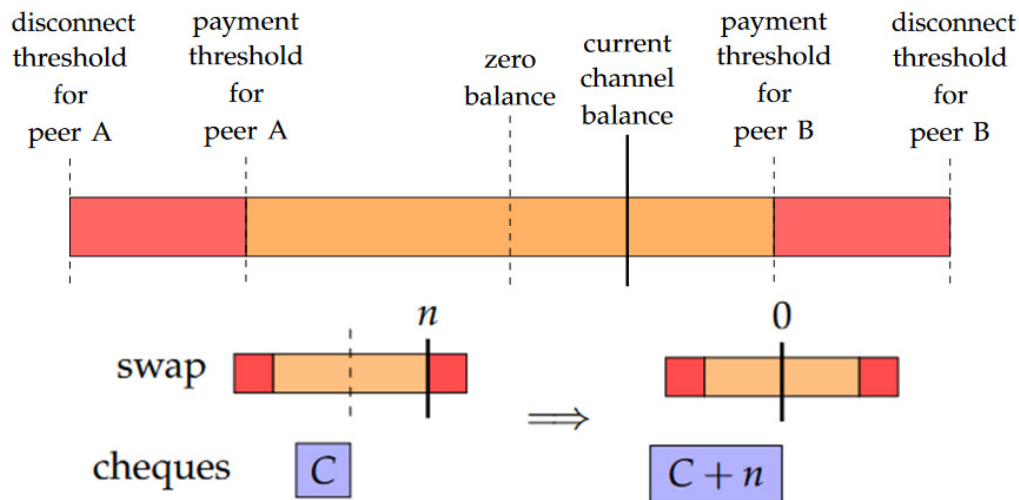


Figure 4.6: Basic functioning of SWAP protocol [51]

Since peers engage each other often, it is reasonable to defer payments and process them in bulk. This strategy reduces transaction costs but at the same time, it increases the risk of settlement failure. The strategy implementation is performed thanks to a smart contract called *chequebook*. It acts like a bank: once the payment threshold is reached the issuer gives the recipient a signed cheque with the beneficiary, amount, and date; then the recipient can cash that cheque by submitting it to the smart contract.

On top of that, SWAP allows peers to exchange cheques to save again on transaction costs. Imagine the scenario in which node A issued a cheque to node B; then B got into debt with A. Instead of issuing another cheque to A, B can simply waive the first cheque signed by A. Chequebook will record the payout.

As the last thing, it is interesting to notice how a newcomer can interact with the network even if it has a null balance. The newcomer can stack cheques by providing services to peers with funds, called the insider, but since cashing cheques has some transaction costs, it won't be able to monetize them. To delete the entry barrier, SWAP has a feature that allows an insider to cash newcomer's cheques and then trigger a payment to the newcomer's address. In this way, it

4.2. INCENTIVE SYSTEM

is possible to get started in the Swarm network without any initial investment.

4.2.3 STORAGE INCENTIVES

The bandwidth incentive system has an important side effect. Since a node gets rewarded when it retrieves a chunk, the more often a chunk is requested, the more a node earns. In this way, profit is directly proportional to content popularity. The ratio between storage cost and retrieval profit is crucial when a node is running out of storage. Indeed the chunks with the lowest profit per storage ratio will be the first to be purged because a node priority is maximizing profit. As a consequence, popular content will be spread across the network while unpopular ones will become rarer and, in the worst scenario, they are likely to be lost. Since data persistence must be ensured, Swarm implements a storage incentive system based on postage stamps and postage lottery.

POSTAGE STAMPS

As the first point, a postage stamp serves as proof of payment. By purchasing a stamp, a user earns his "right to write" on the network. It is a sort of uploading tax.

Stamps are sold in batches. Each batch is created by a smart contract whenever a user sends a proper request to the dedicated endpoint. As we can see from code 4.1, each request has to specify an *amount* of BZZ assigned to the batch, and *batch depth* which is the maximum number of chunks allowed to be in the same bucket [52]. Just as a reminder, Swarm address space is divided into 65536 buckets; during the uploading phase the file is split into 4 kb chunks which are assigned to different buckets according to their address.

```
1  #Requesting a batch
2  $ curl -s -XPOST "http://localhost:1635/stamps/amount/batch_depth
   "
3  #Check on node stamps
4  $ curl "http://localhost:1635/stamps"
5  #Top up a stamp
6  $ curl -X PATCH "http://localhost:1635/stamps/topup/batchID/
   new_amount"
```

Code 4.1: "Console interaction with stamps endpoint"

Going back to request parameters, the *amount* determines the Time To Live (TTL) of a file. Indeed each batch has a *batchTTL*, which is computed according to the storage price during upload under the assumption that the price will be static in a near future [52].

BatchTTL is equal to the number of seconds before the related chunk is considered for garbage collection. As a consequence, once *batchTTL* is expired, the related chunk is moved to the cache. If the node is in capacity shortage, then the cache is pruned [53]. In other words, recency is taken as a popularity predictor. Under these considerations, it is important to notice that postage stamps act also as spam protectors. Imagine a storage system based solely on recency: it would be easy to perform a spam attack to replace stored content with useless chunks. By introducing an uploading cost via postage stamps, Swarm is able to ensure certain data persistence and at the same time an effective defense from spamming.

From a technical point of view, we could describe a postage stamp as a data structure made of chunk *address*, *batchID*, and *witness*. *BatchID* is needed for enabling validity checks via smart contract while the witness is the owner's digital signature. The latter is for associating the *batchID* and the owner address.

The network considers a stamp valid only if it is

- authentic: the *batchID* is valid
- authorized: The witness is signed by the address specified as the owner of the batch.
- founded: the batch has a non-zero TTL

As the last thing, it is important to notice that each user can extend a batch duration by topping up the desired batch using the stamps endpoint, as we can see from code 4.1.

POSTAGE LOTTERY

We have seen that postage stamps are crucial for spam protection and for handling garbage collection but it is not enough to ensure data persistence. Indeed once the stamp is not founded, the related chunk is very likely to be pruned at the first capacity shortage. For this reason, Swarm has another storage incentive: the postage lottery. Since an unpopular chunk doesn't provide a retrieval reward, the lottery goal is to compensate for the missing earnings by

4.2. INCENTIVE SYSTEM

redistributing the postage stamps revenue.

To enter the lottery, each node has to submit a transaction with a commitment

$$C_i = H(s_i|a_i|d|R)$$

where

- H is a hash function
- s_i is a random salt value
- a_i is the overlay address of the applicant node
- d is the storage depth
- R is the Swarm hash of the concatenation of the first k chunk address in applicant node storage

Then the winning R and d are drawn from all the commitments. Among the nodes with $R_i = R$, it is randomly picked the winning node w . As the last step, the smart contract will check that the storage usage of w is above a certain threshold.

As a result, nodes are motivated to store chunks to be eligible for the lottery. On top of that, they have also incentives to sync storage with their neighbors because if they commit several different R_i , the risk of losing increases.

5

CMS Design and Implementation

I have decided to call the application *SwarmAd*. It recalls on purpose *EtherAd*, i.e. the application developed by Martini [3], because it has been the starting point for my thesis.

In this chapter, we are going to see the actual implementation of SwarmAd. First, we are going to point out all the core functionalities that the CMS must provide. Then, I will explain how all the components described in the previous chapters will be assembled to satisfy those functionalities and provide the desired services. Finally, we will have a brief look at the local environment I have set up for developing and testing purposes.

5.1 REQUIREMENTS

Firstly, it is important to remind that the target is to build a platform where SMEs with similar aims can aggregate and cooperate to maintain a common showcase without technical expertise and relevant maintenance expenses. As a consequence, a crucial point is to provide an intuitive application.

In brief, the dApp must:

- have a web User Interface (UI) easy to access and use
- provide a showcase where enterprises can show their product to potential customers
- avoid any overhead, like browser extensions or third-party applications, that are not essential to the functioning of the platform

5.1. REQUIREMENTS

The latter point is important because any redundant entry barrier could undermine the convenience of using the platform since the target customer is not required to have any expertise. On the other hand, we will see how some overheads will be essential to make the application working and to ease some procedures. For example, since it is a dApp, every registered user needs a valid Ethereum account to exploit the app functionalities like uploading a product on the showcase.

As a second point, we have to remember that the membership is reserved for users who are related to the community scope. In this sense, it is very different from the examples like Helios or HiDe that we have seen in Chapter 2, or from the traditional social apps. For example, if the platform is reserved for pottery craftsmen located in a specific Italian village, a user who doesn't meet these requirements has to be rejected by the community.

Being a registered user enables a specific set of services but it also includes some responsibilities.

A registered user must be able to:

- create, read, update, and delete his enterprise profile
- create, read, update, and delete an item
- vote to accept a newcomer
- report any misbehavior from other members
- vote to ban a member who is acting against the rules
- vote to delete an inappropriate item posted by another member
- propose changes in administration settings
- vote about proposals

Besides, we can add the following optional functionalities to improve the usability:

- a user can create his profile page and organize his content according to his liking
- users can discuss relevant decisions in a specific space, a sort of community forum

Meanwhile, a non-registered user can:

- navigate the homepage showcase

- access each enterprise profile page
- report any inappropriate item or enterprise

A report from a non-registered user has to be examined by members of the community before a voting procedure is created.

The administration is fully automated by a set of smart contracts. Indeed, there is no central authority but there is a consensus process where each registered user can take part and express his will. In other words, all the decisions are to be taken from the registered users fairly and democratically. Considering our use case, decisions will be about:

- accepting a newcomer
- banning a user or item
- changing settings in community administration

For example, users could decide to increase the voting procedure duration or decrease the quorum to establish the winning proposal.

To sum up, the automated administration has to:

- keep track of all the registered enterprises
- keep track of all the items posted by registered enterprises
- keep a meaningful mapping between an image and its Swarm hash
- define threshold, quorum, and deadline for user voting
- open a voting procedure when a registered user flags a misbehavior
- keep track of reports from unregistered user
- broadcast a report to registered users
- keep track of active voting procedures
- close a voting procedure when the quorum or deadline is met and broadcast the result
- perform the required action when the voting procedure is closed, like ban a user or remove an item
- broadcast to registered users any relevant event, such as a new poll or a new proposal
- define a reward system to motivate users in community moderation
- define a penalty scheme to punish users who are acting against the rules

5.2. STACK DESIGN

- create a waiting list to handle newcomers' requests
- keep a blacklist of banned users

In particular, rewards and penalties are crucial for the functioning of the application because they motivate users to be active in community moderation and be respectful of the rules of conduct. Without a properly working incentive scheme, the platform would not be able to reach its goals. Here is the case of digress. SwarmAd is an advertising platform, so it is a sort of "business card" for enterprises that participate: if the reputation of the website is good, then the enterprises benefit from it. Otherwise, they can be damaged. As a result, there is an implicit incentive to keep a healthy community. On the other hand, moderating is a time-demanding activity and in the long run, users could be bothered to invest time in an "abstract" reward. Things change if rewards are tangible and pleasing. For example, we could provide a highlight spot on the showcase to the most active users. Since it is an advertising platform, it would be a significant edge over the other members, who de facto are competitors. In a later section, we will delve into the tokenomics design to better understand the whole incentive system.

5.2 STACK DESIGN

A Web3 application is way different in comparison with the traditional web apps. First, the back-end code is running on a decentralized peer-to-peer network instead of a centralized server. As a result, the application state has to be saved on a blockchain. In this regard, among the different options, I have chosen Ethereum. Indeed, it supports smart contracts and it has extensive documentation and a very active community of developers working on it. From a developer's point of view, it is the best choice to enter the Web3 ecosystem. On the other hand, the main drawback of Ethereum is scalability. Nevertheless, if we consider the application use case, the number of registered users should be small hence it wouldn't be a relevant issue. On top of that, I expect a very higher request in reading the blockchain instead of writing on it. In other words, I expect more users who passively navigate the content in comparison with users who are registered and upload content. Since querying the blockchain is a gas-free operation, the transaction costs should not be an issue. For example, if we were planning to build a marketplace, opting for a layer-2 solution such as

Polygon would be taken into consideration. Indeed, customers would perform a transaction to buy a product so the ratio between reading and writing operations would be different.

Second, I have chosen Swarm as distributed storage. This choice is counter-trend compared with the many examples I have illustrated in Chapter 2. As I have underlined in Chapter 4, Swarm is an interesting solution to data persistence because of its built-in storage incentives. It allows a fully decentralized architecture: the network itself is in charge of maintaining the uploaded data. Compared with IPFS, there is no need for an always-online node to be sure that files are accessible.

In Figure 5.1 we can see a complete schema of the technological stack I have chosen to realize my dApp. Further in this section, we are going to analyze each component and understand how they interact with each other.

5.2.1 BACK-END DEVELOPMENT

In a dApp, a blockchain is designed to act as a back-end server by running a set of smart contracts where the business logic is defined.

Before going any deeper, it is important to have a look at the main entities involved in the application workflow. From the requirement analysis, we can extract the following:

- enterprise: a registered user; it has its profile and it is able to create, read, update, and delete an item
- item: it is a product posted by an enterprise to be shown
- poll: a voting procedure
- token: the base unit of the reward system

At first glance, it could be possible to create a single monolithic smart contract to handle all fours. Nevertheless, I think that a modular approach is a better solution because it can enable contract upgradability and it also allows to reuse secure and tested contract standards. The latter procedure is strongly encouraged by the Ethereum Foundation in the best practices for smart contract development [56].

In regards of upgradability, it is also important to remember that smart contracts are immutable by design. As a consequence, if we have to fix a bug or make

5.2. STACK DESIGN

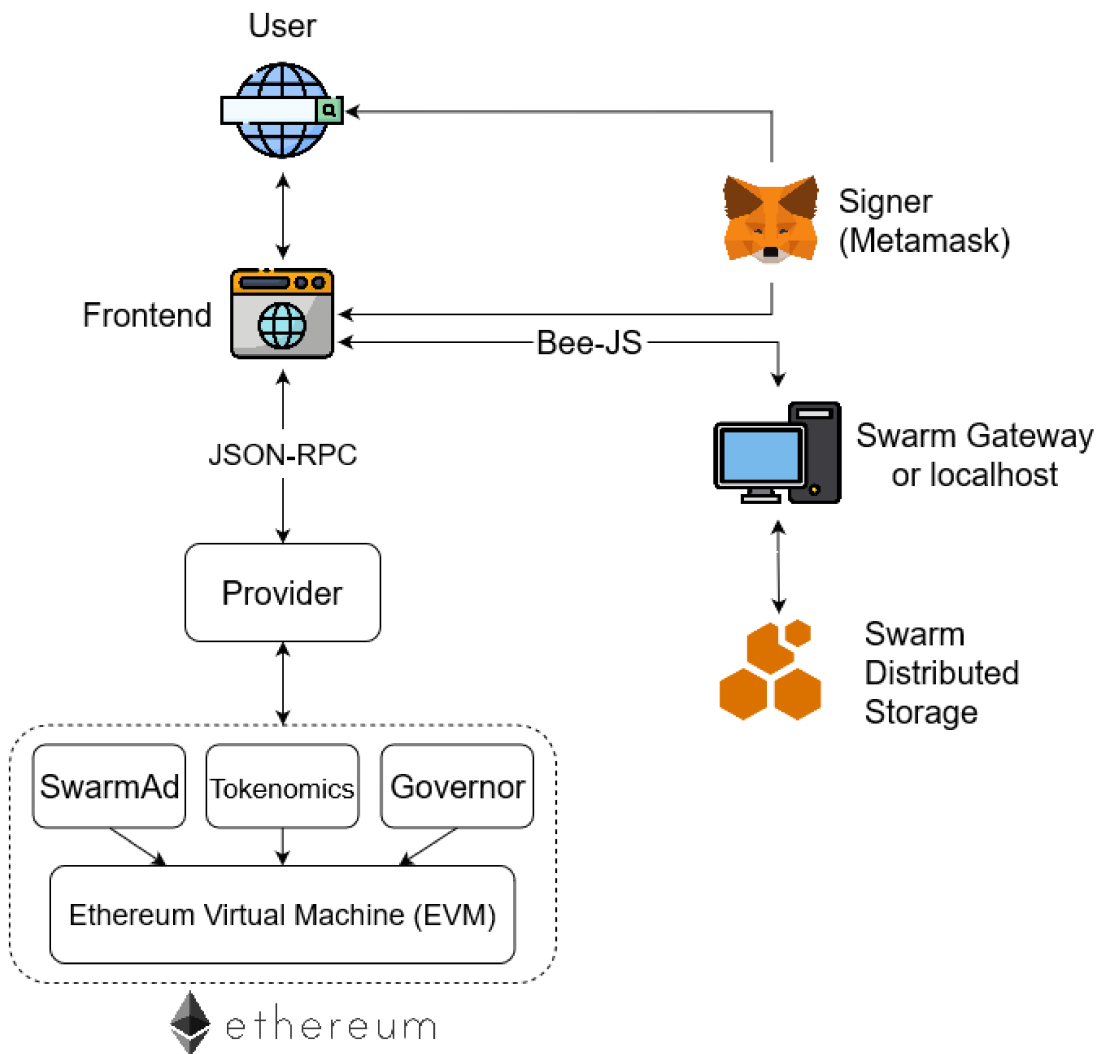


Figure 5.1: dApp architecture

some changes, the only way to update a smart contract is to write a new version and deploy it as a new contract. In this sense, having a modular structure allows a certain degree of upgradability because a simple contract substitution can upgrade or fix a set of functionalities.

On the other hand, when we talk about modularity we have to remind that contract deployment is not free and it could be counter-productive to opt for a too smaller granularity. As an example, creating a different contract for each enterprise is not convenient because it would require a relevant expense without the guarantee of being accepted. From the user's point of view, it is better to pay a transaction for requesting the membership because it is way less expensive than deploying a contract. In other words, excessive modularity adds computational

overhead and increases costs.

After these considerations, I have opted for a data separation pattern where a contract is for storing and managing the data, i.e. enterprises and items, while the other functionalities like tokenomics and community governance are handled by two separate contracts.

As we can see from Figure 5.1, we have three entities:

- SwarmAd: to store data and implement the Create Read Update Delete (CRUD) operations
- SwarmAd Community Token (SWCT): a token based on ERC-20 standard
- SwarmAdGovernor: to handle the voting procedure

Additionally, I could have broken the structure even more by separating data and the CRUD functions. Nevertheless, I think that the three-contract solution is still the best trade-off between modularity and expenses. Indeed, *Item* and *Enterprise* are easy to define in an inclusive and precise manner. It is difficult to imagine a drastic change in the data model because it would implicitly include a drastic change in the application use case. Also, CRUD is a set of basic operations. It is unlikely for them to be changed because they are strongly related to the data model. Finally, gathering together *Enterprise*, *Item* and the related functions seems the most reasonable option.

5.2.2 ENTER THE BLOCKCHAIN

Once we have defined the business logic architecture, it is important to understand how the application will interact with it. As we have seen in Chapter 3, a contract must be deployed to the Ethereum blockchain to be accessible. Once it is deployed, a user must connect to the blockchain in order to interact with it. At this point, we have two different solutions:

- setting up a personal Ethereum node
- relying on a provider

A provider is a third-party service that makes its Ethereum client available to external users. In this way, a user can connect to the blockchain without having a personal node. This option is very convenient because it frees users from having any expertise or spending time on tasks such as setting up a private Ethereum account.

5.2. STACK DESIGN

Nevertheless, a provider is not enough to have full interaction with the Ethereum ecosystem. Indeed, as we have seen in Chapter 3, we can distinguish two different blockchain operations. The first one is reading. It is a gas-free operation because it doesn't change the network state. Consequently, a user needs only a connection to a public Ethereum node to query the blockchain. On the other hand, if a user wants to perform any kind of operation different from reading, he has to submit a transaction. As we have seen in chapter 3, each transaction must be signed with a private key to validate the identity of the origin of the transaction. As a result, a user needs a valid Ethereum account and a wallet to sign and send transactions. Generally speaking, most wallet applications create an Ethereum account for newcomers in order to ease the whole procedure. Since a wallet is fundamental in the Ethereum ecosystem, a user has several options to choose from. For example, a wallet can be:

- physical
- a mobile app
- directly integrated into a browser
- a browser extension
- a desktop application

In this regard, one of the best solutions is Metamask. It provides a wallet via a browser extension or a mobile app. The interesting thing is that it can serve both as a signer and as a provider because it has a connection with a public Ethereum node hosted by Infura. As a result, installing Metamask solves the two problems underlined before and hence it allows full interaction with the application [57]. Obviously, installing a third-party service adds an overhead but in this case, it is inevitable and it is way easier than running a personal Ethereum node or setup other third-party solutions.

5.2.3 ACCESSING THE SWARM

As we have seen in Chapter 2, it is a common strategy to rely on distributed storage to store large files. In this way, it is possible to reduce transaction costs and writing time at the same moment. To be precise, I have decided to adopt the *hash on-chain* pattern [58]: the files will be stored on a distributed storage and the references will be saved on the Ethereum blockchain. I have chosen Swarm as distributed storage because of the built-in incentives and the guarantee of data

persistence, as I have delved into Chapter 4.

It is important to notice that everything that is saved or stored on Ethereum or Swarm is public. By default, no access control restrains the visibility to a specific set of users. When a node has access to a p2p network such as Ethereum and Swarm, it is able to retrieve basically any content. As a result, certain applications must focus carefully on ensuring users' privacy or copyright. Nevertheless, this is not an issue in my use case. Indeed, since SwarmAd is an advertising platform, it is implicit that every content posted on the platform is public. Even better, it is meant to be shown.

Similarly to Ethereum, accessing the Swarm network requires establishing a remote connection to a Swarm node, also called a Bee client. Again, it is possible to choose between running a private node or accessing through a public gateway. Before going into the details, it is important to distinguish three different types of Bee clients:

- full node: the standard one. It participates in retrieval, forwarding, and storing. It is rewarded according to its contributions.
- light node: it doesn't participate in forwarding and storing but it is able to upload content on the network once a valid wallet is associated and funded.
- ultra-light node: it is a light node without a wallet. As a result, it can download files but it cannot upload them since it has no economical resources to buy the postage stamps.

Among the different options, running a full node is without a doubt the best option because it has no limitations and it can be a profitable activity since the user would participate in the Swarm incentive system. On the other hand, it is for sure a complex operation because it requires certain expertise and familiarity with computer science. A light and ultra-light node are easier to set up because it is all handled by a desktop application, called Swarm Desktop. The user has only to download and install the application. Meanwhile, navigating via Swarm Desktop doesn't provide any reward. As the last option, we can use a gateway. For example, Swarm Foundation is developing and maintaining a public gateway. The main drawback is that it has a limitation on bandwidth usage: it is not possible to download or upload a file larger than 10 MB. Table 5.1 summarizes the different options we have to access the Swarm network.

In an ideal case, the best solution is the one where every user runs his own node. It is a win-win situation because users can earn rewards and the platform

5.2. STACK DESIGN

	Gateway	Ultra-light	Light	Full
decentralized	no	yes	yes	yes
download	yes but limited	yes	yes	yes
upload	yes but limited	no	yes	yes
postage stamps	no	no	yes	yes
rewards	no	no	no	yes

Table 5.1: Different ways to access the Swarm network

fault-tolerance is very high. However, this is not a feasible solution. As we have underlined in the requirements, we want an easy and intuitive user experience. Running a node could be an entry barrier very difficult to overcome. On the other hand, relying only on a public gateway is neither the best option because it is limiting the user experience and it adds a single point of failure. Hence, it is in the opposite direction of the design philosophy. I have decided to implement a hybrid solution: using a private gateway. Even if we want to avoid any external interference in the community, there is the need for a public or private organization that is in charge of deploying the whole SwarmAd platform. In my design, one of its responsibilities is to set up a node, or a set of nodes, to provide a trusted entry point to registered users. This gateway I have implemented is the one provided by the Swarm Foundation [59]. It is identical to the public one but it has no limitations on bandwidth usage. On top of that, the organization is rewarded in BZZ tokens according to the node contribution. As the last thing, the gateway is also able to pin content, as any Swarm full node. Pinning the front-end asset folder could provide an extra guarantee of data persistence. Regarding the latter, Swarm supports the "upload and forget" principle: once the content is uploaded, it is maintained by the network itself. As a result, if the organization node went down, the data would still be available. This is a key difference in comparison with IPFS where a node has to be always online.

It is also important to notice that the private gateway is not a single point of failure. Indeed, in case of denial of service, users could access the network by another gateway, such as the public one, or another node, for example, Swarm Desktop. In case of failure of the private gateway, the application has a fallback function that switches the Bee client to the public gateway in order to maintain active the core functionalities. We will see this solution in detail in the next section. In these regards, Swarm Foundation is actively developing a browser extension to ease the switching between a list of Swarm clients. In

this way, a user is able to select his preferred node as the trusted endpoint to channel all the requests [60]. At the moment, it is still in Proof-of-Concept so I have decided to not rely on it but it could be an interesting addition in the future.

5.2.4 FRONT-END INTERFACE

The front-end interface is the mean through which users will be able to interact with Ethereum and Swarm to access all the app functionalities. Since the website hosting is provided by Swarm, we cannot rely on the traditional web app design where the application logic is performed on a server. On top of that, all the data are stored on a blockchain. Consequently, it just needs a UI to query the blockchain and represent the fetched data in a meaningful way. As a result, I have opted for creating a Single Page Application (SPA). A SPA is a single HTML page where are stored all the resources required to make the application work properly. In comparison with a traditional web app, a SPA is loaded only one time because all the further interactions don't require a server request, and hence the page doesn't need reloading. In SwarmAd, the HTML page will be downloaded from Swarm and then, all the data will be fetched from the blockchain and the distributed storage.

In my project, I developed the application using React. It is a Javascript framework developed by Meta. It is very popular in front-end development because it ensures very good performance and efficiency. It is also particularly interesting because it provides a clean separation between logic and UI components, as we will see in a further section. On top of that, it has a large development community, so it is well documented and there are several libraries and tools to ease the coding process.

The front-end interface has to handle both the interactions with the Ethereum blockchain and the ones with the Swarm network.

About the first point, every Ethereum client implements a JSON-RPC specification. It is a collection of methods to provide a canonical interface between clients and the Ethereum network [61].

As a second point, the web interface is remotely connected to the private gateway node via a Javascript library called Bee-js [62]. It is written and maintained by Swarm Foundation. Bee-js has a class called "Bee" to create an instance of a Swarm node. Through this instance, we can access the network and perform

5.2. STACK DESIGN

basic operations such as downloading or uploading a file. In Code 5.1, we can see a simple example of how we can interact with the Swarm network. The code snippet shows an uploading and a downloading process. It is important to notice that in every upload operation a postage batch must be attached. Indeed, as we have seen in Chapter 4, postage stamps work as spamming protecting and guaranteeing data availability for a certain amount of time.

```
1   #Create Swarm node instance
2   import { Bee } from "@ethersphere/bee-js"
3   #In this example, the node is running on localhost and default
   port
4   const bee = new Bee('http://localhost:1633')
5   #Get the postage batch
6   const postageBatchId = await bee.createPostageBatch("100", 17)
7   #Upload a file
8   const result = await bee.uploadFile(postageBatchId, "Hello World!",
   "helloworld.txt")
9   #Download using the hash reference
10  const retrievedFile = await bee.downloadFile(result.reference)
11  #Print
12  console.log(retrievedFile.data.text())
```

Code 5.1: "A simple example of Bee-js functionalities"

5.2.5 ENTERING THE WEBSITE

As we have seen earlier, Swarm also stores the front-end asset folder and it acts as the web host. Considering the Swarm built-in incentives, the web hosting is "free" because we have at least one node, i.e. the private gateway, which is taking part in the network activities. It is an advantage in comparison with the traditional Web2 solution where we would have to pay a monthly fee to a central authority to rent a server to host the application.

Swarm web hosting enables the creation of a fully decentralized application but it also entails a major issue in accessibility. Indeed, Swarm uses a content address protocol. Consequently, we need a 256-bit Swarm hash reference to get the desired content. As an example, this is a Swarm reference of a website folder after it has been uploaded to the Swarm network:

09d1d4049f7163262e5d52695256a6c68b23a77e5afb7ece6ae3252b55bbe476

It is easy to notice that it is not human-readable. As a result, it is highly inconvenient accessing a web interface by typing something like

```
bzz : //09d1d4049f7163262e5d52695256a6c68b23a77e5afb7ece6ae3252b55bbe476
```

in our browser URL search bar. On top of that, a Swarm reference changes every time we update the folder because it is computed as the Keccak hash of the folder itself.

The solution I have decided to implement is to use an ENS domain to translate a Swarm Feed reference into a human-readable domain. It is important to notice that the reference is not to the website folder but it is to a Swarm Feed. This Feed is constantly updated to maintain the reference of the latest folder containing the web interface assets. This step could be seen as a redundant overhead because translating the folder reference is more straightforward. However, any update in that folder would require changing the mapping between the reference and ENS domain. Since ENS saves its records on the blockchain, every update is performed by a transaction and hence it costs gas. In Code 5.2 we can see the script I have written to upload the website folder whenever it gets updated. This script performs two tasks: the first one is uploading the new website directory to the Swarm network, and the second one is writing the reference to that directory in a Swarm Feed.

```

1 #Swarm node instance
2 const gateway = new Bee('swarmad.xyz')
3
4 #Ethereum account and its private key
5 const owner = '0x73a4a14De3E49A63d71c7c0ff2CB3e572c102960'
6 const signer = '0
      x50f88c3b47c3c10581e227fd03fc077bd68945c83c5d7d32eb6179b06cca9480'
7
8 #Uploading the website directory
9 let options: CollectionUploadOptions = {errorDocument: "index.html",
      indexDocument: "index.html", pin: true}
10 const dirUploadResult = await gateway.uploadFilesFromDirectory(
      postageBatchId, "swarmad", options);
11
12 #Writing chunk hash to the Feed
13 const rawTopic="website-folder";
14 const topic = gateway.makeFeedTopic(rawTopic);
15 const writer = gateway.makeFeedWriter('sequence', topic, signer)

```

5.2. STACK DESIGN

```
16 const feedWritingResult = await writer.upload(postageBatchId,
    dirUploadResult.reference)
17
18 #Verifying the Feed with re-download
19 const feedVerification = await writer.download()
20 console.log('Verification result: ', feedVerification.reference ===
    dirUploadResult.reference);
21
22 #Creating chunk hash for manifest that can be used with the BZZ
    endpoint
23 const feed = await gateway.createFeedManifest(postageBatchId, '
    sequence', topic, owner);
24 const websiteURL = '/bzz/${feed.reference}/';
```

Code 5.2: "Updating the web interface via a Swarm Feed"

By design, a Feed is directly related to an Ethereum address, hence it is important to carefully select the Ethereum address that owns the website Feed. Considering my project, I think this is under the responsibility of the organization which is setting up the application and its other components, like the gateway.

As far as security is concerned, the script directly exposed the owner's private key because it is needed to write the Feed. Consequently, in a production-ready version should be preferable to use an ephemeral key or create a web interface able to interact with a wallet like Metamask.

As the last thing, the considerations made in Subsection 5.2.3 are valid also in this context. Since the website URL is like *bzz://swarm-reference*, it is required a Swarm node to communicate using the *bzz* protocol. Hence, every user, registered and not, needs an entry point to the Swarm network to access the web interface. In this regard, the gateway described in Subsection 5.2.3 comes in help. Indeed, it is configured to support *bzz.link* that allows the translation of Swarm references and ENS domains into */bzz* calls. Figure 5.2 quickly sums up the whole process:

1. User asks for *https://swarmad.bzz.link*
2. The request is handled by the gateway
3. The gateway solves the ENS domain into the feed reference
4. The gateway requests for Feed
5. The Feed is retrieved and it is pointing the latest version of the web interface

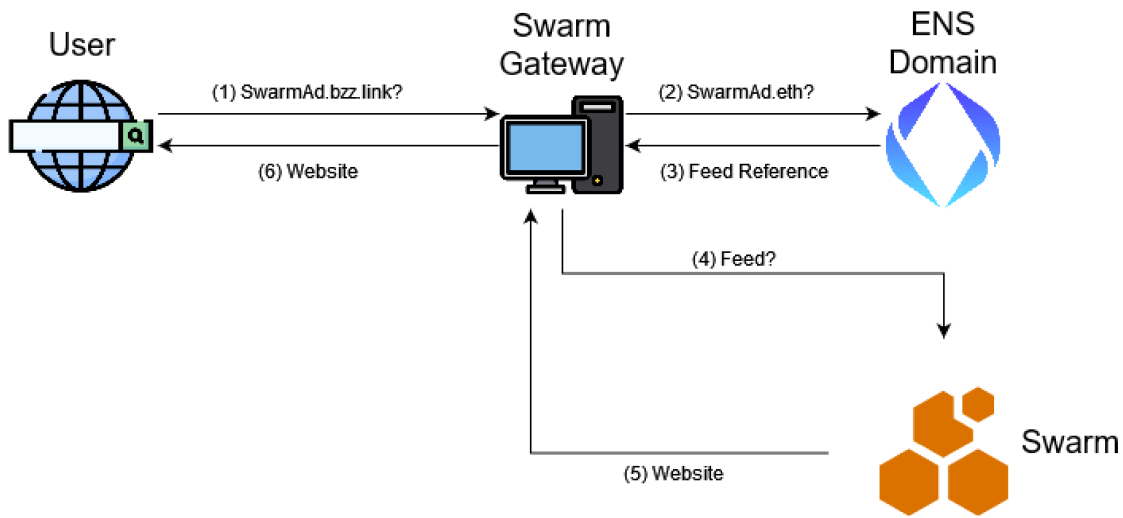


Figure 5.2: How a user will connect to the app

- Web interface is forwarded from the gateway to the end user, who is now able to surf the application

Instead, if a user owns a personal Swarm node things get different. The user himself is able to get the web interface using his node. There is no need to pass through the private gateway. In this case, the application will be available at <http://swarmad.eth.swarm.localhost:1633/>, as we can see from figure 5.3.

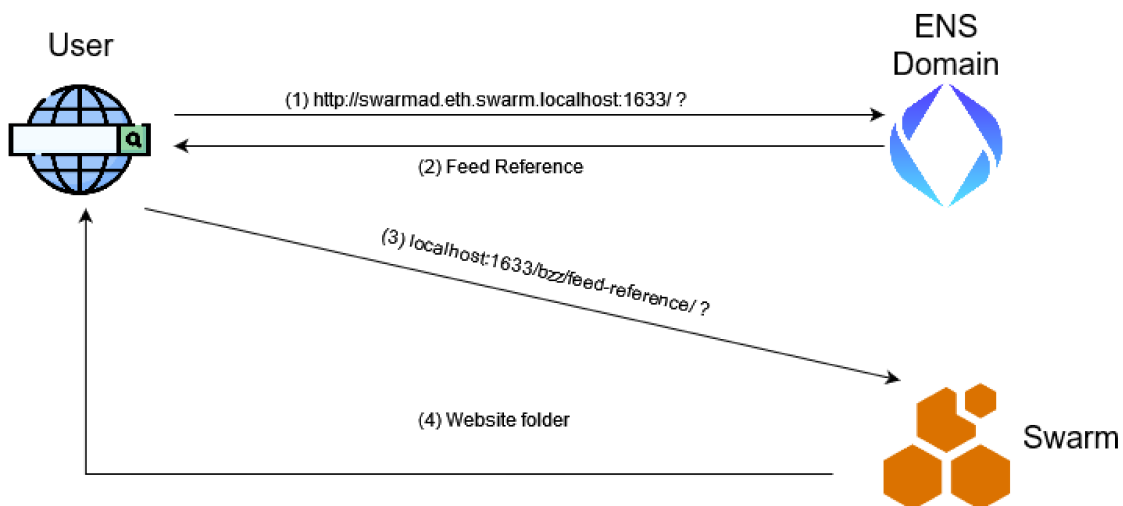


Figure 5.3: How a node owner will connect to the app

5.3 TEST ENVIRONMENT

Developing a smart contract requires a different approach in comparison with web or mobile development. Indeed, contract immutability makes it harder to fix a bug. On top of that, the cost of failure can be really high. In this regard, it is worth mentioning The DAO case where a vulnerability in the contract code caused the theft of about \$9.4 billion USD worth of ETH [63].

These considerations underline the relevance of smart contract testing. We can make a similar argument for Swarm. Posting files on the main-net is not a free operation. It also exposes the files to the whole network and hence it makes them public. As a result, I needed to create a local environment that replicates the entire stack on my local machine for developing and testing purposes. As the first thing, I have used Truffle to compile, test, debug and deploy smart contracts. It is a development environment with a rich set of useful functionalities, like automated contract testing or network management to deploy a contract on a public or private network. Especially the latter is quite interesting. Indeed, deploying and interacting with a smart contract on the Ethereum Mainnet requires the payment of the gas fees. Hence, developers need a cheap and safe version of Ethereum to run all the essential tests. To fulfill this purpose, there are two options: public or local testnet. A testnet replicates all the main-net protocols and functionalities even if it has a smaller number of nodes. The key difference is that ETH on testnet has no real value, hence deployments and interactions are free. In my setup, I have opted for a local testnet, so I have installed Ganache, as suggested by the Ethereum documentation. It is a personal blockchain for Ethereum application development. It also provides a set of ten Ethereum accounts, each of which is funded with 100 ETH. They are easy to impersonate because it is not required any private key. For example, they can be directly imported to Metamask in order to interact with the dApp. I have used these ten Ethereum address to test all the app functionalities without spending real ETH in transactions and gas-fees. Ganache also provides a block explorer. It is a useful feature to rapidly check the blockchain state and the data stored on it.

I also have created a Swarm local testnet. For this purpose, I have used *bee-factory*. It is developed by the Swarm Foundation. It is a Command Line Interface (CLI) to spin up a Docker cluster made of six different containers. [64]. A Docker container is a standalone image that includes all the system

tools, system libraries, and settings that are needed to run an application. In the case of the *bee-factory*, five containers have the image of a Swarm full node. Consequently, running these container is equal to run five Linux server with Bee client installed on them. Meanwhile, the last container is running a blockchain to handle the Swarm incentive system. As a result, I can run a local Swarm network by typing a simple command in my CLI. Thanks to this solution, I can run every test I need without burning real BZZ tokens and posting content on the Swarm main-net. By design, Swarm runs on *localhost:1633* for the main-net













	NAME	IMAGE	STATUS	PORT(S)
	bee-factory-worker-4 332becc0af6d 	etherspher	Exited	41633,41634,41635,43000
	bee-factory-worker-3 76e57b291b72 	etherspher	Exited	31633,31634,31635,33000
	bee-factory-worker-2 70073d6af86a 	etherspher	Exited	21633,21634,21635,23000
	bee-factory-worker-1 ac1dea3222d1 	etherspher	Exited	11633,11634,11635,13000
	bee-factory-queen 79be86ff9039 	etherspher	Exited	1633,1634,1635
	bee-factory-blockchain 0789b840298d 	etherspher	Exited	9545

Figure 5.4: Bee-factory containers

and *localhost:1635* for exploiting the debug API. To ease the remote connection to these entry points, each *bee-factory-worker* has these ports bound to a port on my local machine. As we can see in Figure 5.4 under the "PORTS" column. As a result, I can connect to any *bee-worker* from my machine by *localhost:x1633* where *x* stands for the *bee-worker* number. On top of that, I selected *bee-factory-worker-1* to act as the private gateway. To do so, I installed on it the latest version from the Swarm Github repository. Then, I configured the gateway as follows:

- POSTAGE_DEPTH=20
- POSTAGE_AMOUNT=1000000
- BEE_DEBUG_API_URL=http://localhost:1635
- CID_SUBDOMAINS=true

5.4. TOKENOMICS

- ENS_SUBDOMAINS=true

As we have seen in Chapter 4, the first two values are the ones suggested by Swarm Foundation for purchasing postage stamps. By setting these values, the gateway is enabled to auto-buying stamps and it automatically attaches one to the upload request. This functionality is relevant because it saves time during the upload process. Indeed, buying a stamp could require even a minute. Instead, the last two parameters are set to true in order to enable ENS and CID resolution. In my test environment, they are not mandatory because it is still not possible to solve an ENS domain since it works only on the Ethereum Mainnet or public test-net like Goerli. Anyway, it is a crucial feature in the final version of the application as we have seen in the previous section.

As the last step, I have manually changed the `/src/utils/docker.ts` file in *bee-factory* repository to bind the Bee client port 3000 to the port `x3000` of my local host, where *x* stands for the Bee worker number. Consequently, since the private gateway is running on `localhost:3000` of *bee-factory-worker-1*, I can access the private gateway on port 13000 of my local machine. As a result, once SwarmAd folder is uploaded to the local test-net, it is accessible at <http://localhost:13000/bzz/feed-reference/index.html>.

To sum up, the local environment is made of:

- a local Ethereum blockchain
- a set of 10 Ethereum accounts, each of them founded with 100 ETH
- five Swarm nodes
- a local Ethereum blockchain for Swarm incentives
- one private Swarm gateway running on `localhost:13000`
- a Swarm Feed to save the reference of the latest website version

5.4 TOKENOMICS

In distributed platforms and systems, a tokenomics can be crucial to make the whole application sustainable. As we have seen in Chapters 3 and 4 even Ethereum and Swarm have to rely on such a system of rewards and penalties to ensure all the desired services. SwarmAd is not different. If we recall the target use case where each registered user has his own enterprise and a collection of

products to show, it is easy to notice how users are each others' competitors. As a result, they could be motivated to act selfishly because they don't trust the other members or, even worse, because they want to obstruct them from entering the platform or posting products. Under these considerations, the need for a proper incentive system is clear.

5.4.1 TOKEN DESIGN

The main idea is to design a reputation schema to identify and reward the users who are cooperating and helping to sustain the platform. Since we are talking about an advertising platform, giving a visibility boost to certain enterprises is a relevant edge over their competitors. Consequently, a useful reward could be reserving a certain spot on the website homepage for the most active users. In a further subsection, we will have a look at the content organization. As the first thing, I have created a reputation score to identify the best users. It can increase or decrease according to the user's actions. The design process has not been trivial. Indeed, the simplest solution would be to create an ERC20 token. In this way, users could gain tokens and then burn them into rewards. Nevertheless, there is a major contradiction. Indeed, if the token is spendable to get rewards, it is also possible to sell and buy tokens from other parties. As a consequence, it is possible to buy and sell reputation. It is easy to see that the more liquid a token is, the less meaningful the reputation is. In such a system, even the less-active user could easily buy thousands of tokens and overtake the other members, who actually earned their tokens from the platform reward system.

I would like to make a short digression because this is the same reason why a cross-incentive system linked with Swarm is not feasible. Indeed, as we have seen in Chapter 4, a Bee client can gain BZZ tokens from its contribution to the Swarm network. BZZ tokens are also used as currency, for example for buying stamps. Consequently, it must be purchasable in Decentralized Finance (DeFi) exchanges. In support of this, buying some BZZ tokens is one of the first steps while setting up a Swarm node. Finally, BZZ tokens are not a reliable measure of how much a user is contributing in terms of bandwidth or storage because everyone can buy thousands of them.

Going back to our issue, we have to distinguish two functionalities:

- signaling: the token identifies users who are active and helpful in commu-

5.4. TOKENOMICS

nity duties

- compensating: the token can be spent to get rewards

If we want a token to measure reputation, it must be only a signaling token. Creating one of this kind is straightforward: we can use the ERC-20 standard and make it non-transferable. The latter expedient does not allow to buy and sell the tokens hence it can be a trustworthy measure of users' reputation. On the other hand, a single token solution could restrict the tokenomics benefits because users would not be able to redeem rewards according to their liking. It could fit in competitive video games where organizations have to keep track of users' points and create a global scoreboard. Considering SwarmAd, I think it is important to provide also a proper compensation schema. As a result, the best solution is creating a two-token system to provide signaling and compensating functionalities in a separate way. In detail, SwarmAd has:

- SwarmAd Reputation Point (SWRP): a signaling token to measure the users' reputation
- SWCT: a spendable token to buy rewards

In my idea, users can gain SWRP from various activities like voting a poll or reporting an inappropriate item. SWRP generates SWCT according to an interest rate. We can make a parallelism with the Ethereum validators we have seen in Chapter 3. Each validator is staking Ethereum and its deposit can increase if it follows the rule, or can be slashed if it goes against them. In SwarmAd, users "stake" reputation points which "produce" spendable coins. Then, each user can withdraw SWCT and buy rewards. Thanks to this approach, we create a sort of feedback loop: users are interested to claim rewards using SWCTs but SWCTs are minted only by SWRPs. Hence, users are motivated to farm SWRPs to produce SWCTs. Even if a user is not interested in buying rewards, he can sell SWCTs to others. Finally, he has tangible rewards for his actions.

REPUTATION POINTS

SWRP is a non-transferable ERC20 token. It acts as a *signaling* token so it represents the user reputation in SwarmAd.

In my design, Reputation Points are gained by:

- voting a poll

- creating a poll
- reporting an inappropriate item
- reporting a user who is not acting properly

On the other hand, RPs are burned when

- a user doesn't vote
- a user reports another member or a product but the community rejects the report

Nevertheless, I have felt the need for adding a simple action that can be performed daily. Indeed, if we recall the use case, we do not expect many registered users so the polls for accepting a newcomer could not be very frequent. On top of that, even the last newcomer must have a chance to earn RPs. We can apply the same reasoning for reports: inappropriate items or fake requests could not be frequent because these actions can be expensive due to gas fees. In other words, it is legit to not expect excessive spamming. Under these considerations, daily rewards are crucial to prevent the tokenomics from staling. Inspired by a Peepeth functionality we have seen in Chapter 2, I have introduced a similar one: the *super-like*. It works as follows. Every day, each registered user can vote a product he likes. Obviously, it is not possible to vote for a product the user himself posted. As a reward, users receive a small amount of SWRP. This incentive could overcome the fear of favoring a competitor. It is possible to vote only one product per day. It can be seen just as a login bonus but it also has a meaningful impact on the platform. Indeed, it has three advantages:

- it is easy to perform
- it is useful to reward quality content
- it keeps active the interest in the platform

5.5 IMPLEMENTATION

After having a look at the general structure in the previous sections, we are now able to delve into the actual code implementation.

First thing, I have created the project folder starting from a boilerplate provided by Truffle. It is the *react-truffle* box. It allows to quickly start to write and

5.5. IMPLEMENTATION

compile smart contracts and then interact with them using a React application. After downloading the Truffle box, the project folder is divided into two main directories: *truffle* and *client*. The first one contains the smart contracts and the files for deploying them in the Ethereum blockchain. The second one has all the front-end assets to build the web interface.

5.5.1 BACK-END CODE

If we open the *truffle/contracts* folder we find the following five contracts:

- Migrations.sol
- SwarmAd.sol
- SwarmAdGovernor.sol
- SwarmAdRewarder.sol
- SwarmAdReputationPoints.sol
- SwarmAdCommunityToken.sol

First, it is important to explain how they are able to interact with each others in a safe and authenticated way. Then we will analyze each contract following the order written above.

ACCESS CONTROL

In smart contracts, it is fundamental to define who is allowed to call certain functions and perform the related actions. Indeed, since all the code is public and visible via blockchain, potentially anyone is able to call an external contract. In SwarmAd, we have different contracts and privileged users that have to interact with each other to use the different functionalities. For example, if an enterprise has to be banned from the platform, it is the responsibility of Governor. Or even, we want to restrict the ability to post a product only to registered users. As a result, I have decided to implement a Role-Based Access Control (RBAC) using an API developed by OpenZeppelin [65]. It suits our needs because it provides different levels of authorization. We can distinguish three:

- *GOVERNOR_ROLE*: restrict access only to the Governor
- *ENTERPRISE_ROLE*: specific for registered members. It is granted when an enterprise is accepted in the platform. It is revoked if the enterprise is removed.

- *MINTER_ROLE*: it identifies the entities able to mint SWRP and SWCT, i.e. SwarmAd and Governor

Nevertheless setting up the RBAC has not been trivial. Indeed, we have a sort of circular dependency in contract interactions. To better explain the issue, consider the case in which Governor has to delete an enterprise. It needs the address of the SwarmAd contract to execute its task and at the same time, SwarmAd needs to know the Governor address to grant its role and allow it to perform a critical operation such as deleting a user. Since the deployment is sequential, if we first deploy Governor, we can tell its address to SwarmAd but Governor does not know the SwarmAd one. And vice versa. Luckily, a smart contract address is computed in a deterministic way. Indeed, its address is made from the deployer address and a nonce, which is the number of transactions executed by the deployer. According to Ethereum Improvement Proposal (EIP) 161 [66], the nonce starts from one. As a result, we can pre-compute these addresses before the actual deployment. The procedure is the following:

- convert owner's address and nonce into an array of bytes
- encode the array
- compute its keccak hash
- convert to hex

Imagine we are using a new account to deploy the contract, we can predict all the addresses we need as it is shown in Code 5.3. Notice that the nonce starts from one and then it increases after every deployment.

```

1     migrations = address(uint160(uint256(keccak256(abi.
      encodePacked(bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x01)
      )))));
2     swarmad = address(uint160(uint256(keccak256(abi.encodePacked(
      bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x02) )))));
3     governor = address(uint160(uint256(keccak256(abi.encodePacked(
      (bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x03)))))));
4     swrp = address(uint160(uint256(keccak256(abi.encodePacked(
      bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x05) )))));
5     swct = address(uint160(uint256(keccak256(abi.encodePacked(
      bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x06) )))));

```

Code 5.3: "Pre-computing addresses"

5.5. IMPLEMENTATION

At first glance, I have thought of writing a configuration contract to store all the addresses and set up the access control. Thanks to inheritance, *SwarmAd.sol* and the other contracts would have known addresses and access control roles. In Solidity, we can provide a single level of inheritance by using the keyword *is*. Nevertheless, this solution has seemed like an overkill for my project because RBAC involves only three contracts. Surely if I had had more contracts, I would have decided for introducing such a configuration file. Indeed, it provides an easy way to access the different addresses and to change the access control in case of contract substitution. On the other hand, it requires deploying one contract more and it also requires an external call every time we need an address. Under these considerations, I have decided on an easier model where each contract computes the addresses it will need and it grants the roles.

After solving this dilemma, the actual implementation has been straightforward. As we can see in Code 5.4, each role is defined as a public constant. In this example, the contract computes Governor address and grants the related role when deployed. Then the function modifier *onlyRole(A_ROLE)* restricts access to users who have the role specified in the parameter.

```
1 #Role definition
2 bytes32 public constant GOVERNOR_ROLE = keccak256("GOVERNOR_ROLE");
3 bytes32 public constant ENTERPRISE_ROLE = keccak256("ENTERPRISE_ROLE"
4 );
5 #Grant Governor its role during the contract deployment
6 constructor() {
7     governor = address(uint160(uint256(keccak256(abi.encodePacked(
8         bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x03))))));
9     _grantRole(GOVERNOR_ROLE, governor);
10 }
11 #Restrict a function to Governor
12 function moveEnterprise(address a) public onlyRole(GOVERNOR_ROLE){
13     ... }
14 #Restrict a function to an Enterprise
15 function updateENAME(string memory newName) public onlyRole(
16     ENTERPRISE_ROLE){ ... }
```

Code 5.4: "Access control in SwarmAd"

DEPLOYMENT

To deploy smart contracts on an Ethereum network, we need a set of scripts called *migrations*. For my project, I have written the following ones: *1_initial_migrations.js* and *2_deploy_SwarmAd.js*. The first one is to deploy *Migrations.sol*. It is a smart contract written by Truffle to keep track of which migrations have been done on the current network. Instead, the second migration file is for deploying all the smart contracts needed in SwarmAd. The order is important because the nonce changes after each deployment. Since I have built the RBAC on predicting the addresses, the order has to be consistent with the one used for the computations. In Code 5.5 we can see the sequence. The method `artifacts.require()` returns a contract abstraction we can call in the further code lines.

```

1 const SwarmAd = artifacts.require("SwarmAd");
2 const SWRP = artifacts.require("SwarmAdReputationPoints");
3 const SWCT = artifacts.require("SwarmAdCommunityToken");
4 const Governor = artifacts.require("SwarmAdGovernor");
5 const Rewarder = artifacts.require("SwarmAdRewarder");
6
7 module.exports = function(deployer) {
8   deployer.then(async () => {
9     await deployer.deploy(SwarmAd);
10    await deployer.deploy(Governor);
11    await deployer.deploy(Rewarder);
12    await deployer.deploy(SWRP, Rewarder.address);
13    await deployer.deploy(SWCT, Rewarder.address);
14   });
15 };

```

Code 5.5: "Deploying smart contracts"

SWARMAD

SwarmAd.sol is the core contract of the platform. It contains all the data related to enterprises and products. It is also in charge of providing the desired functionalities like adding a new product or a new enterprise. As the first thing, I have decided to follow Martini's design [3] so I have modeled the two main entities, i.e *Enterprise* and *Product*, as structures. In Solidity, a struct is a composite data type to define a group of several related variables in a single one. The instances of *Enterprise* and *Product* are then stored in mappings. A mapping

5.5. IMPLEMENTATION

is a sort of hash table: given a key, it returns the associated value. In our case, we have *eStructs* and *productStructs*. The first one takes as input the owner's Ethereum address and it returns his enterprise. Meanwhile, the second takes a product identifier and retrieves the product itself. Nevertheless, mappings are not iterable. As an example, it is not possible to iterate over *eStructs* and print all the names of the registered enterprises. Of course, this is a very restricting point. As suggested by Solidity Docs [67], the solution is to create an array storing the mapping keys and then iterate over it. Consequently, I have created *eList* and *productList* to store all the owners' addresses and product identifiers, respectively. Finally, in Code 5.6 we can see how an Enterprise is modeled.

```
1 struct Enterprise{
2     #Enterprise Name
3     string eName;
4     string eMail;
5     #Owner's Ethereum Address
6     address eAddress;
7     #CID provided by Swarm when uploaded
8     string profileImageHash;
9     #Index in list of all Enterprises
10    uint indexEList; //position in eList
11    uint indexWaitingList;
12    #PID of all posted products
13    bytes32[] enterprisePidList;
14 }
15 }
```

Code 5.6: "Defining Enterprise"

Beyond the informative and descriptive fields like *eName* or *eMail*, we can notice *indexEList* which is the variable to store the position in the list of enterprises. On top of that, there is also *enterprisePidList* which stores all the products posted by the enterprise in question. It is not mandatory but it is very functional because it allows a quick retrieval on the front-end side

For example, if we want to get a certain enterprise by knowing its owner's address, we can get the information from the *eStructs* mapping. It is shown by Code 5.7.

```
1 function getEnterprise(address a) public view returns(string memory
    eName, string memory eMail, string memory imgHash){
2     return(eStructs[a].eName, eStructs[a].eMail, eStructs[a].
    profileImageHash);
```

```
3 }
```

Code 5.7: "Querying Enterprise struct"

In a symmetrical way, *Product* and its related functions are defined. As we can see in Code 5.8, we save two distinct indexes: one is specific for the enterprise product list while the other one is for the array containing all the products in the platform. The latter is a small addition in comparison with Martini's design. I have thought that it was needed to improve the data fetching process and item rendering on the front-end interface.

```
1 struct Product{
2     #Keccak hash computed during item creation
3     bytes32 pid;
4     #Owner
5     address enterprise;
6     string productName;
7     string productImageHash;
8     string productDescription;
9     uint productPriceInWei;
10    #Indexes
11    uint indexProductStructs;
12    uint indexEnterprisePidList;
13 }
```

Code 5.8: "Defining Product"

Moving forward, the creation of a new product is a good way to show how things work. In Code 5.9 we can see the function I have written. First thing, we can see that the function usage is restricted to ones who have the *ENTERPRISE_ROLE*. For security reasons, the enterprise creating the product is not taken as input but it is extracted as the caller of the Ethereum transaction using *msg.sender*. Then, it is computed the product identifier, or PID, computing the *keccak256* of enterprise address, product name, and timestamp. In this way, uniqueness should be ensured. As the last thing, the PID is added to the two lists described above.

```
1 function createNewProduct(
2     string memory name, string memory img, string memory description,
3     uint price)
4     #Access Control
5     public onlyRole(ENTERPRISE_ROLE){
6     #Compute PID
```

5.5. IMPLEMENTATION

```
6   bytes32 pid = keccak256(abi.encodePacked(eStructs[msg.sender].
   eAddress, name, block.timestamp));
7   Product memory p = Product(pid, msg.sender, name, img,
   description, price, productList.length, eStructs[msg.sender].
   enterprisePidList.length );
8   #Add to indexes
9   productStructs[pid]=p;
10  productList.push(pid);
11  eStructs[msg.sender].enterprisePidList.push(pid);
12 }
```

Code 5.9: "Create a new product"

As the last functionality, we can have a look at the *superlike*. I have written two mappings: one, *superlike*, to map the user's address into the product he liked. The second, *superlikeTimelock* is to keep track of when a user triggers the superlike function. In this way, the system is able to check if it has passed at least one day and the user can trigger again the function. As we can see in Code 5.10, *SwarmAd* makes also an external call to *Rewarder* to reward the user. At the moment, the function is restricted only to registered users but there is a margin for improvement. For example, it could be developed to become a rating system where also external users can vote for products based on their purchase history. In this sense, since RBAC ensures an high flexibility level, the implementation could require just a little adjustment.

```
1 function assignSuperlike(bytes32 pid) public onlyRole(ENTERPRISE_ROLE
   ){
2   #check item exists
3   require(productStructs[pid].enterprise != address(0));
4   #check owner is not caller
5   require(productStructs[pid].enterprise != msg.sender);
6   #check time lock is expired
7   uint256 differenceTimestamp = SafeMath.sub(block.timestamp,
   superlikeTimelock[msg.sender]);
8   uint256 differenceInDays = SafeMath.div(SafeMath.div(SafeMath.
   div(differenceTimestamp, 60), 60), 24);
9   require(differenceInDays > 0, "time difference is not enough");
10  #update
11  superlike[msg.sender] = pid;
12  superlikeTimelock[msg.sender] = block.timestamp;
13  #call Rewarder to award the user
```

```
14 SwarmAdRewarder(rewarder).addRPsToAccount(msg.sender, 5);
```

Code 5.10: "Superlike function"

SWARMADGOVERNOR

SwarmAdGovernor, or simply Governor, is the smart contract entitled to manage the voting mechanism inside of the platform. Whenever a decision is to be taken, it creates a poll and it allows each registered user to express his opinion by voting.

In my first design, I have thought about implementing a contract provided by OpenZeppelin. Nevertheless, I was skeptical because it was based on coin voting. By definition, coin voting means that a single token is equal to a vote so the voting power is directly proportional to a user's wealth. As a result, it can quickly become a plutarchy if the voting power is condensed into a few wallets. As underlined by Vitalik Buterin [68], coin-based governance has three major issues:

- small-holders have no meaningful influence on decisions
- DAO vision becomes the coin-holders' interest
- there is a clear conflict of interests because the governance is over-exposed to the will of a specific elite group. For example, an investment fund that also holds tokens of other DeFi platforms.

From one side, creating a governance based on Reputation Points seems an intriguing solution because it gives more power to users who actually contribute to the platform. On the other side, I do not think it is the right solution because it does not ensure a democratic voting procedure. In particular, it is not right that a newcomer's vote is worth less than the one from an experienced user. For the above reasons, I have decided to move to proof-of-personhood. In this system, every registered member can express his vote and it counts as one. I have thought about using an alternative version of the OpenZeppelin governor that supports the ERC-721 token, namely Non Fungible Token (NFT), instead of the ERC-20. Since a standard NFT can be sold and bought, it needs a little adjustment to provide proof-of-personhood. In detail, it must be not transferable. A token of this kind is called *soulbound* [69]. The terminology comes from World of Warcraft, a famous video game, where some special items cannot be transferred to other players. In a similar way, I have thought to create a soulbound badge to signal the membership in SwarmAd. Even if the idea

5.5. IMPLEMENTATION

of soulbound items is fascinating, I believe it is an overkill if we consider the platform target. Indeed, SwarmAd keeps track of every user thanks to *eList* and *eStructs* so it is straightforward to know who is registered to the platform and who is not. Under this consideration, I have decided to model the Governor in a simpler way. Since the OpenZeppelin one was not a good fit, I have written the contract from scratch.

Similar to *Enterprise* and *Product*, I have modeled *Poll* as a structure where are stored all the information needed. It has a state which defines the set of actions Governor and users can perform. The state can be *ACTIVE*, if the voting procedure is still open, or *EXECUTED* if the time is expired and the related operations have been performed. It also stores the number of votes for the proposal, the ones against and the number of users who decided to abstain. In Code 5.11, we can see that these variables have type *uint64*. Indeed, I expect a small number of registered users and 64 bits should be enough to represent it. Packing these variables together to take up a single 256-bit slot of memory saves some gas fees [70]. Since the registered users can change during the voting procedure, Governor takes a snapshot of the registered users when the poll is created. In this way, when the poll is closed, the Governor is able to distinguish the users who have not voted for choice from the ones who do not have the voting right. We can apply this reasoning also for quorum computation: since the number of registered users is variable, it is important to have the number of voters and compute the quorum before the poll starts.

```
1 struct Poll{
2     uint64 state;
3     uint64 votesFor;
4     uint64 votesAgainst;
5     uint64 votesNull;
6     uint256 pollId;
7     uint256 maxVoters;
8     uint256 quorum;
9     uint256 votingStart;
10    uint256 votingEnd;
11    address swarmad;
12    address proposer;
13    address [] voters;
14    address [] ableToVote;
15    bytes[] calldatas;
16    string description;
```

```
17     }
```

Code 5.11: "Defining Poll"

Calldata is the function to execute if the proposal is approved. In Code 5.12, we can see how *SwarmAd* calls the governor to create a poll for accepting a newcomer. In this example, the function to execute is *moveAddress* because if the community votes for accepting the newcomer, its enterprise must be moved from the waiting list to the actual list of enterprises. To do so, *SwarmAd* encodes the function and its parameter in ABI format, stores the result in a raw bytes array and passes it to Governor in the *Calldata* field.

```
1  function createEnterprise(string memory name, string memory mail,
2     string memory imgHash) public{
3
4     if(eList.length<1){/* ...the first enterprise is directly added
5     ...*/}
6     else{
7         #Add to waiting list
8         waitingList.push(msg.sender);
9         Enterprise storage e = waitingListStructs[msg.sender];
10        e.eName = name; e.eMail = mail; e.profileImageHash = imgHash; e
11        .indexEList = 0; e.indexWaitingList = waitingList.length -1;
12        #Encode function to execute
13        bytes[] memory transferCalldata = new bytes[](1);
14        transferCalldata[0] = abi.encodePacked(bytes4(keccak256('
15        moveEnterprise(address)')), abi.encode(msg.sender));
16        #Create poll
17        SwarmAdGovernor(governor).createPoll(address(this),
18        transferCalldata, "Accept new user");
19    }
20 }
```

Code 5.12: "Calling Governor to create a poll"

Taking a step back, the waiting list is a mapping similar to *eStructs*. If the registered users decide to accept the newcomer, the Governor will automatically perform the function *moveEnterprise* to insert the new enterprise in *eList* and *eStructs*. In other words, the user is registered and can use all the platform functionalities. As we can see in Code 5.13, the user is granted the *ENTERPRISE_ROLE*. It is also relevant to notice that the function can be performed only by the Governor because of the access control modifier.

5.5. IMPLEMENTATION

```
1 function moveEnterprise(address a) public onlyRole(GOVERNOR_ROLE){
2     eList.push(a);
3     _grantRole(ENTERPRISE_ROLE, a);
4     Enterprise storage e = eStructs[a];
5     e.eName = waitingListStructs[a].eName;
6     e.eMail = waitingListStructs[a].eMail;
7     e.profileImageHash = waitingListStructs[a].profileImageHash;
8     e.indexEList = eList.length-1;
9     e.indexWaitingList = 0;
10    removeEnterpriseFromWaitingList(a);
11    emit moveFromWaitingList(a);
12 }
```

Code 5.13: "Moving an Enterprise from waiting list"

Governor saves all the polls in a mapping called *pollMap*. It maps a *pollId*, which is a unique identifier, into the corresponding poll object. Similar to *SwarmAd.sol*, the keys are stored in an array to solve the mapping iterability issue. A poll is valid in a certain period which is established by *votingStart* and *votingEnd*. If a user tries to cast a vote after the timestamp declared in *votingEnd*, the operation is not permitted. However, the poll closure is not performed automatically. Scheduling a function call is not trivial and generally speaking, it involves third-party services. Martini proposed a solution using AION smart contract [71] but it is no longer feasible since it is not maintained anymore. Another similar project, namely Ethereum Alarm Clock [72], is deprecated as well. SwarmAd tokenomics here comes in help. In my solution, closing a poll is a duty of users who have voted. By default, each voter gets a certain amount of Reputation Points when the poll is closed. On top of that, the one who actually closes the poll by calling the specific function is rewarded with an additional amount of Reputation Points which is inversely proportional to the amount of time passed between the poll deadline and the moment in which the poll is closed. In this way, voters are motivated to close the poll but they are also motivated to do it quickly. To sum up, a voter calls the *closePoll* function which makes the Governor compute the winner and execute the related action if the deadline is met and the quorum is reached. At the moment, I set the voting period to one week and the quorum to 51% of voters. As the last thing, the Governor distributes Reputation Points to the user who has closed the poll and to the ones who have voted. On the other hand, users who didn't vote will lose a certain amount of

Reputation Points because of their carelessness. On top of that, if the voting is about reporting a user or a product and the community decides against the ban, the user who has proposed the poll gets his Reputation Points slashed. The penalty is needed to avoid spamming of ban requests to mine the visibility of other users. Code 5.14 shows the different functions involved in the process.

```

1 function closePoll(uint pollId) public{
2     require(!isVotingActive(pollId));
3     require(!isPollExecuted(pollId));
4     require(hasVotingRight(pollId, msg.sender));
5     if(isQuorumReached(pollId) && isMajorityVotingFor(pollId) )
    executePoll(pollId);
6     giveRewards(pollId);
7     pollMap[pollId].state = EXECUTED;
8     emit pollExecuted(pollId);
9 }
10
11 function giveRewards(uint pollId) internal {
12     require(hasVotingRight(pollId, msg.sender), "Governor: no
    right to vote");
13     SwarmAdRewarder Rewarder = SwarmAdRewarder(rewarder);
14     uint256 amount = 10;
15     address[] memory voters = pollMap[pollId].voters;
16     for(uint i=0; i<voters.length; i++){
17         #Punish who did not vote
18         if(hasNotVoted(pollId, voters[i])) Rewarder.
    burnRPsToAccount(voters[i], amount);
19         #Reward voters
20         else Rewarder.addRPsToAccount(voters[i], amount);
21     }
22     #Compute delay from poll closure and poll deadline
23     uint256 delay = SafeMath.sub(block.timestamp, pollMap[pollId
    ].votingEnd);
24     uint256 delayInHours = SafeMath.div(SafeMath.div(delay, 60),
    60);
25     uint256 delayFactor = Math.max(1, delayInHours);
26     #Compute reward for closing the poll according to delay
27     uint256 closeReward = SafeMath.div(amount, delayFactor);
28     emit assignCloseReward(closeReward, delayInHours, delayFactor
    );
29     #Reward who closed the poll
30     Rewarder.addRPsToAccount(msg.sender, closeReward);

```

5.5. IMPLEMENTATION

31 }

Code 5.14: "Procedure to close the poll"

TOKEN IMPLEMENTATION

Both *SwarmAdReputationPoints.sol* and *SwarmAdCommunityToken.sol* are written following the standard implementation of ERC-20 provided by OpenZeppelin [65]. Concerning Reputation Points, I have written only one addition to make the token non-transferable, as we can see in Code 5.15. The function *beforeTokenTransfer* is a hook that is called before any transfer of tokens. As a result, if the condition in *require* clause is not satisfied, the transfer is blocked. Since I want the contract to be able to mint new SWRPs or burn SWRPs when it is needed, the *require* clause allows the address 0 to call the function and hence perform or receive a transfer. Any transfer that is not performed from or to the contract address, is blocked.

```
1 function _beforeTokenTransfer(  
2     address from, address to, uint256 amount)  
3     internal override(ERC20){  
4         require(from == address(0) || to == address(0),  
5             "Error: Reputation Points are not transferable");  
6         super._beforeTokenTransfer(from, to, amount);  
7     }
```

Code 5.15: "Making ERC20 token non-transferable"

The main problem is how we can compute the interest on the reputation points. As we have already discussed when describing Governor, scheduling a smart contract call is not trivial and requires a third-party service. On top of that, the computations could be performed every hour or every day, hence relying on such a service could be expensive and not sustainable because of transaction costs. For these reasons, I have opted for a lazy evaluation solution.

As the first thing, I have thought about computing the interests when a user asks to redeem them. A sort of simple interest computed according to the number of Reputation Points at stake. The equation is the following:

$$I = (1 + r) * t$$

where r is the interest rate and t the time. In our case, the interest rate is an

arbitrary value. The period is the interval between the last time a user has redeemed its coins and a new request. At a certain frequency, each user can press a "Redeem" button and cash out his SWCT. Nevertheless, this schema is simplistic because interests depend only on the capital at stake during the redeeming phase. Imagine Alice, a user who has 1 SWRP in stake for 29 days. Then, she earns 29 SWRP. If she presses "Redeem", interests are computed for 30 days on 30 SWRP. In fact, she has staked 30 SWRP only for one day. A better evaluation can be performed when a user gains new Reputation Points. Because of smart contract calls, the system is perfectly aware of token transfers. The only drawback is that a user could not be able to cash out the SWCT if he doesn't earn RPs for a long time. It is a very remote scenario since each user can earn at least one RP every day thanks to the super-like.

To make the functioning clearer, we can recall the previous example: Alice earned 1 RP on Monday. After one month, she earns other 29 RPs so she actually staking 30 RPs. When she receives the new points, the smart contract computes the interests she matured on the RPs she has staked for one month. The next time she will earn RPs, the interest will be computed on 30 RPs.

As the last thing, I have thought it was better to separate the logic from the tokens to have a higher degree of upgradability. Indeed, if logic and token are together in a contract, a bug could make the users lose all the tokens in the wallet. By separating the two phases, we can fix the logic without touching the users' wallets. Under these considerations, I have decided to write a third contract, called *SwarmAdRewarder.sol*

It has to:

- mint reputation points
- burn reputation points
- compute interests whenever a user gains reputation points
- keep track of interests
- mint community token when a user redeems its coins

It has two mappings:

- *SWCTVault*: to keep track of the interests gained by each user
- *addressToLastInterestComputation*: it saves the last time a user gained SWRP

5.5. IMPLEMENTATION

Since it has the task of assigning SWRP to users, it knows every time a user gains SWRPs. When this happens, it computes the interests using the function in Code 5.16.

```
1 function computeCompoundInterest(uint256 balance, address a) internal
  view returns(uint256){
2     uint256 duration = SafeMath.sub(block.timestamp,
  addressToLastInterestComputation[a]);
3     uint256 durationInDays = SafeMath.div( SafeMath.div (SafeMath
  .div(duration, 60), 60), 24);
4     uint256 compoundInterest = SafeMath.mul( SafeMath.mul(
  durationInDays, balance), dailyInterestRate);
5     return compoundInterest;
6 }
```

Code 5.16: "Rewarder computes interests"

In Code 5.17, we can see how it can mint or burn SWRP belonging to a certain user.

```
1 function addRPsToAccount(address reward, address a, uint256
  amount) public onlyRole(MINTER_ROLE){
2     SwarmAdReputationPoints SWRP = SwarmAdReputationPoints(reward
  );
3     uint256 balance = SWRP.balanceOf(a);
4     uint256 interest = computeCompoundInterest(balance, a);
5     if(interest>0){
6         uint256 newVaultValue = SafeMath.add(SWCTVault[a],
  interest);
7         SWCTVault[a] = newVaultValue;
8     }
9     SWRP.mint(a, amount);
10    addressToLastInterestComputation[a] = block.timestamp;
11 }
12
13 function burnRPsToAccount(address reward, address a, uint256
  amount) public onlyRole(GOVERNOR_ROLE){
14    SwarmAdReputationPoints SWRP = SwarmAdReputationPoints(reward
  );
15    uint256 balance = SWRP.balanceOf(a);
16    uint256 interest = computeCompoundInterest(balance, a);
17    if(interest>0){
18        uint256 newVaultValue = SafeMath.add(SWCTVault[a],
  interest);
```

```

19     SWCTVault[a] = newVaultValue;
20   }
21   SWRP.burn(a, amount);
22   addressToLastInterestComputation[a] = block.timestamp;
23 }

```

Code 5.17: "Minting and burning RPs"

It is relevant to recall SwarmAd and Governor have *MINTER_ROLE* so they are the only two able to add new RPs to a user wallet.

As the last thing, there is also a *redeem* function to convert the value stored in the user's vault into an actual amount of SWCT. Code 5.18 shows the process.

```

1 function redeem(address coins) public{
2     uint256 amount = SWCTVault[msg.sender];
3     require(amount>0);
4     SWCTVault[msg.sender] = 0;
5     SwarmAdCommunityToken SWCT = SwarmAdCommunityToken(coins);
6     SWCT.mint(msg.sender, amount);
7 }

```

Code 5.18: "Redeeming Community Token"

5.5.2 FRONT-END CODE

The web interface is written in React. Since it is a SPA, the content is dynamically loaded. The interface has a general schema made of a header, a body and a footer. The body is changed according to the user's actions. In other words, while there is a general static structure, the user is also able to navigate through different pages that are loaded on the body. The routing is handled by an *HashRouter* provided by *react-router-dom*. According to the final design, I imagine having five different pages:

- Home: it shows enterprises and products
- Register: a registration form to enable the creation of a new profile
- Create: a page to create and post a new product
- Forum: a page to check the poll results and vote for new polls
- About: a simple landing page that describes the platform

5.5. IMPLEMENTATION

HOME

As it is natural to think, the most important page is *Home*. When the page is loaded for the first time, the function *isWalletConnected* checks if the visitor is using Metamask. Later, when a valid Ethereum account is recognized, the application asks *SwarmAd* if the address is registered. Obviously, the web interface will be loaded according to the response. Indeed, if the user is registered, he will have a navigation bar to check the latest polls, upload a new product or have a look at his reputation score. Otherwise, a button redirecting to the registration form will be loaded. In Code 5.19, we can see how these functionalities are implemented. In particular, *useEffect* is a hook that is executed at the first loading and whenever the Ethereum address in Metamask changes. In this way, if a user switches to another account, the page rendering will change as well.

```
1 useEffect(() => {
2     isWalletConnected();
3     isRegistered();
4 }, [ethAccount]);
5
6 async function isWalletConnected(){
7     #Check if browser is running Metamask
8     const {ethereum} = window;
9     if (!ethereum) return;
10    const accounts = await ethereum.request({method: '
11    eth_requestAccounts'});
12    #Set Ethereum Address as state variable
13    setEthAccount(accounts[0]);
14 }
15 async function isRegistered(){
16     if(!ethAccount) return;
17     const result = await isEthAccountRegistered(ethAccount);
18     console.log(result);
19     setEthAccountRegistered(result);
20 }
```

Code 5.19: "Setting up the Home"

The function *isEthAccountRegistered* is written in a separate file called *SwarmAd.js*. It interacts directly with smart contracts. It uses the *Contract* interface provided by ethers.js to create an abstraction of the smart contract. Every time the front-end needs to interact with SwarmAd, it has to pass through this piece

of Javascript. For example, in Code 5.20 we can see that it calls the *isRegistered* function in *SwarmAd.sol* and it returns the boolean value.

```

1 export const isEthAccountRegistered = async(ethAccount)=>{
2   const result = await swarmAdContract.isRegistered(ethAccount);
3   return result;
4 }

```

Code 5.20: "Querying SwarmAd"

SHOWCASE

The rest of the Home is fulfilled with a React component I have called *Showcase*. It is a set of cards where each card can show a product or an enterprise. In my idea, the Home is divided into five different showcases:

- Best Users: it shows the three users with the highest amount of Reputation Points
- Highlight Spots: a set of purchasable spots using Community Token
- Voted by the community: shows the products with the most super-likes.
- Latest: it shows the latest added products. It is a way to give a boost on newcomers' visibility.

In comparison with Martini's design, I think that this way to organize content motivates users to be active on the platform and to post quality content. Indeed, a random shuffle of content could lead to the tragedy of commons because being a helpful user is not rewarding. On top of that, since it is an advertising platform, it becomes more attractive if it shows the best it can offer.

REGISTER

The registration process is interesting because it involves interactions with the Swarm network and the blockchain at the same time.

It is a form where the user has to insert his enterprise details such as name, email and the company logo. The logo is uploaded on Swarm by *SwarmClient.js*. It creates an instance of a Bee client using the private gateway URL. Then, it uploads the image on Swarm and it returns the related hash reference. We can see the procedure in Code 5.21.

because it involves all five smart contracts.

The sequence is the following:

1. Alice creates a new enterprise. Since it is the first one, it is directly added to the platform.
2. Bob creates his enterprise profile but it is inserted on the waiting list.
3. Governor creates a poll
4. Alice votes for admitting Bob
5. After the deadline, Alice closes the poll and she gains reputation points from the Rewarder
6. Carl asks for entering the platform. A new poll is created and registered users vote for admitting Carl.
7. Again, we move forward to the poll deadline and this time is Bob who closes the poll. Carl's enterprise is moved to *eStructs* and Bob and Alice are rewarded.
8. Alice redeems her Community Tokens.

Consider a reward schema where each action gives 10 SWRPs and a daily interest rate equal to 1. Alice is expected to have 30 SWRPs because she voted twice and closed one poll. As a consequence, when redeeming the SWCTs she earned a total of 140 SWCTs because she gained 20 SWRPs when the first poll have been closed and then she staked that amount for seven days, i.e. when she gained 10 SWRPs more after voting for Carl's acceptance poll. Obviously, the last SWRPs she received have not produced any interest because she decided to redeem her coins on the same day the new SWRPs have been given.

This scenario has been tested using the framework provided by Truffle. The file *GovernorTest.js* precisely simulates every action described above, as if a real user were performing it. On top of that, after every smart contract call, the test script checks if the desired result has been obtained. A Javascript library written by OpenZeppelin comes in help by providing a useful tool to move forward to the poll deadline when it is needed. In the folder *truffle/test* there is another test script called *SwarmAdTest.js*. It is for testing the basic functionalities such as creating a product and changing its name or its image.



Conclusions and Future Works

This work describes a fully decentralized platform to create an authenticated, moderated and shared showcase where SMEs can advertise their products. The blockchain provides a safe way to manage user authentication and data, while a distributed storage system helps in optimizing downloading and uploading costs. Starting from a previous implementation, I have tackled three main issues: data persistence, naming the point of access, and content organization. First, I have changed the storage layer from IPFS to Swarm in order to exploit its built-in incentive system. In particular, its storage incentives are helpful to ensure data persistence over a long period. Second, the naming issue has been solved thanks to the combination of ENS and Swarm Feed. In particular, the latter provides an immutable hash to reference mutable content while ENS translates the immutable reference into a human-readable domain. As the last issue is concerned, I have implemented a tokenomics built on a reputation score to identify the users who are contributing the most and reward them accordingly. In detail, it is a two-token system where the first one is for signaling the users' reputation while the second one can be traded to buy rewards. In this way, I have solved the problem we encounter when the reputation score is tradable. About the platform itself, I have created a moderation mechanism to empower users in community decisions such as accepting a newcomer, reporting a user, or an inappropriate item. This feature is handled by a smart contract in charge of creating a poll whenever it is needed and performing the required actions after the poll is closed and the result is examined. It is special the choice to move away from coin-based governance, as in most of DAOs, in favor of a proof-of-

membership mechanism where each registered user can cast a single vote for each poll.

Nevertheless, creating a production-ready application could require a better membership control mechanism. Indeed, the registration process could be improved by implementing a decentralized identifier to verify digital identities. In this way, each user could provide tangible proofs of his enterprise. The implementation of such a system could extend the platform reach on a global scale because users who do not know each other can trust the newcomer's digital identity when they have to vote about acceptance. About this topic, it is also intriguing to implement a *soulbound* badge to identify the users of a certain community. Such a non-transferable badge could signal the user's membership across other platforms and hence it would embrace the idea of a fully decentralized web. Going even further, the step from an advertising platform to a decentralized marketplace seems a natural extension of my work. Nevertheless, it increases the platform complexity and introduces new issues. In this regard, transaction costs could have to be taken into serious consideration because they could easily drain enterprises' gains and mine the platform sustainability. As a result, it could be crucial to explore several layer-2 solutions which could down-size transaction costs.

Another useful addition could be a distributed rating system to allow users to review and rate an enterprise after a purchase. Of course, this feature should be strongly related to a proof-of-personhood mechanism to avoid review bombing or rating-boosting via fake accounts. In this regard, the idea proposed in [73] could be integrated with the reputation score mechanism. As the last thing, the reward system could be better defined. For example, it would be interesting to mint NFTs related to specific spots on the homepage to allow users to buy visibility. On top of that, it would be precious to explore new rewards to have a more solid and catching incentive schema.

References

- [1] Stanislav Vojí and Jan Kuera. “Towards Re-Decentralized Future of the Web: Privacy, Security and Technology Development”. In: *Acta Informatica Pragensia* 10 (Jan. 2022), pp. 349–369. DOI: 10.18267/j.aip.169.
- [2] Qin Wang et al. “Exploring Web3 From the View of Blockchain”. In: (June 2022).
- [3] Davide Martini. “A distributed CMS for small enterprises aggregation”. MA thesis. University of Padova, 2019.
- [4] [Online]. *IPFS Docs*. <https://docs.ipfs.tech/>.
- [5] [Online]. *Swarm*. <https://www.ethswarm.org/>.
- [6] [Online]. *ENS Documentation*. <https://docs.ens.domains/>.
- [7] Nicolas Six, Nicolas Herbaut, and Camille Salinesi. “Blockchain software patterns for the design of decentralized applications: A systematic literature review”. In: *Blockchain: Research and Applications* 3.2 (2022), p. 100061. ISSN: 2096-7209. DOI: <https://doi.org/10.1016/j.bcra.2022.100061>. URL: <https://www.sciencedirect.com/science/article/pii/S209672092200001X>.
- [8] *Introducing HiD - an Offline-first Decentralized CMS*. [Online] https://hide.ac/articles/m71_Ft2li.
- [9] Jarkko Kuusijärvi et al. *HELIOS: Final system architecture and API specification*. Deliverable.
- [10] Vanessa Clemente et al. *HELIOS: Define Rewarding Strategies*. Deliverable.
- [11] Bokang Jia, Chenhao Xu, and Mateusz Mach. *OPUS: Decentralized music distribution using InterPlanetary File Systems (IPFS) on the blockchain*. Whitepaper.

REFERENCES

- [12] Roneil Rumburg, Sid Sethi, and Hareesh Nagaraj. *Audius: A Decentralized Protocol for Audio Content*. Whitepaper.
- [13] Erik Daniel and Florian Tschorsch. "IPFS and Friends: A Qualitative Comparison of Next Generation Peer-to-Peer Data Networks". In: (2021). DOI: 10.48550/ARXIV.2102.12737. URL: <https://arxiv.org/abs/2102.12737>.
- [14] Barbara Guidi, Andrea Michienzi, and Laura Ricci. "Data Persistence in Decentralized Social Applications: The IPFS approach". In: *2021 IEEE 18th Annual Consumer Communications Networking Conference (CCNC)*. 2021, pp. 1–4. DOI: 10.1109/CCNC49032.2021.9369473.
- [15] [Online]. *Filecoin*. <https://docs.filecoin.io/>.
- [16] [Online]. *Powergate Docs*. <https://github.com/textileio/powergate/>.
- [17] [Online]. *Swarm Scan*. <https://swarmscan.resenje.org/>.
- [18] Zooko Wilcox-O'Hearn. *Names: Distributed, Secure, Human-Readable: Choose Two*. <http://zooko.com/distnames.html>.
- [19] [Online]. *IPNS | IPFS Docs*. <https://docs.ipfs.tech/concepts/ipns/#interplanetary-name-system-ipns>.
- [20] [Online]. *DNSLink | IPFS Docs*. <https://docs.ipfs.tech/concepts/dnslink/#dnslink>.
- [21] [Online]. *DNSLink Standard*. <https://dnslink.dev/>.
- [22] [Online]. *Handshake Developer Documentation*. <https://hsd-dev.org/>.
- [23] *Squaring the Triangle: Secure, Decentralized, Human-Readable Names*. [Online] <http://www.aaronsw.com/weblog/squarezooko>.
- [24] Grant Potter. "Defending Internet Freedom through Decentralization: Back to the Future?" In: 2018.
- [25] Andrea Passarella Barbara Guidi Marco Conti and Laura Ricci. "Managing social contents in Decentralized Online Social Networks: A survey". In: *Online Social Networks and Media* 7 (2018). DOI: <https://doi.org/10.1016/j.osnem.2018.07.001>.
- [26] Monika di Angelo and Gernot Salzer. "Tokens, Types, and Standards: Identification and Utilization in Ethereum". In: *2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. 2020, pp. 1–10. DOI: 10.1109/DAPPS49028.2020.00001.

- [27] Steemit. *Steem: An incentivized, blockchain-based, public content platform*. White Paper.
- [28] Moon Soo Kim and Jee Yong Chung. "Sustainable Growth and Token Economy Design: The Case of Steemit". In: *Sustainability* (2019).
- [29] Usman W. Chohan. "The Concept and Criticisms of Steemit". In: *Economics of Networks eJournal* (2018).
- [30] Simon de la Rouviere. *Curation Markets Whitepaper*. White Paper.
- [31] Zhengdong Li Haimei Xu Yan Cheng and Chunyan You. "Content Sharing Network based on IPFS and Blockchain". In: *IOP Conference Series: Materials Science and Engineering* 1043.052014 (2021). DOI: <https://doi.org/10.1088/1757-899X/1043/5/052014>.
- [32] [Online]. *Etherna*. <https://etherna.io/>.
- [33] *D.tube: Turning the tables in social media industry*. White Paper.
- [34] [Online]. *Bitcoin Wiki*. <https://en.bitcoin.it/wiki/Script>.
- [35] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. White Paper.
- [36] Nick Szabo. *Smart Contracts: Building Blocks for Digital Markets*. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.
- [37] William Metcalfe. "Ethereum, Smart Contracts, DApps". In: Springer, 2020. Chap. Chapter 5, pp. 77–93. URL: https://EconPapers.repec.org/RePEc:spr:ec1chp:978-981-15-3376-1_5.
- [38] Fabian Vogelsteller and Vitalik Buterin. *EIP-20: Token Standard*. Ethereum Improvement Proposals.
- [39] Zibin Zheng et al. "An overview on smart contracts: Challenges, advances and platforms". In: *Future Generation Computer Systems* 105 (2020), pp. 475–491. ISSN: 0167-739X. DOI: <https://doi.org/10.1016/j.future.2019.12.019>. URL: <https://www.sciencedirect.com/science/article/pii/S0167739X19316280>.
- [40] John R. Douceur. "The Sybil Attack". In: *Peer-to-Peer Systems*. Ed. by Peter Druschel, Frans Kaashoek, and Antony Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 251–260.

REFERENCES

- [41] [Online]. *The Merge*. <https://ethereum.org/it/upgrades/merge/>.
- [42] Vitalik Buterin. *Why Proof of Stake*. [Online] <https://vitalik.ca/general/2020/11/06/pos2020.html>.
- [43] Vitalik Buterin et al. "Combining GHOST and Casper". In: (2020). DOI: 10.48550/ARXIV.2003.03052. URL: <https://arxiv.org/abs/2003.03052>.
- [44] Vitalik Buterin and Virgil Griffith. "Casper the Friendly Finality Gadget". In: (2017). DOI: 10.48550/ARXIV.1710.09437. URL: <https://arxiv.org/abs/1710.09437>.
- [45] Vitalik Buterin. *Proof of Stake: How I Learned to Love Weak Subjectivity*. <https://blog.ethereum.org/2014/11/25/proof-stake-learned-love-weak-subjectivity>.
- [46] Ben Edgington. *Upgrading Ethereum : A technical handbook on Ethereum's move to proof of stake and beyond*. <https://eth2book.info/bellatrix/>.
- [47] Vitalik Buterin. *Annotated Ethereum 2.0 Spec*. <https://github.com/ethereum/annotated-spec/blob/master/phase0/beacon-chain.md>.
- [48] [Online]. *Altair – The Beacon Chain*. <https://github.com/ethereum/consensus-specs/blob/dev/specs/altair/beacon-chain.md>.
- [49] Vitalik Buterin. *On Settlement Finality*. [Online] <https://blog.ethereum.org/2016/05/09/on-settlement-finality1>.
- [50] Nick Johnson. *EIP-137: Ethereum Domain Name Service*. Ethereum Improvement Proposals.
- [51] Swarm Foundation. *The Book of Swarm*. White Paper.
- [52] [Online]. *Swarm Bee Documentation*. <https://docs.ethswarm.org/docs/>.
- [53] Swarm Foundation. *Swarm: storage and communication infrastructure for a self-sovereign digital society*. White Paper.
- [54] Robert Axelrod. "Effective Choice in the Prisoner's Dilemma". In: *Journal of Conflict Resolution* 24 (1980), pp. 25–3.
- [55] Trón Viktor et al. *Swap, Swear and Swindle: incentive system for swarm*. Orange Paper.
- [56] Yos Riady. *Best Practices for Smart Contract Development*. [Online] <https://yos.io/2019/11/10/smart-contract-development-best-practices/>.
- [57] [Online]. *Metamask Docs*. <https://docs.metamask.io/guide/>.

- [58] Xiwei Xu et al. "A Decision Model for Choosing Patterns in Blockchain-Based Applications". In: *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. 2021, pp. 47–57. doi: 10.1109/ICSA51549.2021.00013.
- [59] [Online]. *Ethersphere: gateway-proxy*. <https://github.com/ethersphere/gateway-proxy>.
- [60] [Online]. *Swarm Extension*. <https://github.com/ethersphere/swarm-extension>.
- [61] [Online]. *Execution API Specification | JSON-RPC*. <https://github.com/ethereum/execution-apis>.
- [62] [Online]. *Bee-js*. <https://github.com/ethersphere/bee-js>.
- [63] Robbie Morrison, Natasha C. Mazey, and Stephen C. Wingreen. "The dao controversy: The case for a new species of corporate governance?" In: *Frontiers in Blockchain* 3 (2020). doi: 10.3389/fbloc.2020.00025.
- [64] [Online]. *Ethersphere: Bee Factory*. <https://github.com/ethersphere/bee-factory>.
- [65] [Online]. *OpenZeppelin Docs*. <https://docs.openzeppelin.com/contracts/4.x/erc20>.
- [66] Gavin Wood. *EIP-161: State trie clearing*. Ethereum Improvement Proposals.
- [67] [Online]. *Solidity Docs*. <https://docs.soliditylang.org/en/latest/index.html>.
- [68] Vitalik Buterin. *Moving beyond coin voting governance*. <https://vitalik.ca/general/2021/08/16/voting3.html>.
- [69] Vitalik Buterin. *Soulbound*. <https://vitalik.ca/general/2022/01/26/soulbound.html>.
- [70] Lodovica Marchesi et al. "Design Patterns for Gas Optimization in Ethereum". In: *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. 2020, pp. 9–15. doi: 10.1109/IWBOSE50093.2020.9050163.
- [71] [Online]. *AION: A system for Scheduling transactions with arbitrary bytecode on the Ethereum Network*. <https://github.com/ETH-Pantheon/Aion>.
- [72] [Online]. *Ethereum Alarm Clock: Schedule transactions for the future*. <https://github.com/ethereum-alarm-clock/ethereum-alarm-clock>.

REFERENCES

- [73] Andrea Lisi et al. "Rewarding reviews with tokens: An Ethereum-based approach". In: *Future Generation Computer Systems* 120 (2021), pp. 36–54. ISSN: 0167-739X. DOI: <https://doi.org/10.1016/j.future.2021.02.003>. URL: <https://www.sciencedirect.com/science/article/pii/S0167739X21000480>.

Acknowledgments

It has been a difficult path to get to this final page. I have put many hours into studying, researching and writing to produce this. And every second was worth it. I am extremely glad about what I have achieved and I am ready to face what the future holds.

Of course, no one achieves anything alone. I am very thankful for having many people around me, ready to help with every problem or difficulty I have faced. In particular, I have to thank:

- Elda, *moje zlato* : for loving and caring during each day of this journey
- Mamma Giovanna e Papà Sergio, for being a wise guide in the toughest days
- My siblings: Francesco, Anna e Mattia
- My childhood friends, i.e. *TTesse*: Ale, Caretto, CuccyMatty, David, Guerry, SebaMag
- My friends from UniTS: Billo, Davide, Enrico, Leonardo, Luca, Matteo, Pippi, Simone, Riccardo, Xhacu
- My friends from UniPD, Andrea ed Elia
- Prof. Migliardi for guiding me in a wise and precise way
- DEI and UniPD
- Everybody who has spent some time reading this work



Smart Contracts

A.1 MIGRATIONS.SOL

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.4.22 <0.9.0;
3
4 contract Migrations {
5     address public owner;
6     uint public last_completed_migration;
7
8     modifier restricted() {
9         if (msg.sender == owner) _;
10    }
11
12    constructor(){
13        owner = msg.sender;
14    }
15
16    function setCompleted(uint completed) public restricted {
17        last_completed_migration = completed;
18    }
19
20    function upgrade(address new_address) public restricted {
21        Migrations upgraded = Migrations(new_address);
22        upgraded.setCompleted(last_completed_migration);
23    }
```

A.2. SWARMAD.SOL

24 }

Code A.1: Migrations.sol

A.2 SWARMAD.SOL

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.4.22 <0.9.0;
3
4 import "../SwarmAdGovernor.sol";
5 import "../node_modules/@openzeppelin/contracts/utils/Address.sol";
6 import "../node_modules/@openzeppelin/contracts/access/AccessControl.
    sol";
7
8 contract SwarmAd is AccessControl {
9
10     /// @notice List of Ethereum account of registered users
11     address[] public eList;
12
13     /// @notice It contains a list of Ethereum Account who asked for
14     membership and its votation is pending
15     address[] public waitingList;
16
17     /// @notice It contains PID from all products
18     bytes32[] public productList;
19
20     /// @notice Mapping between Ethereum address and Enterprise object
21     mapping(address=>Enterprise) public eStructs;
22
23     /// @notice Mapping between Product PID and Product object
24     mapping(bytes32 => Product) public productStructs; //mapping
25     between product and pid
26
27     /// @notice Mapping between Ethereum address and Enterprise object
28     while its votation is pending
29     mapping(address=>Enterprise) public waitingListStructs;
30
31     /// @notice Mapping an Ethereum address to its liked product, save
32     the last time he call the function
33     mapping(address=>bytes32) private superlike;
34     mapping(address=>uint256) private superlikeTimelock;
35
36     /// @notice Access Control
```

```

33 address private governor;
34 address private rewarder;
35 bytes32 public constant GOVERNOR_ROLE = keccak256("GOVERNOR_ROLE");
36 bytes32 public constant ENTERPRISE_ROLE = keccak256("
    ENTERPRISE_ROLE");
37
38 constructor() {
39     governor = address(uint160(uint256(keccak256(abi.encodePacked(
40         bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x03))))));
41     rewarder = address(uint160(uint256(keccak256(abi.encodePacked(
42         bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x04))))));
43     _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
44     _setupRole(GOVERNOR_ROLE, governor);
45 }
46
47 struct Enterprise{
48     string eName;
49     string eMail;
50     address eAddress;
51     string profileImageHash; /*CID provided by Swarm when uploaded */
52     uint indexEList; //position in eList
53     uint indexWaitingList;
54     bytes32[] enterprisePidList; //contains all product identifiers
55 }
56
57 struct Product{
58     bytes32 pid;
59     address enterprise;
60     string productName;
61     string productImageHash; //CID provided by Swarm when uploaded
62     string productDescription;
63     uint productPriceInWei;
64     uint indexProductStructs;
65     uint indexEnterprisePidList;
66 }
67
68 event newEnterpriseInWaitingList(address owner);
69 event moveFromWaitingList(address owner);
70
71 /// @notice Check if an address is registered in the platform,
72     must have for performing access control
73 /// @param a Enterprise Ethereum Address to check
74 /// @return boolean

```

A.2. SWARMAD.SOL

```
72  function isRegistered(address a) public view returns(bool){
73      if(eList.length==0) return false;
74      return (eList[eStructs[a].indexEList]==a);
75  }
76
77  /// @notice Check if an address is registered in the platform,
78  /// must have for performing access control
79  /// @param a Enterprise Ethereum Address to check
80  /// @return boolean
81  function isInWaitingList(address a) public view returns(bool){
82      if(waitingList.length==0) return false;
83      return (waitingList[waitingListStructs[a].indexWaitingList]==a);
84  }
85
86  /// @notice Create a new enterprise
87  /// @param name Enterprise name
88  /// @param mail Enterprise email
89  /// @param imgHash Image reference to retrieve the file from Swarm
90  function createEnterprise(string memory name, string memory mail,
91  string memory imgHash) public{
92      require(!isRegistered(msg.sender));
93
94      if(eList.length<1){
95          eList.push(msg.sender);
96          Enterprise storage e = eStructs[msg.sender];
97          e.eName = name; e.eMail = mail; e.profileImageHash = imgHash; e
98          .indexEList = eList.length-1; e.indexWaitingList = 0;
99          _grantRole(ENTERPRISE_ROLE, msg.sender);
100     }
101     else{
102         waitingList.push(msg.sender);
103         Enterprise storage e = waitingListStructs[msg.sender];
104         e.eName = name; e.eMail = mail; e.profileImageHash = imgHash; e
105         .indexEList = 0; e.indexWaitingList = waitingList.length -1;
106         emit newEnterpriseInWaitingList(msg.sender);
107         bytes[] memory transferCalldata = new bytes[](1);
108         transferCalldata[0] = abi.encodePacked(bytes4(keccak256('
109         moveEnterprise(address)')), // function signature
110         abi.encode(msg.sender) //function
111         arguments
112         );
113         SwarmAdGovernor(governor).createPoll(address(this),
114         transferCalldata, "Accept new user");
```



```

108     }
109 }
110
111 /// @notice Remove a registered enterprise
112 function removeEnterpriseFromSwarmAd() public onlyRole(
113     ENTERPRISE_ROLE){
114     bytes32 [] memory pidToDelete = eStructs[msg.sender].
115     enterprisePidList;
116     for(uint i=0; i<pidToDelete.length; i++){
117         removeProductByPid(pidToDelete[i]);
118     }
119
120     //pop from eList
121     for(uint i = eStructs[msg.sender].indexEList; i < eList.length-1;
122     i++){
123         eStructs[msg.sender].indexEList--; //update pointer
124         eList[i]=eList[i+1];
125     }
126     eList.pop();
127     renounceRole(ENTERPRISE_ROLE, msg.sender);
128     delete eStructs[msg.sender];
129 }
130
131 /// @notice Remove a registered enterprise
132 function governorRemoveEnterprise(address a) public onlyRole(
133     GOVERNOR_ROLE){
134     bytes32 [] memory pidToDelete = eStructs[a].enterprisePidList;
135     for(uint i=0; i<pidToDelete.length; i++){
136         governorRemoveProductByPid(pidToDelete[i]);
137     }
138
139     //pop from eList
140     for(uint i = eStructs[a].indexEList; i < eList.length-1; i++){
141         eStructs[a].indexEList--; //update pointer
142         eList[i]=eList[i+1];
143     }
144     eList.pop();
145     revokeRole(ENTERPRISE_ROLE, a);
146     delete eStructs[a];
147 }
148
149 /// @notice Remove enterprise from waiting list
150 function removeEnterpriseFromWaitingList(address a) public onlyRole

```

A.2. SWARMAD.SOL

```
(GOVERNOR_ROLE) {
147   for(uint i = waitingListStructs[a].indexWaitingList; i <
waitingList.length-1; i++){
148       waitingListStructs[a].indexWaitingList--; //update pointer
149       waitingList[i]=waitingList[i+1];
150   }
151   waitingList.pop();
152   delete waitingListStructs[msg.sender];
153 }
154
155 /// @notice Move from waiting list to SwarmAd list
156 function moveEnterprise(address a) public onlyRole(GOVERNOR_ROLE){
157     eList.push(a);
158     _grantRole(ENTERPRISE_ROLE, a);
159     Enterprise storage e = eStructs[a];
160     e.eName = waitingListStructs[a].eName;
161     e.eMail = waitingListStructs[a].eMail;
162     e.profileImageHash = waitingListStructs[a].profileImageHash;
163     e.indexEList = eList.length-1;
164     e.indexWaitingList = 0;
165     removeEnterpriseFromWaitingList(a);
166     emit moveFromWaitingList(a);
167 }
168
169 /// @notice Retrieve an enterprise
170 /// @param a Enterprise address
171 /// @return eName enterprise name
172 /// @return eMail enterprise mail
173 /// @return imgHash enterprise reference in Swarm
174 //get an Enterprise by address
175 function getEnterprise(address a) public view returns(string memory
eName, string memory eMail, string memory imgHash){
176     return(eStructs[a].eName, eStructs[a].eMail, eStructs[a].
profileImageHash);
177 }
178
179 /// @notice Retrieve an enterprise product list
180 /// @param a Enterprise address
181 /// @return enterprisePidList
182 function getProductListFromEnterprise(address a)public view returns
(bytes32 [] memory enterprisePidList) {
183     return (eStructs[a].enterprisePidList);
184 }
```

```

185
186 // @notice Update an enterprise image
187 // @param newImageHash new reference to a file saved in Swarm
188 function updateEProfilePicture(string memory newImageHash) public
    onlyRole(ENTERPRISE_ROLE){
189     eStructs[msg.sender].profileImageHash = newImageHash;
190 }
191
192 // @notice Update an enterprise mail
193 // @param newMail new mail to change
194 function updateEMail(string memory newMail) public onlyRole(
    ENTERPRISE_ROLE){
195     eStructs[msg.sender].eMail = newMail;
196 }
197
198 // @notice Update an enterprise name
199 // @param newName name to change
200 function updateEName(string memory newName) public onlyRole(
    ENTERPRISE_ROLE){
201     eStructs[msg.sender].eName = newName;
202 }
203
204 // @notice Create a new product
205 // @param name product name
206 // @param img Swarm reference to an image
207 // @param description literal description of an item
208 // @param price item price expressed in wei
209 function createNewProduct(
210     string memory name,
211     string memory img,
212     string memory description,
213     uint price) public onlyRole(ENTERPRISE_ROLE){
214     bytes32 pid = keccak256(abi.encodePacked(eStructs[msg.sender].
    eAddress, name, block.timestamp));
215     Product memory p = Product(
216         pid, msg.sender, name, img, description
    , price,
217         productList.length,
218         eStructs[msg.sender].enterprisePidList.
    length
219         );
220     productStructs[pid]=p;
221     productList.push(pid);

```

A.2. SWARMAD.SOL

```
222     eStructs[msg.sender].enterprisePidList.push(pid);
223 }
224
225 /// @notice Get a product py pid
226 function getProductByPid(bytes32 pid) public view returns(
227     string memory productName,
228     string memory productImageHash,
229     string memory productDescription,
230     uint productPriceInWei,
231     address enterprise
232 ){
233     Product memory p = productStructs[pid];
234     return(p.productName, p.productImageHash, p.productDescription, p
        .productPriceInWei, p.enterprise);
235 }
236
237 /// @notice Get a product enterpirse py pid
238 function getNameByPid(bytes32 pid) public view returns(
239     string memory productName){
240     Product memory p = productStructs[pid];
241     return p.productName;
242 }
243
244 /// @notice Get a product enterpirse py pid
245 function getEnterpriseByPid(bytes32 pid) public view returns
246     (address enterprise){
247     Product memory p = productStructs[pid];
248     return p.enterprise;
249 }
250
251 /// @notice Remove product from global list of product and from
252     enterprise list
253 function removeProductByPid(bytes32 pid) public onlyRole(
254     ENTERPRISE_ROLE){
255     require(msg.sender == productStructs[pid].enterprise, "Caller is
256         not owner");
257
258     //delete from enterprise
259     for(uint i = productStructs[pid].indexEnterprisePidList; i <
260         eStructs[msg.sender].enterprisePidList.length-1; i++){
261         eStructs[msg.sender].enterprisePidList[i] = eStructs[msg.sender
262             ].enterprisePidList[i+1];
263     }
264 }
```

```

257     eStructs[msg.sender].enterprisePidList.pop();
258
259     //delete from global
260     for(uint i = productStructs[pid].indexProductStructs; i <
productList.length-1; i++){
261         productList[i] = productList[i+1];
262     }
263     productList.pop();
264
265     //delete from global mapping
266     delete productStructs[pid];
267 }
268
269 /// @notice Remove product from global list of product and from
enterprise list
270 function governorRemoveProductByPid(bytes32 pid) public onlyRole(
GOVERNOR_ROLE){
271     address e = productStructs[pid].enterprise;
272     //delete from enterprise
273     for(uint i = productStructs[pid].indexEnterprisePidList; i <
eStructs[e].enterprisePidList.length-1; i++){
274         eStructs[e].enterprisePidList[i] = eStructs[e].
enterprisePidList[i+1];
275     }
276     eStructs[e].enterprisePidList.pop();
277
278     //delete from global
279     for(uint i = productStructs[pid].indexProductStructs; i <
productList.length-1; i++){
280         productList[i] = productList[i+1];
281     }
282     productList.pop();
283
284     //delete from global mapping
285     delete productStructs[pid];
286 }
287
288 /// @notice Update product name
289 /// @param pid to select the product
290 /// @param newName name to change the old one
291 function updateProductNameByPid(bytes32 pid, string memory newName)
public onlyRole(ENTERPRISE_ROLE){
292     require(msg.sender == productStructs[pid].enterprise, "Caller is

```

A.2. SWARMAD.SOL

```
    not owner");
293     productStructs[pid].productName = newName;
294 }
295
296 /// @notice Update product price
297 /// @param pid to select the product
298 /// @param newPrice price to change the old one
299 function updateProductPriceByPid(bytes32 pid, uint newPrice) public
    onlyRole(ENTERPRISE_ROLE){
300     require(msg.sender == productStructs[pid].enterprise, "Caller is
    not owner");
301     productStructs[pid].productPriceInWei = newPrice;
302 }
303
304 /// @notice Update product description
305 /// @param pid to select the product
306 /// @param newDescription description to change the old one
307 function updateProductDescriptionByPid(bytes32 pid, string memory
    newDescription) public onlyRole(ENTERPRISE_ROLE){
308     require(msg.sender == productStructs[pid].enterprise, "Caller is
    not owner");
309     productStructs[pid].productDescription = newDescription;
310 }
311
312 /// @notice Update product image hash
313 /// @param pid to select the product
314 /// @param newImageHash Swarm reference to change the old one
315 function updateProductImageHashByPid(bytes32 pid, string memory
    newImageHash) public onlyRole(ENTERPRISE_ROLE){
316     require(msg.sender == productStructs[pid].enterprise, "Caller is
    not owner");
317     productStructs[pid].productImageHash = newImageHash;
318 }
319
320 /// @notice Retrieve Ethereum addresses of all registered
    enterprises
321 function getEList() public view returns(address [] memory){
322     return eList;
323 }
324
325 /// @notice Retrieve Ethereum addresses of enterprises in waiting
    list
326 function getWaitingList() public view returns(address [] memory){
```

```

327     return waitingList;
328 }
329
330 /// @notice Retrieve all products
331 function getProductList() public view returns(bytes32 [] memory){
332     return productList;
333 }
334
335 /// @notice Superlike a product
336 /// @param pid to select the product
337 function assignSuperlike(bytes32 pid) public onlyRole(
    ENTERPRISE_ROLE){
338     require(productStructs[pid].enterprise != address(0)); //check
    item exists
339     require(productStructs[pid].enterprise != msg.sender); //check
    owner is not caller
340     uint256 differenceTimestamp = SafeMath.sub(block.timestamp,
    superlikeTimelock[msg.sender]);
341     uint256 differenceInDays = SafeMath.div(SafeMath.div(SafeMath.
    div(differenceTimestamp, 60), 60), 24);
342     require(differenceInDays > 0, "time difference is not enough");
343     superlike[msg.sender] = pid;
344     superlikeTimelock[msg.sender] = block.timestamp;
345     SwarmAdRewarder(rewarder).addRPsToAccount(msg.sender, 5);
346 }
347 }

```

Code A.2: SwarmAd.sol

A.3 SWARMADGOVERNOR

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.4;
3
4 import "./SwarmAd.sol";
5 import "./SwarmAdRewarder.sol";
6 import "../node_modules/@openzeppelin/contracts/access/Ownable.sol";
7 import "../node_modules/@openzeppelin/contracts/utils/Address.sol";
8 import "../node_modules/@openzeppelin/contracts/utils/math/SafeMath.
    sol";
9 import "../node_modules/@openzeppelin/contracts/utils/math/Math.sol";
10
11 contract SwarmAdGovernor is AccessControl{

```

A.3. SWARMADGOVERNOR

```
12
13     /// @notice Other contracts' addresses
14     address swarmad;
15     address rewarder;
16
17     /// @notice Translates pollId into Poll object
18     mapping (uint=>Poll) pollMap;
19
20     /// @notice List with all pollId since mapping is not iterable
21     uint[] polls;
22
23     /// @notice voting time (currently 1 day)
24     uint256 votingDelay = 86400;
25
26     /// @notice quorum in percentage to reach to make a poll valid
27     uint256 quorum = 51;
28
29     /// @notice slashing reputation score of who doesn't vote
30     uint256 punishment = 10;
31
32     /// @notice poll state
33     uint64 constant ACTIVE = 1;
34     uint64 constant CLOSED = 2;
35     uint64 constant EXECUTED = 3;
36
37     struct Poll{
38         uint64 state;
39         uint64 votesFor;
40         uint64 votesAgainst;
41         uint64 votesNull;
42         uint256 pollId;
43         uint256 maxVoters;
44         uint256 quorum;
45         uint256 votingStart;
46         uint256 votingEnd;
47         address swarmad;
48         address proposer;
49         address [] voters;
50         address [] ableToVote;
51         bytes[] calldatas;
52         string description;
53     }
54
```



```

55     event pollCreated(uint256 pollId, string description);
56     event pollExecuted(uint256 pollId);
57     event assignCloseReward(uint256 amount, uint256 delayInHours,
uint256 delayFactor);
58
59     constructor() {
60         swarmad = address(uint160(uint256(keccak256(abi.encodePacked(
bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x02))))));
61         rewarder = address(uint160(uint256(keccak256(abi.encodePacked
(bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x04))))));
62         _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
63     }
64
65     /// @notice Compute the number of voters is needed to reach the
quorum
66     /// @param maxVoters users with voting right for the current poll
67     /// @return res i.e. quorum expressed in users
68     function computeQuorum(uint256 maxVoters) internal view returns(
uint256){
69         uint256 prod = SafeMath.mul(maxVoters, quorum);
70         uint256 res = SafeMath.div(prod, 100);
71         return res;
72     }
73
74     /// @notice Create a poll
75     /// @param proposer user or contract who is creating the poll
76     /// @param calldatas function to execute if the majority voted
for
77     /// @param description string with the poll reason
78     function createPoll(address proposer, bytes[] memory calldatas,
string memory description) public {
79         uint256 pollId = uint(keccak256(abi.encodePacked(block.
timestamp, proposer)));
80         //External call to SwarmAd to get current list of users
81         address[] memory maxVoters = SwarmAd(swarmad).getEList();
82         require(maxVoters.length > 0);
83
84         Poll storage p = pollMap[pollId];
85         p.pollId = pollId;
86         p.proposer = proposer;
87         p.ableToVote = maxVoters;
88         p.maxVoters = maxVoters.length;
89         p.quorum = computeQuorum(maxVoters.length);

```

A.3. SWARMADGOVERNOR

```
90     p.votingStart = block.timestamp;
91     p.votingEnd = block.timestamp + votingDelay;
92     p.state = ACTIVE;
93     p.calldatas = calldatas;
94     p.description = description;
95     polls.push(pollId);
96
97     emit pollCreated(pollId, description);
98 }
99
100 // @notice Retrieve a poll from global list
101 // @param pollId uint unique identifier
102 function getPoll(uint pollId) public view returns(
103     address proposer, uint64 votesFor, uint64 votesAgainst,
104     uint64 votesNull,
105     uint256 numberVoters, uint256 votingEnd
106 ) {
107     Poll memory p = pollMap[pollId];
108     return (p.proposer, p.votesFor, p.votesAgainst, p.votesNull,
109     p.voters.length, p.votingEnd);
110 }
111
112 // @notice Execute the function embedded in the poll
113 // @param pollId uint unique identifier
114 function executePoll(uint pollId) internal {
115     require(hasVotingRight(pollId, msg.sender));
116     (bool success, bytes memory returndata) = swarmad.call(
117     pollMap[pollId].calldatas[0]);
118     Address.verifyCallResult(success, returndata, "Governor:
119     reverted without message");
120 }
121
122 // @notice Close the poll if the deadline is met
123 // @param pollId uint unique identifier
124 function closePoll(uint pollId) public {
125     require(!isVotingActive(pollId), "Governor: Poll is still
126     active");
127     require(!isPollExecuted(pollId), "Governor: Poll has already
128     been executed");
129     require(hasVotingRight(pollId, msg.sender), "Governor: User
130     has no voting right to close the poll");
131     if(!isQuorumReached(pollId)) {
132         pollMap[pollId].state=CLOSED;
```

```

126     }
127     if(isMajorityVotingFor(pollId) ){
128         executePoll(pollId);
129     }
130     else if(pollMap[pollId].swarmad != pollMap[pollId].proposer){
131         SwarmAdRewarder(rewarder).burnRPsToAccount(pollMap[pollId
].proposer, punishment);
132     }
133     giveRewards(pollId);
134     pollMap[pollId].state = EXECUTED;
135     emit pollExecuted(pollId);
136 }
137
138 /// @notice Check if user a voted in poll
139 /// @param poll uint unique identifier
140 /// @param a user
141 function hasNotVoted(uint poll, address a) internal view returns(
bool){
142     address[] memory voters = pollMap[poll].voters;
143     for(uint i=0; i<voters.length; i++){
144         if(voters[i]==a) return false;
145     }
146     return true;
147 }
148
149 /// @notice Check if a poll has been already executed
150 /// @param pollId uint unique identifier
151 /// @return bool
152 function isPollExecuted(uint pollId) public view returns(bool){
153     if(pollMap[pollId].state == EXECUTED) return true;
154     return false;
155 }
156
157 /// @notice Check if a poll deadline is already met
158 /// @param poll uint unique identifier
159 /// @return bool
160 function isVotingActive(uint poll) public view returns(bool) {
161     if(block.timestamp > pollMap[poll].votingEnd) return false;
162     if(block.timestamp < pollMap[poll].votingStart) return false;
163     return true;
164 }
165
166 /// @notice Check if quorum is reached

```

A.3. SWARMADGOVERNOR

```
167     /// @param pollId uint unique identifier
168     /// @return bool
169     function isQuorumReached(uint pollId) internal view returns(bool)
170     {
171         if(pollMap[pollId].voters.length > pollMap[pollId].quorum)
172         return true;
173         return false;
174     }
175
176     /// @notice Check if majority voted for, hence if poll has to be
177     /// executed
178     /// @param pollId uint unique identifier
179     /// @return bool
180     function isMajorityVotingFor(uint pollId) internal view returns(
181     bool){
182         Poll memory p = pollMap[pollId];
183         if(p.votesFor > p.votesAgainst) return true;
184         return false;
185     }
186
187     /// @notice Check users who voted and who did not, calls the
188     /// rewarder to mint or burn reputation points
189     /// @param pollId uint unique identifier
190     function giveRewards(uint pollId) internal {
191         require(hasVotingRight(pollId, msg.sender), "Governor: no
192         right to vote");
193         SwarmAdRewarder Rewarder = SwarmAdRewarder(rewarder);
194         uint256 amount = 10; //TODO
195         address[] memory voters = pollMap[pollId].voters;
196         for(uint i=0; i<voters.length; i++){
197             if(hasNotVoted(pollId, voters[i])) Rewarder.
198             burnRPsToAccount(voters[i], amount);
199             else Rewarder.addRPsToAccount(voters[i], amount);
200         }
201         uint256 delay = SafeMath.sub(block.timestamp, pollMap[pollId
202         ].votingEnd);
203         uint256 delayInHours = SafeMath.div(SafeMath.div(delay, 60),
204         60);
205         uint256 delayFactor = Math.max(1, delayInHours);
206         uint256 closeReward = SafeMath.div(amount, delayFactor);
207         emit assignCloseReward(closeReward, delayInHours, delayFactor
208         );
209         Rewarder.addRPsToAccount(msg.sender, closeReward);
```

```

200     }
201
202     /// @notice Check if a user can vote for that poll
203     /// @param poll uint unique identifier
204     /// @param a user
205     /// @return bool
206     function hasVotingRight(uint poll, address a) internal view
returns(bool){
207         address[] memory ableToVote = pollMap[poll].ableToVote;
208         for(uint i = 0; i<ableToVote.length; i++){
209             if(ableToVote[i]==a) return true;
210         }
211         return false;
212     }
213
214     /// @notice Cast vote from a user
215     /// @param poll uint unique identifier
216     /// @param vote 1 = for, 2 = against, 3 = abstained
217     function castVote(uint poll, uint64 vote) public{
218         require(hasVotingRight(poll, msg.sender));
219         require(isVotingActive(poll));
220         require(hasNotVoted(poll, msg.sender));
221
222         if(vote==1) pollMap[poll].votesFor++;
223         if(vote==2) pollMap[poll].votesAgainst++;
224         if(vote==3) pollMap[poll].votesNull++;
225
226         pollMap[poll].voters.push(msg.sender);
227     }
228
229     /// @notice Get all polls created
230     function getAllPoll() public view returns(uint[] memory pollList)
    {
231         return polls;
232     }
233
234
235
236 }

```

Code A.3: SwarmAdGovernor.sol

A.4 SWARMADREWARDER

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.4;
3
4 import "./SwarmAd.sol";
5 import "./SwarmAdReputationPoints.sol";
6 import "./SwarmAdCommunityToken.sol";
7 import "../node_modules/@openzeppelin/contracts/access/AccessControl.
  sol";
8 import "../node_modules/@openzeppelin/contracts/utils/Address.sol";
9 import "../node_modules/@openzeppelin/contracts/utils/math/SafeMath.
  sol";
10
11 contract SwarmAdRewarder is AccessControl{
12
13     uint256 dailyInterestRate = 1;
14
15     /// @notice stores how much interest in form of Community Token
16     users are accumulating
17     mapping(address=>uint256) public SWCTVault;
18     /// @notice keeps track of last time the interests have been
19     computed for each wallet
20     mapping(address=>uint256) private
21     addressToLastInterestComputation;
22
23     /// @notice access control mechanism
24     address governor;
25     address swarmad;
26     address coins;
27     address reputation;
28
29     bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
30     bytes32 public constant GOVERNOR_ROLE = keccak256("GOVERNOR_ROLE"
31 );
32
33     event newValueInVault(uint256 amount);
34     event ValueInVault(address owner, uint256 amount);
35
36     constructor() {
37         swarmad = address(uint160(uint256(keccak256(abi.encodePacked(
38 bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x02) )))));

```

```

34     governor = address(uint160(uint256(keccak256(abi.encodePacked
(bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x03))))));
35     reputation = address(uint160(uint256(keccak256(abi.
encodePacked(bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x05)
))));
36     coins = address(uint160(uint256(keccak256(abi.encodePacked(
bytes1(0xd6), bytes1(0x94), msg.sender, bytes1(0x06) ))));
37     _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
38     _setupRole(MINTER_ROLE, swarmad);
39     _setupRole(MINTER_ROLE, governor);
40     _setupRole(GOVERNOR_ROLE, governor);
41 }
42
43     /// @notice Computes interests
44     /// @param balance reputation points a user has in his wallet
45     /// @param a user
46     /// @return uint256 how much interests a user has accumulateed
from ladt computation
47     function computeCompoundInterest(uint256 balance, address a)
internal view returns(uint256){
48         uint256 duration = SafeMath.sub(block.timestamp,
addressToLastInterestComputation[a]);
49         uint256 durationInDays = SafeMath.div( SafeMath.div (SafeMath
.div(duration, 60), 60), 24);
50         uint256 compoundInterest = SafeMath.mul( SafeMath.mul(
durationInDays, balance), dailyInterestRate);
51         return compoundInterest;
52     }
53
54     /// @notice Add reputation points to a user's wallet
55     /// @param a user ethereum address
56     /// @param amount how many RP
57     function addRPsToAccount(address a, uint256 amount) public
onlyRole(MINTER_ROLE) {
58         SwarmAdReputationPoints SWRP = SwarmAdReputationPoints(
reputation);
59         uint256 balance = SWRP.balanceOf(a);
60         uint256 interest = computeCompoundInterest(balance, a);
61         if(interest>0){
62             uint256 newVaultValue = SafeMath.add(SWCTVault[a],
interest);
63             SWCTVault[a] = newVaultValue;
64         }

```

A.5. SWARMADREPUTATIONPOINTS

```
65     SWRP.mint(a, amount);
66     addressToLastInterestComputation[a] = block.timestamp;
67 }
68
69 /// @notice Burn reputation points from a user's wallet
70 /// @param a user ethereum address
71 /// @param amount how many RP
72 function burnRPsToAccount(address a, uint256 amount) public
onlyRole(GOVERNOR_ROLE){
73     SwarmAdReputationPoints SWRP = SwarmAdReputationPoints(
reputation);
74     uint256 balance = SWRP.balanceOf(a);
75     uint256 interest = computeCompoundInterest(balance, a);
76     if(interest>0){
77         uint256 newVaultValue = SafeMath.add(SWCTVault[a],
interest);
78         SWCTVault[a] = newVaultValue;
79     }
80     SWRP.burn(a, amount);
81     addressToLastInterestComputation[a] = block.timestamp;
82 }
83
84 /// @notice converts interests into Community Token
85 function redeem() public{
86     uint256 amount = SWCTVault[msg.sender];
87     require(amount>0);
88     SWCTVault[msg.sender] = 0;
89     SwarmAdCommunityToken SWCT = SwarmAdCommunityToken(coins);
90     SWCT.mint(msg.sender, amount);
91 }
92 }
```

Code A.4: SwarmAdRewarder.sol

A.5 SWARMADREPUTATIONPOINTS

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.4;
3
4 import "../node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol";
5 import "../node_modules/@openzeppelin/contracts/token/ERC20/
extensions/ERC20Burnable.sol";
```



```

6 import "../node_modules/@openzeppelin/contracts/access/AccessControl.
  sol";
7
8 contract SwarmAdReputationPoints is ERC20, ERC20Burnable,
  AccessControl {
9   bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
10
11   constructor(address rewarder) ERC20("SwarmAdReputationPoints", "
  SWRP") {
12     _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
13     _grantRole(MINTER_ROLE, rewarder);
14   }
15
16   function _beforeTokenTransfer(address from, address to, uint256
  amount) internal override(ERC20){
17     require(from == address(0) || to == address(0), "Error:
  Reputation Points are not transferable");
18     super._beforeTokenTransfer(from, to, amount);
19   }
20
21   function mint(address to, uint256 amount) public onlyRole(
  MINTER_ROLE) {
22     _mint(to, amount);
23   }
24
25   function burn(address to, uint256 amount) public onlyRole(
  MINTER_ROLE){
26     _burn(to, amount);
27   }
28
29
30 }

```

Code A.5: SwarmAdReputationPoints.sol



Testing scripts

B.1 SWARMADTEST.JS

```
1 const SwarmAd = artifacts.require("SwarmAd");
2 const Governor = artifacts.require("SwarmAdGovernor");
3 const Rewarder = artifacts.require("SwarmAdRewarder");
4 const SWCT = artifacts.require("SwarmAdCommunityToken");
5 const SWRP = artifacts.require("SwarmAdReputationPoints");
6 const { time } = require('../node_modules/@openzeppelin/test-helpers'
  );
7
8 //accounts = available addresses in the network
9 contract("SwarmAd", (accounts) => {
10
11   let SwarmAdContract, GovernorContract, SWCTContract,
12   RewarderContract, SWRPContract;
13   let user=accounts[0]
14   let bob=accounts[1];
15   let pid;
16
17   before(async ()=>{
18     SwarmAdContract = await SwarmAd.deployed();
19     GovernorContract = await Governor.deployed();
20     RewarderContract = await Rewarder.deployed();
21
22     SWRPContract = await SWRP.new(RewarderContract.address);
23     SWCTContract = await SWCT.new(RewarderContract.address);
```

B.1. SWARMADTEST.JS

```
23     });
24
25     it("create enterprise", async()=>{
26         await SwarmAdContract.createEnterprise("juve", "juve@gmail.
com", "0x1234", {from: user});
27         let result = await SwarmAdContract.isRegistered(user);
28         assert.equal(result, true, "Enterprises not registered");
29     });
30
31     it("remove enterprise", async()=>{
32         await SwarmAdContract.removeEnterpriseFromSwarmAd({from: user
});
33         let result = await SwarmAdContract.isRegistered(user);
34         assert.equal(result, false, "Enterprises is still registered"
);
35     });
36
37     it("check if remove enterprise removes also products", async()=>{
38         await SwarmAdContract.createEnterprise("samp", "samp@gmail.
com", "0x9001", {from: user});
39         await SwarmAdContract.removeEnterpriseFromSwarmAd({from: user
});
40         let globalPidList= await SwarmAdContract.getProductList();
41         assert.equal(globalPidList.length, 0, "Products not removed")
;
42     });
43
44     it("retrieve enterprise", async()=>{
45         await SwarmAdContract.createEnterprise("samp", "samp@gmail.
com", "0x9001", {from: user});
46         let result = await SwarmAdContract.getEnterprise(user);
47         assert.equal(result[0], "samp", "Enterprise is not retrieved"
);
48     });
49
50     it("change enterprise name", async()=>{
51         await SwarmAdContract.updateEName("juventus", {from: user});
52         let result = await SwarmAdContract.getEnterprise(user, {from:
user});
53         assert.equal(result[0], "juventus", "Name is not changed");
54     });
55
56     it("create new product", async()=>{
```

```

57     await SwarmAdContract.createNewProduct("pogba", "pogba.jpg",
58     "football player", "10", {from: user});
59
60     it("retrieve product by pid", async()=>{
61         let pidList = await SwarmAdContract.
62         getProductListFromEnterprise(user);
63         pid = pidList[0];
64         let result = await SwarmAdContract.getProductByPid(pid);
65         assert.equal(result[0], "pogba", "Product name is not correct
66         ");
67     });
68
69     it("update name by pid", async()=>{
70         await SwarmAdContract.updateProductNameByPid(pid, "dimaria",{
71         from: user});
72         let result = await SwarmAdContract.getProductByPid(pid);
73         assert.equal(result[0], "dimaria", "item is not deleted");
74     });
75
76     it("use superlike", async()=>{
77         await SwarmAdContract.createEnterprise("bob", "bob@gmail.com"
78         , "0x5678", {from: bob});
79         let polls = await GovernorContract.getAllPoll();
80         await GovernorContract.castVote(polls[0], 1, {from:user});
81         let day = 60 * 60 * 24;
82         await time.increase(day*1);
83         await GovernorContract.closePoll(polls[0], {from: user});
84         let pidList = await SwarmAdContract.
85         getProductListFromEnterprise(user);
86         await SwarmAdContract.assignSuperlike(pidList[0], {from: bob
87         });
88     });
89 }

```

Code B.1: SwarmAdTest.js

B.2 GOVERNORTEST.JS

```

1  const SwarmAd = artifacts.require("SwarmAd");
2  const Governor = artifacts.require("SwarmAdGovernor");
3  const SWRP = artifacts.require("SwarmAdReputationPoints");

```

B.2. GOVERNORTEST.JS

```
4 const SWCT = artifacts.require("SwarmAdCommunityToken");
5 const Rewarder = artifacts.require("SwarmAdRewarder");
6
7 const {
8     BN,          // Big Number support
9     constants,  // Common constants, like the zero address and
10    largest integers
11    expectEvent, // Assertions for emitted events
12    expectRevert,
13    time, // Assertions for transactions that should fail
14 } = require('../node_modules/@openzeppelin/test-helpers');
15
16 contract("Testing Governance", (accounts) => {
17
18     let SwarmAdContract, GovernorContract, SWCTContract,
19     RewarderContract, SWRPContract;
20
21     let alice = accounts[1];
22     let bob = accounts[2];
23     let carl = accounts[3];
24
25     before(async ()=>{
26         SwarmAdContract = await SwarmAd.deployed();
27         GovernorContract = await Governor.deployed();
28         RewarderContract = await Rewarder.deployed();
29
30         SWRPContract = await SWRP.new(RewarderContract.address);
31         SWCTContract = await SWCT.new(RewarderContract.address);
32     });
33
34     it("1) Create a new enterprise", async()=>{
35         await SwarmAdContract.createEnterprise("alice", "alice@gmail.
36         com", "0x1234", {from: alice});
37         await SwarmAdContract.createEnterprise("bob", "bob@gmail.com"
38         , "0x5678", {from: bob});
39     });
40
41     it("2) Check enterprise in waiting list", async()=>{
42         let result = await SwarmAdContract.isInWaitingList(bob);
43         assert.equal(result, true, "Enterprise is not in waiting list
44         ");
45     });
46
47     it("3) Vote a proposal", async()=>{
```

```

42     let polls = await GovernorContract.getAllPoll();
43     await GovernorContract.castVote(polls[0], 1, {from:alice});
44 });
45
46 it("4) Execute a proposal", async()=>{
47     let result = await SwarmAdContract.isRegistered(bob);
48     assert.equal(result, false, "already registered");
49     let day = 60 * 60 * 24;
50     await time.increase(day*1);
51     let polls = await GovernorContract.getAllPoll();
52     await GovernorContract.closePoll(polls[0], {from: alice});
53     result = await SwarmAdContract.isRegistered(bob);
54     assert.equal(result, true, "user not added successfully");
55 });
56
57 it("5) Check rewards", async()=>{
58     let result = await SWRPContract.balanceOf(alice);
59     assert.equal(result, 20, "Rewards not received");
60 });
61
62 it("6) New voting", async()=>{
63     //newcomer
64     await SwarmAdContract.createEnterprise("carl", "carl@gmail.
65 com", "0x9101", {from: carl});
66     //voting
67     let polls = await GovernorContract.getAllPoll();
68     await GovernorContract.castVote(polls[1], 1, {from:alice});
69     await GovernorContract.castVote(polls[1], 1, {from:bob});
70     //fast forward to vote end
71     let day = 60 * 60 * 24;
72     await time.increase(day+1);
73     await GovernorContract.closePoll(polls[1], {from: bob});
74     //alice has new RPs, she should be able to redeem her first
75 bunch of SWCT
76     await RewarderContract.redeem({from: alice});
77     ct = await SWCTContract.balanceOf(alice);
78     assert.equal(ct, 20, "SWCT not properly converted");
79     let rp = await SWRPContract.balanceOf(alice);
80     assert.equal(rp, 30, "RP are wrong");
81     let vault = await RewarderContract.getVaultValue(alice);
82     assert.equal(vault, 0, "vault not empty after redeem");
83 });

```

B.2. GOVERNORTEST.JS

```
82 });
```

Code B.2: GovernorTest.js



Front-end

C.1 HOME

```
1 import { ethers } from "ethers";
2 import React, {useState, useEffect} from "react";
3 import { isEthAccountRegistered } from "../apis/SwarmAd";
4 import Footer from "../components/Footer";
5 import Header from '../components/Header';
6 import Showcase from "../components/Showcase";
7
8 //context
9 export const UserContext = React.createContext(null);
10
11 function Home(){
12
13     const [ethAccount, setEthAccount] = useState();
14     const [ethAccountRegistered, setEthAccountRegistered]=useState(
15         false);
16
17     useEffect(() => {
18         isWalletConnected();
19         isRegistered();
20     }, [ethAccount]);
21
22     async function isWalletConnected(){
23         // Check if browser is running Metamask
24         const {ethereum} = window;
```

C.2. REGISTER

```
24     //If not, fallback to Ganache
25     if (!ethereum) ethereum = ethers.getDefaultProvider('http
26     ://127.0.0.1:7545');
27     const accounts = await ethereum.request({method: '
28     eth_requestAccounts'});
29     setEthAccount(accounts[0]);
30   }
31
32   async function isRegistered(){
33     if(!ethAccount) return;
34     const result = await isEthAccountRegistered(ethAccount);
35     console.log(result);
36     setEthAccountRegistered(result);
37   }
38
39   return(
40     <UserContext.Provider value={[ethAccount,
41     ethAccountRegistered]}>
42       <Header />
43       <Showcase/>
44       <Footer />
45     </UserContext.Provider>
46   );
47 }
48
49 export default Home;
```

Code C.1: Home.jsx

C.2 REGISTER

```
1 import { keccak256 } from "ethers/lib/utils";
2 import React, {useState, useEffect} from "react";
3 import { Link, Navigate } from "react-router-dom";
4 import { createEnterprise } from "../apis/SwarmAd";
5 import { uploadImage } from "../apis/SwarmClient";
6
7
8 function Register(){
9
10     //state vars
11     const [ethAccount, setEthAccount] = useState();
12     const [eName, setEName]=useState('');
```

```

13  const [eMail, setEmail]=useState('');
14  const [imageHash, setImageHash]=useState('');
15  const [image, setImage] = useState();
16
17  //error checking
18  const [submit, setSubmit] = useState(false);
19  const [error, setError] = useState(false);
20
21  //get wallet from metamask
22  useEffect(() => {
23      isWalletConnected();
24  }, [ethAccount]);
25
26  async function isWalletConnected() {
27      // Check if browser is running Metamask
28      const {ethereum} = window;
29      if (!ethereum) return;
30      const accounts = await ethereum.request({method: '
eth_requestAccounts'});
31      setEthAccount(accounts[0]);
32  }
33
34  //handle name
35  const handleEName = (e) => {
36      setEName(e.target.value);
37      setSubmit(false);
38  }
39
40  //handle email
41  const handleEMail = (e) => {
42      setEMail(e.target.value);
43      setSubmit(false)
44  }
45
46  //Contract call
47  async function callCreateEnterprise() {
48
49      let txn = await createEnterprise(eName, eMail, imageHash);
50      if(!txn.hash) return;
51      setSubmit(true);
52      setError(false);
53
54      //redirect

```

C.2. REGISTER

```
55     <Navigate to="/home"/>
56   }
57
58   async function handleSubmit(e){
59     e.preventDefault();
60     //validation
61     if(eName==='' || eMail==='' || imageHash===') {setError(true
); return;}
62     callCreateEnterprise();
63   }
64
65   //image handler
66   const onImageChange = (e) =>{
67     const f = e.target && e.target.files && e.target.files[0];
68     setImage(f);
69   }
70
71   async function uploadImageOnSwarm(e){
72     e.preventDefault();
73     const result = await uploadImage(image);
74     setImageHash(result);
75
76   }
77
78   const successMsg = () =>{
79     return(
80       <div className="success">{eName} has been successfully
registered to SwarmAd</div>
81     );
82   }
83
84   const errorMsg = () =>{
85     <div className="error">Error: check all fields before
submitting</div>
86   }
87
88   return(
89     <div className="registerForm">
90       <div><h2>Register Form</h2></div>
91
92       <div className="messages">
93         {errorMsg()}
94         {successMsg()}
```

```

95         </div>
96
97         <form>
98             <label className="label">Enterprise Name</label>
99             <input onChange={handleEName} className="input" value
100 ={eName} type="text"/>
101
102             <label className="label">E-mail Address</label>
103             <input onChange={handleEMail} className="input" value
104 ={eMail} type="text"/>
105
106             <button onClick={handleSubmit} className="btn">
107 Register</button>
108
109         </form>
110
111         <form>
112             <label className="label">Logo</label>
113             <input type="file" name="file" onChange={
114 onImageChange}/>
115
116             <button onClick={uploadImageOnSwarm} className="btn">
117 Upload</button>
118
119         </form>
120
121         <Link to="/">Back to home</Link>
122     </div>
123 );
124 }
125
126 export default Register;

```

Code C.2: Register.jsx

C.3 CREATEITEM

```

1 import { keccak256 } from "ethers/lib/utils";
2 import React, {useState, useEffect} from "react";
3 import { Link, Navigate } from "react-router-dom";
4 import { createEnterprise } from "../apis/SwarmAd";
5 import { uploadImage } from "../apis/SwarmClient";
6 import {createNewItem} from '../apis/SwarmAd'
7

```

C.3. CREATEITEM

```
8
9 function CreateItem(){
10
11     //state vars
12     const [ethAccount, setEthAccount] = useState();
13     const[itemName, setItemName]=useState('');
14     const[itemDescription, setItemDescription]=useState('');
15     const[itemPrice, setItemPrice]=useState('');
16     const [image, setImage] = useState();
17     const[imageHash, setImageHash]=useState('');
18
19     //error checking
20     const[submit, setSubmit] = useState(false);
21     const[error, setError] = useState(false);
22
23     //get wallet from metamask
24     useEffect((() => {
25         isWalletConnected();
26     }, [ethAccount]));
27
28     async function isWalletConnected(){
29         // Check if browser is running Metamask
30         const {ethereum} = window;
31         if (!ethereum) return;
32         const accounts = await ethereum.request({method: '
33 eth_requestAccounts'});
34         setEthAccount(accounts[0]);
35     }
36
37     //handle name
38     const handleItemName = (e) => {
39         setItemName(e.target.value);
40         setSubmit(false);
41     }
42
43     //handle email
44     const handleItemDescription = (e) =>{
45         setItemDescription(e.target.value);
46         setSubmit(false)
47     }
48
49     //handle price
50     const handleItemPrice = (e) =>{
```

```

50     setItemPrice(e.target.value);
51     setSubmit(false)
52   }
53
54   //Contract call
55   async function callCreateNewItem(){
56     let txn = await createNewItem(itemName, imageHash,
itemDescription, itemPrice)
57     if(!txn.hash) return;
58     setSubmit(true);
59     setError(false);
60   }
61
62   async function handleSubmit(e){
63     e.preventDefault();
64     //validation
65     if(itemName==='' || itemDescription==='' || imageHash==='' ||
itemPrice===') {setError(true); return;}
66     callCreateNewItem();
67   }
68
69   //image handler
70   const onImageChange = (e) =>{
71     const f = e.target && e.target.files && e.target.files[0];
72     setImage(f);
73   }
74
75   async function uploadImageOnSwarm(e){
76     e.preventDefault();
77     const result = await uploadImage(image);
78     setImageHash(result);
79
80   }
81
82   const successMsg = () =>{
83     return(
84       <div className="success">{itemName} has been successfully
added to SwarmAd</div>
85     );
86   }
87
88   const errorMsg = () =>{
89     <div className="error">Error: check all fields before

```

C.3. CREATEITEM

```
submitting</div>
90   }
91
92   return(
93     <div className="registerForm">
94       <div><h2>Create new product</h2></div>
95
96       <div className="messages">
97         {errorMsg()}
98         {successMsg()}
99       </div>
100
101       <form>
102         <label className="label">Product name</label>
103         <input onChange={handleItemName} className="input "
value={itemName} type="text"/>
104
105         <label className="label">Description</label>
106         <input onChange={handleItemDescription} className="
input" value={itemDescription} type="text"/>
107
108         <label className="label">Price</label>
109         <input onChange={handleItemPrice} className="input "
value={itemPrice} type="number"/>
110
111         <button onClick={handleSubmit} className="btn">Add
new product</button>
112
113       </form>
114
115       <form>
116         <label className="label">Logo</label>
117         <input type="file" name="file" onChange={
onImageChange}/>
118
119         <button onClick={uploadImageOnSwarm} className="btn">
Upload</button>
120       </form>
121
122       <Link to="/">Back to home</Link>
123     </div>
124   );
125 }
```



```
126
127 export default CreateItem;
```

Code C.3: CreateItem.jsx

C.4 SHOWCASE

```
1 import {useEffect, useState} from 'react';
2 import ItemShowcase from './ItemShowcase';
3 import EnterpriseCard from './EnterpriseCard'
4
5 import { getEnterpriseAddressList, getEnterpriseList,
  getItemListFromEnterprise } from '../apis/SwarmAd';
6 import EnterpriseShowcase from './EnterpriseShowcase';
7
8 function Showcase(){
9
10     const [enterprises, setEnterprises] = useState();
11     const [items, setItems] = useState();
12     const maxItemInShowcase = 5;
13
14     useEffect(()=>{
15         fetchEnterprise();
16     }, []);
17
18     async function fetchEnterprise(){
19         let list = await getEnterpriseList();
20         setEnterprises(list);
21     }
22
23     async function fetchItemsFromEnterprise(enterprise){
24         let list = await getItemListFromEnterprise(enterprise);
25         setItems(list);
26     }
27
28     return(
29         <div>
30         <EnterpriseShowcase enterprises={enterprises} maxNumber='4' />
31         </div>
32     );
33
34 }
35
```

```

36
37 export default Showcase;

```

Code C.4: Showcase.jsx

C.5 ENTERPRISESHOWCASE

```

1 import { useEffect, useState } from "react";
2 import { Box, Button, Container, Grid, Typography } from '@mui/
  material';
3 import EnterpriseCard from "../EnterpriseCard";
4
5 function EnterpriseShowcase({enterprises, maxNumber}){
6
7   const [index, setIndex] = useState(0);
8   const [enterprisesToShow, setEnterprisesToShow] = useState();
9
10  useEffect(()=>{
11    fillWithEnterprises();
12  },[index]);
13
14  function fillWithEnterprises(maxNumber){
15    let list = [];
16    for(let i=0; i<maxNumber; i++){
17      if((index + i + 1) > enterprises.length) index=0;
18      list.push(enterprises[index+i]);
19    }
20    setEnterprisesToShow(list);
21  }
22
23  return(
24    <div>
25      <h2>Enterprises</h2>
26      <Container>
27        <Grid container spacing={1}>
28          {enterprises?
29            enterprises.map((e)=>{
30              return (
31                <Grid item xs={3} key={e.eName} >
32                  <EnterpriseCard eName={e.eName} eMail
33                    ={e.eMail} image={e.image}/>
34                  </Grid>
35                );}) : null

```

```

35         }
36
37         </Grid>
38     </Container>
39 </div>
40 );
41
42 }
43 export default EnterpriseShowcase;

```

Code C.5: EnterpriseShowcase.jsx

C.6 SWARMAD.JS

```

1 import swarmAdJson from '../contracts/SwarmAd.json'
2 import {ethers} from 'ethers'
3
4 //web3
5 const {ethereum} = window;
6 const provider= new ethers.providers.Web3Provider(ethereum);
7 const signer = provider.getSigner();
8 //contract
9 const ABI = swarmAdJson.abi;
10 const ADDRESS = "0xBDff6A11043dc03838dA14D7AD77854B2E4B85D1";
11 export const swarmAdContract = new ethers.Contract(ADDRESS,ABI,signer
    );
12
13 export const getEnterprise = async(ethAddress)=>{
14     const txn = await swarmAdContract.getEnterprise(ethAddress);
15     return {eName: txn[0], eMail: txn[1], image: txn[2]}
16 }
17
18 export const getEnterpriseAddressList = async()=>{
19     const txn = await swarmAdContract.getEList();
20     return txn;
21 }
22
23 export const getEnterpriseList = async()=>{
24     const list = await swarmAdContract.getEList();
25     const enterprises = []
26     for(let i=0; i<list.length;i++){
27         const enterprise = await getEnterprise(list[i]);
28         enterprises.push(enterprise);

```

C.6. SWARMAD.JS

```
29     }
30     console.log(enterprises);
31     return enterprises;
32 }
33
34 export const getItem = async(pid)=>{
35     const item = await swarmAdContract.getProduct(pid);
36     console.log(item);
37 }
38
39 export const getItemListFromEnterprise = async(enterpriseEthAddress)
=>{
40     const list = await swarmAdContract.getProductList(
enterpriseEthAddress);
41     let items = [];
42     for(let i=0; i<list.length;i++){
43         const item = await getItem(list[i]);
44         items.push(item);
45     }
46     return items;
47 }
48
49 export const isEthAccountRegistered = async(ethAccount)=>{
50     const result = await swarmAdContract.isRegistered(ethAccount);
51     return result;
52 }
53
54 export const createEnterprise = async (eName, eMail, imageHash) =>{
55     const txn = await swarmAdContract.createEnterprise(eName, eMail,
imageHash);
56     return txn;
57 }
58
59 export const createNewItem = async(itemName, imageHash,
itemDescription, itemPrice)=>{
60     const txn = await swarmAdContract.createNewProduct(itemName,
imageHash, itemDescription, itemPrice);
61     return txn;
62 }
```

Code C.6: SwarmAd.js

C.7 SWARMCLIENT.JS

```

1 import { Bee, BeeDebug } from "@ethersphere/bee-js"
2 /* BEE FACTORY SETTINGS */
3 //const beeURL = 'http://localhost:11633' //main net value 'http://
  localhost:1633' or 'https://api.gateway.ethswarm.org/' for gateway
4 const beeURL = 'http://localhost:13000';
5 const bee = new Bee(beeURL);
6
7 export const downloadImage = (imageHash)=>{
8   return `${beeURL}/bzz/${imageHash}`
9 }
10
11 export const uploadImage = async (image) =>{
12
13   console.log('uploading');
14   //batch is handled directly by the gateway
15   const postageBatch = '
  0000000000000000000000000000000000000000000000000000000000000000';
16   const result = await bee.uploadFile(postageBatch, image);
17   console.log(result.reference);
18   return result.reference
19 }

```

Code C.7: SwarmClient.js