Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

The Design and Implementation of a Decentralized Smart Contract Descriptor Repository

Christian Schreiner

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Ghareeb Falazi, M.Sc.
Commenced:	January 20, 2022
Completed:	July 20, 2022

Abstract

One of Ethereum's most important innovations was the first implementation of smart contracts. Since its release in 2015, many blockchain technologies were developed, many of them also implementing smart contracts. Typically, smart contracts have an address that is not meaningful to humans, in addition to having different interfaces depending on the used technology. Thus, a requirement for a smart contract registry emerged which allows users to not only register uniform descriptions of their smart contracts but also find contract descriptions from other developers, they might be interested in. This thesis proposes such a registry that is not only functional but also decentralized and thus censorship resistant. To do this, we proposed two architectures that solve this problem, compared them and decided on one that we implemented in the end. First, both store so-called Smart Contract Descriptors (SCD), which are technology independent descriptions of smart contracts. The first architecture can be summed up as a client for an already existing decentralized content-sharing network that utilizes its built in functionality to discover SCDs and to upload them. The other approach was the one we implemented. It solves the problem by storing metadata about SCDs in a smart contract which we call the Registry Contract. This metadata points then to the location of the actual SCD. We call this off-chain location an External SCD Storage. In addition to storing SCD-metadata, the Registry Contract offers querying capabilities that are augmented by an off-chain service that we call the External Search Provider. We propose to integrate all of those pieces with a frontend that is hosted in a decentralized manner. Since it is expected that such a registry would store a significant number of contracts, we also wanted to get insights into the time it takes to query it. Thus, we also conducted a performance test with regard to the amount of stored and retrieved SCD-metadata. This test showed us that the overhead of using such a registry is relatively small. Consequently, we came to the conclusion that a censorship resistant registry can not only be designed and implemented but that is also feasible to use it due to the not too large overhead. Moreover, we created a Smart Contract Descriptor data set by crawling GitHub for Solidity smart contracts which we then transformed to SCDs. The data set consists of 127766 SCDs and is to our knowledge the first large-scale SCD data set in existence, and it can assist further research in the field.

Kurzfassung

Eine der wichtigsten Innovationen von Ethereum war die erste Implementierung von Smart Contracts. Seit der Veröffentlichung im Jahr 2015 wurden viele Blockchain-Technologien entwickelt, von denen viele auch Smart Contracts implementieren. In der Regel haben Smart Contracts eine Adresse, die für Menschen nicht aussagekräftig ist, und je nach verwendeter Technologie auch unterschiedliche Schnittstellen. So entstand der Bedarf an einer Smart Contract Registry, die es Nutzern ermöglicht, nicht nur einheitliche Beschreibungen ihrer Smart Contracts zu registrieren, sondern auch Contract Beschreibungen von anderen Entwicklern zu finden, die für sie von Interesse sein könnten. Diese Arbeit präsentiert eine solche Registry, die nicht nur funktional, sondern auch dezentralisiert und damit zensurresistent ist. Zu diesem Zweck haben wir zwei Architekturen erstellt, die dieses Problem lösen, sie verglichen und uns für eine entschieden, die wir schließlich implementiert haben. Zu aller erst sollte gesagt werden, dass beide sogenannte Smart Contract Descriptors (SCD) speichern. Das sind technologieunabhängige Beschreibungen von Smart Contracts. Die erste Architektur lässt sich als Client für ein bereits bestehendes dezentrales Content Sharing Netzwerk zusammenfassen, der die eingebauten Funktionen des Netzwerks nutzt, um SCDs zu entdecken und hochzuladen. Der andere Ansatz war der von uns implementierte. Er löst das Problem, indem er Metadaten über SCDs in einem Smart Contract speichert, den wir Registry Contract nennen. Diese Metadaten verweisen dann auf den Ort, an dem sich das eigentliche SCD befindet. Wir nennen diesen außerhalb der Blockchain liegenden Ort einen externen SCD Storage. Neben der Speicherung von SCD-Metadaten bietet der Registry-Contract eine Suchfunktionalität, die durch einen außerhalb der Blockchain liegenden Dienst erweitert wird und den wir External Search Provider nennen. Wir integrieren all diese Komponenten in einem Frontend, das dezentral gehostet wird. Da zu erwarten ist, dass eine solche Registry eine beträchtliche Anzahl von Contracts speichern wird, wollten wir auch einen Überblick über die Zeit erhalten, die benötigt wird, um SCD-Metadaten von der Registry abzufragen. Daher haben wir auch einen Performancetest in Bezug auf die Menge der gespeicherten und abgerufenen SCD-Metadaten durchgeführt. Dieser Test zeigte uns, dass der Overhead bei der Verwendung einer solchen Registry relativ gering ist. Folglich kamen wir zu dem Schluss, dass eine zensurresistente Registry nicht nur entworfen und implementiert werden kann, sondern aufgrund des nicht allzu großen Overheads auch praktikabel ist. Darüber hinaus haben wir einen Smart Contract Descriptor Datensatz erstellt, indem wir GitHub nach Solidity Smart Contracts durchforstet haben, die wir dann in SCDs umgewandelt haben. Der Datensatz besteht aus 127766 SCDs und ist unseres Wissens nach der erste große SCD Datensatz und er kann die weitere Forschung auf diesem Gebiet unterstützen.

Acknowledgements

I would like to express my deepest appreciation to my supervisor Ghareeb Falazi who has been a great mentor and thesis supervisor. He offered me invaluable advice and encouragement during my work on this thesis. Without his immense knowledge and support, this thesis would have never been accomplished. I am truly thankful for the opportunity to have worked with him.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ĐApp Decentralized Application. 21

API Application Programming Interface. 29

DAO Decentralized Autonomous Organization. 22

DHT Distributed hash table. 21

DISC Distributed Immutable Store of Chunks. 25

DNS Domain Name System. 29

EDI Electronic Data Interchange. 23

ENS Ethereum Name Service. 33

EVM Ethereum Virtual Machine. 23

HTTP Hypertext Transfer Protocol. 12

JSON JavaScript Object Notation. 13

NFT Non-fungible token. 22

POS Proof-of-Stake. 23

POW Proof-of-Work. 21

RPC remote procedure call. 41

SCD Smart Contract Descriptor. 9

SCDL Smart Contract Description Language. 11

SCIP Smart Contract Invocation Protocol. 11

SCL Smart Contract Locator. 11

SEC U.S. Securities and Exchange Commission. 46

SOAP Simple Object Access Protocol. 33

SWAP Swarm Accounting Protocol. 25

SWIFT Society for Worldwide Interbank Financial Telecommunication. 23

UBR UDDI Business Registry. 33

UDDI Universal Description, Discovery, and Integration. 33

UI User interface. 49

URL Uniform Resource Locator. 27

WSDL Web Services Description Language. 33

XML Extensible Markup Language. 33

YaLBRYf Yet another LBRY frontend. 11

1 Introduction

Imagine being the last living descendant of *Alexander the Great, King of Macedonia and conqueror of Achaemenid Persia* and you know it. As his name and tile suggest, he was a powerful landowner and conqueror, making you technically the heir of a large fortune that would give you the high-live you deserve, without lifting a finger ever again. Sadly, it would be pretty frustrating to live with that knowledge, since there is no existing legal document that lets you claim his empire. Additionally, according to the ancient Greek historian *Diodorus Siculus*, Alexander said on his deathbed that his empire should go to "tôi kratistôi", which when translated means "to the strongest". Thus, forgoing his, at that point unborn son and sparking war between his generals which broke Alexander's kingdom into pieces. Would it not have been better for you, if there had been a technology back then that enabled the man to create some sort of self-enforcing contract which managed his heritage in a smart and more peaceful way, so that you can actually try to lay claim to the fortune of the son of Zeus (even though no one today would accept your claim, even if such a technology existed back then)?

Luckily, today something like this exists. It is called a *smart contract*. They are not only an integral part of the blockchain domain, but are also "promises, specified in a digital form" [Sza96]. In other words, they are small programs that are executed on all nodes of a blockchain deterministically. Like all programs they need to be called and therefore need to be found and addressed somehow. Depending on the underlying technology, calling them works quite differently. However, most smart contracts are typically identified by some sort of address. Such addresses are often long strings that can only be remembered by the most capable of mental acrobats. Therefore, there is a need to make them publicly accessible. A widely applied solution to such discovery problems are central services, like npm[1] or ConanCenter[2]. Doing so provides a place to store some sort of smart contract description and make it possible to search though those descriptions while giving users a functional approach to discovering new ones. Such a registry has been proposed by Lamparelli [Lam20]. Their goal was not the creation of a registry per se, but to integrate smart contracts into conventional software systems and business process engines. For this purpose, they provided an exemplary architecture for how such a system might look like and introduced a data structure called Smart Contract Descriptor (SCD). The latter is a JSON document that describes smart contracts in a technology-agnostic way, while the former contained a centralized SCD registry.

Unfortunately, there would be no reason to trust the operators of such services in the real world, since they have ultimate control over the software they run and can therefore use their power to do serious harm instead of doing what they promised. They could for example remove entries or exclude certain users from using the service to benefit external groups. In addition to that, such an approach would contradict the spirit of blockchain technologies which is opposed to central authorities in general.

[1] https://www.npmjs.com
[2] https://conan.io

Hence, our goal is to create a censorship resistant SCD registry to complement Lamparelli et al.'s work by decentralizing control over it as much as possible while keeping it functional. The registry should also enable users to search through registered SCDs and therefore find new and possibly useful contracts.

We will now give an overview over this work. The first chapter is called *Background* and provides an overview over the necessary topics that are foundational to this work. That is followed by a short chapter about related work. After that, we talk about two architecture alternatives that have been developed as a part of this work. One alternative utilizes an already existing system as its backend and the other one combines multiple existing pieces of software to create a completely new system. We compare both of them and decide on which one we will implement and evaluate. Furthermore, we go into more detail about the system's architecture that we implemented in Chapter 5. Following that architecture description, we talk about the implementation of the system, i. e., how it works, which implementation decisions were made and which technologies are used. This is followed by an evaluation of the system in Chapter 7. Finally, Chapter 8 concludes our work and gives an overview about what can be done in the future.

2 Background

In this chapter, we go through the background topics, relevant for this work. We begin with what a blockchain actually is and then we are talking about Ethereum which is a popular blockchain platform. Following that, we talk about the concept of smart contracts in more depth than we did in the Introduction. After that we dive into the topic of Decentralized Applications, since what we propose in this work is going to be one. Then we are going to look into so-called Distributed hash tables, which serve as a foundation for decentralized storage systems. Lastly, we are presenting two such systems, one called Swarm and the other one called LBRY. The latter in that case is not only a storage system, but it also provides a naming system for stored files.

2.1 Blockchain

Foremost, we have to clarify what a blockchain is. In essence, a blockchain is a distributed database [ER18]. Not distributed in the sense that the data itself is distributed among multiple nodes, but that the transaction verification is distributed among all network participants. Thus, no one has complete control over which transaction are legitimate. Verification is a matter of performing a consensus algorithm, but this will be explained later.

Bitcoin was the first implementation of such a system and therefore created the first blockchain [Rav16]. Consequently, it seems only logical to use it as basis for explaining how all of this works in general. In Bitcoin each network participant has a wallet which consists of a public and a private key [Nak09]. The public key servers as the address, while the private key is used to sign transactions before sending them. Imagine now Alice wants to send Bob 100 BTC. Alice creates that transaction with all necessary information and signs it with her public key. In addition to that, she has to pay a transaction fee, but we will come to that later. She now has to broadcast that block to the network, where it will be picked up by so-called *miners*. Their task is to verify transactions and add them to the blockchain to make them official. They do that by collecting transactions in so-called *blocks*. After a miner gathered enough transactions, they begin to perform a Proof-of-Work (POW). This is done by solving a hard cryptographic puzzle. Bitcoins POW works roughly by guessing a random nonce which is added to the block's data until the hash of that data has a certain number of zeros at the beginning. The miner that succeeds with that before everyone else gets to add their block to the blockchain. A miner does this by broadcasting the block with its hash. Other miners verify if the block's hash is correct, store it and broadcast it to other miners. Consequently, the block is propagated to all miners in the network. Following this, the miners start the same process of creating a block again. This POW and verification process is Bitcoins consensus algorithm. Blocks are chronologically linked by adding the hash of the previous block to the data of the current block (i. e., the hash of the previous block influences the hash of the current one). This results in a chain

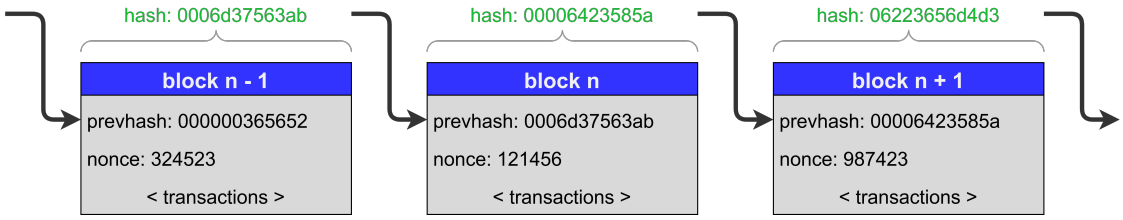

Figure 2.1: This figure shows how blocks are connected. Each block contains the hash of the previous block. This means that the hash of a block influences the hash of its successor block [But22; Nak09].

of blocks, or better called blockchain. Figure 2.1 visualizes this relationship. Bob receives the 100 BTC when Alice's transaction is added to the blockchain. The current state of a wallet can be determined by summing up all transactions that the wallet was involved in.

The POW is performed to prevent illegitimate transactions from entering the blockchain and thus becoming official. We assume that the majority of miners are not malicious and even it they were it is highly unlikely that they all work on the same goal. Solving the puzzle before everyone else is more likely if the attacker has a large amount of computing resources. But since a malicious actor (lets call them Mallory) is unlikely to have more resources than all the other legitimate miners, they will lose that race. Even if, by sheer luck, Mallory succeeds before the rest of the network, the success will not last for long, since the longest chain (and thus the one with the most work put into it) is to be trusted by all network participants. Users should therefore only trust a transaction if a few more blocks have already been mined. Mallory has to continue this lucky streak for quite a while, which is, as we previously determined, unlikely.

Getting back to the transaction fee. It seems pretty nonsensical for miners to burn through their hardware and their electricity bill for charity. To give them an incentive, they receive the transaction fees of all transactions in a block they added to the blockchain, in addition to a BTC reward that they get from the network itself. This reward from the network is the only way for new Bitcoins to be created. Furthermore, this reward decreases over time until it is practically non-existent. Thus the amount of BTC is predetermined to be 21 Million, which makes Bitcoin a deflationary currency. Hence, Bitcoin does not rely on people being nice but on their pursue of self-interest which is a way better motivator.

2.2 Ethereum

The usage of blockchain technologies has picked up steam and branched out into different domains apart from currency [But22]. Such domains include but are not limited to Decentralized Autonomous Organizations, Non-fungible tokens and smart contracts. Ethereum[1] facilitates all of that by providing a Turing-complete programming language that enable users to create and run programs on the blockchain. Therefore, enabling the creation of systems for all kinds of applications. Ethereum is fuelled by a cryptocurrency called *Ether*.

[1] https://ethereum.org

Like Bitcoin, Ethereum is permissionless and uses POW as its consensus algorithm. Furthermore, it also supports permissioned deployments. Every year there are plans to change that algorithm to Proof-of-Stake (POS) but they were never realized at the time of writing this [Eth22a].

2.3 Smart Contracts

Szabo described smart contracts as "a set of promises, specified in digital form, including protocols within which the parties perform on the other promises" in 1996 [Sza96], but people commonly associate them with Ethereum (see Section 2.2), since it was their first implementation [But22]. Szabo also defined four properties commonly found in conventional contracts that also apply to smart contracts. Firstly, *observability*, which implies that the involved parties are able to observe each other or to prove their performance. The second one is *verifiability* and it is about proving the performance or lack thereof to an arbitrator. The third one is called *privity*. It says that control and knowledge about the contract should only be distributed among parties as much as necessary for the performance of the contract. In other words, only designated arbitrators and contract parties are affected or effect the contract. Lastly, comes *enforceability* while at the same time minimizing the necessity for enforcement. Back then he called systems like the Society for Worldwide Interbank Financial Telecommunication (SWIFT)[2] or Electronic Data Interchange (EDI) the forerunners of smart contracts. He also acknowledged that they only implement commercial security models, without concern for the obligations and needs of the involved contract parties.

The concept of smart contracts has found its way into other blockchain technologies over the years. Bitcoin itself provides a small Turing-incomplete language with which scripting short smart contracts is possible. Ethereum is a step-up in that regard, because it provides a Turing-complete language and a runtime environment called the Ethereum Virtual Machine (EVM) to facilitate more advanced smart contracts and applications.

2.4 Đapps

Ethereum enables running arbitrary code on the blockchain via smart contracts and thus enables the creation of so called ĐApps. Such applications can theoretically serve the same functions as the common and centralized applications we use every day, which range from video streaming over cloud storage to online stock trading. In contrast to those traditional applications, ĐApps have a few special properties. According to Raval [Rav16], those are:

- It has to be **open source**. Otherwise, there is no reason for the users to believe that the application is really decentralized. That leads them to favour open source alternatives. In general this is pretty easy to achieve, but problems arise if the application is supposed to make money. Everyone can just fork a project and brand it as their own.
- There has to be an **internal currency**, which fuels the system. After all, transaction fees have to be paid and in most cases the developers need to be paid as well. Scarcity is the key to solve that problem. Therefore, the currency has to be limited by nature and it has to be

[2] https://www.swift.com

necessary to use the application. If more users use it, the demand for it goes up and so does it's value. Hence, using ĐApps can never be free of charge, but sign up rewards are a viable option for new users.

- If **consensus** is required, it has to be **decentralized**. In the old days, this was hard to achieve, but with the advent of blockchains, this became easier. As a result, it is necessary to use one for a ĐApp if consensus is required. Protocols like BitTorrent have no need for a consensus, therefore, they don't require a blockchain.
- The last property is that there is **no single point of failure**. So excessive force like a world wide EMP [3] would be necessary to shut it down. Performing such an action generally entails other risks and it is really expensive. Decentralization is a great tool to achieve that kind of resiliency. To illustrate this, look at the state of online piracy ☠ over the BitTorrent protocol. You can punish a lot of pirates but you will never get all of them.

The second property is not as important to us as the other three, since our prototype is not supposed to be deployed on a real blockchain like Ethereum or make money. Popular ĐApps are PancakeSwap[4], Splinterlands[5] and OpenSea[6].

2.5 Distributed hash tables

An important part of systems of any kind is the storage of data. Data is typically stored in a centralized location in huge data centers. However, ĐApps go for different, decentralized solutions like IPFS[7], Swarm[8] or BitTorrent. Most of those solutions have one thing in common: they use so called DHTs.

A DHT is a peer-to-peer method to store and retrieve data [WGR05]. In general, DHTs distribute the stored data across a number of nodes. With each node managing a specific range of data items. Retrieval is done by implementing a routing scheme which lets one locate the node that manages the data that is to be retrieved. To facilitate that, each node stores a partial view of the global network. Typically, this information should contain the node's nearest neighbors.

Data is identified via a key which is typically a hash of the data, but it can also be anything else. When a request for a specific data item is received by a node, it looks up if it stores that item and if it doesn't it routes the request to the next one. The choice of the next node is in general implementation specific. As an example: Assume keys are just a sequence of numbers (i. e., data1 gets key 1, data2 gets key 2, and so on) and each node is responsible for a range of those keys and nodes that manage neighboring ranges are also neighbors in the network, then it is possible to greedily route to the node that manages the range closest to the key of the data that is to be retrieved.

[3] Electromagnetic pulse
[4] https://pancakeswap.finance
[5] https://splinterlands.com
[6] https://opensea.io
[7] https://ipfs.io
[8] https://www.ethswarm.org

Until now, we assumed that the responsible node actually stores the data. Doing so is definitely a valid approach, but it leads to overhead regarding storage and network bandwidth at that node, because it would need to service all file requests directly, instead of delegating them to other participants. An alternative approach to this is that nodes only store a reference to a data storage that actually stores the data. Such storage are typically the original data uploader or in this case one could call them announcers. Obviously the announcer needs to stay operational as long as the data is supposed to be available.

2.6 Swarm

Swarm is an economically self-sustaining, peer-to-peer distributed data storage network [Swa21b] developed by the Swarm Foundation. In this section, we are going to talk about how Swarm works.

Swarm uses an underlying storage model called Distributed Immutable Store of Chunks (DISC) which consists of nodes that collaborate to store data in a way that maximizes the operator's profits [Swa21b]. Nodes in Swarm form a Kademlia [MM02] which is a DHT. Their Kademlia address is the Swarm address which is distinct from its network address. The degree of node proximity can be determined by counting the common prefix bits of this address. Kademlia requires that nodes form a *Kademlia connectivity*, which means that a node's neighborhood of proximity order d contains at least eight nodes. The node is also connected to eight more peers for each shallower proximity order $d-1, d-2, \ldots, 1, 0$ [Swa21b]. Figure 2.2 visualizes that relationship. Messages are sent or relaid to the neighbor that is nearest to the target address. Through the Kademlia connectivity each of those hops moves the message nearer to the target.

DISC's unit of storage is called a *chunk*. They have a maximum size is 4 kilobytes. Each chunk gets an address and this address determines at which node the chunk is stored. It is stored at the node that has the nearest Swarm address. Consequently, the distance between a node and a chunk is computed the same way as the distance between nodes themselves. Chunks are forwarded using the *push-sync* protocol until they arrive at the node they are supposed to be stored at. In addition to that they are also replicated to the nodes in the storage node's neighborhood to improve availability.

Coming back to the "self-sustaining" part. Foremost, a node posses two kinds of storage: A so-called *reserve* and a *cache*. Both of them are limited and have different strategies to clear. Those are part of the so called Swarm Accounting Protocol (SWAP) which is the mechanism that ensures the necessary collaboration and self sustainability. It works like this: The message exchange between nodes works on the basis of a service-for-service exchange. When the service limit of a node for another node is reached, they can either wait until an equilibrium was reached, or wait until the other node pays them by sending checks, which can later be cashed out for BZZ, which is the currency, the Swarm network uses. Nodes also receive BZZ when they service a request successfully. Therefore, they are motivated to cache or even buy chunks from nodes, to relay them themselves later during other requests. The storage for those chunks is called *cache*.

To upload chunks to the network, so-called *Postage batches* are required and attached to chunks. They associate a value with that chunk. Uploaded data is stored in the reserve. Over time the value of the used Postage batch is burned (i. e., destroyed, no one gets anything for that currently). When the storage limit of the reserve is reached, the chunks with the lowest value are moved to the cache. Consequently, the node tries to maximize the per chunk value of it's reserve. The cache holds those

chunks that were moved from the reserve there and chunks that are stored to relay them later for profit. This storage is pruned regularly when its limit is reached, by removing chunks that were not requested for the longest time. This is a good metric to find out which chunks are the most popular and thus the most profitable in terms of SWAP income.

Postage batches need to be bought with BZZ. Their price depends on the so-called *amount* and the *Batch depth*. The amount specifies the balance of a batch, while the depth specifies how much the chunks are spread throughout the network. Currently, the time to live of chunks cannot be reliably determined beforehand, since it depends on the current price which is variable, but it can be estimated based on the remaining amount batch balance and the current price. The Swarm Foundation therefore suggests buying batches with an amount of 10000000 and a depth of 20, since they expect that amount to keep files in the network for the foreseeable future [Swa21a].

There are two chunk types, *content-addressed chunks* and *single-owner chunks*. The formers address is computed using the Binary Merkle Tree hash function on a binary Merkle tree over small segments of the chunk data. The latter is computed as the hash of the owner's address and an identifier. It is used to allow a user to attach data to a specific part of the Swarm network address space.

On top of DISC it is possible to create many different applications. We will go now deeper into one, since it is the most straightforward use-case of the Swarm network: File storage. If a file is bigger than 4 kilobytes, it is split into chunks which form the leafs of a tree. The intermediate chunks reference 128 other intermediate or leaf chunks. The address of the root chunk becomes the address of the whole file and serves as its checksum to verify if the whole file was retrieved. Figure 2.3 visualizes this construction. Other use-cases are file collections or node-to-node messaging.

2.7 LBRY

In this section, we are talking about a protocol that gained traction in recent years: The LBRY protocol. It was designed to solve the problems, that are shared by platforms like Youtube[9], centralized cloud providers like AWS[10] and the BitTorrent protocol [GK]. Centralized services often engage in behavior that does not align with their users interests. Those behaviors range from extorting fees, arbitrary changes to the terms of service or plain old censorship on the behest of institutions around the world. BitTorrent does not have such faults but this does not mean that there are none. There is no way of getting the file address from the protocol itself. Hence, it has to be known beforehand.

Another problems is that there is no commonly used way known to us of monetizing the published content over the BitTorrent protocol in addition to there being no incentive for users to seed content, other than pure altruism. LBRY solves those problems by utilizing a combination of a Kademlia DHT to store and retrieve files in combination with a permissionless POW blockchain to announce content and to make it possible to pay for it if publishers chose to charge for it. What now follows is a description of the relevant parts of the LBRY protocol. We begin by describing how the blockchain part works.

[9] https://www.youtube.com
[10] https://aws.amazon.com

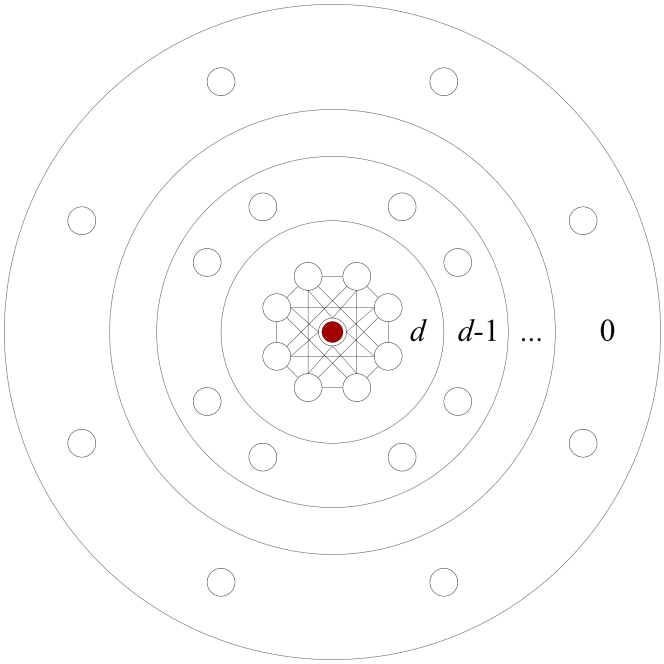

Figure 2.2: This figure shows the connection topology of a Kademlia node. The red nodes neighborhood of proximity order d contains at least eight nodes. It is also connected to eight more peers for each shallower proximity order $d-1, d-2, \ldots, 1, 0$. We call this Kademlia connectivity [MM02; Swa21b].

The first piece to understand are so-called *claims*. A claim contains metadata about things that exist on the network. That metadata contains information, like the title, the file description or a list of tags. They are divided into two types: *Stream claims* and *Channel claims*. The former contains metadata about streams, while the latter contains data about pseudonyms which can serve as a publisher for stream claims. Claims are structured in a so-called *Claim tree* to make it possible to address them with human-readable names. Hence, claims can be referenced by a Uniform Resource Locators like this one `lbry://@GeremiasValier:0/Never-Gonna-Give-You-Up---Rick-Astley--:0`. The claims themselves are stored in the leafs, while the name is stored as the normalized path from the root to the claim. Normalization works by first converting the name using Unicode Normalization Form D [Uni21] and lower casing it, making the URLs effectively case-insensitive. The Claim tree is implemented as a *Merkle tree* of which the root hash is stored in each new block, to enable verification of the claim tree and therefore the URL.

Like Swarm, LBRY uses a Kademlia DHT to find an appropriate host and make the data retrievable. To upload data, the file is first split into a series of *blobs* which are analogous to Swarm's chunks. They have a maximum size of 2MiB. Blobs are hashed with *SHA-384* to create a chunk address. If a node announces blobs to the network, they inform the responsible nodes about the availability of those blobs and that they can be found at the announcing node.

When a node tries to fetch a blob, they send a request to the responsible node to get information about the nodes that actually store that blob. To do so, one first has to query the DHT. In contrast to Swarm, this works by iteratively requesting blobs from intermediate nodes even if they do not store them, instead of the message being relaid by those nodes to the target. With the requests from each

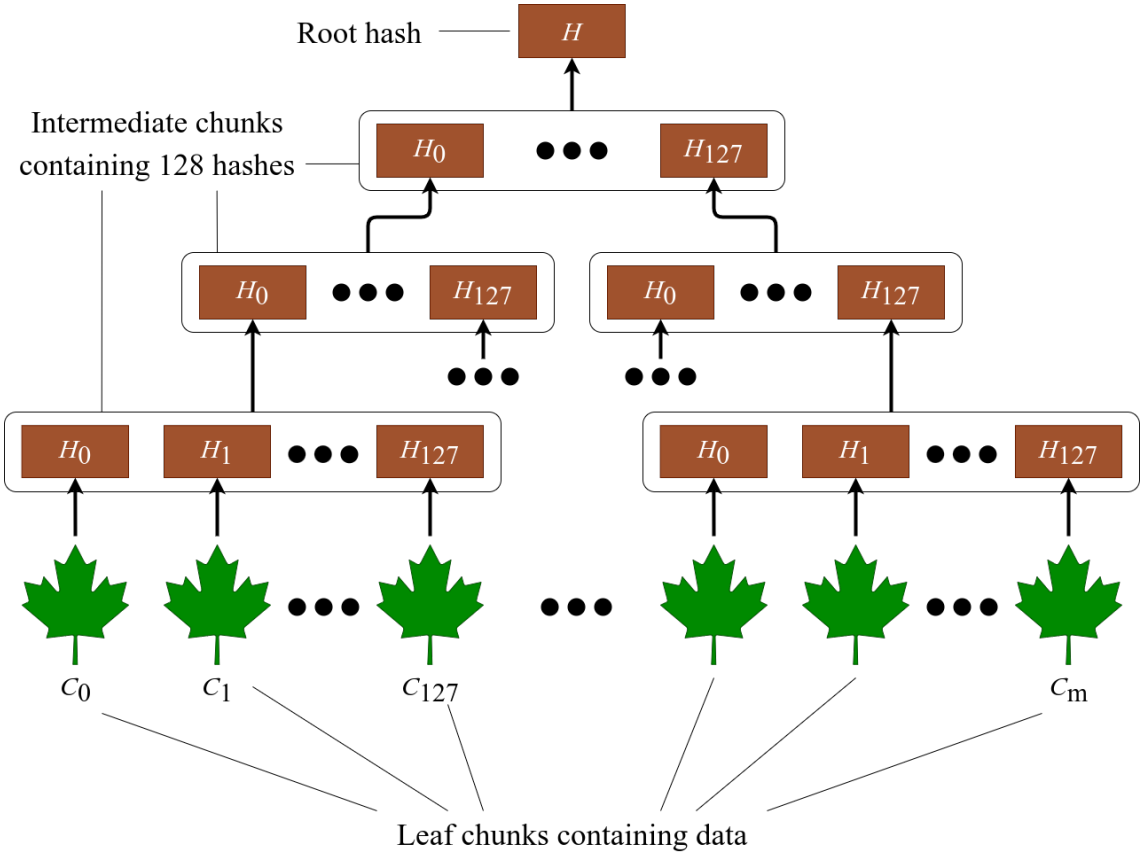

Figure 2.3: Here we can see the chunk tree for file storage in Swarm. Each intermediate chunk contains 128 hashes of child chunks. Those child chunks can either be more intermediate chunks or leaf chunks. The latter contains the actual file data. The root hash serves as the files checksum and address [Swa21b].

iteration targeting nodes that are nearer to the responsible node than the requests from the previous iteration. This works because of the following mechanisms. If a node receives a blob request and it is not responsible for that blob, it returns the list of neighbors that are nearer with respect to their addresses to the requested blob. After that, the querying node sends the same request to the nodes in that neighbor list. If a node is responsible for that blob, then it answers with a list of all peers that currently seed that blob. The querying node then contacts those nodes directly and downloads the blob from them.

All created chunks together form a so-called *Stream* which is then announced to the network by storing it on the blockchain as a Stream claim. Doing so costs LBC which is the currency that fuels the chain. LBRY also enables users to discover claims by integrating Elasticsearch[11] as a search engine into their full nodes [LBRb]. This way, users can search for content they are interested in. Assuming there are a lot of full nodes, then one can trust those search results, because then the possibility of querying a full node that alters the results in a malicious way is low.

[11] https://www.elastic.co

3 Related Work

In this chapter, we talk about work that was created by other authors that relates to our problem of creating a censorship resistant SCD registry. We start with a section about the work of Lamparelli about SCL and SCDL. After that, we describe a system called Namecoin. This is followed by a discussion about the popular blockchain explorer Etherscan. Then comes a section about the Contract Registry Pattern and finally we conclude by talking about UDDI.

3.1 SCL and SCDL

Lamparelli et al. had the goal of integrating smart contracts into conventional software systems and business process engines. For that, they proposed the SCL to identify smart contracts over the internet in addition to the SCDL which allows an abstract description of external smart contract interfaces [Lam20; LFB+19]. These are the foundation of the SCIP [Lam20] which facilitates the uniform interaction with smart contracts that rely on different blockchain technologies. Figure 3.1 shows how all three support an approach that solves the integration problem using a service-oriented approach. An external consumer invokes a smart contract by sending a technology-agnostic SCIP request message to a SCIP Gateway. The SCIP Gateway uses the information from the SCIP message and the information contained in the SCL to invoke the addressed smart contract by leveraging the Application Programming Interface (API) of the blockchain that the smart contract resides on. To construct such a SCIP request, a registry has to be queried for the SCD that corresponds to the smart contract to get the necessary information to perform the actual invocation. SCDs document the external smart contract interface in SCDL.

Their proposal solves the posed problem, but the registry is a centralized component [Lam20]. Therefore, its operator can tamper with it. The work described in this thesis can be seen as follow-up research to Lamparelli et al.'s work, since it serves as a decentralized replacement for the centralized registry they provided.

3.2 Namecoin

Namecoin[1] is a blockchain system with the goal of providing a censorship resistant, decentralized Domain Name System (DNS) [Nam14]. Not only that, but it is also the first solution to *Zooko's triangle* which refers to the problem that it is impossible to create a decentralized naming system that is both meaningful to humans and secure at the same time [WSG13]. This means that every other DNS has to make a compromise between those three properties. Figure 3.2 illustrates that

[1] https://www.namecoin.org

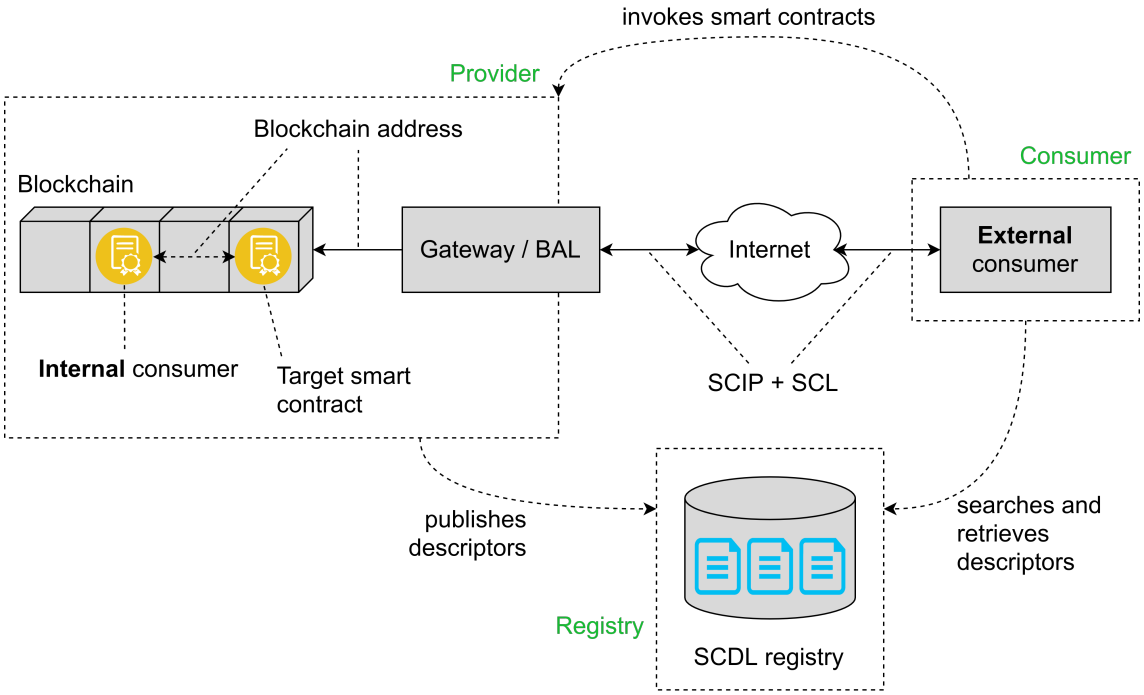

Figure 3.1: This figure depicts how SCIP,SCL and SCDL are used to integrate smart contracts into conventional systems and business process engines on a conceptual level. External consumers query information about a smart contract from an SCDL registry and use it to construct a SCIP message. After that it sends the message to a SCIP Gateway which is located with an SCL. The SCIP Gateway uses the information from the SCIP message and the information contained in the SCL to invoke the addressed smart contract [Lam20; LFB+19].

problem. Domains that are registered with Namecoin have the top level domain *.bit*. Another use case is the creation of identities which can be enriched by arbitrary information. They can be turned into an *OpenID* which may be used to sign into websites that support such identifiers. All of this is achieved by storing key-value pairs on a blockchain which is accessed by client software that enables the lookup of .bit domains.

It is technically possible to use Namecoin as a SCD repository. However, the lack of an enforced, standardized format to describe smart contracts and to query for them makes this infeasible since this would result in many different description formats which users need to adopt before using them.

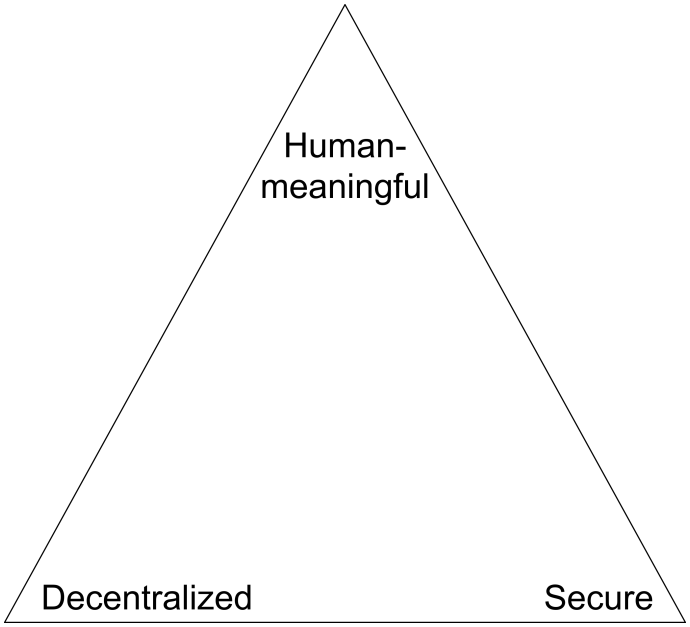

Figure 3.2: Zooko's triangle refers to the problem that it is impossible to create a decentralized naming system that is both meaningful to humans and secure at the same time [WSG13].

3.3 Etherscan

Etherscan[2] is a well known block explorer for the Ethereum blockchain [Eth21]. The service stores information about every block that is created. This information ranges from the amount of gas used to perform all the transactions included into a block. Additionally, this does include information about smart contracts and their validity. The latter is achieved by users uploading the contract code to the service where it is compiled with the exact same compiler version and settings that were used to compile the contract in the first place. After that, the resulting byte code is compared to the code that is deployed on-chain. Assuming both are equal, the contract is marked as verified and the code can be reviewed by users. Lately, Etherscan began indexing those source files to offer search functionality [Eth22b]. Therefore, users can discover new contracts by searching for keywords, the address, the transaction date and many similar attributes.

Etherscan is a centralized service, hence they possess ultimate power to alter their stored data or to exclude certain users from using their services. As the name Etherscan suggests, their service is Ethereum specific. This means that smart contracts from other chains are not indexed and cannot be verified. Thus, they cannot be found in their registry. Consequently, it is not a good solution for our problem, but it serves as concrete example on how a Smart Contract Registry might look like.

[2] https://etherscan.io

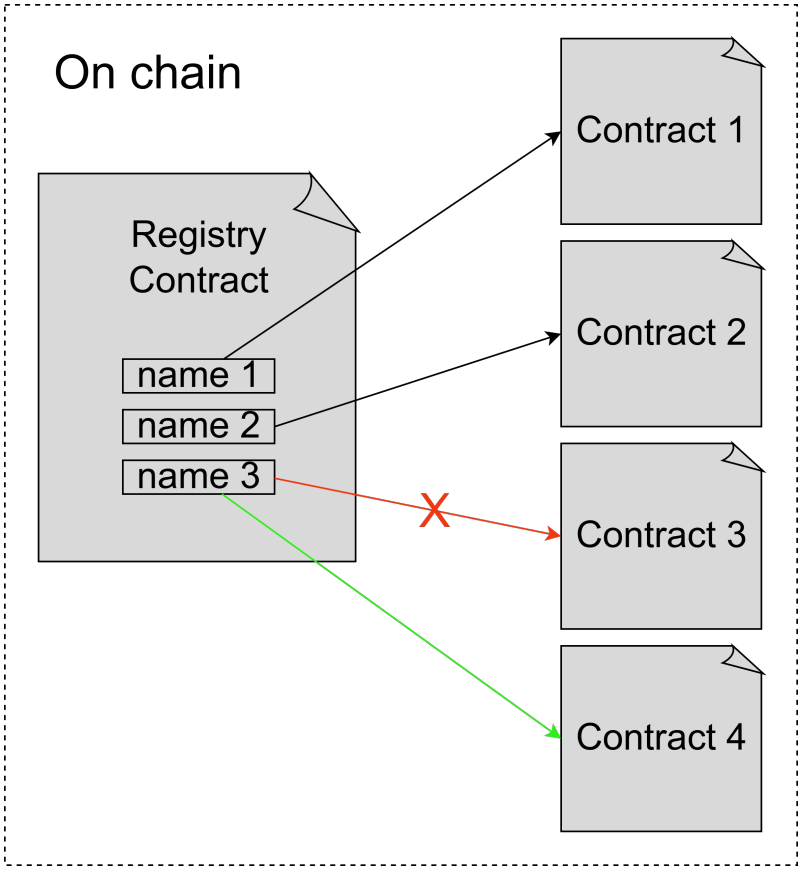

Figure 3.3: This figure shows how a contract is updated in the Contract Registry Pattern. In this case *Contract 3* is updated with *Contract 4* by changing the address *name 3* points to the address of *Contract 4* [Com].

3.4 Contract Registry Pattern

In this section, we talk about a pattern called *Contract Registry* [Com]. It was created to address the problem of upgradeability of smart contracts. Like other pieces of software, smart contracts can have bugs or need new features. Therefore, they need to be updated. Problematically, smart contracts are deployed as a piece of an immutable ledger. Hence, they cannot be updated like other software systems, but need to be completely redeployed. This is a problem if other contracts want to invoke that updated contract, since they only know about the old address. Fixing this implies that the invoking contracts must be updated as well. The Contract Registry solves this problem by storing the contract address in another smart contract which is queried every time the actual contract is invoked. Addresses need to be stored in variables to make them updatable in the future. Following that, an invoking contract needs to first query the Contract Registry to get the address of the target contract before each invocation. Thus, it is possible to keep track of contract addresses without updating the invoking contract as long as the interfaces are the same. Figure 3.3 shows how a contract is updated. In this case, *Contract 3* is updated with *Contract 4* by changing the address *name 3* points to.

It appears feasible to create a public registry using this pattern. Still, it lacks search functionality and does not give users more information about the actual smart contract other than its location. Users consequently need to know the names and the purpose of the smart contract they are interested in beforehand. To our knowledge, such a system also does not exist at the time of writing this, but there are systems that use this pattern. An exemplary system is the Ethereum Name Service (ENS)[3], which is a naming system based on the Ethereum blockchain to resolve the location of resources [Tru]. Another work that makes use of Contract Registries was created by Lu et al. [LBW+21]. Their goal was to support smart contract development with a model-driven approach. Thus they presented a tool called *Lorikeet* which is able to generate Contract Registries and track their changes.

3.5 Universal Description, Discovery, and Integration

Universal Description, Discovery, and Integration (UDDI) refers to a registry of Web service descriptions [CDK+02] as Extensible Markup Language (XML) documents. It can be accessed via the Simple Object Access Protocol (SOAP). The entries are organized into two entities that describe the business and the services they provide. The first is the *businessEntity*. It stores not only information about the name and contact details of a business, but also *businessServices*, which describe services provided by that business. The technical information of those businessServices is stored as references to so called *tModels* which are the second group of important entities. Thus, in order to store the Web Services Description Language (WSDL) definition of a service, a tModel has to be provided at first, before it can be referenced in the corresponding businessService. The reasoning for this is that there can be types or formats that cannot be anticipated. This allows referencing arbitrary information types. Furthermore, businesses and services are categorized to make it possible for users to find them. UDDI facilitates that also by using tModels for each category. Consequently, businessServices reference those tModels in their XML description.

Shortly after the UDDI specification was released, the UDDI Business Registry (UBR) was created. This is a publicly accessible UDDI registry operated by multiple companies. Those are IBM[4], Microsoft[5], SAP[6] and NTT Communications[7] [CW04]. Updates to the registry are propagated to all of the operated nodes. Hence, their databases are replicated. Companies also started to host company internal UBRs to make services discoverable from inside the company network, but not from the outside.

However, UBR is not censorship resistant. Individual companies can just decide to remove entries from the registry. Replication is no sufficient solution, because those four companies may also collude on that task.

[3] https://ens.domains
[4] https://www.ibm.com
[5] https://www.microsoft.com
[6] https://www.sap.com
[7] https://www.ntt.com

4 Architecture alternatives

This chapter concerns itself with two possible architecture alternatives for the SCD registry and decide on one. We will first discuss the requirements. Next, we will describe both architectures and we begin with the Smart contract based registry and end with Yet another LBRY frontend (YaLBRYf). Finally, we conclude with our decision and justify it.

4.1 Requirements

In this section, we discuss the requirements that our SCD registry needs to fulfill. We begin with the functional requirements and then follow up with the non-functional ones.

4.1.1 Functional requirements

Here we outline the functional requirements. They are displayed in a use case diagram in Figure 4.1. We go through them from top to bottom.

FREQ1: Search SCDs The first is the ability to search for SCDs. This is necessary so users can find SCDs that describe smart contracts, they may be interested in. For example: A user might be interested in a smart contract that adds two numbers together. They can then search for SCDs that contain a function called “add” or “addTwoNumbers” or similar ones.

FREQ2: Retrieve SCDs Our second requirement is about retrieving SCDs from where they are stored. We require this, so users can go through them and judge if the represented smart contract might be useful.

FREQ3: Register SCDs This requirement is about the ability to register new SCDs. This functionality is instrumental to fill the registry with information.

4.1.2 Non-functional requirements

We defined four non-functional requirements, of which one has three sub requirements that are directly tied to it and necessary to fulfill it. They are the following:

NFREQ1: Trustworthiness The registry and its entries need to be trustworthy. Otherwise people might be less incentivized to use the registry if they fear that their entries might get removed or hidden from the public now or in the future. Trustworthiness can be improved by fulfilling the subrequirements, which are:

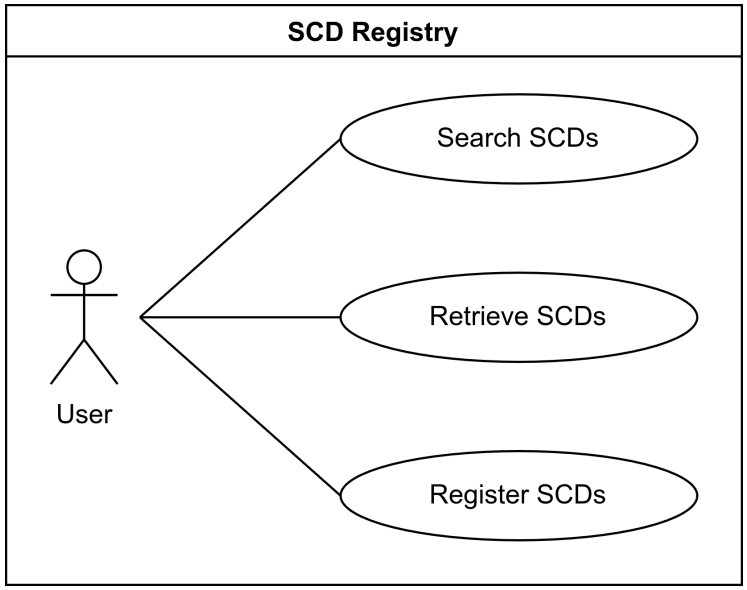

Figure 4.1: This use case shows the three functional requirements for a SCD registry. They are *Search SCDs*, *Retrieve SCDs* and *Register SCDs*.

NFREQ1.1: Data Integrity This requirement is about the verifiability of the uploader (i. e., is the uploader actually the actor they claim to be) and that SCD does not change without the users noticing. Unfortunately, SCDs make no assumptions about the underlying blockchain technology which we would need to make to guarantee complete *Data Integrity* which is "the degree to which data are complete, consistent, accurate, trustworthy, and reliable and that these characteristics of the data are maintained throughout the data life cycle" [PIC21]. We can also not guarantee that the described smart contract is not dangerous (i. e., steals a users funds and NFTs).

NFREQ1.2: Censorship resistance No single actor should control which entries are included into the registry. Furthermore, all already entered entries should be discoverable and not be blacklisted by some mechanism. As a consequence, there needs to be not only a decentralized consensus algorithm that decides which SCDs are included but also no mechanism to blacklist SCDs afterwards. Users can therefore be sure that SCDs are not excluded for arbitrary reasons and that other users or potential registry operators do not alter the entries or hide them from the public.

NFREQ1.3: Transparency The consensus algorithm needs to be traceable. Consequently, the complete algorithm and all used technologies need to be open source as there is no reason to trust closed source software to not interfere with the consensus algorithm. Otherwise, people can never be sure that the system does what it claims to do and nothing else, like for example altering the transactions' data before it is submitted.

NFREQ2: Interoperability The registry needs to offer some sort of API or data sharing capability, so other programs can make use of it and retrieve SCDs to get information about smart contracts programmatically, since it is supposed to be a replacement for the SCD registry which is part of the system described in Section 3.1. It can also be seen in Figure 3.1.

NFREQ3: No single point of failure A single point of failure makes the system easier to attack and might therefore lead to problems down the line. Such attacks can range from conventional cyberattacks, like DoS attacks to the use of laws, to attack such a registry (e. g., one company hosts a necessary part of the system all by itself which can then be forced by a third party to take that part of the system offline, dragging everything down with it). Assuming there exists an operator of that single point of failure, they can also use their power to blackmail the users of the system or change the terms of service at will and without any oversight or consent from the users. To be more precise: it is important to avoid centralization.

NFREQ4: Usability The registry should be easy to use. A complicated setup for users that just want to search through the entries also needs to be avoided. Not fulfilling this might lead users to give up too fast when trying to use the registry.

4.2 Design approach

To come up with our designs, we loosely followed a *Design Science Research* approach. We began by defining the requirements our censorship resistant registry needs to have and we recorded them in Section 4.1 to define our problem. Following that, we started to review existing literature, relating to the topics of:

- service registries
- existing ĐApps
- blockchain technologies
- decentralized storage systems
- decentralized naming systems

The result of that research are the Chapters 2 and 3. With the requirements and the gathered knowledge in mind, we created the two architectures that we are going to describe and evaluate in the following sections.

4.3 Smart contract based registry

We now begin with the first architecture proposal, namely with the *Smart contract based registry*. Figure 4.2 shows an overview over this architecture. As the name suggests, a smart contract is an important part of this registry. We call it the *Registry Contract*. We assume that this contract is deployed on Ethereum, since it is a well known and widely used blockchain platform. Hence we also assume that Ethereum is very decentralized. The Registry Contract exposes an API that realizes the functional requirements of an "ideal" SCD registry. Consequently, it makes SCDs discoverable and searchable. This is done by storing SCD metadata in the Registry Contract. That metadata then contains the actual location of the SCDs. One might ask why we are not storing the SCDs themselves in the contract. Our reasoning for that is that storing complete SCDs on Ethereum might be too expensive. Listing 1 illustrates that problem. That Listing shows a shortened version of the ERC1155 [Opea; Opeb] contract developed by OpenZeppelin. Despite omitting data that would

Listing 1 This listing shows a shortened version of OpenZeppelins ERC1155 [Opea; Opeb] contract. The actual contract can also be seen in the appendix.

```
{
  "scdl_version": "1.1",
  "name": "ERC1155",
  "version": "v4.6.0",
  "latest_URL": "https://somewhere.com/else",
  "description": "@dev Implementation of the {IERC20} interface. This implementation is agnostic to the way
  ↪  tokens are created. This means that a supply mechanism has to be added in a derived contract using
  ↪  {_mint}. For a generic mechanism see {ERC20PresetMinterPauser}. TIP: For a detailed writeup see our
  ↪  guide https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How to
  ↪  implement supply mechanisms]. We have followed general OpenZeppelin Contracts guidelines: functions
  ↪  revert instead returning `false` on failure. This behavior is nonetheless conventional and does not
  ↪  conflict with the expectations of ERC20 applications. Additionally, an {Approval} event is emitted on
  ↪  calls to {transferFrom}. This allows applications to reconstruct" ... ,
  "author": "OpenZeppelin",
  "created_on": "2022-06-24T08:27:43.1599534Z",
  "updated_on": "2022-06-24T08:27:43.1599553Z",
  "life_cycle": "ready",
  "scl": "https://mygateway.com?blockchain=ethereum&blockchain-id=eth-
  ↪  mainnet&address=0xa0b73e1ff0b80914ab6fe0444e65848c4c34450b",
  "blockchain_type": "ethereum",
  "blockchain_version": "v0.5.11+commit.c082d0b4",
  "internal_address": "0xa0b73e1ff0b80914ab6fe0444e65848c4c34450b",
  "metadata": "https://somewhere.com/else/meta",
  "hash": "c89425c4fb9e686fbfec43a3b724db25 -",
  "is_stateful": true,
  "functions": [
    {
      "name": "supportsInterface",
      "description": "",
      "scope": "public",
      "has_side_effects": false,
      "dispatcher": "",
      "inputs": [
        { "name": "interfaceId", "type": "bytes4", "is_indexed": false }
      ],
      "outputs": [{ "name": "", "type": "boolean", "is_indexed": false }],
      "events": []
    },
    ...
  ],
  "events": [
    {
      "name": "TransferSingle",
      "description": "",
      "outputs": [
        { "name": "_operator", "type": "address", "is_indexed": true },
        { "name": "_from", "type": "address", "is_indexed": true },
        { "name": "_to", "type": "address", "is_indexed": true },
        { "name": "_id", "type": "uint256", "is_indexed": false },
        { "name": "_value", "type": "uint256", "is_indexed": false }
      ]
    },
    ...
  ]
}
```

Construct	Description	Syntax element	Value type
Author public key	This is the public key of the SCD metadata uploader (i. e., the public key that belongs to the private key that was used to sign the SCD)	author_pub_key	string

Table 4.1: This table defines our extension to the SCD JSON schema. We propose to extend it with the authors public key.

be part of the SCD, it fills the entire page. Hence, storing something like this on the Ethereum blockchain is expensive. Furthermore, size management of SCDs is hard, since longer contracts typically lead to longer SCDs, because they contain more functions and events, which also need to be described in the SCD. The complete ERC1155 contract can be found in the Appendix.

The complete SCD is stored in a so-called *External SCD Storage*. We make no assumptions about that storage other than, that it stores the SCD in a machine-retrievable way. Therefore many different systems for this task are possible. Some of those possible storages are decentralized storages, like Swarm or IPFS, while others are controlled by centralized services that could be managed by the SCD-metadata uploaders themselves, for example an Apache HTTP Server[1]. Technically, even services like OneDrive[2] or Google Drive[3] are valid storages, even though they are far from the best option if one wants to be in control of their own data, since such services can restrict access, if they choose to do so either by accident or on purpose, like they have done in the past [Goo21; Hol12; Syn17; Wri22].

We also propose the creation of external systems that provide better search capabilities than the Registry Contract. They are necessary to enable a more detailed search through the existing SCDs, since the Registry Contract does not have enough information to offer that. Those external systems are supposed to fetch SCDs when they are registered in the Registry Contract and index them for their own search algorithms. They are called *External Search Providers*.

The metadata stored on-chain also contains the signature of the actual SCD. The signing key is required to be the same one that was used to store the metadata. We are storing the signature to make it possible to verify, if the SCD changed since the metadata was uploaded. Users should not trust an SCD if the signature verification fails. Furthermore, we propose to add a new mandatory field to the SCD JSON schema. We call that field the *Author public key* and it can be further inspected in Table 4.1. This key is compared to the key of the metadata uploader. If they are not equal we have to assume that the metadata uploader tried to fake "ownership" of an already registered SCD by registering the SCD again but this time signing it with their own private key. Thus, users should not trust the metadata if that comparison fails.

[1] https://httpd.apache.org

[2] https://www.microsoft.com/en-ww/microsoft-365/onedrive/online-cloud-storage

[3] https://www.google.com/drive

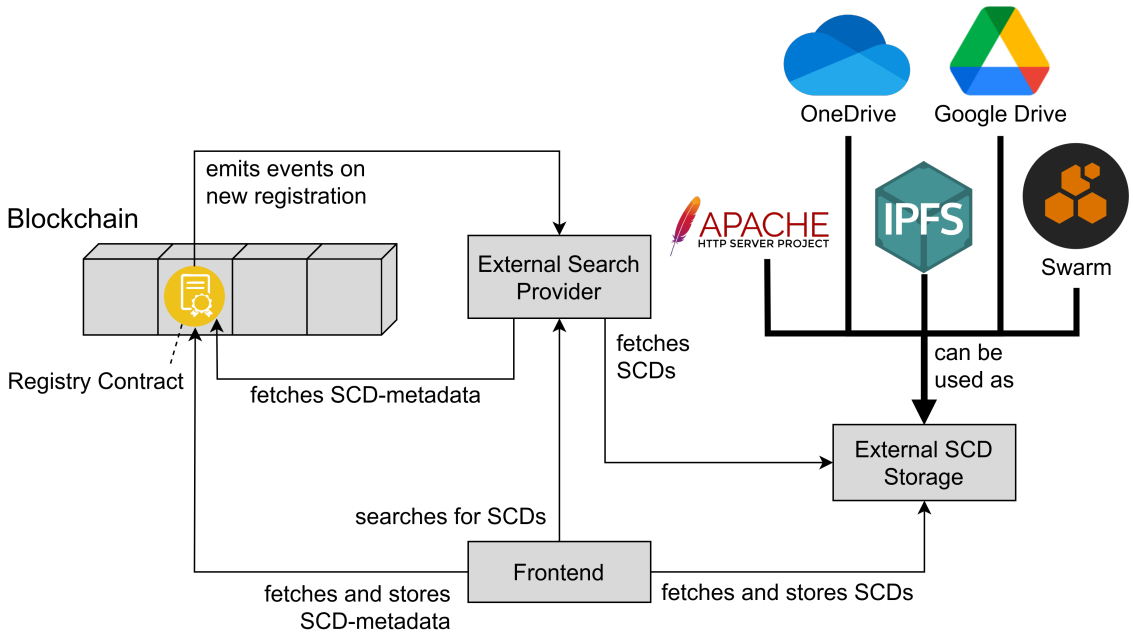

Figure 4.2: This figure shows an overview over the architecture of the Smart contract based registry.

A *Frontend* would serve as the integration point for all those components. We require this frontend to not be reliant on a central authority or service. This can be realized by using decentralized storage solutions that enable hosting of single-page applications like Swarm or IPFS. Another possibility might be the distribution of a client application via BitTorrent. It is also possible to integrate the components manually to create new applications by leveraging their APIs.

4.4 Yet another LBRY frontend

We previously talked about the LBRY protocol. It is easy to notice that it does provide the necessary building blocks for a SCD registry. Those are the ability to store, retrieve and search for files, which in our case are SCDs. The LBRY protocol luckily does this in a decentralized manner. As a result it could be used as a SCD registry.

We previously noted that the files that are stored in the LBRY network also have metadata associated to those files stored on the LBRY blockchain. This metadata is indexed by LBRY nodes to make it possible to search through those metadata instances. Hence, we decided to not only store the SCD as a file in the network but also as that file's description. Consequently, making it possible to perform full text searches for SCDs. In addition to that, we also attach a tag to the file to identify it as an SCD. *Yet another LBRY frontend (YaLBRYf)* is a proposal for an architecture that takes advantage of this protocol. This proposal is not complicated and will be explained in the following.

Figure 4.3 shows a component diagram of that architecture. It only consists of three components: A *Frontend*, the *lbry-sdk* and the LBRY network itself. As the name "Yet another LBRY frontend (YaLBRYf)" suggests it is just a Frontend used to interact with the LBRY protocol. It would therefore

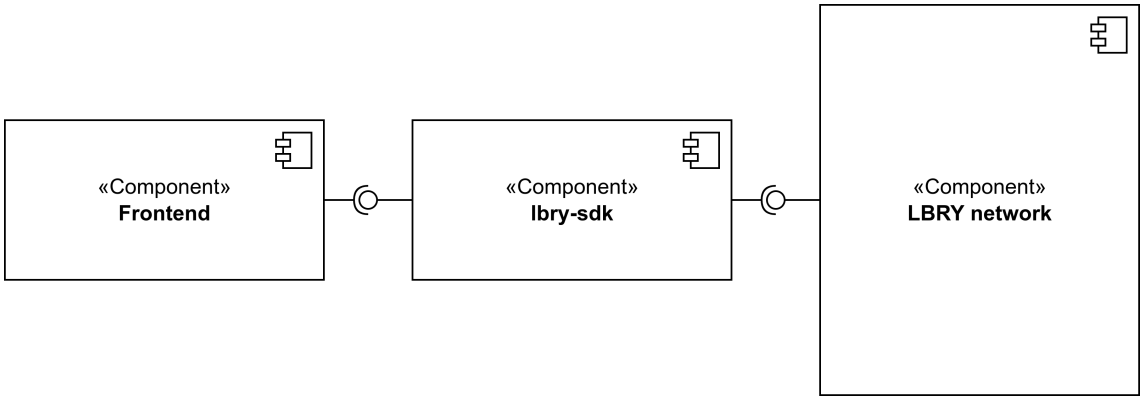

Figure 4.3: This component diagram shows how an architecture based on the LBRY network might look like. It is in general simple as it only consists of a Frontend component which directly interacts with the lbry-sdk. The latter is used to interact with the LBRY network itself.

be interchangeable. Technically, it would be possible to just use already existing frontends like *Odysse*[4] for uploading and retrieving SCDs, since they already offer all the necessary functionality but are typically tailored to consuming and uploading videos.

Our proposed Frontend would make uploading of SCDs easier and make it possible to search specifically for them. Metadata like the title and the description are automatically extracted and set during the upload process. The same is true for the tag that identifies this file as an SCD. During searches, the query is being preprocessed if necessary and then passed to the lbry-sdk. Upon receiving the results, they are filtered, to remove files that do not have the SCD tag. This makes it easier for users to browse through SCDs without being distracted by irrelevant pieces of content, like videos or audio.

Like in the other architecture, we require this frontend to also not be reliant on a central authority or service. Interactions with LBRY itself are performed by invoking functions from the lbry-sdk[5]. The sequence diagrams in the Figures 4.4 and 4.5 show how both uploading and searching would work. The lbry-sdk is a JSON remote procedure call (RPC) server that can be used to interact with the LBRY protocol. It enables users to search for files, upload them, download them and to manage their LBRY wallets.

4.5 Comparison

This section compares the two architectures on the basis of the defined requirements (see Section 4.1) and decides which we are going to implement in the coming chapters.

[4] https://odysee.com
[5] https://github.com/lbryio/lbry-sdk

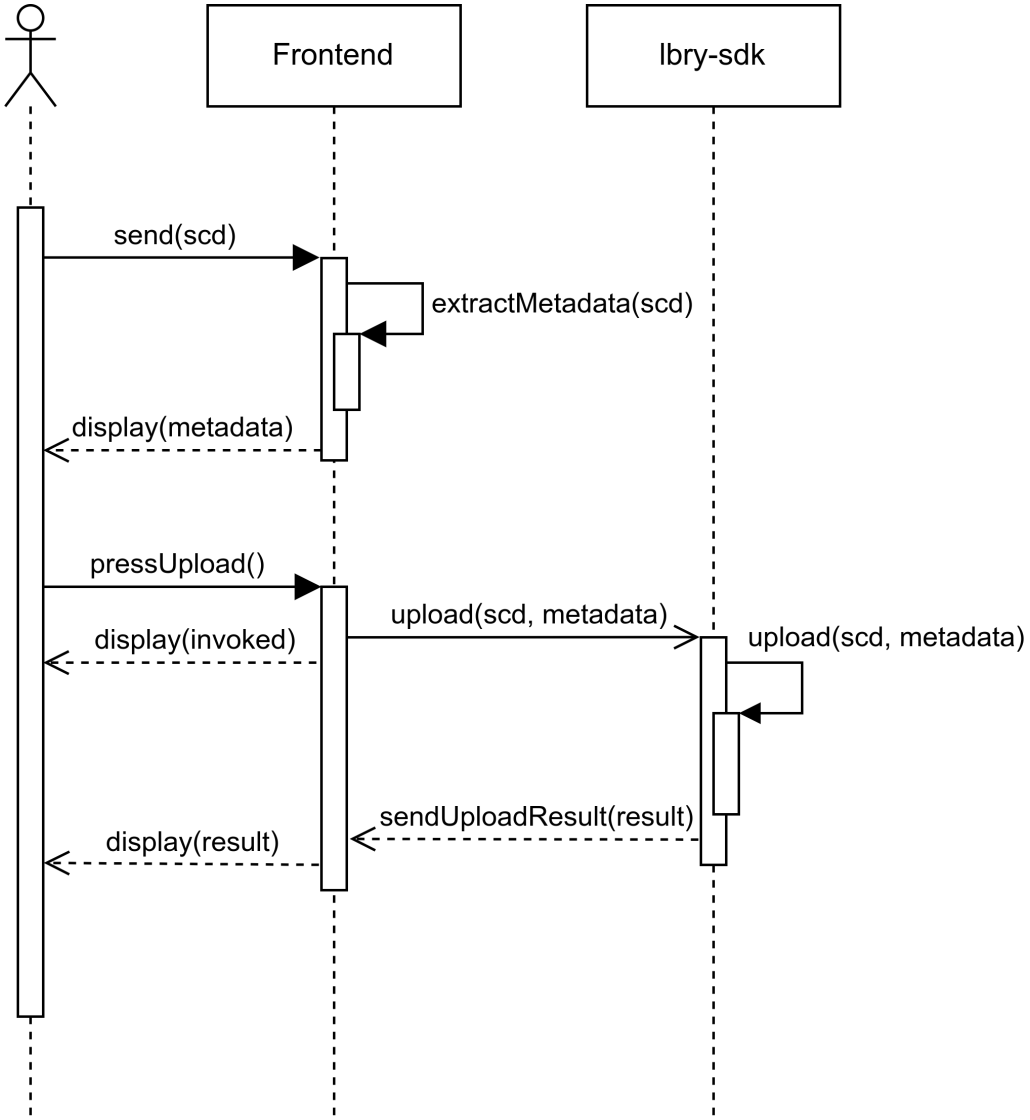

Figure 4.4: This figure shows how uploading a SCD with YaLBRYf would work.

We begin with the functional requirements (FREQ1, FREQ2 and FREQ3), because we deem both architectures capable of fulfilling all of them. YaLBRYf might even be better in that regard, since it provides a single highly capable approach to searching, which can be used right out of the box without any preparation. That offers an enhanced user experience in comparison to relying on External Search Providers. Now follow the non-functional requirements.

4.5.1 YaLBRYf

NFREQ1: Trustworthiness

As we previously mentioned, this requirement is about Trustworthiness and it depends on fulfilling the subrequirements (see Section 4.1.2).

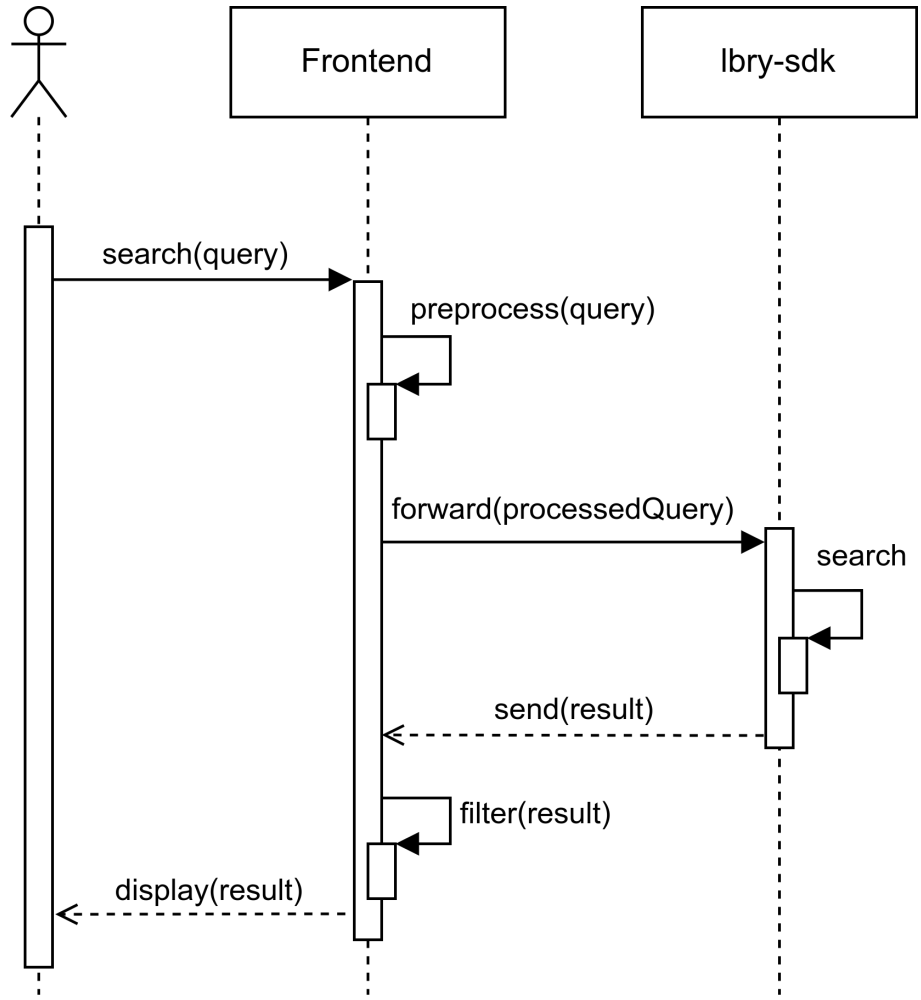

Figure 4.5: This figure shows how searching for SCDs with YaLBRYf would work.

NFREQ1.1: Data Integrity LBRY uses a permissionless POW blockchain to announce the existence of new files to the network. As we previously mentioned, we propose to store the SCD in the file description to have it indexed by the LBRY nodes. This has the additional advantage that SCDs cannot be changed by unauthorized actors. Hence, we consider this requirement to be fulfillable by this architecture.

NFREQ1.2: Censorship resistance As we mentioned previously, LBRY uses a permissionless POW blockchain. Therefore, we get a decentralized consensus that decides which SCDs are included into the registry. Furthermore, this protects against single actors gaining control over the network and thus protects against censorship. Therefore, this is in our opinion also fulfillable.

NFREQ1.3: Transparency LBRY, Inc. publishes all their source code on Github[6] as open source software, hence we consider this requirement to be fulfilled.

[6] https://github.com/lbryio

Since all three subrequirements can be fulfilled, we hold the opinion that NFREQ1 is fulfillable.

NFREQ2: Interoperability

NFREQ2 is about interoperability which can be fulfilled by using the lbry-sdk directly. If this is not enough, additional wrappers can be provided that make the whole process more streamlined and tailored to SCDs.

NFREQ3: No single point of failure

In LBRY data is currently hosted by so-called *reflectors* [GK]. They seed uploaded blobs and can charge a fee for downloads. Except for clients that have the necessary blobs currently stored, they are the main avenue to download files from the network. The discussion online implies that currently all of those reflectors are hosted by LBRY, Inc., but no reputable source was found that confirms this statement. The only one we found is an answer by a moderator in a reddit thread on the /r/lbry subreddit [try22]. That subreddit is dedicated to discussing the LBRY protocol. Furthermore, the lack of detailed documentation on setting up and operating a reflector also supports that suspicion Such a lack of documentation might discourage people from hosting a reflector themselves. But since we also were not able to find sources that contradict our suspicion that LBRY is more centralized than advertised, we chose to believe it. This gives LBRY, Inc. a lot of influence in the LBRY network and makes the whole data network almost completely dependent on them. It is important to mention that our approach does not rely on reflectors since all information is stored on the blockchain, but there are only a few nodes. Furthermore, this is enough to consider decentralization compromised, i.e., attacking a small PoW-based network is easy, and there is a risk that a large portion of the network is controlled by a single entity. This is in our opinion likely, since they are doing the same thing with their reflectors, without providing evidence to contradict that statement. Currently, the price of LBC which is the cryptocurrency that powers the LBRY network is really low. Consequently, this disincentivises people from contributing to the network, if they are doing it for the money and not to keep the network running. Therefore we cannot be certain that NFREQ3, which was about there being no single point of failure, can be fulfilled.

NFREQ4: Usability

NFREQ4 was about usability, which would not be easy to fulfill. Event though the setup only requires installing the lbry-sdk, users also need LBC if they want to upload something. To get it, they either have to mine it, or they have to buy it. The latter requires them to use an exchange [LBR22] which first requires them to buy Bitcoin or Tether [LBR21] and then exchange those for LBC. This process might be easy for an experienced user of cryptocurrencies, but it might be hard for newcomers to the scene.

4.5.2 Smart contract based registry

We now follow with the analysis of the Smart contract based registry.

NFREQ1: Trustworthiness

This requirement can be fulfilled, since the subrequirements are fulfilled.

NFREQ1.1: Data Integrity Like LBRY this approach uses a blockchain. We store metadata about SCDs, which includes the SCD signature. Those signatures can be verified to notice changes or inconsistencies. Consequently, SCDs cannot change without those changes being noticed by users when checking the signature.

NFREQ1.2: Censorship resistance Assuming the blockchain is decentralized enough and uses an applicable consensus algorithm, like Proof-of-Work (POW) or Proof-of-Stake (POS), we deem NFREQ1.2 to be fulfillable.

NFREQ1.3: Transparency NFREQ1.3 can also be fulfilled due to us being able to choose the used software.

NFREQ2: Interoperability

NFREQ2 requires the ability to interact with the system programmatically. This can be done by invoking the Registry Contract.

NFREQ3: No single point of failure

The next requirement which is NFREQ3 is a bit more complicated. First of all the External SCD Storage can be ignored in this case since each user themselves is responsible for maintenance and control. External SCD Storages are systems that uploaders use to store the actual SCD, because only metadata about those SCDs is stored on the blockchain. Unfortunately, there is the possibility that users might become too dependent on external search providers which might use that power to censor SCDs inside their own service. This is a problem, because even though users can verify the entries, they might not choose to do so when they trust the External Search Provider. We thus conclude that this requirement can be fulfilled, assuming that this does not happen.

NFREQ4: Usability

Only the contract address and the location of the blockchain are required for the Registry Contract and a URL to interact with the External Search Provider. Uploading is harder, because it requires setting up the necessary infrastructure to store the actual SCD. The exact difficulty depends on the chosen storage technologies. Hence, we deem this requirement fulfillable, if the user only wants to search through the available SCDs. Otherwise it is hard to fulfill it.

Requirement	YaLBRYf	Smart contract based registry
FREQ1	✓	✓
FREQ2	✓	✓
FREQ3	✓	✓
NFREQ1	✓	✓
NFREQ1.1	✓	✓
NFREQ1.2	✓	✓
NFREQ1.3	✓	✓
NFREQ2	✓	✓
NFREQ3	×	✓*
NFREQ4	–	–

Table 4.2: This table summarizes our decision on which architecture we implemented. Our decision comes down to NFREQ3, because YaLBRYf does not fulfill it. Hence, we decided to implement the Smart contract based registry.
*Regarding NFREQ3 in relation to the Smart contract based registry, the following needs to be noted. There is the possibility that users might become too dependent on external search providers. Even though, users can verify the entries, they might not do that when they trust the External Search Provider. We assume that this does not happen and thus conclude that this requirement can be fulfilled.

4.6 Decision

This section justifies our decision on which system we implemented. A summary of that comparison can be found in Table 4.2.

We decided on implementing the Smart contract based registry. Our reasoning relates mainly to NFREQ3. Not fulfilling this requirement prevents us from creating a truly censorship resistant and therefore decentralized SCD registry. We also found another problem: At the time of writing, the company LBRY, Inc. which develops the LBRY protocol is entangled in a court case with the U.S. Securities and Exchange Commission (SEC) for allegedly offering unregistered securities in the form of their currency LBC [LBRa]. Even though the LBRY protocol and network would theoretically survive the court case even if the SEC wins, development might still slow down or halt completely if no one steps up to continue it, making it an undesirable foundation for our goal. This court case already led some exchanges to no longer allow US customers to exchange LBC which complicates the usage of LBRY for US citizens already [Bit21a; Bit21b; Pol19].

5 System Design

In this chapter, we are going to go deeper into the system design of the Smart contract based registry. First, the Registry Contract does not store the entire SCD, but only a small set of metadata. This brings up the question, why we only store that metadata on chain and not the whole SCD, because doing limits the search capability. The reason is simple: To achieve a high degree of decentralization, a highly decentralized blockchain with many users is necessary. Permissionless blockchains are a good choice for that, since they can become more decentralized over time when more nodes join the network. New nodes can be added without someone preventing that. One such blockchain is the Ethereum blockchain (see Section 2.2). However, transaction fees are relatively too high to allow storing whole SCDs on chain. Such high costs would disincentives people from registering their SCDs in such a registry and thus make it in turn a bad solution.

As a remedy, we propose the possibility of using external search providers which refer to the Registry Contract for verification of their data. A user that wants to search through the stored SCDs can then either use the limited search functionality of the Registry Contract, or a more capable external search provider. But how would users register SCDs? The first step would be to put the SCD in a storage that enables retrieval by users that want to look at the SCD. Such storages might be IPFS, Swarm, LBRY or a simple HTTP server. Following this, the smart contract developer needs to invoke the Registry Contract to store the metadata of the SCD and announce its registration. Table 5.1 shows the metadata of a SCD.

Name	Description	Value type
id	Unique id of the metadata. This value is generated by increasing a counter, every time a new SCD is registered.	uint256
name	Name of the smart contract.	string
author	Public key of the entity that registered this SCD. This key is also compared to the Author public key, stored in the SCD (see Table 4.1)	string
internalAddress	The address of the smart contract. The address format depends on the used blockchain technology.	string
url	Where the complete SCD is located (i. e., a URL).	URL
signature	The signature of the SCD. The signing key has to be the same that signed the transaction that was used to create this entry on the chain.	string
version	Version number of the contract.	string
functions	The names of all invocable functions.	string[]
events	All names of the events the smart contract can emit.	string[]
isValid	Marks the entry as existing on chain if it is true.	boolean
blockChainType	Blockchain type (i. e., on which blockchain is the contract deployed).	BITCOIN=0, ETHEREUM=1, FABRIC=2, NEO=3

Table 5.1: This table defines how SCD-metadata looks like. The name is equal to the actual property name in Solidity.

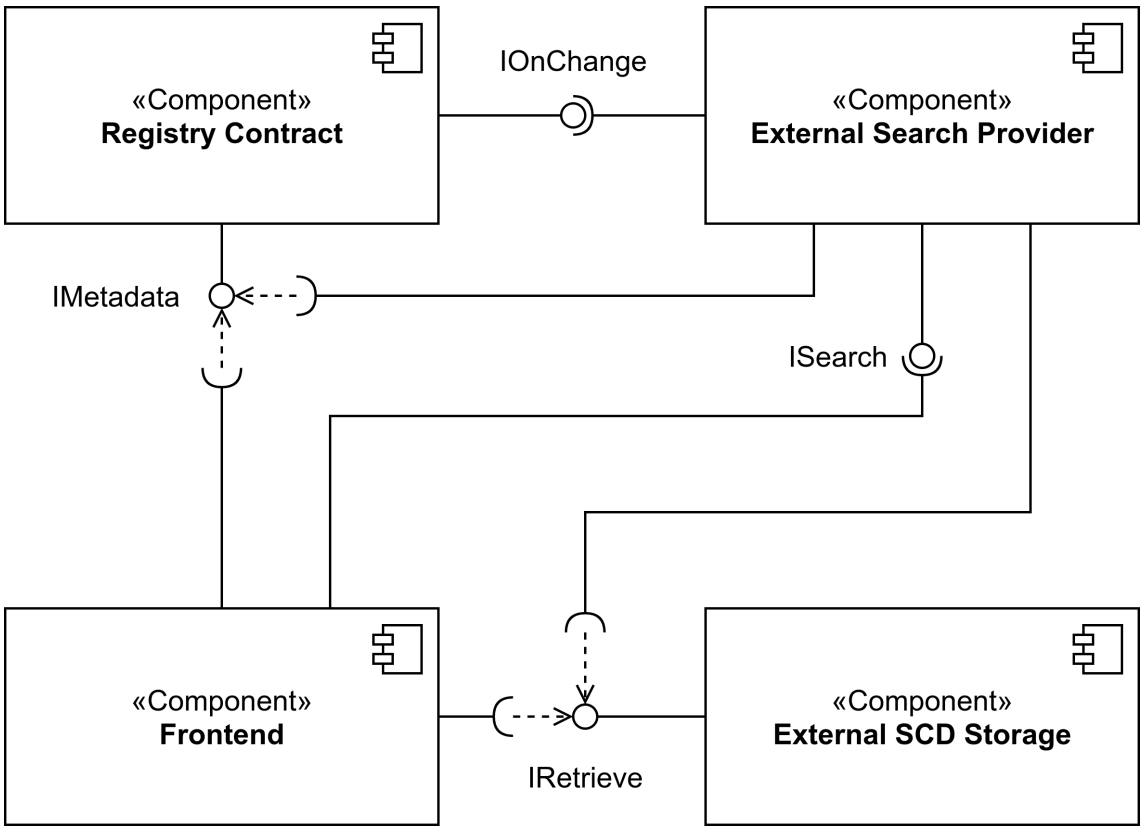

Figure 5.1: This figure shows the component diagram of the Smart contract based registry. It consists of four components, which are the *Registry Contract*, the *Frontend*, the *External Search Provider* and the *External SCD Storage*. We implement the latter both in the form of the Swarm network and of a HTTP server.

The Registry Contract emits an event when a new SCD is registered to announce changes to the External Search Providers. They then fetch the new SCD from the SCD storage. Despite that, the developer needs to keep the storage operational, so that it can still be accessed at a later point in time from the original source. If users use the service of an external search provider, they need to verify the signatures of the results. This is important, because SCDs with wrong signatures should not be trusted, because that means that the actual SCD file changed after the metadata was stored in the Registry Contract. Furthermore, the equality of the public key that is stored in the SCD-metadata and the one that is stored in the SCD is verified. If they are not equal, we can assume that the user that registered the SCD-metadata wants to fake ownership of the actual SCD. Therefore, users should not trust such SCD-metadata if the check fails.

5.1 Components

This section describes the components that are relevant for this design in more detail. Figure 5.1 shows them in a component diagram.

5.1.1 Registry Contract

The Registry Contract is a smart contract that serves as a decentralized storage for the SCD-metadata. It not only facilitates both storing and querying the metadata but it also informs External Search Providers when new SCDs have been published. Two interfaces are provided by this smart contract. They are called *IMetadata* and *IOnChange*. IMetadata is responsible for the storing and querying part, while IOnChange informs the External Search Providers.

5.1.2 Frontend

The *Frontend* component provides a User interface (UI) to all kinds of users, whether they want to register SCDs or search for them. It should also automatically fetch the actual SCD, if a user wants to look at it automatically and perform the necessary signature checks. Furthermore, it also checks if the public key, stored in the SCD metadata and the one, stored in the SCD are the same. The user is informed if any of those checks fail. To do all of this, it relies on three interfaces, which are the Registry Contract's IMetadata interface, the IRetrieve interface of the External SCD Storage and the External Search Provider's ISearch interface. The first two of those are mandatory for the basic functionality, while the last one is optional. IMetadata is needed to enable searching though the available metadata that is stored on chain and to register new SCDs. IRetrieve fetches the actual SCD to verify and display it. The ISearch interface is provided by External Search Providers and supports the search functionality, because only utilizing the IMetadata interface is limiting. In addition to all of this, the Frontend also serves as the main integration point for all the other components.

5.1.3 External SCD Storage

The External SCD Storage is a component that has the task of storing the actual SCD in a machine-accessible way. It provides the IRetrieve interface for that retrieval process. This storage can be realized with multiple different technologies such as webservers, Swarm, IPFS or LBRY, just to name a few. Our example design provides two different External SCD storages: A HTTP server and the Swarm network. Even though we provide an implementation for the former, it is not required to be used, since it serves as a stand-in for any form of location from which SCDs can be retrieved over HTTP. Thus, our HTTP server can easily be exchanged by webservers like an Apache HTTP Server or even cloud storage providers like Google Drive or One Drive.

5.1.4 External Search Provider

We previously mentioned that storing complete SCDs on chain is expensive and thus we only store a small amount of metadata about them. Searches via the Registry Contract are consequently limited by that lack of information. Our proposed solution for this problem are External Search Providers that index SCDs that have been published via the Registry Contract. Hence, they subscribe to events emitted by the Registry Contract via the IOnChange interface and then they fetch the actual changed SCDs from the corresponding External SCD Storage. After that, they index that SCD. To search through the SCDs, clients have to use the ISearch interface.

5.2 Interfaces

This section describes the interfaces that can be seen in the Figure 5.1, namely the *ISearch*, the *IRetrieve*, the *IMetadata* and the *IOnChange* interfaces.

5.2.1 ISearch

ISearch is the interface of the External Search Provider. We require the External Search Provider to be addressable via a URL. It should offer one endpoint that takes in two query parameters. Those are *query* and *onlyId*. The former is the query that is passed to the search provider. If the client only sets this parameter, then the results are the complete SCDs with their corresponding metadata ids. However, if the client sets the onlyId parameter to "true", the server returns only the ids. The URL `http://localhost:3000?onlyId=true&query=quaCoin` serves as an example that returns the SCD-metadata ids of the corresponding SCDs which contain the word "quaCoin" somewhere (i. e., it does a full-text search).

5.2.2 IRetrieve

The IRetrieve interface is used to retrieve SCDs from an External SCD Storage. Hence, it is highly dependent on the used storage which makes it more of a "meta interface". The following two subsections describe the real ones.

HTTP server

This interface works by serving files over HTTP. The SCD JSON is located via an URL that is stored inside the SCD-metadata. Furthermore, the form of this URL is irrelevant as long as it points directly to the SCD.

Swarm

If Swarm is used, this API has additional functionality, besides retrieving the SCD. It is also used to upload SCDs to the Swarm network and buy the necessary postage batches.

5.2.3 IMetadata

Clients can use this smart contract interface to search for SCD-metadata, to register new metadata and to retrieve it. It consists of three functions that are called *query*, *store* and *retrieveById*. The query function is used to query for SCD-metadata and its signature looks like this:

```
function query(string memory _query) public view returns (SCDMetadataWithID[] memory);
```

The only parameter is the *_query* which we are going to describe further in Section 6.1. Furthermore, the results are returned as an array of *SCDMetadataWithID* which is also described in Section 6.1.

The next function is called *store* and is used to store new SCD-metadata on the blockchain. The signature looks like this:

```
function store(SCDMetadataIn memory _metadata) public;
```

It takes in an instance of *SCDMetadataIn*. That struct contains the actual metadata and we describe it in Section 6.1.

The last function which is *retrieveById*, retrieves SCD-metadata by its id. The signature looks like this:

```
function retrieveById(uint256 _id) public view returns (SCDMetadataWithID memory);
```

The input is the identifier of the SCD-metadata and it returns metadata as an instance of SCDMetadataWithID.

5.2.4 IOnChange

This interface is used to notify users about newly added SCDs. To do so, the Registry Contract emits an event called *ContractRegistered* and its signature looks like this:

```
event ContractRegistered(uint256 id);
```

It carries a value called *id* which represents the id of the newly registered SCD-metadata.

5.3 Interaction

In this section, we talk about the interaction a user needs to do to register SCDs and to search for them. First of all, how this works is highly dependent on the form of the chosen External SCD Storage. There are storages that require the user to upload the SCD before starting the actual registration process and others that upload the SCD during the registration. Hence, the beginning of the workflow is different for both provided SCD storages. For the HTTP server, users have to take note of the following. They need to upload the SCD file to that server in beforehand and enter the URL in the corresponding form in the frontend to fetch the file from that and continue the registration process. We did not implement that upload process, because depending on the underlying HTTP server, the possible upload processes might be varied. Consequently, we also did not model that here. The other SCD storage is the Swarm network. To use it, the user needs to have access to a Bee node that enables buying postage batches and uploading files to the network. Thus, both the debug and the normal API need to be accessible to the user. Buying the postage batches and uploading the file is then done from the Frontend during the registration process.

5.3.1 Registration

We now begin with the registration of SCDs. As we talked about in the previous section, there are two different approaches to register an SCD. The first involves a HTTP server that requires uploading the files to it before users can begin with the actual registration process. Figure 5.2 shows how the registration process works. The user begins with entering the URL of the SCD into a form in the Frontend which then fetches the file from the server and displays it, so the user can review it. Following that, the Frontend can extract the necessary metadata and sign the SCD. All of this is then also displayed to the user who then starts the upload process. The Frontend invokes the Registry Contract which stores the metadata on the blockchain. After that, the user gets to see that the registration was executed. If it was successful, the contract emits an event that signals that a new SCD was registered. This event is then picked up by External Search Providers which fetch the metadata from the contract and use that data to find out where the actual SCD can be found. Consequently, they also fetch the SCD. The fetched data is then processed by the External Search Provider.

The second approach utilizes the Swarm network as an External SCD Storage. This approach is shown in Figure 5.3. Users have to begin by either choosing a usable postage batch or by buying new ones. Both of those tasks can be performed in the Frontend. Following this, the user needs to select the SCD file and start the upload. The file is then uploaded to the Swarm network and the user receives the file reference which is used to access the file on the network. After receiving the reference, the frontend immediately fetches the SCD to display it. The remaining steps are the same as in the other approach, beginning with the extraction of metadata and the signing of the SCD.

5.3.2 Querying

We are now talking about the subject of querying the registry. Like before, there are two possible scenarios to do so. In one scenario the Registry Contract is queried, while in the other the External Search Provider is queried.

We begin with the scenario that queries the Registry Contract. The sequence diagram in Figure 5.4 visualizes the scenario. The user begins by typing the query in the search form of the frontend. After the user submits the form, the Frontend calls the query function of the Registry Contract and waits for the results. The contract searches SCDs that match the query. All matches are then sent back as a result. Upon receiving the results, the frontend displays them. Technically this process ends here but let's assume that the user now selects one of the SCDs for further inspection. This leads to the Frontend retrieving the SCD metadata from the Registry Contract based on the metadata id. Following that, the contract sends the metadata back as a response. The Frontend then uses the location from the metadata to retrieve that SCD from the External SCD Storage. After that, the SCD is displayed to the user. Then the signature is verified in addition to comparing the public key that is stored in the SCD-metadata and the public key from the SCD itself. The user is then informed about the results of those checks.

Now follows the second scenario which is illustrated in Figure 5.5. Instead of querying the contract, the frontend first queries the External Search Provider after receiving the query string. The External Search Provider returns the metadata ids (see Table 5.1) of all matching SCDs. Those ids are then

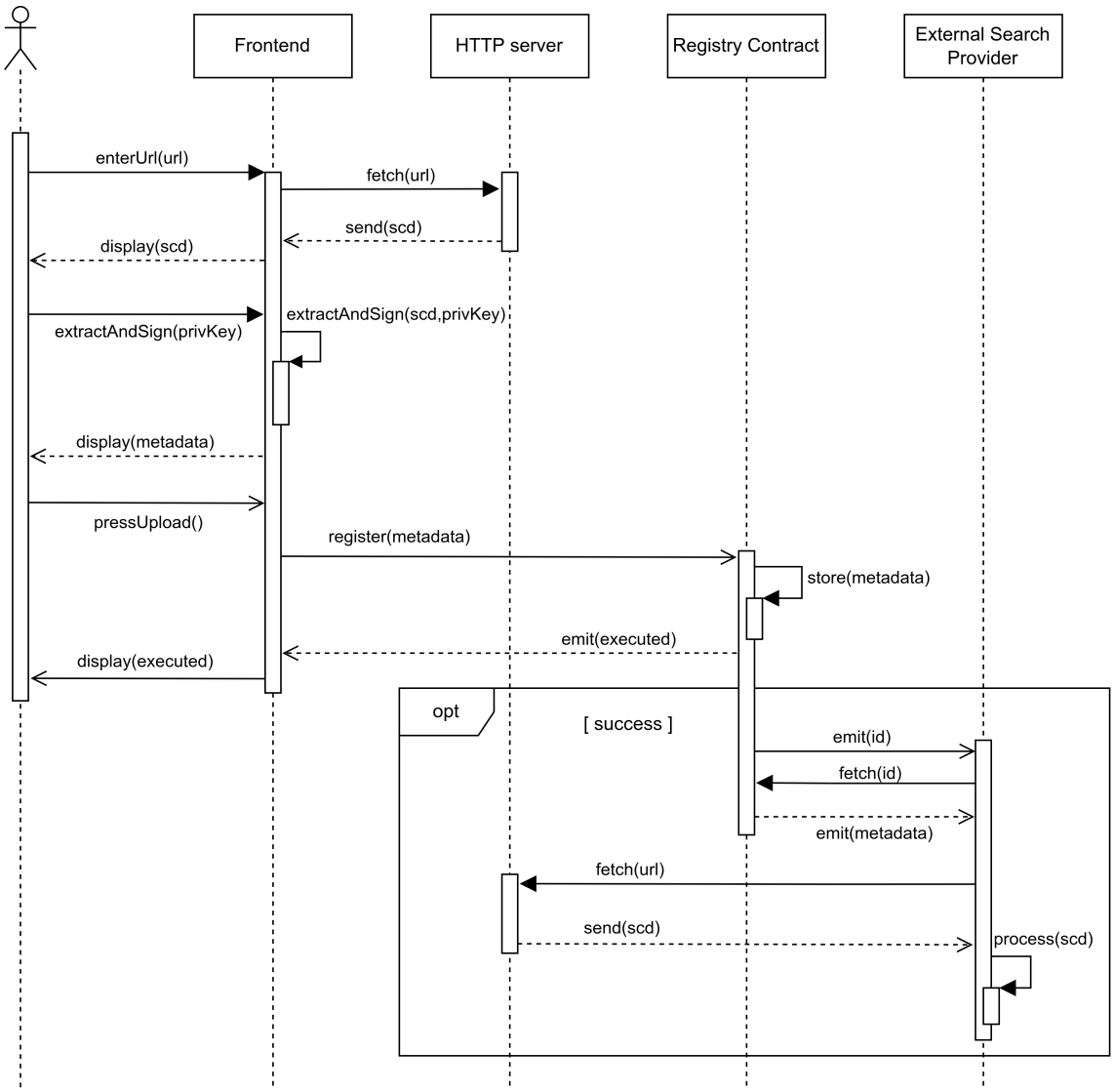

Figure 5.2: This figure shows the registration process for SCDs assuming the user chose the HTTP server as an External SCD storage.

used by the Frontend to retrieve the actual metadata from the Registry Contract. The retrieved metadata is then displayed to the user. All the remaining steps are the same as in the previous scenario. Therefore, we are not going to repeat those.

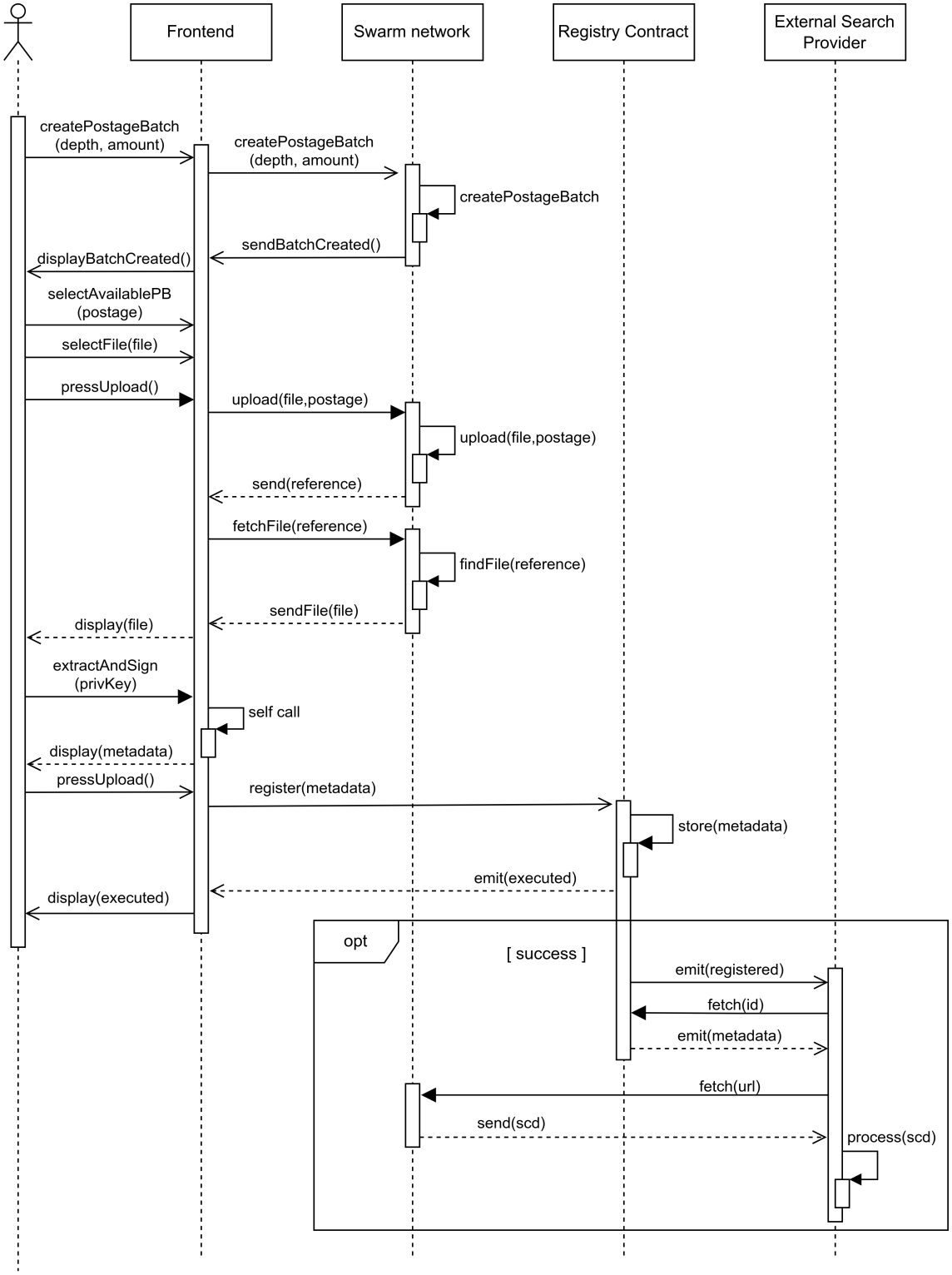

Figure 5.3: This figure shows the registration process for SCDs assuming the user chose the Swarm network as an External SCD storage.

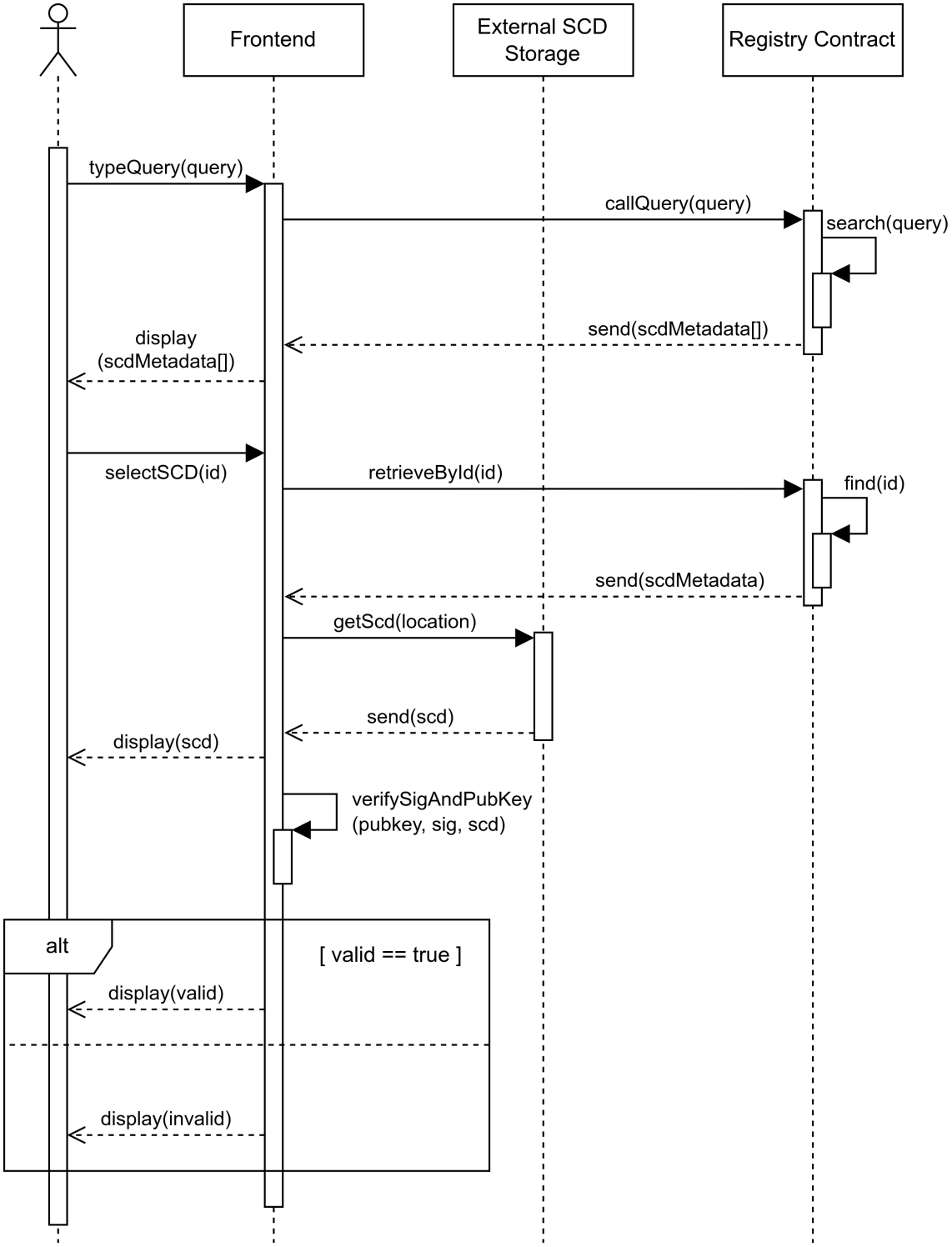

Figure 5.4: This figure shows the querying process for the Registry Contract.

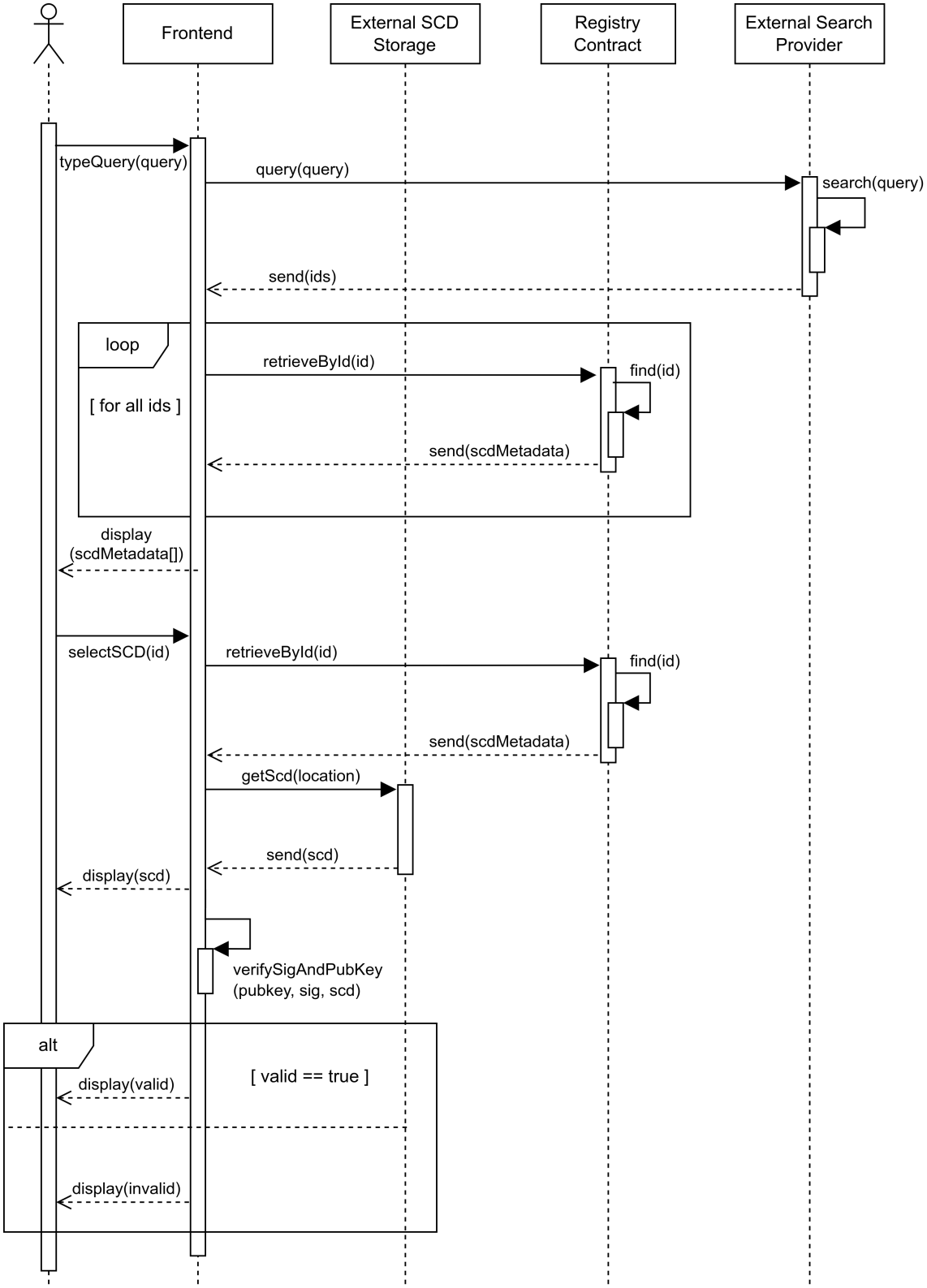

Figure 5.5: This figure shows the querying process for the External Search Provider.

6 Implementation

In this chapter, we talk about the actual system implementation. To be more precise, we describe how the individual components were implemented, how they work and which technologies were used. We begin with the centerpiece of the registry: the Registry Contract. The second section is about the External SCD Storage. Following that, comes a section about the External Search Provider. The next section is about the Frontend and we conclude with a short section about the deployment of a showcase system.

6.1 Registry Contract

We decided to create the Registry Contract[1] for an EVM blockchain and designed it for a deployment on Ethereum. For contract development, we therefore decided on using Solidity[2] as our language of choice. Solidity is a turing complete programming language, designed to write smart contracts for the EVM. It enjoys wide adoption in the Ethereum community while other languages for the EVM, like Vyper do not seem to be as mature. Therefore, we think it is well suited for that task. Furthermore, we chose *Hardhat*[3] as a development environment. Hardhat makes testing easier by providing the local *Hardhat Network*, which is an Ethereum network simulator designed for development and it is well integrated into the hardhat environment.

In addition to the Registry Contract, we implemented two more contracts called *Regex* and *Util*. Figure 6.1 shows their relationship in addition to more information about them. We can see that the Registry Contract needs the Regex contract which in turn needs the Util contract.

The Regex contract is necessary for the *query* function of the Registry Contract which takes in a query string that follows a simple query language. We therefore decided to parse that query with a regex. Unfortunately, Solidity has no utilities for that task. As a consequence, we generated the Solidity code for a corresponding state machine with the *solregex*[4] npm package and wrote the missing utility functions ourselves. Our code generation approach is faster than manually coding the entire parser, because it only requires creating a regex. Future enhancements of the language can consequently also be performed more easily.

The query language consists of space separated key-value pairs of the following form:

(6.1) KEY = 'VALUE'

[1] https://github.com/TIHBS/scd-registry-contract
[2] https://soliditylang.org/
[3] https://hardhat.org/
[4] https://www.npmjs.com/package/solregex

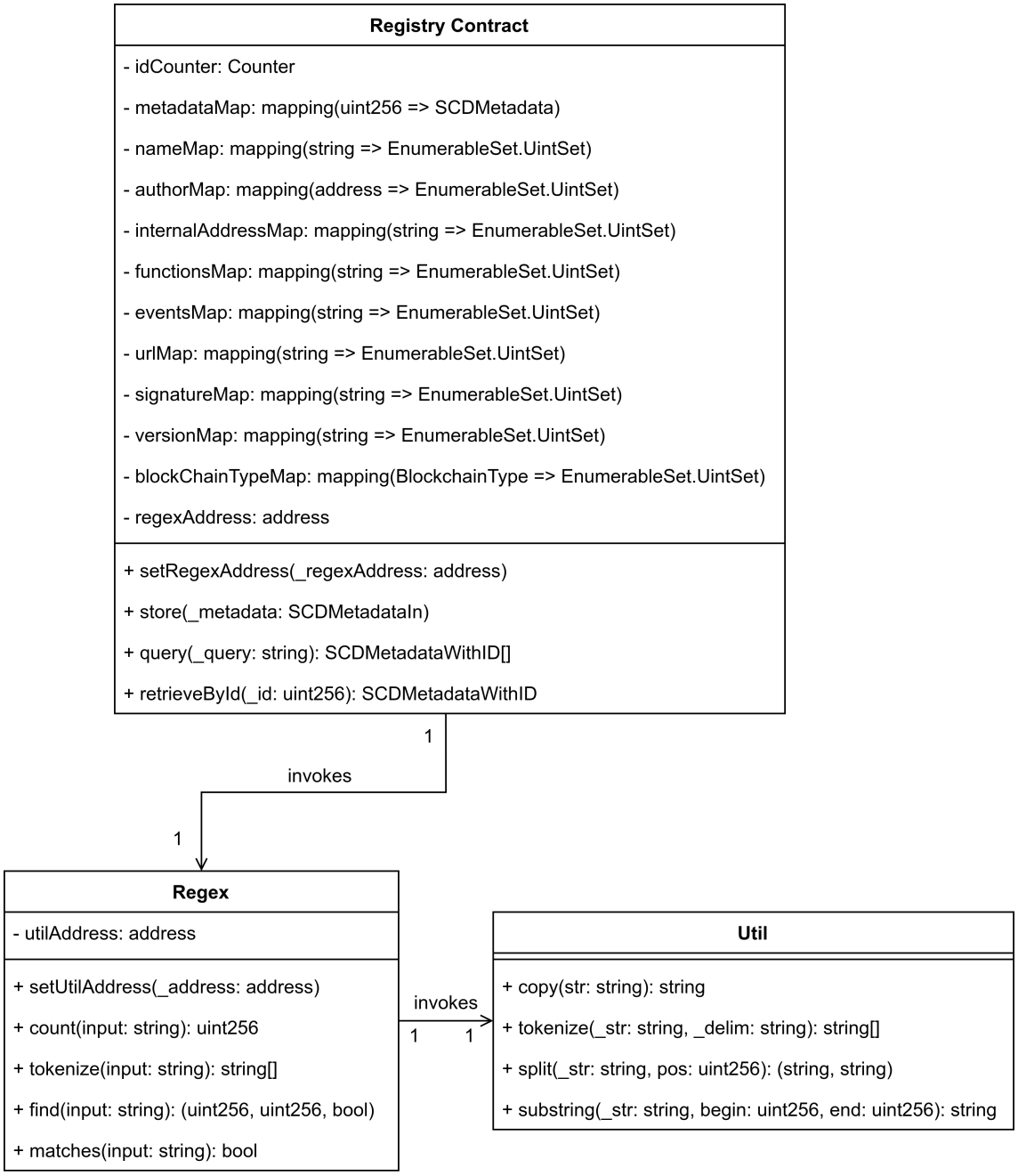

Figure 6.1: This class diagram shows the Registry Contract and the two other contracts it depends on. They are called *Regex* and *Util*.

To illustrate that further, here is an example query:

(6.2) Name = 'funnyAnimalCoin' Version = 'v1.2' BlockchainType = 'Ethereum'

This query would match SCD-metadata items that contain the name "funnyAnimalCoin", the version "v1.2" and the blockchain type "Ethereum". Therefore, we assume an implicit *AND* operator between the key-value pairs. We enclose the value in two ' to allow values with spaces which can be necessary for URLs if they are not percent encoded. For example, the URL `http://localhost:5670/Hallo Leute/Gurke.json` is a valid URL and can therefore be used. In percent encoding, it look like this `http://localhost:5670/Hallo%20Leute/Gurke.json`. We now come to the key. It has to have one of the following values:

- Name
- Author
- InternalAddress
- Url
- Signature
- Version
- Function
- Event
- BlockchainType

Each of them identifies one of the corresponding metadata fields which we described back in Table 5.1. The results of a query are structs with the name *SCDMetadataWithID* and they can be viewed in Listing 2. The *id* field contains the unique identifier, the contract assigns to this metadata item. Therefore, it can be used to fetch a specific metadata item with the function *retrieveById* (i. e., it is the id, defined in Table 5.1).

Coming back to another important function of the Registry Contract: the *store* function. It is used to register and thus store SCD-metadata in the contract. To do so, it takes in a struct which can be seen in the code snippet in Listing 3. The structs name is *SCDMetadataIn* and the author's public key is missing from it. The key is taken from the transaction itself to avoid users providing a wrong key. It also emits an event called *ContractRegistered* to notify potential External Search Providers that a new SCD exists and carries the corresponding metadata id. In Section 6.1.1 we go into more depth about storage and searching.

In our prototype, all of those contracts are deployed on a *Ganache*[5] blockchain simulator, but they can easily be deployed to a private *geth*[6] network or even a live network, for which they were designed. This is facilitated by the *hardhat-deploy*[7] plugin for Hardhat which allows writing

[5] https://trufflesuite.com/ganache/
[6] https://geth.ethereum.org/
[7] https://github.com/wighawag/hardhat-deploy/

Listing 2 This code snippet shows the SCDMetadata struct and the SCDMetadataWithID struct. The former is part of the *metadataMap* mapping, while the latter is used as a return value for the *query* method and the *retrieveById* method of the Registry Contract.

```
enum BlockchainType {
    BITCOIN,
    ETHEREUM,
    FABRIC,
    NEO
}

struct SCDMetadata {
    string name;
    address author;
    string internalAddress;
    string url;
    string signature;
    string version;
    string[] functions;
    string[] events;
    bool isValid;
    BlockchainType blockChainType;
}

struct SCDMetadataWithID {
    uint256 id;
    SCDMetadata metadata;
}
```

Listing 3 This code snippet shows the *SCDMetadataIn* struct. It is the input to the store method of the Registry Contract.

```
enum BlockchainType {
    BITCOIN,
    ETHEREUM,
    FABRIC,
    NEO
}

struct SCDMetadataIn {
    string name;
    string internalAddress;
    string url;
    string signature;
    string version;
    string[] functions;
    string[] events;
    BlockchainType blockChainType;
}
```

Listing 4 Here, the relevant mappings that are used to store *SCDMetadata* in the Registry Contract can be seen. The UintSet is a set that stores values of the type *uint256* [Opec]. SCDMetadata and the BlockchainType are defined in Listing 2.

```
mapping(uint256 => SCDMetadata) private metadataMap;

mapping(string => EnumerableSet.UintSet) private nameMap;
mapping(address => EnumerableSet.UintSet) private authorMap;
mapping(string => EnumerableSet.UintSet) private internalAddressMap;
mapping(string => EnumerableSet.UintSet) private functionsMap;
mapping(string => EnumerableSet.UintSet) private eventsMap;
mapping(string => EnumerableSet.UintSet) private urlMap;
mapping(string => EnumerableSet.UintSet) private signatureMap;
mapping(string => EnumerableSet.UintSet) private versionMap;
mapping(BlockchainType => EnumerableSet.UintSet) private blockChainTypeMap;
```

deployment scripts. To make interactions easier, we generated Typescript method stubs for the smart contract with *TypeChain*[8]. They can be found on GitHub [9]. Using them requires the npm package *ethers*[10].

6.1.1 Searching and storing

We are now going into more depth on how the Registry Contract's search and storage algorithms work. To make understanding easier, we are going to start with the storage part. The Listing 4 shows a code snippet that defines the mappings that are relevant for storage. They are the same that were already visible in Figure 6.1 in the Registry Contract. The most important mapping is the *metadataMap*. It maps the metadata id to the actual metadata which takes the form of a *SCDMetadata* struct. The remaining mappings correspond to the fields of the SCDMetadata struct. Thus, the keys of the mappings are the values stored in the members of the SCDMetadata while the values of the mappings are id sets of the SCDMetadata instances that contain that key as a value. We use those mappings as a simple "index". This is necessary to implement a fast retrieval algorithm, because otherwise all stored SCD-metadata instances need to be examined.

We now continue by talking about the storage algorithm. The relevant code snippet can be seen in Listing 5. The first step is to create a SCDMetadata instance from the *SCDMetadataIn* parameter named *_metadata* (1). We give this created instance the name *toStore*. During that process, the author's public key is taken from the transaction and also stored in toStore (2). The SCDMetadata instance is then stored in the metadataMap with the current value of the *idCounter* as the key (3) (i. e., this is the metadata id). Furthermore, the values stored in toStore are used as keys to find the map that should store the metadata id (4). Events and the functions are special cases. For them, we iterate over all events and functions to use each of them individually as a key (5). As an example, consider the following example: A SCDMetadata instance contains the functions "func1"

[8] https://www.npmjs.com/package/typechain/
[9] https://github.com/TIHBS/scd-registry-common/tree/master/src/wrappers
[10] https://www.npmjs.com/package/ethers

Listing 5 This listing shows the source code of the *store* function of the Registry Contract.

```
Counters.Counter private idCounter;

function store(SCDMetadataIn memory _metadata) public {
    // (1) The _metadata is used to create a SCDMetadata instance
    SCDMetadata memory toStore = SCDMetadata({
        name: _metadata.name,
        // (2) The authors key is taken from the transaction
        author: msg.sender,
        internalAddress: _metadata.internalAddress,
        url: _metadata.url,
        signature: _metadata.signature,
        version: _metadata.version,
        blockChainType: _metadata.blockChainType,
        functions: _metadata.functions,
        events: _metadata.events,
        isValid: true
    });

    // (3) The instance is stored in the metadataMap with the current value of the counter
    metadataMap[idCounter.current()] = toStore;

    // (4) The metadata id is mapped to the values from toStore by adding them to the corresponding set
    nameMap[toStore.name].add(idCounter.current());
    authorMap[toStore.author].add(idCounter.current());
    internalAddressMap[toStore.internalAddress].add(idCounter.current());
    urlMap[toStore.url].add(idCounter.current());
    signatureMap[toStore.signature].add(idCounter.current());
    versionMap[toStore.version].add(idCounter.current());
    blockChainTypeMap[toStore.blockChainType].add(idCounter.current());

    // (5) The same is done for every function and event from toStore
    addMultipleKeysForOneValue(functionsMap,toStore.functions,idCounter.current());
    addMultipleKeysForOneValue(eventsMap, toStore.events, idCounter.current());

    // (6) The same is done for every function and event from toStore
    emit ContractRegistered(idCounter.current());

    // (7) The counter is incremented to generate the metadata id for the next metadata instance that is
↪   going to be stored
    idCounter.increment();
}
```

and “func2” and has the metadata id 4242. This leads to the two mappings below being stored in the functionsMap:

$$\text{func1} \rightarrow \{\ldots, 4242, \ldots\}$$
$$\text{func2} \rightarrow \{\ldots, 4242, \ldots\}$$

After that, the ContractRegistered event is emitted with the metadata id of toStore (6). Finally, the idCounter is incremented (7).

Consequently, if a user wants to fetch all metadata items that contain the name “quaCoin”, they have to fetch the set that is stored behind the key “quaCoin” from the nameMap and then use the ids from that set to get the actual metadata from the metadataMap. Additionally, if the user wants those retrieved metadata instances to also contain a function called “add”, they would have to get the set of ids that has “add” as its key from the functionsMap and intersect it with the name id set that was fetched previously. This resulting intersection is then used to fetch the actual metadata instances from the metadataMap. The query function implements that process. Its source code can be seen in Listing 6. The function starts by first checking if the *_query* is empty (1). Assuming it is not, the algorithm continues. The _query is then tokenized into the key value pairs, we described in Section 6.1 (2). If no pairs are obtained by doing so, an empty result is returned to the user (3). Otherwise, the key-value pair tokens are split into the key and the value and stored together in structs called *KeyValuePair* (4). Then the first metadata id set is fetched from the metadataMap and stored in the *resultArray* variable (5). Following that, the other sets are fetched based on the remaining KeyValuePairs (6,7) and intersected with the resultArray (8). Finally, the actual metadata instances are collected and returned to the client (9).

6.2 External SCD Storage

We previously said that we provide two External SCD Storages. This section talks about both of them.

6.2.1 HTTP Server

The HTTP Server[11] is an Express[12] server that serves SCDs via HTTP. All SCD JSON files are located in the public directory and can be accessed by appending the relative path from the public directory to the SCD JSON to the url (e. g., `http://localhost:49160/directories/insidePublic/contract.json`). This server is a stand-in for numerous possible storages. The only requirement they have is that they can make SCDs accessible via a URL. Hence, it can easily be exchanged with other webservers, like the Apache HTTP Server or even a cloud storage provider, like One Drive.

6.2.2 Swarm network

Since using the real network requires real BZZ, we decided on using a local test network which can easily be deployed with the *Bee factory*[13] project which is a cli-tool, developed by the Swarm team to deploy a local swarm network utilizing docker containers. Using the real network can be done by changing the relevant settings in the frontend and providing the necessary infrastructure (i. e., a real Swarm node to communicate with).

[11] https://github.com/TIHBS/scd-registry-http-storage
[12] https://expressjs.com/
[13] https://github.com/ethersphere/bee-factory

Listing 6 This listing shows the code of the *query* function of the Registry Contract.

```
function query(string memory _query)
    public
    view
    returns (SCDMetadataWithID[] memory)
{
    // (1) Checks if the query is empty
    require(_query.toSlice().len() > 0, "The query should not be empty!");

    // (2) Tokenizenizes the query into key-value pairs of the form KEY = 'VALUE'
    IRegex regex = IRegex(regexAddress);
    string[] memory keyValueStrings = regex.tokenize(_query);

    // (3) Retruns an empty result if there are no key-value pairs.
    if (keyValueStrings.length <= 0) {
        return new SCDMetadataWithID[](0);
    }

    // (4) Splits the tokens into real key value pairs.
    KeyValuePair[] memory keyValuePairs = new KeyValuePair[](
        eyValueStrings.length
    );
    for (uint256 i = 0; i < keyValueStrings.length; i++) {
        (string memory key, string memory value) = queryParamToValue(
            keyValueStrings[i]
        );
        keyValuePairs[i] = KeyValuePair(key, value);
    }

    // (5) Gets the id of set first key-value pair from the metadataMap and stores them in the result set.
    uint256[] memory resultArray = getSetForKey(keyValuePairs[0]).values();

    // (6) Iterates over the remaining key-value pairs ...
    for (uint256 i = 1; i < keyValuePairs.length; i++) {
        // (7) ... and fetches the corresponding metadata id sets from the metadataMap ...
        uint256[] memory current = getSetForKey(keyValuePairs[i]).values();
        // (8) ... and intersects them with the results. The intersection is then stored as the result set.
        resultArray = resultArray.intersection(current);
    }

    // (9) Fetches the actual metadata instances based on the result metadata ids
    return indicesToMetadata(resultArray);
}
```

6.3 External Search Provider

The External Search Provider[14] is an Express server that returns search results, based on a full-text search query and it also subscribes to the ContractRegistered event (see Section 5.2.4). This component subscribes to that event and fetches the SCD metadata with the piggybacked id from the Registry Contract. The metadata contains the location of the actual SCD which is then also fetched.

We did not want to reinvent the wheel for our search algorithm. Consequently, we chose to use an already existing service for that, called *Elasticsearch*[15]. We chose it for being a mature, well known project which supports full-text searches out of the box. Alternatives could be *Solr*[16] or simply, most other database system, since most support some sort of full-text search or it can be implemented on top of them. We need to highlight that Elasticsearch is a standalone component and that the External Search Provider only acts as middleware for it. Hence, we outline the interactions between both briefly. The External Search Provider fetches registered SCDs and passes them over a JSON-RPC interface to Elasticsearch which indexes the SCDs and stores them with their metadata ids. When the External Search Provider receives a query from a client, it just forwards it to Elasticsearch which then searches through its database for matching SCDs. It then takes the metadata ids from them and returns those ids as a search result to the External Search Provider. The latter just forwards them to the client as a response after receiving them.

6.4 Frontend

The Frontend[17] is a single page application built with Vue.js[18]. The framework decision was made arbitrarily, since most frontend frameworks can create single-page applications. Other possible frameworks include, but are not limited to Angular[19] or React[20]. It is obvious that hosting this website on a centralized server infrastructure is not an option, because we want to avoid one entity getting control over the system as much as possible. Fortunately, Swarm can not only be used as a file storage but also as a webserver [Swa21b]. Consequently, we decided on doing that. The following subsections concern themselves with the different views of the frontend and what users can do on them.

6.4.1 SCDs view

This view is the main page of the application and it can be seen in Figure 6.2. It lets users search for SCDs. At the top is a navigation bar that not only lets the user switch to the settings (1) page but also go to one of two registration pages (2). One is for registering SCDs that will be stored in the

[14] https://github.com/TIHBS/scd-registry-external-search-provider
[15] https://www.elastic.co
[16] https://solr.apache.org
[17] https://github.com/TIHBS/scd-registry-frontend
[18] https://vuejs.org
[19] https://angular.io
[20] https://reactjs.org

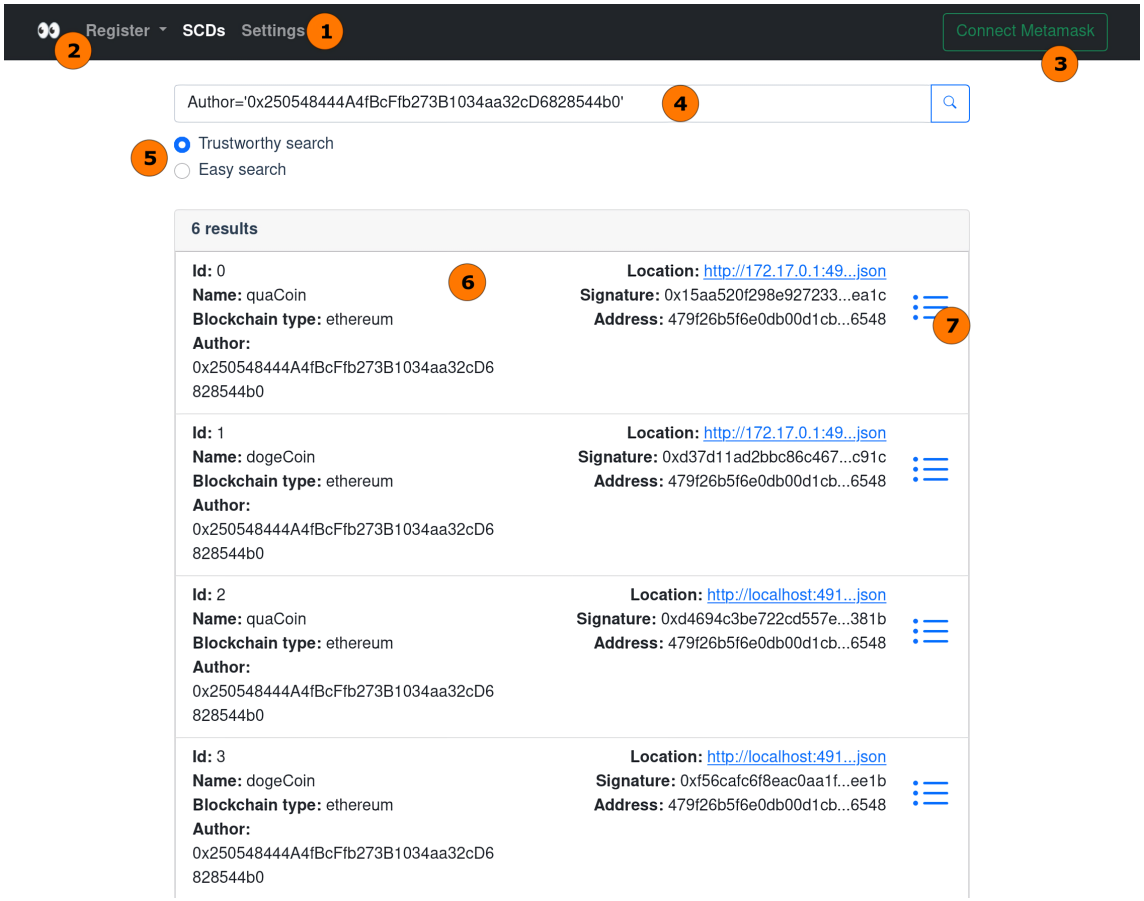

Figure 6.2: This screenshot shows the main page of the Frontend. Users can query here for SCDs.

Swarm network and the other one registers SCDs that can be located over HTTP. The right side of that bar shows the *Connect Metamask* button (3). Smart contract interactions require a wallet that is integrated into the browser. We chose Metamask[21] for that purpose, because it is the de facto standard for managing user access to EVM-based blockchains. Pressing that button lets the user connect it to the frontend. After the connection is established, the button disappears and the user's truncated public key is displayed to indicate the connection. Connecting is only necessary if the user wants to register SCDs. Just querying the registry does not require doing so.

The middle of the page shows the querying view. There is a text input (4) that lets the user enter the query and submit it. Below that are two radio buttons (5) which let the user switch between the two search modes. They are the *Trustworthy search* and the *Easy search*. The trustworthy search interfaces with the Registry Contract while the easy search invokes the External Search Provider. Under those are the query results (6). They show the most important elements of the metadata to distinguish the individual search results. Each result has a list symbol (7) at the right which when clicked takes the user to the corresponding detail page.

[21] https://metamask.io/

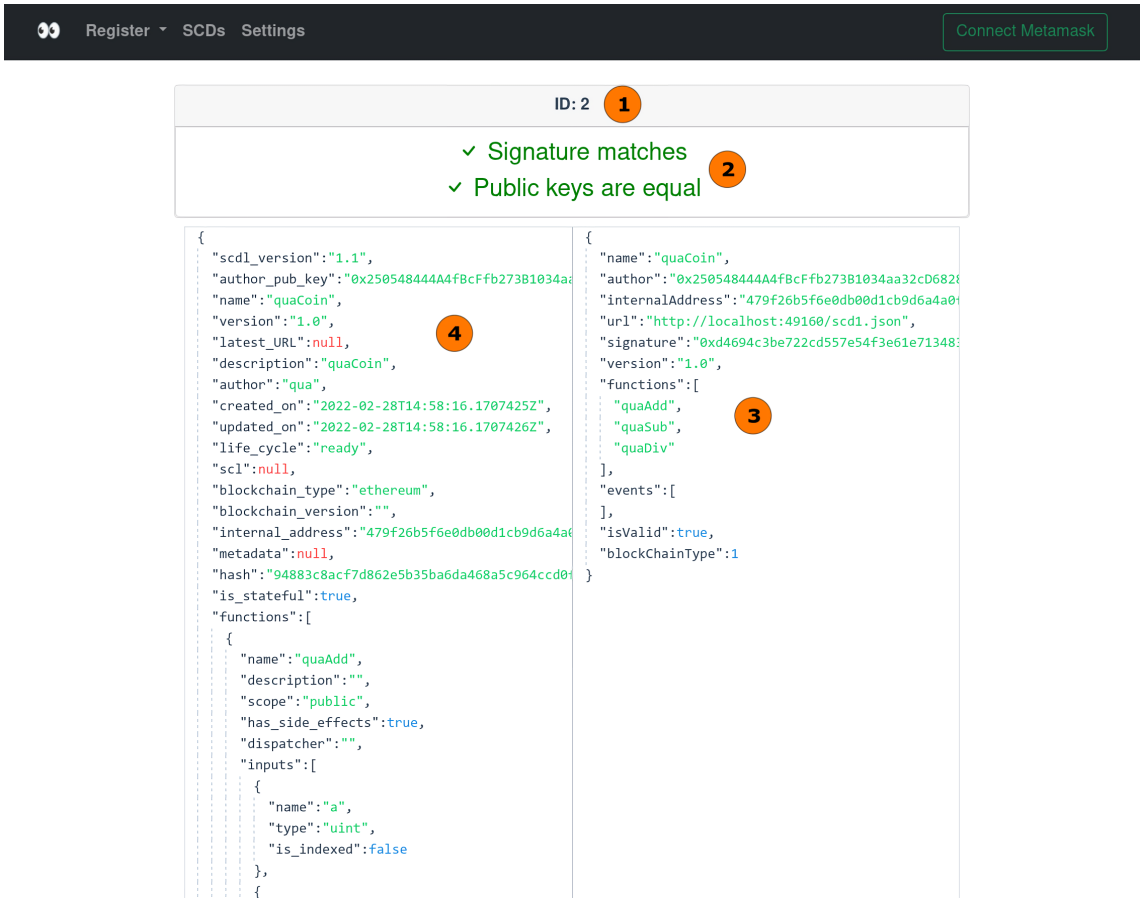

Figure 6.3: This screenshot shows the detail view of a SCD. On the left is the actual SCD while on the right the SCD-metadata can be seen.

6.4.2 Detail view

Figure 6.3 shows the detail view. The top of it shows the id (1) of the SCD-metadata that is stored in the Registry Contract. Below that can be seen if the signature of the SCD is correct and if the public key, stored in the SCD-metadata and the public key, stored in the SCD are the same (2). This turns red if the checks fail. On the right the stored metadata (3) can be seen and on the left (4) is the actual SCD.

6.4.3 Settings page

In Figure 6.4 users can see the settings page. It offers the user a text input for what we call the *Networkish* (1). We use that term for the URL of a blockchain network node or a network id. Furthermore, the Registry Contract's address (2), the URL of the External Search Provider (3), the URL of the Swarm Debug API (4) and finally the URL of the Swarm API (5) can be set here. Those settings are relevant for various tasks that the Frontend can perform.

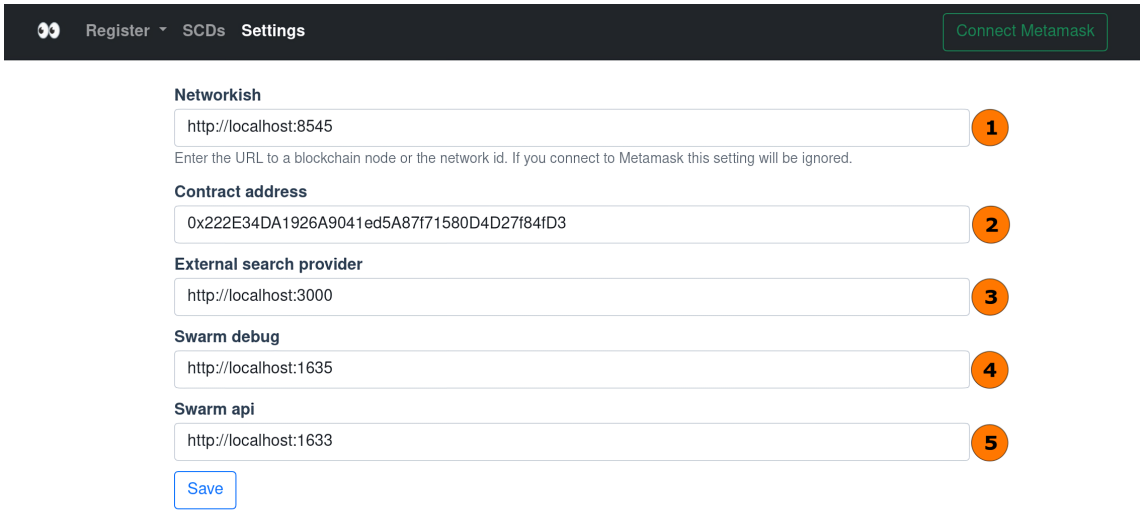

Figure 6.4: This screenshot shows the settings page. Here, users can set the appropriate connection information to the necessary external services.

6.4.4 Register

We now come to the registration of SCDs. To do that, users click on "Register" in the navigation bar. Then they can choose the desired registration approach in a dropdown menu. They will be described in the following sections.

HTTP server

We begin with registering SCDs that can be accessed via HTTP. The corresponding page can be seen in Figure 6.5. At the top, a text input (1) can be seen. Here users have to enter the URL that leads to the SCD and click on "Fetch" (2). The frontend then fetches the file from its source. On the left the files content can be seen (3). After the file has been fetched, the user needs to click on "Sign and transform" (4). This begins the process of signing the SCD with the private key of the current wallet and extracting the necessary metadata (5) from the file. The result of that is on the right. Clicking on Store invokes the Registry Contract to store (6) the metadata.

Swarm

The page to register SCDs and store them in the swarm network can be seen in Figure 6.6. Uploading to Swarm requires so-called Postage batches. To create those, users have to set the amount (1) and depth (2) of them and click on the "Create" (3) button. Available Postage batches can be selected in the card below (4). Pressing upload (6) after selecting a file with the file picker (5) starts the upload. The file is fetched after this is finished, like in the previous registration approach. The rest of this process is also the same (8).

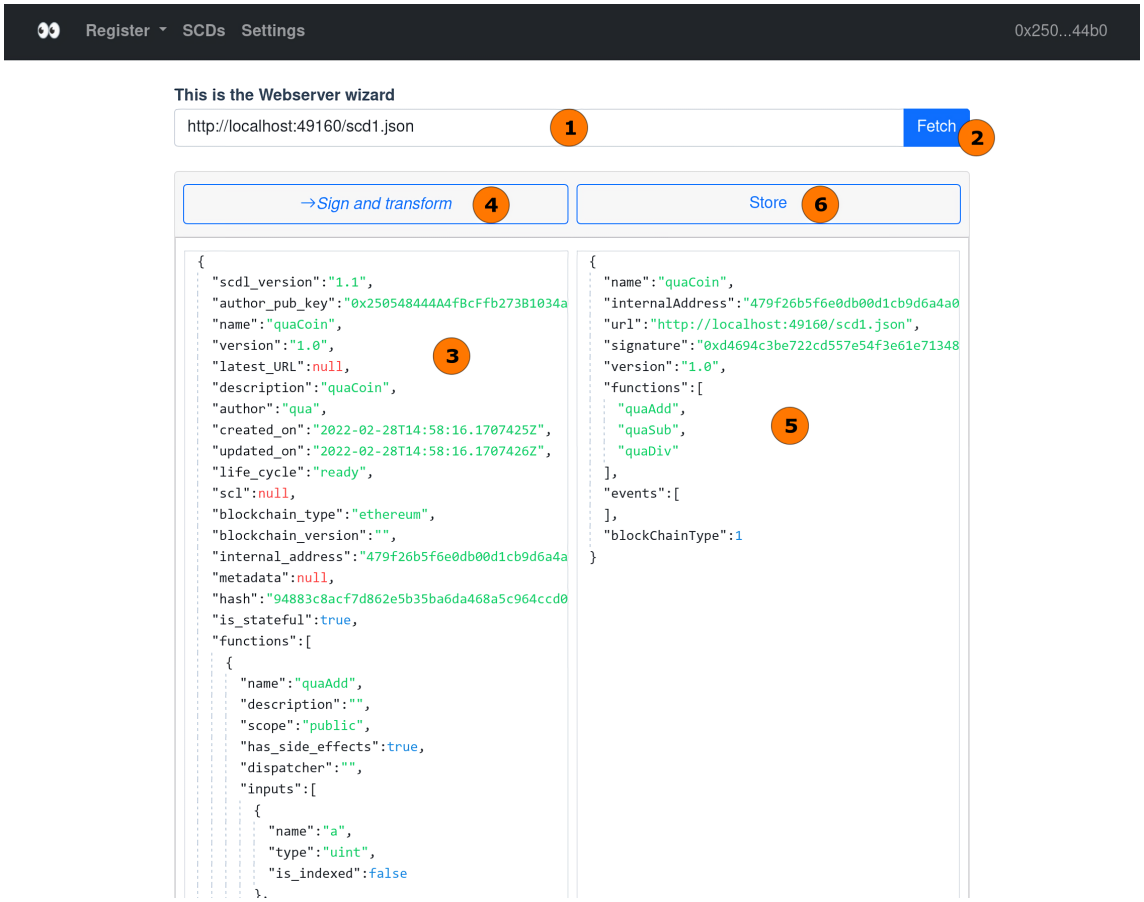

Figure 6.5: This screenshot shows the page on which users can register SCDs if they previously stored them in the HTTP server.

6.5 Deployment of the showcase system

This section talks about the deployment of our showcase system. Hence, we give information about the deployment process itself and the resulting system topology.

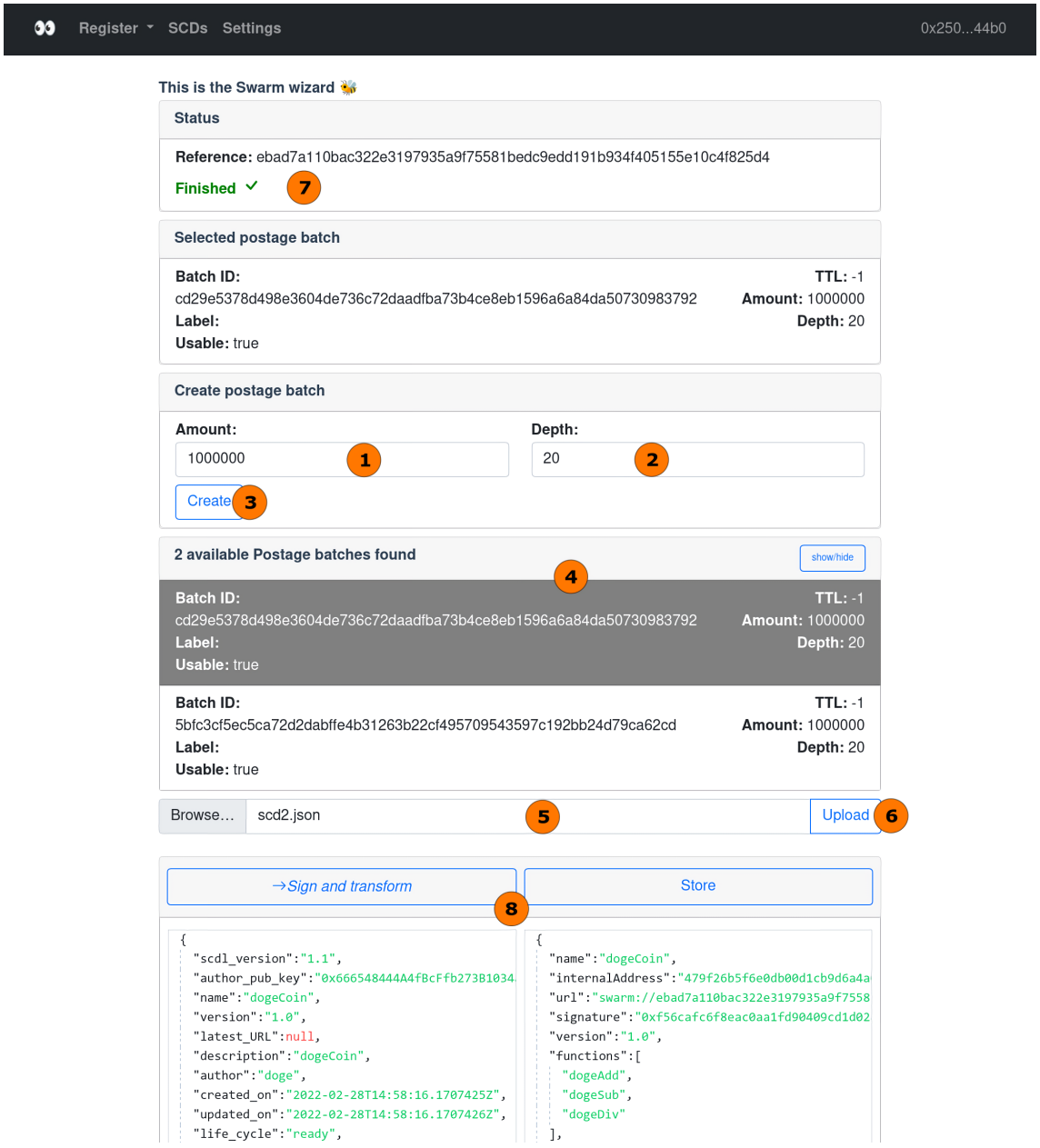

Figure 6.6: This screenshot shows the page on which users can register SCDs if they are supposed to be stored in the Swarm network.

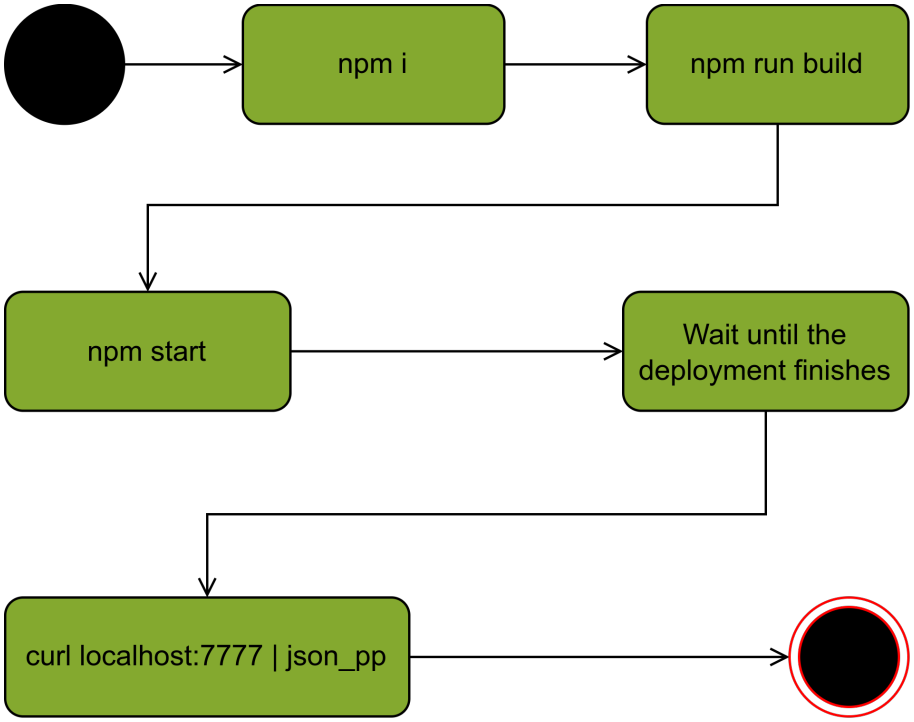

Figure 6.7: This figure shows the deployment process of our showcase system as a activity diagram. It mainly consists of shell commands that the deployer needs to execute.

6.5.1 Setup process

To make it easy for everyone to try out the registry, we provide a project that deploys a local example registry with all components. It can be found on GitHub[22]. The list of requirements is:

- npm
- nodejs
- docker
- docker-compose

In addition to the following description, we also visualize the deployment in Figure 6.7. We begin by installing the necessary npm dependencies:

```
npm i
```

The next step to booting up the registry is building the containers themselves. This can be done by running:

```
npm run build
```

From this point onwards, we recommend turning off the firewall to prevent it from blocking requests to the docker host. Finally, to start the system the following has to be run:

[22] https://github.com/TIHBS/scd-registry-meta

```
npm start
```

The startup process takes a few minutes. After everything has started, it is possible to retrieve the connection information by running:

```
curl localhost:7777 | json_pp
```

This results in a JSON object which looks like the following:


```
{
  "externalSearchProvider": "http://localhost:3000",
  "frontendUrl":
  ↪  "http://localhost:1633/bzz/fff8c8adfa7e57bd81a59d71f35ad3824424a07f32d2eb6c63b81e51683d3778/index.html",
  "networkish": "http://localhost:8545",
  "registryAddress": "0x222E34DA1926A9041ed5A87f71580D4D27f84fD3",
  "swarmAPi": "http://localhost:1633",
  "swarmDebug": "http://localhost:1635",
  "webserverStorage": "http://localhost:49160"
}
```


Consequently, the frontend can be reached via this URL:
`http://localhost:1633/bzz/4299db7c02e78b2c3799f9c7e2eb296acdafa9304bddf933084505411383ce21/index.html`.

6.5.2 Topology

The result of the previous setup process is represented as a TOSCA topology [RLNC19] in Figure 6.8. We can see that everything is hosted on the same Docker host. Thus, we make no assumptions about the underlying machine, other than it having docker installed and being x86 capable.

We will start with the local Swarm network. It consists of the bee-factory-workers, the bee-factory-queen and the Ganache container they are all connected to. The bee-factory-workers and the bee-factory-queen are connected for communication between the nodes. Our tool starts all of those by invoking the Bee Factory (see Section 6.2.2), which we mentioned previously.

The next piece is the Frontend which we deploy on the local Swarm network via the bee-factory-queen. The connection to the External Search Provider is necessary to query it for SCD-metadata ids. Additionally, the connection to the Registry Contract serves a similar purpose. It is used to query for SCD-metadata. Furthermore, the last connection to the External SCD Storage serves the purpose of fetching the actual SCDs.

The External Search Provider connects to the Elasticsearch container and the Registry Contract, since it proxies said Elasticsearch deployment and thus connects it to the rest of the system.

Lastly, comes the Registry Contract which is deployed on another Ganache container together with the Util and the Regex contract.

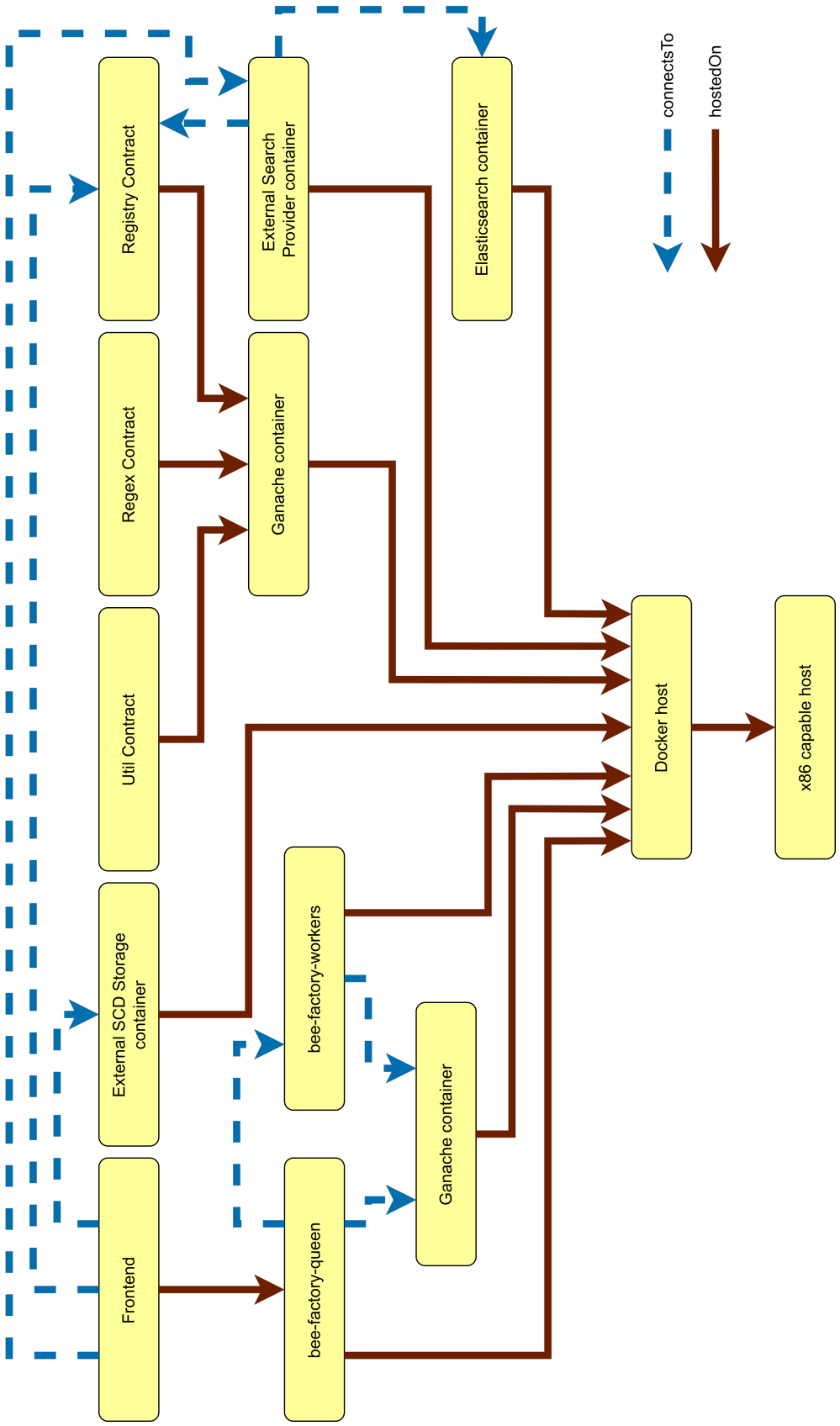

Figure 6.8: This figure shows the TOSCA topology [RLNC19] of the showcase system.

7 Evaluation

This chapter concerns itself with the evaluation of our implemented system. We are most concerned about correlations in the response time of the Registry Contract, because the exact time values are meaningless, since our system represents an ideal case. In addition to that, if it was deployed on a real network, those times would probably be higher due to increased network traffic and hardware utilization. Evaluating parts like the External Search Provider is out of the scope of this thesis, because it is merely a middleware for Elasticsearch. We are also disregarding the time it takes to retrieve the actual SCDs from the External SCD Storage, they are stored in, because this depends on the setup, and on the network speed. We also need to highlight that querying does not change the state of the Registry Contract. Consequently, it does not require gas which lets us neglect cost measurements. Following that, we are going to utilize a SCD data set we created to make the response time measurements. In the performance test, we are going to look into the Registry Contracts performance, depending on the number of search results retrieved and on the number of stored SCD metadata instances.

7.1 Creation of a SCD data set

As we previously mentioned, for the Evaluation we need a SCD data set. SCDs are currently not widely adopted, therefore such a dataset could not be found. Thus, we created that dataset ourselves instead by crawling the GitHub API[1]. Such a big crawling operation generally comes with problems. The first of those is that the GitHub API makes only the first 1000 results available for a given query. Consequently, it was necessary to change the query, so that it never produced more results than that number. This was achieved by querying for repositories that were created in a specific time frame iteratively, starting on January 1st 2016 until June 7th 2022. Smart contracts can be written in many languages depending on the underlying blockchain technology they are developed for. Those languages include, but are not limited to Solidity or general purpose programming languages like C/C++, Javascript or C#. The latter language group makes it hard to automatically decide if a file is a smart contract or not. Thus, we only considered repositories that contained Solidity files which we later collected. With that decision comes the obvious limitation that this data set only contains SCDs of solidity smart contracts. However, this is not a problem since SCDs themselves follow a standard format that does not change based on the source smart contract. Therefore, this suffices for our purposes. The following URL serves as an example query that we sent to the GitHub API: `https://api.github.com/search/repositories?per_page=100&page=2&q=solidity created:`

[1] `https://api.github.com`

`2021-08-08..2021-09-08 language:solidity`. The result of such queries are lists of repositories that contain Solidity files. We then downloaded all those repositories and collected the Solidity files into another directory. The scripts that automates this process can be found on GitHub[2].

This resulted in a set of 235916 Solidity files, which were then transformed with the *SmartContract-DescriptorsGenerator*[3] [Art20] in combination with another script[4]. Sadly, only 127766 files were transformable due to reasons that are out of scope of this work. The SCD set that was obtained had therefore a size of 127766 files which is still a huge number of SCDs. We calculated the average size of those files to be 7.168 kB. To us, this file size seemed small. Thus, we investigated further and found out that an unknown but significant number of SCDs had missing data that should have been present. The missing data ranged from functions and events to missing everything (i. e., all fields stored the value *null*). Therefore, we believe that incomplete files dragged down the file size. In addition to that, a lot of those files could be "Hello World" programs or other short tutorial contracts. All in all, our data set might not be an ideal data set to represent SCDs that describe useful contracts. However, since our experiment only relies on quantity and not quality, it should suffice for our purpose. The SCD data set can be found on GitHub[5].

7.2 Time measurements

This section talks about the evaluation of the Smart Contract based registry. As we previously said, we are only going to look at the response time of the Registry Contract during querying. Our independent variables are the amount of stored SCD-metadata instances and the number of retrieved results. The dependant variable is the time it takes to retrieve the results. One might ask why we exclude the length of the query from our evaluation. The reason for that lies in the fact that the contract iterates over each key-value pair in the query and therefore has at least a time complexity of $O(n)$ for the length of the query. Conclusively, we expect that the time it takes will increase with the length of the query linearly.

For our measurements, we used the dataset that we described in the previous section. The experiment was conducted in the following manner. We stored a total of 10000 SCD metadata instances in the contract in ten steps. Thus, we stored 1000 SCD-metadata instances in each step. Furthermore, the first step contained instances that had a name field containing the value "InitializeableImplementation", with a number from one to ten appended to it (e. g., "InitializeableImplementation3"). The number also specified how many instances with that name were added (i. e., InitializeableImplementation3 was added three times). We also prepared corresponding queries that each matched their specific metadata subset. They can be seen in Listing 7. Each query produced exactly as many results as the number behind the name implies, because that was exactly the amount of metadata instances with that name, contained in the set, with the first matching none. After each step, we ran all of those queries and measured the time it took to retrieve the results. The Figures 7.1 and 7.2 show the results as boxplots. In both figures, the y-axis shows the time the retrieval process took, while the x-axis shows the dependant variable. Figure 7.1 gives us an indication that there might be no

[2] https://github.com/TIHBS/solidity-file-scraper
[3] https://github.com/TIHBS/SmartContractDescriptorsGenerator
[4] https://github.com/TIHBS/invoke-scd-transform
[5] https://github.com/TIHBS/scds

Listing 7 This listing shows the queries we executed during the time measurement experiment. Each of them resulted in exactly the number of results as the number at the end of each of them. The only exception to this is the first one. It does not produce any results.

```
const queries = [
    "Name='kjakhgrlanjklfh3984jklnklasdfhigjöaklgj'", // The case that nothing was retrieved
    "Name='InitializeableImplementation1'",
    "Name='InitializeableImplementation2'",
    "Name='InitializeableImplementation3'",
    "Name='InitializeableImplementation4'",
    "Name='InitializeableImplementation5'",
    "Name='InitializeableImplementation6'",
    "Name='InitializeableImplementation7'",
    "Name='InitializeableImplementation8'",
    "Name='InitializeableImplementation9'",
    "Name='InitializeableImplementation10'",
];
```

correlation between the number of stored SCDs and the time it takes to retrieve them. Our reasoning for that is that we cannot observe a clear tendency of the response time to change if the number of stored SCDs increases. The opposite seems to be true for the time it took and the number of retrieved instances. Figure 7.2 indicates this, because the time it takes does increase with the number of retrieved SCDs. Calculating the Pearson correlation coefficient does not completely confirm the visual analysis. The results are displayed in Figure 7.3. There seems to be a weak negative correlation of -0.19 between the retrieval time and the number of stored SCD metadata instances. The relationship between the retrieval time and the number of results is a strong correlation of 0.67.

7.3 Discussion

As our experiment showed, the retrieval time increases with the number of retrieved results. One reason we found that might explain this is, that this increases the amount of data that needs to be processed by the EVM and passed to the client that invoked the smart contract. Another interesting observation we made was that the time it takes to retrieve SCDs does slightly decrease with the number of stored SCDs. On the one hand, assuming that this observation is not a random occurrence, this is a good thing, since it means that the response time of the Registry Contract improves if more SCD-metadata is stored, on the other hand we think that something like this is unusual, since more data typically means worse performance in the realm of data storage. We assume that the reason for this observation may be related to the querying process coming down to being a retrieval from a mapping, instead of running a more complex algorithm. This would at least explain, why there is no time increase with the amount of stored SCDs, but it does not explain why there is a time decrease.

Conclusively, we want to note the following two things. First, we can say that the registries overhead is not too big to be feasibly used when a large number of SCDs has been stored, since there exists only a weak negative correlation between the time it takes to retrieve SCDs and the amount of stored SCDs. Second, we expect there to be performance problems with regard to querying the

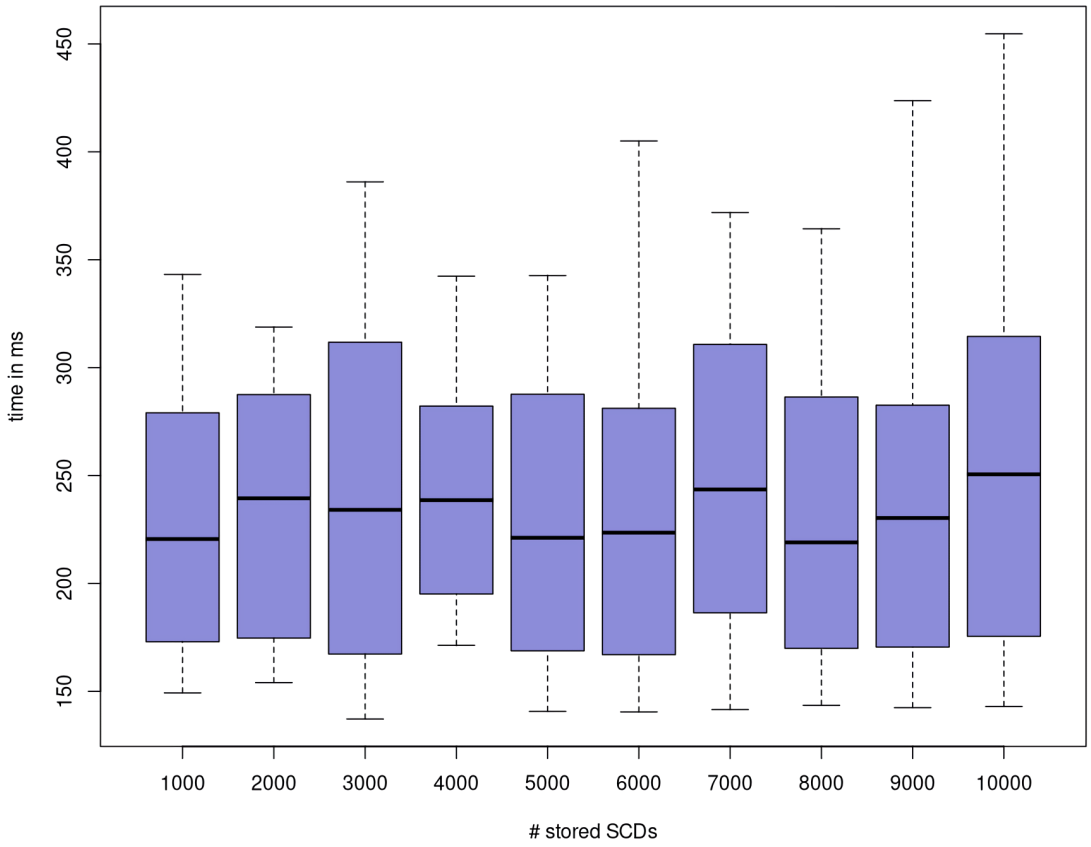

Figure 7.1: The boxplots in this figure show how the time it takes to query for SCD-metadata changes, depending on the number of already stored SCD-metadata instances. There appears to be no obvious correlation.

registry, because of the large correlation between the time it takes to retrieve SCDs and the number of retrieved SCDs. All in all, we still think that this registry implementation can be used and that it is a good starting point for future work.

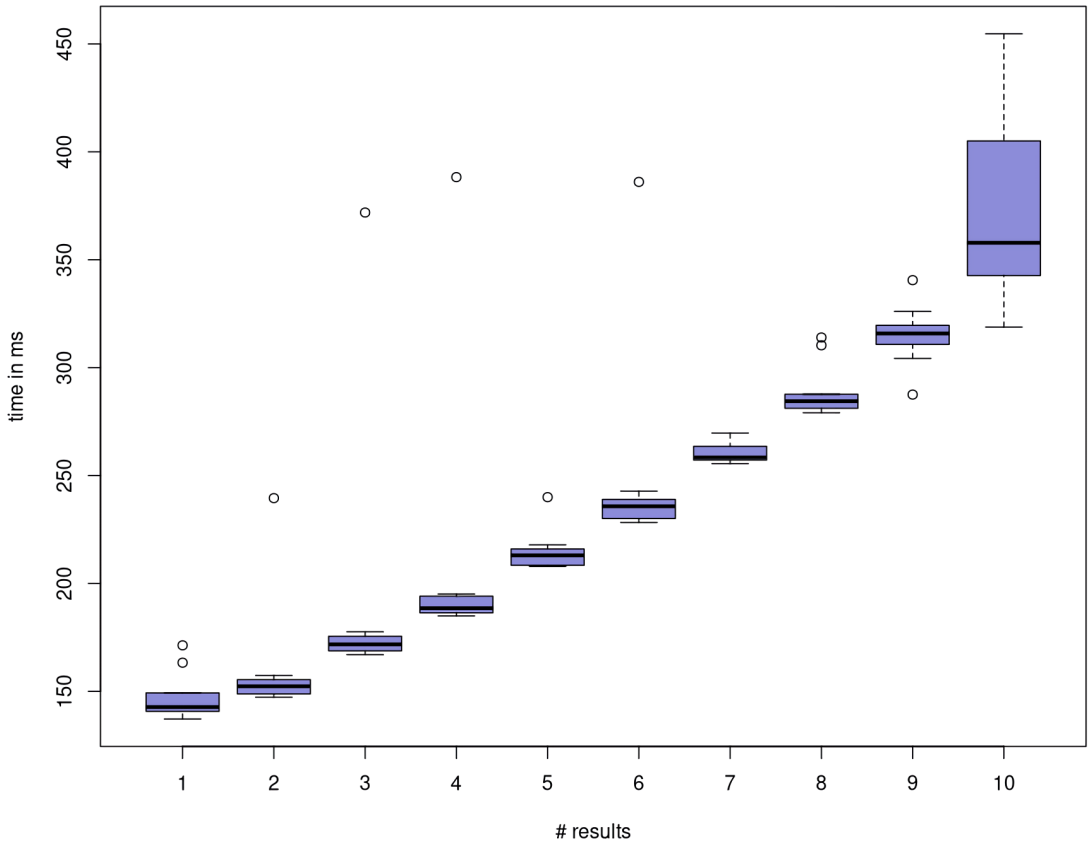

Figure 7.2: The boxplots in this figure show how the time it takes to query for SCD-metadata changes, depending on the number of retrieved SCD-metadata instances. There appears to be a strong correlation.

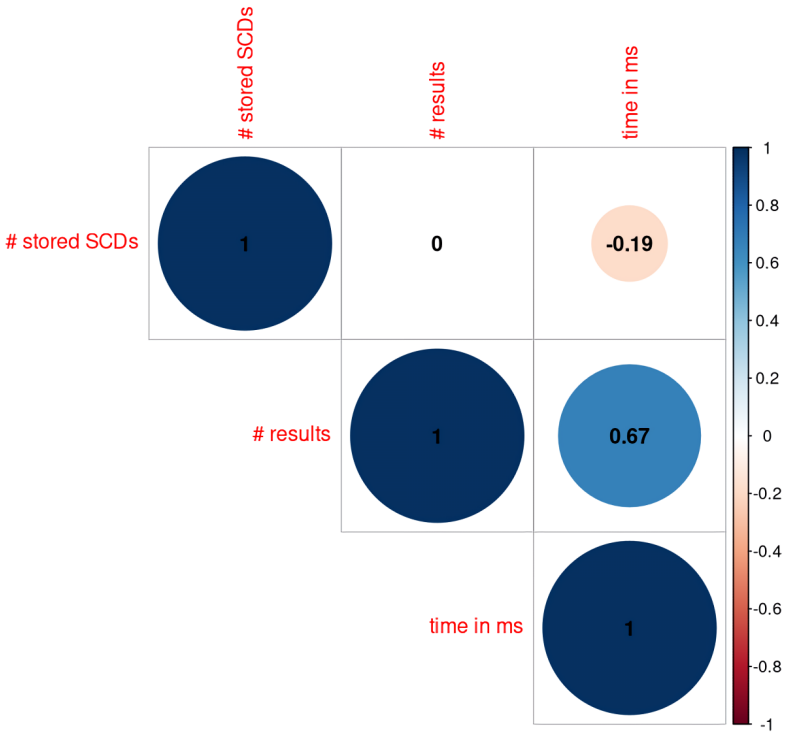

Figure 7.3: This figure shows the calculated Pearson correlation coefficients. The coefficient is -0.19 for the correlation between the time it takes to run the query and the number of already stored SCD-metadata instances. The coefficient for the time and the number of query results on the other hand is 0.67. Moreover, the values on the diagonal can be ignored.

8 Conclusion and Outlook

This thesis had the goal to create a censorship resistant SCD registry. We achieved that by conceptualizing a decentralized registry whose centerpiece is a EVM smart contract which we call the Registry Contract. However, this smart contract does not store those SCDs themselves but only a small amount of metadata about them. Consequently, they need to be stored somewhere else. We proposed two example External SCD Storages, but more are possible. One of those is a HTTP server, that uploaders have to host and maintain themselves. The other is the Swarm network, which is a decentralized storage solution. Furthermore, by only storing SCD-metadata on-chain, the contract only has limited search capabilities. Therefore, we also proposed the implementation of External Search Providers that index the registered SCDs by fetching them from the External SCD Storages and thus augmenting the discoverability of SCDs. All of this is integrated together using a Frontend hosted on Swarm. Therefore, it does not depend on a single party that hosts the Frontend. We described this system in more detail in Chapters 5 and 6. To get to this solution, we created two architectures in Chapter 4 and compared those. The first one is the architecture we just outlined, while the other one was in essence a frontend for the LBRY network. Unfortunately, we came to question the LBRY network, since it might be less decentralized than advertised. Thus we decided to implement the other solution.

We also evaluated the time it takes to query the Registry Contract based on the amount of stored SCDs and the amount of SCDs retrieved (see Chapter 7). During our evaluation we came to the conclusion that there are performance problems with regard to querying the registry but we still think that they are manageable and that the registry can still be used.

For the evaluation, we created a SCD dataset by querying the GitHub API for Solidity smart contracts and then automatically transforming them to SCDs (see Section 7.1). This dataset contains 127766 SCDs.

The conceptualized registry does in our opinion accomplish the goal of creating a censorship resistant SCD registry that we defined in Chapter 1.

Outlook

Even though we think that our approach accomplishes its goal, there are still things that can be improved. For starters, the number of currently supported different External SCD Storages is small. Therefore, we propose that more different storages should be supported. They might include but are not limited to things like IPFS, BitTorrent or Storj[1]. To do so, the frontend needs to be expanded, so it can be used to register and fetch them.

[1] https://www.storj.io/

Furthermore, the query language that the Registry Contract uses can be expanded. An explicit *AND* operator and an explicit *OR* operator can be added. In addition to that, it is possible to add the grouping operators *()*. Thus, more complicated queries can be created. Further enhancements to the language could be mechanisms to sort the results, for example alphabetically by the contract name or by the number of functions. This also would probably have no negative consequences for the gas costs, since querying does not change the smart contracts state. Currently, no fuzzy searches are possible. This means, results always match the query terms completely. Sadly, in the real world, this might be impractical. For example, smart contracts that have a similar purpose, might have similar, but not the exact same names. If we then imagine, a user tries to query for SCD-metadata instances that have a specific name, they will only get results that match that name exactly, but miss instances that have similar names. Thus, in our opinion it might be helpful to make fuzzy searches possible, by utilizing prefix trees, like patricia or radix trees. In contrast to the previous approaches, this one would increase the gas cost for uploaders, because the metadata would have to be stored appropriately into the used data structure. Thus, the costs and benefits need to be weight against each other.

As we previously noted, there are possible performance problems when the registry is queried depending on the number of retrieved SCDs. This comes down to the search algorithm which we think should be improved in future work.

Another enhancement can be achieved by using the SmartContractDescriptorsGenerator. It can be used to directly transform the smart contract to a SCD and thus offer a more seamless experience. For this to work, users would need to host the SmartContractDescriptorsGenerator themselves or use a trusted publicly available instance. In addition to that, the SmartContractDescriptorsGenerator could be enhanced by adding support for more languages in which smart contracts can be created.

Unfortunately, there is currently no functionality to edit previously registered SCD-metadata on chain. Furthermore, it is currently not possible to remove or at least invalidate them. Consequently, updating a SCD requires registering new SCD-metadata, because otherwise the signature that is stored in the SCD-metadata would be incorrect. Doing so is not without problems, because users might only know about the old SCD-metadata and have no way to find out if a new version already exists. Thus, future work should tackle this problem.

Moving on from concrete improvements to the tool itself to more general research. It is unclear how big the improvements in terms of gas for the storage of SCD-metadata in comparison to complete SCDs are. To perform such a cost analysis it is necessary to create a better SCD dataset, since ours is lacking as we previously noted in Section 7.1. Additionally, a SCD-metadata set would have to be created for the comparison. Utilizing those, different experiments could be performed to get a better grasp on the real cost improvements. Such experiments include, but are not limited to, comparing the gas costs for the storage of averagely sized SCDs to their metadata counterparts.

Furthermore, we noted an observation during our experiment in Chapter 7 that we could not explain. We refer to the decrease in the time it takes to query for SCD-metadata if more of them are already in the Registry Contract. Thus, we think that future work should try to provide an explanation for this phenomenon, since it strikes us as being something unusual.

Bibliography

[Art20] U. O. Artuvan. “Automatic Generation of Blockchain Smart Contract Descriptors and Client Application Skeletons”. Bachelor. University of Stuttgart - Institute of Architecture of Application Systems, Sept. 7, 2020 (cit. on p. 76).

[Bit21a] @Bittrex Global. *Twitter Post: Bittrex Global has no plans to delist LBRY Credits (LBC). Though Bittrex US will be delisting LBC, this will only impact U.S. customers and will not impact Bittrex Global customers' access to LBC.* Apr. 8, 2021. URL: https://twitter.com/BittrexGlobal/status/1380211037094432771 (cit. on p. 46).

[Bit21b] Bittrex Team. *LBC Market Removal 04/16/2021*. Apr. 16, 2021. URL: https://bittrex.zendesk.com/hc/en-us/articles/360058775712-LBC-Market-Removal-04-16-2021 (cit. on p. 46).

[But22] V. Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. Feb. 1, 2022. URL: https://ethereum.org/whitepaper/ (cit. on pp. 22, 23).

[CDK+02] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, S. Weerawarana. “Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI”. In: *IEEE Internet Computing* 6.2 (2002), pp. 86–93. DOI: 10.1109/4236.991449 (cit. on p. 33).

[Com] Commonwealth Scientific and Industrial Research Organisation. *Contract Registry*. URL: https://research.csiro.au/blockchainpatterns/general-patterns/contract-structural-patterns/contract-registry/ (cit. on p. 32).

[CW04] S. Chatterjee, J. Webber. *Developing Enterprise Web Services: An Architect's Guide*. Hewlett-Packard professional books. Prentice Hall PTR, 2004. ISBN: 9780131401600. URL: https://books.google.de/books?id=LEpPzQ5mRDoC (cit. on p. 33).

[ER18] D. Efanov, P. Roschin. “The All-Pervasiveness of the Blockchain Technology”. In: *Procedia Computer Science* 123 (Jan. 2018), pp. 116–121. DOI: 10.1016/j.procs.2018.01.019 (cit. on p. 21).

[Eth21] Etherscan.io. *What is Etherscan?* Aug. 2021. URL: https://info.etherscan.com/what-is-etherscan/ (cit. on p. 31).

[Eth22a] Ethereum Foundation. *The Merge*. June 30, 2022. URL: https://ethereum.org/en/upgrades/merge/ (cit. on p. 23).

[Eth22b] Etherscan.io. *How to Use Our Smart Contract Search Tool*. June 14, 2022. URL: https://info.etherscan.com/using-smart-contract-search-tool/ (cit. on p. 31).

[GK] A. Grintsvayg, J. Kauffman. *LBRY: A Decentralized Digital Content Marketplace*. Tech. rep. LBRY Inc. URL: https://lbry.tech/spec (cit. on pp. 26, 44).

[Goo21] Google. *New notifications when Drive content violates abuse program policies*. Dec. 14, 2021. URL: https://workspaceupdates.googleblog.com/2021/12/abuse-notification-emails-google-drive.html (cit. on p. 39).

[Hol12] T. Holman. *Microsoft responds to SkyDrive privacy concerns*. July 21, 2012. URL: https://www.neowin.net/news/microsoft-responds-to-skydrive-privacy-concerns/ (cit. on p. 39).

[Lam20] A. Lamparelli. "Unified Smart Contracts Integration: Proposal of a Service-Oriented Communication Infrastructure". MA thesis. ING - Scuola di Ingegneria Industriale e dell'Informazione, Apr. 29, 2020. URL: https://www.politesi.polimi.it/bitstream/10589/153100/1/Thesis___Unified_Smart_Contracts_Integration.pdf (cit. on pp. 19, 29, 30).

[LBRa] LBRY, Inc. *Case Guide and FAQ*. URL: https://helplbrysavecrypto.com/faq (cit. on p. 46).

[LBRb] LBRY, Inc. *How To Run Your Own Wallet Server*. URL: https://lbry.tech/resources/wallet-server (cit. on p. 28).

[LBR21] LBRY, Inc. *Buying and selling Credits on crypto exchanges*. Aug. 26, 2021. URL: https://lbry.com/faq/buy-sell-bittrex (cit. on p. 44).

[LBR22] LBRY, Inc. *Where can I buy and sell LBC?* May 20, 2022. URL: https://lbry.com/faq/exchanges (cit. on p. 44).

[LBW+21] Q. Lu, A. Binh Tran, I. Weber, H. O'Connor, P. Rimba, X. Xu, M. Staples, L. Zhu, R. Jeffery. "Integrated model-driven engineering of blockchain applications for business processes and asset management". In: *Software: Practice and Experience* 51.5 (2021), pp. 1059–1079. DOI: https://doi.org/10.1002/spe.2931. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2931. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2931 (cit. on p. 33).

[LFB+19] A. Lamparelli, G. Falazi, U. Breitenbücher, F. Daniel, F. Leymann. "Smart Contract Locator (SCL) and Smart Contract Description Language (SCDL)". In: *Service-Oriented Computing – ICSOC 2019 Workshops*. Vol. 11229. Lecture Notes in Computer Science. Springer, Oct. 2019, pp. 195–210. DOI: 10.1007/978-3-030-45989-5_16 (cit. on pp. 20, 29, 30).

[MM02] P. Maymounkov, D. Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *Peer-to-Peer Systems*. Ed. by P. Druschel, F. Kaashoek, A. Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65. ISBN: 978-3-540-45748-0 (cit. on pp. 25, 27).

[Nak09] S. Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep. 2009. URL: http://www.bitcoin.org/bitcoin.pdf (cit. on pp. 21, 22).

[Nam14] Namecoin. *Namecoin.org*. Apr. 20, 2014. URL: https://www.namecoin.org (visited on 01/30/2022) (cit. on p. 29).

[Opea] OpenZeppelin. *ERC 1155*. URL: https://docs.openzeppelin.com/contracts/3.x/api/token/erc1155#ERC1155 (cit. on pp. 37, 38, 87).

[Opeb] OpenZeppelin. *ERC1155.sol*. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC1155/ERC1155.sol (cit. on pp. 37, 38, 87).

[Opec] OpenZeppelin. *Utilities*. URL: https://docs.openzeppelin.com/contracts/3.x/api/utils#EnumerableSet (cit. on p. 61).

[PIC21] PIC/S. *PIC/S Guidance PI 041-1: Good practices for data management and integrity in regulated GMP/GDP environments*. July 2021 (cit. on p. 36).

[Pol19] Poloniex. *Delisted Assets*. Oct. 28, 2019. URL: https://support.poloniex.com/hc/en-us/articles/360040013653-Delisted-Assets (cit. on p. 46).

[Rav16] S. Raval. *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology*. First. O'Reilly Media, Inc., 2016. ISBN: 9781491924549 (cit. on pp. 21, 23).

[RLNC19] M. Rutkowski, C. Lauwers, C. Noshpitz, C. Curescu, eds. *TOSCA Simple Profile in YAML Version 1.3*. Specification Draft 01 / Public Review Draft 01. Latest version: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html. OASIS Committee, Apr. 25, 2019. URL: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/csprd01/TOSCA-Simple-Profile-YAML-v1.3-csprd01.html (cit. on pp. 72, 73).

[Swa21a] Swarm Foundation. *Keep Your Data Alive*. Dec. 24, 2021. URL: https://docs.ethswarm.org/docs/access-the-swarm/keep-your-data-alive (cit. on p. 26).

[Swa21b] Swarm Foundation. *Swarm: Storage and Communication Infrastructurefor a Self-Sovereign Digital Society*. Tech. rep. Swarm Foundation, June 13, 2021. URL: https://docs.ethswarm.org/swarm-whitepaper.pdf (cit. on pp. 25, 27, 28, 65).

[Syn17] G. Synek. *Google Docs is accidentally blocking access to documents*. Oct. 31, 2017. URL: https://www.techspot.com/news/71669-google-docs-accidentally-blocking-access-documents.html (cit. on p. 39).

[Sza96] N. Szabo. *Smart contracts: building blocks for digital markets*. 1996 (cit. on pp. 19, 23).

[Tru] True Names Ltd. *ENS Documentation*. URL: https://docs.ens.domains/ (cit. on p. 33).

[try22] trymeouteh. *Who is seeding all the videos on LBRY?* May 15, 2022. URL: https://www.reddit.com/r/lbry/comments/upyaoj/who_is_seeding_all_the_videos_on_lbry/ (cit. on p. 44).

[Uni21] Unicode, Inc. *Unicode Standard Annex #15: Unicode Normalization Forms*. Tech. rep. Aug. 27, 2021. URL: http://unicode.org/reports/tr15/ (cit. on p. 27).

[WGR05] K. Wehrle, S. Götz, S. Rieche. "Peer-to-Peer Systems and Applications". In: vol. 3485. Jan. 2005, pp. 79–93. ISBN: 978-3-540-29192-3. DOI: 10.1007/11530657_7 (cit. on p. 24).

[Wri22] A. Wright. *Numbers seem to be violating Google Drive's terms of service right now*. Jan. 26, 2022. URL: https://www.androidpolice.com/numbers-violate-google-drives-terms-of-service (cit. on p. 39).

[WSG13] M. Wachs, M. Schanzenbach, C. Grothoff. "On the Feasibility of a Censorship Resistant Decentralized Name System". In: Oct. 2013. DOI: 10.13140/2.1.2725.3765 (cit. on pp. 29, 31).

All links were last followed on July 14, 2022.

Appendix

ERC1155

[Opea; Opeb]

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.6.0) (token/ERC1155/ERC1155.sol)

pragma solidity ^0.8.0;

import "./IERC1155.sol";
import "./IERC1155Receiver.sol";
import "./extensions/IERC1155MetadataURI.sol";
import "../../utils/Address.sol";
import "../../utils/Context.sol";
import "../../utils/introspection/ERC165.sol";

/**
 * @dev Implementation of the basic standard multi-token.
 * See https://eips.ethereum.org/EIPS/eip-1155
 * Originally based on code by Enjin: https://github.com/enjin/erc-1155
 *
 * _Available since v3.1._
 */
contract ERC1155 is Context, ERC165, IERC1155, IERC1155MetadataURI {
    using Address for address;

    // Mapping from token ID to account balances
    mapping(uint256 => mapping(address => uint256)) private _balances;

    // Mapping from account to operator approvals
    mapping(address => mapping(address => bool)) private _operatorApprovals;

    // Used as the URI for all token types by relying on ID substitution, e.g.
↪   https://token-cdn-domain/{id}.json
    string private _uri;

    /**
     * @dev See {_setURI}.
     */
    constructor(string memory uri_) {
        _setURI(uri_);
    }

    /**
     * @dev See {IERC165-supportsInterface}.
     */
```

```
42      function supportsInterface(bytes4 interfaceId) public view virtual override(ERC165, IERC165) returns
↪   (bool) {
43          return
44              interfaceId == type(IERC1155).interfaceId ||
45              interfaceId == type(IERC1155MetadataURI).interfaceId ||
46              super.supportsInterface(interfaceId);
47      }
48
49      /**
50       * @dev See {IERC1155MetadataURI-uri}.
51       *
52       * This implementation returns the same URI for *all* token types. It relies
53       * on the token type ID substitution mechanism
54       * https://eips.ethereum.org/EIPS/eip-1155#metadata[defined in the EIP].
55       *
56       * Clients calling this function must replace the `\{id\}` substring with the
57       * actual token type ID.
58       */
59      function uri(uint256) public view virtual override returns (string memory) {
60          return _uri;
61      }
62
63      /**
64       * @dev See {IERC1155-balanceOf}.
65       *
66       * Requirements:
67       *
68       * - `account` cannot be the zero address.
69       */
70      function balanceOf(address account, uint256 id) public view virtual override returns (uint256) {
71          require(account != address(0), "ERC1155: address zero is not a valid owner");
72          return _balances[id][account];
73      }
74
75      /**
76       * @dev See {IERC1155-balanceOfBatch}.
77       *
78       * Requirements:
79       *
80       * - `accounts` and `ids` must have the same length.
81       */
82      function balanceOfBatch(address[] memory accounts, uint256[] memory ids)
83          public
84          view
85          virtual
86          override
87          returns (uint256[] memory)
88      {
89          require(accounts.length == ids.length, "ERC1155: accounts and ids length mismatch");
90
91          uint256[] memory batchBalances = new uint256[](accounts.length);
92
93          for (uint256 i = 0; i < accounts.length; ++i) {
94              batchBalances[i] = balanceOf(accounts[i], ids[i]);
95          }
96
97          return batchBalances;
```

```
98      }
99
100     /**
101      * @dev See {IERC1155-setApprovalForAll}.
102      */
103     function setApprovalForAll(address operator, bool approved) public virtual override {
104         _setApprovalForAll(_msgSender(), operator, approved);
105     }
106
107     /**
108      * @dev See {IERC1155-isApprovedForAll}.
109      */
110     function isApprovedForAll(address account, address operator) public view virtual override returns
    ↪ (bool) {
111         return _operatorApprovals[account][operator];
112     }
113
114     /**
115      * @dev See {IERC1155-safeTransferFrom}.
116      */
117     function safeTransferFrom(
118         address from,
119         address to,
120         uint256 id,
121         uint256 amount,
122         bytes memory data
123     ) public virtual override {
124         require(
125             from == _msgSender() || isApprovedForAll(from, _msgSender()),
126             "ERC1155: caller is not token owner nor approved"
127         );
128         _safeTransferFrom(from, to, id, amount, data);
129     }
130
131     /**
132      * @dev See {IERC1155-safeBatchTransferFrom}.
133      */
134     function safeBatchTransferFrom(
135         address from,
136         address to,
137         uint256[] memory ids,
138         uint256[] memory amounts,
139         bytes memory data
140     ) public virtual override {
141         require(
142             from == _msgSender() || isApprovedForAll(from, _msgSender()),
143             "ERC1155: caller is not token owner nor approved"
144         );
145         _safeBatchTransferFrom(from, to, ids, amounts, data);
146     }
147
148     /**
149      * @dev Transfers `amount` tokens of token type `id` from `from` to `to`.
150      *
151      * Emits a {TransferSingle} event.
152      *
153      * Requirements:
```

```
154         *
155         * - `to` cannot be the zero address.
156         * - `from` must have a balance of tokens of type `id` of at least `amount`.
157         * - If `to` refers to a smart contract, it must implement {IERC1155Receiver-onERC1155Received} and
    ↪  return the
158         * acceptance magic value.
159         */
160        function _safeTransferFrom(
161            address from,
162            address to,
163            uint256 id,
164            uint256 amount,
165            bytes memory data
166        ) internal virtual {
167            require(to != address(0), "ERC1155: transfer to the zero address");
168
169            address operator = _msgSender();
170            uint256[] memory ids = _asSingletonArray(id);
171            uint256[] memory amounts = _asSingletonArray(amount);
172
173            _beforeTokenTransfer(operator, from, to, ids, amounts, data);
174
175            uint256 fromBalance = _balances[id][from];
176            require(fromBalance >= amount, "ERC1155: insufficient balance for transfer");
177            unchecked {
178                _balances[id][from] = fromBalance - amount;
179            }
180            _balances[id][to] += amount;
181
182            emit TransferSingle(operator, from, to, id, amount);
183
184            _afterTokenTransfer(operator, from, to, ids, amounts, data);
185
186            _doSafeTransferAcceptanceCheck(operator, from, to, id, amount, data);
187        }
188
189        /**
190         * @dev xref:ROOT:erc1155.adoc#batch-operations[Batched] version of {_safeTransferFrom}.
191         *
192         * Emits a {TransferBatch} event.
193         *
194         * Requirements:
195         *
196         * - If `to` refers to a smart contract, it must implement {IERC1155Receiver-onERC1155BatchReceived}
    ↪  and return the
197         * acceptance magic value.
198         */
199        function _safeBatchTransferFrom(
200            address from,
201            address to,
202            uint256[] memory ids,
203            uint256[] memory amounts,
204            bytes memory data
205        ) internal virtual {
206            require(ids.length == amounts.length, "ERC1155: ids and amounts length mismatch");
207            require(to != address(0), "ERC1155: transfer to the zero address");
208
```

```
209         address operator = _msgSender();
210
211         _beforeTokenTransfer(operator, from, to, ids, amounts, data);
212
213         for (uint256 i = 0; i < ids.length; ++i) {
214             uint256 id = ids[i];
215             uint256 amount = amounts[i];
216
217             uint256 fromBalance = _balances[id][from];
218             require(fromBalance >= amount, "ERC1155: insufficient balance for transfer");
219             unchecked {
220                 _balances[id][from] = fromBalance - amount;
221             }
222             _balances[id][to] += amount;
223         }
224
225         emit TransferBatch(operator, from, to, ids, amounts);
226
227         _afterTokenTransfer(operator, from, to, ids, amounts, data);
228
229         _doSafeBatchTransferAcceptanceCheck(operator, from, to, ids, amounts, data);
230     }
231
232     /**
233      * @dev Sets a new URI for all token types, by relying on the token type ID
234      * substitution mechanism
235      * https://eips.ethereum.org/EIPS/eip-1155#metadata[defined in the EIP].
236      *
237      * By this mechanism, any occurrence of the `\{id\}` substring in either the
238      * URI or any of the amounts in the JSON file at said URI will be replaced by
239      * clients with the token type ID.
240      *
241      * For example, the `https://token-cdn-domain/\{id\}.json` URI would be
242      * interpreted by clients as
243      * `https://token-cdn-domain/000000000000000000000000000000000000000000000000000000000004cce0.json`
244      * for token type ID 0x4cce0.
245      *
246      * See {uri}.
247      *
248      * Because these URIs cannot be meaningfully represented by the {URI} event,
249      * this function emits no events.
250      */
251     function _setURI(string memory newuri) internal virtual {
252         _uri = newuri;
253     }
254
255     /**
256      * @dev Creates `amount` tokens of token type `id`, and assigns them to `to`.
257      *
258      * Emits a {TransferSingle} event.
259      *
260      * Requirements:
261      *
262      * - `to` cannot be the zero address.
263      * - If `to` refers to a smart contract, it must implement {IERC1155Receiver-onERC1155Received} and
↪  return the
264      * acceptance magic value.
```

```
265         */
266         function _mint(
267             address to,
268             uint256 id,
269             uint256 amount,
270             bytes memory data
271         ) internal virtual {
272             require(to != address(0), "ERC1155: mint to the zero address");
273 
274             address operator = _msgSender();
275             uint256[] memory ids = _asSingletonArray(id);
276             uint256[] memory amounts = _asSingletonArray(amount);
277 
278             _beforeTokenTransfer(operator, address(0), to, ids, amounts, data);
279 
280             _balances[id][to] += amount;
281             emit TransferSingle(operator, address(0), to, id, amount);
282 
283             _afterTokenTransfer(operator, address(0), to, ids, amounts, data);
284 
285             _doSafeTransferAcceptanceCheck(operator, address(0), to, id, amount, data);
286         }
287 
288         /**
289          * @dev xref:ROOT:erc1155.adoc#batch-operations[Batched] version of {_mint}.
290          *
291          * Emits a {TransferBatch} event.
292          *
293          * Requirements:
294          *
295          * - `ids` and `amounts` must have the same length.
296          * - If `to` refers to a smart contract, it must implement {IERC1155Receiver-onERC1155BatchReceived}
    ↪  and return the
297          * acceptance magic value.
298          */
299         function _mintBatch(
300             address to,
301             uint256[] memory ids,
302             uint256[] memory amounts,
303             bytes memory data
304         ) internal virtual {
305             require(to != address(0), "ERC1155: mint to the zero address");
306             require(ids.length == amounts.length, "ERC1155: ids and amounts length mismatch");
307 
308             address operator = _msgSender();
309 
310             _beforeTokenTransfer(operator, address(0), to, ids, amounts, data);
311 
312             for (uint256 i = 0; i < ids.length; i++) {
313                 _balances[ids[i]][to] += amounts[i];
314             }
315 
316             emit TransferBatch(operator, address(0), to, ids, amounts);
317 
318             _afterTokenTransfer(operator, address(0), to, ids, amounts, data);
319 
320             _doSafeBatchTransferAcceptanceCheck(operator, address(0), to, ids, amounts, data);
```

```
    }

    /**
     * @dev Destroys `amount` tokens of token type `id` from `from`
     *
     * Emits a {TransferSingle} event.
     *
     * Requirements:
     *
     * - `from` cannot be the zero address.
     * - `from` must have at least `amount` tokens of token type `id`.
     */
    function _burn(
        address from,
        uint256 id,
        uint256 amount
    ) internal virtual {
        require(from != address(0), "ERC1155: burn from the zero address");

        address operator = _msgSender();
        uint256[] memory ids = _asSingletonArray(id);
        uint256[] memory amounts = _asSingletonArray(amount);

        _beforeTokenTransfer(operator, from, address(0), ids, amounts, "");

        uint256 fromBalance = _balances[id][from];
        require(fromBalance >= amount, "ERC1155: burn amount exceeds balance");
        unchecked {
            _balances[id][from] = fromBalance - amount;
        }

        emit TransferSingle(operator, from, address(0), id, amount);

        _afterTokenTransfer(operator, from, address(0), ids, amounts, "");
    }

    /**
     * @dev xref:ROOT:erc1155.adoc#batch-operations[Batched] version of {_burn}.
     *
     * Emits a {TransferBatch} event.
     *
     * Requirements:
     *
     * - `ids` and `amounts` must have the same length.
     */
    function _burnBatch(
        address from,
        uint256[] memory ids,
        uint256[] memory amounts
    ) internal virtual {
        require(from != address(0), "ERC1155: burn from the zero address");
        require(ids.length == amounts.length, "ERC1155: ids and amounts length mismatch");

        address operator = _msgSender();

        _beforeTokenTransfer(operator, from, address(0), ids, amounts, "");

```

```
        for (uint256 i = 0; i < ids.length; i++) {
            uint256 id = ids[i];
            uint256 amount = amounts[i];

            uint256 fromBalance = _balances[id][from];
            require(fromBalance >= amount, "ERC1155: burn amount exceeds balance");
            unchecked {
                _balances[id][from] = fromBalance - amount;
            }
        }

        emit TransferBatch(operator, from, address(0), ids, amounts);

        _afterTokenTransfer(operator, from, address(0), ids, amounts, "");
    }

    /**
     * @dev Approve `operator` to operate on all of `owner` tokens
     *
     * Emits an {ApprovalForAll} event.
     */
    function _setApprovalForAll(
        address owner,
        address operator,
        bool approved
    ) internal virtual {
        require(owner != operator, "ERC1155: setting approval status for self");
        _operatorApprovals[owner][operator] = approved;
        emit ApprovalForAll(owner, operator, approved);
    }

    /**
     * @dev Hook that is called before any token transfer. This includes minting
     * and burning, as well as batched variants.
     *
     * The same hook is called on both single and batched variants. For single
     * transfers, the length of the `ids` and `amounts` arrays will be 1.
     *
     * Calling conditions (for each `id` and `amount` pair):
     *
     * - When `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * of token type `id` will be  transferred to `to`.
     * - When `from` is zero, `amount` tokens of token type `id` will be minted
     * for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens of token type `id`
     * will be burned.
     * - `from` and `to` are never both zero.
     * - `ids` and `amounts` have the same, non-zero length.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
     */
    function _beforeTokenTransfer(
        address operator,
        address from,
        address to,
        uint256[] memory ids,
        uint256[] memory amounts,
```

```
435         bytes memory data
436     ) internal virtual {}
437
438     /**
439      * @dev Hook that is called after any token transfer. This includes minting
440      * and burning, as well as batched variants.
441      *
442      * The same hook is called on both single and batched variants. For single
443      * transfers, the length of the `id` and `amount` arrays will be 1.
444      *
445      * Calling conditions (for each `id` and `amount` pair):
446      *
447      * - When `from` and `to` are both non-zero, `amount` of ``from``'s tokens
448      * of token type `id` will be  transferred to `to`.
449      * - When `from` is zero, `amount` tokens of token type `id` will be minted
450      * for `to`.
451      * - when `to` is zero, `amount` of ``from``'s tokens of token type `id`
452      * will be burned.
453      * - `from` and `to` are never both zero.
454      * - `ids` and `amounts` have the same, non-zero length.
455      *
456      * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
457      */
458     function _afterTokenTransfer(
459         address operator,
460         address from,
461         address to,
462         uint256[] memory ids,
463         uint256[] memory amounts,
464         bytes memory data
465     ) internal virtual {}
466
467     function _doSafeTransferAcceptanceCheck(
468         address operator,
469         address from,
470         address to,
471         uint256 id,
472         uint256 amount,
473         bytes memory data
474     ) private {
475         if (to.isContract()) {
476             try IERC1155Receiver(to).onERC1155Received(operator, from, id, amount, data) returns (bytes4
↪  response) {
477                 if (response != IERC1155Receiver.onERC1155Received.selector) {
478                     revert("ERC1155: ERC1155Receiver rejected tokens");
479                 }
480             } catch Error(string memory reason) {
481                 revert(reason);
482             } catch {
483                 revert("ERC1155: transfer to non ERC1155Receiver implementer");
484             }
485         }
486     }
487
488     function _doSafeBatchTransferAcceptanceCheck(
489         address operator,
490         address from,
```

```
        address to,
        uint256[] memory ids,
        uint256[] memory amounts,
        bytes memory data
    ) private {
        if (to.isContract()) {
            try IERC1155Receiver(to).onERC1155BatchReceived(operator, from, ids, amounts, data) returns (
                bytes4 response
            ) {
                if (response != IERC1155Receiver.onERC1155BatchReceived.selector) {
                    revert("ERC1155: ERC1155Receiver rejected tokens");
                }
            } catch Error(string memory reason) {
                revert(reason);
            } catch {
                revert("ERC1155: transfer to non ERC1155Receiver implementer");
            }
        }
    }

    function _asSingletonArray(uint256 element) private pure returns (uint256[] memory) {
        uint256[] memory array = new uint256[](1);
        array[0] = element;

        return array;
    }
}
```

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature