Technische Universität Berlin

Documentation

IOSL: Distributed Identity Management in the Blockchain

Authors:
Filippo Boiani – 387680
Riccardo Sibani – 382708
Ijaz Ullah – 387533
Stefan Stojikovki – 387529
Ahmad Jawid Jamiulahmadi – 380457

Supervisors:
Sebastian Göndör
Prof. Dr. Axel Küpper

August 7, 2017

Contents

1 Introduction

1.1 Background

1.1.1 SONIC Project

Today's communication happens mostly through Online Social Network (OSN), but with their proliferation, some problems have arisen. The main issue is related to the fact that OSN are built in a closed an proprietary manner (think of Facebook, Twitter or LinkedIn) and don't allow a smooth communication among them. Therefore, the user is forced to create different accounts, and build different networks within every platform. All this accounts and information correspond to the same user, but are heavily segregated from one another. This segregation is caused by the platform which aim to bind the user in a lock-in effect.

The SONIC project has been developed under suggestion of Telekom Innovation Laboratories to overcome this problem and facilitate a seamless connectivity between different OSNs [5] allowing the migration of accounts between different platforms.

The SONIC project expects the social profiles to be managed independently from the platform they are hosted on. The profiles can be migrated from one platform to another at any time and, therefore, the identifiers should be domain agnostic and it should be possible to create them in a distributed fashion (i.e. every node of the network is able to create or derive them).

Introducing this domain agnostic identifiers creates the need of a system to resolve them to actual profile locations. In order to translate a GID to an user profile, SONIC has introduced the Global Social Lookup System (GSLS), a distributed directory service built on peer to peer technology using distributed hash tables (DHT), This mapping is possible thanks to datasets called Social Records which are digitally signed. [5].

1.1.2 Blockchain

Blockchain is a type of distributed ledger or decentralized database that keeps records of digital transactions. Rather than having a central administrator like a traditional database, (think banks, governments and accountants), a distributed ledger has a network of replicated databases, synchronized via the internet and visible to anyone within the network. Blockchain networks can be private with restricted membership similar to an intranet, or public, like the Internet, accessible to any person in the world.

When a digital transaction is carried out, it is grouped together in a cryptographically protected block with other transactions that have occurred in some time period and sent out to the entire network. Miners (members in the network

with high levels of computing power) then compete to validate the transactions by solving complex coded problems. The first miner to solve the problems and validate the block receives a reward.

The validated block of transactions is then timestamped and added to a chain in a linear, chronological order. New blocks of validated transactions are linked to older blocks, making a chain of blocks that show every transaction made in the history of that blockchain.The entire chain is continually updated so that every ledger in the network is the same, giving each member the ability to prove who owns what at any given time.

Blocks hold batches of valid transactions that are hashed and encoded into a Merkle tree. Each block includes the hash of the prior block in the blockchain, linking the two. Variants of this format were used previously, for example in Git. The format is not by itself sufficient to qualify as a blockchain. The linked blocks form a chain. This iterative process confirms the integrity of the previous block, all the way back to the original genesis block.

Blockchain's decentralized, open and cryptographic nature allow people to trust each other and transact peer to peer, making the need for intermediaries obsolete. This also brings unprecedented security benefits. Hacking attacks that commonly impact large centralized intermediaries like banks would be virtually impossible to pull off on the blockchain. For example if someone wanted to hack into a particular block in a blockchain, a hacker would not only need to hack into that specific block, but all of the proceeding blocks going back the entire history of that blockchain. And they would need to do it on every ledger in the network, which could be millions, simultaneously.

Blockchains are secure by design and are an example of a distributed computing system with high Byzantine fault tolerance.

1.1.3 Identity Management

Identity management refers to the process of employing emerging technologies to manage information about the identity of users and control access to company resources [7]. The goal of identity management is to improve productivity and security while lowering costs associated with managing users and their identities, attributes, and credentials.

Everyone is using internet to interact with different digital service platforms. It has many forms like accessing social sites, online shopping services, and interacting with your online banking account. Interactions with these service providers require that each user has digital identity so that user is authorized to access digital platform. There are certain reasons digital platform uses identities and storing all the information related to the user to grow their business and improve user experiences, and defend against certain attacks externally as well as to take care of the privacy of the user.

Identities were used in different ways like accessing personal computer we use only username and password. Digital platforms on the other hand like Facebook, Yahoo use a domain name with a combination of a username to isolate users because these platforms have many users so domain name can make username uniquely identifiable [6].

The problem in these digital platforms is the absence of federated directories. Users often forget sign credential when they have many different ones. It also increases administrative overhead when we have separate credential for each application. Microsoft defines federation as "the technology and business arrangements necessary for the interconnecting of users, applications, and systems. This includes authentication, distributed processing and storage, data sharing, and more." [7]

Federated directories interact and trust each other, thus allowing secure information sharing between applications. Companies are currently running isolated, independent directories that neither interact with nor trust each other. This is a result of applications having their own proprietary identity stores. Each proprietary directory requires its own method of user administration, user provisioning, and user access control. The problem with proprietary identity stores is that users require login for every application, which in turn burdens users with having to remember numerous username and password combinations.

The need for blockchain based identity management is particularly noticeable in the internet age, we have faced identity management challenges since the dawn of the Internet. Prime among them: security, privacy, and usability [1]. Blockchain technology may offer a way to circumvent these problems by delivering a secure solution without the need for a trusted, central authority. It can be used for creating an identity on the blockchain, making it easier to manage for individuals, giving them greater control over has personal information and how they access it.

1.2 BCIMS: Blockchain Identity Management System

The aim of this project is to build a blockchain-based, distributed system for self-asserted identities for Distributed Online Social Networks (DOSN) in the Sonic ecosystem. The mission is to: first research the state of the art regarding to identity management and blockchains; then conceptualize and design a service to manage self-asserted identities in a blockchain; and finally implement and evaluate the designed solution.

1.2.1 GSLS Problem And Solution

The current implementation of the GSLS is implemented in Java Spring-Boot and is exposing APIs for creating, updating and querying the Social Records. The Social Records are stored in Distributed Hash Table (DHT) and RSA private-public key encryption is used for validation of the records. However, there is a security issue with this implementation because someone can spawn a malicious node in the DHT which contains validated (signed by the real user), but outdated (old) data and not the most recent one, and there is no way to detect this. The data is checked against the user's public key and it is considered valid if the signature is correct, but we cannot be sure if this data is the most recent one.

To overcome this security issue, we have been working on an implementation which stores the Social Records on the Blockchain, since the Blockchain is

immutable distributed ledger of transactions which can be followed in order to determine the most recent state of the SR.

In order to achieve this, we are building an application which exposes RESTful APIs for creation, modification and retrieving of SR that makes use of the Blockchain technology. Additionally, the service architecture needs to be distributed, without single point of failure and it needs to prevent attempts to write crafted datasets or overwrite datasets.

2 Related Work

2.1 Directory Service

Also known as name service, maps the names of network resources to their respective network addresses. Each resource on the network is considered an object by the directory server. Information about a particular resource is stored as a collection of attributes associated with that resource or object. One of the most representing example is the DNS, which offers internet domain names translation to ip addresses.

2.2 DHT and P2P Networks

A distributed hash table (DHT) is a class of a decentralised distributed system that provides a lookup service similar to a hash table: (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. We took into consideration tomP2P as it is based on Kademlia, one of the most successful DHT (e.g Gnutella and IPFS are using it).

2.2.1 DHT characteristics

Autonomy and decentralization: multiple nodes, no central coordinator;

Fault tolerance: reliable, no single point of failure (like Napster's central server) (Byzantine fault tolerance);

Scalability: the system scales. This means it's working efficiently regardless the workload;

Load balancing (optional);

Data integrity (optional).

2.2.2 Functioning Example

Once these components are in place, a typical use of the DHT for storage and retrieval might proceed as follows. Suppose the keyspace is the set of 160-bit strings. To index a file with given filename and data in the DHT, the SHA-1 hash of filename is generated, producing a 160-bit key k, and a message put(k, data) is sent to any node participating in the DHT. The message is forwarded from node to node through the overlay network until it reaches the single node responsible for key k as specified by the keyspace partitioning. That node then stores the key and the data. Any other client can then retrieve the contents of the file by again

hashing filename to produce k and asking any DHT node to find the data associated with k with a message get(k). The message will again be routed through the overlay to the node responsible for k, which will reply with the stored data.

2.2.3 The Network

Each node maintains a set of links to other nodes (its neighbors or routing table). Together, these links form the overlay network. A node picks its neighbors according to a certain structure, called the network's topology.

Aside from routing, there exist many algorithms that exploit the structure of the overlay network for sending a message to all nodes, or a subset of nodes, in a DHT. These algorithms are used by applications to do overlay multicast, range queries, or to collect statistic. (Flooding).

2.3 Blockchain Implementations

Blockchain introduction

Blockchain's distributed ledger system is used to keep track of Bitcoin transactions. Blockchain eliminates the need for central authorities and enables each user of the system to maintain their own copy of the ledger. It also keeps all copies of the ledger synchronized through a consensus algorithm.Bitcoin miners do the recording and validation of the transactions. The miners are necessary to prevent "double spend".

Bitcoin

Blockchain is a distributed ledger technology used to keep track of Bitcoin cryptocurrency transactions. Distributed ledgers create a data structure – like a chain – where records of every single Bitcoin transaction live. To prevent "double spend", all Bitcoin transactions are validated and then permanently archived in the cryptographic ledger or chain. The validation is done via a peer-to-peer process that is hugely computer-intensive. It is supported by a global network of volunteers – known as "miners" – who are incentivized mainly by Bitcoin's mining reward. In essence, Bitcoin uses cryptography to enable participants on the network to update the ledger in a secure way without the need for a central authority. The key to Blockchain was to agree on the order of entries in the ledger. Once this was in place, distributed control of Bitcoin was possible.

Namecoin

Namecoin is an experimental open-source technology which improves decentralization, security, censorship resistance, privacy, and speed of certain components of the Internet infrastructure such as DNS and identities. Namecoin is a key/value pair registration and transfer system based on the Bitcoin technology. What does Namecoin do under the hood?

Securely record and transfer arbitrary names (keys).

Attach a value (data) to the names (up to 520 bytes).

Transact the digital currency namecoins (NMC).

Like bitcoins, Namecoin names are difficult to censor or seize.

Lookups do not generate network traffic (improves privacy).

Hyperledger / Hyperledger Fabric

Hyperledger is an open source collaborative effort created to advance cross-industry blockchain technologies. It is a global collaboration, hosted by The Linux Foundation, including leaders in finance, banking, Internet of Things, supply chains, manufacturing and Technology. Hyperledger Fabric is a business blockchain framework hosted on Hyperledger intended as a foundation for developing blockchain applications or solutions with a modular architecture. Hyperledger Fabric allows components such as consensus and membership services to be plug-and-play. Hyperledger Fabric establishes trust, transparency, and accountability based on the following principles:

Permissioned network - Provides collectively defined membership and access rights within your business network

Confidential transactions - Gives businesses the flexibility and security to make transactions visible to select parties with the correct encryption keys

No cryptocurrency - Does not require mining and expensive computations to assure transactions

Programmable - Leverage the embedded logic in smart contracts to automate business processes across your network

Ethereum

Ethereum is a decentralized platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third party interference. These apps run on a custom built blockchain, an enormously powerful shared global infrastructure that can move value around and represent the ownership of property. This enables developers to create markets, store registries of debts or promises, move funds in accordance with instructions given long in the past (like a will or a futures contract) and many other things that have not been invented yet, all without a middle man or counterparty risk. On traditional server architectures, every application has to set up its own servers that run their own code in isolated silos, making sharing of data hard. If a single app is compromised or goes offline, many users and other apps are affected. On a blockchain, anyone can set up a node that replicates the necessary data for all nodes to reach an agreement and be compensated by users and app developers. This allows user data to remain private and apps to be decentralized like the Internet was supposed to work.

Storj

Storj is a peer-to-peer cloud storage network implementing client-side encryption would allow users to transfer and share data without reliance on a third party storage provider. The removal of central controls would mitigate most traditional data failures and outages, as well as significantly increase security, privacy, and data control. Peer-to-peer networks are generally unfeasible for production storage systems, as data availability is a function of popularity, rather than utility.

Swarm

Swarm is a distributed storage platform and content distribution service, a native base layer service of the ethereum web 3 stack. The primary objective of Swarm is to provide a decentralized and redundant store of Ethereum's public record, in particular to store and distribute dapp code and data as well as block chain data. From the end user's perspective, Swarm is not that different from WWW, except that uploads are not to a specific server. The objective is to peer-to-peer storage and serving solution that is DDOS-resistant, zero-downtime, fault-tolerant and censorship-resistant as well as self-sustaining due to a built-in incentive system which uses peer to peer accounting and allows trading resources for payment. Swarm is designed to deeply integrate with the devp2p multiprotocol network layer of Ethereum as well as with the Ethereum blockchain for domain name resolution, service payments and content availability insurance.

IPFS

The InterPlanetary File System (IPFS) is a peer-to-peer distributed file system that seeks to connect all computing devices with the same system of files. In some ways, IPFS is similar to the Web, but IPFS could be seen as a single BitTorrent swarm, exchanging objects within one Git repository. In other words, IPFS provides a high throughput content-addressed block storage model, with content-addressed hyper links. This forms a generalized Merkle DAG, a data structure upon which one can build versioned file systems, blockchains, and even a Permanent Web. IPFS combines a distributed hashtable, an incentivized block exchange, and a self-certifying namespace. IPFS has no single point of failure, and nodes do not need to trust each other.

Filecoin

Filecoin is a distributed electronic currency similar to Bitcoin. Unlike Bitcoin's computation-only proof-of-work, Filecoin's proof-of-work function includes a proof-of-retrievability component, which requires nodes to prove they store a particular file. The Filecoin network forms an entirely distributed file storage system, whose nodes are incentivized to store as much of the entire network's data as they can. The currency is awarded for storing files, and is transferred in transactions, as in Bitcoin. Files are added to the network by spending currency.

This produces strong monetary incentives for individuals to join and work for the network. In the course of ordinary operation of the Filecoin network, nodes contribute useful work in the form of storage and distribution of valuable data.

BigChainDB

BigChainDB is aiming to merge database and Blockchain, trying to keep all the properties of the first such as a linear scaling throughput, an efficient query language and permissions in order to interact with it. In particular developing a MongoDB based NoSQL distributed database. With the fundamental features of Blockchain as decentralisation, immutability, creation and movement of digital assets. One interesting approach with BigChainDB is that it can be integrated with Ethereum (sort of its database) and be run on private or public networks.

2.4 Blockstack Use Case

Blockstack is a particular implementation of a decentralized DNS system based on blockchain. It combines DNS functionality with public key infrastructure and is primarily meant to be used by new blockchain applications.

According to the company: "under the hood, Blockstack provides a decentralized domain name system (DNS), decentralized public key distribution system, and registry for apps and user identities" [**BlockStackMainPage**].

The real breakthrough is the architecture the system is built on. It can be described as a three-layer design with the blockchain as the first and lower tier, the storage system as the upper and the peer network as middle layer.

The architecture is shown in 2.1.

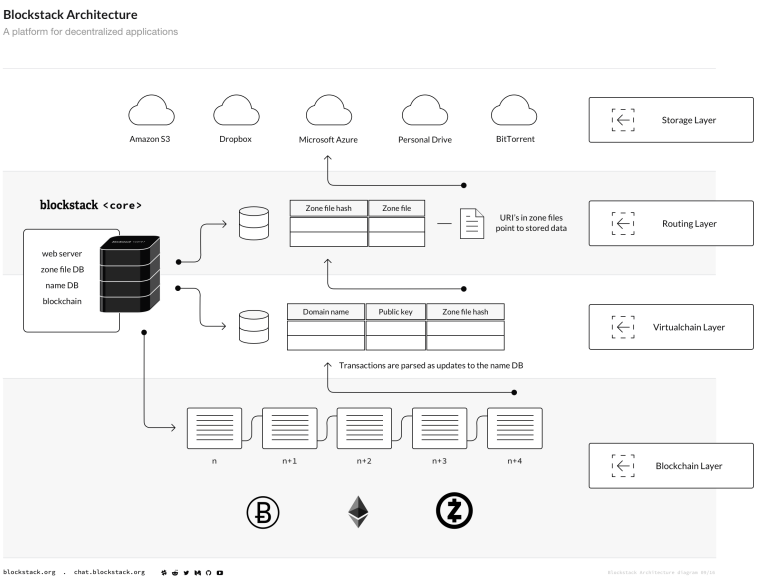

Figure 2.1: Basic Blockstack architecture [2]

2.5 Choosing the best Blockchain

As shown, each blockchain implementation has its pro and cons. In this particular project the authors decided to use the Ethereum project due to the following reasons:

- **Big Community**: Ethereum has the second biggest community after Bitcoin blockchain. This also mean a big user base both when is needed to reach the users and when is matter of mining fees.
- **Clear documentation**: It has one of the best documented website and a section on Stack Overflow.
- **Transaction fees**: The transaction fee (even if not the cheapest) are pretty low in compare to Bitcoin.
- **Easy to store and keep track of the changes**: Besides many other implementations, Ethereum is account base, which means that is possible to understand at a glance the state of one account.

2.6 Solidity

Solidity is a high level programming language developed specifically for Ethereum blockchain. Solidity introduced the concept of *smart contract* which is the base for different use cases like for voting, crowdfunding, blind auctions, multi-signature wallets and more.

Smart contracts are executed inside the Ethereum Virtual Machine (EVM) by any node in the network. The nodes use consensus protocols to agree upon certain actions resulting from the smart's contract code. Like other programming languages, Solidity support inheritance, user defined types, libraries, control structures, comments and a many other features. The syntax of solidity is designed around ECMAScript to make it easier for existing web developers. The latter characteristic made it the preferred language to write smart contracts on Ethereum.

2.7 Public Key Cryptography

2.7.1 Elliptic Curves

ECC is the next generation of public key cryptography, and based on currently understood mathematics, it provides a significantly more secure foundation than first-generation public key cryptography systems like RSA.

With ECC, you can use smaller keys to get the same levels of security. Small keys are important, especially in a world where more and more cryptography is done on less powerful devices like mobile phones. While multiplying two prime numbers together is easier than factoring the product into its component parts, when the prime numbers start to get very long, even just the multiplication step can take some time on a low powered device. While you could likely continue to keep RSA secure by increasing the key length, that comes with a cost of slower

cryptographic performance on the client. ECC appears to offer a better tradeoff: high security with short, fast keys.

2.7.2 RSA

RSA algorithm, along with the Diffie-Hellman represented the first viable cryptographic schemes where security was based on the theory of numbers; it was the first to enable secure communication between two parties without a shared secret. Modern cryptography is founded on the idea that the key that you use to encrypt your data can be made public while the key that is used to decrypt your data can be kept private.

The algorithm multiplies two prime numbers. If multiplication is the easy algorithm, its difficult pair algorithm is factoring the product of the multiplication into its two component primes. Algorithms that have this characteristic — easy in one direction, hard the other — are known as trapdoor functions. Finding a good trapdoor function is critical to making a secure public key cryptographic system.

Its security relies on the fact that factoring is slow and multiplication is fast. In general, a public key encryption system has two components, a public key and a private key. Encryption works by taking a message and applying a mathematical operation to it to get a random-looking number. Decryption takes the random looking number and applies a different operation to get back to the original number. Encryption with the public key can only be undone by decrypting with the private key.

In RSA, this maximum value (call it max) is obtained by multiplying two random prime numbers. The public and private keys are two specially chosen numbers that are greater than zero and less than the maximum value (call them pub and priv). To encrypt a number, you multiply it by itself pub times, making sure to wrap around when you hit the maximum. To decrypt a message, you multiply it by itself priv times, and you get back to the original number.

Specialized algorithms like the Quadratic Sieve and the General Number Field Sieve were created to tackle the problem of prime factorization and have been moderately successful. As the resources available to decrypt numbers increase, the size of the keys needs to grow even faster. This is not a sustainable situation for mobile and low-powered devices that have limited computational power.

3 Concept and Design

This chapter holds and explains in detail the different approaches taken into consideration in order to store social records. Before doing so, it is important to define some terms and conventions which are going to be used in the following sections and chapters. The terms are particularly related to Ethereum, the chosen blockchain implementation.

3.1 Project Dictionary And Legend

Ethereum Accounts

As stated in the Ethereum white paper, the state is made up of objects called accounts, with each account having a 20-byte address. All the state transitions are represented as direct transfers of value and information between accounts. An Ethereum account contains four fields:

The nonce: a counter used to make sure each transaction can only be processed once;

Balance: the account's current ether balance;

Code: the account's contract code, if present;

Storage: the account's storage (empty by default).

There are two types of account:

Externally owned accounts: controlled by private keys. It has no code, and one can send messages from an externally owned account by creating and signing a transaction.

Contact accounts: controlled by contract code. Every time the contract account receives a message its code activates, allowing it to read and write to internal storage and send other messages or create contracts in turn. The contract live as autonomous agents inside the Ethereum execution environment. Wallet accounts are nothing but particular type of contract accounts [4]

Account Key Pair

Each account holds a key pair stored in a file called "UTC JSON Keystore File". This file is password encrypted and it is possible to unlock it only by knowing the pass phrase. The private key is generally used to sign transactions, while the public one is used to identify the account all over the Ethereum network.

The key pair is generated using an algorithm called *secp256k1*, based on elliptic curves cryptography which is more secure and efficient of any algorithm based on RSA. In particular, the private key is a 64-character long hexadecimal string. It is randomly generated at the moment of the account creation. The public key is generated starting from the private one and is publicly accessible. The account address is nothing but a part of the hash of the public key (in particular, the last 20 bytes of the keccak-256 hash of the public key).

Transaction

The term "transaction" is used in Ethereum to refer to the signed data package that stores a message to be sent from an externally owned account. Transactions contain:

The recipient of the message;

A signature identifying the sender;

The amount of ether to transfer from the sender to the recipient;

An optional data field;

A "start gas" value, representing the maximum number of computational steps the transaction execution is allowed to take; and

A "gas price" value, representing the fee the sender pays per computational step.

Raw Transaction

All transactions need to be signed otherwise they are considered invalid transactions and will not be included in the blockchain. A raw transaction is a transaction in raw bytes. If one has the raw bytes of a valid transaction, they can send it to the blockchain. Otherwise, web3.js – the library provided by Ethereum to interact with the blockchain – automatically creates the signed transaction bytes as part of the default implementation. The library converts the JSON transaction to the raw bytes and automatically signs them. In a raw transaction nothing is done automatically, but it is developer's responsibility to first convert each parameter then sign the transaction and eventually send it to a node.

Using some javascript libraries (like ethereumjs-tx) raw transactions can be created and signed offline. After the creation, they can be serialised and sent to the GSLS which is accountable of sending the transaction to the blockchain.

Ethereum Node

A node is a piece of software that connects to other nodes, thus participating in the formation of the network. A node stores an entire copy of the blockchain, and a node may mine (but doesn't have to). All interaction with the blockchain (interaction with contracts, etc) needs to go via a node. It is also referred to as "blockchian client".

Mining

Mining is a computationally intensive work that requires a lot of processing power and time. Mining is the act of participating in a given peer distributed cryptocurrency network in consensus. The miner is subsequently rewarded for providing solutions to challenging math problems. It is done by putting the computer's hardware to use with mining applications.

Other Terms

Term	Definition
User	The one that owns a social record and has the right to create and update it as well as to search for other social records. The user does this by interacting with an application, the GSLS client.
GSLS client	It is a piece of code that allows the user to interact and send request to the GSLS server. In our last approach, the client is implemented as a Desktop application.
GSLS server	The GSLS server is a piece of code that exposes api callable by the GSLS client via HTTP protocol. The server is also a blockchain client in that it is either connected to an Ethereum node or it is an actual node in the network.
GSLS Administrator	Whoever spawns a GSLS server instance and has its control, meaning that she can start and stop it at will.

3.2 Approaches to Store Social Records

3.2.1 Store the Social Record

The most intuitive solution would be to switch from the DHT to the Blockchain, moving each social record through a transaction. Unfortunately after few considerations, this approach does not seem to be optimal:

- Storing the entire social record would occupy a high amount of memory in the distributed node. In particular the limit for the data structure is 44kb, but is best practice to keep the records light.
- Checking the real identity of the uploader would be impossible, one option could be to allow only a specific wallet address to write into it.
- Looking up would not be efficient.

3.2.2 Storing The Hash

This solution is inspired by Blockstack architecture and requires to create a DHT to map the GlobalID to the address of the transaction contained inside the blockchain.

The hash of the Social Record is stored in the data field of the transaction in the Blockchain. In this manner, the blockchain has the role of validation infrastructure rather than storage system. The actual data is stored in the distributed hash table. This solution could be vulnerable in case a malicious node tries to restore an old version of the social record by injecting the wrong address in the DHT table, causing a corrupted record.

3.2.3 Using A Third-Party Solution

Blockstack offers an already implemented solution using a layered architecture similar to the one proposed in solution 2. It also allows to choose among different blockchain implementations and storage systems. Nevertheless, the underlying idea is the same as solution 2: use the blockchain as a validation system and use a DHT to map ids (keys/GlobalIDs) with the corresponding values (Social Records).

According to Blockstack, the applications built on this system are serverless and decentralized. Developers start by building a single-page application in Javascript. Then, instead of plugging the frontend into a centralized API, they plug into an API run by the user. Developers install a library called 'blockstack. js' and do not have to worry about running servers, maintaining databases, or building out user management systems [add reference].

3.3 How To Store The Social Records

It is important to understand how to write transactions in the blockchain, in particular in an ethereum contract.

Given the fact that the GSLS is an open source, and not centrally managed server, a safety threat may arise. In order to ask the server to record a particular transaction, a password is required (otherwise a malicious user could update the social record on behalf of someone else), but of course in case of a malicious node the password could be sniffed. In order to solve this problem asymmetric keys solutions can be implemented (such as the above mentioned RSA and ECC).

The contract will therefore be a *key-value* array with the globalId as key and the as value a json object with the following structure:

In case we store the hashed Social Record

```
{
        publicKey: String,
        hashedSocialRecord: String,
}
```

With this data structure in order to verify the authenticity of the social record will be enough to hash the social record stored in the DHT and compare it with the variable(the hash) stored in the blockchain for that particular user. In order to update the information, the following object needs to be encrypted with the user's private key:

```
{
        globalId: String,
        hashedSocialRecord: String,
}
```

In order to update the record, the signature is checked with the help of the user's public key, and then, if valid, the record is updated in a new transaction in the Blockchain.

In case we store the entire Social Record in the Blockchain by the GSLS server

```
{
        publicKey: string,
        socialRecord: Object
}
```

The record to store is similar, the only difference is that rather than the hashed-SocialRecord string, the contract is storing the socialRecord Object (which implies higher costs). Unfortunately the process to insert the social record for the first time is more complex. The main reason of this complexity is due to the fact that the GSLS system will not look up for the DHT table anymore (and therefore it will be possible to remove it) and the authenticity of the record must be assured by the solidity contract.

The first writing especially, will be extremely difficult with an object as shown:

```
{
        globalId: string,
        publicKey: string,
        encriptedData: string,
}
```

EncryptedData is a string that can be decrypted with the public key and that will be declared authentic if the result of the decryption will be a JSON object as shown:

```
{
        globalId: string,
        socialRecord: Object,
}
```

This approach, even though complicated at first, solves a crucial problem and the biggest barrier of the blockchain technology in general: force all the users to run the blockchain.

In fact, if the users don't want to host an Ethereum node and cannot rely on the authenticity of the server, it is impossible for them to update their record.

The problem with users who don't want to host an Ethereum node could be overcome if they trust the server, but one of our constraints is to build a trestles application. In theory this can be done if the asymmetric key validation is done in a contract in the Blockchain, but unfortunately, so far, an asymmetric library in solidity has not been developed yet.

Even though the implementation of Elliptic Curves or RSA keys would be a useful (although expensive) feature, and the Elliptic curve is already implemented in Ethereum in order to create and verify private and public keys, a solidity implementation is not yet available.

Nevertheless there are several proposals and RFCs both for RSA and Elliptic curves [3] [8] and have been taken place suggestions of implementing asymmetric keys and many other features in the EVM2.0. Because of this lack the described process cannot be pursued.

From here second draft of the possible approaches description Needs to decide which one to maintain.

3.4 Using Accounts Without Hosting An Ethereum Node

The last approach that this paper wants to explain requires the users to create an Ethereum wallet but not to host an Ethereum node and can ensure a complete and working implementation without the DHT.

The *key-value* array this time has the user's wallet address as a key and the social record object as a value; this approach also reduce the complexity of the algorithm that needs to be run in solidity and therefore the costs.

The steps to follow are the same as the user participate in the blockchain with an ethereum node: he creates the transaction with all the data but, instead of uploading it in the network, the transaction is sent to the GSLS node and then finally pushed on the network.

This methodology is called offline transaction because the client can create and sign transactions with his ethereum account without running the node and so: being offline. The only limitation of this solution is that the transaction needs be signed with the private key of the account instead that with the usual combination of username - password created with the account.

This could lead to suspicion over this authentication system because the common sense and the documentation suggests not to share the private key, it is important to notice that the private key is not shared to the server and will never leave the local environment.

3.5 Possible Approaches

3.5.1 DHT Supported By The Blockchain As A Validation System

The first approach is based on the Blockstack architecture and is both cheap and light. The GSLS network is still connected using a DHT and It holds all the social record data like the original implementation. The Blockchain would come into play as a supporting data structure used by the GSLS to validate the social records and detect malicious changes.

How does it work

Creation And Update This implementation does not require the client to host an Ethereum node, but needs a wallet per each user.

The user needs to download a piece of software to run (eg. exe, epp).

The software creates the Sonic Private Key and Sonic Public Key (in case of a new user, otherwise the user will upload his existing keys), and then the user must save the private key in a secure location (otherwise he will loose the possibility to change his SR). Then the social record is created / uploaded as always and stored in the DHT. The SR is then hashed, resulting in a small dimension string. The hash is then uploaded in the blockchain through the server in a Key value array with the Id as Key end the hash as a value.

GET When a user wants to retrieve information starting from a particular GlobalId, he has to pull the SR of that particular GlobalId from the DHT and the hashes from the Blockchain.

If the hashed SR from the Blockchain is the same as the hashed version of the retrieved SR from the DHT, then the record is valid.

This approach is valid and cheap since only the Hash of the entire SR is stored. Unfortunately, the solution can only detect unauthorised changes but not rollback from it.

3.5.2 Without DHT and Etherereum Accounts for the Users

This approach merges the security and immutability of the Blockchain without forcing the User to have a wallet or Ethereum account. Nevertheless, it is important to keep in mind that the server that contacts the Blockchain must run a BC node. In our prototype we can imagine the GSLS as a Server hosting in Ethereum Node and implementing some functionalities explained here below.

How Does It Work?

Creation The user downloads the software, then generates the Sonic Public and Private Key. Once the social record has been created, it is encrypted with the private Key and the following object is sent to the server.

(Global ID, Pub Key, Encrypted (SR))

The server with the (Server wallet) uploads the transaction to the blockchain via a function call in a contract. The contract, stored in Ethereum, will decrypt the encrypted SR by using the Public Key of the user, and then store the SR in the Key Value array with the Global Id as key and the SR object (Public key, SR version 1) as Value.

Update In order to update the SR, the User application needs to retrieve the number of the version, encrypt the number of version + 1 and send the following object to the server (that will update the SR in the blockchain). The Solidity contract will decrypt the version and check if it matches the version +1 in the

Blockchain , if it matches, the SR can be decrypted as well and the record will be updated in the BC.

GET Get is pretty trivial, asking with a GlobalId to the Ethereum node, it will return the social record.

3.5.3 User With Ethereum Wallet

This approach requires the users to have an Ethereum wallet but not to run an Ethereum node. This is optimal because it allows the users to update their credentials even without trusting 3rd party service. Ideally, it is possible to create and change the record even from a different devices such as PC or smartphone.

It is enough to provide the wallet .json file.

How Does It Work?

Creation And Update Once the wallet json file is loaded into the application it is possible to create the social record, sign it with the private key; thus creating the transaction normally. This signed transaction is then uploaded(passed) to the server and then to the BC wallet file json.

What is needed is json file with the private key of the Ethereum wallet account (and other information).

The SR is stored in a Key Value array as Gid-SR

GET Get is pretty trivial, asking with a GlobalId to the Ethereum node, it will return the social record.

This approach forces the users to have an Ethereum wallet, but it is conceptually more correct because it uses the offline transaction form.

4 Implementation

4.1 System Architecture

The architecture is shown in the picture below:

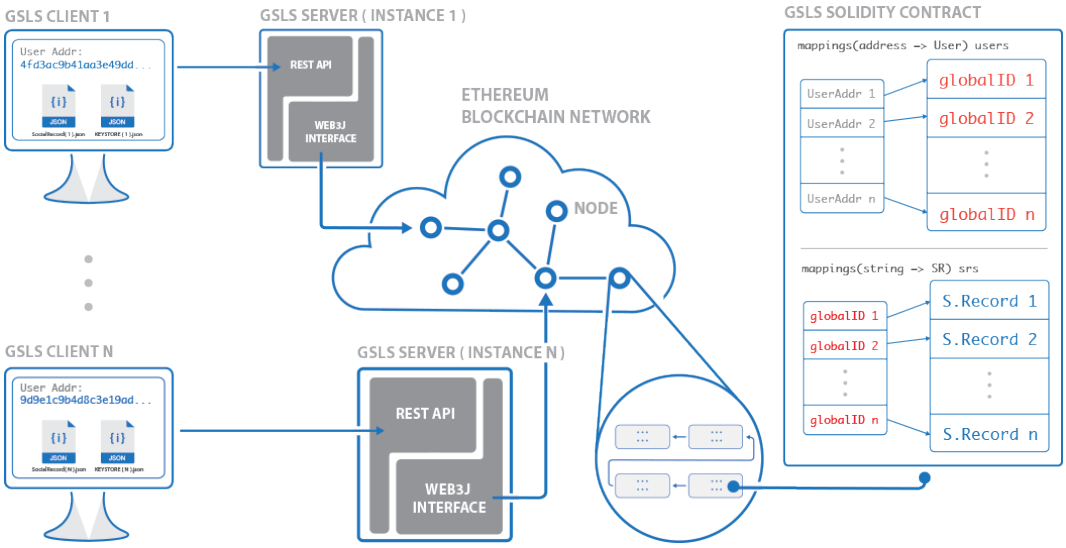

Figure 4.1: Basic architecture: n clients communicate with m GSLS server instances. The GSLS instances can interface with the blockchain because each and every of them is holding an Ethereum node. All the operations are referenced to the contract address storing the list of social records.

4.2 User side

The client can download the software locally (directly from the GSLS) from GitHub or wherever it is hosted). In this case, it is on Electron application running Nodejs on the backend side and React on the frontend side, where it is possible for a user to login, upload the Ethereum wallet file and the password. This information is needed to create the row transaction before sending it to the GSLS server.

4.2.1 POST /socialrecord And PUT /socialrecord

The User can create a new social record or modify an existing one (of course, only if he is allowed to). When the user clicks on the "submit" Button the following operations are performed:

- **To Address**: convert the address of the receiver (in this case the contract) to Hex
- **Get the Gas price**: the gas price is changing constantly, and in order to set the right amount, it is required to run a get HTTP request to the GSLS server.
- **Gas Limit**: the maximum amount of Gas that the user wants to spend for the transaction to flex.
- **Nonce**: This is a scalar vector equal to the number of the transactions sent from a particular address. It is a value that needs to be always different in order to satisfy the proof of work condition. The GSLS Server can retrieve this value, via web3.eth.getTransactionCount(*walletAddress*) and send the result to the user application. Then the nonce is converted to Hex.
- **Data**: The data field is where the meat is. The first 32 bits are reserved for the first 32 bits of the hashed function name (that the user wants to call). Then, since in the ABI File, the number of parameters required (and their type) is specified according to each function : it is possible to grok the bytes needed to append to the data string. In the case of creation of a new social record, 2 string parameters are required, which will be converted to hex and appended to the field data space into the string (padding with 0 according to the number of bytes allocated to the string type).

Since the client is the only part of the system allowed to access the wallet file, it must create the mentioned transaction in hexadecimal *style* and sign it with its private key. The signed transaction can then be sent to the Server via HTTP.

4.2.2 GET /socialrecord/:GlobalID

The get request, used to retrieve the social record for a particular GlobalID, is a simple promise that contacts the server specifying the global id and waiting for the result.

4.3 Server

The GSLS Server can be converted to a REST-ful Server delegated at pulling / pushing social records from / to the blockchain. The following interfaces are needed:

4.3.1 POST /socialrecord & PUT /socialrecord

For a creation of a new social record the following steps are requred:

- GET current Nonce and current Gas Price, asking to the server for the current account nonce and the gasPrice to set in the raw transaction.
- The body of the request only contains a signed transaction. The transaction is pushed into the blockchain and the hash of it returned. The hash is useful

because starting from that is possible to check the status of the transaction (e.g. on https://ropsten.etherscan.io).

- It is possible to create a notification system based on polling, with the client asking for the good or bad result of the operation. It is not possible to send the result in the same HTTP response since it could take more than 90 seconds and therefore time out the request.

4.3.2 GET /socialrecord/:GlobalID

The Server receives an HTTP request with one Global ID as parameter. It gets the social record from the blockchain with the help of the ethereum contract function and then it sends it back to the client as an HTTP response. In this case, it is not required to set up the polling mechanism because the get transaction operates on constant values and therefore does not need to be mined (the mining operation usually takes more time, in Ethereum from 30 seconds to potentially infinite).

4.4 Contract

The contract can be perceived as a class (with constructors and functions) stored in the blockchain. The mechanism (simplified) is that having the contract stored (with a unique address) in the Ethereum BC and one wallet sending one transaction to the contract address containing the method to call: it is possible to obtain the same output in all the nodes that will receive the transaction.

4.4.1 Structures

In the created social record contract there are 2 structures:

SR

```
struct SR {
    string socialRecord;
    bool exists;
}
```

The social record structure allows storing one string and setting one boolean value to mark if the above mentioned social record is valid.

User

```
struct User {
        string globalId;
        bool exists;
    }
```

In the same way, it allows storing one globalId and a boolean variable to mark if the specific globalId is existing

4.4.2 Variables

owner

```
address owner;
```

The creator of the contract is stored in the owner variable with type address.

4.4.3 Mappings

users

```
mapping (address => User) users;
```

The mapping *users* maps each address with one *User* structure. This is used to prove ownership of the SR when someone tries to modify it later.

srs

```
mapping (string => SR) srs;
```

The mapping *srs* maps each globalId (string) with one *SR* structure.

4.4.4 Modifiers

The modifiers define who is authorised to call a specific function.

onlyOwner

```
modifier onlyOwner {
    require(msg.sender == owner);
    _;
}
```

The caller of the function must be the owner of the contract.

onlyUser

```
modifier onlyUser {
    if(users[msg.sender].exists) {
        _;
    }
}
```

The caller of the function must be verified user.

onlyNotUser

```
modifier onlyNotUser {
    if(!users[msg.sender].exists) {
        _;
    }
}
```

The caller of the function must be new user, who is not already saved in the contract.

4.4.5 Constructors

```
function SocialRecord(){
      owner = msg.sender;
      // set the contract creator
      users[msg.sender] = User("Creator", true);
  }
```

The constructor is identified by the capital letter, and it is the same name as the contract. The users map is added with a new address from the contract creator. The address of the creator is stored in the contract: he is the only one allowed to kill the contract and terminate all the login, sign up and get functionalities.

4.4.6 Functions

The first check that needs to be done is to check if the globalId has already been taken (if the condition verifies then return false). Otherwise, the globalId is added to the *srs* map with the social record object (the social record string received in input and the boolean set to true: existing). Then the social record is returned.

getSocialRecord

```
function getSocialRecord(string _globalId) constant returns (string) {

    if (!srs[_globalId].exists) {
       throw;
    }

    return srs[_globalId].socialRecord;
 }
```

This functionality is characterised by the word *constant*, which means that the instructions contained into the function will not change the state of the blockchain: therefore this function can be performed for free. The function, in particular, receives one globalId as a parameter. If the globalId is not passed as a parameter, it throws an exception, otherwise it returns the requested social record.

updateSocialRecord

```
function updateSocialRecord(string _globalId, string _socialRecordString) returns

      // the user is the owner of the social record
      if (sha3(users[msg.sender].globalId) != sha3(_globalId)) {
          return (false, "This account is not allowed to modify the social record
      }

      // the Social Record must exist
      if (!srs[_globalId].exists)  {
```

```
            return (false, "The specified social record doesn't exist");
        }

        srs[_globalId] = SR(_socialRecordString, true);
        // Trigger the event
        SocialRecordUpdated(msg.sender, _globalId, _socialRecordString);
        return (true, _socialRecordString);
    }
```

Two checkers are implemented, one checking if the user is allowed to update the globalId (basically the sender must be the owner and only him can update his social record). The second check is about the existence of the social record in a similar manner described in the *getSocialRecord* function.

Then the *srs* map is changed at index which equals to globalId with a new and updated structure.

deleteSocialRecord

```
    function deleteSocialRecord(string _globalId) returns (bool) {
        delete srs[_globalId];
        delete users[msg.sender];
        return true;
    }
```

The users that call this function can only delete their own social record.

kill

```
    function kill() onlyOwner {
        selfdestruct(owner);
    }
```

In this function one modifier is specified, in particular the *onlyOwner* modifier; in fact, only the creator of the contract can terminate the contract itself.

4.4.7 Events

There are 3 events that are implemented

```
event SocialRecordAdded(address user, string globalId);
event SocialRecordUpdated(address updater, string globalId);
event SocialReocordDeleted(address deleter, string globalId);
```

As each of the names of the events suggests, the events are invoked in case of Social Record creation, update or deletion.

5 Evaluation

5.1 Performance

5.1.1 Get

The get operations are nearly instantly, since the state of the blockchain does not need to be changed. The get functionalities are therefore very similar to the DTH.

5.1.2 Update

The time needed in order to update one social record is not predictable, could be almost instant (if it is directly inserted in a block) or never being mined. One of the key factors in order to have the transaction mined is to increase the transaction price. In fact, miners will be willing to put expensive transactions in their block, aiming to maximise the rewards of the proof of work. An expensive transaction is a transaction with an high *gasPrice*, meaning that who build the transaction is prone to pay one particular price per each gas. Another factor is the *gasLimit*, saying that the total theoretical price for one transaction is given by $maxPrice = gasPrice \times gasLimit$. In the reality, the unused *gasLimit* is returned to the creator of the transaction. For this reason and for the constant changes that are affecting Ethereum, it has not any validity to create a benchmark trying to find an average updating time [9]. It is nevertheless reasonable to estimate it in 1–3 minutes.

5.2 Security

If on the performance side this new approach may appear not optimal, the security is strongly enhanced in compare with the previous DHT implementation.

It has been proved in the concept section the security of this approach, in particular:

Bigger Node: Since the social record is stored in one of the main public blockchains, it is very unlikely to perpetrate an attack aiming at the 50%+1 of the network (proof of immunity).

Contract: The contract has a consistent number of checkers in order to stop attack to change one particular social record (functionalities that were not available within the DHT.

Client Server: Not only the storying method is secure but also the communication between the client and the server since the transactions are sent already encrypted.

Malicious GSLS: in this approach a malicious GSLS server cannot change the content of one social record nor upload an old one (due to the expired nonce of the transaction). The only feasible attack is to send false information to the client (e.g. nonce), resulting in an invalid transaction. Nevertheless the open source nature of the sonic project allows to arbitrary choose the GSLS server the user wants to link to. It will be therefore only necessary to change the server location in the desktop application settings.

6 Conclusion

In conclusion, we have managed to implement the project according to all the requirements and constraints defined. The project has been implemented with three separate entities: Unfortunately after few considerations, this approach seems to be optimal:

- The Desktop client is implemented using Electron framework
- The GSLS Server using Java Spring-Boot for APIs implementation and Web3j for communication with the Ethereum client
- Ethereum Testnet node, for accessing the Ethereum Blockchain

We can say that we have encountered more problems that we have initially anticipated and we had to do a lot of research and trials in order to overcome the problems. Some of the biggest challenges we have faced was lack of information on the Internet since it is a new technology and lack of features of Ethereum that we thought will be supported by just knowing Blockchain in theory.

The biggest challenge was to find the appropriate features and architectural design in order to satisfy all the project requirements about the features and the security of the system. Great part of our work included research of the many new blockchain technologies and their possible application in our project.

The major constraint that we wanted to overcome was that we did not want each End User (Client) to be obliged to have Ethereum client on it's machine in order to be able to communicate with the Ethereum Blockchain. We found the solution to this challenge in signing the raw transactions offline and created a user friendly client application that takes care of this.

While working on this project, we have learned that the Blockchain technology is still very new and under heavy development with many new implementations and use cases emerging almost on daily bases. Additionally we faced a lot of limitations compared to the "regular" programming patterns in terms of features that are not supported while programming contracts in Solidity.

Moreover, since the technology is relatively new, the online community and support is not that strong yet and we had to come up with our solutions to most of the challenges we have faced.

To sum it all up, even though the team had a great challenge to overcome the limitations of the Blockchain technology in our use case, we have found and implemented a working solution according to the requirements. A lot of research had to be done, and in the end we have successfully implemented the project.

A Code

Bibliography

[1] *Blockchain for Identity Management*. 2017. url: https://letstalkpayments.com/22-companies-leveraging-blockchain-for-identity-management-and-authentication/ (visited on 05/14/2017).

[2] BlockStack. *BlockStack architecture*. 2017. url: https://blockstack.org/whitepaper.pdf (visited on 05/06/2017).

[3] *Elliptic Curve Implementation in Solidity*. 2016. url: https://github.com/jbaylina/ecsol (visited on 06/29/2017).

[4] Ethereum. *A Next-Generation Smart Contract and Decentralized Application Platform*. 2017. url: https://github.com/ethereum/wiki/wiki/White-Paper (visited on 05/06/2017).

[5] Sebastian Göndör and Hussam Hebbo. "SONIC: Towards seamless interaction in heterogeneous distributed OSN ecosystems". In: *Wireless and Mobile Computing, Networking and Communications (WiMob), 2014 IEEE 10th International Conference on*. IEEE. 2014, pp. 407–412.

[6] Sebastian Göndör et al. "Distributed and Domain-Independent Identity Management for User Profiles in the SONIC Online Social Network Federation". In: *International Conference on Computational Social Networks*. Springer International Publishing. 2016, pp. 226–238.

[7] *Introduction to Identity Mangement*. 2003. url: https://www.sans.org/reading-room/whitepapers/authentication/introduction-identity-management-852 (visited on 05/14/2017).

[8] *Support RSA signature verification*. 2016. url: https://github.com/ethereum/EIPs/issues/74 (visited on 06/29/2017).

[9] *Why is the average block time 17 seconds?* 2016. url: https://ethereum.stackexchange.com/questions/58/why-is-the-average-block-time-17-seconds (visited on 07/20/2017).