Technische Universität Berlin

Documentation

IOSL: Distributed Identity Management in the Blockchain

Authors:

July 20, 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 for 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 with 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 OSN [3] allowing the migration of accounts between different platforms.

The project is supported by a distributed and domain-independent ID management architecture where a GSLS (Global Social Lookup System) is employed to map Global ID to URL of a social profile. This mapping is possible thanks to datasets called Social Records which are digitally signed.

1.1.2 Blockchain

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.

1.1.3 Identity Management

Add the references

Identity management refers to the process of employing emerging technologies to manage information about the identity of users and control access to company resources[5]. 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[4].

The problem in these digital platforms is the absence of federated directories.User' often forget sign credential when they have many different ones. It also increase 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[5]."

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.

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

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).

Implementations Considered

Kademlia

TomP2P

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 contentaddressed 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 sorage 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 lineat scaling throuhput, an efficient query language and permissions in order to interact with it. In particular developing a MongoDB based NoSQL distributed database. With the foundamental 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: ünder the hood, Blockstack provides a decentralized domain name system (DNS), decentralized public key distribution system, and registry for apps and user identities[̈add reference].

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 **??**.

Why did we choose Ethereum

2.5 Solidity

Solidity is a high level programming language developed specially for Ethereum block chain. Solidity introduced smart contracts which is a base for different use cases like supply chain platforms, Banks, health care system etc. Smart contracts are executed by a computer network that uses consensus protocols to agree upon

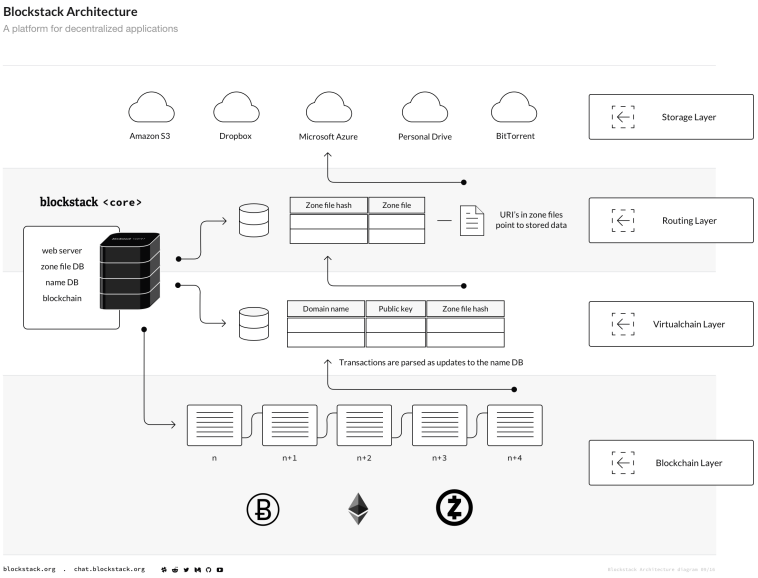

Figure 2.1: Basic Blockstack architecture

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 lot more features. Syntax of solidity language is similar to java script. Due to that reason web developers and programmer can adopt it in very short period of time.

Solidity is still in its early stage while non Ethereum projects are also involved to use solidity. Every language has their own weaknesses while in solidity a lot of things to be improved in near future. Theoretically it's easy to talk about in idea but in reality it's impossible to implement every idea in solidity. Solidity was the best fit for our project "identity management in block chain" through which we created contracts for Ethereum block chain.

2.5.1 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.

2.5.2 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

3.1 Approaches to Store Social Records

3.1.1 Store the Social Record

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

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

3.1.2 Store 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 transaction holds in the data field the hash of the Social Record. 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 currupted record.

3.1.3 Use a third-party solution

Blockstack offers an already implemented solution using a layered architecture similar to the one proposed in solution 2. It allows also 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].

3.2 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 not centrally managed server, a safety threat may arise. In order to ask to the server to conclude 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 mentioned above RSA and ECC).

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

In case of 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 DHT social record and check it with the variable stored in the blockchain for that particular user. In order to update the information is required to encript the following object with the user's private key:

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

So the contract is able to open the message, check if the globalId match with the one of the user and update the record.

In case of we store the entire Social Record

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

The record to store is similar, the only difference is that rather than the hashedSocialRecord string, the contract is storying the socialRecord Object (which imply 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 sistem 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 follow:

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

EncriptedData is a string that can be decrypted with the public key and that will be declared authentic if the result of the decrpytion will be a JSON object as follow.

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

This approach, even though complicated at first, solve 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 the ethereum node and cannot rely on the authenticity of the server, they are impossibilitated to update their record.

It is a common practice to trust the server identity of every service (and when it is not possible to trust the service, it is still possiblo to write in the blockchain), but with this approach the writing/reading problem can be solved even without trusting anyone and without hosting any blockchain.

Unfortunately, so far, an asymmetric library in solidity has not been developed yet.

Event 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 [2] [6] 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.

3.3 Dictionary / Nomenclature

TODO: Make it table

User- The find entity that owns a social record, concrete it, upload, deliver or retrieve it.

Server- The GSLS system can be converted to an Ethereum node with few more functionalities such as an API request interface listening for communication from the client.

Blockchain - the Blockchain is referred as an entity where to store records with transactions.

Transaction- Is an operation that aims at changing the status of the BC through authorized inputs.

Contract - Operations that needs to be performed in particulars contracts when not specified in the doc contract and solidity contract are considered synonyms.

3.4 Use wallet 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 metodology 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 Keep the DHT and verify the authenticity of the Social Record with the blockchain

The first approach wants to be cheap and light.It will keep the DHT implementation and aims to integrate the Blockchain in order to detect malicious charges in the SR stared in the DHT itself.

How it works

Creation & Update This implementation does not require the client to host an Ethereal 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 will upload his keys), 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 string of small dimension. The hash is then uploaded in the blockchain through the server in a Key value array with the Old 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 Global iD from the DHT and the hashes from Blockchain.

If the hashes SR of the Blockchain is the same as the hashing of the retrieved from the DHT, the record is pure.

This approach is valid and cheap since only the Hash of the entire SR is stared. 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 tries to merge the security and immutability of the Blockchain without forcing the User to have a wallet or Ethereum account nevertheless is important to keep in mind that the server that contacts the bc must move a node. In our prototype we can imagine the GSLS as a Server hosting in Ethereum Node and implementing some functionalities explained here bellow.

How it works

Creation The user downloads the software, 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 send to the server.

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

The server with the (Server wallet) uploads the transaction to the blockchain. The contract stored in Ethereum will decrypt the encrypted SR and store the Key Value array with the Global Id as key and the object (Public key, SR version 1) as Value.

Update In order to update SR, the User application needs to retrieve the number of version, encrypt the number of version + 1 and send the following object to the server (that will update it to the blockchain). The Solidity contract will decrypt the version and check if it match the version +1 in the Blockchain , if it match, the SR can be decrypted as well and update the record in the BC.

GET Get is pretty trivied, asking for a Global Id, the SR will returned. This approach even if valid (although expension is not visible because of . . . .)

3.5.3 User with Ethereum Wallet

This approach require the Suers to have an Ethereal wallet but not to run an Ethereal node. This is optimal because it allows the users to update their credentials even without hybrid solutions that port of the the bc (such as ....) Ideally, it is possible to create and change the record even from a random hosting. Pc or mobile device.

It is enough to provide the wallet file .j son file.

How it works

Creation & Update Once the wallet file json loaded on the application its is possible to create the social record, sign it with the private key; creating the transaction normally. This signed transaction is then uploaded to the server and then to the BC wallet file json.

Json file with the private key of the Ethereum wallet account (and other information).

In a Key Value array as follow Gid–SR

GET (same as previous)

Needs to be done

This approach force the users to deliver an Ethereal wallet but it conceptually more correct because it uses the offline transaction from —— link to yellow white paper.

4 Implementation

4.1 User side

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

4.1.1 POST /socialrecord & PUT /socialrecord

The User can create a new social record or change an existing one (of course only if he is allowed to). When the user click 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 in order to set it, 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**: 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 bit are reserved for the first 32 bit of the hashed function name (that the user wants to call). Then, since in the ABI File is specified the number of parameters required from the function (and their type): is possible to grok the bytes needed to append to the data string. In the case of the creation of a new social record, 2 string parameters are required, so they will be both converted to hex and appendend 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 see 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.

4.1.2 GET /socialrecord/:GlobalID

The get request in order to retrieve the social record of one particular GlobalID is a simple promise that contact the server specifying the global id and wait for the result.

4.2 Server

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

4.2.1 POST /socialrecord & PUT /socialrecord

The creation of a new social record happens accordingly to the following steps

- GET NONCE

```
TODO
```

- 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.

4.2.2 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 and then it sends it back to the cleint 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).

4.3 Contract

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

4.3.1 Structures

In the created social record contract there are 2 structures:

SR

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

The social record structure allows to store one string and to set one boolean value to mark if the afore mentioned social record is valid.

User

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

In the same way it allows to store one globalId and to mark if one specific globalId has been market as existing.

4.3.2 Variables

owner

```
address owner;
```

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

4.3.3 Mappings

users

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

The mapping *users* maps each address with one *User* structure.

srs

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

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

4.3.4 Modifiers

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

onlyOwner

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

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

4.3.5 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 the same name as the contract. The users map is added with a new address and construct the contract. 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.

4.3.6 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 funtion, in particular, receives one globald as a parameter, if the globalId is not taken it throw an exception, otherwise it returns the social record requested.

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 can only update his social record). The second check the existence of the social record in a similar manner descibed in the *getSocialRecord* function.

Then the *srs* map is changed at index 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.

EVENTS ARE MISSING

5 Evaluation

6 Conclusion

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

[3] 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.

[4] 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.

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

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