Technische Universität Berlin

Censorship-free Messaging using Blockchain Technology

Martin Westerkamp, Sebastian Göndör
Supervisors

Antal Száva, Charul Daudkhane, Hai Duc Dang, Ferhat Saritas
Team members

Berlin, 2018

TELEKOM INNOVATION LABORATORIES

Service-centric NETWORKING

Contents

Chapter 1

Introduction

1.1 Overview of the application

Blockchat is a user-friendly censorship free chatting app which is secure, private and encrypted. Based on Blockchain technology, Blockstack helps to solve the problem associated with centralized Chatting Applications.

Centralized Messaging Approach

The most popular chatting apps today are centralized: WhatsApp, Skype, Viber, Telegram. With a centralized application, a single cluster of servers contains all logic to conduct the necessary steps. The cluster receives the request, processes it, saves what it needs, and returns the correct response. Due to the central access they suffer from-

- downtime
- security breaches
- zero anonymity/privacy

Decentralized Peer-to-Peer Messaging

In a decentralized environment, messaging is conducted in a peer-to-peer manner. This means that messages are sent from a device directly to another. From the perspective of the user, the main benefit of this is that they can be sure no one else has access to their messages. Since there is no central server to collect, store, and distribute messages, there is no opportunity for the platform owner to eavesdrop on the users. In fact, there is no "owner" to speak of.

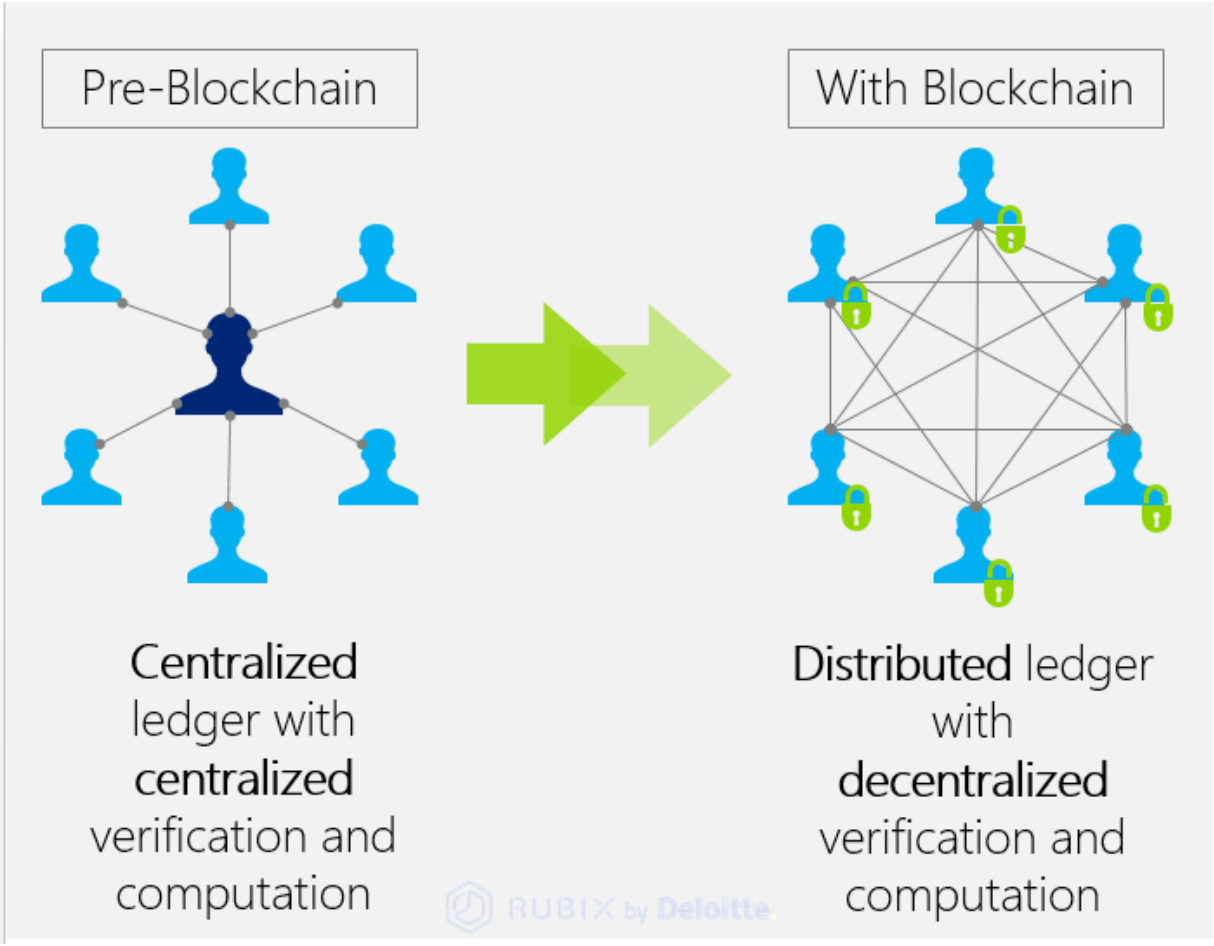


Figure 1.1: Comparison of the centralized and decentralized approaches DecMessagingImage

Federated architecture

In a federated architecture different parts of the system and different technologies are brought together. With this, a public service can be used, although parts of the architecture are kept private for each component.[11]

1.2 Other blockchain messaging applications

1.2.1 Status

Status is a blockchain-based app that operates as a messenger and decentralized browser. It allows their users in addition to chatting the sending of Ethereum-based payments and smart contracts to friends. As it is blockchain based, all chats on the platform are encrypted by default. In addition to that, the user can interact with Dapps running on the Ethereum Network.

1.2.2 Whisper

Using Whisper, clients of the Ethereum blockchain can send peer to peer messages to each other. This protocol is independent of the Ethereum blockchain, thus there is no access to the smart contracts of the network.

Chapter 2

Research background

2.1 Blockchain

The Blockchain still has no statisfactory definition yet but the ISO/TC 307 describes a Blockchain as follows: [Blockchain is] *a shared, immutableledger that can record transactions across different industries, [...] It is a digital platformthat records and verifies transactions in a transparent and secure way, removing the needfor middlemen and increasing trust through its highly transparent nature*

In practice a what this means is that a Blockchain is a distributed computer architecture, where each computer is treated as a node of this network of nodes. Each node has knowledge of all transactions inside the network. Transactions are encrypted and bundled in a so called block. Only one block at a time can be added to the network because it has to be verified that it follows the previous blocks first.

Nowadays a Blockchain have the following traits:

- Distributed: Each node is considered equal and has the full history of all transactions
- Time-Stamped: Consequence of this is that it with each new block it becomes harder to change past entries because they are being build on the information of past blocks.
- Consensus: Through a consensus the nodes know which distributed database they need to address.

For this project we make use of an Ethereum Blockchain which has a virtual machine already built in and allows to deploy our own smart contracts. It is a Blockchain

specifically designed to run distributed applications and acts for those applications like the Internet for web applications.

2.1.1 Cryptography

The secure way of information transfer is top priority in certain use cases. In order to ensure that a certain entity of the network is really the one who he/she claim to be, a piece of information is really coming from this desired sender and that it was not tempered with in any way, there are certain characteristics that need to be provided. These characteristics are described by the concepts of integrity, authentication and non-repudiation.

- Integrity implies that a piece of data remains unchanged while being exchanged.
- Authentication is the process of making sure that a certain party is really someone who he/she claims to be.
- Non-repudiation is described in the following way: "the assurance that someone cannot deny the validity of something". [3]

This means that the initial needs are there, once non-repudiation is provided. Cryptographic hash functions With the use of a cryptographic hash function, it can be ensured that a piece of information is authentic. This algorithm involves transforming a certain piece of data into a so called checksum (unique string). With this approach, we can claim two inputs to be exactly alike in the case when the outputs after applying cryptographic hash function on them separately, are the same. Cryptographic hash-based message authentication is one way to address the problem of secure information . Once a cryptographic hash function and a certain key that is exchanged between two entities are used, integrity and authentication is provided. However, as the key is publicly shared, non-repudiation is not yet provided. [4]

Assymetric encryption and digital signature With the approach of assymetric encryption each client of a certain architecture has its own so called private and public key. Private and public keys are unique keys with which piece of data can be signed digitally. Private keys are solely possessed by clients whereas the belonging public keys are distributed in the network. Whenever someone wants to send a message, first it encrypts the message using the intended receiver's public key, then uses own private key to encrypt again. The receiver would receive this encrypted message, would decrypt it first with the sender's public key (each client would be

able to access every other client's public key) and then uses the own private key to decrypt. With this technique, non-repudiation is provided also. Breach in security only arises once the private key of a certain user is taken by someone.[5]

2.1.2 Consensus algorithm

Consensus decision-making is a group decision-making process in which group members develop, and agree to support a decision in the best interest of the whole.

The first consensus algorithm for cryptocurrencies is Proof-of-Work (PoW). PoW is based on the assumption that work must be put into something to give it value, and in Bitcoin's case this work is doing computations with mining Mining serves as two purposes:

1. To verify the legitimacy of a transaction, or avoiding the so-called double-spending;
2. To create new digital currencies by rewarding miners for performing the previous task.[6]

Following things happen on setting a transaction-

- Transactions are bundled together into what we call a block;
- Miners verify that transactions within each block are legitimate;
- To do so, miners should solve a mathematical puzzle known as proof-of-work problem;
- A reward is given to the first miner who solves each blocks problem;
- Verified transactions are stored in the public blockchain

All the network miners compete to be the first to find a solution for the mathematical problem that concerns the candidate block, a problem that cannot be solved in other ways than through brute force so that essentially requires a huge number of attempts. When a miner finally finds the right solution, he/she announces it to the whole network at the same time, receiving a cryptocurrency prize (the reward) provided by the protocol. From a technical point of view, mining process is an operation of inverse hashing: it determines a number (nonce), so the cryptographic hash algorithm of block data results in less than a given threshold. Some of the other consensus algorithm include Proof of Stake, Proof of activity,Proof of capacity,Proof of elapsed time etc [7]

2.1.3 Ethereum and smart contracts

Digital solutions enabled new ways of insurance when it comes to contractual agreements. Smart contracts digitally execute the terms and conditions that were recorded in contracts by running program code. This way it is ensured that points of the contract are fulfilled and are not corrupted in any way.

Ethereum was the first cryptocurrency to incorporate smart contracts. These contracts are executed automatically once certain criteria are fulfilled. Ethereum's initial goal was to easen the creation of decentralized applications. Developers can leverage the Turing-complete programming language that is provided for Ethereum to program the logic for the application. According to the white paper of the cryptocurreny, there was a focus on five key elements:

- Simplicity: this characteristic is followed even in some cases for the price of lower storage and time efficiency. No extra complexity is to be brought, only if for fundamental optimization purposes.
- Universality: there are no limiting functionalities with Ethereum, through the internal Turing-complete scripting language which it provides, any use case can be implemented
- Modularity: the Ethereum protocol can be broken up into components which function independently of each other
- Agility: characteristics of the protocol may be prone to change, however only in cases when it proves to have a high potential for scalability or security
- Non-discrimination and non-censorship: the implementation of any use case is permitted, a fee is paid for each application running on top of Ethereum

Each transaction contains "standard" fields that are stored in the blockchain: information on the recipient, the signature of the sender, a data field that may be used when deemed so and the number of ether to be transferred with the transaction. Apart from these fields, there are two more values: STARTGAS and GASPRICE. These describe the value that are used for computations. Each computation of a transaction costs a certain amount of gas and there is also starting value. This is needed so that no DoS or other malicious operations leveraging infinite loops can be carried out in the network. [9]

Ethereum Virtual Machine (EVM) The Ethereum Virtual Machine (EVM) is the environment where the smart contracts of the Ethereum blockchain are processed during runtime. It is a completely separate from other networks. The contracts of

the Ethereum blockchain are compiled into a binary format and then stored in the Ethereum blockchain. The EVM is Turing complete, meaning that any mathematical computation can be processed by it. [10]

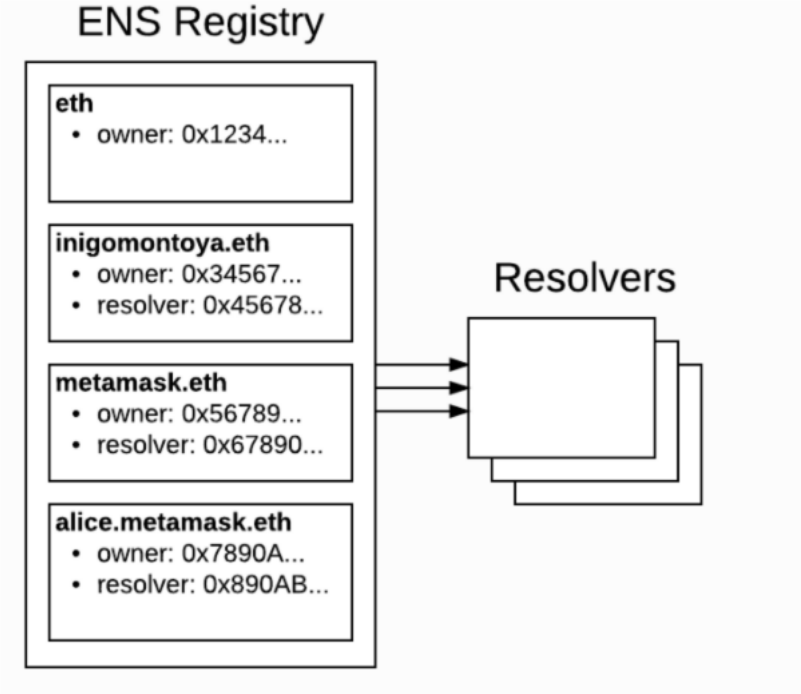
2.1.4 Ethereum Name Service

The Ethereum Name System is an open, distributed service running on the Ethereum Blockchain. It basically maps an Ethereum Address to an human-readable domain name, similiar to the DNS. The toplevel Domain of ENS is .eth.

ENS consists of 2 components:

- ENS Registry
- ENS Resolver

Both of them are smart contracts.



ENS Registry is a smart contract, which maintains a list of all domains and subdomains. For each of these exist 3 pieces of critical information:
- the owner of the domain
- the Ethereum address of the resolver of that domain
- the time-to- live for all records under that domain

ENS Resolver is a smart contract, which translates or resolves the Ethereum Domain Name to an Ethereum Address, which can be the Ethereum Address of the

owner or the address of another smart contract. ENS Registry maintains a list of all domains and subdomains. [1]

2.2 Technologies in use

2.2.1 Truffle

Truffle is a development environment, testing framework and asset pipeline for Ethereum, aiming to make life as an Ethereum developer easier.

Features of Truffle:

- Generators for creating new contracts and tests (like rails generate)
- Instant rebuilding of assets during development (truffle watch)
- Console to easily work with your compiled contracts (truffle console)
- Script runner that lets you run JS/Coffee files with your contracts included (truffle exec)
- Contract compilation and deployment using the RPC client of your choice.

2.2.2 Ganache

Ganache is a personal blockchain for Ethereum development you can use to deploy contracts, develop your applications, and run tests. It is available as both a desktop application as well as a command-line tool.

2.2.3 Metamask

MetaMask is the easiest way to interact with Ethereum based DApps using in a browser. It is an extension for Chrome or Firefox that connects to the blockchain without running a full node on the browser's machine. It can connect to the main Ethereum network, any of the testnets (Ropsten, Kovan, and Rinkeby), or a local blockchain such as the one created by Ganache through injecting a Javascript library accessing the Ethereum network. We chose to work with MetaMask as it provides a seamless user experience and it is widely supported by the community.

2.2.4 NodeJS

Node.js is based on V8 engine developed for Chrome browser by Google corporation, as a framework for javascript execution Node.js uses non-blocking, event-driven I/O

to remain lightweight and efficient in the face of data-intensive real-time applications that run across distributed devices.

2.2.5 MongoDB

MongoDB is an open source database that uses a document-oriented data model. Instead of using tables and rows as in relational databases, MongoDB is built on an architecture of collections and documents. Documents comprise sets of key-value pairs and are the basic unit of data in MongoDB. Collections contain sets of documents and function as the equivalent of relational database tables.

2.2.6 ExpressJS

Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications. Following are some of the core features of Express framework:

- Allows to set up middlewares to respond to HTTP Requests.
- Defines a routing table which is used to perform different actions based on HTTP Method and URL.
- Allows to dynamically render HTML Pages based on passing arguments to templates.

2.2.7 VueJS

VueJS is a progressive framework for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects.

2.2.8 Webpack

It's a module bundler for modern JavaScript applications. When Webpack processes your application, it recursively builds a dependency graph that includes every module your application needs, then packages all of those modules into a small number of bundles - often only one - to be loaded by the browse.

Chapter 3

User Documentation

The steps taken in this chapter are done in a Ubuntu 16.06 environment running a Chrome browser of version 67.0.3396.87 (Official Build) (64-bit). Though configuration steps and details in other environments may differ, they are not prone to any arising difficulties arising because of any implementation details.

3.1 Prerequisites

For our application to be run certain pieces of software are to be configured in the system. The following is a list of these applications with a brief piece of detail for their installation process:

- MongoDB: follow installation details from the official site (`https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/`).
- NodeJS + NPM, node Version 8.9.1 is needed: follow installation details from the official site (`https://nodejs.org/en/download/package-manager/`).
- Truffle: execute the following command

```
1    npm install -g truffle@4.1.11
```

- Ganache-cli: execute the following command

```
1    npm install -g ganache-cli
```

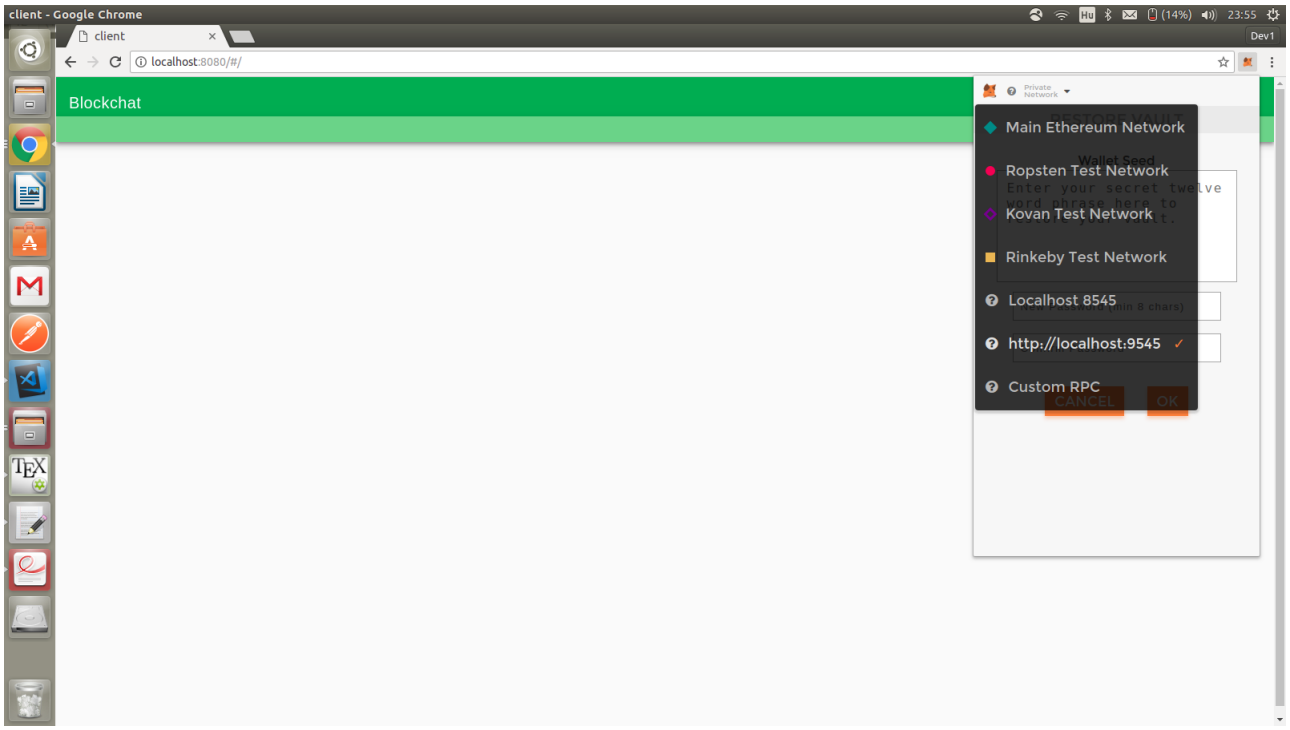
- MetaMask: follow installation details from the official site for the Chrome extension (`https://metamask.io/`).

3.2 Using the application

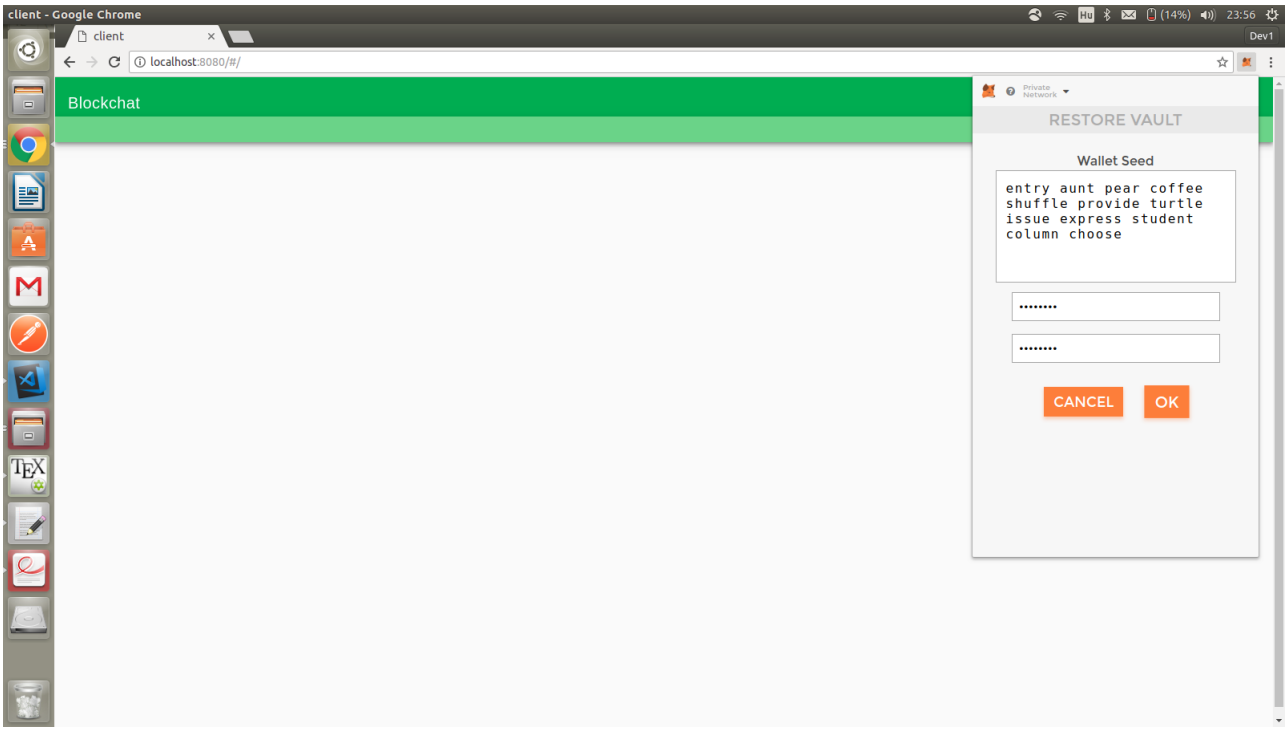
We are assuming that there is an Ethereum blockchain service running and that there are servers of BlockChat to which users can connect to.

3.2.1 Signing up in MetaMask

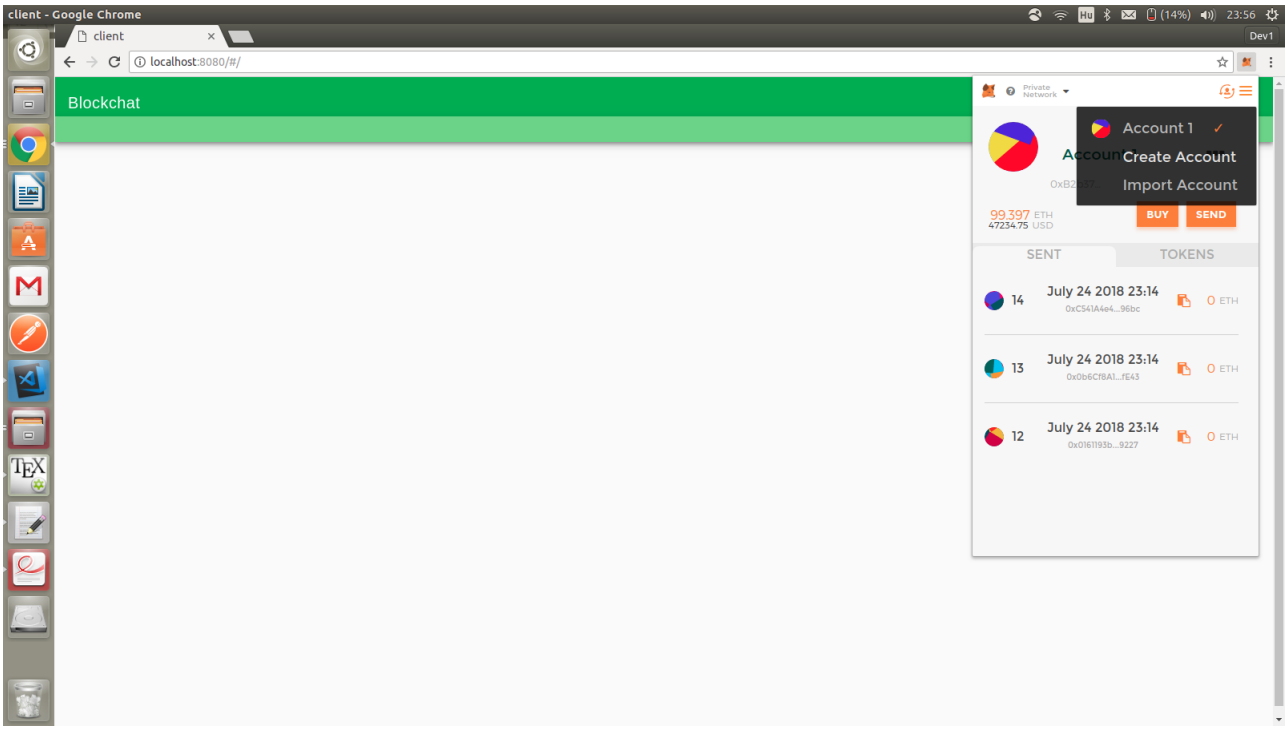
1. Opening a browser with installed MetaMask extension (suggested browser is Google Chrome).



2. Signing up to running Ethereum network through MetaMask (a local Ganache blockchain listeninig on port 9545 is used for development)



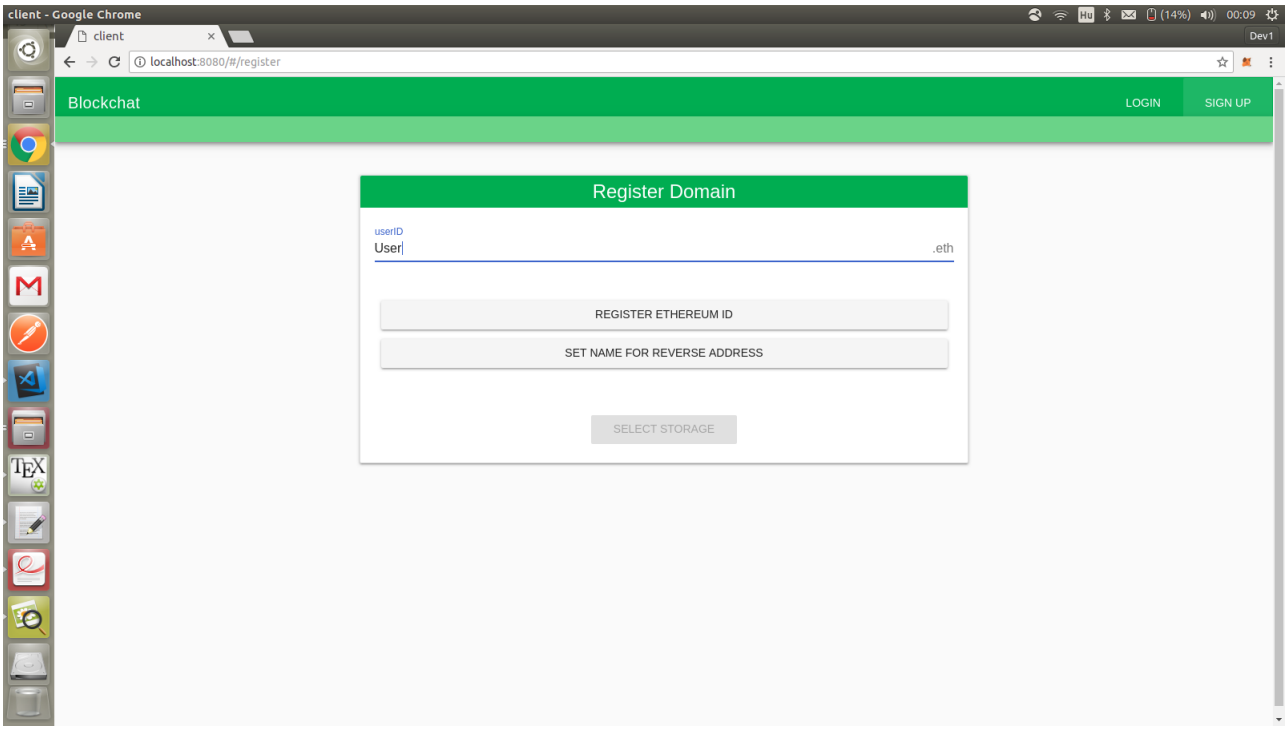
3. Creating a new account for the network.



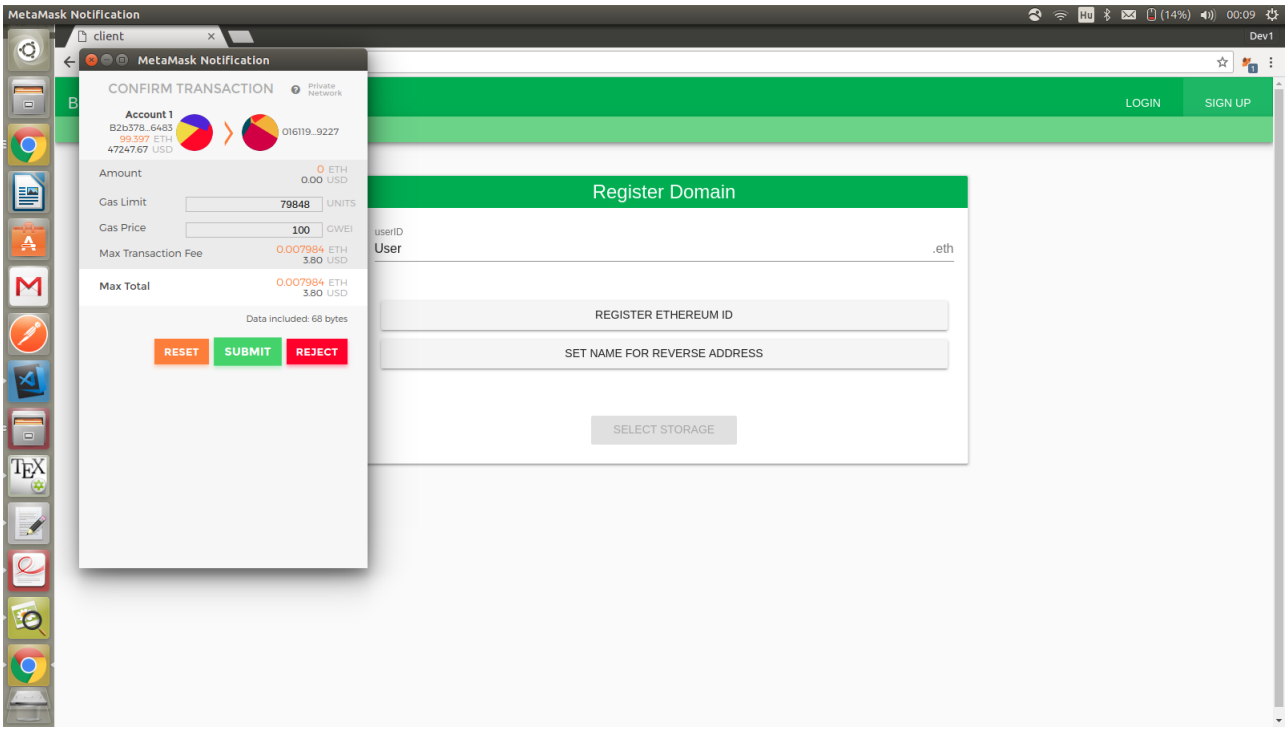
3.2.2 Signing up into BlockChat

1. Visit the designated URL for BlockChat in the browser (used local instance for development).

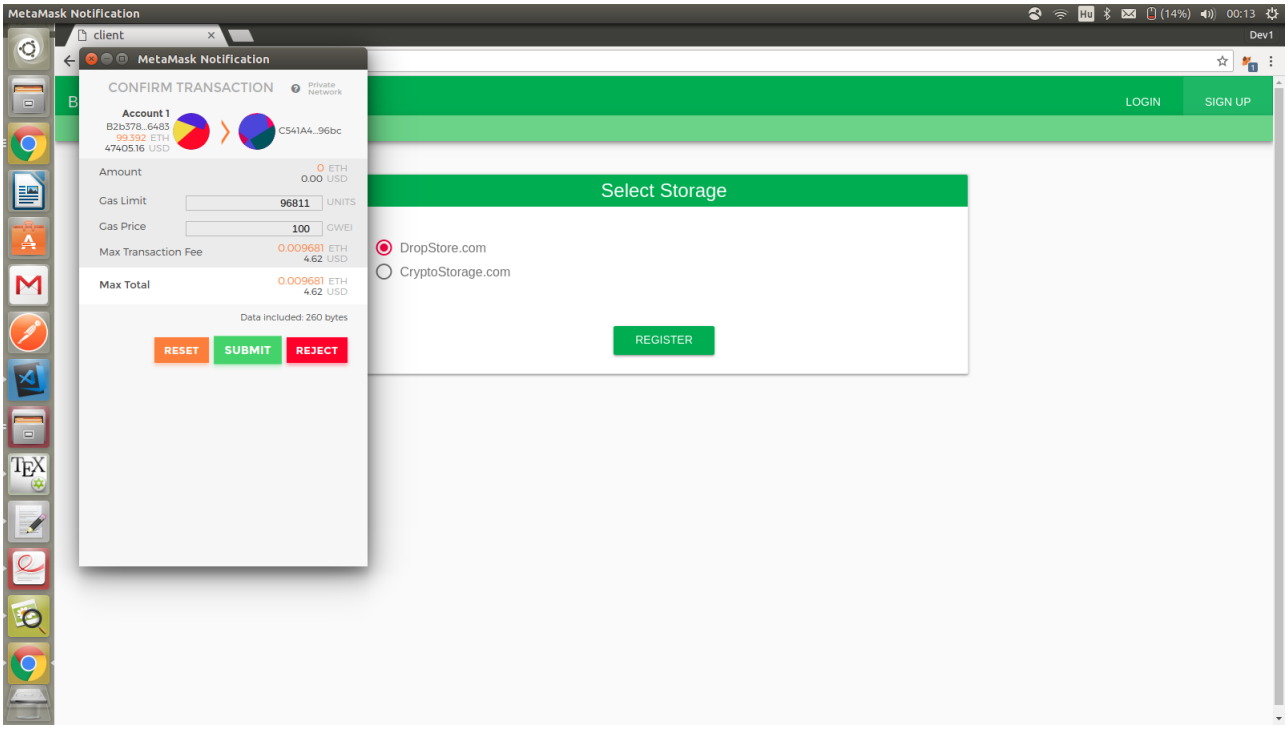
2. Click on the Sign Up button and add a customized username.



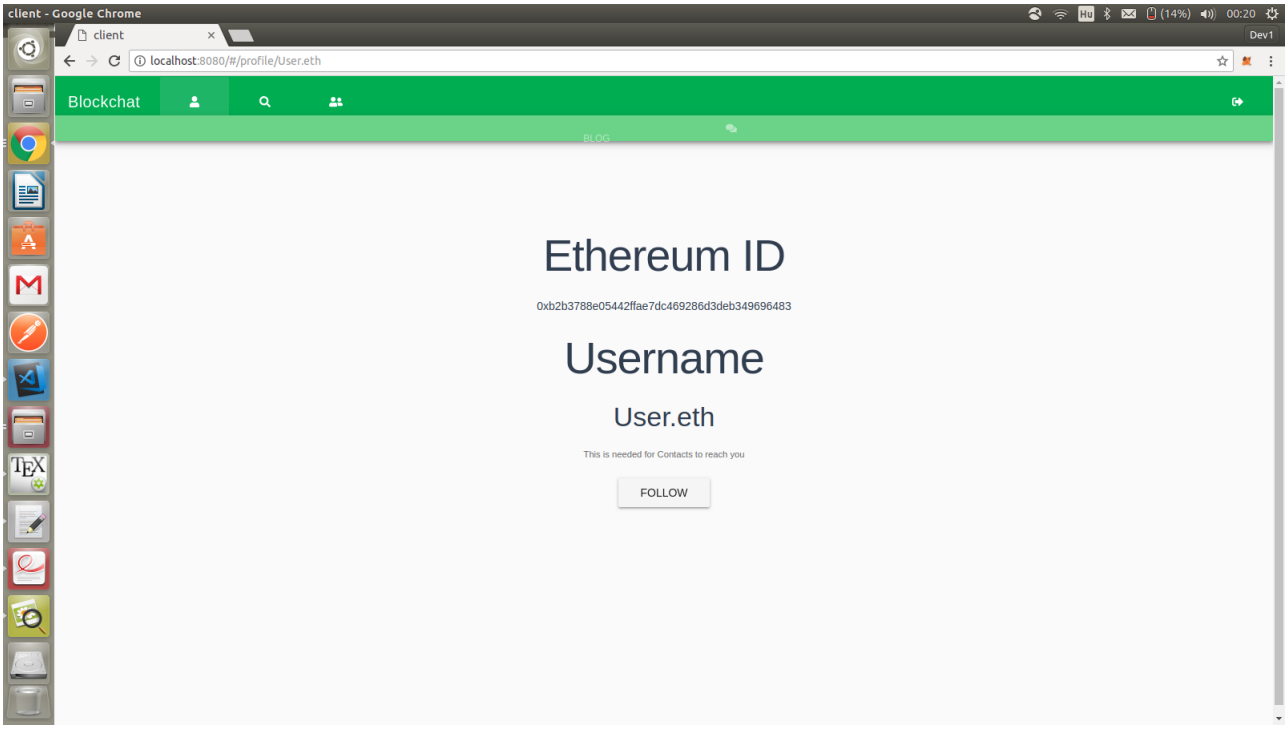
3. Register the Ethereum address and set a Reverse address. Each operation needs to be signed through the pop-up window from MetaMask.



4. Select a specific server where your messages, contacts and blog entries are going to be stored.



5. Congratulations! You have just created your BlockChat account and logged into our service. You can return to this page any time for specific details on your profile.



3.2.3 Using BlockChat

LogIn

For logging in to BlockChat you will have to click on the designated button and sign your request through MetaMask.

Adding a friend

You can add a friend as a contact by clicking on the magnifying glass icon and typing their Ethereum name. Having found them they can be added by clicking on the "Follow" button. You will have to sign your request through MetaMask.

Adding a friend

Having added a friend to the contact list, you can send messages to them by clicking on the messaging icon. Most of the times you will have to sign your first messages to a new user through MetaMask. Your friend will receive your message once he/she is only.

Accepting a friend request

You may receive friend requests from other users. It is your choice to either accept or decline them. You can look for new friend requests by clicking on the requests icon to the right from the search icon.

Creating a blog entry

You may create a blog entry by clicking on the "Blog" button and typing your text into the text area. Once you press "Enter", all your contacts are going to receive your blog entries once they are online.

Looking at blog entries

Once a friend of yours creates a blog entry, a notification appears next to the "Blog" text indicating the number of new blog entries. Once you click on the "Blog" button, you will get the latest blogs coming from the users in your contact list.

Chapter 4

Developer Documentation

4.1 Blockchain

In order to use our Application, the user has to have a Domain Name registered on the ENS. Since our setup makes use of the local Blockchain Ganache, we deployed the ENS on our local test Blockchain. In the current setup the user is registering a Domain through the FIFSRegistrar, which basically means the first user who registers for a free available Domain Name comes into possession of the Domain Name. Setting up our application for use in production though, requires the user to register a Domain Name on the public ENS through an auction registrar. An instruction for that can be found here: `https://www.myetherwallet.com/#ens`

In our Application, the Ethereum Domain Name, owned by a User will act as an unique Identifier and will be necessary to have in order to register for our service to use. For our service, we modified an Smart Contract, provided by Martin Westerkamp, which basically resolves an Ethereum Domain Name to an storage Endpoint. Since our application is distributed, there will be several different storage endpoints, between which the user can select to store his messages and blog entries on. Upon registration, our web application will therefore communicate with this URLResolver smart contract.

The URLResolver smart contract consists of the following important attributes and methods :

```
1 struct Record {
2         string url;
3         PublicKey pubkey;
4    }
```

```
mapping (bytes32 => Record) records;
```

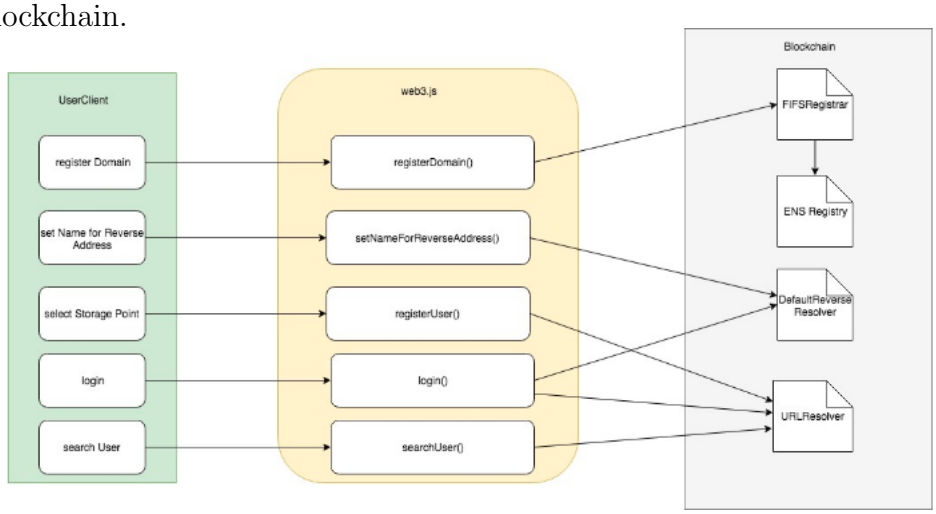
The variable records contains of an array with the following key value pair - ethereum name node mapped to its Record Object, which contains the storage Endpoint Url.

```
function setUrl(bytes32 node, string url, string userName, address
    owner) public only_owner(node) {
        records[node].url = url;
        emit URLChanged(node,url);
    }
```

This function maps and stores the storage Endpoint url to an Ethereum Domain Name (username). Only the owner of the Ethereum Domain Name will be able to call this function. Therefore only users who are in Possession of an Ethereum Domain Name, can use this resolver.

```
function url(bytes32 node) public view returns (string) {
        return records[node].url;
    }
```

This function retrieves the selected url storage endpoint mapped to a specific user. The following diagram gives an overview of how the client is interacting with the Blockchain.



The File web3.js is located in client ¿ services ¿ web3.js and exposes several methods in order for the client to interact with the Ethereum Blockchain. It makes use of the web3 API library, which is a collection of libraries allowing you to interact with a local or remote ethereum node, using an HTTP or IPC connection. As mentioned before, we deployed the ENS on the local test Blockchain, therefore a way of registering a Domain locally has been implemented. Since registering an Ethereum Domain is not a bidirectional mapping, we make make use of the DefaultReverseResolver smart contract, which basically just maps the hash of an Ethereum Domain Name to an Ethereum Domain Name in string format. More practically speaking that

means: Given a Ethereum User set his Reverse Address, one can retrieve the Users Ethereum Domain Name by looking up his Ethereum Address. Our Login Process therefore doesn't need the input of the Username anymore, since the Address is provided by MetaMask and the DefaultReverseResolver gives back the Username. Then the URLresolver will be called, in order to deliver the storageEndpoint, to whom the User should be connecting to in order to retrieve his User Data.

Since our system is distributed, the Client will have to retrieve the storage Endpoints of the Users he/she wishes to connect to from somewhere. This is happening through the URLResolver again, which provides the storage Endpoints given the Ethereum Username. As these endpoints are stored the Blockchain, no one can manipulate the data, since only the owner of an Ethereum Domain Name is allowed to set his storage endpoint.

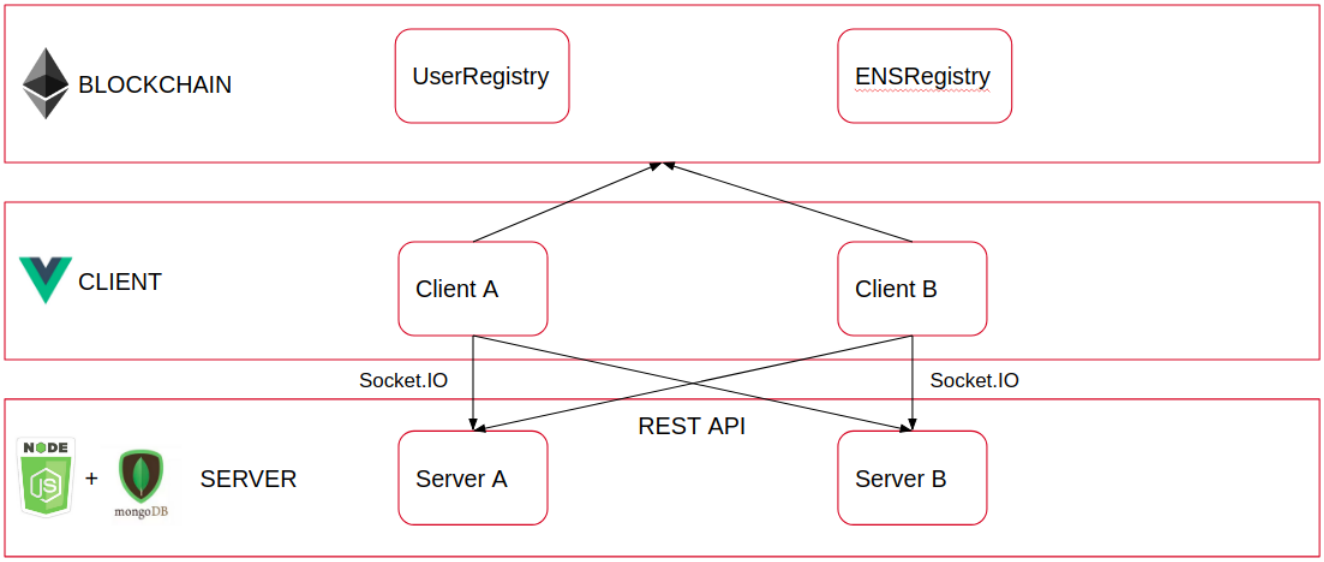
4.2 System Architecture

4.2.1 Concepts of system architecture

- User: a machine connecting to the BlockChat service, upon sign up a User is specifically assigned an Endpoint
- Client: the service provider that allows Users to use the functionalities of the service through the user interface
- Server: service provider that stores messages, contacts and friend requests for specificly assigned Users. Communicates through a socket with its own Users, and it also awaits REST API calls from other Users.
- Endpoint: a specific Server that is assigned to a specific User upon sign up to the service.
- Database: storage accessed by Servers utilising MongoDB technology
- Blockchain: Ethereum blockchain storing the Ethereum name, Ethereum address and Endpoint for each User

4.2.2 Overview of the architecture

While designing the application we thought of how to use the blockchain to empower the user. We wanted the user to have absolute control over his or her data. With that in mind we came up with an architecture displayed in the following image.

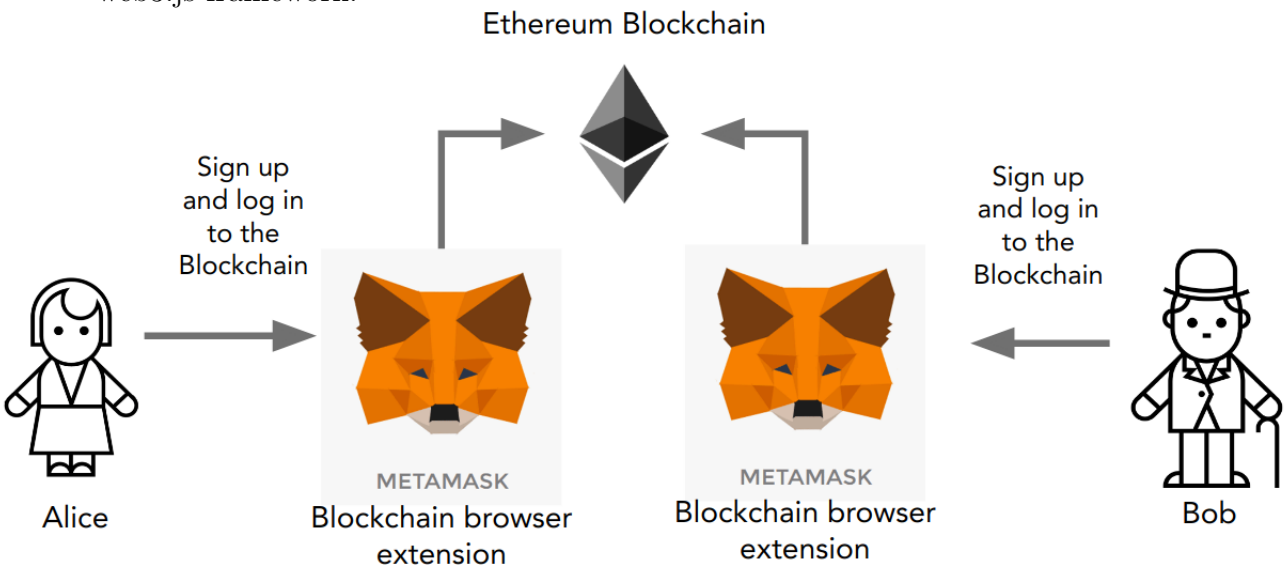


Every user of BlockChat first has to register his eth.id with the url of his or her storage address into the Ethereum blockchain. Through the eth.id other users can find each other in the application. The storage address is the endpoint for BlockChat to store the data of the user and for other users it serves as a connection endpoint. After the user is registered and logged in, a server instance with Express.JS and MongoDB will be running on the storage address and a Vue.JS client instance on the browser. The client accesses BlockChat with MetaMask, which handles all the authentication and is the gateway to our platform. All information shared between users are stored on both storages via Socket.IO and a REST API. Although BlockChat is designed as a distributed application, the basic message transfers are done in a Client-to-Server architecture. This model was created so that we can make use of MetaMask for authentication through REST API calls (see details on authentication later). In the coming sections we will explain in more detail how each core functionality works.

4.2.3 Sign up

As we can see in the figure below, before a user can use BlockChat he or she has to sign up into the Ethereum blockchain. For this they both need MetaMask on their browser, which allows them to browse through DApps and platforms built on blockchain technology. The register form also asks the user to pick a storage address.. On this storage address the server instance will be installed with the assumption that MongoDB is running on the server. If the user is already registered he or she has to authenticate him-/herself every time on the server, when the user wants to log into BlockChat. As said previously all authentication are handled with

MetaMask. All Javascript functions for register for this process are parts of the web3.js framework.

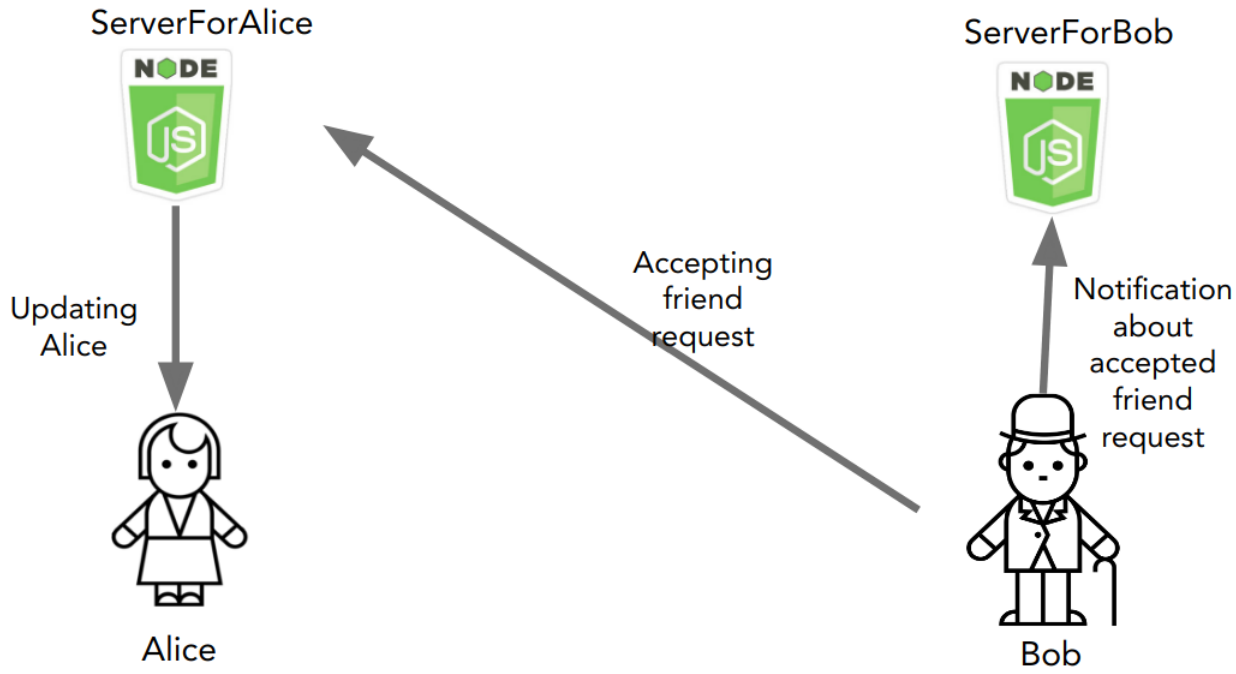


4.2.4 Authentication

Every time a user tries to contact another user in any form or logs in, he or she accesses a specific storage address. This triggers the authentication process, where the server gives an authentication token to the user, if the user succeeds the authentication challenge. With that token the user does not have to take the challenge again for contacting the same person or logging in until the user logs out or closes the browser. The necessary functions for this are in AuthenticationService.js and web3.js for the client while the functions for the server are in AuthenticationController.js.

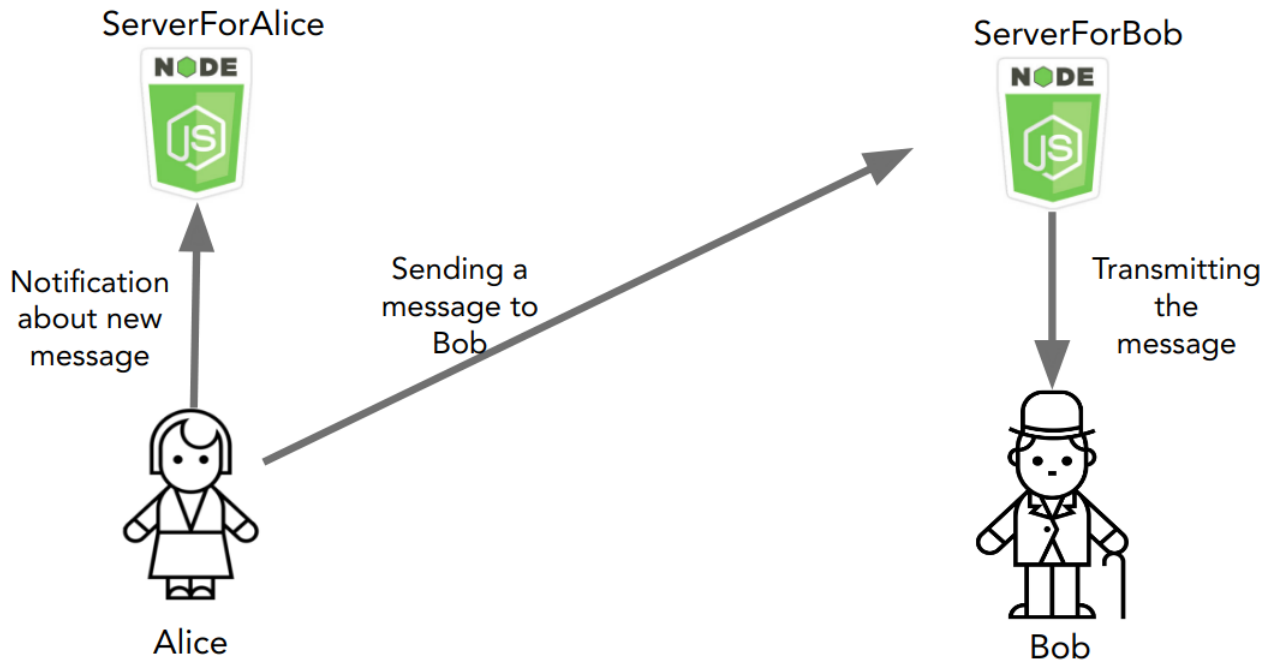
4.2.5 Sending Friend request

In following figure, one can see an example of how BlockChat handles friend requests. First Alice sends a simple friend request via REST API to Bob's storage, but before the request can be sent, Alice has to authenticate herself to Bob's storage. The request will be forwarded to Bob's client where Bob can either accept or decline the request. If Bob accepts the request he has to authenticate himself to Alice's storage. This is followed by two responses, one for Alice's storage, which is sent through a HTTP request to the API, and one Bob's own storage, which is sent through the Socket.IO channel. Both storages update the contact list of their users. Client-side functions for this process are found in Search.Vue, UserService.js and mutations.js while server-side functions are found in UserController.js and ContactController.js.



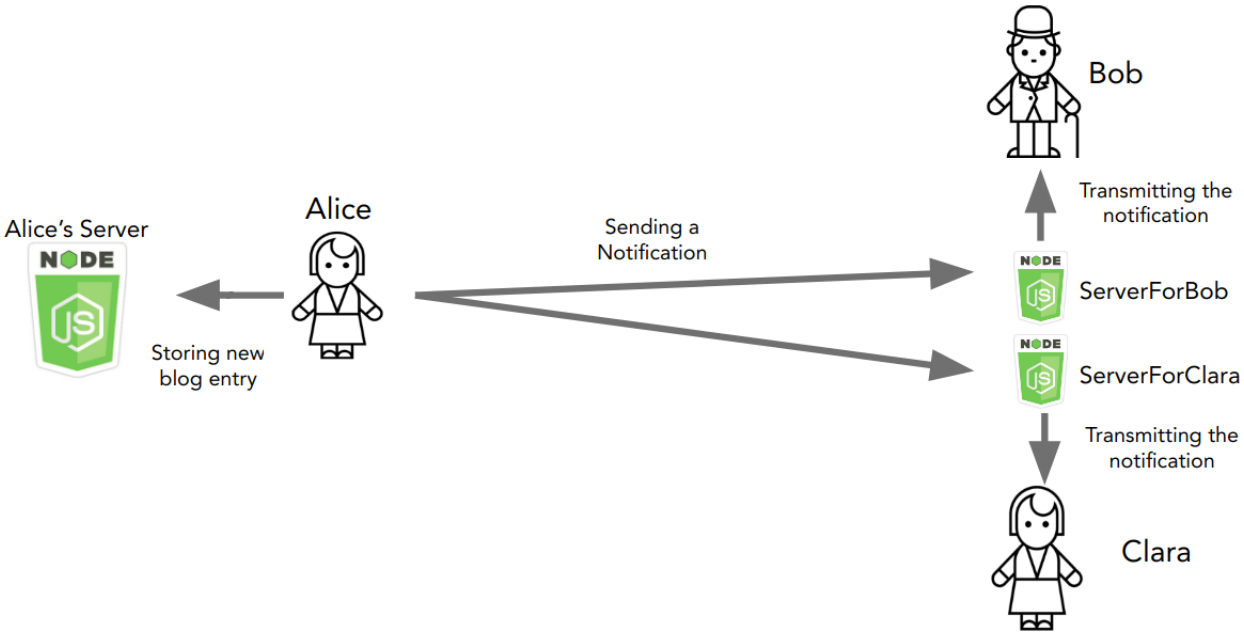
4.2.6 Messaging

Messaging is handled for every user the same. As we can see in the figure below it is mainly a Client-to-Server communication. As soon a conversation partner sends a message, the conversation on both ends will be updated. This happens with a HTTP request to the API of Bob and a notification with the message attached to it with Socket.IO. To open or start a conversation both users have to authenticate, because they again start to send messages to each others storages. The client functions for messaging are in all Vue files in the chat directory and in UserService.js while all server functions are inside MessageController.js.



4.2.7 Microblogging

Microblogging is a bit more complicated. Let us take the example of figure 05. If Alice posts a new entry on her blog, she stores the new entry only in her own storage but also sends a notification to all contacts she has in her contact list. The storing from client to storage happens with a Socket.IO channel while the notification of all contacts is handled with HTTP request to the corresponding API endpoint. The storages of all contacts forward this notification to their client. Now anytime Clara accesses the blogging page while logged in, she will send a HTTP request to Alice's server for her latest entries. After authenticating, Alice's server will respond to Clara's request and the last entries of Alice will be forwarded from Clara's server to the her client. Those entries will be stored nowhere but in the local storage of Clara's browser. This means they will disappear as soon as Clara stops using her browser. The Javascript functions are inside the MicroBlog directory and inside UserService.js for client-side implementation. The server-side implementation can be seen in BlogController.js.



4.3 Details of authentication

Authentication is needed in the application so that no entity may use such functionalities to which they are not entitled to. For this reason, we have made use of MetaMask, with which authentication can be done through signing messages. As it will be further described in the System Architecture part, our system only contains Client to Server communication as

4.3.1 Files transmitted during the authentication procedure

Special files transmitted during the authentication procedure:

- MetaAddress: Used to issue a challenge message for an address
- MetaMessage: Challenge message returned by the client.
- MetaSignature: Signed Challenge returned with MetaMessage.[12]

4.3.2 Steps of authentication

1. The User sends a request to the server (User's MetaAddress in the URL)
2. The Server generates a challenge, sends it back to the User
3. The User signs the challenge and sends the MetaMessage

4. The Server extracts the User's MetaAddress based on the MetaMessage and the MetaSignature and compares it with one sent originally

5. If the two MetaAddresses match, the User is authenticated and is given a token which is used for the given session.[12]

4.3.3 Authentication use cases

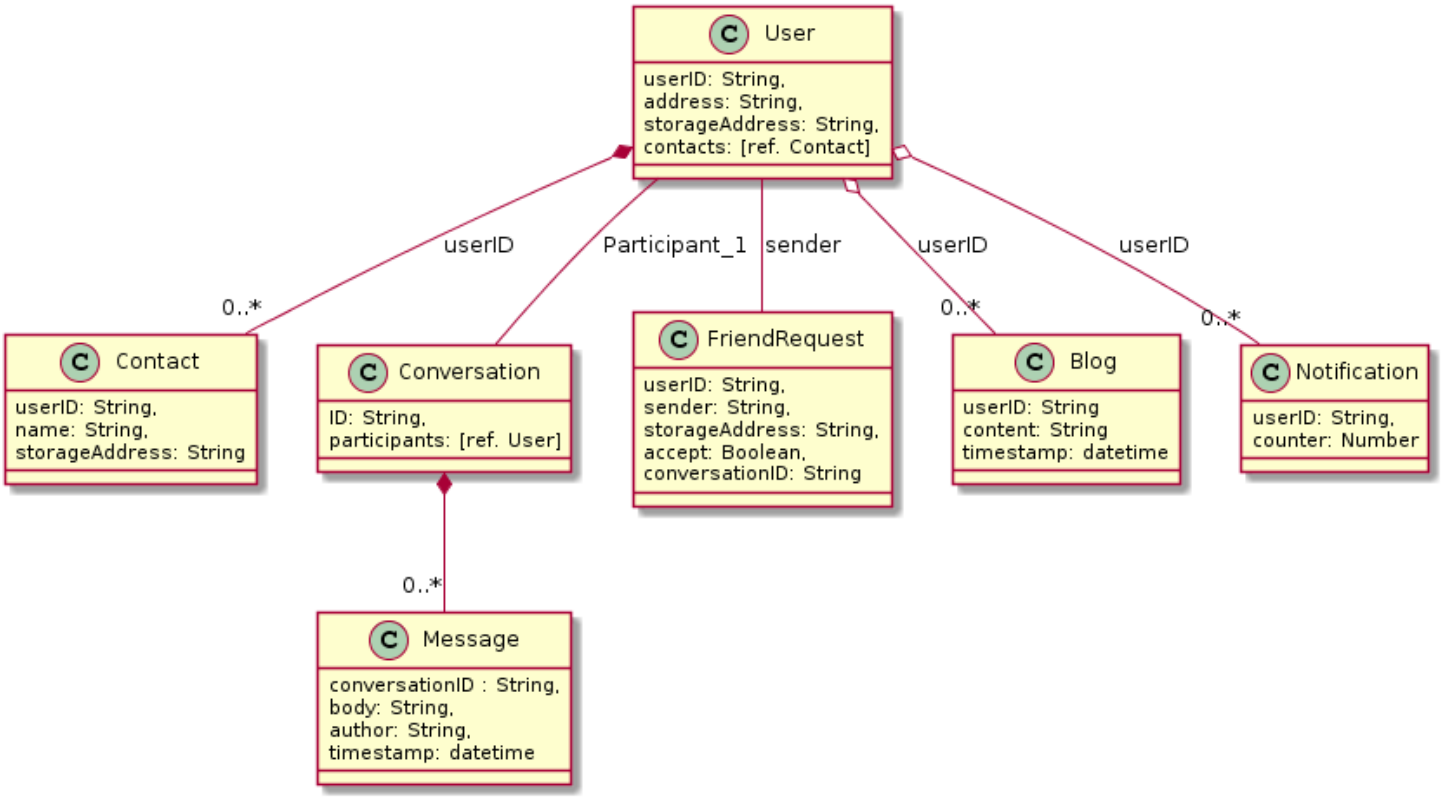
Authentication takes place in the following cases:

- User logs in
- User sends a friend request
- First message sent to a User in the current session (since after this a token is stored in the session, this happens only once for each message sent to a User of the same Endpoint)

4.4 Schema structure

The following diagram shows the connection between the various schemas used on the Server side of the application.

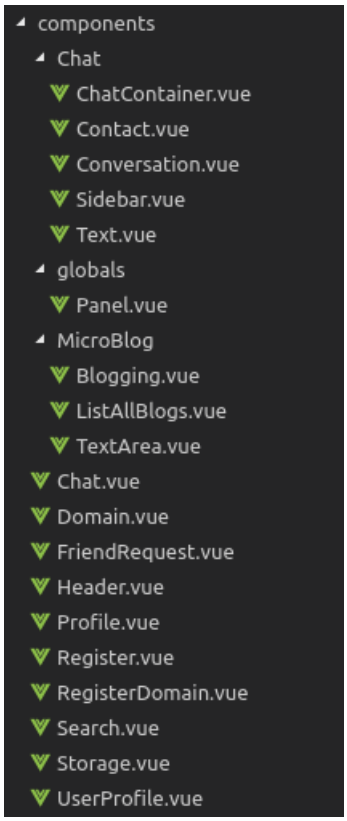
For each schema, where there is no designated ID used, the ID generated by MongoDB is used in the implementation. The listed UserIDs of the schemas are Ethereum names. The ID for each Conversation is generated by the server as a part of the friendrequest procedure (for further details please have a look at the designated part in the REST API section).



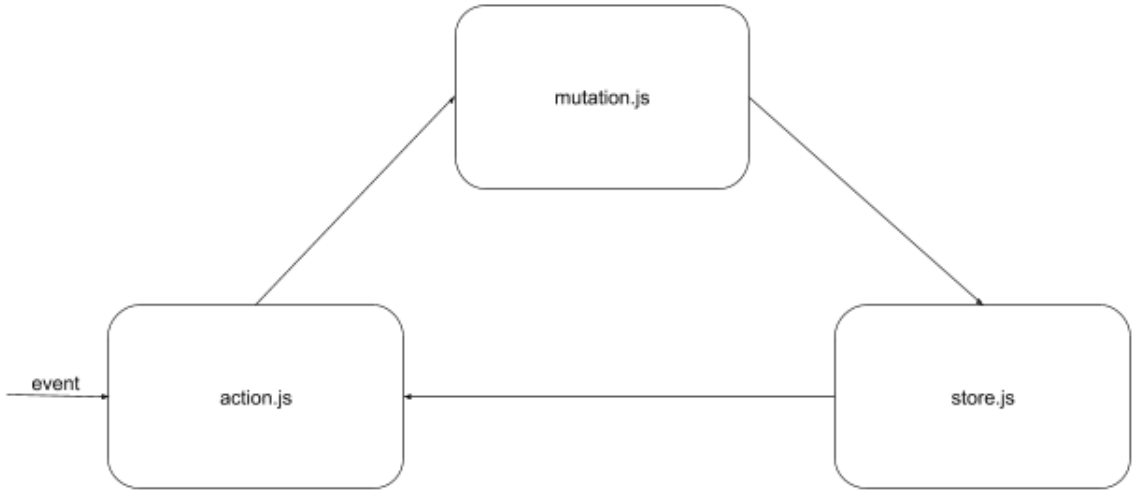
4.5 Frontend

4.5.1 Logic

For the Frontend of BlockChat we used the Components-Pattern of Vue.js . With that, we can implement every page modular and only load parts of a page that need changing. For instance the header, navbar and footer are present in every page. Therefore they don't need to be loaded for every page. Only content in the ¡main¿-tag changes with every page. Vue.js also allows implementing methods which are specific for the component. That is why we implemented most ajax functions inside the components. Function for sending messages is found in Text.vue while the function for loading all incoming messages is in Conversation.vue. The functions for blogging, login and sign up are placed similar to messaging. We split the messaging and blog component into multiple components because of the complexity of both compared to login and sign up. That is why the components directory is structured like this:



Additionally we used an "action-mutation-store" design-pattern with which we manage the data of the client. As pictured in the following image is the relation between all three a cycle.



What happens is that an event triggered by the client starts and action which then may start a mutation. The mutation is called when data, which are stored in store, are being changed or modified. The data each store session has are the following:

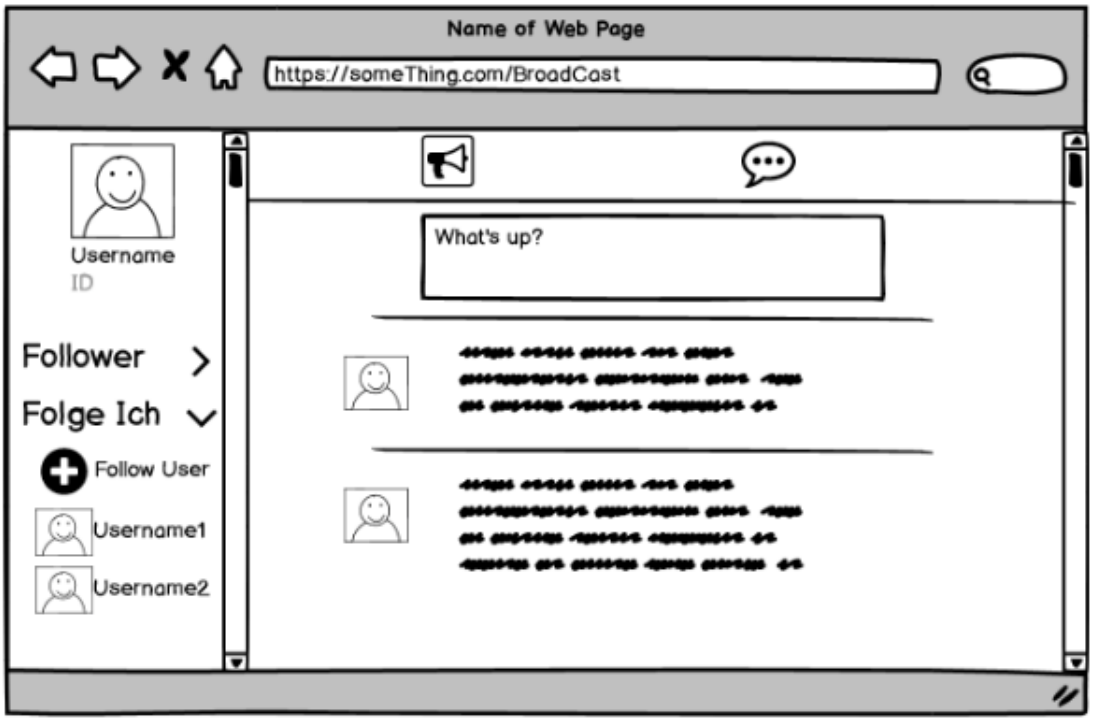
```
1 const initialState = {
2  token: null,
3  user: null,
4  profile: null,
```

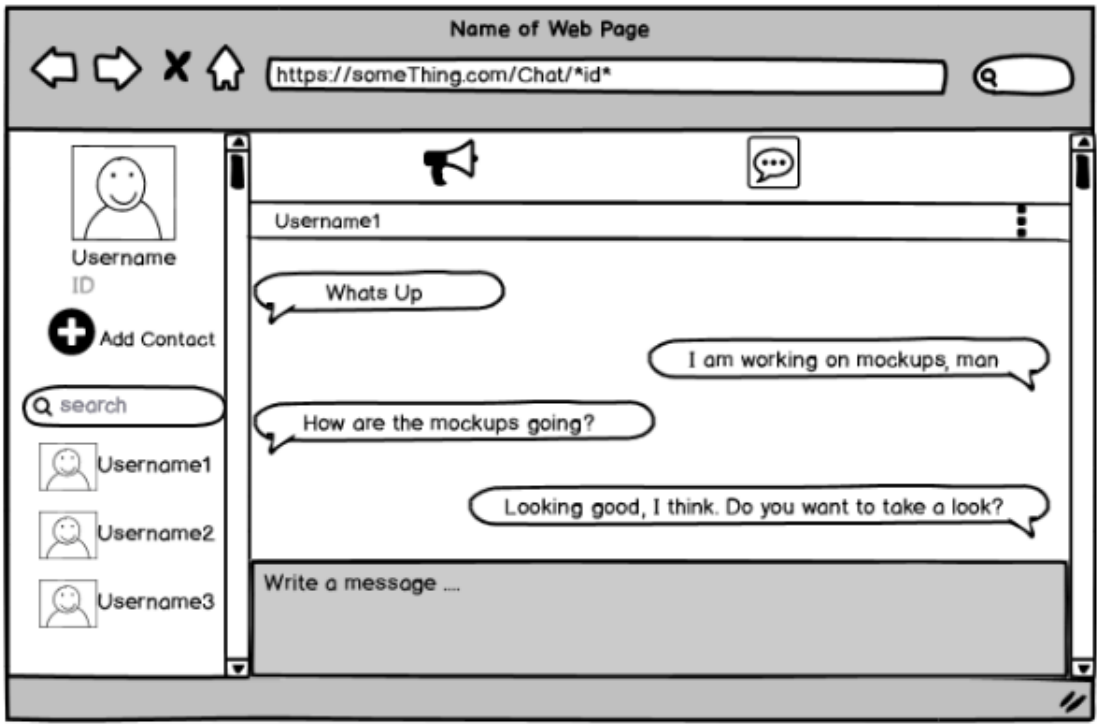
```
5   isUserLoggedIn: false,
6   url: null,
7   conversation: null,
8   friendRequests: [],
9   messages: [] ,
10  blog: [],
11  contacts : [],
12  endpoints:[],
13  currentEndpoint: null,
14  friend: false,
15  blogNotification: 0
16 }
```

The *token* is a string of random letters and numbers and it is given by the server of my conversation partner for authentication. *User* is a json with an array of all contacts, an *id* and a *user_id* which is the etherium id of the user. *Profile* is similar to user but contains additional information like a *storageAddress* and an *address*. *conversatio*n contains everything regarding the current conversation like all participants and the *conversation_id*. *currentEndpoint* is the URI to the contact information of the user while *messages* is an array of all messages the user sent in this conversation. *blogNotification* counts the number of new blog entrys of every member in users contact list. All those blog entries are load into the list *blog*. *isUserLoggedIn* is a boolean and is true if the user is logged into his storage.

4.5.2 Layout

For the layout our goal was it to keep as simple as possible without losing any usability. Therefore we prepared some mock ups. As we can see in the following pictures a sidebar follows the user to all communication pages like blogging and messaging. In both cases it serves a similar but different function. In both cases it should display the current user id with profile picture through which the user can access his or her own profile page. Also in both cases all contacts are displayed in the sidebar. While in messaging the user can switch private chats by choosing the conversation partner through the sidebar, in blogging choosing a contact only filters the blog feed for entries of chosen user.





4.6 RESTful API

For the backend side of the application a Resftul API was used. In this section the main functionalities for it are introduced through diagrams and explanations.

4.6.1 Notations

The following notations are used in the coming diagrams:

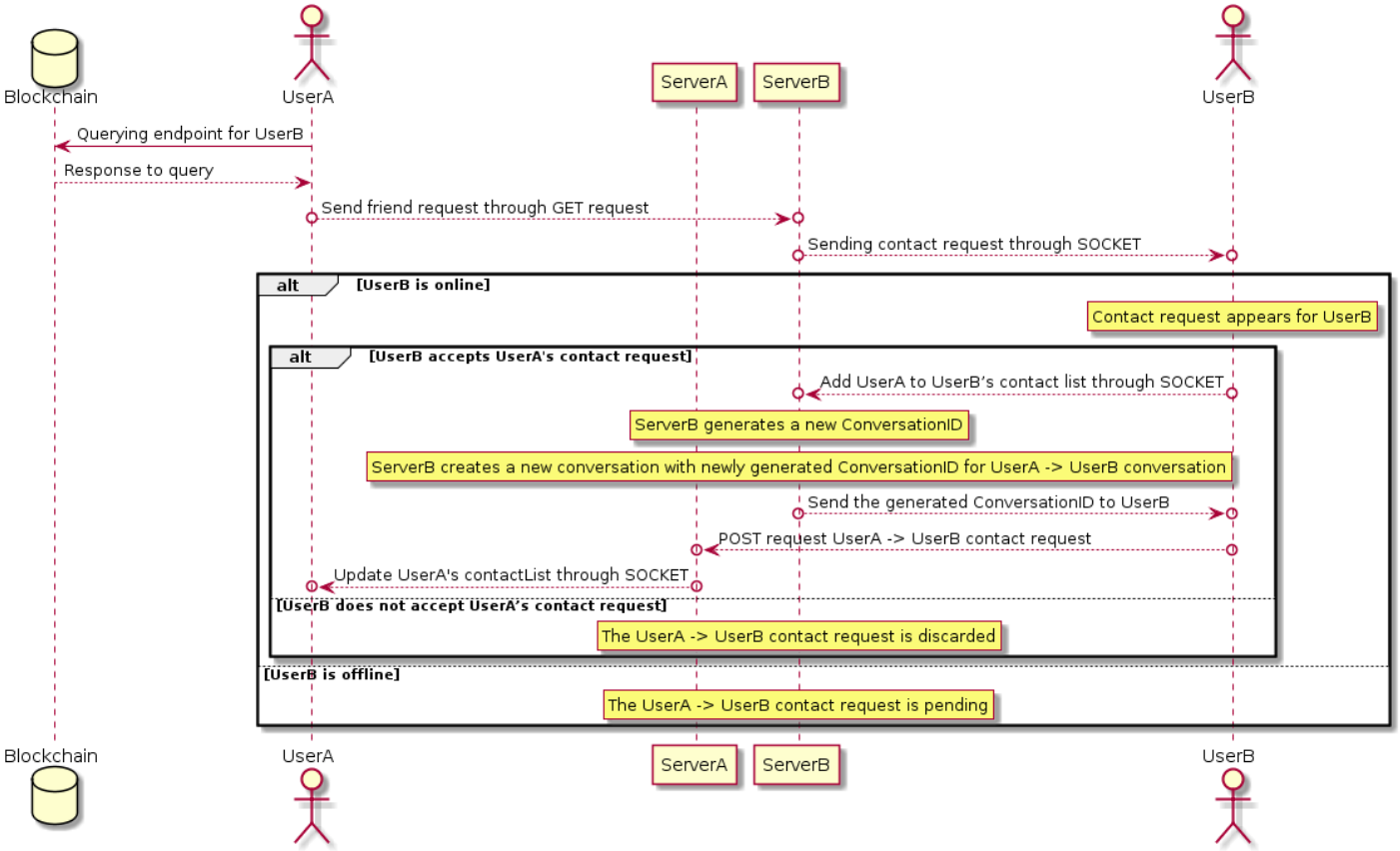
Arrows

- continuous arrow: asynchronous message transmission
- dotted arrow: synchronous message transmission
- arrow with circles: denotes authenticated communication using MetaMask

Abbreviations

- SOCKET: denotes a socket connection between the Client and Server side for a User. This is achieved through the framework Socket.io, a Client fetches all pieces of information whenever the connection is established.
- GET, POST, PUT: HTTP requests for message transmission

4.6.2 Friendrequest



1. UserA would like to add UserB as a contact.

2. UserA fetches the URL for ServerB from the Blockchain.

3. UserA sends a contact request through a GET request to ServerB, while also authenticating oneself using the route /friendRequest/auth/IDofUserA/IDofUserB/:MetaMessage/:MetaSignature

4. ServerB propagates the contact request to UserB using SOCKET

5. If UserB is online, then the contact request appears in the user interface

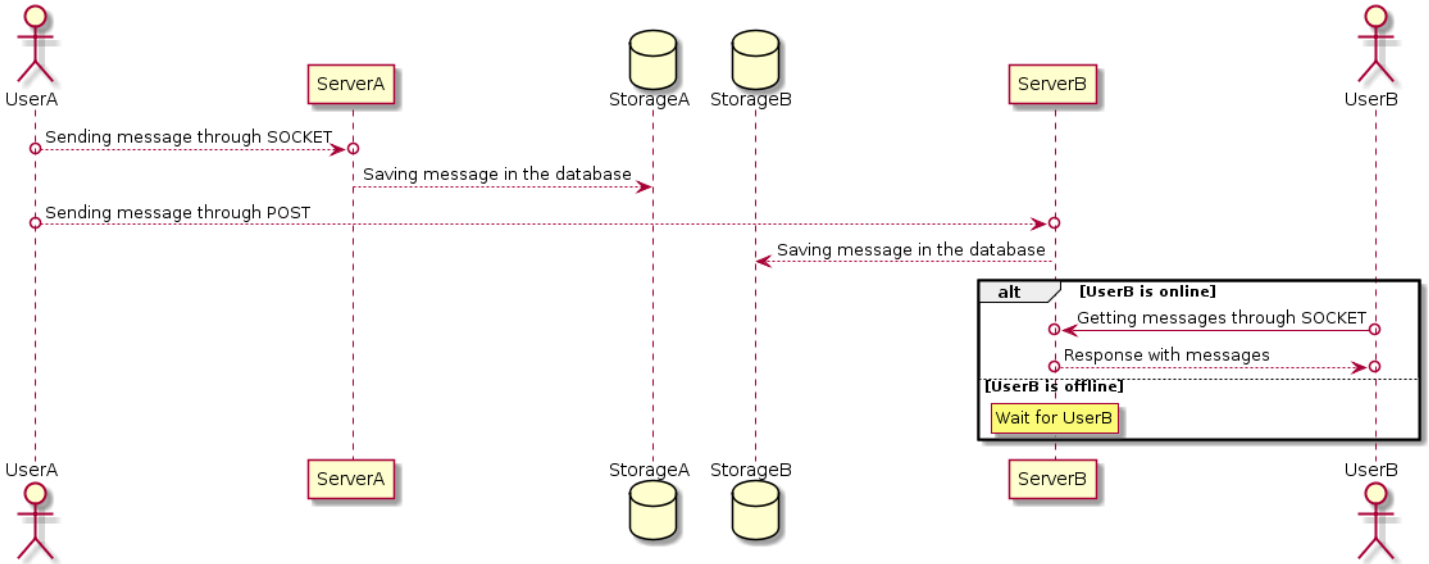
   If UserB accepts the contact request, then

   - It adds UserA to the contact list and sends this information to ServerB for storing
   - When ServerB receives, it generates a ConversationID and then a conversation object for the conversation between UserA and UserB
   - Then ServerB sends this ConversationID to UserB. This step serves server authentication purposes as it solves the problem that may arise when two
   - After this, UserB sends the information on the accepted request and the ConversationID to ServerA through a POST request using the route /receiveFriendRequest/auth/:currentUser/:newContact/:MetaMessage/:MetaSignature
   - After receiving this request, ServerA updates the contact list of UserA through sending a message using SOCKET by adding UserB to the list

   If UserB does not accept the contact request, then the request is discarded

6. If UserB is offline, then the contact request is pending until UserB comes online

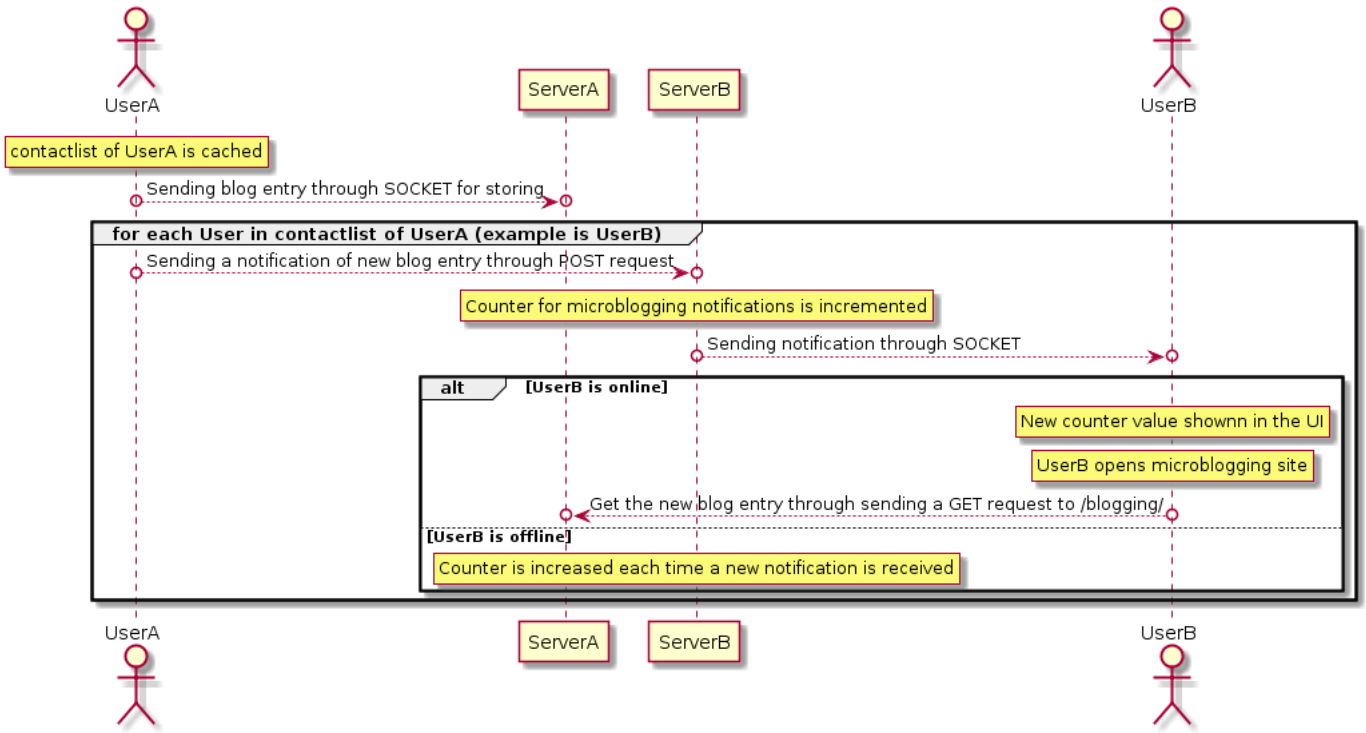
4.6.3 Sending messages



1. ClientA creates a new message.
2. ClientA sends the new message to ServerA for it to be stored in the database.
3. ServerA saves the message in the database
4. ClientA sends the message through a POST request to ServerB using the /messages/ route
5. ServerB saves the message in the database
6. If UserB is online
   - UserB gets the message from ServerB through SOCKET
   - ServerB sends the message through SOCKET

   If UserB is offline
   - ServerB waits for UserB

4.6.4 Microblogging



1. ClientA creates a new blog entry.
2. ClientA sends the blog entry to ServerA for it to be stored in the database (details of message storing is not described in this diagram).

3. ClientA sends a notification (POST request) to all Servers of users in the contact list (addresses for the server are cached), as an example also to ServerB

4. The counter for new microblogs is incremented in ServerB and ServerB sends the notification notice to ClientB, that there is a new blog entry to be fetched through the socket connection

5. Whenever ClientB comes online and clicks on the assigned button in the user interface, the new blog entries are fetched and shown on the microblogging site for ClientB. This is done through GET request to ServerA, its URL is obtained from the cached contactlist object.

Chapter 5

Future outlook

5.1 Debugging

One of the main future objectives would be to work on the bugs that arose during development up to this point. The following list indicates the bugs that we encountered:

- Conversation history is duplicated when scrolling up.

5.2 Further features

Another future objective would be the integration of further features and functionalities. The following list serves as an overview for these possible tasks:

- Integrate auto refresher for MetaMask: currently for the socket connection to be established correctly a page reload is needed due to contemporary malfunction in the MetaMask framework.
- Load history blog entries on scroll down: currently the blog page is created by fetching the last 10 blog entries of each user in the contact list upon opening the microblogging page. These data are loaded into the session and lost upon logout. Consequently, when the microblogging page is revisited, once again only the last 10 blog entries are fetched.

Chapter 6

Bibliography

[1] Ether 1 https://docs.ens.domains/en/latest/introduction.html

[2] Ether 2 https://www.myetherwallet.com/#ens

[3] Crypto https://www.cryptomathic.com/products/authentication-signing/digital-signatures-faqs/what-is-non-repudiation

[4] Crypto https://resources.infosecinstitute.com/non-repudiation-digital-signature/#gref

[5] Crypto https://www.lifewire.com/cryptographic-hash-function-2625832

[6] Consensus 1 https://medium.com/coinmonks/blockchain-consensus-algorithms-an-early-days-overview-2973f0cf49c6

[7] Consensus 2 https://hackernoon.com/a-hitchhikers-guide-to-consensus-algorithms-

[8] Smart contracts https://blockchainhub.net/smart-contracts/

[9] Ethereum https://github.com/ethereum/wiki/wiki/White-Paper#ethereum

[10] EVM http://ethdocs.org/en/latest/contracts-and-transactions/developer-tools.html#the-evm

[11] Federated architecture https://www.progress.com/blogs/distributed-vs-federated

[12] Expressjs User Authentication with MetaMask & meta-authhttps://medium.com/coinmonks/expressjs-user-authentication-with-metamask-meta-auth-630b6da123ef

[13] Decentralised messaging image https://elearningindustry.com/bitcoin-blockchain-impacting-elearning-industry