Censorship-free Messaging using Blockchain Technology

Martin Westerkamp, Sebastian Göndör
Supervisors

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

Berlin, 2018

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 on a peer-to-peer basis. From the user's perspective, 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.

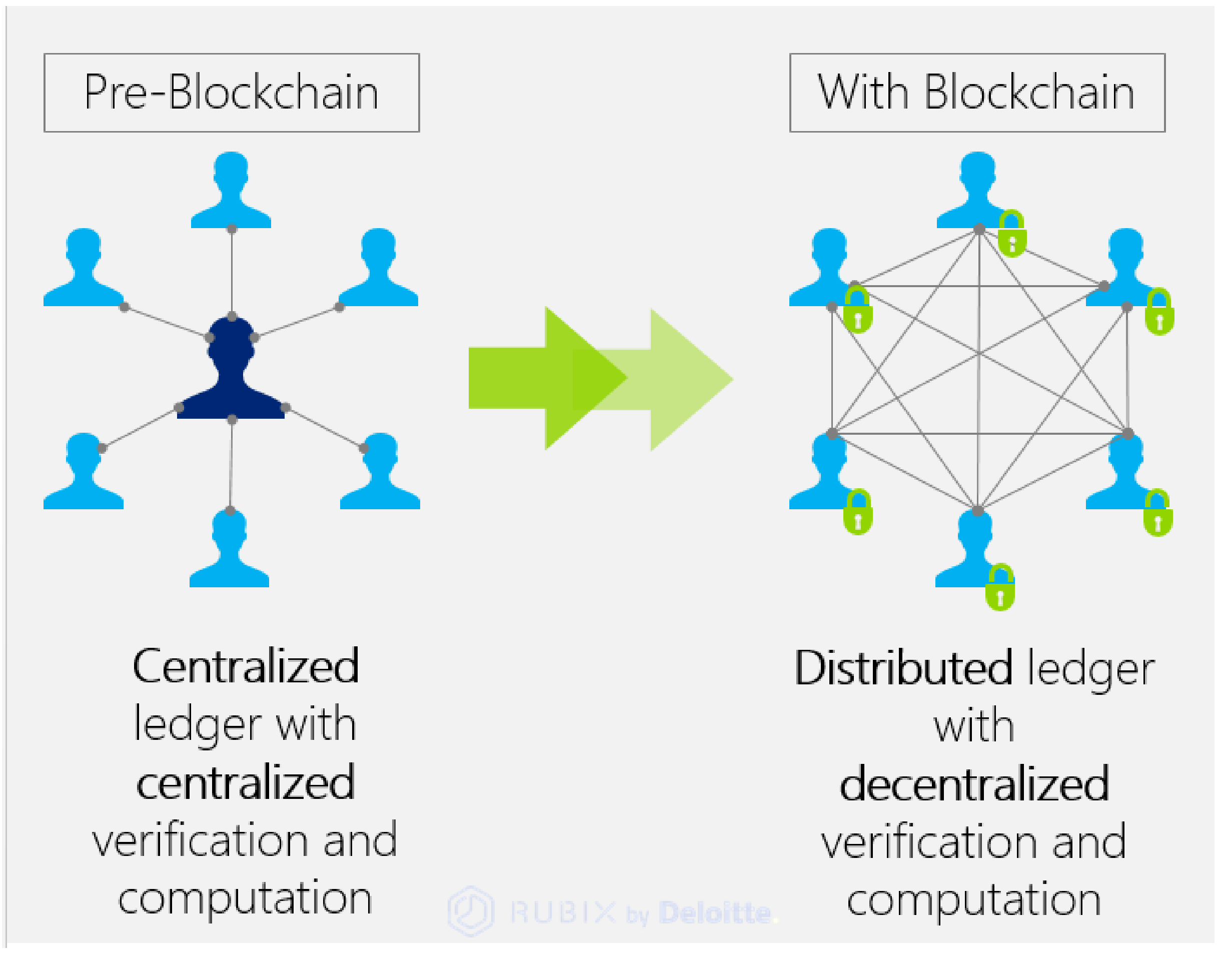

Pre-Blockchain
With Blockchain
Centralized ledger with centralized verification and computation
Distributed ledger with decentralized verification and computation
RUBIX by Deloitte.

Figure 1.1: Comparison of the centralized and decentralized approaches [2]

1.2 Research background

1.2.1 Blockchain and the 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 maintains a list of all domains and subdomains. 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.

More Information about the ENS can be found here: `https://docs.ens.domains/en/latest/index.html`

Prerequisites 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 that 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`

1.3 Technologies in use

1.3.1 Truffle

1.3.2 Ganache

1.3.3 Metamask

1.3.4 Solidity

1.3.5 NodeJS

1.3.6 MongoDB

1.3.7 ExpressJS

1.3.8 VueJS

1.3.9 Webpack

1.4 Design

1.5 Rest API

1.6 Security and authentication

Chapter 2

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.

2.1 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

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

- Ganache-cli: execute the following command

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

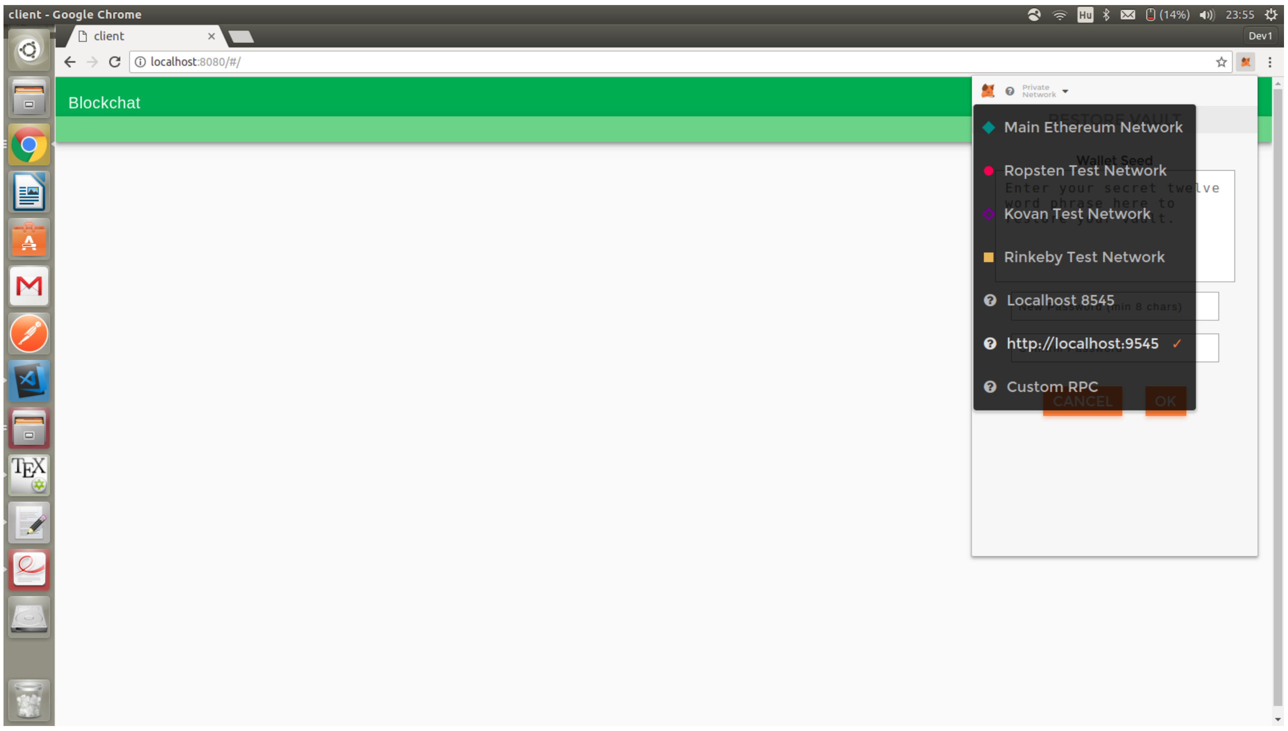
- MetaMask: follow installation details from the official site for the Chrome extension (`https://metamask.io/`).

2.2 Using the application

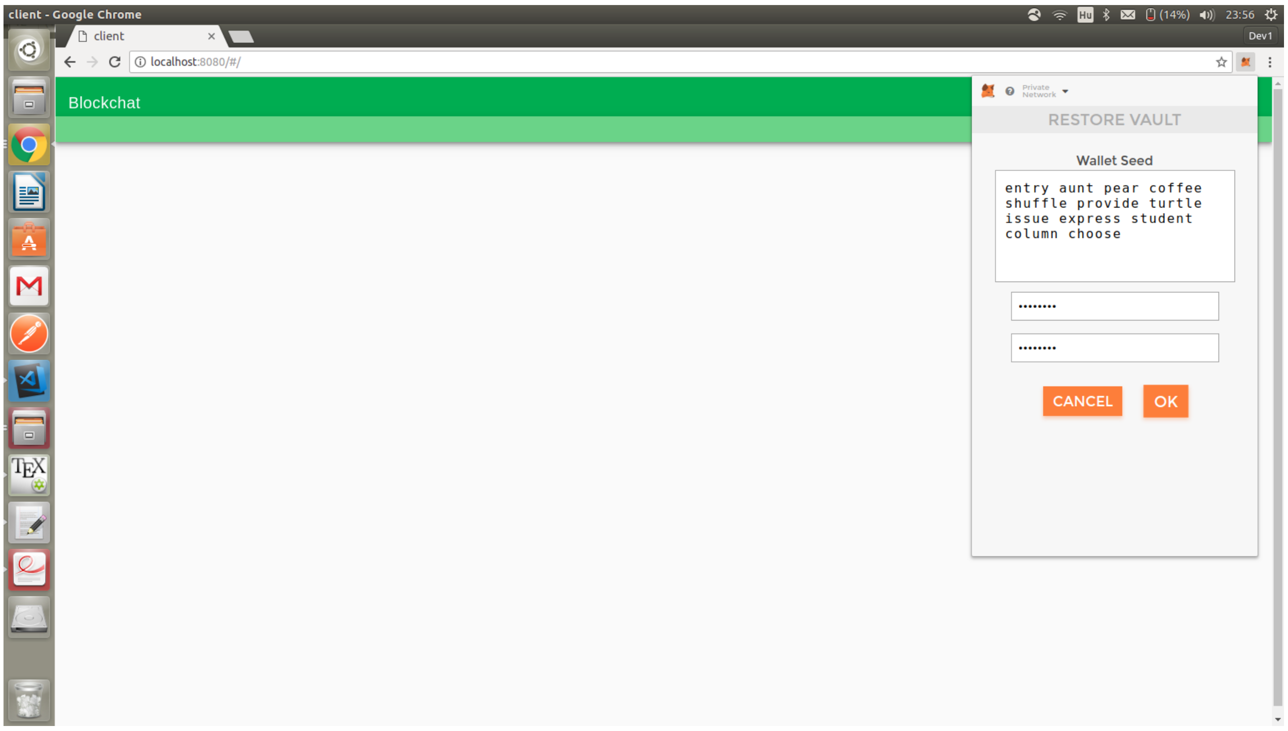
We are assuming that there is an Ethereum blockchain service running and that there are servers of BlockChat to which users can connect to.

2.2.1 Signing up in MetaMask

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

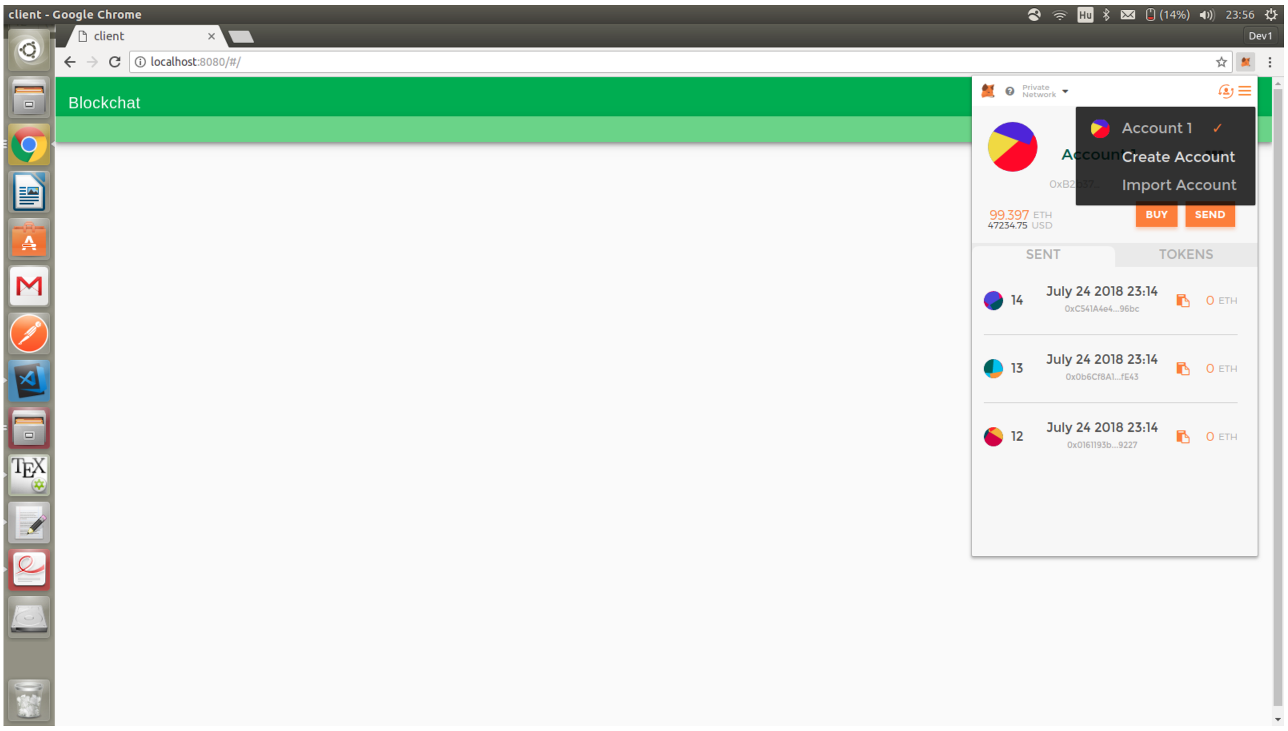


2. Signing up to running Ethereum network through MetaMask (a local Ganache blockchain listeninig on port 9545 is used for development)



```
entry aunt pear coffee
shuffle provide turtle
issue express student
column choose
```

3. Creating a new account for the network.



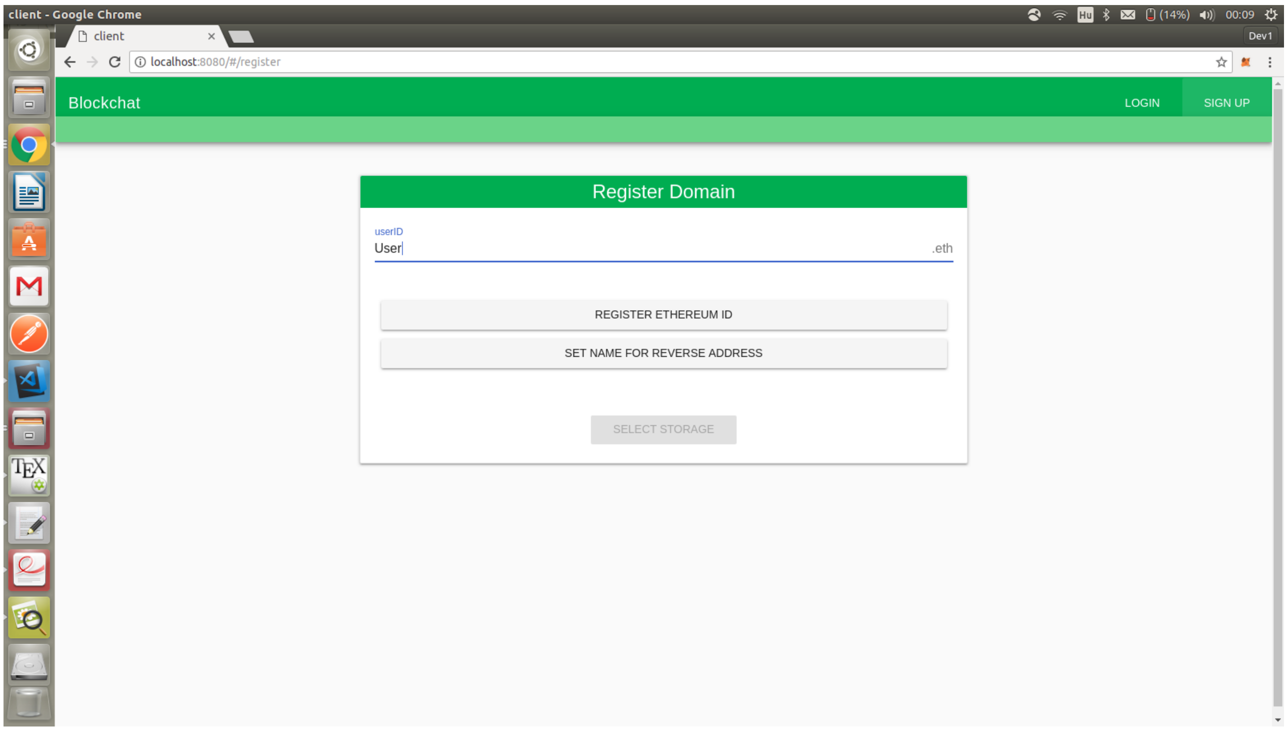
99.397 ETH
47234.75 USD

```
14  July 24 2018 23:14  0xC541A4e4...96bc  0 ETH
13  July 24 2018 23:14  0x0b6Cf8A1...fE43  0 ETH
12  July 24 2018 23:14  0x0161193b...9227  0 ETH
```

2.2.2 Signing up into BlockChat

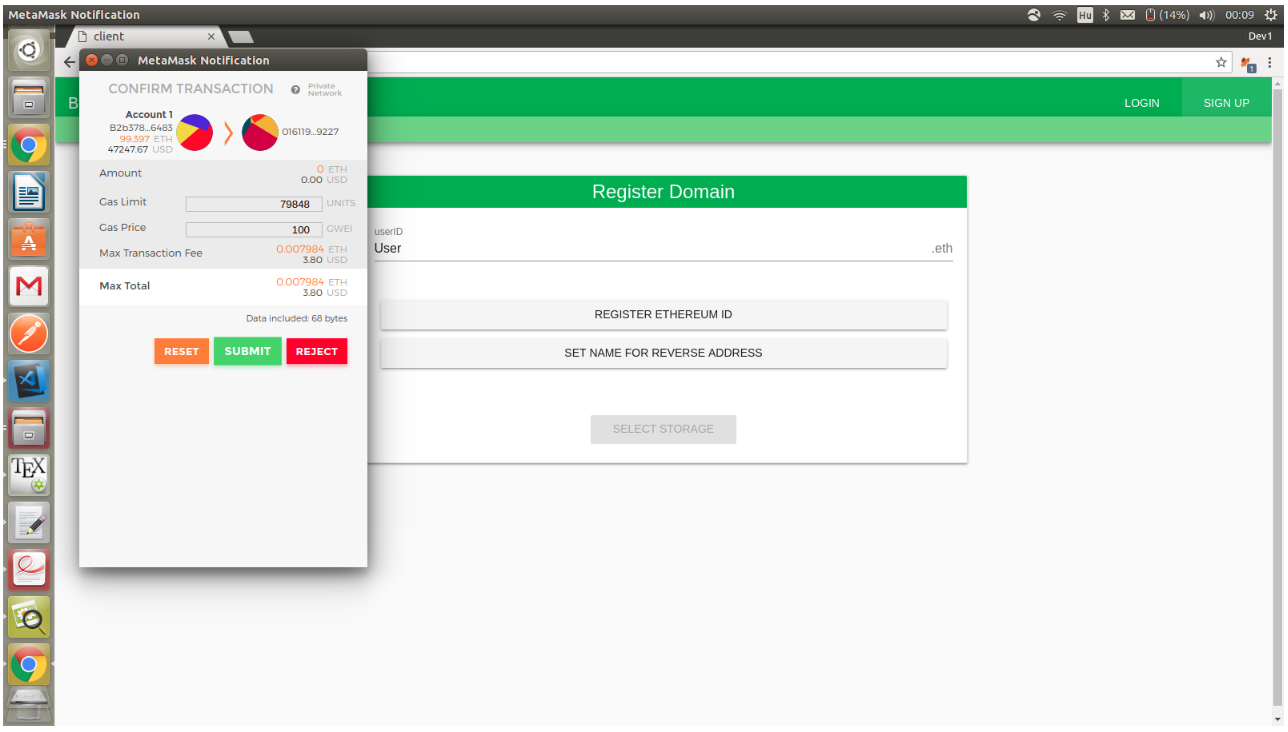
1. Visit the designated URL for BlockChat in the browser (used local instance for development).

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



userID
User .eth

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



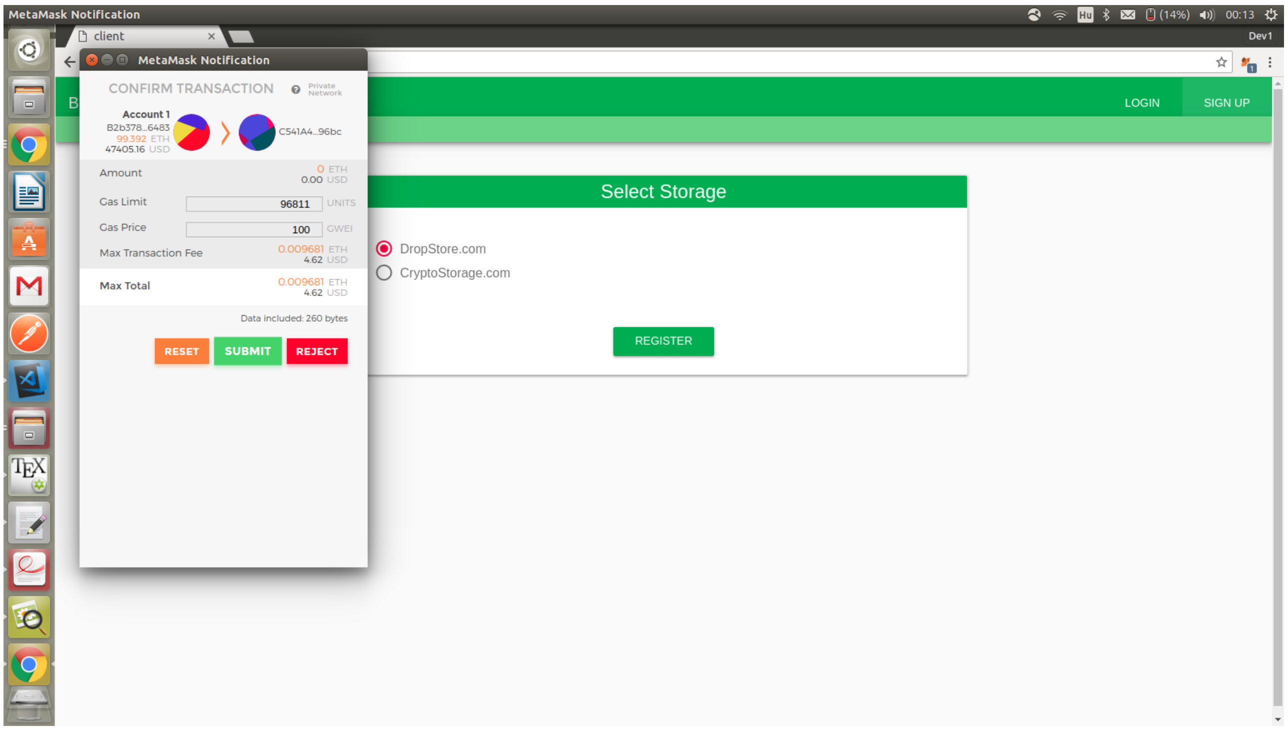
Account 1
B2b378...6483
99.397 ETH
47247.67 USD
016119...9227

Amount	0 ETH 0.00 USD
Gas Limit	79848 UNITS
Gas Price	100 GWEI
Max Transaction Fee	0.007984 ETH 3.80 USD
Max Total	0.007984 ETH 3.80 USD

Data included: 68 bytes

userID
User .eth

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



CONFIRM TRANSACTION Private Network

Account 1
B2b378...6483
99.392 ETH
47405.16 USD
C541A4...96bc

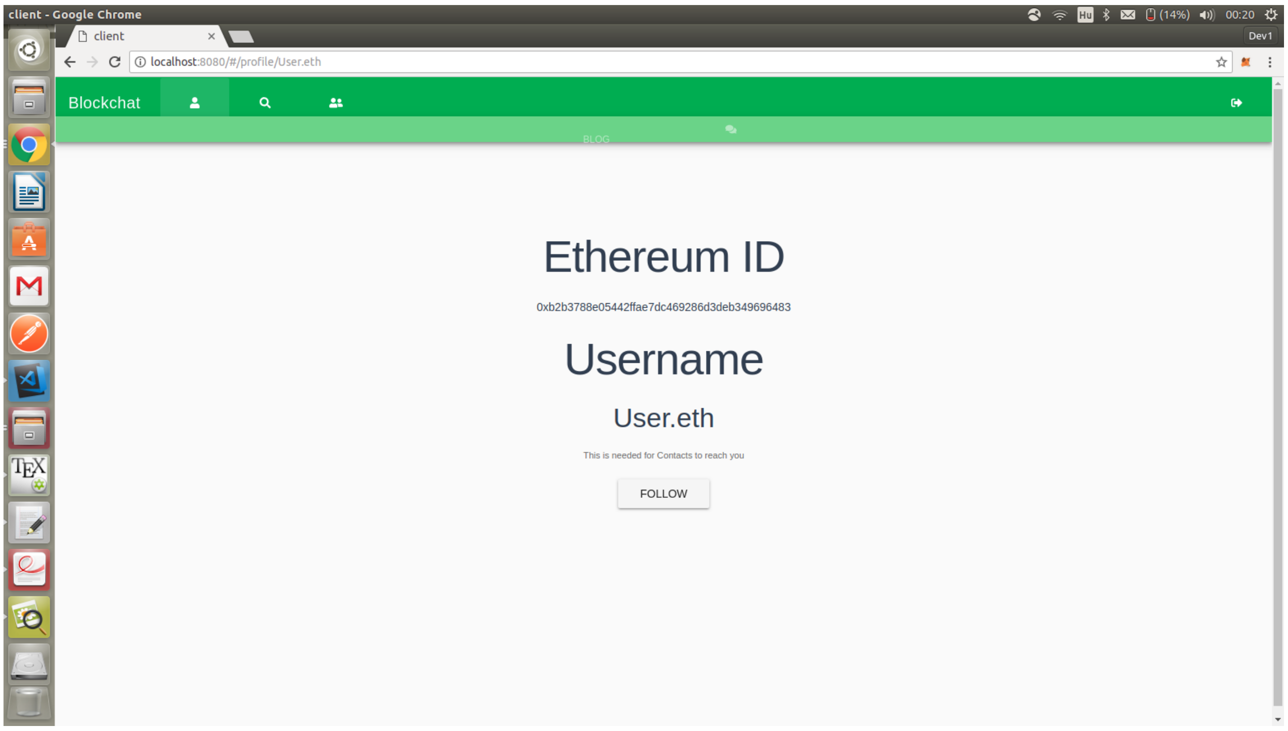
Amount	0 ETH 0.00 USD
Gas Limit	96811 UNITS
Gas Price	100 GWEI
Max Transaction Fee	0.009681 ETH 4.62 USD
Max Total	0.009681 ETH 4.62 USD

Data included: 260 bytes

Select Storage

[✓ DropStore.com]
CryptoStorage.com

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.



localhost:8080/#/profile/User.eth

Ethereum ID

0xb2b3788e05442ffae7dc469286d3deb349696483

Username

User.eth

This is needed for Contacts to reach you

2.2.3 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 3

Developer Documentation

3.1 Blockchain

In our Application, the Ethereum Domain Name, owned by a User will act as an unique Identifier and will be neccessary to have in order to register for our service to use. Smart Contract - URLResolver.sol

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 URL smart contract consists of the following important attributes and methods :

```
struct Record {
        string url;
        PublicKey pubkey;
    }
```

```
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.

This smart contract will be used by our application when registering for our service as well as for searching other users. 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 - in this case the BlockChain. 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. Design Backend The Backend consists of the following main components, which will be further explained in detail:

- controllers
- models
- routes
- Authentication
- server.js
- socket.js
- passport.js

3.2 System Architecture

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

3.3 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

3.3.1 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.[1]

3.3.2 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.[1]

3.3.3 Authentication use cases

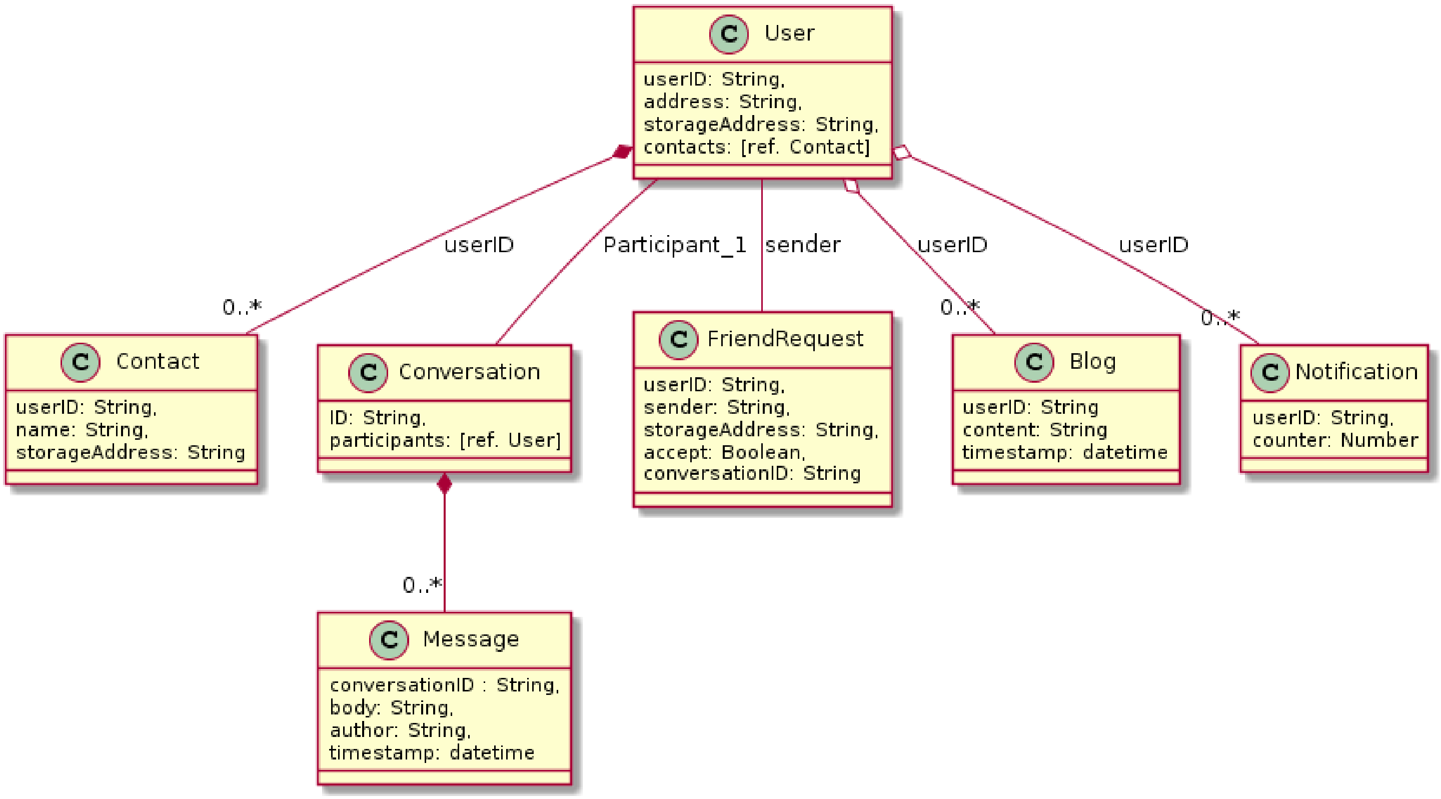
Authentication takes place in the following cases:

- 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)

3.4 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).



3.5 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.

3.5.1 Notations

The following notations are used in the coming diagrams:

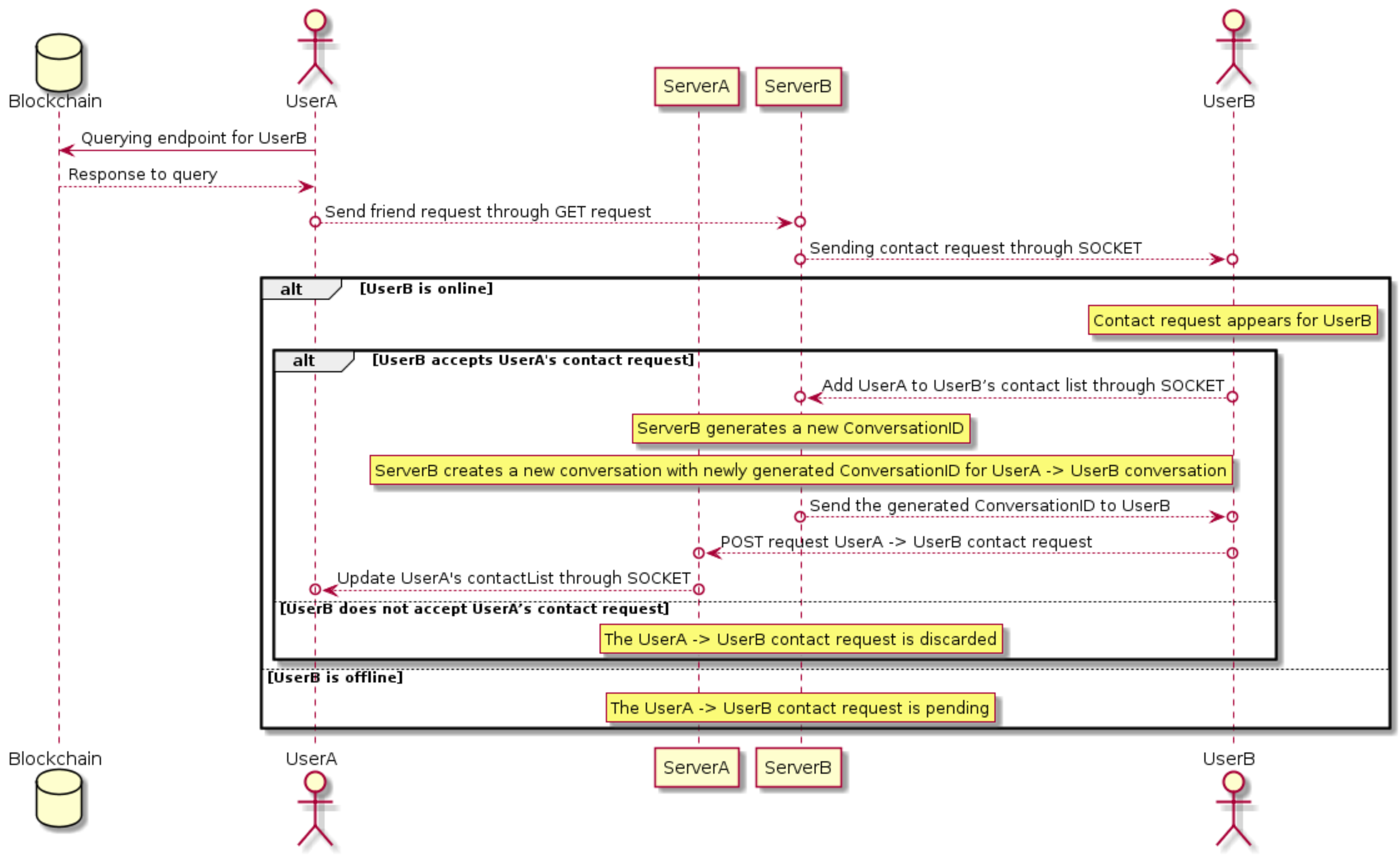
Arrows

- 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

3.5.2 Friendrequest


Blockchain
UserA
ServerA
ServerB
UserB
Querying endpoint for UserB
Response to query
Send friend request through GET request
Sending contact request through SOCKET
alt [UserB is online]
Contact request appears for UserB
alt [UserB accepts UserA's contact request]
Add UserA to UserB's contact list through SOCKET
ServerB generates a new ConversationID
ServerB creates a new conversation with newly generated ConversationID for UserA -> UserB conversation
Send the generated ConversationID to UserB
POST request UserA -> UserB contact request
Update UserA's contactList through SOCKET
[UserB does not accept UserA's contact request]
The UserA -> UserB contact request is discarded
[UserB is offline]
The UserA -> UserB contact request is pending

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

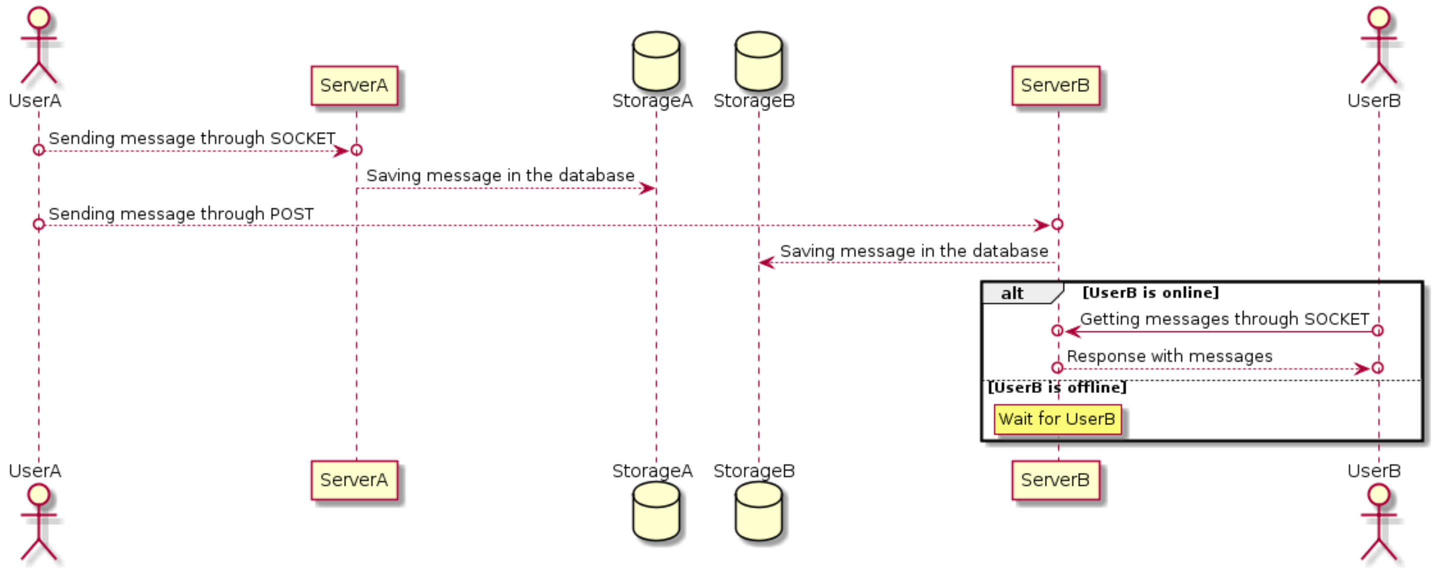
   - It adds UserA to the contact list and sends this information to ServerB for storing
   - When ServerB receives, it generates a ConversationID and then a conversation object for the conversation between UserA and UserB
   - Then ServerB sends this ConversationID to UserB. This step serves server authentication 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

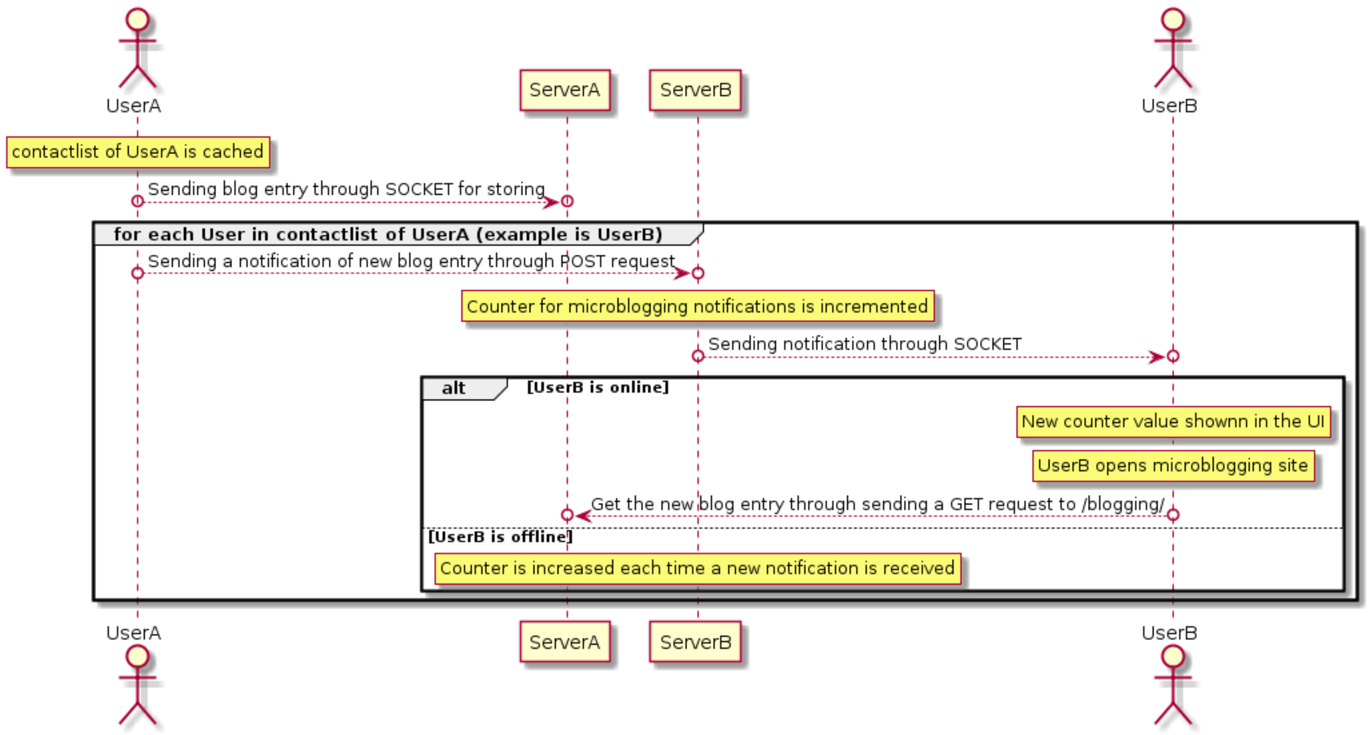
3.5.3 Sending messages


UserA
ServerA
StorageA
StorageB
ServerB
UserB
Sending message through SOCKET
Saving message in the database
Sending message through POST
Saving message in the database
alt
[UserB is online]
Getting messages through SOCKET
Response with messages
[UserB is offline]
Wait for UserB

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

3.5.4 Microblogging


UserA
ServerA
ServerB
UserB
contactlist of UserA is cached
Sending blog entry through SOCKET for storing
for each User in contactlist of UserA (example is UserB)
Sending a notification of new blog entry through POST request
Counter for microblogging notifications is incremented
Sending notification through SOCKET
alt [UserB is online]
New counter value shownn in the UI
UserB opens microblogging site
Get the new blog entry through sending a GET request to /blogging/
[UserB is offline]
Counter is increased each time a new notification is received
UserA
ServerA
ServerB
UserB

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 4

Future outlook

4.1 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.

4.2 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 5

Bibliography

[1] Expressjs User Authentication with MetaMask & meta-auth`https://medium.com/coinmonks/expressjs-user-authentication-with-metamask-meta-auth-630b6da12`

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