Cryptographie appliquée - Build a decentralized messaging app using the Ethereum blockchain

Tristan BILOT, Nora DELFAU
EPITA

24 octobre 2021

`repo: https://github.com/TristanBilot/blockchain-chat-app`

1 Install dependencies

Before diving into blockchain and smart contract development, we have to install all the needed dependencies and environements. Here is a list of the dependencies we need to install before starting coding:

- node.js
- react.js
- web3.js
- metamask
- truffle suite
- ganache

To gain time, we'll clone a template which already have preinstalled the three first dependencies. The template is given by DAppUniversity and can be clones using:

```
git clone https://github.com/dappuniversity/starter_kit
```

Then we can install the dependencies from package.json and run the app using:

```
npm install
npm run start
```

Now we have to install metamask, truffle and ganache:

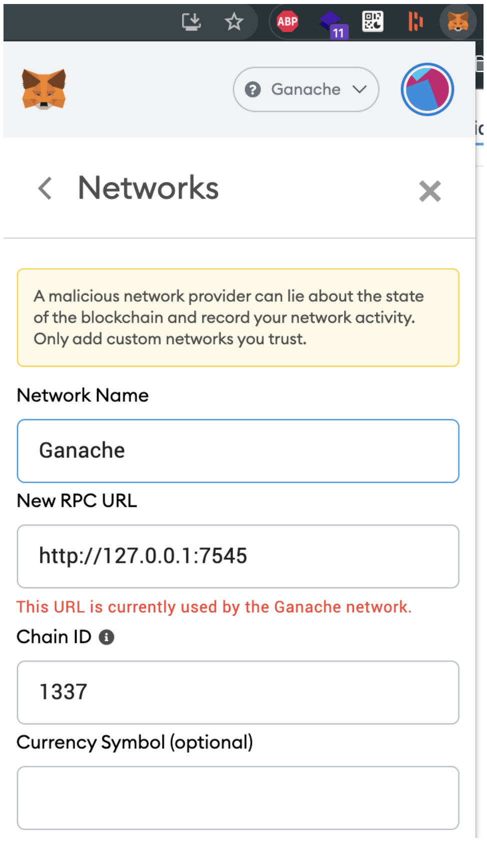
- https://www.trufflesuite.com/
- https://www.trufflesuite.com/ganache
- https://metamask.io/

2 Deploy our first smart contract

2.1 Connect metamask to the browser

First of all, we have to configure metamask. Metamask is a browser plugin used to configure personal ledger wallets used to store tokens that will be used when interacting with our future blockchain. After installing the plugin on the store we create a new network by clicking custom RPC at the top right of the extension.

Then we connect the plugin with the corresponding information used in our project.



A malicious network provider can lie about the state of the blockchain and record your network activity. Only add custom networks you trust.

Network Name

Ganache

New RPC URL

http://127.0.0.1:7545

This URL is currently used by the Ganache network.

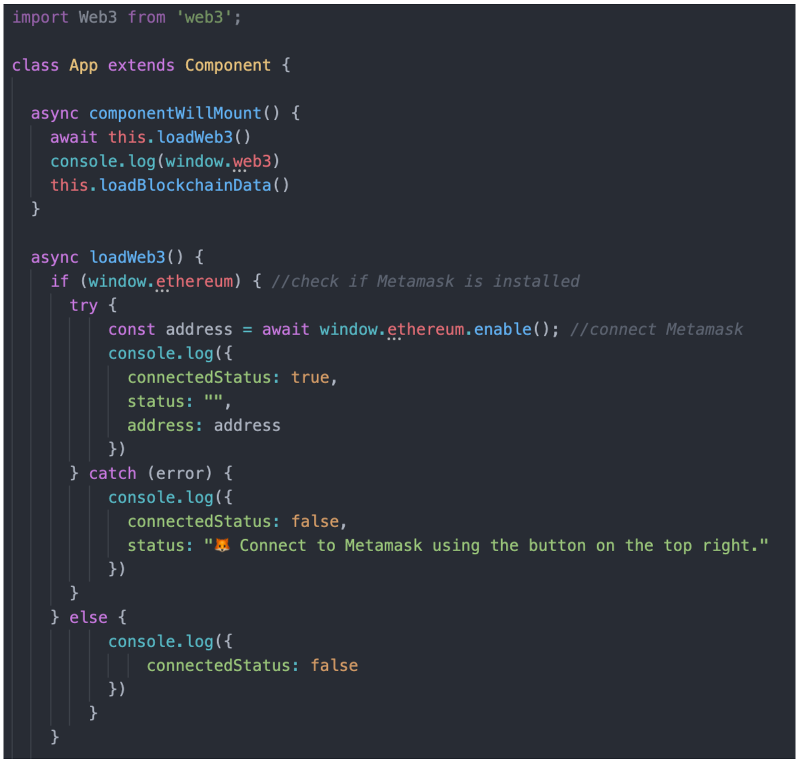
Chain ID

1337

Currency Symbol (optional)

Figure 1: Creating a new metamask network

Then open the file App.js which consist of the ”main” of our web app. We add the following code to connect our browser to the metamask extension when the content of the page is loaded.



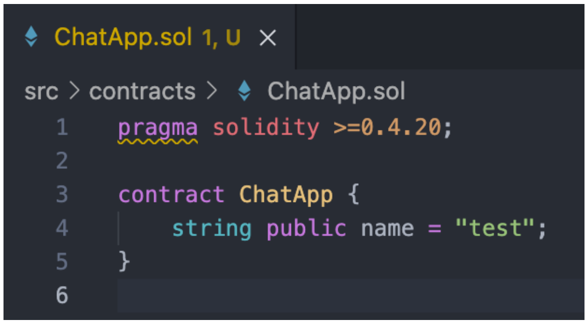
```
import Web3 from 'web3';

class App extends Component {

  async componentWillMount() {
    await this.loadWeb3()
    console.log(window.web3)
    this.loadBlockchainData()
  }

  async loadWeb3() {
    if (window.ethereum) { //check if Metamask is installed
      try {
          const address = await window.ethereum.enable(); //connect Metamask
          console.log({
            connectedStatus: true,
            status: "",
            address: address
          })
      } catch (error) {
          console.log({
            connectedStatus: false,
            status: "🦊 Connect to Metamask using the button on the top right."
          })
      }
    } else {
          console.log({
              connectedStatus: false
          })
        }
    }
```

Figure 2: Method used to connect the browser to metamask

When returning to the web app we should get a pop-in requesting to connect this url to metamask, we simply have to say yes and metamask will be configured.

2.2 Create the smart contract

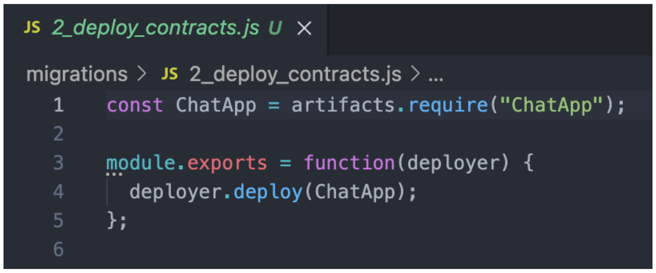
Under src/contracts/, we create a new file Named ChatApp.sol and we're writing this very small and simple contract. This smart contract will be used to verify we're successfuly connected to ganache.



```
pragma solidity >=0.4.20;

contract ChatApp {
    string public name = "test";
}
```

Figure 3: Basic contract used to test the connection between truffle and our app

Then under migrations/ create we create a file named 2_deploy_contracts.js where we tell the blockchain to deploy our smart contract.

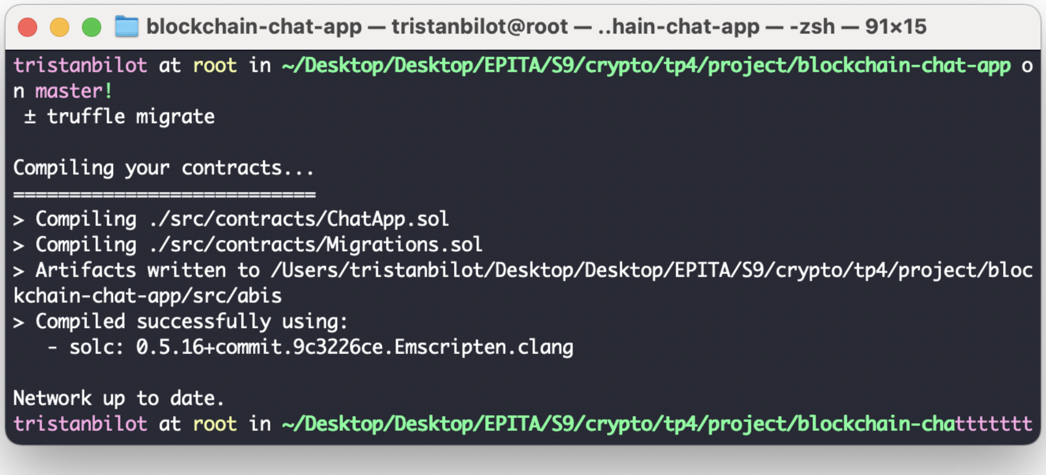

JS 2_deploy_contracts.js U ×
migrations > JS 2_deploy_contracts.js > ...

```
const ChatApp = artifacts.require("ChatApp");

module.exports = function(deployer) {
  deployer.deploy(ChatApp);
};
```

Figure 4: Migration file used by the blockchain to deploy our smart contract

2.3 Deploy the smart contract

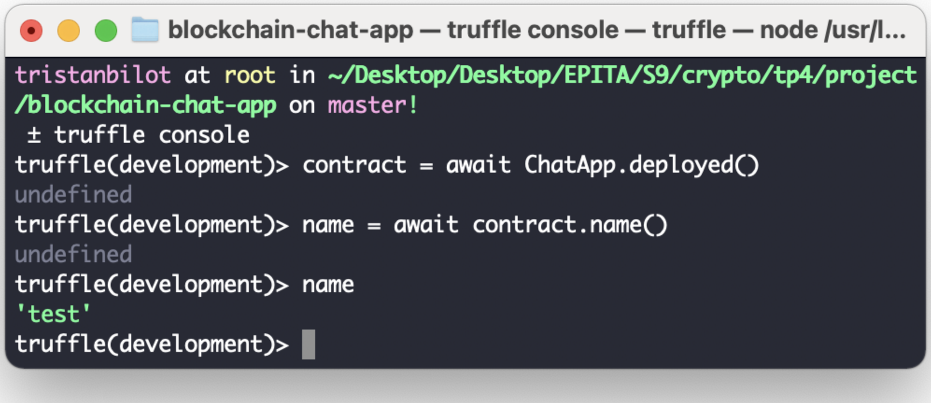
To migrate our contracts, we'll need truffle and ganache. Ganache is a software used to fire a personal Ethereum blockchain on a personal machine. We have to run the ganache app in order to start the server on which we'll can communicate with our blockchain later. Now using the cli of truffle, we can run a migration (deploy all the available smart contracts to the Ethereum blockchain). This migration can be done using this command:


blockchain-chat-app — tristanbilot@root — ..hain-chat-app — -zsh — 91×15

```
tristanbilot at root in ~/Desktop/Desktop/EPITA/S9/crypto/tp4/project/blockchain-chat-app o
n master!
 ± truffle migrate

Compiling your contracts...
===========================
> Compiling ./src/contracts/ChatApp.sol
> Compiling ./src/contracts/Migrations.sol
> Artifacts written to /Users/tristanbilot/Desktop/Desktop/EPITA/S9/crypto/tp4/project/bloc
kchain-chat-app/src/abis
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang

Network up to date.
tristanbilot at root in ~/Desktop/Desktop/EPITA/S9/crypto/tp4/project/blockchain-chattttttt
```

Figure 5: Deploy the smart contracts to the blockchain

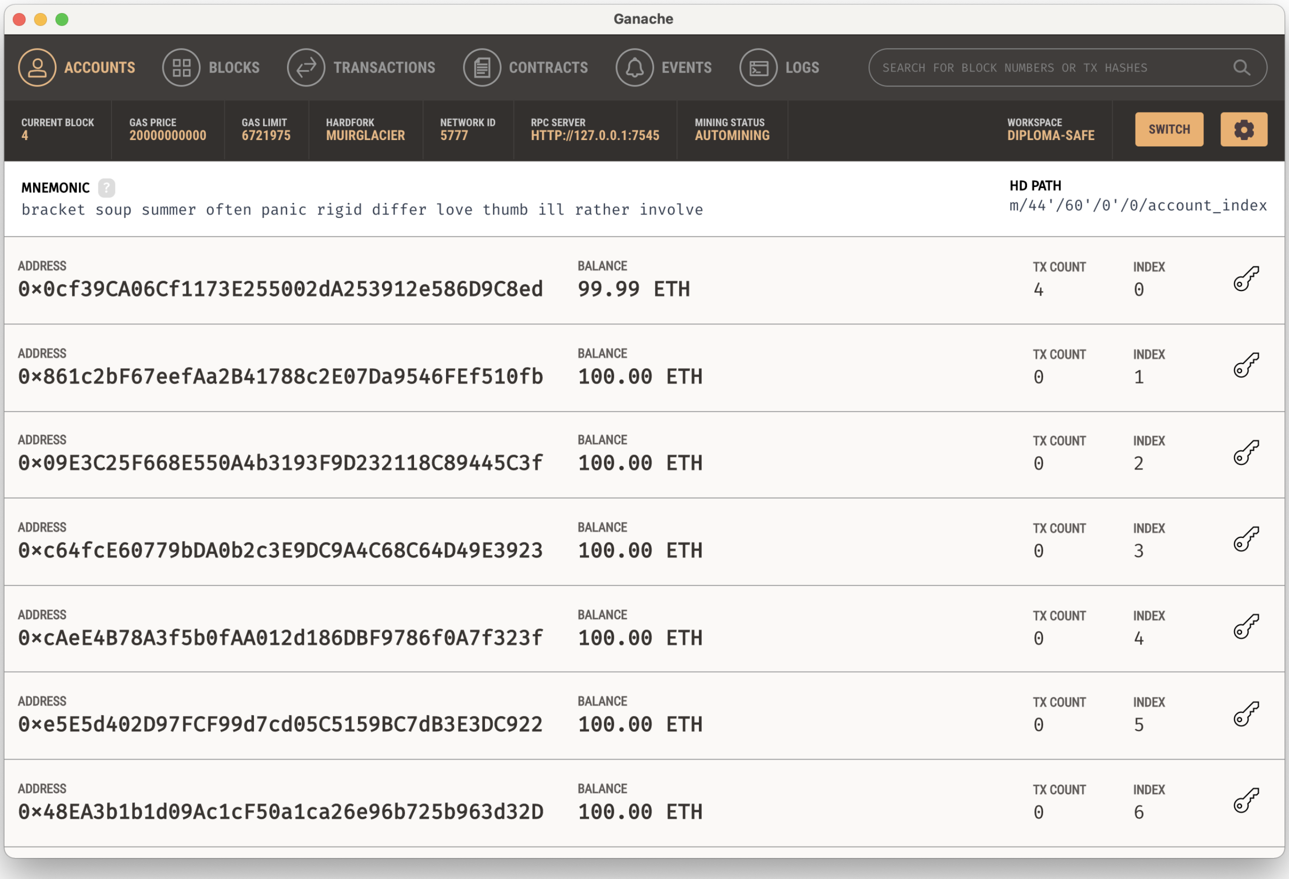
Now that our contract is deployed, we can interact with it using commands in the truffle interpreter. To fire the interpreter, use the truffle console command. When the prompt spawn, we can interact directly with the blockchain. All the smart contracts can be accessed as global variables in the interpreter so let's access our newly created contract and check the name variable. We see that we retrieve the value of the initial variable name. Note that we have to use the await keyword in order to wait for the response of the blockchain asynchronously. Because it can be quite long to get a response to our query, we have to wait until the blockchain answer. Not using this keyword will result in a Promise object, like these used in Javascript.


blockchain-chat-app — truffle console — truffle — node /usr/l...

```
tristanbilot at root in ~/Desktop/Desktop/EPITA/S9/crypto/tp4/project
/blockchain-chat-app on master!
 ± truffle console
truffle(development)> contract = await ChatApp.deployed()
undefined
truffle(development)> name = await contract.name()
undefined
truffle(development)> name
'test'
truffle(development)> 
```

Figure 6: Our contract is deployed to the blockchain

When looking back to our ganache interface, we see that our first wallet adress has decreased by 0.01 ETH. This is because every transaction done on a Ethereum blockchain isn't free, we have to pay for them to be achieved. This price we pay to make transactions is called **gas**. This gas is fixed by the blockchain and is applied to all participants using the blockchain.



CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
4	20000000000	6721975	MUIRGLACIER	5777	HTTP://127.0.0.1:7545	AUTOMINING	DIPLOMA-SAFE

MNEMONIC
bracket soup summer often panic rigid differ love thumb ill rather involve

HD PATH
m/44'/60'/0'/0/account_index

ADDRESS	BALANCE	TX COUNT	INDEX
0x0cf39CA06Cf1173E255002dA253912e586D9C8ed	99.99 ETH	4	0
0x861c2bF67eefAa2B41788c2E07Da9546FEf510fb	100.00 ETH	0	1
0x09E3C25F668E550A4b3193F9D232118C89445C3f	100.00 ETH	0	2
0xc64fcE60779bDA0b2c3E9DC9A4C68C64D49E3923	100.00 ETH	0	3
0xcAeE4B78A3f5b0fAA012d186DBF9786f0A7f323f	100.00 ETH	0	4
0xe5E5d402D97FCF99d7cd05C5159BC7dB3E3DC922	100.00 ETH	0	5
0x48EA3b1b1d09Ac1cF50a1ca26e96b725b963d32D	100.00 ETH	0	6

Figure 7: View from Ganache

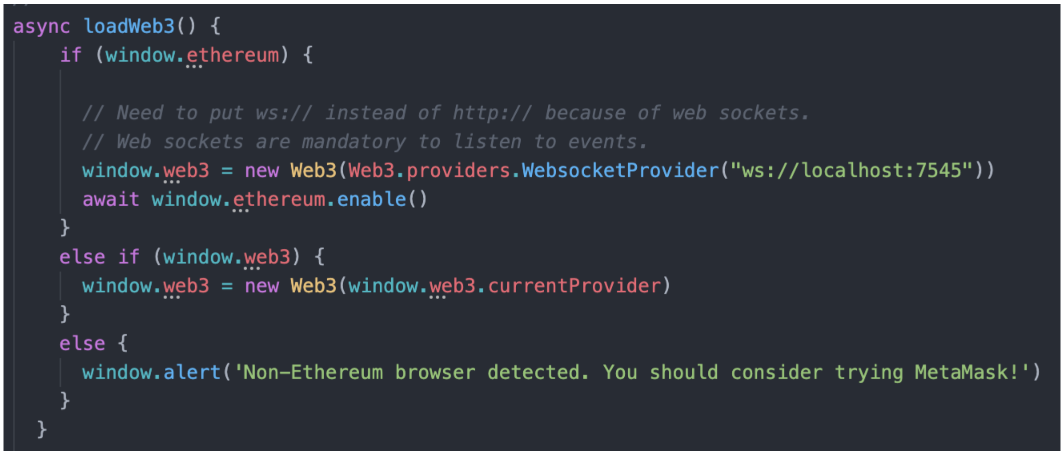
3 Messaging app: blockchain-side

At this step, the blockchain is setup, let's dive into the code of our blockchain messaging app! These are the features implemented by the web app:

- connect to all the available wallet addresses available in Ganache
- send messages between these addresses
- store all the messages in the smart contract in order to fetch them back when the page is reloaded
- monitor the state of the blockchain in real time when the transactions are executed
- bonus: send ethereum between the addresses

3.1 Connect browser to Ganache wallets

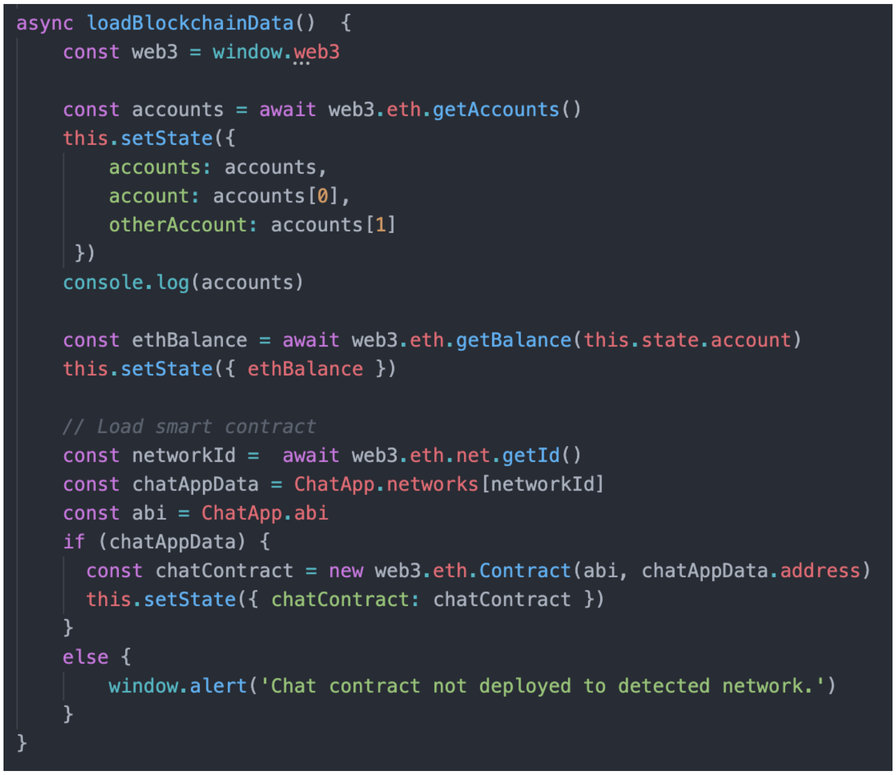
In order to communicate between the blockchain and the browser, we use a JavaScript library named web3.js. In this project, we'll use as UI framework react.js because it is easy to use and allows reactive programming. Basically, web3 is an API which maps high level commands to low level RPC instructions sent to the Ethereum blockchain. We just have to connect web3 to an instance of our Ganache server and we'll access the data of the blockchain via API methods. In our case, we'recommunicating with the blockchain not with http but with websocket because we'll need live data transfer using the events in solidity: more on that later. This is the code used to connect the Ganache server launched on localhost.



```
async loadWeb3() {
    if (window.ethereum) {

      // Need to put ws:// instead of http:// because of web sockets.
      // Web sockets are mandatory to listen to events.
      window.web3 = new Web3(Web3.providers.WebsocketProvider("ws://localhost:7545"))
      await window.ethereum.enable()
    }
    else if (window.web3) {
      window.web3 = new Web3(window.web3.currentProvider)
    }
    else {
      window.alert('Non-Ethereum browser detected. You should consider trying MetaMask!')
    }
  }
```

Figure 8: Connect web3 to the blockchain

Now that the blockchain is connected, it is possible to fetch data from the blockchain. In the following function, we fetch all the available account addresses in Ganache and store them in the state of the react component. Note that we store the first account of the list of accounts as the specific account of the current user on the page (the default address used to send messages later). This is also here that we fetch the instance of the Chat smart contract that we created previously. This instance is fetched using the abi json file of the contract: smart contract serialized as json file when compiling and deploying the smart contract using truffle. This json contains many information in order to find the contract when parsing it. The contract instance will be used later. Note that these two functions are triggered at the loading of the react component (page).



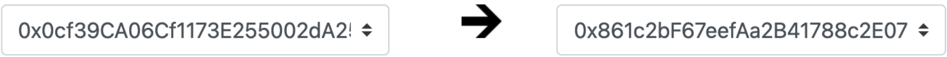
```javascript
async loadBlockchainData()  {
    const web3 = window.web3

    const accounts = await web3.eth.getAccounts()
    this.setState({
        accounts: accounts,
        account: accounts[0],
        otherAccount: accounts[1]
     })
    console.log(accounts)

    const ethBalance = await web3.eth.getBalance(this.state.account)
    this.setState({ ethBalance })

    // Load smart contract
    const networkId =  await web3.eth.net.getId()
    const chatAppData = ChatApp.networks[networkId]
    const abi = ChatApp.abi
    if (chatAppData) {
      const chatContract = new web3.eth.Contract(abi, chatAppData.address)
      this.setState({ chatContract: chatContract })
    }
    else {
        window.alert('Chat contract not deployed to detected network.')
    }
}
```

Figure 9: Fetch users from the blockchain

3.2 Messages

3.2.1 Address selection

To send messages, we suppose we can open two different pages in a browser, select two distinct addresses and send text messages between them. The first thing to do is to create a UI component so the user can select an address to send the messages with. We'll use a select html input for this address selection feature.



0x0cf39CA06Cf1173E255002dA2

0x861c2bF67eefAa2B41788c2E07

Figure 10: Address selection: sender and recipient

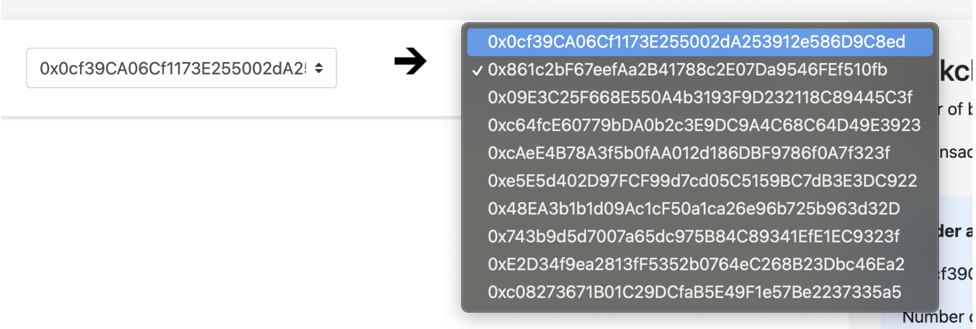
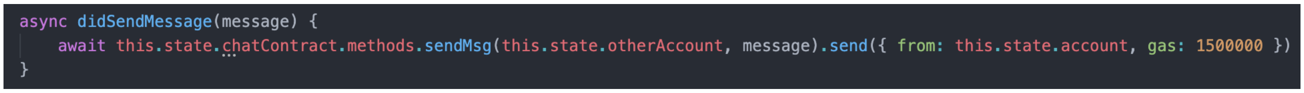

0x0cf39CA06Cf1173E255002dA2!
0x0cf39CA06Cf1173E255002dA253912e586D9C8ed
✓ 0x861c2bF67eefAa2B41788c2E07Da9546FEf510fb
0x09E3C25F668E550A4b3193F9D232118C89445C3f
0xc64fcE60779bDA0b2c3E9DC9A4C68C64D49E3923
0xcAeE4B78A3f5b0fAA012d186DBF9786f0A7f323f
0xe5E5d402D97FCF99d7cd05C5159BC7dB3E3DC922
0x48EA3b1b1d09Ac1cF50a1ca26e96b725b963d32D
0x743b9d5d7007a65dc975B84C89341EfE1EC9323f
0xE2D34f9ea2813fF5352b0764eC268B23Dbc46Ea2
0xc08273671B01C29DCfaB5E49F1e57Be2237335a5

Figure 11: Address selection: sender and recipient

Each selection of a new address triggers a function which save the selected address as the current address of the user.

3.2.2 Send messages

Sending messages via the blockchain will be done using Solidity **events**. Events work the same way as with software architectures like MVVM or Block. A client is **listening** for incoming data after **subscribing** to a type of event. Then an event can be sent and all the clients will be notified. This is a perfect architecture for messaging. Note that each event uses some gas on the blockchain so it has a price. In general, everything that uses power on a blockchain is not free, we have to pay that power using tokens (ETH here).



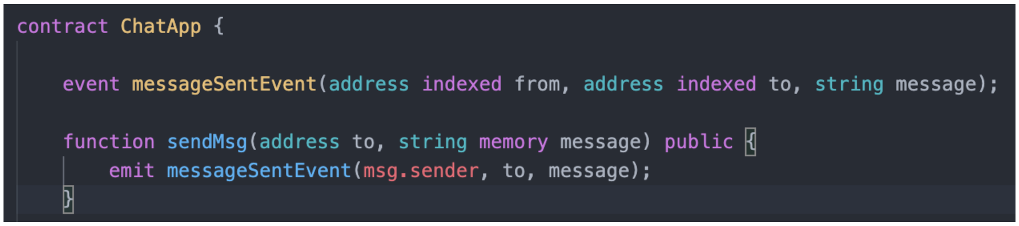
```
async didSendMessage(message) {
    await this.state.chatContract.methods.sendMsg(this.state.otherAccount, message).send({ from: this.state.account, gas: 1500000 })
}
```

Figure 12: Web3 code used to send a message

The message is sent using the ChatApp contract we saved in state previously. We can access the methods of the contract and then we use this method:

$$sendMsg(to\_address, message)$$

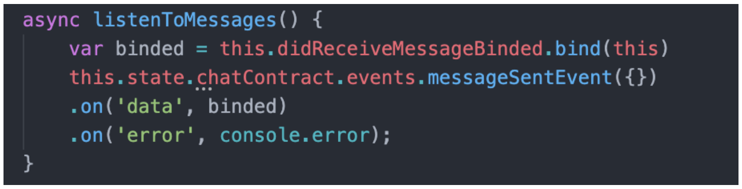
We specify enough gas because the method contains some memory instructions so it uses a big amount of gas.



```
contract ChatApp {

    event messageSentEvent(address indexed from, address indexed to, string message);

    function sendMsg(address to, string memory message) public {
        emit messageSentEvent(msg.sender, to, message);
    }
```

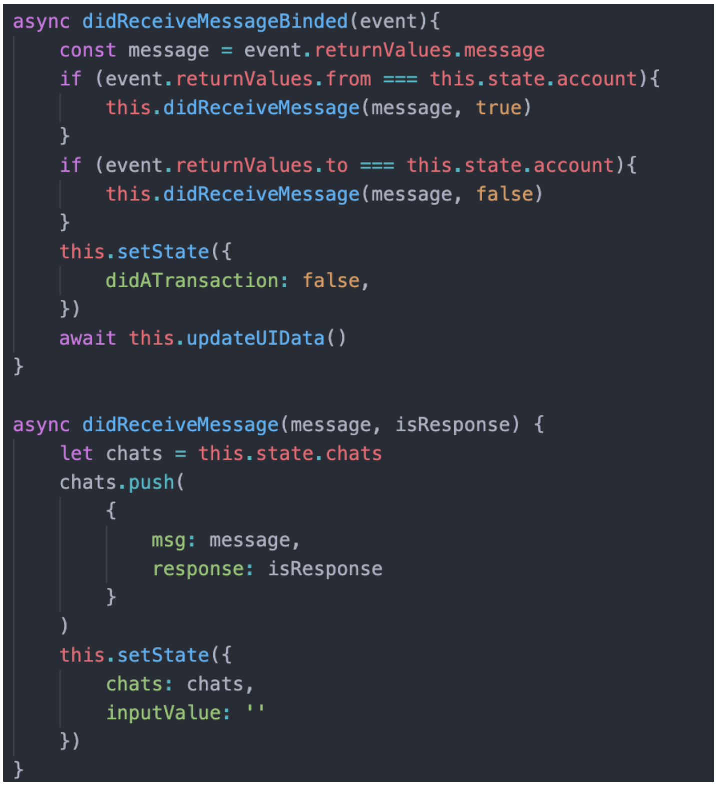
Figure 13: Solidity code used to send an event of received message

When the method will be called, an event will be sent back to the subscribers and trigger this function:



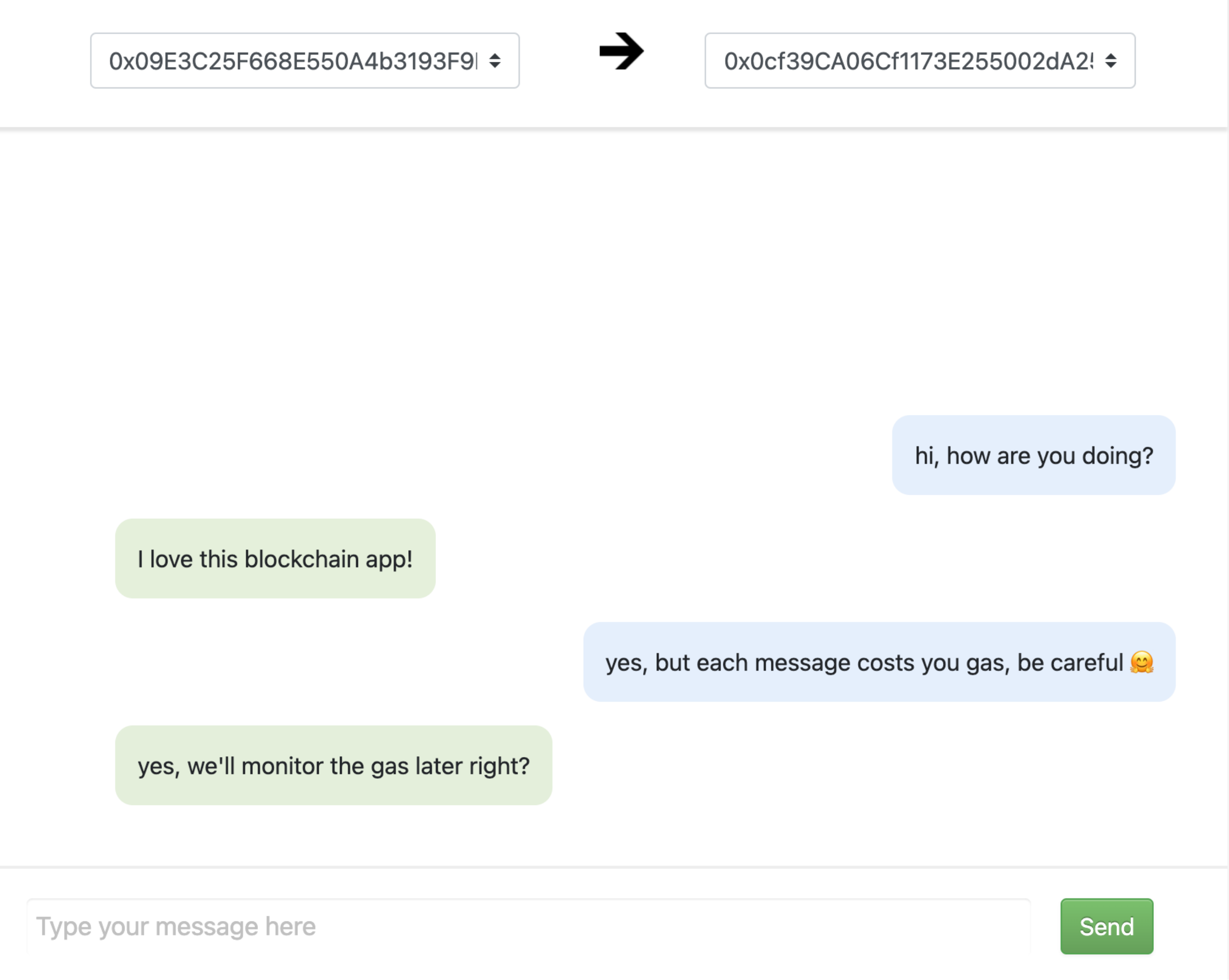
```
async listenToMessages() {
    var binded = this.didReceiveMessageBinded.bind(this)
    this.state.chatContract.events.messageSentEvent({})
    .on('data', binded)
    .on('error', console.error);
}
```

Figure 14: Code used to listen the incoming message events



```
async didReceiveMessageBinded(event){
    const message = event.returnValues.message
    if (event.returnValues.from === this.state.account){
        this.didReceiveMessage(message, true)
    }
    if (event.returnValues.to === this.state.account){
        this.didReceiveMessage(message, false)
    }
    this.setState({
        didATransaction: false,
    })
    await this.updateUIData()
}

async didReceiveMessage(message, isResponse) {
    let chats = this.state.chats
    chats.push(
        {
            msg: message,
            response: isResponse
        }
    )
    this.setState({
        chats: chats,
        inputValue: ''
    })
}
```

Figure 15: Handler function triggered when the event is coming



0x09E3C25F668E550A4b3193F9I → 0x0cf39CA06Cf1173E255002dA2!

hi, how are you doing?

I love this blockchain app!

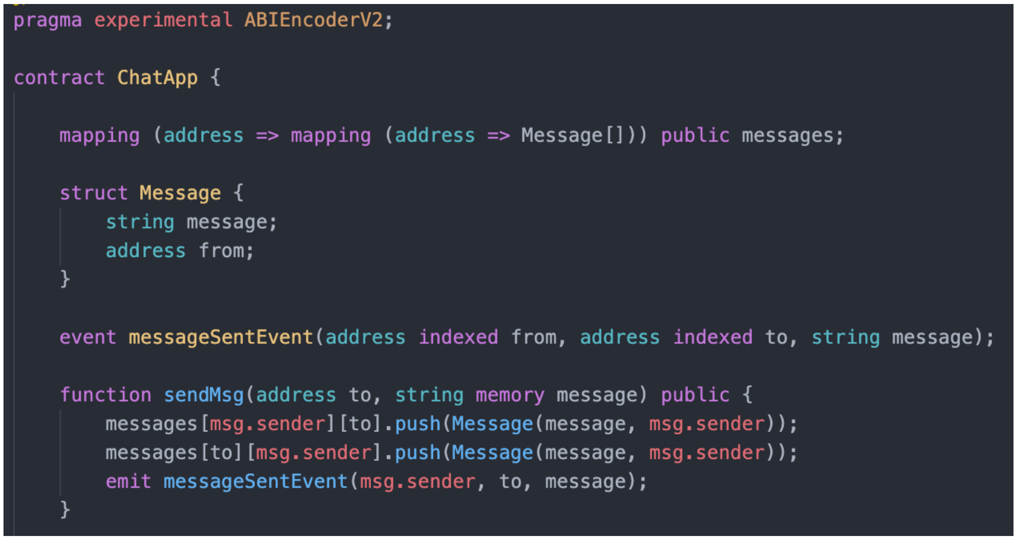
yes, but each message costs you gas, be careful 🤗

yes, we'll monitor the gas later right?

Figure 16: Results of the messaging part

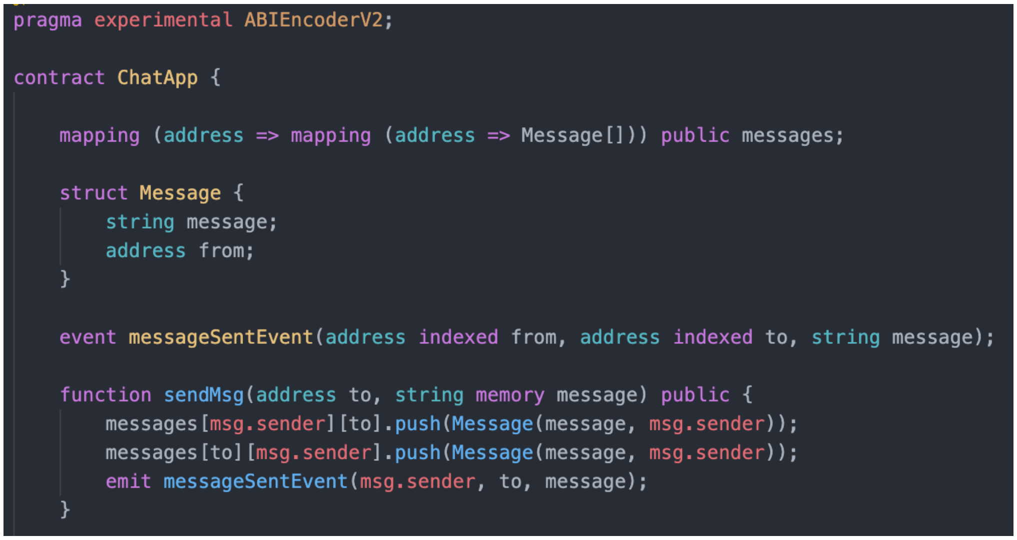
3.3 Store messages in the blockchain

At the moment, the messages are just emitted to the clients but if we reload the page of the browser, messages disappear because there is no persistence. Messages are not saved in memory nor storage. In solidity, the mapping object is a hashmap-like data-structure. We'll use two nested mapping objects: One mapping the address of the sender to the address of the receiver and finally mapping to a list of messages: the conversation. So we can now fetch the messages if we just know the two adresses. Of course, this is not a real use-case and in a real-life project this solution will definitely not be chosen but this is not the aim of the project. The from address in the Message structure is used to determine if the messages are sent or received from each of the two addresses.



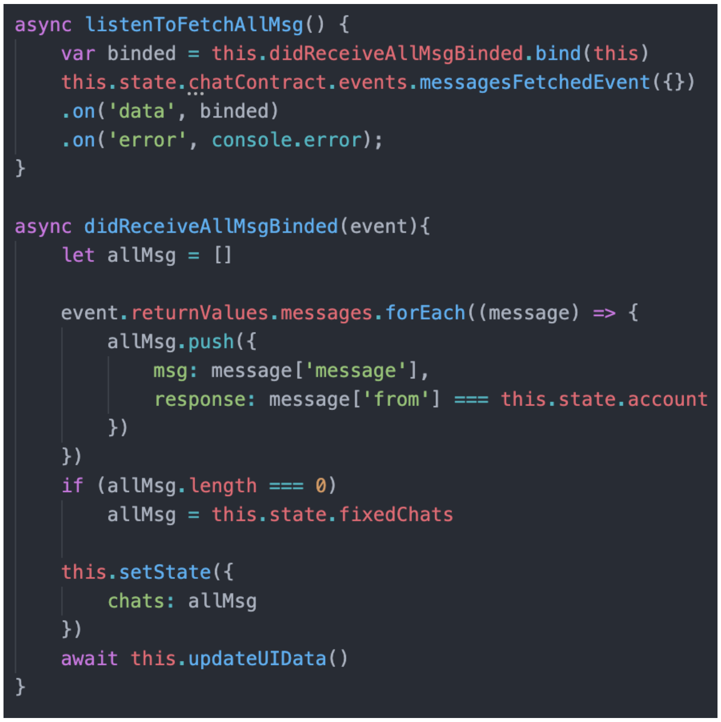
```
pragma experimental ABIEncoderV2;

contract ChatApp {

    mapping (address => mapping (address => Message[])) public messages;

    struct Message {
        string message;
        address from;
    }

    event messageSentEvent(address indexed from, address indexed to, string message);

    function sendMsg(address to, string memory message) public {
        messages[msg.sender][to].push(Message(message, msg.sender));
        messages[to][msg.sender].push(Message(message, msg.sender));
        emit messageSentEvent(msg.sender, to, message);
    }
```

Figure 17: Client method used to fetch all the messagesfor the two selected addresses



```
pragma experimental ABIEncoderV2;

contract ChatApp {

    mapping (address => mapping (address => Message[])) public messages;

    struct Message {
        string message;
        address from;
    }

    event messageSentEvent(address indexed from, address indexed to, string message);

    function sendMsg(address to, string memory message) public {
        messages[msg.sender][to].push(Message(message, msg.sender));
        messages[to][msg.sender].push(Message(message, msg.sender));
        emit messageSentEvent(msg.sender, to, message);
    }
```

Figure 18: Solidity code used to send an event of received message

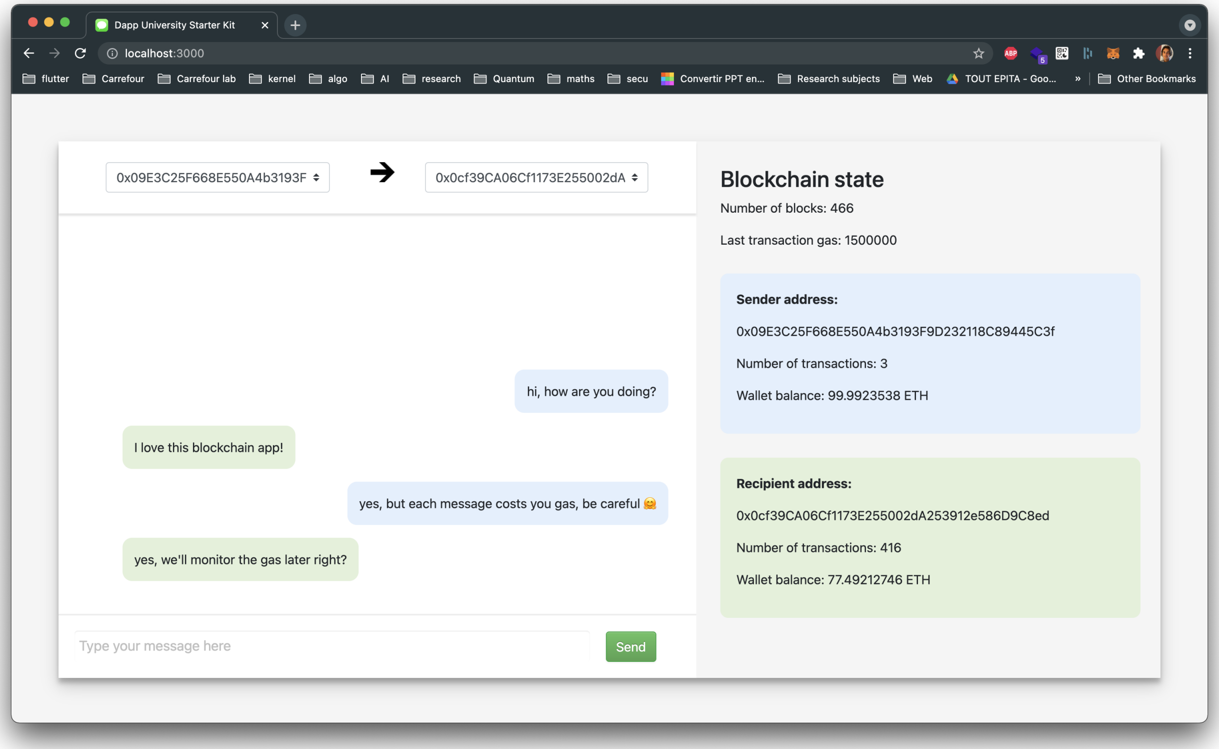


```
async listenToFetchAllMsg() {
    var binded = this.didReceiveAllMsgBinded.bind(this)
    this.state.chatContract.events.messagesFetchedEvent({})
    .on('data', binded)
    .on('error', console.error);
}

async didReceiveAllMsgBinded(event){
    let allMsg = []

    event.returnValues.messages.forEach((message) => {
        allMsg.push({
            msg: message['message'],
            response: message['from'] === this.state.account
        })
    })
    if (allMsg.length === 0)
        allMsg = this.state.fixedChats

    this.setState({
        chats: allMsg
    })
    await this.updateUIData()
}
```

Figure 19: Client method used to listen and handle the display of messages

3.4 Monitor blockchain state

Using web3, it is possible to fetch a vast quantity of information concerning the blockchain. In the following examples, we have fetched the ETH wallet of the two participants of the conversation, their addresses and number of transactions, but also the number of the last block in the blockchain and the gas used for the last transaction. Note that if you change the gas parameter in the send() function, the last transaction gas will be the same as the one used in the send() function.



0x09E3C25F668E550A4b3193F → 0x0cf39CA06Cf1173E255002dA

hi, how are you doing?

I love this blockchain app!

yes, but each message costs you gas, be careful 🤗

yes, we'll monitor the gas later right?

Blockchain state

Number of blocks: 466

Last transaction gas: 1500000

Sender address:

0x09E3C25F668E550A4b3193F9D232118C89445C3f

Number of transactions: 3

Wallet balance: 99.9923538 ETH

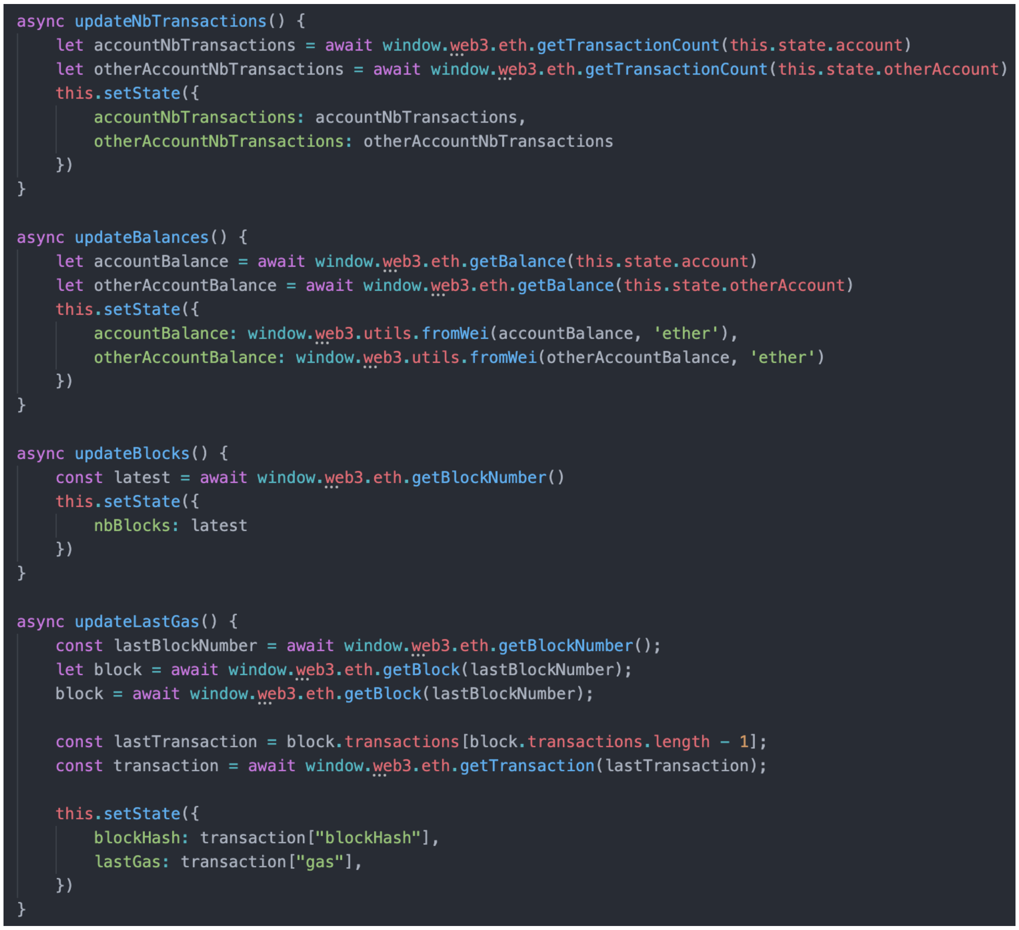
Recipient address:

0x0cf39CA06Cf1173E255002dA253912e586D9C8ed

Number of transactions: 416

Wallet balance: 77.49212746 ETH

Figure 20: Real time monitoring information of the blockchain



```
async updateNbTransactions() {
    let accountNbTransactions = await window.web3.eth.getTransactionCount(this.state.account)
    let otherAccountNbTransactions = await window.web3.eth.getTransactionCount(this.state.otherAccount)
    this.setState({
        accountNbTransactions: accountNbTransactions,
        otherAccountNbTransactions: otherAccountNbTransactions
    })
}

async updateBalances() {
    let accountBalance = await window.web3.eth.getBalance(this.state.account)
    let otherAccountBalance = await window.web3.eth.getBalance(this.state.otherAccount)
    this.setState({
        accountBalance: window.web3.utils.fromWei(accountBalance, 'ether'),
        otherAccountBalance: window.web3.utils.fromWei(otherAccountBalance, 'ether')
    })
}

async updateBlocks() {
    const latest = await window.web3.eth.getBlockNumber()
    this.setState({
        nbBlocks: latest
    })
}

async updateLastGas() {
    const lastBlockNumber = await window.web3.eth.getBlockNumber();
    let block = await window.web3.eth.getBlock(lastBlockNumber);
    block = await window.web3.eth.getBlock(lastBlockNumber);

    const lastTransaction = block.transactions[block.transactions.length - 1];
    const transaction = await window.web3.eth.getTransaction(lastTransaction);

    this.setState({
        blockHash: transaction["blockHash"],
        lastGas: transaction["gas"],
    })
}
```

Figure 21: Some methods used to fetch and display data from the blockchain

3.5 Send ether

To better understand the transations in the Ethereum blockchain, it is interesting to check how work the real ether transactions between wallet addresses.

3.5.1 Input format for ether sending

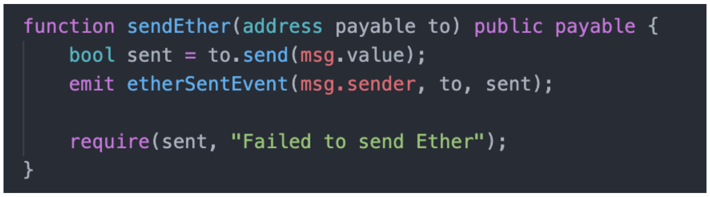
For the sake of simplicity, we'll reuse the text input of the messages and parse a command inside to check if the client has to process an ether or message sending. The following syntax is used to send a transaction of n ether.

$$send\_ether : n$$

Where n is a floating point number such as 0.0002, 3, 0.123 representing the number of ether to send. If the number of ether sent if larger than the displayed wallet amount, an error will appear.

3.5.2 Transaction from the smart contract

Sending ether tokens from an address to another is trivial using Solidity. It can be done using only two lines of code. Note that the function has to be "payable" because we are looking to send tokens. The amount of ether to transfer is passed by msg.value.

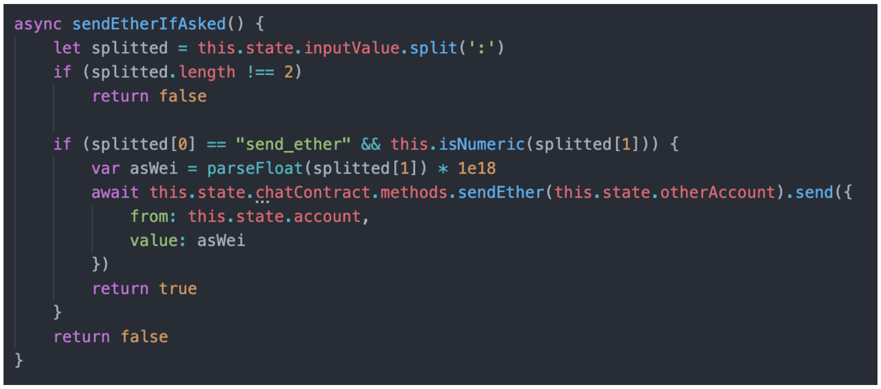


```
function sendEther(address payable to) public payable {
    bool sent = to.send(msg.value);
    emit etherSentEvent(msg.sender, to, sent);

    require(sent, "Failed to send Ether");
}
```

Figure 22: Solidity function used to send ether between two addresses

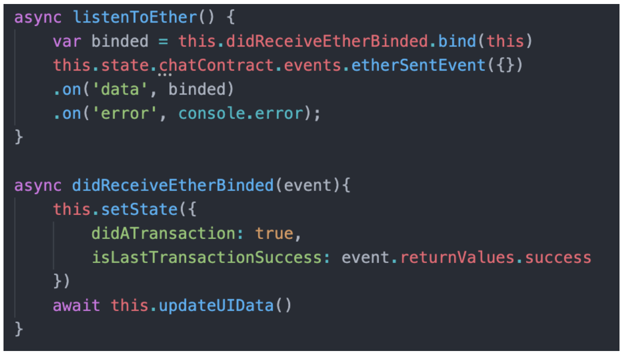
3.5.3 Transaction from the UI

From the UI, it is pretty much the same thing as the previous use cases, we listen to events from incoming ether transactions and apply UI updates when handling.



```
async sendEtherIfAsked() {
    let splitted = this.state.inputValue.split(':')
    if (splitted.length !== 2)
        return false

    if (splitted[0] == "send_ether" && this.isNumeric(splitted[1])) {
        var asWei = parseFloat(splitted[1]) * 1e18
        await this.state.chatContract.methods.sendEther(this.state.otherAccount).send({
            from: this.state.account,
            value: asWei
        })
        return true
    }
    return false
}
```

Figure 23: Client method used to send the transaction

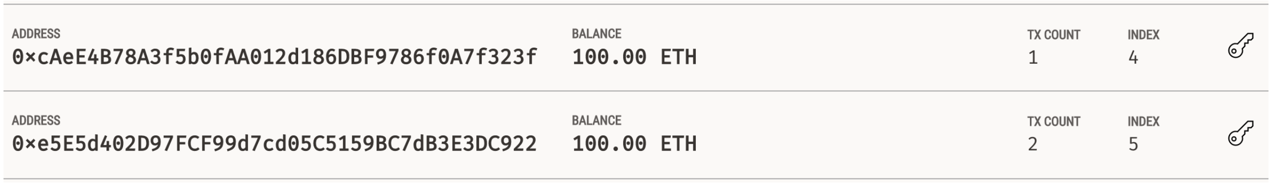


```
async listenToEther() {
    var binded = this.didReceiveEtherBinded.bind(this)
    this.state.chatContract.events.etherSentEvent({})
    .on('data', binded)
    .on('error', console.error);
}

async didReceiveEtherBinded(event){
    this.setState({
        didATransaction: true,
        isLastTransactionSuccess: event.returnValues.success
    })
    await this.updateUIData()
}
```

Figure 24: Client methods used to handle and display the transaction updates

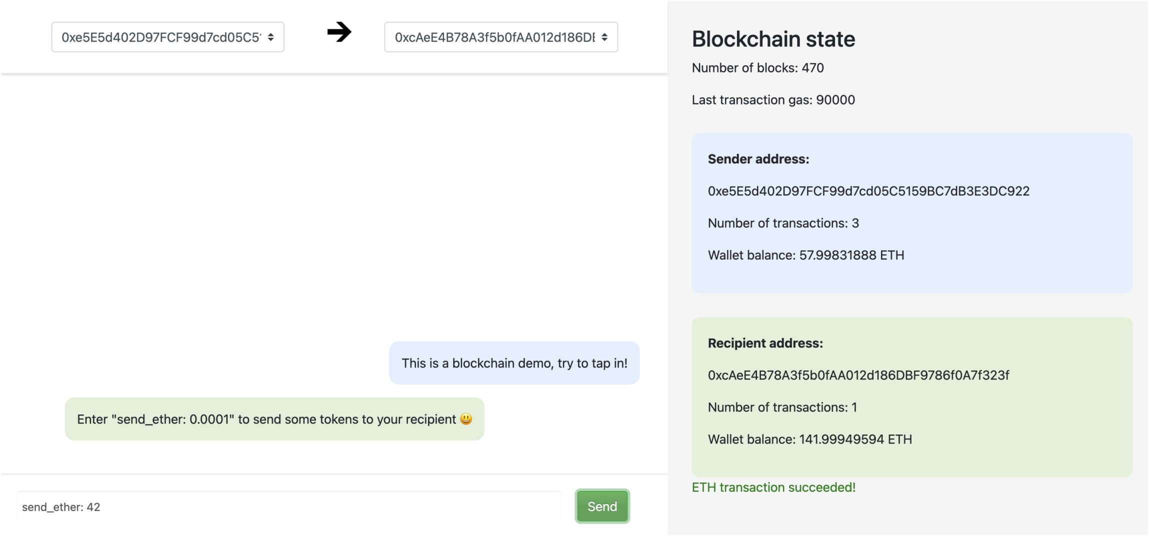
3.5.4 Example

The following is an example of a user sending 42 ether between two selected addresses. Almost instantly after sending the transaction, the balances of wallets are updates, another block is added, and transactions are incremented on both addresses.



ADDRESS	BALANCE	TX COUNT	INDEX
0×cAeE4B78A3f5b0fAA012d186DBF9786f0A7f323f	100.00 ETH	1	4
0×e5E5d402D97FCF99d7cd05C5159BC7dB3E3DC922	100.00 ETH	2	5

Figure 25: Wallets before transaction of 42 ether



This is a blockchain demo, try to tap in!

Enter "send_ether: 0.0001" to send some tokens to your recipient 😀

send_ether: 42

Blockchain state

Number of blocks: 470

Last transaction gas: 90000

Sender address:

0xe5E5d402D97FCF99d7cd05C5159BC7dB3E3DC922

Number of transactions: 3

Wallet balance: 57.99831888 ETH

Recipient address:

0xcAeE4B78A3f5b0fAA012d186DBF9786f0A7f323f

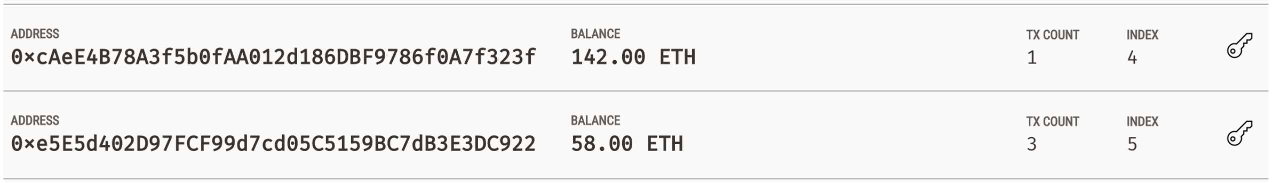
Number of transactions: 1

Wallet balance: 141.99949594 ETH

ETH transaction succeeded!

Figure 26: Transaction processed on the web UI

Looking back to Ganache, the wallet balances are also updated in real time.



ADDRESS	BALANCE	TX COUNT	INDEX
0×cAeE4B78A3f5b0fAA012d186DBF9786f0A7f323f	142.00 ETH	1	4
0×e5E5d402D97FCF99d7cd05C5159BC7dB3E3DC922	58.00 ETH	3	5

Figure 27: Wallets after transaction of 42 ether

4 Improvements

- using RSA encryption for messages
- using IPFS for file sharing
- optimize gas consumption in smart contract