Cloud Computing Project Report
Internet Connection Sharing

Kostas Tzoumpas, Marco Zenere

June 23rd 2021

Contents

1 Introduction **1**
1.1 Intuition . . . 1
1.2 Domain specification . . . 1
1.3 Expected results . . . 1

2 Architecture **2**
2.1 Requirements . . . 2
2.1.1 VPN Server . . . 2
2.1.2 Web Server / Client . . . 2
2.2 Important notes . . . 3
2.3 How to run . . . 3
2.3.1 VPN Server . . . 4
2.3.2 WebServer / Client . . . 4
2.4 Decisions made . . . 4

3 Implementation **6**
3.1 Back-end . . . 6
3.1.1 Smart Contract . . . 6
3.1.2 VPN Server . . . 7
3.2 Front-end . . . 9
3.2.1 Web Server . . . 9

4 Conclusion **14**
4.1 Difficulties and Lessons learned . . . 14
4.2 Possible improvements . . . 14

1 Introduction

1.1 Intuition

Blockchain technology has gained popularity in the recent years due to the events surrounding the most famous cryptocurrency based on this technology, Bitcoin. With the idea of learning more about this topic, we decided to implement an internet sharing system involving a blockchain. The every day case scenario is a user that wants to share his/her home internet connection with other users and get paid for it. This can be useful when someone is not at home and wants to use a safer and faster internet connection than the free hotspots normally available in cafes or in public places.

1.2 Domain specification

The system runs in an Ethereum blockchain which provides smart contracts feature, programs to store on a blockchain that run when predetermined conditions are met. "They typically are used to automate the execution of an agreement so that all participants can be immediately certain of the outcome, without any intermediary's involvement or time loss. They can also automate a workflow, triggering the next action when conditions are met"[1].The contracts available on the system are based on how many hours the customer wants to use the internet connection and the only means of payment is Ethereum cryptocurrency. Lastly, the OpenVPN is used as framework for the availability of the personal connection to the payer of the contract.

1.3 Expected results

The final result we expect is to have a system that can be accessed simply through a webpage where the customer will find the different contracts that can be purchased. When one of them is purchased, the customer must be able to access the network without too many steps and especially in a simple way, so that it is also accessible to the less experienced users.

2 Architecture

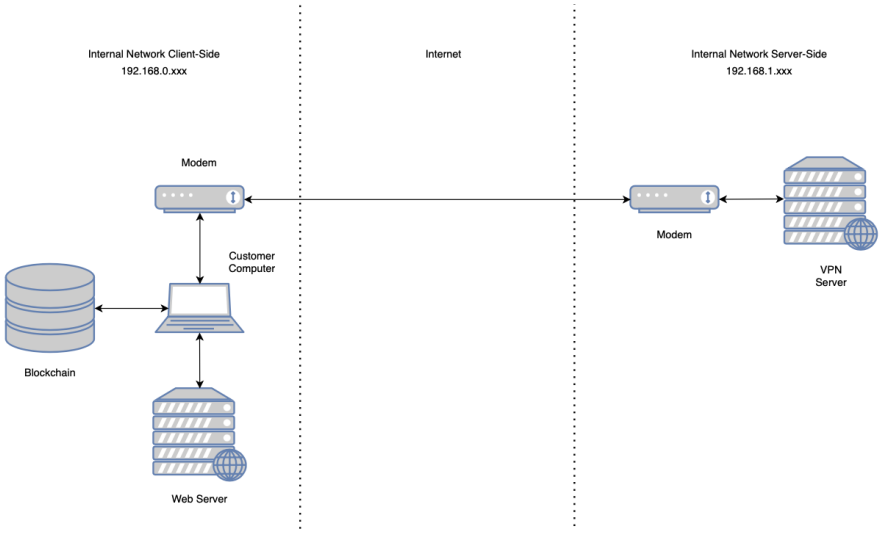

Figure 1: System Architecture

Figure 1 shows the architecture of the system developed. The system is composed of a server-side where there is a VPN server that manages the internet access of the customers connected to the provider's network, alongside the necessary scripts for the communication with the client. In the client-side, there are the Web server and the Blockchain.

2.1 Requirements

2.1.1 VPN Server

- Ubuntu Server OS (Tested with Ubuntu Server OS 32-bit 16.04)
- OpenVPN
- Python (3.5 or newer)
- PIP
- Websockets

2.1.2 Web Server / Client

- NPM
- Node.js

- Python
- PIP
- Chromium based browser (Firefox too, but make sure to disable the HTTPS-Only Mode for the websocket to run)
- Metamask browser extension
- OpenVPN client
- Ganache-cli
- Truffle

2.2 Important notes

- The server has a hostname from www.noip.com (kostasmarco.ddns.net)
- Port forwarding has been applied in the router's settings (server side) for the specific ports that listen to connections (1194 for OpenVPN, 1331 for Websockets)
- The server OS's firewall should be set for allowing the ports to be used for connection:

```
$ sudo ufw enable

$ sudo ufw allow {port_number}
```

- Ganache-cli is the command line version of Ganache. The information of the accounts is available in "ganache_output.txt" (hidden file in the project folder)
- The hidden file ".mnemonic.txt" can be used with the mnemonic phrase of the Ganache workspace so that the running of the program will instatiate the same accounts every time. This would be helpful when the user wants to avoid setting up the Metamask account more than once.
- Before running the starting script, make sure to save any work from running Ganache and front-end npm instances because the script "run.sh" kills them in order to set free the default ports (localhost ports: 7545, 3000)

2.3 How to run

In the Git repository of the Project, there are two README.md files, one in the main folder and one in the server folder. Both contain extensive information on how to deploy and run the systems. A similar guideline is given below.

2.3.1 VPN Server

1. Install OpenVPN:

```
$ sudo AUTO_INSTALL=y ./openvpn-install.sh
```

2. Make sure VPN service is active:

```
$ sudo systemctl enable openvpn
$ sudo systemctl restart openvpn
```

3. Make sure firewall is active (and ports 1194 and 1331 are allowed):

```
$ sudo ufw enable
```

4. Make sure all files of this folder are in the same directory

5. Start the Websocket Server:

```
$ python3 ws_server.py
```

2.3.2 WebServer / Client

1. Run the script:

```
$ ./run.sh
```

2. After the purchase is completed, a pop-up in the web browser appears and after few seconds a configuration file for the VPN is downloaded. To connect to the VPN there are two ways:

   (a) On the command line:

```
$ sudo mv client.conf /etc/openvpn/client/client.conf

$ sudo openvpn /etc/openvpn/client/client.conf
```

   (b) Open the OpenVPN client and drag and drop the configuration file in "FILE" section

2.4 Decisions made

- The pet shop example proposed by Truffle was used as base for our project since we are not expert in web development
- For the purpose of the demo, the Web Server and the blockchain are hosted by the client-side and the contracts aren't valid for hours but minutes. Furthermore, we decided to keep running the transactions on the local network (using Ganache) instead of deploying it on a test network or even the main Ethereum network.

- Using a web browser as a Javascript interpreter, the "require" keyword for getting the libraries doesn't work. We encountered this problem in the final part of the development of the code when we were trying to understand how to connect to our VPN server and get the configuration file. To address the issue we decided to use the WebSocket implementation which is already available to be used from the browser version of Javascript, without requiring any extra packages.

3 Implementation

3.1 Back-end

3.1.1 Smart Contract

```
pragma solidity ^0.5.0;

contract InternetConnection{

    address payable private owner = 0x0000000000000000000000000000000000000000;

    constructor() public {
        owner = msg.sender;
    }

    event productPurchased(address payable owner, address payable buyer, uint product_id, uint price );

    // Buy an internet service
    function purchaseProduct(uint productId) public payable{

        require(productId >= 0 && productId <= 4);
        require(owner != msg.sender);

        owner.transfer(msg.value);

        emit productPurchased(owner, msg.sender, productId, msg.value);
    }

    // Get the owner address
    function getOwner() public view returns (address){
        return owner;
    }
}
```

Figure 2: Smart Contract Code

Figure 2 shows the code written in Solidity developed for implementing a smart contract. The smart contract has three functions:

1. constructor: It is called when the contract is migrated in the blockchain and it takes the address of the user that deploy the contract (in Ganache it is the first account of the list). The address is used during the payment phase and we assumed that the first account is the owner of the products

2. purchaseProduct: It is called every time a customer buys a product. At the beginning of the function, there are a check of the product id and the address of who wants to buy the product. In case all the checks are passed, there is a money transfer from the customer to the owner of the products. In the end, an event is emitted after the purchase is completed successfully.

3. getOwner: It gives back the owner address

3.1.2 VPN Server

The server includes the VPN installation which is achieved with the use of the "openvpn-install.sh" file, as described in the "How to run" section. The other two necessary files we created are "ws_server.py" and "revoke_user.sh".

The first one handles the communication with the client, where:

- It waits for a new connection and prints a welcoming message when one happens.

```
63
64  async def run(websocket, path):
65      # receive message
66      msg = await websocket.recv()
67      print(f"{msg}")
68
69      if 'Contract:\n' in msg:
70          vpn_file = create_vpn_profile(msg)
71
72          # send to client
73          await websocket.send(vpn_file)
74
75  start_server = websockets.serve(run, "192.168.1.138", 1331)
76
77  asyncio.get_event_loop().run_until_complete(start_server)
78  asyncio.get_event_loop().run_forever()
```

Figure 3: Websocket listening to new connections.

- It listens to messages, and if the message is in a specific format (starting with "Contract:"), then the function "create_vpn_profile" is called. It creates a new contract with a random name. Also, using the duration for which the client paid, it calls the function to automatically revoke the access to the VPN of that specific user.

```python

def create_vpn_profile(msg):
    """ This function runs after a new contract is received
    and creates a .ovpn file for that specific contract.
    Return: the link of the .ovpn file. """

    # clean message
    msg = msg.split("\n")[-1]

    # generate name
    name = ""
    for k in range(10):
        name += chr(random.randint(ord('a'), ord('z')))
    duration = msg.split(',')[0].split(':')[-1]
    time_now = msg.split(',')[1].split(':')[-1]
    name += time_now

    # save details
    details = {'name': name, 'duration': duration}

    print('\n\nVPN Profile details:')
    pprint(details)
    print('\n\n')

    # create ovpn file
    print('\n\n================= VPN PROFILE CREATION ================')
    command = f"sudo MENU_OPTION=1 CLIENT={name} AUTO_INSTALL=y ./openvpn-install.sh | grep added"
    os.system(command)
    print('\n\n')

    # automate removal of vpn file
    plan_cancellation(name, duration)

    # get file's context
    with open(f'{name}.ovpn') as f:
        data_file = f.readlines()
```

Figure 4: VPN profile creation.

- The "plan_cancellation" function runs the "revoke_user.sh" script.

```python

def plan_cancellation(name, duration):
    """ This function runs after a vpn profile is created and sends the
    cancellation command after a perido of time
    equal to the duration of the contract."""

    duration_n, unit = duration.split('_')

    # revoke a client with name "name" after time expires
    print('\n\n================= VPN PROFILE CANCELLATION ================')
    os.system(f"PROF_NAME={name} DURATION={duration_n} UNIT={unit} ./revoke_user.sh &")

```

Figure 5: VPN cancellation calling.

The second file ("revoke_user.sh") just runs the revoking using the "openvpnv-install.sh" after the end of duration of the contract. After that, the OpenVPN service is restarted in order to force the disconnection of the revoked user.

```
#!/bin/bash
sleep $DURATION$UNIT
sudo MENU_OPTION=2 CLIENTNUMBER=$(sudo tail -n +2 /etc/openvpn/easy-rsa/pki/index.txt
    | grep '^V' | cut -d '=' -f 2 | nl -s ') ' | grep $PROF_NAME
    | awk '{print $1}' | cut -c1) ./openvpn-install.sh | grep revoked
sudo service openvpn restart


```

Figure 6: Revoking the access.

3.2 Front-end

3.2.1 Web Server

As mentioned in section 2.4, we decided to take the pet shop example provided by Truffle to start our web application development. The web app is a mix of different languages: HTML, CSS and Javascript and it runs in a development environment using lite-server. Figure 7 shows how the homepage of our website looks like.

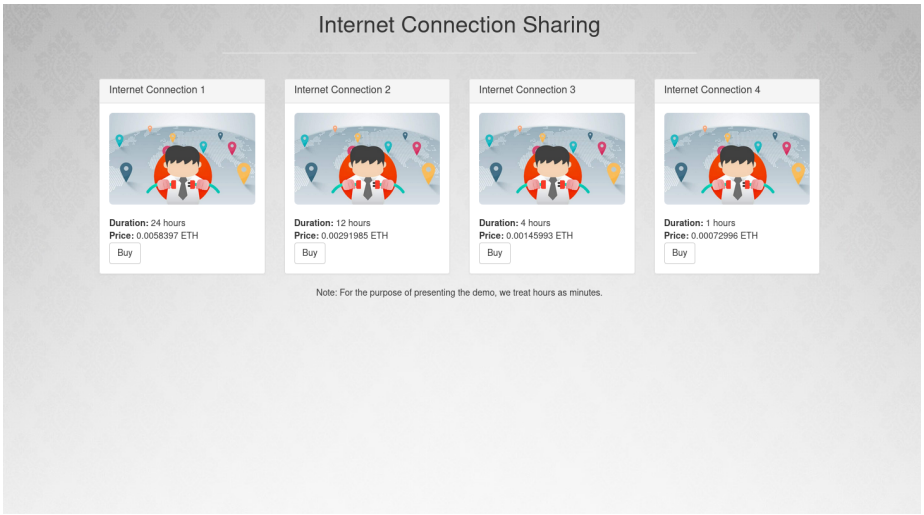

Figure 7: WebSite Homepage

There are four blocks that represent the available products and the characteristic of each one. The functionalities of the website are contained in "app.js" file and it has several changes with the respect to the pet shop in order to satisfy our requirements. Starting from the first access of the customer, we introduced two checks: a check to be sure that there is a connection with the VPN server and a check of the account address. The first one is used to be sure that after the payment the customer can receive the configuration file of the VPN. The

second one is to be sure that the 'buy' buttons are disabled in case the owner of the products access the website or to avoid that the customer can buy contracts after at least one of the owned contract is still valid.

```
connect_to_server: function() {
  // start a websocket client
  var ws = new WebSocket('ws://kostasmarco.ddns.net:1331');

  // check communication to server via websocket
  ws.onerror = function() {
    alert('Connection error with the server. Abort!');
  };

  return ws;
},
```

Figure 8: VPN Server connection check

The VPN server connection check is done by simply using WebSocket with the hostname for reaching the server. In case the server is not reachable, an error appears as a pop-up and the customer cannot buy anything.

```
markNotAvailable: function() {

  var now = new Date();
  var users = []

  // Check if some contracts expired
  for(let i=0; i<localStorage.length; i++) {
    let key = localStorage.key(i);
    if(now.getTime() >= localStorage.getItem(key)){
      localStorage.removeItem(key);
    }
    else{
      users.push(key);
    }
  }

  console.log(localStorage.length,'Items in local storage');

  if(loggedAccount == owner.toLowerCase() || users.includes(loggedAccount)){
    for (i = 0; i < product.length; i++) {
      $('.panel-product').eq(i).find('button').text('Not Available').attr('disabled', true);
    }
  }
},
```

Figure 9: Account logged check

The account logged check is done using the "markNotAvailable" function. In the first part of the function, there is a for loop with an if-else statement that is used to check if there is any timer related to the user and if is still valid in case there exist one. The timer is activated after the customer completed a purchase and the time depends on the contract he/she bought. To save the timer, we decided to use the local storage of the client browser and the data is saved as cookie. In the second part of the function, there is an if statement with the for-loop and it is used to disabled the 'buy' buttons in case the account logged is the owner or the customer is in the list of users with a valid timer associated. Looking at the payment phase, after the customer clicks one of the 'buy' buttons, Metamask interacts with the user in order to complete the purchase. In 'app.js' file, the payment is handled by 'handlePurchase' function.

```
handlePurchase: function(event) {
  event.preventDefault();

  var productId = parseInt($(event.target).data('id'));
  var purchaseInstance;
  var price = product[productId].price;
  var duration = product[productId].duration;

  web3.eth.getAccounts(function(error, accounts) {
    if (error) {
      console.log(error);
    }

    var account = accounts[0];
    console.log(account)

    var price_inWei = web3.toWei(price.toString(),'ether');

    App.contracts.InternetConnection.deployed().then(function(instance) {
      purchaseInstance = instance;

      // Execute purchase as a transaction by sending account
      return purchaseInstance.purchaseProduct(productId, {from: account, value: price_inWei});
    }).then(function(result) {
      console.log(result);
      console.log('Transaction Approved');
      return App.popUpScreen(duration);

    }).catch(function(err) {
      console.log(err.message);
    });
  });
 }
};
```

Figure 10: Code for handle the purchase

The function retrieves the information about the purchased product and after converting the price in Ethereum to Wei, the 'purchaseProduct' smart contract function is called to transfer the money from the customer to the product owner and then record the transaction in the blockchain. In case the purchase is completed successfully, a pop-up screen will appear on the web browser with an 'ok' button to start the download of the VPN configuration file and all instructions for using the file.

```
popUpScreen: function(duration){

  alert("Purchase Confirmed!\n\nBy clicking ok, the downloading of the VPN file will start (it can take about 5 seconds).\n\n
  // start of the contract
  var now = new Date();
  time_now = now.getTime();
  console.log("Sending contract details to the server and awaiting the file...");

  // send now+duration to the server
  ws = App.connect_to_server();

  ws.onopen = function(e) {
    // send details as string of a JSON
    const msg = "Contract:\nduration:"+duration.toString()+"_m"+",time_now:"+time_now.toString();
    // NOTE HERE: We treat hours as minutes, for the purpose of the demo.
    // Otherwise, "_m" should be replaced by "_h".
    ws.send(msg);
  };

  // wait for response (the VPN file)
  ws.onmessage = function(msg) {
    console.log("WebSocket message received.");

    // download it
    App.download_file("config.ovpn",msg.data);
  };

  // end of the contract
  now = now.setMinutes(now.getMinutes() + duration);
  localStorage.setItem(loggedAccount,now);
},
```

Figure 11: Code for the pop-up

The pop-up is managed by 'popUpScreen' function which not only confirms the purchase and displays instructions but also communicates via WebSocket with the server for sending the duration of the contract and receiving the configuration file for the VPN.

In conclusion, the front-end part handles mainly the payment part, while the system is managed by the server which communicates to the client for getting the contract duration and sending back the VPN file.

4 Conclusion

4.1 Difficulties and Lessons learned

The main difficulty we encountered was the link between the VPN server and Web server (client). We can confidently admit that the implementation of the two systems independently was achieved without significant difficulties. In contrast, the communication of the two was for a long period the key we were searching for, mainly because it was the first time we worked on a front-end program using programming languages unknown to our background (e.g. JavaScript). Our main goal was initially to handle an important amount of work on the client side but fortunately we moved to another plan where everything about the VPN is managed by the server, while the client side manages only the Blockchain and the payment. This was achieved by using Websockets for passing important information to the server and receiving the VPN file from it. In general, we learned that Websockets are suitable for easy and straightforward communication between machines, the Blockchain technology is not hard to understand and implement for beginners, and that web-browser JavaScript projects need some experience before trying to achieve advanced targets.

4.2 Possible improvements

Some of the steps forward that could be taken are the following:

- The Web server could be migrated to Amazon AWS using an EC2 instance, so that the webpage of the project can be accessible from anyone, anywhere. For this to happen, there would not be needed much configuration of the architecture as it is right now.
- Similarly, the Blockchain could be run on another network other than the local one (test network like Rinkeby, or even the main Ethereum network). For this improvement, there would be necessary to configure a bit differently the system.
- Finally, we use Websockets for communication but the connection is not secured by encryption (we us ws instead of wss). This happened because we encountered problems using the certificates on the client side, so for the purpose of this project we decided to use Websockets without encryption. This is something that would be necessary to be improved in future steps.

The above ideas are not present in this version of the project; firstly due to lack of remaining time (too much time lost for the solving of the problems mentioned in the above section) but also since already a sufficient amount of work and time has been put for the accomplished results.

References

[1] IBM, *IBM Smart Contracts: https://www.ibm.com/topics/smart-contracts*