Lab
Decentralized Application using Smart Contracts and IPFS: Healthcare Scenario

Version 1.1

LERSAIS
School of Computing and Information
University of Pittsburg

This lab is designed by *Runhua Xu* and also part of lab module for SAHI Project

Goal

The goal of this lab is to illustrate how to manage personal healthcare records using the decentralized application that are related on Ethereum smart contract and IPFS.

In this lab, given a decentralized application demo, you will learn the following objects:

1) The overview and usage of IPFS
2) The principle/usage/development of Ethereum smart contract
3) The combination of IPFS and smart contract to manage the healthcare records

Part 0 Required Tools

This lab requires the following tools:

1) Get *Node* and *npm* installed.
    a. https://www.npmjs.com/get-npm
    b. If you are not sure whether your system installed or not, you can check the following commands in the terminal environment.

    ```
    $ node -v
    $ npm -v
    ```

    If the commands return the corresponding version number, you are all set; otherwise, follow the instruction from the above link to install the tools.
2) Get *ipfs* installed
    a. Following the official instructions to install the *ipfs*. https://docs.ipfs.io/introduction/install/
3) Get *Truffle* installed
    a. Truffle is the most popular development framework for smart contracts.
    b. Follow the official instructions to install *truffle* https://truffleframework.com/truffle
4) Get *Ganache* or *Ganache CLI* installed
    a. https://truffleframework.com/docs/ganache/quickstart
    b. *Ganache CLI* is the command-line version (formerly known as the TestRPC) of *Ganache*. Just have one installed. In the rest of the lab, we will demonstrate using *Ganache*.

Note that the screenshot in the rest of the lab is captured in the macOS environment. The node and npm tool also support other operation system.

Part I IPFS

IPFS provides deduplication, high performance, and clustered persistence to organize the world's information. Besides, IPFS and the Blockchain are a perfect match. You can address large amounts of data with IPFS, and place the immutable, permanent IPFS links into a blockchain transaction. These timestamps and secures your content, without having to put the data itself on the chain.

You can find more details from IPFS documentation (https://docs.ipfs.io/)

As the IPFS has been installed successfully, here you will follow the instructions below to learn how the IPFS works.

1) Open your terminal and go to the lab folder, execute the IPFS initialization command
   `$ ipfs init`
   Where (which folder) is the IPFS node initialized in your environment? What is the peer identity?

2) Go to the IPFS initialization folder and check the content of config file
   `$ cd [your-ipfs-initialization-folder]`
   `$ cat config`
   What is the content of "Addresses" component in the config file?

3) Add the `"/ip4/127.0.0.1/tcp/9999/ws",` entry to your *Swarm* array in *Address* component.
4) Let's start a daemon of IPFS. Move back to your lab folder and execute the command
   `$ ipfs daemon`
   What is the output in your terminal window?

5) As the ipfs is running, you can start to run the demo app of IPFS. Move to app folder, and you need to install and bundle the dependencies to run the app.

```
$ cd ipfs-demo
$ npm install
$ npm run bundle
$ npm start
```

Are you able to run the app? What is the available access URL? List one connected peer.

6) Copy the access URL to your browser and try to upload a file to the IPFS.
What is the file name and file CID? What is the ID and Addresses of the current node?

7) Close the IPFS daemon and the opened browser window.

Part II Smart Contracts in Solidity

In this part, you will have an intuition of the smart contracts in Solidity. Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behaviour of accounts within the Ethereum state.

For more details, check the official document

(https://solidity.readthedocs.io/en/v0.5.4/index.html)

For demonstration of smart contracts, you will try to compile/deploy a very simple smart contact and interacting with your contract.

Try to follow the steps: *create/implement, compile, deploy,* to experience the procedure of your first smart contract.

1) Move to the lab folder and create a new folder called "hello". If you work in the terminal, the possible commands may be as

   $ mkdir hello

   $ cd hello

2) Move to the "hello" folder and initialize a truffle framework that will help you implement, compile and migrate a smart contract in Solidity. The corresponding command is as

   $ truffle init

   Does the command execute successfully? What's the output? What's the structure of the folder?

3) Let's create a "HelloWorld" smart contract using the *Truffle* framework command as follows:

   $ truffle create contract HelloWorld

   After that, is there any changes in the folder? What is the difference?

4) Open the newly created contract “HelloWorld.sol”. You can find that only one *constructor()* function listed in the contract. Let’s create a new simple function as follows:

```
pragma solidity ^0.5.0;


contract HelloWorld {
  constructor() public {
  }

  function hi() public pure returns (string memory) {
    return ("Hello World");
  }
}
```

5) Then, let’s compile the contract to the executable code for the EVM.

```
$ truffle compile
```

After that, what happens in the working folder (hello folder)?

6) Let’s deploy the contract to a simulated Ethereum network. In the *migrations* folder create a js file named “*2_deploy_contracts.js*” with the following content.

```
var HelloWorld = artifacts.require("./HelloWorld.sol");

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

Then, lunch the *Ganache* application and record the RPC SERVER address for your *Ganache*. Also, modify the configuration of the *truffle-config.js* file to adapt to the *Ganache* network.

```
  networks: {
    // Useful for testing. The `development` name is special - truffle uses it by default
    // if it's defined here and no other network is specified at the command line.
    // You should run a client (like ganache-cli, geth or parity) in a separate terminal
    // tab if you use this network and you must also set the `host`, `port` and `network_id`
    // options below to some value.
    //
    development: {
     host: "127.0.0.1",     // Localhost (default: none)
     port: 7545,            // Standard Ethereum port (default: none)
     network_id: 5777,       // Any network (default: none)
    },
```

Note that the network information should be consistent to your *Ganache* application.

7) The next step is run the contact in the simulated network. Let’s migrate the compiled code to the network.

```
$ truffle migrate
```

a) Before you execute the migrate command, check the accounts status. After the migration, is there any change to the account?
b) Which account deploy the contract?
c) How many contracts deployed in the network? What is/are it/them?
d) How many blocks generated in the network? How much gas used for each? How many transactions for each?

8) As the contract is deployed in the *Ganache* network, you can interact with the contract. In your terminal, execute the interactive console:
```
$ truffle console
```
Then execute the following command to call the *saysomething()* function you write in the contract:
```
$ truffle (development)>
HelloWorld.deployed().then(function(contractInstance){contractInstance.
saysomething().then(function(v){console.log(v)})})
```
Do you see the output of the *saysomething()* function?

9) Close all applications used in this part.

Part III A Decentralized Application for Managing Healthcare Records

As you have accomplished the task of the smart contract in the last part, you are familiar with the procedures of smart contract application. In this section, you will combine the IPFS and smart contract to construct a relatively complete application to manage personal healthcare records. Specifically, the personal records are encrypted and then send to decentralized application (dApp). The dApp first stores the received file to the IPFS and then send/store the identifier to the Ethereum network using the smart contract.

Follow the steps in this part.

1) Move to the dApp demo folder.
```
$ cd dapp-ipfs
```
2) Install the dependencies.
```
$ npm install
```

3) Compile the smart contract.

```
$ truffle compile
```

4) Open the *Ganache* and update the corresponding network information into the *truffle-config.js* file, as similar you have done in the previous section.

5) Migrate the smart contract.

```
$ truffle migrate
```

Check the *Ganache,*

a) How many blocks and transactions are generated/deployed?

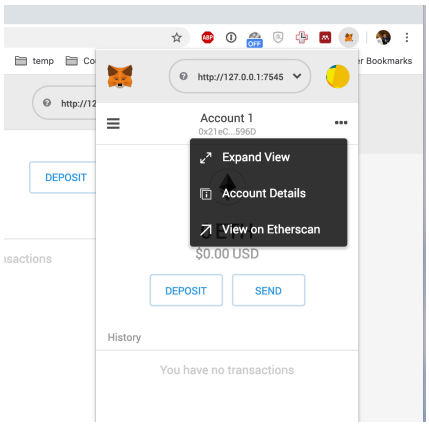
b) What is the cost for each contract?

c) What is the address of each contract?

6) As you can see, the first account in the *Ganache* has cost several ETH, let's use a plug-in to manage your ETH account.

Here we recommend the MetaMask (https://metamask.io/). Follow the instructions on the website to install the MetaMask.

Then you can set up the initial account for the MetaMask, and you can see this



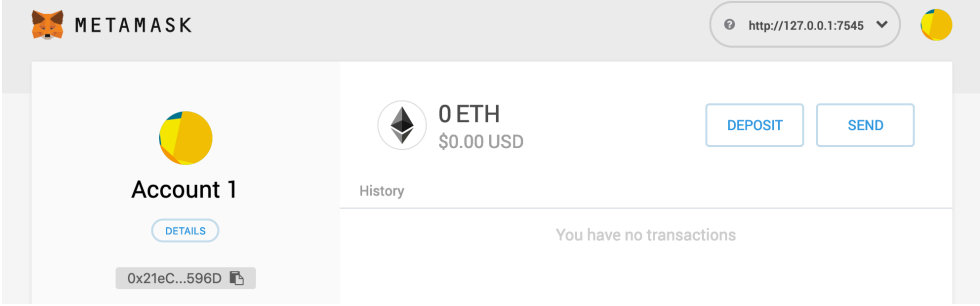
Click the expand view to explore the UI.

The next step is to configure your local simulated network Ganache and add the simulated account.

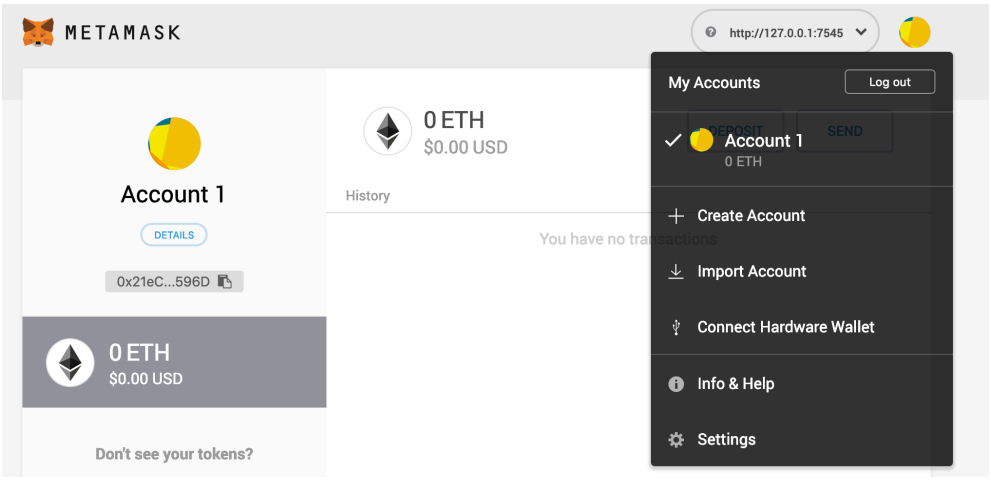
a) In the network option, choose to add a *Custom RPC* (the last option)

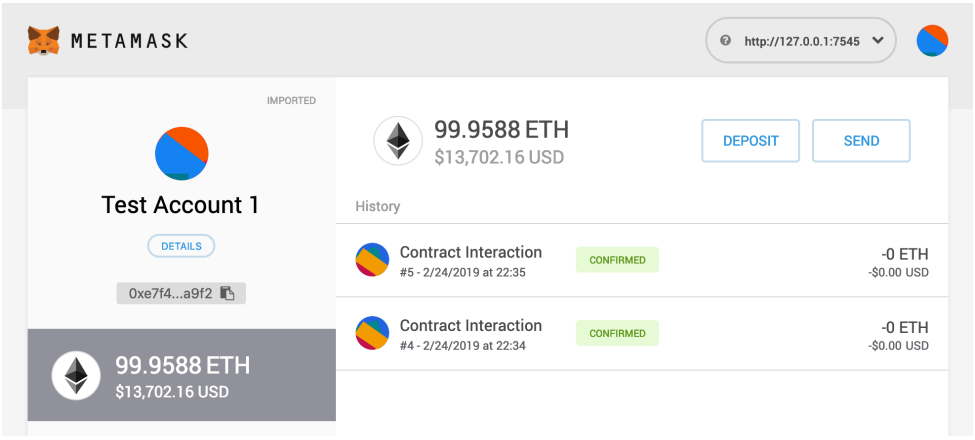
b) In the opened page, add the RPC information into the *New Network*, and Save the configuration.

c) Then you can change to the test RPC network.



d) Then you can add the first simulate account into MetaMask.





In addition, select another account from the *Ganache* and add to the MetaMask. For example, you have two accounts: ***Test 1*** and ***Test 2***, and keep the current account of metamask be ***Test 1***

7) Start the application

```
$ npm run start
```

8) Navigate to http://localhost:3000/ in your browser.

What is the current MetaMask account in the page? Change to another account in the MetaMask, is there any update in the page?

9) Try to upload an encrypted test record in the application and confirm the transaction when the pop-window of metamask plug-in is up.
   Answer the following questions:
   a) Are you successfully uploaded the record?
   b) What is the hash value of the block and transaction?
   c) How much gas is used?
   d) What is the IPFS hash value? Are you able to access/download the file?
   e) Try to change the current account to ***Test 2*** in the metamask, are you able to see the file uploaded with account ***Test 1***?

10) Close all applications used in this lab.