Lab 4: Blockchain Logging for Remote Data Storage

Lab Description

The Blockchain technology can be used in applications beyond cryptocurrency. In this lab, we explore the Blockchain application for logging the operation trace of a remote file-storage service. This features a common use of the Blockchain, that is, making the data about a third-party transparent to invite trusts. The learning objective of this lab is for students to design and implement a simple service for secure distributed file system (SDFS) and use the Blockchain technology to log the service related information. The target system consists of a client supporting multiple file-system users and a server storing a file.

Lab Environment

The SDFS includes classic information-security protocols for permission-based access control and password authentication. For simplicity, in this project you only need to consider the concrete and specific execution sequence as below: Initially, a server stores a single file "fileX". Two clients "userA" and "userB" can interact with the server in the following sequence:

```
user_login("userA", "pwd123");
user_login("userB", "pwd456");
file_permission_set("userA");
file_access("userA"); //success
file_access("userB"); //failure
file_delegate("userA", "userB");
file_access("userB"); //success
user_logout("userA");
user_logout("userB");
```

Tutorial on using geth in javascript/web3

To interact Ethereum blockchain through Javascript, you will need to use `web3` object in the `web3.js` library. The following sample code sends a transaction to Blockchain by Javascript:

```
web3.eth.sendTransaction({from:var1,data:var2,to:var3},function(var4,var5)
{
    if (var4)
      //Failed to send the transaction
    else
      // The transaction is acknowledged successful by the Blockchain
  });
```

The arguments of sendTransaction() are explained below:

- `var1` : This argument is an account address of the sender.
- `var2` : A byte string containing the associated data of the message.
- `var3` : The argument is an account address of receiver.
- `var4` : The argument is the error message if the request fails.
- `var5` : The argument is the result of the request after successful execution.

For more details, refer to the [link]

Lab Setup

Install `node.js`

```
sudo apt install curl
curl -sL https://deb.nodesource.com/setup_8.x | sudo bash -
sudo apt install nodejs
```

You can also refer here: https://nodejs.org/en/download/

You can refer https://nodejs.org/dist/ for specific versions of `node.js` . This lab is tested with: `node.js` (v8.11.1), `npm` (v5.6.0), `web3` (v0.20.0).

You can install the latest versions and work with them if you would prefer.

Instructions for running the provided program

This section requires you to know how to run a `geth` in a terminal which was covered in the first Lab [link]. To do that, you can have two options: S1) re-use all the data from lab1, or S2) hook web3 with a local node run by Ganache.

S1) Reuse Lab 1.

For example:

```
$ mkdir -p sdfs_lab
$ cp -r lab1/bkc_data sdfs_lab
```

- In this lab, you need to enable RPC on your `geth` terminal after you start `geth`

```
> admin.startRPC()
```

S2) Use a local node by Ganache

- Go to website: https://www.trufflesuite.com/ganache and install Ganache
- After Ganache is install, there will be a Ethereum node running on port 7545
- In the second line of [SDFC_lab.js], you can replace "http://127.0.0.1:8545" with "http://127.0.0.1:7545" there.

After an Ethereum node is up and running (either by S1 or S2), you can install web3 and hook it with the Ethereum node.

```
$ npm install web3@^0.20.0
```

Copy the [SDFC_lab.js] file to your current working directory, and run the program

```
$ nodejs SDFC_lab.js
```

Note: Make sure you have enough funds in your `eth.accounts[0]` and also make sure to unlock the account before running the program.

For convenience, we provide some methods that simulate the flow:

1. `user_Login = function(userId, pwd)` : A new session between client and the server starts by client logging into her account with password.
2. `user_Logout = function(userId)` : The session associated with userId is terminated by client logging out from her account.
3. `file_Access = function(user)` : Once logged in, a client can request to read the single file fileX stored in the server.
4. `file_permission_set = function(user)` : In SDFS, fileX has a read permission bit for each user. For instance, if the permission bit for userA is set, userA is allowed to read the file. Function file_permission_set(user) sets the permission bit for username.

Exercise 1: Tracing remote requests [grade 10%]

Define an empty function, called `bkc_logging(log)` . Insert the logging calls to the server code such that the bkc_logging function is called whenever the execution enters any one of the four functions above (i.e., `user_login()` , `user_logout()` , `file_access()` , `file_permission_set()`).

There are two ways to store arbitrary data in Ethereum transactions: 1. Use data field of a transaction 2. Use recipient (to) address. Ethereum addresses are 20 bytes (40 hex characters). So that, we can store up to 20 bytes of arbitrary data (e.g. UTF-8 string) in the recipient address field after we encode our data correctly.

Exercise 2: Blockchain logging using data field [grade 30%]

Implement the function of `bkc_logging(log)` using the `data` field in the `sendTransaction` method. You may want to show/check the log string you logged into the blockchain using `getTransaction` method.

Exercise 3: Blockchain logging using recipient address [grade 20%]

Implement the function of `bkc_logging(log)` that encode logged data in the recipient address (i.e., `to` address) in the `sendTransaction` method. You may want to show/check the log string you logged into the blockchain using `getTransaction` method.

Hint: 1) You may need to use `keccak256` hash algorithm to generate an Ethereum address out of the logged data. 2) Alternatively, you can manually generate a blockchain address of a fixed length. When the logged data is fewer than 20 bytes, you may need to pad the data to 20 bytes. Or when the logged data is longer than 20 bytes, you can separate it into multiple addresses and send multiple transactions to those addresses.

Exercise 4: Transaction fees [grade 20%]

Call user_Login & user_Logout methods and log your data into the blockchain using both data and recipient address fields. Report the transaction fee for each of them.

Exercise 5: Reducing transaction fees 1 [grade 10%]

With the increasing number of transactions, we would have to pay more transaction fees. One way to reduce the fees would be to batch our logs. You can do that by concatenating your logs into a single log. Simulate multiple (5-10) operations (user_Login, user_Logout, etc.) and store your logs (by concatenating them) in a single variable, such as following:

finalLog = logUpdate1 || logUpdate2 || ... || logUpdate5

Then, send that to the blockchain. This way, you'll be sending a single transaction to the blockchain instead of multiple ones like in exercises 2 and 3.

Report the transaction fee.

You can use data field for this task.

Exercise 6: Reducing transaction fees 2 [grade 10%]

The above approach still might result in high transaction fees since our log data keeps getting larger. We can keep its size constant by hashing it. Again, simulate multiple (5-10) operations but this time hash your concatenated logs, such as following:

H = hash function

log1 = H(logUpdate1)

log2 = H(log1||logUpdate2)

...

log5 = H(log4||logUpdate5)

You can hash your logs using the hash methods in `crypto` module of Node.js. Send your final hash value to the blockchain. Report the transaction fee.

You can use data field for this task.

What to submit

1. Source code
2. A report describing your design in detail, implementation issues you encounter, and how you overcome them.
3. Screenshot that shows your code is running successfully.