CITS 3004
Cybersecurity

THE UNIVERSITY OF WESTERN AUSTRALIA

Lab 4: Hashing and Blockchain

Task 1: Hashing

1A) GENERATING MESSAGE DIGEST AND MAC

In this task, we will play with various one-way hash algorithms. You can use the following `openssl dgst` command to generate the hash value for a file. To see the manuals, you can type `man openssl` and `man dgst`.

```
% openssl dgst dgsttype filename
```

Note: you can create a binary output using -binary command.

Please replace the `dgsttype` with a specific one-way hash algorithm, such as `-md5`, `-sha1`, `-sha256`, etc. In this task, you should try at least 3 different algorithms, and discuss your observations with lab instructor/peers. You can find the supported one-way hash algorithms by typing "`man openssl`".

1B) KEYED HASH AND HMAC

In this task, we would like to generate a keyed hash (i.e. `MAC`) for a file. We can use the `-hmac` option (this option is currently undocumented, but it is supported by `openssl`). The following example generates a keyed hash for a file using the HMAC-MD5 algorithm. The string following the `-hmac` option is the key

Complete the following tasks.
(1) Generate a keyed hash using `HMAC-MD5`, `HMAC-SHA256`, and `HMAC-SHA1` for any file that you choose (create your own).
(2)Try several keys with different length.

CITS 3004 Cybersecurity

THE UNIVERSITY OF WESTERN AUSTRALIA

1C) THE RANDOMNESS OF ONE-WAY HASH

To understand the properties of one-way hash functions, we would like to do the following exercise for `MD5` and `SHA256`:

1. Hash the text "this is a hash message" and generate the hash value H1 using a specific hash algorithm.
2. Hash the text "this ir a hash message" and generate the hash value H2 using a specific hash algorithm. Notice one character has been changed.
3. Observe whether H1 and H2 (both should be saved onto files) are similar or not. Use a program Comparer.c to count how many bits are the same between H1 and H2.

Note: the bytesize for comparer is the number of characters to compare in the file you store the hash (i.e. each hex value (e.g. FF) will occupy 2 bytes as 2 characters when stored in a file).

You can hash the text directly using the following command.

```
# echo -n "your message" | openssl dgst -[hash function]
```

Answer Q1 ~ Q5 on LMS

1D) ONE-WAY PROPERTY VERSUS COLLISION-FREE PROPERTY

In this task, we will investigate the difference between hash function's two properties: one-way property versus collision-free property. We will use the brute-force method to see how long it takes to break each of these properties. Instead of using `openssl`'s command-line tools, you are required to write your own C programs to invoke the message digest functions in `openssl`'s crypto library. A sample code can be found from `http://www.openssl.org/docs/crypto/EVP_DigestInit.html`. Please get familiar with this sample code.

Since most of the hash functions are quite strong against the brute-force attack on those two properties, it will take us years to break them using the brute-force method. To make the task feasible, we reduce the length of the hash value to 24 bits. We can use any one-way hash function, but we only use the first 24 bits of the hash value in this task. Namely, we are using a modified one-way hash function. Please design an experiment to find out the following:

1. How many trials it will take you to break the one-way property using the brute-force method? You should repeat your experiment for multiple times, and record your average number of trials.
2. How many trials it will take you to break the collision-free property using the brute-force method? Similarly, you record the average.
3. Based on your observation, which property is easier to break using the brute-force method? You are provided with a C program `OneWayBreaker.c`, which provides a solution for breaking one way property with a brute force method. You are to complete the second part, which is to investigate the collision-free property, where a partially completed C program `CollisionFreeBreaker.c` is also provided.

The makefile to create `OneWayBreaker.c` and partially complete C program `CollisionFreeBreaker.c` is provided.

Answer Q6 ~ Q7 on LMS

CITS 3004 Cybersecurity

Task 2: Blockchain

Written by Jin Hong using online resources

2A) SETUP

We will implement a Blockchain in Python3. First, we need to install Python 3.6+, which is not installed on our SEED Ubuntu (it has 3.5). Make sure you have a backup image that you can always get a fresh start.

```
# sudo add-apt-repository ppa:deadsnakes/ppa
# sudo apt-get update
# sudo apt-get install python3.6
```

Install pip.

```
# curl https://bootstrap.pypa.io/get-pip.py | sudo python3.6
```

Ensure your pip is running for Python3.6, not Python3.5, by checking the version.

```
# pip -V
pip 10.0.1 from /usr/local/lib/python3.6/dist-packages/pip (python 3.6)
```

Install the requests package, which will be used later for handling HTTP requests.

```
# sudo pip install requests
```

Now, install Flask and other dependent libraries

```
# sudo pip install Flask==0.12.2 requests==2.18.4 werkzeug==0.16.1
```

Also, install Postman from the main website. This will be used to test the Blockchain via web requests. Remember, the Linux distro is x86 (not x64). You have to sign up to use (but it's free).

```
https://www.getpostman.com/download
```

You can also choose to use your own HTTP client.

2B) BUILDING A BLOCKCHAIN

The skeleton code `blockchain.py` is provided for you. This is currently incomplete and will not work as of its current state. We will expand this code to make it a functioning Blockchain code. The code will manage the block chains, and handle transactions via helper methods for adding new blocks to the chain. Each Block has an *index*, a *timestamp* (in Unix time), a *list of transactions*, a *proof* (more on that later), and the *hash of the previous Block*.

1) Adding a new transaction

Firstly, we define a method for adding new transactions. We will add one transaction at a time. This method will take three input parameters: sender, recipient and amount. The `new_transaction()` method is provided for you that adds a new transaction. Once a transaction has been added, the method returns the *index* of the block, specifying the next one to be mined (i.e., where the next transaction will be stored). Complete the code for `new_transaction()` method. It will append a dictionary into the `current_transactions` list. The dictionary contains:

Key	Value
"sender"	Sender
"recipient"	Recipient
"amount"	Amount

The values are all input to the method so you just have to use them.

2) Creating a new Block

The initial block without the predecessor, which is called the *genesis* block, is created when the Blockchain is instantiated. Just like every other blocks, the genesis block also needs to have a *"proof"*. Proof is simply specifying the work done to mine the block. This will be discussed later in the mining section. Three methods work to create a new block: `new_block()`, `new_transaction()` and `hash()`.

In the `new_block()` method, create a new block, which is a dictionary containing the following. Key values are all type string.

Key	Value
"index"	Length of the current chain + 1
"timestamp"	time()
"transactions"	Current transactions that the block is storing
"proof"	`proof`
"previous_hash"	Either the `previous_hash` or hash value of the last block on the chain (use self.hash)

3) Proof of Work

To create or *mine* new blocks for the Blockchain, a Proof of Work algorithm (PoW) is required. In simple terms, PoW is a number that solves a difficult mathematical problem. Normally, solving this is difficult, but verifying the solution is quick by anyone in the network. This is done using hashing algorithms. For example, we would like to find a hash given value `x`, such that the last digit of the hash(`x * y`) is 0 (e.g., `hash(x*y)=ac23dc...0`). Let's assume the value `x=5`, and we are using the SHA256 hash algorithm. You can easily write this in Python.


```
from hashlib import sha256

x = 5
y = 0  # start from 0

while sha256(str(x*y).encode()).hexdigest()[-1] != "0":
    y += 1

print('solution value is {}'.format(y))
```

The PoW algorithm used by Bitcoin is called *Hashcash*, which is essentially the same idea as above, but much more difficult (e.g., instead of one 0 at the end, has to have four 0's at the beginning of the hash). Miners use this algorithm in order to create a block, and when they do, the miners are rewarded with a coin in a transaction. The difficulty is determined by the number of characters searched for in a string. The network is able to *easily* verify their solution.

4) Implementing basic Proof of Work

Let's implement a similar algorithm for our blockchain. Our rule will be to find a number *p* that when hashed with the previous block's solution, a hash with 4 leading 0s is produced. The methods `proof_of_work(self, last_proof)` and `valid_proof(last_proof, proof)` are provided for you to complete. The difficulty can easily be adjust by defining the length of the leading zeroes. For our purpose, 4 is enough. You should explore how much extra work is needed by adjusting the number of leading zeroes.

Answer Q8 ~ Q10 on LMS

2C) USING THE BLOCKCHAIN CLASS

At this point, we are ready to test our blockchain. The Python Flask Framework enables us to interact with the blockchain class over the web using HTTP requests. The necessary codes are already provided for you, we just need to get familiar with how to interact with the blockchain class over the HTTP protocol.

The request for a transaction will look like below (i.e., the user sends to the server (blockchain class)):

```
{
 "sender": "my address",
 "recipient": "receiver's address",
 "amount": 1234
}
```

CITS 3004
Cybersecurity

We will mainly use Postman, but you can also use cURL or other HTTP clients to interact, if you wish.

To start, we instantiate the blockchain server (make sure the python call will use python3.6).

```
$ python blockchain.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

First, let's try mining a block. This is done by making a GET request:

```
http://localhost:5000/mine
```

Select the GET request, and type the command to execute. You should be able to receive the reply from the server similarly as shown in Figure 1. This is the first block (one after the genesis block) with the transaction. You can validate the hash by using the proof.

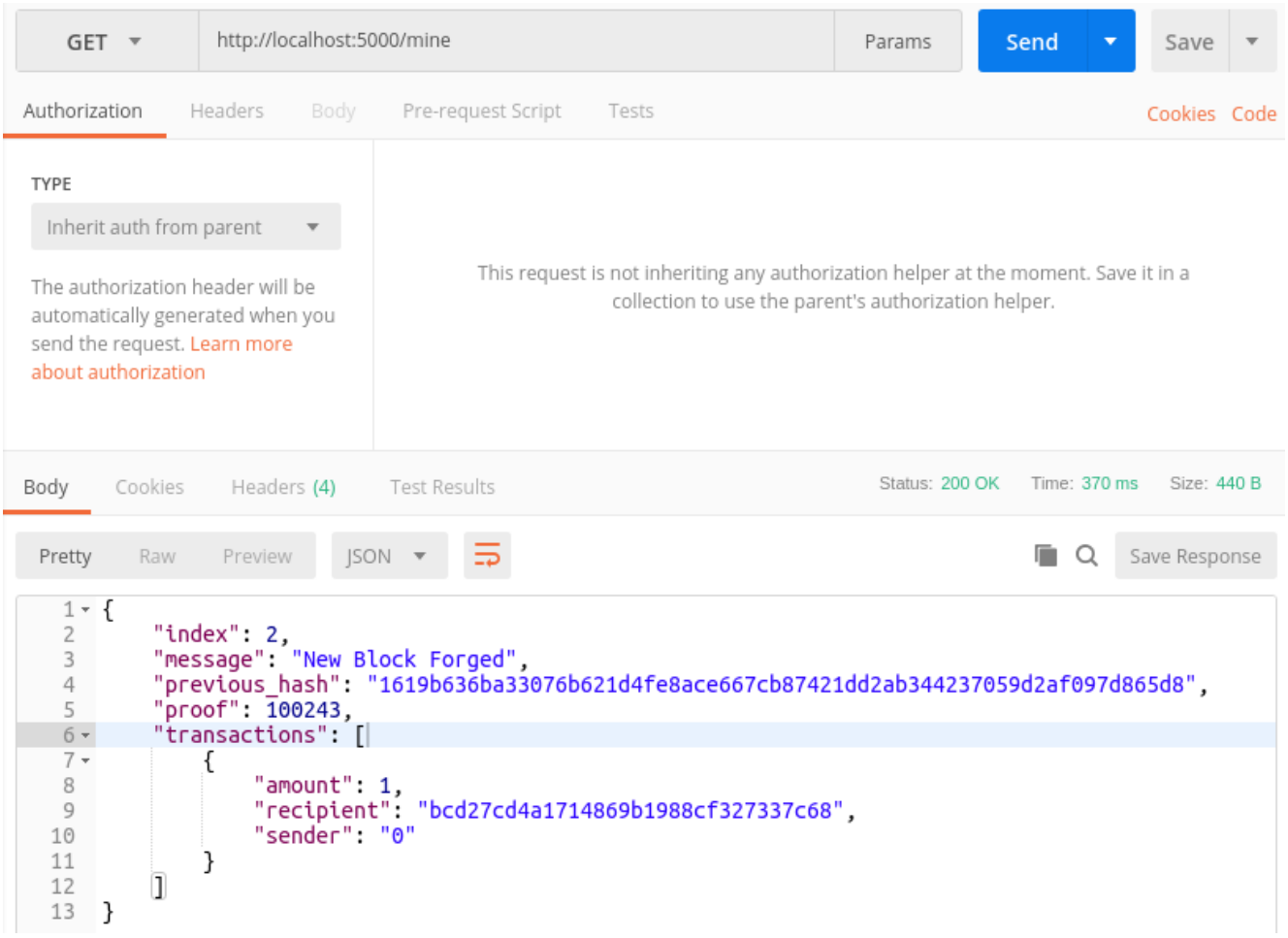

Figure 1. A GET request from the blockchain class

Second, let's create a new transaction. This is done by making a POST request:

```
http://localhost:5000/transactions/new
```

You also have to specify the body, which should contain the transaction structure. For example:

```
{
 "sender": "d4ee26eee15148ee92c6cd394edd974e",
 "recipient": "someone-other-address",
 "amount": 5
}
```

Ensure that you input a *raw* JSON into the body field. If you get an error, you may have to restart the server.

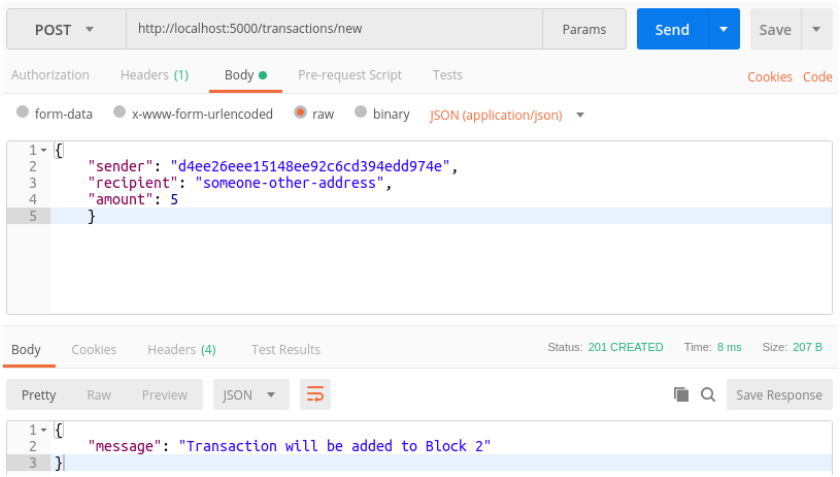

Figure 2. A POST request from the blockchain class with a new transaction

Figure 2 shows the result of creating a new transaction. For this, I restarted my server, and mined two blocks and posted one, to give 3 blocks in total. Let's inspect the full chain by requesting the chain query (Figure 3):

```
http://localhost:5000/chain
```

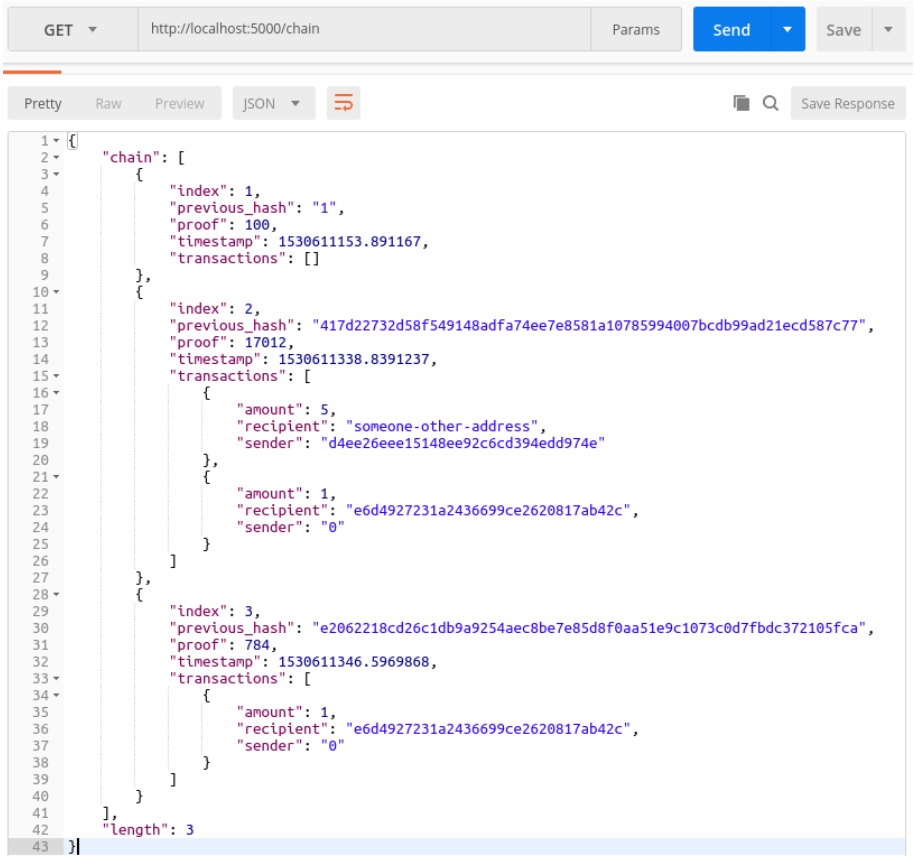

Figure 3. A POST request from the blockchain class with a new transaction

2D) CONSENSUS

So far, we have completed a basic Blockchain, which can create new transactions and mine new blocks. In this task, we will ensure that the Blockchain will be used in a decentralised format. One key aspect of this is to ensure that all the blockchains in the network are the same. This is called the *Consensus* problem.

1) Registering new Nodes

Before we can implement a Consensus Algorithm, we need a way to let a node know about neighbouring nodes on the network. Each node on our network should keep a registry of other nodes on the network. Thus, we'll need some more endpoints.

1. /nodes/register to accept a list of new nodes in the form of URLs.
2. /nodes/resolve to implement our Consensus Algorithm, which resolves any conflicts—to ensure a node has the correct chain.

These endpoints are already implemented for you, but you should inspect the structure of the blockchain class now to get a good overview how multiple nodes will be handled.

2) Implementing the Consensus Algorithm

A conflict can occur in a Blockchain network when one node has a different chain to another node. A simple approach to resolve this problem is specifying rules which chain is the authoritative one. Simply, *the longest valid chain is authoritative* for our exercise (and this is true for typical blockchains).

Two methods, `valid_chain()` and `resolve_conflicts()`, are provided. The `valid_chain()` method ensures that the existing chain is valid by checking each block of its hash and the proof. The `resolve_conflicts()` method will check all the neighbouring nodes, which *downloads* their chains and verifies them using the above method. If we find a valid chain that has a greater length that the existing one, we will replace it.

To test, start another blockchain server on a different port.

```
# python3.6 blockchain.py -p 5001
```

Alternatively, you can manually create a copy of the blockchain.py and rename it (e.g., `blockchain2.py`). Edit the port details of the second blockchain code (e.g., use 5001). Start new terminals and start both `blockchain.py` and `blockchain2.py` on each terminal.

We just started two blockchain nodes, which can be used to test the consensus resolution we just implemented. First, register the second node to the first one (and vice versa) (see Figure 4). Mine a block or two on node 1 (port 5000). Then mine some new blocks on node 2 (port 5001), and make sure the chain is longer than node 1. Finally, call `GET /nodes/resolve` on node 1 and see the output. You can observe that the chain is now replaced by the Consensus algorithm (see Figure 5).

Now, you should be able to create a network of blockchains with your peers!

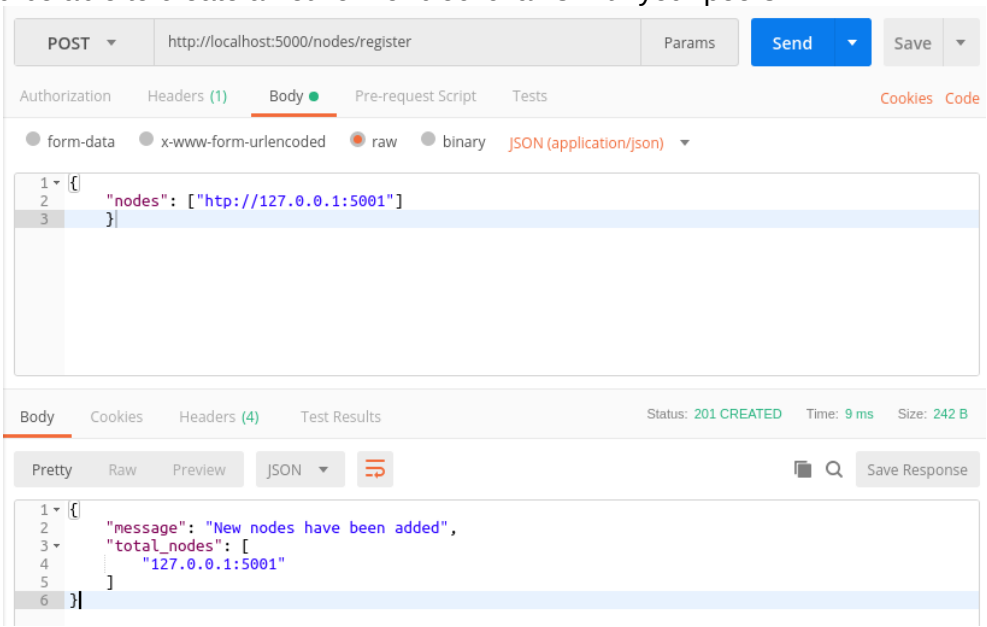

Figure 4. Registering a new Node

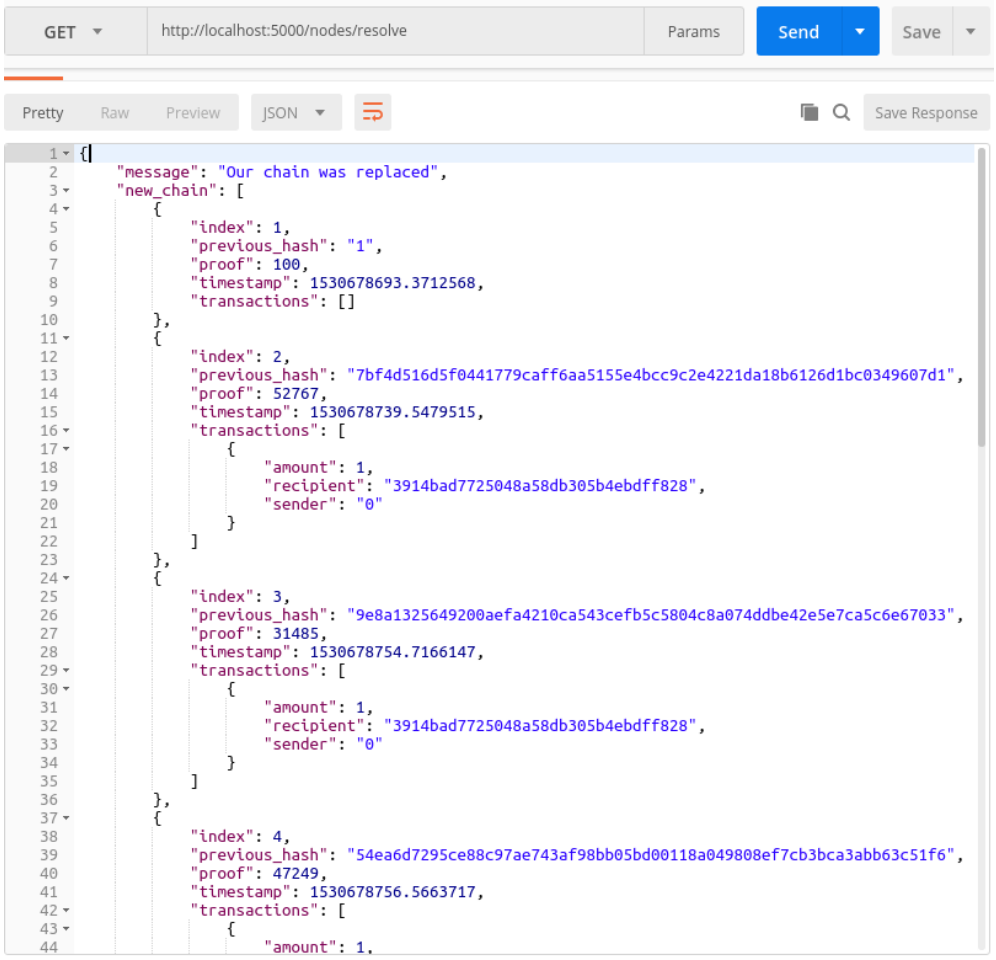

Figure 5. Checking the Consensus Algorithm working

Answer Q11 and Q12 on LMS