**Blockchain Realization Lab**

Date 12/27/2019

Name: Tiancheng Xie



**Department of Computer Science and Electrical Engineering**

**DOCUMENT CONTENTS**

1. Author
2. Objective
3. Configuration
4. UML Graph
5. Code Snippet
6. Deployment
7. References

**AUTHOR**

Author: Tiancheng Xie

Guide by Dr. Choi.

**OBJECTIVE**

This document is mainly to report how to realize the prototype of the blockchain on the python platform. It demonstrates how to build a blockchain framework, and explain the functionality in the code.

Introduction:

Python plays an important role on developing OOP. In a single block of blockchain, there are many attributions, like index, timestamp, transaction information, proof of work, and previous hash. Also, there are many methods for implementing the operation of those property. In this lab, a class called blockchain was built, which can realize the addition of transactions, the interaction of different IP nodes, and the consensus mechanism of PoW, and simple mining.

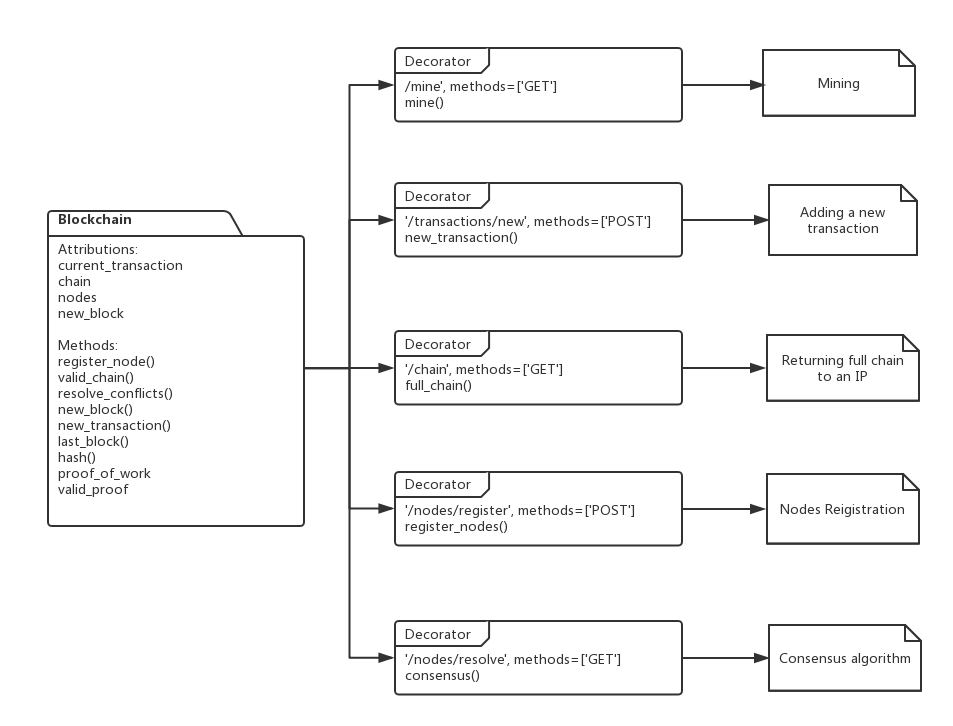
**Configuration**

For executing the lab given in the lab, advanced version of **Python 3.6** is used and the code is built in **PYTHON Pycharm IDLE**.

In addition, **Postman** is used to testify the lightweight web application framework which is also built in this lab.

Main library used in the lab:

import hashlib  
from time import time # For timestamp  
import json  
from urllib.parse import urlparse  
from uuid import uuid4  
  
import requests  
from flask import Flask, jsonify, request  
  
from argparse import ArgumentParser

**UML Graph**

**Code Snippet**

Constructor:

def \_\_init\_\_(self):  
 self.chain = []  
 self.current\_transactions = []  
 self.nodes = set()  
 # Build the No.1 block  
 self.new\_block(proof=100, previous\_hash=1)

The first block would be created once the blockchain was set up. The local attribution nodes is used to store different IP nodes for different users.

Node Registration:

def register\_node(self, address: str) -> None:  
 # http://127.0.0.1:5001  
 parsed\_url = urlparse(address)  
 self.nodes.add(parsed\_url.netloc)

Add a new node to the list of nodes;

:param address: Address of node.

Valid Chain:

def valid\_chain(self, chain) -> bool:  
  
 last\_block = chain[0]  
 current\_index = 1  
  
 while current\_index < len(chain):  
 block = chain[current\_index]  
  
 if block['previous\_hash'] != self.hash(last\_block):  
 return False  
  
 if not self.valid\_proof(last\_block['proof'], block['proof']):  
 return False  
  
 last\_block = block  
 current\_index += 1  
  
 return True

Determine if a given blockchain is valid

:param chain: A blockchain

:return: True if valid, False if not

Consensus:

def resolve\_conflicts(self) -> bool:  
  
 neighbours = self.nodes  
  
 max\_length = len(self.chain)  
 new\_chain = None  
  
 for node in neighbours:  
 response = requests.get(f'http://{node}/chain')  
 if response.status\_code == 200:  
 length = response.json()['length']  
 chain = response.json()['chain']  
  
 if length > max\_length and self.valid\_chain(chain):  
 max\_length = length  
 new\_chain = chain  
  
 if new\_chain:  
 self.chain = new\_chain  
 return True  
  
 return False

Consensus algorithm solves conflicts: using the longest chain in the network.

:return: If the chain was replaced return True, otherwise return False.

Create New Block:

def new\_block(self, proof, previous\_hash=None):  
 block = {  
 'index': len(self.chain) + 1,  
 'timestamp': time(), # Return the time  
 'transactions': self.current\_transactions,  
 'proof': proof,  
 'previous\_hash': previous\_hash or self.hash(self.chain[-1])  
 # Previous hash can be the delivered hash or the hash calculated by the hash function  
 }  
  
 self.current\_transactions = [] # Clear the current transaction  
 self.chain.append(block) # Append the new block  
  
 return block

:param proof: The proof given by the Proof of Work Algorithm

:param previous\_hash: Hash of previous Block

:return: New Block

Add new transaction:

def new\_transaction(self, sender, recipient, amount) -> int:  
 # Current\_transaction is an array storing the block  
 self.current\_transactions.append(  
 {  
 'sender': sender,  
 'recipient': recipient,  
 'amount': amount  
 }  
 )  
  
 return self.last\_block['index'] + 1

Generate new transaction information which would be added into the next mined block

:param sender: Address of the Sender

:param recipient: Address of the Recipient

:param amount: Amount

:return: The index of the Block that will hold this transaction

**Proof of Work:**

def proof\_of\_work(self, last\_proof: int) -> int:  
 proof = 0  
 while self.valid\_proof(last\_proof, proof) is False:  
 proof += 1  
  
 return proof

def valid\_proof(self, last\_proof: int, proof: int) -> bool:  
 guess = f'{last\_proof}{proof}'.encode()  
 guess\_hash = hashlib.sha256(guess).hexdigest()  
  
 # Take the first 4 hash number  
 return guess\_hash[0:4] == "0000"

Simple Testify:

Find a **proof** which made hash(**previous\_proof**) start with “0000”; It can be sixteen ‘0’, considering it takes a lot of time, ‘0000’ is fit.

**DEPLOYMENT**

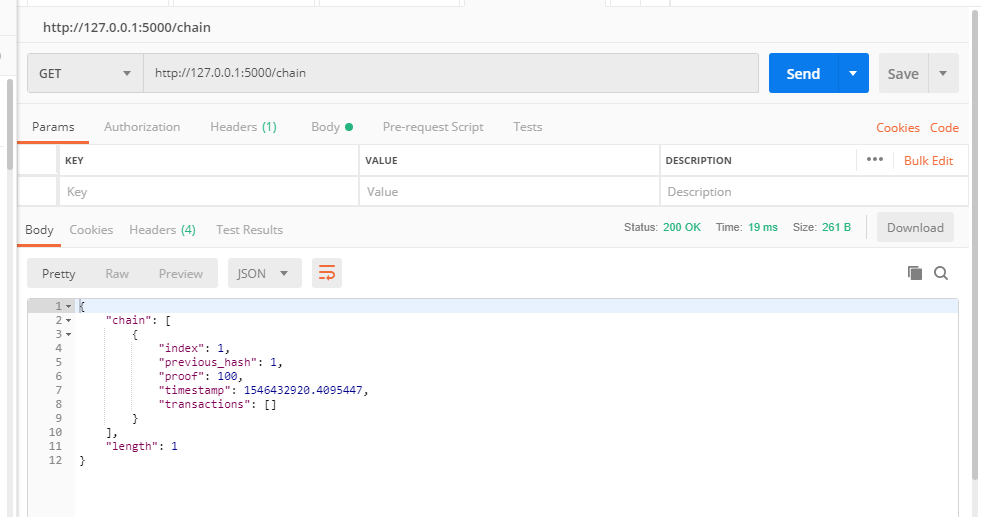
**Run the blockchain.py file with 5000 and 5001 two different nodes.**

C:\Users\a1908\AppData\Local\Temp\1546432981(1).png

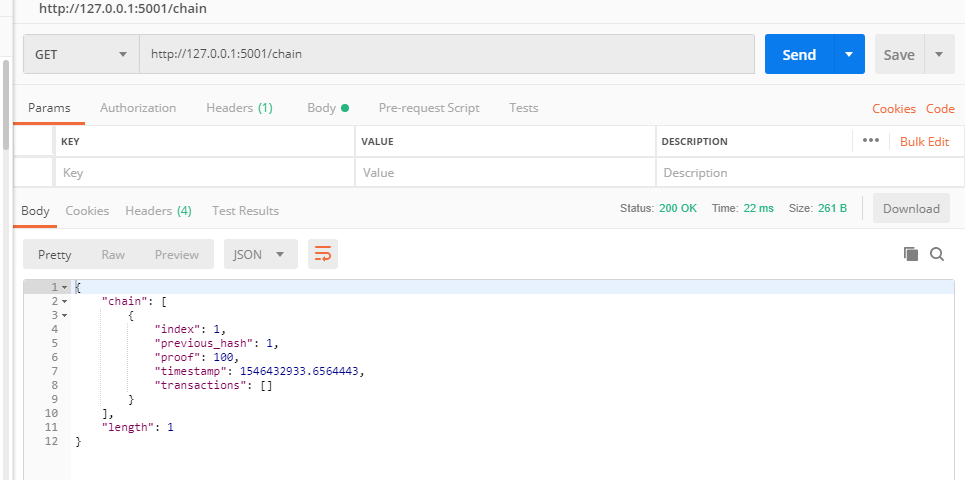


**Run Postman**

**Chain of 5000:**

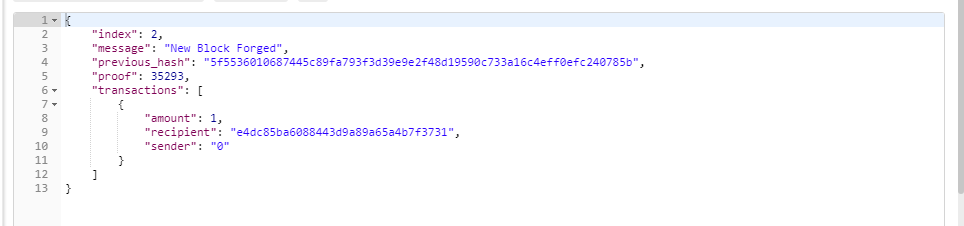


**Chain of 5001:**



**Mine (5000 ):**

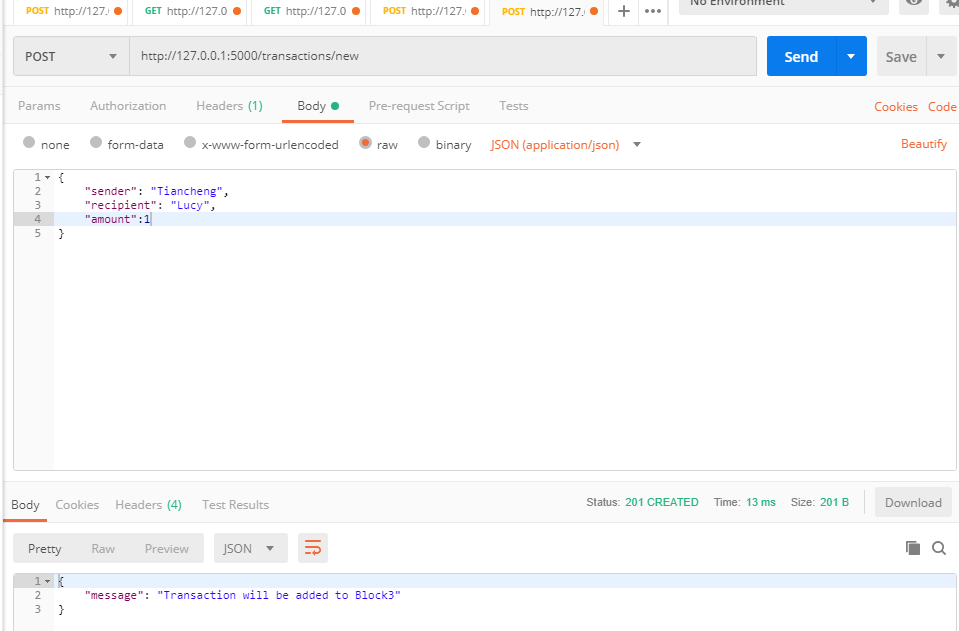
**New block:**



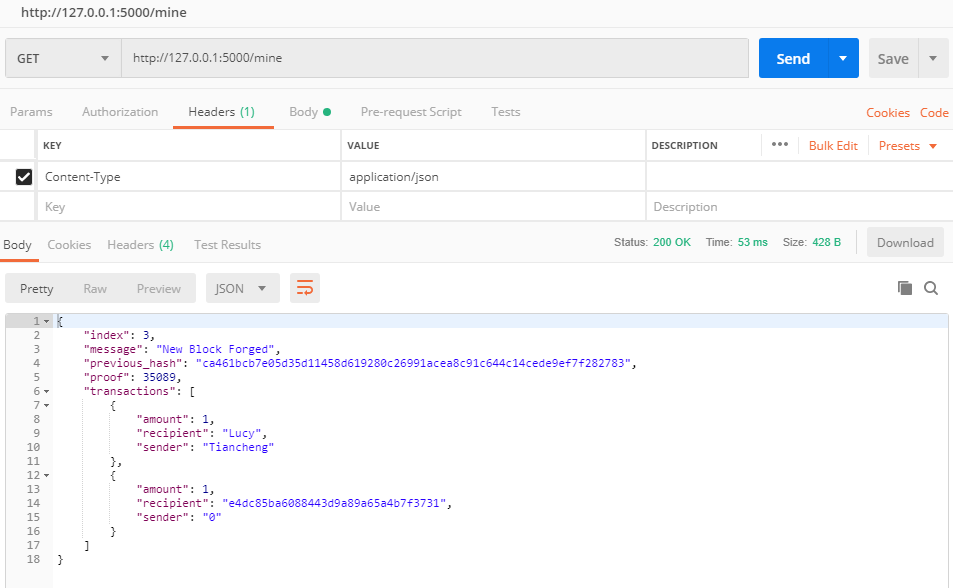
**Full chain:**



**New transaction (5000 )**

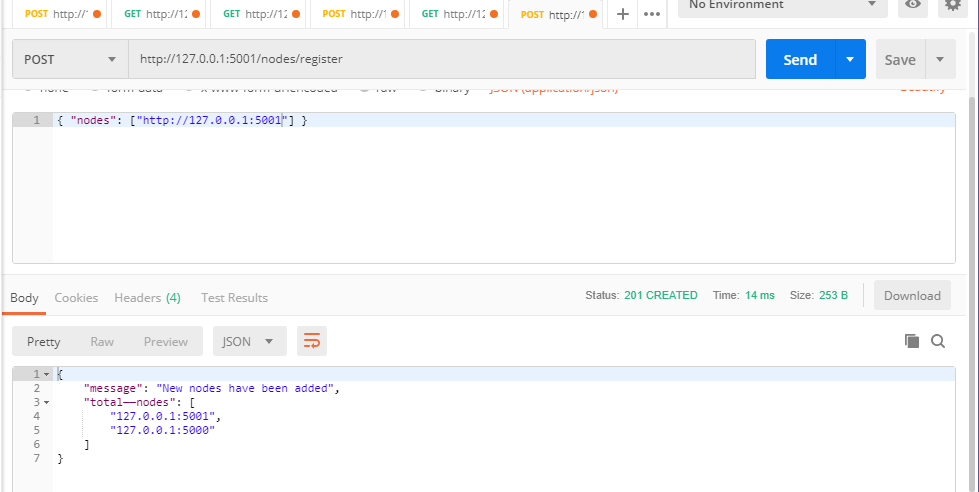


**After mining, transaction information saved(5000 node):**

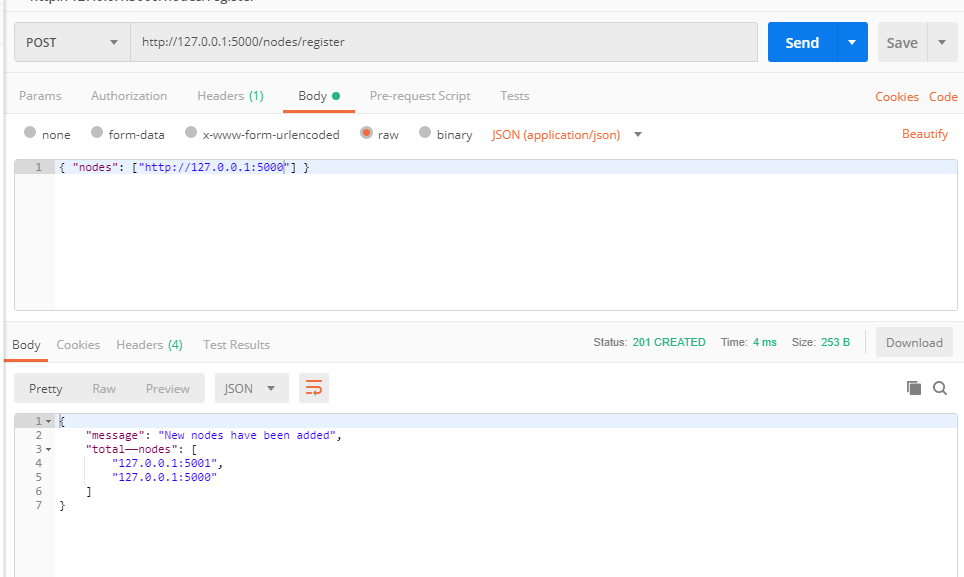


**Node registration:**

**(5001):**

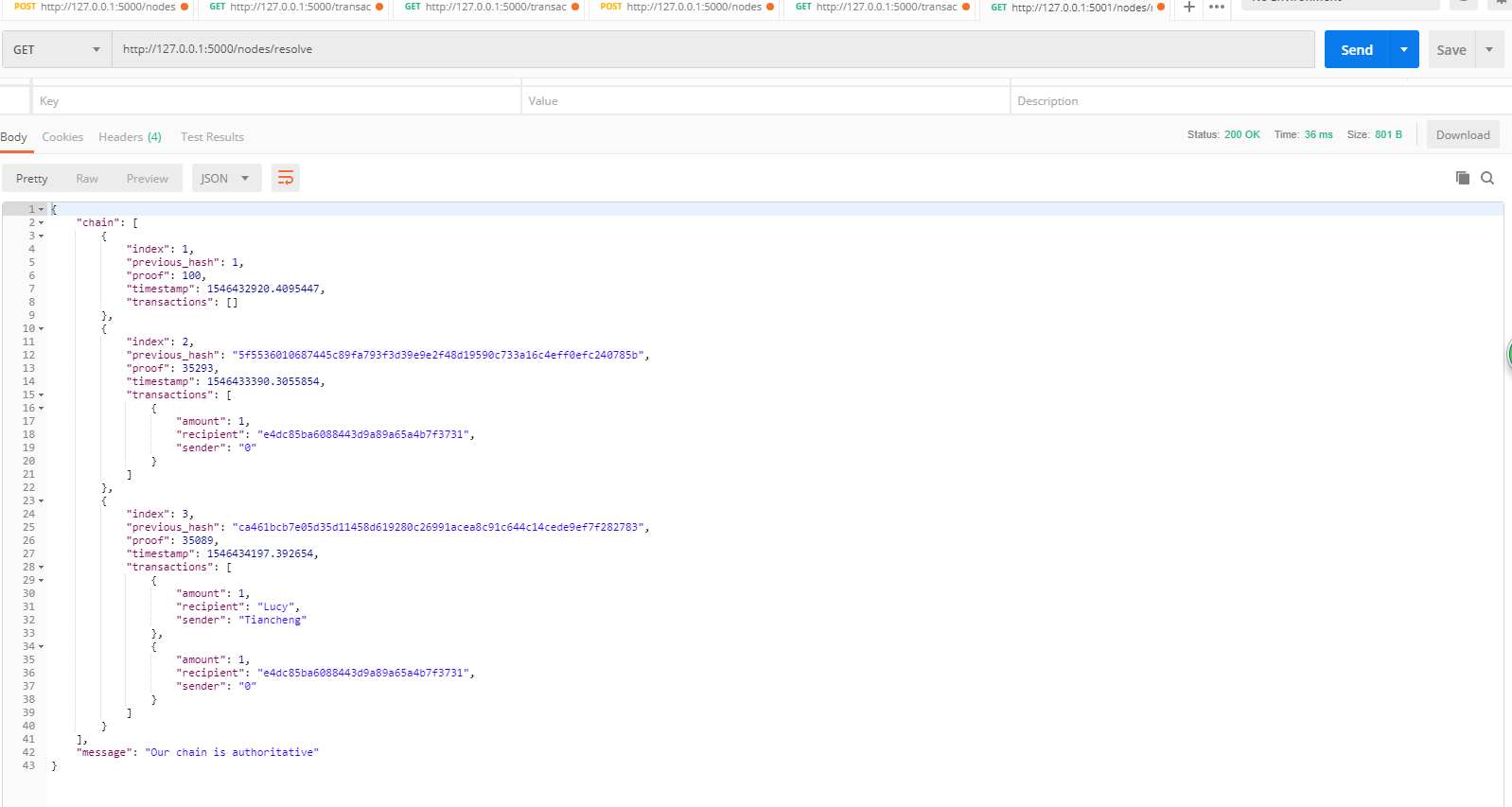


**(5000):**

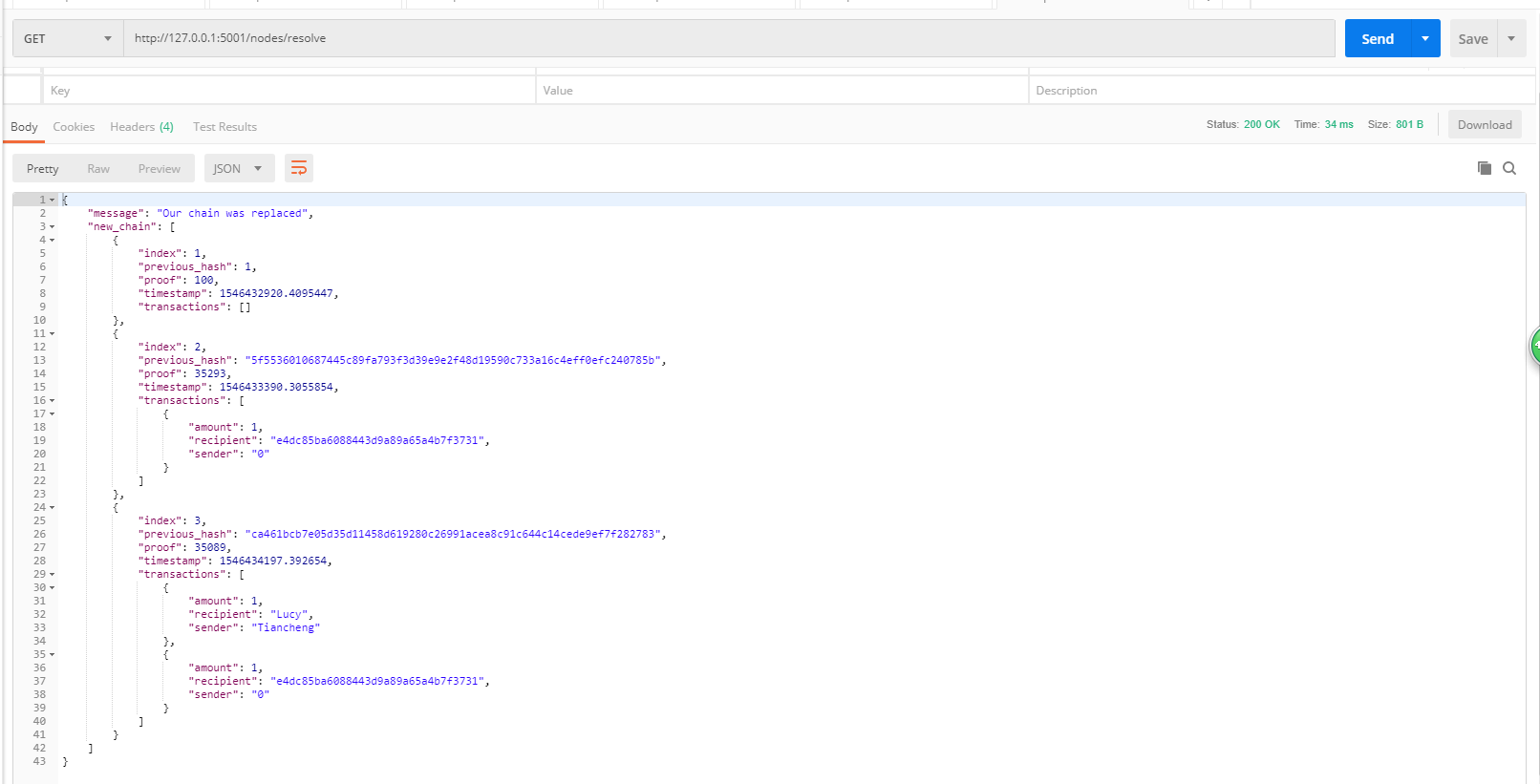


**Consensus test:**

**(5000) main chain : longest**



**(5001) was replaced by the main chain**



**REFERENCES**

**[1]** [**https://www.jianshu.com/p/8c8cedc84220**](https://www.jianshu.com/p/8c8cedc84220)

**[2] Dash**

**[3]** [**http://flask.pocoo.org/**](http://flask.pocoo.org/)