HW2 Q2 Documentation

Jing Jiang & Zheng Liu

a)

```python
class Blockchain:
    def __init__(self):
        self.current_tx = []
        self.chain = []
        self.nodes = set()

        self.new_genesis_block()

    def new_genesis_block(self):
        """
        """
        first_hash = "0000000000000000000000000000000000000000000000000000000000000000"
        first_time = time()

        block = {
            'index': len(self.chain) + 1,
            'timestamp': first_time,
            'transactions': self.current_tx,
            'nonce': self.proof_of_work(first_hash, first_time),
            'prev_hash': first_hash,
        }
        self.current_tx = []

        self.chain.append(block)
        return block
```

In the screenshot above, I created a Blockchain class. Inside the constructor, 3 properties are defined. The *current_tx* is the transactions pending to be added to the next block. The chain stores the data structure of the blockchain. And nodes is related to the consensus algorithms. Then, I initialize it with a method *new_genesis_block*. Unlike the tutorial, I didn't use the *new_block* method, because the author of the tutorial assigned a nonce instead of mining to obtain one, which will cause the initial block having a unverified hash.

```python
    def new_block(self, nonce, prev_hash, timestamp):
        """
        """
        block = {
            'index': len(self.chain) + 1,
            'timestamp': timestamp,
            'transactions': self.current_tx,
            'nonce': nonce or self.proof_of_work(prev_hash, timestamp),
            'prev_hash': prev_hash or self.hash(self.chain[-1]),
        }
        self.current_tx = []

        self.chain.append(block)
        return block

    def new_transaction(self, recipient, sender, amount):
        """
        """
        self.current_tx.append({
            'recipient': recipient,
            'sender': sender,
            'amount': amount,
        })

        return self.last_block['index'] + 1
```

In the *new_block* method, a dictionary called block is defined and its nonce is mined to make the block verified. The *current_tx* will be reset, meaning that all transactions are added to the new block. And then the block is appended to the chain.

In the *new_transaction* method, a dictionary with keys: recipient, sender and amount is appended to the transaction list. The method will return the index of the block which this transaction will be added to.

```python
    @property
    def last_block(self):
        if len(self.chain)>0:
            return self.chain[-1]
        else:
            return None

    @staticmethod
    def hash(block):
        """
        """
        block_string = json.dumps(block, sort_keys=True).encode()
        return hashlib.sha256(block_string).hexdigest()

    def proof_of_work(self, last_hash, timestamp):
        """
        """
        if self.last_block is not None:
            current_index = self.last_block['index'] + 1
        else:
            current_index = 1

        nonce = 0
        while self.valid_proof(current_index, timestamp, self.current_tx, nonce, last_hash) is False:
            nonce += 1

        return nonce

    @staticmethod
    def valid_proof(index, timestamp, transactions, nonce, last_hash):
        """
        """
        block_to_verify = {
            'index': index,
            'timestamp': timestamp,
            'transactions': transactions,
            'nonce': nonce,
            'prev_hash': last_hash,
        }

        guess_string = json.dumps(block_to_verify, sort_keys=True).encode()
        guess_hash = hashlib.sha256(guess_string).hexdigest()
        return guess_hash[:4] == "0000"
```

As shown above, the *last_block* returns the last block of the chain if the chain is not empty. The hash method uses the json library and the hashlib library to provide the hash code of a block.

The *proof_of_work* method and the *valid_proof* method are related to the verification of a block. The while loop in the *proof_of_work* method tries to pass a integer as a nonce to the *valid_proof* method to see if this nonce can make the block verified. If *valid_proof* returns true, the loop ends and the nonce is returned.

Inside the *valid_proof* method, every time a dictionary called *block_to_verify* is created. This is basically a block pending to be verified. Every time this block is JSON serialized and hashed. If the hash code has 4 preceding zeros, this method returns true, indicating that this block is verified.

b)

In the example, the author simplified the hash puzzle of the proof of work greatly by only passing the nonce of the previous block to the method to calculate the hash. He concatenated the previous nonce (what he called proof) with the current nonce and hashed them until the hash got four preceding zeros. However, what we discussed in class is that the hash of the block should be verified with 4 zeros, not a concatenated string. So I implemented my own method to replace the simplified method, as shown and explain in part a.

c)

```
@app.route('/mine', methods=['GET'])
def mine():
    timestp = time()
    blockchain.new_transaction(
        recipient=node_id,
        sender="coinbase",
        amount=100,
    )
    last_block = blockchain.last_block

    prev_hash = blockchain.hash(last_block)

    nonce = blockchain.proof_of_work(prev_hash, timestp)

    block = blockchain.new_block(nonce, prev_hash, timestp)

    response = {
        'message': "New Block Forged",
        'index': block['index'],
        'transactions': block['transactions'],
        'nonce': block['nonce'],
        'prev_hash': block['prev_hash'],
        'block_hash': blockchain.hash(block)
    }

    return jsonify(response), 200
```

The first method(endpoint) is */mine*. This method allows me to mine the new block. Every time this method is called, a block reward transaction is added. The sender of this transaction is *"coinbase"*, and the receiver is whoever who mined the block. Then the nonce is find and a new block is created. A response showing the information of this block is returned.

```python
@app.route('/transactions/new', methods=['POST'])
def new_transaction():
    values = request.get_json()

    required = ['recipient', 'sender', 'amount']

    if not all(k in values for k in required):
        return 'Not enough values', 400

    ind = blockchain.new_transaction(values['recipient'], values['sender'], values['amount'])

    response = {'message': f'The transaction will be added to Block {ind}'}

    return jsonify(response), 201
```

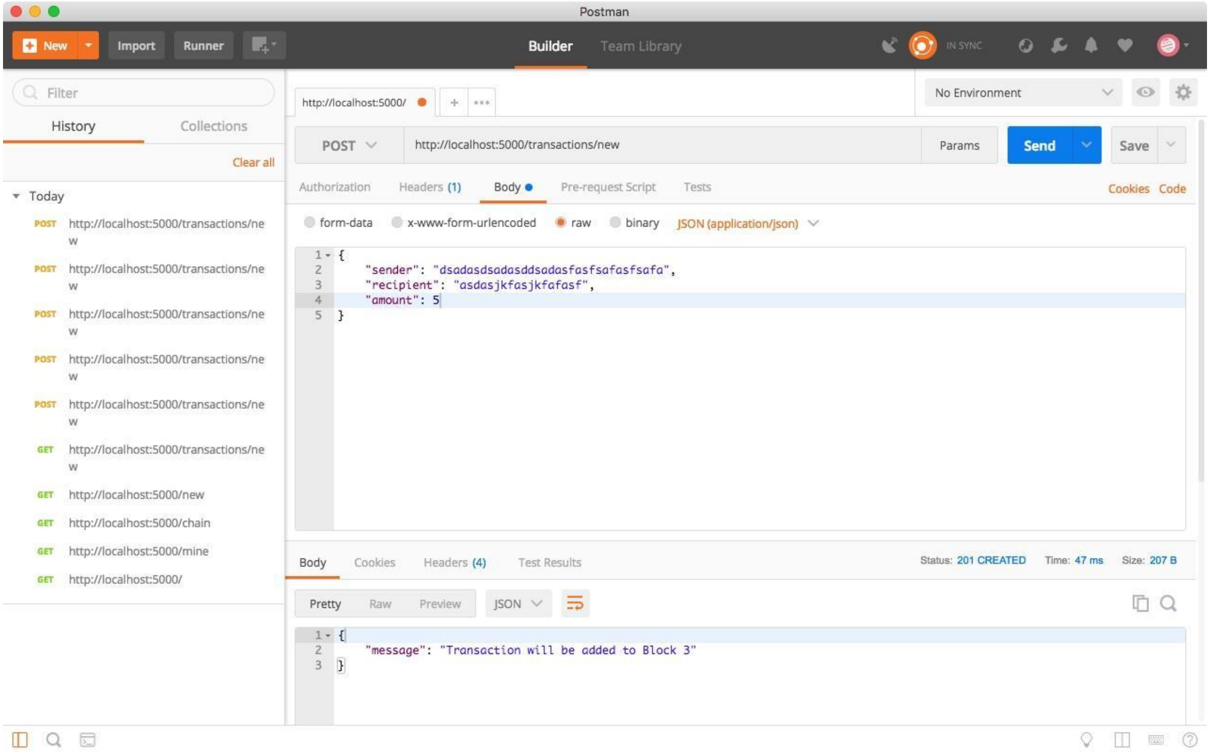
The endpoint above is */transactions/new*. This method creates new transactions. Unlike */mine*, this method is a POST request, in which information are sent from local to server. The keywords *'recipient', 'sender'* and *'amount'* are specified and a response is returned if this works correctly.

```python
@app.route('/chain', methods=['GET'])
def full_chain():
    response = {
        'BlockChain': blockchain.chain,
        'length of the chain': len(blockchain.chain),
    }

    return jsonify(response), 200
```

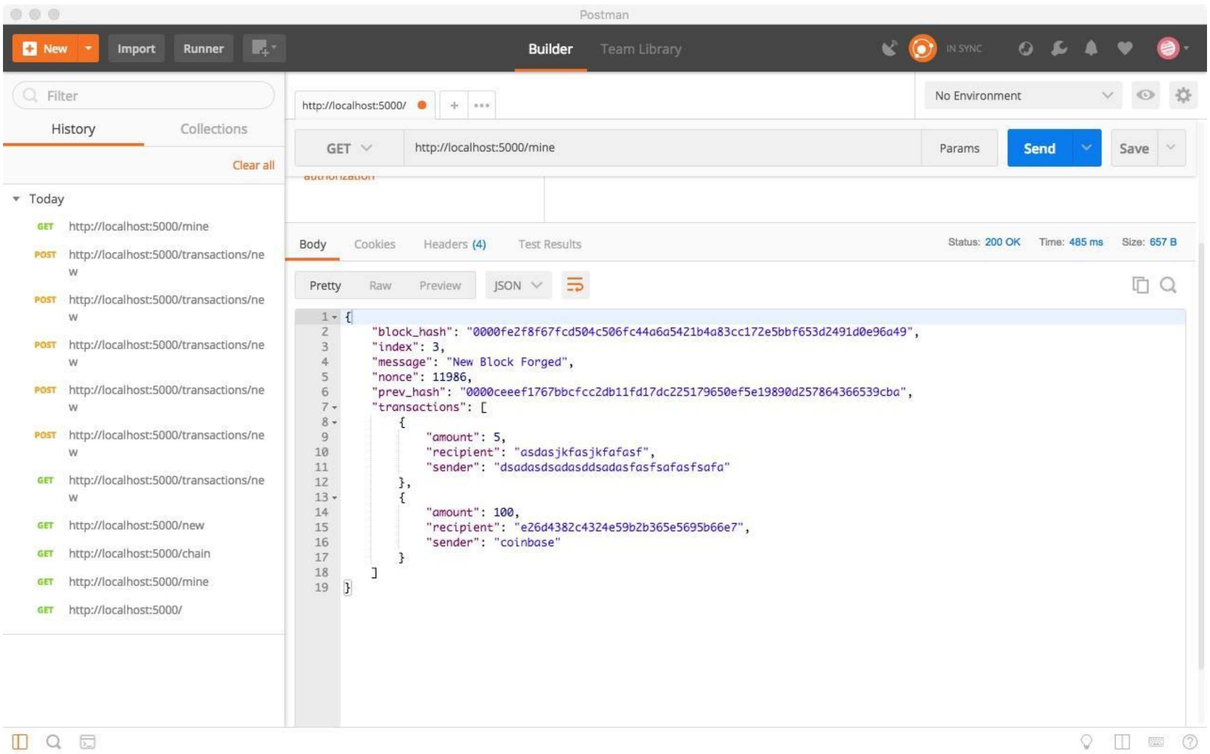
This endpoint above is */chain*, which returns a response showing the entire blockchain and the length of the blockchain.

Endpoints are like places on a server. People can access it by adding endpoints after the server address and it retrieves information if selecting "Get" and add information from local if selecting "Post". When people access the endpoints, there could be just returned strings and could also trigger operations. Generally, endpoints are address/directions pointed to some kind of information.

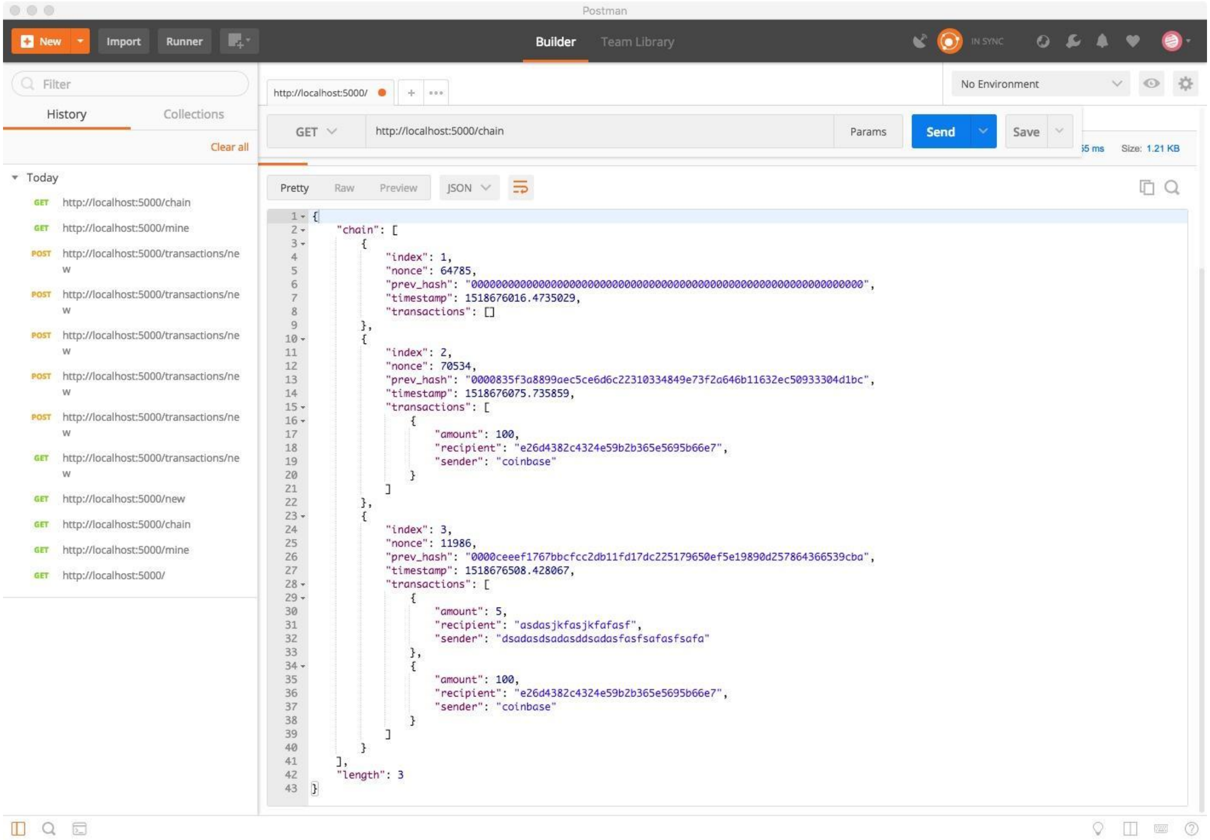
d) Endpoint *transactions/new* creates a transaction the specifies the “sender”, “recipient”, etc and assigns it to the new block that will be mined the next.



And the next miner will mine (using */mine*) the block with the newly added transaction.



The new chain is formed and can be retrieved by */chain*.



e) When the program tries to detect conflicts and solve them, the following chunk of code checks whether the chain at any one of the neighbors is longer and valid. If so, that chain will be taken as the valid chain and replace the current one of the host. Therefore the conflicts are solved.

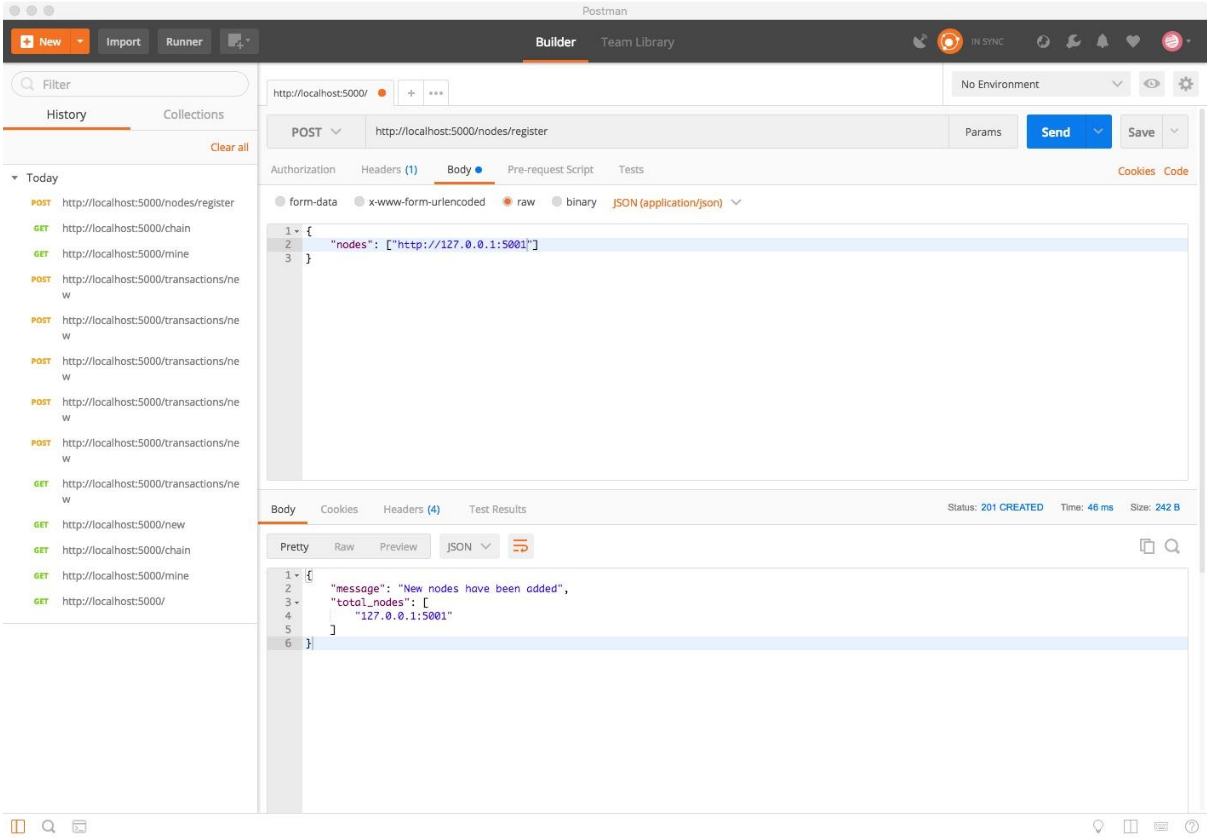
```python
for node in nghbrs:
    rp = requests.get(f'http://{node}/chain')

    if rp.status_code is 200:        #validation
        length = rp.json()['length']
        chain = rp.json()['chain']

        if length > max_length and self.valid_chain(chain):
            max_length = length
            new_chain = chain
```

The code above calls *valid_chain()* to validate the chains at neighbor nodes. The most important part that analyzes the chain is the following:

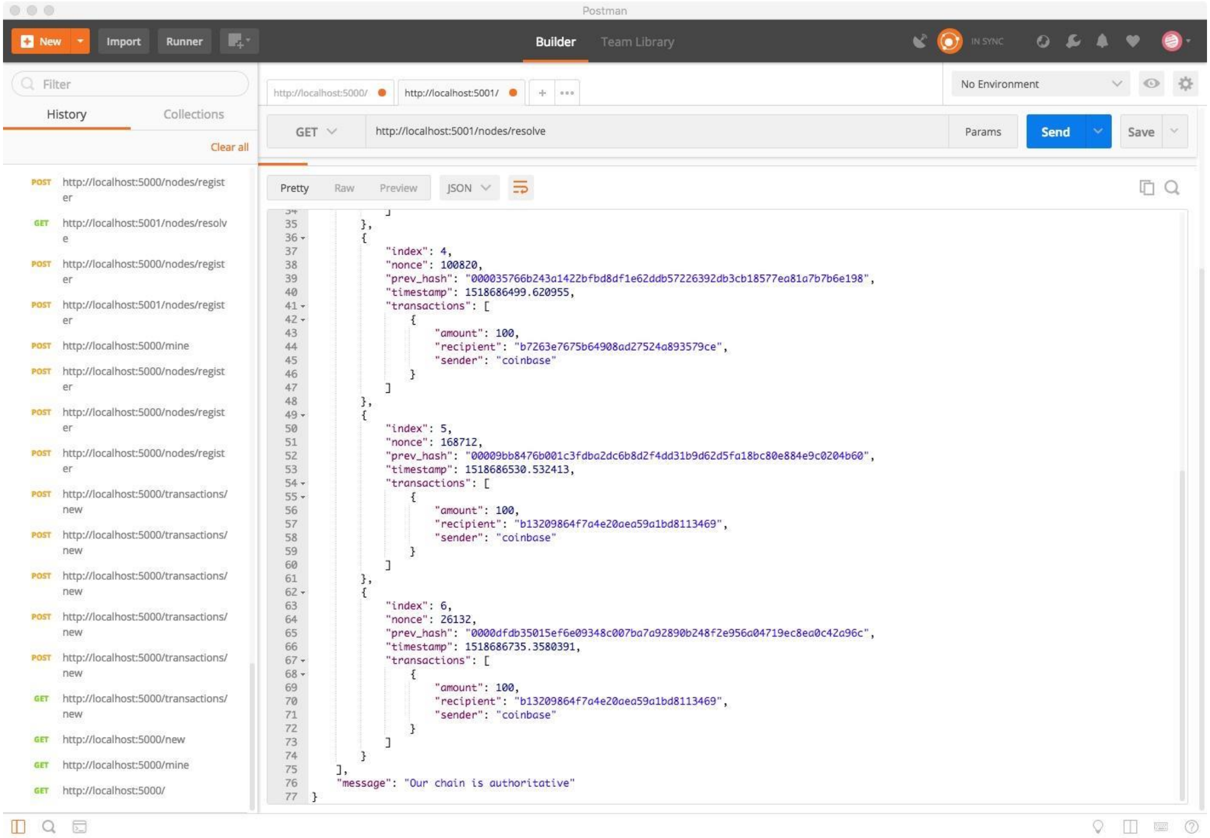
```python
    if block['prev_hash'] != self.hash(last_block):
        return False

    if not self.valid_proof(block['index'], block['timestamp'], block['transactions'], block['nonce'], block['prev_hash']):
        return False
```

The first “if” checks if the block belongs to the chain by checking the hash of the previous block and being checked by the next block. The second “if” checks whether the proof of work is valid by checking the if the hash meets the requirements. If the chain passes two “if”’s, it is to be proved valid. Following is an example of the use of */resolve* (*resolve_conflicts()*).

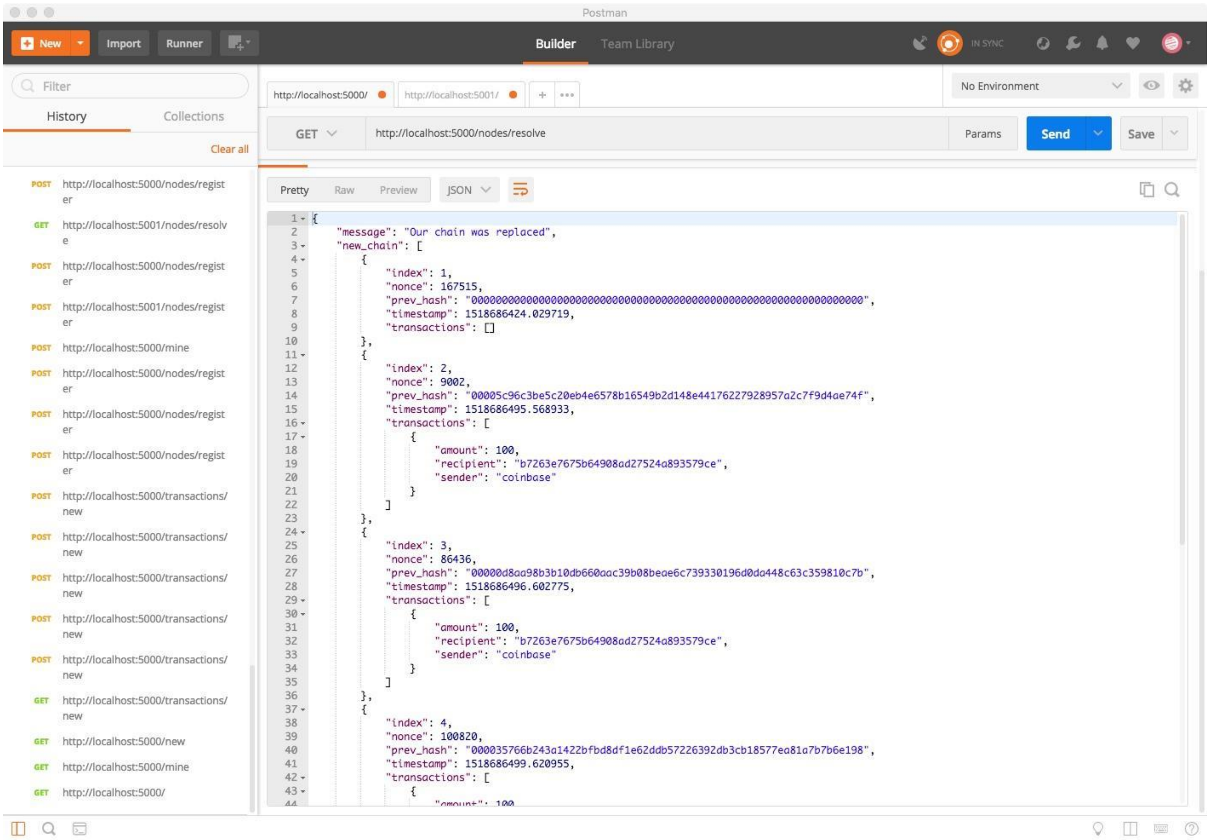
Initially port:5001 is registered by port:5000.



Port:5001 then mines 5 times so it now has 6 blocks in its chain while port:5000 has only one(the genesis block). There is a conflict. Port:5001 has the authorative chain.



And the chain at port:5000 is replaced.



Roles:

We discussed this question and read the tutorial together.
Jing is mainly in charge of part a, b and c
Zheng is mainly in charge of part d and e.
We wrote the documentation together.