**EE4017 Internet Finance**

**Short notes on setting up a cryptocurrency network for the Bitcon project**

## 1. Environment and tools

Please installed the following packages for the examples below.

- Python 3.6.8
- cURL
- PyCharm IDE 2018.3.7 (Community Edition)

Remarks: it is not recommended to use co-lab at this point because co-lab does not support concurrent progress for the free version.

## 2. Import python modules

Please Import the following modules at the beginning of your python application:

```
import Crypto
import Crypto.Random
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
from Crypto.Hash import SHA
import binascii
import json
import requests
from flask import Flask, jsonify, request
from urllib.parse import urlparse
```

## 3. Initializing Flask framework and the client

Flask is a lightweight web application framework for Python. We can use it to build APIs to interact with the blockchain client through http requests. To begin creating the Flask object, his line is needed after modules import.

```
app = Flask(__name__)
```

Then, we can write the main method. In the main method, we need to initialize the wallet, the blockchain and run the Flask object.

```
if __name__ == '__main__':
    myWallet = Wallet()
    blockchain = Blockchain()
    port = 5000
    app.run(host='127.0.0.1', port=port)
```

## 4. Develop APIs using Flask framework

The first API we need is adding transactions to the transaction pool. The Transaction class is from the last tutorial. Note that Flask uses the @app.route() decorator to define an API. Note that all API return messages in JSON file format and a number behind it. That number is the HTTP status code.

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

    # Check that the required fields are in the POST'ed data
    required = ['recipient_address', 'amount']
    if not all(k in values for k in required):
        return 'Missing values', 400
    # Create a new Transaction

    transaction = Transaction(myWallet.identity, values['recipient_address'], values['amount'])
    transaction.add_signature(myWallet.sign_transaction(transaction))
    transaction_result = blockchain.add_new_transaction(transaction)

    if transaction_result:
        response = {'message': 'Transaction will be added to Block '}
        return jsonify(response), 201
    else:
        response = {'message': 'Invalid Transaction!'}
        return jsonify(response), 406
```

**All Flask API must be placed outside of all classes and the main method!!!**

The API to get the transaction pool:

```
@app.route('/get_transactions', methods=['GET'])
def get_transactions():
    # Get transactions from transactions pool
    transactions = blockchain.unconfirmed_transactions
    response = {'transactions': transactions}
    return jsonify(response), 200
```

Now we need to handle the chain. The first one returns the last 10 blocks only, while the second API returns the whole blockchain. The reason is, transferring the whole chain is time consuming especially when the size of chain is long. For example, the blockchain of Bitcoin takes 234GB, transferring it on the internet is difficult and resource consuming. Sometimes, we just need the last few blocks to confirm our transactions.

```
@app.route('/chain', methods=['GET'])
def last_ten_blocks():
    response = {
        'chain': blockchain.chain[-10:],
        'length': len(blockchain.chain),
    }
    return jsonify(response), 200


@app.route('/fullchain', methods=['GET'])
def full_chain():
    response = {
        'chain': json.dumps(blockchain.chain),
        'length': len(blockchain.chain),
    }
    return jsonify(response), 200
```

Now, we can write a method to register other nodes. First, we need to store the IP addresses of other nodes in the cryptocurrency network. A set in Python is an unordered collection of unique elements. We can **modify** the **Blockchain class** and add an empty set named 'nodes' to store these IP addresses.

```
class Blockchain:
    difficulty = 2
    nodes = set()
```

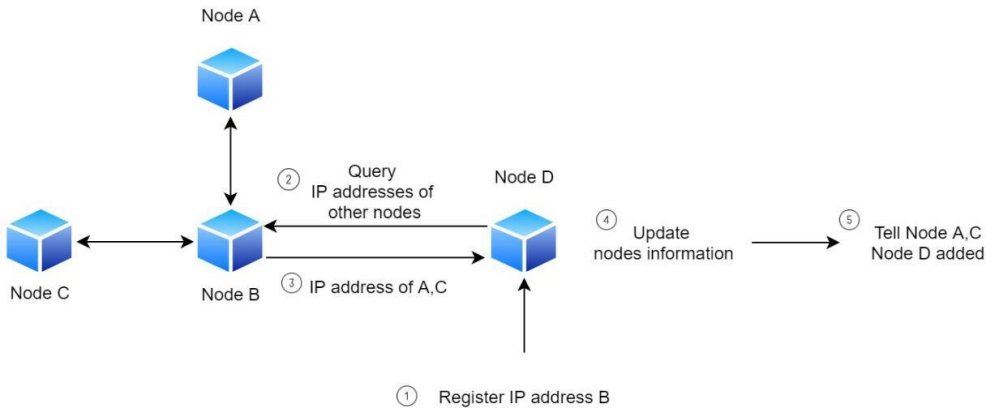
We need to provide an API to access these IP addresses for other nodes. This is crucial for the network formation.

```
@app.route('/get_nodes', methods=['GET'])
def get_nodes():
    nodes = list(blockchain.nodes)
    response = {'nodes': nodes}
    return jsonify(response), 200
```

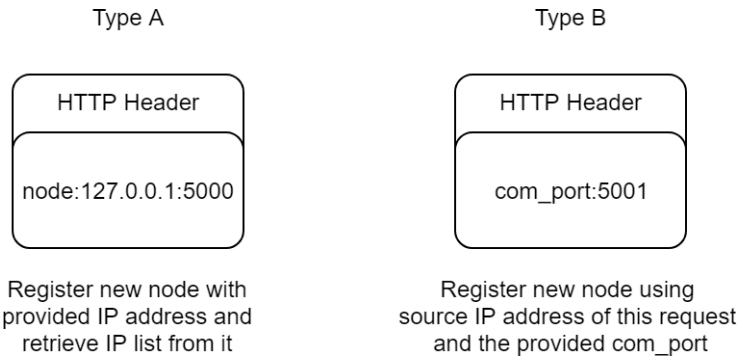
Then, we can create a method to register the new node inside **Blockchain class**.

```
 def register_node(self, node_url):
     """
     Add a new node to the list of nodes
     """
     # Checking node_url has valid format
     parsed_url = urlparse(node_url)
     if parsed_url.netloc:
         self.nodes.add(parsed_url.netloc)
     elif parsed_url.path:
         # Accepts an URL without scheme like '192.168.0.5:5000'.
         self.nodes.add(parsed_url.path)
     else:
         raise ValueError('Invalid URL')
```

Finally, the API to register new node. This is much more complicated because we need to propagate the IP address of all nodes. The diagram below shows the steps we need to make to connect Node D to the cryptocurrency network. At the beginning, Node A and C are connected to Node B. Therefore, we can use the API '/get_nodes' to obtain the IP addresses of Node A and Node C from Node B. In fact, Node A, B and C should have the same sets of IP addresses. After this API call, all nodes should add the IP address of this new node to their list.




To achieve such a design, we need to handle 2 types of requests. The first one retrieves the nodes list from the given IP and add them to the local list. The second one is used to notify A and C that D has added their IP so that they can add D.



```
@app.route('/register_node', methods=['POST'])
def register_node():
    values = request.form
    node = values.get('node')
    com_port = values.get('com_port')
    if com_port is not None:
        blockchain.register_node(request.remote_addr + ":" + com_port)
        return "ok", 200
    if node is None and com_port is None:
        return "Error: Please supply a valid list of nodes", 400

    blockchain.register_node(node)
    node_list = requests.get('http://' + node + '/get_nodes')
    if node_list.status_code == 200:
        node_list = node_list.json()['nodes']
        for node in node_list:
            blockchain.register_node(node)
    for new_nodes in blockchain.nodes:
        requests.post('http://' + new_nodes + '/register_node', data={'com_port': str(port)})

    replaced = blockchain.consensus()
    if replaced:
        response = {
            'message': 'Longer authoritative chain found from peers, replacing ours',
            'total_nodes': [node for node in blockchain.nodes]
        }
    else:
        response = {
            'message': 'New nodes have been added, but our chain is authoritative',
            'total_nodes': [node for node in blockchain.nodes]
        }
    return jsonify(response), 201
```

Note that the above API is a method called ‘consensus’ from the **Blockchain class**. Since a cryptocurrency network is decentralized, no entity can define rules. Therefore, all clients in a blockchain network must have a consensus on the rules before deployment. These consensus rules must be hard coded into the client. Consensus is important in a cryptocurrency network because it has economic value, which could bring potential conflict of interest. In this tutorial, we use proof-of-work as consensus algorithm, but we still need some more rules to solve conflict when there are multiple blocks in the network. They are:

- Broadcast a new block to the network once found
- Longest chain is authoritative

```
def consensus(self):
    """
    Resolve conflicts between blockchain's nodes
    by replacing our chain with the longest one in the network.
    """
    neighbours = self.nodes
    new_chain = None

    # We're only looking for chains longer than ours
    max_length = len(self.chain)

    # Grab and verify the chains from all the nodes in our network
    for node in neighbours:
        response = requests.get('http://' + node + '/fullchain')

        if response.status_code == 200:
            length = response.json()['length']
            chain = response.json()['chain']

            # Check if the length is longer and the chain is valid
            if length > max_length and self.valid_chain(chain):
                max_length = length
                new_chain = chain

    # Replace our chain if we discovered a new, valid chain longer than ours
    if new_chain:
        self.chain = json.loads(new_chain)
        return True

    return False
```

In a cryptocurrency network, we might receive a full of copy of the chain from other nodes. We should validate this chain before replacing it with ours. The above code is called ‘valid_chain’ method from **Blockchain class**. This method looks like this:

```
def valid_chain(self, chain):

    check if a bockchain is valid
    current_index = 0
    chain = json.loads(chain)
    while current_index < len(chain):
        block = json.loads(chain[current_index])
        current_block = Block(block['index'],
                              block['transactions'],
                              block['timestamp'],
                              block['previous_hash'],
                              block['hash'],
                              block['nonce'])
        if current_index + 1 < len(chain):
            if current_block.compute_hash() != json.loads(chain[current_index + 1])['previous_hash']:
                return False
        if isinstance(current_block.transactions, list):
            for transaction in current_block.transactions:
                transaction = json.loads(transaction)
                if transaction['sender'] == 'Block_Reward':
                    continue
                current_transaction = Transaction(transaction['sender'],
                                                  transaction['recipient'],
                                                  transaction['value'],
                                                  transaction['signature'])
                if not current_transaction.verify_transaction_signature():
                    return False
            if not self.is_valid_proof(current_block, block['hash']):
                return False
        current_index += 1

    return True
```

A consensus API is needed for other nodes to notify us that a new block is formed and should have initialized a synchronization process.

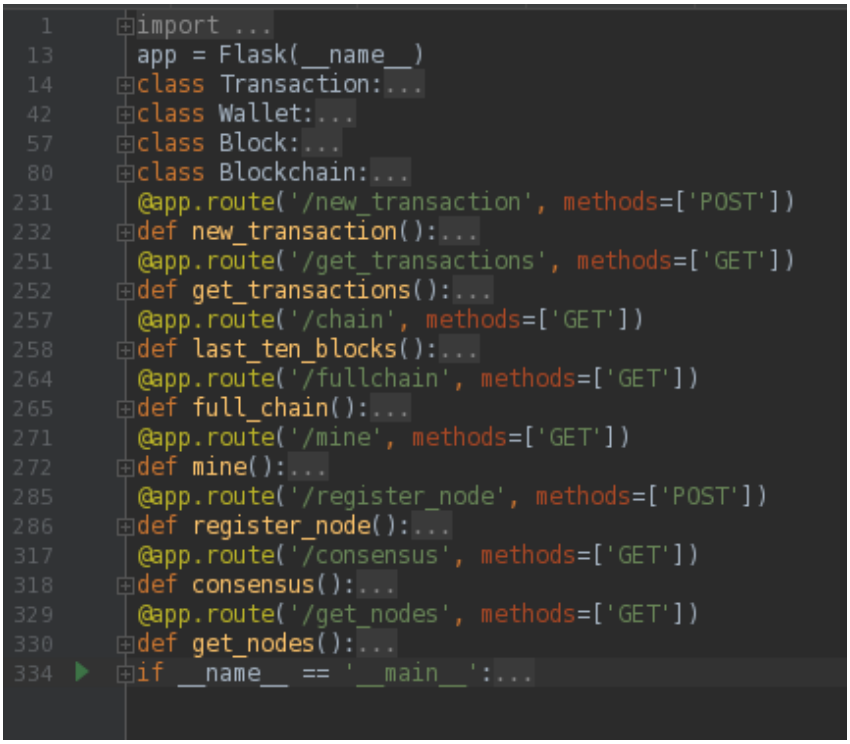
```
@app.route('/consensus', methods=['GET'])
def consensus():
    replaced = blockchain.consensus()
    if replaced:
        response = {
            'message': 'Our chain was replaced',
        }
    else:
        response = {
            'message': 'Our chain is authoritative',
        }
    return jsonify(response), 200
```

Finally, we can create a mining API:

```
@app.route('/mine', methods=['GET'])
def mine():
    newblock = blockchain.mine(myWallet)
    for node in blockchain.nodes:
        requests.get('http://' + node + '/consensus')
    response = {
        'index': newblock.index,
        'transactions': newblock.transactions,
        'timestamp': newblock.timestamp,
        'nonce': newblock.nonce,
        'hash': newblock.hash,
        'previous_hash': newblock.previous_hash
    }
    return jsonify(response), 200
```

**A simple Cryptocurrency client has been built!**

The structure of the whole application should look like this. Note that the details of classes and functions have been shortened.

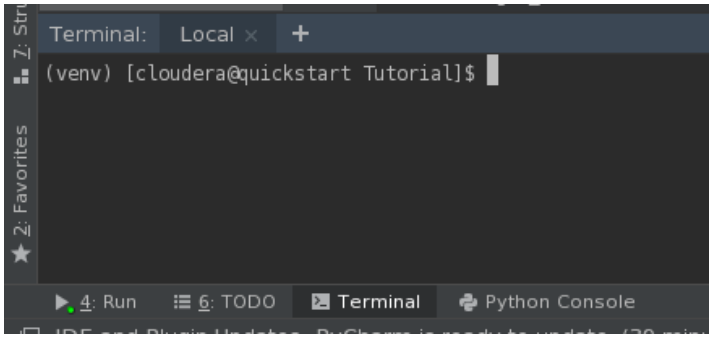



## 5. Test APIs with cURL

Before playing with the APIs, you need to duplicate the application and change the port numbers to emulate multiple nodes in the same machine. The above example uses port 5000, you can use port 5001 and port 5002 as your second and third clients. To do so, change the port number in the main function.


After running three clients, you should see their file name in the 'Run' tab such as below:

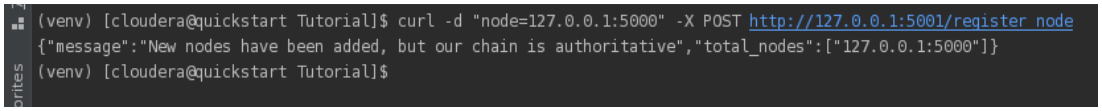
'curl' is used in command lines or scripts to transfer data to a URL address. We can use it to test our APIs in the terminal. Pycharm provides a terminal access in the bottom of the development interface.




Command:

```
curl -d "node=127.0.0.1:5000" -X POST http://127.0.0.1:5001/register_node
```

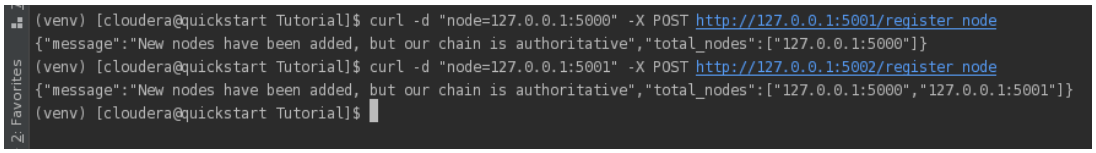
This command tells 127.0.0.1:5001 to register 127.0.0.1:5000 as a node.



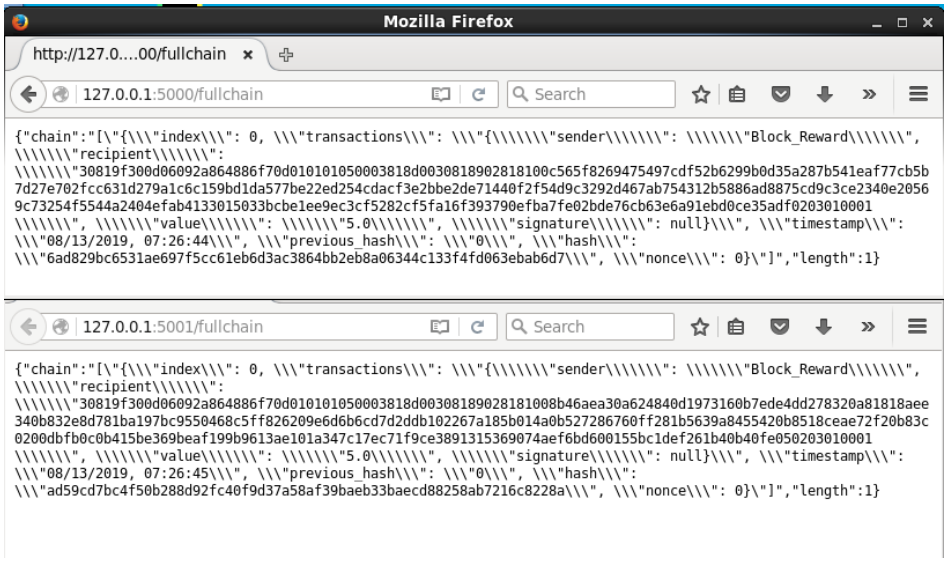

Command:

```
curl -d "node=127.0.0.1:5001" -X POST http://127.0.0.1:5002/register_node
```

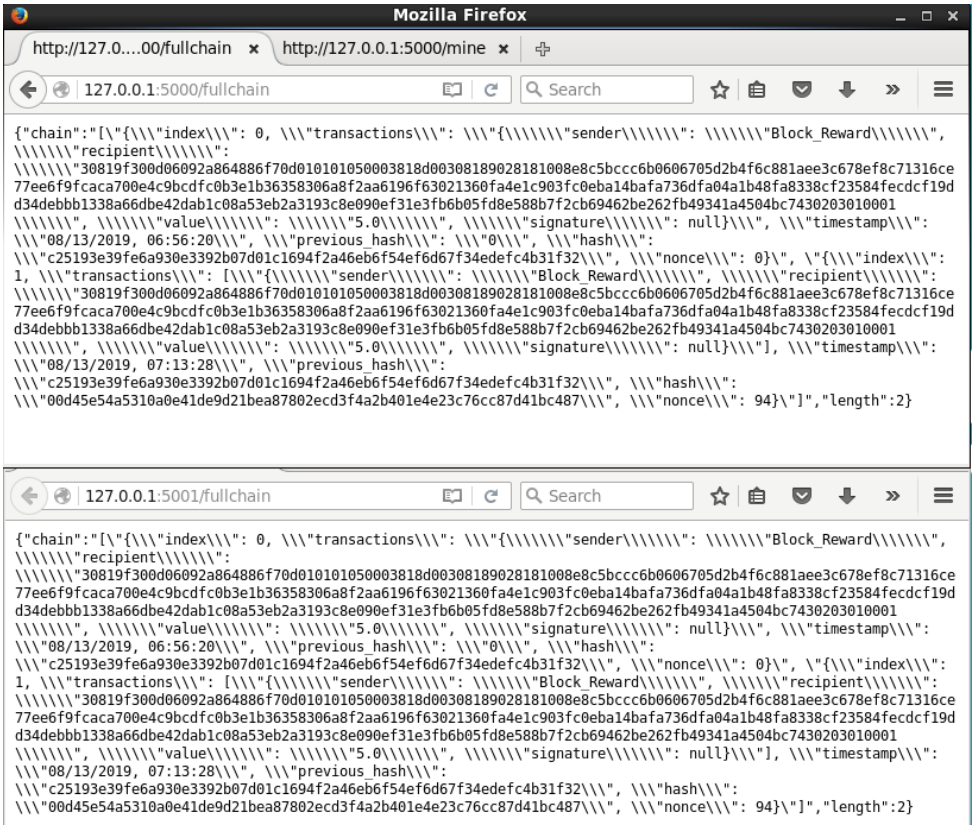
This command tells 127.0.0.1:5002 to register 127.0.0.1:5001 as a node. During the register process, 127.0.0.1:5001 responds the IP list which contains 127.0.0.1:5000.




Now, the three nodes have registered each other. You can use the browser to get the full chain as below. Note that they are two different chains.



Now, try to call ‘mine’ API in a client and refresh both ‘fullchain’ page. Both are synchronized and display the same copy of blockchain.




To submit a transaction to the node you can use the following command:

```
curl -d "recipient_address=$wallet_public_key&amount=5" -X POST
http://127.0.0.1:5002/new_transaction
```

Note: You need to replace $wallet_public_key with an actual wallet public key. Then you need to perform mining operations to write this transaction to the block.