**Blockchain Network Exercise – Blockchain Independent Study**

**Introduction**

In the previous exercise, we explored most of the fundamentals of creating a blockchain, including:

* Defining an individual block
* Defining a blockchain
* Defining a consensus mechanism (PoW)
* Defining a mining procedure

In this exercise we’ll be adding on to that exercise. We’ll add functionality which enables us to interact with the blockchain as if it was a protocol. To do this we'll make use of a light python web framework called Flask to create a RESTful API which integrates with our current blockchain implementation. We’ll also be using the Postman platform to locally interface with this API.

**Instructions**

(TODO: Add instructions for how students should install Postman, setup an environment which they can install the Flask & requests libraries in, and how they can obtain the “Blockchain-Network-Template.py” file)

Now that all the necessary resources have been setup, we can begin to develop the API. To start, add the following code below to at the bottom of the python file:

app = Flask(\_\_name\_\_)

blockchain = Blockchain()

These lines are responsible for defining the API and for creating a local instance of the blockchain.

Now, the first endpoint, called ‘/get\_chain’, is ready to be shown. When queried, it will return relevant information about the blockchain, such as the length of the chain and the chain itself. This endpoint has been created for you to demonstrate the syntax of Flask endpoints. Examine and add the following endpoint code to the end of the python file:

@app.route('/get\_chain', methods = ['GET'])

def get\_chain():

chain\_data = []

for block in blockchain.chain:

chain\_data.append(block if type(block) is dict else block.\_\_dict\_\_)

response = {'chain': chain\_data, 'length': len(blockchain.chain)}

return jsonify(response), 200

Along with this endpoints name, it can be seen in its header that it’s a ‘GET’ endpoint. This means that users who query it want to interact with the blockchain and *get* information in return. The response that’s sent back to the user who queried the endpoint will contain the information they requested in json form, as well as a 200-level response code, indicating success.

To ensure that everything is working so far, let’s try running and querying our API. To do this we’ll start by adding the following line of code to the end of the python file:

app.run(port=5000)

Next, using a command prompt, we’ll navigate to where the python file is located. Once it’s found, execute the python file. You should see text that is similar to the following:

Text

Description automatically generated

After this, open the Postman application, create a new ‘GET’ request using the API’s address (found in the command prompt where the API is running), in combination with the ‘/get\_chain’ endpoint, and then send said request. This request should instantly return the genesis block, and a length of 1 (indicating that only one block exists). If not, contact your instructor for assistance.

Graphical user interface, text, application, email

Description automatically generated Graphical user interface, text, application

Description automatically generated

Now that a platform for debugging has been established, we’ll begin creating the rest of the endpoints. The next endpoint which can be created will be called ‘/is\_valid’, and is a ‘GET’ endpoint. This endpoint, as well as all following endpoints can be inserted in between the ‘/get\_chain’ endpoint and the ‘app.run’ statement. Here is the initial code stub for the ‘/is\_valid’ endpoint:

@app.route('/is\_valid', methods = ['GET'])

def is\_valid():

# TODO

return jsonify(response), 200

After this, another ‘GET’ endpoint can be created, which will be called ‘/mine\_block’. This endpoint will be responsible for mining one block and returning the blockchain’s outputted information to the user once mined. The code stub for the ‘/is\_valid’ endpoint can be re-used for this endpoint, however, don’t forget to change all instances of ‘is\_valid’ to be ‘mine\_block’.

Next, a different type of endpoint can be made, this will be a ‘POST’ endpoint which will be called ‘/add\_transaction’. This endpoint will be responsible for interacting with the blockchain through the means of receiving data which is *posted* to it by any user who queries it. More specifically, a user will send a transaction to this endpoint that they want to be added to the mempool. This transaction will be represented as json data containing a ‘sender’, ‘receiver’ and ‘amount’. The endpoint will verify that all these elements of the transaction are all present before adding the transaction to the mempool and sending a response that the transaction was successfully added. If any of the elements of the transaction are missing, the endpoint will return a response with a 400-level response code, indicating failure, as well as a message clarifying what’s wrong. Here is the initial code stub for the ‘/add\_transaction’ endpoint:

@app.route('/add\_transaction', methods = ['POST'])

def add\_transaction():

json = request.get\_json()

# TODO

return jsonify(response), 201

To test the functionality of this endpoint once it has been made, open up Postman once again. Create a new ‘POST’ request, use the same API address as before, however, this time use the ‘/add\_transaction’ endpoint. To send the include the transaction with this query, **(1)** navigate to the ‘Body’ tab, **(2)** click the ‘raw’ radio button, **(3)** select ‘JSON’ in the rightmost dropdown menu, and then **(4)** add a json object, which has a sender, receiver, and amount, in the text box.

Graphical user interface, text, application, email

Description automatically generated

4

3

2

1

With this endpoint now complete, all of the key features from the previous exercise have now been implemented using the API. This brings us to the main focus of this exercise, demonstrating decentralization through treating the blockchain as a protocol by giving it a network. To create the infrastructure to this, we’ll need to add a few methods and endpoints to our code which allow us to have multiple users, or *nodes*, which will be in control of the blockchain. We’ll begin by revisiting and adding two new functions the blockchain class.

The ‘add\_node’ method is responsible for parsing any formatted URLs that it’s given that represent the addresses of other nodes in the blockchain network and appending said addresses to an unordered set. This way each node can keep track of all the other nodes on the network. This method is given to you, add it to the bottom of the Blockchain class.

def add\_node(self, address):

self.nodes.add(urlparse(address).netloc)

The ‘replace\_chain’ method is responsible for iterating through each of the nodes in the network, determining if any of them have a longer chain then what they currently have, and if so, replacing their own chain with the chain of said node. This can be done by using the [requests](https://docs.python-requests.org/en/latest/) library to get each of the other node’s chains [ex: response = requests.get(f'http://{node}/get\_chain')]. For each of these chains, their length should be compared to the length of the current node’s chain, and if the current node’s chain is shorter in length, it should be replaced. If this method finds a longer chain and replaces its own with said chain, it should return True, otherwise it should return False.

def replace\_chain(self):

# TODO

Next, we’ll create endpoints that individual nodes can interact with which use the methods that were just created. First, we’ll create a ‘POST’ endpoint called ‘/connect\_node’. This endpoint will accept a list of the addresses of all other nodes on the network. In order for a new node to join the network, or many new nodes, all existing nodes will need to query this endpoint, using a list of the new node’s addresses as the query’s parameter. Using the provided code stub, create a method which receives a json object that should contain a list of nodes, verifies that the list contains at least one node, adds all of said nodes to the blockchains node set, and then responds with a conformation message (this message can optionally contain a list of all of the blockchain’s nodes).

@app.route('/connect\_node', methods = ['POST'])

def connect\_node():

json = request.get\_json()

nodes = json.get('nodes')

# TODO

return jsonify(response), 201

Lastly, we’ll create an ‘GET’ endpoint, called ‘/replace\_chain’ which will allow us to ensure that all the chains in the network stay up to date. To do this, complete the provided code stub such that the ‘replace\_chain’ method is called and the node who queries the endpoint will receive an informative message as to whether their chain was replaced with another node’s chain.

@app.route('/replace\_chain', methods = ['GET'])

def replace\_chain():

# TODO

return jsonify(response), 200

To check that these endpoints work, we’ll open Postman once again, as well as three command prompts in the same directory as the python file. One these are open, now duplicate the python file three times in its own directory, call these three copies BNN1.py, BNN2.py, and BNN3.py (BNN standing for blockchain network node). Modify the ‘app.run’ line of each of these files such that they’re running on different ports [ex: 5001, 5002, 5003]. Next, run each of these nodes/API’s in the three separate command prompts, and then head over to Postman. For each of the nodes, create a post request which queries the ‘/connect\_node’ endpoint with similar parameters to the following screenshot. Observe that the address of the node querying the endpoint isn’t included in the list of node addresses.

Graphical user interface, text, application, email

Description automatically generated

When all the nodes have successfully connected to each other, a decentralized network has now been formed around the blockchain. To demonstrate the network in action, follow these steps:

* Start by calling the replace\_chain endpoint on one of the nodes, a response should be returned saying that the chain wasn’t replaced, as all chains are currently equal in length
* On a separate node, add a transaction, mine a block, and call the get\_chain endpoint
* On the node that initially called replace\_chain, call the get\_chain endpoint, notice how this chain is different than the chain on the other node, because the network hasn't been synchronized yet
* Call the replace\_chain method on both nodes which didn’t mine the block, if everything’s working properly, all nodes should now have the same chain and a message similar to the following should be returned

Graphical user interface, text, application, email

Description automatically generated

This marks the end of the exercise, feel free to continue exploring the blockchain network that you made. One flaw in the system when compared to large scale networks is the lack of a communal transaction mempool. The correct functionality would be if any of the nodes added a transaction to the mempool, when any other nodes mined a block, it would contain said transaction. For extra credit, you can attempt to implement a communal network mempool so that all nodes have an updated transaction list when mining blocks. There are multiple ways to do this such as creating a replace\_transactions endpoint, checking transactions pools of all nodes before mining, etc.