Introduction to Blockchain: Technology and Applications
Programming Assignment # 1

Out: Feb 3, 2022. Due: Feb 13, 2022 at 11:55 pm via LMS

Course Rules

The objective of this assignment is to enhance your understanding of Bitcoin's protocol for **mining** blocks and **validating** chains (blocks and transactions within the blocks). Those two functionalities are mimicked in this assignment. The programming environment for this assignment is the Linux machine for the course `ispl01.lums.edu.pk`. If you are enrolled in class, an account has been created for you and login details have been shared by email.

You are welcome to discuss any issues and confusions with the course staff but **NOT with anyone else**. There is a strict plagiarism policy. **Share blockchains, not code.**

Introduction

Each student in the class will run a full node that will be able to create and broadcast candidate blocks to everyone else in the class (through us). You will be writing the code for block creation (i.e., mining) and we will (through a utility function) take care of broadcasting your candidate blocks to others in class.

Candidate blocks broadcast by others do not automatically get to you. You will have to request us (through a utility function call) for receiving newly created blocks.

You will also be required to write code for validating the transactions you pick from a mempool (using a utility function we provide) before you mine a block.

You will further be required to validate individual blocks broadcast by other students before adding those to your own journals (your own view of the longest valid blockchain).

To facilitate above tasks, we have provided you with a skeleton code. You are only allowed to edit `FullNode.py`, and even within `FullNode.py`, do not remove any skeleton code (or redefine any constants). You are, however, free to use the utility functions provided in `util.py` for general operations such as picking transactions (from the mempool) and saving blocks (that either you have created or others have broadcast to you) to your chain saved in the file system (folder named `valid_chain` in your working directory).

1 Getting Started

There are two parts to this assignment:
Part 1 – Mining blocks by including valid transactions (50 Marks)
Part 2 – Validating of blocks broadcast by others (50 Marks)

Both parts require you to run the file `FullNode.py`, by using the following command on your terminal:
python3 FullNode.py

2 Mining

You will be using the Proof-of-Work mechanism: creating blocks by extracting transactions from a mempool and searching for a valid nonce. There are several steps to ensure your program mines the correct blocks, given a set of transactions. For this part, we recommend implementing these steps in the given order (choose valid transactions, create a possible block object, and search for a valid nonce in the block

object to create a candidate record).

Note that some of the transactions extracted and provided to you from the mempool are purposely corrupted. You will be required to disregard such corrupted transactions (details below) as we will only need valid transactions in each block of the global ledger.

Mining will occur when you run the 'mine' command. For this part of the assignment, ignore the 'validate', 'rl', and 'ra' commands. You will be using them later.

Tip: We highly recommend keeping your code modular, as you may reuse functions you implement in this part in the next part.

2.1 Choosing transactions

Takeaways: Being able to check if the UTXO and digital signatures of transactions are valid, marking corrupt transactions as such (and adding those to your corrupted transactions list).

We have provided a function to extract from the mempool the transactions you do not yet have in your valid chain (and are also not in your corrupted transactions list). The extracted transactions list (that is stored as `self.unconfirmed_transactions`) has exactly MAX_TRANSACTIONS transactions, however, some of them are corrupted. It is your responsibility to select only valid transactions (that will be added to the block you mine later). We are using an extremely simplified version of Bitcoin protocol where you only have to check for two things:

1. UTXO: Is the UTXO input and output mathematically correct?
2. Digital Signatures: Is the digital signature valid according to the provided public key?

It is your responsibility to mark transactions as corrupt when you have identified them so that they should not be picked from the mempool in the future either. Store these corrupt transactions in `self.corrupt_transactions` to disallow picking these up again from the mempool.

We require each block to contain a maximum of MAX_TRANSACTIONS transactions (and a minimum of 1 transaction). Keep this in mind when picking the transactions from the mempool.

For the above task, you will need to add your code to **mine(self)** function.

2.2 Creating blocks

Takeaways: Understanding the structure of block, being able to use the Block() function to create object, and saving a block object to the chain.

For this part as well, you would need to edit **mine(self)** function.
The `valid_chain` folder stores your view of the longest valid chain in the network. At the start, let us assume we are not connected to the network. For now, let us work on populating this folder by creating some dummy blocks to get the hang of things.

Each blocks is stored as a separate file with the .block extension in this folder. To save a block here, you must first create a Block object (which is conveniently already defined for us in the `Block.py` file).

Identify which arguments need to be passed for the Block initialization (see Appendix). Moreover, try running the **save_object(*block, file_path*)** function on this block object to save it as a file (with .block extension).

Note: You can always manually delete blocks from your directory, however **do not delete the genesis block i.e., block0.block**.

2.3 Hash Function & PoW

Takeaways: Using the hash function to calculate the hash value of a given block, iterate through nonce till you have the correct number of preceding zeroes in the hash value.

For this part, add your code to **proof_of_work(self, block)** and **mine(self)** functions.

We learnt about the SHA-256 algorithm in class. Now it is time to implement it. The function that takes a block as an input and outputs its SHA-256 hash value is already given to you. You will need this. After you have done the previous steps, you have all the necessary building blocks to mine a block by iterating through the **nonce** value and find one that outputs a hash value that suits the difficulty level provided as a constant self.DIFFICULTY (this constant stores the minimum number of preceding zeroes for the hash of a valid block in our protocol).

You are free to use any method to find the nonce. Just ensure that once you have created a valid block with the correct nonce, you save it to your chain using the **save_object(*block, file_path*)** function. To understand the file path and directory structures well, please refer to the appendix section below.

3 Validating

Now that your node is able to mine blocks, let's put it in an environment where everyone else can too! For this part of the assignment, you will be receiving blockchains from other students enrolled in this course in a folder labelled 'pending_chains' (after you request them with the 'ra' or 'rl' commands; the former requests chains from **all** students and the latter requests the longest as perceived by us).

Note that you will not be receiving entire blockchains every time; you will only receive the portion of the blockchain that you do not already have in your valid_chain folder. That way, if you and your friend have the same blockchain and they mine an extra block, when you request for their chain, you will only receive one block.

It is your program's job to go through these chains and replace/add to your view of the valid chain (stored in the valid_chain folder). The ultimate goal is to have the **longest valid chain in the network** (i.e., your class) in your 'valid_chain' folder.

This validation should occur whenever you use the 'validate' command. The file reading and iteration through all pending chains and blocks has been handled for you, however it is your job to validate individual blocks.

After the validation, the system cleans up all the pending chains folders. These folders will get populated again if you request blocks again using the 'ra' or 'rl' commands.

Note: To let the rest of the nodes know that you have updated your blockchain (either after you have mined a block, or you have gone through validation of pending chains, or even if you have manually changed the .block files), use the 'send state' command. Your state is also automatically sent to us when the `FullNode.py` file is run. Your updated state is sent by us to others whenever they request it.

Add your code to **verify_chain(self, current_longest,temp_chain,last_block_hash)** function for this part. Again, we recommend implementing this in the following order:

3.1 Validate hash

Firstly, check if the computed hash of a given block in the pending chain meets the Proof-of-Work condition. If not, stop!

3.2 Check number of transactions

Do the number of transactions in this block match the criteria (minimum and maximum number of transactions in the block) we have set for you? If not, discard the block.

Note: *Skip this check for the genesis block since it has been structured in a certain way that it contains seeding transactions that fuel the subsequent transactions in next blocks, so its transaction count is more than the standard count.*

3.3 Check Previous Hashes

Now we will check if the hashes of all the blocks in a given pending chain are valid. For this, iterate through the chain and check if the previous hash of a block equals computed hash of the previous block.

3.4 Validate Individual transactions

If the hash of the block is valid, we now need to check every individual transaction to see if any corrupt ones made their way (perhaps, by another malicious node). Luckily, you did this when you were mining your own blocks so feel free to reuse that functionality here. Here are the two checks you need to reuse here:

1. UTXO check
2. Digital signatures check

Another extra check you will have to do here is **ensure there are no repeated transactions (use transaction IDs for this)**.

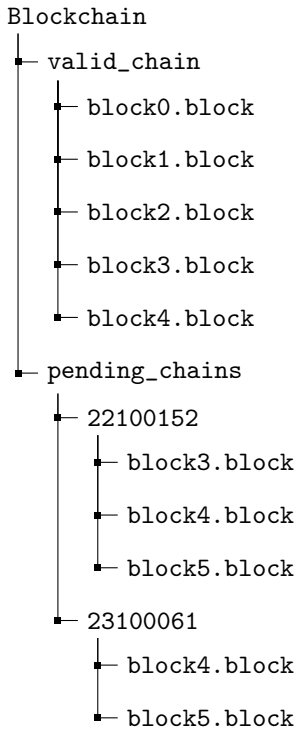
3.5 Bonus

While implementing your program, you probably noticed that it chooses the longest chain but if there were multiple chains of the same length, it chooses the **first** valid chain it checks. As a bonus, try to implement the following condition:

If multiple pending chains have the same number of blocks, prefer the chains with more transactions in them. If multiple chains have the same number of transactions, pick a random chain from among them.

Appendix

I. **Directory Structure**
This following diagram shows how directories and files are structured in the main Blockchain directory provided to you.



Unpacking the tree above:

- **'valid_chain'** folder contains the current blockchain (journal, if you like) of the node.
- A chain is composed of its blocks, and each block is stored in the format: **block{*index_number*}.block**
- In order to append a new block to the **valid_chain**, follow the above mentioned format.
- **'pending_chains'** folder contains the incoming blockchains of other nodes. These chains, in turn, are stored in sub-directories that are named after the *roll numbers* of the sending nodes.

II. **Utility Functions**

1. **save_object(block, file_path)**

   Input parameters: block object, file path.

   Functionality: This method takes in a block and saves it in the specified path.

   Example Usecase: **save_object(new_block, "valid_chain\block6.block")**

2. **save_chain(chain)**

   Input parameters: a blockchain stored as a list of blocks.

   Functionality: This method takes in a chain and saves it in the valid_chain folder.

   Example Usecase: **save_chain(new_chain)**

3. **self.last_block**

   Input parameters: None

   Functionality: Returns the latest block stored in your 'valid_chain' folder.

III. **Block structure**

Block	
index	A numerical index signifying where this block is in the blockchain
transactions	A list of transactions
time_stamp	The time this block was mined stored as a string in the format "day-month-year (hour:minute:second)"
previous_hash	Hash of the previous block in this chain
nonce	self explanatory
miner	ID of the person who mined this block. Place your ID here when you're mining

Table 1: The structure of a block stored in .block files

IV. **Transaction structure**

Transaction	
id	Unique transaction ID
render	Sender public key
receiver	Receiver public key
value_receiver	The value the receiver receives
value_sender	The value the sender keeps
UTXO_input	The input the sender put into the transaction
signature	Digital Signature
time_stamp	The time this transaction was done stored as a string in the format "day-month-year (hour:minute:second)"
signature_token	The token that was signed by the sender

Table 2: The structure of a transaction object