CS-3812: Introduction to Blockchain: Technology and Applications
Assignment # 1: Simulating a Node

Lead TA: Zaeem Mohtashim Khan, Huzaifa Ahmad

Release date: September 11 2025, Deadline: September 28, 2025

Course Rules

The objective of this assignment is to enhance your understanding of Bitcoin's protocol for mining and validating chains. You are welcome to discuss any issues and confusions with the course staff, however **DO NOT ask other students for guidance. There is a strict plagiarism policy. Share blockchains, not code.**

Introduction

Not too long ago, your teaching assistant became *very* motivated after one late-night lecture on Bitcoin. Inspired by Satoshi, they decided to launch their own blockchain. But there was a small problem: a blockchain with only one node isn't much of a blockchain at all. So, they turned to the class for help. Each of you will now run your own node, validate transactions, mine blocks, and share your chains with others — mirroring the structure and protocols of Bitcoin. In other words, you're not just students completing an assignment — you're early "miners" in Zaeem's grand experiment. Whether this chain grows into something legendary or collapses into chaos depends on how honestly and carefully you build your part of it.

To make this possible, we have provided a skeleton code. **You are only allowed to edit FullNode.py, and even within FullNode.py, do not remove any skeleton code/redefine any constants.** You are free to use the utility functions provided in util.py for general operations such as receiving transactions and saving blocks to your chain. You will work on the ISPL server for this assignment. A guide to setting up SSH and connecting to ISPL has been uploaded on LMS.

In this assignment, we will also be running an honest miner, 24/7. Its primary objective is to facilitate the inclusion of your valid block into the blockchain. Additionally, this continuous operation serves to verify that you have successfully added a valid block to the chain, guaranteeing the allocation of marks by some of the test cases. This process is further elaborated in Section 2 onwards.
First and foremost, **to run the file**, use the following command on your terminal:

```
python3 main.py
```

This will open an interactive menu you will use for this assignment.
There will be two parts to this assignment:

Part 1 – UTXO Database Construction (40 Marks)
Part 2 – Mining (20 Marks)

1 UTXO Database Construction

This is the first part of this assignment and involves you understanding how transactions are verified and how double spending is avoided in the Bitcoin blockchain.

1.1 Structure of a transaction

You are provided with a set of `Transactions` in your "mempool". The structure of these `Transactions` is available in the `Transaction.py` file. Each `Transaction` is a **python dictionary** and NOT an object.

Here are the fields (keys) of this `Transaction` dictionary:

1. **id:** This is the **hash** of the transaction. The exact mechanism to compute the hash of a transaction is to convert it into a string and then apply the SHA256 function on this string. Fortunately, the `stringifyTransaction` function in `hashing.py` converts a transaction into a string and the `calculateHash` function can compute the hash from any string and return a hash (as an integer).
2. **COINBASE:** This is a boolean variable that states whether the Transaction is a coinbase transaction or not. If it is a coinbase transaction, then it should have no inputs. Instead, it has 50e8 satoshis. It has, however, at least 1 output.*(You may assume that each unique user whose transactions are present in the mempool has only one coinbase transaction.)*
3. **inputs:** This is a python `list` of inputs. The exact details of inputs will be provided below.
4. **outputs:** This is a python `list` of outputs. The exact details of outputs will be provided below.
5. **number:** This is an integer. It is inconsequential and we only added it for your convenience; the transactions are ordered (roughly) according to this number. It is not used in the hash function or any signature.

We will now define what an `input` is. Recall that `Transaction['inputs']` is a list of multiple `inputs`. Each input is a tuple `(prevTxnId, output_number, signature, PubKey)`.

- prevTxnId: This is a hash of the parent transaction (as an integer).
- output_number: This is the output number in the parent transaction. Recall that a transaction may have multiple outputs. This specifies which output is being unlocked.
- signature: This is the signature associated with PubKey that is used to unlock the output. We will give more details about the signature verification later in this handout (in the subsection labelled "Transaction validity verification").
- PubKey: This is the public key whose corresponding private key may be used to unlock the output.

What you should note is that an `input` does NOT specify how many satoshis are being unlocked.
We will now define what an `output`. Each output is a tuple `(value, PubKeyHash)`.

- value: This is the number of satoshis being locked.
- PubKeyHash: This is the hash of the Public Key that can be used to unlock this output. The function `hashPubKey` can be used to hash a Public Key.

These fields are the only piece of information that you must use to verify the integrity and validity of a transaction. You may access these transactions from the list of transactions in `self.unconfirmed_transactions`. Please note that these are in no particular order.

1.2 Transaction validity verification

There are **two** parts to verifying whether a transaction is legitimate or not.

The first part verifies whether a transaction has integrity. In order for a transaction to be valid, **ALL** the signatures present in this transaction MUST be correct signatures. Here are the steps to verify whether a transaction is corrupt or otherwise – For each input:

1. Pick the `sig` from the input.
2. Store the parent hash as `parentHash`. This is already present in the inputs as **PrevTxnID**.
3. Compute the hash of the stringified current transaction EXCLUDING all the signatures present in this transaction. This may involve using the `stringifyTransactionExcludeSig` function. This hash is an integer `currentHash`.

4. Compute a final string `str(parentHash) + ':' + str(currentHash)`.
5. Finally, take the hash of the final string as `finalHash`.
6. Apply the Public Key to `sig` to get a tentative hash.
7. if the tentative hash and finalHash do not match, the transaction is corrupt!
8. Finally, if the PubKeyHash in the parent transaction's output is the same as the hash of the PubKey present in this input, the transaction is not corrupt!

If a transaction is corrupt, it should be discarded; you may manually move it to `self.corrupt_transactions`. The second part of verifying a transaction relates to the validity of a transaction.
If a transaction is valid, it is:

1. NOT sending out more satoshis than those unlocked by the inputs.
2. NOT attempting to unlock an output that has already been unlocked.

Tip : It is possible for a transaction in a block to use the output of another transaction present earlier in the same block. Make sure your code does ***NOT*** *consider these blocks invalid.*

1.2.1 Verifying a signature on a string

We have provided you with a function `VerifySignature(string, signature, PubKey)`. You may use it to verify signatures.

1.3 Keeping track of UTXOs

UTXOs are unspent transaction outputs. Recall, when validating a transaction you have to ensure that the same output of a transaction is not being used by two child transactions. Why is this necessary?

Keeping track of this requires the construction of a **UTXO DATABASE**. A `UTXO DATABASE` is constructed by *sequentially* going over all the transactions in a block and then through all the blocks in a blockchain. It keeps tracks of which transaction output has been spent (i.e. has already been unlocked in a child transaction).

Here are the steps necessary to construct a `UTXO DATABASE` from a list of (not-corrupt) transactions:

1. Pick a transaction. For each input, find its parent transaction's output (from the provided id and output number). If this output does not exist, this transaction is invalid and should be rejected.
2. If this (parent) output exists in the UTXO database, remove it from the UTXO database.
3. Add the outputs of THIS transaction into the UTXO database.
4. If the sum of inputs (acquired from the UTXO database) is greater than or equal to the sum of outputs of this transaction (as written in this transaction) then this transaction is valid.

   If this transaction is invalid, revert changes brought on by this transaction to the UTXO database.
   Ignore the transactions of the first block, i.e., Genesis Block, in the construction of your UTXO.

The `UTXO DATABASE` may be implemented in many ways. You may implement it using a python list or a python dictionary. We recommend using a python dictionary.

If you use a dictionary to implement a `UTXO DATABASE`, this about the following two questions:

1. What will the "keys" be of this dictionary?
2. What will the "values" be for this dictionary? [1]

[1] Python is very flexible. Each "value" may be another dictionary or a list. However, a key must be something simple like a string or a number.

1.4 The verifyTransaction function

This is the only function you need to consider in order to pass the first few test cases. It checks whether a transaction is not corrupt AND valid. Furthermore, for each valid transaction, this function should update `UTXO_Database_Pending`. Moreover, when checking the validity of a transaction, this function should utilize `UTXO_Database_Pending`.

1.5 Querying user balance

The function `showAccounts` should print the balance (in satoshis) for each user. In our case, each user is identified by their **PubKeyHash** [2]. The function should print the sum of UTXOs locked to a PubKeyHash. Furthermore, it should return a dictionary with the PubKeyHash as the "key" and the sum of UTXOs as the "value". *Kindly note that in the case where a user's all UTXOs gets deleted from the database, you need to consider them as a user with a zero balance.*

2 Mining and Proof-Of-Work

This part involves you creating blocks filled with valid transactions to create a blockchain and verify the chains of your peers.

2.1 The update_UTXO function

This function should go through all the transactions in the blocks in your *valid_ chain* folder and update `UTXO_Database` accordingly. Note that this does not need to modify `UTXO_Database_pending`.

2.2 Collecting valid transactions

Before you can mine a block, you must collect valid but unconfirmed transactions from your "mempool" (i.e. self.unconfirmed_transactions). You are free to traverse the mempool in any order in-order. The appropriate function for this is the `findValidButUnconfirmedTransactions` function. By default, this function should return 5 valid transactions.

2.3 Making a block

You will modify the `mining` function for this part.

Call the `update_UTXO` function and `findValidButUnconfirmedTransactions` function in order to acquire 5 valid but unconfirmed transactions. Create an object of the `Block` class and write an appropriate index, list of transactions, unix-time (as a string), last block hash and a nonce. You may initialize this nonce as `0`.

Then, you need to do the following:

- Do Proof-Of-Work on this block (as described in the next subsection) to find a nonce. This involves calling the `self.proof_of_work` function on this block.
- Append this block to `self.valid_chain`.
- Write this block in your valid_chain folder. Writing this will involve using the `save_object` function.*(Syntax for this function call is provided in `FullNode.py`).*
- You may choose to update your UTXO database or your pending UTXO database.

Once you perform these steps, you blockchain will be stored in your valid_chain folder. You may broadcast it to the rest of your peers by typing "send state".

[2]This is what we call an address

2.4 Doing Proof-Of-Work on this block

Recall **nonces** from the lectures. When performing Proof-Of-Work, we iterate through the nonce such that the block hash[3] meets a certain "leading zeros" criteria.

It is now your job to find a nonce such that the block hash (computed using `self.computeBlockHash`) meets this leading zero criteria. The number of leading zeros is given by `self.DIFFICULTY`. You may modify the nonce in any way that you like. This function should take in a block and return the final, valid nonce.

2.5 Verifying other people's chains

You will verify the chains received from other students.
You can ask for chains from other students by typing the following two commands:

ra: This will give you the chains of all students.

rl: This will give you the longest chain.

It should be noted that you will only receive the difference of your chain and the received chain. That is if you have a chain till block 3 and the recipient chain is till block 5 then you will only receive block 4 and block 5 in the temp_chain folder.

In order to verify the chain you have to do the following four tasks:

- Verify if the indexes of the blocks chain correctly, i.e. Block N follows Block N-1.
- Verify if the blocks are chained together correctly using the hash, i.e. the previous hash in block N matches the Hash of the Block N-1. You can use the `self.computeBlockHash` function to compute the hash of a block.
- Verify if the Blocks meet the leading zero criteria. The number of leading zeros is given by self.DIFFICULTY.
- Verify if all the transactions in the block are valid, i.e no corrupt transactions are present in the block. Luckily, you did this in part 1.2 so feel free to reuse that functionality here. There are two parts to verifying whether a transaction is legitimate or not.
  - Check if ALL the signatures present in this transaction are correct signatures.
  - UTXO check
  - If a block contains fewer than 5 valid transactions, refrain from invalidating it, as our miner has the capability to generate blocks with less than 5 transactions.

  To implement the above task you need to edit the `self.verify_chain` function. This function should return True if the given longest_chain/temp_chain is valid else it should return a False.

2.6 Test cases for UTXO Construction

For this part of the assignment, you can run the given test case file. Use the following code for this task:

```
python3 testCode.py
```

Once the code is run, you will get an output of the marks recieved out of a total of 40 for Part 1. The output will also contain what specific part of the test cases failed.

[3]In bitcoin, it is the block header hash but that is not the case here.

2.7 Test cases for block creation

You may verify whether your blocks meet the difficulty criteria. However, verifying whether your block is being mined perfectly is left up to a miner "humza" being run on the back end.

In order to get the final 20 points, you must mine a block with *5* transactions, append it to the longest chain and broadcast it. If it gets picked up by our miner as part of *its* longest chain, then we will know your block is being mined properly. We have made sure that our miner works slowly and regularly requests for blocks that you have broadcasted.

Appendix

2.8 Directory Structure

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

```
Blockchain
├─ valid_chain
│  ├─ block0.block
│  ├─ block1.block
│  ├─ block2.block
│  ├─ block3.block
│  └─ block4.block
└─ pending_chains
   ├─ 24100053
   │  ├─ block3.block
   │  ├─ block4.block
   │  └─ block5.block
   └─ 241001003
      ├─ block4.block
      └─ block5.block
```

Unpacking the tree above:

- **'valid_chain'** folder contains the current blockchain 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.

2.9 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.

2.10 Tips and Tricks

- Always use deep copy when dealing with your UTXO database. If you don't you'll be modifying the global UTXO, even if the transaction turns out to be invalid.
- Always ensure that the valid_chain directory contains block0 since it is required for your code to work properly. If you accidentally end up deleting the block, you can always copy it from the longest chain you received from the server.
- Do not mine more blocks than needed. If your block has been mined correctly, it will become part of the valid chain. You are sharing the chain with all your other classmates - let them mine blocks as well. We have access to everyone's individual chain and any attempt at tampering will be dealt with accordingly.
- In case your code is flawed and that has caused you to mine invalid blocks, you can delete all blocks (except block0) from the valid_chain folder and reconstruct your chain from there using the process described in the manual.
- Ensure you always follow the correct order of operations. Before mining you should request the longest chain, validate it, and them mine a block. After this you have to broadcast your state so our miner can pick up your block and append to it.
- Start your assignment early and before writing any code try to have a map of what each function will do. Make sure you check for all details mentioned in the class, manual, and tutorial.

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

Transaction structure

Transaction	
id	Hash of the Transaction
COINBASE	Boolean value denoting is it a coinbase transaction?
inputs	This is a python list of inputs
outputs	This is a python list of inputs
number	This is an integer. It is inconsequential and It is not used in the hash function or any signature.

Table 2: The structure of a transaction object

Input and Output structure

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

```
Transaction
 ├─
 │   ├─
 │   ├─
 │   └─
 │
 ├─ Transaction['inputs']
 │   ├─ Tuple1:  (prevTxnId, output_number, signature, PubKey)
 │   │
 │   ├─ Tuple2:  (prevTxnId, output_number, signature, PubKey)
 │   │
 │   ├─ Tuple3:  (prevTxnId, output_number, signature, PubKey)
 │   │
 │   ├─ Tuple3:  (prevTxnId, output_number, signature, PubKey)
 │   │
 │   └─ TupleN: ...
 │
 └─ Transaction['outputs']
     ├─ Tuple1:  (value, PubKeyHash).
     │
     ├─ Tuple2:  (value, PubKeyHash).
     │
     ├─ Tuple3:  (value, PubKeyHash).
     │
     ├─ Tuple3:  (value, PubKeyHash).
     │
     └─ TupleN: ...
```

Submission Instructions

You will be asked to submit only `FullNode.py`. Please upload `FullNode.py` to the LMS tab.