Introduction to Cryptocurrencies - Ex2

Due date: Tuesday, April 12th
(This exercise can be solved in pairs)

1 High Level Goal

In the first exercise you built a toy payment system that was based on a trusted authority – the bank.

Although the system was based on cryptographic primitives that the bank cannot break (e.g. public/private key signatures), the bank could still refuse to admit transactions, and we did not fully verify blocks that it produced. It could also print as much money as it wanted.

In this exercise, we will replace the Bank with nodes running according to the longest-chain protocol.

As before, we still assume that transactions have only a single input and a single output and always carry a single coin.

2 Description

The API for "Node" in this exercise is a combination of the APIs of banks and wallets from ex1 with some extensions. Our nodes can both mine new blocks, as well as send and receive funds. Each node manages its own mempool, and its own copy of the blockchain and UTxO set. Each node has its own address (public key) to handle its funds.

As before, we provide the APIs that you should implement. For some objects like Block and Transaction, you will find your implementation from ex1 useful (naturally, you can re-use your code). In other cases, you'll need to make adjustments.

As in ex1, we will use the cryptography package (for signatures), hashlib (for SHA256), and the secrets module (to generate a random sequence of bytes that will be included in money creation transactions).

Below we outline some of the processes that were added to the APIs or changed from ex1.

Communication Between Nodes Nodes can connect to each other (via an API call). Upon connection, nodes notify each other about the tip of their blockchain. Preexisting mempool transactions are not shared upon connection.

Connected nodes notify each other of new blocks they mine, as well as blocks they receive (after learning of a new blockchain-tip from someone, all of the

nodes' neighbors are informed of that new tip). Such notifications only take place if the blocks received were valid, and ended up being part of the longest chain.

Nodes additionally notify their neighbors about each new transaction that they hear about that has made it into their mempool (transactions that did not enter the mempool because they double-spend another mempool transaction are not propagated).

Block Validation and Chain-Reorgs Nodes notify each other about the hash of a new block that they have mined or discovered. If the hash is part of an unknown block (to the notified node), then the block itself is requested in response. If this block in turn specifies an unknown previous block (as the previous block in the chain), that too is requested. This is repeated until a known block (or the Genesis block) is reached.

Occasionally, during this process nodes may discover that another node holds a longer chain than they do. In this case, you need to update the blockchain to be the newer chain, along with all related data-structures (such as the UTxO set and the mempool).

Other nodes we are connected to may be malicious, and may try to alter blocks or transactions; therefore, it is important to verify that the blocks and transactions that are received are correctly formed.

Malformed blocks may be too large (exceed the block size limit that is set as a constant in the code), have malformed transactions in them, double spends, or a bad amount of money creation transactions (more or less than 1). Be extra careful not to accept malformed or invalid blocks into the chain.

Mining Our Nodes will not really use a proof-of-work process to decide when they mine blocks. Instead, an external call (by the test script, for example) will notify a node that its turn to create a block has arrived (in regular PoW mining, all nodes would constantly try to create blocks, and the one that finds a correct nonce would succeed – We prefer to save CPU cycles, and just pretend we are running this process). As such, there is no need to check that the hash of a block is below some target.

Money Creation Each block should contain exactly one transaction that creates money. This transaction assigns the money to the miner.

Transactions that create money have no input, and instead of a signature, have a random sequence of bytes (as in ex1).

Sending Money Since nodes function both as wallets and as miners, when a node creates a transaction it is immediately placed in its mempool and propagated to its neighbors. Nodes attempt not to double-spend their own transactions, so they will not re-issue transactions that spend outputs that are already spent by other transactions in their mempool. However, clearing the mempool of a node will allow it to re-try to spend those outputs again.

In addition to placing transactions in their own mempool, nodes notify all their neighbors of new transactions. They in turn notify their neighbors, and so on (this only occurs if transactions did indeed enter the mempool).

3 Tests

As in Ex1, we are providing you with some basic pytest tests for the API. These should help clarify the expected behavior.

The provided tests are partial, you are encouraged to write more to supplement them. See for example pytest-cov, a tool which checks how much of your code is covered by the tests.

What we will not test: There are many nuances to how nodes behave. One such complication is that nodes that change their longest chain may have transactions in their blocks that can be re-included in the new chain (they are not double spends of what appeared in the other chain). We will not check your mempool to see if you included these. We will also not check your mempool's behavior in case your node attempted to change to a new chain that was later discovered to be malformed.

Lastly, we will not attempt to access or modify internal fields in your Nodes or Blocks. Tests will only consist of normal expected behavior or of "attacks" that external attackers can commit: sending badly signed messages, malformed blocks, etc.

4 Submission

Submit your files in a single zip file called ex2.zip. The structure of the zip file should be flat, containing only files from the ex2 subdirectory. You are allowed to add additional python files to this directory (e.g., ones containing additional classes) if you wish to. You are naturally allowed to modify the files we provided as long as you do not break the API that we describe (e.g., you can add private methods, instance variables, and any additional classes or functions you wish to have). In addition to .py files, the zip should also contain a `README` with a brief description of the project.

Finally, the zip should also contain a file named `AUTHORS` that has 1 or 2 lines (one line per author of the exercise). Each line should contain a user name and a student ID separated by a comma.

For example:

```
buggs_bunny22,055512345
elmer_fuddd,032112457
```

Be sure to have only one partner submit the exercise in moodle (the grade will be given to both partners in the AUTHORS file).

To run your solution we will import from 'ex2' as the tests do. Make sure this will only define the relevant classes (as in the template files) and will not run additional code.

Make sure that your submission passes the presubmit tests so that we can test it successfully.

Good luck!