CS 5433 SP23: Homework 2

Instructor: Ari Juels
TAs: Andres Fabrega, Zhi Liu, Sishan Long
Due: 13 March 2023

Policies and Submission

Submit your written solutions and code to CMSX.

Integrity. For all assignments, you may make use of published materials, but you must acknowledge all sources, in accordance with the Cornell Code of Academic Integrity. Additionally, you must ensure that you understand the material you are submitting; you must be able to explain your solutions to the course instructor or TA if requested. You must complete this homework assignment *on your own.*

Setting Up. Full setup instructions and documentation for the provided code is available by opening docs/index.html in a web browser. Testing will be done using Python 3.9. Some test cases are provided with the code but additional tests may be used for grading.

Problem 1 - Signatures, Hashes, Sealing

In this problem, we explore basic cryptography and its applications to Bitcoin. *Hash functions* are used to secure blockchains through a mechanism called *Proof of Work*; the course textbook has more in-depth background on "PoW". We now implement proof of work and a popular alternative.

(1a) - Proof of Work

Proof of Work is required by the network for block validity; each miner in a cryptocurrency system is constantly attempting to produce valid blocks by finding a block such that: `H(block)` < `target`.
More precisely, SHA256(SHA256(`block`)) < `target`. In Bitcoin, "`H`" is SHA256^2.

One natural question is how blocks are represented in the system; our provided files `blockchain/block.py` and `blockchain/chain.py` provide an (unoptimized) infrastructure for representing and managing blocks.

In Proof of Work, what is hashed is in fact not the block but the **header** of the block; some binary metadata designed to be compact enough to efficiently communicate quickly through the network. Our block headers take the following format:

```
block = height‘timestamp‘target‘parent_hash‘is_genesis‘merkle_root‘seal_data
```

An example value is:

```
block = 100‘1519668704‘
45231284858326638837332416019018714005183587760015845327913118753091066265 6‘
0074aa074e3fee902d5e2a251e90f37f528425ebcef4ae1212f988a388877acc‘False‘
8f212c0356b46c1df1d909d0ddeee09a923129dfa0c36b949df4c5fa357db158‘184
```

represented as a Python string, with hexadecimal encodings for all binary values ("Merkle Root," i.e., root of a Merkle Tree, and signature) and no spaces.

The critical value here for header validity is the *seal data*, additional data which a protocol can specify as input to a function that checks whether a block is valid. In proof of work, a nonce counter is used for this seal data; a valid seal occurs when the header (including the seal data) hashes to below the target. Proof of Work mining then tries different seal data via brute force, checking if the block is valid until a valid seal (header hash) is found. A block is called *validly sealed* if its seal data causes the block to verify properly. In Proof of Work, the seal data is also called a nonce ("number that is used once").

Notice that there are no transactions in these blocks. The Merkle Root is a short hash that summarizes all transactions in a block, which are propagated by the network separately.

The mining lifecycle in our codebase for Proof of Work consists of the following process:

- A block object is created with full transaction data. This object is an *unsealed* block, as it starts with the default nonce of 0, and so may not be a valid PoW block.
- The mining loop checks whether the block's seal is valid (a.k.a. the hash of the block's header is below the target). If not, it increases the nonce/seal data of the block by 1 and tries again, until the block is valid.

For this problem, you will use the code provided in folder `p1-release`. An example of this process is given in `examples/add_single_pow_block.py`. You will first code some helper functions for this process.

When there's a fork, i.e., two or more competing blocks at the same height (mined at roughly the same time), To choose which block to use as the next block for mining and transacting on, the network uses what is called the "best chain" fork-choice model. Because blocks can have varied difficulties, just because a chain is longer does not mean more work was expended to produce it (it is possible to have a high-difficulty short chain if all targets are low, or a low-difficulty long chain if all targets are high). To solve this, each block is given a "weight", or an estimate of how hard the block was to create. The best block in the chain is then called the "heaviest" block, and is the block that is on the chain with the most weight. A block B's "chain" is the string of blocks between genesis and B, obtained by repeatedly following the parent hash pointers in the block header.

You should now complete the following functions (we recommend completing them in this order):

- `blockchain/util.py/sha256_2_string(string_to_hash)`: returns the SHA256(SHA256(`string_to_hash`)) hash of a given string.
  Provided tests: `tests/hash.py`
- `blockchain/pow_block.py/get_weight(self)`: Gets the "consensus weight" of a block's chain, as described above. In Proof of Work, the "weight" is the same as the total work on the chain. Calculate the weight using the ratio of maximum possible target to the block's target (`Block.target`), where the max possible target is 2^{256}. Notice that in the current codebase, all PoW mining is constant-difficulty, but your method should be able to handle later extensions to include blocks with different targets.
  Provided tests: `tests/weight.py`
- `blockchain/chain.py/get_chain_ending_with(self)`: Gets a list of blockhashes in the blockchain ending with the block represented by `block_hash`. The first item in the list should be the provided block hash, and the last item in the list should be the hash of the genesis block (as above, check with the `Block.is_genesis` attribute).
  Provided tests: `tests/get_chain.py`

Please refer to the provided Python docs for full documentation of all the data structures and methods required for this problem; for more complex methods, we provide some hints, but you may need to explore the documentation additionally on your own.

It is not enough for a block's seal to be valid; for a block to be accepted into the chain, the block must also obey all the rules of the currency system it is running (such as enforcing that no money is created out of thin air or no input is spent twice). Coding this validation will be the most substantial component of this problem. Please complete the following method:

- `blockchain/block.py/is_valid(self)`: returns True if a block is valid. A list of conditions for block validity is provided as comments in the included code, as well as a list of methods and datastructures you may find helpful. Note that the provided test file includes multiple unit tests; we will be assigning partial credit for implementing partial validation and passing some but not all tests.
  Provided tests: `tests/validity.py`

Add your solutions to the appropriate files, at (1a) placeholders therein.

(1b) - Proof of Authority

Proof of Work is an inherently wasteful system. In many systems that use blockchains, such as supply-chain blockchains (https://www.ibm.com/blockchain/supply-chain/), sometimes participants share a single root of trust or set of trusted parties. In such use cases, it is possible to construct a blockchain without the use of Proof of Work. Instead of the above seal validity conditions, a Proof of Authority seal is valid when

```
Verif(PK, [block, sig]) = accept
```

(where Verif is a function in the digital signature primitive described in Lectures 1 and 2).

For a block to be valid, it must be *signed* by the public key of a valid authority, who is trusted by all blockchain participants. Some version of this scheme is used in many enterprise deployments, where the ability to produce blocks can be cleanly delegated to a trusted party, but where all system participants want the ability to rigorously and cryptographically audit the actions of this party.

- You will implement the `mine` method corresponding to the provided method in the PoA block class; we provide utility methods `get_private_key` and `get_public_key` in the authority file; in a real consensus system, node implementations would only need the public key, with the corresponding private key available only to the trusted authority. To simulate this, notice that we **do not** use the `get_private_key` function in the provided `seal_is_valid`.

  Tests are provided in `tests/poa.py`.

Add your solution to the `blockchain/poa_block.py` file, replacing the (1b) placeholders therein.

Problem 2 - UTXO Management in Wallets

In our lecture on Bitcoin transaction structure, we looked at the representation of Bitcoin as inputs and outputs. Additional reading on this "UTXO" model of representing account state is available in the NBFMG textbook.

We have provided some code that generates a random proof-of-work blockchain; in this code, we simulate various users transacting with wallet software, choosing which UTXO to use randomly from the UTXOs available to a given user.

Unfortunately, this strategy is clearly not efficient: wallets want to maintain as few UTXOs as possible in Bitcoin, since transactions are charged according to size, and using / creating more output costs money.

Consider an enterprise business called Moonbase, which must make and receive hundreds of thousands of transactions to millions of users every day, paying millions of dollars in network fees. Moonbase's strategy is as follows: for every user that wants to deposit money, Moonbase receives a new UTXO from (and created by) that user. For every user that wants to withdraw money, they create a withdrawal transaction funded by a random UTXO that is large enough to fund this transaction.

You may make any reasonable simplifying assumptions you find helpful, as long as you explicitly state them. One particular helpful assumption may be, for example, that all malicious users in the first part of the below are able to communicate, and that these malicious users can spy on the blockchain to gain knowledge of Moonbase's full UTXO set. We do not require formal proofs for the below; rigorous and justified arguments will suffice.

1. Identify a denial-of-service vector in this process, i.e., an adversarial attack strategy by dishonest and wealthy users that could result in failed withdrawals for legitimate users of Moonbase. Provide a fix for this DoS vector and argue informally that user withdrawals will never fail.

2. For each user withdrawal in the provided scheme, recall from class that *two UTXOs* actually need to be generated: one paying the target user, and one that is kept by Moonbase representing any leftover

"change". Provide a modification to the above strategy that will reduce the number of UTXOs Moonbase must maintain in its database.

Add your solution to a `solutions.pdf` file.

Problem 3 - Basic P2P

In the previous problems, we implemented some of the features of a real cryptocurrency, including block validation and fork-choice rules, and proof of work and proof of authority mining.

Several features we learned about in recent lectures are missing in this homework. One of these is a peer-to-peer network, which allows several nodes to coexist across a network and synchronize their view of a blockchain.

For this problem, you will use the code provided in folder `p3-release` with some code from the first problem. We have made several changes to the previous codebase in support of this. We suggest you explore:

- The `run_node.py` file, a script that runs a self-contained node with a block explorer and connection to the peer-to-peer network. The operation of this script is explained in the web-based documentation under "Running Nodes" of the index page (`docs/index.html`). Configuration of available nodes on the peer-to-peer network is provided in `config.py`.
- The file `p2p/gossip.py` has been added, and the `p2p.interfaces` module has also been added to serialize and deserialize blocks and transactions as they are exchanged over the network. We recommend inspecting the interfaces, though their modification is not required for the assignment. All functions to handle incoming messages are provided (`gossip.handle_message`) , dispatching them to the appropriate submodule.
- The `generate_example_pow_chain.py` script will gossip blocks to any available nodes on the peer-to-peer network. Your code will have nodes re-gossip messages to any available nodes on *their* p2p network. Each message should only be forwarded once; code for this is provided in `handle_message` as part of re-gossip.

We recommend exploring the provided web documentation for a complete overview of the modules added to this version. Before starting this problem, please copy some of your implementation of the blockchain module from the first question (just the implementation of the following methods, not the entire scripts). Specifically:

- method `is_valid()` from `block.py`.
- method `mine()` from `poa_block.py`.
- method `get_weight()` from `pow_block.py`.
- method `get_chain_ending_with()` from `chain.py`, be aware that in this codebase `chain.py` is located in `blockchain/chaindb/`.
- method `sha256_2_string()` from `util.py`

Your implementation for these methods will not be used for grading p3, but they are necessary for you to finish the tasks in p3. The only code you need to submit for p3 is `p2p/gossip.py` (see the submission instructions at the end of this doc). We now ask you to complete the following (with solutions to 2, 3 placed in a **solutions.pdf** to be submitted to CMSX):

1. In `p2p/gossip.py`, complete the `gossip_message` function
   Provided tests: `tests/gossip.py`

2. First clear your databases for nodes 1-6 and the parent/master node (see the instructions on the web documentation under "Database Management" of the index page (`docs/index.html`)). Run 2-6 nodes (remember to modify the config file accordingly to comment out peers you don't run) in separate command line windows, and then run the `generate_example_pow_chain.py` file, which might take a few minutes. This will generate a blockchain of at most 100 blocks, although the exact number will vary due to the random generation. Furthermore, this file has been modified to broadcast its generated blockchain to all nodes in the gossip network. Open the block explorers of each of your nodes at the end of that script's execution; do they appear synchronized? Now, repeat the experiment, but this time stop one of the nodes around halfway through the execution, and restart it at the end. What do you observe, and does this suggest any additions required to our gossip protocol (if so, what changes, otherwise, why not)? You may consult the documentation of Bitcoin's gossip protocol, available at https://en.bitcoin.it/wiki/Protocol_documentation.

3. Notice that our node's list of peers is hard-coded in `config.py`. Does this suggest another missing component required to achieve a permissionless blockchain? Why not, or if so, what is the closest analogous message type in the Bitcoin p2p protocol documentation linked above?

Submission Instructions

Directory Structure:

```
hw2-<NetID>
 ├─ p1
 │   ├─ util.py
 │   ├─ pow_block.py
 │   ├─ chain.py
 │   ├─ block.py
 │   └─ poa_block.py
 ├─ p2
 │   └─ solutions.pdf
 └─ p3
     ├─ gossip.py
     └─ solutions.pdf
```