Homework 9: A Tiny Blockchain in Python

In this homework, you will write a simple miner on a permissionless blockchain using Proof-of-Work. This handout will guide you through the implementation steps, but feel free to diverge from the suggested design. As the final task of this homework, you will run you miner against an existing implementation that we provide. Hence, you must follow the specification for the messaging and the format of the blockchain.

NOTE: using the HTML version of this handout might be preferable for copy-pasting the code snippets.

Setup

You will use Python 3 to implement the miner. Start by creating a new directory for the project. We assume it is called `hw9`, and that it is your working directory. Inside this folder, create directory named `miner`, where you will implement your python miner. Also create an empty file `miner/__init__.py`, in order to be able to import `.py` files inside the `miner` directory.

The blocks

The basic data structure you will need is the block, so you should create a class for that. Create a new file `miner/block.py` containing the following template code:

```
class Block:
    def __init__(self, data, previous):
        self.data = data
        self.previous_hash = previous
        # TODO

    def hash(self):
        # TODO

    def encode(self):
        # TODO

    @staticmethod
    def decode(b):
        # TODO
```

This class has two class variables `data` and `hash`, holding the payload data of the block, and the hash of the previous block in the chain. The following invariant should hold for the class variables:

- `data` is a `bytes` object and can be of length 0 to 4096
- `previous_hash` is a `bytes` object of length 32.

Your first task is to implement the missing methods. The `hash` method must return a `bytes` object containing the hash of the block. The hash must be computed as `SHA-256(self.data||self.previous_hash)` where `||` denotes the bytes concatenation. The `encode` method must return a `bytes` object containing a UTF-8 encoded JSON object with the following fields:

- `data`: a base64-string representation of the `self.data` variable
- `previous`: the `self.previous_hash` as an hexadecimal string

The decode method must be a static method accepting a bytes object and returning a decoded Block instance according to the structure defined by the encode method.

The chain

The second data-structure that we need is a storage for the blocks. Create a new file miner/blockchain.py, that contains the following template code:

```
class BlockChain:

    def __init__(self, genesis_block):
        self.root = genesis_block
        # TODO

    def append(self, new_block):
        # TODO
```

The BlockChain class is initialized with the genesis block. Such block can have any valid payload and has null bytes as previous_hash. It keeps a collection of blocks in a data structure that is left for you to define. Only those blocks having a previous_hash path leading to the self.root block shall be stored stored in the collection.

You must implement the append method, which takes a Block object as argument, and appends it to the blockchain, if possible. The append method must return the newly appended block if it was appended to the chain, or None if it wasn't.

You should test your model carefully before proceeding to the main program implementation. You may find the pptree useful for printing your chain.

The miner

The next step is to implement the miner program. For that, create a new file miner/miner.py that contains the following template code:

```
import sys

class Miner:

    def __init__(self, host, port, miners, genesis):
        pass # TODO

    def broadcast(self, block):
        pass # TODO

    def run(self):
        pass # TODO


if __name__ == '__main__':
    if len(sys.argv) < 3:
        print("Usage: python3 miner.py [addr] [others] [genesis|]")
        print("\taddr:\t\taddress of the miner in the format" \
             "host:port")
        print("\tothers:\t\tcomma-separated list of the other" \
              "miners' addresses \n\t\t\tin the format host:port," \
```

```
            "host:port,...")
        print("\tgenesis:\toptional, \"genesis\" if the miner must" \
                "generate\n\t\t\tthe genesis block")
        sys.exit(0)
    # TODO
```

Run this program using `python3 miner/miner.py` to see what command-line arguments your implementation must support. As you can see, we simplify the implementation for this homework by assuming that a miner is responsible for generating the genesis block, and that the set of miners is fixed. Again, your task is to implement the class methods.

The `broadcast` method takes a `Block` object as argument, and sends it to all other miners through the UDP socket.

The `run` method is the main program loop. As a first test implementation, the genesis miner must send some blocks to other miners (without yet listening for block), who should update their chain accordingly. You can test this by starting several of your miner programs in different terminals. Thus, before proceeding, you should have a miner program that can either produce **or** receive blocks.

Concurrency

Naturally, the miners should be able to produce **and** receive blocks, so a little bit of concurrency will be required to implement both sending and listening routine. Feel free to implement your routines in your own way, but here are some advices:

- The multiprocessing package proposes various primitives for concurrent python programs. The `multiprocessing.Process` class can be used directly, or extended to implement the sending and listening loops.
- Unless you want to make your `Blockchain` objects thread safe, only **one** process should be updating the chain.
- As new blocks can be produced and received, this is *multiple producers, single receiver* situation where the `multiprocessing.Queue` primitive can be used as a convenient and elegant solution. You should investigate how you can solve this whole concurrency problem with a single `Queue` object.

Test your implementation by having each miner periodically (e.g., each second) producing new blocks on its own *fork* of the chain. This will test that your implementation of the various routine is correct before proceeding: Indeed, you eventually want the miners to agree on a single version of the history, i.e., to produce blocks on the same chain.

Consensus

In blockchain systems, consensus is achieved by accepting the longest chain as the current state. This introduces an incentive for miners to base their new blocks on the longest chain, and ensures progression of the system. Hence, you must change your miner to implement this behavior, and test that the miners can make the longest chain progress.

At this point, assuming reliable communication and honest miners, you have a working blockchain. Unfortunately, the minors can misbehave. More specifically, a miner willing to impose its own version of the history can try to produce many blocks to make its own chain the longest.

To ensure that miners cannot create too many blocks in a short time, you will implement a Proof-of-Work mechanism.

Proof of Work

To publish a new block on the blockchain, the miners will need to solve a cryptographic puzzle. Hence, the difficulty of this puzzle enforce the global rate limit for new blocks. This puzzle consists in finding blocks for which the bytes of the hash value, when interpreted as an unsigned integer in the a big-endian representation, is smaller or equal to the `TARGET` value. For the purpose of this exercise `TARGET` can be a constant in your program.

Augment your `Block` class with a `p_o_w` class variable, which must be a 8 bytes array. The hash of a block must now be computed as

```
SHA-256(block.data||block.previous_hash||block.p_o_w)
```

You must also augment the JSON encoding with a `p_o_w` field, that must contain the Proof-of-Work bytes as an hexadecimal string.

Test again your program, and think about what strategy should your miner use if they want to maximize the number of blocks they published on the longest chain.

(Optional) More Testing

In order to test your miner against another implementation, we provide a dockerized miner. This task is optional, as we have noticed that Docker does not handle parallel CPU-intensive tasks on some systems. It is nevertheless recommended to attempt it, as it can point out implementation mistakes, and test cases where other miners do not behave exactly as you would expect. You will use docker-compose to start a small system with 3 miners, one of which will be you own. Yout have to copy the following `docker-compose.yml` file at the same level as your `miner` folder.

```
version: '3.3'

services:
  miner_1:
    image: com402/hw9_miner
    container_name: miner-1
    hostname: miner.1
    command: "miner.1 miner.2,miner.s genesis"
    networks:
      bc_network:
        aliases:
        - miner.1
    logging:
      driver: none
  miner_2:
    image: com402/hw9_miner
    container_name: miner-2
    hostname: miner.2
    command: "miner.2 miner.1,miner.s"
    networks:
      bc_network:
        aliases:
          - miner.2
    logging:
      driver: none
  miner_s:
```

```
    hostname: miner.s
    container_name: miner-s
    build: .
    command: "miner.s miner.1,miner.2"
    networks:
      bc_network:
        aliases:
          - miner.s
    logging:
      driver: json-file

networks:
  bc_network:
    driver: bridge
```

In order for docker-compose to start your miner, it needs to be dockerized. Add the following Dockerfile to your working directory, at the same level as the `miner` folder.

```
FROM python:3.7-alpine

RUN pip3 install pptree

ADD miner /miner/
WORKDIR /miner

ENTRYPOINT ["python3", "miner.py"]
```

Don't forget to add `RUN pip3 install [lib]` lines to include all the additional requirements for your implementation in the docker container. For this test, the `TARGET` variable is `2**235`. Once everything is set, you can start the test by running:

```
docker-compose up --force-recreate --build
```

This will start the three miners and display the output of your implementation (you can ignore the warnings at the beginning). Again, try to optimize the strategy of your miner so that it produces as many blocks on the longest chain as possible. Also, try to make these strategies as resilient as possible against malicious nodes that could send invalid or *uninteresting* blocks. Indeed, these strategies should be realistic: they should not rely on the fact that you are running all the nodes on your own computer!