THE UNIVERSITY OF SYDNEY

Due: Friday, Week 11 by 23:59

COMP3221 Assignment 3: Blockchain

The goal of this project is to implement a peer-to-peer (P2P) blockchain system in Python.

1 Introduction

As seen during the lectures, a blockchain is a chain of blocks implemented in a distributed fashion. Each block stores a series of transactions just like the page of a ledger. The goal of blockchain is to track the ownership of digital assets. Blockchain finds applications in finance, health, supply chain, etc.

In this assignment, you will develop your own blockchain by applying the knowledge learned during the lectures. The nodes or *peers* developed by different students should be able to interact peer-to-peer (P2P) to form a blockchain network.

1.1 Learning Objectives

- Enhance the understanding of essential blockchain terms, characteristics and concepts.
- Understand how a P2P model works and how to design and implement a P2P system.
- Develop programming skills in the fields of blockchain and P2P.

2 Assignment Specifications

To implement a blockchain system one has to simply write the code of a single blockchain node. Spawning multiple of these nodes that connect with each other is sufficient to create a blockchain P2P network. The node must act as a "peer", behaving both as a "client", requesting services from other nodes, and a "server", offering services to other nodes. These nodes or peers request and offer services by sending network messages to each other.

2.1 Formats

Network messages have maximum length of 65535 bytes. Each message is prefixed with network-byte order encoded (big-endian) 2-byte length, followed by data.

The data in the messages is encoded in JSON format (cf. `https://www.json.org/json-en.html`). Message is a JSON object with two keys: "type" and "payload". Type can have two string values: "transaction" or "values".

A payload value for a transaction is a JSON object representing the transaction with the following keys: "sender", "message", "nonce", and "signature":

- The sender is a valid hex-encoded 32-byte ed25519 public key string. For more information about the ed25519 cryptographic scheme refer to http://ed25519.cr.yp.to/).
- The message is a string, which should contain at most 70 alphanumeric characters.
- The nonce is an integer representing the number of transactions that the sender already has in the blockchain at the moment of sending the transaction.
- The signature is a valid hex-encoded 64-byte ed25519 signature, corresponding to the public key of the sender and the current transaction.

A block is a JSON object with the following keys: "index", "transactions", "current_hash", and "previous_hash".

- The index represents the number of blocks in the blockchain, including itself.
- The value associated with the key "transactions" is a list containing the transaction objects described above.
- The current hash is the SHA-256 hash of the string representation of the block object including other three keys, with keys being sorted lexicographically.
- The previous hash corresponds to the value of the "current_hash" of the block with the previous index, with the exception of the genesis block.

For the message type "values", the value associated with the key "payload" is a list of block objects that were proposed in the current consensus round.

```
{
    "type": "transaction",
    "payload": {
        "sender": "a57819938feb51bb3f923496c9dacde3e9f667b214a0fb1653b6bfc0f185363b",
        "message": "hello",
        "nonce": 0,
        "signature": "142e395895e0bf4e4a3a7c3aabf2f59d80c517d24bb2d98a1a24384bc7cb29c9d593ce3063c5dd4f12ae9393f3345174485c052d0f5e87c082f286fd60c7fd0c"
    }
}
```

Figure 1: Example `transaction` request.

```
True
```

Figure 2: Example `transaction` response

```
{
    "type": "values",
    "payload": 2
}
```

Figure 3: Example **values** request

```
[
    {
        'index': 2,
        'transactions':
        [
            {
                'sender': 'a57819938feb51bb3f923496c9dacde3e9f667b214a0fb1653b6bfc0
                    f185363b',
                'message': 'hello',
                'nonce': 0,
                'signature': '142e395895e0bf4e4a3a7c3aabf2f59d80c517d24bb2d98a1a
                    24384bc7cb29c9d593ce3063c5dd4f12ae9393f3345174485c052d0f5e87c
                    082f286fd60c7fd0c'
            }
        ],
        'previous_hash': '03525042c7132a2ec3db14b7aa1db816e61f1311199ae2a31f3ad1c
            4312047d1',
        'current_hash': '5c0ada1107f87eee93b675cc9e7d772424013add94e202a8d578a16298
            c30c19'
    }
]
```

Figure 4: Example **values** response

2.2 Protocol

The peer operates a multithreaded TCP server on a specified port and receives two types of messages described in the previous part. A separate thread should be created for each connected client. Each node is connected to every other node through an independent long-live TCP connection and exchanges messages through these connections. But the node should handle possible connection drops with a small timeout as well. When a connection to a remote node is created, we open a socket without a timeout to allow the remote node to start. When we send a first request, we change the socket timeout parameter to 5 seconds. If the connection is broken or timed out, we retry to create a new socket 1 time, and if that fails, we consider the remote node as crashed, and don't try to reconnect anymore.

In addition to the TCP server, the peer should be running a pipeline thread as described below, where the consensus logic is executed. The thread should execute the loop with the following operations.

1. Wait for the transaction pool to be ready. Transaction pool can become ready on two events.

    - The first event is when the pool becomes non-empty, for example when a transaction is received.
    - The second event is when the node receives a request from another remote node asking for the value of the next round. This happens when the remote node has received a transaction and is ready to propose a non-empty block.

    Once its transaction pool becomes ready, a node should wait for 2.5 seconds before moving to operation (2) in order to potentially receive more transactions and batch them all altogether.

2. With the transactions received from the previous call, create a block proposal based on the current blockchain, containing the transactions, correct index and the previous block hash.

3. Start the consensus broadcast routine. As described in Lab 8, the node should execute the crash-fault tolerant synchronous consensus protocol. Timeout for values request is 5 seconds and the timer starts when the values are requested. The timer can be set in the socket **`settimeout`** call. If the socket is closed, or the timeout is reached, retry once. After one retry with the same timeout, the remote node is considered as crashed, and should not be contacted in the next rounds. Note that the consensus protocol must decide on a block value while the original consensus algorithm decides on the minimum of the received integer values. This is why this implementation should decide on the *minimum block* defined as the block with at least one transaction whose hash has the lowest lexicographical representation.

4. Once the block is decided, append the block to the blockchain and remove the included transaction from the transaction pool. Note that there could be conflicting transactions from the same user based on the transaction nonce, which could have been received during the round execution. The first of these transactions can be included but any following transactions that conflict with it should be removed from the block, since they became invalid and cannot be committed.

2.3 Transaction Validation

Upon reception of a transaction formatted as specified in Part 2.1, the peer must validate it according to the following rules:

- The transaction nonce is equal to the correct value based on the current state.
- There is no transaction from the same sender and the same nonce in the pool.

The nonce is monotonically increased for each transaction from the same user and starts with 0. The capacity of the transaction pool should be 3. If the pool size is less than the defined capacity, then a newly received transaction should be added to the pool and **`True`** JSON value should be returned to the client that sent the transaction. Otherwise, the content of the response should be **`False`**.

3 Program structure

You can use the skeleton code provided on Canvas as the base. All files should be located in the folder named “SID1_SID2_A2” with no subfolders. All python files should be correct and do not forget to remove any dummy files that do not count as source files. Please zip the SID1_SID2_A2 folder and submit the resulting archive SID1_SID2_A2.zip to Canvas by the deadline. The program should be compatible with Python 3.6.3, and should not contain platform-specific code.

3.1 Interface

As declared in the skeleton code, you should implement the following classes and methods.

`Blockchain` class with the following method.

- `set_on_new_block(self, on_new_block)` – sets a callback function that will be called when a new block is appended to the blockchain. The callback function should have one argument, which is the block dictionary. It will be called as `on_new_block(block)`.

`RemoteNode` class with the following methods.

- `__init__(self, host: str, port: int)` – creates a client for a remote node with the given host and port.
- `transaction(self, transaction: dict) -> bool` – blocking request to send transaction to the network. Should return `True` if the transaction was added to the pool, and `False` otherwise.

`make_transaction(message, private_key, nonce) -> dict` – create the transaction dictionary with the signature. The resulting dictionary should be passed to the `RemoteNode` `transaction` method.

`ServerRunner` class with the following methods.

- `__init__(self, host: str, port: int, f: int)` – creates an object with the given host, port, and maximum number of tolerated failures.
- `start(self)` – non-blocking function that starts the node, together with the threads and TCP server.
- `stop(self)` – blocking function that shuts down the node and waits until the threads have joined.
- `append(self, remote_node: RemoteNode)` – adds a connection to a `RemoteNode` instance. Used for communication between the nodes.
- `blockchain` – data attribute of type `Blockchain` representing the underlying data structure

3.2 Submission

The final version of your assignment should be submitted electronically via CANVAS by 23:59 on the Friday of Week 11. **Students will work individually or in groups of two members**.

- Week 8: release of the assignment description.
- Weeks 7-11: 4 labs to guide the in the tasks.
- End of week 11: assignment submission deadline.
- Lab of week 12: demo to assess the codebase.
- Any late submission implies penalty points (5% per day) after the submission deadline.

You are required to submit your source code and a short report to Canvas.

- Code (zip file includes all implementation, readme) **`SID1_SID2_A3.zip`**, where SIDx is the student ID.
- Code (including all implementation in one file exported in a txt file for Plagiarism checking) **`SID1_SID2_A3_Code.txt`**.
- Readme: Clearly state how to run your program.

Please note that you must upload your submission BEFORE the deadline. The CANVAS would continue accepting submissions after the due date. Late submissions would carry penalty per day with maximum of 5 days late submission allowed.

README.md text/mark-down file should indicate the OS and the version of Python you used to test your code successfully.

4 Academic Honesty / Plagiarism

By uploading your submission to CANVAS you implicitly agree to abide by the University policies regarding academic honesty, and in particular that all the work is original and not plagiarised from the work of others. If you believe that part of your submission is not your work you must bring this to the attention of your tutor or lecturer immediately. See the policy slides released in Week 1 for further details.

In assessing a piece of submitted work, the School of Computer Science may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program. A copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.

5 Marking

Marking scheme:

- -2% Arbitrary crash penalty
- 15% Step 1: node runs locally, receives a transaction, and prints it on stdout
- 30% Step 2: node receives enough transactions locally to create a block and prints it on stdout
  - 15% for printing a well formatted block
    - 5% correct list of transactions
    - 5% correct previous hash
    - 5% no conflicting transactions (e.g. same nonce)
  - 5% can include multiple transactions in a block
  - 10% for printing the correct current block hash
- 20% Step 3: node receives [10%] and sends [10%] a block before decision is reached
- 10% Step 4: node reaches a correct decision at the end of the consensus protocol execution
- 10% Step 5: node reaches a correct decision despite crashes
  - 5% one node crash tolerance
  - 5% two node crash tolerance
- 15% Step 6: invalid transactions are rejected
  - 5% wrong nonce is rejected
  - 5% wrong signature is rejected
  - 5% wrong field format is rejected