THE UNIVERSITY OF SYDNEY

Due: Friday, Week 11 by 23:59

COMP3221 Assignment 3: Blockchain

The goal of this assignment is to develop your own peer-to-peer (P2P) blockchain system in Python by applying the knowledge learned during the lectures.

1 Assignment Specifications

To implement a blockchain system, one has simply to 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 request and offer services by sending network messages to each other. *The nodes developed by different students should be able to interact peer-to-peer (P2P) to form a blockchain network.*

1.1 Formats

Network messages have a maximum length of 65535 bytes. Each message is prefixed with its length encoded over two bytes. The length of the message is encoded using network-byte order (i.e., big-endian). You can use the code in the supplied **`network.py`** file available on Canvas to send and receive messages in the correct format.

The content of a message is encoded in JSON format (see https://www.json.org/json-en.html): each message is a JSON object with two attributes: **`"type"`** and **`"payload"`**. The **`"type"`** attribute can take one of the two following values:

- **`"type": "transaction"`**, if the message is a transaction request,
- **`"type": "values"`**, if the message is a block request.

```
{
  "type": "transaction",
  "payload": {
    "sender": "a57819938feb51bb3f923496c9dacde3e9f667b214a0fb1653b6bfc0f185363b",
    "message": "hello",
    "nonce": 0,
    "signature": "142e395895e0bf4e4a3a7c3aabf2f59d80c517d24bb2d98a1a24384bc7cb29c9d593ce
    3063c5dd4f12ae9393f3345174485c052d0f5e87c082f286fd60c7fd0c"
  }
}
```

Figure 1: Sample transaction request.

Transaction Request. If the `"type"` attribute has the value `"transaction"`, then the `"payload"` value contains a JSON object representing a transaction (see Figure 1). A transaction has the following attributes: `"sender"`, `"message"`, `"nonce"`, and `"signature"`.

- `sender`. The sender is an hexadecimal string of 64 alphanumeric characters representing a valid 32-byte ed25519 public key. For more information about the ed25519 cryptographic scheme refer to http://ed25519.cr.yp.to/.
- `message`. The message is a string that contains at most 70 alphanumeric characters.
- `nonce`. The nonce is an integer representing the sequence number of the transaction. This sequence number is strictly monotonically increased and corresponds to the number of transactions issued by the same sender. Each sender starts with a sequence number equal to 0 and increments its sequence number before issuing a new transaction.
- `signature`. The signature is an hexadecimal string of 128 alphanumeric characters representing a valid 64-byte ed25519 signature for the transaction. The signature can be verified using the public key of the issuer of the transaction.

```
{
  "response": True
}
```

Figure 2: Sample transaction response.

Upon receiving a transaction request, a process validates the transaction (see Section 1.3) and responds with a transaction response (see Figure 2). The transaction response is a JSON object containing a unique boolean attribute `"response"` that takes the value `True` if the transaction validation is successful, `False` otherwise. If a process validates a transaction successfully, then it adds the transaction to its transaction pool.

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

Figure 3: Sample block request.

Block Request. If the `"type"` attribute has the value `"values"`, then the payload value is an integer corresponding to the index of the block for which the consensus process is currently being executed (see Figure 3). A block is a JSON object with the following attributes: `"index"`, `"transactions"`, `"current_hash"`, and `"previous_hash"` (see Figure 4).

- `index`. The index represents the index of the block in the blockchain. Indices start at 0 for the genesis block and are incremented by 1 for each block added to the blockchain.

- **`transactions`**. The value associated with the attribute **`"transactions"`** is a list of transaction objects as described above.
- **`current_hash`**. This value is the SHA-256 hash of the string representation of this block. The string representation of a block is a JSON string with the three block fields **`"index"`**, **`"transactions"`** and **`previous_hash`**, and where the keys are sorted lexicographically. This can be achieved by using the **`json.dumps`** method and passing **`True`** for the **`sort_keys`** parameter. An example of how to achieve this is presented in Figure 5.
- **`previous_hash`**. The previous hash corresponds to the value of the **`"current_hash"`** of the previous block in the blockchain (i.e., previous index), with the exception of the genesis block where the value is 32-byte representation of zero.

Upon receiving a block request, a node responds with a list of the block proposals that were proposed in the current consensus round.

```
[
  {
    "index": 2,
    "transactions":
    [
      {
        "sender": "a57819938feb51bb3f923496c9dacde3e9f667b214a0fb1653b6bfc0f185363b",
        "message": "hello",
        "nonce": 0,
        "signature": "142e395895e0bf4e4a3a7c3aabf2f59d80c517d24bb2d98a1a24384bc7cb29c9d
    593ce3063c5dd4f12ae9393f3345174485c052d0f5e87c082f286fd60c7fd0c"
      }
    ],
    "previous_hash": "03525042c7132a2ec3db14b7aa1db816e61f1311199ae2a31f3ad1c4312047d1",
    "current_hash": "5c0ada1107f87eee93b675cc9e7d772424013add94e202a8d578a16298c30c19"
  }
]
```

Figure 4: Sample block response.

```
import hashlib
import json

json_string = json.dumps({"index": block.index, "transactions": block.transactions, "
    previous_hash": block.previous_hash}, sort_keys=True)

block_hash = hashlib.sha256(json_string.encode("utf-8")).hexdigest()
```

Figure 5: Building the block hash of a block **`block`**.

1.2 Protocol

A node operates a multi-threaded TCP server listening on a specified port where it receives the two types of messages described in the previous part. A separate thread should be created to handle each connection. A node is connected to each other node through an independent long-lived TCP connection and exchanges messages with this node through this connection. A node should also handle possible connection drops.

- When a connection to a remote node is created, the node opens a socket without a timeout to allow the remote node to start.
- When a node starts the consensus broadcast routine, the socket timeout parameter is changed to 5 seconds. If the connection is broken or timeouts, the node tries to create a new socket 1 time, and if that fails, the remote node is considered crashed and is not contacted anymore.
- When the consensus broadcast routine is completed for a specific round, the timeout for the nodes considered alive should be reset (i.e. `settimeout(None)`) to allow idle between receiving transactions from the clients.

Each node takes as a parameter the path to a file containing the list of the other nodes. The format of the file containing the list of nodes is detailed in Appendix A. Blockchain should initially contain the genesis block defined in Appendix B.

In addition to the TCP server, the node runs a pipeline thread where the consensus logic for each index is executed. The pipeline thread runs a loop over the blockchain indices, and for each index of the blockchain the node executes the following steps:

1. Wait for the transaction pool to be ready. Transaction pool becomes ready when any of the two following events occur.
   - The pool becomes non-empty, for example when a transaction is received.
   - 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.
2. With the transactions collected during the previous step, create a *block proposal* containing a correct `index`, the `transactions`, the hash `previous_hash` of the previous block and the current hash `current_hash` as illustrated in the list of blocks of Figure 4. A block proposal can have zero transactions if the node did not receive any transactions but received a block proposal for the current index from another node.
3. Start the consensus broadcast routine. As described in Consensus tutorial, the node should execute the crash-fault tolerant synchronous consensus protocol. Nodes should request values from each other using block request as in Figure 3 where `"payload"` is the block index in the current consensus round. The response should contain a list of blocks corresponsing to the set of values seen by the node, formatted as in Figure 4. Request `"payload"` and the `"index"` in every block contained in the response should be the same.

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 connection is broken or timeouts, the node tries to create a new socket 1 time, and if that fails, the remote node is considered crashed and is not contacted anymore. Make sure to reset the timeout (**`settimeout(None)`**) for the nodes considered alive after the broadcast routine has completed.

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 from the local transaction pool all the transactions that are in the appended block.

1.3 Transaction Validation

Upon reception of a transaction formatted as specified in Section 1.1, the node must validate it (i.e., make sure the transaction is valid) according to the following rules:

- Transaction attributes are valid according to the transaction request specification described in Section 1.1;
- The transaction nonce is equal to the number of transactions issued previously by the sender;
- There is no transaction from the same sender and the same nonce in the pool.

If the transaction is valid then it is added to the local transaction pool and the node returns **`True`** to the client, otherwise the transaction is discarded and the node returns **`False`**.

1.4 Program Structure

The node program is named **`COMP3221_BlockchainNode.py`** and requires two command-line arguments for execution as follows.

```
python COMP3221_BlockchainNode.py <Port-Server> <Node-List>
```

- **<Port-Server>**: The port number on which the server listens for incoming connections from other nodes and clients.
- **<Node-List>**: A path to the list of the addresses of all nodes following the format described in Appendix A.

Example usage:

```
python COMP3221_BlockchainNode.py 8888 node-list.txt
```

2 Submission

All files should be located in the folder named **`SID1_SID2_A3`** (or **`SID1_A3`** if there is only one student in the group) with no subfolders, where SIDx is the your nine-digit Student Identification Number. 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_A3`** folder and submit the resulting archive **`SID1_SID2_A3.zip`** to Canvas by the deadline. The program should be compatible with Python 3.6.3, and should not contain platform-specific code.

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.** Note that if one student receives a special consideration, it will automatically be applied to all the members of the group.

You are required to submit your source code to Canvas.

- Code (zip file includes all implementation, readme) **`SID1_SID2_A3.zip`**.
- 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 website would continue accepting submissions after the due date.

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

3 Marking Scheme

Demo sequence. Your program will be assessed in person in week 12 during your lab. To this end, the tutor will spawn a *reference node* interacting with the blockchain network, of which your *student node* will be part. Arrive on time in your lab and follow these steps:

1. connect your usual computer to the university network and check its IP address;
2. within the first 5 minutes of the lab, write the IP address and port number your student node will use for the demo on the whiteboard;
3. 5 minutes into the lab, create a file with the IP addresses on the whiteboard following the format described in Appendix A;
4. re-download your submitted node from Canvas and start it with the arguments described in Section 1.4;
5. follow the instructions of your tutor.

Assessment criteria. Assuming that full marks is 100 points (pts) for this assignment, the marking scheme used during the demo is as follows:

Note that the **`current_hash`** and message payload are defined in Section 1.1, and block proposal is defined in Section 1.2.

- **Receive transactions**: 40pts

10pts If the student node receives either a valid or invalid transaction, then it displays:

`[NET] Received a transaction from node {node_ip}: {message payload}`

5pts If the student node receives a transaction with a wrong sender, then it discards it and displays:

`[TX] Received an invalid transaction, wrong sender – {message payload}`

5pts If the student node receives a transaction with a wrong message, then it discards it and displays:

`[TX] Received an invalid transaction, wrong message – {message payload}`

5pts If the student node receives a transaction with a wrong nonce, then it discards it and displays:

`[TX] Received an invalid transaction, wrong nonce – {message payload}`

5pts If the student node receives a transaction with a wrong signature, then it discards it and displays:

`[TX] Received an invalid transaction, wrong signature mssage – {message payload}`

10pts If the student node receives a valid transaction, then it adds it to its transaction pool and displays:

`[MEM] Stored transaction in the transaction pool: {signature of the transaction}`

- **Proposal creation**: 20pts

10pts If the student node adds a first transaction to its transaction pool, then it creates a valid block proposal and displays:

`[PROPOSAL] Created a block proposal: {block proposal}`

5pts If the student node adds a second transaction to its transaction pool, then it creates a valid block proposal and displays:

`[PROPOSAL] Created a block proposal: {block proposal}`

5pts When the student node adds a third transaction to its transaction pool it creates a valid block proposal and displays:

`[PROPOSAL] Created a block proposal: {block proposal}`

- **Consensus**: 20pts

10pts If the student node creates a block proposal, then the reference node receives a block request as a message from the student node and displays:

```
[BLOCK] Received a block request from node {node_ip}: {message payload}
```

10pts If the student node reaches a consensus decision on a block, the student node appends the block to the blockchain and displays:

```
[CONSENSUS] Appended to the blockchain: {current_hash of the block}
```

- **Fault-tolerance**: 20pts

10pts Even after the reference node has crashed, upon reception of a valid transaction, the student node adds the transaction to its transaction pool, creates a valid block proposal and displays:

```
[PROPOSAL] Created a block proposal: {block proposal}
```

10pts Even after the reference node has crashed, upon consensus termination the student node still appends blocks to the blockchain and displays:

```
[CONSENSUS] Appended to the blockchain: {current_hash of the block}
```

- **Late submission**: -5pts per late day (no submission will be accepted after 5 late days).
- **Arbitrary crash penalty**: -2pts each time a crash happens

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.

A Node Discovery

This section details the format of the file used for node discovery. Each node will take as argument the path to a file containing the list of nodes. The file will contain one node per line and each line consists of the IP and the listening port of the node separated by a colon. A sample file is shown in Figure 6.

```
192.168.1.43:8888
192.168.1.56:8888
192.168.1.58:8888
192.168.1.16:8888
```

Figure 6: Sample file with a list of nodes.

B Genesis Block

```
{
  "index": 1,
  "transactions": [],
  "previous_hash": "0000000000000000000000000000000000000000000000000000000000000000",
  "current_hash": "03525042c7132a2ec3db14b7aa1db816e61f1311199ae2a31f3ad1c4312047d1"
}
```

Figure 7: Genesis block.