Toby, Owen (500262636, 500550430)

COMP3221 Blockchain Report

Our P2P Blockchain system consists of 3 peers, interconnected with each other in the network, with each Peer consisting of a Server, a Client, and a Miner. The blockchain network is constructed using python and connected via python's *socket* library.

Blockchain Data

The blockchain data in each Peer is stored within a blockchain class object. The object keeps track of:

1. Blockchain: chain of blocks
2. Current Transaction Pool
3. Limit to transactions per block

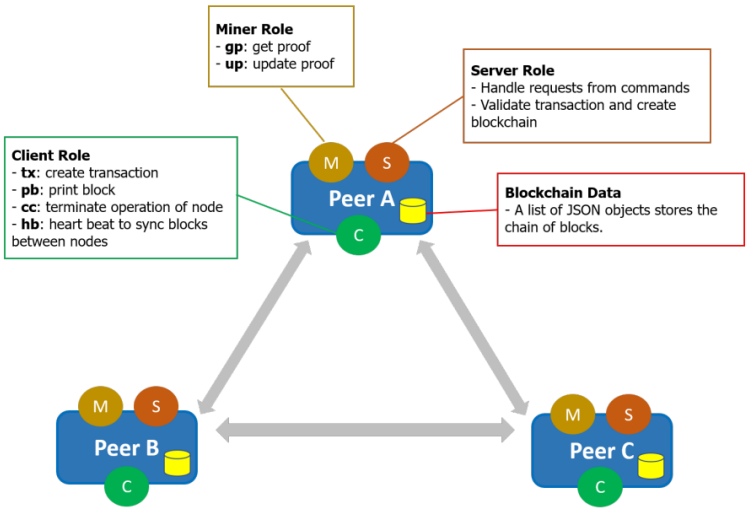

Figure 2: Peer-to-Peer Blockchain Network

The class will also include the template for blocks in the chain with parameters of:

1. Index : *int*
2. Timestamp : *datetime*
3. Proof : *int*
4. Previous Hash : *SHA256 Hash*
5. Current Hash : *SHA256 Hash*

It also contains various utility functions to interact with the block chain such as the creation of a *NewBlock()*, *addTransaction()* or calculating the *currentHash()* and *previousHash()*.

Blockchain: SERVER

Each Blockchain Peer has a server that is responsible for listening on a specific port to handle incoming requests created by other client peers. The server utilizes the python library *socket* to listen for incoming connections. The library *pickle* is also used to transmit encoded python object data through the socket, in which the server will receive and decode the transmission. The transmission is formatted to be classified by the first 2 characters; being the OP codes. The server can handle:

- **tx**: First validating the format of the transaction, then adding it to the Blockchain's pool awaiting to be processed in a block.
- **pb**: The server will return the blockchain data of the blocks back to the requesting client in the form of JSON.
- **cc**: The server will terminate the connection with the client through *socket.close()* and exiting the loop.
- **gp**: The server will send the current proof of the last block in the chain to the requesting miner through the *blockchain.lastBlock()["proof"]*.
- **up**: After receiving proof from a miner, the server will check if the proof is valid (*proof[:2] == "00"*), then process the transactions into a block, notifying all other peers in the network about the new block through the *hb* request from other peer clients.
- **hb**: The server will return the current blockchain and current pool to the requesting client, allowing other peers to assume synchronous amongst peers every 5 seconds.

The server handles multiple clients by creating a new thread for each connecting client, using the python library *_thread*. As the server reads and writes information to the shared blockchain data store, mutex locks are used to prevent race conditions and errors in reading and writing, thus achieving thread safety.

Blockchain: MINER

The miners role in each blockchain peer is to solve the proof of work algorithm, thus providing proof of work and validation for the creation of a new block to be added to the blockchain. The PoW algorithm is characterized by being hard to find a proof, however, is easy to check if the proof is valid. The PoW algorithm implemented is:

*calculateHash(newProof ** 2 - previousProof ** 2)[:2] == "00"*

Once a proof is calculated, it will collate 5 transactions in a block and send the block to other peers every 5 seconds. The miner is connected to the peer via a socket, thus being able to communicate with the peer through commands such as:

- **gp**: where the miner will receive the latest proof from the server.
- **up:** where the miner will post the newly calculated proof to the server for validation and creation of a new block.

Our miner is deployed on a separate thread thus the server can listen to and be connected to multiple miners. Our miner will also poll the server every second through the socket connection to check if a new proof is needed.

Blockchain: Client

The Clients main role in each Blockchain is to take input from the user and transfer those commands to the server to perform respective tasks. These commands include ***tx, pb, cc***. Our implementation also uses the client to broadcast the **hb** message to all other neighbors, to keep the blockchain network synchronous. The comparison function we use is influenced from bitcoin, where the longest branch is kept, while shorter ones are pruned. Furthermore, we enforce a timestamp at every transaction to ensure that transactions are not double spent, comparing the timestamp of transactions to new blocks as well.

Problems in development:

Connection of Sockets

Our original implementation did not have the connection of sockets wrapped in a while loop, and as *socket.connect()* is a blocking function, the first Peer to be launched will not be able to connect to other servers, as the other Peer servers aren't initialized. This issue was due to the nature of the *socket.connect()* function as well as race conditions between different Peers. Originally we dealt with this problem by enforcing a *time.sleep(5)* on all clients, such that other servers can be initialized before any clients connect. However, soon we figured that if we wrapped the connect in a while loop to constant attempt to connect, we can connect to the listening port, even if the server starts up later than the client.

Closing of Threads

We found trouble in the closure of all of our threads generated by each peer as many of the threads did not have an active socket connection to the server, thus it was difficult to pass information to them without the use of SIGNALS. However, we found that if we utilize the *os* library, we can initalise a manual shutdown of the process from the parent thread - Peer, in which the python environment will handle the clean up of children threads for us. This was useful in the implementation of **cc** to close all connections from the given Peer.