COMP3221 Assignment2: A P2P Blockchain System in Python

1 Brief introduction of our system

Our system is a decentralized peer-to-peer blockchain network that relies on proof of work algorithms to reach consensus. In this network, each node (peer) is connected to all other nodes, and users can access any node through the client and perform operations such as adding transactions or querying the blockchain.

The main components of a node are client, server, miner and blockchain. The client provides a command-line interface for users to add transactions or query the blockchain; the server receives requests from clients and miners and returns blockchain-related data, or adds blocks/transactions; the miner is responsible for checking the blockchain every second whether it is updated, if so, call the proof of work algorithm to calculate a new proof and send the proof to the server. In addition, the main thread of the peer also sends a heartbeat message to all other peer servers every 5 seconds to obtain the blockchains of other peers, and will automatically set its own blockchain to the longest of them all (If all blockchains are the same length then choose the one with the earliest last block created).

2 Network Typology

In general, peer-to-peer is the most notable feature of the blockchain network we have developed. Every peer in the network is connected to each other, forming a fully connected graph. "Connected" here means each peer and its client is able to send messages to the servers of all other peers and receive response from them. Thus, it is sufficient to say that the network topology of our system is mesh topology.

3 Techniques and Methodology

- **Client-server architecture:** For each peer, it follows the relationship of so-called client-server architecture, where the client primarily serves as the user interface and pass user input as command for the server to handle. In this context, the client is just a command-line interface that is purely responsible for receiving and passing user requests, it does not access the peer's blockchain, nor does it touch the peer's internal logic.
- **Multi-threading with socket:** Socket is used to set up a communication channel between the message sender and the message receiver. To ensure multiple threads can be executed simultaneously with no race condition, lock is utilised to limit the resource access to a particular thread in a single process (a peer process). Specifically, when a peer joins the network, three sub-threads will be generated, each of which corresponds to the client, miner and server. They will establish socket connections with different objects and pass requests and data as required. For example, when the server thread continues to listen for requests sent to its own peer's port, the miner thread will be sending gp requests to the server every second, or calculating proofs. They don't interfere with each other.
- **Proof of work:** This method allows each peer in the blockchain network to reach consensus. In our case, such consensus is reached with the proof determined when the hexadecimal hash has '00' as prefix. Each peer can generate a new block and broadcast the block to other peers only after calculating the correct proof of the next block, and when different peers compete for computing power on the proof of the block in the same location , in the end, only the earliest calculated result and generated block will be accepted by all peers.

4 Implementation

- **Transaction.py:** This program contains the construction of the **Blockchain** class. It contains methods to create new block, get the last block, calculate and get hash, and add new transactions to the last block.
- **Blockchain.py:** This program contains a method **validateTransaction** checking the validation of a new transaction, which will be used by a **BlockchainServer** object.
- **BlockchainPeer.py:** This program contains the construction of the **BlockchainPeer** class. When running this program, the peer id, port and config file will be given as command line arguments in that order. The program will record those and put add neighbor peer ids (as keys) and their corresponding ports (as values) to a dictionary **id2port**, which is further sent to each role of this peer: client, miner and server. All peer ids and port information is passed as the first argument to each role from **BlockchainPeer**. The first id-to-port pair corresponds to each role's id and port. The program then create and run a BlockchainPeer object. When creating a peer, it will create three objects: **BlockchainServer, BlockchainClient** and **BlockchainMiner** with the id-to-port dictionary passed to them. When running the peer, these three roles will be run and **hb** will be sent to all neighbors via socket every 5 second.

- **BlockchainClient.py:** This program contains the construction of the **BlockchainClient** class. When a client starts, it will connect to its corresponding ip (localhost) and port as those are passed as the first key-value pair in **id-to-port** by a **BlockchainPeer** object. When the connection is established, it will enter an infinite loop continuously asking user input from the terminal until it receives 'cc', which indicates connection termination. In each iteration, the client will say **"Please enter your command"** to start receiving user input. The client will firstly check the validation of the command, where the valid ones are a)tx, b)pb and c)cc. When the command is a)tx, the client will firstly check the length of the content, which should be of the format of tx [sender] [content]. If the message has three parts, the client will further send such new transaction to servers of the neighbouring ports. When the command is b)pb or c)cc, the client will send it to the server at the same port. A b)pb command should cause the server to send the blockchain back to the client in the format of a string, which will be print to the terminal by the client. A c)cc command will further break the infinite loop and close the client connection.

- **BlockchainMiner.py:** This program contains the construction of the **BlockchainMiner** class. When a miner runs, it will connect to localhost with the port sent by **BlockchainPeer**. A "gp" command will be sent to such location per one second requesting its server to send blockchain back to it. Once the miner receives the proof of the last block, it will compare it with **previous_proof**, a variable storing the latest proof the miner has found before. If two proofs are different meaning that new block has been added, the miner will call a **proof_of_work** method continuously incrementing a **new_proof** until the first two characters of the hash is **00**. The algorithm will return **new_proof**, which will be updated as the current **previous_proof** of the miner. Then the miner will send the "up" command with **new_proof** to the server.

- **BlockchainServer.py:** This program contains the construction of the **BlockchainServer** class. When a server runs, it will create a socket on localhost with port passed by the peer. The program will enter an infinite loop until "cc" from the client is received. A server will receive messages sending to the exact location and call the **serverHandler** method. **serverHandler** continously receives messages sent to the server (from all clients and its corresponding miner). a) If the command is "tx", the server will call the *validateTransaction* method of a Transaction object to validate the received transaction. b) If the command is "pb", it will send the entire blockchain object back to the sender as a string. c) If the command is "cc", it will break the inner loop in **serverHandler** and close the client-server connection, where the outer infinite loop will be closed furthermore wit the socket object terminated. c) When the command is "gp", the server will send back the proof of the last block. d) When the command is "up", the server will check whether the hashed value calculated using the proof from the sender and that of the last block has **"00"** at the prefix. e) If the command is "hb", the server will return the blockchain object as a string. If the last block has more than 4 transactions, the server will create a new block for its **Blockchain** object.

5 Problems encountered and solutions

- **How to ensure each role of the peer knows all other peers' ids and ports?** We solve this by passing a id-to-port dictionary read from the config file from the peer to every role of it.

- **How to ensure that all clients see the same results without race condition?** Lock (mutex) is used in all client, server and miner to enforce synchronization that no other thread can access resources when the current thread has not finished. Besides, to ensure every peer knows the same blockchain (especially the same hash), if the client receives blockchains both of the same longest length, but with different timestamp, it will pick the one with the earliest timestamp.

- **How to close all threads for a peer when the client receives 'cc' command?** When client receives 'cc', it will send it to its corresponding server, where the server further end its infinite loop for receiving and sending messages and close its socket connection. The client will break its infinite loop and close the socket connection. Peer will constantly check whether client is alive, if it is not, this peer will end, which will terminate the miner. All other peers will not receive and send to such dead peer.

6 Remark

Due to space limitations, this report does not include simulation results of each requirement, but the README.md of the project provides the detailed usage of the system, and any user can actually use the system to confirm whether each function is running correctly.