CS305 2025 Spring Project

Blockchain Network Simulation

1. Introduction

A blockchain network is formed by a set of blockchain peers (*simply* peers) who are responsible for verifying users' transactions in the network and packaging valid transactions in the form of blocks. Then, each block is verified by all peers and linked to the blockchain (i.e., the chain of valid blocks) if the majority of peers accept it. In this process, each peer decides about the validity of transactions and blocks independently without any centralised entity. Moreover, each peer stores a copy of the blockchain locally in its host. The simple operation procedure of a blockchain system is as follows:

1) Users generate new transactions and submit them to peers in the blockchain system.
2) Peers receiving the transactions verify the transactions' validity. If the transactions are valid, peers add them to their local transaction pool and broadcast them to other peers for verification.
3) In each block period, one of the peers is selected as a block generator to package transactions in its pool in the form of a block, which is broadcast to all peers in the network for verification.
4) If the majority of peers think that the block is valid and accept it, each peer appends the block to the blockchain. Note that each peer stores a copy of the blockchain locally.

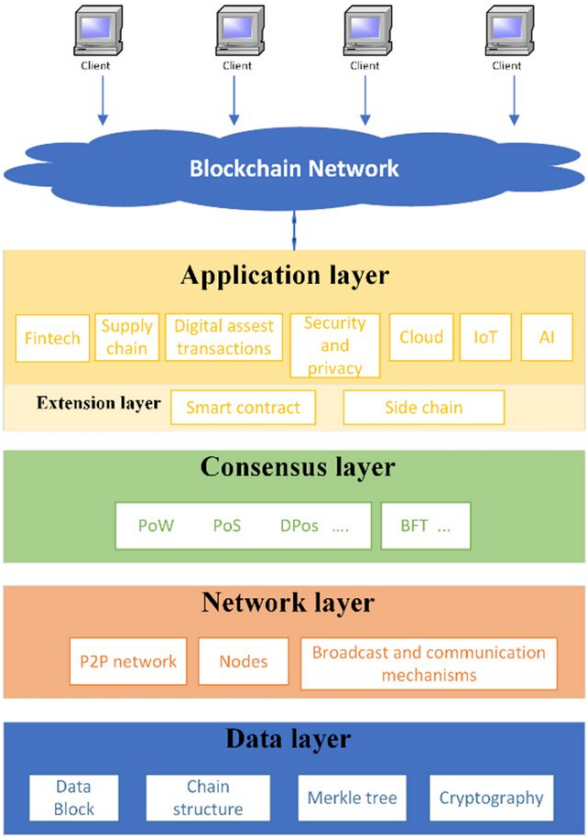

Figure 1: A general blockchain architecture [1].

As shown in Figure 1, a blockchain system can be roughly divided into four layers as follows (from bottom to top):

- Data Layer: Define the data structure of transactions, blocks, the blockchain (i.e., the chain of blocks), and the techniques (e.g., cryptography) to ensure data security.

- Network Layer: Define the network of blockchain peers (*simply* peers) and how they communicate to discover each other, exchange data, and verify the validity of transmitted data.
- Consensus Layer: Define the consensus mechanism for selecting block generators and verifying blocks among peers to reach a consensus about newly generated blocks.
- Application Layer: Provide users with dApps, such as cryptocurrency exchange, data sharing, etc.

This project focuses on simulating the network layer, which can help you understand how peers in a blockchain network interact. The following section will detail the network layer in a blockchain system.

2. Functionality of the Network Layer

This section will introduce the functionalities in a blockchain system's network layer, as shown in Figure 2. The functionalities of a peer can be roughly split into six parts as follows:

Part 0: Peer Initialisation

- Configure its IP address, port, and the number of target peers while gossiping messages.
- Decide whether to act as a normal/malicious, lightweight/full, NATed or non-NATed peer.
- Create a TCP socket to receive incoming messages.

Tips:

1）When broadcasting blocks and transactions, each peer sends them to a random subset of its known peers instead of all known peers to reduce redundant messages in the network.

2）If the peer acts as a normal peer, it always generates correct transactions and blocks. Instead, a malicious peer can generate incorrect transactions and blocks (e.g., with the wrong block ID).

3）If the peer acts as a full peer, it always engages in block generation and verification and store all blocks locally. Instead, a lightweight peer never generates and verifies block and only stores the header of blocks locally.

4）This project considers network address translation (NAT). A NATed peer is generally located in a local network and cannot interact directly with peers outside the local network. Instead, non-NATed peers in the local network act as NAT routers or relaying peers between NATed peers and peers outside the local network. Typically, while forwarding external messages to a peer in a local network, a relaying peer must find the destination peer's IP address in the local network based on the NAT translation table. Here, to reduce the complexity, we only simulate the logic of NAT and ignore the NAT translation table; that is, a NATed peer has only one IP address across the network.

Part 1: Peer Discovery

The peer discovers peers existing in the network and periodically checks if they are alive. The peer must know some peers running in the network before joining it. The procedure of peer discovery is as follows:

- Say “hello” to its known peers while joining the network.
- Add a new peer to the list of known peers when receiving a “hello” message.
- Send a “ping” message to all known peers periodically.
- Update the state of known peers while receiving “pong” messages.
- Remove unresponsive peers if no “pong” messages are received before the timeout.

Part 2: Block and Transaction Generation and Verification

Since block generator selection and block verification are implemented in the consensus layer, we simplify these two functions in this project. In detail, each full peer can generate transactions with random senders and receivers, and packages received and generated transactions into a new block periodically. A new transaction or block is valid if its ID is correct. Moreover, a new peer must obtain the latest blockchain from known peers before generating transactions and blocks. The procedure of the transaction and block generation, and verification is as follows:

- Synchronize the latest blockchain from known peers while joining the network.
- Start generating transactions with random senders and receivers.
- Broadcast the transactions to known peers for verification.
- Add the valid transactions to the local transaction pool.
- Package the transactions in the local pool into a new block.
- Broadcast the block to known peers for verification.
- Add the valid block to the local blockchain.

Tips:

1) When a peer sends a block to another, the sender usually sends an “INV” message with the block’s metadata instead of the block itself. If the receiver finds that it has not yet received the block, the receiver will reply with a “GETDATA” message to request the block. This can reduce the network overhead.
2) When receiving a “GETDATA” message, a peer replies with the block if the sender is a full peer; otherwise, a peer replies with the block's header because lightweight peers only store the header of blocks.

Part 3: Sending Messages

To maintain the fairness of sending messages and control the peer’s sending capacities, all messages are put into an outbox queue before being sent out. Then, the peer sends the messages in the queue one by one.

- Add messages to an outbox queue each time sending messages.
- Read a message from the queue based on their priorities.
- Find the best relaying peer if the destination of a message is a NATed peer.
- Send the message to the relaying node or the destination directly.

Part 4: Receiving Messages

The peer dispatches and processes received messages based on the message types.

- Check whether the message sender is banned. Drop the message if the sender is banned.
- Check whether the number of messages sent by the sender is within the limit. Drop the message if the sender sends messages too frequently.
- Check the types of messages and process the messages according to their type:
  - Msg.type=TX,
    - ✧ Check the validity of the transactions.
    - ✧ Check whether the transactions have been received.
    - ✧ Record the count of redundant transactions if they have been received.
    - ✧ Add the new transactions to the local pool if they have not been received.
    - ✧ Broadcast the new transactions to known peers.
  - Msg.type=BLOCK,
    - ✧ Check the validity of the blocks.
    - ✧ Check whether the blocks have been received.
    - ✧ Record the count of redundant blocks if they have been received.
    - ✧ Add the new blocks to the list of orphaned blocks if they have not been received, but the parents of the new blocks do not exist.
    - ✧ Add the new blocks to the local blockchain if they have not been received, and the parents of the new blocks exist.
    - ✧ Check whether the new blocks are the parents of orphaned blocks.
    - ✧ Broadcast the new blocks to known peers.
  - Msg.type=INV,
    - ✧ Check whether the blocks in the INV message have been received.
    - ✧ Request missing blocks from the message sender.
  - Msg.type=GETDATA,
    - ✧ Check whether the blocks requested are in the local blockchain.
    - ✧ Check whether the message requester is a full or lightweight peer.
    - ✧ Reply to the full message sender with the requested blocks or the lightweight message sender with the header of the requested blocks if the blocks are stored locally.
    - ✧ Request the blocks from known peers if the blocks are not stored locally.
  - Msg.type=GET_BLOCK_HEADERS
    - ✧ Reply with the header of blocks in the local blockchain.
  - Msg.type=BLOCK_HEADERS
    - ✧ Check the validity of the list of block headers by checking whether the parent of each block exists in the blockchain.

- ✧ Request the missing block from known peers if the peer is a full peer.

Part 6: Start Dashboard

Start a dashboard server to display the following message:

- Localhost: port/peers: display the set of known peers.
- Localhost: port/status: display the status of peers in the network.
- Localhost: port/transactions: display the transactions in the local pool.
- Localhost: port/blocks: display the blocks in the local blockchain.
- Localhost: port/orphan: display the orphaned blocks.
- Localhost: port/latency: display the transmission latency between peers.
- Localhost: port/capacity: display the sending capacity of the peers.
- Localhost: port/redundancy: display the number of redundant blocks and transactions received.
- Localhost: port/scores: display peers' scores in the network.

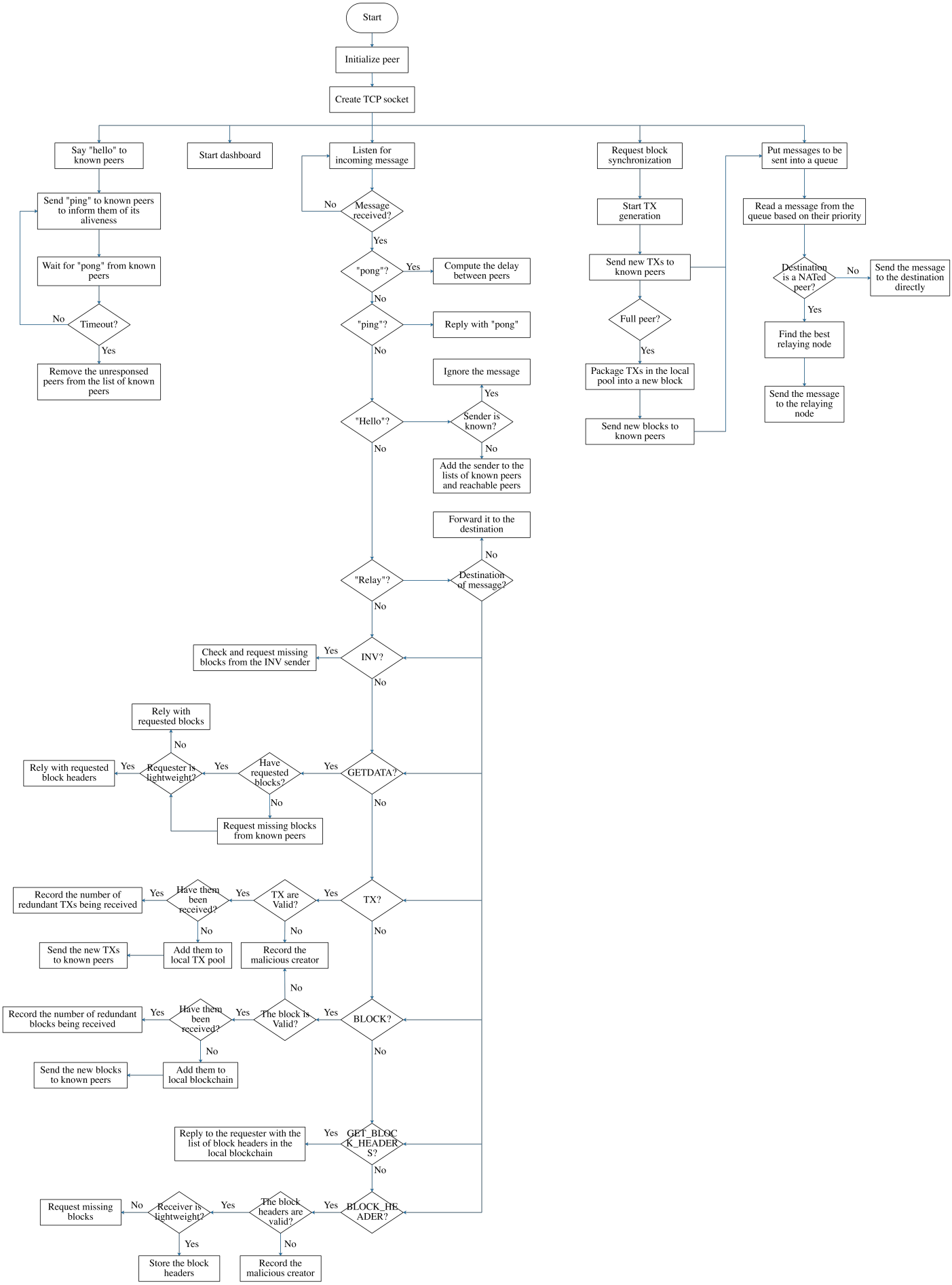

Figure 2: Operation logic of the blockchain network layer

3. Functions to Complete

The operation logic of the project is given in the main function of node.py. In the following, the functions to be completed in each part are explained as follows.

Part 0: Peer Initialization

1) **Start_socket_server (socket_server.py)**
   - Create a TCP socket and bind it to the peer's IP address and port.
   - Start listening on the socket for receiving incoming messages.
2) **Listen_on_port (node.py)**
   - Pass the received data to the function **dispatch_message** for message processing.

Part 1: Peer Discovery

1) **Start_peer_discovery (peer_discovery.py)**
   - Define the JSON format of a "hello" message, which should include: {message type, sender's alias, IP address, port, flags, and message ID}. A sender's alias can be "peer_port". The flags should indicate whether the peer is NATed or non-NATed, and full or lightweight. The message ID can be a random number.
   - Send a "hello" message to all known peers and put the messages into the outbox queue.
2) **Handle_hello_message (peer_discovery.py)**
   - Process received "hello" message.
   - If the sender is unknown, add it to the list of known peers and record their flags.
   - Update the set of reachable peers. Each peer can only receive "hello" messages from reachable peers and never forward "hello" messages. If a peer receives "hello" messages from a NATed peer, it can act as the relaying peers of the NATed peer.

Part 2: Block and Transaction Generation and Verification

1) **Request_block_sync (node.py)**
   - Define the JSON structure of a "GET_BLOCK_HEADERS" message, which should include: {message type, from_id}. from_id is the sender's alias.
   - Send a "GET_BLOCK_HEADERS" message to all known peers and put the messages into the outbox queue.
2) **Start_transaction_broadcast (node.py)**
   - Generate the set of transactions with random senders and receivers (selected from known peers). The structure of a transaction is defined in TransactionMessage in transaction_message.py.
   - Broadcast the transactions to known peers based on gossip and put the message into the outbox queue.

3) **Start_block_generation (node.py)**
   - Exploit the function **create_dummy_block** in block_handler.py to generate blocks periodically.
   - Create an "INV" message for the new blocks.
   - Send the "INV" message to known peers and put the messages into the outbox queue.

4) **Create_dummy_block (block_handler.py)**
   - Define the JSON format of a block, which should include {type: Block, Block Generator's ID, Timestamp, Block Height, Block ID, ID of Block's Parent, and Transactions}. The block ID is the hash value of block structure except for the item of the block ID. A new block's parent is the last block in the blockchain, to which the new block will be linked. The height of a block is the height of its parent plus one. If the block generator is malicious, it can generate random block ID.
   - Read the transactions in the local pool using the function **get_recent_transactions** and create a new block.
   - Clear the local transaction pool and add the new block into the local blockchain in receive_block.

5) **Add_transaction (tx_pool.py)**
   - Add new transactions to the local pool tx_pool.

6) **Get_recent_transactions** (tx_pool.py)
   - Read the transactions in the local pool.

7) **Clear_pool (tx_pool.py)**
   - Remove transactions in the pool after packaging them into new blocks.

8) **Create_inv (network/inv_message.py)**
   - Create the JSON format of an "INV" message, which should include: {message type, sender's ID, blocks' ID, and message ID}.

Part 3: Sending Messages

1) **Enqueue_message (outbox.py)**

This function put all sending messages into an outbox queue. The procedure of the function is as follows:
   - Check if the sender sends message to the receiver too frequently using the function **Is_rate_limited**. If yes, drop the message.
   - Check if the receiver is banned using the function **Is_banned**.
   - Classify the priority of the sending messages based on the message type using the function **Classify_priority**.

- Add the message to the queue if the length of the queue is within the limit QUEUE_LIMIT, or otherwise, drop the message.

2) **Is_rate_limited (outbox.py)**
   - Check how many messages were sent from the peer to a target peer during the TIME_WINDOW that ends now.
   - Check if the sending frequency exceeds the sending rate limit RATE_LIMIT.
   - Record the current sending time into peer_send_timestamps.

3) **Is_banned (peer_manager.py)**
   - Check if a peer is in the blacklist.

4) **Record_offence (peer_manager.py)**
   - Record the offence when detecting malicious behavior of a peer.
   - Ban the peer using the function **Ban_peer** if the number of offences exceeds the limit OFFENCE_LIMIT.

5) **Ban_peer (peer_manager.py)**
   - Add a peer to the blacklist.

6) **Send_from_queue (outbox.py)**
   - Read the message in the queue. Each time, read a target peer's message with the highest priority. Then, move to the next target peer's message. This can ensure the fairness of sending messages to different target peers.
   - Check sending the message to the target peer directly or through a relaying node using the function **relay_or_direct_send**.
   - Retry a message if it is sent unsuccessfully and drop the message if the number of retry exceed the limit MAX_RETRIES.

7) **Classify_priority (outbox.py)**
   - Classify the priority of a message based on the message type.

8) **Get_outbox_status (outbox.py)**
   - Read the sending message in the queues.

9) **Relay_or_direct_send (network/link_simulator.py)**

- Check if the target peer is NATed. If yes, use the function **get_relay_peer** to find the best relaying peer.
- Send the message to the target peer or relaying peer using the function **send_message**.

10) Get_relay_peer (network/link_simulator.py)

- Read the latency between the sender and other peers in rtt_tracker in peer_manager.py.
- Find the set of relay candidates reachable from the target peer.
- Select the best relaying peers with the smallest latency.

11) Send_message (node.py)

- Send the message to the target peer. Wrap the function **send_message** with the dynamic network condition in the function **apply_network_condition**.