CS305 2025 Spring Final Project - Blockchain Network Simulation Report

> This project is hosted on GitHub at:
> https://github.com/OptimistiCompound/SUSTech_CS305_BlockChain

Group Members: Zhong Yong, Chen Jialin, Ni Huizhi

Responsibilities

	Work Items
Zhong Yong	socket_server.py, peer_discovery.py, outbox.py, message_handler.py
Chen Jialin	peer_manager.py, block_handler.py, inv_message.py, transaction.py, dashboard.py
Ni Huizhi	

Module Implementations

Part 1: Peer Initialization (socket_server.py)

Creates a TCP socket, binds to IP and port, and starts listening:

- Whenever a message is received, it uses json.loads to parse the message and calls dispatch_message to handle distribution.
- If an empty message or invalid JSON is received, it is skipped or logged to ensure stable service.

```
RECV_BUFFER = 4096

def start_socket_server(self_id, self_ip, port):

    def listen_loop():
        # Create a TCP socket and bind it to the peer's IP address and port.
        peer_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        peer_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        peer_socket.bind((self_ip, port))
        peer_socket.listen()
        print(f"Listening on {self_ip}:{port}")
        # When receiving messages, pass the messages to the function
`dispatch_message` in `message_handler.py`.
        while True:
            try:
                conn, addr = peer_socket.accept()
                conn.settimeout(10)  # Prevent waiting indefinitely
                with conn:  # Use 'with' to ensure the connection is properly
closed
                    try:
                        # Use a file object to read line by line
```

```
                        f = conn.makefile()
                        for line in f:
                            line = line.strip()
                            if not line:
                                continue
                            try:
                                msg_dict = json.loads(line)
                                dispatch_message(msg_dict, self_id, self_ip)
                            except json.JSONDecodeError:
                                print(f"Received invalid JSON data from {addr}:
{line}")
                    except Exception as e:
                        print(f"❌ Error receiving message: {e} in peer {self_id}
at {self_ip}:{port}")
            except Exception as e:
                print(f" ▽  Error accepting connection: {e} in peer {self_id} at
{self_ip}:{port}")
                continue

    # ☑ Run listener in background
    threading.Thread(target=listen_loop, daemon=True).start()
```

Part 2: Peer Discovery

peer_discovery.py

When a peer is initialized, start_peer_discovery is invoked to begin peer discovery. This function creates a reachable_peer_list based on peer_config:

- By using nested loops, it calculates each peer's reachable_by set, indicating which peers can reach that node (considering NAT and LAN constraints).
- Non-NAT nodes can be reached by all peers. NAT nodes can only be reached by peers in the same LAN. Once the global reachable_peer_list is created, a thread is started to call discover_peers every DISCOVERY_INTERVAL seconds. discover_peers iterates over reachable_peer_list and sends HELLO messages to each peer, then waits for responses. When a response is received, handle_discover_response is called:
- If the sender is not in known_peer, the node is added to known_peer.
- If the sender is not in reachable_by[self_id], the sender is added to reachable_by[self_id].

```
def start_peer_discovery(self_id, self_info):
    ...
    ...
        for peer_id in peer_config:
            reachable_by[peer_id] = set()

        for target_id, target_info in peer_config.items():
            for candidate_id, candidate_info in peer_config.items():
                if candidate_id == target_id:
                    continue
                target_nat = target_info.get("nat", False)
```

```
                candidate_nat = candidate_info.get("nat", False)
                target_localnet = target_info.get("localnetworkid", -1)
                candidate_localnet = candidate_info.get("localnetworkid", -1)
                # Non-NAT peer
                if not target_nat:
                    if not candidate_nat:
                        reachable_by[target_id].add(candidate_id)
                    else:
                        if target_localnet == candidate_localnet:
                            reachable_by[target_id].add(candidate_id)
                # NAT peer can only be reached by peers in the same LAN
                else:
                    if target_localnet == candidate_localnet:
                        reachable_by[target_id].add(candidate_id)


def handle_hello_message(msg, self_id):
    ...
    # If the sender is unknown, add it to the list of known peers (known_peer) and
record their flags (peer_flags).
    if sender_id not in known_peers:
        known_peers[sender_id] = (sender_ip, sender_port)
        peer_flags[sender_id] = {
            "nat": sender_nat,
            "light": sender_light
        }
        new_peers.append(sender_id)
    ...

    # Update the set of reachable peers (reachable_by).
    if sender_id not in reachable_by[self_id]:
        reachable_by[self_id].add(sender_id)
```

peer_manager.py

This module monitors each node's liveness status, manages RTT latency information, and implements a simple blacklist mechanism.

Key functions:

- start_ping_loop(self_id, peer_table, interval=15): Periodically sends PING messages to all known nodes to detect if they are online.
- handle_pong(msg): Upon receiving PONG responses, measures RTT latency and updates node liveness status.
- start_peer_monitor(timeout=10, check_interval=2): Periodically checks whether a node has timed out and marks it as "UNREACHABLE".
- Blacklist mechanism: Offenses (e.g., invalid blocks) are counted. Once a threshold is exceeded, the node is added to a blacklist to ensure security and robustness.

```
def start_ping_loop(self_id, peer_table, interval=15):
    def loop():
       while True:
            cur_time = time.time()
            for peer_id, (ip, port) in peer_table.items():
                if peer_id == self_id:
                    continue
                msg = {
                    "type": "PING",
                    "sender": self_id,
                    "timestamp": cur_time,
                    "message_id": generate_message_id()
                }
                enqueue_message(peer_id, ip, port, msg)
            time.sleep(interval)
    threading.Thread(target=loop, daemon=True).start()

def handle_pong(msg):
    sender = msg.get("sender")
    sent_ts = msg.get("timestamp")
    now = time.time()
    if sender is not None and sent_ts is not None:
        rtt = now - sent_ts
        rtt_tracker[sender] = rtt
        update_peer_heartbeat(sender)
```

```
def record_offense(peer_id):
    peer_offense_counts[peer_id] += 1
    if peer_offense_counts[peer_id] > 0:
        blacklist.add(peer_id)
        print(f"[{peer_id}] has been added to the blacklist due to repeated
offenses.")
```

Part 3: Block and Transaction Generation and Verification

block_handler.py

Mainly in charge of generating and verifying blocks.

- block_generation and create_dummy_block handle new block generation.
- handle_block receives blocks, checks whether the block_id is valid, and then adds the block to received_blocks. If more than 3 invalid messages are received from the same peer, it is marked as malicious. If the block's previous block ID cannot be found, the block is placed in the orphan_blocks list.

```
def block_generation(self_id, MALICIOUS_MODE, interval=20):
    from inv_message import create_inv
    def mine():
        while True:
```

```
            # Create a new block
            block = create_dummy_block(self_id, MALICIOUS_MODE)
            if block is not None:
                # Generate INV message and broadcast
                inv_msg = create_inv([block["block_id"]], self_id)
                gossip_message([peer for peer in known_peers if peer != self_id],
inv_msg)
            time.sleep(interval)
    threading.Thread(target=mine, daemon=True).start()
```

Newly generated blocks are broadcast to known nodes via the gossip_message function in outbox.py.

```
def gossip_message(self_id, message, fanout=3):

    from peer_discovery import known_peers, peer_config, peer_flags
    selected_peers = set()
    for peer in peer_config:
        if peer == self_id:
            continue
        light = peer_flags[peer].get("light", False)
        if light and message["type"] == "TX":
            continue
        selected_peers.add(peer)
        if len(selected_peers) == fanout:
            break
    for peer in selected_peers:
        enqueue_message(peer, known_peers[peer][0], known_peers[peer][1], message)
```

```
def handle_block(msg, self_id):
    block_id = msg.get("block_id")
    expected_hash = compute_block_hash(msg)
    if block_id != expected_hash:
        record_offense(msg.get("sender"))
        print(f"Invalid block ID {block_id} from {msg.get('sender')}, expected
{expected_hash}")
        return False
    if any(b["block_id"] == block_id for b in received_blocks):
        return False
    prev_id = msg.get("previous_block_id")
    if prev_id != "GENESIS" and not any(b["block_id"] == prev_id for b in
received_blocks):
        orphan_blocks[block_id] = msg
        return False
    receive_block(msg)
    return True
```

transaction.py

Handles generating and verifying transactions and maintains the local transaction pool.

Key features:

- TransactionMessage class: encapsulates transaction properties (sender, receiver, amount, timestamp, unique Hash ID, etc.) to facilitate serialization and network transmission.
- transaction_generation(self_id, interval=15): Periodically creates and broadcasts new transactions at set intervals, simulating real network traffic.
- add_transaction(tx): De-duplicates and adds the new transaction to the local transaction pool.
- get_recent_transactions(): Returns all transactions in the transaction pool as a dictionary list, so they can be displayed or included in a block.
- clear_pool(): Empties the transaction pool after the transactions are included in a new block.

```
class TransactionMessage:
    def __init__(self, sender, receiver, amount, timestamp=None):
        self.type = "TX"
        self.from_peer = sender
        self.to_peer = receiver
        self.amount = amount
        self.timestamp = timestamp if timestamp else time.time()
        self.id = self.compute_hash()
    def compute_hash(self):
        tx_data = {
            "type": self.type,
            "from": self.from_peer,
            "to": self.to_peer,
            "amount": self.amount,
            "timestamp": self.timestamp
        }
        return hashlib.sha256(json.dumps(tx_data,
sort_keys=True).encode()).hexdigest()
    def to_dict(self):
        return {
            "type": self.type,
            "tx_id": self.id,
            "from": self.from_peer,
            "to": self.to_peer,
            "amount": self.amount,
            "timestamp": self.timestamp
        }
```

```
def transaction_generation(self_id, interval=15):
    def loop():
        while True:
            candidates = [peer for peer in known_peers if peer != self_id]
            if not candidates:
                time.sleep(interval)
                continue
            to_peer = random.choice(candidates)
```

```
            amount = random.randint(1, 100)
            tx = TransactionMessage(self_id, to_peer, amount)
            add_transaction(tx)
            gossip_message([peer for peer in known_peers if peer != self_id],
tx.to_dict())
            time.sleep(interval)
    threading.Thread(target=loop, daemon=True).start()
```

inv_message.py

Handles the INV message mechanism for block broadcasting and synchronization across the blockchain network.

Key features:

- create_inv(block_ids, sender_id): Constructs an INV message (dictionary) based on the given block IDs and the sender ID to notify other nodes about new blocks to sync.
- get_inventory(): Returns a list of all block IDs stored locally.
- broadcast_inventory(self_id): Automatically builds an INV message and broadcasts it to all other known nodes, informing them of newly acquired block information to accelerate cross-node synchronization.

Implementation details:

- Uses gossip_message() to broadcast the message, automatically excluding the local node so that only the other nodes receive it.
- Cooperates with block generation and block request modules to achieve full network synchronization of the main chain.

```
def create_inv(block_ids, sender_id):
    return {
        "type": "INV",
        "sender": sender_id,
        "block_ids": block_ids,
        "message_id": generate_message_id()
    }

def broadcast_inventory(self_id):
    inv_msg = create_inv(get_inventory(), self_id)
    peer_ids = [peer_id for peer_id in known_peers if peer_id != self_id]
    gossip_message(peer_ids, inv_msg)
```

Part 4: Sending Message Processing (outbox.py)

Responsible for sending messages. Maintains a message queue for storing pending messages. Messages are sent according to priority: high-priority messages get sent first, and only after high-priority messages are sent completely do lower-priority messages proceed. If the target peer is unreachable and NATed, the message is sent to the corresponding relay node, which forwards the message through its queue to the final NAT node.

Other files can call enqueue_message to add messages to the queue. The actual sending logic is implemented in outbox.py.

```
def enqueue_message(target_id, ip, port, message):
    from peer_manager import blacklist, rtt_tracker

    # Check if the peer sends messages to the receiver too frequently using the
function is_rate_limited. If yes, drop the message.
    # Check if the receiver exists in the blacklist. If yes, drop the message.
    # Classify the priority of the sending messages based on the message type
using classify_priority.
    # Add the message to the queue (queues) if the length of the queue is within
the limit QUEUE_LIMIT, otherwise drop it.
    if is_rate_limited(target_id):
        return
    if target_id in blacklist:
        return
    priority = classify_priority(message)

    if message["type"] == "HELLO":
        print(f"🟢 Hello from {target_id}")

    with lock:
        if len(queues[target_id][priority]) < QUEUE_LIMIT:
            queues[target_id][priority].append((ip, port, message))
        else:
            print(f"[{target_id}]🈲 Drop due to queue limit")
            drop_stats[message["type"]] += 1
            return
```

The core sending logic is in send_from_queue. enqueue_message decides whether to send immediately or later based on message priority and the destination status, then calls relay_or_direct_send. If direct communication is possible, send_message is called. If the destination is a NATed peer and cannot be reached directly, get_relay_peer tries to find the relay node with the lowest latency, wraps the original message in a RELAY payload, and calls send_message. Retransmissions are done through simple repeated transmission, up to a maximum of 3 attempts, after which the message is considered dropped.

```
def send_from_queue(self_id):
    def worker():
        while True:  # continuous loop
            for target_id in list(queues.keys()):
                with lock:
                    if (queues[target_id]["HIGH"] or queues[target_id]["MEDIUM"]
or queues[target_id]["LOW"]):
                        ip, port, message = None, None, None
                        if queues[target_id]["HIGH"]:
                            ip, port, message = queues[target_id]
["HIGH"].popleft()
                        elif queues[target_id]["MEDIUM"]:
```

```
                            ip, port, message = queues[target_id]
["MEDIUM"].popleft()
                        elif queues[target_id]["LOW"]:
                            ip, port, message = queues[target_id]["LOW"].popleft()
                        else:
                            continue
                        retries[target_id] = 0
                    else:
                        continue

                success = relay_or_direct_send(self_id, target_id, message)

                # Retry a message if it fails to send. Drop it if retry exceeds
MAX_RETRIES.
                if not success:
                    if retries[target_id] < MAX_RETRIES:
                        retries[target_id] += 1
                        print(f"Retrying: {retries[target_id]}/3")
                        time.sleep(RETRY_INTERVAL)
                        enqueue_message(target_id, ip, port, message)
                    else:
                        drop_stats[message["type"]] += 1
                        retries[target_id] = 0
                else:
                    retries[target_id] = 0
            time.sleep(0.01)  # prevent excessive CPU usage

    threading.Thread(target=worker, daemon=True).start()

def relay_or_direct_send(self_id, dst_id, message):
    from peer_discovery import known_peers, peer_flags, reachable_by
    from utils import generate_message_id

    if message["type"] == "HELLO":
        print(f"🟢 Sending HELLO to {dst_id}")

    # Check if the target peer is NATed.
    nat = peer_flags.get(dst_id, {}).get("nat", False)

    if self_id in reachable_by[dst_id]:
        return send_message(known_peers[dst_id][0], known_peers[dst_id][1],
message)
    if nat:
        relay_peer = get_relay_peer(self_id, dst_id) # (peer_id, ip, port) or None
        if relay_peer:
            relay_msg = {
                "type": "RELAY",
                "sender": self_id,
                "target": dst_id,
                "payload": message,
                "message_id": generate_message_id()
            }
            return send_message(relay_peer[1], relay_peer[2], relay_msg)
        else:
```

```
            print(f"🟡 No relay peer found for {dst_id}")
            return False
    else:
        return send_message(known_peers[dst_id][0], known_peers[dst_id][1],
message)
```

Additionally, the RateLimiter class simulates realistic network sending rate limits, and apply_network_conditions adds random delay/drop behavior.

```
# === Sending Rate Limiter ===
class RateLimiter:
    def __init__(self, rate=SEND_RATE_LIMIT):
        self.capacity = rate               # Max burst size
        self.tokens = rate                # Start full
        self.refill_rate = rate           # Tokens added per second
        self.last_check = time.time()
        self.lock = Lock()

    def allow(self):
        with self.lock:
            now = time.time()
            elapsed = now - self.last_check
            self.tokens += elapsed * self.refill_rate
            self.tokens = min(self.tokens, self.capacity)
            self.last_check = now

            if self.tokens >= 1:
                self.tokens -= 1
                return True
            return False

rate_limiter = RateLimiter()

def apply_network_conditions(send_func):
    def wrapper(ip, port, message):

        # Use rate_limiter.allow to check if sending rate is out of limit. If yes,
drop the message and update drop_stats.
        if rate_limiter.allow() == False:
            drop_stats[message["type"]] += 1
            return False

        # Generate a random number. If it is smaller than DROP_PROB, drop the
message to simulate random channel loss. Log in drop_stats.
        if random.random() < DROP_PROB:
            drop_stats[message["type"]] += 1
            return False

        # Add random latency to simulate transmission delay, then call send_func.
        time.sleep(random.uniform(*LATENCY_MS) / 1000)
        return send_func(ip, port, message)
```

PART 5: Receiving Message Processing (message_handler.py)

Handles incoming messages. The dispatch_message method classifies messages by type. It also maintains seen_message_ids, seen_txs, redundant_blocks, redundant_txs, and message_redundancy to handle duplicate messages. An is_inbound_limited check limits inbound message rate.

Below is an outline of dispatch_message:

PING/PONG are handled to update liveness:

```
elif msg_type == "PING":
        update_peer_heartbeat(msg["sender"])
        pong_msg = create_pong(self_id, msg["timestamp"])
        target_ip, target_port = known_peers[msg["sender"]]
        enqueue_message(msg["sender"], target_ip, target_port, pong_msg)

    elif msg_type == "PONG":
        update_peer_heartbeat(msg["sender"])
        update_peer_heartbeat(self_id)
        handle_pong(msg)
```

INV messages: if received block_ids are not in the local block list, a GETBLOCK message is sent back to request the missing blocks.

```
    elif msg_type == "INV":
        local_block_ids = get_inventory() # list of block_id
        rcv_block_ids = msg.get("block_ids", [])
        missing_block_ids = [block_id for block_id in rcv_block_ids if block_id
not in local_block_ids]
        if missing_block_ids:
            getblock_msg = create_getblock(self_id, missing_block_ids)
            target_ip, target_port = known_peers[msg["sender"]]
            enqueue_message(msg["sender"], target_ip, target_port, getblock_msg)
```

GETBLOCK messages: if a requested block_id is found locally, a BLOCK message is sent back; otherwise, it requests the missing block from known peers.

```
    elif msg_type == "GETBLOCK":
        print(f"[{self_id}] Received GETBLOCK from {msg['sender']}, requesting
blocks: {msg.get('block_ids', [])}")

        rcv_block_ids = msg.get("block_ids", [])
        ret_blocks = []
        missing_block_ids = []

        # 1. Look for blocks locally
```

```
        for block_id in rcv_block_ids:
            block = get_block_by_id(block_id)
            if block:
                ret_blocks.append(block)
                print(f"{self_id} Found block: {block_id}")
            else:
                missing_block_ids.append(block_id)
                print(f"[{self_id}] Missing block: {block_id}")

        # 2. Send locally found blocks
        for block in ret_blocks:
            try:
                # Check serializability
                json.dumps(block)
            except Exception as e:
                print(f"[{self_id}] Block not serializable: {e}, block={block}")
                continue
            print(f"Sending BLOCK: {block['block_id']}")

            try:
                sender = msg["sender"]
            except Exception as e:
                print(f"🆘 Exception in Key")
            try:
                print(f"enqueue_message args: sender={msg.get('sender')},
peer_config={peer_config.get(msg.get('sender'))}")
                enqueue_message(
                    sender,
                    peer_config.get(sender)["ip"],
                    peer_config.get(sender)["port"],
                    block
                )
            except Exception as e:
                print(f"🆘 Error calling enqueue_message: {e}, msg={msg},
peer_config_keys={list(peer_config.keys())}")


        # 3. If there are missing blocks, request them from other peers
        if missing_block_ids:
            for peer_id in known_peers:
                if peer_id == self_id:
                    continue
                get_block_msg = create_getblock(self_id, missing_block_ids)
                enqueue_message(peer_id, peer_config[peer_id]["ip"],
peer_config[peer_id]["port"], get_block_msg)

            # 4. Up to 3 retries, each wait 10s
            retry_cnt = 0
            while missing_block_ids and retry_cnt < 3:
                retry_cnt += 1
                print(f"[{self_id}] get block retry {retry_cnt} times, missing:
{missing_block_ids}")
                time.sleep(10)
                found_block_ids = []
```

```
                for block_id in missing_block_ids:
                    block = get_block_by_id(block_id)
                    if block:
                        try:
                            json.dumps(block)
                        except Exception as e:
                            print(f"[{self_id}] Block not serializable: {e},
block={block}")
                            continue
                        print(f"Sending BLOCK: {block['block_id']}")
                        enqueue_message(
                            msg["sender"],
                            peer_config[msg["sender"]]["ip"],
                            peer_config[msg["sender"]]["port"],
                            block
                        )
                        found_block_ids.append(block_id)
                # Remove found blocks
                for block_id in found_block_ids:
                    missing_block_ids.remove(block_id)
                # Continue requesting from other peers
                if missing_block_ids:
                    for peer_id in known_peers:
                        if peer_id == self_id:
                            continue
                        get_block_msg = create_getblock(self_id,
missing_block_ids)
                        enqueue_message(peer_id, peer_config[peer_id]["ip"],
peer_config[peer_id]["port"], get_block_msg)
```

Other message types primarily call functions in other files, so we skip redundant details.

Part 6: Dashboard Visualization (dashboard.py)

This module is based on Flask and provides a real-time visualization panel for the simulated blockchain network. By visiting HTTP endpoints, users can observe and debug network status and performance.

Key endpoints:

- / : Home page, shows a title.
- /blocks : Displays the block data the node has received (received_blocks).
- /peers : Displays details for all known nodes, including NAT/Light flags and status.
- /transactions : Displays all transactions in the local pool.
- /latency : Shows RTT (round-trip) information to other nodes for network delay analysis.
- /capacity : Reserved endpoint for future extension (e.g., bandwidth or throughput display).
- /orphans : Shows orphan_blocks to track blocks not accepted into the main chain.
- /queue : Displays message queue contents by node and priority.
- /redundancy : Shows redundancy statistics by calling get_redundancy_stats().
- /blacklist : Shows the blacklist for observing malicious nodes.

Debug endpoints, such as /reachable, /peer_config, /known_peers, /peer_flags, and /drop_stats, are available to monitor and debug internal states of the network.

How to Run

1. Use docker compose up --build to start all nodes. Each node runs in a separate container.
2. Each node automatically completes initialization, peer discovery, message receiving and sending, and block/transaction generation.
3. By visiting each node's localhost:port along with various endpoints, you can observe real-time data on blocks, transaction pools, queues, and node status.

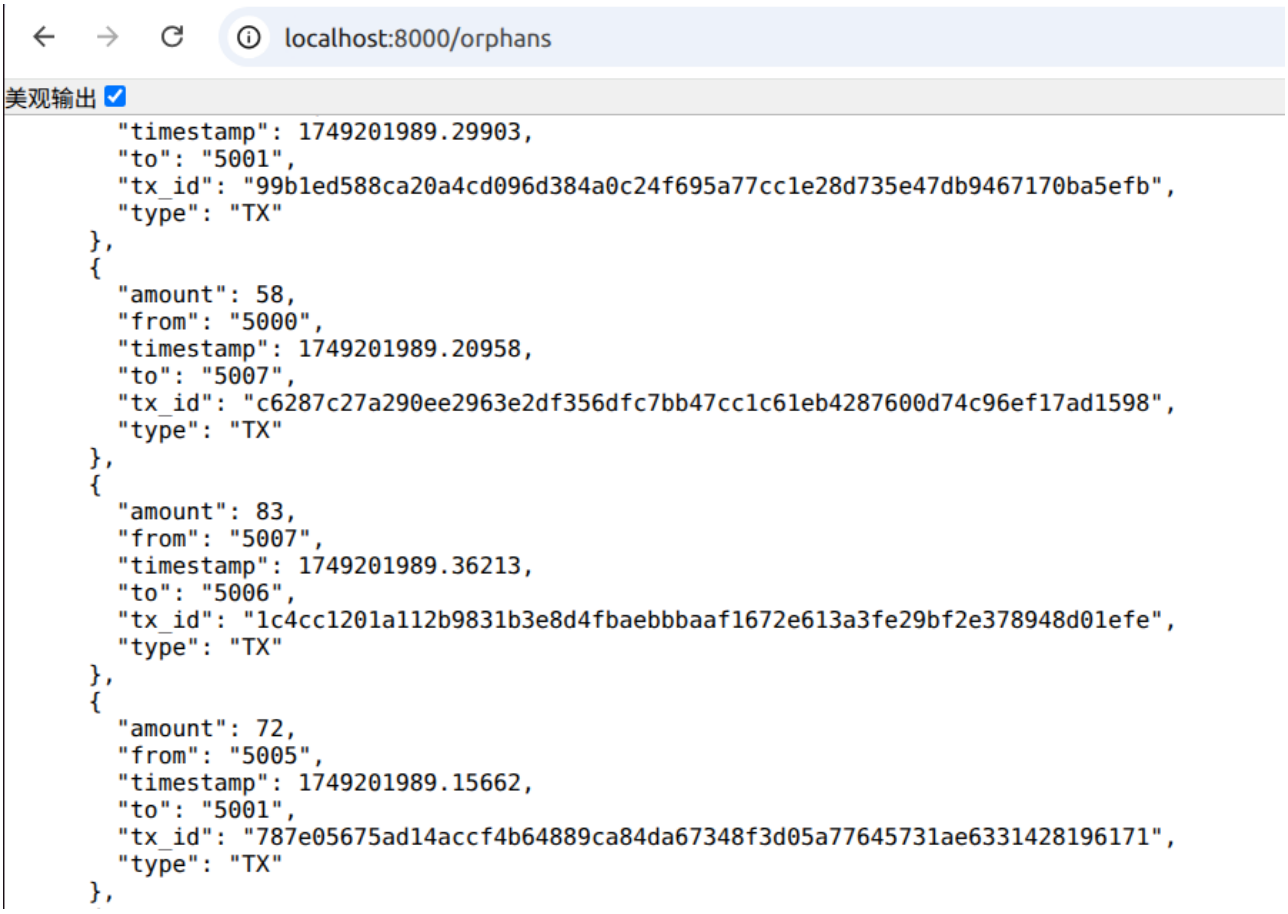
For full nodes:

- Automatically generate new blocks.
- Automatically generate new transactions, which are cleared after being included in a block.
- Block pool example:


```
[
  {
    "block_id": "af56794a973b4c43602b559cad74ffeef251d7c0c48df1cc98f94d99bc0185f3",
    "previous_block_id": "GENESIS",
    "sender": "5000",
    "timestamp": 1749200632.26488,
    "transactions": [
      {
        "amount": 15,
        "from": "5000",
        "timestamp": 1749200632.26467,
        "to": "5006",
        "tx_id": "91db430aeb60c2177b8550dbb21c629eb683d526581f09c3de8d92b365daa5c0",
        "type": "TX"
      }
    ],
    "type": "BLOCK"
  },
  {
    "block_id": "cb1fc5e1e3a27db934c8deb85c1855b21cb602ea41ee3ca8ed3a1efd100fca92",
    "previous_block_id": "GENESIS",
    "sender": "5003",
    "timestamp": 1749200632.15647,
    "transactions": [
      {
        "amount": 88,
        "from": "5003",
        "timestamp": 1749200632.15611,
        "to": "5001",
        "tx_id": "d43f64df519fac581282d0e8ca7088a0adc66a132b3eb36fc241a735e18638e4",
        "type": "TX"
      }
    ],
    "type": "BLOCK"
  },
  {
    "block_id": "0c6c462f73c83a450407bea46072ef69b50e68939487704c6209f249db51795e",
    "previous_block_id": "GENESIS",
    "sender": "5002",
    "timestamp": 1749200632.3423,
    "transactions": [
      {
        "amount": 80,
        "from": "5002",
        "timestamp": 1749200632.34206,
        "to": "5007",
        "tx_id": "d5176e9a01506967965400d40404947243de264a3211c65bf5f687ded8c34988",
        "type": "TX"
      }
    ],
    "type": "BLOCK"
  }
]
```

- Transactions pool example:

- Orphan pool example:



For light nodes:

- They do not generate new blocks.
- They do not generate new transactions.
- They only receive block headers.

```
[
  {
    "block_id": "30c5486f787026a79aa4f4527a1810c804ac703ef919453f4d072bd706946bd4",
    "previous_block_id": "GENESIS",
    "sender": "5002",
    "timestamp": 1749201530.85813
  },
  {
    "block_id": "b13f92b93d393452436abfe51c84370e508c5017175fac06f75ab6833ff55b72",
    "previous_block_id": "GENESIS",
    "sender": "5003",
    "timestamp": 1749201530.7299
  }
]
```


For NAT nodes:

- They cannot directly communicate with peers in a different localnetworkid and require a relay node.
- Below is the queue of node on port 5000. Peer 5003 is a NAT peer in the same local network. Peers 5007 and 5005 need to reach 5003, so messages from 5007 and 5005 to 5003 are relayed through 5000 as RELAY messages (payload containing the original message).


```
},
"5003": {
  "HIGH": [],
  "LOW": [
    "('172.28.0.13', 5003, {'type': 'RELAY', 'sender': '5007', 'target': '5003', 'payload': {'type': 'PING', 'sender': '5007', 'timestamp': 1749200857.399377, 'message_id': 'bba265f6-fb59-479e-9894-c3346abc0af3'}, 'message_id': '04276849-ffd9-4cca-ad0d-890edf0c4cbb'})",
    "('172.28.0.13', 5003, {'type': 'RELAY', 'sender': '5005', 'target': '5003', 'payload': {'type': 'PING', 'sender': '5005', 'timestamp': 1749200857.58792, 'message_id': '76bf47b9-2b3a-48be-9e59-4874d4288d49'}, 'message_id': '1065f8e0-9d7a-48cc-b43e-8765455c45a3'})"
  ],
  "MEDIUM": []
},
"5004": {
  "HIGH": [],
  "LOW": [],
  "MEDIUM": []
},
"5005": {
```


Other Parameters


```
[
  {
    "ip": "172.28.0.10",
    "light": false,
    "nat": false,
    "peer_id": "5000",
    "port": 5000,
    "status": "ALIVE"
  },
  {
    "ip": "172.28.0.11",
    "light": false,
    "nat": false,
    "peer_id": "5001",
    "port": 5001,
    "status": "ALIVE"
  },
  {
    "ip": "172.28.0.12",
    "light": false,
    "nat": true,
    "peer_id": "5002",
    "port": 5002,
    "status": "ALIVE"
  }
```

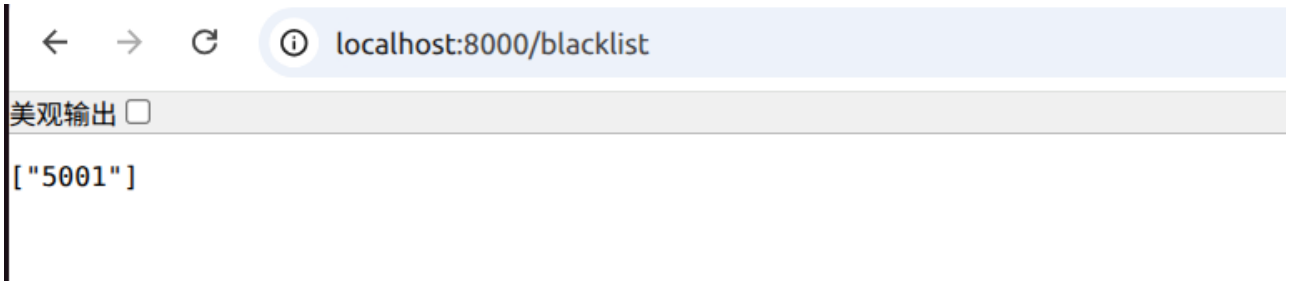
```
  },
  {
    "ip": "172.28.0.13",
    "light": false,
    "nat": true,
    "peer_id": "5003",
    "port": 5003,
    "status": "ALIVE"
  },
  {
    "ip": "172.28.0.14",
    "light": false,
    "nat": false,
    "peer_id": "5004",
    "port": 5004,
    "status": "ALIVE"
  },
  {
    "ip": "172.28.0.15",
    "light": false,
    "nat": false,
    "peer_id": "5005",
    "port": 5005,
    "status": "ALIVE"
  },
  {
    "ip": "172.28.0.16",
    "light": true,
    "nat": true,
    "peer_id": "5006",
    "port": 5006,
    "status": "ALIVE"
  },
  {
    "ip": "172.28.0.17",
    "light": false,
    "nat": true,
    "peer_id": "5007",
    "port": 5007,
    "status": "ALIVE"
  }
]
```


- Peers example:

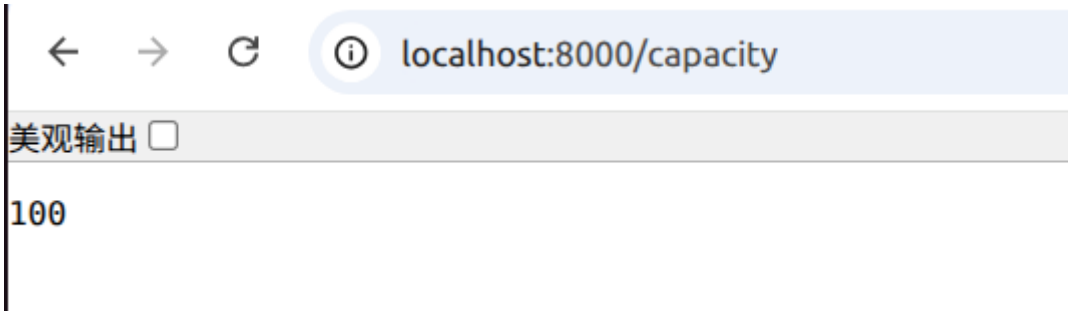
- Latency example:


```
{"5000":7.060009479522705,"5001":10.409663438796997,"5002":12.580609321594238,"5003":8.758962154388428,"5004":Infinity,"5005":0.5959677696228027,"5006":5.570429801940918,"5007":6.167625904083252}
```

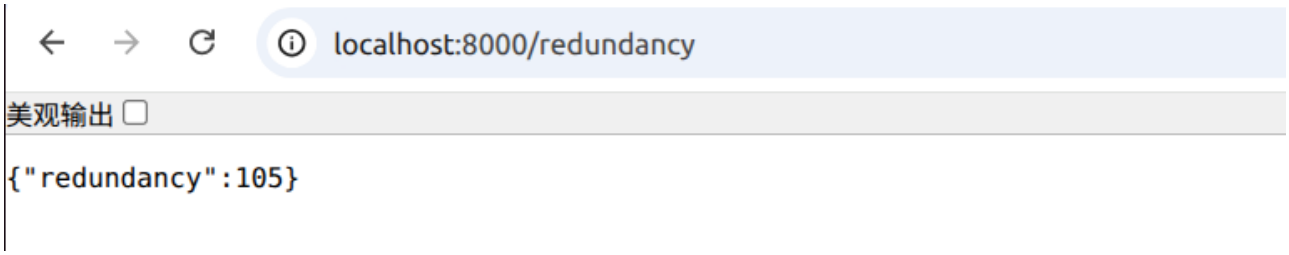
- Blacklist example:



- Capacity example:



- Redundancy example:



- drop_stats example: