BlockChain: Tendermint Logic simulation

Table of contents:

Introduction to Tendermint

Tendermint is a software that can replicate applications securely and consistently across multiple machines. From a security perspective, even if 1/3 of the random nodes in a distributed system go down, Tendermint can still ensure that the distributed system operates normally. In addition, in terms of data consistency, the failed nodes can see the same transaction log and calculate the same state. Safe and consistent replication is a fundamental problem in distributed systems and plays a key role in many applications, such as monetary systems, elections, and infrastructure orchestration systems.

Distributed systems need to be able to tolerate nodes being offline or failing, which is called Byzantine fault tolerance. The Byzantine fault tolerance theory has a long history, and the successful implementation of blockchain technology, such as Bitcoin and Ethereum, has made the theory popular in the computer field. Blockchain technology focuses on the evolution of Byzantine fault tolerance by combining peer-to-peer (P2P) networks and cryptographic authentication. Transactions are batched into blocks, and each block is connected to the previous block through cryptographic hashing to form a chain, which is why the technology is named Blockchain.

Features of Tendermint

Tendermint Core:

This is Tendermint's consensus engine. It functions on Proof of Stake (PoS), where a selected node from a validator set proposes the new block to be added to a blockchain. Other Validators must then vote before the block goes through. Multiple systems on Tendermint can view the same transactions at the same time and in the same order.Furthermore, since one cannot always detect malicious intent, Tendermint maintains security using Byzantine Fault Tolerance. BFT is a mechanism that allows for the Consensus to be resistant to up to ⅓ malicious nodes.

Application Blockchain Interface (ABCI):

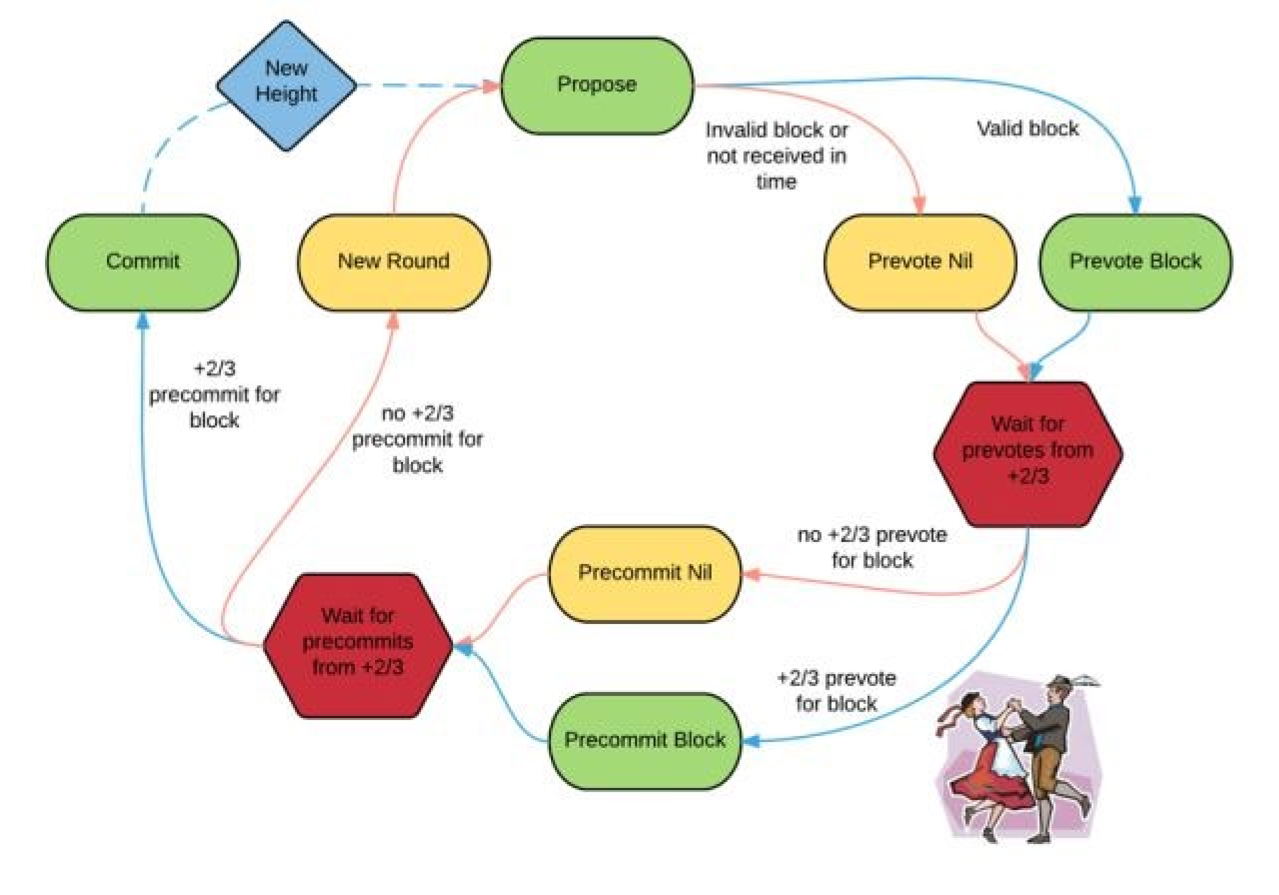
This is the Tendermint toolkit of ready-to-use software for replicating or launching blockchains. Tendermint is unlike most major blockchains in that it has a modular architecture in contrast with the popular monolithic architecture. Its modular architecture makes it possible for a wide range of applications and their different languages to be integrated with the Tendermint core. ABCI is a major part of tendermint's networking layer, a channel for all transactions and interactions. Any application Layer logic must go through to ABCI to reach the consensus engine. Furthermore, ABCI supports any programming language on application logic layers connected to it.

How Tendermint works

There are two roles in the protocol:
(1) Validator: a role or node in the protocol. Different validators have different powers in the voting process.
(2) Proposer: generated by validators in turn.

As can be seen from the figure, to successfully submit a block, two stages of voting must be performed, called pre-vote and pre-commit. When more than 2/3 of the validators have pre-committed the same block in the same round of proposals, the block will be submitted.

Due to reasons such as offline or network delays, the proposer may fail to propose a block. This situation is also allowed in Tendermint, because the validator will wait for a certain period of time before entering the next round of proposals to receive the block proposed by the proposer.



How Tendermint works

To summarize briefly, tendermint is mainly divided into three stages:

1.Propose: Select the proposer to broadcast the block.

2.Prevote: Validators check the block and vote.

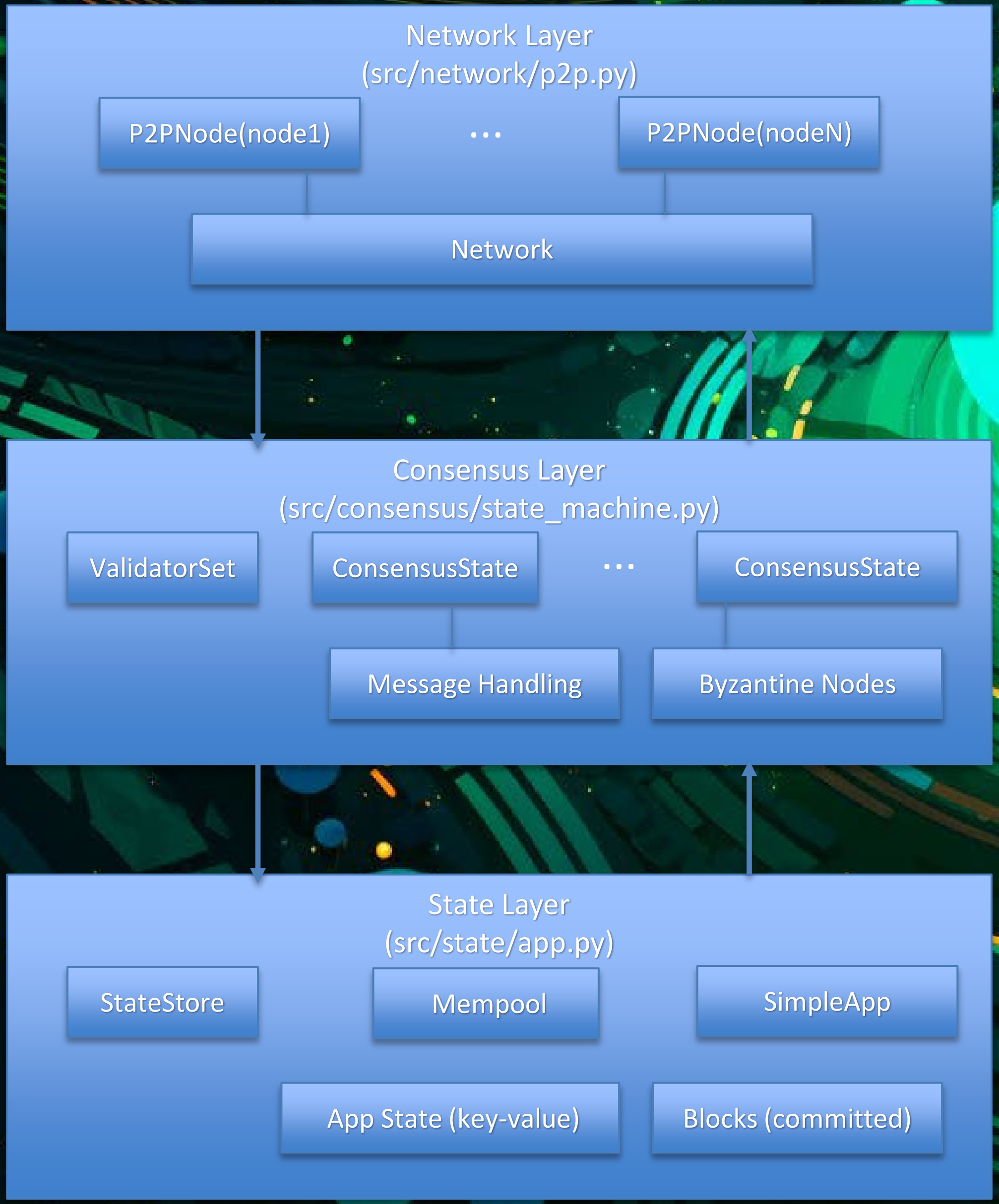
3.Precommit: Submit the block after the vote is passed.

This project

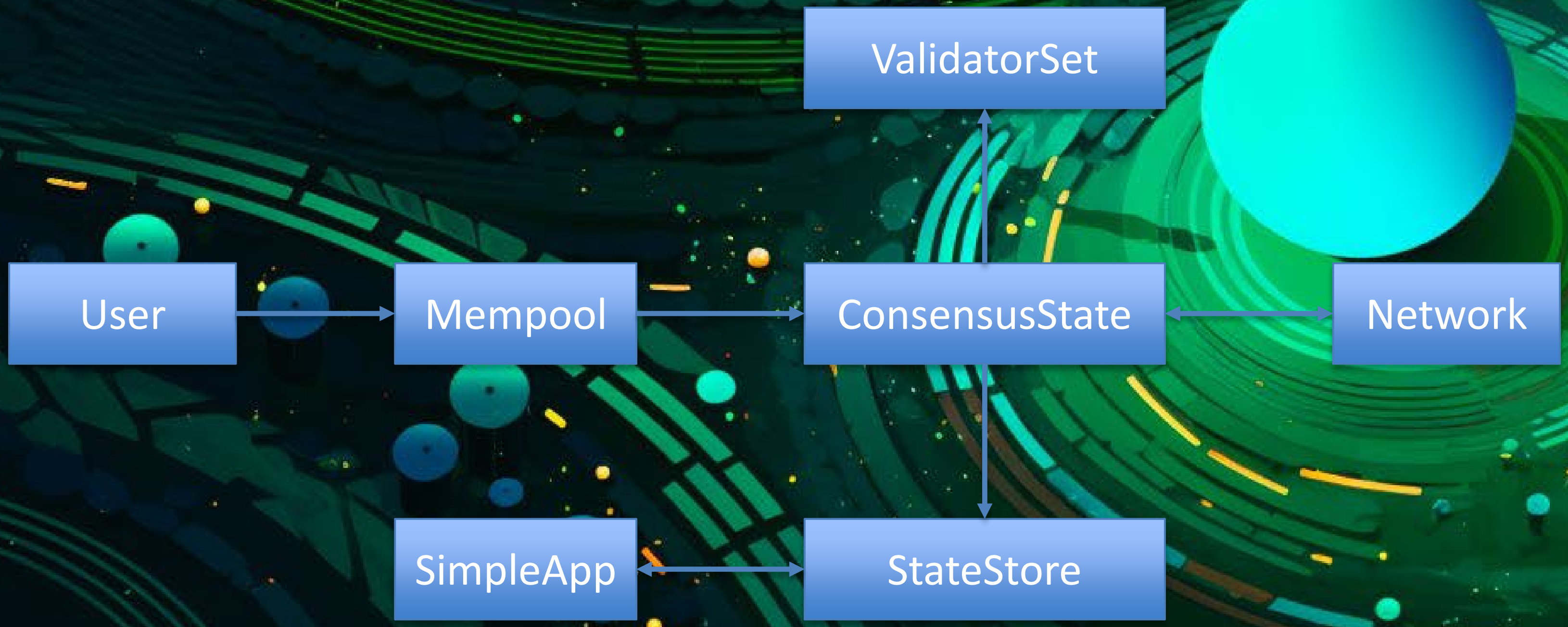
This is a simplified Tendermint implementation, designed to allow for learning the tendermint workflow while exploring the Byzantine Fault Tolerant consensus mechanism.

The main goal of the project is to simulate the behavior of distributed nodes. Implement core consensus logic (proposal, pre-voting, submission). And support basic status updates and transaction pool functions.

Project architecture design



Workflow of this project



Workflow of this project in detail

1.User submits transaction
Trigger point: User submits transaction to a node through the interface.
Module call:
Mempool.add_tx(tx):
Convert transaction object to JSON string and store it in txs list. Transaction pool manages storage and extraction of multiple transactions.
Output result: Transaction enters transaction pool and waits for proposer to package.

2. Propose Step
Trigger point: The proposer node of the current round.
Module call:
ConsensusState.broadcast_proposal():
Get transaction data from Mempool.get_txs() and generate a block.
Use compute_block_hash() to calculate the block hash and fill the block header.
Broadcast proposal message (Message type: propose).
Data flow:
The proposal message is propagated to other nodes through Network.send_message().

3. Prevote Step
Trigger point: The node receives a proposal message.
Module call:
ConsensusState.handle_propose():
Verify whether the height, round, and hash of the proposed block are correct. If legal, enter the prevote stage.
ConsensusState.broadcast_prevote():
If the block is legal, broadcast the Prevote message supporting the block. If the block is illegal, broadcast an empty Prevote message.
Data flow:
Prevote messages are broadcast via Network.send_message().

4. Precommit Step
Trigger point: The node receives enough pre-vote messages.
Module call:
ConsensusState.handle_prevote():
Collect pre-votes to determine whether 2/3 support a block. If the threshold is reached, enter the pre-vote phase.
ConsensusState.broadcast_precommit():
Broadcast the Precommit message for the block.

5. Block Commit (Commit Step)
Trigger point: The node receives enough commit vote messages.
Module call:
ConsensusState.handle_commit():
Call StateStore.commit_block() to commit the block to local storage.Call SimpleApp.apply_block() to update the state.
StateStore.commit_block():
Add the block to the blockchain storage.
Traverse the transactions in the block and update app_state.
Output result:
The block is successfully submitted and the system status update is completed.

Module Introduction

```python
async def handle_propose(self, msg):
    proposal = msg.data["block"]
    proposal_height = proposal["header"]["height"]
    proposal_round = proposal["header"]["round"]

    if proposal_height in self.committed_heights:
        logger.debug(f"{self.validator_id} already committed block at height {proposal_height}, ignoring proposal.")
        return

    self.proposal_block = proposal
    self.height = proposal_height
    self.round = proposal_round
    logger.debug(f"{self.validator_id} received proposal block at H:{proposal_height} R:{proposal_round}")
    # Record the start of the pre-voting timer after receiving the proposal
    asyncio.create_task(self.set_step_timeout("prevote_timeout", 0.5))
```

ConsensusState: consensus state machine
(src/consensus/state_machine.py)
Core code functions
Implement the core logic of the consensus algorithm, including proposal, pre-voting, and submission voting.
Manage the current status of the node (height, height round, etc.).
Process received messages and advance the status according to the consensus logic.

```python
async def check_prevote_threshold(self):
    # Simple check: if 2/3 prevotes are collected, then enter precommit
    needed = (2 * len(self.validator_set.validators) // 3) + 1
    if len(self.prevotes) >= needed:
        logger.debug(f"{self.validator_id} got {needed} prevotes")
        await self.enter_precommit_step()
```

State management:

Manage the current state of the node through the properties height, round and step.
Use proposal_block to temporarily store the proposed block, and prevotes and precommits to store votes.
Message processing:

handle_propose: Verify the proposed block and start the pre-voting stage.
handle_prevote: Process the pre-vote and check whether the voting threshold reaches 2/3.
handle_precommit: Check the submission vote and trigger the block submission.

```python
async def commit_block(self):
    if self.height in self.committed_heights:
        logger.debug(f"{self.validator_id} has already committed block at height {self.height}, skipping commit.")
        return

    # Submit block to state
    block = self.build_block_from_proposal(self.proposal_block)
    self.state_store.commit_block(block, self.app)
    self.committed_heights.add(self.height)
    logger.info(f"{self.validator_id} committed block at height {self.height}")
```

State update:

commit_block: Submit the verified block to the state storage, update the node state, and enter the next height.

ValidatorSet: Validator Management (src/consensus/validator.py)

Core Code Functions

Proposer Selection:

Use height and round to calculate the current proposer.

Voting Signature and Verification:

Use sign_vote to generate a simple signature for the vote.

Use verify_vote to verify the legitimacy of the signature.

```python
def get_proposer(self, height, round_):
    # Simplified: select the proposer based on the modulus of height + round
    index = (height + round_) % len(self.validators)
    return self.validators[index]

def is_validator(self, vid):
    return vid in self.validators

def sign_vote(self, validator_id, block_hash):
    # Simple signature: f"{validator_id}:{block_hash}"
    return f"{validator_id}:{block_hash}"

def verify_vote(self, vote):
    # Simple verification, check signature format and verifier identity
    if not self.is_validator(vote.validator_id):
        return False
    expected_sig = f"{vote.validator_id}:{vote.block_hash}"
    return vote.signature == expected_sig
```

P2PNode & Network: Network simulation (src/network/p2p.py)

Core code functions

Message passing:

send_message simulates delay and packet loss behavior.

Each node receives messages through the inbox queue.

Network partitioning:

Supports isolating some nodes to prevent them from sending or receiving messages.

```python
async def send_message(self, target_id, message):
    if self.partitioned and target_id in self.partitions.get("isolated", []):
        # Simulate network partition and isolate messages
        return
    # Fault injection: delayed sending
    await asyncio.sleep(self.delay)
    # Fault injection: probabilistic packet loss
    if random.random() > self.drop_rate:
        await self.nodes[target_id].inbox.put(message)

def get_node(self, node_id):
    return self.nodes[node_id]

def partition_network(self, isolated_nodes):
    """Isolate specific nodes in the network."""
    self.partitioned = True
    self.partitions = {"isolated": isolated_nodes}
```

Mempool: Simple transaction pool (src/state/mempool.py)

Core code functions

Transaction management:

Supports adding transactions to the pool.

Extract a specified number of transactions from the pool for packaging.

```python
def add_tx(self, tx):
    # tx can be a dictionary, converted to json string storage
    if isinstance(tx, dict):
        tx_str = json.dumps(tx)
    else:
        tx_str = tx
    self.txs.append(tx_str)

def get_txs(self, num=5):
    # Get at most num transactions from mempool, if not enough, return all
    # For simplicity, take from the front and delete after taking
    batch = self.txs[:num]
    self.txs = self.txs[num:]
    return batch
```

StateStore: State storage (src/state/store.py, src/state/app.py)

Core code functions

Block submission:

Write the block to the storage through commit_block.

Use apply_block to apply the transactions in the block to the state.

```
def commit_block(self, block, app):
    self.blocks.append(block)
    # Simplified application execution: Assume that each transaction is {"action":"set","key":"k","value":"v"}
    app.apply_block(block, self)
```

```
def apply_block(self, block, state_store):
    for tx_str in block.transactions:
        tx = json.loads(tx_str)
        if tx.get("action") == "set":
            state_store.app_state[tx["key"]] = tx["value"]
    state_store.blocks.append(block)
```

Comparison with Tendermint

Commonalities

Same core idea:
BFT consensus logic.
Modular design (separation of consensus and state).

Functional implementation:
Proposer selection logic.
Message processing and inter-node communication.

Differences

Unimplemented functions:
ABCI interface: does not provide a standard protocol between applications and consensus engines.
Network protocol: does not implement the Gossip protocol, only simulates message propagation.
Fast synchronization: the node cannot quickly synchronize the state after startup.

Simplified parts:
Signature and verification logic: use simple string simulation.
State storage: only supports key-value state updates.

Code Explanation

ConsensusState: Consensus state machine

Function overview

Implement the core consensus logic, including processing proposals, pre-voting, submission and other steps.

Manage node status: height (block height), round (consensus round), step (current stage).

Coordinate the consensus process between nodes through message processing (handle_message).

Key implementation

Message processing:

handle_propose: Receive and verify the proposed block and start the pre-voting stage.

handle_prevote: Collect pre-votes and check whether the 2/3 threshold is reached.

handle_precommit: Collect submission votes and trigger block submission.

Timeout processing:

Timeout triggers step conversion, such as entering pre-voting from the proposal stage.

State update:

Call commit_block to submit the block and broadcast the submission message.

```
async def handle_message(self, msg):
    if msg.type == "timeout":
        await self.handle_timeout(msg.data["event_type"])
    elif msg.type == "propose":
        await self.handle_propose(msg)
    elif msg.type == "prevote":
        await self.handle_prevote(msg)
    elif msg.type == "precommit":
        await self.handle_precommit(msg)
    elif msg.type == "commit":
        await self.handle_commit(msg)
```

Code Explanation

ValidatorSet: Validator Management

Function Overview

Manage the identity of validators and select proposers.

Verify the legitimacy of the vote (signature and identity authentication).

Key Implementation

```
def get_proposer(self, height, round_):
    # Simplified: select the proposer based on the modulus of height + round
    index = (height + round_) % len(self.validators)
    return self.validators[index]
```

Proposer Selection:

Calculate the proposer index using height and round.

Signature and Verification:

Generate and verify voting signatures to ensure legitimacy.

```
def verify_vote(self, vote):
    # Simple verification, check signature format and verifier identity
    if not self.is_validator(vote.validator_id):
        return False
    expected_sig = f"{vote.validator_id}:{vote.block_hash}"
    return vote.signature == expected_sig
```

Code Explanation

Block and Vote: Data structure

Function overview

Block and Vote provide the basic structure of blocks and votes.

Support serialization and deserialization for easy transmission between nodes.

Key implementation

BlockHeader:

Contains the metadata of the block (height, round, proposer ID, block hash).

Provides dictionary conversion methods to_dict and from_dict.

Vote:

Contains validator ID, block hash, height, round and signature.

```python
class BlockHeader:
    def __init__(self, height, round, proposer_id, block_hash):
        self.height = height
        self.round = round
        self.proposer_id = proposer_id
        self.block_hash = block_hash

    def to_dict(self):
        return {
            "height": self.height,
            "round": self.round,
            "proposer_id": self.proposer_id,
            "block_hash": self.block_hash
        }

    @staticmethod
    def from_dict(data):
        return BlockHeader(
            height=data["height"],
            round=data["round"],
            proposer_id=data["proposer_id"],
            block_hash=data["block_hash"]
        )
```

```python
class Vote:
    def __init__(self, validator_id, block_hash, height, round, signature):
        self.validator_id = validator_id
        self.block_hash = block_hash
        self.height = height
        self.round = round
        self.signature = signature

    def to_dict(self):
        return {
            "validator_id": self.validator_id,
            "block_hash": self.block_hash,
            "height": self.height,
            "round": self.round,
            "signature": self.signature
        }

    @staticmethod
    def from_dict(data):
        return Vote(
            validator_id=data["validator_id"],
            block_hash=data["block_hash"],
            height=data["height"],
            round=data["round"],
            signature=data["signature"]
        )
```

Code Explanation

P2PNode and Network: Network simulation

Function overview

Provide communication mechanism between nodes.

Simulate network delay, packet loss rate and network partition.

Key implementation

Node communication:

Use asynchronous queue inbox to store received messages.

Messaging between nodes is implemented through send_message method.

Network failure:

Simulate delay and packet loss by setting delay and drop_rate parameters.

Support partition isolation nodes to prevent messages from being sent or received.

```
class P2PNode:
    def __init__(self, node_id, network):
        self.node_id = node_id
        self.network = network
        self.inbox = asyncio.Queue()

    async def receive_message(self):
        return await self.inbox.get()

    async def send_message(self, target_id, message):
        await self.network.send_message(target_id, message)
```

Code Explanation

Mempool and StateStore: state and transaction management

Function overview

Mempool:

Manage transaction pool, support transaction addition and batch extraction.

Provide interfaces add_tx and get_txs.

StateStore:

Save the state of the blockchain and the application state.

Receive blocks and apply transactions through commit_block.

```
class Mempool:
    def __init__(self):
        self.txs = []

    def add_tx(self, tx):
        # tx can be a dictionary, converted to json string storage
        if isinstance(tx, dict):
            tx_str = json.dumps(tx)
        else:
            tx_str = tx
        self.txs.append(tx_str)

    def get_txs(self, num=5):
        # Get at most num transactions from mempool, if not enough, return all
        # For simplicity, take from the front and delete after taking
        batch = self.txs[:num]
        self.txs = self.txs[num:]
        return batch

    def size(self):
        return len(self.txs)
```

```
class StateStore:
    def __init__(self):
        self.blocks = []
        self.app_state = {}

    def commit_block(self, block, app):
        self.blocks.append(block)
        # Simplified application execution: Assume that each transaction is {"action":"set","key":"k","value":"v"}
        app.apply_block(block, self)
```

Code Explanation

Main program logic

Function overview

Integrate various modules, initialize network, state and consensus logic.

Simulate distributed node operation.

Key implementation

Initialize nodes:

Create independent StateStore, SimpleApp and ConsensusState instances for each node.

Start consensus process:

Broadcast proposal message.

Simulate timeout trigger process.

```python
async def main():
    network = Network(NODE_IDS, delay=0.01, drop_rate=0.0)
    validator_set = ValidatorSet(NODE_IDS)

    # Create state stores and applications for all nodes
    state_stores = {vid: StateStore() for vid in NODE_IDS}
    apps = {vid: SimpleApp(mempool=Mempool()) for vid in NODE_IDS}

    # Initialize the mempool for all nodes and add some transactions
    for vid in NODE_IDS:
        for i in range(20):
            tx = {"action": "set", "key": f"key{i}", "value": f"value{i}"}
            apps[vid].mempool.add_tx(tx)

    # Initialize consensus states for all nodes
    consensus_states = {}
    for vid in NODE_IDS:
        p2p_node = network.get_node(vid)
        cs = ConsensusState(vid, validator_set, state_stores[vid], apps[vid], p2p_node)
        consensus_states[vid] = cs

    # Start the consensus state machine for all nodes
    tasks = [asyncio.create_task(cs.run()) for cs in consensus_states.values()]

    # Assign Byzantine behavior to some nodes
    num_byzantine = len(NODE_IDS) // 3  # Up to 1/3 of nodes can be Byzantine
    byzantine_nodes = assign_byzantine_nodes(consensus_states, num_byzantine)
    print(f"Byzantine nodes: {byzantine_nodes}")

    # Find the current proposer and broadcast the first proposal
    proposer = validator_set.get_proposer(consensus_states["node1"].height, consensus_states["node1"].round)
    proposer_cs = consensus_states[proposer]
    await proposer_cs.broadcast_proposal()

    # Trigger propose timeout for all nodes to start the consensus process
    for vid in NODE_IDS:
        p2p_node = network.get_node(vid)
        timeout_msg = Message(type="timeout", sender="timer", data={"event_type": "propose_timeout"})
        await p2p_node.send_message(vid, timeout_msg)

    # Run the system for a short time for testing
    await asyncio.sleep(5)
    for t in tasks:
        t.cancel()
```

Conclusion and future work

Conclusion

Implementation of Tendermint core consensus process

Implementation content: This project reproduces the core logic of Tendermint, including the Propose, Prevote, and Precommit phases, as well as message communication between distributed nodes.

Simplified implementation suitable for teaching and verification

Simplified Tendermint implementation written, with clear modularization and easy analysis.

Simulates challenges in real distributed systems such as network delay, packet loss, and Byzantine nodes.

Understand the message synchronization problem between nodes.

Explore how the system reaches consensus when some nodes are unreliable (Byzantine behavior).

Future work

Improvement direction

1. Introduce encrypted signatures and strict verification logic
Problem: The current voting signature uses a simple string simulation, lacking real encryption and security verification.
Improvement: Use asymmetric encryption algorithms (such as ECDSA) to generate and verify signatures to ensure the immutability of voting messages. Introduce validator authentication based on public and private key pairs to increase the security of the system.

2. Implement a complete ABCI interface
Problem: The current state management of the project is directly bound to the application logic, lacking a standardized interface.
Improvement: Implement Tendermint's Application Blockchain Interface (ABCI) to decouple the consensus layer from the application layer. Support external applications to provide state update logic (such as smart contracts) through ABCI.

3. Network layer optimization
Problem: The current network implementation simulates delays and packet loss, but lacks more complex behavior support.
Improvement: Implement the Gossip protocol: optimize the efficiency of message propagation so that each message only needs to be transmitted a limited number of times to reach the entire network. Dynamic partitioning and node crash simulation: support more complex network failure scenarios for testing the robustness of the system.

4. Support fast synchronization
Problem: After a new node joins, it cannot quickly synchronize the current blockchain status.
Improvement: Implement Tendermint's Fast Sync mechanism to enable new nodes to quickly join the network by downloading blockchain snapshots.

5. Improve concurrent performance
Problem: The current system runs through a single-threaded asynchronous task, with low performance.
Improvement: Use multi-threading or multi-process technology to achieve parallel transaction processing and consensus calculation. Optimize transaction pool management to support higher transaction throughput.

Thanks for your attention