Distributed OS Project 2 Report

Kaidi Fu Yufeng Zhao Kaisheng Tang

January 20, 2024

1 Baseline Description

In the first part code constitutes a robust implementation of a peer-to-peer (P2P) blockchain system in the Go programming language. The implementation demonstrates key components necessary for a decentralized blockchain network, including the handling of blocks, transactions, peer management, and communication using gRPC.

We initialize the P2P server by setting up a peer list, transaction pool, and communication channels. It reads the blockchain from a local file, initializing the transaction pool from the existing blocks. The SaveChainToLocal function converts the blockchain to .json format and writes it to a local file. This ensures the persistence of the blockchain data between server restarts.

The Broadcast function facilitates communication between peers. It sends new blocks to all connected peers and updates their blockchains if needed.

The Mine function in the main package represents the miner component of the P2P blockchain system. This function runs in a continuous loop, responsible for generating new blocks by aggregating transactions from the transaction channel (Server.NewTrans). The miner collects up to ten transactions from the NewTrans channel and converts them into a json-formatted string. We utilize the GenerateBlock function described above to create a new block. It takes the hash of the current blockchain's tail block, the aggregated transaction string, and the length of the existing blockchain as parameters. It then append the newly generated block to the blockchain. If successful, it broadcasts the block to connected peers, and the blockchain is saved locally.

Also, we implement the hash with difficulty function. It essentially performs proof-of-work by iteratively hashing the input data with an incremented nonce until a hash is found that satisfies the specified difficulty level. The difficulty is defined by the number of leading zero bytes in the hash. The higher the difficulty, the more computational effort is required to find a valid hash.Here is a general graph illustrating the relation of speed and difficulty, and we ensure that the time used to generate a block is relatively stable:

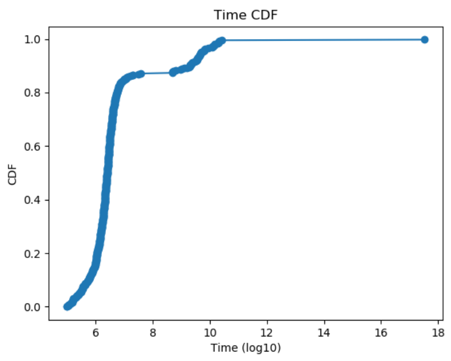

Figure 1: latency vs difficulty

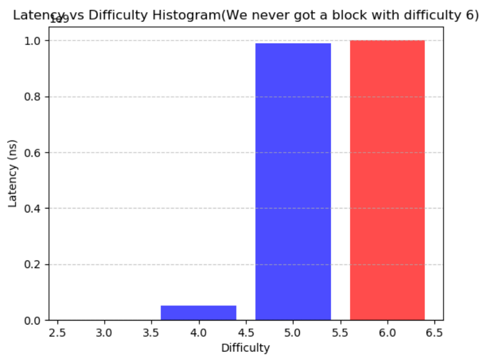

Figure 2: latency cdf

As for the blockchain, implements a basic blockchain structure with functionality for creating, appending, fetching, and replacing chains of blocks. The blockchain is a decentralized and secure ledger that stores a series of blocks, each containing a list of transactions.

A ‘Block‘ represents a unit of data in the blockchain and contains the following fields:

- `PVHash`: Previous block hash.
- `Timestamp`: Timestamp of the block’s creation.
- `Data`: Serialized transaction data.(changed in the utxo part)
- `Index`: Index of the block in the blockchain.
- `Nonce`: Nonce value for proof-of-work.
- `Hash`: Hash of the block.

The ‘CreateChain‘ function initializes a new blockchain by creating a genesis block. It initializes a transaction pool with a single transaction and sets up the initial block with a hash of ”0”.The ‘AppendChain‘ function appends a valid block to the blockchain’s tail. It locks access to the blockchain using a mutex to ensure thread safety.The ‘GetChainLen‘ function returns the length of the blockchain.

As for the longest chain role, the ‘ReplaceChain‘ function replaces the existing blockchain with a longer valid chain. It checks the validity of each block and ensures that the replacement chain is longer than the current chain. To prevent race conditions, a mutex named ‘lock‘ is used to synchronize access to critical sections of code, such as appending or replacing blocks in the blockchain.

2 Improvements

We have made three key improvements in addition to our baseline implementation: Merkletree, UTXO, and serveral cost-saving strategy compared with the baseline.

2.1 Merkletree

In our enhanced blockchain implementation, we introduced a Merkle tree to optimize the structure of transaction data within each block. The Merkle tree is a fundamental cryptographic data structure that enhances the efficiency and security of data verification in a blockchain. The MerkleNode struct represents a node in the Merkle tree. It contains pointers to the left and right child nodes, along with the hash of the data it representsThis tree is built by creating leaf nodes for each transaction hash and then combining pairs of nodes until a single root node is formed.The GenerateBlock function now incorporates the Merkle tree. It calculates the Merkle root hash for a list of transaction hashes.The calculated Merkle root is included in the block, ensuring an efficient and secure representation of all transactions within the block.The IsValid function, which checks the validity of a block, can be extended to verify the correctness of the Merkle root in the block.

The introduction of the Merkle tree in our blockchain implementation provides several advantages. It enables efficient verification of transaction integrity and enhances the security of the blockchain. This improvement contributes to a more robust and reliable system for handling transaction data.

2.2 UTXO

In our blockchain system, we have initially allocated 10,000 bitcoins in the genesis block and distributed 100 bitcoins to each of the ten recipients (alice1 to alice10). This distribution is represented in the form of Unspent Transaction Outputs (UTXOs). Subsequently, for every transaction from C to A, a new transaction is created, where the input refers to the hash of the previous transaction from C to A (encrypted with C's private key), and the output is associated with B's public key.

2.2.1 Transaction Structure

A transaction consists of inputs and outputs. The input of a transaction is a reference to a previously received transaction, and the output is the public key of the recipient. During verification, only the input is considered, and the goal is to validate that A can spend the input. Each block maintains a UTXO set for efficient verification.

2.2.2 Utxoset Maintainence

Initialization: In the genesis block, 10,000 bitcoins are allocated and distributed to alice1 to alice10, each receiving 100 bitcoins. This creates the initial UTXO set.

When A initiates a payment to B, a new transaction is created with the input referencing the hash of the transaction from C to A. The UTXO set is consulted to verify that the public key associated with the input exists in the set. If not found or the content is incorrect, the transaction is deemed invalid and rejected. The verified transaction remains in the UTXO set but is marked as spent, preventing it from being used again. Spent Transaction Removal: Once a transaction is included in a block, it is removed from the UTXO set, ensuring it cannot be double-spent.

2.2.3 Signature Verification

The public key (C's public key) associated with the input is used to verify that the content of the TxIn is indeed a hash from C to A.

2.2.4 Transaction Boardcasting

After successful verification, A signs the new transaction with their private key and broadcasts it to the network. This ensures the integrity and authenticity of the transaction.

2.2.5 Code Explaination

We need to change the P2PServer to ultilize utxo. Below is a detailed explaination of the P2PServer: P2P Server Implementation: The P2P server is a crucial component of our blockchain system, responsible for managing peer connections, handling block and transaction broadcasts, and ensuring synchronization across the network. Below are the explanations for the relevant sections of the provided code:

Initialization and Local Storage

The Init function initializes the P2P server, reading the blockchain and UTXO set from local storage. The blockchain is replaced with the stored chain, and the UTXO set is loaded into the localset field.

Chain and UTXO Set Updates

The UpdateBlockChain function is invoked when the server receives a longer blockchain from a peer. It updates the local blockchain, saves it to local storage, and broadcasts the latest block to peers.

New Block and Transaction Handling

In the NewBlock function, after validating and appending the new block to the blockchain, the UTXO set is updated with new transactions from the block. The NewTransaction function validates incoming transactions against the local UTXO set and then broadcasts the transaction to the network.

Peer Management

The NewPeer function adds a new peer to the local peer list and saves the updated list to local storage. These additions to the P2P server implementation contribute to the overall functionality of the blockchain network by ensuring proper synchronization, maintaining local UTXO sets, and managing peer connections.

2.2.6 Functional Validation

We have demonstrated that with utxo strategy, one can not double-spend a previous transaction output. And it is also validated that if one try to use a irrelevant signature to spend a valid input, he/she will be denied. However since we only implement simple rpc response, and client won't be informed of the state of his or her transaction release attempt, so it is hard to put some pictures here and they are all presented in our presentation. S

2.3 Cost-saving Strategy

A further enhancement is made to our blockchain system, focusing on two key aspects: the evolution of the p2p peerlist and the implementation of a thread concurrency strategy to optimize transaction processing.

2.3.1 P2P Peerlist Transformation

In the initial architecture, each client was required to broadcast messages to all other nodes in the network, leading to potential scalability challenges. The recent improvement introduces a more efficient approach by leveraging the majority of nodes. Now, instead of broadcasting messages to every client, messages are selectively sent to a randomly chosen subset of nodes, ensuring the dissemination of information without overloading the network.

Strategy	Latency(ns)	Speedup
Baseline	1589350	1
Peerlist	1325115	1.199
Concurrency	1431910	1.109

Table 1: Improvment Strategy

Selective Majority Broadcasting

As we have learned from the lecture, the majority can ensure the consensus. The system now targets the majority of nodes rather than broadcasting to all, significantly reducing message redundancy.

Randomized Receivers

To prevent potential shutdowns or targeted attacks on specific nodes, messages are distributed to random receivers in each communication round.

Enhanced Scalability

The updated p2p peerlist strategy enhances the scalability of our blockchain, allowing for more nodes without compromising network efficiency.

2.3.2 Threads Concurrency Strategy

To augment the responsiveness and transaction processing capabilities of our p2p server, a thread concurrency strategy has been implemented. This strategic enhancement ensures that the system can efficiently handle a larger number of incoming transactions concurrently.

The introduction of a thread concurrency model allows the p2p server to concurrently process multiple transactions, reducing latency and improving overall system responsiveness.By leveraging threads effectively, the system can maximize the utilization of available resources, ensuring efficient processing of transactions even during peak loads.The enhanced concurrency strategy contributes to improved throughput, enabling our blockchain to handle a higher volume of transactions per unit of time.

2.3.3 Conclusion

The combined improvements in the p2p peerlist and the adoption of a thread concurrency strategy mark a significant leap forward in the robustness and efficiency of our blockchain architecture. These enhancements not only address previous scalability concerns but also position our blockchain for future growth and increased transactional demands. The selective peerlist approach and thread concurrency model showcase our commitment to continuous innovation and optimization in the evolving landscape of blockchain technology.