Report of Project 2: Bitcoin-like Blockchain Implementation

Beier Li, Kairui Li, Jiaze Wei

January 16, 2026

1 Implementation Overview

This project implements a Bitcoin-like blockchain system based on the Nakamoto consensus protocol. The system consists of two main components: miners (peers) that participate in the consensus protocol, and clients (wallets) that submit transactions to the chain.

Check README.md for detailed instructions on how to build and run the system. Also, the three shell scripts provided can help automate testing and performance evaluation. There usage is explained in the README.md.

1.1 System Architecture

The system is implemented in Go. Each miner process maintains its own copy of the blockchain and the associated UTXO set. The miners communicate via RPC to broadcast transactions and blocks. Clients interact with miners to submit transactions and query blockchain status.

The P2P network uses TCP-based RPC for communication. Transactions and newly mined blocks are broadcast to all peers. Nodes can request chains from peers for synchronization and apply the longest chain rule when receiving a longer valid chain. The current implementation uses a **static peer list** where all miner IP addresses are pre-configured.

There is a strange behaviour. The RPC in go seems to create new threads each time, but they are not used to parallelize the mining, but just for handling incoming requests. Too many RPC threads will crash the miner. To solve this problem, we should set a limit on the number of RPC threads. For example, `export GOMAXPROCS=4`.

1.2 Block Structure

Each block in the blockchain contains: the block index (height in the chain), a timestamp, a list of transactions, the Merkle root of transaction hashes (when enabled), a hash pointer to the previous block, the SHA-256 hash of the current block, a nonce value found during mining, the difficulty level, and the miner's identifier. The genesis block uses a special previous hash of all zeros and index 0.

1.3 Validation Mechanisms

1.3.1 Proof of Work

The PoW algorithm requires finding a nonce such that the block's hash has at least D leading zero bits, where D is the configurable difficulty. Mining starts from a random nonce to distribute

attempts across miners, then repeatedly calculates the block hash and checks if it meets the difficulty requirement. Each additional bit of difficulty doubles the expected number of hash attempts required.

1.3.2 Digital Signatures

Transactions are signed using ECDSA with the P-256 curve. Each wallet has an ECDSA key pair, where the public key serves as the wallet address. To spend a UTXO, the owner signs the transaction data with their private key. Validators verify signatures against the public key stored in the referenced UTXO.

1.3.3 Block Validation

When receiving a new block, miners verify that the block index is correct, the previous hash pointer matches, the block hash is valid, the PoW requirement is satisfied, all transactions are valid against the UTXO set, exactly one coinbase transaction exists at the beginning, and the coinbase reward does not exceed the allowed amount.

1.4 Transaction Model and Data Ownership

The system uses a UTXO (Unspent Transaction Output) model similar to Bitcoin. Each transaction has inputs (references to previous outputs to be spent) and outputs (new UTXOs created). Coinbase transactions are special transactions that create new coins as mining rewards.

Data Ownership: Each miner maintains its own independent copy of the entire blockchain and its associated UTXO set in memory. The UTXO set is derived from all transactions in the blockchain—when a block is added, the spent UTXOs (referenced by transaction inputs) are removed and new UTXOs (transaction outputs) are added. Since each miner has its own blockchain copy, each miner also has its own UTXO set. When miners synchronize and adopt a longer chain, they rebuild their UTXO set from the new chain. This distributed ownership ensures that no single point of failure exists, and each miner can independently validate all transactions.

2 Part I: Baseline Performance Analysis

2.1 Environment Setup

Performance evaluation was conducted on a Kubernetes cluster with 5 miner nodes. Each pod was configured with 4 CPU and 8 GiB memory, communicating via internal cluster networking.

2.2 Results

Table 1 shows the performance results from our evaluation, where we deployed different numbers of miners (1, 3, 5) with varying difficulty levels (15, 18, 20) and measured blocks mined in a 120-second window.

Miners	Difficulty	Blocks Mined	Blocks/Second
1	15	855	7.13
3	15	1514	12.62
5	15	1598	13.32
1	18	124	1.03
3	18	176	1.47
5	18	233	1.94
1	20	29	0.24
3	20	49	0.41
5	20	54	0.45

Table 1: Performance Results: Blocks Mined in 120 Seconds

2.3 Analysis

The mining difficulty has an exponential impact on block generation rate. Increasing difficulty from 15 to 18 (3 more leading zero bits) reduces throughput by approximately 8×, and increasing from 18 to 20 further reduces it by approximately 4×. This matches the theoretical expectation that each additional bit of difficulty doubles the expected number of hash attempts required.

Adding more miners increases the overall block generation rate, but the scaling is sub-linear. With difficulty 15, going from 1 miner (855 blocks) to 3 miners (1514 blocks) yields 1.77× improvement, while 5 miners (1598 blocks) yields only 1.87× improvement. The sub-linear scaling is due to heavy network overhead (such as synchronization) and UTXO performace bottleneck. The latter has a significant impact, and is explained in Part II.

3 Part II: Improvements

3.1 Merkle Tree Implementation

We implemented a Merkle tree data structure for efficient transaction verification. The Merkle tree is built bottom-up from transaction hashes: leaf nodes contain the SHA-256 hash of each transaction ID, and internal nodes contain the hash of the concatenation of their children. If a level has an odd number of nodes, the last node is duplicated. The root hash (Merkle root) is stored in the block header.

The implementation supports Simplified Payment Verification (SPV), allowing lightweight clients to verify transaction inclusion without downloading the entire block. A proof consists of the sibling hashes along the path from the transaction to the root. The Merkle tree feature can be enabled or disabled via a global configuration flag.

We compared block generation rates with and without Merkle tree enabled using difficulty 18 and a 60-second mining window. With Merkle tree enabled, miners generated 84 blocks; without Merkle tree (controlled by a flag), miners generated 86 blocks. The number of blocks drops because we send transactions. The difference is negligible, indicating that the Merkle tree computation overhead is minimal compared to the PoW mining cost.

The Merkle tree provides efficient transaction verification without impacting mining performance, which is difficult to test due to the dominant PoW cost.

3.2 WebUI Interface

We developed a web-based visualization interface using React, TypeScript, and Vite with Chakra UI for styling. The WebUI provides a blockchain dashboard showing real-time status, a wallet manager for creating and managing wallets, a transfer manager for submitting transactions, a block explorer for browsing blocks and transactions, and settings for configuring miner connections.

By `make environment`, we can build the environment with both backend miner and frontend WebUI. The WebUI connects to the miner's RPC endpoints to fetch blockchain data and submit transactions. More details (including figures) are in README.md.

3.3 UTXO Model for Double-Spend Prevention

3.3.1 Overview and Storage

The UTXO (Unspent Transaction Output) model is a fundamental mechanism for preventing double-spending attacks. Unlike account-based models, UTXO explicitly tracks which outputs have been spent.

Each miner stores its UTXO set as part of its local state, derived from its copy of the blockchain. The UTXO set is a mapping from transaction ID and output index to the unspent output data (value and owner's public key). When a miner processes a new block, it updates its local UTXO set by removing the spent outputs and adding the newly created outputs. Since each miner maintains its own blockchain, each miner also maintains its own UTXO set independently.

3.3.2 Double-Spend Prevention Logic

When validating a transaction, the system first checks that each referenced UTXO exists in the current UTXO set—if any input references a non-existent or already-spent UTXO, the transaction is rejected. Then it verifies that the signature on each input is valid against the public key of the UTXO owner, ensuring only the rightful owner can spend the funds. Finally, it checks that the total input value is at least equal to the total output value.

When a block is accepted, the UTXO set is updated atomically: all spent UTXOs (referenced by inputs) are removed, and all new UTXOs (outputs) are added. This atomic update ensures consistency and prevents double-spending within the same block.

3.3.3 UTXO as Performance Bottleneck

While the UTXO model provides strong double-spend protection, it has become a performance bottleneck in our implementation. Finding all UTXOs for a given address requires iterating through the entire UTXO set, as the primary indexing is by transaction ID rather than by owner address. Additionally, validating a block's transactions requires creating a temporary copy of the entire UTXO set for atomic validation, which has $O(n)$ complexity where n is the UTXO set size. What's more, when a client requests its balance, the miner must scan the entire UTXO set to sum unspent outputs belonging to that address.

The $O(n)$ complexity of synchronization leads to significant slowdowns as the blockchain grows. Miners spend increasing amounts of time managing and exchanging the UTXO set (and the entire blockchain) rather than focusing on mining new blocks. This bottleneck limits scalability and reduces the benefits of adding more miners, as they cannot effectively utilize their computational resources.

3.4 Dynamic Difficulty Adjustment

We implemented a dynamic difficulty adjustment mechanism that automatically adjusts the mining difficulty based on recent block generation rates. The target is to produce one block every 10 seconds. The adjustment is checked every 6 blocks (approximately 1 minute at the target rate).

The algorithm calculates the actual time taken to mine the last 6 blocks and compares it with the expected time. If blocks are being mined too fast, difficulty increases; if too slow, difficulty decreases. The adjustment is bounded to prevent drastic changes, and difficulty is clamped between a minimum of 1 and maximum of 32 bits.

This feature can be enabled with the `-dynamic-difficulty` flag when starting a miner. When enabled, the system self-regulates to maintain a consistent block generation rate regardless of network hash power changes.

3.5 Parallel Mining

We implemented parallel mining to utilize multiple CPU cores for faster PoW computation. The parallel mining algorithm spawns multiple worker goroutines, each exploring a different region of the nonce space. Each worker starts from a random nonce offset by its worker ID to avoid duplication, and workers increment their nonce by the total worker count to ensure non-overlapping search spaces.

Table 2 shows the parallel mining performance with different thread counts, measured with difficulty 20 over a 60-second window on a single miner node.

Threads	Blocks Mined
1	16
2	34
4	39
8	35

Table 2: Parallel Mining Performance (Difficulty 20, 60 seconds)

The results show that parallel mining provides significant speedup. Using 2 threads yields 2.1× improvement over single-threaded mining, and 4 threads yields 2.4× improvement. However, 8 threads shows diminishing returns (slightly worse than 4 threads), likely due to context switching overhead and the pod's CPU limit of 4 cores. The optimal thread count should match the available CPU cores.

4 Proposal: Known Limitations and Future Work

4.1 Static Network Topology

The current implementation assumes a static network where all miner IP addresses are known in advance. This means no support for dynamic node joining or leaving, no peer discovery mechanism, and no handling of node failures. A gossip-based peer discovery protocol could allow nodes to discover and connect to new peers dynamically.

4.2 UTXO Performance Bottleneck

As described above, the UTXO model has performance limitations due to inefficient indexing and full-copy validation. Secondary indexing by address and persistent storage with efficient key-value lookups could address these issues.

4.3 No Persistent Storage

The current implementation stores the blockchain entirely in memory. All data is lost when the miner process restarts, requiring re-synchronization from peers. Memory usage also grows unbounded with chain length. Implementing persistent storage using a database like LevelDB would solve these problems.