Blockchain Project Documentation

August 2025

Contents

1 Project Overview

This document provides a comprehensive explanation of a simple blockchain project implemented in C++, including file responsibilities, workflow, terminologies, cryptocurrency fundamentals, and practical applications of blockchain technology. The project demonstrates core blockchain concepts such as block creation, hashing, proof-of-work, and chain validation.

2 File Responsibilities & Workflow

This section outlines the roles of each file in the blockchain project and the workflow for key operations.

2.1 main.cpp - Driver / Demo Controller

- **Purpose**: Runs the program and demonstrates the blockchain in action.
- **Functionality**:
  - Creates a `Blockchain` object.
  - Adds new blocks to the chain.
  - Prints messages to show progress.
  - Validates the chain's integrity.

2.2 Blockchain.cpp - Chain Manager

- **Purpose**: Manages the vector of blocks.
- **Functionality**:
  - Creates the genesis block (index 0, data = "Genesis Block", prevHash = "0").
  - `addBlock()`: Links a new block's `prevHash` to the previous block's hash and calls `mineBlock()`.
  - `isChainValid()`: Verifies chain integrity by checking hashes and `prevHash` links.
  - Stores the difficulty value for mining.

2.3 Block.cpp - Block Blueprint

- **Purpose**: Defines the structure of a block.
- **Functionality**:
  - Structure: Includes `index`, `timestamp`, `data`, `prevHash`, `hash`, and `nonce`.

- `calculateHash()`: Combines block fields and calls `sha256()` from `CryptoUtils.cpp`.
- `mineBlock(difficulty)`: Increments `nonce` until the hash starts with the required number of zeros (proof-of-work).
- `printBlock()`: Outputs block details.

2.4 CryptoUtils.cpp - Hashing Engine

- **Purpose**: Provides cryptographic hashing functionality.
- **Functionality**:
  - Implements `sha256()` to compute SHA-256 hash from a string input.
  - Used by `Block.cpp` to secure blocks.

2.5 Headers (.h files) - Class Blueprints

- **Block.h**: Declares the `Block` class.
- **Blockchain.h**: Declares the `Blockchain` class.
- **CryptoUtils.h**: Declares hashing function prototypes.

2.6 Workflow

- **Initialization**: `main.cpp` → `Blockchain.cpp` (`createGenesisBlock`) → `Block.cpp` (`mineBlock`) → `CryptoUtils.cpp` (`sha256`).
- **Adding a Block**: `main.cpp` calls `addBlock()` → `Blockchain.cpp` sets `prevHash` → `Block.cpp` mines the block → block is pushed to the chain.
- **Validation**: `main.cpp` calls `isChainValid()` → `Blockchain.cpp` checks all hashes and `prevHash` links.

2.7 Output Mapping

- *Simple C++ Blockchain Demo*: `main.cpp`
- *Chain has 1 block(s)*: `Blockchain.cpp`
- *Added block X*: `main.cpp`
- *Block details*: `Block.cpp`
- *Valid chain? Yes*: `Blockchain.cpp`

3 Blockchain Terminologies

This section defines key blockchain concepts relevant to the project.

1. **Block**: A container holding `index`, `timestamp`, `data` (transactions), `prevHash`, `hash`, and `nonce`. In the project: Defined in `Block.cpp`, printed via `printBlock()`.
2. **Blockchain**: A linked list of blocks starting from the genesis block. In the project: Managed in `Blockchain.cpp`, which creates the genesis block, adds new blocks, and validates the chain.
3. **Genesis Block**: The first block with `prevHash` = "0". In the project: Created by `createGenesisBlock()` in `Blockchain.cpp`.
4. **Hash**: A fixed-length code from SHA-256 representing block content uniquely. In the project: Generated in `Block.cpp` via `calculateHash()`, using `CryptoUtils.cpp`
5. **SHA-256**: Secure Hash Algorithm producing 64-character hex strings. In the project: Implemented in `CryptoUtils.cpp`.
6. **Nonce**: "Number Only Used Once" — changes to find a valid hash. In the project: Incremented in `mineBlock()` until the hash meets proof-of-work.
7. **Proof-of-Work**: Rule requiring the hash to start with N zeros (difficulty level). In the project: Implemented in `mineBlock()` in `Block.cpp` with difficulty set in `Blockchain.cpp`.
8. **Difficulty**: Number of leading zeros required in the hash. In the project: Fixed value (e.g., 3) stored in `Blockchain.cpp`.
9. **Mining**: The process of finding the correct `nonce` to satisfy proof-of-work. In the project: `mineBlock()` tries `nonce` values and calls `calculateHash()`.
10. **Previous Hash (prevHash)**: Links the current block to the previous block. In the project: Set in `addBlock()` in `Blockchain.cpp` to the hash of the last block.
11. **Chain Validation**: Ensures all hashes match and `prevHash` links are intact. In the project: `isChainValid()` in `Blockchain.cpp`.
12. **Transaction Data**: The content stored in a block (e.g., "Alice → Bob: 10"). In the project: Passed from `main.cpp` to the `Block` constructor.

4 How Cryptocurrency Works

This section explains the fundamentals of cryptocurrencies and their connection to the project.

4.1 What is Cryptocurrency?

Cryptocurrency is digital money that uses cryptography for security and operates on a blockchain — a decentralized ledger maintained by many computers (nodes) worldwide. It is not controlled by any single authority and relies on protocol rules for security and trust.

4.2 Basic Flow of a Transaction

Example: Alice sends 1 coin to Bob.

1. Alice creates a transaction with sender and receiver addresses and signs it with her private key.
2. The transaction is broadcast to the network.
3. Validators check the transaction for sufficient balance and valid signature.
4. Valid transactions are grouped into a block.
5. The block is validated by miners (Proof-of-Work) or validators (Proof-of-Stake).
6. Once validated, the block is added to the blockchain.
7. Bob's balance updates to reflect the received coin.

4.3 How Coins Are Generated

- **Proof-of-Work (PoW)** (e.g., Bitcoin):
  - Miners solve computational puzzles to add blocks.
  - The first to solve gets a block reward (newly minted coins) plus transaction fees.
- **Proof-of-Stake (PoS)** (e.g., Ethereum post-upgrade):
  - Validators stake coins to be chosen to confirm blocks.
  - They earn transaction fees and sometimes small rewards in newly minted coins.

4.4 Validators

- **In PoW**: Validators are miners who solve puzzles.
- **In PoS**: Validators are stakers chosen to create blocks.
- **Role**: Verify transactions, add blocks, and maintain network security.

4.5 Why Validators/Mining Are Important

Validators prevent double-spending, ensure only valid transactions enter the blockchain, and maintain consensus across the network.

4.6 Connection to Your Project

The blockchain project includes:

- Mining with a small difficulty (similar to PoW but simpler).
- A single validator (the program itself) instead of a network.

- No coin creation — only transaction storage.

To extend the project into a cryptocurrency:

- Add balances to track ownership.
- Reward miners with coins for adding blocks.
- Enable networking for multiple validators.

5 Applications of Blockchain Technology

This section explores practical applications of blockchain technology, highlighting how the concepts in this project can be extended to real-world use cases.

5.1 Cryptocurrency

- **Description**: Blockchain enables decentralized digital currencies like Bitcoin and Ethereum, allowing secure, transparent peer-to-peer transactions without intermediaries.
- **Relevance to Project**: The project's block structure, hashing, and proof-of-work mechanisms mirror the core components of cryptocurrencies. By adding balance tracking and network support, the project could simulate a basic cryptocurrency.

5.2 Supply Chain Management

- **Description**: Blockchain provides a tamper-proof record of goods movement, ensuring transparency and traceability in supply chains (e.g., tracking food or pharmaceuticals from origin to consumer).
- **Relevance to Project**: The project's chain validation and immutable block structure could be adapted to store and verify supply chain data, ensuring integrity of transaction records.

5.3 Smart Contracts

- **Description**: Self-executing contracts with terms coded on the blockchain, automatically enforced when conditions are met (e.g., automatic payments in Ethereum).
- **Relevance to Project**: The project's transaction data storage could be extended to include executable code, enabling simple smart contract functionality.

5.4 Identity Verification

- **Description**: Blockchain enables secure, decentralized identity management, allowing users to control their personal data and verify identity without relying on central authorities.

- **Relevance to Project**: The project's secure hashing and chain validation could be used to store and verify identity records, ensuring data integrity and authenticity.

5.5 Healthcare Records

- **Description**: Blockchain can securely store patient records, ensuring privacy, interoperability, and immutability while allowing authorized access across healthcare providers.
- **Relevance to Project**: The project's block structure and validation mechanisms could be adapted to store encrypted medical records, maintaining a secure and verifiable history.