Blockchain Simulator Design Document

Contributors: Vardan Verma, Atharva Bendale, Vishal Bysani

February 9, 2025

Contents

1 Introduction

This document provides a detailed design overview of the blockchain simulator. The simulator is designed to emulate a network of blockchain mining nodes, each maintaining its own private blockchain view and interacting with peers through events. The system models network latency, block creation, transaction propagation, and consensus via chain selection based on the longest chain rule.

2 System Overview

The simulation is built to mimic real-world blockchain dynamics through modular components. Key components include:

- **Simulator:** Orchestrates the simulation, schedules and processes events, and generates network visualization.
- **Miner:** Represents an individual node that creates transactions and blocks, maintains its local blockchain copy via a `BlockTree`, and communicates with other nodes.
- **BlockTree:** Maintains the blockchain as a tree structure. Supports adding blocks, chain validation, and switching to the longest (main) chain.
- **Block & Transaction Structures:** Define the data structure for blocks and transactions. Each block contains transaction data along with metadata such as timestamps, identifiers, and references to the parent block.
- **Event System:** Uses events to drive the simulation. Events include block creation, broadcast, reception of transactions/blocks, etc. The events are handled in chronological order via a priority queue.
- **Network Topology:** A randomly generated network topology emulates real-life network latencies between miners. The system distinguishes between fast and slow nodes (with high or low CPU).

3 Architecture Details

3.1 Simulator Module

The Simulator (defined in `sim.hpp` and `sim.cpp`) is the central controller of the simulation.

- Maintains an array of `Miner` pointers and a priority queue of scheduled events.
- Initiates simulation by parsing command line arguments (through `parser.cpp/hpp`) and setting simulation parameters (e.g., total nodes, block/transaction intervals, network delays).

- Processes events such as block and transaction broadcasts, updating the state of each miner.
- Generates Graphviz DOT files for visualizing the network topology and block tree.

3.2 Miner Module

Each miner, implemented in `miner.cpp/hpp`, simulates a blockchain node with the following responsibilities:

- **Block and Transaction Generation:** Uses exponential delays (from `utils.cpp`) to simulate realistic mining and transaction timings.
- **Maintaining Local Blockchain:** Each miner maintains a `BlockTree` that represents its view of the blockchain. This includes handling forks and switching to the longest chain.
- **Event Handling:** Generates new events (e.g., block broadcasts) when a new block is confirmed or a transaction is created.
- **Peer Communication:** Selects neighbors from the global network topology (a 2D vector of network delay and capacity pairs) and broadcasts its newly mined blocks or transactions.

3.3 BlockTree and Block Structures

The BlockTree (located in `blockTree.cpp/hpp`) is used to store the miner's blockchain:

- **Tree Representation:** Each node (of type `BlockTreeNode`) holds a block, its arrival time, and pointers to parent and children.
- **Chain Switching:** Implements logic to switch to a longer branch when a new valid block is added to a fork.
- **Transaction Processing:** Updates miner balances using methods `processTransaction` and `deProcessTransactions` when blocks are added or removed.

The `Block` structure itself contains metadata like block ID, height, parent ID, timestamp, and owner.

3.4 Transaction Module

Defined in `transaction.cpp`, the transaction module describes:

- Basic information (sender, receiver, amount, transaction ID).
- The role of transactions during block creation, where miners bundle transactions and validate them.

3.5 Event System

The event system is driven by the `Event` struct outlined in `event.hpp`. The key aspects include:

- **Event Types:** Examples include `BLOCK_CREATION`, `BROADCAST_BLOCK`, `RECEIVE_BROADCAST_BLOCK`, `SEND_BROADCAST_TRANSACTION`, etc.
- **Priority Queue:** Events are scheduled based on their timestamp so that the simulation can process them in chronological order.
- **Event Propagation:** Events trigger actions in miners, e.g., a block broadcast event causes each neighbor to schedule a receive event based on network latency.

3.6 Network Topology Utility Functions

Network behavior and randomization are handled by functions in `utils.cpp` and `utils.hpp`:

- **Network Generation:** Uses a configuration model to create a 4-regular connected graph and then augments it. The resulting structure is represented as a 2D vector where each entry is a pair representing network latency and capacity.
- **Random Generators:** Uniform and exponential random generators to simulate network delays, transaction delays, and block propagation times.
- **Miner Classification:** Slow and fast ministers are randomly assigned using a permutation function.

3.7 Parser & Configuration

The simulation parameters (e.g., total nodes, transaction delay (`TTX_TIME`), block interval (`BLK_TIME`)) are specified on the command line and parsed using the parser (`parser.cpp/hpp`). This ensures that the simulator's configuration can be modified easily without changing the source code.

3.8 Build System (Makefile)

The `Makefile` provided in `Part1/Makefile` is used to compile the project:

- It sets the compiler to use C++23 and includes options for debugging (`-g`) or warnings (`-Wall`) based on flags.
- The file lists all source files and defines object file rules, making it easier to build or clean the project using standard targets (`all`, `run`, `analyze`, `clean`).
- The build process also calls ancillary tools such as Graphviz (`dot`) to generate network visualizations.

4 System Workflow

4.1 Initialization

1. The program starts in `main.cpp` where command-line parameters are parsed to initialize simulation settings.
2. A network topology is generated, and miners are classified as either fast or slow.
3. A `Simulator` object is instantiated with all the required parameters, and each miner is initialized with its genesis block.

4.2 Event Scheduling & Processing

1. The `Simulator` enters its main loop, extracting events from the priority queue (sorted by their scheduled timestamp).
2. Each event type triggers a specific processing sequence:
   - **Block Creation Event:**
     (a) A `BLOCK_CREATION` event signals the initiation of a new block by a miner.
     (b) The miner collects pending transactions and creates a candidate block.
     (c) Upon receiving this event, the miner locally confirms the block using `confirmBlock`. This step ensures the block meets all validation criteria.
     (d) If the block is confirmed, a follow-up `BROADCAST_BLOCK` event is generated to disseminate the block to neighboring nodes.
   - **Broadcasting Events:**
     (a) For `BROADCAST_BLOCK` or `BROADCAST_TRANSACTION` events, the miner computes the network latency based on the simulation's network topology.
     (b) The event is then forwarded, appearing as a `SEND_BROADCAST_xxx` event and subsequently scheduled as a `RECEIVE_BROADCAST_xxx` event at the intended destination.
     (c) Upon reception, the destination miner processes the event by adding the block to its `BlockTree` or the transaction to its mempool, with optional further propagation.
   - **Transaction Handling:**
     (a) Similar to blocks, transaction events follow a propagation sequence: first through a `SEND_BROADCAST_TRANSACTION` event, followed by latency calculation, and finally delivered as a `RECEIVE_BROADCAST_TRANSACTION` event.
     (b) The receiving miner processes the transaction by adding it to its mempool and may further relay it to its neighbors.
3. After an event is processed, the miner updates its state:
   - If a new block is successfully confirmed, it is added to the miner's `BlockTree`.

- The miner then checks if switching to the longest chain is warranted. If so, the miner updates its current chain view and rebalances its mempool as needed.

4.3 Visualization & Logging

- The simulator periodically generates DOT files for the peer network and each miner's block tree.
- These visualizations are converted to PNG images via Graphviz, aiding in debugging and analysis.
- Log files capture detailed summaries (such as total blocks generated, main chain length, and branch lengths) for each miner.

5 Assumptions and Limitations

- The simulation assumes a fixed number of nodes with parameters that determine whether they are fast or slow.
- Network delays are simulated through random distributions and may not faithfully reflect all complexities of a real-world network.
- The block validation and fork resolution logic is implemented using simple heuristics (e.g., always switching to the longer chain).

6 Conclusion

This design document describes the complete architecture and design rationale for the blockchain simulator. Through its modular design, the system cleanly separates concerns such as network topology, event processing, miner behavior, and blockchain state management, making it both flexible and extendable for future improvements.