Winter 2025 CS271 Project1

January 8, 2025

Abstract

Blockchain is a list of blocks that are linked using cryptographic hash pointers. Each block consists of multiple transactions and each transaction transfers some amount of currency from one client (sender) to another client (receiver). In a blockchain, each block may contain one or more transactions along with the hash value of the previous block.

In this first project, you will create multiple clients, each of them storing a copy of the blockchain. Each client will also maintain a copy of a banking server that keeps track of every client's balance in a balance table. A client needs to get mutual exclusion in order to complete transactions to transfer money. You should develop the application logic using Lamport's Distributed Solution to achieve mutual exclusion.

1 Application Component

We will assume three clients, each starting with a balance of $10.
A client can issue two types of transactions:

- A transfer transaction.
- A balance transaction.

When a client initiates a transfer transaction, it needs to communicate with the other clients in order to achieve exclusive access to the blockchain. If the client tries to transfer more money than it has, the server should abort the transaction.

2 Contents of Each Block

Each block in a blockchain consists of the following components:

- **Operations**: A single $< S, R, amt >$ transaction representing the sender, receiver, and amount of money transferred respectively.

- **Hash Pointer**: Is a pair consisting of a *pointer* to the previous block for traversal purposes and the *hash* of the contents of the previous block in the blockchain. To generate the hash of the previous block, you will use the cryptographic hash function (SHA256). You are **not** expected to write your own hash function and can use any appropriate pre-implemented library function. SHA256 returns an output in hexadecimal format consisting of digits 0 through 9 and letters a through f.

$$O_{n+1}.Hash = SHA256(O_n.Operation||O_n.Hash) \tag{1}$$

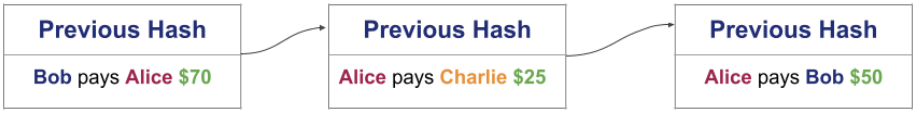

Figure 1: Structure of blockchain

3 Implementation Detail Suggestions

1. Each client has two main data structures: a local copy of the blockchain and a local copy of the Balance Table.
2. The blockchain can be implemented as an insert-only data structure stored on each client, with each block containing the components outlined above.
3. For simplicity in this assignment, each block contains only one transaction.
4. The Balance Table is a simple Key-Value store, where the key is the client name and the value is the corresponding balance.
5. Each client maintains a Lamport logical clock. As discussed in the lecture, we should use the Totally-Ordered Lamport Clock, i.e. $\langle Lamportclock, Processid \rangle$, to break ties, and each client should maintain its request queue/blockchain.
6. Each time a client wants to issue a transfer transaction, it will first execute **Lamport's distributed mutual exclusion protocol**. Once it has mutex, the client verifies if it has enough balance to issue this transfer using the local Balance Table. If the client can afford the transfer, then it inserts the transaction block at the head of the blockchain and send that block directly to all other clients, who also insert the block at the head of their local copy of the blockchain. Once inserted in the blockchain, the local copy of the Balance Table is updated. Then mutex is released. If the client does not have enough balance, the transaction is aborted and mutex is released.

7. You will need to implement **Lamport's distributed mutual exclusion protocol**, with its Request Queues and follow all its details.

8. Once a node is added to a blockchain, the local balance table is updated to reflect the transaction transfer.

9. Each time a client wants to issue a balance transaction, it will check the local copy of the balance and immediately reply with the requesting client's balance. No mutex is necessary. **Note**: No new node is added to the blockchain.

4 User Interface

1. When starting a client, it should connect to all the other clients. You can provide a client's IP or other identification info that can uniquely identify each client. Alternatively, you can accomplish this via a configuration file or other methods that are appropriate.

2. Through the client user interface, we can issue transfer or balance transactions to an individual client. Once a client receives the transaction request from the user, the client executes it and displays on the screen "SUCCESS" or "FAILED" (for transfer transactions) or the balance returned from the server (for balance transactions).

3. The client user interface should allow us to print out the client's blockchain, including the details of each block. It should also allow us to print out the balance table containing the balances of all clients.

4. You should log all necessary information on the console for the sake of debugging and demonstration, e.g. Message sent to client XX. Message received from client YY. When the local clock value changes, output the current clock value. When a client issues a transaction, output its current balance before and after.

5. You should add some delay (e.g. 3 seconds) when sending a message. This simulates the time required for message passing and makes it easier for demoing concurrent events.

6. Use message passing primitives TCP/UDP. You can decide which alternative to use and explore their trade-offs. We will be interested in hearing about your experience.

5 Demo Case

For the demo, you should have 3 clients. At startup, they should all display the following information:
Balance: $10

Then, the clients will issue transactions to each other, e.g. A gives B $4, etc. You will need to maintain a copy of the Balance Table at all clients.

6 Teams

This project must be done individually.

7 Deadlines and Deployment

This project will be due Friday 01/31/2026. We will have a short demo for each project as well as submitting to Gradescope.
The demo portion of the project will be conducted over Zoom during your time slot. The signup for demo time slots and zoom link will be posted on Piazza. You can deploy your code on several machines. However, it is also acceptable if you just use several processes on the same machine to simulate a distributed environment.
Your codebase must be submitted to Gradescope by 11:59 pm on the same day as demos.