Winter22 CS271 Project1

January 5, 2022

Abstract

Blockchain is a list of blocks which are linked using cryptography. Each block consists of multiple transactions and each transaction has money transfer from one (sender) to another (receiver). That is, when a (sender) sends money to a (receiver), that interaction is stored as a transaction. In blockchain, each block may contain one or more transactions along with the hash value of the previous block.

In this first project, you will develop a "non-replicated" blockchain that will be stored at an untrusted server known as the "blockchain master". Access to the blockchain is controlled by mutual exclusion. In other words, a client needs to get mutual exclusion over the blockchain in order to add a transaction to the blockchain. You should develop the application logic that uses Lamport's Distributed Solution to achieve mutual exclusion.

1 Application Component

We will assume multiple clients. Each starts 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 access to the blockchain. If the client tries to send more money than it has, the transaction is aborted.

2 Contents of each block

Each block consists of the following components:

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

- **Hash Pointer**: Since the blockchain is implemented as an append only data structure, the previous block is always immediately before the current block. All you need to include is the *hash* of the contents of the previous block in the blockchain. To generate the hash of the previous block, you will use the cryptograhic 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.

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

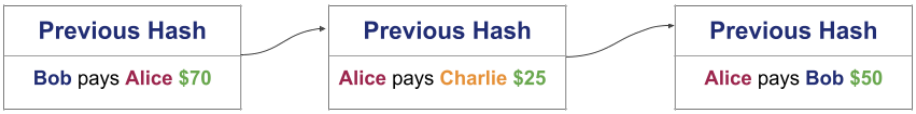

Figure 1: Structure of blockchain

3 Implementation Detail Suggestions

1. The blockchain can be implemented as a centralized append-only data structure stored on the untrusted "blockchain master", with each block containing the components outlined above.
2. For simplicity in this assignment, each block contains only one transaction.
3. Clients don't need to keep track of their current balance. They just need to know their initial balance and can check with the blockchain to figure out the rest.
4. Each client should maintain a Lamport logical clock. As discussed in the lecture, we should use the Totally-Ordered Lamport Clock, ie, $\langle Lamportclock, Processid \rangle$ to break ties and each client should maintain its request queue.
5. Each time a client wants to issue a transaction, they first execute **Lamport's distributed mutual exclusion protocol**. Once they have mutex, they send their transaction to the blockchain. There are two cases:
   (a) balance transaction: The blockchain master traverses the blockchain, calculates the balance and returns its value. **Note**: No new node is added to the blockchain.
   (b) transfer transaction: The client first issues a balance to the "blockchain master" and checks if it has enough cash to issue this transfer transaction. It also retrieves the last block in the blockchain to calculate the hash. If valid, then create a new block with this transaction to the

blockchain and send the block to the "blockchain master" who adds it to his blockchain. If not, the client should output INCORRECT and the transaction should be aborted.

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. Or this could be done 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 "INCORRECT" (for transfer transactions) or the balance (for balance transactions).

3. Through the "blochchain master" user interface we should be able to print out the blockchain.

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 for message passing and makes it easier for demoing concurrent events.

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

5 Demo Case

For the demo, you should have 3 clients. Initially, they should read from the same content file and display the following initial information:
Balance:$10
Then the clients issue transactions to each other: A gives B $4, etc. You will need to maintain each client's balance (via checking the blockchain) and display the order of transactions.

6 Teams

Projects can be done individually or in a team of 2.

7 Deadlines and Deployment

This project will be due 01/20/2020. We will have a short demo for each project For this project's demo via Zoom (sign-up sheet 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 in the same machine to simulate the distributed environment.