CS 171 : Introduction to Distributed Systems
Final Project
Due: June 7, 2019
NO LATE DEMOS

Transferring money between any two accounts is one of the most prominent applications in today's world. Instead of a centralized system of exchange, you will build a peer to peer money exchange application called Benmo, on top of a private blockchain to create a trusted but decentralized system such that transactions between 2 clients can be executed without any middle-man.

1 Theory

In this project, you will be building a simplified version of Blockchain using Paxos as a consensus protocol. Both Paxos and Blockchain provide a way to replicate the log (in Paxos) or a chain of blocks (in Blockchain) across all the participating nodes. Paxos provides a way to obtain consensus on a value for *one* entry in a replicated log, whereas, Blockchain obtains consensus on a block *eventually*, after the chain grows by a certain length. (You will learn more about Blockchain later in the class). Blockchain is usually built on top of a network where the nodes do not trust each other, but we will simplify the problem by assuming trust among the nodes in this project. Finally, in Blockchain, nobody trusts anybody else, while in Paxos, failures are all assumed to be crash.

For this project, you will be using Paxos as the method for reaching agreement on the next block to be appended to the blockchain, which is maintained by each participating node. None of the nodes are assumed to be malicious, but some might crash fail.

2 Implementation Details

We decompose this project into **three parts**.

1. The Basic Blockchain
2. Paxos for Transaction Execution
3. Handling Recovery

In the first part, you will implement the basic blockchain structure. In the second part, you will implement incorporate Paxos protocol into your blockchain to support the addition of a new block that stores transactions. Finally, you will handle the recovery of a site after it has failed.

2.1 Blockchain Settup

- The money exchange system you will build will have five clients, Alice, Bob, Carol, Devon, and Elizabeth, each with a starting *balance* of $100.
- A client, say Alice, executes *transactions* to transfer some amount of money, say $5, from herself to another client say Bob. The client then sends it to ONE of the servers.
- There will be **five** servers, each of them storing a copy of the blockchain.

Every server will maintain two data structures:

- Set of Transactions *TRANS*: Each server keeps a set of all the transactions it hears about from clients.
- Modified Blockchain is a linked list of *blocks*. This is consistently replicated across the five servers.
  - The modified blockchain is represented as a log (i.e, an array and not a linked list) and there will be **no** direct pointers to previous blocks.
  - Each block B has two major components: a **header** and a **list of two transactions**. (B-1) means the previous block.
    - The header consists of *three* components
      - current block depth
      - $H(B-1)$, which is the hash of previous block
      - The nonce
    - A List of transactions
      - txA
      - txB
  - $H(B-1)$ is the SHA256 hash of the **previous** block, ie, the SHA256 of the string concatenation of (B - 1), $H(B-2)$, nonce of (B-1) and txA and txB of(B-1).
  - txA and txB are two transactions where each transaction is a string in the format: "A B amount" (which means Alice sends Bob an amount of 5). And there are **exactly two** transactions in each list of transactions(in each block).

    NOTE:
    - Please see **Block Example** below for a visualization of a block B.
    - In the first block, $H(B-1)$ would be "NULL".
    - A block will be added to the blockchain **only if** there are two transactions (i.e. If there are odd number of transactions, then the last transaction wouldn't be added to the blockchain until another transaction comes along). Furthermore, the server must verify that the two transactions are legal, ie, the amount transfered does not exceed the balance present with respect to the local copy of the blockchain. in that node.

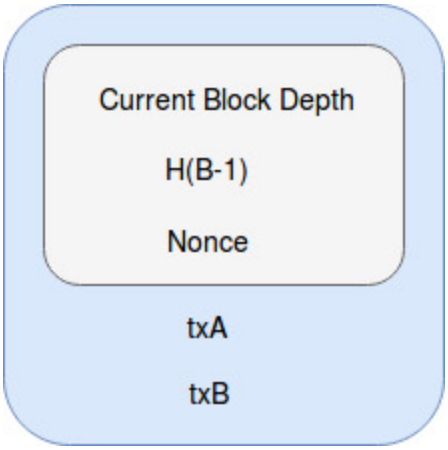

Figure 1: Block Example

The *nonce* is a random string such that: Taking the SHA256 of the **concatenation** of txA, txB of the current block **and** the nonce would give a resulting hash value with the last character being digit 0 or 1. In order to do that you will create the nonce randomly. The length of the nonce is up to you. If the calculated hash value does not end with the digit 0 or 1 as its last character, you will have to try another nonce again. In other words, you need to create the nonce randomly until the rightmost character of the resulting hash value is either 0 and 1. This is a simplification of the concept of Proof of Work (PoW) used in Bitcoin.

2.2 Paxos Protocol

When a server has created a block, it starts the Paxos protocol to insert it in the blockchain. Note that more than one server can try to become a proposer. If successful, it appends the block to its copy of the blockchain and then sends out decision messages to all the server, upon which they append the block to their blockchain. Finally, it removes the two transactions from its *TRANS*, its transaction set.

The Paxos algorithm described in class obtains agreement on a value for a *single* entry in a replicated log. Since your application needs agreement on a sequence of blocks, the Ballot number should additionally capture the depth (or index) of the block being proposed. Hence, the ballot number will be a tuple of $< seq\_num, proc\_id, depth >$. An acceptor does not accept prepare or accept messages from a contending leader if the depth of the block being proposed is lower than the acceptor's depth of its copy of the blockchain.

2.3 Handling Recovery

Your project should handle crash failures. The system should make progress as long as a majority of the servers are alive. This is guaranteed since you are using Paxos. Once a server recovers, it should be able to update its blockchain from the other servers and can take input from the user and be ready to become a leader. If a server, $\mathcal{N}$ crashes and meanwhile if the blockchain grew longer, either upon receiving a decision message from the current leader or when $\mathcal{N}$ sends a stale prepare message (lower depth value), $\mathcal{N}$ realizes that

its blockchain is not up-to-date. You are free to choose a way of updating the crashed server (either to poll the current leader or a randomly picked server to send the latest version of blockchain).

3 Delays

We recommend you add delays to your system for ease of demonstration. You can use the network process you have used earlier in PA1 and PA2, or add delays to your sockets (A delay in the message being received/sent by the socket). Any other method of adding delays that does not jeopardize the protocol's execution is acceptable.

4 User Interface

There are four types of commands that users can input:

1. *moneyTransfer(amount, client1, client2)*: The transaction transfers *amount* of money from *client*1 to *client*2.
2. *printBlockchain*: This command should print the copy of blockchain on that node.
3. *printBalance*: This command should print the balance of the 5 clients.
4. *printSet*: This command should print the set of transactions recorded at the node.

5 System Configuration

NOTE: We do not want any front end UI for this project. All the processes will be run on the terminal and the input/output for these processes will use stdio.

1. All the nodes are connected to each other. Use a configuration file with the IP and port information, so that the nodes can know about each other.
2. You should print all necessary information on the console for the sake of debugging and demonstration, for example: Committing the block , Sending proposal with ballot number A, Received acknowledgment for ballot number A etc.
3. Use message passing primitives TCP/UDP. You can decide which alternative and explore the trade-offs. We will be interested in hearing your experience.

6 Extra Credit

You may consider the following two additions to the project:

1. Network Partitioning: Paxos can tolerate network partitioning. Provide us the with necessary tools to test your project in the presence of network partitioning.

2. Data Persistence: In a real system, when a server crashes, it loses all the data it has in its memory. Hence, upon recovery, a server would need to copy all data structures from the other servers. This is overkill. Instead, all relevant data structures need to be stored on **disk**. After recovery, the server data structures are only missing all that happened during failure. For extra credit, add persistence to your implementation by storing all relevant information in a file that survives failures.

7 Submissions

You must submit all code needed to run your system on Gauchospace by **Friday June 7, 2019 at 9:00 a.m**. Along with your code, you must submit a README file that indicates the installation steps needed to get your code running on CSIL.

8 Demo

For the demo, you should have 5 servers, each representing a node in system. We will download your code from Gauchospace and use your README to install and start your program.
The demo signup sheet will be made available on Piazza. Signups for demos close on **Thursday June 6, 2019 at midnight**. No modifications to your slot will be allowed after that time. The demo will be on **Friday June 7, 2019** in CSIL. NO LATE DEMOS.

9 Teams

Projects can be done in teams of 2 or individually. You can use Piazza to form teams.