CS 171 : Introduction to Distributed Systems
Final Project
Demos: Monday December 8, 2025

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

1 Overview

In this project, you will be building a simplified version of a Blockchain using Paxos as a consensus protocol. For this project, you will be using Paxos as the method for reaching agreement on the next block to be appended to the blockchain. None of the nodes are assumed to be malicious, but *some might crash fail*.

2 Functional Specification

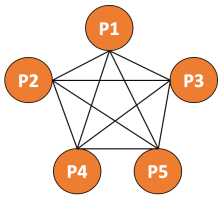

Figure 1: Processes connected to each other

The money exchange system you will build will have five nodes, $P1$ through $P5$ (as shown in Figure 1), each with a *balance* starting with \$100. You can think of each node corresponding to the account of a specific user, and each node provides an interface for the user to transfer money to another node. Every node will maintain two data structures:

- Blockchain: Blockchain is a linked list of *blocks*. This is the consistently replicated entity across the five nodes. A *block* contains one transaction. A *transaction* is a money transfer operation between a pair of nodes, i.e. <*sender_id*, *receiver_id*, *amount*>.
- Bank Accounts Table: This is a table which contains the current balance for all of the users, in this case 5. The table has a key, which is the client-id and the corresponding balance. Initially, all tables are identical, and have 5 clients-id and an initial balance of 100

2.1 Contents of each block

Each block consists of the following components:

- **Transaction**: A single transaction of the form $< sender\_id, receiver\_id, amount >$.
- **Hash Pointer**: It 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 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.

$$T_{n+1}.Hash = SHA256(T_n.Txns||T_n.Nonce||T_n.Hash) \tag{1}$$

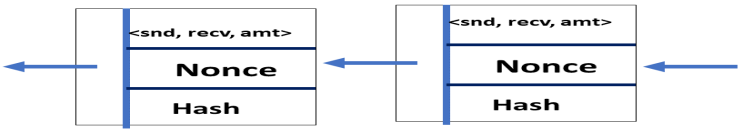

Figure 2: Contents of each block

- **A Nonce**: A *nonce* is a random string. The nonce is calculated using the content of the *current* block transaction (unlike a hash pointer, which is based on the previous block). Basically, you need to find a nonce so that when the nonce is concatenated to the transaction in the current block, the last character of the hash is a *digit* between 0 and 4.

$$h = SHA256(Txns||Nonce) \tag{2}$$

  Hence, in Equation 2, a successful nonce will produce a *digit* between 0 and 4 as the last character of h. In order to do that you will create the nonce randomly. *The length of the nonce is up to you.* If h does not end with a digit between 0 and 4 as its last character, you will have to try another nonce. In other words, you need to create a sequence of strings randomly until the right-most character of the resulting hash value is a digit between (0-4). This is a simplification of the concept of *Proof of Work (PoW)* used in Bitcoin. Although you will use Paxos for consensus, the idea of nonce increases the tamper resistance of the system. To tamper with the blockchain, an adversary will need to recalculate all hash pointers, and for each hash of the previous block, it will have to calculate its **correct nonce**. If we add more restrictions on the calculation of the nonce, the amount of computational resources the adversary needs to have will increase, hopefully prohibitively.

2.2 Protocol implementation details

When a node receives a transaction, it runs Basic Paxos, ie, it tries to become the leader and propose the next block in the blockchain to the other *acceptor* nodes. The node starts a Paxos leader election phase. Note that more than one node can try to become a leader. In this project, you are to incorporate the leader election phase of Paxos to support blockchains as follows:

1. A node intending to become the leader sends prepare messages and must obtain a majority of promise messages for its ballot number.

2. Before it can propose the block, the node should compute the acceptable hash value (the last character of the hash must be a digit between 0 and 4) by finding an appropriate nonce. Note that if a process takes too long to compute the acceptable hash, another process might time out and start leader election.

Once the node becomes a leader after performing the last two steps, it proposes the block using accept messages. After a majority of nodes accept the block, the leader appends the block to its chain, *updates the local Bank Accounts Table*. The leader then sends out decision messages to all the nodes, upon which they append the block to their blockchain and *update the balances in the local Bank Accounts Table* depending on the transaction in the block.

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 the blocks of the replicated blockchain, 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. For each new block (depth or index) in the blockchain, *seq_num* should be reinitialized. Also, for each new depth, all the Paxos parameters are also initialized for that agreement process, ie, $accept\_Val$ and $accept\_Num$ are initialized for this agreement.

If a node $\mathcal{N}$ 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 node (either to poll the current leader or a randomly picked node to ask for the latest version of blockchain).

When a node is elected to be a leader, it collects information about the depth of all the blockchains among the participants. This information is compared and the leader attempts to update all nodes based on the most up to date blockchain (longest in terms of number of blocks). This part is tricky. Keep it till the end and think about all the edge cases.

This recover algorithm is an extra credit of 10%. Please address all other tasks first and then comeback to tackle the recovery. You are free to choose any ways to recover the log of the crashed or partitioned nodes and update their corresponding Bank Accounts Table. One suggestion is to maintain in information of the first uncommitted index at both the acceptors

and the proposer (leader). The first uncommitted index is attached to the accept, accepted and decision messages. For instance, when the leader receives the accepted message from an acceptor, if the attached index is older than what the leader has, then the leader will send all necessary information to repair the blockchain of that particular acceptor. On the other hand, if the index of the leader is older than the acceptor, then the leader will request all necessary information to repair its blockchain from that particular acceptor. The full details of this recovery is left for you to discover.

2.3 Persisting blockchain to disk

To ensure persistence in the presence of failures, you will persist both the Bank Accounts Table as well as the blockchain in a file on disk. If the node is the leader, when agreement is reached on a block, the leader writes this block to a file. If the node is a participant, when it receives a block from the leader it needs to write the block to the file and tag is as *tentative*. When it receives the decide message on a block from the leader, it tags the block in the file as *decided*. Make sure that you write all the contents of the block to the file and you should be able to read that file and reconstruct the blockchain in case of node failures. Once a block is decided on disk, a transfer operation in the block is executed on the Bank Accounts Table (on disk too).

3 User Interface

The user must be able to input the following commands:

1. *moneyTransfer(debit_node, credit_node, amount)*: The transaction transfers *amount* of money from *debit_node* to *credit_node*.
   Note that we assume that the *debit_node* is always the node where the transaction is initiated and your interface should not allow the user to input a value that exceeds the balance available in that node.

2. *failProcess*: This input kills the process to which the input is provided.

3. *fixProcess*: This input will restart the process after it has failed. The process should resume from where the failure had happened and the way to do this would be to store the state of a process on disk and reading from it when the process starts back.

4. *printBlockchain*: This command should print the copy of blockchain on that node.

5. *printBalance*: This command should print the balance of all 5 accounts on that node.

4 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 directly connected to each other. You can use a configuration file with the IP and port information, so that the nodes can know about each other.

2. All message exchanges should have a **constant** delay (e.g., 3 seconds). This simulates the network delays and makes it easier for demoing concurrent events. This delay can be added when sending a message to another process.

3. You should print all necessary information on the console for the sake of debugging and demonstration. You are expected to print the nonce and the hash values once the correct hash is computed, so we can verify that your nonce results in a hash with the correct specifications. Also, print the hash pointers in the blockchain. You can also print details such as: Committing the block, Sending proposal with ballot number N, Received acknowledgment for ballot number N, etc.

5 Failure scenarios

Your project should handle crash failures. The system should make progress as long as a majority of the nodes are alive. A leader failure should be handled as described in Paxos in the lectures.

Once the node is brought back, the node should be able to update its blockchain from the other nodes and can take input from the user and be ready to become leader.

6 Test scenarios

While developing your project, you can test it for scenarios such as: sequential money transfer of different processes, concurrent transfers on different processes (multiple concurrent leaders), failing one or more processes and eventually bringing them back, partitioning the network and eventually fixing the network.

7 Demo

For the demo, you should have 5 processes, each representing a process in system. We will have a short demo. The demo will be on **Monday, December 8** via Zoom. Zoom details will be posted on Piazza.

8 Teams

Projects can be done in team of 2.