CS 171 : Introduction to Distributed Systems
Final Project
Demos: Monday June 8, 2020

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 (instead of mutual exclusion used in PA2). None of the nodes are assumed to be malicious, but *some might crash fail*.

2 Functional Specification

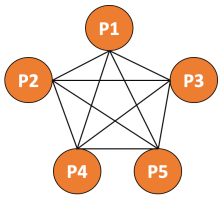

Figure 1: Processes connected to each other

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: A *transaction* is a money transfer operation between a pair of nodes. A transaction is similar to PA2, i.e. <*sender_id*, *receiver_id*, *amount*>. A *block* contains one or more transactions. Blockchain is a linked list of *blocks*. This is the consistently replicated entity across the five nodes.
- Queue: This is to store all the transactions, i.e. money transfer requests provided by the user of this node, until the set of transactions gets appended to the blockchain.

2.1 Contents of each block

Each block consists of the following components:

- **Transactions**: A set of $< sender\_id, receiver\_id, amount >$ transactions.
- **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.

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

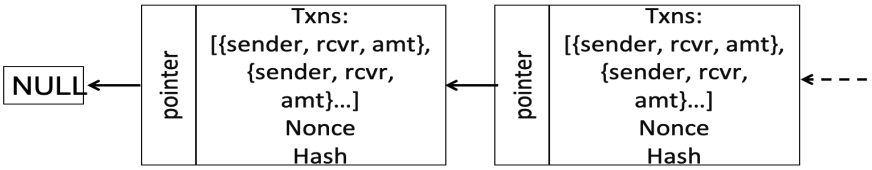

Figure 2: Contents of each block

- **A Nonce**: A *nonce* is a random string. The nonce is calculated using the content of the *current* block (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 transactions 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

Any node with a **non-empty** queue can become the leader and propose the next block in the blockchain to the other *acceptor* nodes. Each node with a non-empty queue has a random timeout. The timer starts as soon as a transaction is added to an empty queue. Upon timeout, 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 containing all the pending transactions that were stored in its queue, 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 prepare messages. After a majority of nodes accept the block, the leader appends the block to its chain, *updates its local balance*, and clears the queue. The leader then sends out decision messages to all the nodes, upon which they append the block to their blockchain and *update the balance* depending on the transactions present 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 multiple 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.

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).

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*. A node can have more than one transaction before it becomes a leader and proposes the block of transactions.
   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, i.e, assuming the current balance and all transactions in the local queue will succeed (of course, a node i might have more balance if another node j had concurrently initiated a transfer to i, but i is still not aware of that).

2. *failLink(src, dest):* This command must emulate a communication link failure between the *src* process and *dest* process. For example: if we want the communication link between P1 and P2 to fail, we will use this interface as a user input on process P1. And your program should then treat the link between the two nodes P1 and P2 as failed and hence should not send any message between the two nodes. For simplicity, links are considered bidirectional and breaking a link allows neither the *src* nor the *dest* to communicate with each other.

3. *fixLink(src, dest):* This is a counterpart of failLink input. This input fixes a broken communication link between any two processes, upon which the two nodes will be able to communicate with each other again. This input will be provided on the *src* process.

   A suggestion to simulate such a behaviour could be to use a dictionary data structure that has the network links in the system and has an active or non-active boolean. You can use such a map to check if the network link is active before sending out any message. This is one suggestion; you can use any other technique as long as the behaviour of network failure and fixing of it is emulated.

4. *failProcess*: This input kills the process to which the input is provided. You will be asked to 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.

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

6. *printBalance*: This command should print the balance on that node.

7. *printQueue*: This command should print the pending transactions present on the queue.

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. Unlike PA1 and PA2, all the nodes in this assignment 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., 5 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. The node timers that trigger Paxos should set their random timeout delays in such a way that would allow a client at the node to submit **three** or more transactions before starting leader election.

4. 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 and network 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 network is fixed or 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, June 8** via Zoom. Zoom details will be posted on Piazza.

8 Teams

Projects can be done in team of 2.

9 Afterword

In this project, we use Paxos as a consensus protocol to agree on the next block of transactions to be added to the blockchain. Alternatively, we could have used a blockchain protocol as a consensus mechanism.

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 using communication and quorums. Blockchain, on the other hand, obtains consensus on a block using computational power as in Proof of Work. (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 amongst the nodes in this project. Finally, in blockchain, nobody trusts anybody else, while in Paxos, nodes are trusted, and failures are all assumed to be crash.