CS 271: Advanced Topics in Distributed Systems
Project 3
Due: March 12, 2020 (11:59 pm)

Abstract

In this project, you will develop a distributed banking application on a set of servers. Each server is responsible to process the outgoing transactions of a single account (client) locally and maintain them on its local log. However, if an account does not enough money to initiate a requested transactions, the server will run a modified Paxos protocol among all severs to get the most recent transactions from all other servers. Once consensus has been established, each server adds a block of transaction to its blockchain.

The bank has *three* servers to keep track of all transactions made by clients. It uses a modified version of **Paxos** as its underlying consensus protocol to keep an updated transaction history stored in a blockchain on all servers to ensure proper replication, and at the same time, to ensure fault-tolerance from server crash failures.

1 Application

In this project, you need to implement a simple banking application where each client has a single account. Clients can send requests to transfer money from their accounts to other accounts. A transfer transaction (S, R, amt) initiated by some client c consists of a sender, S, a receiver, R, and an amount of money amt. The transaction is valid if c is the owner of the sender account and the account balance has at least x.

2 Example

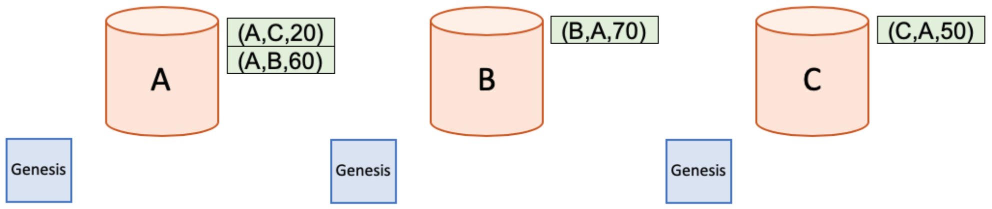

Figure 1: First Snapshot

Consider a very simple scenario with three servers A, B, and C and three corresponding clients A, B, and C where each server is responsible for processing the outgoing transactions of its corresponding client. Each client initially has 100 units in its account. Figure 1 presents the log of each server after a few transactions. As can be seen, client A has initiated two transactions $(A, C, 20)$ and $(A, B, 60)$. Client B has initiated a single transaction $(B, A, 70)$ and Client C has initiated a single transaction $(C, A, 50)$. At this point, all transactions have been executed locally and the servers are not aware of the transactions that are executed on other servers. The blockchain on each server is in its initial state, which includes a single genesis block.

Now, let's assume client A initiates a transaction $(A, B, 30)$. Server A checks the client's account and realizes that client A has only 20 units in its account. Therefore, it cannot execute the transaction locally. Server A, thus, initiates a modified Paxos protocol to get the most up-to-date information from servers B, and C. As a result of establishing consensus (is explained in more detail later), each server gets all executed transactions on other servers and as shown in Figure 2, appends a block of transactions to its blockchain.

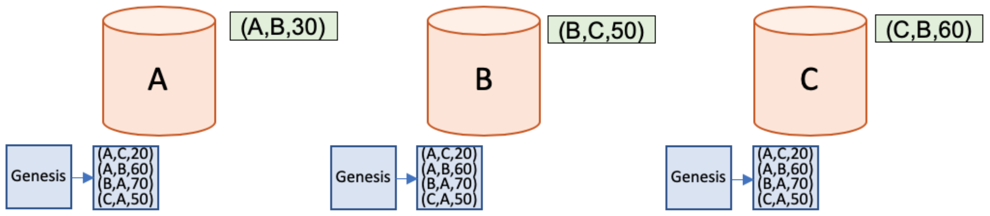

Figure 2: Second Snapshot

Now, the balance of client A is 140 and A can easily initiate transaction $(A, B, 30)$. Similarly, clients B and C also initiate transactions $(B, C, 50)$ and $(C, B, 60)$ which are executed locally on servers B and C respectively.

At this point client B sends transaction $(B, A, 60)$, however its balance is 40 and server B cannot process the transaction. As a result, server B will initiate consensus protocol among all servers to get the most recent list of transactions. As shown in 3, as a result of consensus, a block of transactions will be added to the blockchain and server B can also execute the requested transaction.

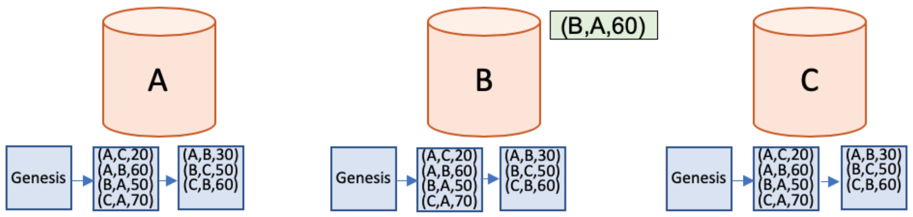

Figure 3: Third Snapshot

3 Implementation Details

We assume there will be **three** servers where each server processes the outgoing transactions of a single client.

Each client sends its requests to the corresponding server. Each request is a money transfer (S, R, amt) from the account of sender client S to receiver client R. There will be two situations: (1) If the balance on the client account is at least amt, the server logs the request, and executes the transfer locally. (2) Otherwise (when account balance $< amt$), the server runs a modified Paxos protocol among all servers to get the most up-to-date transactions from other servers and then updates its blockchain copy.

The blockchain is represented as a linked list where each block B consists of a *sequence number* and a *list of transactions*.

3.1 Modified Paxos protocol

Paxos is a *leader-based* approach to consensus. We explain the protocol in three main parts: (1) Leader Election, (2) Normal Operations, and (3) Node Failure.

In the first part, you will implement the leader election of the Paxos protocol. In the second part, you will implement the normal operations of the Paxos protocol along with the blockchain as a database to store transaction information. Finally, in the last part, you will deal with the leader failure of Paxos protocol while at the same time, supporting normal operations. Here we will assume f us the maximum number of possible failures that can be tolerate, i.e., $f = 1$.

Part I: Leader Election

Follow the Paxos leader election protocol so that among these three servers, there will be *exactly one leader*, and *two followers* after the leader election stage. A server tries to become the leader only if it wants to execute a requested transaction (S, R, amt), however, the balance of account S is less than amt.

Part II: Normal Operations

Once a server S becomes a leader, it multicasts an accept message $\langle \textsc{accept}, n, TX(S) \rangle$ to all other servers where n i the sequence number and $TX(S)$ is a list of the transactions that were executed locally on server S but have not appended to the blockchain.

Upon receiving a valid accept message from the leader, each server R sends an accepted message $\langle \textsc{accepted}, n, TX(R) \rangle$ to the leader where $TX(R)$ is a list of the transactions that were executed locally on server R but have not appended to the blockchain.

The leader logs all accepted messages. Once the leader receives f accepted messages from different servers (plus itself becomes $f + 1$, a majority of the servers), it collects all $TX(S), TX(R), ...$ within the accepted messages and also its accept message, constructs block B, and multicasts a commit message $\langle \textsc{commit}, n, B \rangle$ to every server. The leader, then appends block B to its blockchain ledger, executes the requested transaction, and sends a reply to the client. Upon receiving a commit message from the leader, each node appends block B to the blockchain ledger. The client also waits for a valid reply from the leader to accept the result.

Part III: Node Failure

A failed node might be a follower (non-leader) or a leader. If a follower fails, the program should still be able to perform normal operations. In case of leader failure, Paxos will begin a new term with leader election and then it should be able to perform normal operations. Note that once a node fails, its copy of the blockchain might become out-of-date.

4 Demo Case

For the demo, you should have 5 clients where each server is responsible to process the transactions that are initiated by a single client. Your program should first read a given input file. This file will consist of a set of triplets (S, R, amt). Assume all clients start with 10 units. Then the clients initiate transactions: $(A, B, 4)$, etc. Each client sends its transactions to the corresponding server.

- Your program should be able to **prompt user** for standard input of transactions.
- Your program should have a *PrintBalance* function which prints the balance of a given client.
- Your program should have a *PrintLog* function which prints the log of a given server.
- Your program should have a *PrintBlockchain* function which prints the current blockchain.

NOTE:

1. We do not want any front end UI for this project. Your project will be run on the terminal and the input/output for the demo will use `stdio`.
2. 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 Deadlines, Extension and Deployment

This project will be due March 12. We will have a short demo for each project om Friday March 13. For this project's demo, 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.