Design Document

-By Sushant Chaudhari

Project 1 : Advanced Operating Systems

Problem statement: To implement a centralized multi user concurrent bank account manager.

Server code(server.cpp):

The server code takes port as an argument. Commands used to compile and run the server code are:

Compile: gcc -pthread -o server server.cpp

Run: ./server 8080

The server.cpp is the server side code of this application. The server runs indefinitely, can handle requests from any client at any time. Basically, it creates sockets to accept requests from multiple clients through these sockets. Theses sockets are then used to send and receive data. Threads are used to handle multiple connections simultaneously. In my code, the server assigns a new thread to each client to handle its request through a connection handler. All the threads share the same memory. To avoid any deadlock and handle concurrency control mutex locks are used to protect the variables from the critical section. Initially the server loads the records from the records.txt file. A structure named records is created which has three fields viz. 'Account number', 'Name' and 'Account balance'. Each time a record is fetched from the records.txt file it is decoded and each string token is assigned to its respective variable from the structure. The array of structures store all the records from the records.txt file. The server code has three important functions which are explained as below:

Withdraw(): When a server decodes an instruction received from the client and identifies it as a withdraw request it assigns the request to the withdraw function. The withdraw function then matches the account number from the instructions with the account numbers stored in the array of structures. If there is a match it checks if the requested amount is less than the available account balance. If all the conditions are met it applies a mutex lock, deducts the amount from the account and updates it in the array. It then unlocks the mutes and the transaction is completed. Acknowledgement is sent to the client after completion of the transaction.

Deposit(): When a server decodes an instruction received from the client and identifies it as a deposit request it assigns the request to the deposit function. The withdraw function then matches the account number from the instructions with the account numbers stored in the array of structures. If there is a match it applies a mutex lock, adds the amount from the transaction to the account and updates it in the array. It then unlocks the mutes and the transaction is completed. Acknowledgement is sent to the client after completion of the transaction.

Intrst_calc(): This function is used to calculate the interest for each account. When a server receives a request from the interest_client code to update the accounts with the interest amount it assigns it to the intrst_calc() function. This function then iterates over all the account numbers in the array structure and calculates the interest amount using the interest percentage. The interest amount is then added to the accounts respectively. If any account has zero balance, then no interest would be calculated for that account.

Formula used to calculate interest:

Account_balance = Account_balance + (Account_balance*Interest_percentage)

Client code(client.cpp):

The client code takes hostname, port and the transactions.txt filename as an argument. Commands used to compile and run the code are:

Compile: gcc -o client client.cpp

Run: ./client hostname 8080 transactions.txt

The client.cpp is the client side code of the application. This code first establishes a connection with the server using the socket. Once the connection is established it calculates the number of transactions in the transaction.txt file and sends it to the server side. Transaction type can be a withdrawal or deposit request. The transactions are then fetched from the transactions file one at a time. The client maintains its timestamp in a counter variable which is incremented using usleep(1000000). The wait time is passed as an argument to the client code which is multiplied to the usleep to calculate the relative timestamps for each transaction. If the timestamp of a transaction is greater than the timestamp of the client it waits until its timestamp is equal to the client's timestamp. Once both timestamp values are equal the transaction is issued to the server. Similarly, all the transactions from the transactions.txt are processed and sent to the server.

Client_interest code(client_interest.cpp):

The client interest code takes hostname and port as an argument. Following commands are used to compile and run the code:

Compile: gcc -o client_interest client_interest.cpp

Run: ./client_interest hostname 8080

The client_request code is like the other client code in terms of connecting to the server and sending its request. This code sends the interest percentage to the server side on a regular interval. Sleep() is used to maintain the interval between the request. It receives an acknowledgement from the server each time a request is processed.

Scope for improvement:

The design can be further improved by using vector clock instead of a counter clock on both client and server side to have a better synchronization. The interest calculation method can be further improved by using the rules used in real time banking applications. The failure handling mechanism should be improved to have a reliable connection. Security features can be implemented to secure the transaction. We can use a MD5 algorithm to decode the transactions.