Name: Pankaj Tekwani
Banking Server Application Project

Purpose:

This design document is intended to explain the architecture of Banking server applications along with the function api's, correctness and scalability of the program.

Client:

The client reads the transaction file which is has the list of all the transaction requested by the client. The client then stores all the transaction data in the dynamically allocated fixed size array. In addition it stores a dummy record with A/c id = -1, to keep track of the end of the records. Each transaction in an array is a C structure which stores Timestamp, Account No, Transaction Type (Withdraw or Deposit) & Amount. Following is the syntax of the Transaction record:

```
struct tsn_record
{
        int id;
        int time_st;
        char ttype;
        float amt;
}tsn_record;
```

Dataset for Transaction Record:

Id	Integer
Timestamp	Integer
Transaction Type	Character
Amount	Float

This same transaction record is used by the server program to read array of transactions from the client socket. Hence this C structure is defined in "common.h" header file. The client sends the transaction one by one based on the timestamp of the transaction. It then waits for the status of the transaction sent. Once it receives the status it will then prints it on the terminal.

Server:

On the server end, the server reads the records file which has the list of all the account details like Account No, Account Name & Current Balance. The server stores all this details in a dynamically allocated fixed size array. In addition it stores a dummy record with Account id = -1, to keep track of the end of the records. Following is the Account details structure:

```
struct acc_record
{
        int id;
        char name[200];
        float bal;
}acc_record;
```

Dataset for A/C Record:

Id	Integer
Name	Character Array
Balance	Float

The server then prepares the listen socket on which it will attempt to listen the socket connection from the client. After the connection is accepted the server creates one thread per client. The thread runs parallely on the server side and perform all the transactions one by one as sent by the client. Appropriate locks are applied on each transactions in order to prevent any race conditions. After each transaction is performed the respective thread then sends the update message to the respective client regarding the status of the transaction. Once, all the transactions of one client is performed the thread closes the client connection.

The server also spawns a Interest thread when it starts. This thread adds the interest balance to all the accounts periodically.

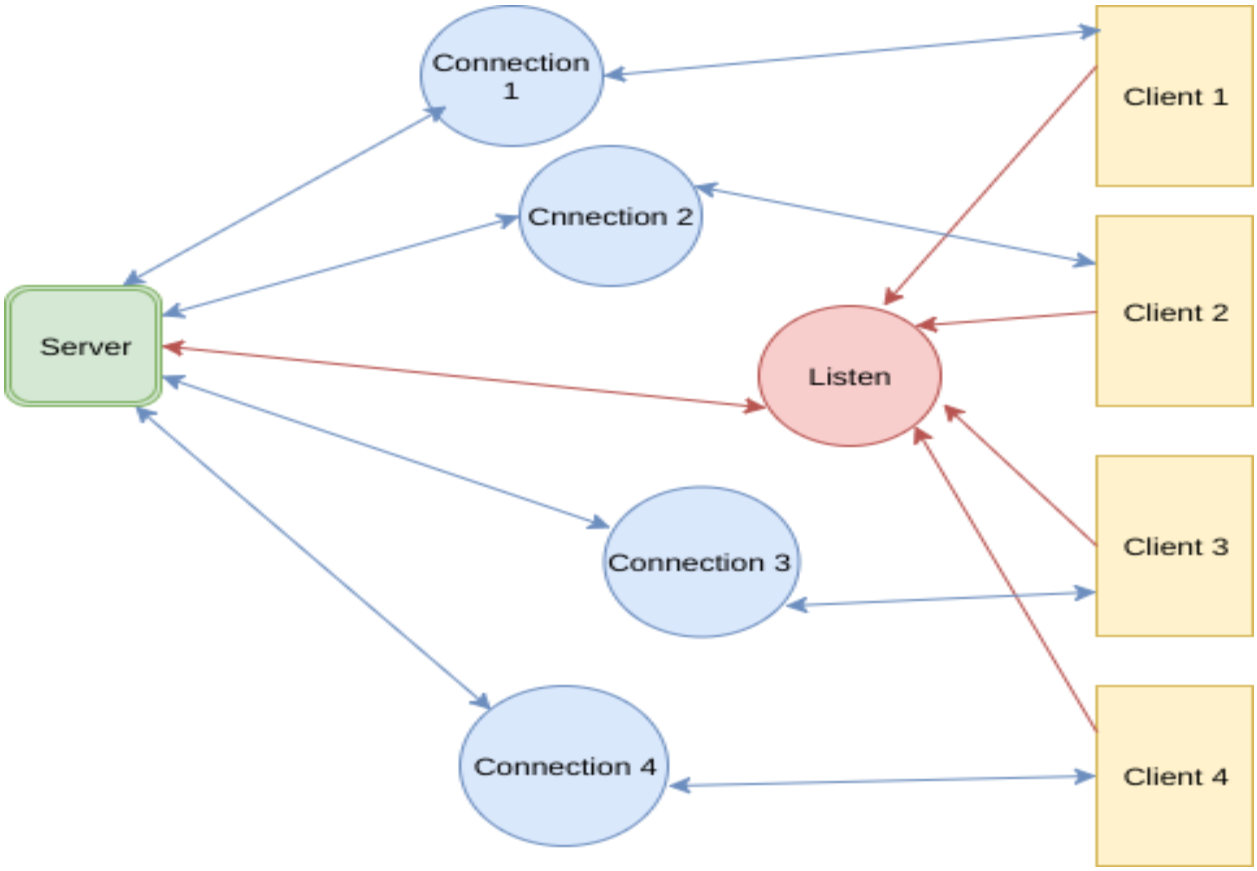
The concurrency over the account records is maintained using the Mutex lock. So, each thread who wants to update the A/C records at the server end, be it transaction request or interest update has to acquire a mutex lock of that A/c, before performing any updates on the A/C records. This way the server maintains the atomicity of the update operation performed on the A/C records.

Product Design Constraints & Tradeoff

1. The application is designed confirming the requirement of the problem statement. So, no functionality exists in the application to create a new account for the customer.
2. The request from the client is limited to “withdraw” & “deposit ”. No functionality to review Customer A/c balance.
3. The server maintains its own data structure in main memory after reading the records file. So, it does not modify the records file. It simply updates the data structure and send

update to the client. Thus, if any point of time server crashes, restarting the server will erase all the previous transaction updated. It will start fresh.

Block diagram:



Functions:

Server Side:

void perform_tsn(struct acc_record acc, struct tsn_record tsn, char *msg)*	Function to perform one transaction of the particular Client. Passing A/C records (acc), Transaction (tsn) to be performed and the char * (msg) to retrieve the details of the transaction performed.
*void * perform_cli_tsn(void *arg)*	Function to perform all transaction of single client. After accepting the connection from each client the Server creates a thread for the client connection and assigns this function to it to perform all transaction of that client.

*void * add_interest(void *arg)*	Function to add interest to all the accounts after a certain interval. When the server starts it creates the Interest thread after preparing its account table. So, this function runs parallely in the infinite loop and add interest to the account after a fixed interval. This function has used locks to prevent any race condition.

Client Side:

*struct tsn_record * getRecords(char file[], int *totalTransactions)*	This function reads the transaction file and prepares the array of structures to pass it to the server in order to perform transactions

Commands:

First of all go into the directory where all the source code is present and then,

- To make *.o files and type:
  -> make

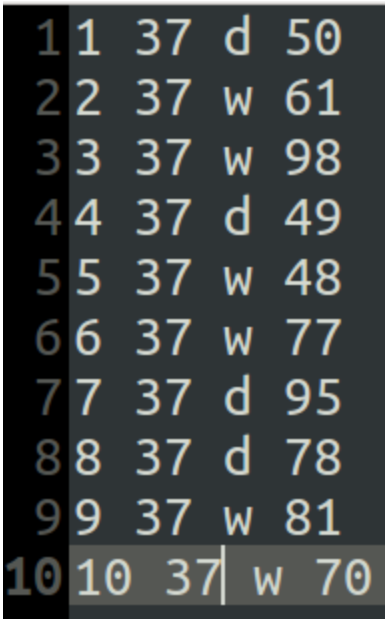
- To clean *.o files type:
  -> make clean

- To start a server, type the following:
  -> ./server <Records.txt> <Port No.>

- To start a client, type the following:
  -> ./client <Transactions.txt> <IP Address> <Port No.>

Concurrency Control and Correctness

Pthread mutex locks are used to prevent race conditions in the data. Each account has its own lock. So, when any thread tries to modify any account, it has to acquire lock of that particular client in order to prevent any race conditions. Following is the snapshot of 10 transactions run simultaneously on one particular account.



10 Transactions fired simultaneously from 2 clients.

Client 1 & Client 2 transactions status respectively:

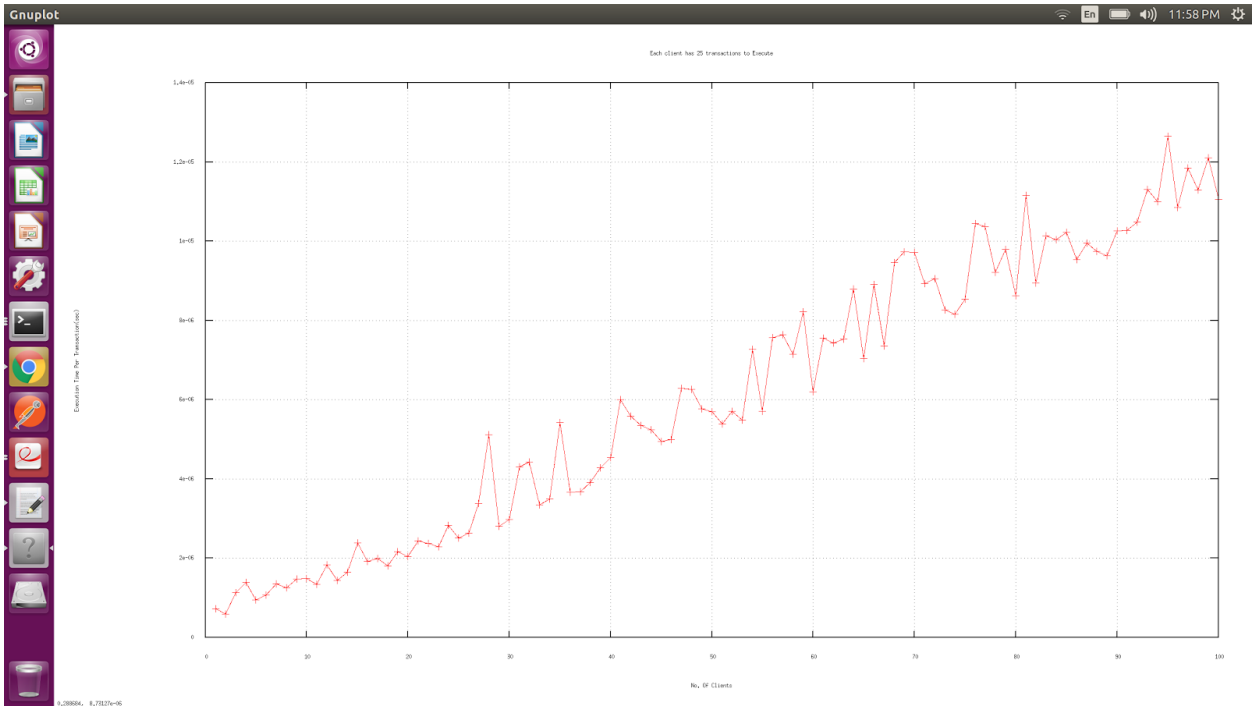
```
1
2 Connection Accepted
3 [0.000015]secs  A/c No:  37, Dpst Amt:  50, New Bal:201.00
4 [0.000019]secs  A/c No:  37, Wthdrwn Amt:  61, New Bal:190.00
5 [0.000020]secs  A/c No:  37, Wthdrwn Amt:  98, New Bal:31.00
6 [0.000019]secs  A/c No:  37, Dpst Amt:  49, New Bal:80.00
7 [0.000013]secs  A/c No:  37, Wthdrwn Amt:  48, New Bal:81.00
8 [0.000012]secs  A/c No:  37, No Suffient Bal! Rqstd:  77,Curr Bal:33.00
9 [0.000017]secs  A/c No:  37, Dpst Amt:  95, New Bal:128.00
10 [0.000027]secs  A/c No:  37, Dpst Amt:  78, New Bal:301.00
11 [0.000029]secs  A/c No:  37, Wthdrwn Amt:  81, New Bal:298.00
12 [0.000017]secs  A/c No:  37, Wthdrwn Amt:  70, New Bal:147.00
13
14 Total Time for All Transactions: [0.000188] secs
```

```
1
2 Connection Accepted
3 [0.000045]secs  A/c No:  37, Dpst Amt:  50, New Bal:251.00
4 [0.000051]secs  A/c No:  37, Wthdrwn Amt:  61, New Bal:129.00
5 [0.000055]secs  A/c No:  37, No Suffient Bal! Rqstd:  98,Curr Bal:31.00
6 [0.000014]secs  A/c No:  37, Dpst Amt:  49, New Bal:129.00
7 [0.000016]secs  A/c No:  37, Wthdrwn Amt:  48, New Bal:33.00
8 [0.000016]secs  A/c No:  37, No Suffient Bal! Rqstd:  77,Curr Bal:33.00
9 [0.000022]secs  A/c No:  37, Dpst Amt:  95, New Bal:223.00
10 [0.000021]secs  A/c No:  37, Dpst Amt:  78, New Bal:379.00
11 [0.000026]secs  A/c No:  37, Wthdrwn Amt:  81, New Bal:217.00
12 [0.000023]secs  A/c No:  37, Wthdrwn Amt:  70, New Bal:77.00
13
14 Total Time for All Transactions: [0.000289] secs
```

From the above snapshots we can see that the transactions are fired simultaneously on the same A/C 37 and **no race condition** has occurred. From the above snapshots we can also figure out that when the requested amount greater than account balance, the server aborts the transaction and and replies with “no sufficient balance!!”. Hence, **data accuracy** is maintained in all the accounts.

Scalability

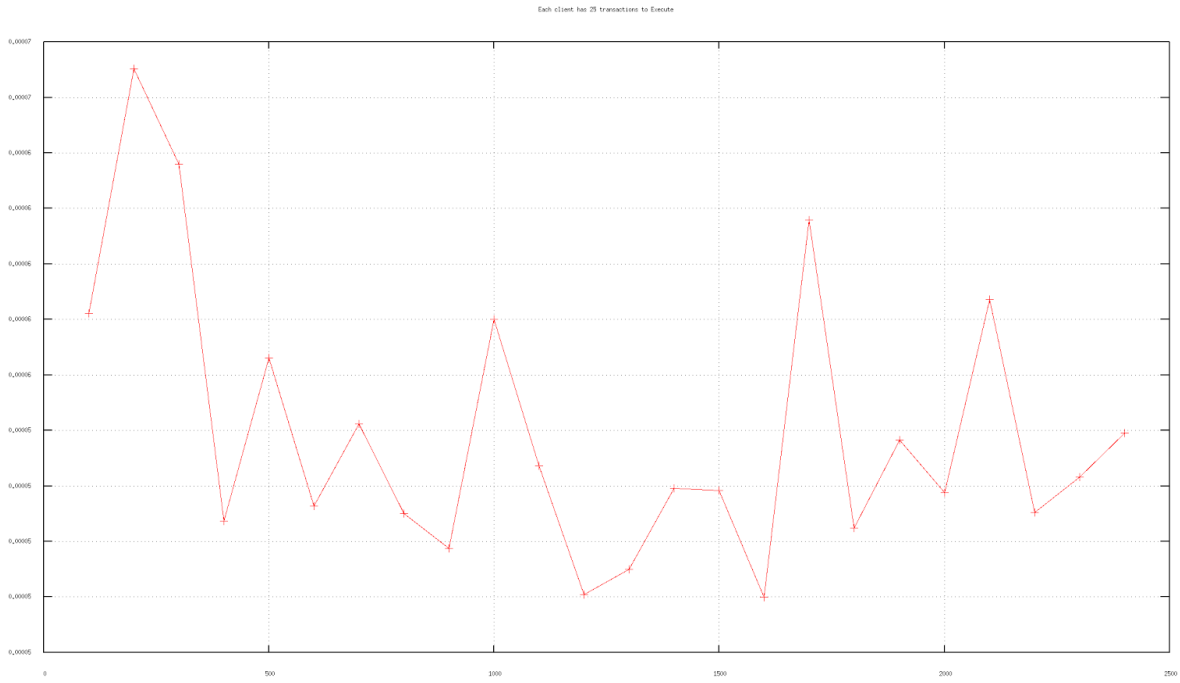
Graph 1 is plotted against **No. of clients on X axis (From 1 to 100)** and **Execution time per Transaction(sec) on Y axis.** Each client has 25 transactions and from the graph it is visible that the execution time per transaction is increasing as the no. of client increases.



X Axis: No. of Clients

Y Axis: Execution time per transaction

Graph 2:



X Axis: Time Stamp Differences (msecs)
Y Axis: Execution time per transaction (sec)

Graph 2 is plotted against **Transaction Timestamp Difference on X axis** and **Execution time per Transaction(sec) on Y axis.** As per the above graph, we can say that as the time interval between the request increases, the execution time per transaction decreases. The sudden spikes in the graph can be accounted for the interest service thread which can lock the accounts and can delay the transaction threads.