OS PROJECT 1 DESIGN

Centralized Multi-User Concurrent Bank Account Manager

Author: Nikhil Kumar Mengani

CLIENT SIDE:

Client is used to send the requests of the transactions to the server. Each client has the Transactions.txt file, in which there are bunch of transactions to be performed. Transaction requests from the clients are sent one by one. Client establishes the connection with the server by creating a socket.

For each transaction record in the file, new socket is created and connects to server to send that transaction details. The maximum transaction record length I considered as 255 bytes and used that on the server side also. The two operations that can be done by the client are withdrawal and deposit. Client waits till the it receives the acknowledge from the server about the transaction.

SERVER SIDE:

Server accepts the transaction requests from the clients and processes them. It can handle the multiple requests from the clients concurrently. The details of all the account holders are maintained at the server side in the file Records.txt. Every line in this file corresponds to each account holder information like account no, balance etc. Whenever the transaction of the customer is completed, the balance of that customer is updated accordingly in the file. So, the record length of each account holder may vary as per the transaction. If I use the same format of the Records.txt, since the records are stored in the sequential locations in file, then there is the possibility that all the records get messed up. So, I preprocessed the Records file by using the other temporary file. For this, I read the records in the records file and then fix the max length of the record as 128 bytes and then add the new line '\n' character after that. This is only the initial setup of the server. By doing this, I can avoid my file getting messed up after updating the records in the records file after the transaction has been requested.

Core Design of the server:

The server should be able to process more than one requests from clients concurrently. To accomplish this, I am maintaining a queue and using the threads to process the requests. Whenever the new request from the client hits, the socket id of the request is pushed into the queue. When the server is started, it spawns 20 threads. These threads wait for the requests to be pushed into queue. As soon as the it appears, one of the 20 threads will pop the socket id from the queue and processes that transaction. Having 20 threads all the time, will save lot of time in handling requests. Queue is always in work in accepting the requests from clients.

Here queue acts as producer and threads as consumers. I dealt the feature of processing concurrent requests by turning this into producer and consumer problem. So, there are conditional variables. Queue signal the pool of threads as soon as request is hit and threads wait till that. When queue becomes full it waits till one of the threads in the pool consumes the request.

When one of the thread consumes the request, it signals the queue saying that it can accepts new requests into it.

- **Thread pool:**

  The whole processing of the single transaction is completed in the thread. The main thread in the server pushes the request into queue. Now the thread receives the record information through the socket id it takes from the queue. It gets the information, on which account it must perform the operation. Traversing through the file with help of account number, it reaches to the record on which the operation must be performed. Now we should check which operation should be performed on the record like withdrawal or deposit and process it accordingly. But while withdrawing, the account will be checked if it has enough balance. If there is no enough balance, the message is sent to the client. After completion of the transaction, the message is sent to the client saying that transaction is completed and the socket will be closed.

- **Synchronization using mutex locks:**

  At the start of the server, for every record in the Records file, I assigned unique number. This information is stored in Hash map where the key is the account number of that record and the value is the number assigned to the record.

  While different threads try to access the same record in file and try to read while the other one is writing or both trying to write, then there is the problem of the consistency of that account balance. To solve this problem, I used the mutex locks on the records. For every record in the file, I have the lock number associated with it. The thread reading the record locks that record using its lock number from the Hash map information. Then if there is other thread trying to access the same record, it checks whether the lock number associated with it is locked or not. Since there is already the lock on that record, it waits till the record gets unlocked. The same thing happens if more than one thread tries to manipulate the same record.

SCALABILITY:

The design that I implemented seems to scale incredibly well. From testing my implementation using the large number of clients each having decent number of transactions, I observed server handling the requests quite well.

IMPROVEMENTS AND EXTENSIONS:

In the present design, I am not using any kind of cache. It can help in reducing the amount of time in processing the account number in the Records file. Average time for the transactions can be drastically reduced by implementing the cache buffer. And, while searching for the account

number, I am using linear search in the current design. Implementing the binary search, we can reduce the time in fetching the record. These are the improvements that can be embedded to the current design.

As an extension, implementing the feature to add new account number to the server database can be done. If some account holder takes the loan and server should be able be track that.

RUNNING THE PROGRAMS:

Server : **./server <port_no> <records_file>**
Client : **./client <server_ip_address> <port_no> <transaction_delay> <Transaction_file>**

I am printing the server messages and client messages on to the console directly. To view them properly please write them in to the file using ">>" operator while running the program.