CMSC 621

Project 3: Two-phase commit protocol

Anurag Pawar (LA22171)

- **Requirements specifications**

The goal of the project is to design a fault-tolerant baking system with three backend servers and a frontend server to act as a mediator between backend servers and clients. Request from the client will be sent to the backend server to process via the frontend server. The frontend server will decide whether to commit or abort the transaction based on the responses from the backend servers. The frontend server will convey the decision to backend servers and if it is a commit decision then they will process the transaction.

- **Program Design**

The project consists of two files.

1) *back_server.cpp*
2) *front_server.cpp*
3) *client.cpp.*

The frontend server will accept the connection from the client and process the transaction request with the help of the backend server. The frontend server will be connected to three backend servers. Once the client requests a transaction, the frontend server will ask backend servers to vote for either commit or abort. The backend server will make the decision and convey it to the frontend server If one of the servers says abort then the final decision will be aborted otherwise it will process it.

- **Code Overview**

The connection between client and the both types of the server is achieved with the help of *UNIX-based TCP sockets*. Following socket, programming functions are used for communication purposes.

socket(): To create a communication endpoint and will return a file descriptor.

accept(): To accept new connections.

bind(): To assign the address to the socket.

listen(): To make server to listen (look for) connection requests.

connect(): Used by the client to connect to a server.

To handle multiple clients, *POSIX threads* are used. Each client is handled by a thread. Following commands are used for threads.

pthread_t : Declaration of thread.

pthread_create : Creation of thread function.

pthread_join : Command to wait for a thread to finish.

For avoiding race condition locking mechanism is used with *lock* variable, commands as follows:

pthread_mutex_lock(&lock): to apply the lock

pthread_mutex_unlock(&lock): to release the clock

setsockopt(): To set the current value for a socket option associated with a socket of any type, in any state

The flow of the code

1. The three backend servers will start first.
2. Then frontend server will start and it will connect to all three backend servers.
3. Lastly, the client will start and it will request the transaction.
4. The frontend server will receive the transaction and ask backend servers for a vote to commit or abort.
5. If any of the backend servers says abort then the transaction will be aborted and the message will be conveyed to the client as well as the backend server.
6. If all three backend server says commit then the frontend server will send the actual transaction request to backend servers and they will process it.
7. Then the backend servers will inform to frontend server that they have processed the transaction and the frontend server will inform the client the same.

- **Compiling instructions**

1. Run the make file.
2. The backend server will start first. Run *'./back_server'* to start 3 servers with command-line arguments as:
   './back_server 54000', './back_server 54001', './back_server 54002'
3. Then frontend server will start with the command: *'./front_server'*
4. Then the client will start with the command: *'./client'*

NOTE: Please try again if you see 'Transaction aborted', as I have used random function to take decision in backend server's code it might result into number of ABORTs.

- **Two phase commit protocol**

It has two phases, first is prepare and second is commit

Prepare phase

1. Each database store (slave) sends a "DONE" notification to the transaction coordinator once it has completed its transaction locally. When the coordinator receives all of the slaves' messages, it sends them a "PREPARE" message.
2. Each slave replies to the "PREPARE" message by sending the coordinator a "READY" message.
3. The coordinator sends a global "ABORT" message to all slaves if a slave answers with a "NOT READY" message or does not react at all. The coordinator considers the entire transaction aborted only after obtaining acknowledgement from all slaves that the transaction has been aborted.

Commit phase

1. When the transaction coordinator receives the "READY" message from all slaves, it sends a "COMMIT" message to them all, which contains the transaction details that must be put in the databases.
2. Each slave completes the transaction and sends the coordinator a "DONE" acknowledgment message.
3. When the coordinator receives a "DONE" message from all of the slaves, it deems the transaction complete.

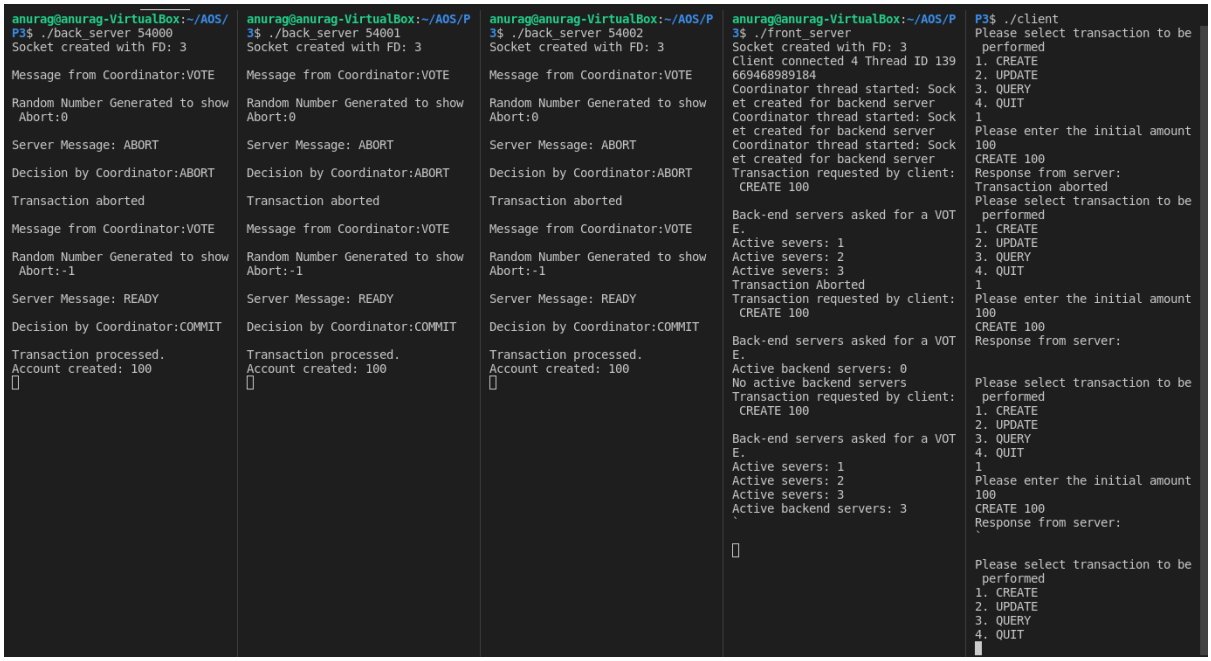
- **Tradeoffs and possible improvements**

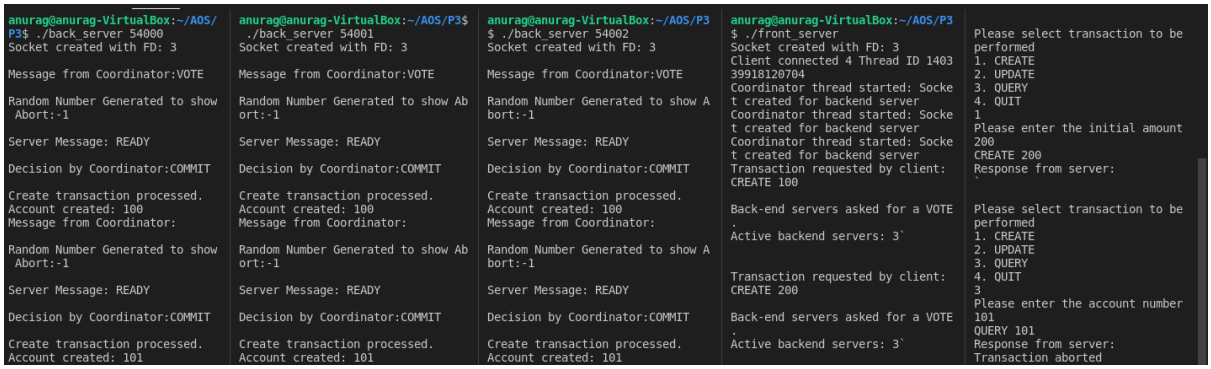
The code will work for one client. To handle multiple clients, the code of the frontend server needs to be changed. An array of threads is required while accepting the client connection so that the code will be able to handle the number of clients.

- **Thoroughness of evaluation**

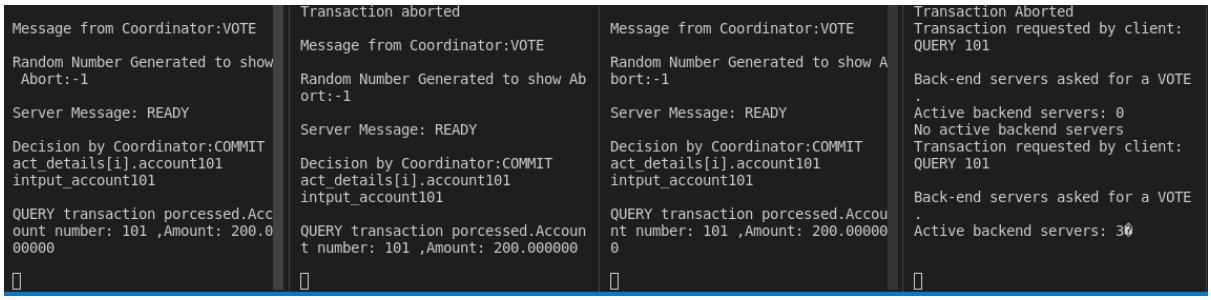
Code has been tested with one client, screenshot as below:

Creating an account



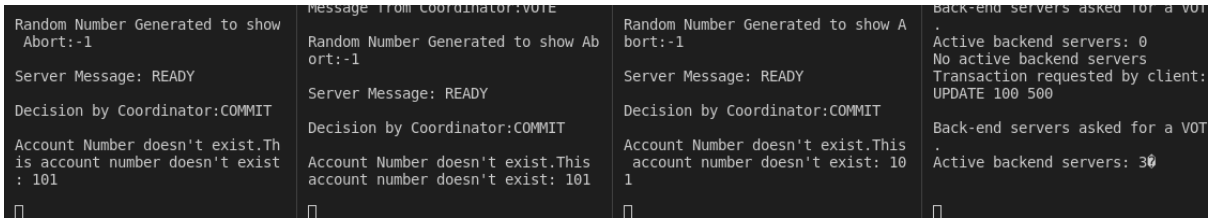


Query



Update:

1.



2.