CMSC 621

Advanced Operating Systems

Ajinkya Wakhale
AN75014
Project3 Report-

We were asked to implement distributed banking service that permits multiple concurrent operations and exhibits fault tolerance.

So, total there are total three backend servers. In my case the program name for backend servers is **server1.C** And, there is a client program called **client.c** which connects to a front-end server called **front_server.c**.

Flow is like that; first client will connect to front end server which in turn will connect to all the backend servers. Both front end servers and Backend servers are multithreaded. For each client, front end server will create a thread and will assign a handler called **createthread** which will work for that specific client. So, for two phase commit, in my program front_server is like coordinator, and we are assuming that it will not fail.

Once the client connects to front-end server, that specific thread will connect to all backend servers. Backend server's ports are hardcoded in my program. So, for total three servers, ports are 7868,7869, and 7870.

So, once client connects to front-end server, it will create three sockets and these three sockets will make TCP connection will all the backend servers. Once all the connections are successful, the thread at frontend server will receive transactions one by one from client. In my case, I am sending transactions from screen. So, one by one you must write transaction on client and front end server will receive that transaction.

I handling three cases of fault tolerance in my project. Firstly, we are assuming that there are total three backend servers, so a case in which one of the servers are not available at initial stage only, coordinator will assume that there are total 2 servers available and will continue to work in a proper fashion.

Secondly, I am handling the case when before voting, one of the servers gets crashed, i.e before **INIT** stage.

So, for two phase commit, first front end server will send a **VOTE** request to all the three servers. This is a **INIT** stage. It sends **VOTE** to all the servers and if those servers are active, they will send back the **COMMIT** message back to front end server. If the front-end servers get the commit message from all the back-end servers, it will send them **GLOBAL COMMIT** message. If the count of **COMMIT** message received at front end server is less than total number of backend servers, it means that server is not available and server got crashed before **INIT** stage only, and it will send **GLOBAL ABORT** to the remaining servers which are available and will abort the transaction as not enough votes are received. So, for the query for which it happened, the program will give technical error, as it is a **GLOBAL ABORT,** but next transactions will be executed properly. So, we can test this case in our project in two ways. Before writing a new transaction at client, you can close a backend server, so for this transaction, it will give you a message, it will give you a message, saying that it is technical error. Second, after transaction is written on client, you can put a sleep before sending voting request, and then check the transaction message response.

In third case, if front end server receives all the **COMMIT** votes and then one of the servers got crashed, it means that server moved from **INIT** to **READY** state, then at that time one of the backend servers will never know the outcome of whether it was **GLOBAL COMMIT** or other. So, if at **READY** stage one of the servers got crashed, it will send **GLOBAL COMMIT** to remaining available servers, which will execute the instruction and will send back the updated result to the client. So, even though one of the servers got crashed, but remaining servers will make sure that, the transaction is executed at the time when they are in **READY** state. It won't work when the coordinator fails.

So, after getting all the votes, if I close one of the backend servers, still transaction will get executed. This is how, we are implementing fault tolerance and ensuring atomicity via 2-phase commit. To test we can put a sleep after receiving commit from all backend servers and then try to crash one server. Even though one of the server got crashed, other servers will make sure that instruction is executed and the result is sent back to the client.

Once all the operations are successful, backend server will read the transaction and will update its database. In my case, I am taking database as an array of structures. So, each server will have a unique array of structures which is global and its values will change as per the transaction. Even for multiple clients, values will change as that array is global.

As there is update transaction and create transaction, which client can send, we must use locking to maintain consistency. So, before each update operation and create operation, I am putting lock, which will make sure that only one thread access that

critical section and others wait. This way, we are maintaining consistency. For example- if at two clients there are two transactions UPDATE 1 200 and UPDATE 1 300. At run time at backend server, the thread which comes later will update the balance for account number 1. Later, if you try to give transaction at both client as QUERY 1, value will be 300, assuming second thread comes later. Other transactions are simply read and hence locking does not matter.

Multiple clients can access backend servers and front end servers simultaneously. For each send and receive from front end server, I am checking how many servers are available, when state is READY, because we need to commit transaction once the coordinator comes to READY state. So, after READY state, I will commit transaction no matter what, to the remaining servers. GLOBAL ABORT will happen only, when servers are not available at the INIT stage.

So, for example, in client

./client 7867 localhost

OK
Create 100
Ok 1
Query 1
Ok 100
Now, if you crash one of the servers, before writing transaction or even after writing a transaction, it will give technical error.
Technical Error: Please try again later.
Now again, if you write,
Query 1
OK 100
Query 1
Now let's say, after receiving votes from all backend servers, if one of the servers gets crashed, the transaction will execute properly. But for the next transaction, it will abort, but remaining transactions will work properly.

QUIT

OK