MP3: DISTRIBUTED TRANSACTIONS

Submission Information

Name of student: Junda Feng

Netid of student: jundaf2

Github Repo url: https://github.com/JundaFeng/mp3-distributed-transaction.git

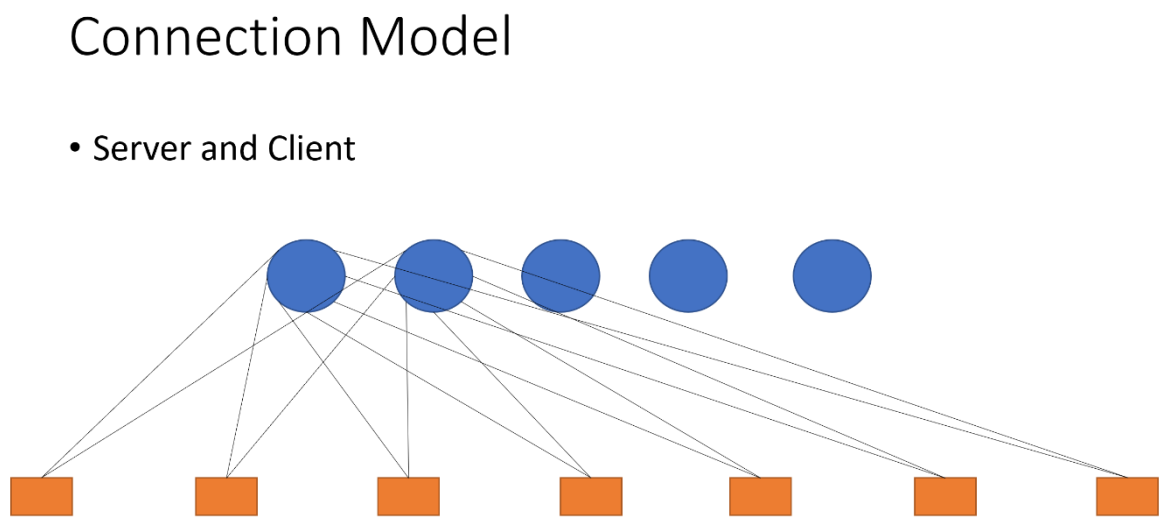
Executable is in the 'build' folder.

Or you can create a new build folder and type 'cmake ..' and then 'make'.

Design Description and Configuration

In this MP we implemented a distributed transaction system that represents a collection of accounts and their balances. The goal is to support transactions that read and write to distributed objects while ensuring full ACI properties.

To communicate across the network, we use the blocking duplexing TCP/IP protocol to use the same socket file descriptor to send and receive. The basic network connection model is shown below.



The accounts are stored in five different branches (named A, B, C, D, E). An account is named with the identifier of the branch followed by an account name; e.g., A.xyz is account xyz stored at branch A. Account names will be comprised of lowercase english letters. Because a server that represents a branch need to keep track of all accounts in that branch, we choose to

1. use the server name as the identifier to tell which server a client should communicate to when it deicides to commit the transaction
2. use per account read-write lock and two-phase locking to ensure serial equivalence and increase parallelism. Since the GCC compiler on the cluster does not support C++17, which supports std::shared_mutex, we will implement read and write lock by ourself using the primitive std::mutex and std:: condition variable.

Because each transaction contains arbitrary accounts distributed in 5 servers and at most one transaction from one client can be executed on the server at same time, we use the client identifier (a string) to identify the current transaction of the client on the server.

Atomicity

Transactions should execute atomically. In particular, any changes made by a transaction should be rolled back in case of an abort (initiated either by the user or the server) and all account values should be restored to their state before the transaction.

We choose to roll back by counting and reversing the amount of balance change the transaction to be aborted has made to the permanent account-balances.

Consistency

Consistency thus can be checked at the following two points:

1. WITHDRAW or BALANCE: a transaction should not reference any accounts that have not yet received any deposits in a WITHDRAW or BALANCE command
2. COMMIT: at the end of a transaction no account balance should be negative. When a user specifies COMMIT any balances are negative, the transaction should be aborted. However, it is OK for accounts to have negative balances during the transaction, assuming those are eventually resolved.

Isolation

We choose to guarantee the serializability of the executed transactions suing two-phase locking. This means that the results should be equivalent to a serial execution of all committed transactions. Aborted transactions should have no impact on other transactions.

To support concurrency between transactions that do not interfere with each other while guarantee the serializability of the executed transactions. If two transactions are serialized, one may need to wait for a message from another to proceed. Since account modifications happen at the serves, we should also maintain the account objects each of which owns a two-phase read/write lock.

Server

Each server must take two arguments. The first argument identifies the branch that the server must handle. The second argument is a configuration file. E.g.: ./server C config.txt

The configuration file has 5 lines, each containing the branch, hostname, and the port no. of a server. The servers need to both send and receive messages from the clients. The configuration file provided to each server is the same, the content of which is listed below.

```
A fa21-cs425-g59-01.cs.illinois.edu 1234
B fa21-cs425-g59-02.cs.illinois.edu 1234
C fa21-cs425-g59-03.cs.illinois.edu 1234
D fa21-cs425-g59-04.cs.illinois.edu 1234
E fa21-cs425-g59-05.cs.illinois.edu 1234
```

When a client is trying to connect, each server accept and launch a new thread to handle the transaction messages (RPCs) from the client. Each thread owns a message queue with a mutex for inserting and popping the queue. Each thread who parsing messages and pushing into the queue also owns another thread that can be cancelled and detached again (when an ABORT RPC comes in).

Every time a server commits any updates to its objects, it should print the balance of all accounts with non-zero values. Other than the above, the servers should not print any additional messages.

Client

A client takes two arguments – the first argument is a client id (unique for each client), and the second argument is the configuration file (same as above), which provides the required details for connecting to a server when processing a transaction. E.g., ./client asdf config.txt

At start up, the client should automatically connect to all the necessary servers and start accepting commands typed in by the user. The clients need to both send and receive messages from the servers. The user will execute the following commands:

- BEGIN: Open a new transaction and reply with “OK”.
- DEPOSIT server.account amount: Deposit some amount into an account. Amount will be an unsigned integer type. The client will need to send the deposit amount to the server and inform the server that this is a ‘write’ operation. The account balance should increase by the given amount temporally. If the account was previously unreferenced, it should be created with an initial balance of amount. The client should reply with OK to the user.
- BALANCE server.account: The client needs to reach to the identified server for the permanent account information. This message will inform the server that this is a ‘read’ operation. As soon as the client get the information, it should display the current balance in the given account. If a query is made to an account that has not previously received a deposit, the client should print to the user ‘NOT FOUND, ABORTED’ and abort the transaction.

- WITHDRAW server.account amount: Withdraw some amount from an account. The client will need to send the withdraw amount to the server and inform the server that this is a 'write' operation. The account balance should decrease by the withdrawn amount temporally. The client should reply to the user with 'OK' if the operation is successful. If the account does not exist (i.e, has never received any deposits), the client should print to the user 'NOT FOUND, ABORTED' and abort the transaction.
- COMMIT: Commit the transaction, making its results visible to other transactions. Not until this commit operation at the server can the server make the temporal modifications permanent. The client should reply either with 'COMMIT OK' or 'ABORTED', in the case that the transaction had to be aborted during the commit process.
- ABORT: Abort the transaction. All updates made during the transaction must be rolled back. Because the transactions that are not committed or committed with failure are temporal modifications, 'roll back' can be doing nothing. The client should reply with 'ABORTED' to confirm that the transaction was aborted.

Rules

There are some rules that also need to be followed and our solutions to these rules:

- Ignore any commands occurring outside a transaction (other than BEGIN). We choose to implement this by using two string parsing while loops, the outer one is for identifying BEGIN and the inner one is for other commands. After getting the response of COMMIT and ABORT from server and output response to the CLI, we break inner loop and continue the outer loop.
- Assume that all commands are using valid format.
- A transaction should see its own tentative updates. Whether updates from other transactions are seen depends on whether those transactions are committed and isolation properties.
- If an operation requires a lock, you should not reply until the lock has been acquired. However, the user may type ABORT before getting a response and you must abort the transaction immediately. This requirement let us decide to use a separate thread to send the ABORT RPC to the server at the client end and to receive messages from the corresponding client and perform abortion operations at the server end in addition to the threads that perform non-abortion operations. Specifically, we can cancel a current thread and roll back its transaction and relaunch the thread again to perform the immediate abortion operation. We are not canceling the thread for receiving messages but the thread that perform the transaction operations. To be more specific, we use a method close to 'dynamic parallelism' to launch and cancel thread on the fly. Here, we choose to use the function sleep() as the so-called cancellation point.
- Suppose a transaction creates an account (by issuing a deposit on it for the first time), and the transaction gets aborted, the account will be considered non-existent by a future transaction. To achieve this, we use another account-balance pair to record the previous state of the accounts and balances. When doing roll-back, those the previous transactions had will be rolled back while others are erased.

- Spontaneously abort a transaction while waiting for the next command from the user. The user will need to open a new transaction using BEGIN. We use a queue to receive and store user's commands. The thread for receiving commands and the threads for processing commands are separate. We treat each user command message as RPC in json format, which is easy to convert and interpret back and forth. We launch a thread with mutex to push RPC into the RPC queue as soon as we get the user input from the CLI. In the main loop of main thread, we process these RPCs and send them to the corresponding server and wait the server to respond. All RPCs except for the ABORT RPC are processed in the same sequence they enter the queue. If any of the RPC is ABORT, we choose to send this abort RPC immediately and pop all the other RPCs before the this ABORT (inclusive) in the RPC queue at client side without further processing of the remaining queued ones.