ECE 428/CS 425 Distributed Systems
Spring 2020
MP 3
Cluster: g22
URL: https://gitlab.engr.illinois.edu/asahdev2/ece428
Revision Number: bb34fca8d42334bc9f37d160468f4f45bdbcbaa6

Navid Mokhlesi, navidm2
Amber Sahdev, asahdev2

To build the code run the following within the MP3 directory of the repository

```
> make all
```

After running the makefile, we can run a branch/server as follows (not necessary):

```
> ./branch [node port number]
```

To run all 5 services (A, B, C, D, E) at once, we have designed a launcher service in go which can be launched as such:

```
> ./launcher
```

We fetch the service/branch address and ports from MP3/branchAddresses.txt

You can then launch a client with the following command:

```
> ./client
```

System Design:

1. **A walk-through of a simple transaction that clarifies the roles that the clients, servers, and coordinator (if any) play; i.e., what messages are sent, what state is maintained by which of the nodes, etc.**
   In this design our client acts as the coordinator. See answer to Question 3 for justification.

   To start a transaction, you run ./client and input into the terminal “BEGIN”. Any input that’s not between a BEGIN and ABORT/COMMIT is discarded. User has an option of entering one of the 4 following commands: DEPOSIT, BALANCE, WITHDRAW, ABORT. For DEPOSIT/WITHDRAW, the client sends the corresponding message to the branch over TCP and this client’s connection acquires the necessary write lock on the branch side, as well as recording the net balance change map (for rollback), and also updates the current account balance. Similarly, for BALANCE, the client’s connection on the branch side acquires a read lock (unless already acquired a write or read lock). On the client side how we verify if we’re in the middle of transaction, or waiting on reply from the branch is by using flags. We have two flags, one that shows whether we are in a valid transaction (a transaction that has been started by BEGIN but not COMMITed or ABORTed yet), and another flag that tells us if we’re waiting on the branch to reply, if this wait flag is set we discard any input by the user unless it’s an ABORT.

   For COMMIT we have a 2-step way of committing transactions. First the client sends all the branches a CHECK message, if all the branches reply with an OK, we then send all the branches a COMMIT message and print COMMIT OK.

2. **A detailed explanation of your concurrency control approach. Explain how and where locks are maintained, when they are acquired, and when they are released. If you are using a lock-free strategy, explain the other data structures (timestamps, dependency lists) used in your implementation.**
   Each branch leverages go’s concurrency on a per client (transaction) basis so that all of these transactions can occur in parallel when there is no data conflict. A branch maintains a global map of account name to account balance and lock. The lock is a custom written lock that allows for shared reading and exclusive writing with no writer starvation. Additionally the reason we simply couldn’t use the default golang RW mutex was because to have shared 2PL we needed to have a function that allowed for promotion from reader to writer lock without releasing the read lock. We achieved this in our custom RW-locking mechanism. Write locks are acquired for Withdraw and Deposit requests and read locks are acquired on balance requests. Locks are released after a transaction is rolled back (after an abort) or when a transaction is committed.

3. **A description of how transactions are aborted and their actions are rolled back. Be sure to mention how you ensure that other transactions do not use partial results from aborted transactions.**

   Upon a branch receiving an Abort Message from the client, all locks are released and rollback of transactions occurs. Upon receiving a transaction commit from the client, the acquired locks are released and no rollback is required. The set of acquired locks are

tracked by each transaction thread, and the account balance changes are tracked as well so that in the case of an abort, we can rollback the transactions. A check message is also sent by the client when a client desires to actually commit a transaction, and only if all branches reply with COMMIT OK (no negative balances) does the client send the commit message to all the branches. If a branch has negative balances it will reply with ABORTED instead of COMMIT OK and the client will tell all the other branches to abort instead of commit (since, in our system design, the client also doubles as the transaction coordinator, but in a design where security is concerned, it would be a simple matter of using a netcat frontend to decouple the client from the coordinator).