ECE 428 MP3 Report

Name: Qihao Wang, Xianfan Gu
Netid: qw2, xianfan2

Repository URL:https://gitlab.engr.illinois.edu/qw2/ece428-mp3.git
Version number: 3d769dd5f281667fd1f407eb3b516d9f2324441e

We use vm06 (port 3000) to run the coordinator server, vm01-05 (port 3000) to run the participants server, and vm01-10 (port 4000) to run the clients.

- Instructions for building and running your code

Copy the file from the git repository.

Start coordinator vm06 (in coordinator folder):
“go run coordinator.go 3000”

Start servers vm01-05(in server folder):
“go run server.go 3000”

Start clients vm01-10(in client folder):
“go run client.go 3000 4000”

Follow the above steps to start the nodes

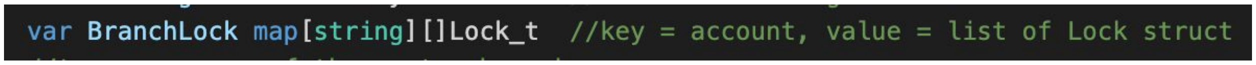
The client node will print the OK reply when adding the transaction information, and print the transaction id when BEGIN the transaction. BALANCE command will print the balance of the account on the terminal. The servers will print the transaction request when receiving the requests from clients. Both the servers and the coordinator will print the connection state when then connections are built up.

The walk-through of a simple transaction that clarifies the roles that the clients, servers, and coordinator.

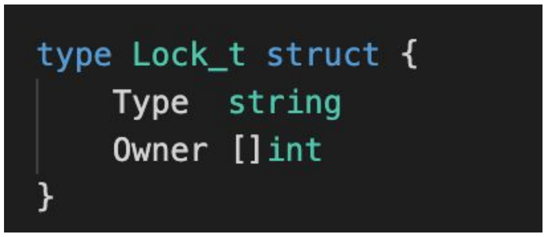
In the clients terminal, the user should type BEGIN before to execute the transactions and the coordinator would assign a transaction id to the client when the client BEGIN the transaction. After the client gets the transaction id, it will notify all servers there is a new transaction. For every transaction, it (which include WITHDRAW, BALANCE, DEPOSIT) sends directly to designated servers and the wrong format of the commands cannot be sent to the participant servers and coordinator. The servers would send new commands to the Coordinator once they receive the request from the client. For DEPOSIT command, it will try to acquire a write lock at first and then add balance for the specific account. Once it finishes, it will reply “OK” to the client. For WITHDRAW and BALANCE requests, it will check whether this account has been deposited before at first. If not, the server will reply “NOT FOUND” and abort the transaction. Otherwise, it will acquire the lock(read lock for BALANCE request and write lock for WITHDRAW request) and process the request. COMMIT and ABORT would send directly to the coordinator, for the COMMIT command, the coordinator will send the requests to all connected servers. In the meantime, the servers should check the rationality of the transaction; i.e., the balance of the account should be a zero or positive number. The commit command works if the coordinator received all votes True and replies to the client with “COMMIT OK”, the coordinator should send COMMIT to the servers, so the server will make the commit decision and make the transaction visible by adding the temporary balance value from the temporary memory into the balance. If some of the replies are False, then the client should reply with False, and the coordinator should send ABORT to the servers to let all servers abort the decision and roll back the transactions, the transactions in the temporary memory will be removed. Once the client receives the ABORT request, it will notify the coordinator immediately to abort the transaction.

A detailed explanation of your concurrency control approach. Explain how and where locks are maintained, when they are acquired, and when they are released.

For the concurrency control approach, here we use R/W 2 Phase Locking approach. To implement the R/W 2PL, for each branch, we create a BranchLock hashmap for storing the locks from different accounts. The key for BranchLock is the account name and the value are the type of the lock and a list of transaction-id try to acquire the lock.



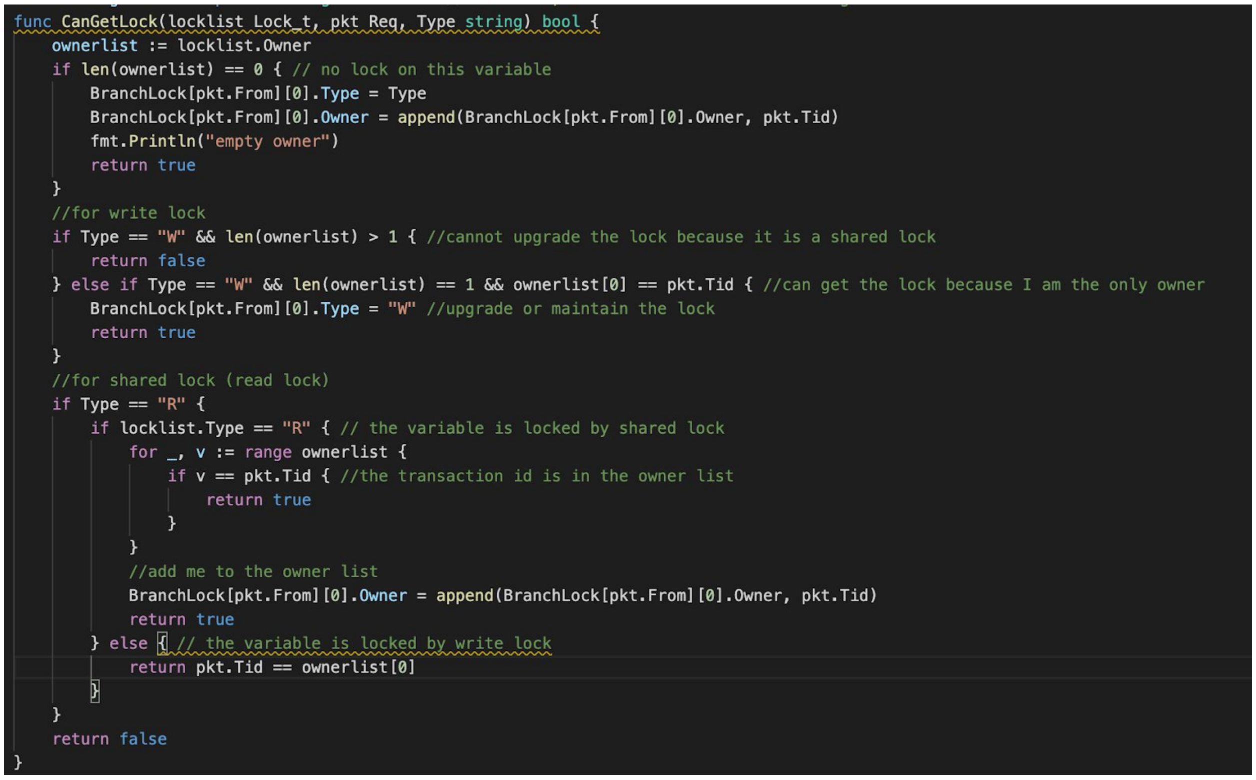
```
var BranchLock map[string][]Lock_t  //key = account, value = list of Lock struct
```



```
type Lock_t struct {
    Type  string
    Owner []int
}
```

When the client sends a read command(BALANCE) to the server, the server would query the locklist of the designated account, if the first lock in the locklist is read lock, since the lock is a shared lock, so we can push the transaction to this lock and execute the transaction in the temporary memory. If the first lock is write lock, the pending command can still execute if the command is in the same transaction of the first lock. Otherwise, the command will be pushed into the next readlock in the lock list. When the client sends a write command(DEPOSIT, WITHDRAW) to the server, the server would query the locklist of the designated account, if the command is the only pending transaction in the locklist, then the server create a write lock of promote the lock to the write lock(if the transaction is existed in the locklist) for the transaction and push it at the beginning of the locklist. If the command is not in the first pending transaction then the write command would be pushed to the end of the lock list and waiting until the former locks are released. The lock will be released as the client type a COMMIT command and ready to commit the transactions, or ABORT command to abort the transaction. Then the locks of the transactions will be popped from the lock list. When the readlock is released, the server needs to check which transaction in the read locks can be promoted to the write locks so as to run the next transactions. To avoid the race condition, for each of these global variables, we will have a RWlock for each of them.

The following picture is the lock to check whether the request can get the lock:



```go
func CanGetLock(locklist Lock_t, pkt Req, Type string) bool {
    ownerlist := locklist.Owner
    if len(ownerlist) == 0 { // no lock on this variable
        BranchLock[pkt.From][0].Type = Type
        BranchLock[pkt.From][0].Owner = append(BranchLock[pkt.From][0].Owner, pkt.Tid)
        fmt.Println("empty owner")
        return true
    }
    //for write lock
    if Type == "W" && len(ownerlist) > 1 { //cannot upgrade the lock because it is a shared lock
        return false
    } else if Type == "W" && len(ownerlist) == 1 && ownerlist[0] == pkt.Tid { //can get the lock because I am the only owner
        BranchLock[pkt.From][0].Type = "W" //upgrade or maintain the lock
        return true
    }
    //for shared lock (read lock)
    if Type == "R" {
        if locklist.Type == "R" { // the variable is locked by shared lock
            for _, v := range ownerlist {
                if v == pkt.Tid { //the transaction id is in the owner list
                    return true
                }
            }
            //add me to the owner list
            BranchLock[pkt.From][0].Owner = append(BranchLock[pkt.From][0].Owner, pkt.Tid)
            return true
        } else { // the variable is locked by write lock
            return pkt.Tid == ownerlist[0]
        }
    }
    return false
}
```

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.

For processing the aborted transactions, we create a temporary branch hashmap for storing the temporary balance of the transaction. For the DEPOSIT and WITHDRAW requests, it will add/subtract money from the temporary branch. For the BALANCE request, it will check the balance in the master branch and temporary branch. As the client sends transactions to the server, the server will process the transaction and update the balance in the temporary hashmap with the transaction id key. Once the servers receive the ABORT request from the coordinator, or the temporary balance will cause false balance value when receiving the COMMIT request from the coordinator, the server would abort the temporary balance by querying with the transaction id from the temporary hash map. Meanwhile, the transaction lock stored in the Branchlock would also be released and the command waiting on the lock can quit if it finds its transaction id is aborted. Since the temporary branch merges to master branch if and only if we can commit successfully, the aborted transaction cannot change the balance in the account, and the aborted transactions are rolled back from the account.