MP3 Report

Instructions for Building and Running the Code

The Java version we use is **OpenJDK 19**.
We have already packaged *server.java* and *client.java* into *server.jar* and *client.jar* and converted them to compiled executables as *server* and *client*, stored in *test/src*.

Package an application into a Jar (Optional)

If you use IntelliJ IDEA, follow the instructions here
https://www.jetbrains.com/help/idea/compiling-applications.html#package_into_jar
and package *server.java* and *client.java* into two jar files.

Convert a Jar to Complied Executable (Optional)

Run

```
cd test/src
./convert.sh client.java client
./convert.sh server.java server
```

Run the Program

Upload *server* and *client* to corresponding VMs.
Run

```
./server [server_name] [config_file]
./client [client_name] [config_file]
```

Design

Communication between Client and Server

We use Java Remote Method Invocation (RMI) system, which is an alternative to RPC calls.

Client

A client will get its randomly selected server's *ClientService* object through *Registry* lookup and use it to make RPC calls to that server.

Server

A server has a *ClientService* object and a *ServerSevice* object that both extend the *Remote* interface. The server will bind its IP address, port, and path (e.g., "/client" corresponds to *ClientService*) to the two objects. It also stores a map of *Account*.

- ClientService receives RPC calls from the client and sends back the results
- ServerService forwards clients' requests to the corresponding server

Each server uses a ConcurrentHashMap to store *Account*. An account has three fields related to concurrency control:

- lastCommitID, timestamp of the last commit
- RTS, a *TreeSet* of read timestamp

- TW, a *TreeMap* of <tentative write timestamp, the balance after current write>

The TreeSet and TreeMap can keep the timestamps ordered.

Account

Accounts are stored on a server (ConcurrentHashMap<accountName, Account>).
An account has the following fields:

- String name;
- int balance; // committed value
- String lastCommitID; // transaction id of the last committed transaction, default: -1
- TreeSet<read timestamp> RTS
- TreeMap<transactionID, balance after operation> TW

Transaction

Transactions are stored on the server (ConcurrentHashMap<clientName, Transaction>). It is an inner class of *ClientService* with the following fields:

- String transactionID
- ConcurrentHashMap<ServerService, HashSet<transactionID>> operations

When to abort, we find the corresponding ServerService and make an RPC call.

Concurrency Control

We use timestamp ordering to design our transaction system. When a transaction begins, a randomly selected server will generate a string "[localTimestamp] [serverId]" as the transactionID. We use *synchronized* blocks on the account when we try to do an operation on it. This is because timestamp ordering does not guarantee absolute concurrency (e.g., write and read upon RTS and TW have to be synchronized).
Operations:

- BALANCE: Follows the read rules of timestamp ordering on the lecture slides. It will check the account's lastCommitID and the size of TW. If lastCommitID is null and (TW is empty or TW's first timestamp is larger than the current timestamp), the transaction will ABORT because the account does not exist. We use a *synchronized* block on the account. If the BALANCE operation has to wait for some previous transaction to commit or abort, it will *wait()* on the account that has conflicts until notified.
- DEPOSIT/WITHDRAW: It will first do a BALANCE operation to get the balance it can update. Then it follows the write rules to put the updated value into TW.
- COMMIT: We use a similar approach to the two-phase committing. When a transaction starts to COMMIT, the server will first call *tryCommit()* to check if all write operations associated with the transaction can proceed and the modified balances are valid, then the server calls *commit()* to commit the transaction. We use a *synchronized* block on each of the accounts involved during each iteration. If the transaction cannot proceed, it will *wait()* until all previous transactions are committed or aborted. If there is any negative

balance, the transaction will be aborted. The server will call *notifyAll()* on the account that has changes committed.

- ABORT: It will remove all tentative write from every account the transaction tries to write. We use a *synchronized* block on all the accounts involved. The server will call *notifyAll()* on the account that has changes aborted.

Deadlock Prevention

We use timestamp ordering so there will not be an issue with deadlocks.

Note: We provide very detailed comments in our code, including some explanation of our design. You can also refer to that.