CONCORDIA UNIVERSITY

DEPARTMENT OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

SOEN 423, Fall 2017 Instructor: R. Jayakumar

ASSIGNMENT 1

Issued: Sep. 21, 2017 Due: Oct. 5, 2017

Note: *The assignments must be done individually and submitted electronically.*

Distributed Banking System (DBS) using Java RMI

In the assignments and project, you are going to design and implement a simple Distributed Banking System (DBS): a distributed system used by bank customers to perform financial operations on their accounts and bank managers to create and manage customer accounts for a bank with branches at multiple cities.

Consider four branches: British Columbia (BC), Manitoba (MB), New Brunswick (NB) and Quebec (QC) for your implementation. A customer account record contains the following fields:

- First name.
- Last name.
- Account Number.
- Address.
- Phone.

These records are placed in several lists that are stored in a hash map according to the first letter of the last name and indicated in the records. For example, all the records with the last name starting with an "A" will belong to the same list and will be stored in a hash map (acting as the database) and the key will be "A". Each server also maintains a log containing the history of all the operations that have been performed on that server. This should be an external text file (one per server) and shall provide as much information as possible about what operations are performed, at what time and who performed the operation.

The users of the system are *customers* and *managers* identified by a unique *customerID* or *managerID* respectively, which is constructed from the acronym of the branch and a 4-digit number (e.g. QCC1111 for a customer and QCM1111 for a manager). Whenever the user performs an operation, the system must identify the server that user belongs to by looking at the *ID* prefix (i.e. QCC or QCM) and perform the operation on that server. The user should also maintain a log (text file) of the actions they performed on the system and the response from the system when available. For example, if you have 10 users using your system, you should have a folder containing 10 logs.

Manager Operations:

The operations that can be performed by a manger are:

- *createAccountRecord (firstName, lastName, address, phone, branch)* :

  When a manager invokes this method through a program called *ManagerClient*, the server associated with the indicated *branch* attempts to create a customer record with the information passed, assigns a unique *customerID* and inserts the customer record at the appropriate location in the hash map maintained at the indicated *branch*. The server returns information to the manager whether the operation was successful or not and both the server and the client store this information in their logs.

- *editRecord (customerID, fieldName, newValue)*

  When invoked by a manager, the server associated with this customer, (determined by the unique *customerID*) searches in the hash map to find the customer record and changes the value of the field identified by *fieldname* to the *newValue*, if it is found. Upon success or failure it returns a message to the manager and the logs are updated with this information. If the new value of the fields such as branch, does not match the type it is expecting, it is invalid. For example, if the *fieldName* is branch and *newValue* is other than BC, MB, NB, QC, the server should return an error. The fields that should be allowed to change are address, phone and branch.

- *getAccountCount()*

  A manager invokes this method and the server associated with that manager's branch concurrently finds out the number of customer accounts in the all the branches using UDP/IP sockets and returns the result to the manager. For example if there are 400 customer accounts in BC, 300 in MB, 500 in NB, and 700 in QC, this function should return the following list of (string, integer) pairs: BC 400, MB 300, NB 500, QC 700

Customer Operations:

The operations that can be performed by a customer are:

- *deposit* (*customerID*, *amt*):

  A customer with *customerID* invokes this method through a *CustomerClient*. This method increases the balance of the customer's account by the specified amount *amt* and returns a message indicating the success/failure of the operation and the new account balance.

- *withdraw* (*customerID*, *amt*):

  A customer with *customerID* invokes this method through a *CustomerClient*. This method decreases the balance of the customer's account by specified amount *amt* if possible and returns a message indicating the success/failure of the operation the new account balance.

- *getBalance* (*customerID*):

  A customer with *customerID* invokes this method through a *CustomerClient*. This method returns the current balance of the customer's account.

Thus, this application has a number of *Servers* (one per branch) each implementing the above operations for that branch, customers and managers invoking the operations at the appropriate *Server* as necessary. When a *Server* is started, it registers its address and

related/necessary information with a central repository. For each operation, the *CustomerClient/ManagerClient* finds the required information about the associated *Server* from the central repository and invokes the corresponding operation. ***Your server should ensure that a customer can only perform a customer operation and cannot perform a manager operation; and a manager can perform all (both the customer and manager) operations.***

In this assignment, you are going to develop this application using Java RMI. Specifically, do the following:

- Write the Java RMI interface definition for the *Server* with the specified operations.
- Implement the *Server*.
- Design and implement a *ManagerClient,* which invokes the server system to test the correct operation of the DBS invoking multiple *Server* (each of the servers initially has a few records) and multiple managers.
- Design and implement a *CustomerClient,* which invokes the server system to test the correct operation of the DBS invoking multiple *Server* (each of the servers initially has a few records) and multiple managers.

You should design the *Server* maximizing concurrency. In other words, use proper synchronization that allows multiple passengers and managers to perform operations for the same or different records at the same time.

Marking Scheme

[30%] *Design Documentation*: Describe the techniques you use and your architecture, including the data structures. Design proper and sufficient test scenarios and explain what you want to test. Describe the most important/difficult part in this assignment. You can use UML and text description, but limit the document to 10 pages. Submit the documentation and code by the due date; print the documentation and bring it to your demo.

[70%] *Demo in the Lab*: You have to register for a 5-minute demo. Please come to the lab session and choose your preferred demo time in advance. You cannot demo without registering, so if you did not register before the demo week, you will lose 40% of the marks. Your demo should focus on the following.

[50%] C*orrectness of code:* Demo your designed test scenarios to illustrate the correctness of your design. If your test scenarios do not cover all possible issues, you'll lose part of mark up to 40%. You will also be evaluated on the implementation of your design.

[20%] *Questions:* You need to answer some simple questions (like those discussed during lab tutorials) during the demo. They can be theoretical related directly to your implementation of the assignment.

Questions

If you are having difficulties understanding any aspect of this assignment, feel free to email your teaching assistant Ms. Akriti Saini at akrit123@outlook.com. It is strongly recommended that you attend the tutorial sessions which will cover various aspects of the assignment.