COMP6231/1 – Sections BB – Summer 2018

ASSIGNMENT #1
Due: Sunday, May 27th by midnight 11:59 PM

Important Note

- The work is realized in a **team of 4 students**.
- Submit code by the due date.
- Your program should be compiled, executed and return the expected results; otherwise a mark 0 (zero) will be assigned.

Distributed Class Management System (DCMS) using Java RMI

In the assignments and project, you are going to implement a simple Distributed Class Management System: a distributed system used by center managers to manage information regarding the teachers and students across different centers.

Consider three centers locations: Montreal (MTL), Laval (LVL) and Dollard-des-Ormeaux (DDO) for your implementation. The server for each center (called *CenterServer*) must maintain a number of *Records*. There are two types of *Records*: *TeacherRecord* and *StudentRecord*. A *Record* can be identified by a unique *RecordID* starting with TR (for *TeacherRecord*) or SR (for *StudentRecord*) and ending with a 5 digits number (e.g. TR10000 for a *TeacherRecord* or SR10001 for a *StudentRecord*).

A *TeacherRecord* contains the following fields:

- First name
- Last name
- Address
- Phone
- Specialization (e.g. french, maths, etc)
- Location (mtl, lvl, ddo)

A *StudentRecord* contains the following fields:

- First name
- Last name
- CoursesRegistered (maths/french/science, note that student could be registered for
- multiple courses)
- Status (active/inactive)
- Status Date (date when student became active (if status is active) or date when student became inactive (if status is inactive))

The *Records* are placed in several lists that are stored in a hash map according to the first letter of the last name 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". We do not distinguish between teacher records and Student records when inserting them into the hash map (i.e. a list may contain both teacher and Student records). 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 center managers. They can be identified by a unique *managerID*, which is constructed from the acronym of the center and a 4-digit number (e.g. MTL1111). Whenever a manager performs an operation, the system must identify the center that manager belongs to by looking at the *managerID* prefix and perform the operation on that server. The managers carry with them 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 managers using your system, you should have a folder containing 10 logs.

The operations that can be performed are the following:

- ***createTRecord* (*firstName*, *lastName, address, phone, specialization, location*) :**

When a manager invokes this method from his/her center through a client program called *ManagerClient*, the server associated with this manager (determined by the unique *managerID* prefix) attempts to create a *TeacherRecord* with the information passed, assigns a unique *RecordID* and inserts the *Record* at the appropriate location in the hash map. 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.

- ***createSRecord* (*firstName*, *lastName*, *courseRegistered*, *status, statusDate*)**

When a manager invokes this method from a *ManagerClient*, the server associated with this manager (determined by the unique *managerID* prefix) attempts to create a *StudentRecord* with the information passed, assigns a unique *RecordID* and inserts the *Record* at the appropriate location in the hash map. 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.

- ***getRecordCounts* ()**

A manager invokes this method from his/her *ManagerClient* and the server associated with that manager concurrently finds out the number of records (both TR and SR) in the other centers using UDP/IP sockets and returns the result to the manager. Please note that it only returns the record counts (a number) and not the records themselves. For example if MTL has 6 records, LVL has 7 and DDO had 8, it should return the following: MTL 6, LVL 7, DDO 8.

- ***editRecord* (*recordID*, *fieldName*, *newValue*)**

When invoked by a manager, the server associated with this manager, (determined by the unique *managerID*) searches in the hash map to find the *recordID* and change 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 location (Teacher), status (Student), does not match the type it is expecting, it is invalid. For example, if the found *Record* is a *TeacherRecord* and the field to change is *location* and *newValue* is other than mtl, lvl or ddo, the server shall return an error. The fields that should be allowed to change are *address, phone and location* (for TeacherRecord), and *course registered, status and status date* (for StudentRecord).

Thus, this application has a number of *CenterServers* (one per center) each implementing the above operations for that center and *ManagerClients* (one per center) invoking the manager's operations at the associated *CenterServer* as necessary. When a *CenterServer* is started, it registers its address and related/necessary information with a central repository. For each operation, the *ManagerClient* finds the required information about the associated *CenterServer* from the central repository and invokes the corresponding operation.

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 *CenterServer* with the specified operations.
- Implement the *CenterServer*.
- Design and implement a *ManagerClient,* which invokes the center's server system to test the correct operation of the DCMS invoking multiple *CenterServer* (each of the servers initially has a few records) and multiple managers.

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

MARKING SCHEME

[**40%**] ***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.

[**60%**] ***The correctness of code:*** your designed test scenarios to illustrate the correctness of your design. If your test scenarios do not cover all possible issues, you will lose part of marks up to 50%.

EDUCATIONAL GUIDELINES

- The work is realized in a **team of 4 students**.
- The delivery must be made no later than **Sunday, May 27th, 2018 by midnight 11:59 PM** using the Website submission as mentioned in the course outline.
- If you are having difficulties understanding sections of this assignment, feel free to email the Teaching Assistants It is strongly recommended that you attend the Lab sessions which will cover various aspects of the assignment.