CONCORDIA UNIVERSITY

DEPARTMENT OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

COMP 6231, Winter 2022 Instructor: R. Jayakumar

ASSIGNMENT 1

Issued: Jan. 26, 2022 Due: Feb. 9, 2022

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

Distributed Appointment Management System (DAMS) using Java RMI

In this assignment, you are going to implement a distributed appointment management system (DAMS) for health care: a distributed system used by hospital admins to manage the information about medical appointments and patients to book or cancel a medical appointment across different hospitals within the medicare system.

Consider three hospitals in different cities: Montreal (MTL), Quebec (QUE) and Sherbrooke (SHE) for your implementation. The users of the system are hospital's appointment admins and appointment booking patients. *Admins* and *Patients* are identified by a unique *adminID* and *patientID* respectively, which is constructed from the acronym of their hospital's city and a 4-digit number (e.g. MTLA2345 for an admin and MTLP2345 for a patient). Whenever the user performs an operation, the system must identify the server to which the user belongs by looking at the ID prefix (i.e. MTLA or MTLP identifies the server for Montreal) and performs 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.

In this DAMS, there are different admins for 3 different servers. They create appointment slots corresponding to the appointment type as per the availability of doctors. There are three appointment types for which slots can be created: *Physician*, *Surgeon* and *Dental*. A patient can book an appointment in any of the hospital, for any appointment type, if the appointment is available for booking (if the appointment slots corresponding to an appointment on a particular date is not yet full). A server (which receives the request) maintains a booking-count for every patient. A patient cannot book more than one appointment with the same appointment id and same appointment type**.** Also, a patient is not allowed to have more than one booking of same appointment type in a day. There are three time slots available for each appointment type on a day: *Morning* (M), *Afternoon* (A) and *Evening* (E). An *appointmentID* is a combination of city, time slot and appointment date (e.g. MTLM100222 for a morning appointment on 10th February 2022 in Montreal, QUEA151022 for an afternoon appointment on 15th October 2022 in Quebec and SHEE201122 for an evening appointment on 20th November 2022 in Sherbrooke). You should ensure that if the availability of an appointment is full, more patients cannot book the appointment. Also, a patient can book as many appointments in his/her own city, but only at most 3 appointments from other cities overall in a week.

The appointment records are maintained in a hashmap as shown in Figure 1. Here appointment type is the key, while the value is again a sub-hashmap. The key for sub-hashmap is the *appointmentID*, while the value of the sub-hashmap is the information about the appointment.

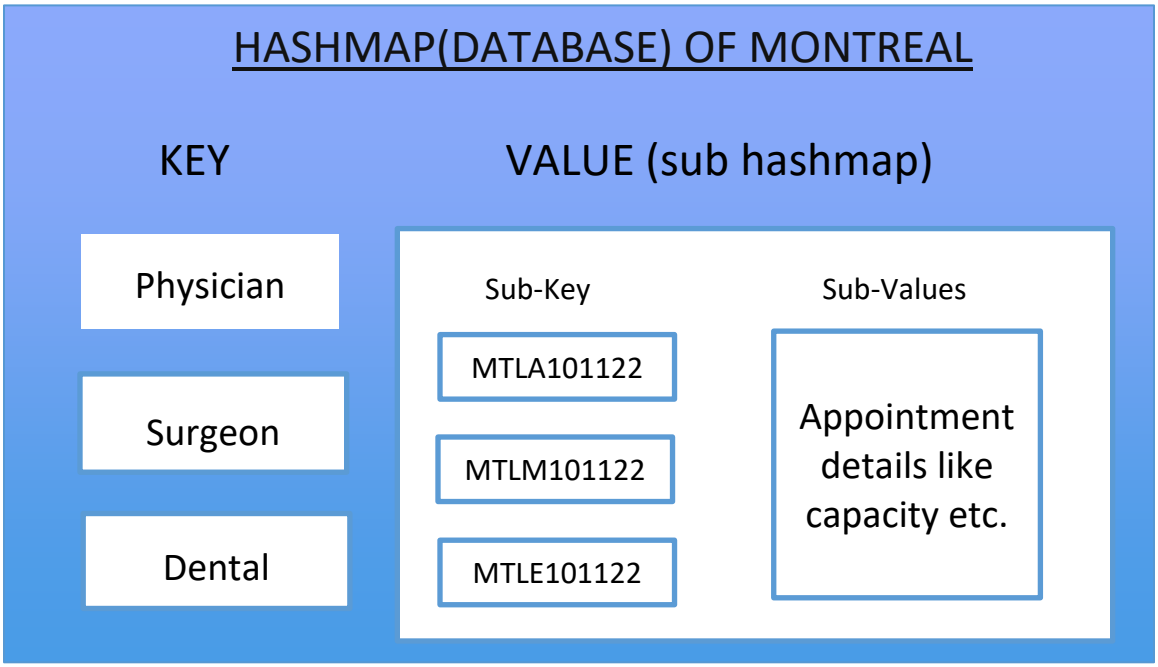

Fig. 1 Hashmap of a single hospital/city

Each server also maintains a log file 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. These are some details that a single log file record must contain:

- Date and time the request was sent.
- Request type (book an appointment, cancel an appointment, etc.).
- Request parameters (patientID, appointmentID, etc.).
- Request successfully completed/failed.
- Server response for the particular request.

Admin Role:

The operations that can be performed by an admin are the following:

- *addAppointment* (*appointmentID, appointmentType, capacity*):

  When an admin invokes this method through the server associated with this admin (determined by the unique *adminID* prefix), attempts to add an appointment with the information passed, and inserts the record at the appropriate location in the hashmap. The server returns information to the admin whether the operation was successful or not and both the server and the client store this information in their logs. If an appointment

already exists for same appointment type, the admin can't add it again for the same appointment type. If an appointment does not exist in the database for that appointment type, then add it. Log the information into the admin log file.

- *removeAppointment* (*appointmentID, appointmentType*):

  When invoked by an admin, the server associated with that admin (determined by the unique *adminID*) searches in the hashmap to find and delete the appointment for the indicated *appointmentType*. Upon success or failure, it returns a message to the admin and the logs are updated with this information. If an appointment does not exist, then obviously there is no deletion performed. Just in case that, if an appointment exists and a patient has booked that appointment, then, delete the appointment and book the next available appointment for that patient. Log the information into the log file.

- *listAppointmentAvailability* (a*ppointmentType*):

  When an admin invokes this method from his/her hospital/city through the associated server, that hospital/city server concurrently finds out the number of spaces available for each appointment in all the servers, for only the given *appointmentType*. This requires inter server communication that will be done using UDP/IP sockets and result will be returned to the client. Eg: Surgeon - MTLE131122 3, QUEA061222 6, SHEM181122 0, MTLE191222 2.

Patient Role:

The operations that can be performed by a patient are the following:

- *bookAppointment* (*patientID, appointmentID, appointmentType*):

  When a patient invokes this method from his/her city through the server associated with this patient (determined by the unique *patientID* prefix) attempts to book the appointment for the patient and change the capacity left in that appointment. Also, if the booking was successful or not, an appropriate message is displayed to the patient and both the server and the client stores this information in their logs.

- *getAppointmentSchedule* (*patientID*):

  When a patient invokes this method from his/her city's hospital through the server associated with this patient, that city's branch server gets all the appointments booked by the patient and display them on the console. Here, appointments from all the cities, Montreal, Quebec and Sherbrooke, should be displayed.

- *cancelAppointment (patientID, appointmentID):*

  When a patient invokes this method from his/her city's hospital through the server associated with this patient (determined by the unique *patientID* prefix) searches the hash map to find the appointment*ID* and remove the appointment. Upon success or failure, it returns a message to the patient and the logs are updated with this information. It is required to check that the appointment can only be removed if it was booked by the same customer who sends cancel request.

Thus, this application has a number of servers (one per city) each implementing the above operations for that hospital, *PatientClient* invoking the patient's operations at the associated server as necessary and *AdminClient* invoking the admin's operations at the associated server. When a server is started, it registers its address and related/necessary information

with a central repository. For each operation, the PatientClient/AdminClient finds the required information about the associated server from the central repository and invokes the corresponding operation. ***Your server should ensure that a patient can only perform a patient operation and cannot perform any admin operations, but an admin can perform all 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 *PatientClient,* which invokes the server system to test the correct operation of the DAMS invoking multiple servers (each of the servers initially have few records) and multiple patients.
- Design and implement an *AdminClient,* which invokes the server system to test the correct operation of the DAMS invoking multiple servers (each of the servers initially have few records) and multiple admins.

You should design the server maximizing concurrency. In other words, use proper synchronization that allows multiple users 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 electronically by the due date; print the documentation and bring it to your DEMO.

[70%] *DEMO*: You have to register for a 5–10 minutes demo. You cannot demo without registering, so if you did not register before the demo week, you will lose 40% of the marks. The demo should focus on the following:

- [50%] *The correctness 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 will lose part of marks up to 40%.
- [20%] *Questions:* You need to answer some simple questions (like what we have 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 contact your teaching assistants (Lab FI: Rajkumar Rakoli rokalirajkumar@gmail.com, Lab FJ: Brijesh Lakkad brijeshlakkad22@gmail.com, Lab FK: Stallone Macwan stallonemacwan@gmail.com). It is strongly recommended that you attend the lab sessions, as various aspects of the assignment will be covered there.