CONCORDIA UNIVERSITY

DEPARTMENT OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

COMP 6231, Winter 2023 Instructor: R. Jayakumar

ASSIGNMENT 1

Issued: Jan. 23, 2023 Due: Feb. 6, 2023

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

Distributed Movie Ticket Booking System Using Java RMI

In this assignment, you are going to implement a Distributed Movie Ticket Booking System (DMTBS) for a chain of movie theatres: a distributed system used by theatre managers who manage the information about the ongoing shows and upcoming shows running in theatres as well as customers who can book and cancel the movie ticket across the different theatres.

Consider three theatres in three different areas: Atwater (ATW), Verdun (VER) and Outremont (OUT) for your implementation. The users of the system are admins and customers. Admins and Customer are identified by a unique *adminID* and *customerID* respectively, which is constructed from the acronym of their area and a 4-digit number (e.g. ATWA2345 for an admin and ATWC2345 for a customer). Whenever the user (admin or customer) performs an operation, the system must identify the server to which the user belongs by looking at the ID prefix (i.e., ATWA or ATWC) 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.

In DMTBS, there are different admins for 3 different servers. They create slots for movies with movie type and booking capacity. There are three movies for which booking can be created which are *Avatar*, *Avengers* and *Titanic*. A customer can book a movie ticket in any theatre, for any movie, if the movie is available for booking (if the movie shows is not yet full on a particular date). A server (which receives the request) maintains a booking-count for every customer. There are three time slots available for each movie on a day: Morning (M), Afternoon (A) and Evening (E). A *movieID* is a combination of area, time slot and movie date e.g. ATWM160523 for a morning movie on 16 May 2023 in Atwater, VERA190323 for an afternoon movie show on 19th March 2023 in Verdun and OUTE270123 for an evening movie on 27th Jan 2023 in Outremont). Customer can book multiple tickets for same show (i.e. customer ATWC3432 can book multiple tickets of the movie named avengers for the morning show) if the capacity is not full. You should ensure that if the capacity of the movie is full, more customers cannot book the movies. Also, a customer can book as many movie tickets in his/her own area, but only at most 3 movies from other areas overall in a week. A customer can book extra ticket for the movie if required but cannot book the tickets of the same movie for the same show in different theatres. Moreover, an admin can create the movie shows for next one week from the current date. The movie ticket booking records are maintained in a HashMap as shown in Figure 1. Here movie name is the key, while the value is a sub-

HashMap. The key for sub-HashMap is the *movieID*, while the value of the sub-HashMap is the information details about the movie.

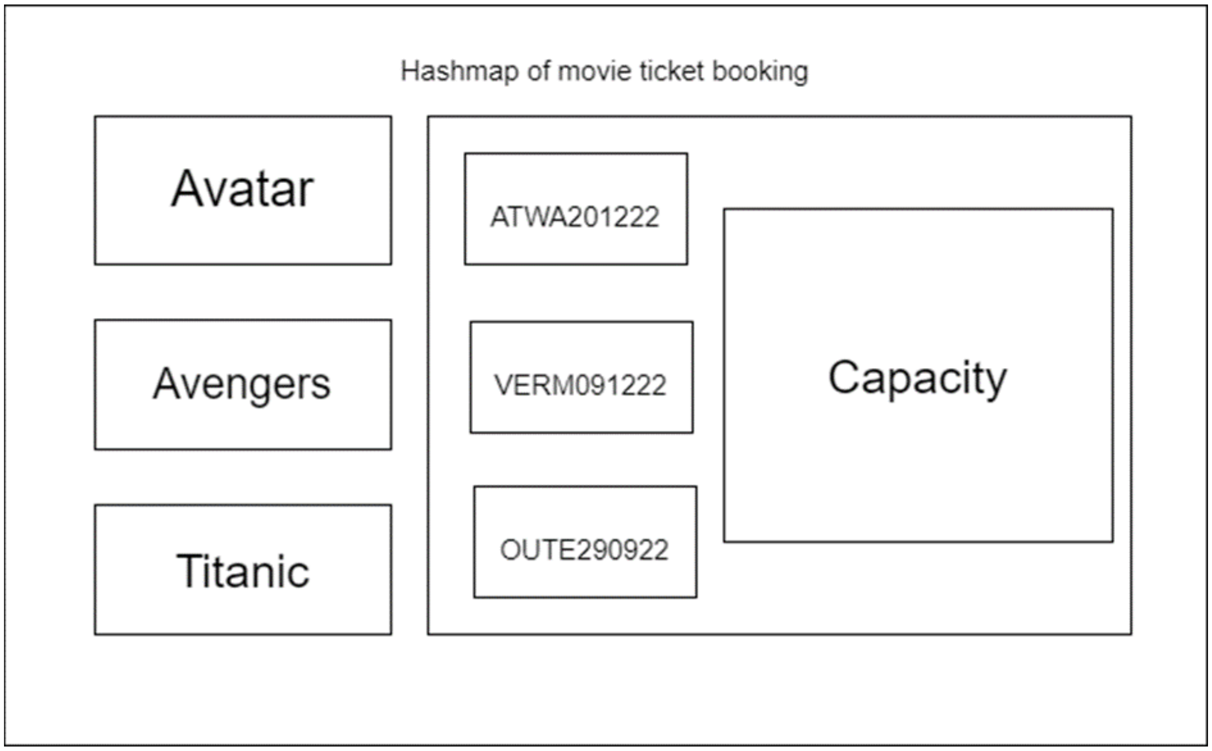

Fig. 1 HashMap of a single area's Theatre

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 a movie ticket, cancel a movie ticket, etc.).
- Request parameters (*customerID*, *movieID*, 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:

- *addMovieSlots* (*movieID*, *movieName*, *bookingCapacity*):

  When an admin invokes this method through the server associated with this admin (determined by the unique *adminID* prefix), attempts to add a movie with the information passed, and inserts the record at the appropriate location in the hash map. 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 a movie record with *movieID* and *movieName* already exists, *bookingCapacity* will be updated for that movie record. If a movie does not exist in the database for that movie name, then a new movie slots is added. Log the information into the admin log file.

- *removeMovieSlots* (*movieID*, *movieName*):

  When invoked by an admin, the server associated with that admin (determined by the unique *adminID*) searches in the hash map to find and delete the movie for the indicated *movieName* and *movieID*. Upon success or failure, a message is returned to the admin and the logs are updated with this information. If a movie slot does not exist, then obviously there is no deletion performed. If a movie show exists and a client has booked that movie ticket, then delete the movie and book the next available movie show (with same *movieName*) for that client. Apart from this admin cannot delete the movie show that occurred from the before the current date. Log the information into the log file.

- *listMovieShowsAvailability* (*movieName*):

  When an admin invokes this method from his/her area's theatre through the associated server, that area's theatre server concurrently finds out the number of tickets available for each movie shows in all the servers, for only the given *movieName*. This requires inter server communication that will be done using UDP/IP messages and the result will be returned to the client. Eg: Avatar: ATWE150623 3, OUTM161223 6, VERE181123 1.

Customer Role:

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

- *bookMovieTickets* (*customerID*, *movieID*, *movieName, numberOfTickets*):

  When a customer invokes this method from his/her area through the server associated with this customer (determined by the unique *customerID* prefix) attempts to book the *numberOfTickets* tickets for a particular movie show for the customer and change the capacity left for that movie show. Also, an appropriate message is displayed to the customer whether the booking was successful or not and both the server and the client stores this information in their logs.

- *getBookingSchedule* (*customerID*):

  When a customer invokes this method from his/her area through the server associated with this customer, that area's server gets all the movie tickets for a particular show booked by the customer and display them on the console. Here, bookings from all the area's theatre such as Atwater, Outremont and Verdun should be displayed.

- *cancelMovieTickets* (*customerID*, *movieID*, *movieName, numberOfTickets*):

  When a customer invokes this method from his/her area's theatre through the server associated with this customer (determined by the unique *customerID* prefix) searches the hash map to find the *movieID* with the *movieName* and cancel the *numberOfTickets* tickets for that movie show. Upon success or failure, it returns a message to the customer and the logs are updated with this information. It is required to check that the movie ticket can only be canceled if it was booked by the same customer who sends cancel request.

Thus, this application has a number of servers (one per theatre) each implementing the above operations for that area, *customerClient* invoking the customer'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. A *customerClient/adminClient* 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 customer operations and cannot perform any admin operation where as an admin can also perform all customer operations on behalf of any customer along with his/her own operations.

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: Anurag Shekhar anurag.shekhar@mail.concordia.ca, Lab FJ: Kiana Nezami nezami.kiana@gmail.com, Lab FK: Yash Patel ycpatel1999@gmail.com). It is strongly recommended that you attend the lab sessions, as various aspects of the assignment will be covered there.