Comp 6231 – Assignment 1
RMI

Saman Saadi Alekasir
Student ID: 40009949

October 8, 2017

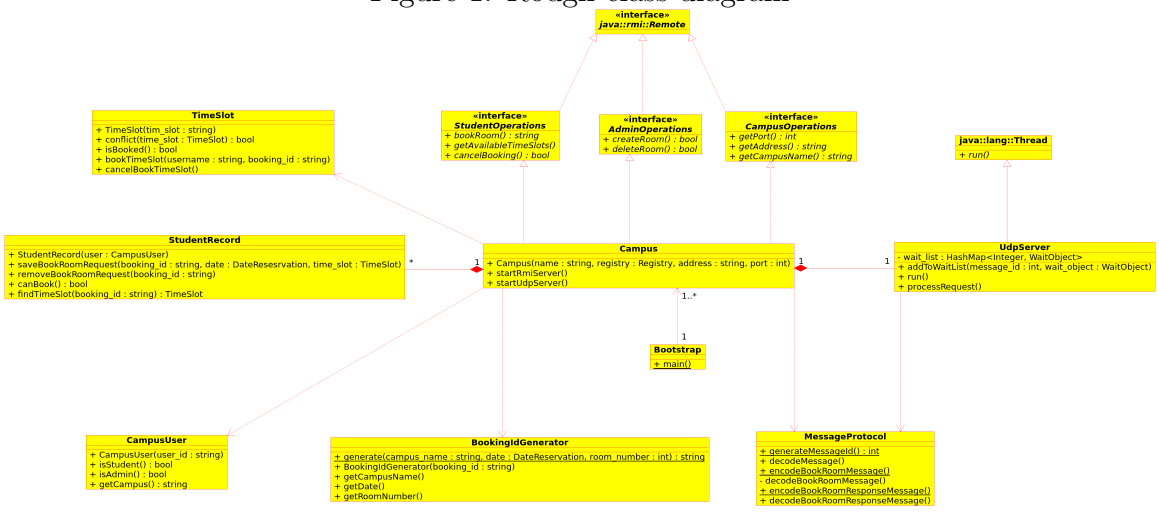
1 Campus Servers Overview

Each campus server is represented by one instance of class *Campus*. Since the problem statement indicated there are only 3 servers, I instantiate this class 3 times. But my implementation easily works if I instantiate *Campus* more. In other words it's easy to have more than 3 campus servers.

Class *Campus* implements three interfaces: *AdminOperations*, *StudentOperations* and *CampusOperations*. *AdminOperations* defines the interface for tasks which are available for an admin user. Since *AdminClient* only knows *AdminOperations*, it cannot see other methods which are available to other users. In the campus side I also double check that only an admin user do admin operations. You can see signatures like *createRoom* and *deleteRoom* in this interface. Student-related operations are defined in *StudentOperations* interface. *StudentClient* only knows about *StudentOperations*. So it cannot access operations for other type of users. I also double check in every method that the user is student. There is another interface named *CampusOperations* which has methods like *getPort*, *getAddress* and *getCampusName*. For example when campus "DVL" try to send a request to campus "KKL", it uses this interface to get the address and port of campus "KKL" through RMI. After that it uses UDP to send and receive requests. You can see a rough class diagram in figure 1. As you can see the startup class is *Bootstrap*. We can run three campuses in one process or 3 separate processes. In the first case we run *Bootstrap* once and it instantiate 3 *Campus* objects. In the second scenario we run Bootstrap three times (each has their own process) and instantiate one *Campus* object. In Campus constructor we run another thread and it initialize a UDP server. Then it listen on a specified port for handling requests. Upon receiving a request it create another thread to handle the received request. By doing that we can immediately return to listen on that port.

Class *MessageProtocol* is responsible for my simple text-based protocol. Whenever the campus decided to communicate with another campus, it uses this class to create a message for sending over UDP socket. On the other hand the receiver campus uses this class to decode the string message into POJOs. Every message has a header and a body. The header consists of two fields: message type and message id. Message id is unique in each campus and by using that the campus can determine which student is corresponding to the received message. I also use message id for detecting duplication and discard them. Since each request has a different message

Figure 1: Rough class diagram



format, I use message type to inform the receiver how to parse the message.

As you can see there are some utility classes. All student activities are stored in a *StudentRecord* object. *BookingIdGenerator* is responsible for generating a booking id in such a way that we can extract campus name, reservation date and room number from it. It is obvious this booking id should be unique among all campuses. *CampusUser* class can extract the type of user (student or admin) and the campus which user belongs to, from user ID. I also developed a class named *TimeSlot* which stores the time slot and booking information.

2 Synchronization

According to Java SE 8 documentation [1]: "if multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.). This is typically accomplished by synchronizing on some object that naturally encapsulates the map". So it is not safe to use *HashMap.get* when another thread try to change *HashMap*'s structure [2]. I should also mention *HashMap* is not thread-safe for different keys [3].

2.1 Main Database

I designed three types of object locks for handling *HashMap* database for multi-threaded environment. One locks the first key (date of reservation), the other locks the second key (room number) and the third locks a list of time slots for a specific date and room number.

I've noticed *createRoom* is similar to *deleteRoom* but some parts are different. For handling that I design a system which is used in event-handling systems. I define a private method named *traverseDb* which handles all synchronization when we traverse the database and call some abstract methods for handling different implementation between *createRoom* and *deleteRoom* operations. In other words, those methods should implement abstract class methods define in listing 1. I removed the arguments and the return value for simplicity. As you can see all of them are thread-safe (unless we try to access another database or container). In other words, *createRoom* and *deleteRoom* should only implement these thread-safe callbacks. Handling the synchronization is done in *traverseDb* method.

Listing 1: DbOperations abstract class

```
1 private abstract class DbOperations
2 {
3   //All the following methods
4   //are thread-safe for campus time slots database
5   public abstract onNullValue();
6   public abstract onNullSubValue();
7   public abstract void onSubValue();
8 }
```

You can see a snippet of *traverseDb* method implementation in listing 2. It is important to put lines 2 - 8 in a *synchronization* block. In *createRoom* method it implements *onNullValue* by creating a new *HashMap* for room number and list of time slots. If more than one thread try to execute it, each of them has their own new *HashMap* of room number and list of time slots which is not correct because one of those two threads has stale value.

Listing 2: Snippet from traverseDb method

```
1 synchronized (date_db_lock) {
2   val = db.get(date);
```

```
3     if (val == null)
4     {
5       val = db_ops.onNullValue();
6       if (val == null)
7         return;
8       db.put(date, val);
9     }
10  }
```

According to problem statement, we need to update student record when we delete a time slot which that user has booked. I create a thread for every time slot which is booked by a user from another campus which is responsible to get notification about the status of student record's update from another campus. At the end of method I'm going to wait for all created threads to finish their job.

2.2 Student Database

This database store the records of a particular student reservation and make sure it does not exceed 3 times per week. I follow the same pattern as main database. I've noticed *bookRoom* and *cancelBooking* have a lot in common. So I created a private method named *traverseStudentDb* which handles traversing student database and calls some callbacks which are implemented differently in *bookRoom* and *cancelBooking* methods. You can see the abstract class which both methods should implement in listing 3. All these methods are thread-safe, unless we want to access another database or container.

Listing 3: UserDbOperations Abstract Class

```
1  private abstract class UserDbOperations
2  {
3    //All the following methods are tread-safe for user database
4    public abstract boolean onUserBelongHere(record);
5    public abstract StudentRecord onNullUserRecord(user);
6    public abstract boolean onOperationOnThisCampus(time_slots);
7    public abstract void onOperationOnOtherCampus(message_id);
8    public abstract void onPostUserBelongHere(record);
9    public abstract ArrayList<TimeSlot> findTimeSlots();
10 }
```

I also try to lock student's record during the transaction. For example when user wants to book a room on another campus, the student record should be locked until we receive answer through UDP communication. When the campus send booking request to another campus, it create a $WaitObject$ object and then calls its *wait* method. It's the responsibility of UDP server to notify it that its response has arrived.

2.2.1 Getting Information About Available Time Slots

This method is different from *bookRoom* and *cancelBooking* methods. So it doesn't use *traverseStudentDb* method. You can see a simple pseudocode in algorithm 1. As you can see we need two functions F and G if we want to implement it correctly. Function F tries to obtain the available time slots in this campus and function G tries in other campus through UDP.

Algorithm 1 Pseudocode For Available Time Slots

```
1: function GET-AVAILABLE-TIMESLOS(date)
2:     S = F(date)
3:     for all c ∈ {Campuses} − {this_campus} do
4:         S = S ∪ G(c, date)
5:     end for
6: end function
```

Since we don't know which campus response arrives first, I create some threads equalt to the number of campuses minus 1 (in this assignment 2) which are responsible for extracting information from its corresponding campus. At the end of function I wait for all created threads to finish their job and combine the results and return it to the user.

2.3 Container Synchronization

According to *ArrayList* documentation [4]: "If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list.". So I need to synchronize accessing *ArrayList* (or other collections) when I want to access an element or traverse it. According to *Collections* documentation [5]: "It

is imperative that the user manually synchronize on the returned list when iterating over it (see listing 4)".

Listing 4: Collections Synchronization

```
1 List list = Collections.synchronizedList(new ArrayList());
2       ...
3 synchronized (list) {
4   Iterator i = list.iterator(); // Must be in synchronized block
5   while (i.hasNext())
6     foo(i.next());
7 }
```

2.4 Socket Synchronization

It is safe to have two threads which one of them read from socket and the other write into it concurrently [6]. I only have one reader thread so I don't need to do synchronization here. But I can have multiple thread for writing into socket so synchronization is necessary.

3 Testing Scenario

I've utilized JUnit for testing simple scenarios. I have a unit test for all student and admin operations. I tried to test most common situations. For example a user try to book an invalid time slot or a before-booked time slot.

For testing my program I created multiple time slots in different campuses so I tested the functionality of admin operations completely. I also tried to do admin operations with a student account. All the operations failed as expected. Then I tried to test student operations. I tried to book a room in user's campus and then I tried to book in other campuses and did the same for delete operation. Then I tried to delete a room which the current user booked in other campus for testing this difficult situation. I did a lot of thread synchronization for handling this situation very well.

4 Conclusion

As you can see in my document, a lot of sections are related to thread synchronization which I spent a lot of time to design. It is the most important and difficult part of this assignment. It's not easy to write some unit tests for it. So I read the critical sections multiple times to see that I handle all concurrent situations. I merged critical section for *createRoom* and *deleteRoom* into a single function which calls some callbacks which are implemented differently in those methods to avoid duplication. I revised critical sections multiple times. Since both methods share the same critical section it was easy to apply those changes to those methods. I follow the same pattern for student database.

Other parts were trivial. Because they are handled by Operating System or JDK. Another important part was designing a simple text-based protocol for communicating over network. I design a class for it. The protocol is designed in such a way that it can detect duplication and discard them.

References

[1] Oracle. HashMap (Java Platform SE 8). `https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html`. [Online; accessed 07-October-2017].

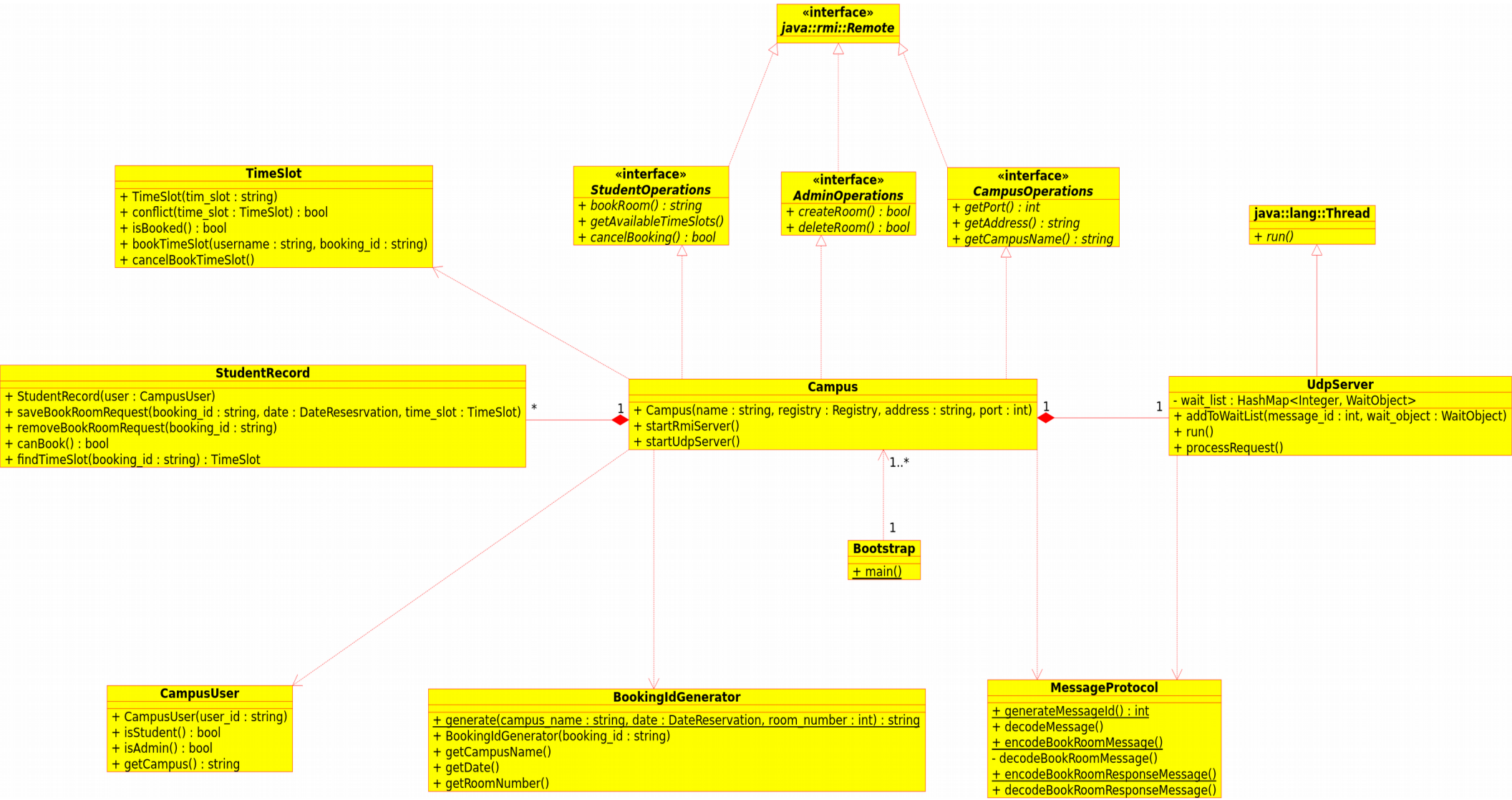
[2] StackOverflow. HashMap in multithreaded environment. `https://stackoverflow.com/questions/11050539/using-hashmap-in-multithreaded-environment`. [Online; accessed 07-October-2017].

[3] StackOverflow. Is a HashMap thread-safe for different keys? `https://stackoverflow.com/questions/2688629/is-a-hashmap-thread-safe-for-different-keys`. [Online; accessed 07-October-2017].

[4] Oracle. ArrayList (Java Platform SE 8). `https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`. [Online; accessed 07-October-2017].

[5] Oracle. Collections (Java Platform SE 8). `https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html`. [Online; accessed 07-October-2017].

[6] StackOverflow. Do Java sockets support full duplex? https://stackoverflow.com/questions/6265731/do-java-sockets-support-full-duplex. [Online; accessed 08-October-2017].


«interface»
java::rmi::Remote
«interface»
StudentOperations
+ bookRoom() : string
+ getAvailableTimeSlots()
+ cancelBooking() : bool
«interface»
AdminOperations
+ createRoom() : bool
+ deleteRoom() : bool
«interface»
CampusOperations
+ getPort() : int
+ getAddress() : string
+ getCampusName() : string
java::lang::Thread
+ run()
TimeSlot
+ TimeSlot(tim_slot : string)
+ conflict(time_slot : TimeSlot) : bool
+ isBooked() : bool
+ bookTimeSlot(username : string, booking_id : string)
+ cancelBookTimeSlot()
StudentRecord
+ StudentRecord(user : CampusUser)
+ saveBookRoomRequest(booking_id : string, date : DateResesrvation, time_slot : TimeSlot)
+ removeBookRoomRequest(booking_id : string)
+ canBook() : bool
+ findTimeSlot(booking_id : string) : TimeSlot
*
1
Campus
+ Campus(name : string, registry : Registry, address : string, port : int)
+ startRmiServer()
+ startUdpServer()
1
1
UdpServer
- wait_list : HashMap<Integer, WaitObject>
+ addToWaitList(message_id : int, wait_object : WaitObject)
+ run()
+ processRequest()
1..*
1
Bootstrap
+ main()
CampusUser
+ CampusUser(user_id : string)
+ isStudent() : bool
+ isAdmin() : bool
+ getCampus() : string
BookingIdGenerator
+ generate(campus_name : string, date : DateReservation, room_number : int) : string
+ BookingIdGenerator(booking_id : string)
+ getCampusName()
+ getDate()
+ getRoomNumber()
MessageProtocol
+ generateMessageId() : int
+ decodeMessage()
+ encodeBookRoomMessage()
- decodeBookRoomMessage()
+ encodeBookRoomResponseMessage()
+ decodeBookRoomResponseMessage()