NANYANG TECHNOLOGICAL UNIVERSITY

CZ4013 Distributed Systems Project

Academic Year 20/21 Semester 2

Name	Matriculation number	Job scope
Tiong Jun Hua, Ryan	U1721346L	Server
Beh Chee Kwang Nicholas	U1821425F	Client
Adrian Goh Jun Wei	U1721134D	Database & Cache

Content Page

Background of Requirements **3**

Client-Server Architecture Design **5**

Network Diagram 5

Request-Reply Message Structure 5

Client Implementation **8**

Server Implementation **9**

Cache / Database Implementation **10**

Assumptions 10

Database Design 10

Cache Implementation 11

Experiments on Invocation Semantics **12**

At-most-once 12

At-least-once 12

Background of Requirements

The goal of this project is to design and implement a distributed facility booking system based on client-server architecture. These requirements include:

- The server stores the information of all facilities (e.g., meeting rooms, lecture theatres), including:
    - The name of each facility (a variable-length string)
    - The availability of the facility over seven days of a week
    - The time represented in the form of day/hour/minute
    - The day as an enumerated type with possible values from Monday to Sunday
    - Hours and minutes as integers
    - Bookings made by users.
- Each facility requires exclusive use.
- The server program implements a set of services on the facilities for remote access by clients.
- The client program provides an interface for users to invoke these services.
- On receiving a request input from the user, the client sends the request to the server.
- The server performs the requested service and returns the result to the client.
- The client then presents the result on the console to the user.
- The client-server communication is carried out using UDP.

Functionalities of the program include:

1. A service that allows a user to query the availability of a facility on a selection of one or multiple days by specifying the facility name and the days. If there does not exist any facility with the specified name, an error message should be returned.
2. A service that allows a user to book a facility for a period of time by specifying the facility name and the start and end times of the booking. On successful booking, a unique confirmation ID is returned to the client and the availability of the facility should be updated at the server. In case of incorrect user input (e.g., non-existing facility name or the facility is completely or partially unavailable during the requested period due to existing bookings), a proper error message should be returned.
3. A service that allows a user to change its booking by specifying the confirmation ID of the booking and an offset for changing (e.g., to advance the booking by 1 hour or to postpone the booking by 30 minutes). The change does not modify the

length of the time period booked. On successful change, an acknowledgement is returned to the client and the availability of the facility should be updated at the server. In case of incorrect user input (e.g., non-existing confirmation ID or the facility is completely or partially unavailable during the new requested period due to existing bookings), a proper error message should be returned.

4. A service that allows a user to monitor the availability of a facility over the week through callback from the server for a designated time period called monitor interval. To register, the client provides the facility name and the length of monitor interval to the server. After registration, the Internet address and the port number of the client are recorded by the server. During the monitoring interval, every time a booking or an update is made by any client to the facility, the updated availability of the facility over the week is sent by the server to the registered client(s) through callback. After the expiration of the monitor interval, the client record is removed from the server which will no longer deliver the availability updates to the client.

Client-Server Architecture Design

Network Diagram

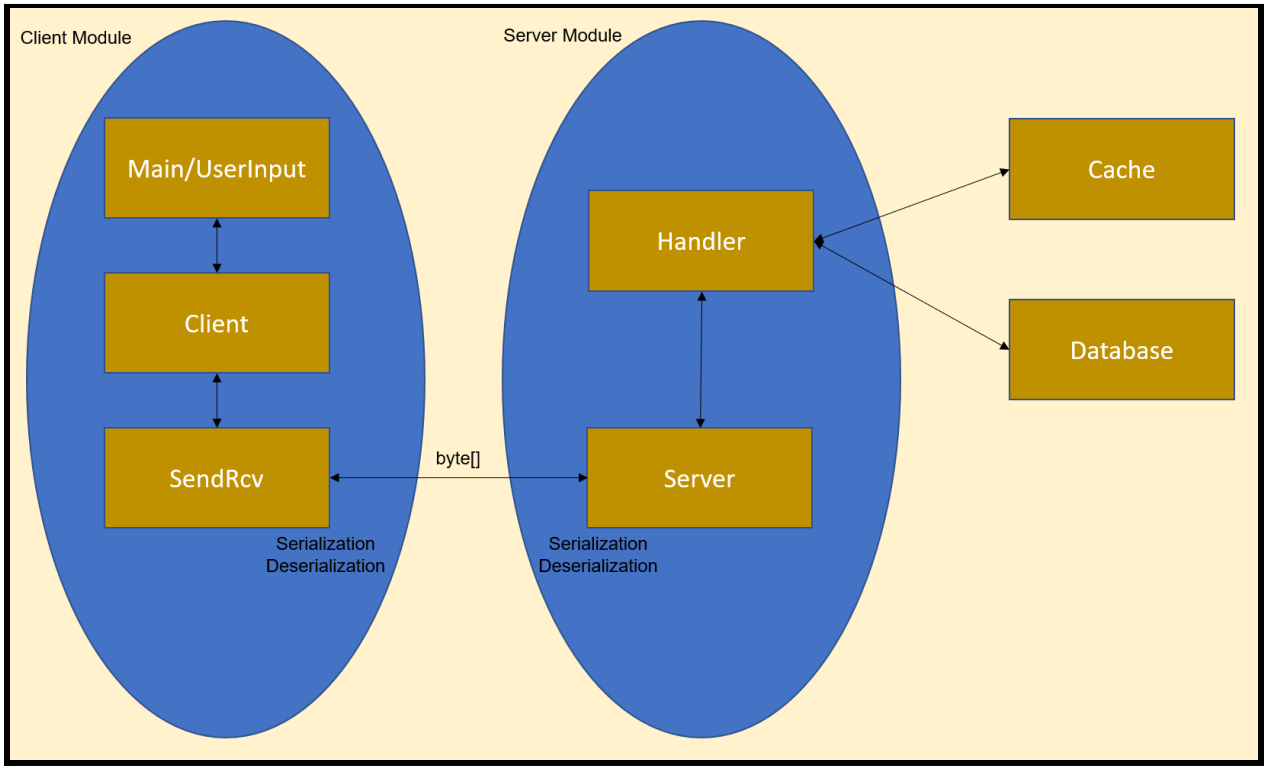

Figure 1: Network Diagram of Implementation

Request-Reply Message Structure

messageType	int (0=Request, 1=Reply)
requestId	int
application-specific request contents or reply contents	

Figure 2: Datagram Packet Message Structure

A Datagram Packet message structure follows the above diagram (as mentioned in the lecture slide page 69). In addition, the request/reply contents require a structure such that the client and server are synchronised on the semantics of the data within.

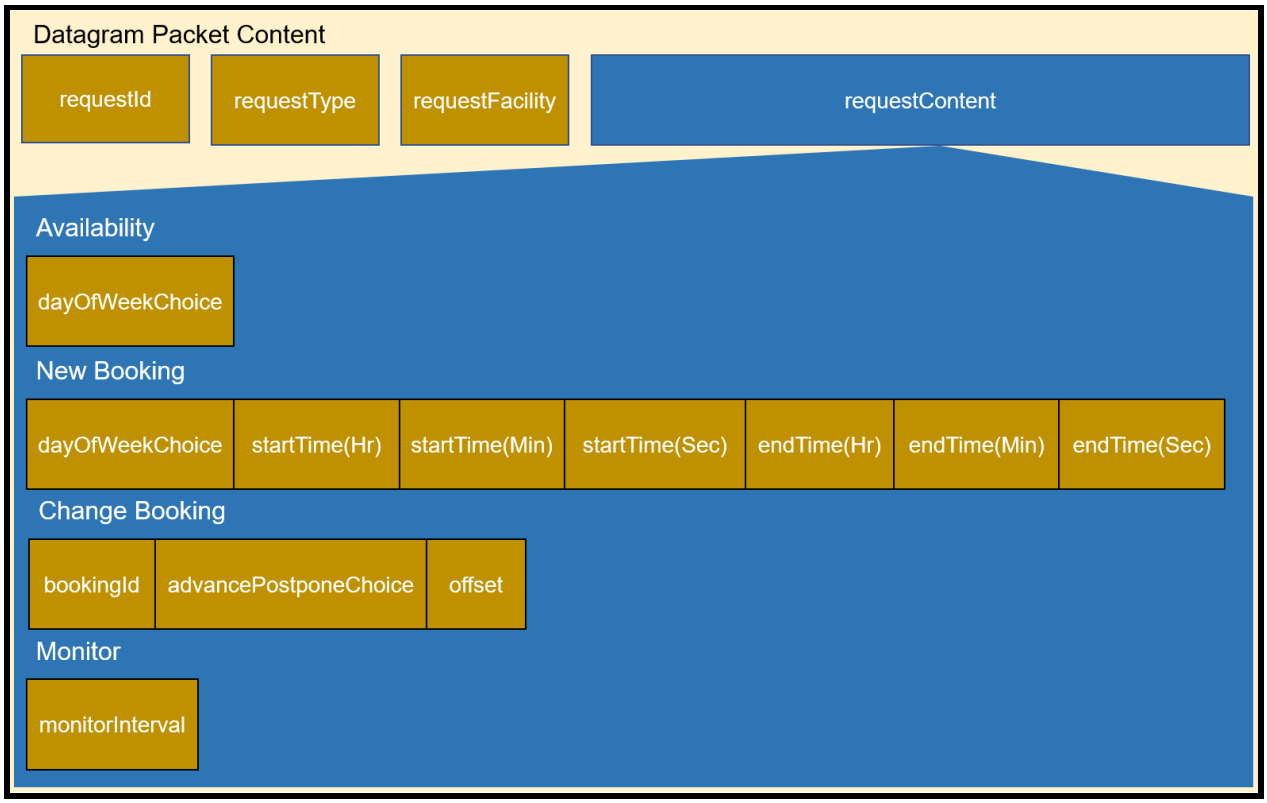

Figure 3: Request Message Content Structure

The above figure depicts the content of a request message from a client to server. The Datagram Packet Content contains a requestId, requestType, requestFacility and requestContent. These fields are of String type and concatenated using the character '/' as a separator such that the server is able to differentiate them. The requestId is a unique ID for identifying the request message and differentiating new requests from retransmissions. A requestType indicates if the request is checking for availability, creating a new booking, change of a current booking, or monitoring a facility. The requestContent contains the parameters required for processing the respective requestType operations. These parameters are of String type and concatenated by a ',' such that the server is able to differentiate them.

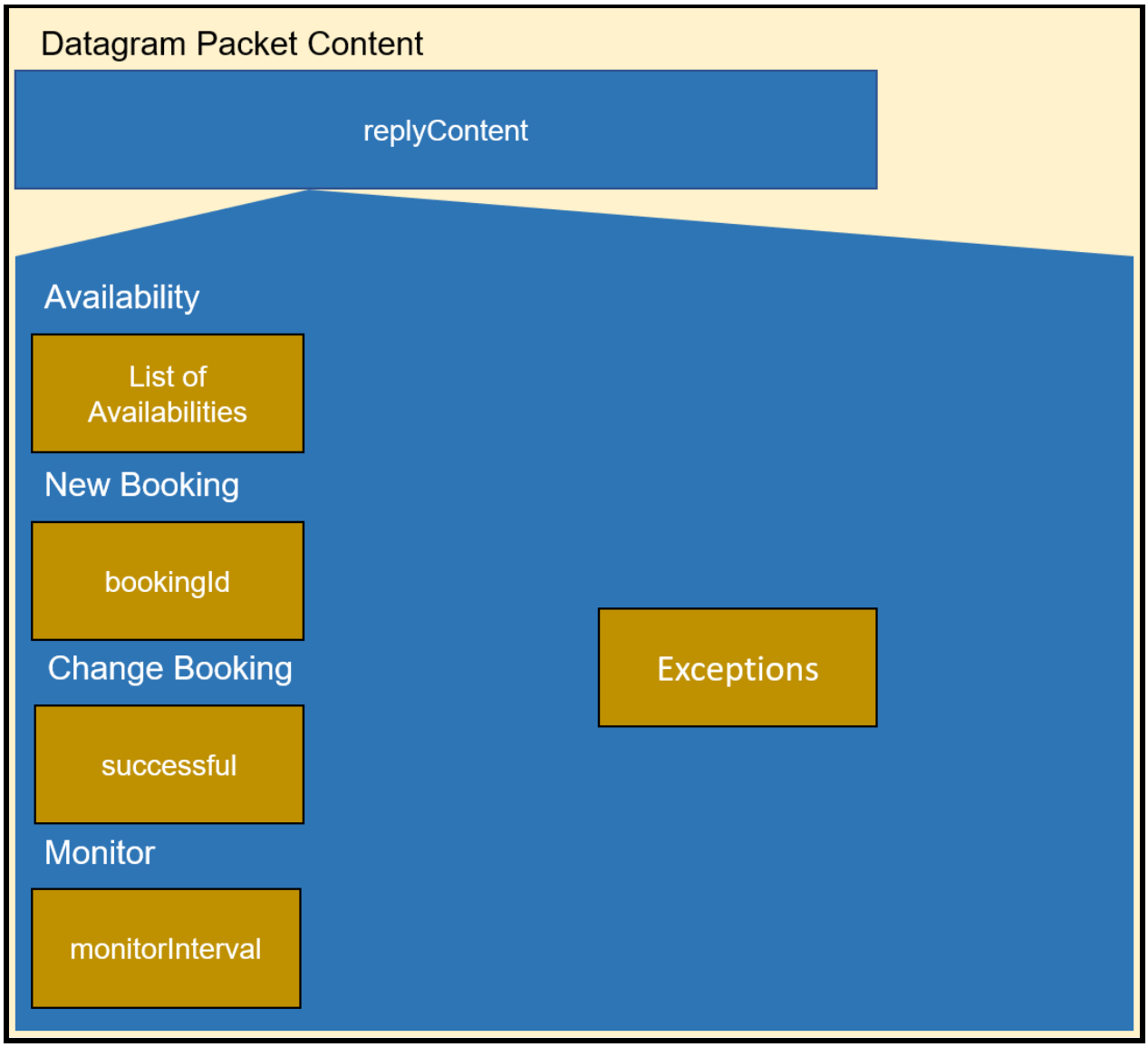

Figure 4: Reply Message Content Structure

After completion of the requestType operations, the return type of the response varies with respect to the requestType. In the event of exceptions, the error message will be returned as a String type. The reply content would then be serialised into a byte array before being returned to the server.

Client Implementation

The client provides a console-based interface where the user can access the functionality of the booking system. It consists of 4 classes, namely Main, Client, SendRecv and UserInputTools.

The Main class is used to display the main menu of the client application to the user. It also initializes the datagram socket used to send and receive data from the server and instantiates the necessary instances of classes Client and SendRecv with the necessary information such as the server IP and port, and the client port. Once the user has selected which service to use, the relevant method of Client is called.

The Client class is called to execute the necessary operations for the services. It contains the logic for each of the different services and prompts the user to provide the relevant data needed for the server to successfully execute those operations. Once the client has provided the data, it is then put together into a String, and passed on as parameters to the sendRequest method of SendRecv.

The SendRecv class handles interprocess communication through the network between the client and the server. The actions taken is dependent on the type of request selected, through a parameter passed when calling the method. A unique request ID is generated so that the server is able to distinguish if a request had been executed before and enables the implementation of at-most-once invocation capabilities on the server. The request data is then combined into a string and marshalled by the Serializer class, before being sent over the network as a UDP packet. It also includes timeout and retransmit functionality in the event that the client does not receive a response from the server. Once a response is received from the server, it is unmarshalled by the Deserializer class. The received data is processed based on the original type of request selected, and the results of the operation is displayed to the user.

The UserInputTools class is used to handle input from the user. Because of the need to repeatedly capture various types of data (e.g. String, Integer, LocalTime) from the user multiple times in both Main and Client, these functions are encapsulated in a separate class where they can be called. It also ensures proper exception handling as the input data is checked if it matches its intended output and prompts the user to re-enter the data if it is erroneous.

Server Implementation

The server provides the functionalities of unmarshalling datagram packets from the clients, retrieving the contents and executing the necessary operations on the database. The server also keeps track of the invocation semantics (at-least-once or at-most-once) to ensure that non-idempotent operations do not get executed multiple times due to retransmission of client requests. The server consists of 2 classes, a communication module (server.java) and a skeleton (handler.java) for invoking operations that satisfy the requirements of the program.

Server.java opens a datagram socket to listen for connections on a port. This port has to be made known to the client beforehand. When a datagram packet is received by the server, it contains the client IP address, client port, and message content in a byte array. This byte array has to be deserialized to retrieve the message content in the actual object type (string in this program). As explained in the request-reply message structure, the message content and order of arguments within varies with the message type.

Before the operations and respective arguments are passed to the database to be executed, the server first checks if the client request is a retransmission with the use of a hashmap. The hashmap stores previously completed request and response. In the event that the invocation semantics is set to at-most-once, the server would check for the presence of a previously sent response for a requestId in the hashmap. If a requestId and response key-value pair is found, the response would be retransmitted to the client without re-execution of the operation. This prevents non-idempotent operations from being executed more than once. Upon return of response from the database, the handler.java serializes the response into a byte array for transmission back to the client.

In addition, the handler.java maintains a list of monitors which will be updated each time an operation is executed, with default set to null. The list of monitors contain the clients that are monitoring the facility involved in the former operation. If the handler completes a booking or change booking operation, the list of monitors is updated to reflect the clients that need to be notified of the changes.

To simulate packet loss over the network, the server implements a check using a random math value upon receiving a datagram packet and before sending a datagram packet. If the random math value is less than the packet loss value that is configurable, the client request is not processed to simulate request message loss or the reply message is not sent to simulate reply message loss.

Cache / Database Implementation

Assumptions

- Facilities and Availabilities are pre-generated and will not undergo any changes. Thus, only Read (SELECT) operations needed for this 2 classes
- Availabilities for a given facility can be different between different days, but are the same for each individual day. E.g. Monday and Tuesday can have different booking availabilities, but every Mondays will have the same availability
- Bookings are made for a given day, and not a date.

Database Design

We are storing our data using a Postgres relational database. The server will interact and perform CRUD operations through the models classes, which will directly interact with the database through SQL queries, using the Postgres Driver for Java.

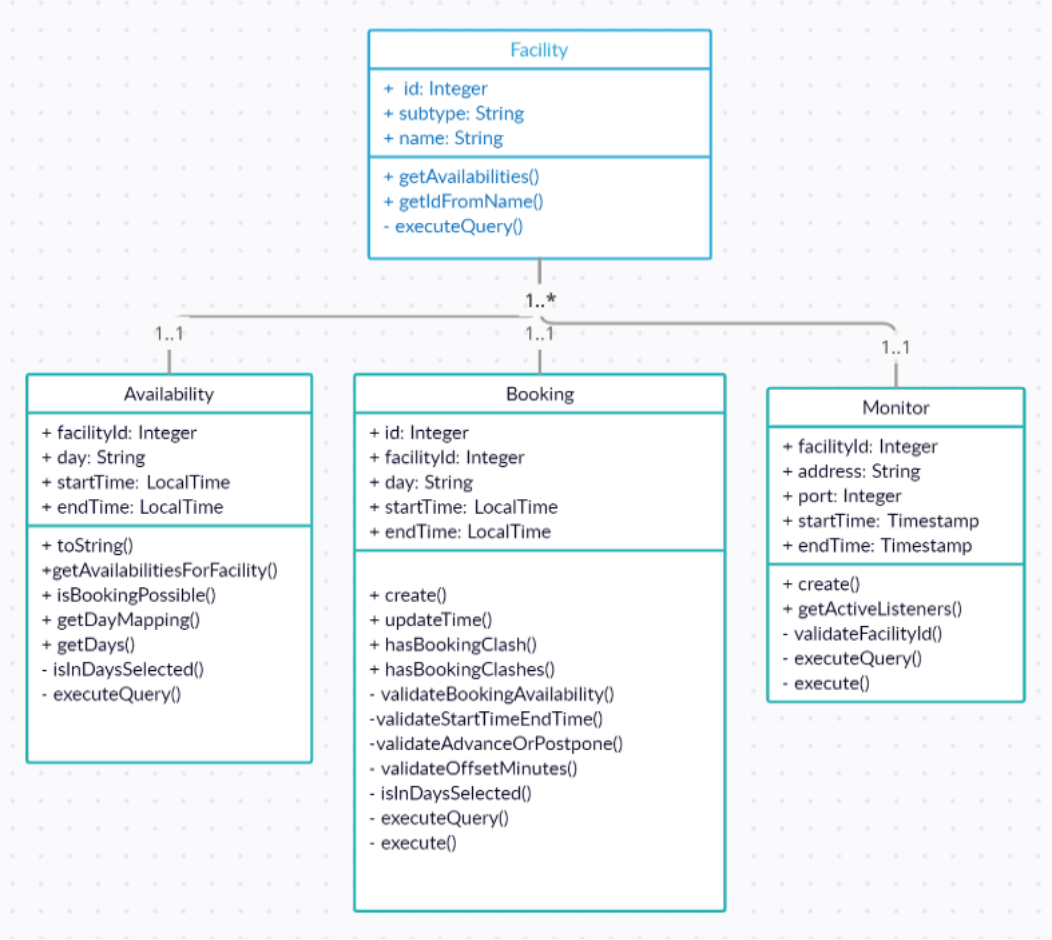

Figure 5: Class diagram of the models involved

Models:

- Facility - List of facilities
- Availability - List of availabilities of a given facility, for a given day of the week
- Booking - List of bookings made by users
- Monitor - List of listeners, as well as their active period

Cache Implementation

When a read operation is executed, the model will first check if the existing records are in the cache instance. If yes, it will retrieve the information from the cache. Otherwise, it will make a call to the database to retrieve this information.

When a create/update operation is executed, the model will perform those operations on the database, and then update the cache with the updated value. If the cache for a particular class is full, we will evict one of the entries in the cache using a Random Replacement Policy to free up space for the new values.

```
public class MonitorCache extends Cache {

  public static int size = 2;

  public static HashMap<Integer, List<Monitor>> cache = new HashMap<Integer, List<Monitor>>();

  // Get the total size of the cache
  public static int getSize() { ... }

  // Retrieve cached objects based on ID of a facility
  public static List<Monitor> get(Integer facilityId) { ... }

  // Insert single object to be cached
  // Note: only add to cache if it's facility_id already exist in cache (for simplicity)
  public static void put(Monitor monitor) { ... }

  // Insert objects to be cached
  // Evict record from cache if cache is full
  public static void put(List<Monitor> monitors) { ... }

  // Generate a randomized cache index to evict
  private static void evictRandomCacheEntry() { ... }

}
```

Figure 6: Structure of a typical cache class implemented

Experiments on Invocation Semantics

At-most-once

In our application, we implement at-most-once invocation semantics through the use of a unique identifier for each and every request. This is the default behavior of our application. When the client sends a request, it assigns a unique identifier to the request, so that the server can identify duplicate requests and prevent the request from being executed more than once. This is particularly important for non-idempotent functions such as the Change Booking service, which offsets a booking by a specified interval. Should this function be executed more than once, then the booking could be offset by an interval more than what was specified by the user.

In our experiment, we set the probability of packet loss on the server to **75%.** When the client sends a request, and fails to receive a response, the client will timeout and send the same request once again. In the case where the response is lost, the server will recognize the request identifier as having been executed before, and send the response to the client once again. There is no issue here as the non-idempotent function of Change Booking is only executed once, and the end result reflects the intentions of the user.

At-least-once

To implement at-least-once invocation semantics in our application, we disable the logic used to determine if a request had been executed before on the server. As such, when the server receives a request, it disregards the request identifier and simply executes the function, regardless if it was executed before. This behavior has no tangible effect on idempotent functions such as Check Availability, apart from using up computing resources. As mentioned earlier, at-least-once invocation semantics can result in unintended consequences for non-idempotent functions.

To compare with at-most-once invocation semantics, we use the same variables to perform this experiment. The behavior of the client does not change, and thus it will continue to retransmit requests if a response is not received. In this case, because the server does not check if the request has been executed before, the Change Booking function is executed multiple times as a result of at-least-once invocation semantics. This results in the booking being offset far beyond what the user had originally intended.