Booking System

Drosou Maria

28/05/2020

Contents

Introduction

In the present work a Java RMI hotel room reservation system was implemented. The hotel was considered to have the following rooms:

- 30 type A rooms (single rooms) which cost 50 Euro per night
- 45 type B rooms (double rooms) which cost 70 Euro per night
- 25 type C rooms (twin rooms) which cost 80 Euro per night
- 10 type D rooms (triple rooms) which cost 120 Euro per night
- 5 type E rooms (quad rooms) which cost 150 Euro per night

The hotel java RMI server can handle the following requests from a client:

- Rooms list display
- Reservation of a number of rooms of a specific type in one name (guest)
- Display hotel guests
- Cancel booking of a number of rooms of a specific type by one name

In addition, in case of non-availability of several rooms for a reservation, the customer can subscribe to a list and be informed of any cancellations in reservations of the specific room type. This is achieved with the Callback method.

The Java files created to implement the above are the following:

Server side:

- **HRServer.java**
  Includes the class that has the main function and starts the Server.
- **HRImpl.java**
  Includes the basic Server class, which contains the list of rooms and clients, as well as all the methods used by the Server. The object of this file is created in the main of HRServer.java and with RMI Registry some of its methods are communicated to the client so that it can call them.
- **HRInterface.java**
  Includes reference to HRImpl.java methods, which the client can also access through the RMI Registry.
- **Room.java**
  Includes information of a room type, ie the name of the type (A, B, C, D or E), the number of this type rooms available and the price. It also has methods that return this information. Used only by the Server in the HRImpl.java file which contains a list with the 5 Room type objects.
- **Client.java**
  Includes the data of a customer, ie his name and his reservations together with the corresponding methods of returning this data. This code is used only by the Server for keeping a list of hotel guests.
- **HRInterface.idl**
  Includes reference to the methods declared in the HRInterface.java file, but in CORBA IDL this time. In this way and with other actions, which were not implemented in this project, communication between client and server can be achieved, even if they are written in a different programming language.

Client side:

- **HRClient.java**
  Includes the class that has the main function and starts the Client.
- **CancelEventListener.java**
  Includes reference to the only method of the customer, called by the Server using the Callback technique and in order to inform the customer about booking cancellations for rooms that the customer has expressed interest.

The above is explained in more detail in the documentation that follows.

Code documentation

Java RMI Server

The files HRServer.java, HRImpl.java, HRInterface.java, Room.java, Client.java, HRInterface.idl were created for the implementation of the Server, as mentioned above..

The latter is not used anywhere in the present implementation. Further reference to this and the role it could play if used could be found in the CORBA IDL subsection. The remaining files are presented and explained below.

First of all, Room.java and Client.java files are two classes used by the Server to store information for each room type and each client respectively. More specifically, the content of the Room class is as follows:

```
public class Room {
	private String type;
	private int availableRooms;
	private long price;

	public Room(String type, int availableRooms, long price) {
		this.type = type;
		this.availableRooms = availableRooms;
		this.price = price;
	}

	public void setType(String type) {
		this.type = type;
	}

	public void setAvailableRooms(int availableRooms) {
		this.availableRooms = availableRooms;
	}

	public void setPrice(long price) {
		this.price = price;
	}

	public String getType() {
		return type;
	}

	public int getAvailableRooms() {
		return availableRooms;
	}

	public long getPrice() {
		return price;
	}

	public String roomToString() {
		String result = availableRooms + " " + type + " type rooms - " + price + " euros
per night";
		return result;
	}
}
```

As shown above it includes the name of the room type (A, B, C, D or E), the number of available rooms of this type and the price corresponding to this type. As methods it has the getters and setters that give access to its data from the Server class, as well as a method called roomToString () which returns all the data of the class as a String.

Respectively the Client class has the following form:

```
import java.util.HashMap;

//Class for guest (client)
//which contains their name and their reservations
public class Client {
	private String name;
	private HashMap<String, Integer> reservations;
```

```
	public Client(String name) {
		this.name = name;
		reservations = new HashMap<String, Integer>();
		reservations.put("A", 0);
		reservations.put("B", 0);
		reservations.put("C", 0);
		reservations.put("D", 0);
		reservations.put("E", 0);
	}

	public void setName(String name) {
		this.name = name;
	}

	public void setReservations(HashMap<String, Integer> reservations) {
		this.reservations = reservations;
	}

	public String getName() {
		return name;
	}

	public HashMap<String, Integer> getReservations() {
		return reservations;
	}

	public String clientToString() {
		int aNum = reservations.get("A");
		int bNum = reservations.get("B");
		int cNum = reservations.get("C");
		int dNum = reservations.get("D");
		int eNum = reservations.get("E");
		String result = "";
		result += name + ": \n";
		result += "\t- Type A rooms booked: " + aNum + "\n";
		result += "\t- Type B rooms booked: " + bNum + "\n";
		result += "\t- Type C rooms booked: " + cNum + "\n";
		result += "\t- Type D rooms booked: " + dNum + "\n";
		result += "\t- Type E rooms booked: " + eNum + "\n";
		long totalAmount = aNum * 50 + bNum * 70 + cNum * 80 + dNum * 120 + eNum * 150;
		result += "\tTotal cost: " + totalAmount + " euros";
		return result;
	}

}
```

As shown the class details are the customer's name and reservations. Reservations are saved in Hashmap format per room type, so it is easier to find the reservation that corresponds to a specific room type. As methods the class includes, like the previous one, the getters and setters and a clientToString () method, which returns all the data of the class in String format.

The above two classes are used by the HRImpl class. This class implements the entire Server component, except the main function. Because its content is large, it is presented in sections and each piece is explained below.

Initially this class extends UnicastRemoteObject and implements HRInterface and Runnable. The first two are necessary to be able to access some class methods from HRClient. HRInterface includes a statement of these methods and its contents are as follows:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.ArrayList;

public interface HRInterface extends Remote {
	public String listFunction() throws RemoteException;
	public long bookFunction(String type, int number, String name) throws RemoteException;
	public String guestsFunction() throws RemoteException;
	public String cancelFunction(String type, int number, String name) throws RemoteException;
	void addCancelEventListener(CancelEventListener ceListener, String type)
		throws RemoteException;
```

```
        void removeCancelEventListener(CancelEventListener ceListener, String type)
                throws RemoteException;
}
```

As it is shown, the methods that HRClient has access to are:

- **String listFunction()**
  Returns as String the list of rooms of each type and their information (availability and price).
- **long bookFunction(String type, int number, String name)**
  Executes the process of booking 'number' of type ‘type’ rooms in the name 'name' and returns long which is equal to the number of rooms if it fails or the total cost of the booking if it succeeds.
- **String guestsFunction()**
  Returns as String the list of hotel guests and the information for each one of them (name and reservations).
- **String cancelFunction(String type, int number, String name)**
  Executes type ‘type’ room cancellation from client with name ‘name’. If it fails it returns a failure message, otherwise it returns a String with the updated customer information (name and reservations).
- **void addCancelEventListener(CancelEventListener ceListener, String type)**
  Performs the addition of an HRClient to the list of people interested in being notified whenever a booking cancellation is made for type ‘type’ rooms.
- **void removeCancelEventListener(CancelEventListener ceListener, String type)**
  Performs the removal of an HRClient from the list of people interested in being notified each time a booking cancellation of type ‘type’ room is made.

All of the above methods throw RemoteException due to the possibility that something may not work properly when HRClient communicates with them.

Therefore the HRImpl class extends the UnicasRemoteObject and implements the HRInterface presented above to enable the HRClient to access the above methods. However, the Callback technique is used to inform the interested customers about cancellations. The implementation of this technique needs to be done asynchronously, so that at the same time many customers are served, without one process blocking the execution of the other. This is why a thread is used to inform customers about any booking cancellations in rooms that interest them. In order for this to work, the HRImpl class also implements Runnable.

The following is the first part of the HRImpl.java file that contains the input of the necessary libraries, and the first part of the HRImpl class that contains its data (attributes), and the constructor:

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.HashMap;

public class HRImpl extends UnicastRemoteObject
implements HRInterface, Runnable {

        private ArrayList<Client> clients; //list of guests
        private ArrayList<Room> rooms; //list of rooms
        //list of listeners for each room type:
        private ArrayList<CancelEventListener> aCancelEventListerners;
        private ArrayList<CancelEventListener> bCancelEventListerners;
        private ArrayList<CancelEventListener> cCancelEventListerners;
        private ArrayList<CancelEventListener> dCancelEventListerners;
        private ArrayList<CancelEventListener> eCancelEventListerners;
        //flags which become true after a cancel in order to inform run()
        //that it has to notify the corresponding listeners for the cancellation
        volatile boolean aCancelEventFlag;
        volatile boolean bCancelEventFlag;
        volatile boolean cCancelEventFlag;
        volatile boolean dCancelEventFlag;
        volatile boolean eCancelEventFlag;
```

```
//constructor
public HRImpl() throws RemoteException{
	super();
	//initialise all lists
	rooms = new ArrayList<Room>();
	clients = new ArrayList<Client>();
	aCancelEventListerners = new ArrayList<CancelEventListener>();
	bCancelEventListerners = new ArrayList<CancelEventListener>();
	cCancelEventListerners = new ArrayList<CancelEventListener>();
	dCancelEventListerners = new ArrayList<CancelEventListener>();
	eCancelEventListerners = new ArrayList<CancelEventListener>();

	//initialise flags to false which means there is
	//no need for notifying CancelEventListeners for now
	aCancelEventFlag = false;
	bCancelEventFlag = false;
	cCancelEventFlag = false;
	dCancelEventFlag = false;
	eCancelEventFlag = false;

	//add existing rooms
	rooms.add(new Room("A", 30, 50));
	rooms.add(new Room("B", 45, 70));
	rooms.add(new Room("C", 25, 80));
	rooms.add(new Room("D", 10, 120));
	rooms.add(new Room("E", 5, 150));
}
```

As shown above, the class data are:

- 1 list with Room class objects
- 1 list with Client class objects
- 5 lists with customers who are interested to be informed in case of cancelations (CancelEventListener class objects), one for each room type
- 5 boolean flags used to indicate when a cancellation was made in the respective room type (become true after a cancellation and false again when the corresponding customers have been informed)

The above elements are initialized in the constructor. The flags are all set to false and the room list is filled with five room types that include availability and price information (one Room class object for each room type).

The following is the listFunction () function that returns a String with the rooms and the corresponding information for each room type:

```
public String listFunction() throws RemoteException {
	System.out.println("listFunction called.");
	String result = "";
	if(rooms.isEmpty())
		result = "There are no rooms";
	else
		for (Room r : rooms) {
			result += (r.roomToString() + "\n");
		}
	return result;
}
```

As shown, the function runs through the list of rooms and passes their information to the returning String using the roomToString () function that belongs to the Room class. The function also predicts the possibility that there are no rooms in the hotel, returning a corresponding message. Of course, this is not really possible, since the list of rooms is filled from the beginning through the constructor and never empties, just the information of the availability of the rooms is updated.

The following is the bookFunction () function that takes over the room reservation process:

```
public long bookFunction(String type, int number, String name) throws RemoteException {
	long result = number;
	boolean clientFlag = false;
	System.out.println("bookFunction called.");
	for (Room r : rooms) {
		if(r.getType().equals(type))
			if(r.getAvailableRooms() >= number) {
				for (Client c : clients) {
					if(c.getName().equals(name)) {
						clientFlag = true;
						HashMap<String, Integer> reservations =
								new HashMap<String, Integer>();
						reservations = c.getReservations();
						reservations.put(type, reservations.get(type) +
								number);
						c.setReservations(reservations);
						result = number * r.getPrice();
					}
				}
				if(!clientFlag) {
					Client c = new Client(name);
					HashMap<String, Integer> reservations =
							new HashMap<String, Integer>();
					reservations = c.getReservations();
					reservations.put(type, reservations.get(type) + number);
					c.setReservations(reservations);
					clients.add(c);
					result = number * r.getPrice();
				}
				r.setAvailableRooms(r.getAvailableRooms() - number);
			}
			else {
				result = r.getAvailableRooms();
			}
	}
	return result;
}
```

As shown above, this function runs through the list of room types (with a for loop), in order to identify the type requested and to check if the availability of this type of room is sufficient to be able to make the reservation of the 'number' of rooms requested. If the availability is high enough, it checks the list of clients in the same way in order to locate the customer by the name given. If the customer is found, the function adds the corresponding number of rooms in the customer's reservations and assigns to the output variable (result) the cost of the reservation (number of rooms multiplied with the price). If the customer is not found, the function creates a new one and does the same process. It then subtracts from the availability of the respective room type the number of rooms for which the reservation was made. If the rooms requested by the customer are more than available, then the booking process is not completed and the number of available rooms is returned. So, if HRClient calls this function and it results in a number less than the number of rooms requested it means that the booking has failed, otherwise the booking has succeeded and the returned value corresponds to the total cost.

The following is the guestsFunction () function that returns the list of clients as a Sting:

```
public String guestsFunction() throws RemoteException {
	System.out.println("guestsFunction called.");
	String result = "";
	if(clients.isEmpty())
		result = "There are no guests";
	else
		for (Client c : clients)
			result += (c.clientToString() + "\n");
	return result;
}
```

As shown above, the function runs through the client list and adds the information of each to the returned String (result). If it does not find customers, a corresponding message is assigned to the result.

The following is the cancelFunction () function that undertakes the cancellation of reservations:

```
	public String cancelFunction(String type, int number, String name) throws RemoteException
{
		String answer = "Cancel failed - Rooms not found";
		boolean clientFlag = false;
		System.out.println("cancelFunction called.");
		for (Room r : rooms)
			if(r.getType().equals(type)) {
				for (Client c : clients)
					if(c.getName().equals(name)) {
						clientFlag = true;
						HashMap<String, Integer> reservations =
								new HashMap<String, Integer>();
						reservations = c.getReservations();
						if(reservations.get(type) >= number) {
							reservations.put(type, reservations.get(type) -
									number);
							c.setReservations(reservations);
							r.setAvailableRooms(r.getAvailableRooms() +
									number);
							answer = c.clientToString();
							//inform other clients for cancellation
							if (type.equals("A"))
								aCancelEventFlag = true;
							else if (type.equals("B"))
								bCancelEventFlag = true;
							else if (type.equals("C"))
								cCancelEventFlag = true;
							else if (type.equals("D"))
								dCancelEventFlag = true;
							else if (type.equals("E"))
								eCancelEventFlag = true;
						}
					}
				if (!clientFlag)
					answer = "Cancel failed - Guest not found";
				}
		return answer;
	}
```

As shown above, the function goes through the list of room types and if it identifies the type of room for which the cancellation is, it also runs through the list of customers to locate the customer who wants to cancel. If the customer reservations for this room type are at least equal to the number of rooms to be canceled, then the corresponding number of rooms is deducted from the customer reservations and added to the availability of this room type. The flag of the respective room type is then made true, in order to notify any interested customers about the cancellation. If the name of the customer requesting the cancellation is not found, then the above procedure is not performed. The output in case of cancellation is defined as a String with the updated information of the customer reservations (result of the clientToString function of the customer). On the contrary, in case of failure, a corresponding message is returned.

Following are functions related to the Callback part for informing the interested customers for any cancellations. Initially a function undertakes the addition of interested customers to the corresponding list. This function is addCancelEventListener (), the contents of which are as follows:

```
	public synchronized void addCancelEventListener(CancelEventListener ceListener,
							String type) throws RemoteException
	{
		if (type.equals("A"))
			aCancelEventListerners.add(ceListener);
		else if (type.equals("B"))
			bCancelEventListerners.add(ceListener);
		else if (type.equals("C"))
			cCancelEventListerners.add(ceListener);
		else if (type.equals("D"))
			dCancelEventListerners.add(ceListener);
		else if (type.equals("E"))
			eCancelEventListerners.add(ceListener);
```

```
        System.out.println("Added new listener for " + type + " type rooms");
}
```

As can be seen, the function receives as input the CancelEventListener which is the interface of the interested customer. In addition, it accepts as input the type of room for which the customer is interested, in order to register them in the respective list and notify them only for cancellations in rooms of this type. On its body it performs the aforementioned registration and prints a relevant message on the Server.

Correspondingly, the removeCancelEventListener () function is implemented, which takes the same inputs to remove a client from the corresponding list. Its content is the following:

```
public synchronized void removeCancelEventListener(CancelEventListener ceListener,
                                                    String type) throws RemoteException
{
        boolean successFlag = false;
        if (type.equals("A"))
                if (aCancelEventListerners.remove(ceListener))
                        successFlag = true;
        else if (type.equals("B"))
                if (bCancelEventListerners.remove(ceListener))
                        successFlag = true;
        else if (type.equals("C"))
                if (cCancelEventListerners.remove(ceListener))
                        successFlag = true;
        else if (type.equals("D"))
                if (dCancelEventListerners.remove(ceListener))
                        successFlag = true;
        else if (type.equals("E"))
                if (eCancelEventListerners.remove(ceListener))
                        successFlag = true;
        if(successFlag)
                System.out.println("Removed listener for " + type + " type rooms");
        else
                System.out.println("Listener is already removed");
}
```

In addition to the functions that add a customer to the corresponding list, there is also the function that notifies the clients of the corresponding list. This function is notifyCancelEventListeners () and its format is as follows:

```
private synchronized void notifyCancelEventListeners(String type,
ArrayList<CancelEventListener> cancelEventListerners) {
        ArrayList<CancelEventListener> toRemove = new ArrayList<CancelEventListener>();
        int availability = 0;
        //get updated availability of rooms of this type
        //so that it will be send to clients via method roomsCancelled
        for (Room r : rooms)
                if(r.getType().equals(type))
                        availability = r.getAvailableRooms();
        for (CancelEventListener ceListener : cancelEventListerners) {
                try {
                        ceListener.roomsCancelled(type, availability);
                }
                catch (RemoteException aInE) {
                        // If the listener is not responding
                        // and the call to the listener's method fails,
                        // remove listener from the list
                        System.out.println("Removing listener");
                        toRemove.add(ceListener);
                }
        }
        for (CancelEventListener toRemListener : toRemove) {
                if (cancelEventListerners.contains(toRemListener)) {
                        cancelEventListerners.remove(toRemListener);
                        System.out.println("Removed listener for " + type + " type rooms");
                }
        }
}
```

As can be seen, the above function is what the Callback executes, that is, it calls a function of the client, roomsCancelled (). This function accepts as inputs the type of room and the availability and prints a corresponding message on the client side. Thus, the customer is notified that a cancellation has been made and is informed about the availability of the rooms of the respective type after this cancellation. There is a check in case the client has interrupted the communication with the Server, so in this case it is removed from the list of clients. This subtraction cannot be done directly because the function is currently running through the for loop client list. For this reason the clients to be removed are saved in a new local list and deleted when the for loop is finished. A corresponding information message is printed on the Server for each client that is removed.

Finally, the HRImpl class includes the run () method, which runs the thread that notifies the customers of each list in case of cancellation of rooms of the corresponding type. This method has the following content:

```
public void run() {
	System.out.println("Notifying thread started");
	for(;;) {
		if (aCancelEventFlag) {
			System.out.println("Notifying listerners for A type rooms
					cancellation");
			aCancelEventFlag = false;
			notifyCancelEventListeners("A", aCancelEventListerners);
		}
		if (bCancelEventFlag) {
			System.out.println("Notifying listerners for B type rooms
					cancellation");
			bCancelEventFlag = false;
			notifyCancelEventListeners("B", bCancelEventListerners);
		}
		if (cCancelEventFlag) {
			System.out.println("Notifying listerners for C type rooms
					cancellation");
			cCancelEventFlag = false;
			notifyCancelEventListeners("C", cCancelEventListerners);
		}
		if (dCancelEventFlag) {
			System.out.println("Notifying listerners for D type rooms
					cancellation");
			dCancelEventFlag = false;
			notifyCancelEventListeners("D", dCancelEventListerners);
		}
		if (eCancelEventFlag) {
			System.out.println("Notifying listerners for E type rooms
					cancellation");
			eCancelEventFlag = false;
			notifyCancelEventListeners("E", eCancelEventListerners);
		}
	}
}
```

As shown above, the function checks each flag that corresponds to each room type and if it is true, then a corresponding message is sent to the Server for notifying, the flag becomes false and notifyCancelEventListeners () is called with the appropriate parameters to notify the corresponding client list about the cancelation of a corresponding room type. That is, the flags become true after each successful cancellation of the respective room type via cancelFunction () and when the thread finds them true it returns them to false and notifies the clients via notifyCancelEventListeners (). For this reason flags are set to volatile, in order to be aware of changes in their value from all running processes. Otherwise if cancelFunction () changed the value of a flag to true, the thread function would not know it and would not inform the interested customers about the cancellation.

The above function completes the contents of the HRImpl class. The latest Server file is HRServer.java which is the class from which the Server starts. This class has the main () function and its contents are as follows:

```
import java.io.*;
import java.rmi.*;
import java.rmi.registry.LocateRegistry;

public class HRServer {
```

```
	public static void main(String argv[]) {
		try { //special exception handler for registry creation
			LocateRegistry.createRegistry(1099);
			System.out.println("java RMI registry created.");
		} catch (RemoteException e) {
			//do nothing, error means registry already exists
			System.out.println("java RMI registry already exists.");
		}

		try {
			HRImpl hrI = new HRImpl();
			Naming.rebind("rmi://localhost/HRServer", hrI);
			Thread lThread = new Thread(hrI);
			lThread.start();
		} catch(Exception e) {
			System.out.println("HRServer: "+e.getMessage());
			e.printStackTrace();
		}
	}
}
```

As shown above, this class creates the RMI Registry, if it does not already exist, and then creates an object of the HRImpl class, which corresponds to rmi: // localhost / HRServer. Through this path the client will be able to access the methods that exist in the interface of the class of this object. Then the thread that undertakes to inform customers about cancellations is created and starts running. In case any of the above fails the corresponding exception is printed.

This class completes the Server part of the server. The Client part follows.

Java RMI Client

For the Client side only two files were implemented. One is the interface that states the roomsCancelled () method used during the callback. Its content is as follows:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

// This interface's method is implemented by the client application used by the server
// application to notify the client for cancellations.
public interface CancelEventListener extends Remote
{
    void roomsCancelled(String type, int availability) throws RemoteException;
}
```

This interface is implemented by the HRClient class that is implemented in the second file which includes the main function that runs the Client. This class also extends UnicastRemoteObject so that the Server can access the roomsCancelled () method via the interface above.

Because the HRClinet.java file is relatively large, it is presented in sections below to make it easier to understand its structure and explain its methods. Initially in the first part of the file are declared the libraries, the beginning of the class and its constructor:

```
import java.io.*;
import java.rmi.*;
import java.util.ArrayList;
import java.util.Scanner;
import java.rmi.server.UnicastRemoteObject;

// The client class, implements the client interface
public class HRClient extends UnicastRemoteObject implements CancelEventListener {

	// Constructor method, does nothing
	public HRClient() throws RemoteException{
	super();
    }
```

The constructor is followed by a function that takes over the printing of instructions for the correct HRClient call, ie the client process:

```
//function for printing usage depending on the client's call mistake
public static void printUsage(int flag) {
	if (flag == 0) {
		System.out.println("Usage: java HRClient parameters");
		System.out.println("Possible parameters:");
		System.out.println("\tlist <hostname>");
		System.out.println("\tbook <hostname> <type> <number> <name>");
		System.out.println("\tguests <hostname>");
		System.out.println("\tcancel <hostname> <type> <number> <name>");
	}
	else if (flag == 1) {
		System.out.println("Usage: java HRClient list <hostname>");
	}
	else if (flag == 2) {
		System.out.println("Usage: java HRClient book <hostname> <type> <number>
				<name>");
	}
	else if (flag == 3) {
		System.out.println("Usage: java HRClient guests <hostname>");
	}
	else if (flag == 4) {
		System.out.println("Usage: java HRClient cancel <hostname> <type> <number>
				<name>");
	}
}
```

As shown above, in case the client does not enter the first parameter correctly, (correct arguments: list / book / guests / cancel), then the function prints a message regarding the correct HRClient call for the specific parameter. Otherwise the correct HRClient call is printed for each parameter case. Which instructions will be printed each time depends on a flag that the function takes as input. In the main, where the parameters given by the user are checked, depending on the error is called printUsage () with the corresponding flag.

This is followed by bookProcess (), which calls the bookfunction () of the Server whose interface it takes as input. The content of the function is as follows:

```
private static boolean bookProcess(HRInterface hrI, String type, int number, String name)
					throws RemoteException
{
	Scanner input = new Scanner(System.in);
	long result = number;
	boolean exitFlag = false;
	boolean successFlag = false;
	while (!exitFlag) {
		//call server function for booking
		result = hrI.bookFunction(type, number, name);
		if(result >= number) {
			//if enough rooms were available the booking succeeded
			//and the result is the total cost
			System.out.println("Reservation succeeded");
			System.out.println("Total cost: " + result + " euros");
			exitFlag = true;
			successFlag = true;
		}
		else if(result == 0) {
			//if there are no rooms,
			//there is no point to ask guest if they want to book only them.
			//instead a message of failure is displayed:
			System.out.println("Reservation failed, because there are no type " +
					type + " rooms available.");
			exitFlag = true;
		}
		else {
			//if there are rooms but not enough
			//guest is asked if they want to book only them
			//if yes, then the booking process (while loop) will be
			//reapeated for the new number of rooms
			System.out.println("There are only " + result + " type " + type +
					" rooms.");
			System.out.println("Do you want to book only them? (y/n)");
			String answer = input.nextLine();
```

```
                        if(answer.equals("n"))
                                exitFlag = true;
                        else
                                number = (int) result;

                }
        }
        return successFlag;
}
```

As can be seen, the above function, in addition to reference to the Server, also takes as inputs the type of rooms to be booked, the number of these rooms and the name in which the booking will be registered. After calling the Server function (bookFunction ()) it is checked if the booking was successful. If it was successful, a corresponding message and the cost of the reservation made are printed and the function returns true. If it was not successful there are two possible cases. The first possibility is that there are no rooms available of this type at all, so a corresponding message comes out and the function terminates by returning false. The second possibility is to have rooms available, but these are less than required. In this case the user is asked if he wants to book only these rooms. If he answers yes, then the process for the new number of rooms is repeated. If the answer is no, then the function terminates by returning false.

For the other procedures (list, guests, cancel) no separate function was implemented, like the one above for the book, because they were simpler with fewer commands and it was preferred that their procedures be fully included in the main.

But before the main there is the implementation of another function, which is the one that is known to the Server through the interface, in order to make the Callback. This function, as mentioned above, is roomCancelled () and its contents are a simple message printout, as shown below:

```
//function called from server to inform interested clients for cancellations
public void roomsCancelled(String type, int availability)  throws RemoteException {
        System.out.println("Due to a cancellation there are " + availability + " " +
                        type +" type rooms available now.");
}
```

The following is the main function ():

```
public static void main(String argv[]) {
        if(argv.length < 1) {
                printUsage(0);
                System.exit(0);
        }

        String lookUpName = "";
        HRInterface hrI;
        Scanner input = new Scanner(System.in);

        if(argv.length > 1) {

                lookUpName = "rmi://" + argv[1] + "/HRServer";
        }

        if(argv[0].equals("list")) {
                if(argv.length < 2) {
                        printUsage(1);
                        System.exit(0);
                }
                try {
                        //call method for option list
                        hrI = (HRInterface) Naming.lookup(lookUpName);
                        String rooms = hrI.listFunction();
                        System.out.println(rooms);

                } catch(Exception e) {
                        System.err.println("HRServer exception: "+ e.getMessage());
                        e.printStackTrace();
                }
        }
```

```
        else if(argv[0].equals("book")) {
            if(argv.length < 5) {
                printUsage(2);
                System.exit(0);
            }
            try {
                //call method for option book
                hrI = (HRInterface) Naming.lookup(lookUpName);
                String type = argv[2];
                int number = Integer.parseInt(argv[3]);
                String name = argv[4];
                boolean success = bookProcess(hrI, type, number, name);
                if (!success) {
                    System.out.println("Would you like to be informed if there is a
                                cancel for type " + type + " rooms? (y/n)");
                    String answer = input.nextLine();
                    if(answer.equals("y")) {
                        HRClient hrC = new HRClient();
                        hrI.addCancelEventListener(hrC, type);
                        System.out.println("Listener registered");
                    }
                }

            } catch(Exception e) {
                System.err.println("HRServer exception: "+ e.getMessage());
                e.printStackTrace();
            }

        }
        else if(argv[0].equals("guests")) {
            if(argv.length < 2) {
                printUsage(3);
                System.exit(0);
            }
            try {
                //call method for option guests
                hrI = (HRInterface) Naming.lookup(lookUpName);
                String clients = hrI.guestsFunction();
                System.out.println(clients);

            } catch(Exception e) {
                System.err.println("HRServer exception: "+ e.getMessage());
                e.printStackTrace();
            }
        }
        else if(argv[0].equals("cancel")) {
            if(argv.length < 5) {
                printUsage(4);
                System.exit(0);
            }
            try {
                //call method for option cancel
                hrI = (HRInterface) Naming.lookup(lookUpName);
                String result = hrI.cancelFunction(argv[2], Integer.parseInt(argv[3]),
                                                   argv[4]);
                System.out.println(result);


            } catch(Exception e) {
                System.err.println("HRServer exception: "+ e.getMessage());
                e.printStackTrace();
            }
        }
        else {
            printUsage(0);
            System.exit(0);
        }

    }
```

As it appears, main first checks if the parameters are correct in terms of their number and if not, calls printUsage () with the appropriate flag to print the correct HRClient call and then terminates. It then gets the server host from the

second parameter given when calling the HRClient (argv [1]). It creates the lookup Name based on this host and then looks at the first parameter (argv [0]), to identify which process the client wants to perform (list, book, guests or cancel). If none of the existing tasks in the first argument is detected, then the message for the correct HRClient call is displayed again. If any of the processes is detected, then the number of arguments based on the specific process is checked, since each process has a different number of arguments. If the number of arguments for the process found is not correct, printUsage () is called in this case with the appropriate flag, in order to print the correct call of the HRClient for the specific process specifically.

If all went well and the arguments are correct, then the lookup () function returns an object of the Server interface. From this object is called in each case the function that performs the corresponding process and based on the return of this function the corresponding printing is done. An exception to this tactic is the case of the book, for which the corresponding Server function is not called directly. Instead, bookProcess () is called, which inside, as mentioned above, calls the corresponding Server function and prints the result. In addition, in case bookProcess () fails, main prints message to the user, asking if he wishes to be informed each time a booking cancellation is made about the type of room he tried to book, but did not managed. If the customer accepts, the main, instead of terminating, waits for any notifications from the Server for cancellations of reservations of the specific type of rooms. In any other case after the end of the respective procedure the main () terminates and therefore terminates the client program (HRClient), as it has fulfilled the purpose for which it was called.

CORBA IDL

All the above files are enough for the program from both the Server side and the Client side, to be fully functional. However, in the context of this work, another file was created which is a description of the Server interface (HRInterface) in CORBA IDL. This file with the appropriate actions allows the Server to communicate with clients that are written in a different programming language than its own. The contents of the HRInterface.idl file are as follows:

```
interface HRInterface {
	string listFunction();
	long long bookFunction(in string type, in long number, in string name);
	string guestsFunction();
	string cancelFunction(in string type, in long number, in string name);
};
```

As it appears, the above interface does not include the methods for adding customers to lists for Callback and removing them. Hence, the implemented CORBA IDL does not support the Callback process.

So, since the above file has been created, to achieve communication between a Server in Java and a Client in C++ the following actions are required:

1. Compile HRInterface.idl using idlj for the Server that is in Java and using omniORB for the Client that is in C++. This generates utilities that communicate between two different language programs..
2. To implement the interface from the Server, ie to write code for the functions mentioned in the declared interface.
3. Write the Java code of the Server which will instantiate (register in the Registry) the remote object.
4. Write the C ++ code of the Client, which communicates with the Registry and gets reference to the remote object, so that it can call the methods included in the corresponding interface of the Server.
5. Start the execution of the naming service
6. Start the execution of Server
7. Start the execution of Client

In case the Server is in C ++ and the Client in Java, the same procedure is done again with the difference that during the compile this time omniORB is used for the Server and idlj for the Client.

For RPC communication, instead of CORBA IDL, jRPC could be used, so that communication between programs written in a different language would be possible.

Test runs

Initially all the files presented above, except HRInterface.idl, are compiled with the following command in terminal:

javac *.java

A terminal window then opens in which the Server starts running with the following command:

java HRServer

The result of this command is shown below:

```
[user@localhost ergasia_2]$ java HRServer
java RMI registry created.
Notifying thread started
```

The Server has therefore started and is expecting calls from clients.

Then a new terminal window opens in which the client program is executed to view the available hotel rooms with the following command:

java HRClient list localhost

The result is as follows:

Server

```
listFunction called.
```

Client

```
[user@localhost ergasia_2]$ java HRClient list localhost
30 A type rooms - 50 euros per night
45 B type rooms - 70 euros per night
25 C type rooms - 80 euros per night
10 D type rooms - 120 euros per night
5 E type rooms - 150 euros per night

[user@localhost ergasia_2]$
```

In the next step, the customer process is performed again, but with the guests parameter this time to verify that for the time being the customer list is empty, ie there are no customers who have made any reservation at the hotel. The corresponding command is the following:

java HRClient guests localhost

The result is the following:

Server

```
guestsFunction called.
```

Client

```
[user@localhost ergasia_2]$ java HRClient guests localhost
There are no guests
[user@localhost ergasia_2]$
```

Then the customer procedure with book parameter is called to book 5 type D rooms in the name Maria, using the following command:

java HRClient book localhost D 5 Maria

Server

```
bookFunction called.
```

Client

```
[user@localhost ergasia_2]$ java HRClient book localhost D 5 Maria
Reservation succeeded
Total cost: 600 euros
[user@localhost ergasia_2]$
```

Then the client process is called again for booking but in another name for other rooms and then the client process is called with guest parameter to display the two clients. The corresponding commands are as follows:

java HRClient book localhost E 2 Nikos

java HRClient guests localhost

The results are the following:

Server

```
bookFunction called.
guestsFunction called.
```

Client

```
[user@localhost ergasia_2]$ java HRClient book localhost E 2 Nikos
Reservation succeeded
Total cost: 300 euros
[user@localhost ergasia_2]$ java HRClient guests localhost
Maria:
        - Type A rooms booked: 0
        - Type B rooms booked: 0
        - Type C rooms booked: 0
        - Type D rooms booked: 5
        - Type E rooms booked: 0
        Total cost: 600 euros
Nikos:
        - Type A rooms booked: 0
        - Type B rooms booked: 0
        - Type C rooms booked: 0
        - Type D rooms booked: 0
        - Type E rooms booked: 2
        Total cost: 300 euros

[user@localhost ergasia_2]$
```

Indeed, it is observed that the changes have been made to the users.

The client is then executed with 'list' option, to show that the rooms added to the customers were also removed from the availability in the respective room types. The results are shown below:

Server

```
listFunction called.
```

Client

```
[user@localhost ergasia_2]$ java HRClient list localhost
30 A type rooms - 50 euros per night
45 B type rooms - 70 euros per night
25 C type rooms - 80 euros per night
5 D type rooms - 120 euros per night
3 E type rooms - 150 euros per night

[user@localhost ergasia_2]$
```

It is observed that the type D rooms were actually reduced from 10 to 5 and the type E rooms were reduced from 5 to 3, as expected.

Then an attempt is made to rent 6 type D rooms in the name of Voula. Due to the fact that there are fewer type D rooms left, the process is not expected to succeed. The customer will therefore first be asked if she wants to close only the remaining rooms. If she refuses, she will be asked if she wants to be registered in the notification lists for any cancellations of room reservations type D. Answering yes, she will be registered in the list and will wait for notifications. The results of these actions are shown below:

Server

```
bookFunction called.
Added new listener for D type rooms
```

Client

```
[user@localhost ergasia_2]$ java HRClient book localhost D 6 Voula
There are only 5 type D rooms.
Do you want to book only them? (y/n)
n
Would you like to be informed if there is a cancel for type D rooms? (y/n)
y
Listener registered
```

Then the existing Nikos customer (in a new window, since both windows are now busy) tries to rent 6 type D rooms, with the difference that when asked if he wants to rent only what is left, he answers yes. The result is shown below:

Server

```
bookFunction called.
bookFunction called.
```

Client

```
[user@localhost ergasia_2]$ java HRClient book localhost D 6 Nikos
There are only 5 type D rooms.
Do you want to book only them? (y/n)
y
Reservation succeeded
Total cost: 600 euros
[user@localhost ergasia_2]$
```

The reason the Server bookFunction function was printed twice is because it was actually executed twice. The first time it was called with a parameter in the number of rooms to be booked at 6 and failed. Instead the second time it was called with the same parameter equal to the number of available type D rooms, ie 5, and succeeded.

In order to have a clear picture of what is valid so far, the client process with the guests parameter is called and the result is the following:

Server

```
guestsFunction called.
```

Client

```
[user@localhost ergasia_2]$ java HRClient guests localhost
Maria:
        - Type A rooms booked: 0
        - Type B rooms booked: 0
        - Type C rooms booked: 0
        - Type D rooms booked: 5
        - Type E rooms booked: 0
        Total cost: 600 euros
Nikos:
        - Type A rooms booked: 0
        - Type B rooms booked: 0
        - Type C rooms booked: 0
        - Type D rooms booked: 5
        - Type E rooms booked: 2
        Total cost: 900 euros

[user@localhost ergasia_2]$
```

It is noticed that Nikos new reservations have been added and the total amount is correspondingly increased. Also Voula who is waiting for any cancellations for type D rooms, has not been added to the list, as she has not made a reservation and therefore does not yet belong to the hotel guests.

A new customer, Petros, then tries to book 4 type E rooms, but fails because there are only 3 available and chooses to be notified of any cancellations:

Server

```
bookFunction called.
Added new listener for E type rooms
```

Client

```
[user@localhost ergasia_2]$ java HRClient book localhost E 4 Petros
There are only 3 type E rooms.
Do you want to book only them? (y/n)
n
Would you like to be informed if there is a cancel for type E rooms? (y/n)
y
Listener registered
```

So now there are two customers who have asked to be informed, one (Voula) for the type D rooms and one (Petros) for the type E rooms.:

Server

```
bookFunction called.
Added new listener for D type rooms
```

Client

```
[user@localhost ergasia_2]$ java HRClient book localhost D 8 Mina
Reservation failed, because there are no type D rooms available.
Would you like to be informed if there is a cancel for type D rooms? (y/n)
y
Listener registered
```

It is noticed that because all the type D rooms are now booked, this user is not asked if she wants to book the rest, since the rest is 0. The question asked directly is if she wants to receive cancellation notifications.

Here the available rooms of each type are displayed:

Server

```
listFunction called.
```

Client

```
[user@localhost ergasia_2]$ java HRClient list localhost
30 A type rooms - 50 euros per night
45 B type rooms - 70 euros per night
25 C type rooms - 80 euros per night
0 D type rooms - 120 euros per night
3 E type rooms - 150 euros per night
```

Subsequently, 3 type D room reservations are canceled by the Maria customer:

Server

```
cancelFunction called.
Notifying listerners for D type rooms
cancellation
```

Client (Maria)

```
[user@localhost ergasia_2]$ java HRClient cancel localhost D 3 Maria
Maria:
        - Type A rooms booked: 0
        - Type B rooms booked: 0
        - Type C rooms booked: 0
        - Type D rooms booked: 2
        - Type E rooms booked: 0
       Total cost: 240 euros
[user@localhost ergasia_2]$
```

Client (Petros)

Nothing printed.

Client (Voula)

```
Listener registered
Due to a cancellation there are 3 D type rooms available now.
```

Client (Mina)

```
Listener registered
Due to a cancellation there are 3 D type rooms available now.
```

Apparently, everyone was informed about the cancellation, except Petros, who is waiting for notifications for other types of rooms.

After that, 2 type E rooms are canceled by the Nikos customer:

Server

```
cancelFunction called.
Notifying listerners for E type rooms
cancellation
```

Client (Nikos)

```
[user@localhost ergasia_2]$ java HRClient cancel localhost E 2 Nikos
Nikos:
        - Type A rooms booked: 0
        - Type B rooms booked: 0
        - Type C rooms booked: 0
        - Type D rooms booked: 5
        - Type E rooms booked: 0
       Total cost: 600 euros
[user@localhost ergasia_2]$
```

Client (Petros)

```
Due to a cancellation there are 5 E type
rooms available now.
```

Client (Voula)

Nothing printed.

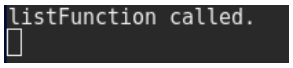
Client (Mina)

Nothing printed.

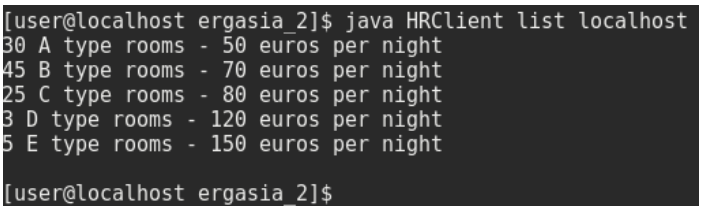
As expected, a cancellation message was printed only to the customer who has expressed interest in the specific room type.

Finally, the list of rooms is printed again, so that the change after the cancellations can be seen:

Server



Client



It is observed that the type D rooms from 0 became 3 and the type E rooms from 3 became 5 after the respective cancellations.

For clients waiting for notifications, they will continue to receive them on each cancellation until their program ends with ctrl + C. In other words, it is not provided that the function that was implemented to delete a customer from the update list is ever called. Of course, if the client terminates its program, then the next time the Server attempts to notify it of a cancellation, it will fail and thus remove it from the list.