Remote communication: Java RMI

Building Distributed Applications using Java Remote Method Invocation (RMI)

---

1 Introduction

Remote method invocation (RMI) allows client applications to invoke methods on remote (server) objects. RMI is used to build distributed applications in Java, with some code running locally while other (e.g. more resource-intensive) code is running on another machine. Remote method invocation is an object-oriented variant on remote procedure calls (RPC), of which Java RMI is one example.

Goal: Java RMI [1] is a simple middleware platform for Java applications, only providing transparent distribution. The goal of this session is to familiarize yourself with building and running a basic Java RMI application, in order to get a better understanding of lower-level concepts such as marshalling and remote invocations that form the base for any distributed application. You should be able to apply these concepts while modifying and extending the application according to a given set of functional requirements.

Opening the start code as IntelliJ project. The code from which you start can be found on Toledo, under 'Assignments'. Make all your changes based on this application code: extend or modify this code when necessary, even if not explicitly described in the assignment. All tools that you need should already be installed in your Ubuntu Desktop VM, assuming you did your homework. We will use IntelliJ as well as some command line tools such as *java*, *javac* and *ant*.

Unzip the file you have downloaded from Toledo, and open the top-level folder to create an IntelliJ project (via `File >`). We require you to use Java SE Development Kit (JDK) version **17**. If necessary, tell IntelliJ to use the Java 17 SDK on your machine, rather than another version (e.g. 1.8 or 16). In the IDE, go to: `File -> Project Structure -> Project Settings -> Project`. Set the Project SDK to 17 and Project language level to the SDK default.

2 The Hotel Booking Application

In this exercise session, we will develop a simplified version of a **hotel booking system** using the core concepts of Java RMI. We have given you (the skeleton of) an application that functions *locally*. It represents a simple hotel booking system that keeps track of the rooms in a hotel and manages room bookings. For example, a room can be booked for a guest on individual nights, with the hotel booking system maintaining the state of these bookings.
The current application consists of three classes: **`BookingManager`**, **`Room`**, and **`BookingDetail`**.

1. **`BookingManager`** stores the list of rooms and has a non-parameterized/default constructor that calls the `initializeRooms()` method when the application starts up. Rooms are identified by a unique number taken from an arbitrary, potentially non-sequential set of numbers (101, 102, 201, 203). We already provide a method that gets the information of all the rooms using `getAllRooms()`.
2. Each individual **`Room`** has a unique room number and contains a list of its booking details. This **`BookingDetail`** encapsulates information about guests who have booked rooms on specific dates.

The **Room** class offers methods to query these booking details: `getRoomNumber()`, `getBookings()`, and `setBookings(List<BookingDetail> bookings)`.

3 Testing your development environment

To refresh your basic Java knowledge, we will kick-off with compiling and running the basic local code from the command line.

To compile your code from the commandline, we will have to specify where the source code is, and where we want the compiled class files. The standard directory structure is to put source code in src and compiled code in out. From your project directory, you can compile the code on the command-line as follows:

```
javac -d out -sourcepath src src/staff/BookingClient.java
```

To run your compiled code, you can just use the java commandline and specify the out folder as classpath with the compiled code. The last argument is the full name of the class with the main method, including it's package name.

```
java -cp out staff.BookingClient
```

4 Assignment

The goal of this assignment is to convert this local hotel booking application into a *distributed* application using Java RMI, allowing hotel staff (the main users of this application) to book the hotel rooms remotely. More specifically, you have to provide the following basic functionality in your implementation:

1. A booking system (which uses the server application) offers a remote `BookingManager` object, which hotel staff can use to make bookings. Allow the hotel staff client application to locate this remote instance of the booking manager object in a naming registry.
2. Allow the hotel staff (who use the client application) to check if a specific room is available for a given date, using `isRoomAvailable(Integer roomNumber, LocalDate date)`.
3. The hotel staff should be able to create a new booking by collecting the necessary information (guest, room no, date) into a `BookingDetail` object. It supplies these details to the booking manager, which will try to add a new booking for the given guest in the given room on the given date through `addBooking(BookingDetail bookingDetail)`. If the room is not available, throw a suitable Exception.
4. Good news! The hotel staff love your system. However, they find it rather tedious to check the availability of each room separately. Therefore, implement a method to request the availability of all rooms of the hotel on a given date, using `getAvailableRooms(LocalDate date)`.

5 Getting up and running

For the **client** application, use the given `BookingClient` class and implement the inherited methods. This application will execute a given test scenario, which covers all the required functionality as described in the assignment. Do *not* change the abstract class (`AbstractScriptedSimpleTest`) in the `staff` package. Otherwise, you may adapt and extend all classes. For the **server** application, you are free to structure the code according to your design, as long as you have one `main` method that launches all server code.

Code organization. In a realistic environment, the client and server components of the distributed hotel booking application would be separate code bases, developed in isolation (e.g. two projects). In our lab setting, however, for simplicity, we recommend to use a single code-base (~ single project) for both components.

Checking the output. Executing your `BookingClient` outputs the results of several test cases for booking new rooms and retrieving available rooms. Check this output to see if it yields the expected result.

6 Executing your code

Apache Ant. Apache Ant[1] is a tool for automating the build and execution process of your application, in order to support efficient development of a distributed application. The main artifact is an Ant build script that contains several tasks for compiling your source code (both client and server), initializing the RMI registry, firing up the server component before the client component using the correct classpath, and finally terminating all applications – all in one go. This build script accepts several parameters that allow you to customize the build process for the structure of your application.

While developing your application, you can test whether it is functioning correctly by executing your code. Next to the compilation, Ant will take care of

1. Starting up the RMI Registry with the default port in the right directory with the compiled code.
2. Starting the server.
3. Starting the client.
4. Stop all processes afterwards.

Prepare the Ant script:

- Put the package and class names for your server's and client's `main`-method classes into the `build.xml` file.

Run your code: Within the main directory of your project (i.e. where `build.xml` is located), you can use the following commands:

- `ant run` → will compile your code, start the `rmiregistry` and run your application.
  This is the default (and most common) option.
- `ant run-wo-compile` → will run `rmiregistry` and your application.
  Use this option if you handle compilation yourself (e.g. or if you just want to rerun the three processes without recompilation).
- `ant run-wo-registry` → will compile your code and run your application.
  Use this option if you handle the setup of the `rmiregistry` yourself.
- `ant run-wo-compile-registry` → will run your application.
  Use this option if you handle compilation and the setup of the `rmiregistry` yourself.

IntelliJ has built-in support for running Ant targets. If the Ant build file was not automatically recognized, you can right-click `build.xml` and select `Add As Ant Build File`. Afterwards, you can right-click a specific target through the `Ant` panel and select `Run Target` to run it. You will find the output in the `Messages` tab (expand the tree, or click the `Toggle Tree/Text Mode` button). If the `Messages` tab is not visible, you can open it via `View > Tool Windows > Messages`.

Running your code manually and locally. Instead of using Ant, you can also run your compiled code manually, leveraging the plain Java command on the commandline. Make sure you get all ports and execution directories right.

To simplify the deployment and loading of your compiled code, execute all processes (including `rmiregistry`) in the same directory as your compiled code (`.class` files). If required, add additional folders to your classpath to make class files accessible:

```
java -cp .:<additional folder> <class>
```

There is no real remote code deployment or remote communication between machines in our current setting. There are just 3 processes (rmiregistry, the client and the server) that communicate over the localhost network on your local machine, all running from one single directory.

Shipping your code to the remote world. In one of the later practical sessions we will run the RMI server code on a remote server in the cloud, and the client on your local machine. For that purpose, we will need to package and deploy the client and server code separately. At this time you can already start

[1] `https://ant.apache.org/`

thinking about which code belongs on the client, and which code on the server. Using the jar tool of the JDK[2], you can package the client code in a client.jar and the server code in a server.jar.

You can already try on your local machine to run client.jar and server.jar from separate directories outside your compilation directory.

7 Assignment extensions

When you are ready with the basic functionality above, you can start working on the following interesting extensions. You might need to adapt and/or extend the test client classes for these extensions.

1. The hotel booking system may be used by a number of hotel staff members at once so your implementation must be thread-safe. In addition, efficiency is important for the hotel booking system, as such two bookings can be made concurrently as long as they are for different rooms. Test a scenario with many clients requesting the same room or different rooms at the same time. In order to trigger concurrency problems on the server, you need to slow down time, e.g. using `Thread.sleep()`.
2. Extend the BookingManager to request an IBookingSession object, which is a remote object located on the server, that will keep track of all state of the client session on the server. A staff member will use this session to add different BookingDetails into a shopping cart, and then book them all at once as one transaction. Design the interface of IBookingSession such that it has at least two operations: addBookingDetail() and bookAll(). When calling bookAll(), if one booking in the shopping cart is no longer possible, all bookings in the shopping cart should be canceled. Hence, either all bookings in the shopping cart succeed, or all are ignored when one booking is no longer possible.

Note that booking details will always be transfered by value over the remote connection (i.e. as a copy). However, the remote session object is an object that has pass-by-reference semantics when transfered over a remote connection. The remote session object should not be registered in the rmi registry. This remote session object must be returned by reference by the booking manager. The booking manager is responsible for creating the session object, generating the stub, and returning it to the requesting client.

References

[1] Oracle. *Java RMI Specification.* `https://docs.oracle.com/en/java/javase/17/docs/specs/rmi/index.html`

[2] Oracle. *An Overview of RMI Applications.* `https://docs.oracle.com/javase/tutorial/rmi/index.html`

[3] *Remote Method Invocation (RMI) Tools and Commands.* `https://docs.oracle.com/en/java/javase/11/tools/remote-method-invocation-rmi-commands.html`

[4] Oracle. *Frequently Asked Questions: Java RMI and Object Serialization.* `https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/faq.html`

[5] Apache Ant. *Java-based build tool.* `https://ant.apache.org/manual/index.html`

[2] `https://docs.oracle.com/javase/tutorial/deployment/jar/index.html`