Distributed Systems: Java RMI session 1

Stefan Walraven, Wouter De Borger, Fatih Gey, Majid Makki and Wouter Joosen

October 2, 2017

Overview

There will be 3 exercise sessions on Java RMI:

1. 10/10/17 in pc lab:
   - Introduction to Java RMI
   - **Submit** the final version of the code on Toledo **before Friday, October 13, 19:00**.
2. 24/10/17 in pc lab:
   - Extensive Java RMI application.
3. 31/10/17 in pc lab:
   - Extensive Java RMI application (cont.).
   - **Submit** the final version of *(1) the design report* and *(2) the code* on Toledo **before Friday, November 3, 19:00**.

In total, **each** student must submit 2 assignments to Toledo.

1 Introduction

Goal: Java RMI is a simple middleware platform for Java applications, only providing transparent distribution. The goal of this session is familiarizing yourself with building and running a *basic* Java RMI application. The concepts mentioned in the lessons (e.g. distribution, marshalling) will be the subject of this session. You will have to be able to apply these concepts while modifying and extending the application according to a given set of functional requirements.

Approach: This session must be carried out in groups of *two* people. You will have to team up with the same person for each of the sessions in this course.

Submitting: On Friday at 19:00, **each** student must submit his or her results to the Toledo website. To do so, first ensure that the main-method classes of your client and server are correctly specified in `build.xml` file (see Section 4) and create a zip file of your source code using the following command from your source code directory:

```
ant zip
```

Then submit the zip file to the Toledo website (under Assignments) before the deadline stated in the overview above.

Important:

- When leaving, make sure `rmiregistry` is no longer running. You can stop any remaining `rmiregistry` processes using the following command: `killall rmiregistry`
- Retain a copy of your work for yourself, so you can review it before the exam.

2 The Car Rental Application

All exercise sessions will use a car rental application. It consists of six classes: `CarRentalCompany`, `Car`, `CarType`, `ReservationConstraints`, `Quote`, `Reservation`, and `ReservationException`, as shown in Figure 1.

1. `CarRentalCompany` has a constructor that accepts the name of the rental company, a list of regions it has offices in, and a number of cars as its arguments. Each car rental company stores a list of cars and a map of car types. `CarRentalCompany` has a method to make inquiries about the availability of car types during a certain period (using `getAvailableCarTypes(Date, Date)`). It also provides methods to reserve cars: `createQuote(ReservationConstraints, String)` creates a quote (i.e. a tentative reservation), which can be confirmed by using the `confirmQuote(Quote)` operation. It is possible to cancel a reservation using `cancelReservation(Reservation)`.
2. The information about a `Car` is captured in a `CarType` object (e.g. the number of seats and the rental price per day). Each `Car` also contains a list of reservations. The class offers methods to query and manage these reservations: `addReservation(Reservation)`, `isAvailable(Date, Date)` and `removeReservation(Reservation)`.
3. `Quote` contains the details of a tentative reservation of a car type: the name of the car renter that makes the reservation, a start and end date (rental period), the total price of the reservation and the car type. A `Reservation` specifies the details of a (final) reservation of a specific car (by means of the car ID) in addition to the details described in the quote.

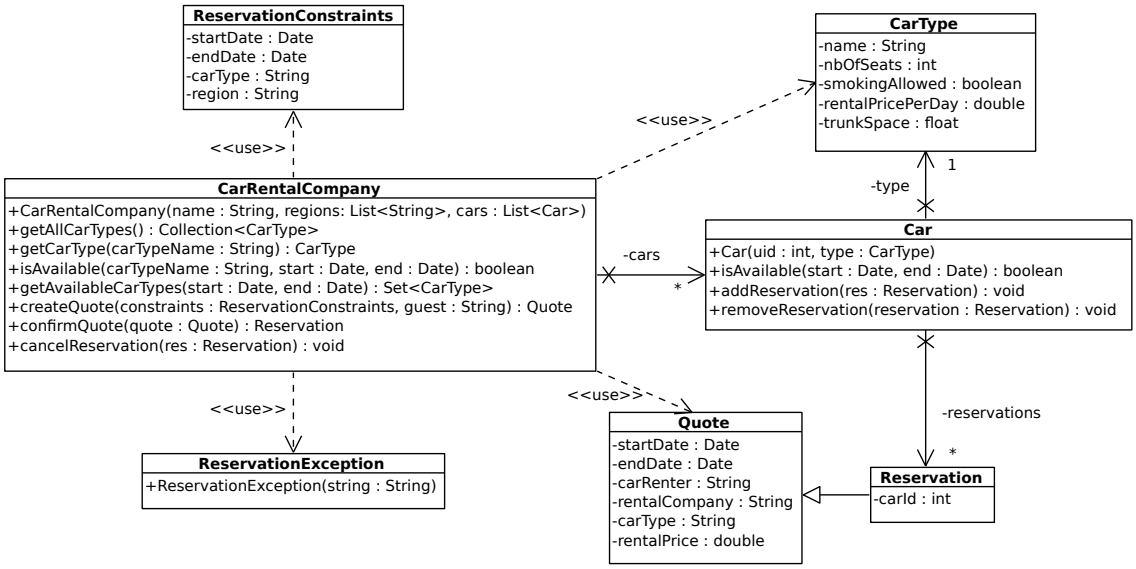

Figure 1: Design of the local car rental application.

3 Assignment

Convert the local application into a distributed application. The course material on Java RMI on Toledo and the Java RMI demo should serve as a good manual for this purpose. First sketch a design of the solution that clearly shows which classes are remote and which are local (deployment diagram). Also indicate of which classes objects will get sent over the wire. You don't have to submit this sketch.

Basically, you will have to build the following functionality:

1. A server offers a remote `CarRentalCompany` object, which clients can use to make reservations. To allow clients to locate this remote instance of the rental company, it is registered

in the RMI registry.

2. The client application can check which car types are available in a certain period. Let the client print this list of available car types. Then the client application collects the constraints necessary to create a quote (car type, region of interest, start and end date). Next, it contacts the car rental company and tries to create a quote by supplying a `ReservationConstraints` object. Print the details of this tentative reservation. Afterwards, the client tries to confirm the quote. The result of this process is that the reservation is registered on the server.

Afterwards, extend your solution with the following functionality:

1. Add an option to request all reservations made by a specific car renter. For each reservation, the client prints the reserved car type and car ID, the reservation period and the price.
2. Add an option to let the manager of the car rental company retrieve the number of reservations for a specific car type. Please note that the same client application will be used by the manager. No authentication is needed to distinguish the manager from car renters.

4 Practical information

The code of the local application used in this session can be found on Toledo, under assignments. Make all your changes based on this application. Extend or modify this code when necessary, even if not explicitly described in the assignment. Use your preferred IDE or the command line.

Use the given `Client` class and implement the inherited methods. The client application will execute a given test scenario (i.e. `simpleTrips` script file), which covers all the required functionality as described in the assignment. *The client application might have redundant parameters*, depending on your implementation of the server application. Do *not* change the abstract classes (`Abstract...`) in the `client` package, the provided script and .csv files.

Security Manager. The security policy of the `SecurityManager` does not allow `connect` and `accept` operations on the sockets of the localhost. There are two ways to solve this:

1. Avoid this problem by setting the `SecurityManager` to `null`. This is the recommended strategy for this session.
2. Use the `SecurityManager`, but you will have to write a custom security policy. A security manager is only required when you want to execute downloaded classes (cf. Bonus, Section 5). You can try this if you have spare time left. The course material on Java RMI (see slides) and the Java RMI Tutorial [Tutorial] can help you. You'll have to run your Java RMI application manually and use the command line argument `-Djava.security.policy=`*`<your policy file>`*.

Code Organization, running your code using Apache Ant. In a industrial environment, the client and server components of the distributed car rental agency would be separate code bases, developed in isolation (e.g. two Eclipse projects). In our lab setting, however, we recommend to use a single code-base ($\sim$ single Eclipse project) for both components. Moreover, we provide additional automation to support an efficient development of a distributed application: an Ant script that compiles your source code (of both components, initializes (a fresh instance of) the rmiregistry and fires up the server component before the client component, and finally terminates all applications – all in one go.

Prepare the automation:

- Put the fully-qualified class names of your server's and client's `main`-method classes into the `build.xml`

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

- `ant run` $\rightarrow$ will compile and run `rmiregistry` + your application
- `ant run-wo-compile` $\rightarrow$ will run `rmiregistry` + your application
  Use this option when you are developing with Eclipse, as external compilation may cause strange errors in Eclipse.

- `ant zip` → will zip your solution for submission
  In case of problems, alternatively use `zip -r firstname.lastname.zip <your source directory>` to attain a zip for submission.

Running your code manually. Instead of using Ant, you can also run manually:

1. Execute all processes (including `rmiregistry`) in the same directory as your compiled code (.class files). In case you want to make certain code available for download, you have to specify the path to the code by means of the following argument for the Java process `-Djava.rmi.server.codebase=<URL>` [Properties, Dyn]. When the codebase is not a jar file, this argument becomes `-Djava.rmi.server.codebase=file:<full path to jar file>`. You also have to set `java.rmi.server.useCodebaseOnly` to false for the Java process that needs to download the code [Properties].
2. It is best to work with Java SE 6 or higher. You can test this by checking the output of the command `java -version`. If you see an older version, you can fix this by adding the bin directory of JDK6 to your PATH. To do this, run the following command in every terminal you open: `export PATH=/usr/lib/jvm/java-6-openjdk/bin:$PATH` or by adding this line to your `.bashrc` file in your home directory.

Checking the output. After running the `simpleTrips` script file and before submitting the assignment, check the output to see if it is the expected result.

Important note. The results of the `java.util.Map` methods `keySet()`, `entrySet()` and `values()` are only views onto the collection and are not serializable independently. *These view collections are intentionally not serializable.*

5 Bonus: distributed setup

The following instructions are for the "die hards" who want to try to run a distributed setup of their Java RMI application. *This part of the assignment is not mandatory and therefore should be submitted separately on Toledo.* First submit the first part, and then continue with this part of the assignment. When you do this part, youll have to build and run your code *manually.*

Separate client and server code. During lab sessions the client and server code are located in the same project folder. This way the client code can access all server code. In an actual client-server setup, server code will be located on a different host and will not be accessible. So create two separate project folders, and place client and server code in the respective project folder.

The client code will not compile any more since it depends on the remote interface of the server application. Therefore you should make the remote interface and all the custom classes it uses available to the client. Create a jar file containing the necessary *class* files:

```
jar cf <jar file> <class files>
```

Add this jar to the class path of the client project. Now you can compile the client code with the following command (or your IDE can do it automatically):

```
javac -cp .:<jar file> client/*.java
```

Locate rmiregistry. The client code needs to locate the RMI registry. Originally the RMI registry executed on the same host as the client, but now it will execute at the server side. Therefore you should update the code in the client that locates the RMI registry: `LocateRegistry.getRegistry(<host>);` or `Naming.lookup(//<host>/<name>);`.

Run server application. There is no need for dynamic downloading of code to run the car rental application in a distributed context, so we can keep the security manager set to `null`. First start the RMI registry. To start the server application we do need to specify the host name of the server (do you know why?):

```
java -Djava.rmi.server.hostname=<server host> rental.RentalServer
```

If you want to add dynamic downloading of code, you need to use a security manager and specify policy files, as described in Section 4. The Java RMI Tutorial [Tutorial] also presents an example that uses dynamic downloading of code. In this example the code is downloaded from the client to the server.

Run client application. Login to another computer in the pc lab (we suggest to login remotely via ssh). The client application doesn't need any extra arguments. However, it is still required to add the jar file to the class path, similar to the compilation step:

```
java -cp .:<jar file> client.Client
```

References

[Dyn] Oracle. *Dynamic code downloading using Java^{TM} RMI.* `http://download.oracle.com/javase/7/docs/technotes/guides/rmi/codebase.html`

[Properties] Oracle. *`java.rmi` Properties.* `http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/javarmiproperties.html`

[Tutorial] Oracle. *An Overview of RMI Applications.* `http://download.oracle.com/javase/tutorial/rmi/index.html`

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

Good luck!