Distributed Systems: Java RMI session 2 & 3

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

November 01 & 08, 2016

Overview

There will be 3 exercise sessions on Java RMI:

1. 18/10/16 in pc lab:
   - Introduction to Java RMI
   - **Submit** the final version of the code on Toledo **before Friday, October 21, 19:00**.
2. 01/11/16 virtual lab:
   - Extensive Java RMI application.
3. 08/11/16 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 11, 19:00**.

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

1 Introduction

Goal: In the following two sessions, we will extend the car rental application into a distributed rental agency system that allows to plan trips (i.e. make reservations at several rental companies). Unlike the first session, the system should now also allow multiple users to book *concurrently* in a safe way. Session 2 will focus on designing the distributed rental agency system, whereas Session 3 will serve for the implementation of this system.

Approach: These sessions must be carried out in groups of *two* people. You will have to work with the same person as the previous sessions.

Submitting: On Friday, at the end of the week of the third Java RMI session, **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` (see Section 5) and create a single zip file of *1) your design report (PDF)* and *2) your source code* using the following command:

```
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 Session Overview

In the first Java RMI session, we built a car rental application that could be used to make reservations at one car rental company. Similar to the first Java EE session, we will now extend this application to support reservations at multiple rental companies. Remember that we are working with plain Java RMI, which means that you cannot rely on the functionality of the Java EE container. You will have to implement the (useful) functionality offered by this container yourself. Concretely, you will have to provide a number of new artifacts:

Sessions and Life Cycle Management: Like the Java EE lab, clients will use the system through *sessions*, which can keep all the conversational state. This also includes *life cycle management of sessions*: creating, storing and cleaning (i.e. removing) sessions.

Concurrency Control: What happens when a `createQuote`-invocation in one session occurs at the same time as a `confirmQuote`-invocation in another session? Or when multiple `confirmQuote`-invocations of different sessions happen at the same time? As you may have noticed in the previous session, things can go wrong when multiple users try to confirm quotes at the same time. The system should be able to handle multiple concurrent users. The Java language provides the `synchronized` keyword[1] to support concurrency control.

Naming Service: The different car rental companies execute independently of the car rental agency. There will be a need for a *naming service* that allows to register and locate the car rental companies, regardless of the system on which they execute. Imagine this like a phone book with contact details for each rental company.

Because converting a simple car rental application into a fully distributed system is non-trivial, we will split up this task into two parts. The first part, i.e. the design of this system, will be the subject of the second Java RMI session. The implementation of this system will be the subject of the last session. However, when your design is already finished, you can immediately start with the implementation.

3 Car Rental Agency Application

Recall from the previous Java RMI session that we had a very simple reservation system: one car rental company with a remote interface for manipulating its reservations. We will now extend this system step-by-step into a distributed rental agency system (cf. http://www.carrentals.com).

3.1 Sessions

The distributed rental agency will have to provide two different services: one for car renters and one for managers of the rental agency. Therefore we need respectively `ReservationSession`s and `ManagerSession`s (cf. EJB session beans). Note that it is not required to do any form of access control, but you have to foresee that a car renter and a manager are different persons (in other words, a car renter shouldn't execute manager operations and vice versa).

Each car renter has its own (unique) session to keep all the conversational state. A `ReservationSession` provides *at least* the following operations. As in the first Java EE session, quotes are first created and stored in the session through the method `createQuote`. The

[1] More info can be found on http://download.oracle.com/javase/tutorial/essential/concurrency/.

method `getCurrentQuotes` gives an overview of all pending quotes ("the bill") and the method `confirmQuotes` effectively makes reservations for specific cars. You may assume that the bill will be paid offline. The latter method may only succeed if *all* quotes are confirmed. However, it is not the purpose of this exercise session to implement distributed transactions (no 2-phase commit implementation!), simple rollback is sufficient. In addition, car renters can check the availability of car types in a certain period via `getAvailableCarTypes`, and can search for the cheapest car type in that period for a specified region using `getCheapestCarType`. Add extra operations when necessary. Below you can find example scenarios for this functionality (you do not have to implement these scenarios).

The manager session enables the rental agency manager to register and unregister car rental companies into the system (see Subsection 3.2), as well as to request the full list of registered car rental companies and information about their cars (i.e. car types). In addition, the rental agency manager should be able to retrieve different statistics to support customer profiling and advertising: the number of reservations for a particular car type in a car rental company, the best customer (highest number of (final) reservations), and the most popular car type of a car rental company for a given calendar year. For that consider the starting date of renting period. There is no need to keep any conversational state, however the rental agency manager (i.e. client) may not get access to actual reservations because of privacy reasons.

Example scenarios

1. The car renter starts a new session.
2. The car renter wants to reserve cars at two car rental companies: A and B:
   (a) The car renter checks the availability of a car type at car rental company A and notices that there is a car available.
   (b) The car renter creates a quote at car rental company A.
   (c) The car renter checks the availability of a car type at car rental company B and notices that there is a car available.
   (d) The car renter creates a quote at car rental company B.

3A. *Alternative A: reservation succeeds*
The car renter wants to confirm these quotes and closes the session:
   (a) The session confirms the quote at car rental company A.
   (b) The session confirms the quote at car rental company B.

4A. The reservation succeeded.

3B. *Alternative B: reservation fails*
The car renter wants to finalize these bookings and closes the session:
   (a) The session confirms the quote at car rental company A.
   (b) The session confirms the quote at car rental company B, but an error occurs: no car of the required car type is available any more.
   (c) The session cancels the reservation at car rental company A.

4B. The reservation failed, the car renter is notified of this failure.

3.2 Naming Service

In going from a system that only has one car rental company towards a distributed system that supports multiple car rental companies (and each of these might run on a different server), we will need to provide a central naming service. This is a lookup service to locate *car rental companies* based on their names. In addition, car rental agency managers register and unregister different car rental companies (e.g. `hertz` and `dockx`) at this naming service via their `ManagerSession`.

4 Assignment

4.1 Java RMI Session 2

Make a detailed design of the distributed car rental agency system described above, and submit it as a design report. **The design report consists of diagrams and maximally 3 pages of design decisions, written in English and combined into 1 PDF. Be concise and to the point!** Don't forget to put your names on the front page of your report, and store the report in the main directory of your solution (next to `build.xml`), such that the final zip-creation process includes these reports.

First, describe in 1 or 2 paragraphs the overview of your design. Which are the core parts/components and their responsibilities? *This is not a sequential story!* Next, at least the following design decisions[2] should be discussed:

- Which classes are remotely accessible and why? Which classes are serializable and why?
- Which remote objects are located at the same host (or not) and why?
- Which remote objects are registered via the built-in RMI registry (or not) and why?
- Briefly explain the approach you applied to achieve life cycle management of sessions.
- At which places is synchronization necessary to achieve thread-safety? Will those places become a bottleneck by applying synchronization?

Further, you will have to make the following design artifacts: a *full class diagram* (thus showing both classes and methods) and a *deployment diagram* which shows which classes are executed on which system[3]. Indicate on the class diagram which classes are remotely accessible (by a `<<Remote>>` stereotype) and which classes will be exchanged between the different systems (by a `<<Serializable>>` stereotype). Also create *sequence diagrams of the chosen operations* to illustrate the entire booking process, including example scenario B with the reservation failure (see Subsection 3.1).

Note. You will have to implement this design in the next session. Keep this in mind while creating the design diagrams. Don't lose too much time on the simple aspects, but put sufficient detail into designing the harder parts. Do not lose track of the big picture and make sure it remains implementable!

4.2 Java RMI Session 3

Implement the design from the previous session. You may start with the code from the first Java RMI session, but this is not a strict requirement.

Test your implementation by writing a client which extends the given `AbstractTestManagement` class and uses the `trips` script file (see Toledo). Implement the inherited methods, and create a `main` method that creates a new client and executes the `run` method. This will start a test scenario, which gives you an *indication* of the correct working of your application.

Provide a single `main` method to boot your distributed car rental agency (i.e. this does not include your client). Since you are executing it on a single machine, this simplifies the testing process, and documents how to start your distributed car rental agency. This method is your "server's main method" with respect to the Apache Ant script (see Section 5).

Software development is an iterative process, so it is possible that the design still changes during implementation. If there are any differences, be sure to document these in the design report and describe why these were necessary. Make sure that the final design artifacts (i.e. the diagrams) are in sync with the final implementation.

[2] With design decisions we mean design choices, possible alternatives and trade-offs.

[3] This diagram has to be documented as if it is a real distributed deployment, not the lab deployment where everything runs on the same machine.

Note. There is no need to make an interactive or multithreaded test application. Extra scenarios may be tested in addition to the provided scenario to show the correct working of your system, but do not adapt the given class. What is important is that you show correct usage of Java RMI and an understanding of distributed computing.

5 Practical Information

For the final design artifacts you can use Visual Paradigm:

`/localhost/packages/visual_paradigm/Visual_Paradigm_12.1/bin/Visual_Paradigm`

To activate the licence of Visual Paradigm, follow these steps after its startup:

- Click on `Subscription / Academic License` → `Academic Partner Program License`.
- Enter your name, your KU Leuven e-mail address and fill in the following activation code: `U99RD-J7P9D-T25F7-J25NP-D3J3A`.
- You are asked to enter a verification code which you will receive in an e-mail within the next minutes.

Sequence diagrams can also be drawn using http://websequencediagrams.com/.

Running your code using Apache Ant. As with the first RMI lab assignment, you may use Apache Ant to run your code. **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` → will compile and run `rmiregistry` + your application
- `ant run-wo-compile` → 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.

Good luck!