Distributed Systems (H0N08A)

Java RMI: assignment 2
Setting up distributed applications with remotely invoked objects

---

Practical arrangements

There will be 3 exercise sessions on Java RMI:

1. Tuesday 15 October 2019:
   - Assignment 1: introduction to Java RMI.
   - Submission 1: Submit the final version of the code on Toledo before Friday 18 October 2019, 19:00.
2. Tuesday 22 October 2019:
   - Assignment 2 *(this assignment)*: extensive Java RMI application.
   - No submission.
3. Tuesday 29 October 2019:
   - Assignment 2 *(this assignment)*: extensive Java RMI application (continued).
   - Submission 2: **Submit** the final version of *(1) the report* and *(2) the code* on Toledo **before Friday 1 November 2019, 19:00**.

In total, **each** student must submit their solutions to the 2 RMI assignments to Toledo.

Submission: Before Friday 1 November 2019 at 19:00, **each** student must submit their 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 the first assignment) and create a zip file of *1) your report (PDF)* and *2) 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. Make sure that the name of the zip file is in the `rmi2.firstname.lastname.zip` format.

Important:

- These sessions must be carried out in groups of *two* people. You will have to work with the same person as the previous session.
- Retain a copy of your work for yourself, so you can review it before the exam.
- When leaving the computer labs, make sure `rmiregistry` is no longer running. You can stop any remaining `rmiregistry` processes using the following command: `killall rmiregistry`

Good luck!

1 Introduction

Goal: 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.

2 The Car Rental Application

In the first Java RMI session, we built a car rental application that could be used to make reservations at one car rental company. We will now extend this application into a distributed rental agency system (cf. `https://www.carrentals.com`), to support reservations at multiple rental companies. Concretely, you will have to provide three new artifacts.

2.1 Rental agency and Naming service

In this assignment, a car rental company sets up its own infrastructure for processing bookings (i.e. it stores its own data). A central *rental agency* enters into contracts with these companies for reselling their cars on the agency platform. Clients therefore only need to interact with the central agency, which manages all aspects of interacting with the different available companies.

To achieve this centralization, we will need a central *naming service*. This is a lookup service that allows to locate and register individual car rental companies (based on their names), regardless of the system on which they execute (each of these might run on a different server, independently of the agency). Imagine this like a phone book with contact details for each rental company. For this assignment, the agency supports two companies: `hertz` and `dockx`.

2.2 Sessions

In order to manage the conversational state in creating quotes and reservations, each **car renter** obtains their own (unique) `ReservationSession` that the distributed rental agency will provide. A `ReservationSession` provides *at least* the following operations:

1. Quotes are first created and stored in the session through the method `createQuote`. If multiple companies can provide a quote with the given constraints, any one (but only one) of them can be offered to the customer.
2. The method `getCurrentQuotes` gives an overview of all pending quotes ("the bill").
3. The method `confirmQuotes` effectively makes reservations for specific cars. You may assume that the bill will be paid offline. The 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.
4. In addition, car renters can use their session to 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`.

You will also need to add a **manager** session, which enables the manager of the central rental agency to control the agency across all the car rental companies.

First, the manager uses this session to register and unregister car rental companies with which the agency has a contract into the system (however, these companies may be launched separately), as well as to request the full list of registered car rental companies and information about their cars (i.e. car types).

Second, the rental agency manager should be able to retrieve different statistics to support customer profiling and advertising:

1. the number of reservations for a particular car type in a car rental company,
2. the best customers (*all* renters who have the highest number of final reservations),
3. the number of reservations made by a particular car renter,
4. the most popular car type of a car rental company for a given calendar year (for this, consider the starting date of the renting period).

There is no need to keep any conversational state for this manager session. The rental agency manager (i.e. client) may not get access to actual reservations because of privacy reasons. For now, you do not need to consider how the manager authenticates themselves.

Supporting both car rental and manager sessions also requires appropriate *life cycle management*: creating, storing and cleaning (i.e. removing) sessions if necessary. For cleaning sessions, you may assume that clients will explicitly close their session; you do not need to support implicit closing of the session when the client exits. In general, add extra operations to support sessions where necessary.

2.3 Concurrency Control

What happens when a `createQuote`-invocation in one session occurs at the same time as a `confirmQuote`-invocation in another renter's session? Or when multiple `confirmQuote`-invocations of different sessions happen at the same time? Things can go wrong when multiple users try to confirm quotes simultaneously.

One solution to prevent race conditions when multiple clients are running concurrently is to use *synchronization*: controlling (restricting) access to critical methods and/or objects. The Java language provides the `synchronized` keyword to support concurrency control through synchronization: see `https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html`. Ensure that your application is able to properly handle multiple concurrent users without entering an invalid state.

3 Assignment

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 design of this system and its implementation. Allow sufficient time to properly develop your design. Of course, once your design is finished, you can start with the implementation. Conversely, software development is an iterative process, so it is possible that the design still changes during implementation. Be sure to update your report such that the final version is in sync with the final implementation.

3.1 Design

Start by designing a solution for the three expansions of the application, i.e. the naming service, sessions and concurrency control. Think of where the different remote objects would be deployed in a distributed setup, which objects need to be serializable/remotely accessible and how to fulfill the requirements set out in the three expansions.

3.2 Implementation

Implement the three expansions of the application based on the design that you have created. You may start with the code that you wrote for the first Java RMI session, but this is not a strict requirement.

Write a **client** that extends the given `AbstractTestManagement` class. Implement the inherited methods: reading the JavaDoc available in the `AbstractTestManagement` and `AbstractTestBooking` classes helps to understand the operations of the `trips` test script. Create a `main` method that creates a new client with the provided `trips` script file and then executes the `run` method. This will start a test scenario, which gives you an *indication* of the correct working of your application.

To bootstrap your distributed car rental agency and the two car rental companies (i.e. this does not include your client), create a **server** with a single `main` method. Since you are executing it on a single machine, this simplifies the testing process and documents how to start your distributed car rental agency.

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 provided classes. What is important is that you show correct usage of Java RMI and an understanding of distributed computing.

3.3 Reporting

We would like you to report on your design and the decisions you have made[1], and on your understanding and implementation of distributed concepts. Provide your answers in English, and keep them short, concise and to the point: a few lines is often enough. You are encouraged to use the LaTeX template on Toledo. Don't forget to put your names in the report, and store the report (as one PDF called `report-rmi.pdf`) in the main directory of your solution (next to `build.xml`), such that the final zip-creation process includes these reports.

Compile a report that answers the following questions:

1. How would a client complete one **full cycle of the booking process**, for both a successful and failed case? Base yourself on the example scenarios in Figure 1. Create sequence drawings to illustrate this.
2. When do classes need to be **serializable**? You may illustrate this with an example class.
3. When do classes need to be **remotely accessible** (`Remote`)? You may illustrate this with an example class.
4. What **data** has to be **transmitted** between client and server and back when requesting the number of reservations of a specific renter?
5. What is the reasoning behind your **distribution of remote objects over hosts**? Show which hosts execute which classes, if run in a real distributed deployment (not a lab deployment where everything runs on the same machine). Create a component/deployment diagram to illustrate this: highlight where the client and server are.
6. How have you implemented the **naming service**, and what role does the built-in RMI registry play? Why did you take this approach?
7. Which approach did you take to achieve **life cycle management** of sessions? Indicate why you picked this approach, in particular where you store the sessions.
8. Why is a Java RMI application **not thread-safe by default**? How does your application of **synchronization** achieve thread-safety?
9. How does your solution to **concurrency control** affect the **scalability** of your design? Could synchronization become a **bottleneck**?

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

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 then 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 then 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.

Figure 1: Example scenarios

4 Getting up and running

Running your code. Check the first RMI lab assignment on how to run your code; we have again provided a Ant script to automate this process.
The remote setup is still available to get a good understanding of and hands-on experience with a real distributed environment, and we recommend to try it if your implementation is finished.

Note: in order to conserve system resources, RMI is designed to support the reuse of sockets (i.e. ports). If you are working in a simple scenario with one Java Virtual Machine, you can therefore safely use the same port number (within your port range) for multiple exported objects when using the remote setup.

Important: if you try the remote setup, make sure to properly parse and use the `LOCAL` and `REMOTE` flags in the code that you submit: we will primarily verify that your `LOCAL` implementation is working.

For creating diagrams, you can use Visual Paradigm:

```
/localhost/packages/visual_paradigm/Visual_Paradigm
```

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

- Click on `Subscription / Academic License` → `Academic Partner Program License`. If the subscription pop-up does not appear, go to the `Window` menu and click on the `License Manager`.
- Enter your name, your KU Leuven e-mail address and the activation code found at `https://ap.visual-paradigm.com/kuleuven`.
- You are asked to enter a verification code which you will receive in an e-mail within the next minutes (this can take a while).

Sequence diagrams can also be drawn using `https://www.websequencediagrams.com/`.