Distributed Systems (H0N08A)

Java RMI: assignment 1
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 *(this assignment)*: 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: extensive Java RMI application.
   - No submission.
3. Tuesday 29 October 2019:
   - Assignment 2: 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 18 October 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 Section 4.1) 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. Make sure that the name of the zip file is in the `rmi1.firstname.lastname.zip` format.

Important:

- 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.
- 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

Remote method invocation (RMI) allows client applications to invoke methods on remote (server) objects. This provides a distributed approach to running applications, 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. The RPC model also forms the basis of Google's gRPC[1] or the Apache Thrift[2] framework developed by Facebook.

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.

2 The Car Rental Application

The recurring theme in all exercise sessions will be a **car rental platform** for booking and managing bookings of rental cars: multiple companies will be able to offer rentals on a shared service, while clients will be able to book cars remotely with this service. For now, we will focus on a simplified version of the car rental platform with only one company. We will then gradually expand this application towards a more advanced setup over the coming sessions.
The current application consists of six classes, as shown in Figure 1: `CarRentalCompany`, `Car`, `CarType`, `ReservationConstraints`, `Quote`, `Reservation`, and `ReservationException`.

1. `CarRentalCompany` has a constructor that accepts as its arguments the name of the rental company, a list of regions (where it has offices to rent out cars) and a list of `Cars`. Each car rental company stores the list of regions, the list of `Cars` and a map of their `CarTypes`. `CarRentalCompany` has a method to make inquiries about the availability of cars of a certain type during a certain period (for any region) 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 using the `confirmQuote(Quote)` method. It is possible to cancel a reservation using `cancelReservation(Reservation)`.
2. The `Cars` owned by a company are each of a certain `CarType`. This `CarType` encapsulates information that is common to all cars that have that same type: e.g. the number of seats and the rental price per day. Each individual `Car` then contains a list of its reservations. The class offers methods to query and manage these reservations: `isAvailable(Date, Date)`, `addReservation(Reservation)`, and `removeReservation(Reservation)`.
3. A `Quote` contains the details of a *tentative* reservation for a car of the given type: the rental company, the name of the car renter, a start and end date (rental period), the total price and the car type. A `Reservation` is an extension of a `Quote` that represents a specific *final* reservation: it assigns one specific car (by means of the car's ID) to fulfill the underlying quote (company, renter, period, price).

3 Assignment

We have given you (the skeleton of) an application that would function *locally*: the car renter and the company would have to run their applications on the same machine. Of course, this is not very convenient, neither for the renter as they cannot book remotely nor for the company as they would have to allow many different users to have access to their infrastructure. The goal of the assignment is therefore to convert this local application into a *distributed* application using Java RMI, allowing car renters to access the company's platform remotely.

[1] `https://grpc.io/`
[2] `https://thrift.apache.org/`

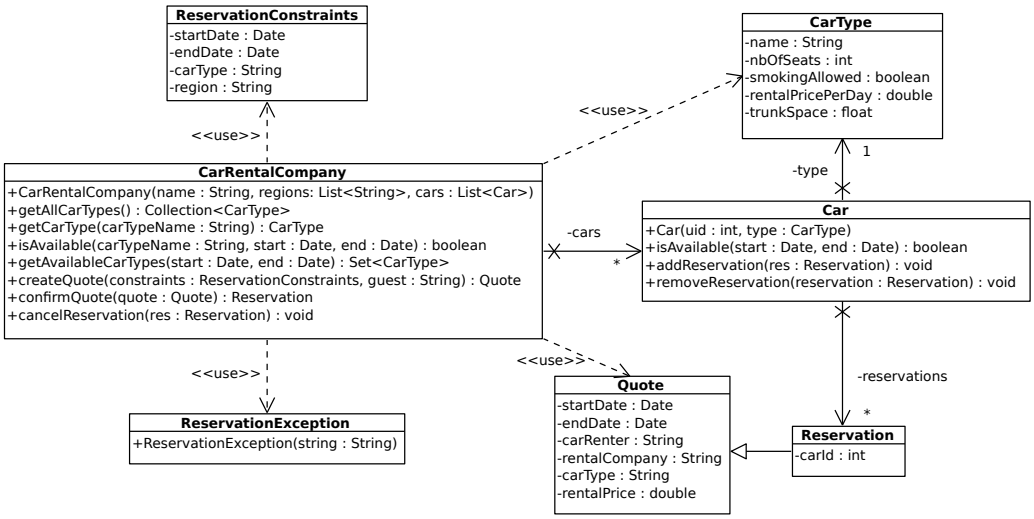

Figure 1: Design of the local car rental application.

3.1 Design

Start by sketching a design of your application, extending the design in Figure 1 to a *distributed* solution. Think about which classes are remote and which are local, and of which classes objects will get sent over the wire. You do not need to submit this design sketch.

3.2 Implementation

You will have to provide the following functionality in the implementation of your design:

1. A company (which uses the server application) offers a remote `CarRentalCompany` object, which clients (car renters) can use to make reservations. Allow clients to locate this remote instance of the rental company object in a naming registry.
2. Renters (who use the client application) first check which car types are available in a certain period. Let the client print this list of available car types.
   Then, the client application should create quotes by collecting the necessary constraints (car type, region in which to collect the car, start and end date) into a `ReservationConstraints` object. It supplies these constraints to the car rental company, which will try to create a corresponding `Quote`. Print the details of this tentative reservation.
   Finally, the client tries to confirm the provided quote, converting it into a final `Reservation`. Print the confirmation of this reservation.
3. Car renters should be able to see the reservations that they have already made. Add a method to the client 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.
4. Managers for the car rental company should be able to retrieve statistics on the reservations made across the company. Add a method to retrieve (and print) the number of reservations for a specific car type. In this simplified application, the manager uses the same client application to retrieve this information; no authentication is needed to distinguish the manager from car renters.

4 Getting up and running

The code of the local application 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. Use your preferred IDE or the command line.

In the **computer labs**, you can use the preinstalled Eclipse IDE:

`/localhost/packages/ds/eclipse-rmi/eclipse`

If you want to develop your code **on your own machine**, check the 'Bring Your Own Device Guide' on Toledo – we require you to use Java SE Development Kit (JDK) version 8.

For the **client** application, use the given `Client` class and implement the inherited methods. This application will execute a given test scenario (i.e. the `simpleTrips` test 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 (`AbstractTesting` and `AbstractTestBooking`) in the `client` package, the provided test script and `.csv` file.

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 (this can launch any number of Java programs if necessary). We already provide a `RentalServer` class with a `loadData` method that parses the provided `.csv` file to obtain the data for the car rental company.

Code organization. In a realistic 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, for simplicity we recommend to use a single code-base (~ single Eclipse project) for both components.

Checking the output. Executing your client runs the `simpleTrips` script file, which outputs the results of several test cases for retrieving quotes and making reservations. Check this output before submitting the assignment to see if it yields the expected result.

Collections. 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.*

4.1 Executing your code

Apache Ant. Apache Ant[3] is a tool for automating the build and execution process of your application, in order to support an 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.

We provide two ways of testing your application: running fully locally on your own machine, or running in a distributed fashion with the server code running on our remote cloud platform and the client code running on your local machine. **The `main` methods of your client and server classes should accept a `localOrRemote` parameter that is used to configure your code according to your setup (pointing to the correct IP addresses, ports, ...).** If you're not trying the remote setup yet, you can leave the remote part to throw an `UnsupportedOperationException`.

4.1.1 Running locally.

While developing your application, you can test whether it is functioning correctly by executing your code locally.

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:

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

- `ant run.local` → will compile your code, start the `rmiregistry` and run your application.
  This is the default (and most common) option.
- `ant run.local.wo-compile` → will run `rmiregistry` and your application.
  Use this option if you handle compilation yourself (e.g. Eclipse may throw strange errors with external compilation).
- `ant run.local.wo-registry` → will compile your code and run your application.
  Use this option if you handle the setup of the `rmiregistry` yourself.
- `ant run.local.wo-compile-registry` → will run your application.
  Use this option if you handle compilation and the setup of the `rmiregistry` yourself.

In the Eclipse GUI, you can right-click `build.xml` → `Run As` → `Ant Build...` → select (a) specific target(s). Alternatively, you can open `build.xml` in the edit window, open the `Outline` panel and right-click a target → `Run As` → `Ant Build` directly.

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). Where necessary, add additional folders to your classpath to make files accessible (e.g. the `.csv` files): `java -cp .:<additional folder> <class>`

4.1.2 Running remotely.

After testing locally, we provide a setup that allows you to experience running your server code on a remote machine in our cloud. Your client code can then interact with this remote setup. This allows you to thoroughly test whether your application actually works in a distributed manner (as opposed to 'discovering' some local objects to interact with). **It is not yet necessary to support this option in this session; ensure that your implementation is complete and that you thoroughly understand all concepts before using this remote testing setup.**

As you maintain one project, your client is able to retrieve all necessary class definitions locally from your server code. However, your client should be prepared to interact with (the objects on) a remote server for the actual operations on the car rental company.

You are assigned the endpoint to which your client will have to connect as well as a username and password to connect to the remote machine, allowing you to upload your code. Enter these parameters into the appropriate fields of `build.xml`. You also receive a range of ports that you can use for all communication between client and server. **Use only these assigned ports,** otherwise communication will fail as it is blocked by our firewall.

Important: Given that your code needs to be uploaded to the remote server, an available slot on this server has to be allocated to you and your code has to be started, **only try running remotely once your code is sufficiently developed!** We use the username/password to track whether you are not needlessly using server resources.

Locate rmiregistry. The client code needs to locate the remote RMI registry: in your original implementation the RMI registry ran on the same host as the client, but now it will execute at the remote side. Therefore, you should update the code in the remote block of your client that locates the RMI registry. You can configure where to find the RMI registry if you use `LocateRegistry.getRegistry(<host>, <port>);`. Alternatively, you can combine RMI registry discovery and stub retrieval if you use `Naming.lookup(//<host>:<port>/<stub_name>);`. Use only your assigned ports.

You have the option of automatically starting the RMI registry on the remote server, or starting it from within your server code on one of your assigned ports (`LocateRegistry.createRegistry(port);`). In the `build.xml` file, set `remote.startRegistry` to `true` or `false` accordingly; in the first case, also set `remote.registryPort` to the (assigned) port on which you want to reach the RMI registry.

Exporting objects. Make sure to export your remote objects on **one of the ports that you were assigned**, otherwise your client will not be allowed to access the object (because of our firewall). The second argument of `UnicastRemoteObject.exportObject` allows you to set this port. This also means that you can no longer rely on extending your class with `UnicastRemoteObject`: this causes ports to be randomly selected and therefore likely fall outside your assigned range.

Prepare the Ant script:

- Put the provided endpoint, username and password in the relevant properties of the `build.xml` file.
- Make sure that the client and server packages are correctly filled in, as these will determine which server code is uploaded to the remote machine.

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.remote.server` → packages your server code in a JAR, sends this JAR to our remote machine, requests an execution slot on our machine and starts your server code on our machine, while transmitting the server console output back to your computer.
- `ant run.remote.client` → will start your client code. **Only run this task after your server code has properly started, i.e. after the previous server task, and then only once you see a `'Deployment successful'` message!**

As you run the server and client tasks separately, their outputs will appear in two separate consoles. **Important:** Once your client test script has finished, don't forget to terminate the server (in the console).

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

Run server application.

- Generate a `MANIFEST.MF` file that contains the following line: `Main-Class: <server-package>.<server-mainclass>`. Place this file into a folder called `META-INF`.
- Package your server code by creating a JAR file containing the necessary *class* files (located in `bin`), manifest file and `csv` files:
  `jar cf <JAR file> <class files> META-INF *.csv`
- Upload this JAR file and request a server slot, either by executing `main` in `RemoteSetup` with the endpoint URL, username, password and JAR file as arguments:
  `java RemoteSetup <endpoint> <username> <password> <start RMI registry on server: true/false> <RMI registry port> <JAR file>`
  or by uploading the file manually, e.g. using `curl`:
  `curl -X POST -H "X-Registry-Start: <true/false>" -H "X-Registry-Port: <port>" --data-binary @<JAR file> http://<username>:<password>@<endpoint>`

Run client application. Run the main class of your client, passing `REMOTE` as an argument to use the remote server: `java -cp . <client-package>.<client-mainclass> REMOTE`

4.1.3 Preparing your submission.

`ant zip` → will zip your solution for submission on Toledo.

In case of problems, alternatively use `zip -r rmi1.firstname.lastname.zip <your source directory>` to obtain a zip for submission.

References

[1] Oracle. *Java RMI Specification.* `https://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmiTOC.html`

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

[3] Oracle. `java.rmi` *Properties.* `https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/javarmiproperties.html`

[4] Oracle. *Dynamic code downloading using Java RMI.* `https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/codebase.html`

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

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