Distributed Systems Java RMI

Charles Van Damme , Nils Lefever

2 November 2018

1 Overview

The car rental agency application is a distributed application that allows clients to make bookings at several car rental companies, through the central car rental agency hub. Also, it allows a manager to retrieve statistics about the rental agency. The application itself basically consists of four major components. The **client** interacts only with the car rental agency, from which it retrieves its reservation session. The **rental agency** serves as a central point for clients to interact with the car rental companies and manages the sessions. It knows which companies are connected with the agency by looking up the naming service. The **naming service** is responsible for the (de)registering of companies at this agency. Finally, multiple **car rental companies** can be registered at the naming service. The companies keep fulfilling their initial role and can live on their own servers. They do not have to be aware of the distributed setting of an agency, which increases the flexibility of the system.

2 Design

2.1 Remotely accessible classes

This subsection describes the classes that are remotely accessible, which means that they extend the class Remote. Instances of classes that are remote always get returned by remote reference and never by value. This is necessary to ensure that changes made by different clients of this distributed application happen consistently. If the instances of these classes would get returned by value, each client would modify its own instance, which is not convenient for the purpose of a car rental agency. More specifically, the following classes in our application are declared remote, because their methods are invoked non-locally:

· **ICarRentalAgency** : The remote interface that a client uses to obtain a reservation session, or a manger to obtain a manager session.

· **ICarRentalCompany** : The remote interface of a car rental company which a reservation session uses to call the company's methods.

· **INamingService** : The remote interface that a car rental agency uses to retrieve which companies are associated to it.

· **IReservationSession** : The remote interface that a client mainly uses to confirm and create quotes.

· **IManagerSession** : The remote interface of a manager session that allows a manager to (un)register companies and retrieve statistics.

2.2 Serializable classes

As opposed to the Remote classes described above, instances of serializable classes are returned by value. Serializable classes implement Java's Serializable class. The classes below extend serializable because they get transferred between the different distributed components. For example, when a client calls the method 'getAllCarTypes()' through its reservation session at a certain car rental company, a collection of the type 'CarType' is returned.

· **Car** : A class that describes a car, with a type and an ID.

· **CarType** : A class that describes a certain type of car.

· **Quote** : A class that describes a quote.

· **Reservation** : A class that describes a Reservation.

· **ReservationConstraints** : A class that describes the constraints of a reservation (start and end date, company and renter name).

2.3 Locations of remote objects

The reservation session and manager session are located on the same host as the rental agency. All other remote objects reside on their own servers. The naming service lives on the naming server and each car rental company has its own server. However, multiple car rental companies can also reside on the same server. This increases the systems flexibility, since a car rental company with its own server can be added to the naming service without any changes to the system.

2.4 Remote objects in the RMI registry

In the distributed car rental agency application, the remote objects of car rental agency and the naming service are registered in the RMI registry. When a client wants to start a session, it queries the car rental agency via the RMI lookup method. The rental agency on his turn retrieves the naming service via an RMI lookup. The naming service contains all the car rental companies associated with our particular agency, and hence do not need to be registered themselves.

2.5 Sessions and life cycle management

Because our application works with plain RMI, we need to implement some aspects that were provided by Java EE in the previous lab ourselves. To cover the session part of Java EE in our RMI application we defined an abstract session class from which the reservation and manager session inherit. A session gets created by the car rental agency upon request of the client and stores a creation date and ID number. This way the administrators can decide how long to keep certain sessions alive. Also, by distinguishing between a reservation and a manger session, we can shield the client from being able to call manager methods and vice versa.

2.6 Synchronization and thread-safety

To support concurrency control, some methods in our application are synchronized. More specifically, the methods for (un)registering a car rental company are synchronized because they cannot be called simultaneously by multiple clients. The same goes for the confirmation and removal of quotes. When multiple clients are booking the same car at the same time, the synchronized method will take care of this conflict. If not, our agency might have double reservations for the same car at the same time. This way of handling concurrency control might become a bottleneck if our agency grows larger, which means that many clients could be trying to make bookings at the same time. Because of the synchronized methods, the server would only handle one request at a time, which could obviously slow down the system dramatically.

3 Design Artifacts

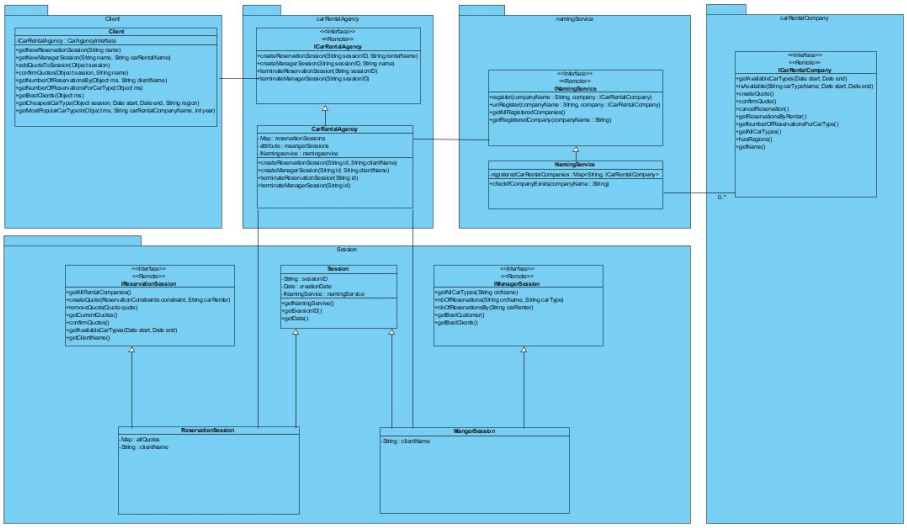

Figure 1: UML Class Diagram

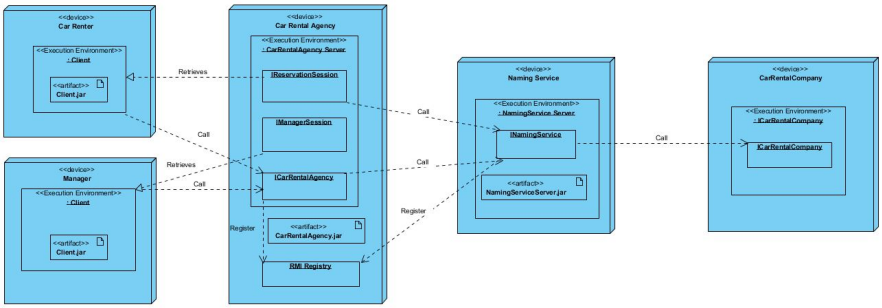

Figure 2: UML Deployment Diagram

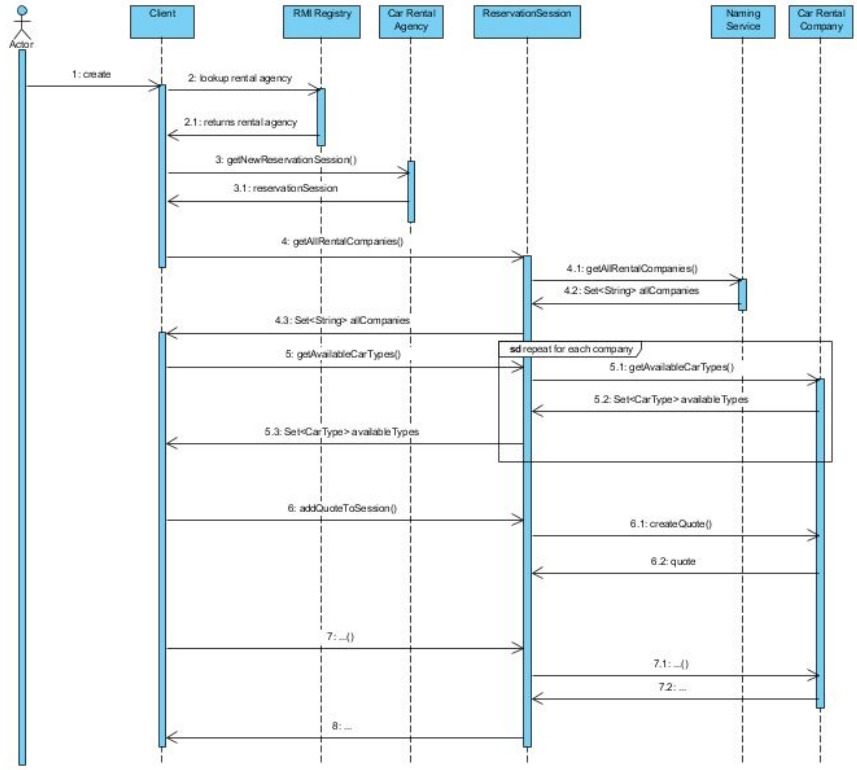

Figure 3: UML Sequence Diagram