RMI session 2-3 report

Stien Vanderhallen, Joke Denissen

October 2018

1 Report

1.1 Overview

This design of a car rental agency consists of five components, which can all run on different servers or devices in a distributed system. A first component is meant to run on the device of the **client** of the car rental agency, whether that is a regular client wanting to make a reservation or a manager of a car rental company. A client interacts with the car rental agency by means of **sessions**, which can be hosted on yet another device. Two types of sessions can be used by the client; either a reservation session or a manager session. A central managing service is implemented in a third component, the **session master**. A client interacts with that session master-module when he/she wants to create a new session or retrieve its session currently active. The session master-component is meant to be run on a device or server, managed by the car rental agency.

The car rental agency is meant to manage data of different car rental companies. The **rental** component in the project is meant to represent those companies; the module can be run on a server or device in each car rental company affiliated with the car rental agency and entails the implementation of all methods the client can run against a company; it answers all questions a client can ask about the company and is able to perform the actions the client requests (e.g. finalizing a reservation). On yet another device or server, a **naming service** is run. That service is meant to keep track of all car rental companies associated with the car rental agency. The car rental agency, as well as the reservation- and manager-sessions, interact with that naming service to retrieve the distributed objects, associated with the names of car rental companies. A manager of a car rental company first has to register its company at this naming server, in order for it to be accessible.

1.2 Classes

1.2.1 Remotely accessible

Each component of the project is in some way remotely accessible, which is necessary to be able to speak of a distributed system; the different components

are located at different servers and have to be able to use each others objects. When instances of the remotely accessible classes listed below have to be used, the components involved do not use the actual values of these instances, but they communicate by remote references to those instances. That way, no clones of the same instance (and thus inconsistency of those clones) can exist, which of course is necessary when handling e.g. reservations. To manage all remote references, the RMI-registry is used, as it was the purpose of this assignment. The implementation of all remotely accessible objects is hidden from their users by means of interfaces, which adds a certain layer of security and flexibility to the project.

The interfaces listed below implement the java.rmi.Remote interface, which makes them and all classes implementing then remotely accessible:

- **ISessionMaster**: accessed by a client to e.g. start a new session.
- **IManagerSession**: accessed by a manager to e.g. register a new car rental company.
- **IReservationSession**: accessed by a client to e.g. attempt to make a new reservation.
- **ICarRentalCompany**: accessed by e.g. a reservation session to find all available car types in a certain period.
- **INamingService**: accessed by e.g. a reservation session to retrieve all car rental companies registered at the car rental agency.

1.2.2 Serializable

Serializable classes implement the interface java.io.Serializable. Contrary to remotely accessible classes, instances of these classes are returned by a value, which does result in inconsistent clones. The client is thus allowed to view and use the properties of these serializable objects, but is not able to modify the properties of the "root" object that resides at the server where it was created. The process of serialization is used when object need to be (un)marshalled when used as arguments in remote method invocations.

The serializable classes in this project are Car, CarType, Quote, Reservation and ReservationConstraints. The class ReservationException is also serializable; although it does not directly implement the java.io.Serializable interface, it extends the Exception class, which extends the Throwable class, which implements the Serializable interface.

1.3 Remote objects

1.3.1 Location

The session master is hosted by the same device as the reservation- and manager-sessions, which makes sense because of the close association of the session master and all sessions. The NamingService is hosted by a different server, since it is accessed by multiple components and thus cannot simply be 'merged' with some particular component. A CarRentalCompany can be on a separate server or multiple CarRentalCompanies can reside on the same server, depending on the load that can be handled by the servers involved.

1.3.2 Registration

All remotely accessible objects are registered at the built-in RMI registry, in order for them to be accessible across the system by some name used in the registration. An extra layer of registration is used in the naming service, in which references to all car rental companies registered at the agency are managed. The session master manages the registration of all active sessions.

1.4 Life cycle management

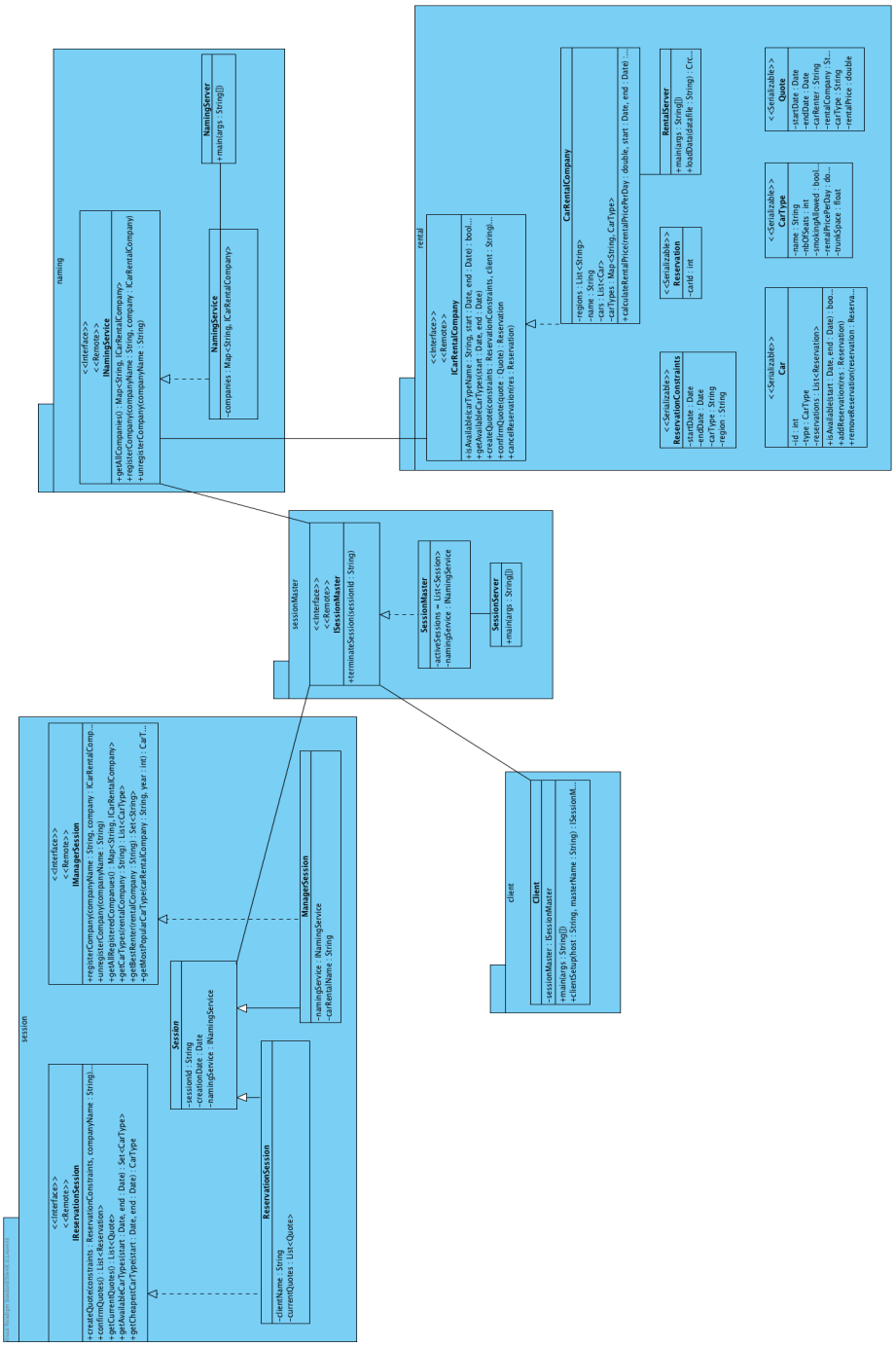
The session master manages the life cycle of sessions; it provides the client with methods to either (re)start or terminate its session currently active. To this end, each session is provided with an ID. More specific, when a client requests the session master for a session with a certain ID, the session master either returns the already existing session with that ID, or it creates a new session with that ID. A client can terminate its currently active session through the session master. No security measures have been taken to prevent unauthorized users to terminate sessions. In conclusion, the session master covers the full life cycle of a session; creation, active phase and deployment and termination.

1.5 Synchronization

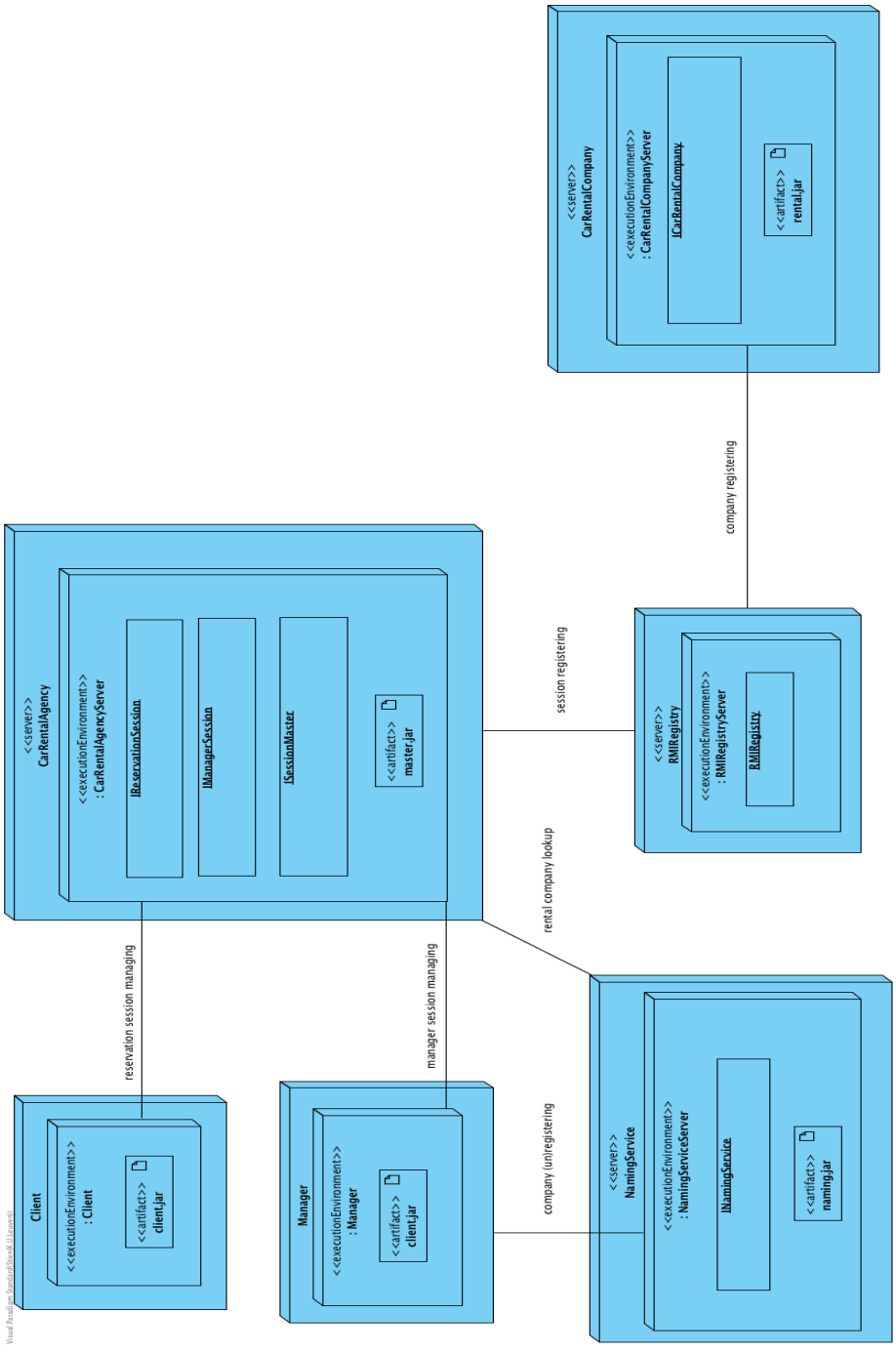
All *critical section*-methods, i.e. methods that can be concurrently invoked and modify remotely accessible features, have to be synchronized in order to guarantee atomicity, consistency, integrity and durability of the system. Some examples of methods that have to be synchronized are the (un)registering of car rental companies at the naming service and all methods dealing with the confirmation and cancellation of reservations at a car rental company. Such synchronized methods could become bottlenecks when a lot of users are using them at the same time.

2 Diagrams

2.1 Class diagram



2.2 Deployment diagram



2.3 Sequence diagram