Gedistribueerde Systemen

RMI Session 2-3: CarRentalAgency

Thomas Uyttendaele s0215028

Matthias van der Hallen s0219692

Burgerlijk ingenieur Eerste Master
Computerwetenschappen

Begeleiders:
Prof. S. Walraven

2012/11/02

Inhoudstafel

1   Overview Design

In this section there will be an overview of the most important components and their responsibilities. This will all be discussed with the aid of the following *class diagram* 1.
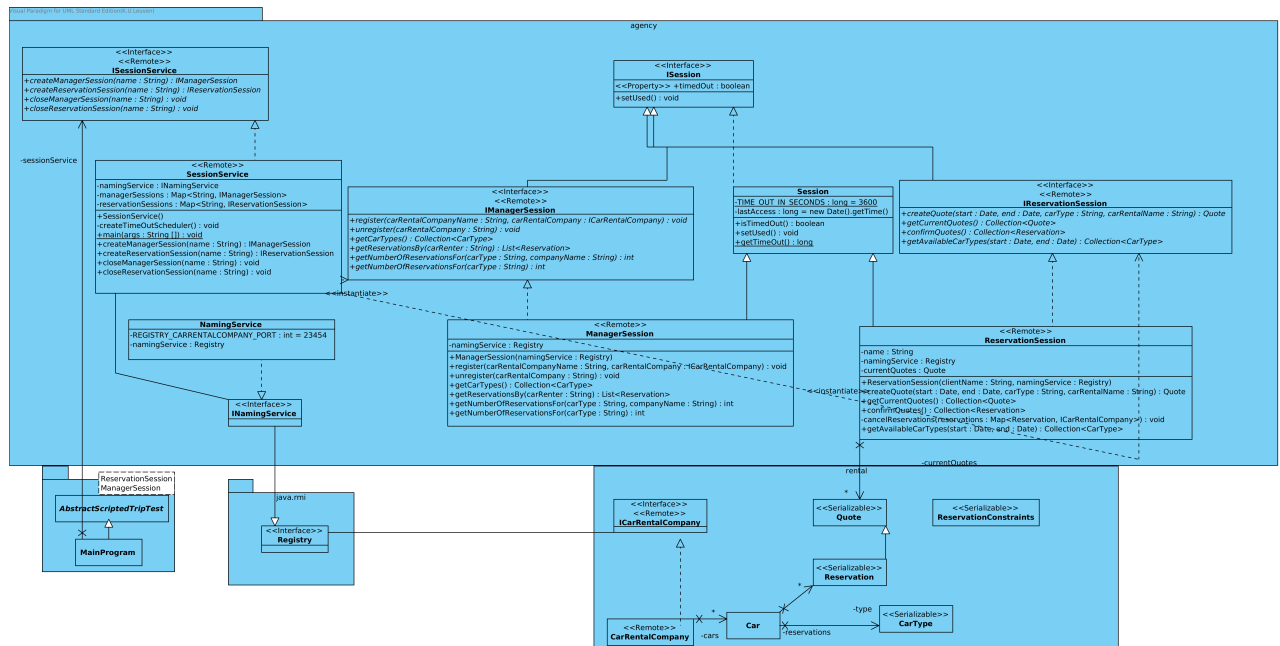

Figuur 1: Overview of the design

1.1   SessionService

This is the factory class for the creation of new `Manager-` and `ReservationSession`s. During initialization, it registers itself at the standard JAVA RMI Registry. `SessionService` runs on the travel agency server. On this server, it is a stand alone program and it provides the main access point to our agencies service. By holding a link to the CarRentalCompany `Namingservice`, which is a second JAVA RMI Registry that can run on a different server, it can provide the sessions it creates with access to this naming service.
`SessionService` is also responsible for the garbage collection. It has a link of all the active sessions and in normal use the sessions should be closed with the `close...Session`-method. When a session is inactive for more than an hour and not closed by the client, it will automatically be closed by the `SessionService`. (In our implementation, for testing purposes, the inactivity timeout is only 3.6 seconds instead of the aforementioned one hour.)

1.2   ManagerSession

This class is used by managers to access the agency server and perform actions on this server. They can register and unregister new `ICarRentalCompany`s at the CarRentalCompanies `NamingService` which was mentioned before.
It can also perform various statistics related methods, such as `getReservationsBy(renter:String)`.
The normal behaviour of the registration process is given in the following *interaction diagram* 2.

1.3   ReservationSession

This class is used by clients to make reservations on the server. This works in 2 separate steps, to ensure transactional behaviour: first the client creates all the necessary `Quote`s and afterwards the client can confirm them, at which point they are all reserved, or the entire confirmation fails.
The transaction behaviour means this process can fail sometimes, when multiple clients are concurrently making reservations. In the first step, by making a `Quote`, the Car is not yet put aside for the client. This means that if someone else makes a `Quote` and confirms it in the meantime, then the first clients confirmation
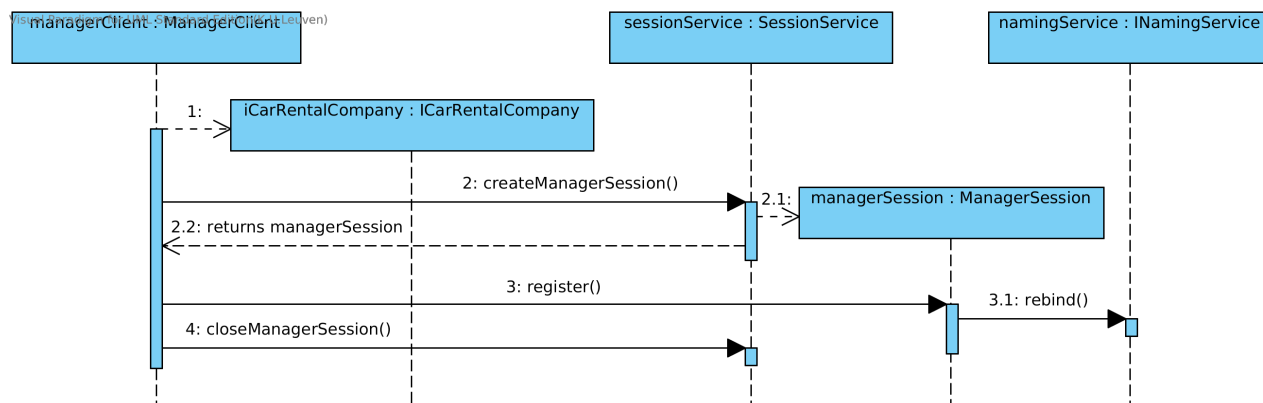
Figuur 2: Interaction diagram of the registration process

will fail if the company doesn't have any identical cars left. In this case, an exception will be thrown. This behaviour is shown in the following *interaction diagram* 3.

Access to these services are synchronised on the `NamingService` so only one client at a time can be in these critical sections, for example during a confirmation it won't be possible for others to make a confirmation.
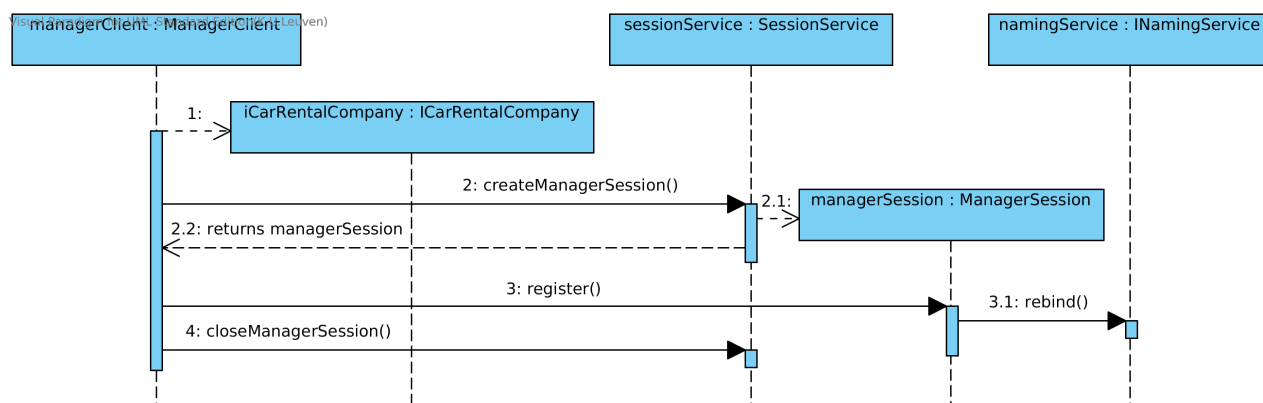
Figuur 3: Interaction diagram of the registration process

1.4 `CarRentalCompany`

This is the regular CarRentalCompany from the first JAVA RMI session. It implements `Remote` so it could run on the server of the CarRentalCompany itself instead of running on the agency server.

The rest of the behaviour of this class has stayed the same as in the first JAVA RMI session.

2 Design Decisions

2.1 Remote Access

As can be found in the *class diagram* (figure 1), the following classes are remote: `ISessionService`, `IManagerSession`, `IReservationSession`, `INamingService` and `ICarRentalCompany`. These provide an interface to programs running elsewhere, meaning they should be remotely accessible. The sessions provide a service interface for the client, `ISessionService` provides a means to create sessions. The `ICarRentalCompany` is kept separate from all this because it's property of a different company, and the `INamingService` offers a lookup service where `ICarRentalCompany`s can be registered.

`Quote`, `CarType`, `ReservationConstraints` and `ReservationException` are *Serializable* because they are data-classes that never change after initialization, and because they have to be passed from the travel agency server to the client and vice versa.

2.2   Deployment of the application

The deployment of the application is shown on the *deployment diagram* 4. Most importantly, the `CarRentalCompanys` run on a different host than the `SessionService` and the `NamingService`. This is because they are property of the respective `CarRentalCompanys` themselves.
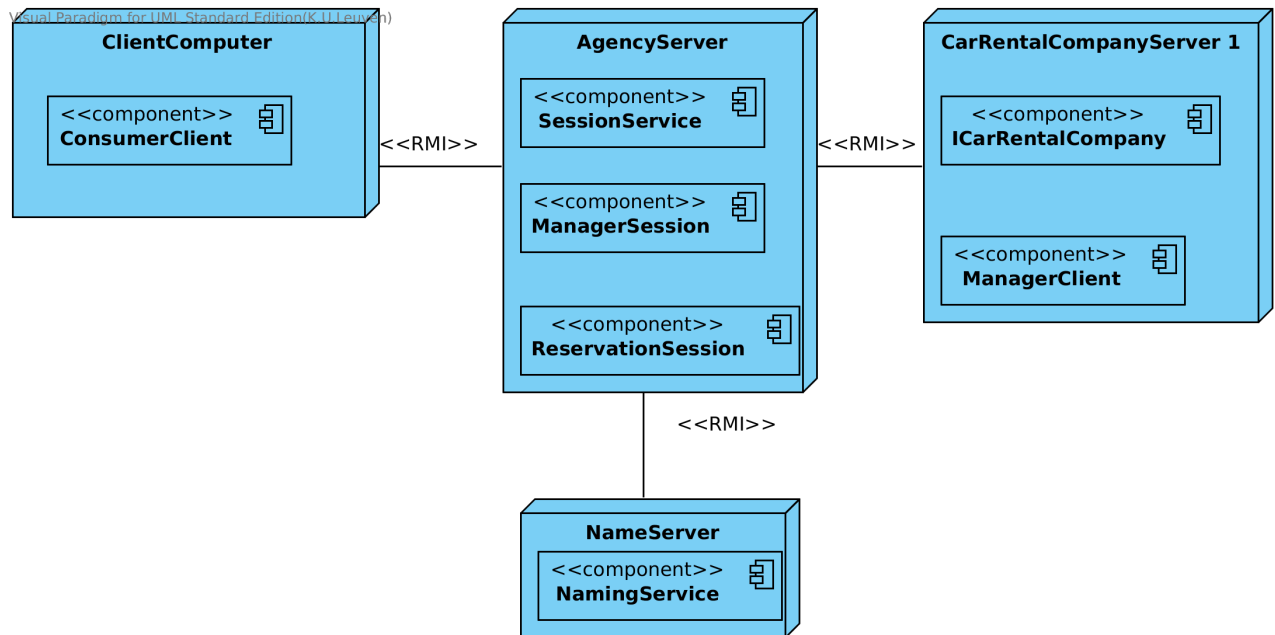
Figuur 4: Deployment of the application

2.3   RMI Registration

The `NamingService` and the `SessionService` are registered at their own local JAVA RMI Registry. These are standalone applications running on their own respective servers. `SessionService` has to know the host and port of the Java RMI Registry that the `NamingService` bound itself to, so that it can acquire a link to the `NamingService`. The `SessionService` is registered on its own local JAVA RMI so that the client can perform a lookup to acquire a hook to the `SessionService`.

`CarRentalCompanys` are registered by a `ManagerSession` at the `NamingService`, which functions the same way as a JAVA RMI Registry. (It offers registration, unregistration and lookup of objects implementing the Remote interface)

2.4   Synchronization

In the `ReservationSession` class, thread-safety is of utmost importance. When multiple clients are making quotes and confirming them, things could behave unexpectedly. To prevent frustrations and problems, the actions available to this class are performed in a *critical session* by using the JAVA *synchronised* keyword on `NamingService`. This prevents concurrent users from reserving cars at the same time.

These methods are short, but require remote calls, which means there might be latency from the network which could cause long execution times. This suggests that they are quite prone to becoming a bottleneck. One slight optimization would be to lock on only the companies used in the transaction, instead of all the companies, as in the current implementation. But there are likely other more efficient methods, such as locking on the `CarType`s (however this is serializable, so extra locking logic is needed) and the corresponding the `ICarCompany`s involved. That way, more concurrent modifications are possible, but it's also much more difficult to implement, and the synchronization and the locks need more time.