Distributed Systems: Java RMI session 2/3

Jago Gyselinck, Armin Halilovic

November 11, 2016

1 Overview

Our design consists of three major components: the client(s), the rentalAgency and sessions, and the car rental company server(s). A lot of code is inherited from the previous JEE and RMI assignments so responsibilities are similiar.

The rental agency serves as a central hub that the clients communicate with, and serves as a naming server where car rental companies are registered. It also manages the sessions that managers or renters can use. Those sessions can then directly interact with the car rental companies registered with the rental agency.

The client interacts only with the central rental agency, which serves it sessions it can use to perform the required managerial/reservation tasks in the assignment.

The car rental companies fulfill their traditional role, they live on separate servers that load their respective data files, but aren't adjusted much compared to previous assignments.

1.1 Serializable classes

The following classes are serializable:

- `Car, CarType, Quote, Reservation, ReservationConstraints`
- We made these classes serializable as data of their type has to be communicated between different distributed components.
- An example of this could be the `getAvailableCarTypes` method which returns `Set<CarType>` and is made available in the `CarRentalCompanyRemote` interface. The class `ReservationSession` makes use of this method, but resides on a different distributed component. For the method invocation on a remote reference of type `CarRentalCompanyRemote` to succeed, `CarType` must be serializable.

1.2 Remote classes

The following classes are remotely accessible:

- `CarRentalCompanyRemote, RentalAgencyRemote, ManagerSessionRemote, ReservationSessionRemote`
- We made these objects remotely accessible because their methods will be invoked from a non-local context, and remote references of their type will be passed along between different distributed components.

1.3 Remote Object Locations

- Only the sessions (Manager and Reservation) and the Rental Agency reside on the same host. This allows the client to request a remote reference for the `CarRentalAgency` via the rmiregistry, and remote references to sessions can be requested via that remote reference. Those session remote references are kept in the remote `CarRentalAgency` object so they can be removed later. This organisation also has the advantage that when a method is invoked on a `(Manager/Reservation)Session` through a remote reference, the `CarRentalAgency` it has to interact with is a static object on the same component, so no remote interaction is required.
- All other remote objects are located on different hosts. The car rental companies each reside on their own server as they execute independently of the car rental agency and there can be an arbitrary number of them. The client does not interact with them except through the RentalAgency. The client is of course also on a separate host, as there too can be as many or as few of them as you want, each executing independently and interacting only with the RentalAgency.

1.4 Registering of remote objects

- The `CarRentalCompanies` and the `RentalAgency` are registered in the built-in RMI registry. We register the `RentalAgency` as it is the central starting point for clients to interact with. The `CarRentalCompanies` are also registered when their respective `CarRentalServer` starts. Again this is done so that there is some initial reference available that `ManagerSessions` can use to add the companies to the `CarRentalAgency`. From then on the `CarRentalAgency` functions as a **naming server** for the clients so they can look up `CarRentalCompanies` by name and not via the rmi registry. The rmi registry offers this functionality too, but would require the client to know the names of the `CarRentalCompanies` in advance.
- `ManagerSessions` and `ReservationSessions` are not registered in the rmiregistry as they are requested and managed via the `RentalAgency`.

1.5 Life cycle management

We were not exactly sure how to handle this at first. During the practical session the assistant told us that it was sufficient to implement the methods `getNewReservationSession` and `removeReservationSession`. The former is called by the client when it wants a

remote reference to a `reservationSession`, if a `reservationSession` already exists for the client's name it, then it is returned. Otherwise, it is created. The latter is called by the client when they are finished doing their operations and causes the removal of the reservationSession coupled to the client's name.
This works in a different way for `ManagerSessions`, they are stateless and thus only one `ManagerSession` is kept alive at all times, whose remote reference is returned to serve all managers.

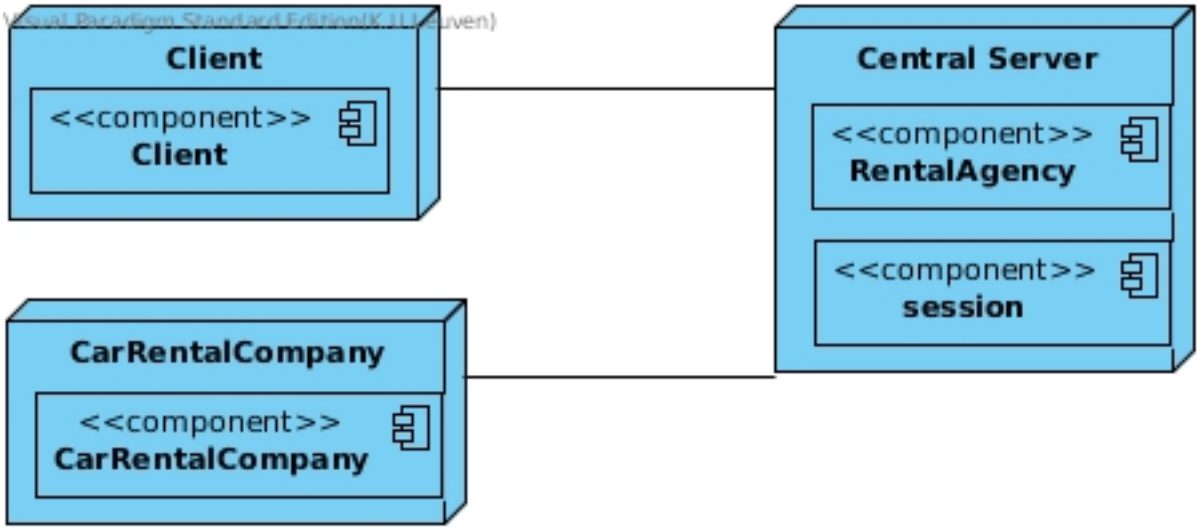
1.6 Synchronization

We applied synchronization where multiple calls could possibly modify a resource concurrently. Examples of this are most of `ReservationSession`'s methods. This will cause a bottleneck if the application has a lot of concurrent users, but for our testing purposes the difference is not noticeable.

2 Full class diagram

See 'class-diagram.jpg'.

3 Deployment diagram



4 Sequence diagram

Sequence diagrams of the booking process have been included in the project. See 'sequence-diagram-success.jpg' and 'sequence-diagram-fail.jpg'.