Design report

Sebastiano Valente, Thomas Vandendijk

November 3, 2017

1 Introduction

First we'll briefly discuss a quick overview of the most important parts of the implementation. Then we'll talk about which classes are remotely accessible and which classes are serializable and why. Afterwards we discuss the three big sections which are remote from each other. After that follows a discussion of the use of RMI. Then we'll talk about the life cycle management of each session and lastly we discuss the different methods that needs to be synchronized.

2 Short overview

This section discusses the most important parts of the implementation. Our design exists of three major components: we have the client side, the session side and the server side. The client side exists of purely a client class (Client) which calls upon the session side to perform actions. the session side exists out of a session handler, and two separate session beans. The session handler is responsible for managing all the separate beans that are created. One session bean named RentalSession exists to serve a renter. This session is stateful. A second type of session exists to serve a manager, the ManagerSession. This manager session is stateless. The session side also contains a central naming server which we called RentalStore. RentalStore contains a map of the company names and their respective interfaces. The final side is the server side, which contains the database and methods a company can perform independently. The class who performs these methods is CarRentalCompany.

3 Remotely accessible and serializable classes

This section explains which classes are remotely accessible and why. After that follows a brief discussion of the serializable classes. We defined four interfaces which are remotely accessible. The first one is ISessionHandler. This class handles the life cycles of the sessions. Because the sessions have to be created on the client side, the clients need to have remote access to those methods (e.g. CreateManagerSession()). Also the clients need to invoke methods of the sessions itself. These sessions are handled on a different location than the local computer of the client so we need remote access to a rental session and a manager session. Therefore we defined the IRentalSession and IManagerSession as Remote. The fourth interface is at the server side. Each CarRentalCompany could be on a different server aswell as on a different location from the classes that invoke the session methods (ManagerSession and RentalSession). Therefore these companies have to be accessed remotely. The interface ICarRentalCompany which extends Remote makes this possible.

The implementation also needs serializable classes because some objects need to be marshalled before we can send them over the network. These are the inputs and outputs of the functions a distributed party invokes. In total there are three classes that were manually made serializable: CarType, Quote and ReservationConstraints. We made the cartype serializable because it contains a lot of information that could be of help to the client. So we made it possible for clients to receive cartypes. Another option would be to only send the name of the cartype as a string. Then CarType doesn't have to be serializable but to us sending CarType's directly doesn't cause a lot of overhead and gives nice extra information which isn't confidential. The second class which is serializable is Quote. Quotes have to be kept in a session which are held at a different location than the server of the CarRentalCompany and have to be sent as an input to confirm a quote

in a company. Reservation inherits from Quote so it automatically becomes serializable which is handy because the client gets a list of Reservations as output from the method confirmQuotes(). The last serializable class is ReservationConstraints which the clients has to send as an input for the method CreateQuote(). These classes were convenient to send over the network, and don't contain information that can be abused by the ones who receive them. A last remark is that ReservationException is actually also serializable because it extends from Exception which is Throwable. All Throwable objects are also serializable.

4 Division of remote objects

As mentioned in the short overview the project is split in three sections: the client section, the session section and the server section. We want to prevent remote access as much as possible because it slows down the execution of client methods. The client sections stands for the renters and the managers who can invoke the given methods on their local computers. Next is the session section which contains the life cycles of the sessions aswell as the naming service for the different CarRentalCompanies. We linked those two together because we wanted to prevent remote access as much as possible but still create an implementation that is well distributed. The sessions need the information of the different CarRentalCompany's, so it seemed logical to put them together with the class who has remote access to them namely the naming service RentalStore. We could have made RentalStore remotely accessible to the sessions, but to us the performance overhead wasn't worth the extra agileness. Lastly we gave the server section which stands for each CarRentalCompany. There is a remote connection between the naming service and the CarRentalCompany. So the naming service contains a map with the names of the company as keys (which are also the lookup string) and the remote interface as value. The sessions have local access to the map so they can call methods on those interfaces. To conclude we have an encapsulation of a client who can create session, terminate sessions via the remote ISessionHandler interface, the client can then either call the methods from the IManagerSession or the IRentalSession. Then we have an encapsulation of the session section which is used to handle the session by remote access to the needed CarRentalCompany via the remote interface ICarRentalCompany and finally the server section which stands for a CarRentalCompany that is remote accessible by the earlier named remote interface.

5 Objects in the rmi-registry

This section discusses how we used the RMI and which remote objects we registered and why. The four objects that are registered with the RMI registry are the ISessionHandler, IManagerSession, IRentalSession and ICarRentalCompany. The ISessionHandler needs to be found by every client so we register it once under a name that's known by all the clients: ISessionHandler.class.toString(). Now every client knows which lookup it has to perform. After that the ICarRentalCompany interfaces have to be remotely accessible by the naming service. They first have to be binded in the RMI registry (we assume the company servers do this themselves) and then they can be added to the naming service by the manager. The registerCompany(String lookup) assumes the company that is registered is already binded with the java RMI. Because of this the naming service (RentalStore) can perform a lookup on the registered company and store the remote interface when the method registerCompany(String lookup) executes. When a renter creates a new session an interface gets created from a new RentalSession, this is because it has to be stateful. This remote interface binds with the RMI registry so that renter can look it up and get access to it. Notice that for every new session we bind a new interface with the java RMI. So we sacrifice memory in our java RMI but we win performance because all interfaces can be accessed in parallel. Lastly we have the IManagerSession. We don't have to create a new session for every manager because the session is stateless. So when a manager creates a new manager session we'll first check if the IManagerSession is already binded with the RMI, if so then the lookup can happen to the (by everyone known) name IManagerSession.class.toString(). If it wasn't binded then we'll first bind it with the RMI registry.

6 Life cycle management

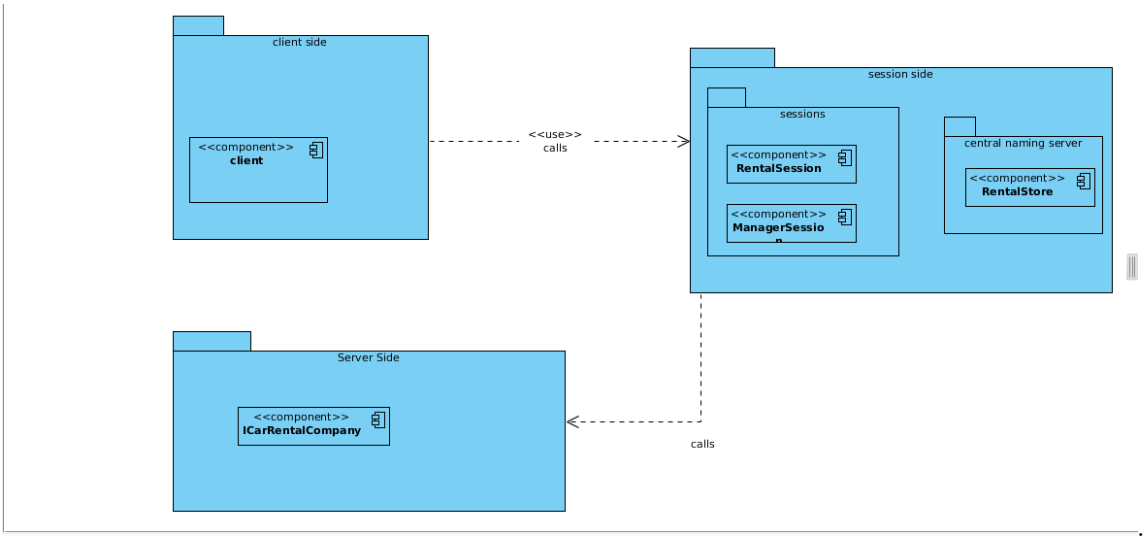
The life cycle management of our sessions is fully operated in our SessionHandler. The SessionHandler gives the option to create and terminate a session. When a session is created a remote interface gets binded to the RMI registry like discussed in last section. For the manager session mostly one interface is binded with the registry. For the rental session one for every renter that is performing a session is binded. The termination of the sessions is up to the user. The SessionHandler contains a function to terminate a renter or manager session which the user can execute. This will unbind the interface from the registry. Another option to extend this termination is to keep a timer for every session which automatically terminates the session after some time has passed. Be aware that the timer can't be too short because renters need time to make reservations. This timer feature is not implemented but could serve as an idea for the future. Finally SessionHandler also offers a private function to terminate all current renter sessions. This can be used to terminate the SessionHandler which unbinds all the sessions that were being made (see terminate()). After the unbinding of the sessions, the garbage collector cleans up the unreferenced objects.

7 Synchronization

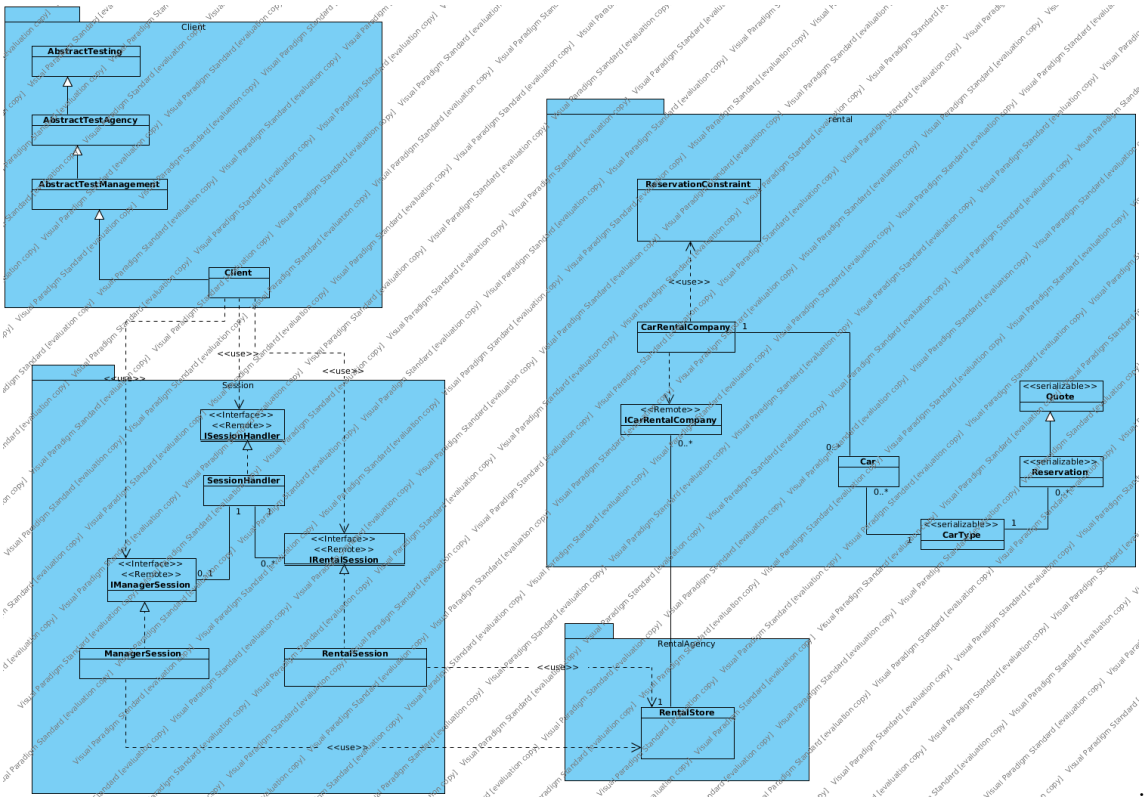
In this code, there are five methods which are synchronized: confirmQuote, cancelReservation in the class CarRentalCompany and registerCompany, unregisterCompany and getRentals in the class RentalStore. It is not possible for two invocations of synchronized methods on the same object to interleave. So we made methods synchronized to prevent race conditions on shared objects. We synchronized confirmQuote and cancelReservation to prevent interleaved operations on the availability of cars. Notice that we synchronize it on the methods that actually modify the data and not on the more standard methods like confirmQuotes() in the rental sessions. Next we have the registerCompany, unregisterCompany, these methods all access or manipulate the same map of companies that are registered. Also here we want to prevent interleaved operations because it could return the wrong data or change the map in a wrong way.

8 images

8.1 deployment diagram



8.2 Class Diagram



8.3 Booking sequence