Java RMI session 2/3

Jorden Belis & Wouter Vandenputte
november 2, 2018

Introduction

An interface `ICarRentalCompany` is an abstract talking point for methods between client and server. The server (the physical machine for al rental companies) implements this methods with a class instance from this interface and registers this remote reference on a certain port to the RMI registry. A client then locates the registry and gets access to these interface methods. Every method is thus executed server side.

For the above to work, some classes have been made serializable so they can travel between the talking points. The Quote (and thus Reservation), `CarType` and `ReservationConstraints` classes implement this `serializable` interface. This is necessary for a client to make reservations. Certain parameters need to have been marshalled in order for the server to unmarshall them.

Q&A

Which classes are remotely accessible and why?

- `RemoteCarRentalCompany, RemoteManagerSession, ReservationSessionRemote`

- Their methods will be called from different machines. Remote references will be passed along different distributed components.

Which classes are serializable and why?

- The `Quote` (and thus `Reservation`), `Car`, `CarType` and `ReservationConstraints` classes are `serializable`.
- The `ReservationConstraints` class is `serializable` in order to send certain constraints in Java classes (after being deserialized) to the server for it to process them. Thus every class composed by the client and sent to the server is serializable.

Which remote objects are registered via the built-in RMI registry (or not) and why?

- `CarRentalCompany` as well as `SessionControl` will be registered in Java RMI registry. When the CarRentalAgency server starts, companies can register through it and the SessionControl object is used for the management of sessions.
- Some classes are not registered. Obviously the simple domain classes and also the sessions. The latter is managed by the SessionControl class.

Briefly explain the approach you applied to achieve life cycle management of sessions.

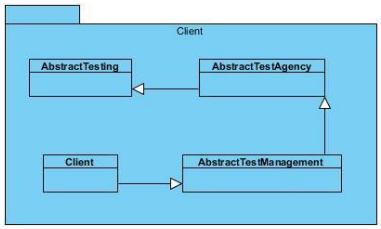
- The system stores sessions in the SessionControl class. The client is able to start a new sesssion through the means of `getNewReservationSession` in the `Client` class wich calls the **createNewReservationSession** method in the **SessionControl** class. Here, the session will be stored in a List. Deletion is not implemented in this assignment.

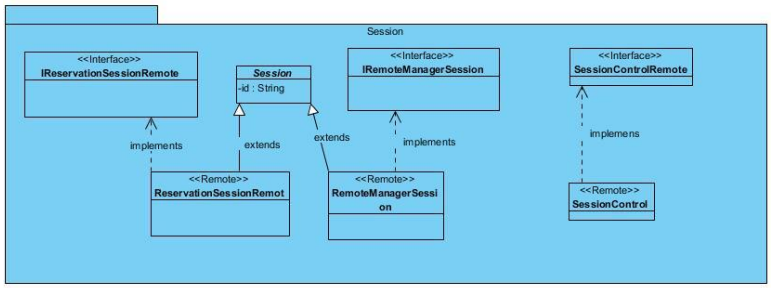
At which places is synchronization necessary to achieve thread-safety? Will those places become a bottleneck by applying synchronization?

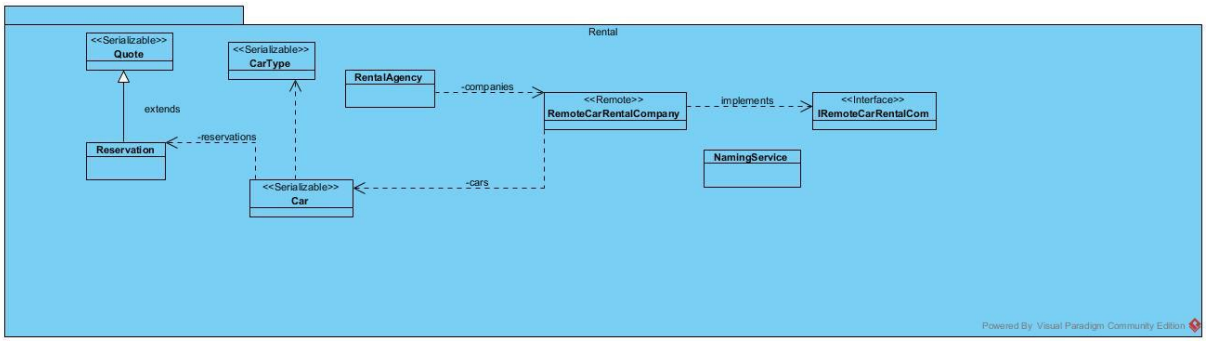
- Synchronization is necessary whenever multiple clients or entities use the distributed system. In this case, synchronization problems may occur at concurrent requests. To tackle this problem. The methods `CreateQuote` and `confirmQuote` (both in `ReservationSessionRemote`) have been marked synchronized. This prevents the method from being called more than once at the same time.
- Of course this creates the bottleneck issue. However, the frequency of the use of this method is rather negligible.

Diagrams

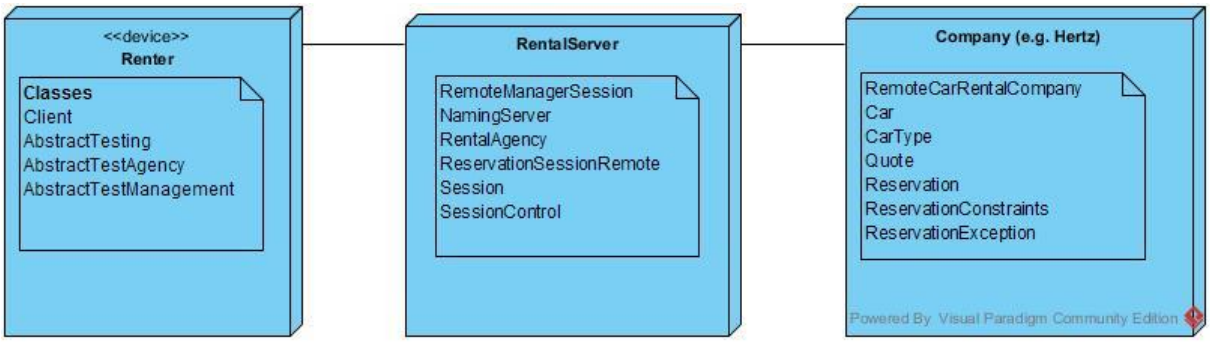
- Class Diagram



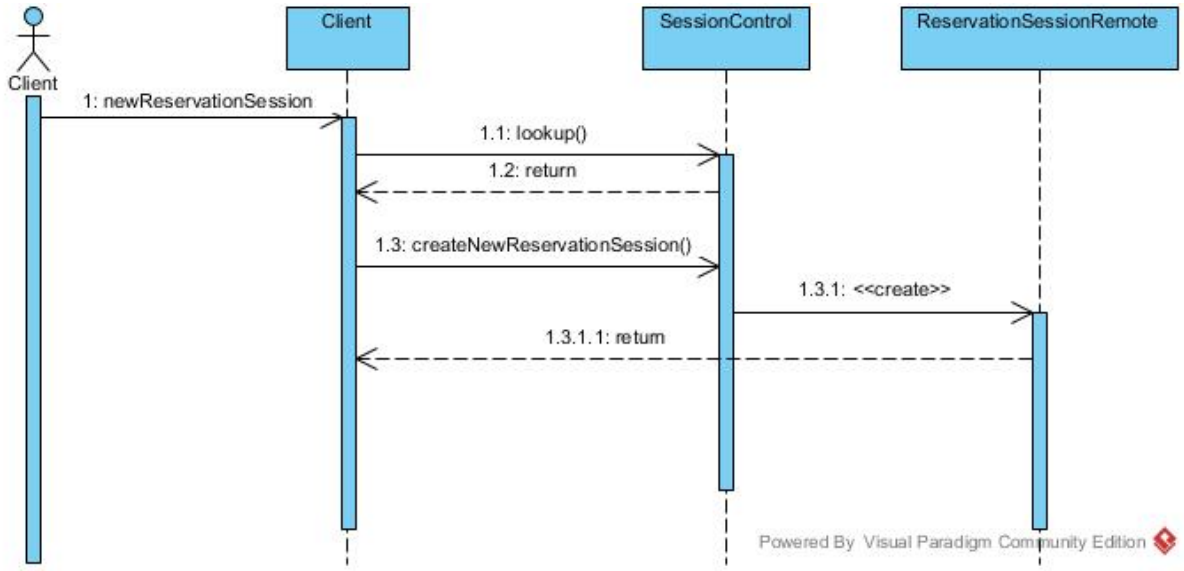


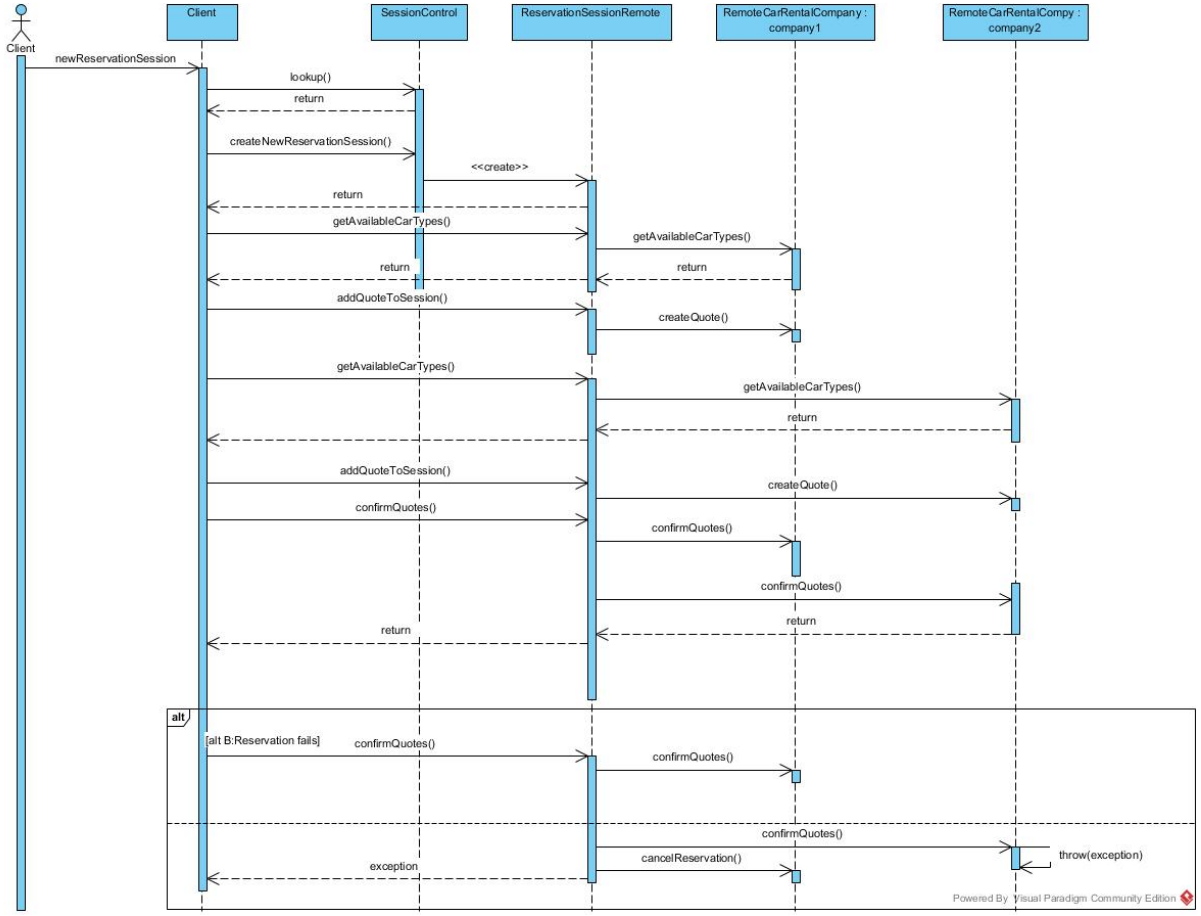


- Deployment diagram



- Sequence diagrams




Client
Client
SessionControl
ReservationSessionRemote
RemoteCarRentalCompany : company1
RemoteCarRentalCompy : company2
newReservationSession
lookup()
return
createNewReservationSession()
<<create>>
return
getAvailableCarTypes()
getAvailableCarTypes()
return
return
addQuoteToSession()
createQuote()
getAvailableCarTypes()
getAvailableCarTypes()
return
addQuoteToSession()
createQuote()
confirmQuotes()
confirmQuotes()
confirmQuotes()
return
return
alt
[alt B:Reservation fails]
confirmQuotes()
confirmQuotes()
confirmQuotes()
throw(exception)
cancelReservation()
exception
Powered By Visual Paradigm Community Edition