TaxiService

Design Document

Authors: Jacopo Silvestri 773082 Simone Penati 850448
Reference Professor: Mirandola Raffaela
Release Date February 2016

POLITECNICO
DI MILANO

1. INDEX

2.Introduction

2.1 Purpose
In this document we focused on the Structural Design and on how the interaction between our software and the users (both Taxi Driver and Customer) has to be. The aim of this document is to clearly explain the architectural design of the application, alongside all the justification for our choices in the matter.

2.2 Scope
The scope of the TaxiService project is to manage, design, build, and implement a service aimed at facilitating public transportation via taxi. The service will be available by means of either mobile or desktop applications, and will allow customers to request or reserve a ride to and from any two addresses inside the city boundaries. It will also allow drivers to manage their requests, and will automatically form taxi queues in order to provide a fair management of the passengers' load. The scope of this project includes all the requirement gathering, planning, designing of the service. The R.A.S. Document is complementary to this in delineating the various aspects of the project.

2.3 Definitions, Acronyms, Abbreviations
- RASD.: Requirement Analysis Specification Document. The Document complementary to this one in describing the service.
- DD: Design Document. This document is the DD.
- JEE: Java Enterprise Edition. The Java platform to implement this service with.
- DBMS: Database Management System.
- COTS: Commercial-Off-The-Shelf. A third party component of any kind used inside a proprietary project.
- SOA: Service-Oriented Architectural Style.
- OO: Object Oriented. A type of architecture which considers each separated entity as an object, focusing on their attributes and the way each objects interacts between each other.

2.4 Document Reference
The following documents had been used for the creation of this Design Document:

- JEE Documentation
- Microsoft Developer Network Guidances and Documentation
- R.A.S.D.

3.Architectural Design

3.1 Overview

3.2 High level Components and their Interactions

TaxiService is strongly based on Java Enterprise Edition and MySQL for its dedicated database DBMS. JEE has a four tiered architecture divided as:

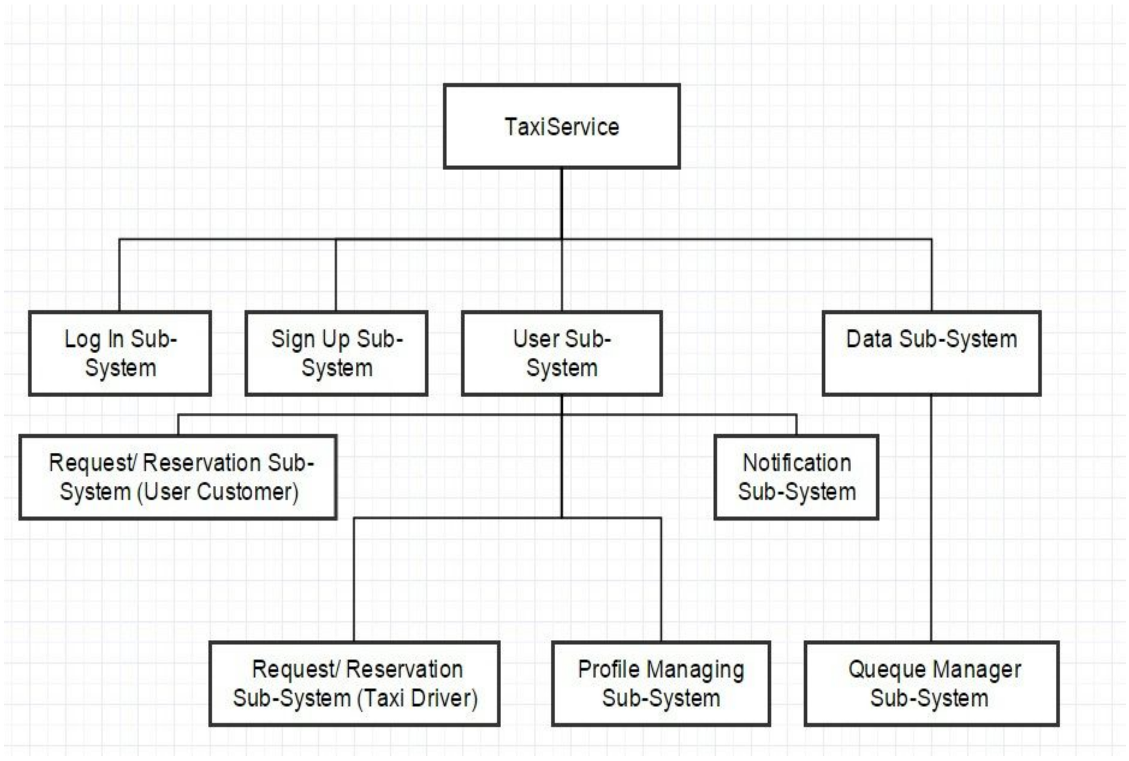
- Client
- Web
- Business Logic
- Dedicated Database

The first tier, the Client, singlehandedly controls all the application’s User Interfaces via our Web Browser/Mobile Browser managing the Java bean via J2EE7 and JSF pages, and it is also the layer that directly interacts with users. The Web Tier has the specific duty to manages the interactions between the Business Logic and the Client; it also contains all the Servlets and Dynamic Web Pages that needs to be elaborate in any instances. The Business Logic contains and controls all the TaxiService’s business logic, which also contains Enterprise Java Beans and the JPE (Java Persistence Entities). Finally, the Dedicated Database contains all the data source and manages the operations recovery, uploading and writing.

3.2.1 Subsystem Structure

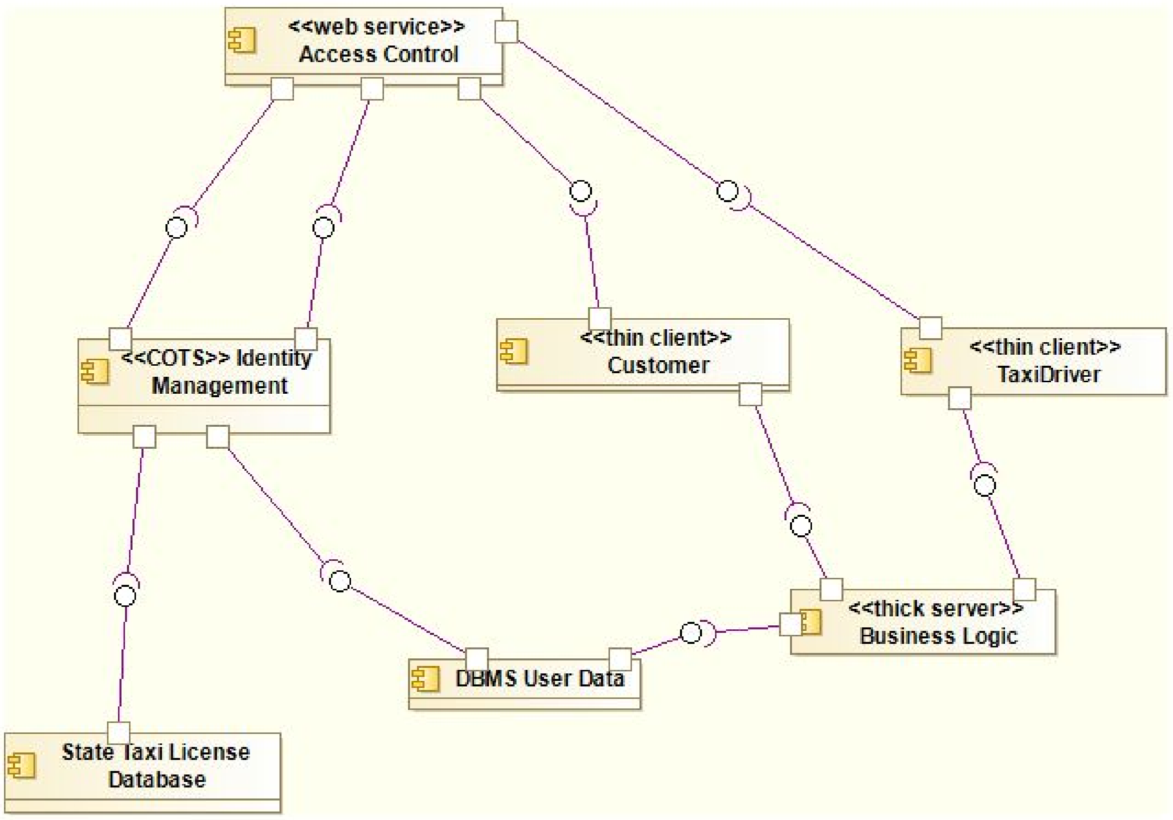
TaxiService hierarchical system structure can be decomposed in many sub-systems to represent the functionalities of our software and to make them clearer. We separate our systems into these sub-systems:

- Sign Up Sub-System
- Log In Sub-System
- User Sub-System
  - Profile Managing Sub-System
  - Notification Sub-System
  - Request/Reservation Sub-System (TaxiDriver)
  - Request/Reservation Sub-System (UserCustomer)
- Data Sub-System
  - Taxi Queue Management Sub-System


TaxiService
Log In Sub-System
Sign Up Sub-System
User Sub-System
Data Sub-System
Request/ Reservation Sub-System (User Customer)
Notification Sub-System
Request/ Reservation Sub-System (Taxi Driver)
Profile Managing Sub-System
Queque Manager Sub-System

3.3. Component View

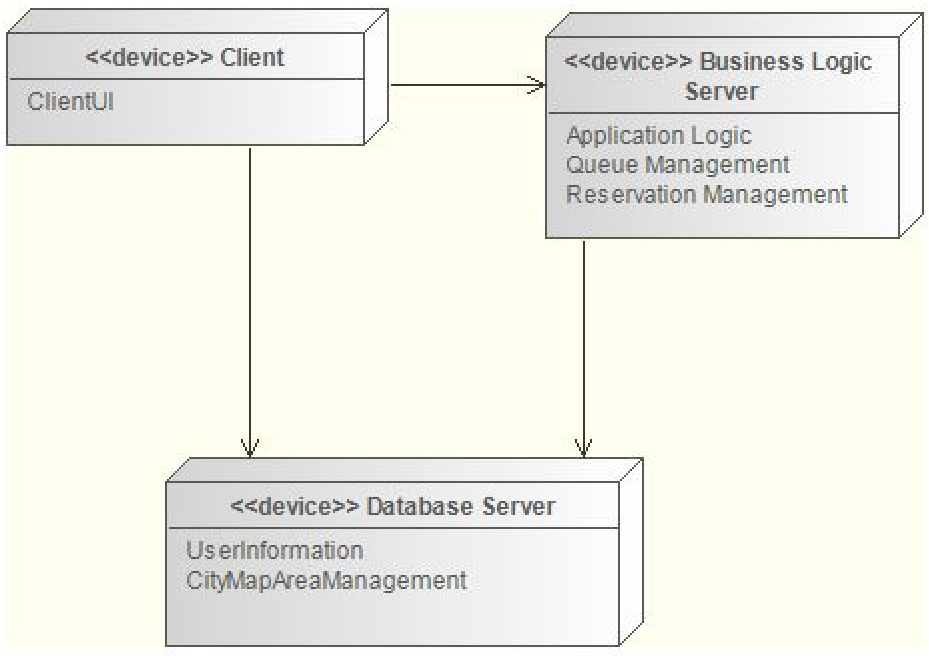
We based the architecture on a thin client/thick server logic. Separated clients in case of access by a Customer or a TaxiDriver communicate using a Business Logic server, managing the messages and responses between them and handling all the application logic. It also constitutes part of the Controller in a MVC pattern approach, whereas the clients represent the View. The Model is instead represented by the DBMS, for a database containing all the users' information that the system acquired via registration, and it uses it to authorize and authenticate the users in concurrence with the Access Control service. It also stores all the information related to the city's map, location and areas used in our software. The Access Control also interacts with a legacy database of taxi licenses in order to verify the registration information submitted by a TaxiDriver-to-be. We thought the Identity Management, handling all the access verifications and logic, to be COTS (for example, services by third party providers such as PortalGuard, Access Manager, or Okta).



<<web service>> Access Control
<<COTS>> Identity Management
<<thin client>> Customer
<<thin client>> TaxiDriver
<<thick server>> Business Logic
DBMS User Data
State Taxi License Database

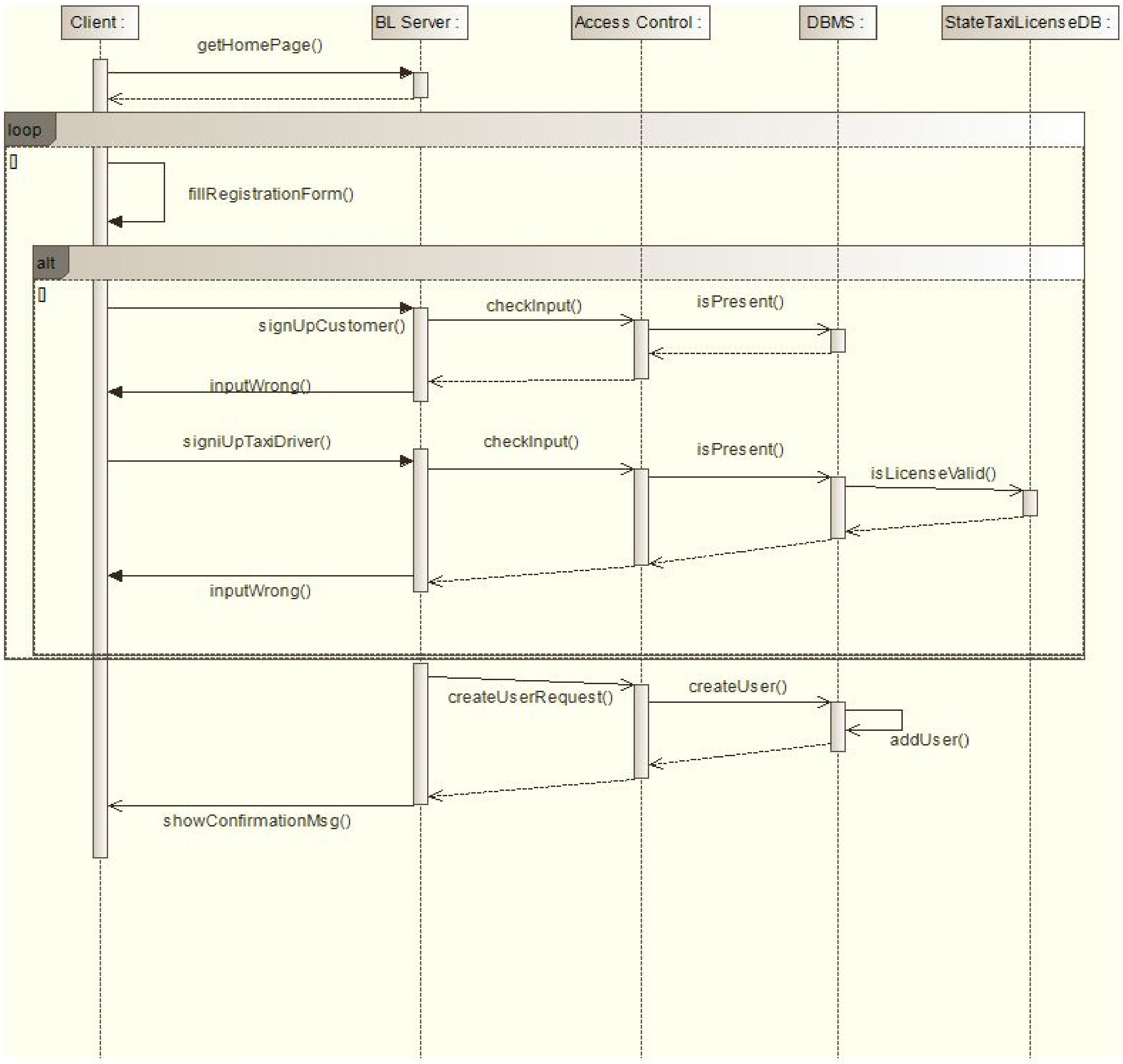
3.4 Deployment View

Ther services are mainly concentrated on the DBMS server cluster. This is because, since it is also a mobile application, we want TaxiService's clients to be as light as possible, in order to diminish loading time and memory occupation. The Business Logic Server manages all the messages exchanged between different clients and accesses the DBMS when needed.

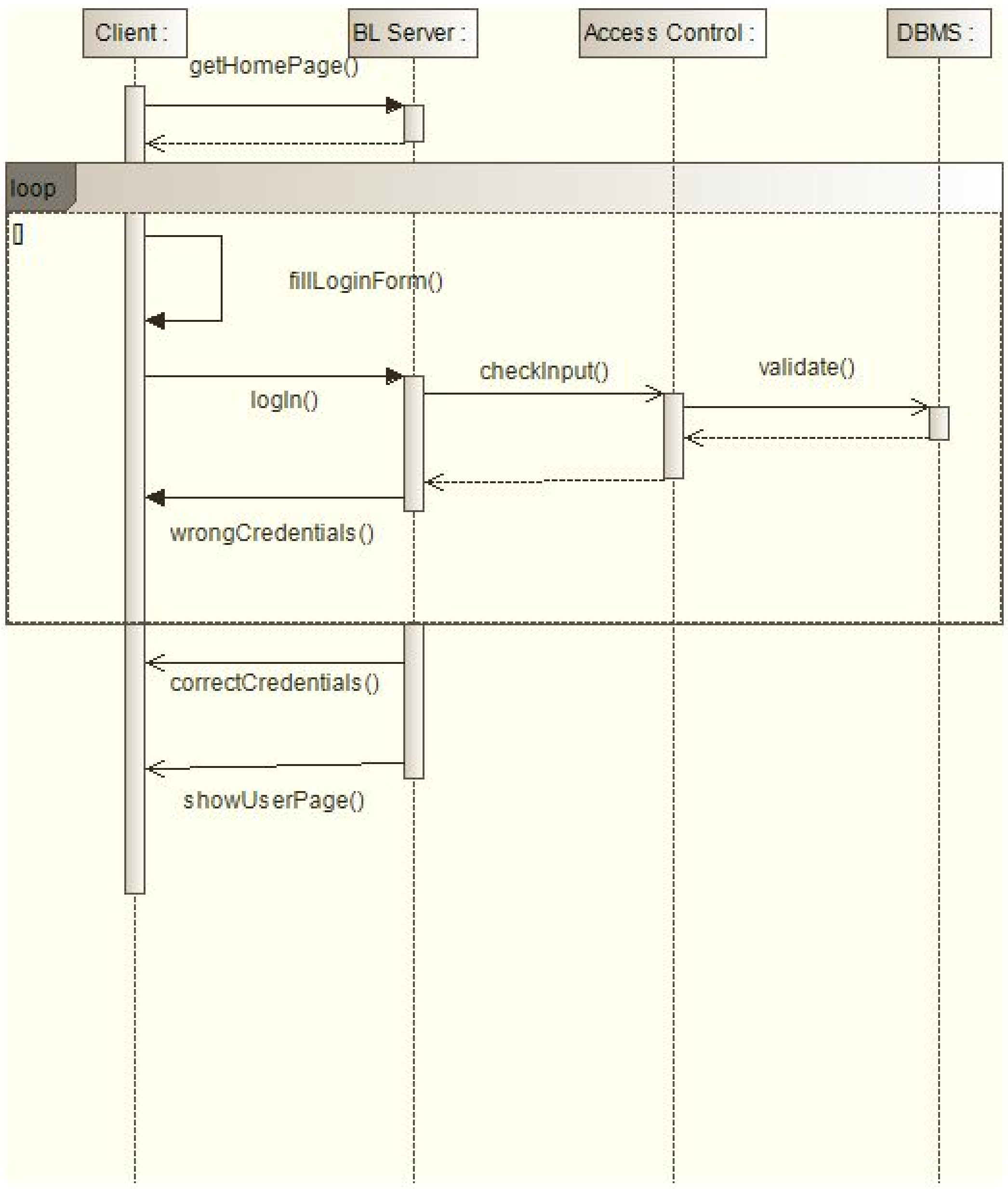


3.5 Runtime view

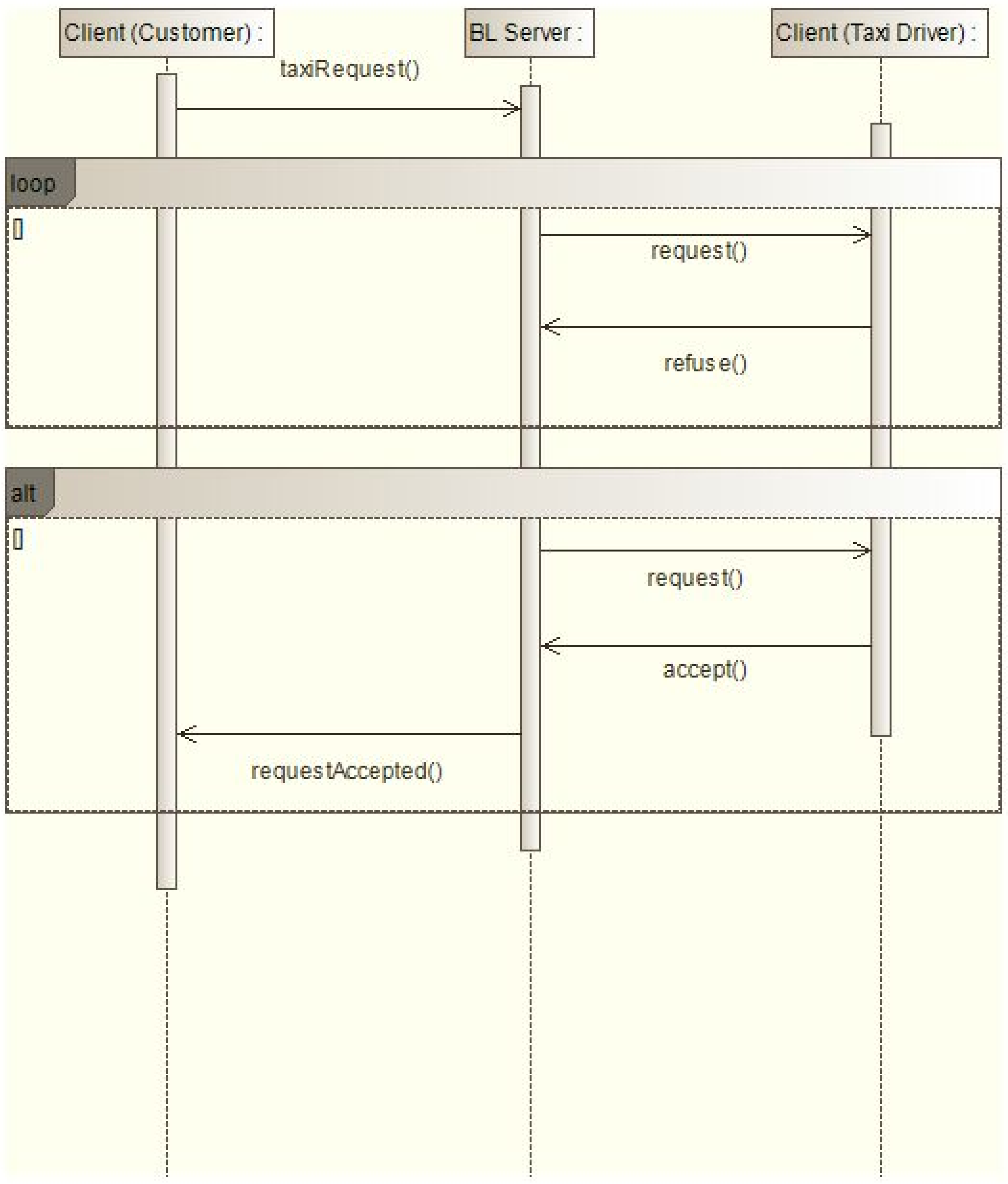
This sequence diagrams explains how components interact during a Registration activity. The Sign Up type (UserCustomer/TaxiDriver) is specified in the fillRegistrationForm() function.



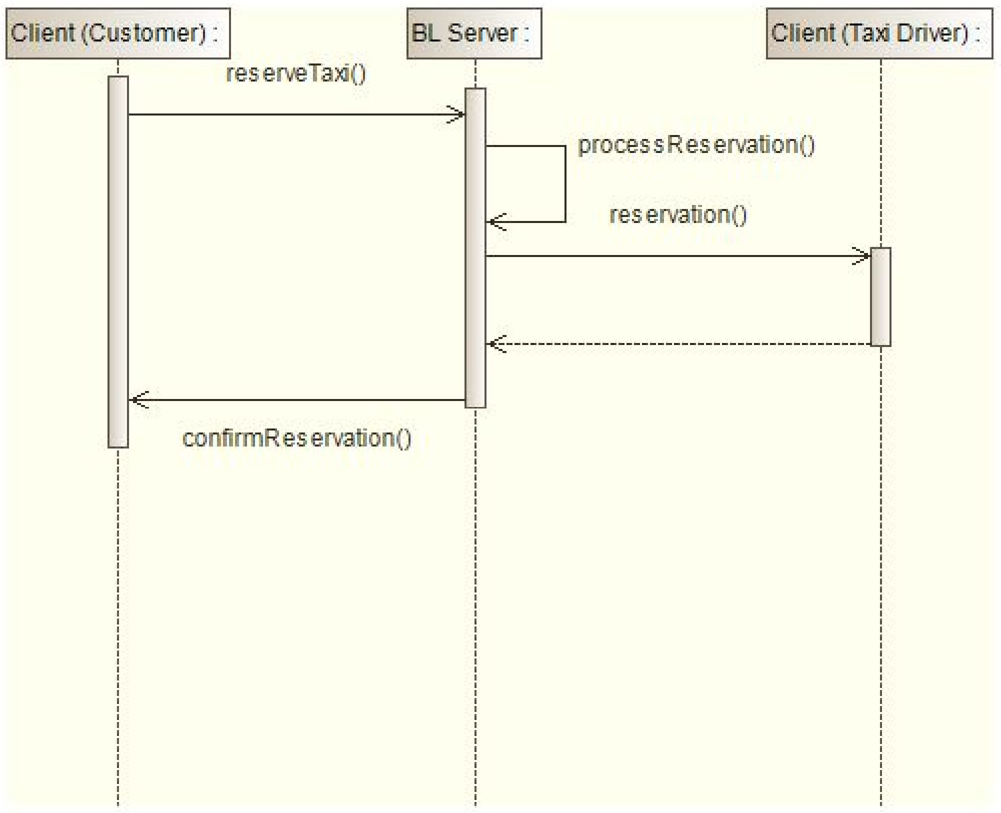
This sequence diagrams explains how components interact during a Log In activity. The Log In tipe (UserCustomer/TaxiDriver) is specified in the fillLogInForm() function.



This sequence diagrams explains how components interact during a Taxi Request activity. All the location data and the Taxi Driver’s licence plate number are specified in the request() function and in the last accept() function respectively.



This sequence diagrams explains how components interact during a Taxi Reservation activity. All the location data and the Taxi Driver's licence plate number are specified in the reserveTaxi() function and in the last confirmReservation() function respectively.



3.6. Component Interfaces

3.6.1 Customer Client/Taxi Driver Client

The client contains a UI Generator as application component (the UI is generated distinctively, depending upon the type of Customer), which receives and processes all allowed input and displays messages. The UI connects to the interface with the Access Control where an input calls for the sign up or login functions. It also connects with the Business Logic Server interfaces, either when a Customer requests or reserves a ride, or when a Taxi Driver responds to a request or updates his status.

3.6.2 Business Logic Server

The business logic server contains a Message Management component, which processes and manages the whole message flow between a Customer and his/her Taxi Driver, ensuring security via encryption.
It also interfaces with the DBMS server cluster in order to retrieve the map information, and to generate the queues based upon the information of each Taxi of the current City Area, using the Taxi-Enqueueing Algorithm in a sub-component of its own.

3.6.3 Access Control

The Access Control service interacts with the Clients during the sign up and login functions. It needs to interface with a third party software, a off-the-shelf Identity Management platform, as aid during the verification and validation processes.

3.6.4 DBMS

It contains a B+ Tree active database, with an 3-key index ordering method. Detached type triggers are mostly used in this system, for a better efficiency in the management of variations of stock indices values after several exchanges. It has to guarantee all the interfaces for the Identity Manager, which is a COTS component.
It is structured as a cluster, in order to achieve a moderate amount of information redundancy and prevent data loss: all the servers in the cluster must then provide each other with interfaces in order to guarantee a safe and sound DB structure.

3.7. Selected Architectural Styles and Patterns

We have chosen an OO Client/Server architectural style, characterized by a thin client to be installed as application by the customer and a central thick Server. We have considered the Client/Server architectural style because our application is server based, since it will support many clients and will have many business processes that will be used by people throughout the city. The ease of the maintenance, the high security and the centralized data access have been fundamental elements that made us make this choice.

We also adopted some characteristics of the SOA (Service-Oriented Architectural Style) because it guarantees benefits in terms of discoverability, interoperability and domain alignment, which allows us to reuse all the common services with standard interfaces (such as the GPS service or Google Maps) and it increases business and technology opportunities and reduces cost.

4.0 Algorithm Design

A critical aspect of the TaxiService application is a fair management of the passenger load throughout all the taxis.
In order to do so, we opted for a dynamic queue generation: each time a call (be it a request or a reservation) is made, in order to contact the most suitable vehicle, the server analyzes the list of TaxiDrivers in the correspondent City Area. It then proceeds to assign to each driver a value, based on the effective road distance that such driver must travel before reaching the customer location. Then, all the drivers are organized in the queue in ascending order, from the closest to the farthest.
In this way, we ensure that no driver will ever be obliged to run across the entire area, if someone closer is available. This also ensures that a driver, having changed City Area, is not automatically last in a queue where he could potentially be the closest, the best choice to make.

More in detail, the value is assigned to each driver by calculating the minimum costing path from his/her vehicle to the customer's location, applying a Dijkstra's shortest path algorithm. Each road is seen as an arc, each intersection as a node, starting from the node represented by the taxi's location and ending at the customer's location. The cost of each road is determined by its length in meters. Optionally a road can have an artificially increased cost, maybe even $+\infty$, in case of blocked roads, rush hour traffic, etc.
After determining the cost of the shortest path for each driver, the system puts all the drivers in an ordered queue, where the first is the one with the minimum cost (i.e. the nearest to the customer), and all the others in ascending order. The call is then forwarded to the first taxi in the queue, which can then accept it or refuse it. In case of refusal, the call is forwarded to the second, and so on. A taxi that refuses a call is removed from the queue.
Here a pseudocode summary of the algorithm:

Taxi-Enqueuing Algorithm

```
BEGIN
/* initialization */
Q:=; S:=; T:=locCustomer; A:= false;
        /* computing shortest paths for available taxis in the area */
        FOR taxi IN currentAreaTaxiList[] WHERE taxi.status = available;
        taxi.cost = dijkstra(Starting = taxi.location, Goal = T);
        Q.tail = taxi;
        END FOR
/* sorting the queue and assigning the call to the first */
bubblesort(Q);
        /* waiting for call acceptance */
        WHILE !A
        S = Q.head;
                IF forward(S) THEN A = true;
                ELSE Q.head = Q.head+1;
                IF Q.head = Q.tail THEN refused();
        END WHILE
END
```

5.0 User Interface Description

All the details about User Interfaces description and functionalities can be found in the R.A.S.D. document of TaxiService attached to this one. It is important to say that all the graphic interfaces, especially the ones concerning the mobile application, has to be as light as possible in order to guarantee a good performance, low memory cost and short download time. The mobile interface will be implemented to be functional with touch-screen devices, and kept as simple and sober as possible to ease its usability by people not used to mobile technology.

6.0 Requirement Traceability

6.1 Registration functionality

(The system has to provide the capability for a Guest to register his data into the dedicated Database as a User (Customer) or a User (Taxi Driver).)

In order to fulfill this requirement we designed the architecture considering a DBMS able to manage and store all the users' data, for registration, validation and verification purposes. Using a client/server pattern allows us to be more at ease on the security side, plus the database will implement state-of-the-art protection systems to prevent data stealing.

6.2 Login functionalities.

(A User (either Customer or TaxiDriver) has to perform the Login action. The Login gives access to different services based on the type of User.)

The Login functionalities are also supported by the architecture design, using a combination of the DBMS and an Access Control application, possibly off-the-shelf, in order to ensure a secure login and prevent as much as possible hacking.

6.3 Taxi Request.

(The system has to provide a function that allows Users (Customer) to request a taxi through the TaxiService application.
The system will then have to inform the passenger about the license plate of the incoming taxi and waiting time.
The system has to provide a function that allows a user to visualize his pending request and the ability to cancel it before the taxi arrives.)

The client/server architecture allows for a lightweight access, especially for mobile usage, so as to make requesting a taxi easier and faster. The Taxi-Enqueueing algorithm allows us to manage taxis, making it so that the closer available taxis are the first ones to get the call forwarded to them.

6.4 Taxi Status Tracking.

(The system has to provide Taxi Drivers a mobile application to inform the system about their availability and to confirm that they are going to take care of a certain call.
The System should also provide the traceability of every Taxi Vehicle via GPS.)

This important detail has been further explained by the adoption of the S.O.A. architectural style, which guarantees service-wise interoperability between different application and therefore can synchronize our software with the Taxi's GPS devices.

6.5 Area-based Taxi queue Management.

(The system should optimize waiting time through the correct management of taxi queues by dividing the city in taxi zones (approximately 2 km squared each) and by associating to each zone a queue of taxis.)

This requirement have been further specified by the creation of the pseudocode concerning the Taxi-Enqueuing Algorithm reported in the Algorithmic Design chapter. It guarantees a correct and CPU-unintensive management of multiple lists of taxi which are then analyzed to search that single vehicle that suits the customer position better. This is explained further in the Flowgraph presented in the RASD document.

6.6 Taxi Reservation;

(The system has to provide the capability to reserve a Taxi, to be in a certain place at a certain hour. After a reservation has been made, the system has to forward that request ten minutes before the selected time.)

We have implemented new sequence diagrams to further explain how our system components interact whenever a new reservation has been made. All reservation are treated as normal call by our Taxi-Enqueuing Algorithm just like all the other requests, but they cannot be refused and are forwarded fifteen minutes before the time inserted by the customer.

7. Reference

Microsoft MDSN documentation

- Chapter 3: Architecturals Patterns and Styles
- Chapter 24: Designing Mobile Applications

TaxiService R.A.D.S.

Windows API