Design Document

Michele Madaschi Lidia Moioli Luca Martinazzi

November 27, 2015

Contents

Introduction

1.1 Purpose

In this document we aim to provide a description for the architecture and design of MyTaxiService. This document is targeted towards the future developers of the system.

1.2 Scope

The application will be developed using a client-server paradigm. The server-side application must recognize an user (either an unregistered guest, a passenger, a taxi driver or an administrator), and accordingly signal the client the available actions.
The server-side application must manage the city-wide taxi deployment, by the means explained in the RASD document[1].
The client-side application must show a UI to which the users can interact.
The application must implement a report system, in order to incentive the good behavior of the users involved.

1.3 Definition, Acronymus, Abbreviation

1.4 Reference Documents

1.5 Document Structure

[1] see reference documents

Architectural Design

2.1 Overview

The distributed application is composed by a server side, and a client one. The client side interacts with users (or guests), showing the correct activity, and sends requests to the server side, when needed. The client side must be able to interact with the GPS. The server side manages requests coming from the client side, and notifies the users (or the guests) involved in the requests. The server side must also interact with a map system, in order to retrieve information about the route and the length of the ride.

2.2 High level components and their interaction

The client side is composed by a set of activities composed by one or more actions and displayed through the user interface. The client side has also an interface which manages the interaction with the server. The server side has a controller for each connected client, that manages the requests coming from the users (or guests), taking data from the ride manager. It also sends messages to the clients in order to resolve the requests. The controller interacts with the clients through a network interface.

2.3 Component view

2.3.1 Client

Activity : an activity is a set of messages and actions, that the application must display to the users. No more than one activity can be displayed at the same time; The default activity is the "guest home" activity. Each time the user taps a button, the application must execute the related action, and select the next activity.

Action : an action is something that a user can do, in order to interact with the application. If an action needs some data, the userinterface must display a field, for each input needed, that allows the human to provide the necessary incormations. Some actions can also select the next

activity that must be shown, or/and send informations to the server, in order to complete their job.

Userinterface : is the component that directly interacts with the human. It contains the activity that must be displayed, and read the components of the activity, in order to display them. It also launches the actions selected by the human.

Clientnetworkinterface : is the component that allows the others to exchange messages with the server side.

2.3.2 Server

Controller : we have one controller for each client connected; initially it will be a guest controller, which allows login/register. After a successful login, the guest controller will create a taxi driver/passenger controller, and substitutes himself with the new controller. The controller must manage requests coming from the client, taking informations from the ridesmanager, and eventually adding new objects to it. In order to sends and receives message from the client, the controller communicates with a Servernetworkinterface. The controller also contains a User object, that allows him to retrieves information about the specific logged user.

Ridesmanager : it is a singleton, that contains informations about the queues, the active rides. This corresponds to the model in the MVC pattern and it is shared by all the controllers.

User : is the component that contains every possible information about a logged user, taken from the database. User is a part of the controller (obviously only in case of passenger/taxi driver controller). User is split in taxi driver and passenger, cause there are some differences between the two kind of user. Also user is a part of the model, but it is not shared.

Ride : this class allows to create object, containing all information about a specific ride. As soon as a passengers make a taxi requests, the controller create a rides containing information recevied from the client. In case of shared ride, a ride object will be created and added to the corresponding sharedride ones.

Sharedride : the sharedride object contains the set of the ride object, the total cost of the ride and the route. When a new passenger is added to the shared ride, the controller will adjust the path, the total cost, and the cost of each rides object, contained in the current sharedride.

Servernetworkinterface : is the component that allows the interaction with the client side. It converts messages coming from the other server component, in messages readable from the Clientnetwrokinterface, and vice versa.

2.3.3 Master view

2.4 Runtime view

2.5 Component interfaces

The passenger controller class must interfaces with Google map sevice, using its API, in order to get the shortest path and the chilometers of the ride.

2.6 Selected architectural styles and patterns

The application follows the MVC (Model-View-Controller) pattern. The model is entirely contained in the server side, and it's composed by a singleton class (Ridesmanager), that contains informations shared by all the controllers, and a user class (User), that contains informations about a specific user, and is accessible only by the controller assigned to the corresponding user. Furthermore, the view is completely contained in the client side; it is composed by the " userinterface" class, that must display the activity's objects. The controller is mainly represented by the homonym class, in the server side. There are few controller's functions , that don't require data contained in the server, implemented in the action class.

2.7 Other design decisions

The server side structure is composed as shown in the picture below: The load balancer is needed to equally divides requests coming from different clients, in order to increase performances. Different application servers must be placed in different physical zones, in order to react to possibly partial blackout and other local physical accidents. Client side there's only one constraint, the obbligation, for the taxi driver, to use a divece with a working GPS.

Algorithm Design

3.1 Outsourced algorithms

3.2 In-house developed algorithms

3.2.1 Shared ride

The following pseudocode describes how share taxi request should be handled

```
function sharedRequest(passenger)
    shared ← findSharedRideAvailable(fromZone, toZone, timeout)
    if exists shared then
        shared.addReservation()
    else
        taxi ← getAvailableTaxi(fromZone)
        if exists taxi then
            create new shared ride
        else
            error message
        end if
    end if
end function
```

The following function (dependency of *sharedRequest*), describes how the system should search for an available shared ride, compatible with the constraints provided via arguments

```
function findSharedRideAvailable(fromZone, toZone, timeout)
    for all ride in Rides do
        if ride is shared then
            if ride.fromZone == fromZone AND ride.toZone == toZone
then
                if not ride.isFull() AND not ride.isReserved() then
                    if now() − ride.allocationTime < timeout then
                        return ride
                    end if
```

```
            end if
        end if
    end if
  end for
  return empty set
end function
```

The following function (dependency of *sharedRequest*), describes how the system should fetch an available taxi. Keep in mind that the process of issuing the call to a driver and managing his/her response happens inside this function (or its sub-routines)

```
function getAvailableTaxi(fromZone)
  for all queue in Queues do
    if queue.getZone()==fromZone AND not queue.isEmpty() then
      taxi ← queue.getFirstTaxi()
      if taxi.sendRequest() then return taxi
      else
        manage queue and retry
      end if
    end if
  end for
end function
```

3.2.2 Billing calculation

The following formula describes how the system should calculate the amount of money each passenger has to pay after a shared ride (to put it simply, how to "split the bill")

B_i = Amount paid by the i-th passenger

D_i = Distance traveled by the i-th passenger

D = Total traveled distance

C = Total amount calculated by the taximeter

$$B_i = \frac{D_i}{D} * C|_{\text{rounded to the nearest 0.1}}$$

$$T = \sum_{i=0}^{n} B_i = \text{Total cost, calculated as the sum of the n partial costs}$$

User Interface Design

Requirements Traceability

References