Politecnico di Milano

Software Engineering II

---

myTaxiService
Design Document

Authors
Giovanni Beri 852984
Biagio Festa 841988

Thursday 3rd December, 2015

Contents

1 Introduction

1.1 Purpose

Our team will project myTaxiService, which is a platform that will be used by the administration of a large city to manage the system of the taxi drivers in the city area. In particular myTaxiService will offer to the users the possibility to effectuate requests of a taxi ride using a web application or a mobile app, and it will also manage the choice of the taxi to assign to every user's request using a system of queues. Advanced features will also allow users to make a reservation of a ride, specifying origin, destination and picking up time of the ride.

This document aims to:

- describe the software and hardware architectural choices we made during the design phase of emphmyTaxiService
- analyze more in depth some algorithms sketched in *myTaxiService* RASD
- describe the user experience in *myTaxiService* applications
- show how the the taken decisions are coherent with the requirements idientified in *myTaxiService* RASD

This document is intended to be examinated by the team of developers that will implement our system, to make the implementation consistent with the system project.

1.2 Problem Description

The government of a large city aims at optimizing its taxi service. In particular, it wants to: i) simplify the access of passengers to the service, and ii) guarantee a fair management of taxi queues.

Passengers can request a taxi either through a web application or a mobile app. The system answers to the request by informing the passenger about the code of the incoming taxi and the waiting time.

Taxi drivers use 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 guarantees a fair management of taxi queues. In particular, the city is divided in taxi zones (approximately $2km^2$ each). Each zone is associated to a queue of taxis. The system automatically computes the distribution of taxis in the various zones based on the GPS information it receives from each taxi. When a taxi is available, its identifier is stored in the queue of taxis in the corresponding zone.

When a request arrives from a certain zone, the system forwards it to the first taxi queuing in that zone. If the taxi confirms, then the system will send a confirmation to the passenger. If not, then the system will forward the request to the second in the queue and will, at the same time, move the first taxi in the last position in the queue.

Besides the specific user interfaces for passengers and taxi drivers, the system offers also programmatic interfaces to enable the development of additional services (e.g., taxi sharing) on top of the basic one.

A user can reserve a taxi by specifying the origin and the destination of the ride. The reservation has to occur at least two hours before the ride. In this case, the system confirms the reservation to the user and allocates a taxi to the request 10 minutes before the meeting time with the user.

1.3 Glossary

Software *or* System
: The collection and the parts of programmes that compose the project. Also we will indicate with these words the components finalized at the user (*documentation* and *user interface* application).

Actor
: An entity that interacts with the system. It can be either a physical user or a external component.

Taxi zones
: Geographical areas of approximately $2km^2$ in which the interested city is divided. Taxi zones are not overlapping, so each point of the city is assigned to exactly one taxi zone.

Ride request
: A request that is sent by a registered user to signal that he needs to be picked up by a taxi in the immediate future. The request includes the GPS coordinates of the position in which the registered user desires to be picked up.

Ride reservation
: A request that an user sends to book a taxi ride in the "future". For "future" we intend an instant of time that is almost 2 hours distant from the instant in which the request is sent. It must contain origin and destination of the ride and picking up time and date.

Fake request
: A Ride reservation or ride request that turns out to be not a real one, because the passenger who sent it did not show up in the indicated place (and, in the case of a ride reservation, at the indicated time and date)

ETA
: The *estimated time of arrival*. It can be used to generate estimated times of arrival depending on either a static timetable or through measurements on traffic intensity. [1]

API
: *Application Programming Interface*: is a set of routines, protocols, and tools for building software applications. [2]

[1] Wikipedia definition: `https://en.wikipedia.org/wiki/Estimated_time_of_arrival`

[2] Wikipedia definition: `https://en.wikipedia.org/wiki/Application_programming_interface`

1.4 Used Tools

The tools used in the creation of this RASD are:

Latex (Live 2013)
to redact and to typeset this document.

StarUML (2.0.0-beta 10)
to create UML Diagrams (*Use Cases, Activity, ...*).

Inkscape (0.91)
to support for the Scalable Vector Graphics (*SVG*).

Dropbox *and* Google Docs
to provide us a simple mechanism to share files and synchronize our work.

Pronto.io
to create mockups. (`https://proto.io/`).

2 Architectural Design

2.1 Overview

In this section we are giving an overall description of the architecture of *myTaxiService*. Using a *bottom-up* approach in the identification of the software components, so we first insert a low-level component view and then a high-level component view. In the deployment view we show how the components will be distributed in physical devices, while in the runtime view we'll show how the components interact with each other in some runtime cases. In both runtime and deployment view, the considered components are the low-level ones. In *section 2.8* [*Other Design Decisions*] we describe the structure of *myTaxiService Data Layer* and we make some considerations about the programming language used to implement the system.

2.2 Component View

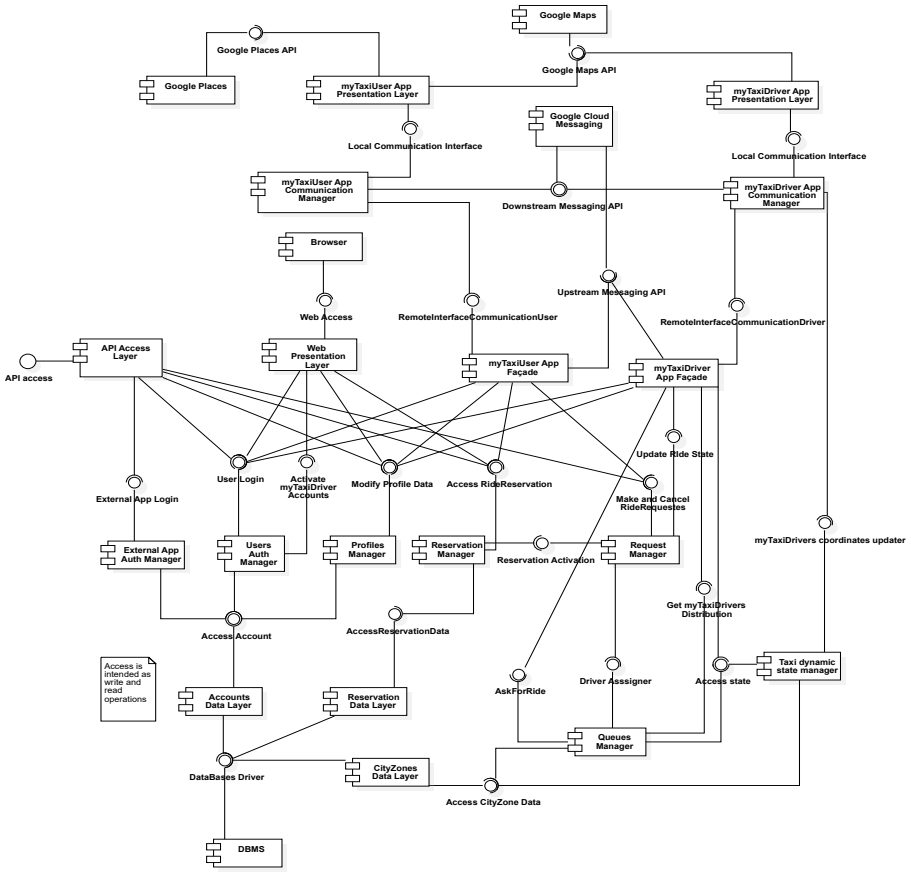


Figure 1: Component Diagram - Low Level

Google Maps API

It's an external Google component that provides *myTaxiService* system

some interfaces able to get data about map informations from the same database of Google Maps[3].

Google Places API
It's an external Google component that provides *myTaxiService* system some interfacesable to get data about the geographical location of point of interest from the same database of Google+ Local[4].

Google Cloud Messaging API
It's an external Google component that provides *myTaxiService* system some interfaces able to implement a system of *push notifications* (i. e. asynchronous messages from the server to the client). Every notification is sent by the *myTaxiService* server to the GCM servers, which carries out the management of its delivering to the client.

myTaxiUser App Presentation Layer
Offers to the user a GUI that:

- displays the data obtained by the interaction with *myTaxiUser App Façade*.
- allows *myTaxiUSers* to access the functionalities the system provides him.

myTaxiDriver App Presentation Layer
Offers to the user a GUI that:

- displays the data obtained by the interaction with *myTaxiDriver App Façade*.
- allows *myTaxiDrivers* to access the functionalities the system provides him.

myTaxiUser App Communication Manager
This component handles all the communications with *myTaxiService* servers that are needed to accomplish the *myTaxiUser App Presentation Layer* tasks. All the messages from *myTaxiUser* client app to the *myTaxiService* servers are sent from this component.

myTaxiDriver App Communication Manager
This component handles all the communications with *myTaxiService* servers that are needed to accomplish the *myTaxiDriver App Presentation Layer* tasks. All the messages from *myTaxDriver* client app to *myTaxiService* servers are sent from this component.

API Access Layer
It manages the access to the *myTaxiService* API, handling all the incoming requests. We decided not to give external applications access directly to the others components of the system for security reasons, so this component have at its disposal the interfaces of all the other components

[3] `https://www.google.com/intx/en_uk/work/mapsearth/products/mapsapi.html?utm_source=HouseAds&utm_medium=cpc&utm_campaign=2015-Geo-EMEA-LCS-GEO-MAB-DeveloperTargetedHouseAds&utm_content=Developers&gclid=CjwKEAiA7f-yBRDAgdv4jZ-78TwSJAA_WdMa-9y4K68HmUUiCcNbQ5NnUiVz11u7PVfvpitp4jknGxoCaUTw_wcB`

[4] `https://developers.google.com/places/`

(*External App Auth Manager*, *Users Auth Manager*, *Profiles Manager*, *Reservation Manager*, *Request Manager*) needed to take care of the incoming requests

Web Presentation Layer
It is a web server that accept HTTP messages from the users (*Administrator User* or *myTaxiUser*) who access *myTaxiService* via web application. This component interacts with all the components that permit

myTaxiUser App Façade
It's a component implementing a *Façade Pattern*: it's the only component that interacts with the *myTaxiUser App Presentation Layer* component, and provides it an interface for the functionalities provided to *myTaxiUsers*

myTaxiDriver App Façade
It's a component implementing a *Façade Pattern*: it's the only component that interacts with the *myTaxiDriver App Presentation Layer* component, and provides it an interface for the functionalities provided to *myTaxiDrivers*

External App Auth Manager
It supports all the functionalities concerning the authentication or registration of the app that has sent the request. To accomplish that, it can access the *Account Data Layer* component and retrieve informations in the database.

Users Auth Manager
It supports all the functionalities concerning the authentication or registration of the user (either a *myTaxiDriver* or *myTaxiDriver*) that is logging in. To accomplish that, It can access the Account Data Layer component and retrieve informations in the database.

Profiles Manager
This component is able to access the Account Data Layer component in order to update the information associated to the user accounts (e.g. Name, licence number, password)

Reservations Manager
This component is able to:

- access the *Reservation Data Layer* component in order to access the stored *RideReservations*
- activate a *RideReservation* 10 minutes before its expected time of arrival.

Request Manager
It handles all the incoming *RideRequest* by interacting with the *Queues Manager*; it manages all the *Ride Reqeust* since the creation, until:

- the state of the *RideRequest* is changed by the *myTaxiDriver*
- the *RideRequest* is rejected by the sytem because no *myTaxiDriver* can carry it out

Accounts Data Layer
It can access all the informations about the Accounts Data stored in the database.

CityZones Data Layer
It can access all the informations about the geographical distribution of the *City zones* stored in the database.

Reservations Data Layer
It can access all the informations about the *RideReservations* stored in the database.

Taxi Dynamic State Manager
It contains all the informations concerning the dynamic status of the *myTaxiDrivers*:

- State: available, busy, riding
- *city zone* in which the *myTaxiDriver* is queued
- priority level

Queue Manager
It contains the taxi queues information, and initialize the queues using the informations retrieved accessing the *CityZones Data Layer* component; the *Queue Manager* observes the Taxi Dynamic State Manager (through the *Observer* design pattern) in order to update the queues when it's necessary.

DBMS
It' an external component tha is able to interact with the database where all the system data are stored via query.

2.3 Runtime View

2.3.1 Ride Request *myTaxiUser* Side

The following (*figure 2*) sequence diagram shows us how the components interact among them when a ride request is caught by the user's interface.

In accordance with the specifications designed in the *RASD* document, we can see how in early passages the presentation layer determines the exact user's position (using *Google Services API*).

Once the position has been found, the application can initialize the communication with the remote *server*. The *client* app sends all data and the request through the *App Communication Manager* which waits for a response. Note that the user's app cannot perform another request until the previous one has been solved.

The *queue manager component* starts to search the first taxi driver available in according to the priority criteria. The taxi will be removed from the queue because whether he accepts or not he has to be moved into a not available state.

Whether a taxi has been found the *presentation layer* will be notified and it will displays a message for the user.

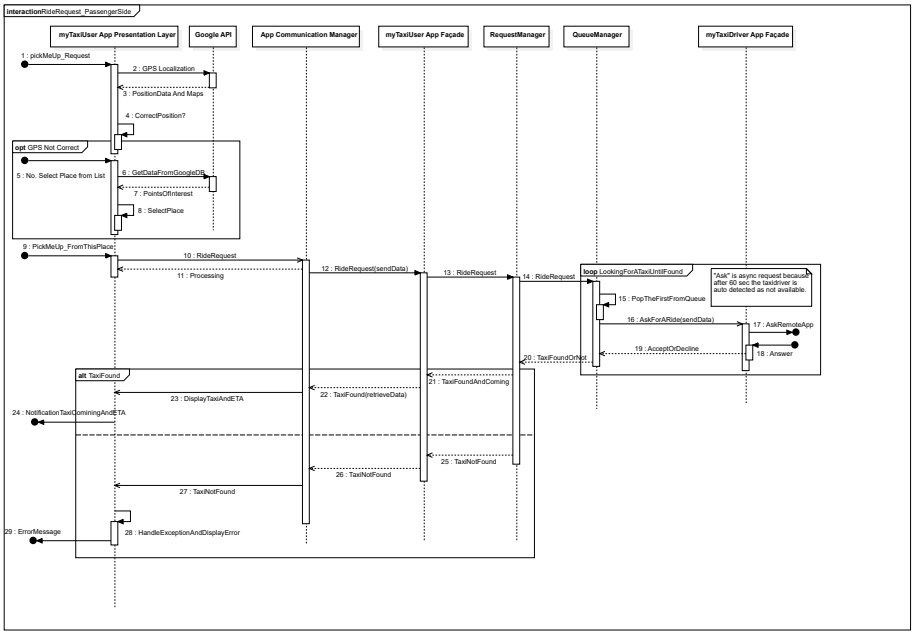


Figure 2: Sequence Diagram. Ride Request Passenger Side.

2.3.2 Ride Request *myTaxiDriver* Side

In this sequence diagram (*see figure 3*) we can see how the components work together in order to find a proper available taxi driver and to notify him for a ride request.

In short words the sequence shows a set of steps for the message exchange:

- The queue manager gets the first proper taxi driver available in accordance with priority policy and the city zone of the ride request.
- The request is send through the dedicated component, the latter send an asynchronous request via *Google Cloud Messaging* (an appropriate study on this mechanism will be formalized later in this document).
- When a taxi driver's answer is retrieved then the state of the taxi driver will be updated. Whether a positive answer has been retrieved on time the taxi driver will be set as "*riding*" and the request will be marked as "*satisfied*".

Note the *queue manager component* will send an *asynchronous* message and it will wait for a maximum amount of time. As we're going to see later, if the request is not received within that temporal window the *queue manager* will select another taxi driver and so on until the queue is empty. We remind you that the algorithm's policy provides a mechanism if a queue is empty: it will search in another queue. Whether all queues are empty, the queue manager has to raise an exception and notify the client for that.

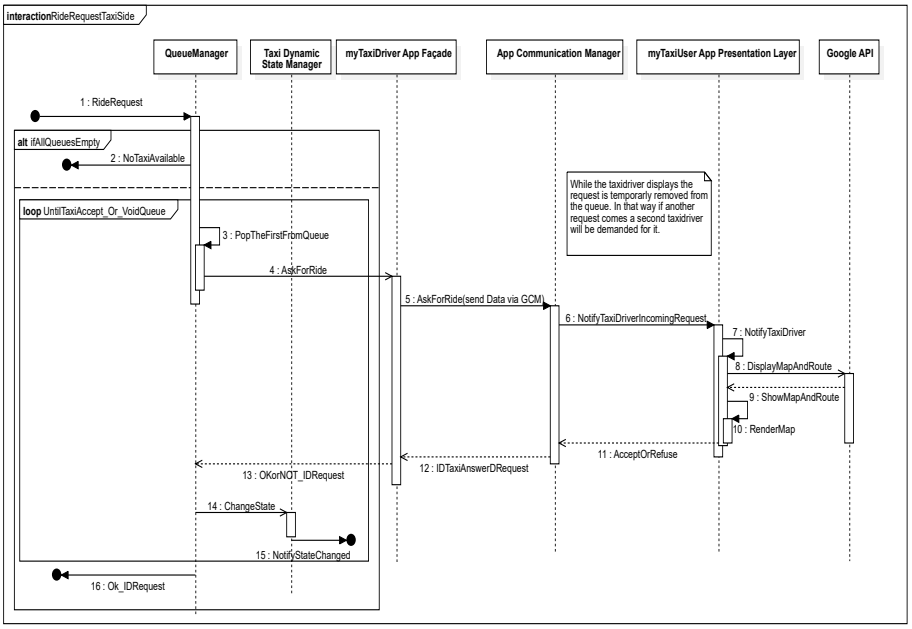


Figure 3: Sequence Diagram. Ride Request Taxi Driver Side.

2.3.3 Google Cloud Messaging and Push Notifications

In this other sequence diagram (*see figure 4*) we can see more in detail how the asynchronous messages are exchanged through *Google Cloud Messaging*.

When an incoming request has been caught, the *Queue Manager* looks for the available taxi driver with more priority. The request is forwarded to the *Google Cloud Messaging* Server and the *Queue Manager* waits at most 60 seconds for a positive answer. Whether an answer is not received within that time, the Queue Manager starts to find another appropriate taxi driver.

In that sequence diagram we can see a particular case in which the taxi driver's answer is retrieved too late (*e.g. the mobile client is out of signal for a while and delays the answer*). In that case the Queue Manager has already begun to look for another taxi driver and so the taxi driver state (who has answered late) will be changed as *busy* and a notification will be send back to the mobile app in order to notify that exceptional event. This because we want to satisfy the ride requests as soon as possible avoiding the creation of particular cases in which one taxi driver stalls the service.

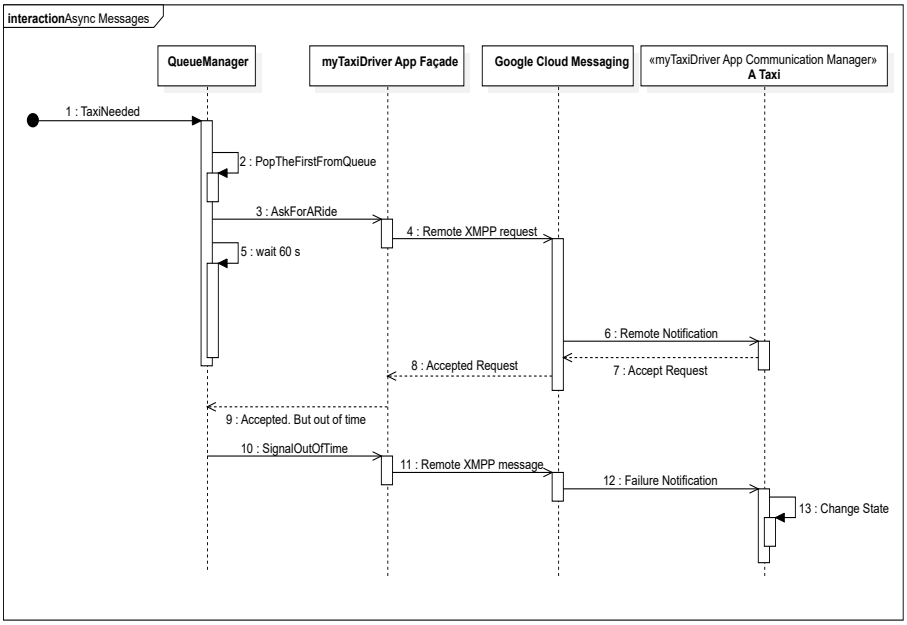


Figure 4: Sequence Diagram. Async Messages.

2.3.4 API Interaction from External Application

In this sequence diagram (see *figure 5*) we can see a particular case of communication between an external application (*developed by third party*) and our system via *API*.

In this case the external component (represented by "*External Application*") wants to update its user's profile information.

The external application opens a communication channel with the *API Access Layer* and sends an *key* in order to identify itself. A component, called *External App Auth Manager*, is dedicated to ensure the proper identification of all external component. The identification of every third party application is fundamental to guarantee a mechanism to control and management. In that way it could be possible to add new functionalities in further which allow to block or to limit requests from one application.

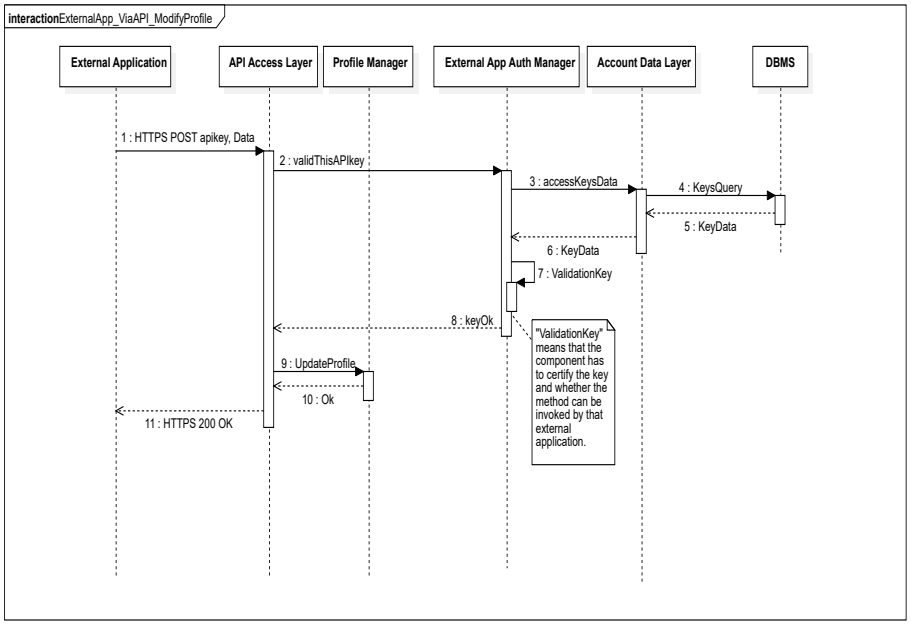


Figure 5: Sequence Diagram. External App, API interaction.

2.3.5 Update Taxi State

Another sequence diagram is shown in the *figure 6*. As we can see, it shows how the components work in order to perform an state update.

The main component which we want to focus on is *Drivers Dynamic State*. That component is responsible for keep tracking of all information about every taxi driver. In the diagram there is an initialization phase in which that component retrieves all geographical information about the *City Zones*. From now these informations are loaded in memory and can be used.

An environmental event happens (e.g. the taxi drivers moves and changes his gps coordinates), it is caught and forwarded to the *Dynamic State Component*.

The *Dynamic State Component* can identify the zone in which the taxi driver finds himself as it can convert the driver's coordinates into a *city zone* in accordance with the geographical informations loaded previously.

In event the taxi driver city zone is changed, through an *observed pattern* the *Queue Manager* is notified so it can update and refresh the queues.

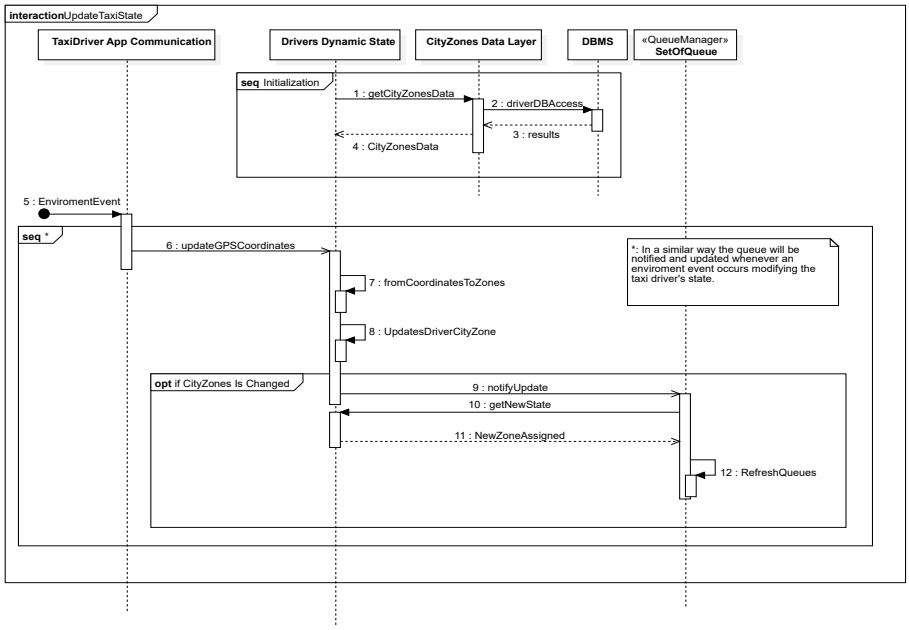


Figure 6: Sequence Diagram. Update a taxi state.

2.4 Component Interfaces

For every component listed at paragraph 2.2, we are now going to give a description of the interfaces they offer to other components; we also describe how those interfaces are used by the components that use them.

Google Maps

The Interface **Google Maps API** provides:

- a graphical representation of geographical maps
- a service of address research
- an evaluation of the ETA of a trip

This interface is used by:

- *myTaxiUser App Presentation Layer* to:
  - embed graphical maps and allow the *myTaxiUser* to visualize the position detected by his GPS, or possibly manually insert an address as origin or destination of a ride.
- *myTaxiDriver App Presentation Layer* to:
  - embed graphical maps and allow the *myTaxiDriver* to visualize the picking up point of an incoming request
  - show the ETA to reach the picking up position of an incoming *Ride Request* or *RIde Reservation*
  - visualize a map of the city including the informations about the distribution of the taxis in the city

Google Places

The interface **Google Places API** provides:

- a service of retrieval of remarkable locations placed around a given position

This interface is used by:

- *myTaxiUser App Presentation Layer* to:
  - display a list of points of interest located near the a given position, to give *myTaxiUser* the possibility to choose one of them as origin of the ride.

Google Cloud Messaging

This component offer 2 different interfaces: The interface **Upstream Messaging API** provides:

- the possibility to delegate the dispatching of a notification to an user to the Google Cloud Messaging service

This interface is used by:

- *myTaxiUser App Façade*, in order to send push notification to *myTaxiUsers* (i.e. assignment of a taxi to a *Ride Request* or *Ride Reservation*)
- *myTaxiDriver App Façade*, in order to send push notification to *myTaxiDrivers* (i.e. incoming *RideRequest* or *Ride Reservation*)

The interface **Downstream Messaging API** provides:

- the possibility to get notifications that were earlier sent using the *Upstream Messagging API*

This interface is used by:

- *myTaxiUser Communication Manager*, to get push notifications sent by *myTaxiUser App Façade*
- *myTaxiDriver Communication Manager* to get push notifications sent by *myTaxiDriver App Façade*

myTaxiUser App Communication Manager

The interface **Local Communication Manager** provides:

- the possibility to build a message that must be sent to *myTaxiService* servers

This interface is used by:

- *myTaxiUser Presentation Layer*, in order to exchange informations with the server; all the synchronous messages sent by *myTaxiUser* (e. g. *Ride Reservation*, *Ride Request*, login) are sent using this interface

myTaxiDriver App Communication Manager

The interface **Local Communication Manager** provides:

- the possibility to build a message that must be sent to *myTaxiService* servers

This interface is used by:

- *myTaxiDriver Presentation Layer*, in order to exchange informations with the server; all the synchronous messages sent by *myTaxiDriver* (e. g. login, change status) are sent using this interface

API Access Layer

The interface **API access** provides:

- access to all the functionalities the system offers to the external application accessing *myTaxiService* by API, i.e. all the functionalities offered to *myTaxiUsers*

This interface is used by:

- external applications, that are going to access the system by sending HTTP messages to *API Access Layer* component containing the data of the request.

Web Presentation Layer

The interface **Web Access** provides:

- access to all of the functionalities offered via web application to *Administrator User* and *myTaxiUser*.

This interface is used by:

- *myTaxiUsers* or *Administrator Users*, to access *myTaxiService* using a web browser.

myTaxiUser App Façade

The interface **Interface Communication User** provides:

- access to all the functionalities offered to *myTaxiUser* via proprietary protocols

This interface is used by:

- *myTaxiUser App Communication Manager* component to: send and cancel *Ride REqeust* or *Ride Reservations* and modify account data.

myTaxiDriver App Façade

The interface **Remote Interface Communication Driver** provides:

- access to all the functionalities offered to *myTaxiDriver* via proprietary protocols

This interface is used by:

- *myTaxiDriver App Communication Manager* component to: manage incoming *ride requests* or *ride reservations*, cancel already accepted rides, signal *fake requests*, confirm rides or change the driver status.

The interface **Ask For Ride** provides:

- possibility to send a specific *myTaxiDriver* a *RideRequest* or *RideReservations* to take care of, and get back the response of the *myTaxiDriver*

This interface is used by:

- *Queues Manager* component to ask the selected *myTaxiDriver* to take care of a *RideRequest* or *RideReservations*

External App Auth Manager

The interface **External App Login** provides:

- the check of an APIkey which validity and permissions must be verified

This interface is used by:

- *API Access Layer*, to check if a message received on *API access* interface include a valid APIkey, and if the permissions of the external app are compatible with the content of the incoming message.

Users Auth Manager

The interface **User Login** provides:

- the check of the validity of a registered user (*myTaxiUser*, *myTaxiDriver* of *Registered User*) authentication data
- to insert and verify the compatibility with the system of the authentication data proposed by a new user

This interface is used by:

- *Api Access Layer* component to verify the correctness of the authentication data contained in an incoming API message
- *Web Presentation Layer* component component to verify the correctness of the authentication data contained in a message received by users that are accessing *myTaxiService* via web application
- *myTaxiUser App Façade* component to verify the correctness of the authentication data sent by a *myTaxiUser* app
- *myTaxiDriver App Façade* component to verify the correctness of the authentication data sent by a *myTaxiDriver* app

The interface **Activate myTaxiDriver accounts** provides:

- the possibility to get the unevaluated *myTaxiDriver* registration requests

This interface is used by:

- *Web Presentation Layer* to communicate *Administrator Users* the list of unevaluated *myTaxiDriver* registration requests and permit them to evaluate

Profiles Manager

The interface **Modify Profile Data** provides:

- the possibility to signal a change request of the profile data of an user (e.g. password, name, email address)

This interface is used by:

- *Api Access Layer* component to signal a change request of the profile data received by *myTAxiService* API
- *Web Presentation Layer* component to signal a change request received by an user accessing *myTaxiService* via web application
- *myTaxiUser App Façade* component to signal a change request sent using *myTaxiUser* app
- *myTaxiDriver App Façade* component to signal a change request sent using *myTaxiDriver* app

Reservations Manager

The interface **Access Ride Reservations** provides:

- the possibility to access the stored ride reservations (i. e. add a new *RideReservation*, modify or cancel an existing one)

This interface is used by:

- *API Access Layer* component to carry out a *Ride Reservation* operation (i. e. cancel or creation) received from *myTaxiService* API
- *Web Presentation Layer* component to carry out a *Ride Reservation* operation (i. e. cancel or creation) that has been sent from a *myTaxiUser* through the web application
- *myTaxiUser App Façade* component to carry out a *Ride Reservation* operation (i. e. cancel or creation) received from a *myTaxiUser* app

Request Manager

The interface **Reservation Activation** provides:

- the possibility to convert a Ride Reservation in a Ride Request, with the same ride parameters (origin, destination, picking up time)

This interface is used by:

- *Reservation Manager* component to convert every *RideReservation* in a *RideRequest* 10 minutes before the specified picking up time

The interface **Make and Cancel RideRequest** provides:

- the possibility to add a new *RideRequest* or to cancel an existing one

This interface is used by:

- *myTaxiUser App Façade* component to carry out a *RideRequest* operation (i. e. cancel or creation) received from a *myTaxiUser* app

- *API Access Layer* component to carry out a *RideRequest* operation (i. e. cancel or creation) received from *myTaxiService* API

The interface **Update Ride State** provides:

- the management of the status of the ride; every ride can be:
  - unassigned: a taxi hasn't been selected yet
  - assigned: a taxi driver has been selected, but he has not reached the passenger(s) yet
  - confirmed: the taxi driver has actually picked up the passenger
  - fake: the taxi driver assigned to the ride has signaled that the ride was actually a *Fake Request*
  - cancelled: the assigned taxi driver has canceled the ride

This interface is used by:

- *myTaxiDriver App Façade* component to carry out a change of the ride status signaled by a taxi driver using a *myTaxiDriver* app

Accounts Data Layer

The interface **Access Accounts** provides:

- access to the sections of *myTaxiService* database where the users accounts and authorized external applications data are stored

This interface is used by:

- *External App Auth Manager* component to access the database and carry out the authentication of an external app
- *Users Auth Manager* component to access the database and carry out the authentication of a *myTaxiUser*, *myTaxiDriver* or *Administrator User*
- *Users Auth Manager* component to access the database and create new *myTaxiUser*, *myTaxiDriver* or *Administrator User* accounts
- *Profile Manager* component to carry out a request of change of some profile data of a registered user

CityZones Data Layer

The interface **Access CityZone Data** provides:

- access to the sections of *myTaxiService* database where the *CityZones* data are stored
- a function that is able to link the GPS coordinates of a position to the *CityZone* it belongs to.

This interface is used by:

- *Queues Manager* component to initialize the queues at the set up of *myTaxiService* system
- *Taxi Dynamic State Manager* component to determine the *CityZone* in which every *myaxiDriver* finds himself

Reservations Data Layer

The interface **Modify Profile Data** provides:

- access to the sections of *myTaxiService* database where the *RideReservations* data are stored

This interface is used by:

- *Reservation Manager* component to store *RideReservations* that has just been sent, and to retrieve *RideReservations* that must be handled in 10 minutes.

Taxi Dynamic State Manager

The interface **Access State** provides:

- possibility to get or modify the status (i.e. state, priority, *CityZone*) of a *myTaxiUser*

This interface is used by:

- *myTaxiDriver App Façade* component to signal change of the state of a *myTaxiDriver*
- *Queues Manager* component to update the taxi queues everytime a taxi driver change its *Taxi Zone* or its state

The interface **myTaxiDrivers Coordianates Updater** provides:

- the possibility to send the GPS coordinates of the position of myTaxiDriver

This interface is used by:

- *myTaxiDriver App Communication Manager* the possibility to send the GPS coordinates (if they are changed) of the position of every myTaxiDriver every 10 seconds.

Queues Manager

The interface **Driver Assigner** provides:

- the possibility to get a taxi driver to assign to a ride

This interface is used by:

- *Request Manager* component to get a *myTaxiDriver* to assign to every *RideRequest*

The interface **Get myTaxiDrivers Distribution** provides:

- possibility to get information about the distribution of the *myTaxiDrivers* in the *CityZones*

This interface is used by:

- *myTaxiDriver App Façade* to get the distribution of the *myTaxiDrivers* with higher priority than the one who is asking for this information

DBMS

The interface **DataBase Driver** provides:

- access via SQL to the content of the database

This interface is used by:

- *Accounts Data Layer* component to access the data concerning the accounts of the various users of the system
- *Reservation Data Layer* component to access the data concerning the *RideReservations stored in the system*
- *CityZones Data Layer* component to access the data concerning the *CityZones* geographical distribution

2.5 High Level Components and Interactions

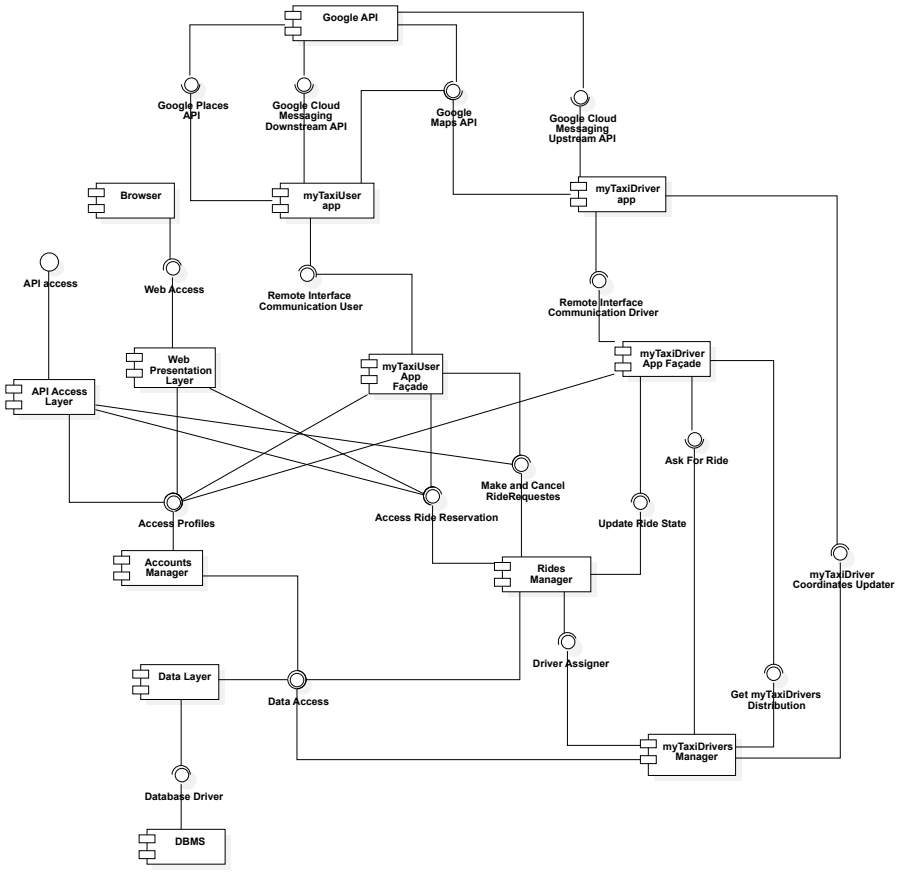


Figure 7: Component Diagram - High Level

We used a *bottom-up* strategy to indentify the components of *myTaxiService* system; so we built the Low Level Component View, and then we grouped some of its components or interfaces to obtain the High Level Component View. For

this reason in this section we are going to describe only the components and interfaces that differs from the Low Level Component View; for all not described components and interfaces holds the description in paragraph 2.2.

Components Description

Google API
It contains all the Google components which APIs are going to be used by *myTaxiService* system. It is the union of the following low-level components:

- *Google Maps*
- *Google Places*
- *Google Cloud Messaging*

myTaxiUser app
It contains the components constituting the presentation and communication layers of the *myTaxiUser* application. It is the union of the following low-level components:

- *myTaxiUser Presentation Layer*
- *myTaxiUser App Communication Manager*

myTaxiDriver app
It contains the components constituting the presentation and communication layers of the *myTaxiDriver* application. It is the union of the following low-level components:

- *myTaxiDriver Presentation Layer*
- *myTaxiDriver App Communication Manager*

Accounts Manager
It contains the components related to the management the profile of the users: login, authentication, modification of profile data. It is the union of the following low-level components:

- *External App Auth Manager*
- *Users Auth Manager*
- *Profiles Manager*

Data Layer
It contains all the component that interact with the DBMS to carry out interrogation on the database. It is the union of the following low-level components:

- *Accounts Data Layer*
- *Reservation Data Layer*
- *CityZones Data Layer*

Rides Manager

It contains the component dedicated to the creation, modification and cancellation of *RideRequests RideReservations*. It is the union of the following low-level components:

- *Requests Manager*
- *Reservations Manager*

myTaxiDrivers Manager

It contains the components dedicated to the management of *myTaxiDrivers*, their distribution in the queues and their assignment to the rides. It is the union of the following low-level components:

- *Queues Manager*
- *Taxi Dynamic State Management*

Interfaces Description

Access Profile

It groups together the interfaces dedicated the authentication and access to profile data of users and external applications:

- *External App Login*
- *User Login*
- *Modify Profile Data*

Data Access

It groups together the interfaces dedicated to the interaction with the DBMS in order to access the *myTaxiService* database:

- *Access Accounts*
- *Access Reservation Data*
- *Access CityZone Data*

2.6 Deployment View

In the *figure 8* it's possible to see the deployment diagram. It shows the hardware of the system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another.

As we can see the main node is *myTaxi Service Server* in which the logic of system is concentrated. That node contains another node which is a *web server*, installed locally.

We decided to design another dedicated node called *State Updater* which contains the *Taxi Driver State Manager* component. That's because of an intense data traffic coming from the mobile taxi driver application which sends his position.

Let's make a rough estimate on the data. There are about 37 taxis every 10,000 people in Milan[5], and a population of 1,343,817[6]. Let's suppose every taxis sends his coordinates every 10 seconds. Then:

[5] `http://www.ilmattino.it/ITALIA/PRIMOPIANO/l_italia_delle_caste_trincea_taxisti_poche_auto_e_costi_elevati/notizie/175572.shtml`

[6] `https://it.wikipedia.org/wiki/Milano`

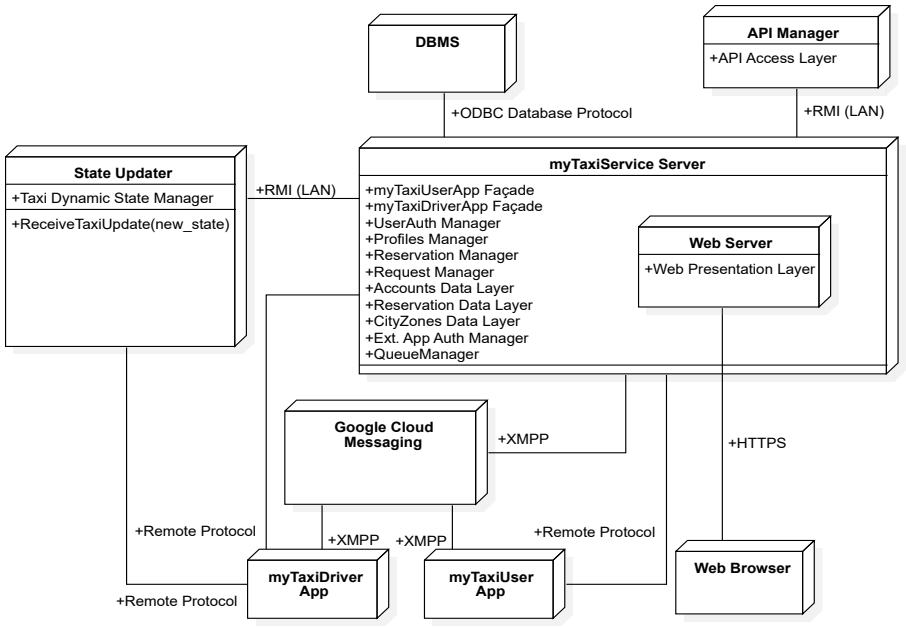


Figure 8: Deployment Diagram

$$\text{Number of Taxis:} \quad 1343817/37 \quad = \quad 36319$$
$$\text{Number of GPS Update every seconds:} \quad 36319/10 \quad = \quad 3632$$

Every GPS update is sent through a TCP/IP stream and let's suppose every message contains about 148 bytes:

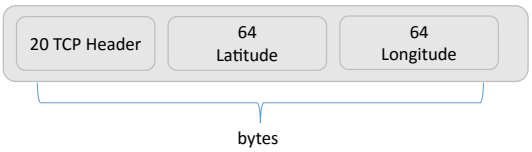


Figure 9: An approximation of the packet sizes.

That means every second there will be a information traffic of about:

$$148[\text{Bytes/Update}] \cdot 3632[\text{Update/s}] = 537536[\text{Bytes/s}] \quad = \quad 4.3[Megabit/s]$$

Further the local data traffic between the *State Updater* and the *Server* is filtered. As described in *section 2.3.5* the *Queue Manager* (which is deployed on the main node) is notified only when a taxi driver changes the city zone where he finds himself. That's why the local traffic between the two node is strictly limited avoiding bottleneck.

Moreover the diagram shows the node represents the *Google Cloud Messaging* which is explained in more details in the *section 2.2*. This service allows us to implement a simple mechanism for push notification. The protocol used for the

message exchange is the *XMPP*[7] as explained in the service Google guide[8].

The last point is the node called *API Manager*. We decided to design a dedicated node responsible for the *API* access in order to provide another layer from the *core system* and guarantee more security. Indeed in that way it is possible avoid to overload the main server with requests coming from third party applications. Moreover external application shouldn't have a way to access to the services directly.

2.7 Selected Architectural Styles and Patterns

2.7.1 Overall Architecture

Our system is essentially a *server-client* based architecture and there are three different type of clients which can connect to it.

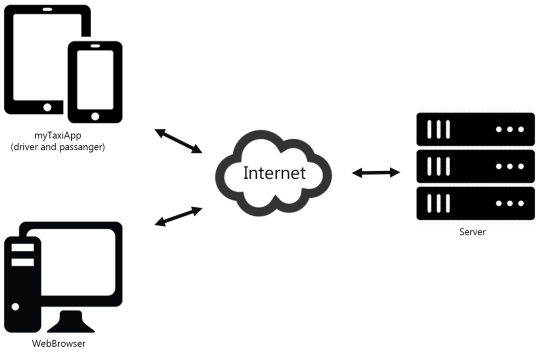


Figure 10: Client-Server Architecture

Web Browser Client The web browser is a thin client, it represents just a presentation layer rendering *web pages* and *javascript*. We needed to differentiate that client because we decided in the design phase that client cannot perform *ride request*, but only *reservation*. This because we have thought that a web browser is accessed mainly via home computer; if somebody needs to call a taxi while he finds himself in the street, he should use the mobile application. Moreover we want to encourage the usage of our mobile application.

Mobile Applications The mobile applications are both *thin* because *business logic* is implemented entirely on the server or Google's services. The *taxi driver*'s mobile application allows him to display a map of the incoming request which is generated via *Google Maps Service*.

In accordance with the *RASD*, the *passenger*'s mobile application has another functionality which allows him to select a nearby place around himself in case the GPS technology is not working. Also in that case the client is going to use a external service called *Google Places Service*.

[7] `https://en.wikipedia.org/wiki/XMPP`
[8] `https://developers.google.com/cloud-messaging/`

2.7.2 Design Patterns and Architectural Choices

Handle Updates Although the main architecture has been designed as an client server based system, there are some conceptual aspects to underline.

As we can see in the sequence 2.3.5 we have used a *observer pattern*: when an environment event happens because the taxi changes city zone, the *Queue Manager* and every component responsible for the update of the user's interface have to be notified. In that way the *Queue Manager* can update its queues and shift the taxi driver previously assigned to a queue into another one.

Google Cloud Messaging Another important architectural choice is the usage of the service called *Google Cloud Messaging*. As we have already written that service allow us to implement a system based on *push notification* in a useful and practical way.

The reason of this choice is quite simple. First of all, the usage of that service simplifies the implementation stage (*coding*) of our system. That because it could be an hard problem the designing of an robust *asynchronous* protocol for message exchanging: what happens when a client loses connection for a while?

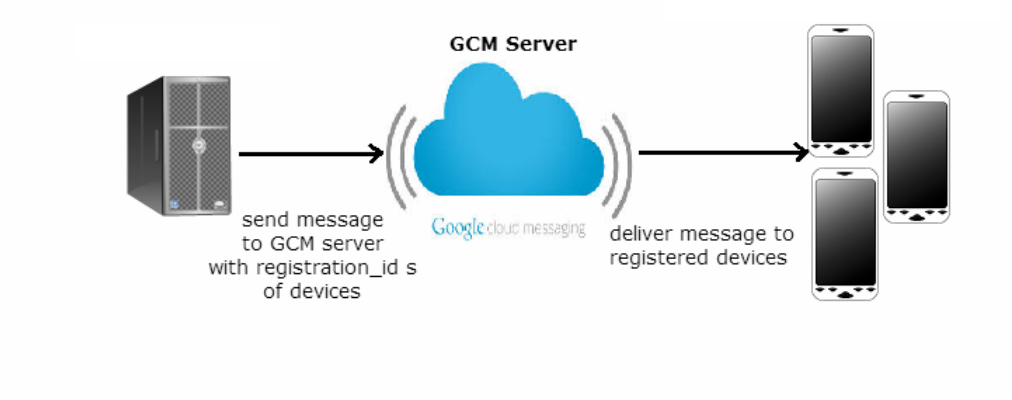


Figure 11: Google Cloud Messaging

Robustness Aspect The principal usage of the system is done through *mobile* applications which are running on mobile terminals. As we can imagine, in that domain the *robustness* of the system is an important aspect to keep in mind. The mobile devices are often subject to loss of connectivity and for that reason the communication between the server and a client could be not available in various time points.

Through the usage of *Google Cloud Messaging Service* the server can upload the request for the taxi driver without caring whether he has a connection stream or not. Every push notification is so handled. In this way in case of communication problem we ensure that the system keep working and it is going to ask to another taxi driver for the satisfaction of the ride request.

2.8 Other Design Decisions

2.8.1 Programming Language

For the implementation of *myTaxiService* we choose Java as programming language. This choice is based on the following considerations:

- Java is a very widespread programming language, so we are sure that there will be more availability of skilled programmers in this technology.
- The system shall be developed on the Java EE platform, that ensures us large-scale, multi-tiered, scalable, reliable, and secure properties.

2.8.2 Data Layer

In this paragraph we report an ER diagram that describes the structure of *myTaxiService* database.

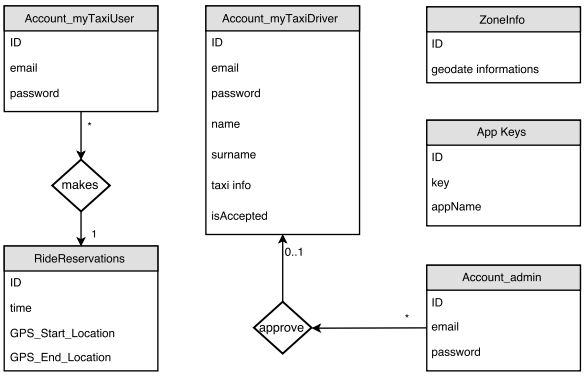


Figure 12: ER Diagram for Data Layer

3 Algorithm Design

We want to describe more precisely the algorithms that rule the queue management and the priority calculation, because we think they are important aspects of the development of the application.

3.1 Priority Calculation

In our system, a logged *myTaxiDriver* status can be exactly one between the following: *Available*, *Busy*, *Riding*. The *myTaxyDriver* can manually switch only between *Available* state and *Busy* state. The complete mechanism of state changes is described in the finite state automata below:

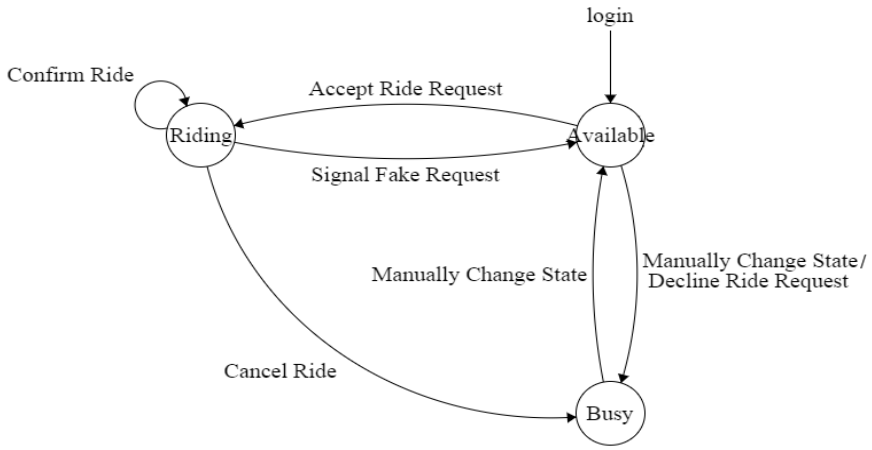


Figure 13: Finite State Automata

Every time a *myTaxiDriver* logs in, the variable *priority* is set at value 0; otherwise, the value of the priority level increases and decreases according to the following algorithm, written in pseudo code:

```
while(true)
        if(loggedInJustNow)
                priority.reset();

        while(state == available)
                priority.increment()

        if(endOfTheRide == true)
                if(fakeRequestSignaled ||
                                rideReservationSatisfiedOnTime)
                        state = available;
```

```
                else
                        priority.reset();
                endOfTheRide = false;

        if(rideRequestDeclined == true)
                state = Busy
                priority.reset()
                rideRequestDeclined = False;
```

3.2 Queue Management

This algorithm will run on the server, because it needs priority and position informations about all the available *myTaxiDriver* of the city. Periodically every *myTaxiDriver* client sends to the server the coordinates (detected by the GPS) of the device on which the client application is running. Every time the server detects that a *myTaxiUser* has changed its taxi zone or has just logged in, the associated queue is updated with the insertion of the new *myTaxiUser* in a position that maintains the queue in order of priority.

When a *ride request* or a *ride reservation* is processed, the server application start the research of an available *myTaxiDriver* that takes on the responsibility of the ride, starting from the queue associated to the *taxi zone* in which the passenger must be picked up, and progressively in the adjacent *taxi zones*. The research is done in according to the algorithm below, written in pseudo code:

```
function assignMyTaxiDriver
        startCityZone cz <- GPStoCityZone(request.gps)
        Queue q <- GetQueueOfCityZone(cz)

        while(requestHasBeenServed or notMoreQueuesToScan)
                while(q.queueIsNotEmpty)
                        TaxiDriver td <- q.pop
                        answer <- SendRequestTo(td,requestInfo)
                        wait_for_seconds(30)
                        if(answer = OK)
                                SetTaxiStateInTravel(td)
                                return OK,ID_Taxi
                        else(answer = NO or answer=NOT_RECEIVED)
                                SetTaxiStateBusy(td)
                end
                q <- nearToQ(q)
        end

        return NO_TAXIS
end function
```

How the request is sent to the taxis is better described in the *section 2.3.3*.

4 User Interface Design

4.1 Guest User Graphic Interface

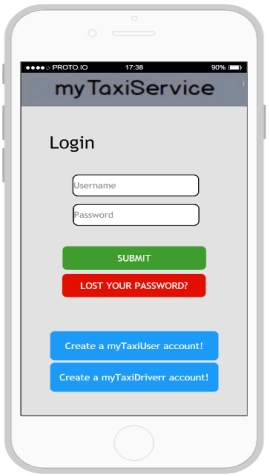
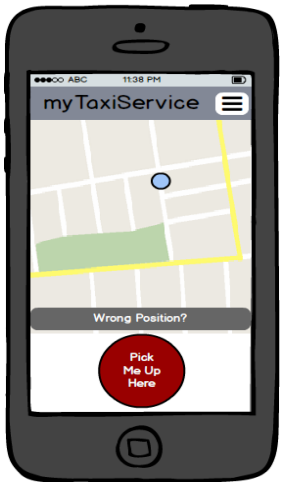


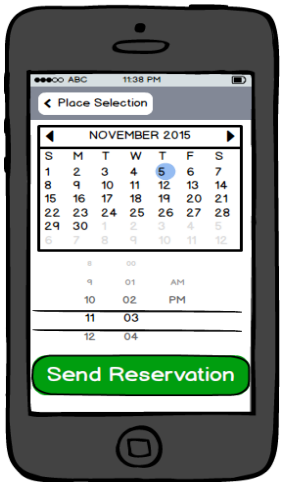
Figure 14: Login Interface.

4.2 myTaxiUser App Graphic Interface



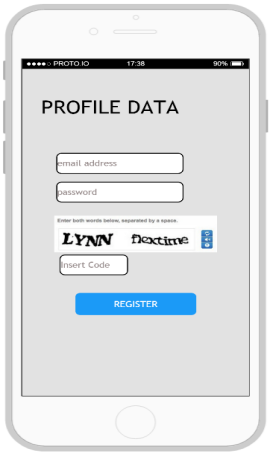
(a) *myTaxiUser home screen.*

(b) *Manual Starting Point Address Insertion.*

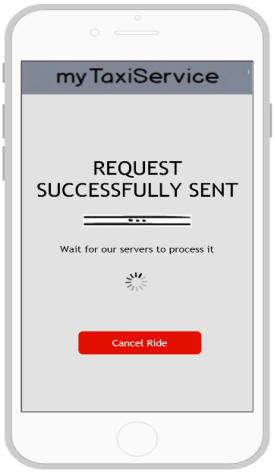


(c) *Make a Reservation.*

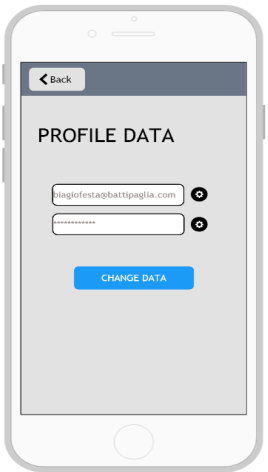
(d) *Waiting For Taxi Arrival.*



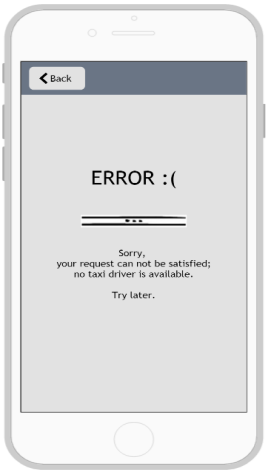
(a) *Registration myTaxiUser Account*



(b) *Request Sent*



(c) *Update Profile Information*



(d) *Error Screen*

Figure 15: Other mockups user interface for passenger user.

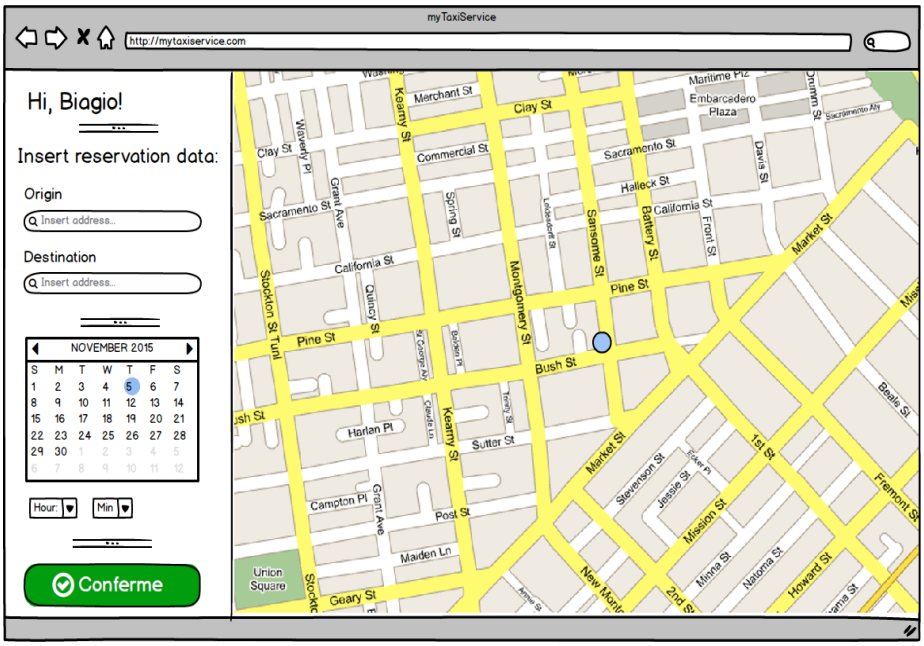
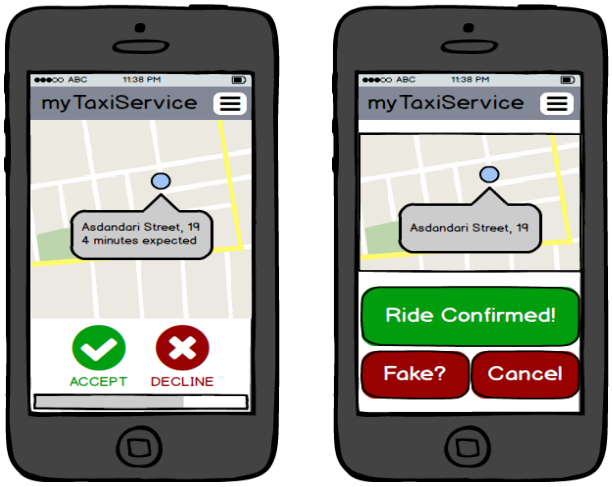


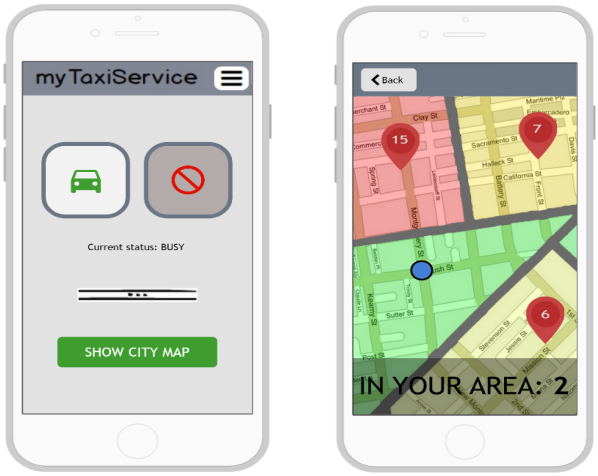
Figure 16: User web interface mockup

4.3 myTaxiDriver App Graphic Interface



(a) *Manage Incoming Request*

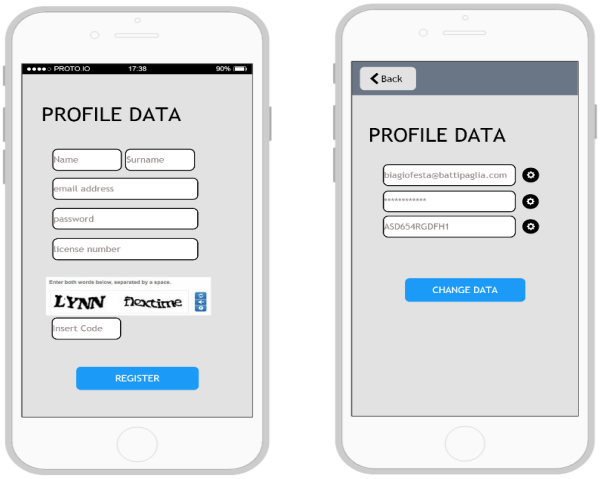
(b) *Collecting The Passenger*



(c) *Stadby Taxi Driver Screen*

(d) *City Map Screen*

Figure 17: Some mockups user interface for taxi driver.



(a) *Registration Account*

(b) *Uodate Profile Information*

Figure 18: Other mockups user interface for taxi driver.

4.4 Flow Graph for Screens Interface

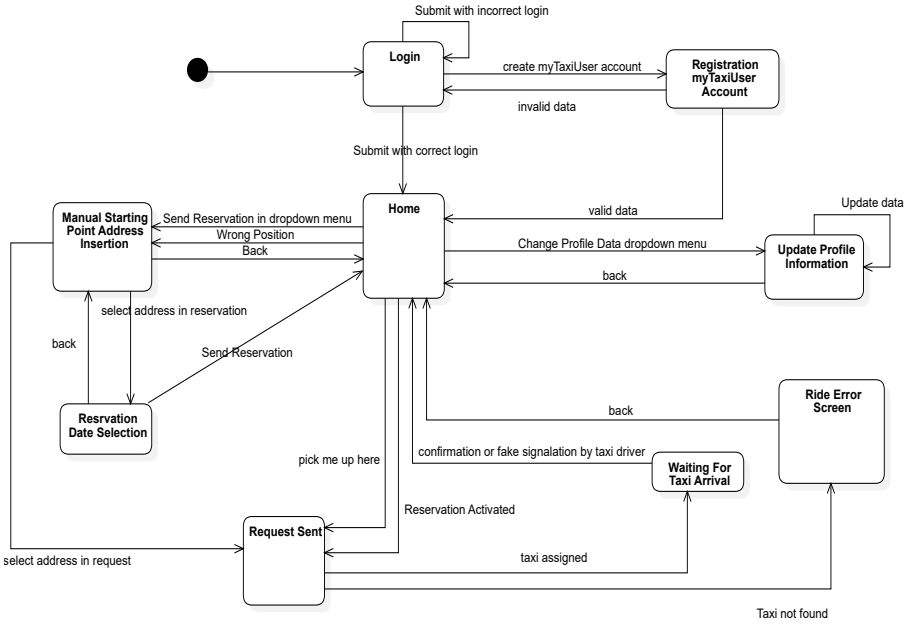


Figure 19: Flow *myTaxiUser* Screens

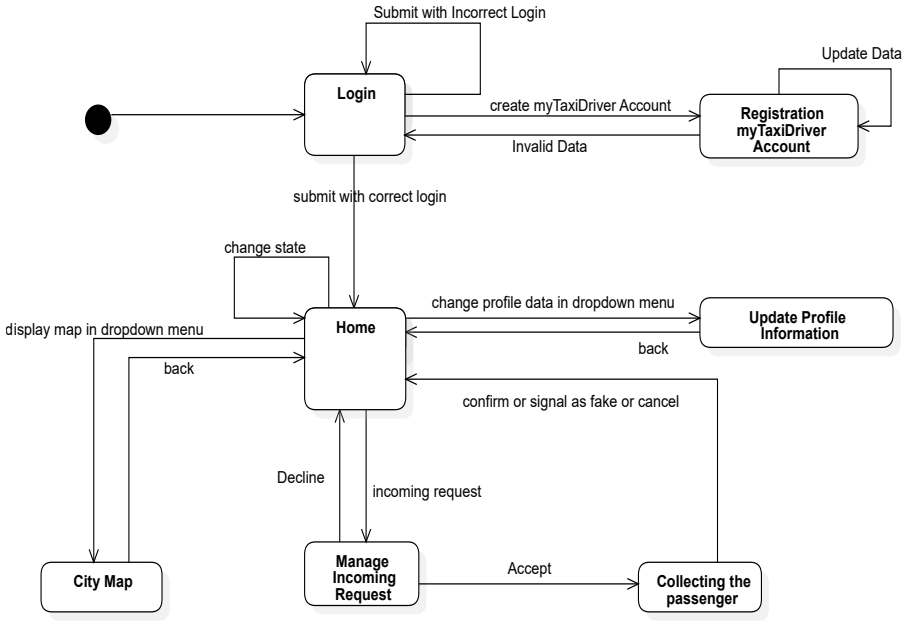


Figure 20: Flow *myTaxiDriver* Screens

5 Requirements Traceability

6 Requirements Traceability

All functional requirements are covered obviously by the interaction between the different presentation layers (i.e. *myTaxiDriver App Presentation Layer*, *myTaxiUser App Presentation Layer*, *Browser*) and the other component of the system. So, in the following tables, we'll only list the components that interact with the presentation layer and not the involved presentation layer itself.

6.1 Guest User

Reference	Requirement	Component
[1]	Registration as *myTaxiUser*	Users Auth Manager
[2]	Request to become a *myTaxiDriver*	Users Auth Manager

6.2 Registered User

Reference	Requirement	Component
[3]	Login	Users Auth Manager
[4]	Password Recovery	Users Auth Manager
[5]	Update Account Data	Profiles Manager

6.3 myTaxiUser

Reference	Requirement	Component
[6]	Send Ride Request	Request Manager
[6.1]	Detect GPS	System call of the device in which the application is running
[6.2]	Manually insert address	Google Maps
[6.3]	Select position from points of interest	Google Places
[7]	Check ETA	Google Maps
[8]	Cancel *RideRequest* or *RideReservation*	Reservation Manager and Request Manager
[9]	Send *RideReservations*	Profiles Manager

6.4 myTaxiDriver

Reference	Requirement	Component
[10]	Manage incoming request	Request Manager
[10.1]	Visualize ride details	Google Maps
[10.1.1]	GPS coordinates of picking up place	Google Maps
[10.1.2]	Check ETA	Google Places
[10.2]	Accept or decline incoming request	Queues Manager
[11]	Cancel acceptance	Request Manager
[12]	Signal state	Taxi Dynamic State Manager
[13]	Signal *fake request*	Request Manager
[14]	Confirm a ride	Request Manager

6.5 Guest User

Reference	Requirement	Component
[15]	API usage	API Access Layer

6.6 Administrator User

Reference	Requirement	Component
[16]	Create admin accounts	Request Manager
[17]	Visualize position and state of *myTaxiDrivers*	Google Maps
[18]	Activate *myTaxiDrivers* accounts	Users Auth Manager

7 References

Google Maps API and Place
https://www.google.com/intx/en_uk/work/mapsearth/products/mapsapi.html?utm_source=HouseAds&utm_medium=cpc&utm_campaign=2015-Geo-EMEA-LCS-GEO-MAB-DeveloperTargetedHouseAds&utm_content=Developers&gclid=Cj0KEQiA4eqyBRDUh7Omv9vCtsoBEiQAspfs8tPEemehd7UJJGwwp2i-YPqwyPPCnJXUhqEdYq9oeYIaAv558P8HAQ.

Google Cloud Messaging
https://developers.google.com/cloud-messaging/gcm.

Requirements Analysis and Specification Document
Previously written.

Assignments 1 and 2 (*RASD* and *DD*)
Which can be retrieved on the beep page of the course.

8 Hours of Work

- Beri Giovanni: ~ 40 hours
- Festa Biagio: ~ 40 hours