

Design Document

Version 1.0

Giorgio Pea(Mat. 853872), Andrea Sessa(Mat. 850082)

4/12/2015



Contents

1	Introduction	2
1.1	Purpose	2
1.2	Scope	2
1.3	Terms Definition	2
1.3.1	Glossary	2
1.3.2	Acronyms	2
1.4	Reference Documents	3
1.5	Document Structure	3
2	Architectural Design	4
2.1	Overview	4
2.2	High level components and their interaction	4
2.3	Component View	6
2.4	Runtime View	7
2.5	Components Interface	14
2.6	Selected architectural styles and patterns	16
2.7	Deployment View	19
3	Algorithms Design	21
4	User Interface Design	25
5	Requirements Traceability	26
6	Appendix	27
6.1	Tools	27
6.2	Hours	27

1 Introduction

1.1 Purpose

This document represents the Design Document(DD). The purpose of the Design Document is to provide a medium/base level description of the design of MyTaxiService in order to allow for software developers to proceed with an understanding of what is to be built and how it is expected to be built.

The main goal of this document is to completely describe the system-to-be by:

- Detecting high-level components of the software to be
- Describing how these components communicate and interact with each other
- Describing how software components are distributed on the architecture's tiers
- Motivating and describing the adopted architectural style

1.2 Scope

The aim of this project is to develop MyTaxiService, a web/mobile application that makes easier and quicker taking taxis within the city's borders. Thanks to MyTaxiService, anyone can request or book a taxi and get realtime information about how long it will take to be picked up or about the taxi's current position and identification code. In addition to that, MyTaxiService provides an efficient way to allocate taxis by dividing the city in zones and using a queue based allocation system, in order to reduce the average waiting time and city's traffic.

This Software Design is focused on the base level system and critical parts of the system.

1.3 Terms Definition

1.3.1 Glossary

- **Tier:** Refers to a possible hardware level in a generic architecture
- **Layer:** Refers to a possible software level in a generic software system
- **Mtaxi:** A taxi that joined MyTaxiService

1.3.2 Acronyms

- **DD:** Design Document
- **MVC:** Model View Controller
- **FIFO:** First In First Out
- **API:** Application Programming Interface

- **GUI:** Graphic User Interface
- **GPS:** Global Positioning System

1.4 Reference Documents

- RASD version 1.1

1.5 Document Structure

Introduction

This section provides a general description of the Design Document by clearly stating purpose and aim of the project. It also includes a disambiguation section to help the reader in the process of resolving the ambiguity generated by the use of natural language

Architecture Design

The first part of this section provides a detailed description of the high-level components of MyTaxiService and of how these components interact. The second part introduces the architectural style chosen for MyTaxiService. The focus is on motivation, advantages and **possible** disadvantages of the chosen architecture.

Algorithms Design

The section aims to provide a very medium/low level description of some routine functionalities of MyTaxiService. Some code is included.

User Interface Design

In this section are provided some mockups describing the requirements of the user interface to-be

Requirements Traceability

This section provides a matrix of traceability that allows the reader to map functional requirements on the previously defined software components

2 Architectural Design

2.1 Overview

The architectural design section is divided into two main parts:

Software components description and interaction

In this section is provided a detailed description of the main components of the software system and of their interactions. To exemplify the above mentioned description a set of UML diagrams(Component, Deployment, Sequence diagrams) is included.

Architectural styles and patterns

In this section the software system architecture is illustrated using a schematic diagram. In addition to that all the architectural choices, patterns and styles considered are motivated and described.

2.2 High level components and their interaction

For a detailed description of the components interaction see section 2.5 and 2.6

Controller Components:

1. Dispatcher

This component provides the functionality of dispatching messages from **MyTaxiService(B)** to users, mtaxies and administrators. Examples of these messages are :

- (a) For users : approximate waiting time for being picked up, the requested mtaxi's identification code
- (b) For mtaxies : zone change order, new ride request
- (c) For administrators : report of an accident, report of bad behavior

2. Request Manager

This component is in charge of:

- (a) Managing all types of requests coming from the Requests Receiver(this requires an interaction with the User Messages Dispatcher, the Taxi Messages Dispatcher, the Queue Manager, the Location Manager and the Data Manager)
- (b) Managing a situation of non **fair distribution of mtaxies** in the city's zones(this requires an interaction with the Queue Manager)

3. External Services manager

This component is in charge of the interaction of MyTaxiService(B) with external services such as the traffic information service or the mtaxies GPS data and allows other components to access these data.

4. **Location Manager**

This component requests and digests data from the External **Communication** Manager; the Location Manager provides also services that elaborate these data and make them accessible by other components

5. **Queue Manager**

This component is in charge of:

- (a) Distributing available mtaxies in city's zones and organize them in zone per zone queues
- (b) Checking periodically if the distribution of mtaxies in city's zones is fair and computing the mtaxies that need to be moved to other zones

6. **Request Receiver**

This component manages incoming messages from users, mtaxies and administrators. According to the type of message, the request receiver will properly invoke services provided by the Request Manager.

Model Components:

1. **Data Manager**

This component is in charge of managing all accesses to the database system. It works as a sort of intermediate layer between the application logic and the database system

View Components:

For each of these view components a communication manager is associated: this component manages the dispatching of requests to and the responses from MyTaxiService(B)

1. **MyTaxiServiceAppUserGUI**

This component manages the interaction between a user and the whole software system. This interaction refers to the MyTaxiService mobile app

2. **MyTaxiServiceAppDriverGUI**

This component manages the interaction between a mtaxi driver and the whole software system. This interaction refers to the MYT device

3. **MyTaxiServiceWebUserGUI**

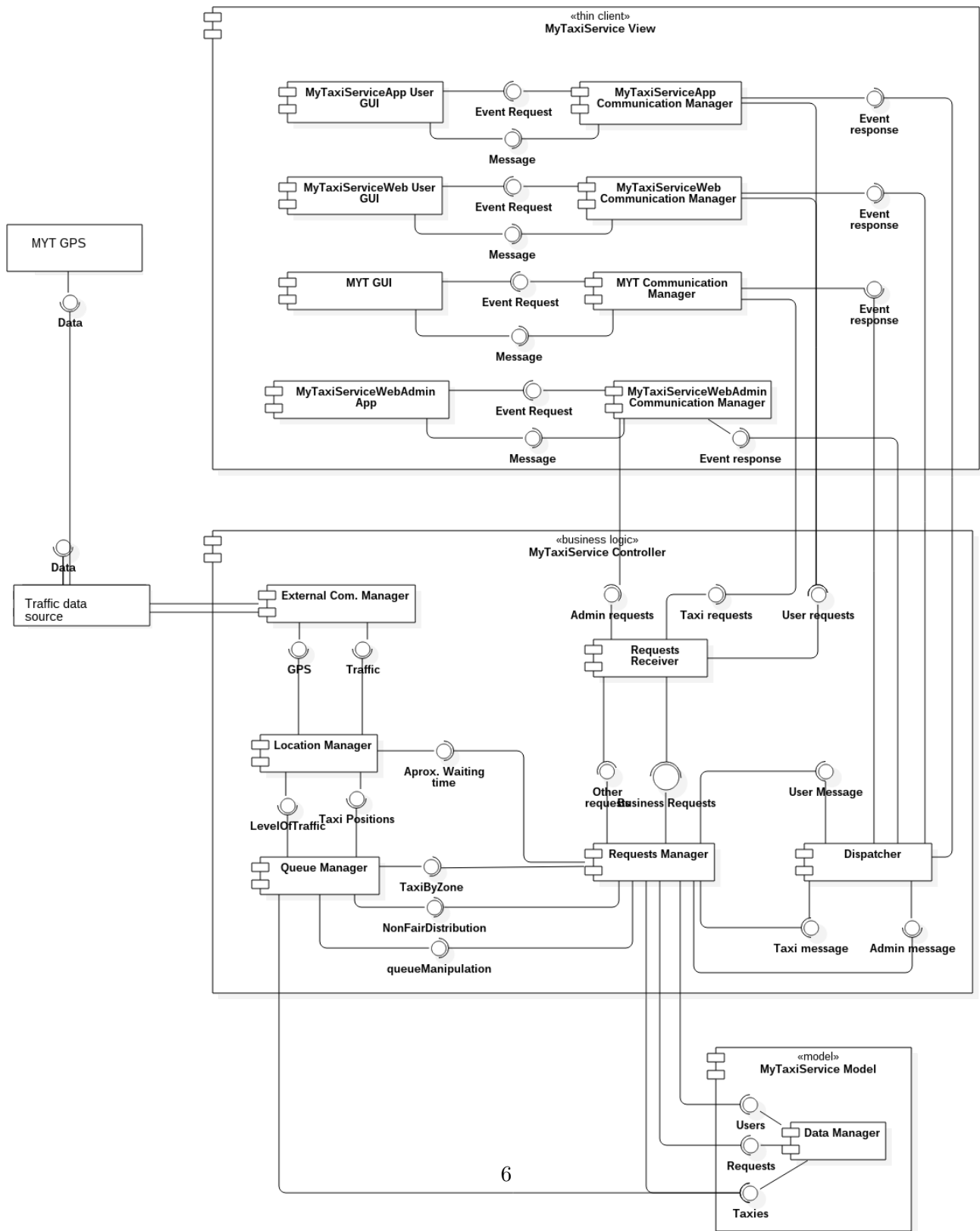
This component manages the interaction between a user/a taxi driver and the whole software system. This interaction refers to the MyTaxiService web app

4. **MyTaxiServiceWebAdminGUI**

This component manages the interaction between an administrator and **the whole software system**. This interaction refers to the MyTaxiService web app for administrators

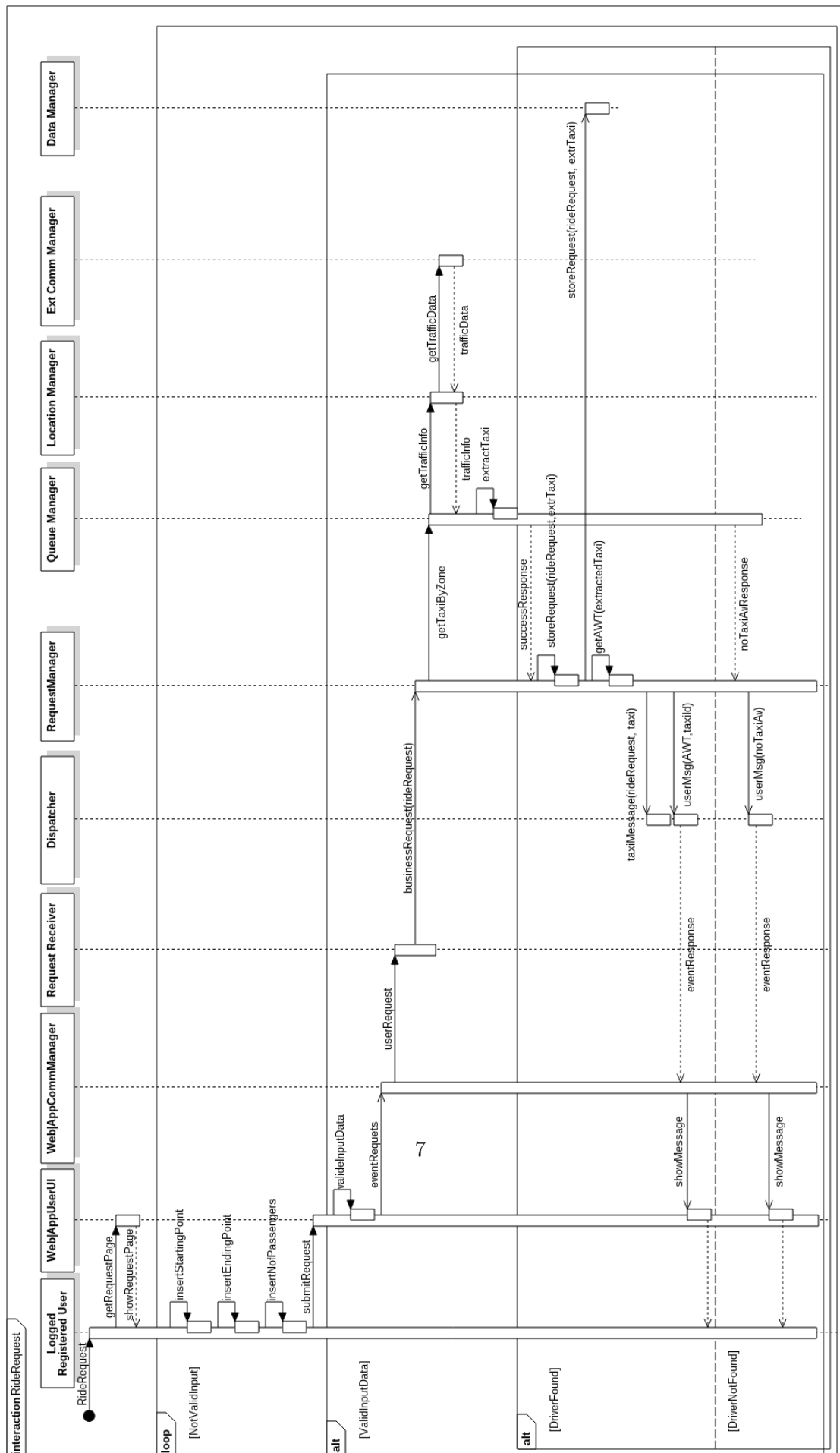
2.3 Component View

In this section is included a general schema of the components and their interactions

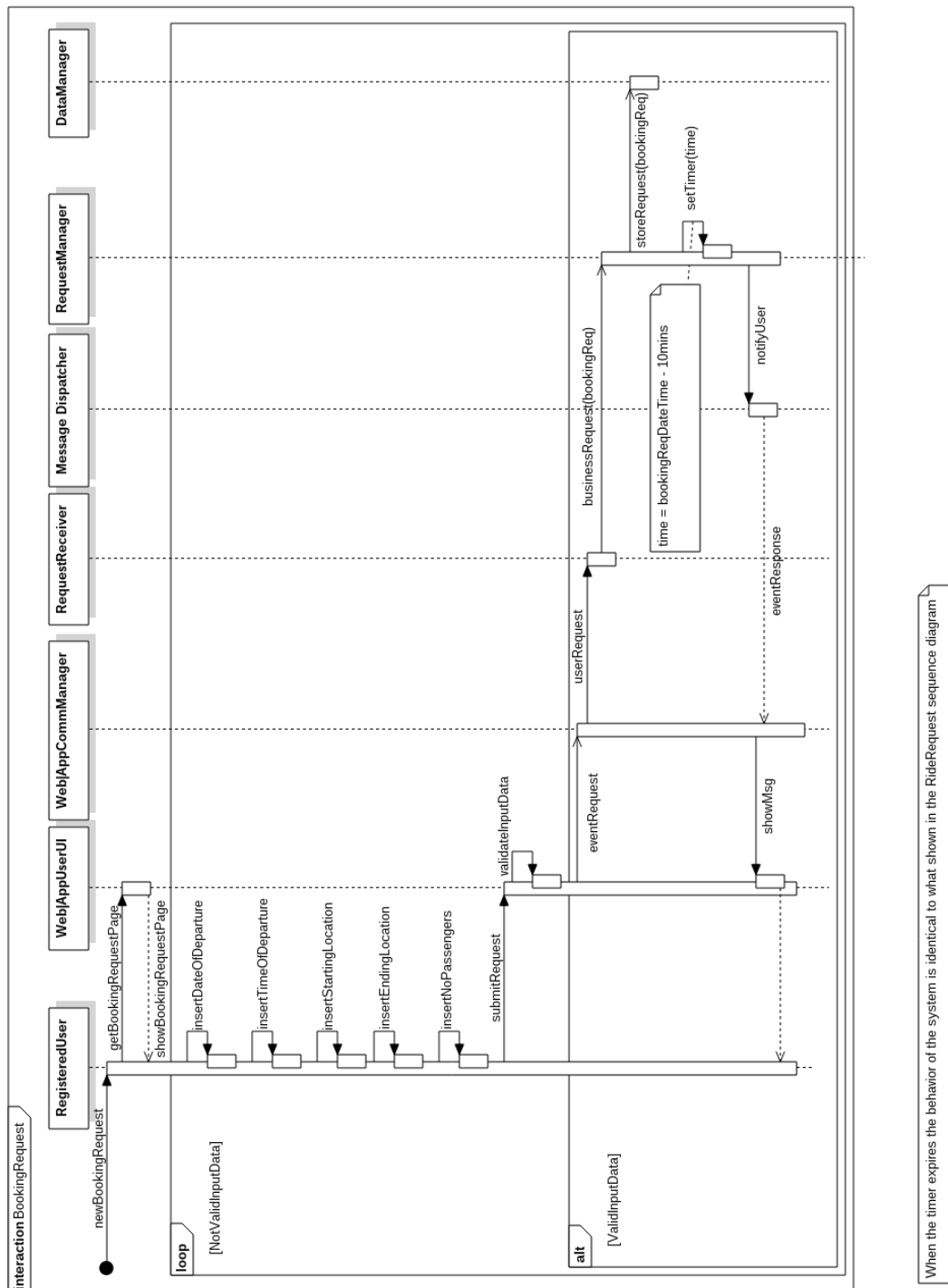


2.4 Runtime View

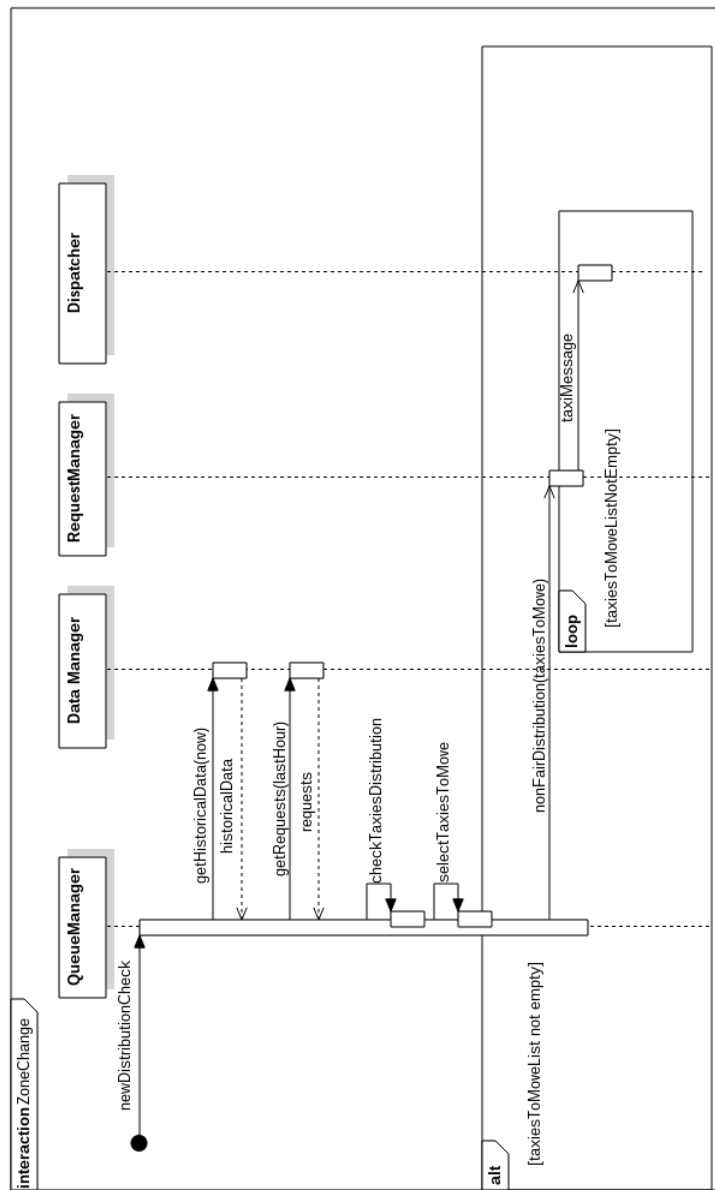
Components interaction in case of ride request.



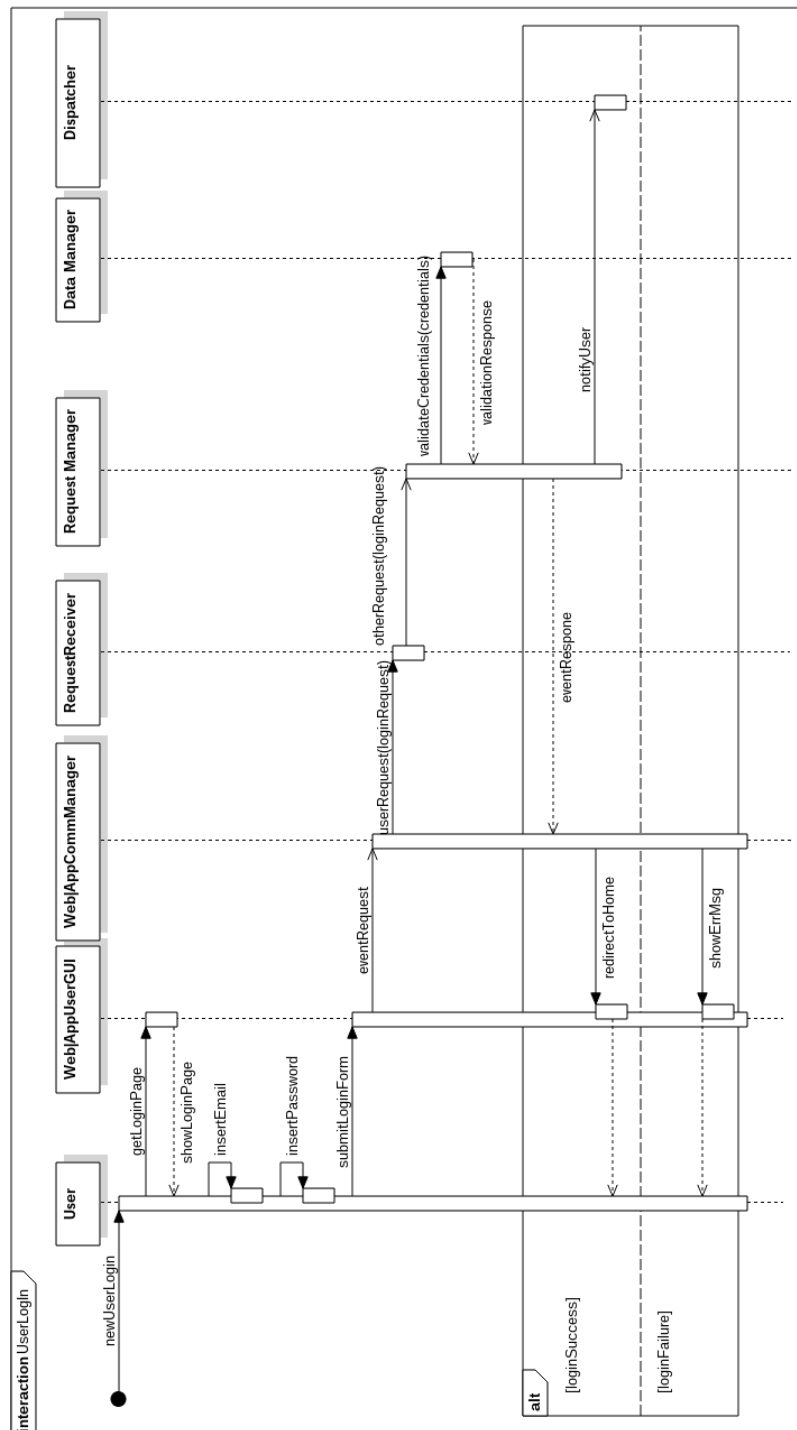
Components interaction in case of booking request.



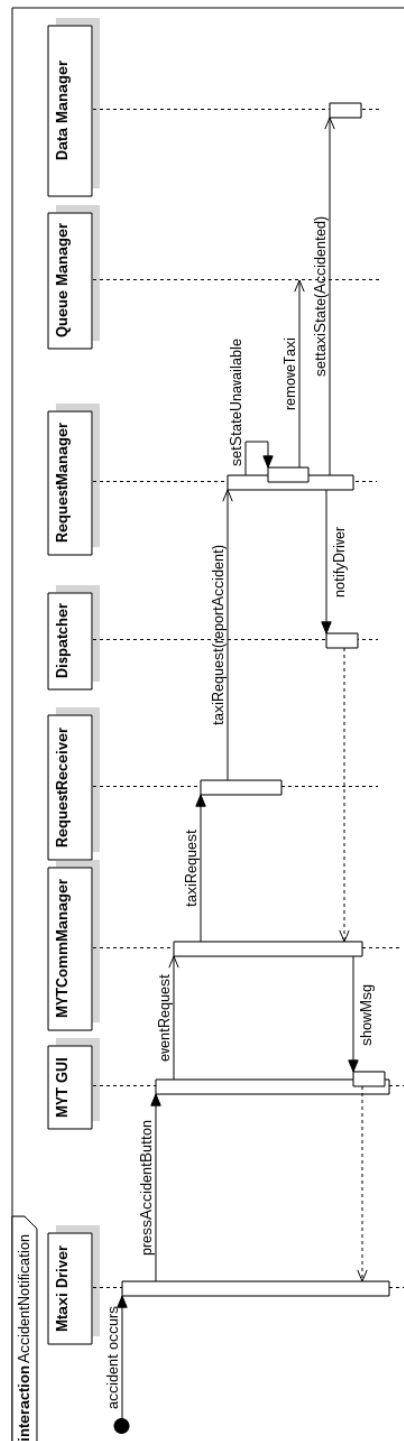
Components interaction when a new taxis redistribution is needed .



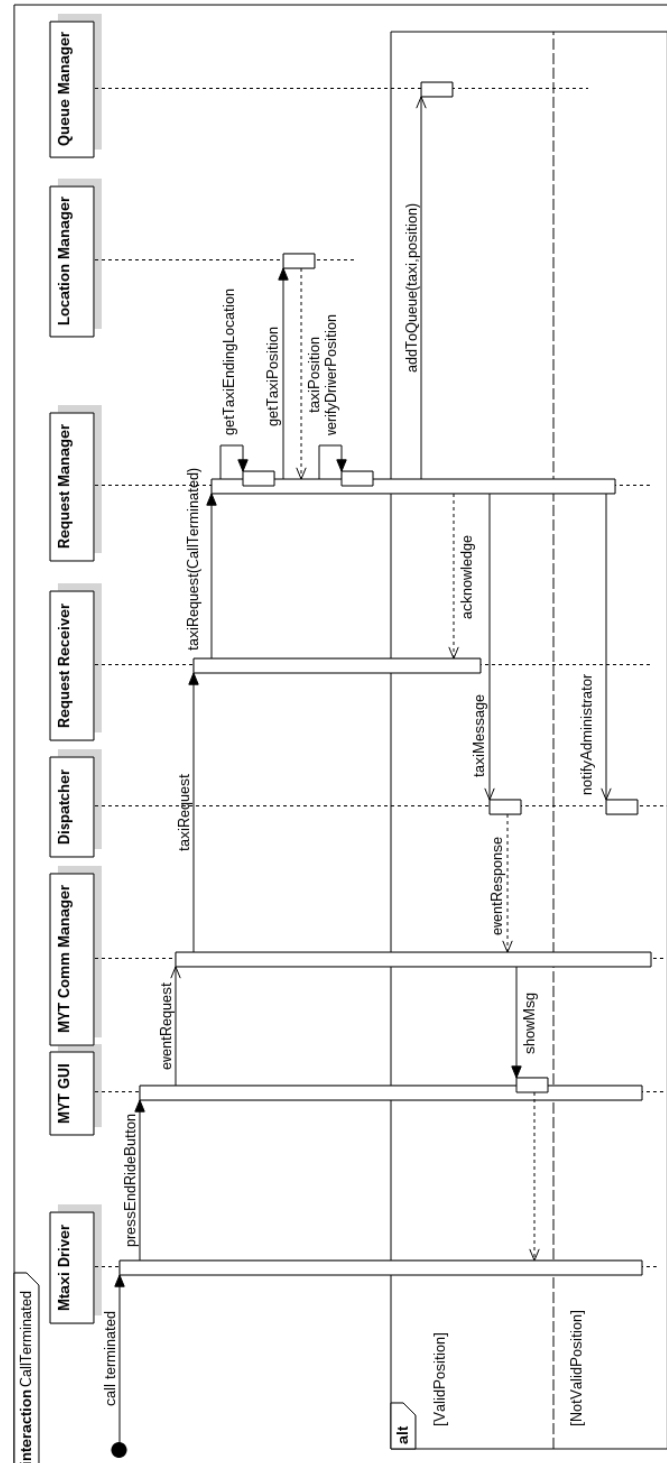
Components interaction in case of user login.



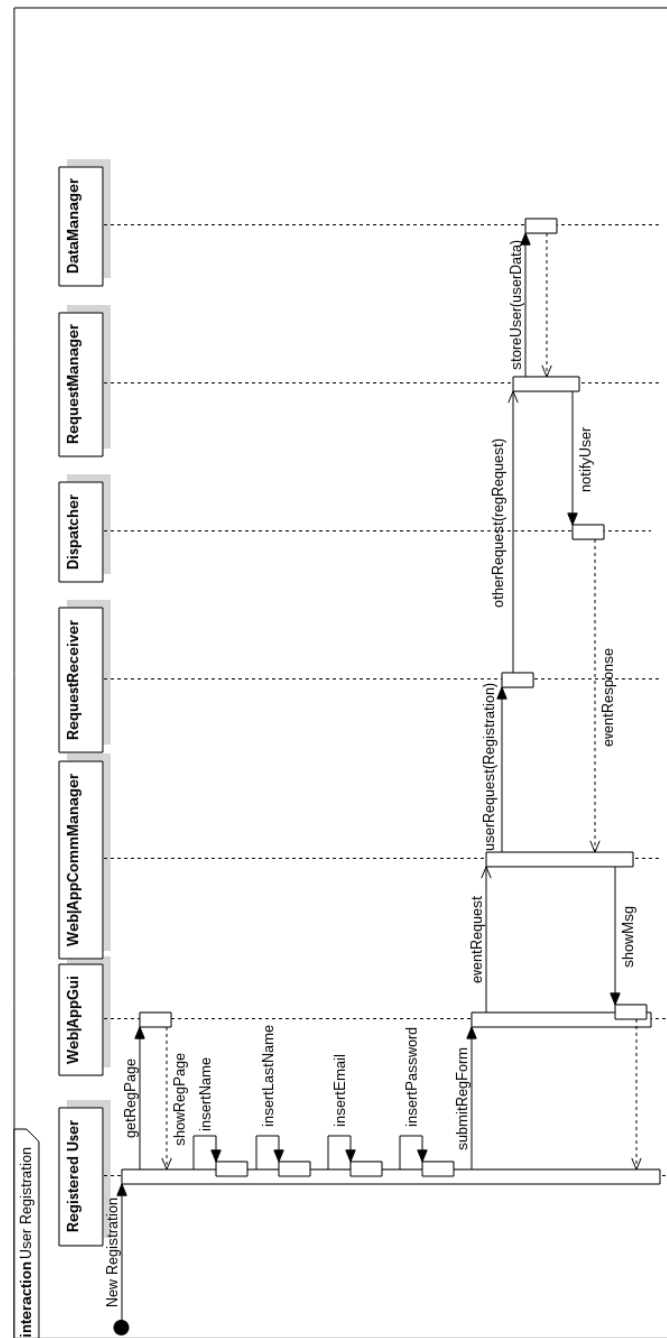
Components interaction in case of accident notification from a driver.



Components interaction when the driver notify a ride end.



Components interaction in case of user registration.



2.5 Components Interface

Request Receiver

This component interfaces with the communication managers of the view in three ways: it receives and pre-process requests from users (users requests), from mtaxi drivers(taxies requests) and from administrators(admin requests). In addition to that, it directly communicates with the Request Manager so that the above mentioned requests are fully elaborated and executed; this is done using three interfaces: one for **other requests**, another for **business requests** and last one for **admin requests**. All messages are exchanged in synchronous way on both side.

Request Manager

This components interfaces with all the components of the controller except for the External Services Manager, it also interfaces all the components of the model. In particular the Request Manager communicates with:

1. Location Manager: via the AWT interface in order to get the **AWT** for a mtaxi to reach a user (synchronous communication)
2. Queue Manager: via three interfaces:
 - (a) TaxiByZone: in order to get the first available mtaxi for a given zone (synchronous communication)
 - (b) NonFairDistribution: the Request Manager observes the Queue Manager for a non fair distribution of mtaxies among city's zones and so for a set of mtaxies **the** need to be moved to other specified zones (asynchronous communication)
 - (c) ManipulateQueue: in order to manipulate the mtaxies in a given queue (for example when a mtaxi reports and accident, then it should be made non available and removed by its zone associated queue) (synchronous communication)
3. Dispatcher: via three interfaces:

All the communications are synchronous

 - (a) Users messages : in order to send a response to a user request
 - (b) Taxi messages : in order to send a response to a mtaxi request
 - (c) Admin messages : in order to send a response to an admin request
4. Data Manager: via three interfaces:

All the communications are synchronous

 - (a) Users : in order to store/retrieve data about users
 - (b) Taxies : in order to store/retrieve data about mtaxies
 - (c) Requests : in order to store/retrieve data about requests

Queue Manager

This components offers to the Request Manager the interfaces already described above. In addition to that it communicates with:

1. Location manager : via two interfaces:
All the communications are asynchronous
 - (a) Taxi positions: in order to get the position(in terms of city zone) of mtaxies
 - (b) LevelOfTraffic : in order to get the level of traffic for a mtaxi and so use another mtaxi to fullfill a request
2. Data Manager via an already considered interface:
 - (a) Taxies : in order to retrieve data about mtaxies and populate the zone per zone queues

Data Manager

This component offers to the Request Manager the interfaces already described above to retrieve/store data from to the database

Location Manager

This component offers to the Request Manager and to the Queue Manager the interfaces already described above. In addition to that it communicates with the External Services Manager via two interfaces:

All the communications are synchronous

1. GPS : in order to get raw gps data from mtaxies thanks to to GPS system integrated in their MYT
2. Traffic : in order to get raw data about the traffic from a generic external provider

Dispatcher

This component offers to the Request Manager the interfaces already described above. In addition to that, it communicates(indirectly using the web) with the communication managers of the view via the event response interfaces; this is done in order to deliver responses to requests and to deliver asynchronous messages (asynchronous and synchronous communication)

External Services Manager

This component offers to the Location Manager the interfaces already described above. In addition to that, it communicates with:

1. The MYT device GPS system via the data interface; this is done in order to get the gps coordinates of a mtaxi
2. A generic traffic source via the data interface; this is done in order to get raw data about the traffic conditions of the city

All communications are asynchronous

MyTaxiServiceApp/MyTaxiServiceUserWeb/MyTaxiServiceAdminWeb/MYT GUI

These components offers to their relative communication managers a message interface They communicate with their relative communication managers via the event request interface; this is done in order to forward a generic request to MyTaxiService(B) All communications are synchronous

MyTaxiServiceApp/MyTaxiServiceWeb/MYT/MyTaxiServiceAdminWeb Comm. Manager

These components offers to the Dispatcher the already described interfaces. In addition to that they communicate with the relative GUIs using the message interfaces; this is done in order to make the GUI display a response to a request or an asynchronous message All communications are asynchronous

2.6 Selected architectural styles and patterns

Some important preliminary considerations:

- The number of requests and of users could be very high
- In the future MyTaxiService can be made available for other cities
- The number of requests can have a very high variance(e.g if an important event is held in the city then the number of requests will probably increase by a lot)

Due to these simple considerations the chosen architecture for MyTaxiService is a 3 tiers Client/Server architecture. This kind of architecture has been chosen because it guarantees:

- Scalability: the nodes that compose the application server and db server can be increased in number without the need of redesign the whole system
- Reliability: since the chosen architecture is distributed, failures can be isolated easily and db replications can be implemented without impacting too much on the design of the whole system
- Adaptability: the chosen architecture can easily be enriched with a load balancer that distributes and activates nodes in relation to the number of incoming requests
- Security: since the chosen architecture divides strictly business logic and data and since the application server and the db server are isolated from the web using firewalls, malicious attacks can be prevented and high level of security granted

The chosen architecture composes of:

1. A tier(view) which include:
 - (a) Smartphones in which the MyTaxiService mobile app is installed
 - (b) Smartphones and computers trough which MyTaxiService User website is accessible

- (c) Smartphones and computers through which MyTaxiService Admin website is accessible
 - (d) MYT devices through which mtaxi drivers can manage ride requests
2. A tier(controller) that is composed by an application server(can be distributed) that manages requests coming from users and mtaxies by applying the business logic at the base of the whole system
 3. A tier that is composed by a database server(can be distributed) that manages data manipulation and access requests from the system's application server
 4. A set of external services used by the application server to implement part of its logic. It is supposed that these services are distributed to the system using a SOA architecture.

The website's assets are hosted by a webserver.

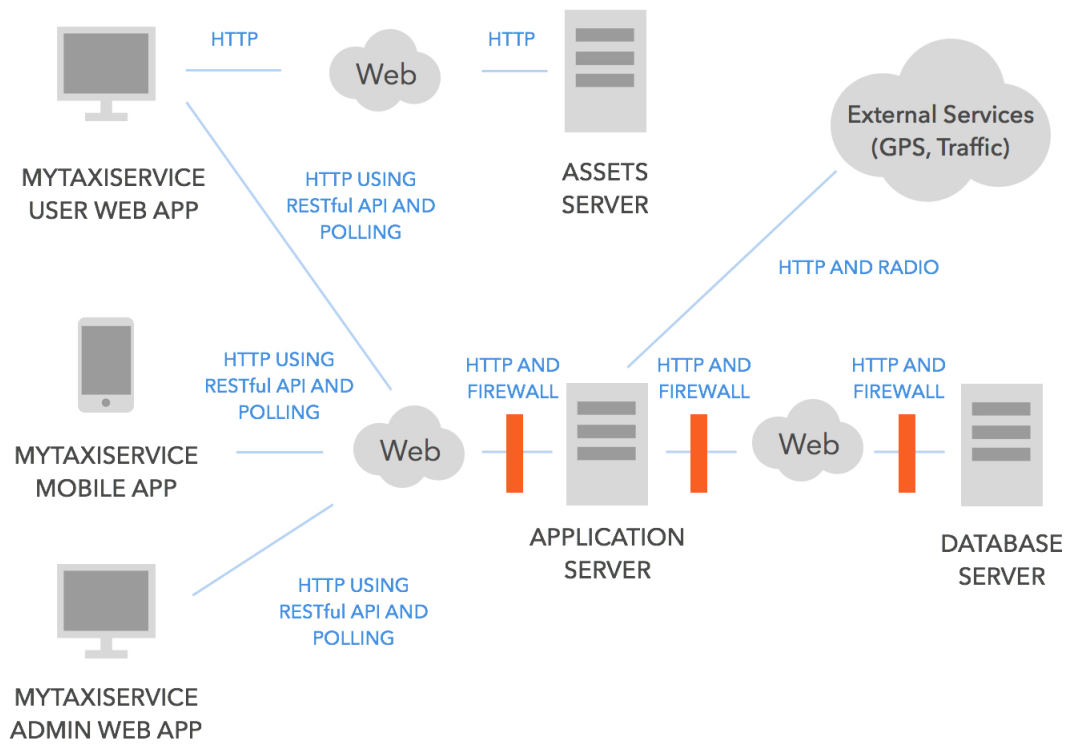
The client in this architecture is a "thin" client, it just contains only an interaction/visualization layer, all the logic of the application is condensed in the application server, which is, in this way, a "fat" server.

As already mentioned above the central system makes use of external service regarding the traffic information **in the city**. It could be assumed that information are distributed to **MyTaxiService** via a SOA architecture because this is the most suitable(and most used) way to distribute data and services to external companies.

The role of polling

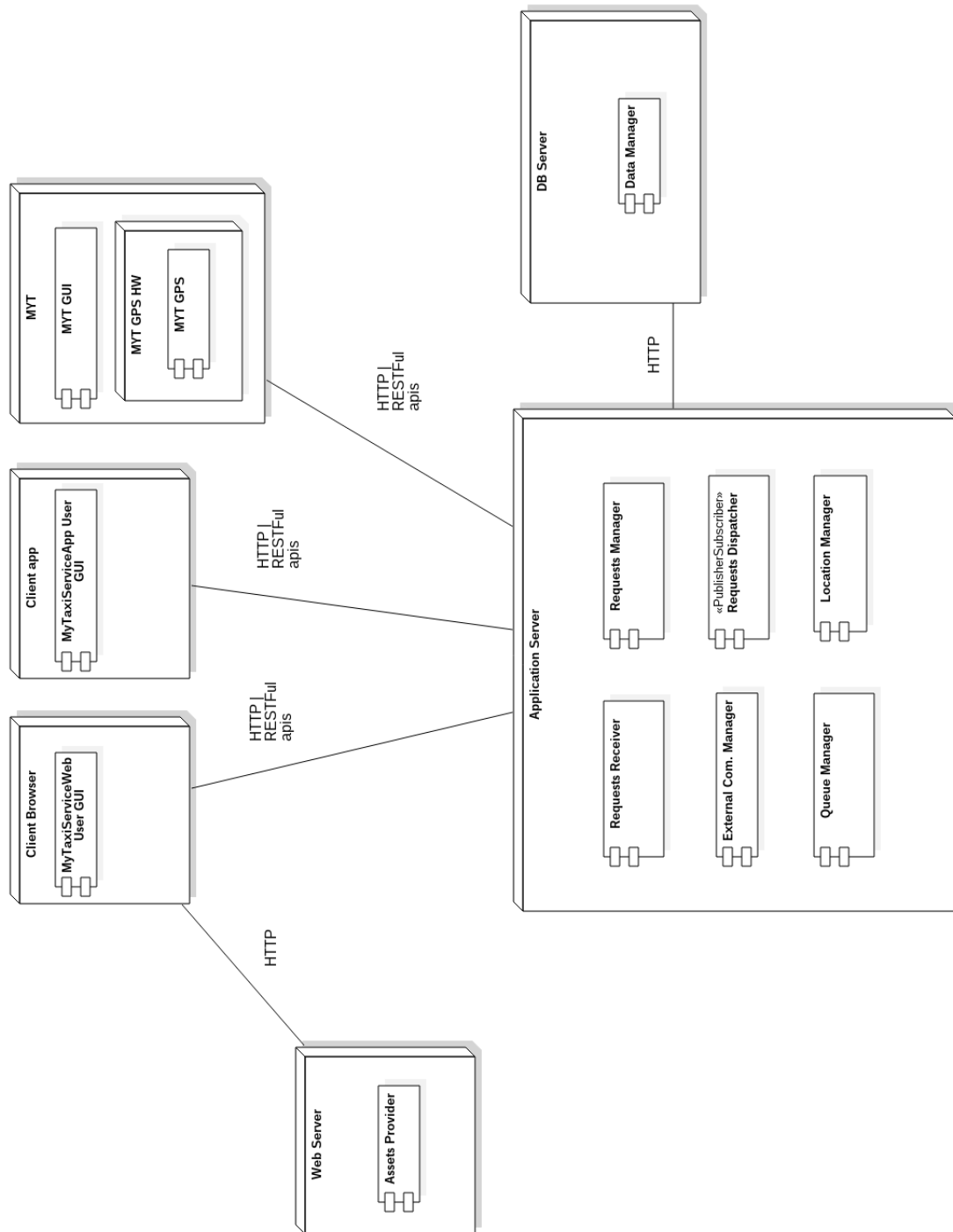
The business logic behind the system demands that users and mtaxies don't receive information only as a response to a request, but also as a non requested message from MyTaxiService(B). Maintaining a continuously opened connection between the client and the application server seems not an **efficient and sustainable strategy**, so is assumed that users and mtaxies check periodically if MyTaxiService(B) has any messages for them. This strategy is known as polling.

Follows a schematic diagram of the system architecture:



2.7 Deployment View

In this section is included a deployment diagram showing how high level components are mapped on the architecture tiers.



3 Algorithms Design

The following algorithm is run in the Queue Manager component periodically (each hour). Its objective is to try to redistribute taxis present in the city's zones in order to match the expected demand of taxis in each zone. To accomplish this objective the first part of the algorithm computes, for each zone, the expected demand by means of the following formula:

$$expectedTaxis = ((historicalAvg * w1 + currentAvg * w2) * zone.getWeight())$$

This formula is based both on historical data for the current day and time and the last hour data registered in the database. Each of these two terms is "weighted" ($w1, w2$) in order to determine how the old and new data contributes to the final result. The above formula takes also into account the size of the zone by means of a third weight typical of each zone ($zone.getWeight()$), the more the value is near to 1 the more the zone is bigger. This is necessary to consider the fact that smaller zones in general have a smaller number of requests wrt to bigger zones. For each zone the number of taxis already present is subtracted from the expectedTaxis value (for that zone). This operation produces the taxisToMove variable that if positive indicates a zone that has a "surplus" of taxis and if negative indicates a zone with a "deficit" of taxis. Queues are then divided into two separated lists (surplusQueues, deficitQueues) according to the value of taxisToMove. The lists are sorted according to the value of taxisToMove. The algorithm redistributes the taxis by initially considering the couple of queues which has the highest number of taxis in respectively in deficit/surplus. The algorithm proceeds by selecting queues with decreasing number of taxisToMove. The function returns when:

- The expected demand of taxis has been fulfilled
- There are no more taxis to fulfill the expected demand

```
1 // 0 < w1, w2 < 1
2 function computeDistribution(float w1, float w2) {
3     /*The following for cycle computes, for each zone, the number of taxis
4     *required (taxiesToMove < 0) or the number of taxis in surplus (
5         taxiesToMove > 0)
6     */
7     foreach(zone in zones){
8         //Get the average number of requests for the current day/hour
9         historicalAvg = getHistData(now.date, now.hour, zone);
10        //Get the number of requests registered in the last hour
11        currentAvg = getLastHourData(zone);
12        //Computes the expected number of taxis required in this zone for
13        the following hour
14        taxiesToMove = ((historicalAvg*w1+currentAvg*w2)*zone.getWeight())/2
15        - zone.getTaxis();
16        //Assigns the expected number of taxis to the relevant queue
17        zone.getQueue().setTaxisToMove(taxiesToMove);
18    }
19 }
```

```

16
17 //surplusQueues is the list of queues that has some taxies to offer
18 surplusQueues = [/*List of queues with taxiesToMove > 0*/];
19 //deficitQueues is the list of queues that requires some taxies
20 deficitQueues = [/*List of queues with taxiesToMove < 0*/];
21
22 //Sort surplusQueues by deacreasing taxiesToMove
23 Sort(surplusQueues);
24 //Sort deficitQueues by deacreasing taxiesToMove
25 Sort(deficitQueues);
26
27 //i iterates over surplusQueues
28 //j iterates over deficitQueues
29 while(i < surplusQueues.length && j < deficitQueues.length){
30
31     //If the current surplusQueues has more taxies to offer than the
        current deficitQueues
32     if(surplusQueues[i].getTaxiesToMove > deficitQueues[j].
        getTaxiesToMove){
33         deficitQueues[j].setTaxiesToMove(0);
34         surplusQueues[i].setTaxiesToMove(oldValue-deficitQueues[j].
        getTaxiesToMove);
35         //Select the next deficitQueues
36         j++;
37         //Send the order
38         foreach(taxi in surplusQueues[i].getTaxies()) {
39             moveTaxi(deficitQueues[j],taxi)
40         }
41     }
42 }
43 //If the current deficitQueues reuires more than what the current
        surplusQueues has to offer
44 else if(surplusQueues[i].getTaxiesToMove < deficitQueues[j].
        getTaxiesToMove){
45     serplusQueues[i].setTaxiesToMove(0);
46     deficitQueues[j].setTaxiesToMove(oldValue-surplus[j].
        getTaxiesToMove);
47     //Send the order
48     foreach(taxi in surplusQueues[i].getTaxies()) {
49         moveTaxi(deficitQueues[j],taxi)
50     }
51     //Select the next surplusQueues
52     i++;
53 }
54 //If the current deficitQueues requires exaclty what the current
        surplusQueues offers
55 else{
56     serplus[i].setTaxiesToMove(0);
57     deficit[j].setTaxiesToMove(0);
58     //Send the order
59     foreach(taxi in surplusQueues[i].getTaxies()) {
60         moveTaxi(deficitQueues[j],taxi)
61     }

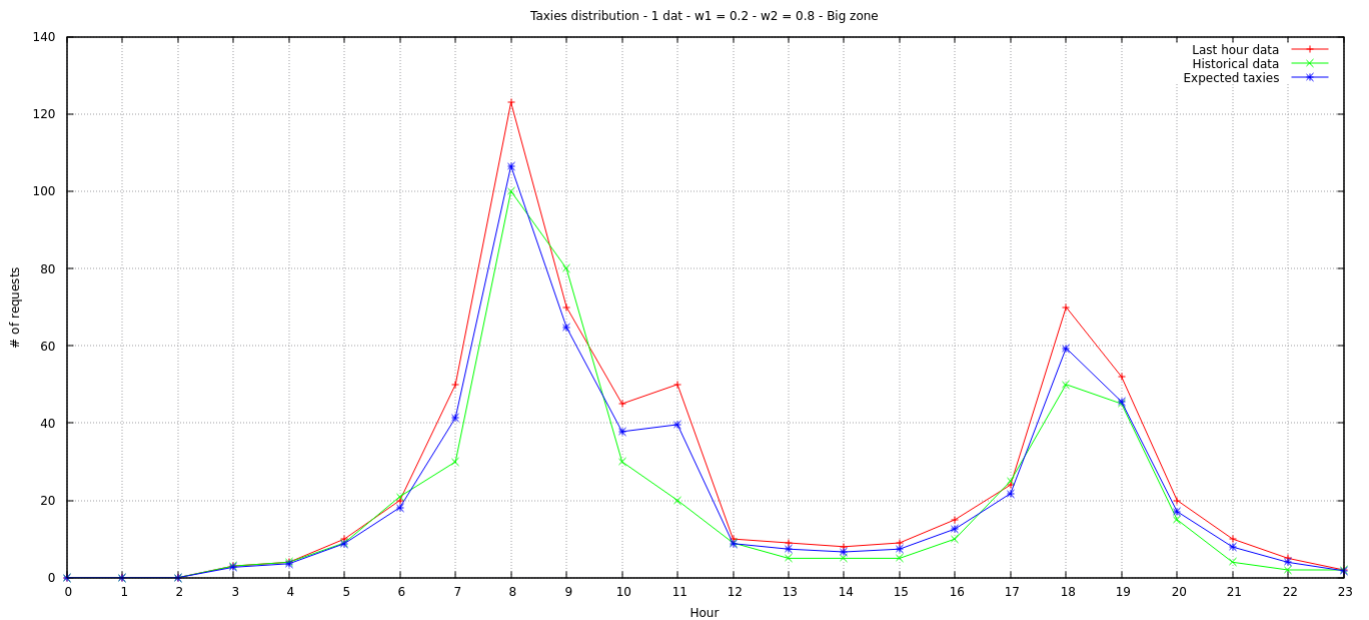
```

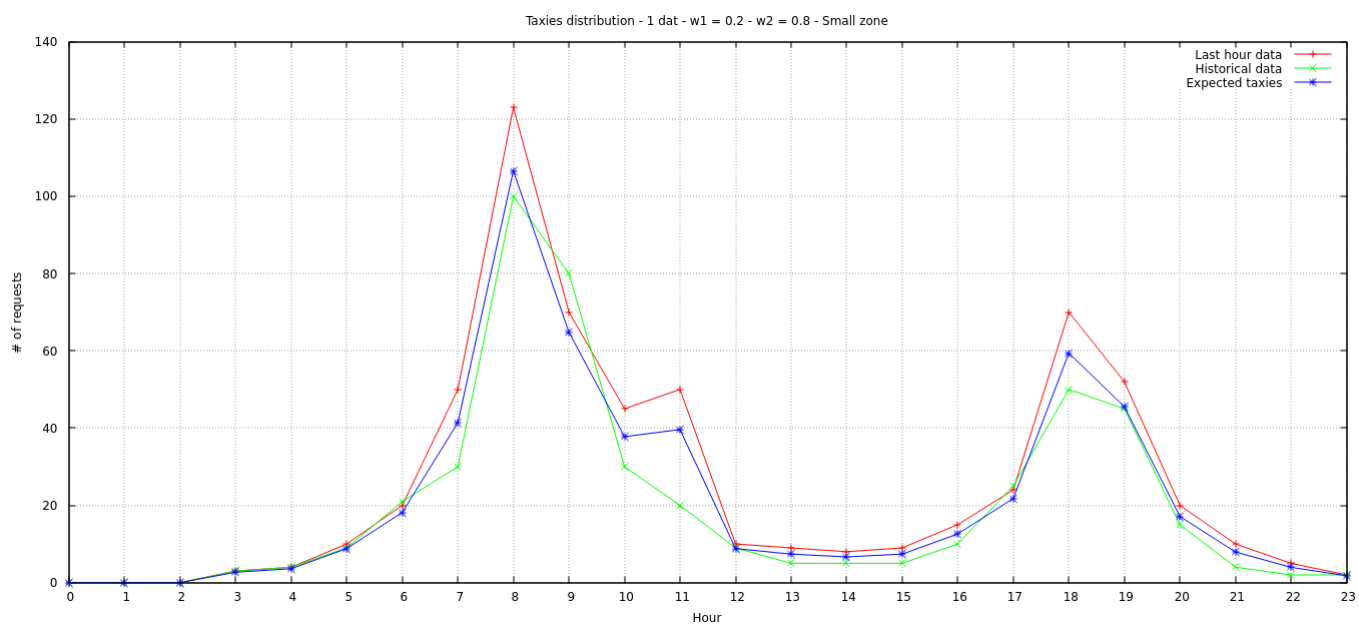
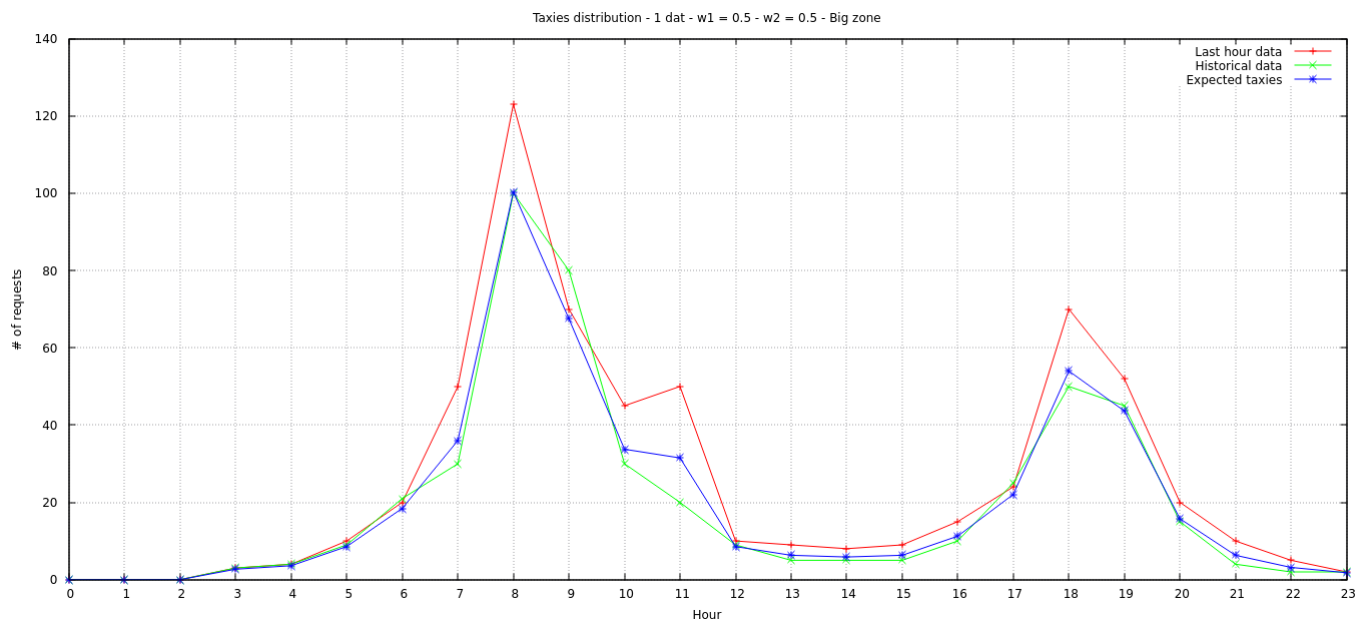
```

62
63     //Select the new surplus/deficitQueues
64     i++;
65     j++;
66 }
67 deficitQueuesurplusQueues[i].getTaxiesToMove
68 }
69 }
70
71 /* Adds taxi to endingQueue
72 * Sends a change zone order to taxi via a primitive provided by
73 * the Request Manager component
74 */
75 function moveTaxi(endingQueue, taxi) {
76     endingQueue.add(taxi);
77     sendChangeOrder(taxi);
78 }

```

Following several graphs showing the evaluation of the expectedTaxies function in three different situations:





4 User Interface Design

No new useful user interface needs to be specified in this section.

For a detailed description of the user interfaces, please refer to section 2.2.2 of the RASD v1.1 document.

5 Requirements Traceability

Component	Requirements
Dispatcher	(32) (34) (42)
Request Manager	(1a) (2a) (2b) (3a) (3b) (3c) (3e) (4a) (6) (15) (29) (34) (42)
External Services Manager	(19) (20)
Location Manager	(19) (20) (26) (30) (41) (42)
Queue Manager	(27) (28) (31) (33) (39)
Request Receiver	
Data Manager	(1a) (2a)(3b) (3c) (4a) (37) (38)

Table 1: Requirements Traceability matrix

6 Appendix

6.1 Tools

- **StarUML:** to draw the sequence diagrams.
- \LaTeX / **Atom:** to redact this document
- **GNUPlot:** to draw the **graphs**

6.2 Hours

- Andrea Sessa:
- Giorgio Pea: