Design **D**ocument

Version 1.1

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

4/01/2016

Contents

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 taxies 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 taxies 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
- **User:** Refers to either a logged registered user or a generic user(see RASD)
- **MyTaxiService(B):** see RASD
- **Administrator:** see RASD
- **Mtaxi bad beahvior**: see RASD

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
- **AWT:** Approximate waiting time to be picked up

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 Software Design Patterns

Model View Controller(MVC)
MyTaxiService's components are organized and designed such that they comply with the MVC philosophy;
components are classified in:

- Model Components: Those components which manage the persistent data of the application
- Controller Components: Those components which implement the business logic of the application
- View Components: Those components which manage the interaction between the application and its users(the general meaning of the term)

2.3 High level components and their interaction

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

Controller Components:

Dispatcher

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

1. For users : approximate waiting time for being picked up, the requested mtaxi's identification code
2. For mtaxies : zone change order, new ride request
3. For administrators : report of an accident, report of bad behavior

Request Manager

This component is in charge of:

1. 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)
2. Managing a situation of non fair distribution of mtaxies in the city's zones(this requires an interaction with the Queue Manager)

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

Location Manager

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

Queue Manager

This component is in charge of:

1. Distributing available mtaxies in city's zones and organize them in zone per zone queues
2. 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

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:

Data Manager

This component is in charge of the interaction with 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)

MyTaxiServiceAppUser GUI

This component manages the user interaction. This interaction refers to the MyTaxiService mobile app

MyTaxiServiceMYT GUI

This component manages the driver interaction. This interaction refers to the MYT device

MyTaxiServiceWebUser GUI

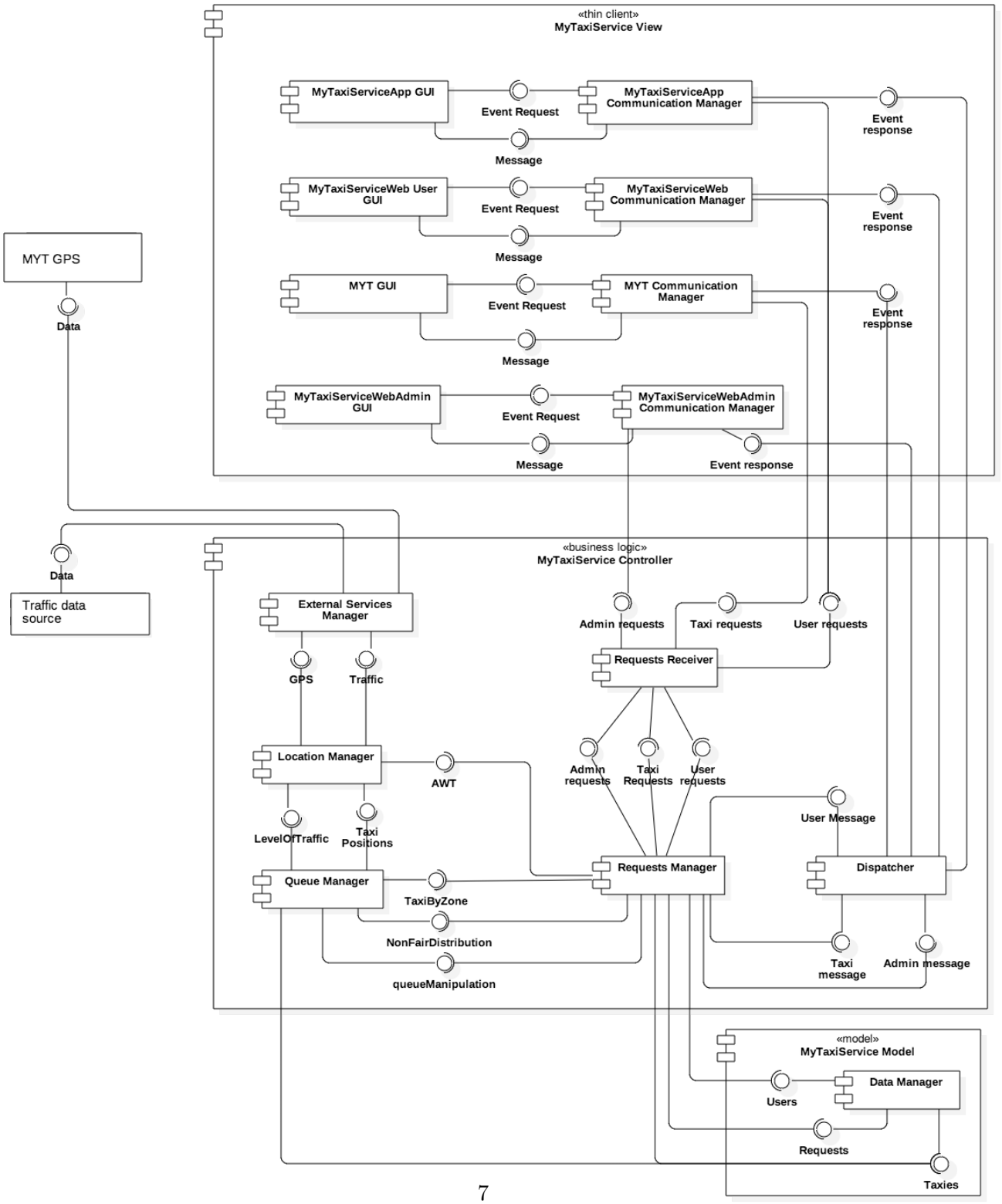
This component manages the user interaction. This interaction refers to the MyTaxiService web app

MyTaxiServiceWebAdmin GUI

This component manages the administrators interaction. This interaction refers to the MyTaxiService web app for administrators

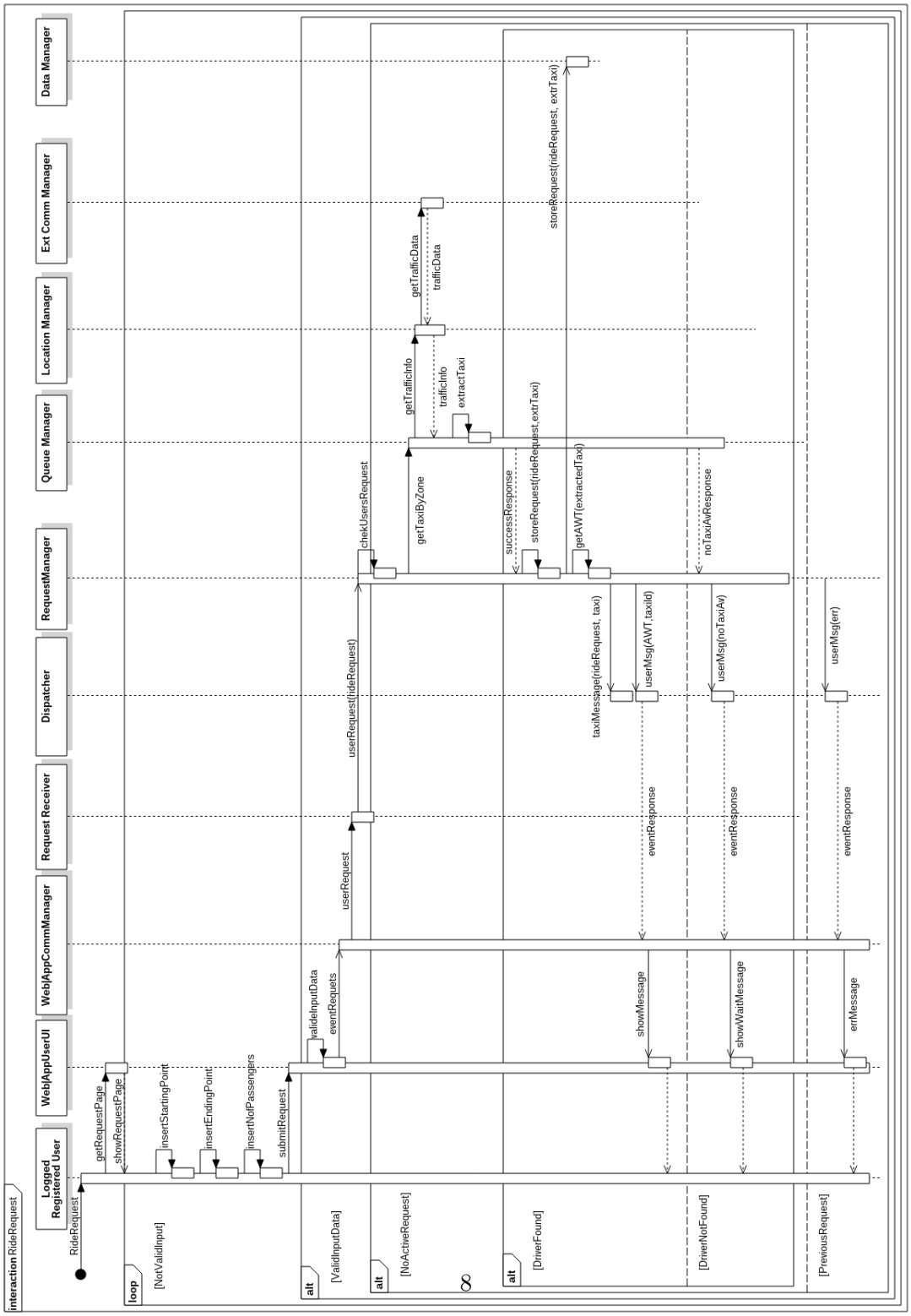
2.4 Component View

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

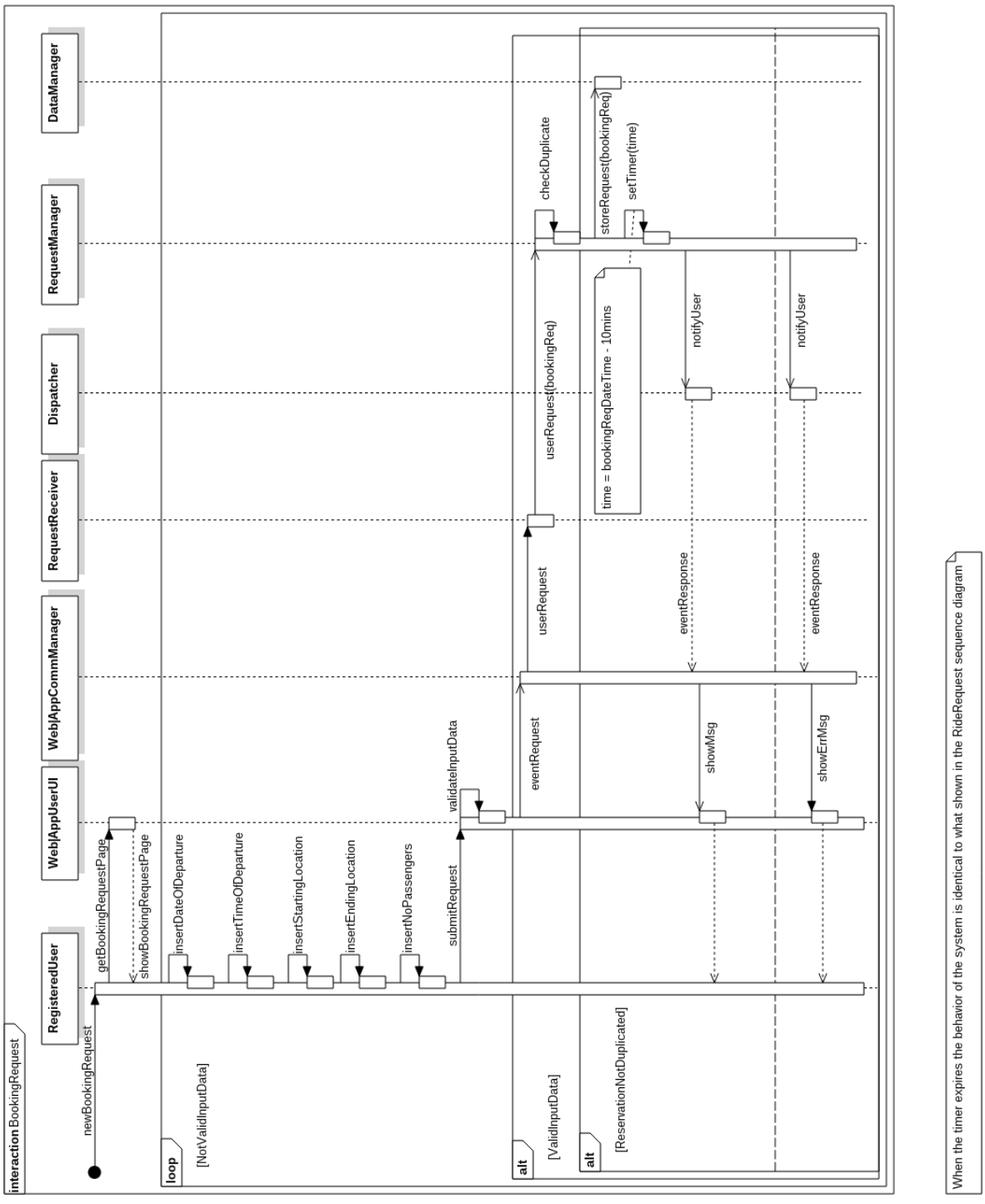


2.5 Runtime View

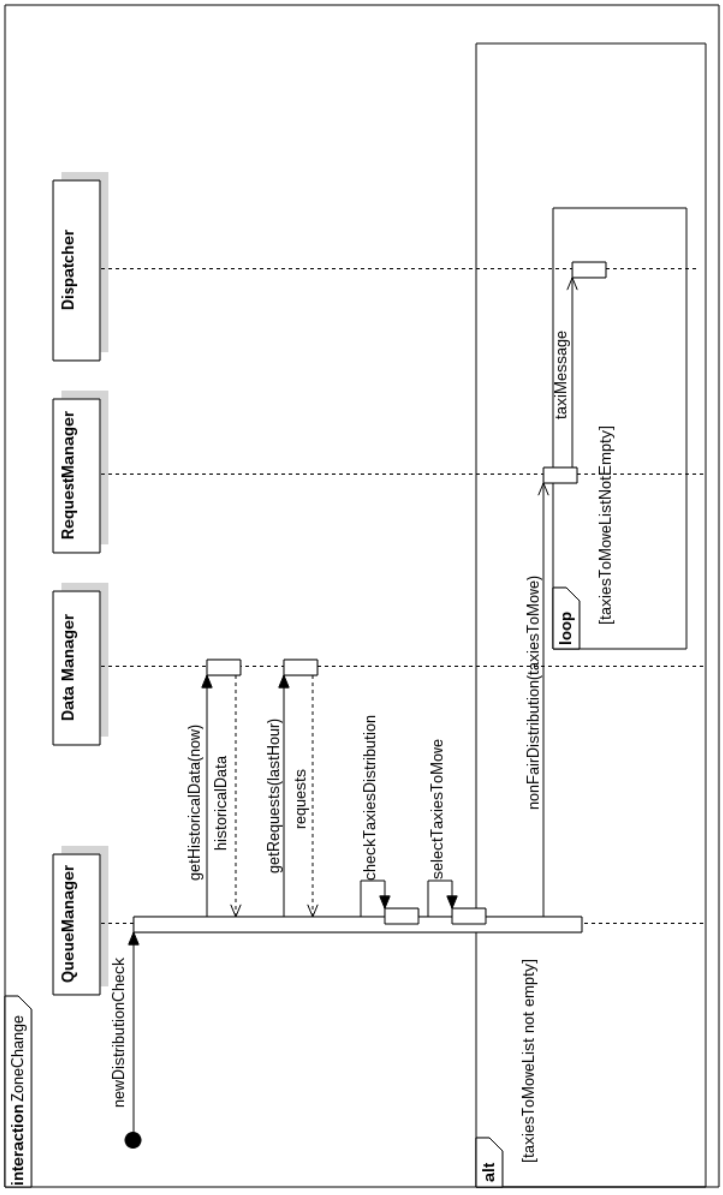
Components interaction in case of ride request.



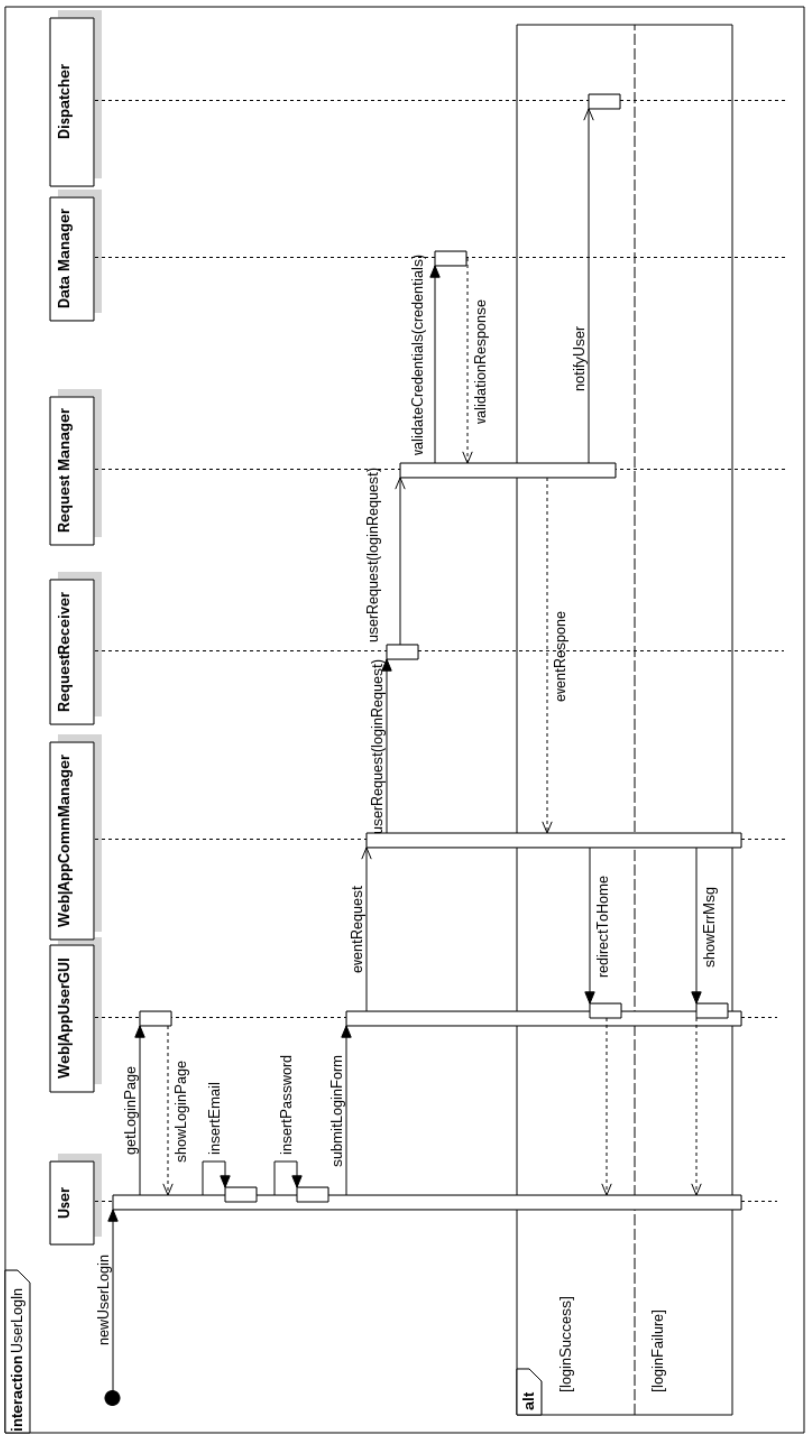
Components interaction in case of booking request.



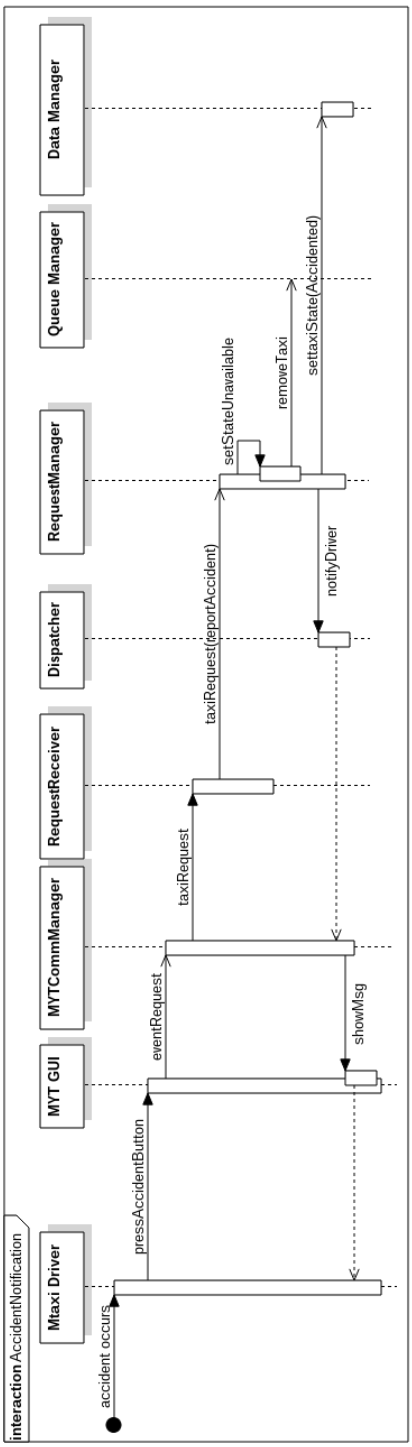
Components interaction when a new taxies redistribution is needed .



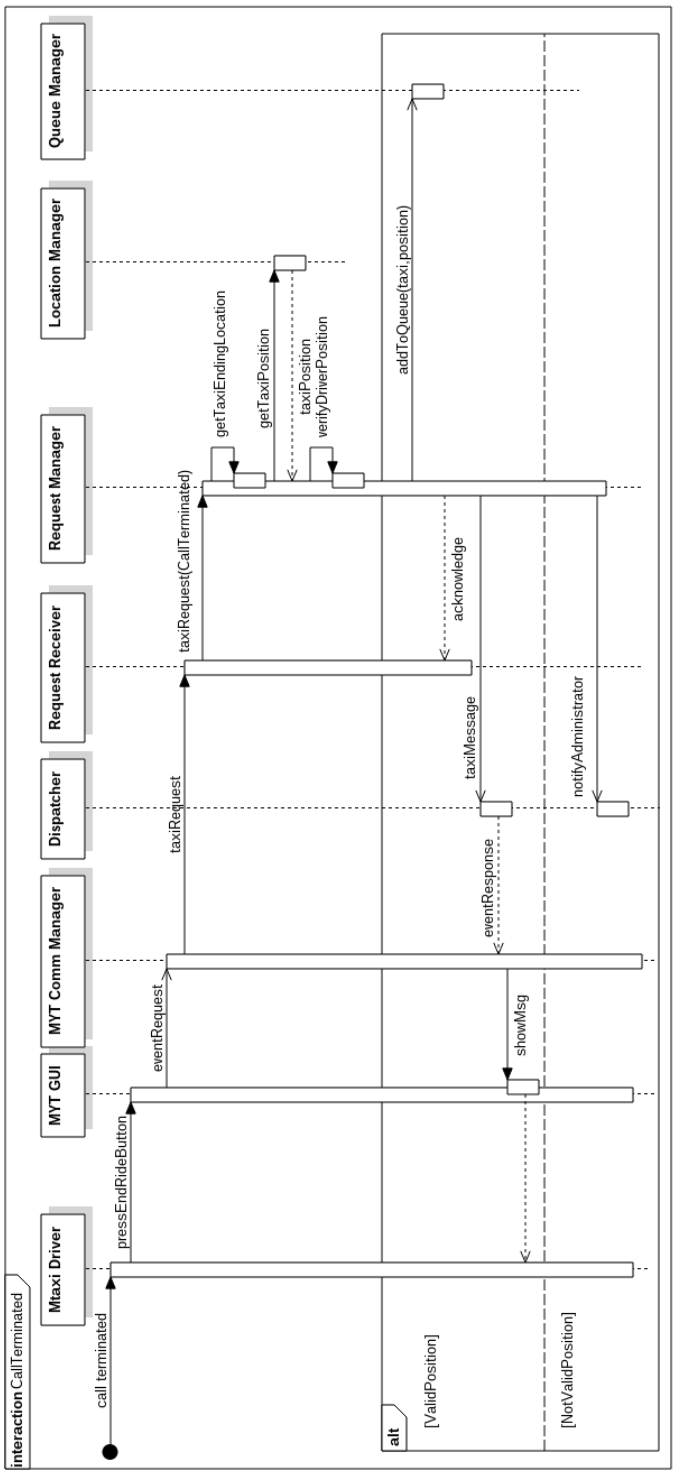
Components interaction in case of user login.



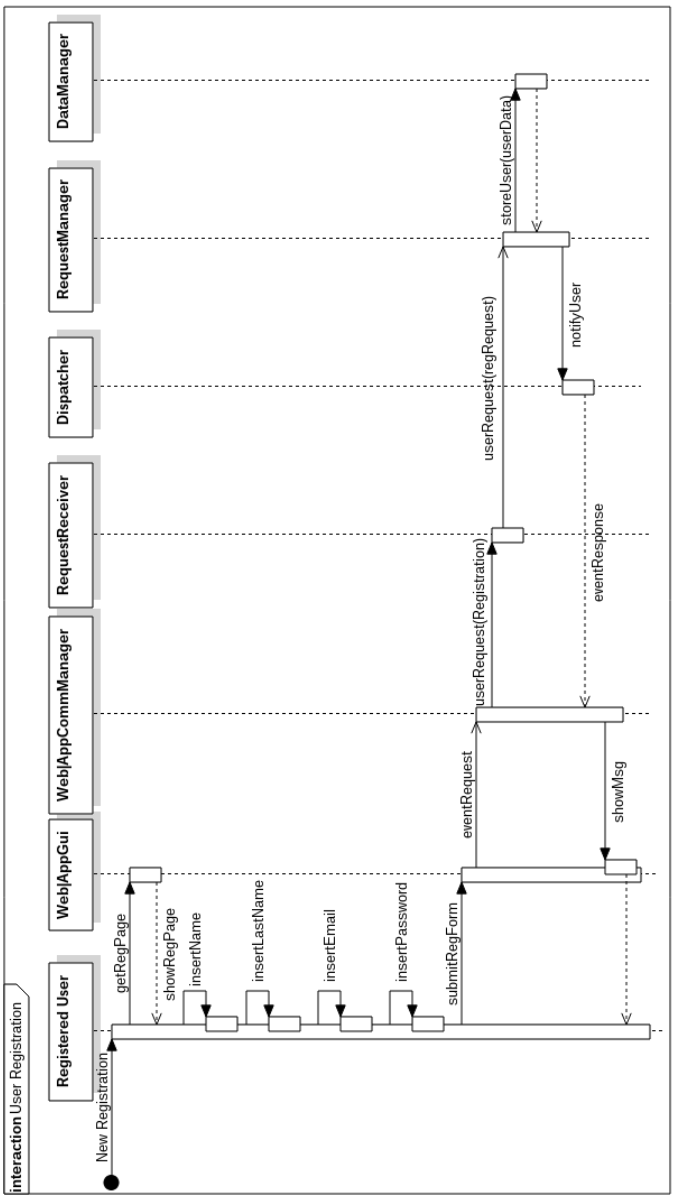
Components interaction in case of accident notification from a driver.



Components interaction when the driver notify a ride end.



Components interaction in case of user registration.



2.6 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:
   (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

   All communications are synchronous
4. Data Manager, via three interfaces:
   (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

   All communications are synchronous

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:

   (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

   All the communications are asynchronous
2. Data Manager, via one 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:

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

All the communications are synchronous

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 aboive. 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

View GUIs

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

View Communication Managers

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

2.7 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:

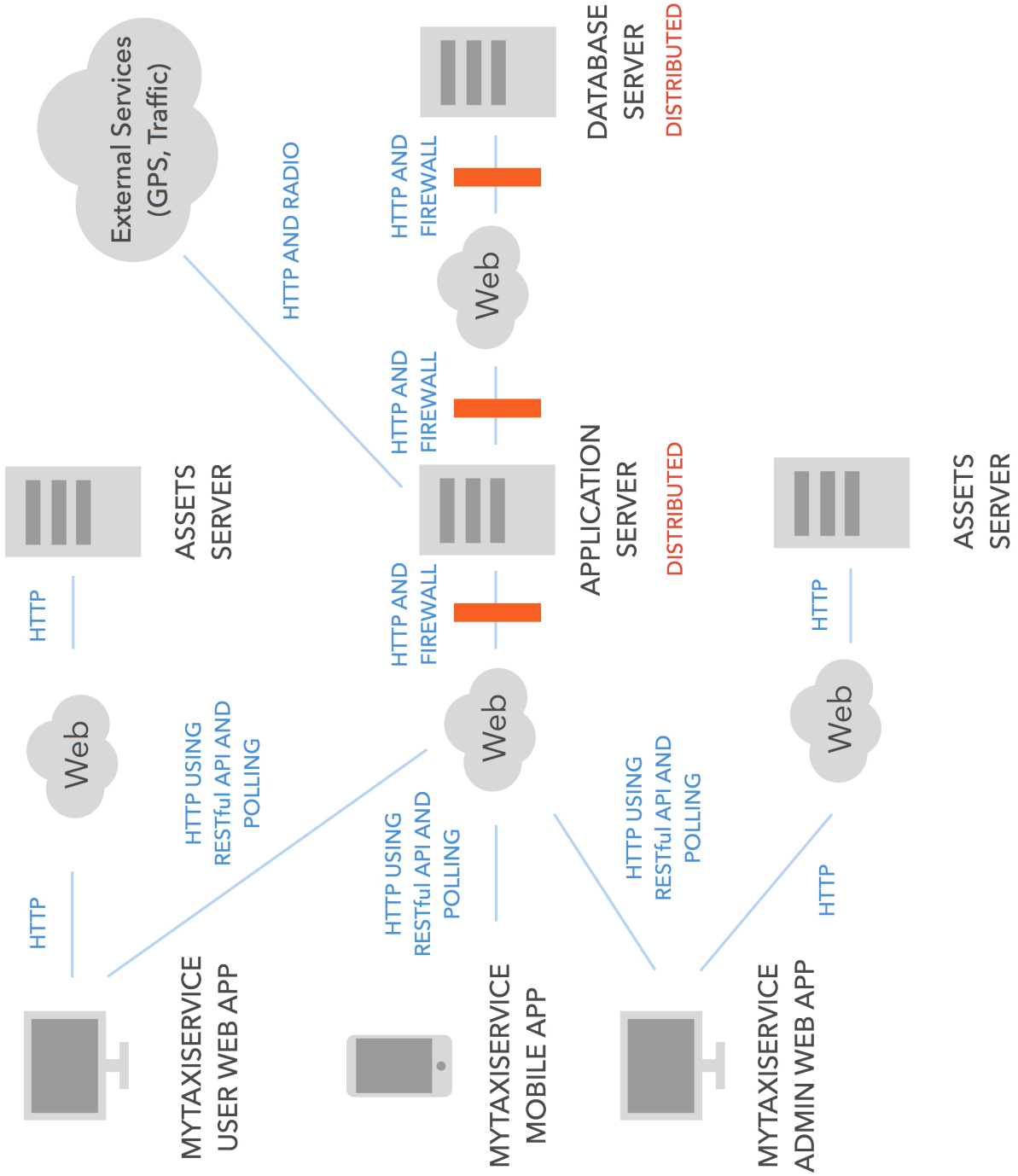
1. A tier(view) which include:
   (a) Smartphones in which the MyTaxiService mobile app is installed
   (b) Smartphones and computers trough which MyTaxiService User website is accessible
   (c) Smartphones and computers trough which MyTaxiService Admin website is accessible
   (d) MYT devices trough which mtaxi drivers can manage ride requests
2. A tier(controller) that is composed by an application server(can be distributed) that manages incoming requests from users, mtaxies and administrators by applying the bussines logic at the base of the whole system
3. A tier(model) 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 websites 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 business logic of the application is condensed in the application server, which is, in this way, a "fat" server.

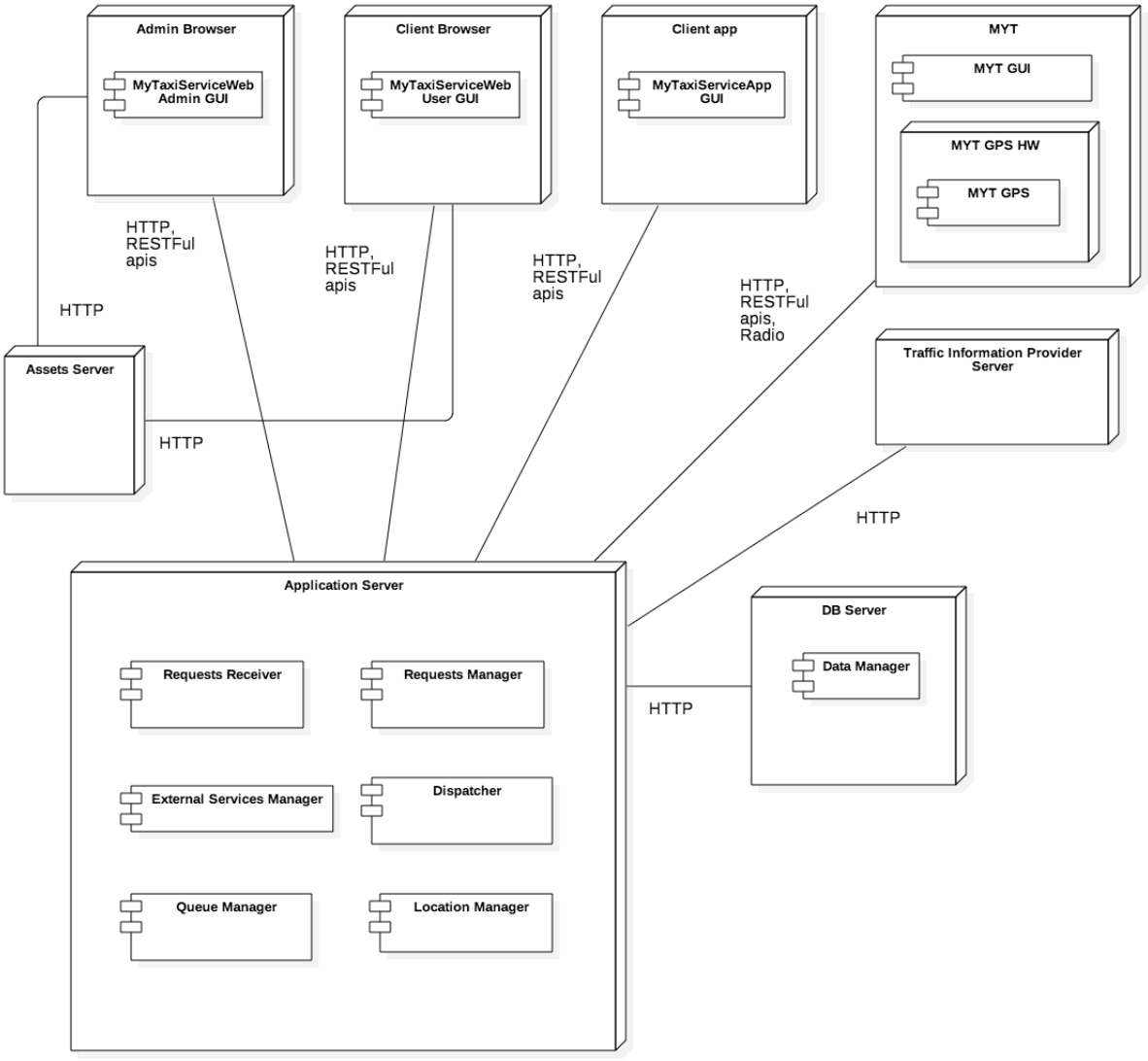
The role of polling
The business logic behind the system demands that users, mtaxies and administrators don't receive information only as a response to a request, but also as a non requested message from MyTaxiService(B). Mantaining a continuosly opened connection between the client and the application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

A schematic diagram of the system's architecture:



2.8 Deployment View

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



3 Algorithms Design

Distribution Checker
The following algorithm is run in the Queue Manager component periodically(each hour). Its objective is to try to reditribute taxies present in the city's zones in order to match the expected demand of taxies 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:

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

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 this 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 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 request wrt to bigger zones. For each zone the number of taxies already present is subtracted from the expectedTaxies value(for that zone). This operation produces the taxiesToMove variable that if positive indicates a zones that has a "surplus" of taxies and if negative indicates a zone with a "deficit" of taxies. Queues are then divided into two separated lists(surplusQueues, deficitQueues) according to the value of taxiesToMove. The lists are sorted according to the value of taxiesToMove. The algorithm reditribute the taxies by initially considering the couple of queues which has the highest number of taxies in respectively in deficit/surplus also taking into account the distance between the two exchanging zones. The algorithm proceeds by selecting queues with deacreasing number of taxiesToMove. The function return when:

- The expected demand of taxies has been fullfilled
- There are no more taxies to fullfill the expected demand

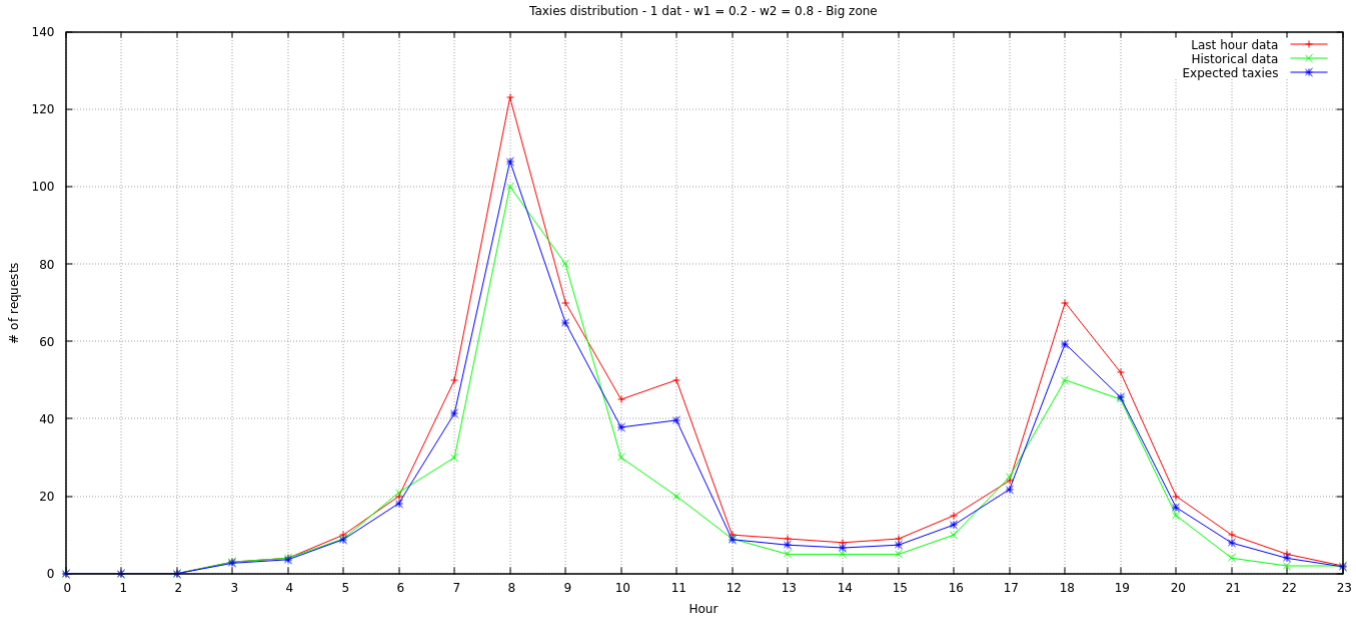
```
1  // 0 < w1, w2 < 1
2  /* This constant represents the maximum admissable distance between
3   * two exchanging zones(in kilometers)
4   */
5  const MAX_DISTANCE = 1;
6
7  // 0 < w1, w2 < 1
8  function computeDistribution(float w1, float w2) {
9    /*The following for cycle computes, for each zone, the number of taxies
10    *required(taxiesToMove < 0) or the number of taxies in surplus(
          taxiesToMove > 0)
11    */
12   foreach(zone in zones){
13     //Get the average number of requests for the current day/hour
14     historicalAvg = getHistData(now.date, now.hour, zone);
```

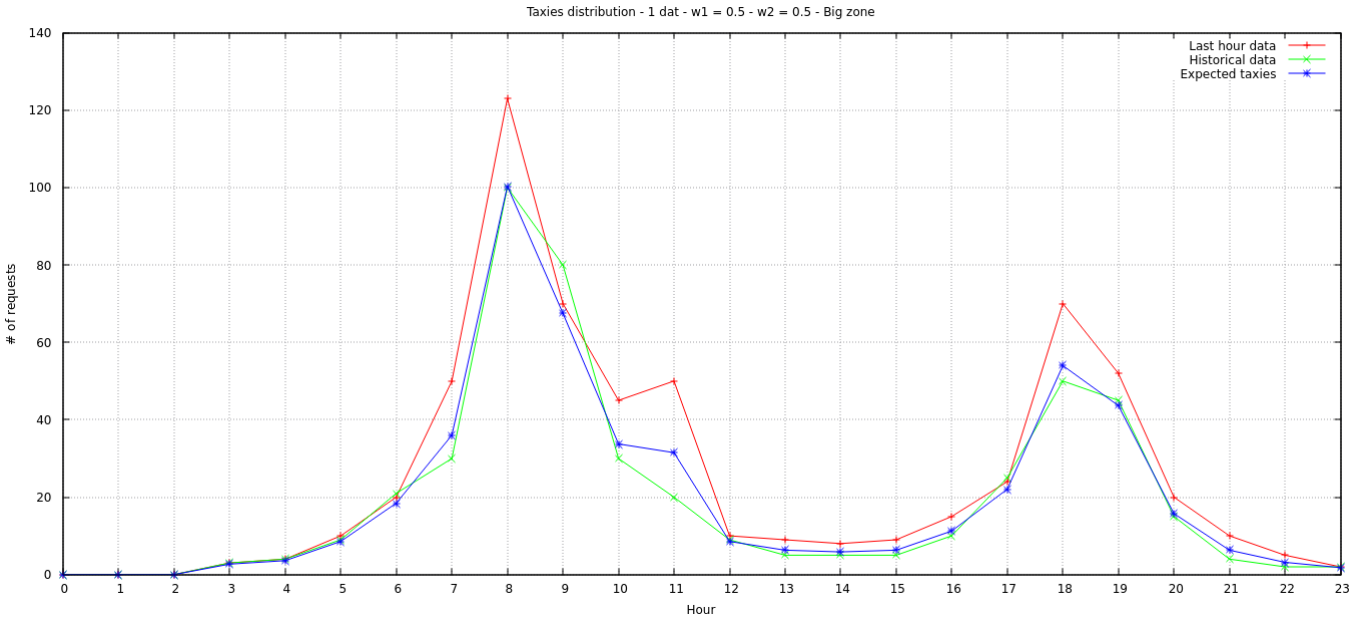
```
15      //Get the number of requests registered in the last hour
16      currentAvg = getLastHourData(zone);
17      //Computes the expected number of taxies required in this zone for
            the following hour
18      taxiesToMove = ((historicalAvg*w1+currentAvg*w2)*zone.getWeight())/2
            - zone.getTaxies();
19      //Assigns the expected number of taxies to the relevant queue
20      zone.getQueue().setTaxiesToMove(taxiesToMove);
21    }
22
23    //surplusQueues is the list of queues that has some taxies to offer
24    surplusQueues = [/*List of queues with taxiesToMove > 0*/];
25    //deficitQueues is the list of queues that requires some taxies
26    deficitQueues = [/*List of queues with taxiesToMove < 0*/];
27
28    //Sort surplusQueues by deacreasing taxiesToMove
29    Sort(surplusQueues);
30    //Sort deficitQueues by deacreasing taxiesToMove
31    Sort(deficitQueues);
32
33    //i iterates over surplusQueues
34    //j iterates over deficitQueues
35    while(i < surplusQueues.length  && j < deficitQueues.length){
36
37      /*Tries to minimize the distance between the two exchanging zones
38      * by selecting a new deficitQueue. The procedure is repeated at most
39      * 3 times(count variable).
40      */
41      if(distance(deficitQueues[i], surplusQueues[j]) > MAX_DISTANCE) {
42        count = 0;
43        actualQueue = deficitQueues[i];
44        do {
45          ShiftOne(deficitQueue, deficitQueues);
46          count++;
47        }while(distance(deficitQueues[i], surplusQueues[j]) > MAX_DISTANCE
              && count < 3);
48        //Sorts deficitQueues starting from the position j
49        PartialSort(deficitQueues,j);
50      }
51
52      //If the current surplusQueues has more taxies to offer than the
            current deficitQueues
53      if(surplusQueues[i].getTaxiesToMove > deficitQueues[j].
            getTaxiesToMove){
54        deficitQueues[j].setTaxiesToMove(0);
55        surplusQueues[i].setTaxiesToMove(oldValue-deficitQueues[j].
            getTaxiesToMove);
56        //Select the next deficitQueues
57        j++;
58
59        //Send the order
60        foreach(taxi in surplusQueues[i].getTaxies()) {
61          moveTaxi(deficitQueues[j],taxi)
```

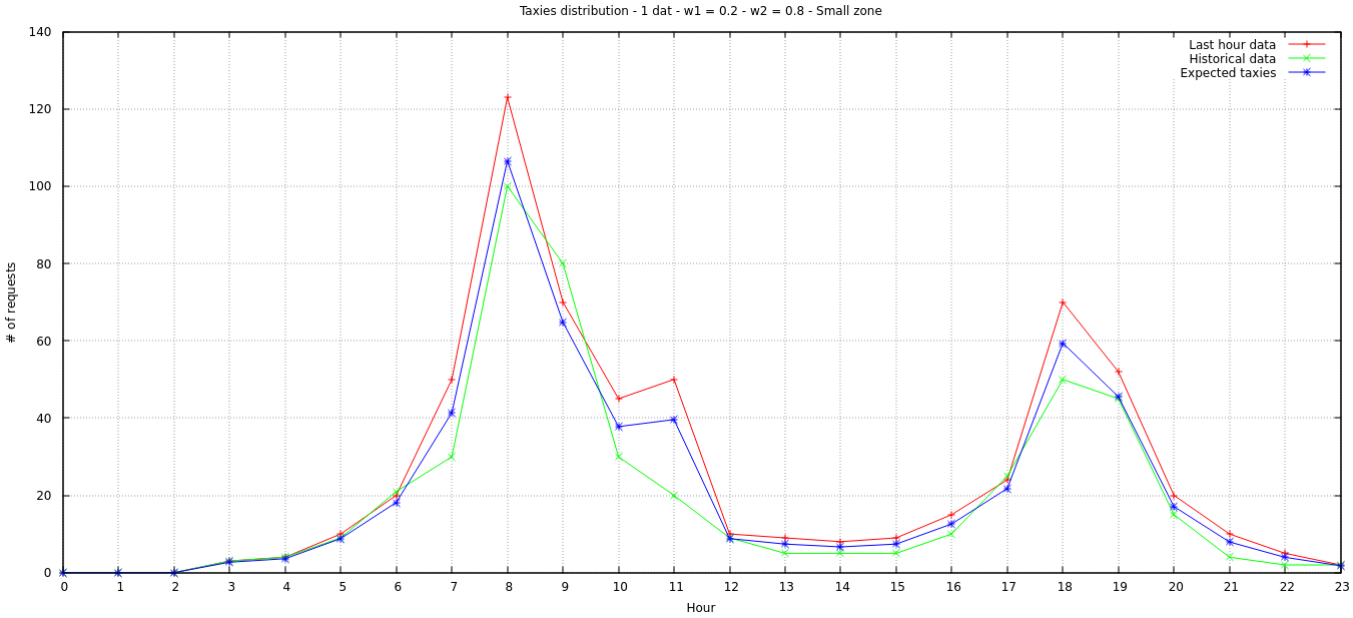
```
62        }
63
64      }
65      //If the current deficitQueues reuires more than what the current
            surplusQueues has to offer
66      else if(surplusQueues[i].getTaxiesToMove < deficitQueues[j].
            getTaxiesToMove){
67        serplusQueues[i].setTaxiesToMove(0);
68        deficitQueues[j].setTaxiesToMove(oldValue-surplus[j].
              getTaxiesToMove);
69        //Send the order
70        foreach(taxi in surplusQueues[i].getTaxies()) {
71          moveTaxi(deficitQueues[j],taxi)
72        }
73        //Select the next surplusQueues
74        i++;
75      }
76      //If the current deficitQueues requires exaclty what the current
            surplusQueues offers
77      else{
78        serplus[i].setTaxiesToMove(0);
79        deficit[j].setTaxiesToMove(0);
80        //Send the order
81        foreach(taxi in surplusQueues[i].getTaxies()) {
82          moveTaxi(deficitQueues[j],taxi)
83        }
84
85        //Select the new surplus/deficitQueues
86        i++;
87        j++;
88      }
89      deficitQueuesurplusQueues[i].getTaxiesToMove
90    }
91  }
92
93  /* Adds taxi to endingQueue
94   * Sends a change zone order to taxi via a primitive provided by
95   * the Request Manager component
96   */
97  function moveTaxi(endingQueue, taxi) {
98    endingQueue.add(taxi);
99    sendChangeOrder(taxi);
100 }
101
102 /*
103 * Moves the swappingQueue at the end of queuesList
104 */
105 function ShiftOne(swappingQueue, queuesList) {
106   queuePosition = queuesList.indexOf(swappingQueue);
107   Queue tmpQueue = queuesList[queuePosition];
108   //Shift the queues
109   for(int i = queuePosition; i < queuesList.length; i++) {
110       queuesList[i]  = queuesList[i+1];
```

```
111     }
112     queuesList[queuesList.length-1] = tmpQueue;
113 }
```

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






Taxies distribution - 1 dat - w1 = 0.2 - w2 = 0.8 - Small zone
Last hour data
Historical data
Expected taxies
# of requests
Hour
0
20
40
60
80
100
120
140
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

Mtaxies Picker Follows a second algorithm to model the selection of a taxi driver to fullfill a ride/booking request.

```
1  /**
2    Checks if there exists a mtaxi in the queue relative to the specified
         zone.
3    This mtaxi is checked to be at most at the limitTo position from the
         top of the queue above mentioned.
4    This taxi is checked to be  in the conditions of fullfilling a ride
         request(that means that the traffic condition
5    relative to its location is good)
6    @param zone The zone relative to the ride request to be fullfilled
7    @param limitTo The upper bound limit on the position of a mtaxi from
         the top of its queue so that it can be selected
8    to fullfill a ride request
9    @return firstInQueue The mtaxi selected following the criteria above
         mentioned
10   @return -1 If it does not exists a mtaxi that can be selected using the
          criteria above mentioned
11
12 **/
13 function checkTaxi(zone,limitTo){
14   //Gets the queue relative to the zone specified
15   var queue = findQueueByZone(zone);
16   var i=0;
17   var firstInQueue = queue.dequeue();
18   //The queue must be non empty, the amount of mtaxies in bad traffic
         conditions
19   //must not be equal to the number of all mtaxies in the queue, i must
         not consider
20   //mtaxies whose position from the top of the queue exceeds the given
         limitTo
21   while(!queue.isEmpty && i<limitTo && queue.puttedBottom !== queue.
         length){
22     //Checks traffic conditions
23     if(LocationManager.checkTrafficByTaxi(firstInQueue)!== 'HIGH'){
24       return firstInQueue;
25     }
26     else{
27       //Puts a mtaxi in bad traffic condition on the bottom of the queue
28       queue.putInBottom(firstInQueue);
29       i++;
30     }
31   }
32   return -1;
33 }
34
35
36 /**
37 Checks if there exists a mtaxi in the queue relative to the specified
       zone or to
38 the zones adjacent to the one specified or to the zones adjacent to the
```

```
    ones adjacent to the one initially
39  specified(depth = 2)
40  @param zone The zone relative to the ride request to be fullfilled
41  @param firstLimitTo The upper bound limit on the position of a mtaxi from
        the top of its queue so that it can be selected
42  to fullfill a ride request
43  **/
44  function searchTaxi(zone,limitTo){
45    //Helper queues
46    firstHelperQueue = [];
47    secondHelperQueue = [];
48    thirdHeleprQueue = [];
49
50    //depth = 0
51    //checking mtaxies
52    result = checkTaxi(zone,limitTo);
53
54    if(result !== -1){
55      return result;
56    }
57    else{
58      //depth = 1
59      //fills the first and second helper queue with the zones
60      //adjacent to the given one
61      foreach(z in zone.adjacentZones){
62        firstHelperQueue.push(z);
63        secondHelperQueue.push(z);
64      }
65      //extracts the adjacent zones and checks them for mtaxies
66      while(firstHelperQueue.length !== 0){
67        zone = firstHelperQueue.pop();
68        result = checkTaxi(zone,limitTo);
69        if(result !== -1){
70          return result
71        }
72      }
73      //depth = 2
74      while(secondHelperQueue.length !== 0){
75        zone = secondHelperQueue.pop();
76        //fills the third helper queue with the zones adjacent to the ones
77        //adjacent to the initial one
78        foreach(z in zone.adjacentZones){
79          thirdHeleprQueue.push(z)
80        }
81        //extracts the adjacent zones and checks them for mtaxies
82        while(thirdHeleprQueue.length !== 0){
83          zone = thirdHeleprQueue.pop();
84          result = checkTaxi(zone, limitTo);
85          if(result !== -1){
86            return result
87          }
88        }
89        //empties the third helper queue
```

```
90          thirdHeleprQueue.flush();
91        }
92        //no taxi found also going deep in the analysis
93        return -1;
94      }
```

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) (17) (25) (6a) (6b)(6c) (44) (45)(46)(47)
External Services Manager	(19) (20)
Location Manager	(19) (20) (26) (30) (41) (42)
Queue Manager	(27) (28) (31) (33) (39)
Request Receiver	(1a) (2a) (3a) (3b) (3c) (3d) (3e) (4a) (5a) (5b) (5c) (5d) (6a) (6b) (6c) (44) (45)(46)(47)
Data Manager	(1a) (2a)(3b) (3c) (4a) (37) (38) (47)
MyTaxiService User Web/App GUI/CommManager	(1a) (2a) (2b) (3a) (3b) (3c) (3d) (3e) (7) (8) (10) (11) (12) (18) (21) (22) (23) (24)
MyTaxiServiceMYT GUI/CommManager	(1a) (2a) (2b) (3a) (3b) (3c) (3d) (3e) (7)
MyTaxiServiceWebAdmin GUI/CommManager	(6a)(6b)(6c) (44) (45)(46)(47)

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: 22 hours
- Giorgio Pea: 20 hours

6.3 Revision

Date	Description	Authors
01/01/2016	Improved the taxi distribution algorithm	Sessa
04/01/2016	Improved architecture	Pea