**3. Introduction**

* 1. **Purpose**  
     This is the Design Document for the MyTaxiService system. The aim of this document is to provide a detailed description of the system’s architecture, its main components and their interaction. It is mainly addressed to developers and testers.
  2. **Scope**

The aim of this project is to develop a system to improve the taxi service of Milan, accessible both via web and mobile applications. For further details see RASD, section 1.2

* 1. **Definitions, Acronyms and Abbreviations**
     1. Definitions  
        See RASD.
     2. Acronyms

DD: Design Document

RASD: Requirement Analysis and Specification Document

Java EE: Java Enterprise Edition

HTTP: HyperText Transfer Protocol

EJB: Enterprise Java Bean

JPA: Java Persistence API

JAX-RS: Java API for RESTful Services

P2P: peer-to-peer

MVC: Model-View-Controller

API: Application Programming Interface

* + 1. Abbreviations

app: application

* 1. **Reference Documents**

RASD.pdf

* 1. **Document Structure**

We provide here the overall structure of this document:

* In section 1 the problem is introduced;
* In section 2 the architecture of the final system is proposed, focusing on the main components and their interaction;
* In section 3 the main algorithms used by the application are presented;
* In section 4 the user interfaces are detailed furtherly;
* In section 5 the requirements defined in the RASD document are mapped to components of the proposed architecture.

**2. Architectural Design \***

**2.1 Overview**

This part of the document provides a detailed description of the system architecture.   
Section 2.2 illustrates the components at high level, both from a physical and a logical perspective.  
Sections 2.3 and 2.4 furtherly detail the components, while section 2.5 focuses on their interaction.  
Section 2.6 TODO  
Sections 2.7 and 2.8 explain and justify design choices.  
  
**2.2 High level components and their interaction**The architecture contemplates a central system, which provides all the required services and manages data, and three types of clients:

* Web application for passengers, entirely accessible via a web browser;
* Mobile app for passengers, an hybrid app accessing the same contents as the web application;
* Mobile app for drivers, a native application.

The system will be implemented using Java EE technologies.

2.2.1 Hardware architecture overview

PICTURE

Clients access the services through the Internet. Client side devices are:

* Mobile phones running a standard operating system (Android, iOS or Windows Phone), on which the mobile applications are installed;
* Any device running a web browser, from which the web application is accessible.

Server side devices are:

* Router: connects the central system to the internet.
* Firewall: provides protection to the back-end system.
* Application server: a machine providing all the functionalities of a web server and hosting the business logic. It handles all application logic: client requests, queue management, taxi allocation...  
  It communicates both with clients and with the Database Server. Considering the nonfunctional requirements specified in the RASD document, in particular availability and performance, a single machine could not be enough to properly manage the whole process. Thus, a cluster of two dedicated machines will host replicated instances of the server.
* Load balancer: redirect incoming requests to the machines in the cluster, in order to maximize performances.
* Database Server: the DBMS storing the system's data.

2.2.2 Software architecture overview

The central system is 3-tiered. A diagram showing the tiers involved is provided below:

(PICCIA)

The server-side tiers are:

-Web tier: receives HTTP requests from the web application and the passenger mobile app, processes them by invoking services on the business tier and returns HTTP responses to the client.

-Business tier: includes several components providing the system functionalities, to be invoked both by the web tier and, remotely, by the driver mobile app. Some of these components are accessible as web services, so that external developers may use them in their application. The components access the database to store and retrieve data. A detailed description of this tier's components is provided in section 2.3.

-Data tier: receives query requests from the business tier and processes them.

The client tier includes:

-Web application: a set of dynamic web pages accessed through a browser, which communicates through HTTP with the web tier.

-Passenger app: an hybrid mobile application (that is, an application directly displaying web contents), which communicates through HTTP with the web tier. Considering that the web application must also communicate using HTTP with the web tier and its functionalities are the same of those provided by the mobile one, an hybrid app has been chosen instead of a native. This choice will also simplify the deployment on different mobile operating systems.

-Taxi driver app: a native mobile application, which directly communicates with the business tier components according to a peer-to-peer paradigm (specified in section 2.3 and 2.7).

The software layers are distributed between the specified tiers according to the distributed presentation paradigm:

(PIC.CIA)

The application server, indeed, assembles the web pages, i.e. the client's view. Thus, a part of the GUI is handled by the application server. Note that this concerns only the web application and the passenger app. The driver app takes care of all the GUI layer by itself, thus the paradigm is, in this case, remote presentation.

2.2.3 Implementation technologies

As already mentioned, the system will be implemented with Java EE. More precisely, the following technologies will be used:

-Application Server: Glassfish 4.1.1;

-Database Server: MySQL;

-Web tier: JavaServer Faces;

-Business tier: EJB (for the components), JAX-RS (for the web services), JPA (for the database communication);

**2.3 Component View**

The business tier includes the following components:

-Login manager: handles all login requests. It distinguishes passenger logins from driver logins. The driver login function can be invoked remotely. The component checks the validity of user data, sets the new state accordingly and returns the result. It also provides a function to check whether a given user is already logged in.

-Registration manager: handles registrations of new passengers to the system. It can be divided in two sub-components. The first one generates validation codes and sends them (via SMS) through an external web service and checks the validity of submitted codes. The second one checks the validity of user data in a registration request, creates new users and returns the result to the invoker (the web tier).

-Logout manager: handles the automatic logout (due to timeout) of taxi drivers.

-Taxi call manager: handles all taxi calls. The component checks the validity of input data and forward valid requests to the Taxi allocation manager. This component is also invoked by a public web service, in order to provide the taxi call function to external developers.

-Taxi reservation manager: handles all taxi reservations. The component checks the validity of input data and stores valid requests into the database. This component is also invoked by a public web service, in order to provide the taxi reservation function to external developers.

-Taxi allocation manager: processes call requests and periodically checks the database for reservations whose meeting time is in ten minutes. In both cases, forwards the request to the corresponding queue (in the queue manager).

-Queue manager: manages all the queues. This component handles the insertion of a driver in a queue (after he sets his state to FREE) and his removal (after his state is set to BUSY). Moreover, it processes calls forwarding to driver, according to the algorithm mentioned in the RASD document (see section 3 for more details).

-Communication manager: handles the message exchanges with the taxi drivers. The component receives messages and sends responses. It also sends asynchronous requests to the drivers. Thus, the overall architecture of this sub-system is p2p (see section 2.7 for further details).

-GPS manager: manages the communication with all taxi GPSs.

**2.4 Deployment view**

**2.5 Runtime view**

**2.6 Component interfaces**

**Login Manager**

Login

* loginPassenger(): performs passenger login. INPUT: cell phone number and password. OUTPUT: the result of the operation (either success or error).
* loginDriver(): performs driver login. INPUT: cell phone number, taxi ID, password. OUTPUT: the result of the operation (either success or error).
* isLoggedIn(): checks whether a given user is already logged in. INPUT: cell phone number. OUTPUT: either true (the user is already logged in) or false (viceversa).

**Registration Manager**

Registration

* register(): performs passenger registration. INPUT: cell phone number, password, name, surname. OUTPUT: the result of the operation (either success or error).

Validation

* generateCode(): generates a random validation code. INPUT: none. OUTPUT: the generated code.
* checkCode(): checks whether the input code corresponds to the one generated for the given passenger. INPUT: cell phone number, code. OUTPUT: either true (the code is valid) or false (viceversa).

**TaxiCall Manager**

TaxiCall

* call(): performs a taxi call operation. INPUT: cell phone number, meeting point. OUTPUT: the result of the operation (either success or error).

**TaxiReservation Manager**

TaxiReservation

* reserve(): performs a taxi reservation. INPUT: cell phone number, meeting point, meeting time, destination. OUTPUT: the result of the operation (either success or error).

**TaxiAllocation Manager**

TaxiAllocation

* allocate(): forwards a request to the corresponding queue. INPUT: request. OUTPUT: none.

**Queue Manager**

QueueModification

* addDriver(): adds the given driver to the specified queue. INPUT: cell phone number, queue. OUTPUT: the result of the operation (either success or error).
* removeDriver(): removes the given driver from the specified queue. INPUT: cell phone number, queue. OUTPUT: the result of the operation (either success or error).

RequestForwarding

* forward(): forwards the given request to the drivers in the given queue according to the already mentioned algorithm. INPUT: request, queue. OUTPUT: none.

**GPS Manager**

Localization

localize(): localizes a taxi given its driver. INPUT: cell phone number. OUTPUT: either the taxi location or an error (if the taxi driver is currently not driving).

**Communication Manager**

MessageReception

* setFree(): invoked remotely by drivers. Forwards the request to the corresponding queue (computed through GPS Manager). INPUT: cell phone number. OUTPUT: the result of the operation (either success or error).
* setBusy(): invoked remotely by drivers. Forwards the request to the corresponding queue (computed through GPS Manager). INPUT: cell phone number. OUTPUT: the result of the operation (either success or error).
* login(): invoked remotely by drivers. Forwards the request to the Login Manager. INPUT: cell phone number, password. OUTPUT: the result of the operation (either success or error).

MessageDelivery

* sendRequest(): sends the given requests to the specified drivers and waits for response. INPUT: cell phone number, request. OUTPUT: the driver's answer.
* sendQueuePosition(): sends the current queue position to the given driver. INPUT: cell phone number, queue position. OUTPUT: none.
* checkStatus(): checks whether the given driver is still connected. INPUT: cell phone number. OUTPUT: either true (the driver replied) or false (no response was received).

**Public Web Services**

WebService

* callService(): processes a taxi call request from external systems. INPUT: the developer's cell phone number, the caller's cell phone number, the meeting point. OUTPUT: the result of the operation (either success or error).
* reserveService(): processes a taxi reservation request from external systems. INPUT: the developer's cell phone number, the caller's cell phone number, the meeting point, the meeting time, the destination. OUTPUT: the result of the operation (either success or error).

**2.7 Architectural styles and patterns**

Considering that different types of client need different communication paradigms, the system is designed according to two different architectural styles:

-Client-Server: it is used for the communication with passenger clients (i.e., the web application and the hybrid mobile app). Clients make HTTP requests to the web tier. The latter processes them, invokes components of the business tier and finally produces responses. In this case, the server never opens a connection with the client by itself. Thus, the communication paradigm is a simple request-response.

-Peer-to-peer: it is used for the communication with driver apps. In this case, both the server and the client can open a connection with each other. The server sends asynchronous messages to the clients (e.g. taxi requests) and, vice-versa, the client sends asynchronous messages to the server(e.g. set FREE or set BUSY). Thus, no server role is defined and the overall style is p2p (even though the clients do not communicate with each other).

The following architectural patterns have been considered:

-MVC: the client-server part of the system is designed according to the MVC pattern. JavaServer Faces, indeed, already implements this pattern. The view part is constituted by a set of facelets, whereas the model part is constituted by the java beans. The mediation between this two parts is managed by the faces servlet, which represents the controller of the MVC pattern. A better explanatory picture is provided below.

(PICCIA)

-Publisher-Subscriber: the p2p part design is inspired by the publisher-subscriber pattern. In fact, when a taxi driver sets his state to FREE, he is subscribing to the queue of his current zone. When the system allocates a taxi, it sends the request to the drivers in the corresponding queue. However, the system forwards the request to one taxi at time and in a precise order, according to the already mentioned algorithm. Thus, this is just a variant of publisher-subscriber, in which the messages are not broadcasted and the system must know the identity of each subscriber. A diagram illustrating this pattern is provided below.

(PICCIA)

**2.8 Other design choices**

The system provides APIs to external developers in the form of public web services. This choice has been made for several reasons:

-accessibility: web services are platform independent and they can be accessed by using standard protocols (HTTP in this case);

-extensibility: it is easy to add new web services that call the business tier functions;

-compatibility: the Glassfish application server directly supports web services.

-security: external developers must be registered user (as normal passengers) and the responsibility of every request is on them. Thus, security is guaranteed.

**3. Algorithm design**

We provide in this section a description of the most relevant algorithms used by our application.

**3.1 Queue management taxi allocation**

We provide here a detailed description of the already mentioned algorithm for the allocation of a taxi. Let us first define the following classes:

* Request: represents a request to a driver;
* Queue: represents a queue of Taxi. It provides the functions: enqueue(Taxi), which adds a Taxi object to the last position in the queue, dequeue(), which removes the first Taxi in the queue and returns it;
* Taxi: an object representing a single taxi.
* TaxiDriver: represents a taxi driver.

Let us define the following symbols:

* r: a Request object, representing the request to be forwarded;
* Q: a Queue object, representing the queue in which the request is going to be forwarded;
* response: a variable which contains the driver response at each cycle;
* nextTaxi: a variable containing the next taxi to which the request is going to be forwarded;

**3.2 Waiting time estimation**

We provide now a detailed description of the algorithm the application uses for estimating the waiting time for a taxi. The city is modeled as a graph, in which each node represents a cross and each arc represents a road connecting two crosses (of course, the graph is directed). To each node is associated a couple of coordinates. To each arc is associated a cost, computed as follows:

* c(i->j)= distance(i,j)/AverageSpeed+trafficFactor(i,j)

In which:

* AverageSpeed is the average speed of a taxi (estimated to be 40km/h);
* distance(i,j) is the length of the road connecting crosses i and j (obtained through Google maps services);
* trafficFactor(i,j) is a delay depending on the traffic condition in the zone containing i and j (obtained through an external service of traffic estimation for Milan);

The TaxiAllocation Manager maintains this graph and periodically updates it.

Let us define the following symbols:

* d: a TaxiDriver object representing the driver who accepted the request;
* r: a Request object representing the request for which the waiting time is going to be computed;
* G: the city graph;
* getUpdatedSubGraph(location): returns a subgraph of the last updated graph, including only the nodes in the same zone as the given location and the ones in adjacent zones;
* nearestNode(location): returns the nearest node in G to the specified location;
* s: the node in which the driver is located;
* t: the node in which the meeting point is located;
* S: a variable storing the nodes computed during Dijkstra's algorithm;
* d: a vector, of size the number of nodes, containing the cost of the minimum path from s to each node;

The algorithm proposed uses Dijkstra's algorithm to find the minimum path on the graph from the driver location to the meeting point and returns its cost (which is an estimation of the waiting time).

**4. User interfaces**

For a complete description of user interfaces (with mockups), see the RASD document. In this section we provide a graphical overview of the connections between screens.

(PICCIA)

(PICCIA)

**5. Requirements traceability**

The requirements defined in the RASD document are mapped to the components of the proposed architecture as follows:

* R1  
  -Component: TaxiAllocation Manager and TaxiCall Manager;  
  -Comment:
* R2  
  -Component: TaxiAllocation Manager;  
  -Comment:
* R3  
  -Component: Queue Manager and Communication Manager  
  -Comment:
* R4  
  -Component: TaxiAllocation Manager and GPS Manager  
  -Comment:
* R5,R6  
  -Component: TaxiAllocation Manager  
  -Comment: the notification is saved in the database and eventually retrieved after the next user HTTP request;
* R7  
  -Component: Communication Manager and Queue Manager  
  -Comment:
* R8  
  -Component: Queue Manager  
  -Comment:
* R9  
  -Component: Queue Manager  
  -Comment:
* R10  
  -Component: Communication Manager, Queue Manager and Logout Manager  
  -Comment: the state of a driver is turned to busy either manually or automatically (following a disconnection).
* R11  
  -Component: Queue Manager  
  -Comment:
* R12  
  -Component: Logout Manager  
  -Comment:
* R13  
  -Component: Reservation Manager  
  -Comment:
* R14  
  -Component: Public Web Services  
  -Comment:
* R15  
  -Component: Communication Manager  
  -Comment:
* R16  
  -Component: Logout Manager  
  -Comment:
* R17  
  -Component: Login Manager  
  -Comment:
* R18  
  -Component: Logout Manager  
  -Comment:
* R19  
  -Component: Login Manager  
  -Comment: