Politecnico di Milano

A.Y. 2015/2016

My Taxi Service

Design Document

Bernardis Cesare matr. 852509 Dagrada Mattia matr.852975

December 4, 2015

Contents

1 Introduction **2**
1.1 Purpose 2
1.2 Scope 2
1.3 Definitions, Acronyms, Abbreviations 2
1.4 Reference Documents 2
1.5 Document structure 2

2 Architectural Design **3**
2.1 Overview 3
2.2 High Level Components and their interaction 3
2.3 Component view 4
2.4 Deployment view 5
2.5 Runtime view 6
2.6 Component interfaces 7
2.7 Architectural styles and patterns 7
2.8 Other design decisions 7

3 Algorithm Design **8**
3.1 Request management 8
3.1.1 Pseudocode 8
3.2 Queue balancing 9
3.2.1 Pseudocode 9
3.2.2 Clarifications 9

4 User Interface Design **11**

5 Requirements Traceability **12**

6 References **13**

1 Introduction

1.1 Purpose

The main goal of this document is to describe the system in terms of architectural design choices, going in details about the architectural styles and patterns, and also to show how these are implemented via pseudo-code utilizing meaningful examples.

1.2 Scope

MyTaxiService is a system designed in order to improve the user experience of a taxi service and also improving the quality of life of taxi drivers. It was decided to design both a web application and a mobile application for the customers. These applications allow customers to easily request taxis and also to reserve a taxi ride, even if this feature is only for registered users. Taxi drivers on the other hand have access to the system only via mobile application. The system sends them requests that they can either accept or refuse and they are also able to set themselves available or unavailable. The system holds taxi queues in different areas of the city and depending on the position of the customers will forward requests to a specific queue instead of another.

1.3 Definitions, Acronyms, Abbreviations

1.4 Reference Documents

1.5 Document structure

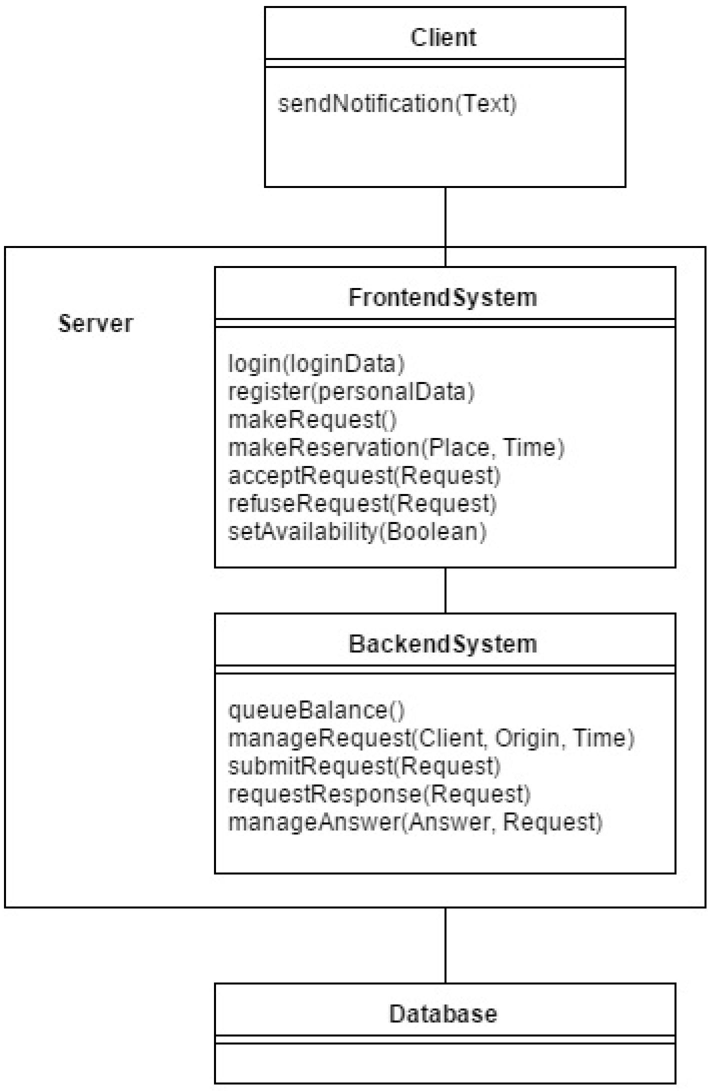
This document is structured as following:

1. **Introduction**: this section represents a generic description of the document.
2. **Architectural Design**: this section gives informations about the architectural choices, showing also which styles and patterns have been selected.
3. **Algorithm Design**: this section focuses on the definition of the most relevant algorithmic part of this project.
4. **User Interface Design**: provides an overview on how the user interface(s) of the system will look like.
5. **Requirements Traceability**: explains how the requirements defined in the RASD map into the design elements.

2 Architectural Design

2.1 Overview

2.2 High Level Components and their interaction



2.3 Component view

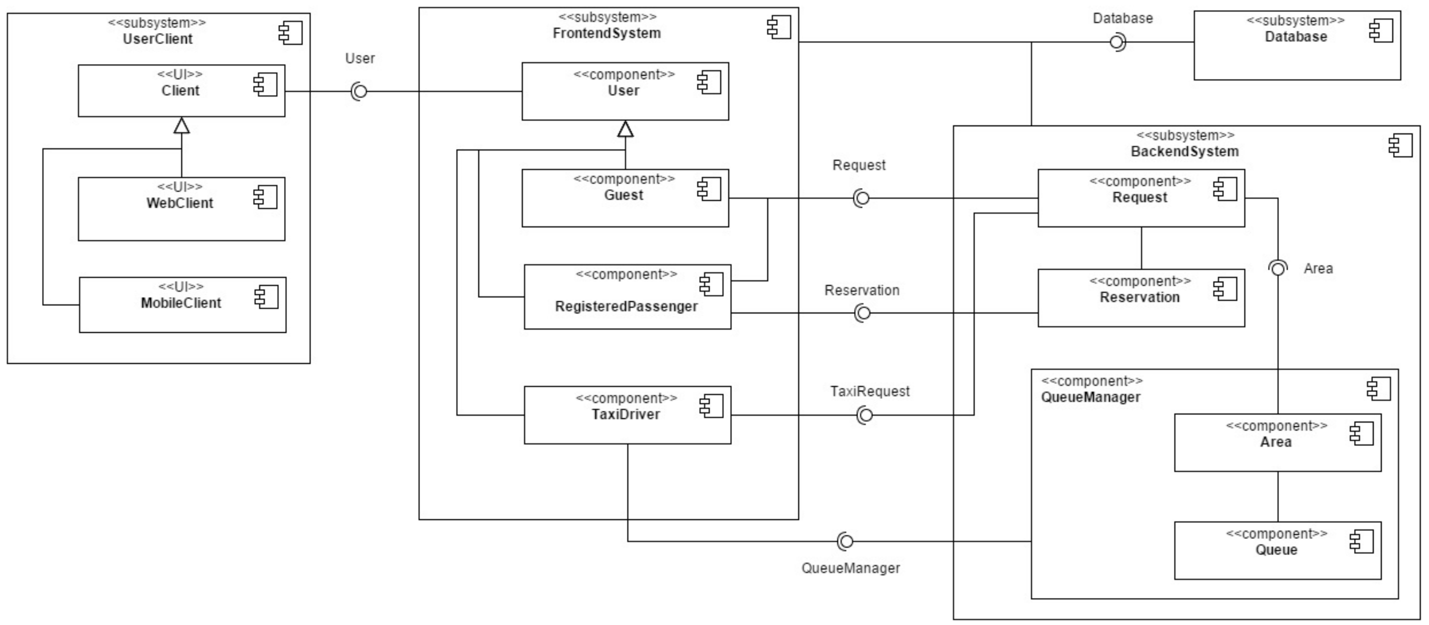


Figure 1: Component Diagram

2.4 Deployment view

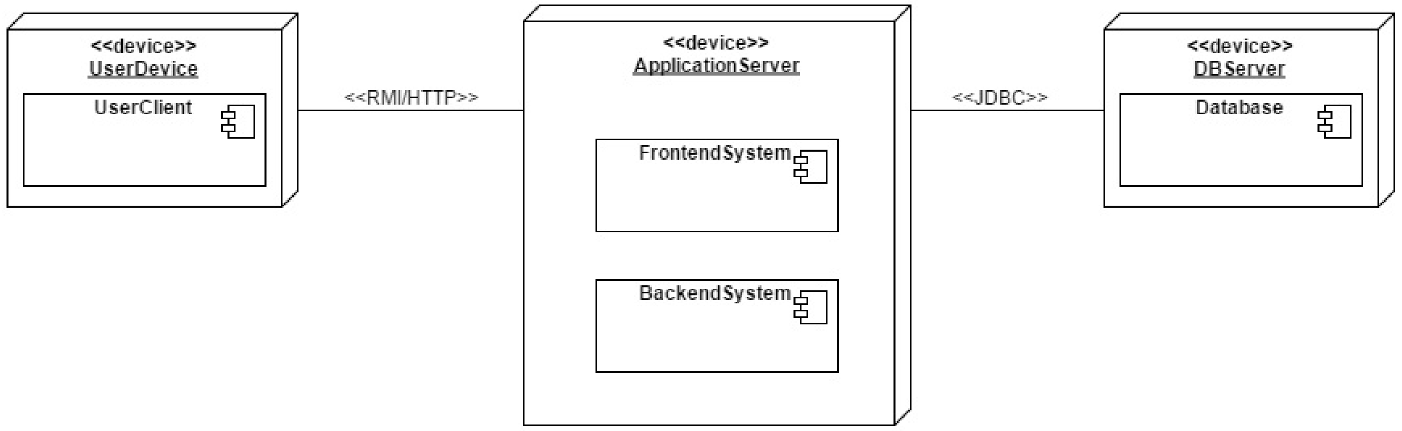


Figure 2: Deployment Diagram

2.5 Runtime view

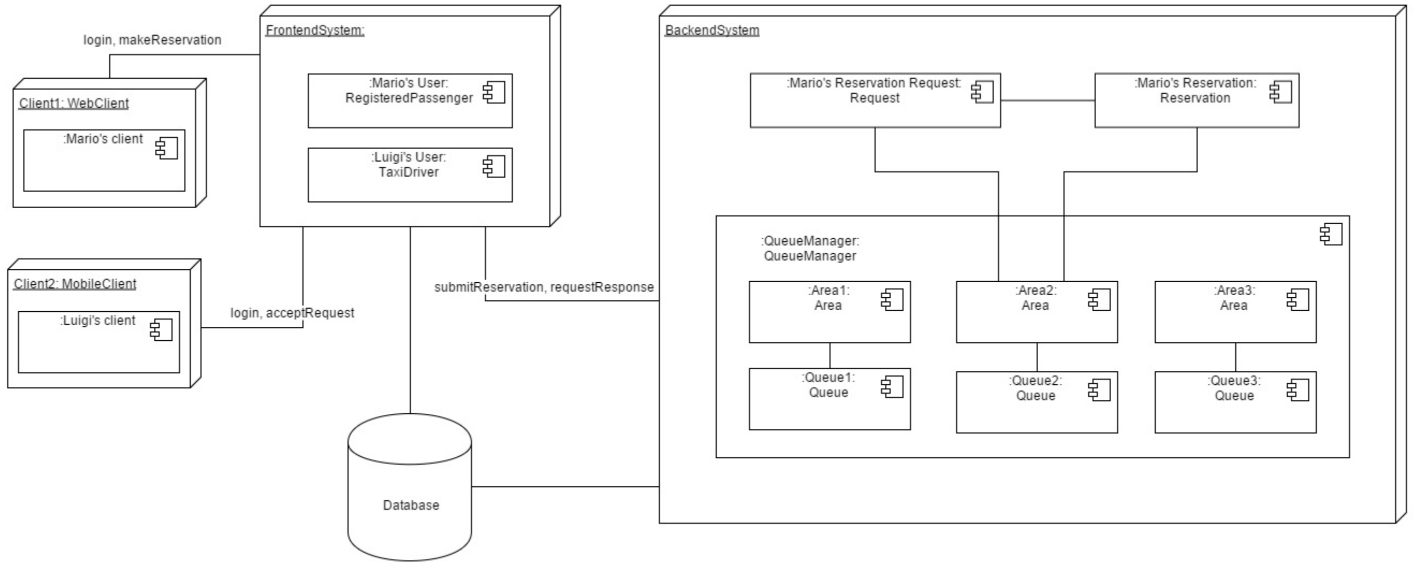


Figure 3: Runtime Diagram

2.6 Component interfaces

2.6.1 TaxiRequest

This interface is used by the taxi to interact with a request. It is necessary because we expect an answer from the taxi driver after he receives the notification of a request. The "manageAnswer" function is the most important functionality provided, which allows to submit and elaborate, at the request level, the answer that the taxi driver gives.

2.6.2 Area

This interface can be widely used to interact with an Area. It is also useful to act indirectly on the queues through the area.

2.6.3 QueueManager

This interface is used by taxis to insert themselves into area queues. It avoids that the TaxiDriver, that belongs to the FrontendSystem, interacts directly with a core component of the BackendSystem.

2.7 Architectural styles and patterns

2.8 Other design decisions

3 Algorithm Design

3.1 Request management

This algorithm

3.1.1 Pseudocode

```
procedure MakeRequest(user)
    if $!validRequest(user)$ then
        $sendMessage(SPAM\_REQUEST\_ERROR, user)$
    else
        $rqst \leftarrow new\ Request(user)$
        $storeRequest(rqst)$
        $area \leftarrow retrieveArea(origin)$
        $taxi \leftarrow area.queue.remove(0)$
        $taxi.submitRequest(rqst)$
    end if
end procedure
```

```
procedure SubmitRequest(request)
    $timer \leftarrow newTimer(60)$
    $timer.start()$
    $sendMessage(rqst, taxi)$
end procedure
```

```
procedure RequestResponse(answer, request)
    $request.manageAnswer(answer)$
    if $answer \equiv REFUSE$ then
        $taxi.declined++$
        if $taxi.declined \equiv 3$ then
            $taxi.setUnavailable()$
        else
            $queueManager.add(taxi)$
        end if
    else
        $taxi.setUnavailable()$
    end if
end procedure
```

```
procedure ManageAnswer(answer)
    if $answer \equiv REFUSE$ then
        $area \leftarrow retrieveArea(rqst.origin)$
        $taxi \leftarrow area.queue.remove(0)$
        $taxi.submitRequest(rqst)$
    else
        $user.sendConfirmation(estimatedTime(), taxi)$
    end if
end procedure
```

3.2 Queue balancing

This algorithm shows the mechanism that every two minutes balances the area queues. The idea is that a taxi may move from its position at any time, for any reason, and, perhaps, pass through other areas than the one he belonged to when he was first inserted into the system. So a simple job running this algorithm can keep the system up to date and avoid inconsistencies (like sending a notification to a taxi out of the interested area).

3.2.1 Pseudocode

3.2.2 Clarifications

The first section of the algorithm updates the area queues, moving taxis from the areas they belonged last time they were inserted to the one they belong now, only if they are different (we assume that *TaxiDriver.Area* retrieves the actual position of the taxi and returns the corresponding area). After the update, the system tries to balance his queues where the values are out of some critical values. It is simple: if the length of an area queue is under a *minCricitalSize*, that area needs to be "filled" with taxis. If the length of an area queue is above a *maxCricitalSize*, that area has too many taxis without a reason an the last taxis in those queues can be moved elsewhere. So the system notifies these *surplusTaxis*, advising them to move where they are really needed.

```
for all $TaxiDrivers$ available as $TaxiDriver$ do
    $actualArea \leftarrow TaxiDriver.Area$
    $oldArea \leftarrow TaxiDriver.Queue.Area$
    if $actualArea \neq oldArea$ then
        $oldArea$.extract($TaxiDriver$)
        $actualArea$.insert($TaxiDriver$)
    end if
end for
$criticalAreas \leftarrow$ List
for all $Areas$ as $Area$ do
    if $Area.Queue.size \leq Area.minCriticalSize$ then
        $criticalAreas$.add($Area$)
    end if
end for
$surplusTaxis \leftarrow$ List
for all $Queues$ as $Queue$ do
    for all $TaxiDrivers$ with index $\geq Queue.Area.maxCriticalSize$ do
        $surplusTaxis$.add($TaxiDriver$)
    end for
end for
$nofitication \leftarrow$ "In the areas ".$criticalAreas$.toString()." we have a taxi shortage. Go there and get back to work!"
for all $surplusTaxis$ as $TaxiDriver$ do
    $TaxiDriver$.sendNotification($notification$)
end for
```

4 User Interface Design

5 Requirements Traceability

6 References