Politecnico di Milano

Software Engineering 2 Project
a.y. 2015-16

MyTaxiService
Design Document
Version 1.0

Casati Fabrizio, 853195
Castelli Valerio, 853992

Referent professor: Di Nitto Elisabetta

December 4, 2015

Contents

Chapter 1

Introduction

1.1 Purpose

This document represents the Software Design Description (SDD), also simply called Design Document, for myTaxiService.

The purpose of this document is to share with all the interested parties a more detailed description of how myTaxiService is designed and architected, with a particular emphasis on what design decisions the development team has made and the rationale behind them.

1.2 Scope

One of the key decisions a software engineer must make while designing a system is between which software components should be actually developed and implemented, and which can be taken for granted as already existing on the market.

This decision has an obvious impact on what a design document should describe and what components are instead left out of the analysis and simply considered ready to be used.

For this reason, throughout this document we will mainly focus on the peculiar characteristics of myTaxiService and provide explanations, diagrams and descriptions for those software components that are specific to this project. On the other hand, we will assume to be able to use a few commercial components for the most common tasks. This is done for a couple of practical reasons: firstly, it allows us to focus on the critical aspects of the project and thus to save budget; and secondly, it lets us leverage a set of well-tested, robust and scalable components that are specifically design and optimized to accomplish certain tasks, thus obtaining a result which is better than anything we could design from scratch to implement the same functionalities. We will clearly annotate which components will need to be developed as parts of our system and which ones will be obtained from a

third party.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Here we present a list of significant, context-specific terms used in the document.

- ACID properties: set of properties of transactions in a relational database management system. Stands for Atomicity, Consistency, Isolation and Durability.

1.3.2 Acronyms

- SDD: Software Design Description.
- DD: Design Document. Used as a synonym of SDD.
- DMZ: Demilitarized Zone.
- DBMS: Database Management System.
- API: Application Programming Interface.
- RASD: Requirement Analysis and Specification Document.
- KPI: Key Performance Indicator.

1.3.3 Abbreviations

- Req. as for Requirement

1.4 Reference Documents

- Assignment document: Assignments 1 and 2 (RASD and DD).pdf
- Template for the Design Document (Structure of the design document.pdf)
- IEEE Systems and software engineering — Architecture description (ISO/IEC/IEEE 42010, first edition)
- IEEE Standard for Information Technology —Systems Design — Software Design Descriptions (IEEE Std 1016^{TM}-2009)
- Microsoft Application Architecture Guide, 2nd edition (published in 2009, ISBN: 9780735627109)

- The Humane Interface (Jef Raskin, 2000, Addison Wesley - ISBN: 0201379376)

1.5 Document Structure

In order to let stakeholders easily navigate through the document, we will now briefly discuss its structure and give a short description of each section.

In the Architectural Design section, we will go in depth describing how the system is designed from different point of views. In particular, we will provide:

- A general overview of how the system is architected from a high level point of view
- A description of the main components that make up the system, their inner structure and how they interact with each other
- A view of how the components of the system are actually deployed on the physical infrastructure
- A detailed view of the normal runtime conditions in which the system operates, with sequence diagrams of the main tasks
- A list of the significant architectural styles and patterns that have been chosen to design the system

In the Algorithm Design section, we will focus on defining the most relevant and critical algorithms that drive the system operations. In particular, for each of them we will outline the key steps using a short pseudo-code representation.

In the User Interface Design, we will mainly reference the existing UI sketches that were already defined in the RASD and further refine them.

Finally, the Requirements Traceability section will explain how our software architecture fulfills the requirements that were identified in the requirement analysis phase and how those requirements have influenced our design decisions.

Chapter 2

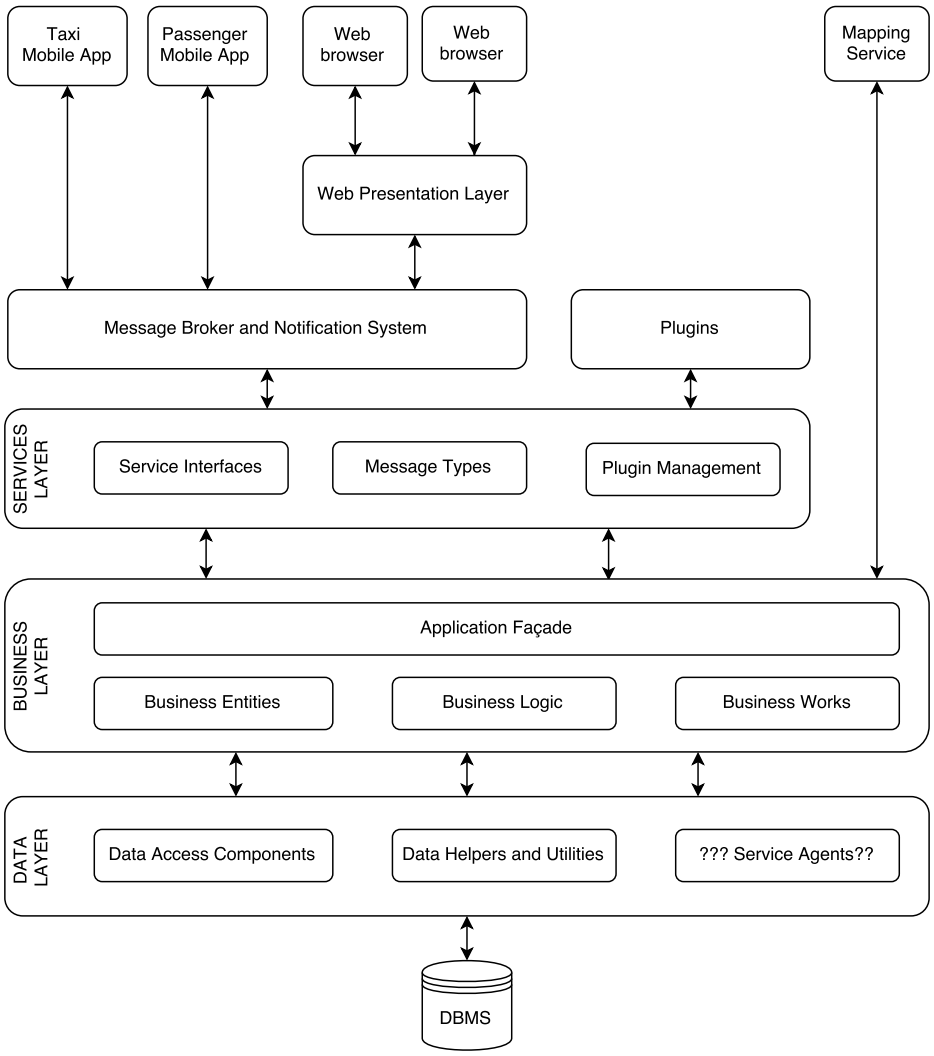
Architectural Design

2.1 Overview

In the following paragraphs we'll present a general overview of how the system is architected, with specific focus on the distinction between logically separated layers.

Specifically, myTaxiService has been envisioned from the ground up to be fully scalable and easily deployable on any number of servers. This characteristic is not only desirable, but actually fundamental for the system to fully accomplish the tasks it's been designed for. In order to guarantee this level of flexibility and modularity, we have settled for a system architecture that inherently enables a wide range of hardware configurations for all kinds of workloads.

The system architecture is thus logically organized in a set of interplaying software layers, whose detailed description is hereby presented.



The lowest layer of the architecture is composed by the relational **Database Management System (DBMS)** that supports all data storage operations. This component fully supports ACID distributed transactions and is meant to be run inside a Demilitarized Zone (DMZ) which is separated from the rest of the system for security reasons. (N.B. questo va messo qui?) In order to provide a higher level of abstraction to all components that need access to data and to be as platform agnostic as possible, the DBMS interface is not directly exposed to the classes that implements the business logic of the system. Instead, an intermediate **Data Layer** is responsible of performing queries on the DBMS while exposing a more flexible, customized interface to the upper layers.

The **Business Layer** implements all core functionalities of the system.

In particular, all operations related to handling taxi requests and reservations, taxi availability and zone management are performed by components of this layer. Data is stored and retrieved using the APIs exposed by the Data Layer. Core functionalities are exposed to clients through a unified Application Façade that allows fine-grained tuning of which operations can be invoked from outside the central system.

The Business Layer also depends on an external **Mapping Service** for the implementation of reverse geocoding and waiting time estimation operations. This external service is directly invoked by classes of the Business Layer by using a public API provided by the Mapping Service itself.

However, not every functionality exposed by the Application Façade may actually be made publicly available to every client. In principle, several levels of permissions could be offered to third parties while maintaining control over private APIs which should only be used by "official clients". Read-only reporting functionalities, for example, could be made available to any requesting party, while more critical operations could be offered only to selected developers after having verified certain requirements are satisfied. Furthermore, specific sets of APIs could be made available only to approved plugins and not be offered to remote services. For this reason, the **Services Layer** provides a comprehensive interface to all kinds of third party services and plugins by carefully defining which methods are available for remote and local invocation, what protocols should be followed for invoking them and what kind of messages can be exchanged between a remote component and the central system.

While locally invokable APIs are made available only to plugins, remotely invokable APIs are also offered to components living outside the perimeter of the central system. The **Message Broker and Notification System** is precisely concerned with guaranteeing an efficient and reliable communication channel with these remote entities by supporting message queues, publish/subscribe communications and dynamic, asynchronous event notifications.

The **Web Presentation Layer** is responsible for the implementation of the web application. It generates the dynamic web pages, offers them to the client via a web server, accepts requests and forwards them to the business layer by means of the communication layer and of the services layer.

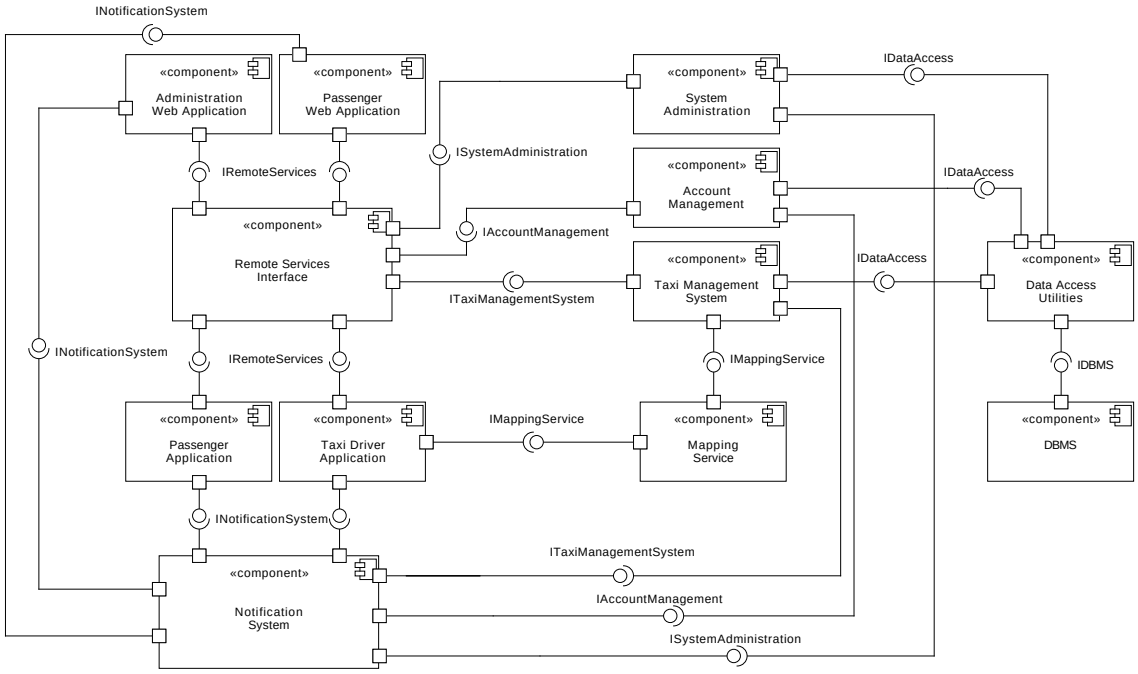
Finally, the **Mobile Applications** let users access the functionalities offered by the system on their smartphones and tablets in a native way. As for the web application, they interact with the central system using the communication layer and the services layer.

2.2 High level components and their interactions

The purpose of this section is to present the main components in which myTaxiService is divided and the relations between them.

In particular, it should be noted that the components discussed here are very abstract clusters of functionalities which do not directly map to any specific module in the final system. Instead, they are intended as an useful representation to show how the main functionalities of the system are grouped together and interact with each other. Further details on how these components are realized in the actual system will be discussed with greater depth in the Component view subsection.

A schematic representation of the component structure of the system is the following.



The **Account Management** component is responsible for all operations related to user accounts. More specifically:

- It implements the passenger registration process
- It implements the login procedure for all users
- It supports operations on existing accounts, including settings management and password retrieval operations

The **Taxi Management System** is the single most important component in the system. It is responsible for:

- Maintaining the availability status of each taxi updated
- Managing the taxi queue associated with each zone in the city
- Accepting and managing taxi reservations
- Fulfilling taxi requests by selecting the first available taxi in the corresponding taxi zone

The **System Administration** component offers system configuration and monitoring functionalities. It enables the insertion, update and deletion of taxis and taxi drivers and the definition of the boundaries of the zones in which the city is divided. It also lets administrators perform queries to obtain system statistics including uptime, number of served requests per day and other key performance indicators. The user interface for interacting with this component is provided by a web application.

The **Database Management System (DBMS)** is the component responsible for storing and retrieving data in a persistent, reliable way. It should be noted that this component will not be implemented from scratch; instead, a commercial solution will be used.

The **Data Access Utilities** component provides an abstraction layer to all those components that need to store data into the DBMS or retrieve data from it.

The **Mapping Service** component is provided by a third party and is accessed via a publicly available API. It is used to perform reverse geocoding and to compute the ETA of the taxi assigned to a certain request.

The **Remote Services Interfaces** component provides external applications and clients a way to invoke the services offered by the central system. Specifically, it implements a platform-independent, SOAP-based web service which exposes the full set of remotely invokable methods defined by the public API specification of myTaxiService.

The **Notification System** component implements an event-based mechanism to notify remote applications and clients of specific changes in the status of the central system. In particular, this is employed to let users known when the system has assigned a taxi to their reservation.

As for how the system can be accessed by its users, the **Taxi Driver Application** and the **Passenger Application** provide a way for taxi drivers and passenger respectively to interact with the system through their smartphones, while the **Passenger Web Application** allows passengers perform their operations through a web browser. Finally, administrators can access the system using the aforementioned **Administration Web Application**.

2.3 Component view

In this section, we'll provide a more detailed description of the most significant components that must be developed as part of myTaxiService.

It should be noted that, when multiple operations or tasks are managed by a certain sub-component, not all of them will be explicitly mentioned in this section. This is because, at this level of detail, we want to put the focus on the primary goal of every sub-component; an exhaustive description of the functionalities implemented by each sub-component will be presented in the Requirements Traceability section of this document.

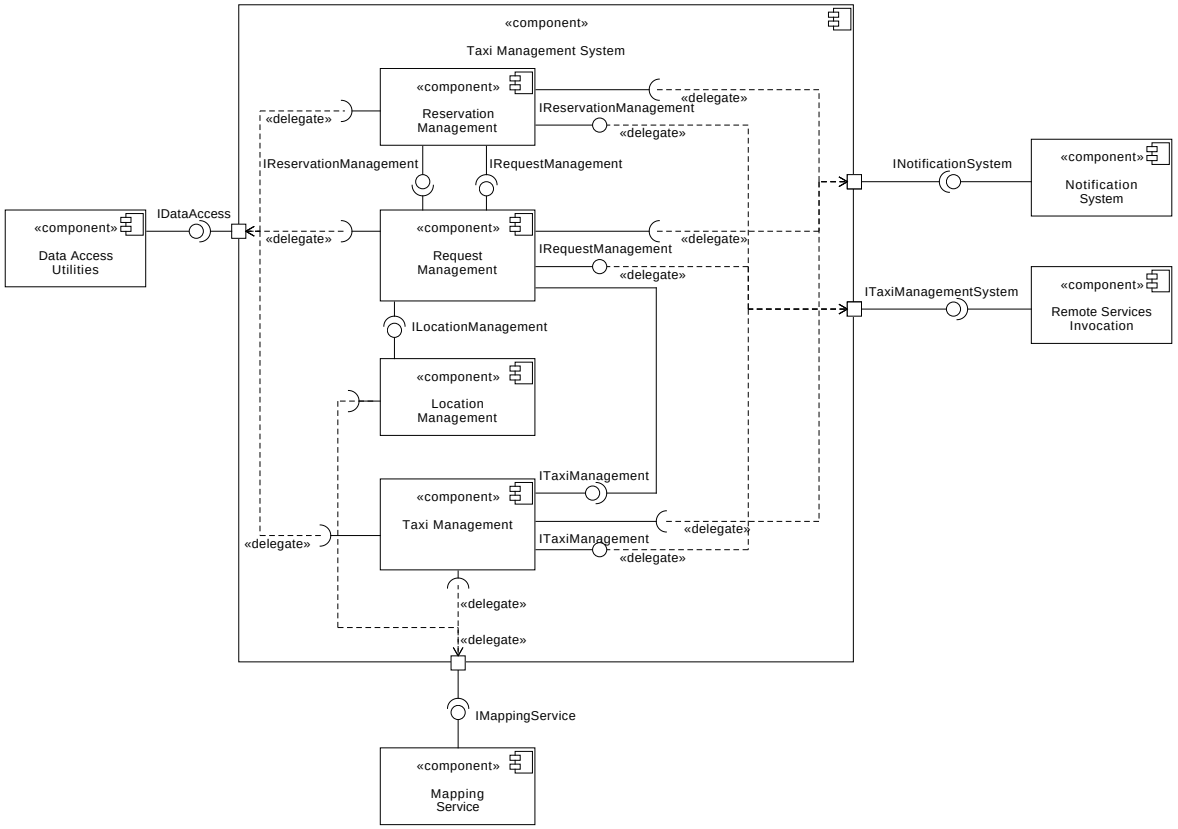
The first component that we'll examine is the **Taxi Management System**. As we already mentioned, this component is primarily responsible for all operations related to taxi management. Although it may appear to be an atomic component, it is in fact composed of three different sub-components, each of them dedicated to handling a specific subset of operations:

- The **Reservation Management** sub-component performs all the activities related to reservation handling. In particular, it is able to receive a request for a new reservation, store it into an internal queue and periodically look for reservations that have come to expiration. Once such a reservation is found, it creates a request for it and transfers the computation to the Request Manager.

- The **Request Management** sub-component performs all the activities related to request handling. In particular, it is able to receive a new taxi request (either from an end user or from the reservation manager), retrieve the associated geographical coordinates through invocation of the Mapping Service API if the request only contains the meeting address, and use them to discover the zone from which the request is coming by invocation of an appropriate method of the Location Management sub-component. It then forwards the necessary information to the Taxi Management sub-component.

- The **Location Management** sub-component is essentially responsible for checking if the geographical coordinates of a certain place are inside the city and, if they are, it computes the zone they belong to.

- The **Taxi Management** sub-component implements the methods to allocate a suitable taxi to a given request, keep the taxis' statuses updated and manage the zones' queues accordingly to the requirements which have been defined in the RASD.

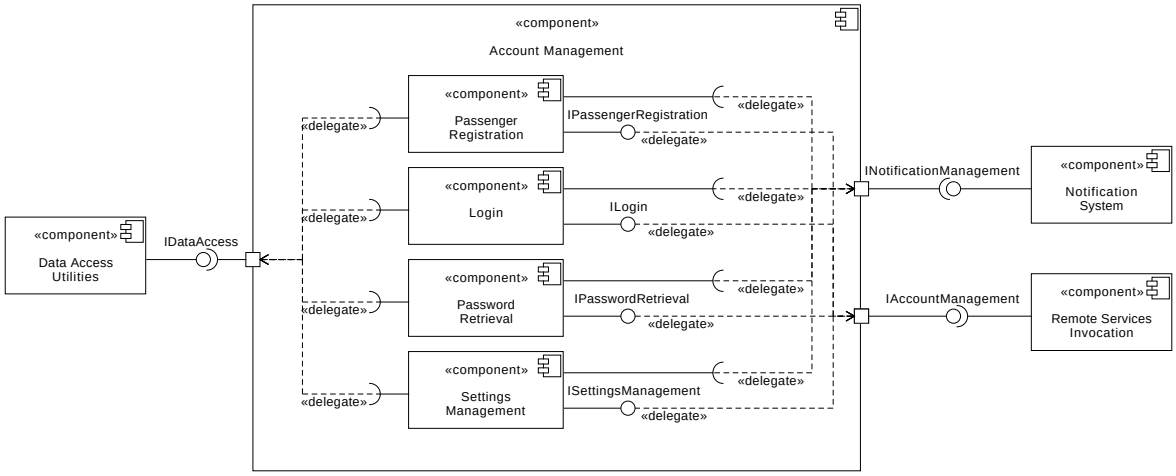


The second component that we'll examine is the **Account Management** component. As we already mentioned, this component is primarily responsible for all operations related to user accounts handling. More specifically, this component is divided into four sub-components, each related to a different kind of operation:

- The **Passenger Registration** sub-component enables passengers to register to the taxi service and create their own account. In particular, it validates the required user data for formal consistency (i.e. checks that the date of birth, email address, mobile phone number and password are in valid format) and creates a temporary user account. It then sends a verification message containing a validation link to the specified email address and, upon user's confirmation, it enables the user account for full usage.

- The **Login** sub-component performs all necessary operations to let registered users log into the system. The user can be either a registered passenger, a taxi driver or a system administrator; depending on the user type, different login data may be required and a different level of privileges will be granted.

- The **Password Retrieval** sub-components implements the password retrieval procedure for all registered users. Depending on the kind of user, a different recovery procedure may be followed.
- The **Settings Management** sub-component contains all the logic that is related to manipulation of an existing account by its owner. Depending on the kind of user, different options may be allowed.



The third component that we'll examine is the **System Administration** component. As we already mentioned, this component is primarily responsible for all operations related to system configuration and monitoring functionalities. More specifically, this component is divided into five sub-components, each related to a different kind of operation:

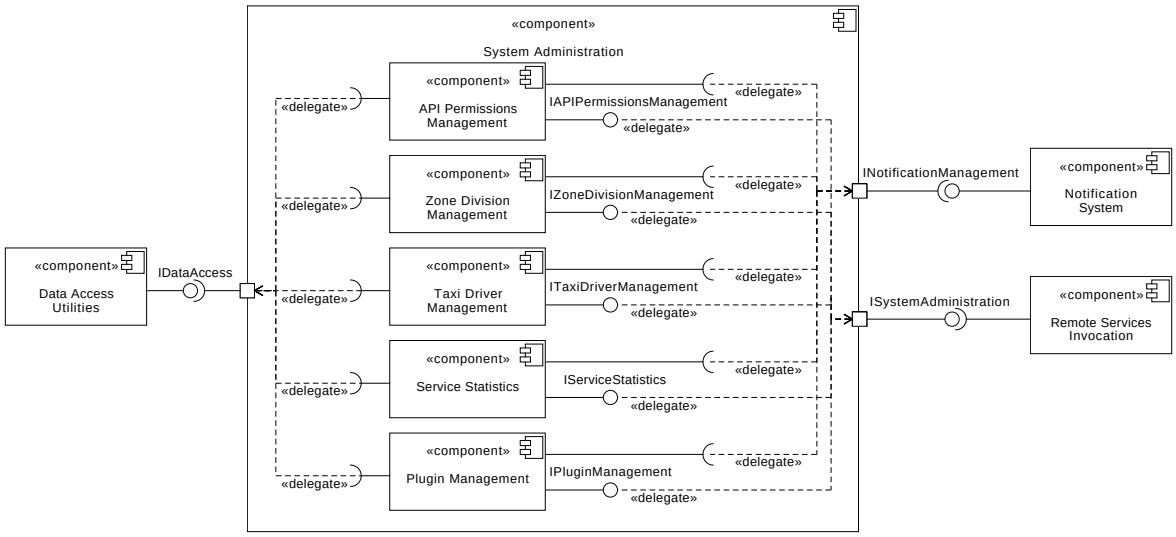
- The **API Permissions Management** sub-component give administrators fine-grained control over which APIs can be publicly accessed and what level of permissions is required to invoke them. More specifically, for each method exposed by the Remote Services Interface component it defines a visibility level that can be either "public" or "restricted". If the method is marked as "restricted", it also allows specifying a specific private key that must be held by the invoking application in order for the invocation to be successful.
- The **Zone Division Management** sub-component implements all the methods for inserting and updating the zone division of the city. In particular, it allows insertion of new taxi zones and deletion or modification of existing ones. It also performs all the necessary checks to ensure that zone consistency is preserved: in particular, this means that overlapping zones won't be accepted. For security reasons, this

component verifies that its methods are only invoked by a user with a sufficient level of privileges.

- The **Taxi Driver Management** sub-component implements all the methods for inserting and updating information about taxi drivers and the related taxis in the system. In particular, it allows insertion of new taxi drivers and taxis and deletion or modification of existing ones. It also performs all the necessary checks to ensure that information consistency is preserved. in particular, this includes validation of taxi driver licenses and taxi plates and enforces the one-to-one correspondence between a taxi driver and its taxi. For security reasons, this component verifies that its methods are only invoked by a user with a sufficient level of privileges.

- The **Service Statistics** sub-components is focused on offering a set of Key Performance Indicators (KPI) about the system operational status to all interested parties with sufficient privileges. This information can include uptime, number of served requests per day, average waiting time and other indicators.

- The **Plugin Management** sub-component let administrators install, enable, disable and remove plugins. It also gives administrators a list of all permissions required by each plugin to work and allows them to selectively grant and revoke them.



2.4 Deployment view

The main purpose of this section is to show how the various components of the system are actually deployed on the hardware infrastructure.

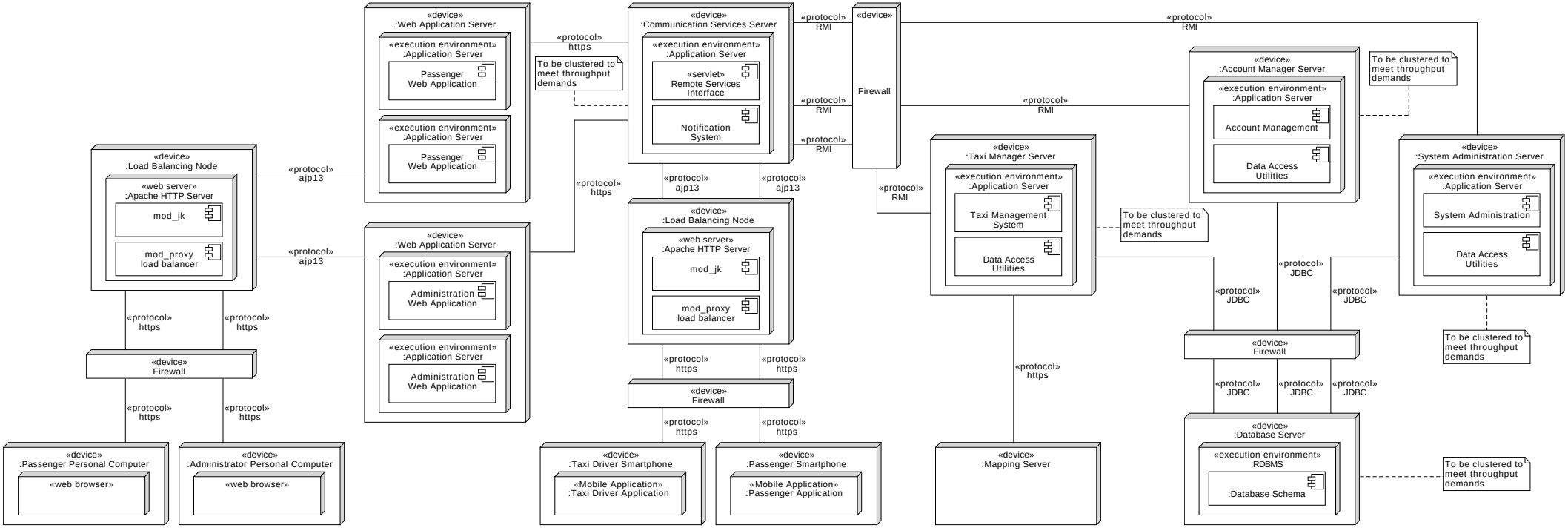
The design of the hardware architecture was mostly influenced by these concerns:

- It had to be sufficiently reliable, available and scalable with respect to the non-functional requirements stated in the RASD
- It had to be compatible with the limited financial resources of a city administration
- It had to be manageable by the city administration staff

For these reasons we have decided to avoid implementing a custom server farm specifically for this project. Instead, we have chosen to deploy myTaxiService on a third-party cloud infrastructure offered as a Platform as a Service.

The following diagram describes which are the most important devices and how software components are deployed on them.

It should be noted that, for readability reasons, we have explicitly depicted only a single instance of each hardware component despite the fact that they are actually arranged in replicated clusters. A load balancer is used to redistribute the workload across the instances.


«device»
:Load Balancing Node
«web server»
:Apache HTTP Server
mod_jk
mod_proxy
load balancer
«protocol»
ajp13
«protocol»
ajp13
«device»
:Web Application Server
«execution environment»
:Application Server
Passenger
Web Application
«execution environment»
:Application Server
Passenger
Web Application
«device»
:Web Application Server
«execution environment»
:Application Server
Administration
Web Application
«execution environment»
:Application Server
Administration
Web Application
«protocol»
https
To be clustered to
meet throughput
demands
«protocol»
https
«device»
:Communication Services Server
«execution environment»
:Application Server
«servlet»
Remote Services
Interface
Notification
System
«protocol»
ajp13
«protocol»
ajp13
«device»
:Load Balancing Node
«web server»
:Apache HTTP Server
mod_jk
mod_proxy
load balancer
«protocol»
RMI
«protocol»
RMI
«protocol»
RMI
«device»
Firewall
«protocol»
RMI
«protocol»
RMI
«device»
:Taxi Manager Server
«execution environment»
:Application Server
Taxi Management
System
Data Access
Utilities
«device»
:Account Manager Server
«execution environment»
:Application Server
Account Management
Data Access
Utilities
To be clustered to
meet throughput
demands
«device»
:System Administration Server
«execution environment»
:Application Server
System Administration
Data Access
Utilities
To be clustered to
meet throughput
demands
To be clustered to
meet throughput
demands
«protocol»
https
«protocol»
https
«protocol»
https
«protocol»
https
«protocol»
https
«protocol»
https
«protocol»
JDBC
«protocol»
JDBC
«protocol»
JDBC
«device»
Firewall
«protocol»
JDBC
«protocol»
JDBC
«protocol»
JDBC
«device»
Firewall
«protocol»
https
«protocol»
https
«protocol»
https
«protocol»
https
«device»
:Passenger Personal Computer
«web browser»
«device»
:Administrator Personal Computer
«web browser»
«device»
:Taxi Driver Smartphone
«Mobile Application»
:Taxi Driver Application
«device»
:Passenger Smartphone
«Mobile Application»
:Passenger Application
«device»
:Mapping Server
«device»
:Database Server
«execution environment»
:RDBMS
:Database Schema
To be clustered to
meet throughput
demands

2.5 Runtime view

The main purpose of this section is to show how the various components of the system interact with each other in some real world scenarios.

The first picture represents a snapshot of the system during its execution. In this scenario Alice is a passenger who has requested a taxi using the passenger mobile application installed on her smartphone, Bob is a taxi driver who has marked himself as available, Charles and Diana are two administrators that are performing queries on the system. The diagram clearly shows the set of processes and threads running on all the devices; in particular:

- The Taxi Management Server is running a single Taxi Management System Process that is responsible of different threads, one for every kind of taxi-related operation. For example, the Request Manager thread is shown controlling a set of taxi requests, including Alice's; the Reservation Manager thread is shown controlling a set of taxi reservations; and the Taxi Manager Thread is shown managing the taxi queues of the different zones and the list of unavailable, out-of-city and currently riding taxis.

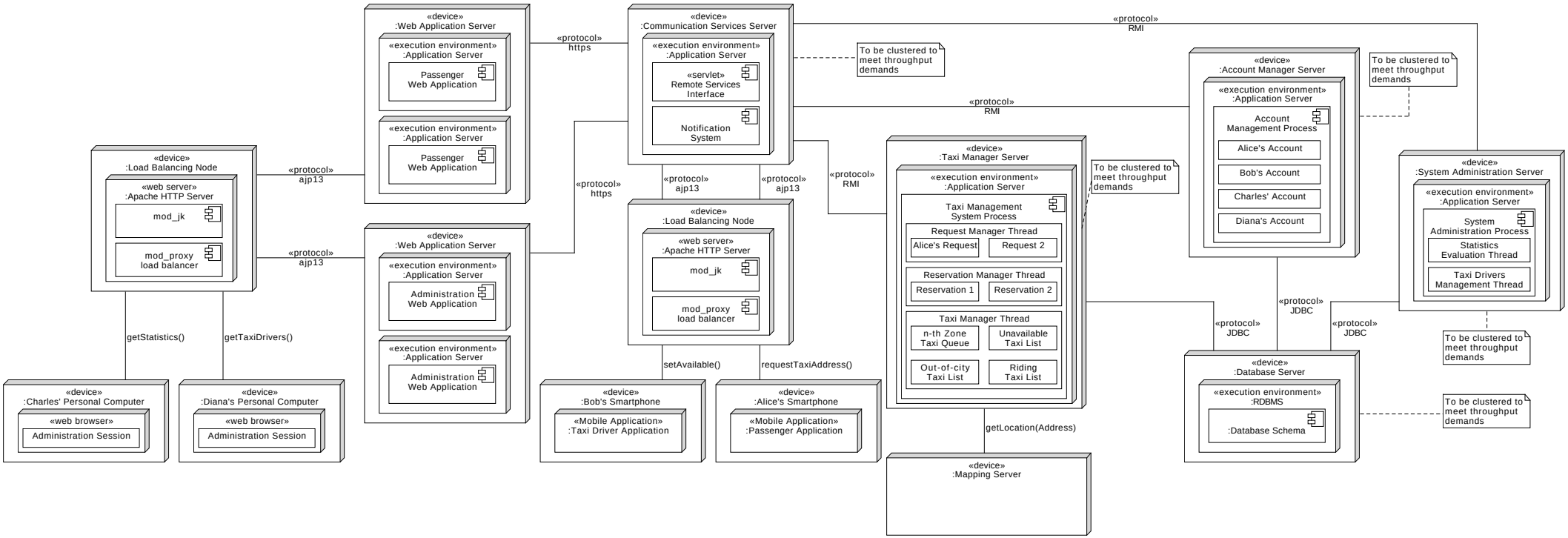
- The Account Manager Server is running a single Account Management process that contains a list of all active user accounts. In particular, in the picture it is shown containing Alice's, Bob's, Charles' and Diana's accounts.

- The System Administration Server is running a single System Administration process that is responsible of different threads, one for every administrative transaction operated on the system. Specifically, the picture shows the Statistics Evaluation thread invoked to satisfy Charles' query and the Taxi Drivers Management thread invoked to satisfy Diana's request.

- The Communication Services Server is running an instance of the Remote Services Interface servlet to expose the key functionalities of the system to remote clients and an instance of the Notification System.

- The Web Application Servers are running instances of the Passenger Web Application presentation logic and of the Administration Web Application presentation logic.

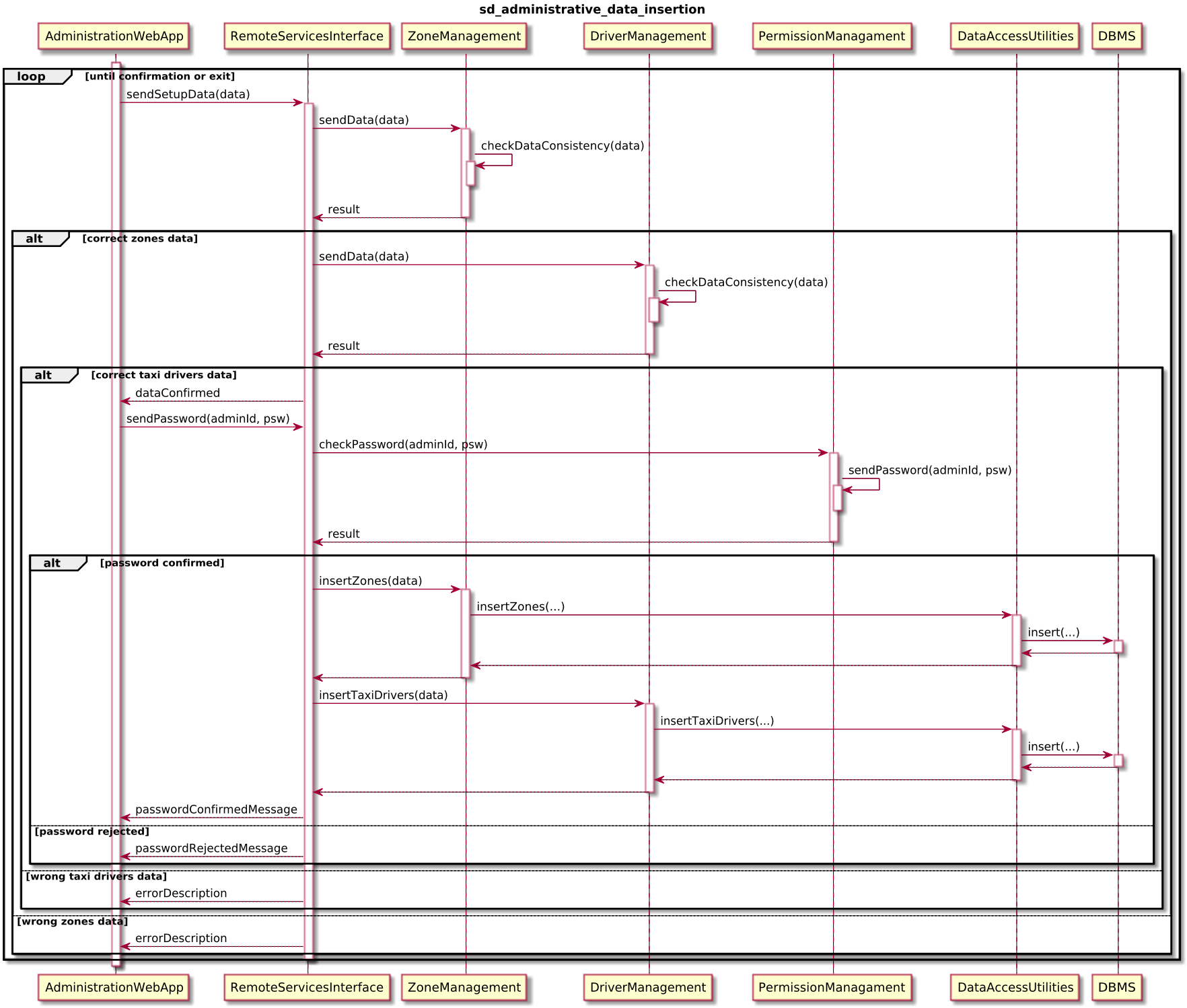
- The Load Balancing nodes are running instances of the Apache Web Server to redirect HTTP requests to different instances of the business logic nodes.

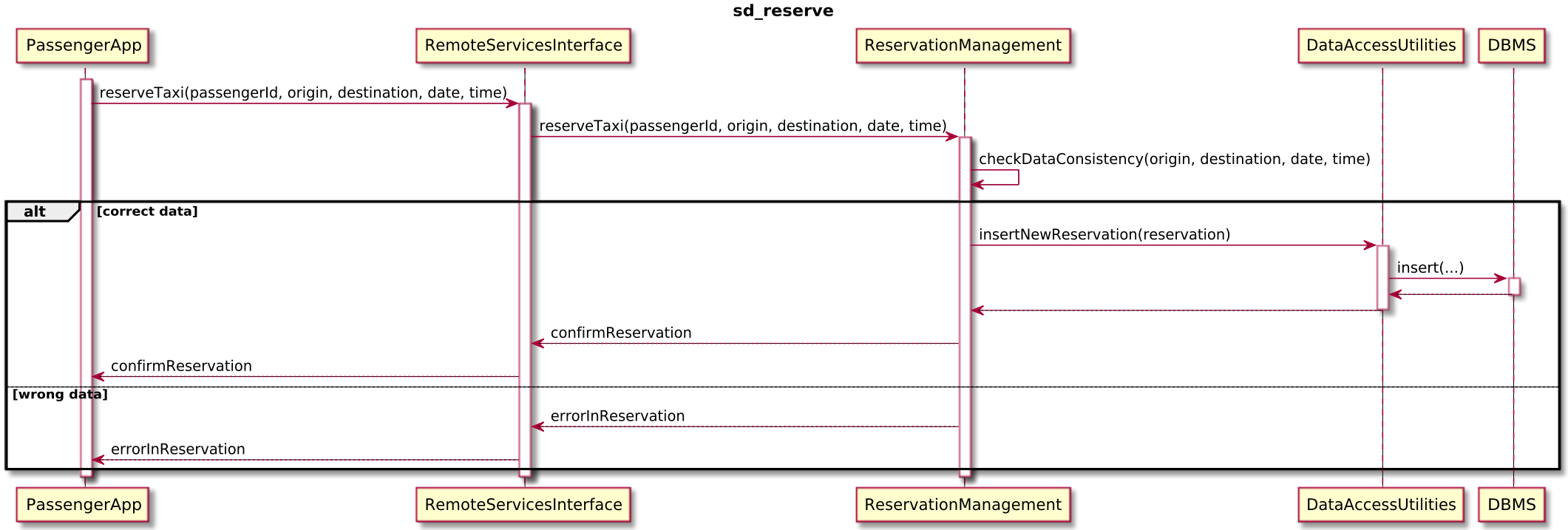

«device»
:Load Balancing Node
«web server»
:Apache HTTP Server
mod_jk
mod_proxy
load balancer
«protocol»
ajp13
«protocol»
ajp13
«device»
:Web Application Server
«execution environment»
:Application Server
Passenger
Web Application
«execution environment»
:Application Server
Passenger
Web Application
«device»
:Web Application Server
«execution environment»
:Application Server
Administration
Web Application
«execution environment»
:Application Server
Administration
Web Application
«protocol»
https
«protocol»
https
«device»
:Communication Services Server
«execution environment»
:Application Server
«servlet»
Remote Services
Interface
Notification
System
To be clustered to
meet throughput
demands
«protocol»
ajp13
«protocol»
ajp13
«device»
:Load Balancing Node
«web server»
:Apache HTTP Server
mod_jk
mod_proxy
load balancer
«protocol»
RMI
«protocol»
RMI
«device»
:Taxi Manager Server
«execution environment»
:Application Server
Taxi Management
System Process
Request Manager Thread
Alice's Request
Request 2
Reservation Manager Thread
Reservation 1
Reservation 2
Taxi Manager Thread
n-th Zone
Taxi Queue
Unavailable
Taxi List
Out-of-city
Taxi List
Riding
Taxi List
To be clustered to
meet throughput
demands
«protocol»
RMI
«device»
:Account Manager Server
«execution environment»
:Application Server
Account
Management Process
Alice's Account
Bob's Account
Charles' Account
Diana's Account
To be clustered to
meet throughput
demands
«device»
:System Administration Server
«execution environment»
:Application Server
System
Administration Process
Statistics
Evaluation Thread
Taxi Drivers
Management Thread
To be clustered to
meet throughput
demands
«protocol»
JDBC
«protocol»
JDBC
«protocol»
JDBC
«device»
:Database Server
«execution environment»
:RDBMS
:Database Schema
To be clustered to
meet throughput
demands
getStatistics()
getTaxiDrivers()
«device»
:Charles' Personal Computer
«web browser»
Administration Session
«device»
:Diana's Personal Computer
«web browser»
Administration Session
setAvailable()
requestTaxiAddress()
«device»
:Bob's Smartphone
«Mobile Application»
:Taxi Driver Application
«device»
:Alice's Smartphone
«Mobile Application»
:Passenger Application
getLocation(Address)
«device»
:Mapping Server

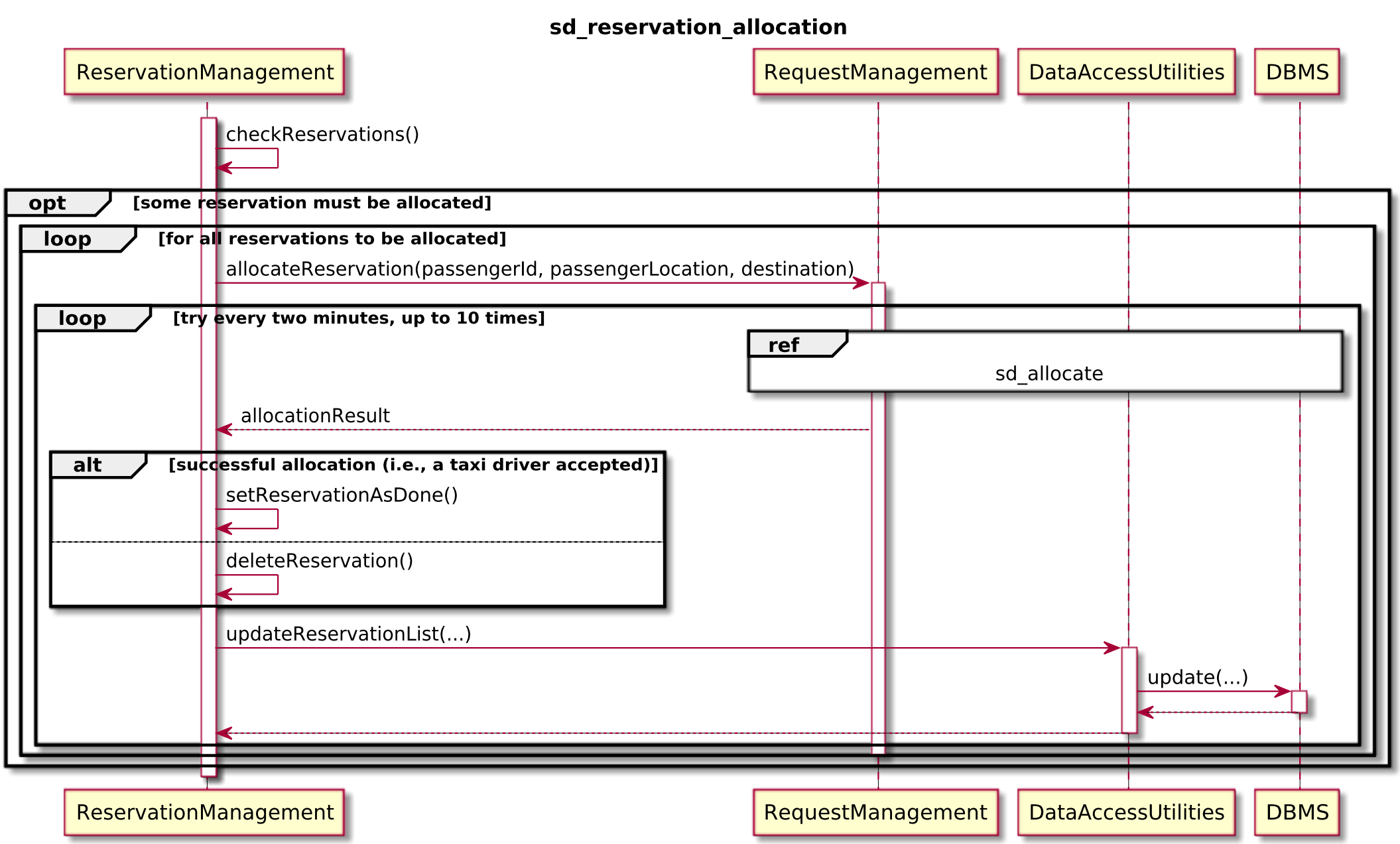
All the other pictures depict the flow of events (in particular method invocations) for the most important operations in the form of sequence diagrams.

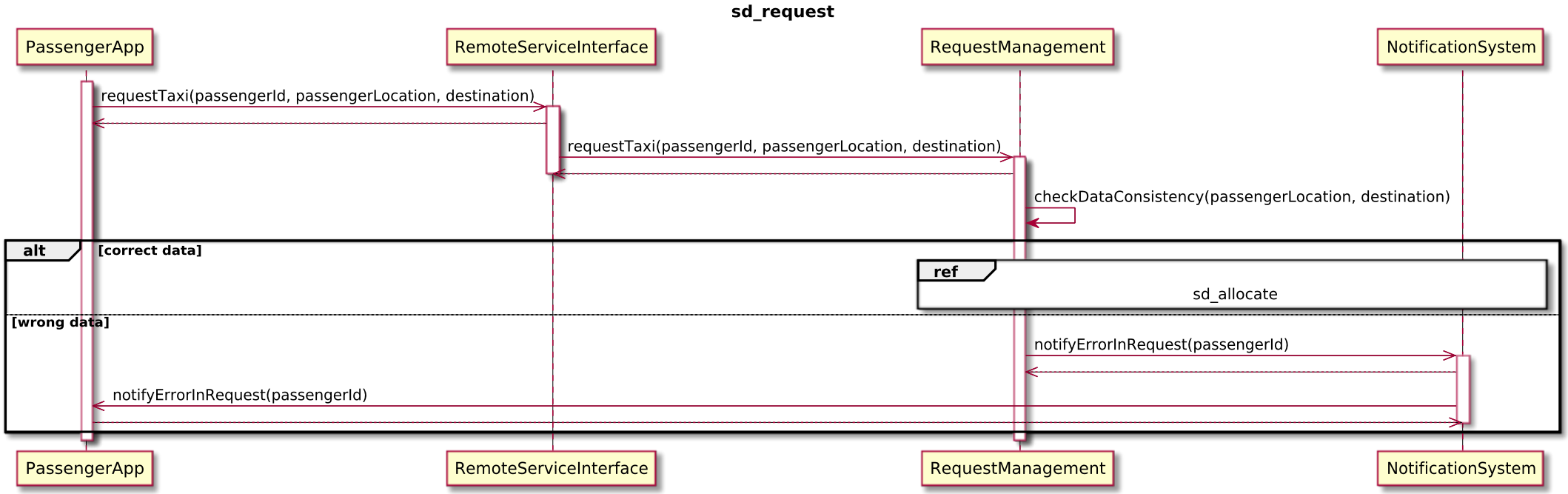
To help clarifying the scope and meaning of each diagram, we'll provide a brief description of each one:

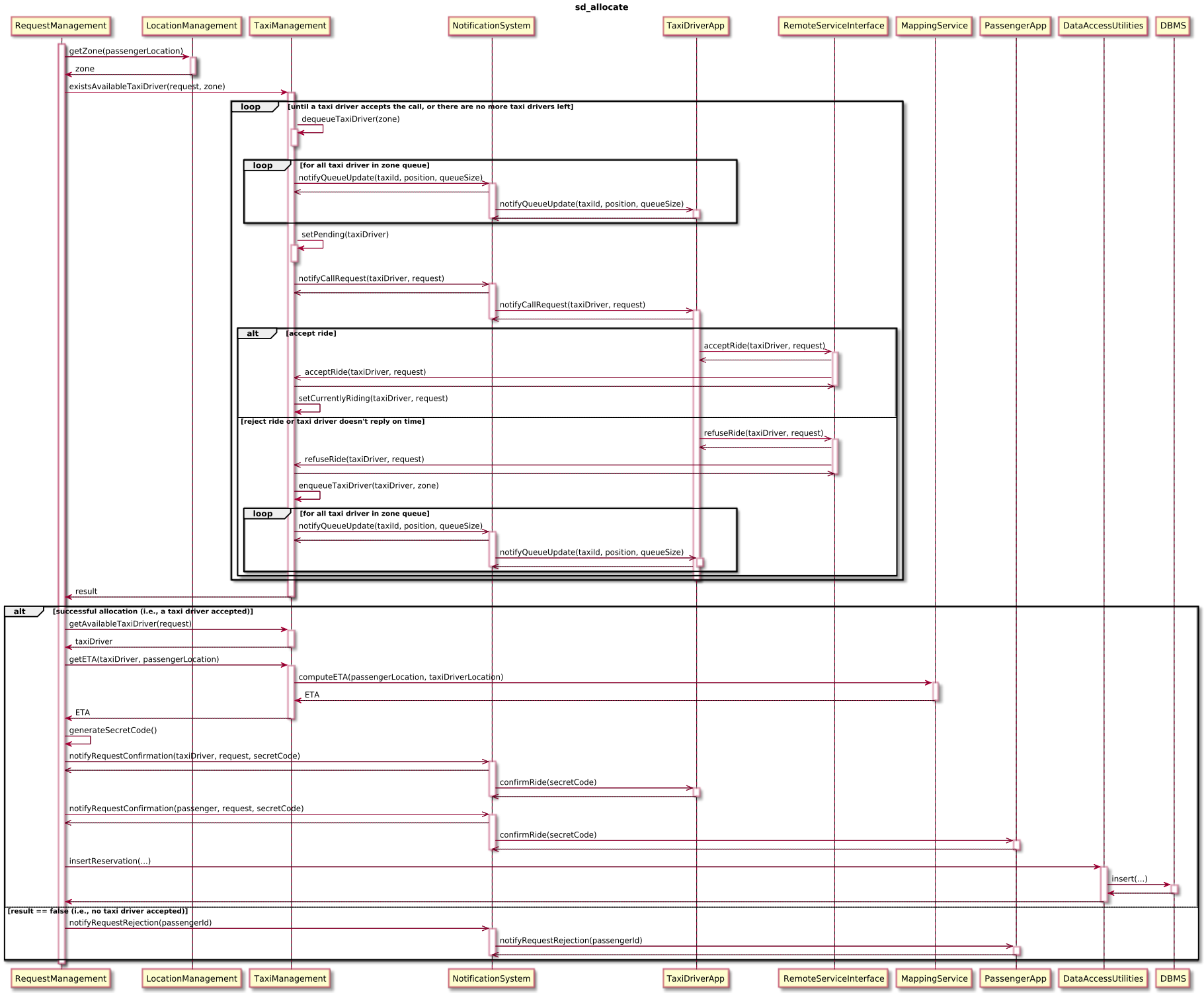
- sd_administrative_data_insertion: illustrates how configuration data is inserted by the administration personnel at the first start of the system
- sd_reserve: describes how the taxi reservation procedure works
- sd_reservation_allocation: shows how the system allocates a previously stored reservation, by converting it into a taxi request
- sd_request: illustrates how the system handles a taxi request coming from a passenger
- sd_allocate: shows how the system matches a request of a passenger with a taxi driver and how it notifies the passenger and the taxi driver about the event
- sd_show_reservations: describes the procedure that allows a passenger to visualize all his past and pending reservations
- sd_zone_update: shows how the system allows the administration personnel to update the information about the city zone division
- sd_drop_call: illustrates how the system handles the call drop procedure
- sd_end_ride: shows the procedure that is triggered when a taxi driver has ended a ride and has notified the system about it
- sd_taxi_location_changed: shows all the operations executed by the system to correctly handle the change of a taxi driver location
- sd_toggle: describes the operations that are triggered when a taxi driver changes his availability status using the toggle button in the mobile application

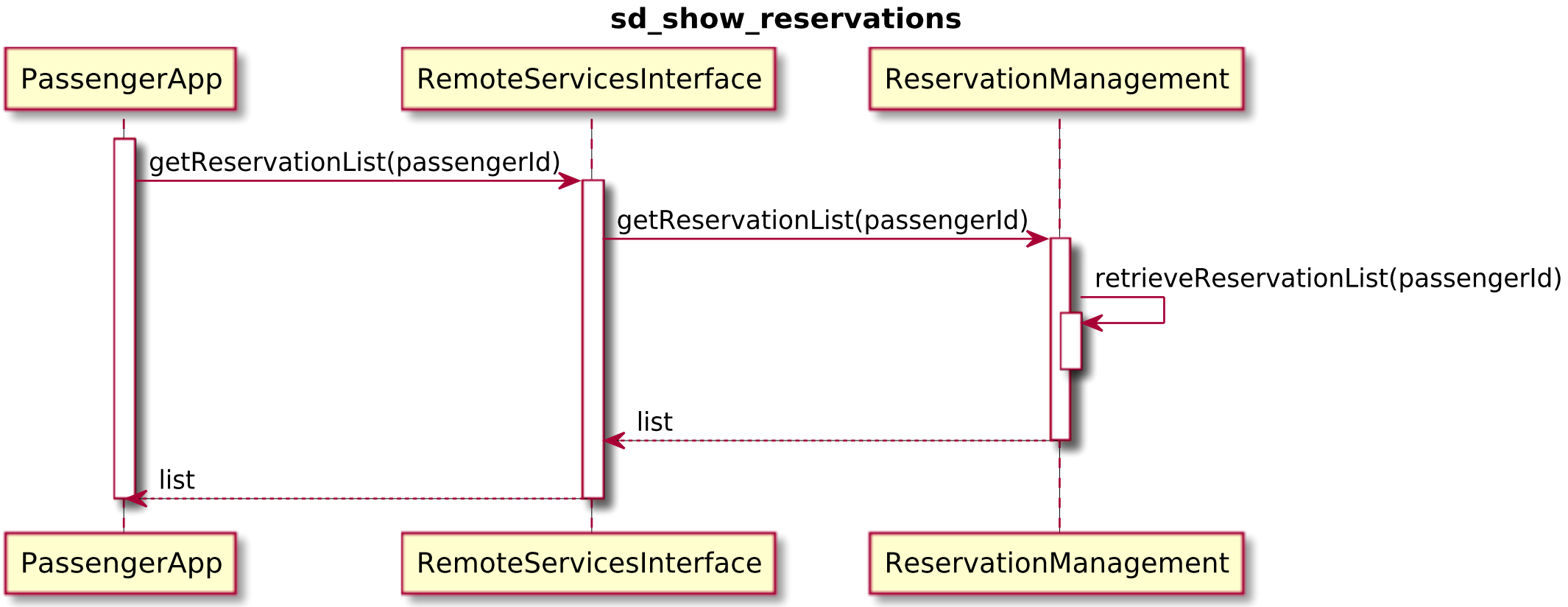

sd_administrative_data_insertion
AdministrationWebApp
RemoteServicesInterface
ZoneManagement
DriverManagement
PermissionManagament
DataAccessUtilities
DBMS
loop
[until confirmation or exit]
sendSetupData(data)
sendData(data)
checkDataConsistency(data)
result
alt
[correct zones data]
sendData(data)
checkDataConsistency(data)
result
alt
[correct taxi drivers data]
dataConfirmed
sendPassword(adminId, psw)
checkPassword(adminId, psw)
sendPassword(adminId, psw)
result
alt
[password confirmed]
insertZones(data)
insertZones(...)
insert(...)
insertTaxiDrivers(data)
insertTaxiDrivers(...)
insert(...)
passwordConfirmedMessage
[password rejected]
passwordRejectedMessage
[wrong taxi drivers data]
errorDescription
[wrong zones data]
errorDescription

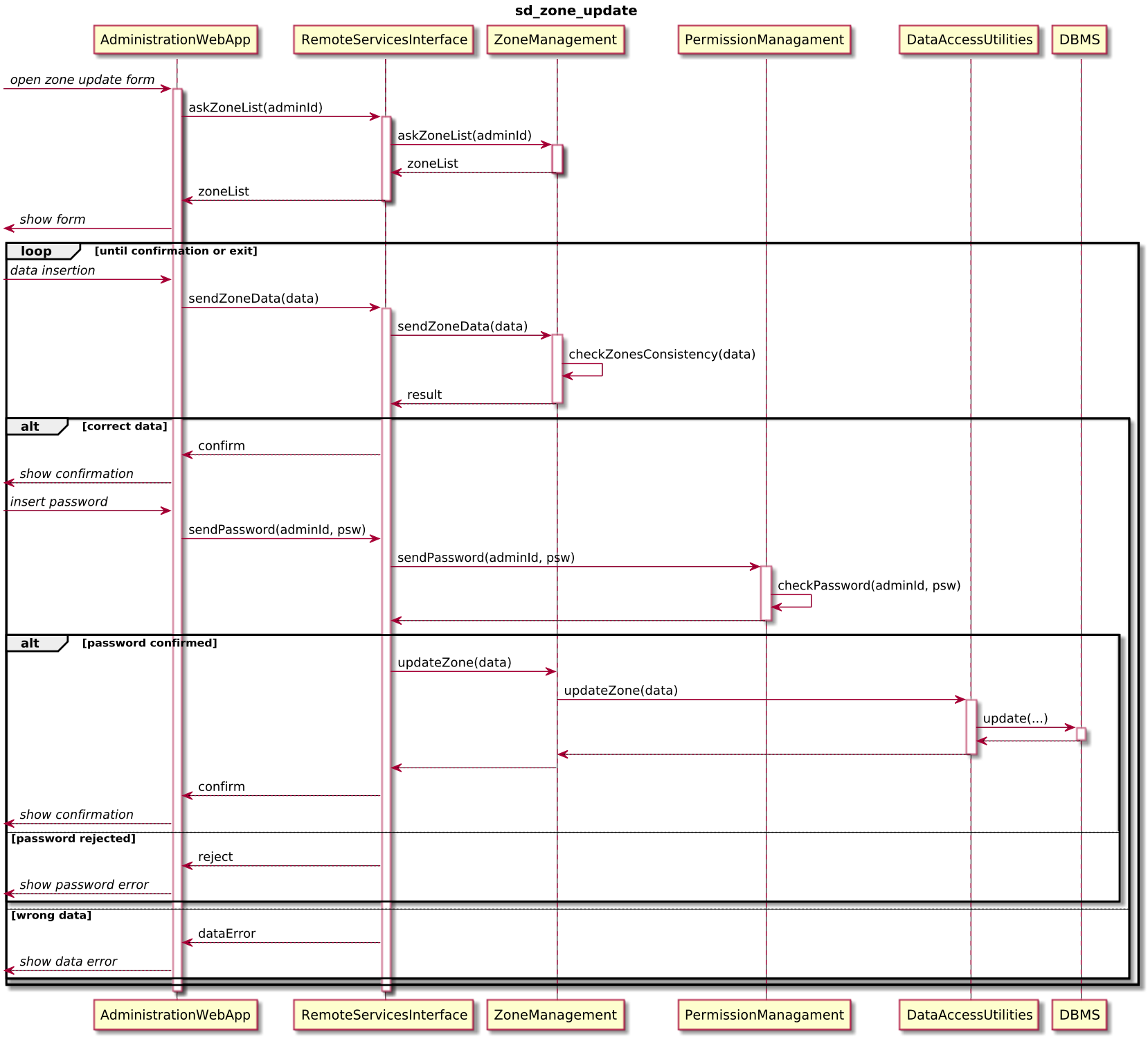

sd_reserve
PassengerApp
RemoteServicesInterface
ReservationManagement
DataAccessUtilities
DBMS
reserveTaxi(passengerId, origin, destination, date, time)
reserveTaxi(passengerId, origin, destination, date, time)
checkDataConsistency(origin, destination, date, time)
alt
[correct data]
insertNewReservation(reservation)
insert(...)
confirmReservation
confirmReservation
[wrong data]
errorInReservation
errorInReservation
PassengerApp
RemoteServicesInterface
ReservationManagement
DataAccessUtilities
DBMS

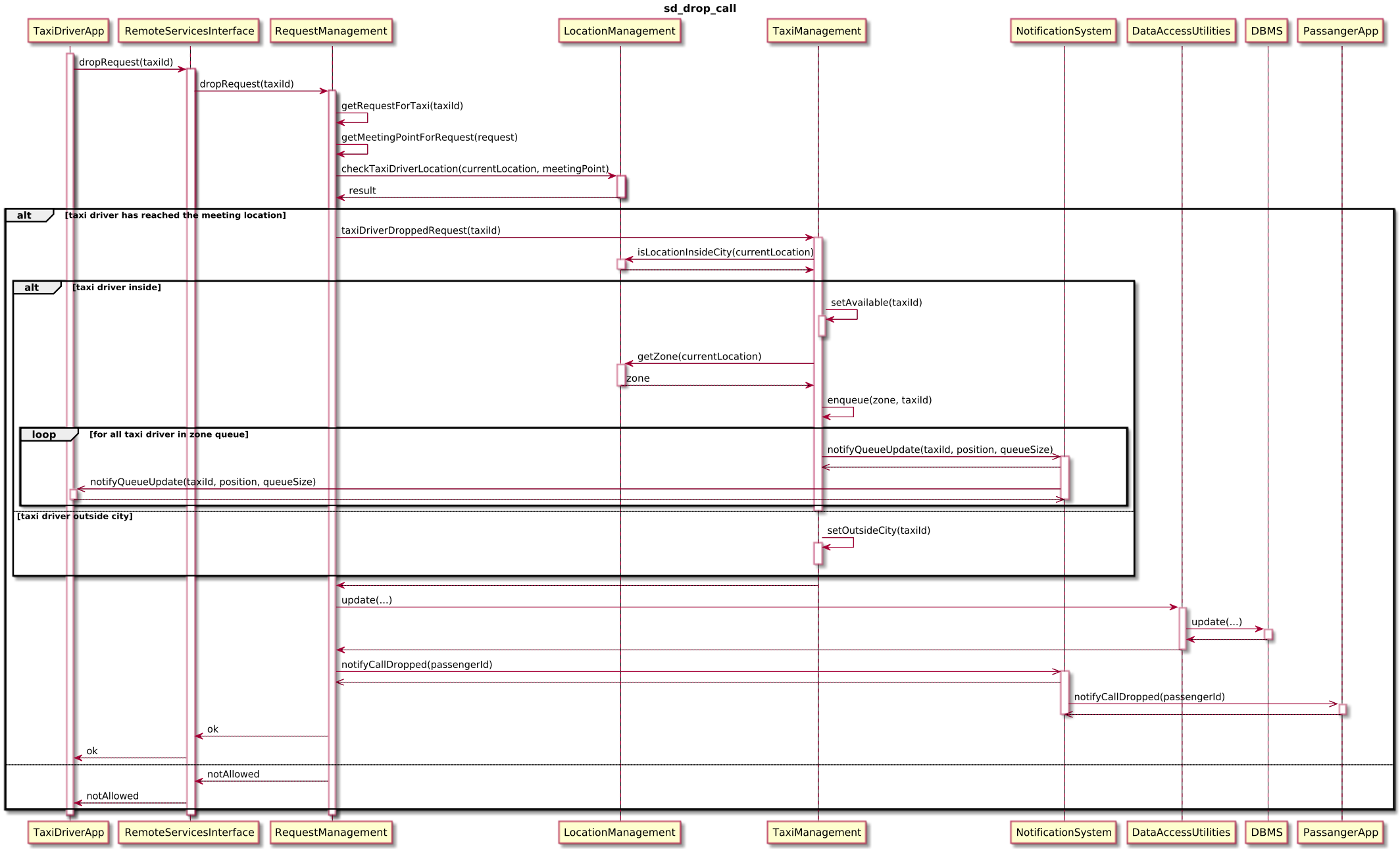

sd_reservation_allocation
ReservationManagement
RequestManagement
DataAccessUtilities
DBMS
checkReservations()
opt
[some reservation must be allocated]
loop
[for all reservations to be allocated]
allocateReservation(passengerId, passengerLocation, destination)
loop
[try every two minutes, up to 10 times]
ref
sd_allocate
allocationResult
alt
[successful allocation (i.e., a taxi driver accepted)]
setReservationAsDone()
deleteReservation()
updateReservationList(...)
update(...)
ReservationManagement
RequestManagement
DataAccessUtilities
DBMS

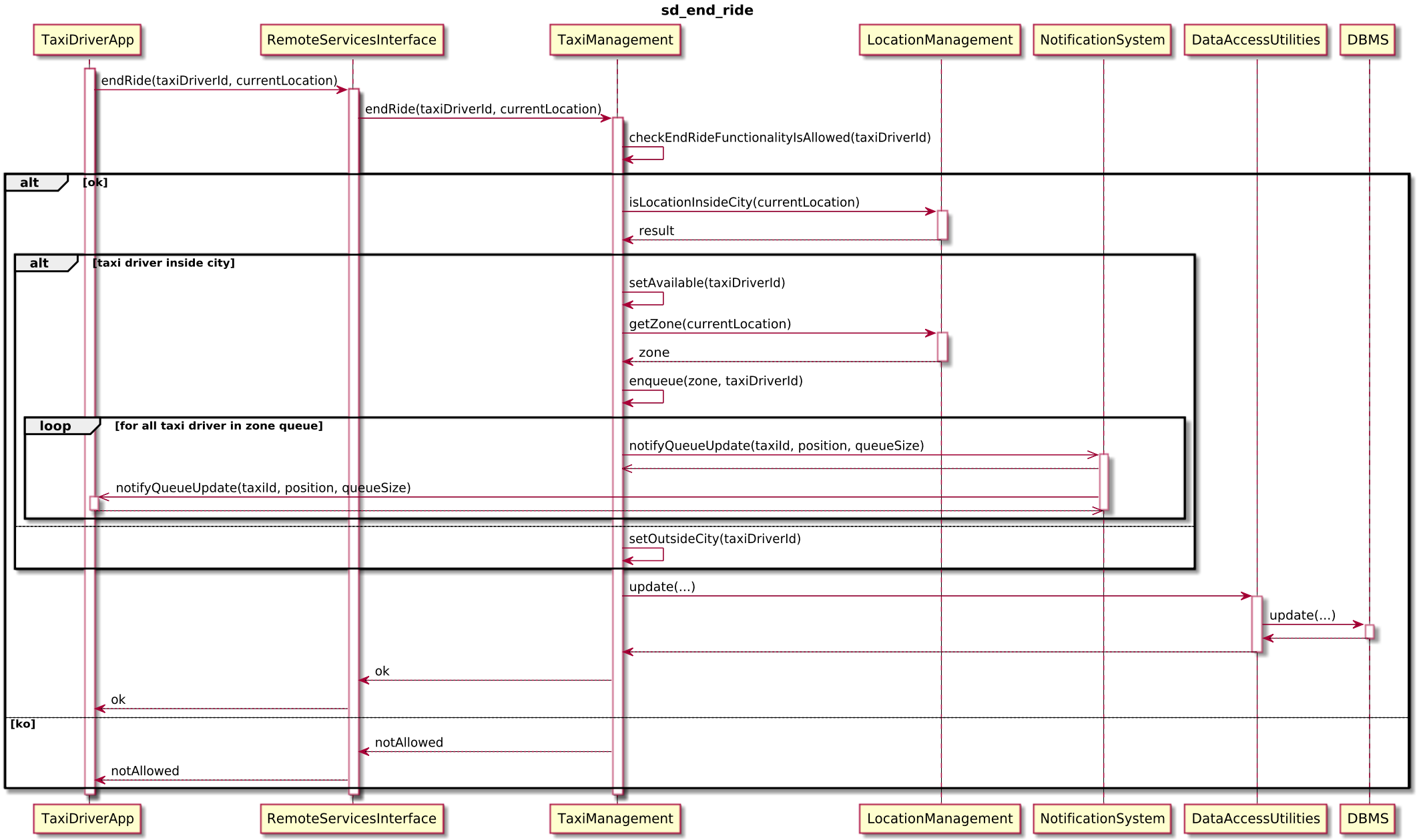

sd_request
PassengerApp
RemoteServiceInterface
RequestManagement
NotificationSystem
requestTaxi(passengerId, passengerLocation, destination)
requestTaxi(passengerId, passengerLocation, destination)
checkDataConsistency(passengerLocation, destination)
alt
[correct data]
ref
sd_allocate
[wrong data]
notifyErrorInRequest(passengerId)
notifyErrorInRequest(passengerId)
PassengerApp
RemoteServiceInterface
RequestManagement
NotificationSystem

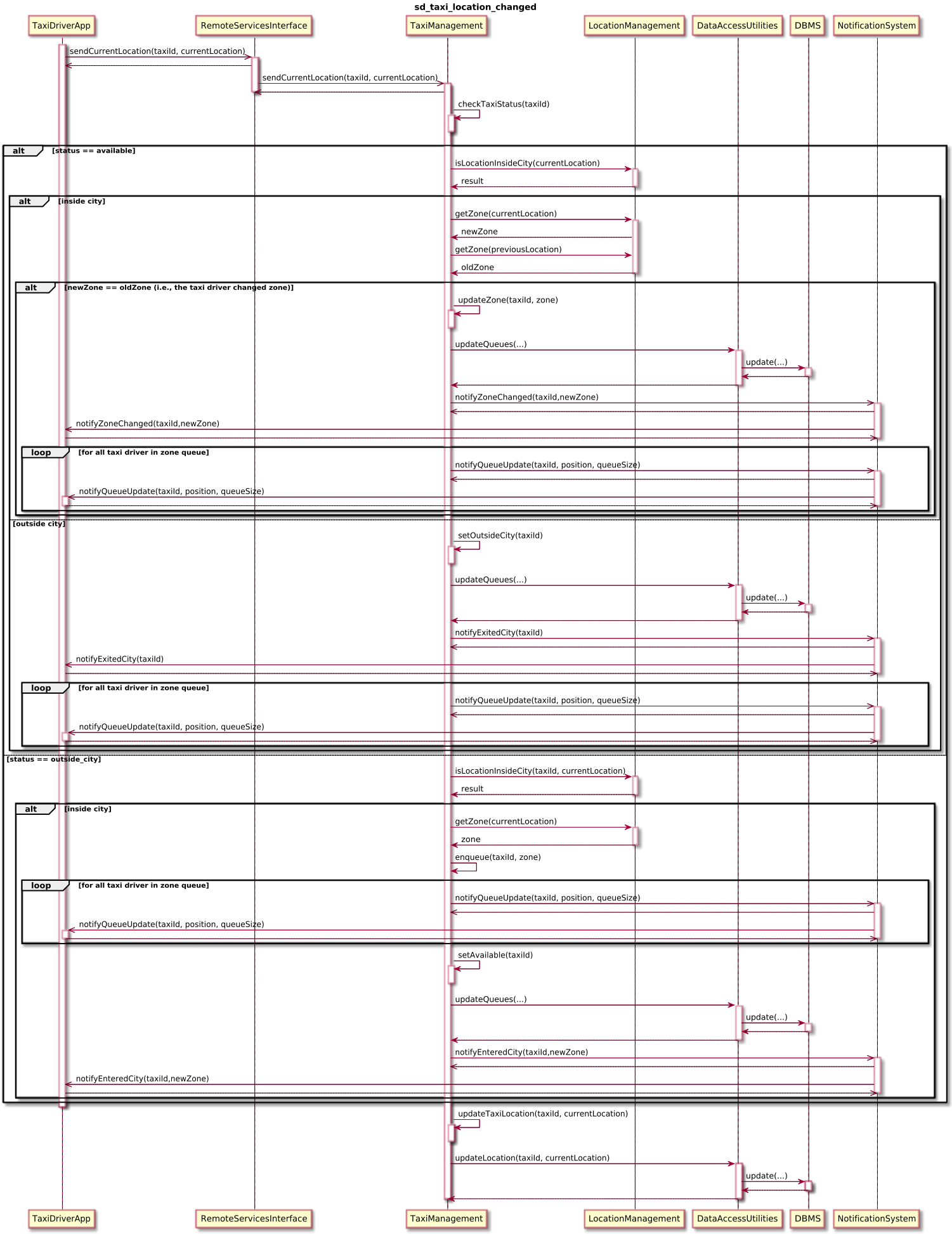

sd_allocate
RequestManagement
LocationManagement
TaxiManagement
NotificationSystem
TaxiDriverApp
RemoteServiceInterface
MappingService
PassengerApp
DataAccessUtilities
DBMS
getZone(passengerLocation)
zone
existsAvailableTaxiDriver(request, zone)
loop
[until a taxi driver accepts the call, or there are no more taxi drivers left]
dequeueTaxiDriver(zone)
loop
[for all taxi driver in zone queue]
notifyQueueUpdate(taxiId, position, queueSize)
notifyQueueUpdate(taxiId, position, queueSize)
setPending(taxiDriver)
notifyCallRequest(taxiDriver, request)
notifyCallRequest(taxiDriver, request)
alt
[accept ride]
acceptRide(taxiDriver, request)
acceptRide(taxiDriver, request)
setCurrentlyRiding(taxiDriver, request)
[reject ride or taxi driver doesn't reply on time]
refuseRide(taxiDriver, request)
refuseRide(taxiDriver, request)
enqueueTaxiDriver(taxiDriver, zone)
loop
[for all taxi driver in zone queue]
notifyQueueUpdate(taxiId, position, queueSize)
notifyQueueUpdate(taxiId, position, queueSize)
result
alt
[successful allocation (i.e., a taxi driver accepted)]
getAvailableTaxiDriver(request)
taxiDriver
getETA(taxiDriver, passengerLocation)
computeETA(passengerLocation, taxiDriverLocation)
ETA
ETA
generateSecretCode()
notifyRequestConfirmation(taxiDriver, request, secretCode)
confirmRide(secretCode)
notifyRequestConfirmation(passenger, request, secretCode)
confirmRide(secretCode)
insertReservation(...)
insert(...)
[result == false (i.e., no taxi driver accepted)]
notifyRequestRejection(passengerId)
notifyRequestRejection(passengerId)

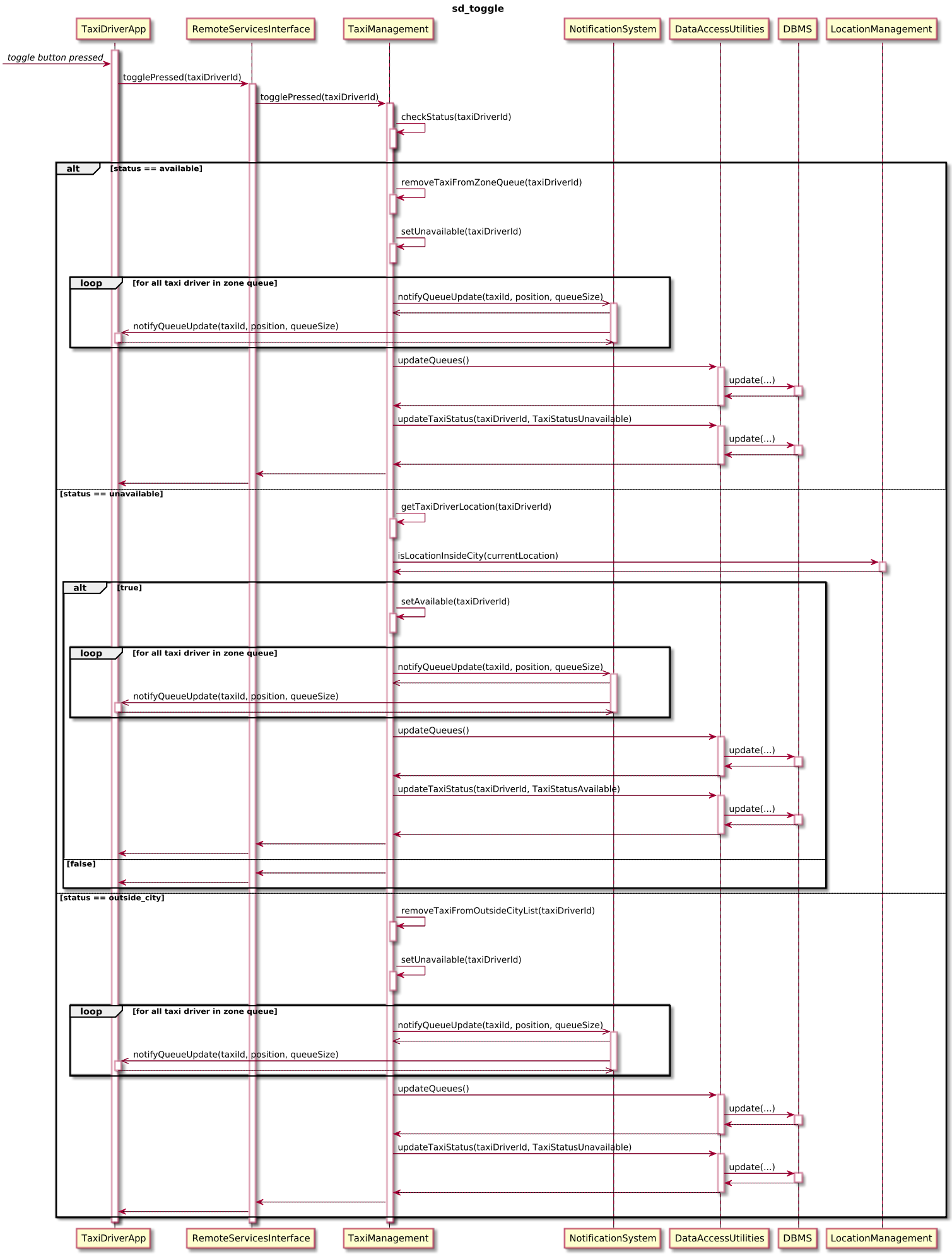

sd_show_reservations
PassengerApp
RemoteServicesInterface
ReservationManagement
getReservationList(passengerId)
getReservationList(passengerId)
retrieveReservationList(passengerId)
list
list
PassengerApp
RemoteServicesInterface
ReservationManagement


sd_zone_update
AdministrationWebApp
RemoteServicesInterface
ZoneManagement
PermissionManagament
DataAccessUtilities
DBMS
open zone update form
askZoneList(adminId)
askZoneList(adminId)
zoneList
zoneList
show form
loop
[until confirmation or exit]
data insertion
sendZoneData(data)
sendZoneData(data)
checkZonesConsistency(data)
result
alt
[correct data]
confirm
show confirmation
insert password
sendPassword(adminId, psw)
sendPassword(adminId, psw)
checkPassword(adminId, psw)
alt
[password confirmed]
updateZone(data)
updateZone(data)
update(...)
confirm
show confirmation
[password rejected]
reject
show password error
[wrong data]
dataError
show data error
AdministrationWebApp
RemoteServicesInterface
ZoneManagement
PermissionManagament
DataAccessUtilities
DBMS


sd_drop_call
TaxiDriverApp
RemoteServicesInterface
RequestManagement
LocationManagement
TaxiManagement
NotificationSystem
DataAccessUtilities
DBMS
PassangerApp
dropRequest(taxiId)
dropRequest(taxiId)
getRequestForTaxi(taxiId)
getMeetingPointForRequest(request)
checkTaxiDriverLocation(currentLocation, meetingPoint)
result
alt
[taxi driver has reached the meeting location]
taxiDriverDroppedRequest(taxiId)
isLocationInsideCity(currentLocation)
alt
[taxi driver inside]
setAvailable(taxiId)
getZone(currentLocation)
zone
enqueue(zone, taxiId)
loop
[for all taxi driver in zone queue]
notifyQueueUpdate(taxiId, position, queueSize)
notifyQueueUpdate(taxiId, position, queueSize)
[taxi driver outside city]
setOutsideCity(taxiId)
update(...)
update(...)
notifyCallDropped(passengerId)
notifyCallDropped(passengerId)
ok
ok
notAllowed
notAllowed
TaxiDriverApp
RemoteServicesInterface
RequestManagement
LocationManagement
TaxiManagement
NotificationSystem
DataAccessUtilities
DBMS
PassangerApp


sd_end_ride
TaxiDriverApp
RemoteServicesInterface
TaxiManagement
LocationManagement
NotificationSystem
DataAccessUtilities
DBMS
endRide(taxiDriverId, currentLocation)
endRide(taxiDriverId, currentLocation)
checkEndRideFunctionalityIsAllowed(taxiDriverId)
alt
[ok]
isLocationInsideCity(currentLocation)
result
alt
[taxi driver inside city]
setAvailable(taxiDriverId)
getZone(currentLocation)
zone
enqueue(zone, taxiDriverId)
loop
[for all taxi driver in zone queue]
notifyQueueUpdate(taxiId, position, queueSize)
notifyQueueUpdate(taxiId, position, queueSize)
setOutsideCity(taxiDriverId)
update(...)
update(...)
ok
ok
[ko]
notAllowed
notAllowed
TaxiDriverApp
RemoteServicesInterface
TaxiManagement
LocationManagement
NotificationSystem
DataAccessUtilities
DBMS


sd_taxi_location_changed
TaxiDriverApp
RemoteServicesInterface
TaxiManagement
LocationManagement
DataAccessUtilities
DBMS
NotificationSystem
sendCurrentLocation(taxiId, currentLocation)
sendCurrentLocation(taxiId, currentLocation)
checkTaxiStatus(taxiId)
alt
[status == available]
isLocationInsideCity(currentLocation)
result
alt
[inside city]
getZone(currentLocation)
newZone
getZone(previousLocation)
oldZone
alt
[newZone == oldZone (i.e., the taxi driver changed zone)]
updateZone(taxiId, zone)
updateQueues(...)
update(...)
notifyZoneChanged(taxiId,newZone)
notifyZoneChanged(taxiId,newZone)
loop
[for all taxi driver in zone queue]
notifyQueueUpdate(taxiId, position, queueSize)
notifyQueueUpdate(taxiId, position, queueSize)
[outside city]
setOutsideCity(taxiId)
updateQueues(...)
update(...)
notifyExitedCity(taxiId)
notifyExitedCity(taxiId)
loop
[for all taxi driver in zone queue]
notifyQueueUpdate(taxiId, position, queueSize)
notifyQueueUpdate(taxiId, position, queueSize)
[status == outside_city]
isLocationInsideCity(taxiId, currentLocation)
result
alt
[inside city]
getZone(currentLocation)
zone
enqueue(taxiId, zone)
loop
[for all taxi driver in zone queue]
notifyQueueUpdate(taxiId, position, queueSize)
notifyQueueUpdate(taxiId, position, queueSize)
setAvailable(taxiId)
updateQueues(...)
update(...)
notifyEnteredCity(taxiId,newZone)
notifyEnteredCity(taxiId,newZone)
updateTaxiLocation(taxiId, currentLocation)
updateLocation(taxiId, currentLocation)
update(...)
TaxiDriverApp
RemoteServicesInterface
TaxiManagement
LocationManagement
DataAccessUtilities
DBMS
NotificationSystem


sd_toggle
TaxiDriverApp
RemoteServicesInterface
TaxiManagement
NotificationSystem
DataAccessUtilities
DBMS
LocationManagement
toggle button pressed
togglePressed(taxiDriverId)
togglePressed(taxiDriverId)
checkStatus(taxiDriverId)
alt
[status == available]
removeTaxiFromZoneQueue(taxiDriverId)
setUnavailable(taxiDriverId)
loop
[for all taxi driver in zone queue]
notifyQueueUpdate(taxiId, position, queueSize)
notifyQueueUpdate(taxiId, position, queueSize)
updateQueues()
update(...)
updateTaxiStatus(taxiDriverId, TaxiStatusUnavailable)
update(...)
[status == unavailable]
getTaxiDriverLocation(taxiDriverId)
isLocationInsideCity(currentLocation)
alt
[true]
setAvailable(taxiDriverId)
loop
[for all taxi driver in zone queue]
notifyQueueUpdate(taxiId, position, queueSize)
notifyQueueUpdate(taxiId, position, queueSize)
updateQueues()
update(...)
updateTaxiStatus(taxiDriverId, TaxiStatusAvailable)
update(...)
[false]
[status == outside_city]
removeTaxiFromOutsideCityList(taxiDriverId)
setUnavailable(taxiDriverId)
loop
[for all taxi driver in zone queue]
notifyQueueUpdate(taxiId, position, queueSize)
notifyQueueUpdate(taxiId, position, queueSize)
updateQueues()
update(...)
updateTaxiStatus(taxiDriverId, TaxiStatusUnavailable)
update(...)
TaxiDriverApp
RemoteServicesInterface
TaxiManagement
NotificationSystem
DataAccessUtilities
DBMS
LocationManagement

2.6 Component interfaces

In this section we're going to describe the interfaces through which the various components of the system communicate. For each interface we'll provide a summary and a description of the main exposed methods.

It should be noted that, in order to provide a greater level of detail, we will actually distinguish between the various interfaces exposed by each subcomponent of every single component instead of limiting ourselves to the highest level components.

In the implementation phase, many of these interfaces will actually be further specialized in different classes as needed. The **ITaxiManagementSystem** interface is actually an umbrella interface which covers the following different sub-interfaces:

- The **IReservationManagement** interface, which represents the way components can interact with the ReservationManagement sub-component. It exposes the following methods:
  - reserveTaxi(passengerId, origin, destination, date, time): schedules a new reservation.
  - getReservationList(passengerId): returns the list of reservations placed by a passenger.
- The **IRequestManagement** interface, which represents the way components can interact with the RequestManagement sub-component. It exposes the following methods:
  - dropRequest(taxiId): entry point for requesting a call drop operation.
  - requestTaxi(passengerId, passengerLocation, destination): implements the necessary operations to request a taxi.
- The **ILocationManagement** interface, which represents the way components can interact with the LocationManagement sub-component. It exposes the following methods:
  - checkTaxiDriverLocation(currentLocation, meetingPoint): verifies that the taxi driver has effectively reached both the meeting point.
  - isLocationInsideCity(location): returns true if the location belongs to the city area, false otherwise.
  - getZone(location): if the location belongs to the city, returns the zone to which it belongs, otherwise it raises an exception.

The **ITaxiManagement** interface, which represents the way components can interact with the TaxiManagement sub-component. It exposes the following methods:

- taxiDriverDroppedRequest(taxiId): verifies if the taxi driver is allowed to drop the request and, if so, performs the necessary operations to drop it.
- endRide(taxiId, currentLocation): performs the necessary operations to register that the driver has ended a ride.
- acceptRide(taxiDriver, request): registers that the driver has accepted to serve the specified request.
- refuseRide(taxiDriver, request): registers that the driver has refused to serve the specified request.
- existsAvailableTaxiDriver(request, zone): verifies if there is a taxi driver able to serve the given request and, if so, contacts him and waits for his response; if no taxi driver is available to serve the request, it returns false.
- getAvailableTaxiDriver(request): it returns the taxi driver that has been associated to the given request.
- getETA(taxiDriver, passengerLocation): returns the estimated time of arrival (ETA) of the taxi driver to the given location.
- sendCurrentLocation(taxiId, location): updates the current location of a taxi and performs the necessary update operations to the queues.
- togglePressed(taxiDriverId): updates the availability status of a taxi driver given its previous availability status and the position of the availability toggle.

For the same reason, the **ISystemAdministration** interface actually comprises many different interfaces, one for each sub-component of System Administration:

- The **IAPIPermissionsManagement** interface, which represents the way components can interact with the API Permissions Management sub-component. It exposes the following methods:
  - checkPassword(adminId, password): verifies that the logged in user is an administrator and that the security password to enable the usage of a certain critical functionality is correct.
  - verifyPermission(appId, operation): verifies that the remote application associated with the appId security token has sufficient privileges to execute the desired operation.

- The **IZoneDivisionManagement** interface, which represents the way components can interact with the Zone Division Management sub-component. It exposes the following methods:
  - sendData(data): receives a set of candidate zones that will have to be checked for consistency.
  - insertZones(data): receives a set of candidate zones that have already been checked for consistency and inserts them inside the database.
  - updateZones(data): updates the data associated with the specified zones inside the database; if a zone doesn't exist yet it is added, while if a zone is marked as ready for deletion it is removed.
  - askZoneList(adminId): returns the set of zones stored in the database.
- The **ITaxiDriverManagement** interface, which represents the way components can interact with the Taxi Driver Management sub-component. It exposes the following methods:
  - sendData(data): receives a set of candidate taxi driver data items that will have to be checked for consistency. The actual set of data attributes is described in depth in the RASD.
  - insertZones(data): receives a set of candidate taxi driver data items that have already been checked for consistency and inserts them inside the database.The actual set of data attributes is described in depth in the RASD.
  - askDriverList(adminId): returns the set of taxi drivers stored in the database.
  - updateDrivers(data): updates the data associated with the specified taxi drivers inside the database; if a taxi driver doesn't exist yet it is added, while if a taxi driver is marked as ready for deletion it is removed.
- The **IServiceStatistics** interface, which represents the way components can interact with the Service Statistics sub-component.
- The **IPluginManagement** interface, which represents the way components can interact with the Plugin Management sub-component.

As for the AccountManagement component, access to its functionalities is made possible by the **IAccountManagement** interface. Much like

previous components, actual operations are carried out using the interfaces exposed by its sub-components, which are the **IPassengerRegistration**, **ILogin**, **IPasswordRetrieval** and **ISettingsManagement** interfaces. Specific methods for these components haven't been defined at this stage. Functionalities offered by the Data Access Utilities component are offered through its **IDataAccess** interface, which exposes a set of methods through which components can easily access the database both for read and write operations. Update, insert, delete and select queries are directly supported and exposed for each kind of key data type; a few examples of these methods can be identified in the sequence diagrams. The **INotificationSystem** interface let components request all sort of notification services. In particular, it allows both unicast and multicast notifications via the following methods:

- notifyCallRequest(taxiDriver, request): notifies a taxi driver that he's been assigned a request.
- notifyRequestConfirmation(user, request, secretCode): notifies a user (which can be a taxi driver or a passenger) that the specified request has been successfully handled and gives him the corresponding security code.
- notifyRequestRejection(passengerId): notifies the passenger that his request has been rejected.
- notifyCallDropped(passengerId): notifies the passenger that his request has been dropped.
- notifyErrorInRequest(passengerId): notifies the passenger that his request could not be fulfilled because the required data has been found inconsistent.
- notifyZoneChanged(taxiId, newZone): sends a notification to the taxi driver identified by taxiId to alert him that his zone has changed.
- notifyExitedCity(taxiId): sends a notification to the taxi driver identified by taxiId to alert him that he has exited the city.
- notifyEnteredCity(taxiId): sends a notification to the taxi driver identified by taxiId to alert him that he has entered the city.
- notifyQueueUpdate(taxiId, position, queueSize): sends a notification to the taxi driver identified by taxiId to alert him that the status of his queue has changed; specifically, it sends him his current position in the queue and the total number of taxis queuing in his zone.

Finally, the **IRemoteServices** interface let remote applications invoke a specific service offered by the central system. In particular, the following methods are exposed:

- sendSetupData(data): receives the initialization data for the system, which is made of tuples containing zone configuration data and taxi drivers data.
- sendZoneData(data): receives a set of candidate zones.
- sendTaxiData(data): receives a set of candidate taxi drivers.
- sendPassword(adminId, password): receives the security password required to perform critical operations and performs the necessary validation procedures.
- askZoneList(adminId): returns the list of all stored zones.
- acceptRide(taxiDriver, request): notifies the central system that the specified taxiDriver has accepted the specified request.
- refuseRide(taxiDriver, request): notifies the central system that the specified taxiDriver has refused the specified request.
- dropRequest(taxiId): allows a taxi driver to request a call drop operation.
- endRide(taxiDriverId, currentLocation): notifies the central system that the specified taxiDriver has ended his currently assigned ride, and also provides his current location.
- requestTaxi(passengerId, passengerLocation, destination): allows a passenger to request a taxi at his current location for a specified destination.
- reserveTaxi(passengerId, origin, destination, date, time): allows a passenger to reserve a taxi for a journey going from the specified origin to the specified destination, with pickup scheduled at the given date and time.
- getReservationList(passengerId): returns the list of reservations placed by the specified passenger.
- sendCurrentLocation(taxiId, location): lets the taxi driver application periodically notify his new coordinates.
- togglePressed(taxiDriverId): notifies the central system that the specified taxi driver has switched the availability toggle in his mobile application.

2.7 Selected architectural styles and patterns

While designing the software architecture for myTaxiService we looked at all the possible options that enabled us both to satisfy current functional and non-functional requirements and to have enough flexibility for future needs and updates.

In this section we're going to highlight the main architectural styles and patterns that we've decided to use as part of our software architecture, giving a brief description for each of them and explaining why we chose them.

- Software Oriented Architecture (SOA). This architectural style allows us to design our system as a set of services which can be separately offered to third parties. The adoption of this architectural style provides several benefits. First, it allows us to easily implement the APIs that must be exposed by the core system, which can be seen as a set of services offered by myTaxiService to remote clients and applications. Second, this give us great flexibility when it comes to further expansions of the set of supported functionalities: by modeling our system as a SOA, it is possible to start offering a set of core services and then increase the number of public APIs as additional components and services are developed.

- Layered Architecture. This architectural style enables a great level of separation of concerns between the various components of the system, such that every component operates at a single logical layer of abstraction. Furthermore, each layer can be separately instantiated on a different machine in a completely orthogonal way, allowing great flexibility in terms of hardware configurations. Finally, this design choice allows us to obtain a greater level of clarity and flexibility: this simplifies the implementation phase, makes testing easier and more effective and dramatically improves maintainability and extendability.

- N-Tier Physical Architecture. This architectural style allows us to run the various components of the system on different devices, not necessarily located in the same server farm. This flexibility in terms of allocation of hardware resources provides a number of key advantages. First of all, it improves security and resilience with respect to attacks: in fact it clearly separates the hardware which runs processes directly connected to the outside world (the web servers) from the hardware running more sensitive and critical services, like the account manager, the administration services and the taxi management system. Furthermore, this gives us the opportunity to easily scale the number of machines of each tier to more efficiently adapt to variations of demand.

- Publish/Subscribe. This architectural style is primarily used in the implementation of the notification system. Even though one-to-one

communications between the central system and remotely connected devices could be achieved without the usage of the publish/subscribe architectural style, this design choice provides us greater flexibility with respect to future expansions of the core functionalities. An example of situation in which this architectural style could be useful is the addition of taxi sharing.

2.8 Other design decisions

The main infrastructure of the system, which comprises the Remote Services Interface, the Taxi Management System, the System Administration, the Account Management and the Data Access Utilities components will be implemented using the Java Enterprise Edition platform. This design decision has been taken due to the several advantages that this platform has over competing solutions:

- It supports multi-tiered applications
- It is highly reliable, thanks to its ability to automatically manage multiple instances of the same component
- It supports interoperability with different platforms by using SOAP to expose web services to remote clients
- Applications are run inside a managed environment which provides support for reliable database transactions, secure network communications and authentication protocols, thus enabling developers to concentrate their efforts on the peculiar functionalities of myTaxiService
- There are many highly efficient and robust application servers which support load balancing

As for the mobile applications, we have decided to use the native libraries offered by each mobile platform in order to achieve a better look & feel and greater performance. This decision comes at the expense of portability of the mobile application code. However, this doesn't actually represent a problem, for three main reasons:

- The amount of code contained in each mobile application is extremely small compared to the whole code base of the project, since it's only related to presentation functionalities
- All the application logic is already performed on the server and is exposed to mobile applications via standard SOAP web services, which are accessible by any platform without the need to reuse code on the client side

- The look & feel of each platform is peculiar and would require specific libraries to be achieved anyway

The web applications will be implemented using HTML5 technology, in order to be as standard-compliant as possible.

No specific choice has been made as for the DBMS, except for the fact that it will need to be a relational database management system with support for distributed data storage.

Chapter 3

Algorithm Design

In this section we will give an overall description of the main algorithms necessary for myTaxiService to work.

Each algorithm will be explained using a pseudo-code notation inherited from the Java programming language; for the sake of simplicity, we will assume to have a class for each sub-component, even though in the real implementation the code will be split among multiple helper classes.

The first algorithm we will focus on is the Taxi Request processing algorithm, which takes a Taxi Request in input and performs the necessary operations to handle it. The main algorithm is described by the requestTaxi(...) method of the RequestManagement component.

The most critical aspect of the algorithm is the proper management of the queues in a concurrent environment. In fact, every request is processed independently in a different thread for efficiency reasons: this means that accesses to queues must be managed in a way that ensures proper and consistent sorting of taxis.

This is achieved by using an additional mapping data structure in each thread that associates to every taxi contained in the queue of the interested zone a special checked/unchecked status. At the beginning of the requestTaxi() method, all taxis in the interested zone are inserted in the map with an unchecked status. A taxi is then considered checked if the algorithm has already considered it for the given request and the taxi driver has refused it; this way, it won't be considered again for the same request.

In order to keep this map data structure updated with respect to the insertion and removal of taxis in/from the interested zone queue, an instance of the observer design pattern is implemented between the TaxiManagement component and the zone queue object. Specifically, the thread responsible of a specific request registers itself as an observer of the interested zone queue object by exposing a couple of callbacks that are called every time a taxi is inserted or removed in/from the zone queue. The behavior of these callbacks is designed to keep the internal thread data structure consistent

with the behavior of a queue; more specifically:

- When a taxi is removed from the original zone queue, it is removed from the map only if it hasn't already been checked. In fact, to guarantee convergence the algorithm must remember the taxis that have already been checked. If the taxi is still marked as unchecked, then it can be safely removed; in this case, two scenarios are possible:

  1. The taxi has been removed because it's become unavailable or it has gone out of the zone.
  2. The taxi has been selected to serve another request.

  In both cases, it is correct to remove the taxi from the map, since it wouldn't be available to serve our request anyway. The removal of the taxi from the map allows the system to insert it back later on if it becomes available again.

- When a taxi is added to the original zone queue, it is added to the map only if it hasn't already been stored inside it. In this way, a taxi can be stored in the map only if one of these two conditions holds:

  1. The taxi is coming from another zone. In this case, it represents a new taxi for us that wasn't previously available for selection.
  2. The taxi was already present in our zone, it had been selected to serve another request but has refused to serve it. In this case, the taxi could still choose to serve our request, so it must be considered for selection.

  In both cases, the taxi is added into the last position of the map. This is consistent with the specifications, as it will respect the priority order of the queue and will allow the system to consider more taxis as they become suitable for selection.

It should be noted that this add/remove policy ensures that no taxi is ever considered twice for the same request: even if it is taken from the pool to serve another request, if it had already been considered for this one his checked status will be kept and this will avoid multiple checks of the same taxi.

In the end, this guarantees that the algorithm converges in at most N iterations, N being the total number of taxis in the system (if they are all available and present inside the same zone, and all of them refuse the call).

```
1 public class RequestManagement{
public boolean requestTaxi(passengerID, address){
3   Location l = locationManagement.getLocation(address);
   return requestTaxi(passengerId, l);
5 }
```

```
public boolean requestTaxi(passengerID, passengerLocation,
    destination){
  boolean consistency = checkDataConsistency(passengerID,
    passengerLocation, destination);
  if (consistency){
    // We create a new request and write it into the database
    Request request = new Request(passengerID, passengerLocation
    );
    dataAccessUtilities.writeToDB(request);
    /* We compute the zone of the source */
    Zone z = locationManagement.getZone(passengerLocation);
    /* Is there an available taxi? */
    if (taxiManagement.existsAvailableTaxiDriver(request, zone))
    {
      /* If so, get it and compute the ETA and the secret code
    */
      TaxiDriver td = taxiManagement.getAvailableTaxiDriver(
    request);
      int ETA = taxiManagement.getETA(td, passengerLocation);
      String secretCode = generateSecretCode();
      /* We notify it to the taxi driver and to the passenger*/
      notificationSystem.notifyRequestConfirmation(td, request,
    secretCode);
      notificationSystem.notifyRequestConfirmation(passenger,
    request, secretCode)
  }
    else
      /* Otherwise we reject the request */
        notificationSystem.notifyRequestRejection(passenger);
    /* Finally we update the status of the request in the
    database */
    dataAccessUtilities.updateDB(request);
  }
  else
    /* Data is inconsistent, notify error */
    notificationSystem.notifyErrorInRequest(passengerID);
  return consistency;
}

private boolean checkDataConsistency(origin, destination){
  Location origin_l = locationManagement.getLocation(address);
  if (locationManagement.isLocationInsideCity(origin_l)
    return true;
  return false;
}
}

public class LocationManagement{
public Location getLocation(address){
  return mappingService.getLocation(address);
}
}
```

```
51 public class RemoteServicesInterface{
   public void acceptRide(td, request){
53   taxiManagement.acceptRide(td, request);
   }
55 }

57 public class TaxiManagement{

59 private boolean accepted;
   private boolean answered;
61 /* We keep a list of already checked taxis */
   private SortedMap checkedTaxi = new SortedMap();
63
   private void notifyQueueUpdateToTaxis(zone){
65   ZoneQueue queue = getZoneQueue(zone);
     for (int i = 0; i<queue.getSize(); i++)
67     notificationSystem.notifyQueueUpdate(queue.get(n), n, queue.
       getSize());
   }
69

71 public void acceptRide(td, request){
     answered = true;
73   accepted = true;
   }
75
   public void refuseRide(td, request){
77   answered = true;
     accepted = false;
79 }

81 public int getETA(passengerLocation){
     return mappingService.computeETA(passengerLocation, td.
       getLocation());
83 }

85 private void setPending(td){
     td.setStatus(pending);
87   dataAccessUtilities.updateTaxiStatus(td.getStatus());
   }
89
   private void setCurrentlyRiding(td, request){
91   td.setStatus(currentlyRiding);
     moveTaxiToCurrentlyRidingList(td);
93   td.associateRequest(request);
     dataAccessUtilities.updateTaxiStatus(td.getStatus());
95 }

97 /* Callback invoked by the zone queue update method when a new
       taxi driver is inserted in the zone queue */
   public void callbackNotifyAddedTaxi(TaxiDriver td){
99   if (!checkedTaxi.contains(td))
       checkedTaxi.add(td, UNCHECKED);
101 }
```

```
103 /* Callback invoked by the zone queue update method when a taxi
    driver is removed from the zone queue */
105 public void callbackNotifyRemovedTaxi(TaxiDriver td){
      if (checkedTaxi.getValue(td)==UNCHECKED)
107     checkedTaxi.remove(td);
    }
109
    /* The queue is frozen while I'm filling the hashmap */
111 private synchronized void initializeHashMap(ZoneQueue queue){
      for (int i = 0; i < zoneQueue.getSize(); i++)
113     checkedTaxi.add(zoneQueue.get(i), UNCHECKED);
    }
115
    public boolean existsAvailableTaxiDriver(zone){
117
      ZoneQueue zoneQueue = getZoneQueue(zone);
119   initializeHashMap(zoneQueue);
      zoneQueue.addObserver(this);
121   accepted = false;
      answered = false;
123   /* While there are still taxis to be checked and nobody has
        accepted */
      for (int i = 0; !accepted && (i < checkedTaxi.getSize()); i++)
125   {
        /* We pop the first available taxi */
127     TaxiDriver td = checkedTaxi.get(i);
        zoneQueue.remove(td);
129     /* We set it as pending */
        setPending(td);
131     /* And notify him the request */
        notificationSystem.notifyCallRequest(td, request);
133     /* Wait for the answer, or timeout */
        waitUntil(answered || timeout);
135     if (!accepted){
          zoneQueue.enqueue(td);
137       checkedTaxi.setValueForKey(i, CHECKED);
        }
139     else
          setCurrentlyRiding(td, request);
141     /* In both cases, we have to notify other drivers of the
        update of the queue */
        notifyQueueUpdateToTaxis(zone);
143   }
      return accepted;
145 }

147 }
```

The second algorithm we will focus on is the Taxi Reservation processing algorithm, which takes a Taxi Reservation in input and performs the necessary operations to handle it. The main algorithm is described by the reserveTaxi(...) method of the ReservationManagement component.

```
public class ReservationManagement{

  public boolean reserveTaxi(passengerID, origin, destination,
    date, time){
    /* Verifies that inserted data are consistent and valid */
    boolean consistency = checkDataConsistency(passengerID,
    origin, destination, date, time);
    if (consistency){
      /* Allocate the reservation and store it into the database
     */
      Reservation reservation = new Reservation(origin,
    destination, date, time);
      insertNewReservation(reservation);
      /* Notify the passenger that the reservation has been
    registered */
      notificationSystem.confirmReservation(passengerID,
    reservation);
    }
    else
      /* Data is inconsistent, notify error */
      notificationSystem.notifyErrorInReservation(passengerID);
    return consistency;
  }

  private insertNewReservation(reservation){
    dataAccessUtilities.insertReservation(reservation);
  }

  private boolean checkDataConsistency(origin, destination, date
    , time){
    Location origin_l = locationManagement.getLocation(address);
    if (locationManagement.isLocationInsideCity(origin_l) &&
    (timeInterval(currentDateTime(),createDateTime(date,time))
    between 2 hours and 15 days))
      return true;
    return false;
  }
}
```

The third algorithm we will focus on is the one that manages the behavior of the availability toggle for taxi drivers. The main algorithm is described by the togglePressed(taxiDriverId) method of the TaxiManagement component.

```
public class TaxiManagement{

  public void togglePressed(taxiDriverId){
    TaxiStatus status = checkStatus(taxiDriverId);
    Zone zone = locationManagement.getZone(taxiDriverId);
    switch(status){
      case TaxiStatusAvailable:{
        removeTaxiFromZoneQueue(taxiDriverId);
        setUnavailable(taxiDriverId);
```

```
10          dataAccessUtilities.updateQueues();
            dataAccessUtilities.updateTaxiStatus(taxiDriverId,
    TaxiStatusAvailable);
12          notifyQueueUpdateToTaxis(zone);
            break;
14        }
          case TaxiStatusOutsideCity:{
16          remoteTaxiFromOutsideCityList(taxiDriverId);
            setUnavailable(taxiDriverId);
18          dataAccessUtilities.updateQueues();
            dataAccessUtilities.updateTaxiStatus(taxiDriverId,
    TaxiStatusAvailable);
20          notifyQueueUpdateToTaxis(zone);
            break;
22        }
          case TaxiStatusCurrentlyRiding:{
24        /* A taxi cannot change his status while on a ride */
          notificationSystem.rejectAvailabilityChange(taxiDriverId);
26        }
          case TaxiStatusPending:{
28        /* A taxi cannot change his status while he's got a
    pending request */
          notificationSystem.rejectAvailabilityChange(taxiDriverId);
30        }
          case TaxiStatusUnavailable:{
32          Location l = getTaxiDriverLocation(taxiDriverId);
            if (locationManagement.isLocationInsideCity(l)){
34            setAvailable(taxiDriverId);
              notifyQueueUpdateToTaxis(zone);
36            notificationSystem.confirmAvailability(taxiDriverId);
            }
38          else
              notificationSystem.rejectAvailabilityChange(
    taxiDriverId);
40          break;
          }
42      }
    }
44
    private getTaxiById(taxiDriverId){
46    /* We have to get the corresponding taxi */
      /* First look through the zones to see if the driver is
      available */
48    for (int zone = 0; z < zoneNumber; z++){
        for (int n = 0; n < getZoneQueue(zone).getSize(); n++)
50        if (getZoneQueue(zone).get(n).getDriverId ==
    taxiDriverId)
            return getZoneQueue(zone).get(n);
52    }
      /* If not, is he outside city? */
54    for (int n = 0; n < outsideCityList.getSize(); n++)
        if (outsideCityList.get(n).getDriverId == taxiDriverId)
56        return outsideCityList.get(n);
      /* If not, is he on a ride? */
```

```
58      for (int n = 0; n < currentlyRidingList.getSize(); n++)
          if (currentlyRidingList.get(n).getDriverId == taxiDriverId
        )
60          return currentlyRidingList.get(n);
        /* If not, is he unavailable ? */
62      for (int n = 0; n < unavailabilityList.getSize(); n++)
          if (unavailabilityList.get(n).getDriverId == taxiDriverId)
64          return unavailabilityList.get(n);
        /* If we still haven't found him, we have a serious problem
        */
66      throw new TaxiNotFoundException();
    }
68
    private TaxiStatus checkStatus(taxiDriverId){
70    Taxi taxi = getTaxiById(taxiDriverId);
      return taxi.getStatus();
72  }
}
```

The fourth algorithm we will focus on is the one that manages the transition of a taxi from a zone to another. The main algorithm is described by the sendCurrentLocation(taxiId,currentLocation) method of the TaxiManagement component.

```
1  public class TaxiManagement{

3    public void sendCurrentLocation(taxiId,currentLocation){
       Taxi taxi = getTaxiById(taxiDriverId);
5      TaxiStatus status = checkStatus(taxiDriverId);
       Zone oldZone = locationManagement.getZone(taxi.getLocation()
       );
7      /* The zone change has an impact only if the taxi is
       available */
       if (status==TaxiStatusAvailable){
9        if (locationManagement.isLocationInsideCity(
       currentLocation)){
           /* If it's inside the city, it should change zone */
11         Zone newZone = locationManagement.getZone(
       currentLocation);
           if (oldZone!=newZone)){
13           updateZone(taxi,newZone);
             /* We have to notify the other drivers in the new zone
        of the change */
15           notifyQueueUpdateToTaxis(newZone);
             dataAccessUtilities.updateQueues();
17
           notificationManager.notifyZoneChanged(taxiId,newZone);
19         }
         }
21       else{
           /* Otherwise it should be set as "out of city" */
23         setOutsideCity(taxiId);
           dataAccessUtilities.updateQueues();
25         notificationManager.notifyExitedCity(taxiId);
```

```
        }
27      /* We have to notify the other drivers in the old zone of
    the change */
        notifyQueueUpdateToTaxis(oldZone);
29    }
    else if (status==TaxiStatusOutsideCity){
31    /* Are we getting inside the city? */
      if (locationManagement.isLocationInsideCity(currentLocation)
    ){
33      /* If so, we have to compute the zone in which we have
    entered */
        Zone newZone = locationManagement.getZone(currentLocation)
    ;
35        setAvailable(taxi);
          enqueueTaxiDriver(taxi,newZone);
37        /* We have to notify the other drivers in the new zone
    of the change */
          notifyQueueUpdateToTaxis(newZone);
39        dataAccessUtilities.updateQueues();
          notificationManager.notifyEnteredCity(taxiId,newZone);
41
      }
43    /* Otherwise we don't really care */
    }
45  updateTaxiLocation(taxiId,currentLocation);
    }
47
    private void updateTaxiLocation(taxi,currentLocation){
49    taxi.setLocation(currentLocation);
    }
51
}
```

Chapter 4

User Interface Design

One of the main concerns with designing an Information System is how to make it as functional and user-friendly as possible. The simplicity and clarity of the user interface is crucial in making sure that the system can actually be employed by the intended user base in an efficient way; in increases user satisfaction with the services and reduces the need to have a support staff. For this reason, we have designed the User Interfaces of our client applications in accordance to the main principles of UI Design outlined by Jef Raskin in his book "The Humane Interface", as referenced at the beginning of this document.

Another important aspect that we have considered while designing the UI of our client applications is related to the adaptability of the UI to different screens and device form factors.

In order to satisfy this requirement, the mobile applications have been designed to natively support both smartphones and tablets. More specifically, this implies that each mobile application actually includes two different user interfaces, each uniquely optimized and tailored to the specific kind of device being used. To keep the user experience consistent across devices, the two UIs are built around the same design language and share many visual traits.

Furthermore, the mobile applications UIs have been designed to be resolution independent. In particular, we have chosen an interface design which can be easily displayed on screen sizes between 3" and 6" (when used on a mobile phone) and between 7" and 12" (when used on a tablet); this requirement will be effectively satisfied in the implementation phase by making usage of appropriate libraries that support vector graphics, proportional spacing and scaling functionalities.

Both the Passenger Web Application and the Administration Web Application components have to be implemented to properly address dynamic web page resizing by proportionally scaling header, footer and side bars to the amount of available space without stretching individual components and

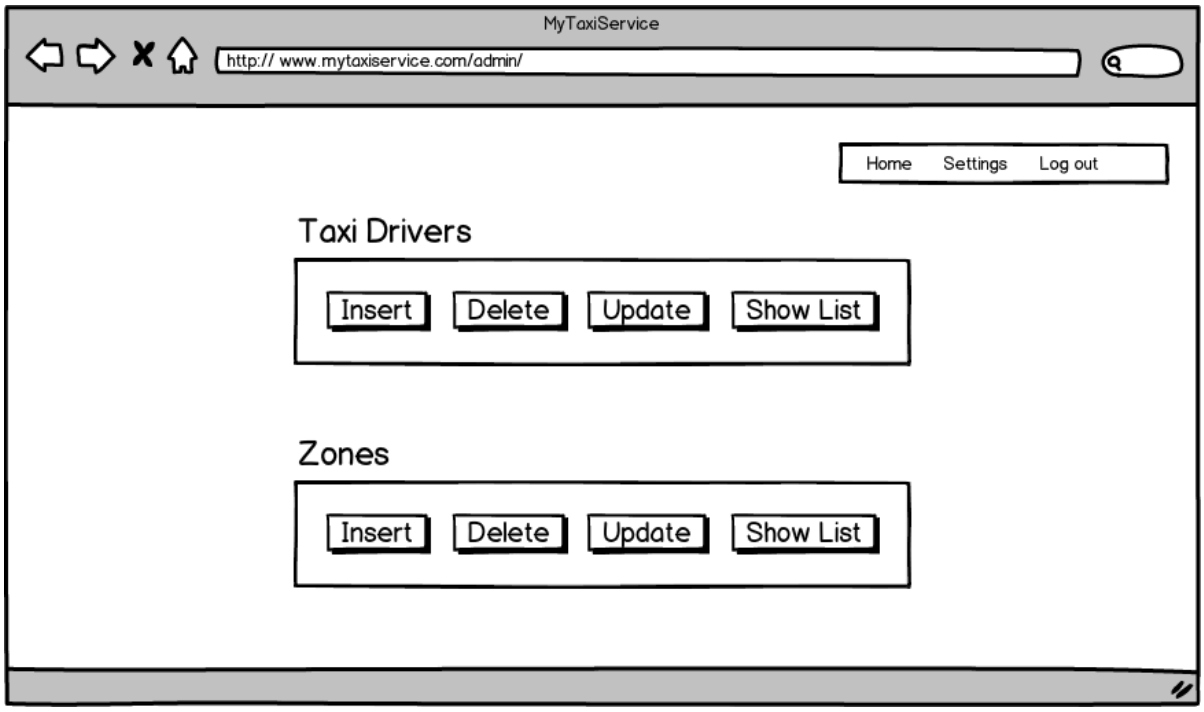
forms. Furthermore, they have to be offered both in a desktop version and in a mobile-friendly version; for simplicity, in the sketches only the desktop version is shown, with the mobile-friendly version closely resembling their application counterparts.

In addition, both the mobile applications and the web applications are implemented to be accessible by blind people. This requirement is translated differently for the two kinds of application:

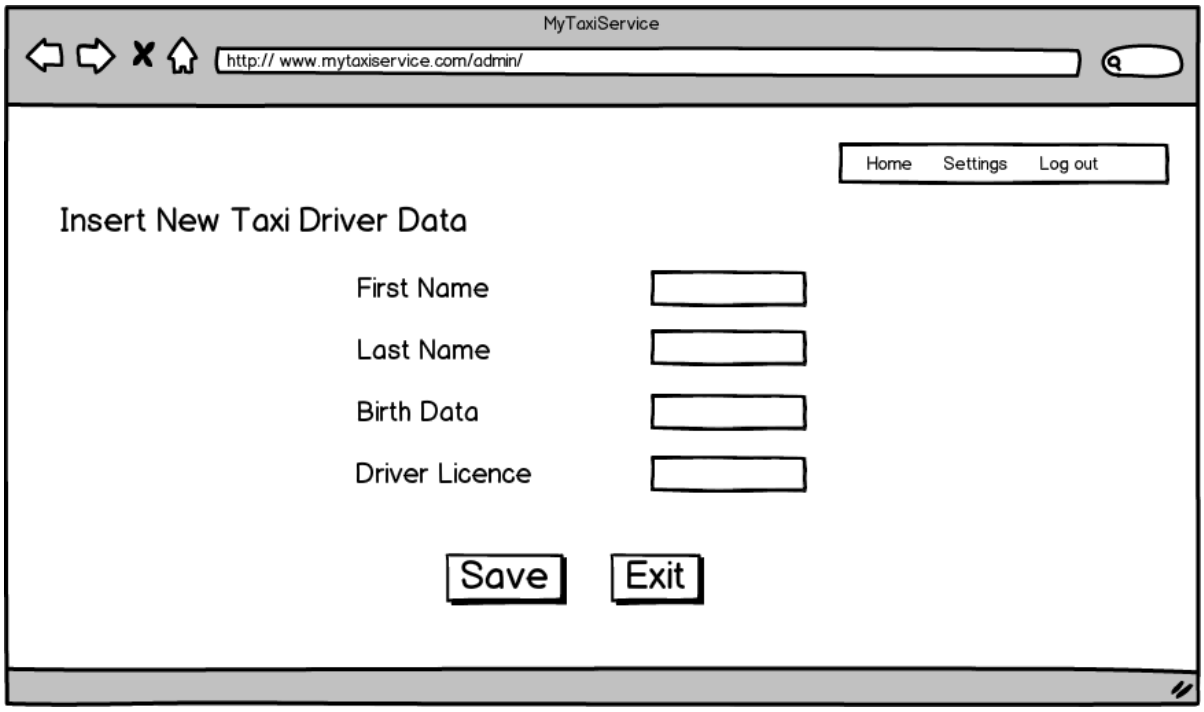
- The Web Applications must use standard HTML components and include the appropriate tags to be interpreted by screen reader software in a meaningful way
- The Mobile Applications must implement appropriate voice description APIs offered by the platform they're running on (as VoiceOver in iOS)

Many detailed mockups regarding the Taxi Driver Mobile Application, the Passenger Mobile Application and the Passenger Web Application have already been included in the RASD, so we will not not include them again in this document. However, we are including a few mockups of the Administration Web Application.

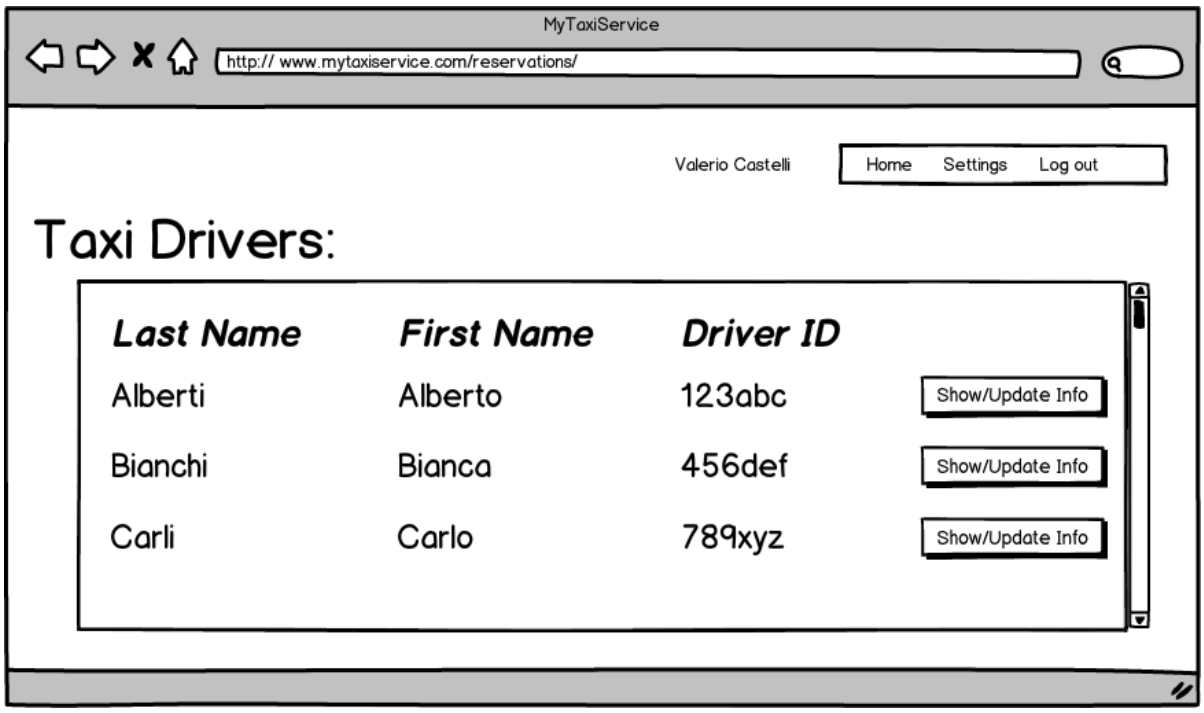
The following sketch shows a possible implementation of the main page of the admin web application.



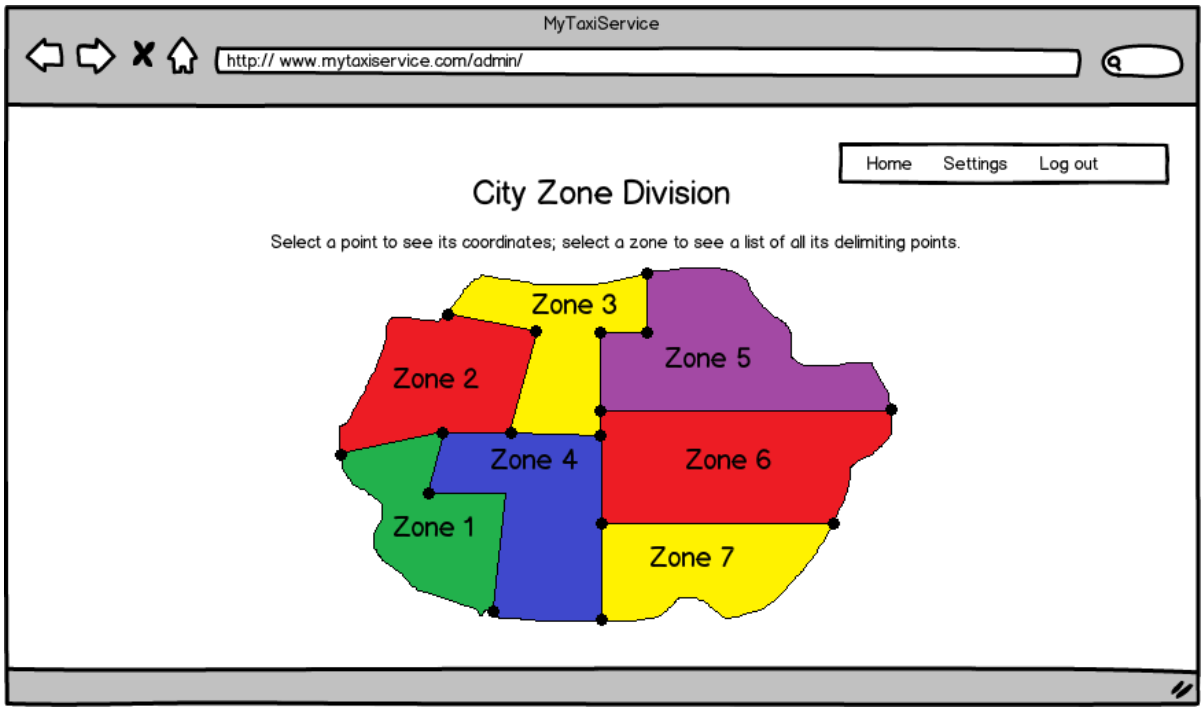
The following sketch shows how the taxi driver insertion page is designed.



The following sketch shows how the list of all taxi drivers is displayed.



The following sketch shows how the zone update page is designed.



Chapter 5

Requirements Traceability

In this section we'll specifically cover in detail how the requirements identified in the SRS document are satisfied by our myTaxiService architecture.

5.1 Functional Requirements

For sake of readability, we will use the notation **req. x.y** to make reference to the y-th requirement associated with the x-th goal.

- **Goal 1, 2 and 3** are essentially administration-oriented requirements; as such, they are satisfied by:
  - The **System Administration** component, which provides the backend business logic to handle the required data modification operations. In particular, operations related to taxi and taxi drivers (goal 1, 2) are handled by the Taxi Driver Management sub-component, while operations related to the definition of taxi zones (goal 1, 3) are handled by the Zone Division Management sub-component.
  - The **Administration Web Application** component, which provides administrators an appropriate user interface to interact with the backend.
- **Goal 4** describes the way in which the availability status of a taxi driver is updated. As such, several components are involved in its fulfillment:
  - The **Taxi Driver Application** component, which provides taxi drivers the ability to interact with the central system to notify a change in their availability status. In particular, this component is responsible for reqs. 4.1 to 4.6 and reqs. 4.8, 4.11.

  - The **Taxi Management System** component, which provides the backend business logic to handle the update operations. In particular:
    * Req. 4.6 and 4.7 is handled by the Location Management sub-component for the zone computation part and by the Taxi Management sub-component for the update operation on taxi queues
    * Reqs 4.9 and 4.10 are handled by the Taxi Management sub-component by moving the taxi to the unavailability list and setting its status to "unavailable"
    * Req. 4.8 and 4.11 are handled by the Taxi Management sub-component
  - The **Notification System** component, which provides to the central system the dispatch mechanism it needs for sending notifications to the taxi driver application. This component is essentially involved in satisfying reqs. 4.6, 4.8 and 4.11.

- **Goal 5** requires the allocation and distribution of taxis to be managed fairly and consistently. This is achieved by an interplay of components:
  - The **Taxi Management System** is the key component that concurs to the fulfillment of the goal. More specifically, reqs. 5.1 to 5.12 are handled by the Taxi Management sub-component, with the aid of the Location Management sub-component as for those requirements that involve checking whether the taxi driver is still inside the city or not.
  - The **Taxi Driver Application** component is crucial in fulfilling reqs 5.13 and 5.14 by providing an appropriate management of the UI to prevent incorrect availability assignments and by sending the GPS coordinates of the taxi driver to the central system with a given frequency.
  - The **Remote Services Interface** component, which enables the taxi driver application to request services to the central system and thus provides the necessary callbacks to satisfy req. 5.14.

- **Goal 6** is related to the ability of a taxi driver to receive, accept and refuse ride requests. This involves:
  - The **Taxi Management System** component, which is responsible of fulfilling reqs. 6.1, 6.2, and 6.4 to 6.6 by means of the Taxi Management sub-component.
  - The **Mapping Service** component, which provides the reverse-geocoding capability to fulfill req. 6.2 and the map data to fulfill req. 6.8.

  - The **Taxi Driver Application** component, which is responsible of fulfilling reqs. 6.1, 6.3, 6.7 and 6.8 by providing to the taxi driver an appropriate UI.
  - The **Notification System** component, which provides to the central system the dispatch mechanism it needs for sending notifications to the taxi driver application and fulfill req 6.1.
  - The **Remote Services Interface** component, which enables the taxi driver application to request services to the central system and thus provides the necessary callbacks to satisfy reqs. 6.4 and 6.5.

- **Goal 7** gives a taxi driver the ability to drop a request if the passenger doesn't show up. In order for this functionality to work correctly, several components should cooperate:

  - The **Taxi Driver Application** component, which provides a suitable UI to drop the request (req. 7.1) and retrieves the current GPS location fo the taxi driver in order to let the central system check if it matches the agreed meeting point (req. 7.2).
  - The **Taxi Management System** component, which is responsible of satisfying all the three requirements related to this goal, in particular my means of the Taxi Management and Request Management sub-components.
  - The **Passenger Application** component, which is responsible of implementing a suitable UI view to notify a passenger that his request has been dropped (req. 7.3).
  - The **Notification System** component, which provides to the central system the dispatch mechanism it needs for sending notifications to both the taxi driver application and the passenger application and fulfill req 7.3.
  - The **Remote Services Interface** component, which enables the taxi driver application to request services to the central system and thus provides the necessary callbacks to satisfy reqs. 7.1 and 7.2.

- **Goal 8** gives a taxi driver the ability to notify the central system when he has completed a ride. In order for this functionality to work correctly, several components should cooperate:

  - The **Taxi Driver Application** component, which provides a suitable UI to mark the end of the ride (req. 8.1) and retrieves the current GPS location fo the taxi driver in order to let the central system check if it matches the agreed meeting point (req. 8.2) and the destination point (req. 8.3).

  - The **Taxi Management System** component, which is responsible of satisfying all the five requirements related to this goal, in particular my means of the Taxi Management and Request Management sub-components. Also, the Location Management sub-component is specifically employed to satisfy req. 8.4 and 8.5.
  - The **Remote Services Interface** component, which enables the taxi driver application to request services to the central system and thus provides the necessary callbacks to satisfy reqs. 8.1 to 8.3.

- **Goal 9** is arguably one of the most important functionalities of the system, as it is related to the ability of passengers to request taxi rides. As such, it involves many different components:
  - The **Taxi Management System** component, which implements all the backend business logic to handle taxi requests. In particular:
    * Reqs. 9.3 and 9.4 are handled by the Request Management and the Location Management sub-components.
    * Req. 9.8 is handled by the Location Management sub-component, which is responsible for computing the taxi zone associated with a certain location.
    * Req. 9.9 to 9.16 are handled by the Taxi Management sub-component and, for those operations involving the status of a pending request, by the Request Management sub-component.
  - The **Mapping Service** component, which is used to calculate the ETA for req. 9.10 and to perform the reverse-geocode location needed to fulfill reqs. 9.3, 9.4 and also 9.8 if the location is an address.
  - The **Passenger Application** component, which provides a suitable UI to fulfill reqs. 9.1, 9.2.1, 9.2.2, 9.5 to 9.7 and 9.10.
  - The **Passenger Web Application** component, which provides a suitable UI to fulfill reqs. 9.1, 9.2.3 and 9.10.
  - The **Taxi Driver Application** component, which provides a suitable UI to alert the taxi driver he has been moved to the last position of its zone queue after refusing a request (req. 9.11).
  - The **Remote Services Interface** component, which enables the passenger applications (web and mobile) to request services to the central system and thus provides the necessary callbacks to satisfy req 9.2.

  - The **Notification System** component, which provides to the central system the dispatch mechanism it needs for sending notifications to both the taxi driver application and the passenger applications and fulfill reqs 9.10 to 9.12 and 9.16.

- **Goal 10** requires the system to offer passengers a way to register. This goal is satisfied by an interplay of four components:
  - The **Account Management** component, specifically the Passenger Registration sub-component, is responsible of offering the backend services that enables passenger registration and validation of the required data.
  - The **Passenger Web Application** component, which provides the registration entry form.
  - The **Remote Services Interface** component, which enables the passenger web application to request services to the central system and thus provides the necessary callbacks to invoke the registration procedure.
  - The **Notification System** component, which lets the central system send a notification containing the result of the registration process to the passenger web application.

- **Goal 11** is arguably one of the most important functionalities of the system, as it is related to the ability of passengers to reserve taxi rides. As such, it involves many different components:
  - The **Account Management** component, which is responsible for providing the required authentication capability using its Login sub-component and for maintaining the association between a passenger and its requests (req. 11.1).
  - The **Taxi Management System** component, which implements all the backend business logic to handle taxi reservations. In particular:
    * Reqs. 11.3 is handled by the Reservation Management and the Location Management sub-components.
    * Reqs. 11.6 to 11.15 are handled by the Reservation Management sub-component.
    * Req. 11.10 is handled by the Location Management sub-component, which is responsible for computing the taxi zone associated with a certain location.
    * Req. 11.10 is also handled by the Request Management sub-component, which is used to associate a reservation to a taxi request.

    - Req. 11.10, 11.14 and 11.15 are handled by the Taxi Management sub-component.
  - The **Passenger Application** component, which provides a suitable UI to fulfill reqs. 11.1, 11.2.1, 11.4 to 11.7, 11.11 to 11.13 and 11.15.
  - The **Passenger Web Application** component, which provides a suitable UI to fulfill reqs. 11.1, 11.2.2, 11.6, 11.7, 11.11 to 11.13 and 11.15.
  - The **Taxi Driver Application** component, which provides a suitable UI to notify a taxi driver that he has been assigned a taxi request, as in goal 9.
  - The **Remote Services Interface** component, which enables the passenger applications (web and mobile) to request services to the central system and thus provides the necessary callbacks to satisfy req 11.2.
  - The **Notification System** component, which provides to the central system the dispatch mechanism it needs for sending notifications to both the taxi driver application and the passenger applications and fulfill reqs 11.9, 11.14 and 11.15.

- **Goal 12** is related to the generation and validation of the security code associated to each ride. This involves:
  - The **Taxi Management System** component, in particular the Taxi Management and Request Management sub-components which are responsible for the generation of the security number (req. 12.1).
  - The **Notification System** component, which is used to send the security number to both the taxi driver application and the passenger application (either the web app or the mobile app, depending on the case) in order to satisfy reqs. 12.2 and 12.3.
  - The **Taxi Driver Application**, Passenger Application and Passenger Web Application components, which implement a suitable UI to show the received security number.

5.2 Quality Requirements

In this section we will describe in detail how myTaxiService has been designed to guarantee certain important Quality of Service (QoS) attributes.

In particular, we will focus on Performance, Reliability, Availability, Security and Portability requirements.

The usage of a Platform as a Service (PaaS) cloud infrastructure, combined with the n-tier and multi-layered software architecture we have chosen

to adopt, provides myTaxiService a great level of performance. Specifically, the division of myTaxiService in a plethora of components implementing functionalities at different levels of abstraction and running on separate machines enables us to scale the hardware infrastructure as needed to achieve exactly the level of performance required. Concurrency requirements, for example, are satisfied by accurate load-balancing of requests across multiple servers running different instances of the same core services, in order to achieve a high level of parallelism and be able to serve as many requests as needed, with the option to increase the capabilities of the system in the future by renting more computation capabilities.

The same n-tier infrastructure also let us achieve satisfying levels of availability and reliability, as the system components are distributed across several servers, possibly housed in different server farms by the cloud provider. Because of the way the PaaS cloud is designed, in case of failures requests can be smoothly transitioned from a server to another one without any interruption in the system operation; also, data is backed up across different locations, to provide a great level of resilience to natural disasters.

Security is achieved through both system-level and hardware-level measures. At the system level, the Account Manager and the System Administration components are responsible for checking that methods can only be invoked by authorized personnel. Furthermore, APIs are exposed via the Remote Services Interface in a way that prevents SQL injection attacks to take place and are invoked via secure token and encryption mechanisms to prevent man-in-the-middle attacks. DDoS attacks are mitigated by the presence of multiple load-balancers to redistribute requests across all the servers and thus minimize the possibility of a server going down due to the excessive amount of requests; a hardware firewall is also positioned at the boundaries of the central system to limit the number of ports that are publicly exposed and to temporarily ban IP addresses that are performing an excessive amount of requests in an attempt to block DDoS attacks. Other firewalls are positioned to separate the Communication Services Server from the servers providing the core functionalities of the system (Taxi Management, Account Management, System Administration and Database) and to separate the Database Server from everything else. All passwords are encrypted at the software level before being stored in the DBMS; to achieve better security, disks are physically encrypted at the hardware-level.

Finally, portability is achieved by exposing APIs using an open, standard web service mechanism based on SOAP. This allows the system to be invoked from any platform which can access the internet without restricting it to any particular operating system, hardware vendor or programming language.

Appendix A

Hours of work

To redact this document, we spent 50 hours per person.