mytaxy

Document Design

Matteo Maria Fusi, Matteo Locatelli

0.1 Document version

- 1.0 - 4/12/2015: first release

0.2 Time Spent

- Matteo M. Fusi: ~25h
- Matteo Locatelli: ~25h

Contents

1 Document Overview

1.1 Document Purpose

The Design Document (DD) contains a functional description of the mytaxi system. This document explains every component that is inserted into the system, the architectural styles used and the design patterns that are implemented to guarantee the satisfaction of all the requirements The components will be described both at high level and more specifically, illustrating and explaining all the sub-components every component is made of. In this way, the connection between components is illustrated; every person that will read this document will have a clear idea about its architecture both hardware and software, whether he wants to have a detailed description of the system or a more general one.

1.2 Definitions, Acronyms, Abbreviations

For the definitions, acronyms and abbreviations look to the "Glossary" section of the RASD document.

1.3 Reference Documents

- RASD document previously delivered.

1.4 Document Structure

Section 1 describes the main purpose and the structure of this Design Document. Section 2 contains the description of the components used in the system, firstly a high level view is illustrated, then every component is explored and each sub-component is described, highlighting the connections between components and sub-components. The second section includes also the deployment view, that describes the physical deployment of the system i.e. which hardware the components are inserted into, the runtime view that illustrates the behaviour of the system when it's running and, finally, the architectural styles and the design patterns that are the most suitable to be used for the system. Section 3 contains the main algorithm implemented in the core of the system, that manages the taxi queue and the sending of messages to customers and taxi drivers. Section 4 shows how every functional requirement described in the RASD document is covered by the components of the system. Lastly, section 5 contains all the references to external documents needed to fully understand this paper.

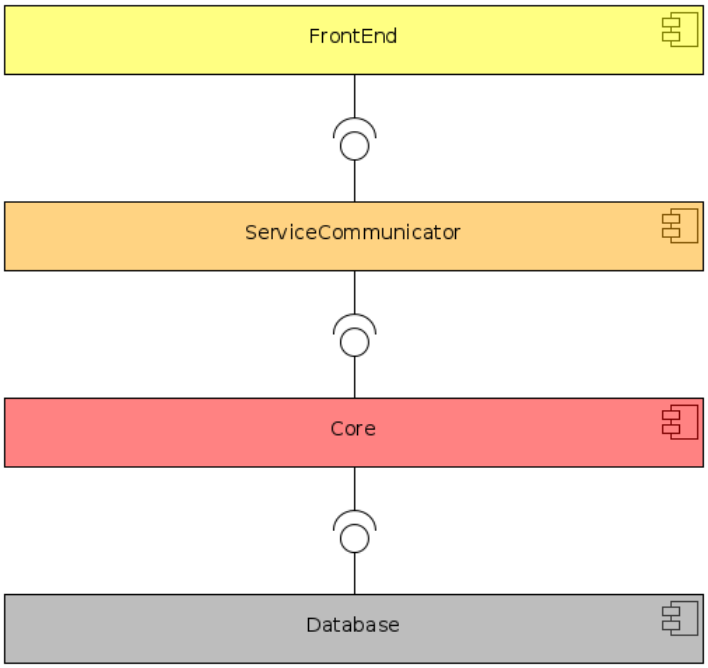
2 Architectural Design

2.1 Overview

The architecture we based our system on is the client-server architecture with two tiers. One tier (the server tier) is composed by the server logic and the

database, the other tier (the client tier) is composed by the client logic and the GUI that allows the user to interact with the system. The fact that both the server and the client tiers have an application logic means that the application is distributed; this, added to the fact that we decided to use a service oriented architure, allows our system to be scalable, so other services can be added without modifying the structure of the system. For example, if a future implementation of the system needs a different type of client, the new client can use the same services provided by the server, without modifying the implementation of its logic. Furthermore, using a two tier architecture allows our system to be easy to maintain and repair if one of its components breaks.

2.2 High level components and their interaction



The main high level components of the system are:

1. Frontend
2. Service Communicator
3. Core
4. Database.

Frontend is used to allow customers and taxi drivers to interact with the system, so every customer can make requests and reservations and every taxi driver can

accept or refuse a ride, appear available or not to the system etc. Core has the task to manage taxi queues and manage the comunication between customers and taxi drivers, by means of messages. Service Communicator has to ensure the comunication between the front end and the core, providing services to both the components. Database trivially must store data that system generates and exploits.

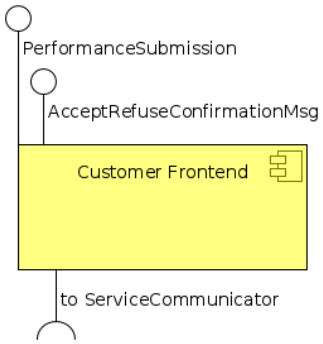
2.3 Component view

2.3.1 Frontend

The frontend must take the input of users and present the several responses that the system generates for the user. Two different kinds of frontends are proposed: one for customers and one for taxi drivers.

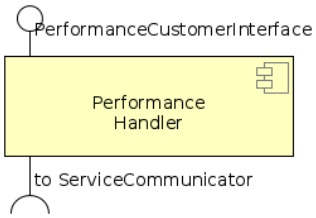
Customer Frontend

Name	Customer Frontend
Required interfaces	1. Interface to ServiceCommunicator: needed for dialogate with the core of the system
Provided interfaces	1. ***AcceptRefuseConfirmation***: handles offers of performances when there are no available taxis in the zone. 2. ***PerformanceSubmission***: handles request of generic performances made by customers (for example request or reservations)
Description	This component must interface the customer with the system. A customer can submit generic requests using the *PerformanceSubmission* interface. The *AcceptRefuseConfirmation* is used from customer for accepting or reject the offers of generic performance proposed by the system if there aren't available taxis in the zone where the customer is.

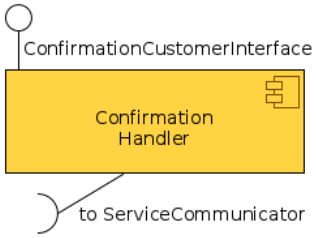


Customer Frontend contains the two submodules described below.

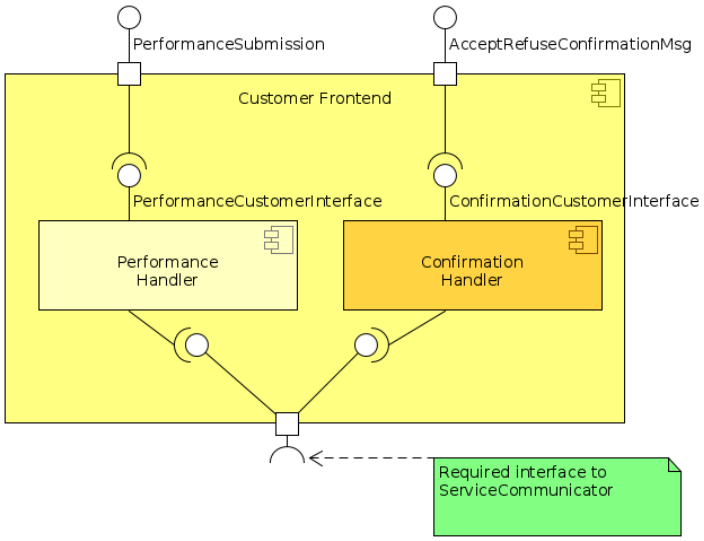
Name	Performance Handler
Required interfaces	1. Interface to ServiceCommunicator
Provided interfaces	1. ***PerformanceCustomerInterface***: Interface which purpose is dialogate with the user and informing him about the performance that he submits to the sytem.
Description	The duty of performanceHandler is to present generic requests made by the customer using the ServiceCommunicator.



Name	Confirmation Handler
Required interfaces	1. Interface to ServiceCommunicator to inform the system about the confirmation or rejection by the customer.
Provided interfaces	1. ***ConfirmationCustomerInterface***: Interface that presents data to customer and takes his input of accepting or rejecting an offer out of the zone.
Description	He links the system with the customer using the ServiceCommunicator in terms of offers out of the zone proposed by the system.

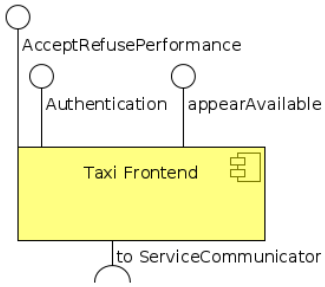


Using the 2 components listed above the Customer Frontend is structured as indicated in the diagram below.



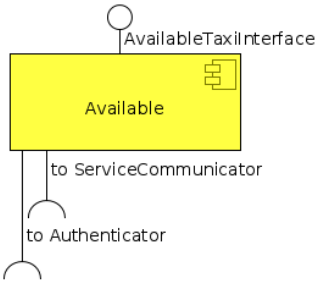
Taxi Frontend

Name	Taxi Frontend
Required interfaces	1. Interface to ServiceCommunicator for dialogates with the system.
Provided interfaces	1. ***AcceptRefusePerformance:*** interface provided for accept or reject ride offers proposed by the system. 2. ***Authentication***: needed for authenticate taxi drivers. 3. ***AppearAvailable***: drivers can use this interface for inform the system if he's available or not.
Description	This component should present all the functionalities of the system that taxi driver can exploit to them. See RASD document for deeper details.

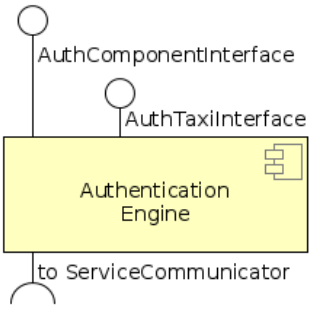


Taxi Frontend is composed by the following 3 modules.

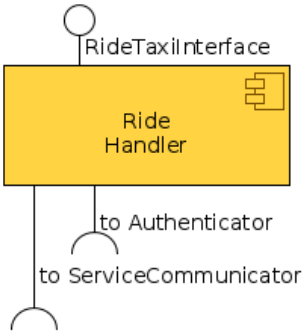
Name	Available
Required interfaces	1. Interface to ServiceCommunicator to inform the system about the availability of the taxi driver. 2. Interface to Authenticator for verifying identity of taxi driver.
Provided interfaces	1. ***AvailableTaxiInterface:*** interface used by taxi driver for setting their status to available or not.
Description	It manage the availability of a taxi. Using this component taxi drivers can set their status to available or unavailable.



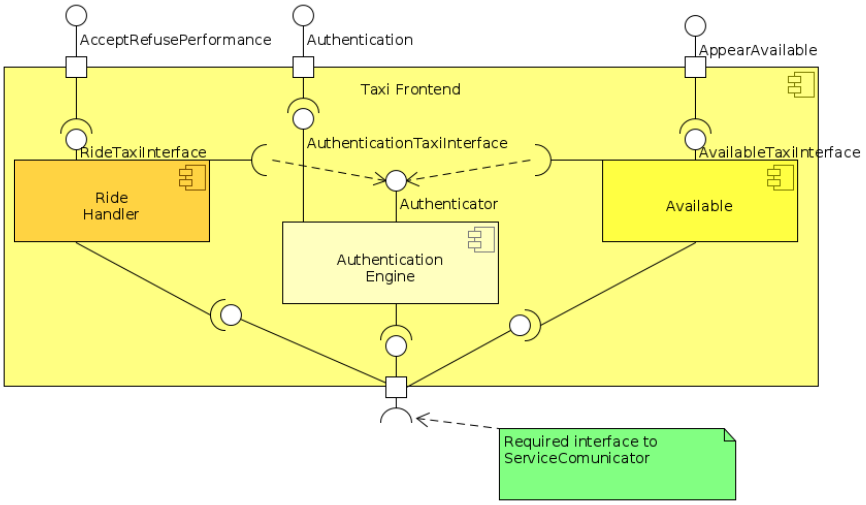
Name	Authentication Engine
Required interfaces	1. Interface to ServiceCommunicator to validate authentication.
Provided interfaces	1. ***AuthComponentInterface:*** interface used for other components for verifying taxi driver if needed. 2. ***AuthTaxiInterface***: Used by taxi driver for logging into the system.
Description	This component has the duty of authenticate taxi driver in the context of the system. It also provides an interface to frontend modules for guaranteeing informatics security.



Name	Ride Handler
Required interfaces	1. Interface to ServiceCommunicator to inform the system about the driver response about a generic request. 2. Interface to Authenticator for verifying identity of the taxi driver.
Provided interfaces	1. ***RideTaxiInterface:*** used by driver for accept or reject ride offers proposed by the system.
Description	Using this component drivers can accept or refuse generic ride offers proposed by the system.



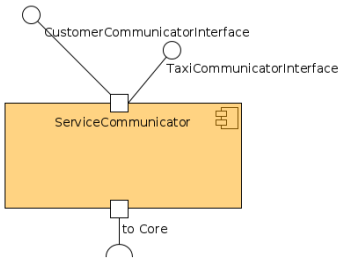
Assembling the subcomponents described we can observe the structure of Taxi Frontend.



2.3.2 ServiceCommunicator

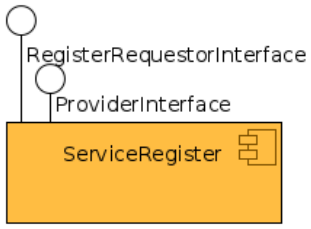
The ServiceCommunicator component has the task to link the Frontend to the Core component, by means of services that the service provider makes available to the service requestor. The service register contains all the services that are needed to the system to run properly.

Name	Service Communicator
Required interfaces	1. Interface to Core
Provided interfaces	1. ***CustomerCommunicatorInterface:*** interface used by customers for exploiting Service Communicator functionalities. 2. ***TaxiCommunicatorInterface:*** interface used by taxi drivers for exploiting Service Communicator functionalities.
Description	Service Communicator has the duty of connecting Frontend and core using SOA.

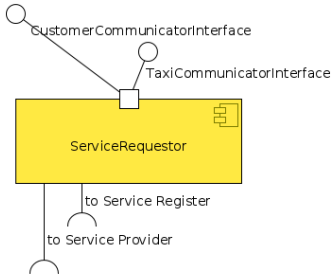


As said in the previous description, a Service Oriented Architecture has been used for develop Service Communicator module. Now the composition and the sub-components of the inquiring module will be discussed.

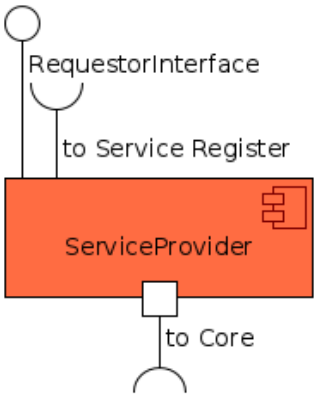
Name	Service Register
Required interfaces	none
Provided interfaces	1. ***RegisterRequestorInterface:*** interface used by the service requestor to know the services it can access and use. 2. ***ProviderInterface:*** interface used by the service provider to know the service that can be accessed by the service requestor.
Description	This component contains all the services available that can be accessed by the requestor, if made available by the provider.



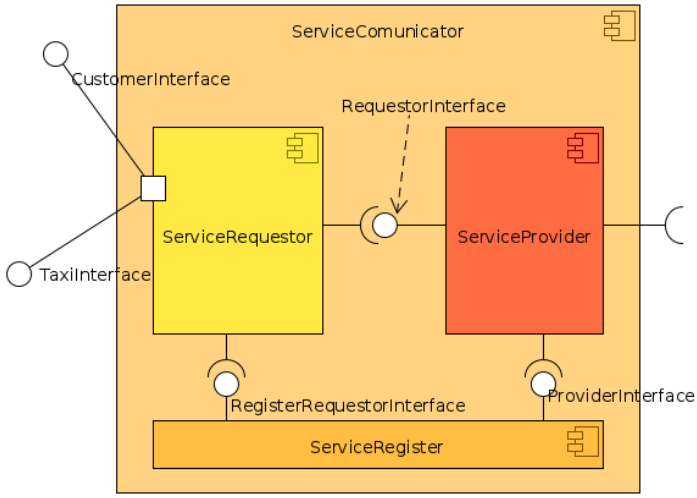
Name	Service Requestor
Required interfaces	1. Interface to Service Register to know which services are available to the requestor. 2. Interface to Service Provider to access the available services.
Provided interfaces	1. ***CustomerCommunicatorInterface:*** interface used by the Customer Frontend to access the services available to the customer. 2. ***TaxiCommunicatorInterface:*** interface used by the Taxi Frontend to access the services available to the taxi driver.
Description	This component is used by the Customer Frontend and the Taxi Frontend components to access all the available services they need no guarantee the frontend functionalities.



Name	Service Provider
Required interfaces	1. Interface to Core that is used to communicate to the core when a customer asks for a performance and to know when to send a message to a customer or to a taxi driver. 2. Interface to Service Register to access the services stored in the service register.
Provided interfaces	1. ***RequestorInterface:*** interface used by the service requestor to access the services available to it.
Description	This component allow the service requestor to access all the services it is able to access.

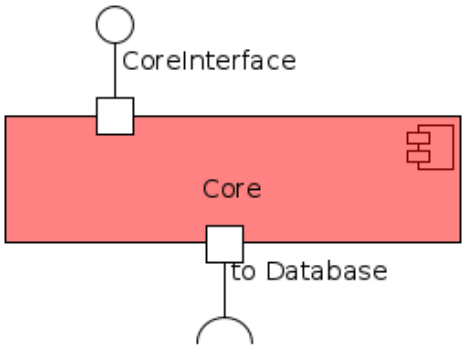


Assembling the listed components we obtain the following diagram that represents the composite structure of the Service Communicator.

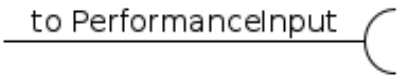


2.3.3 Core

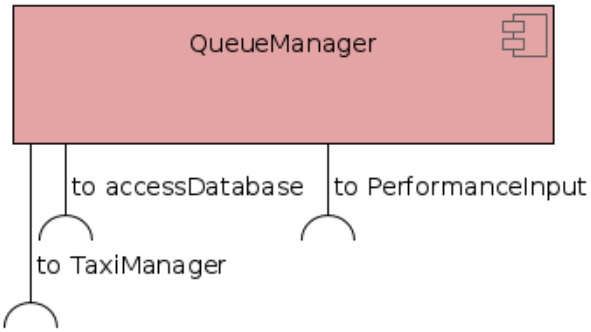
Name	Core
Required interfaces	1. Interface to Database, that make possible to access every information in the database such as the zones of the city and the taxis.
Provided interfaces	1. ***CoreInterface:*** interface used to communicate with the ServiceProvider and to manage services according to the core functions.
Description	This component has the task to execute all the main functionalities of the system, such as the queue management, sending message to customers and taxi drivers, etc.



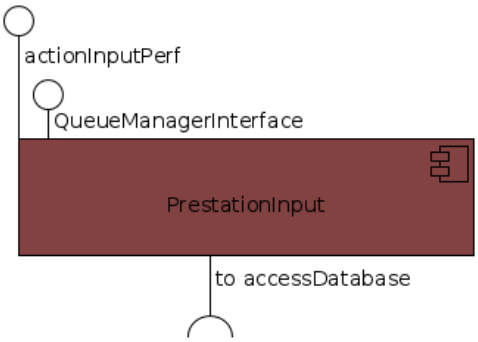
Name	CoreFacade
Required interfaces	1. Interface to PerformanceInput to make possible to external components to input a performance. 2. Interface to TaxiManager to make possible to modify the current information about a taxi.
Provided interfaces	1. ***CoreFacadeInterface:*** interface used to make external components access the facade of the core, allowing a simpler access to the many functionalities of the system.
Description	This component is used to guarantee an easy access to the core functionalities.



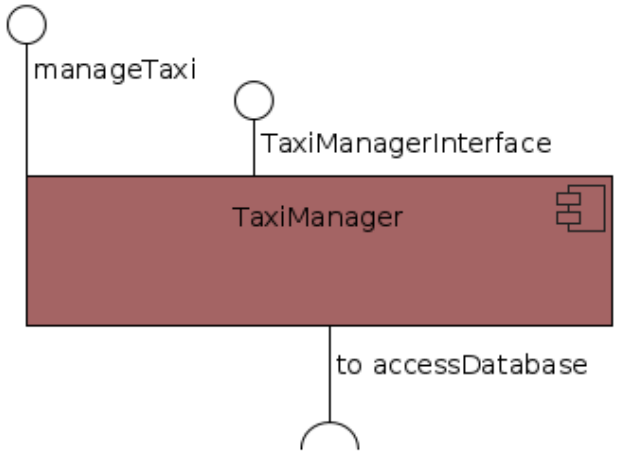
Name	QueueManager
Required interfaces	1. Interface to PerformanceInput that is needed by this component to know when a performance is sent by a customer. 2. Interface to TaxiManager to retrieve information about taxis, such as their availability. 3. Interface to accessDatabase to retrieve information useful to manage the queues.
Provided interfaces	None.
Description	This component manages the taxi queue of any zone of the city and communicates to other components when to send a message to a customer.



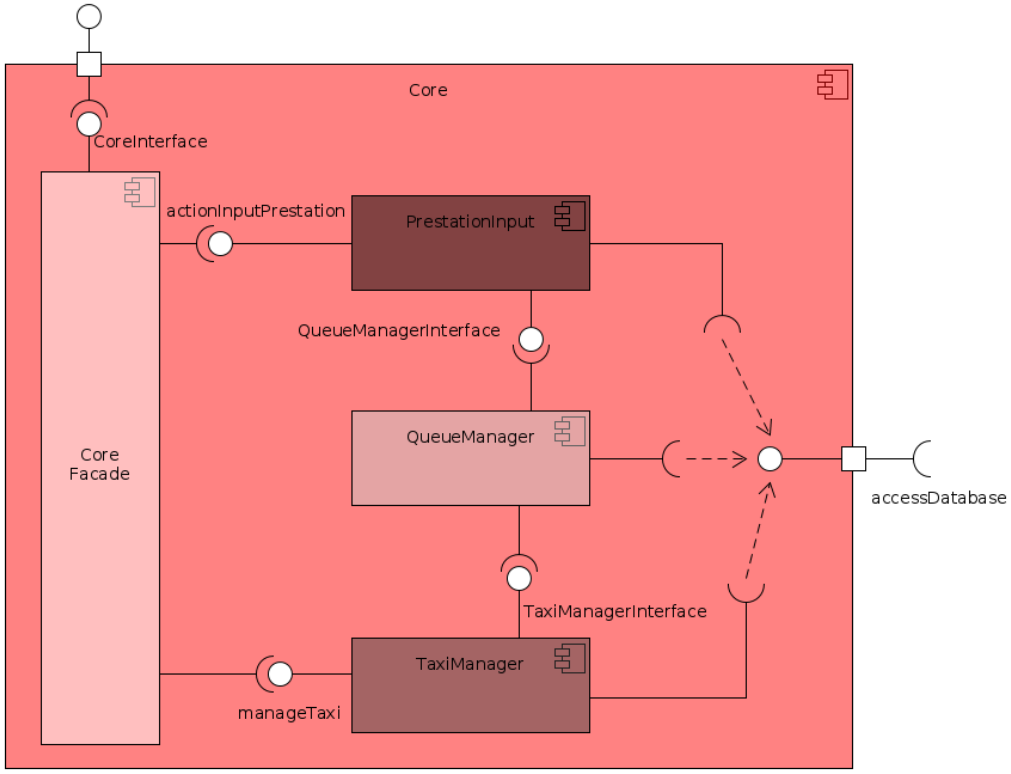
Name	PerformanceInput
Required interfaces	1. Interface to accessDatabase, needed to retrieve information about taxis and performances.
Provided interfaces	1. ***actionInputPerf:*** used to take in input any performance sent by a customer. 2. ***QueueManagerInterface:*** used to communicate to the QueueManager component when a performance is sent by a customer.
Description	This component manages the input of any performance sent by a customer and the sending of messages to both customers and taxi drivers.



Name	TaxiManager
Required interfaces	1. Interface to accessDatabase, needed to retrieve information about taxis.
Provided interfaces	1. ***TaxiManagerInterface:*** used to communicate to the QueueManager the status of a taxi. 2. ***manageTaxi:*** used to take in input an update of the status of a taxi or the credentials used to log into the system.
Description	This component manages the status of every taxi and allow taxi drivers to log into the system.

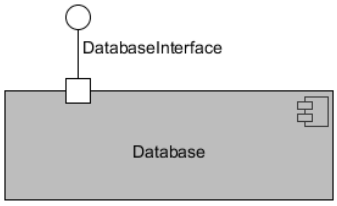


Assembling all the core components previously described the complete Core component is obtained.



2.3.4 Database

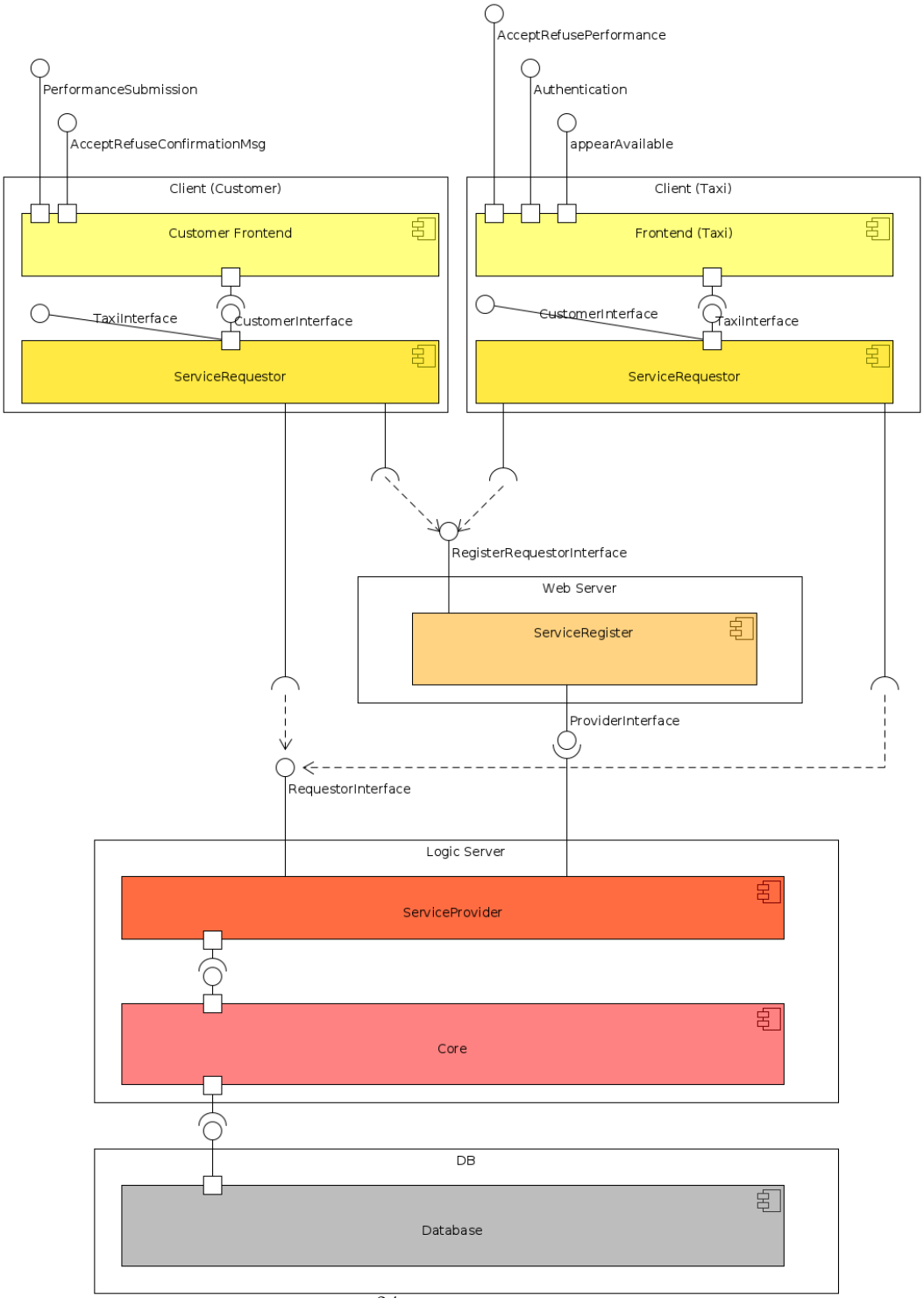
Name	Database
Required interfaces	None.
Provided interfaces	1. ***DatabaseInterface:*** this interface allow the core and the database to communicate, so the core can access all the required information to guarantee its functionalities.
Description	This component contains all the data needed by the core, such as the zone of the city and all the taxis available.



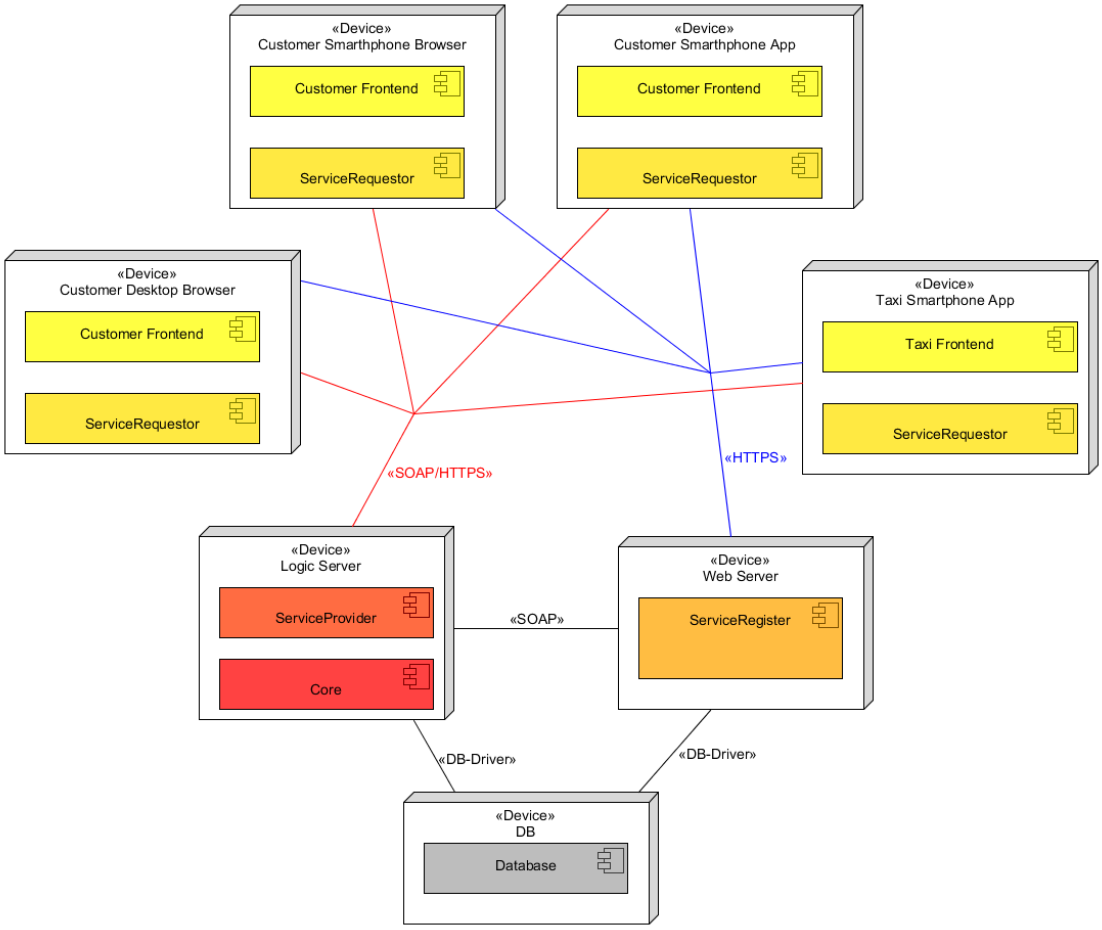
2.4 Deployment view

In the first proposed diagram is proposed to the reader the interactions between several components distributed of physical devices. We can easily observe that the component Service Communicator is distributed in three different devices:

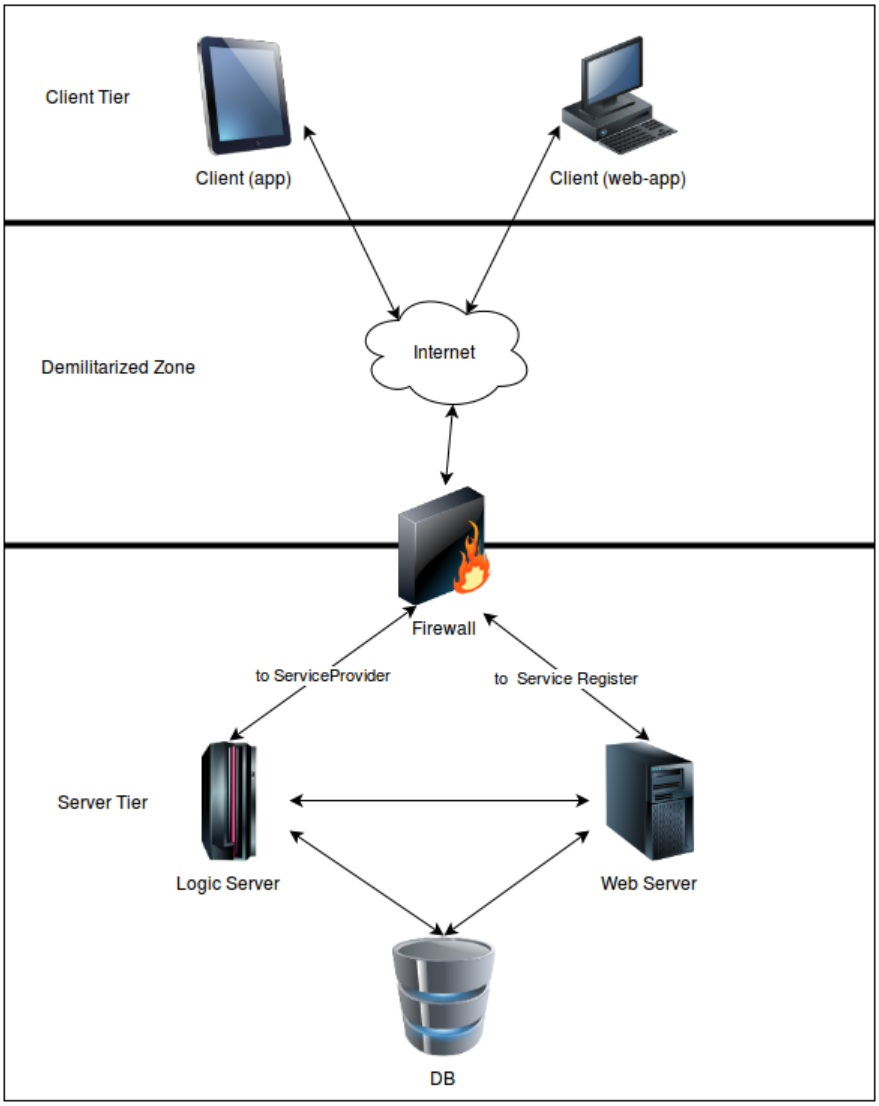
1. Client
2. Web server
3. Logic server.


AcceptRefusePerformance
PerformanceSubmission
Authentication
AcceptRefuseConfirmationMsg
appearAvailable
Client (Customer)
Client (Taxi)
Customer Frontend
Frontend (Taxi)
TaxiInterface
CustomerInterface
CustomerInterface
TaxiInterface
ServiceRequestor
ServiceRequestor
RegisterRequestorInterface
Web Server
ServiceRegister
ProviderInterface
RequestorInterface
Logic Server
ServiceProvider
Core
DB
Database

Considering that a client-server architecture has been chosen to implement the system, there are four elements in which the components have to be installed. In the second view the linkage between several devices has been emphasized. DB-driver will depend on language and DB implementation.



A "*full-hardware*" view is now proposed to suggest to the reader a complete overview of the deployment structure. We can easily note a 2-tier client server architecture.



2.4.1 Customer Client

This client contains the Customer Frontend component, so a user is able to submit a performance (request or reservation) and to decide to accept or refuse a refusable message (this message is sent to the customer when the first taxi driver available is found outside the customer's zone). The customer's client is used on customer's smartphone or pcs, depending on his choice to use the app or webapp version of the service.

2.4.2 Taxi Client

This client contains the Taxi Frontend component and it's used by the taxi driver to access every available functionality that he can do, such as accepting or refusing a request and appear available to the system. This client will be installed on the taxi driver's smartphone and it's important to underline that this version of the client will be only available on taxi drivers' smartphone, in this way customers cannot accept the services reserved to drivers.

2.4.3 Logic Server

The server contains two components: Core (responsible for the execution of the main functionalities of the system) and Service Provider (that allows clients to access the services they need). Service Provider communicates with the service register to access the services that the clients are enabled to use.

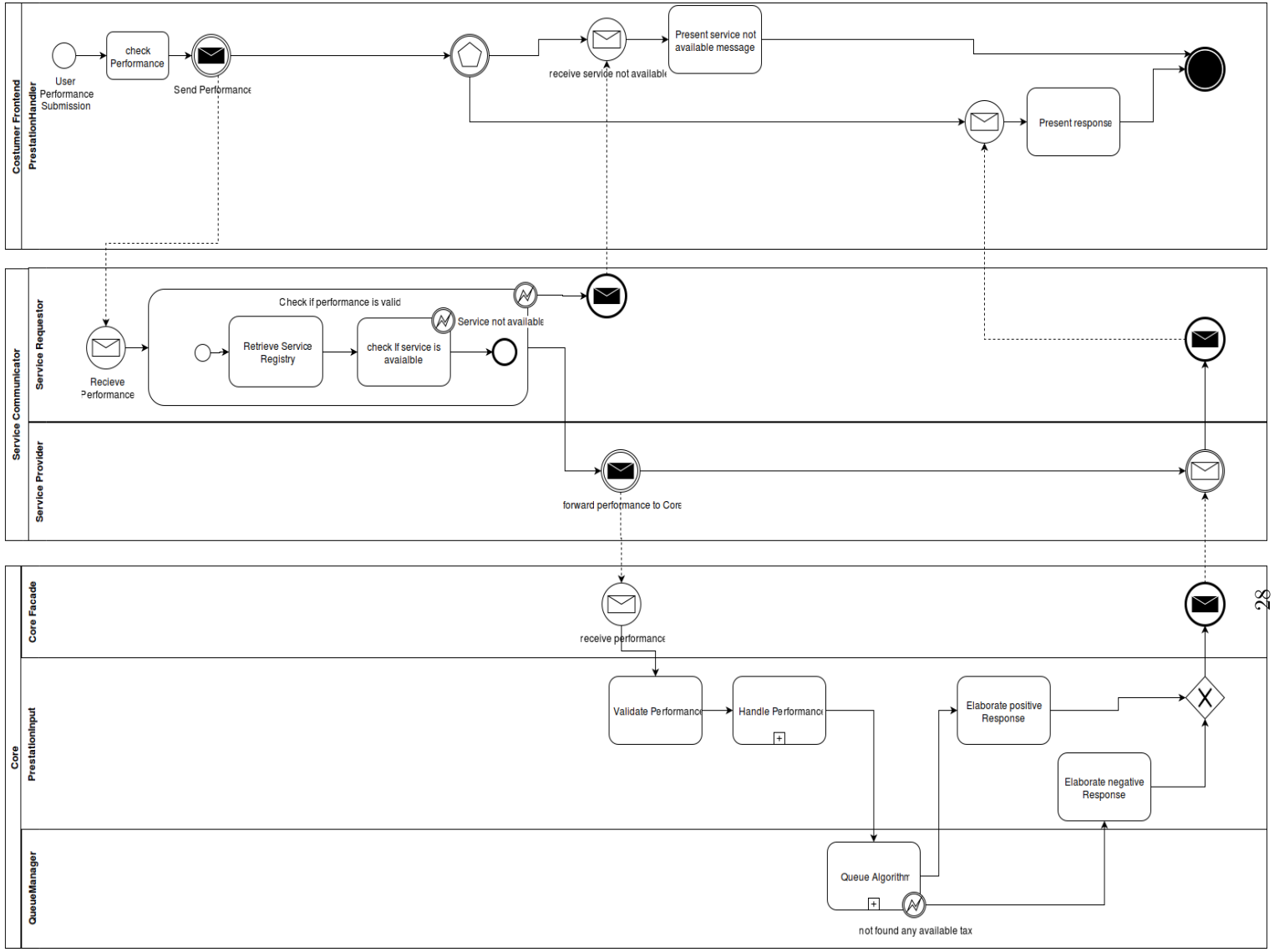
2.4.4 Web Server

The service register is located on a web page, so both the clients and the server can access it at any time and are ifnormed about the services used.

2.5 Runtime view

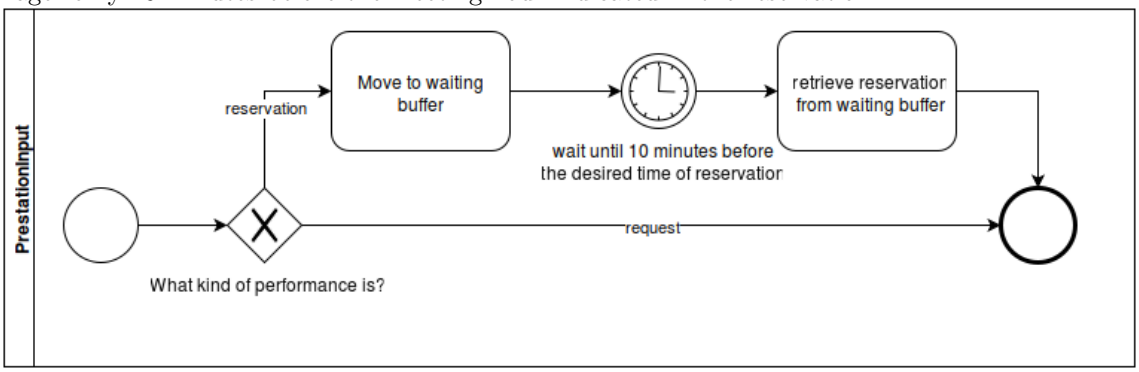
2.5.1 Request for a performance

When a customer submits a performance to the system, Customer Frontend does a raw check of the validity of attributes of the submitted element. After that the performance is sent to the Service Requestor component attached to the inquiring Customer Frontend. The last mentioned component has the duty of checking that the requested service is available (the type of service requested depends by the type of performance submitted). If the service is unavailable then the customer is notified with an error and the process terminates else the correct flow of the process continues, which means that the performance request is forwarded to the Core through Service Provider. The performance request is received by Core Facade, then PerformanceInput checks again the performance request and handles it (see the subsection below for further details), then the performance is forwarded to Queue Manager component that search for a taxi that can takes in charge of the performance. In the end the Core elaborates a response that sends to Customer. The response can be positive (a taxi has been found and so the response contains details about the chosen taxi and ETA) or negative (a simple message that the service is not available).


Costumer Frontend
PrestationHandler
User Performance Submission
check Performance
Send Performance
receive service not available
Present service not available message
Present response
Service Communicator
Service Requestor
Recieve Performance
Check if performance is valid
Retrieve Service Registry
check If service is avaialble
Service not available
Service Provider
forward performance to Core
Core
Core Facade
receive performance
PrestationInput
Validate Performance
Handle Performance
Elaborate positive Response
Elaborate negative Response
QueueManager
Queue Algorithm
not found any available tax

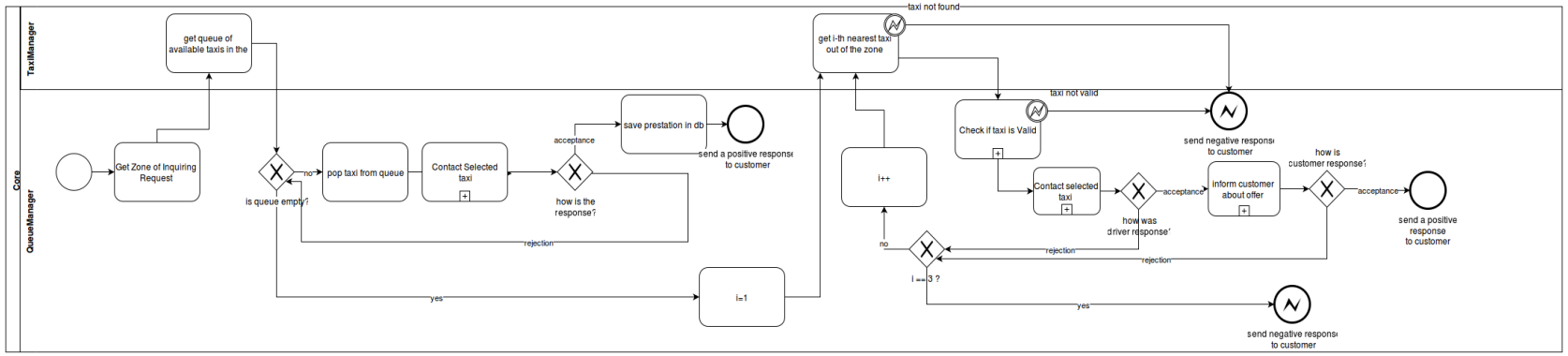
2.5.2 Core Performance Handling

If the type of the performance is a request is immediatly forwarded to Queue Manager. If the type of the performance is a reservation, this one is moved in a waiting buffer. The reservation will be resumed and forwarded to Queue Manager only 10 minutes before the meeting hour indicated in the reservation.



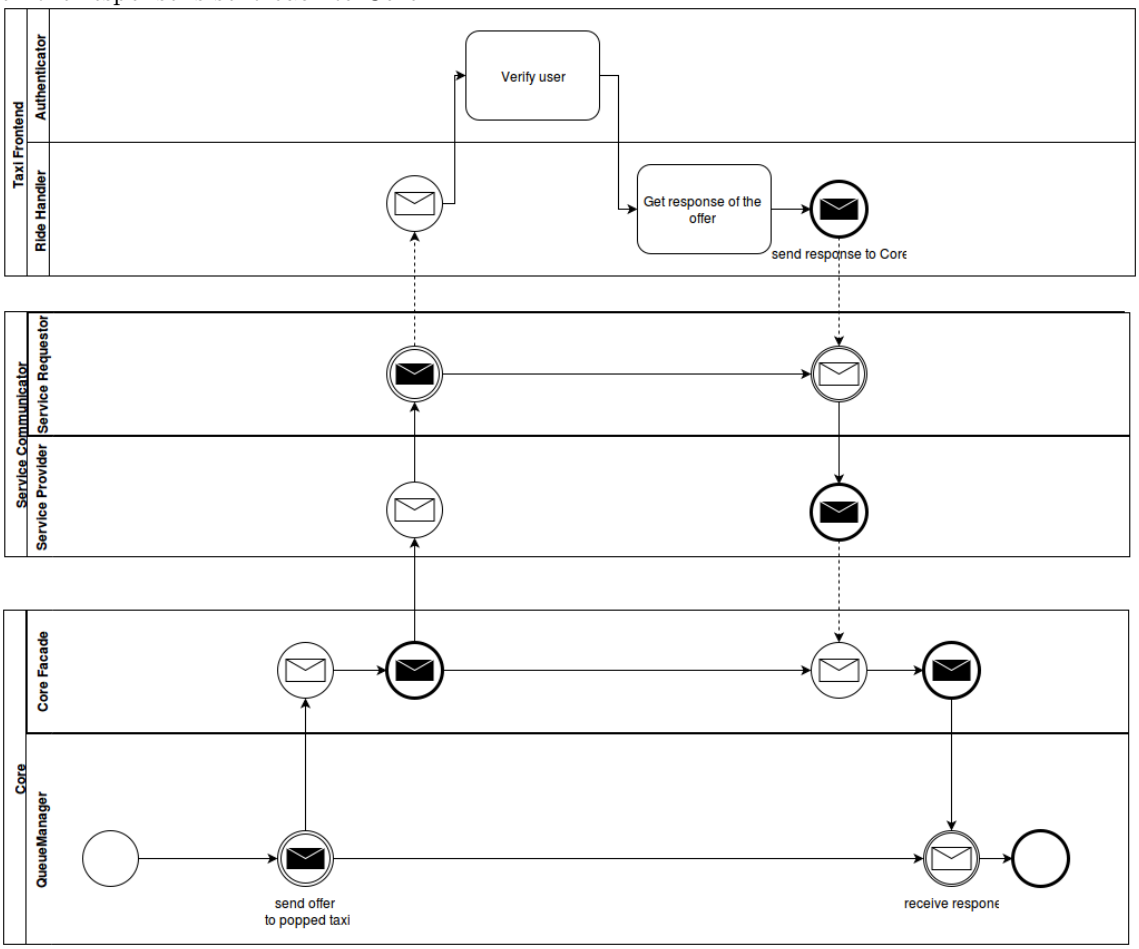
2.5.3 Queue Algorithm

The processing of a performance in the Queue Manager starts with retrieving the zone where the ride indicated by the performance will start. In this way is possible to get the queue of taxis available in the zone. The Queue Manager will start to sequentally contact all the taxis in the queue asking them if they want to accept the ride until the queue is empty or when a taxi driver accepts the performance. If the queue is empty and a taxi has not been found yet then Queue Manager begins to contact taxis out of the specified zone. Queue Manager can contact at most 3 taxis out of the zone and it will contact sequentally the first, the second and then the third nearest taxi out of the zone, but if and only if they're far at most 10 minutes from the meeting point because they must arrive in time. For every taxi is sent a message if they want accept the performance just like done with the taxis in the queue. If a taxi accepts the request an offer of the ride is sent to the customer. The customer can accept or refuse the offer. If the customer accept the process terminates else Queue Manager contact the next nearest taxi. If a suitable taxi has not been found an error signal is thrown.


Core
TaxiManager
QueueManager
get queue of available taxis in the
Get Zone of Inquiring Request
is queue empty?
no
pop taxi from queue
Contact Selected taxi
how is the response?
acceptance
save prestation in db
send a positive response to customer
rejection
yes
i=1
i++
get i-th nearest taxi out of the zone
taxi not found
Check if taxi is Valid
taxi not valid
send negative response to customer
Contact selected taxi
how was driver response?
acceptance
inform customer about offer
how is customer response?
acceptance
send a positive response to customer
rejection
rejection
no
i == 3 ?
yes
send negative response to customer

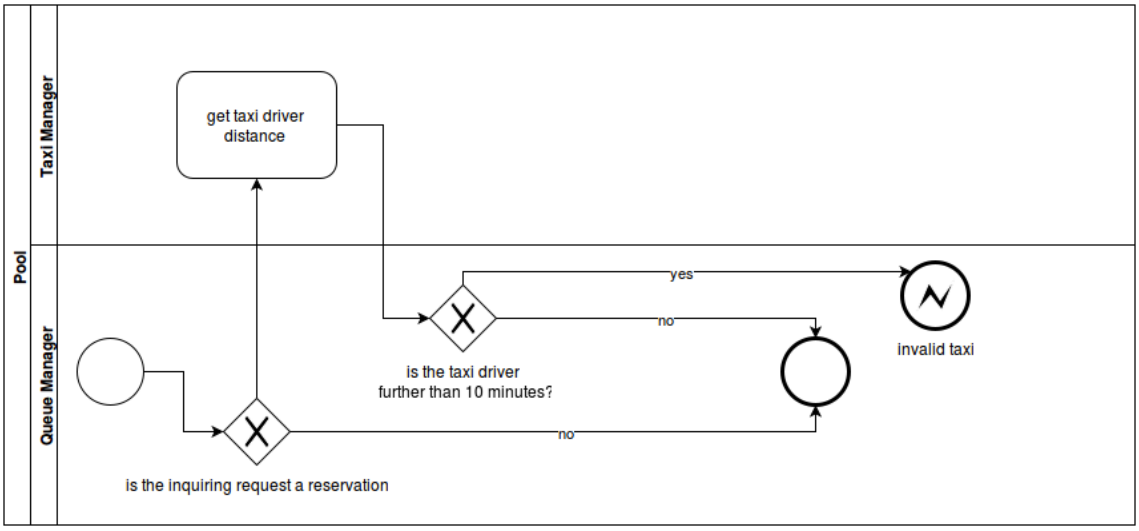
2.5.4 Contact selected taxi

This process just send an offer to a specific taxi driver through Service Communicator. When the offer is received by Taxi Frontend the system verifies that is the correct taxi and wait for the response of the taxi driver. When the input is taken the response is sent back to Core.



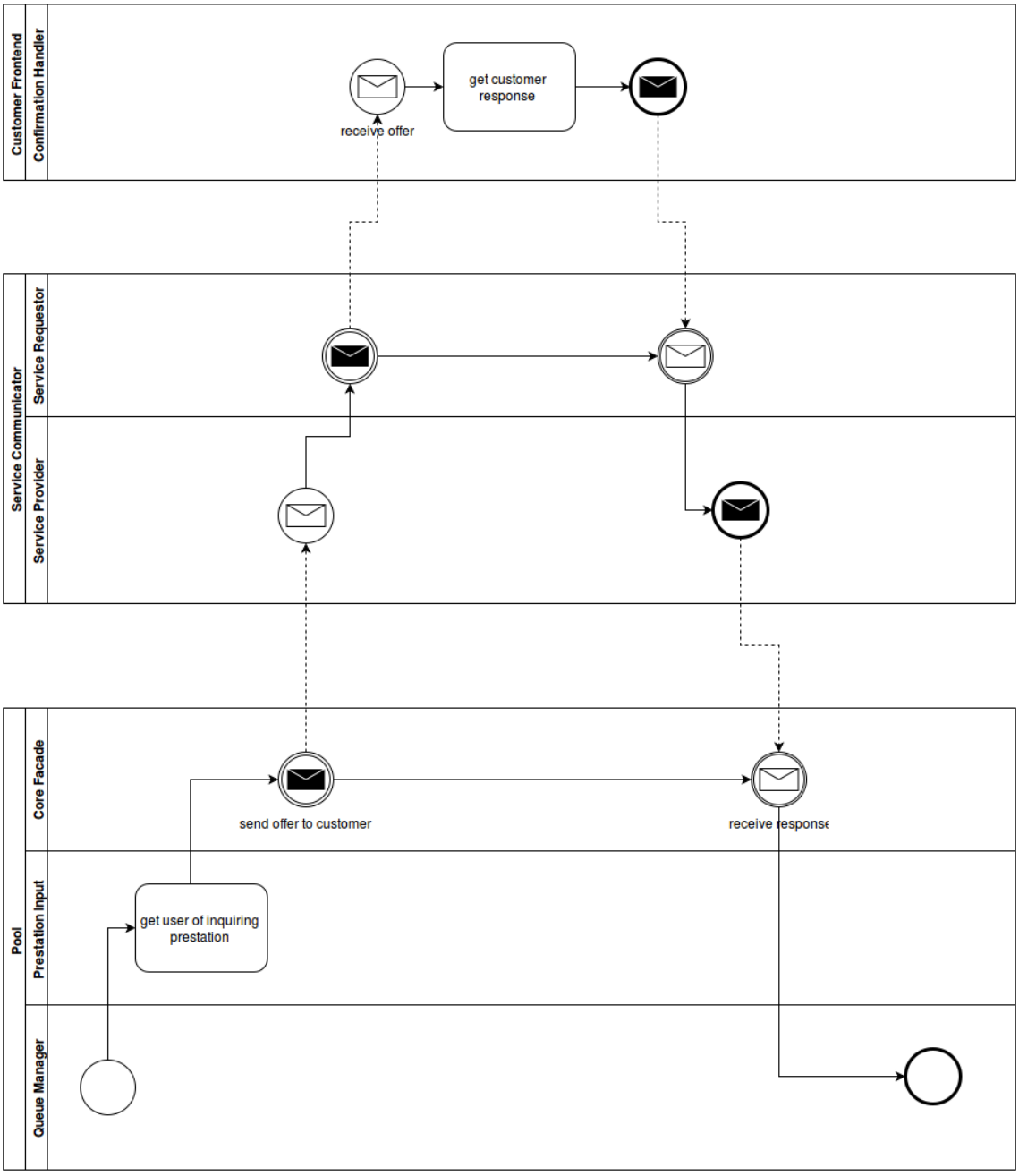
2.5.5 Check if taxi is valid for a performance

This process just calculate the distance of a taxi driver from a specified meeting point. If the taxi is further than 10 minutes an error signal is thrown.



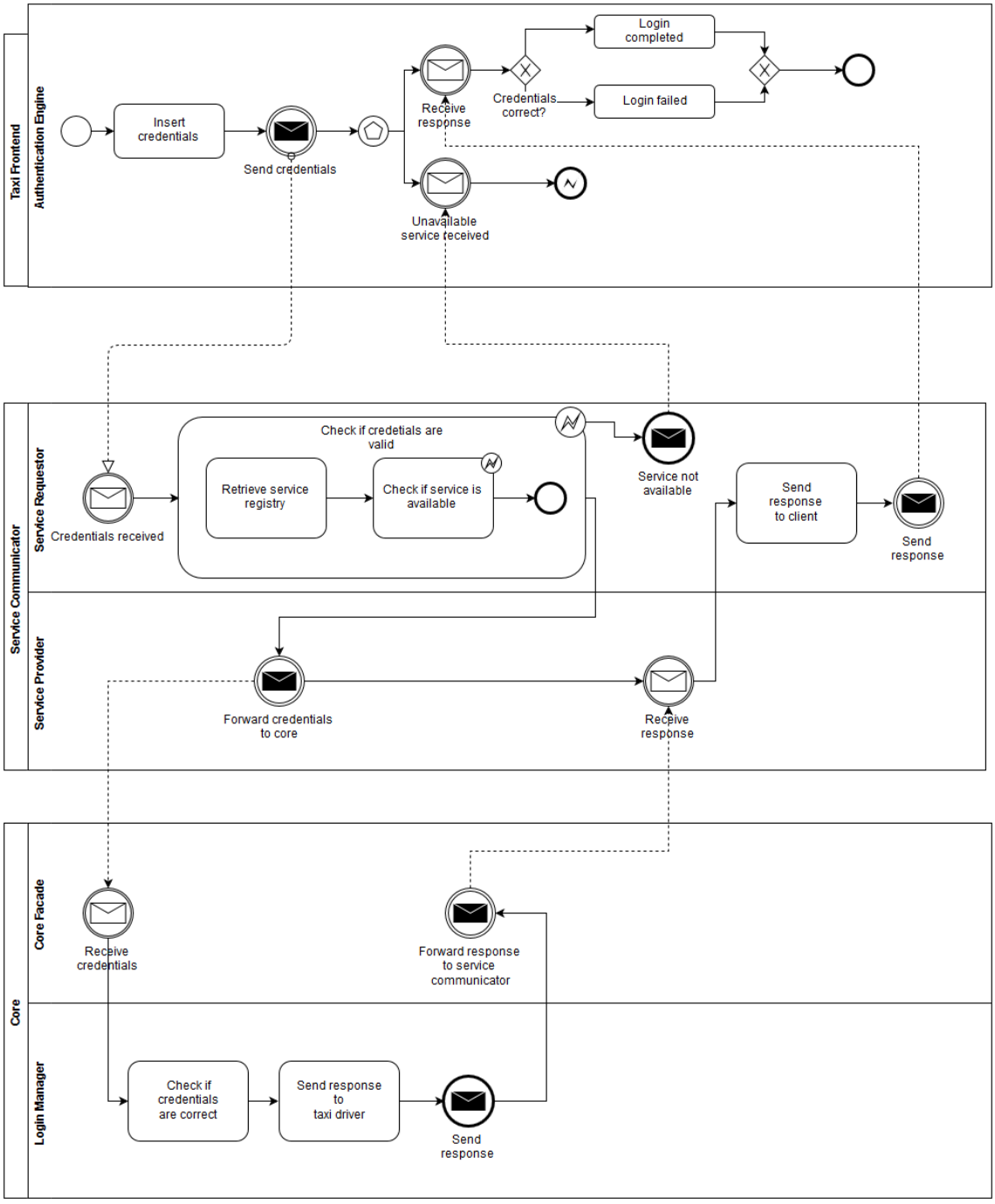
2.5.6 Send Taxi out of the zone offer

When a taxi out of the zone accepts a performance an offer message is sent to customer. The customer can accept or refuse the offer using Confirmation Handler component.


Customer Frontend
Confirmation Handler
receive offer
get customer response
Service Communicator
Service Requestor
Service Provider
Pool
Core Facade
send offer to customer
receive response
Prestation Input
get user of inquiring prestation
Queue Manager

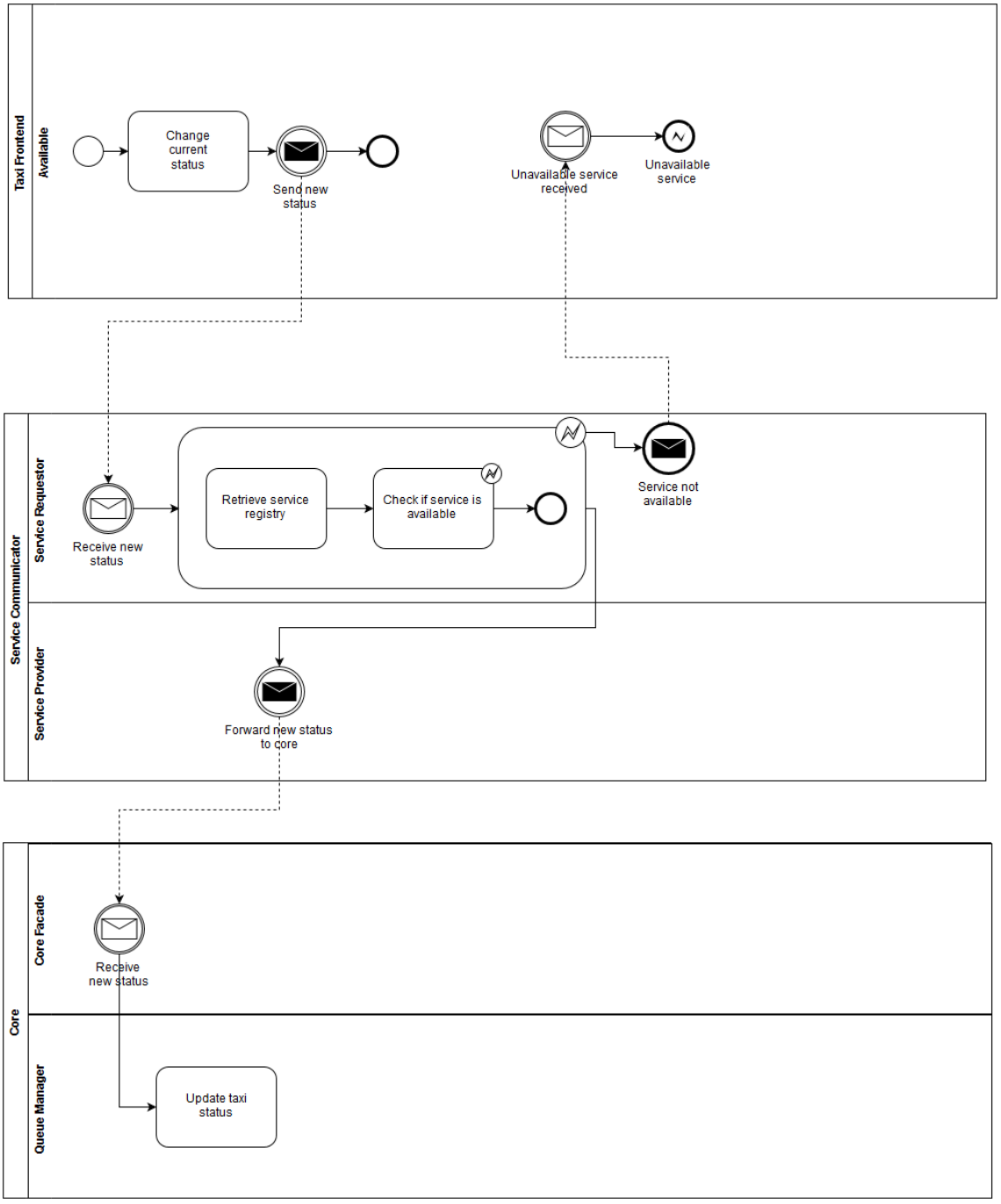
2.5.7 Taxi driver authentication

To complete his authentication, the taxi driver must insert his credentials using the Taxi Frontend. Then Authentication Engine sends the credentials to the Service Requestor, asking for the service that allows the taxi driver to log into the system. The Service Requestor retrieves the service registry and check if the service is available to the taxi driver: if the service is not available a message is sent to the Authentication Engine informing the Taxi Frontend (and thus the taxi driver) that the service is unavailable to him; if the service is instead available the execution of the log in operation continues. The credentials are sent to the Service Provider that forwards them to the Core Facade that in turn sends them to the Login Manager. The Login Manager component checks if the taxi driver's credentials are correct and then sends a message that contains the result of the login operation. The message is sent to the Service Provider via the Core Facade, which forwards it to the Service Requestor that sends the response to the Taxi Frontend. Depending on the content of the message, the taxi driver is notified if the login operation has been successful or not.


Taxi Frontend
Authentication Engine
Insert credentials
Send credentials
Receive response
Credentials correct?
Login completed
Login failed
Unavailable service received
Service Communicator
Service Requestor
Credentials received
Check if credetials are valid
Retrieve service registry
Check if service is available
Service not available
Send response to client
Send response
Service Provider
Forward credentials to core
Receive response
Core
Core Facade
Receive credentials
Forward response to service communicator
Login Manager
Check if credentials are correct
Send response to taxi driver
Send response

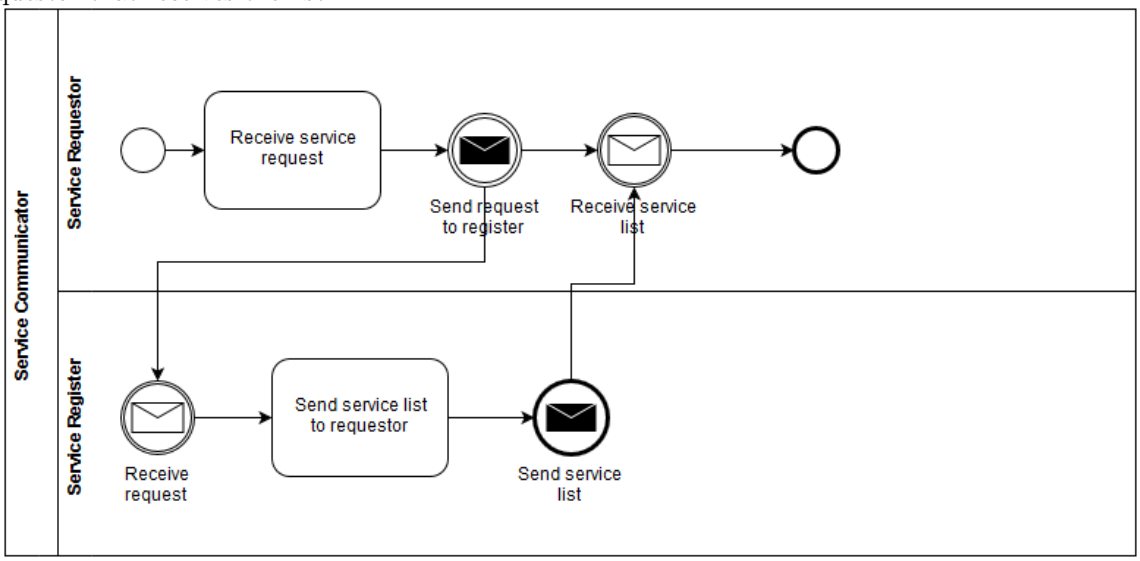
2.5.8 Taxi available operation

When a taxi driver wants to change his current status (from available to unavailable or vice versa), the Available component of the Taxi Frontend is used. It sends the new desired status to the Service Requestor, that checks if the service requested by the Taxi Frontend is available to the driver: if it isn't, an error message is sent to the Taxi Frontend saying that the service is unavailable, otherwise the operation continues. The new status is sent to the Service Provider, that forwards it to the Queue Manager component of the Core using the Core Facade. The Queue Manger then updates the status of the taxi.


Taxi Frontend
Available
Change current status
Send new status
Unavailable service received
Unavailable service
Service Communicator
Service Requestor
Receive new status
Retrieve service registry
Check if service is available
Service not available
Service Provider
Forward new status to core
Core
Core Facade
Receive new status
Queue Manager
Update taxi status

2.5.9 Service list retrievement

When a Frontend asks for a service to be accessed, a service request is sent to the Service Requestor. It forwards this request to the Service Register, which, upon the receiving of the request, sends the service list back to the Service Requestor that receives the list.



2.6 Component interfaces

2.6.1 CustomerInterface

1. `void(AbstractPerformance pre)`: sending the passed performance to Core using ServiceCommunicator.
2. `void notify(Message msg)`: notify something to Customer Frontend using the passed msg.
3. `void (DecoratedAbstractPerformance pre, CustomerResponse resp)`: notify Core about his decision on pre with resp. this method is used for accepting or refusing a performance when a taxi is contacted out of the zone of the customer. DecoratedAbstracPerformance contains infos about the inquiring performance plus infos about taxi (id of taxi, ETA).

2.6.2 TaxiInterface

1. `void sendTaxiResponse(Taxi taxi,Abstractperformance pre, TaxiResponse resp)`: notify Core about his decision on pre with resp.
2. `void login(Taxi taxi,string id, string hashPsw)`: log in the taxi driver into the system using the passed id and the hash of the password inserted by the driver.

3. `void setAvailable(Taxi taxi, bool available)`: set the status of the inquiring taxi available or not as imposed as passed available variable imposes.

4. `void notify(Message msg)`: notify something to Taxi Frontend using the passed msg.

2.6.3 RegisterRequestorInterface

1. `ServiceRegistry retrieveServiceRegistry(User user)`: retrieve a ServiceRegistry. the object returned will depend on the type of user that makes the request. A customer's service registry will be different than a taxi driver's one becuase they can access to different services.

2.6.4 ProviderInterface

1. `void updateServiceRegistry(GlobalServiceRegistry newRegistry)`: replace the current serviceRegistry instance the the new passed one.

2.6.5 CoreInterface

1. `void changeTaxiStatus(Taxi taxi, bool available)`: ask Core to change the status of the inquiring taxi as available variable imposes.

2. `void login(Taxi taxi,string id, string hashPsw)`: validate login in the Core of the passed taxi with the passed id and hashPsw.

3. `void notifyUser(User user, Message msg)`: notify a generic user with msg.

4. `void contactTaxi(Taxi taxi, AbstractPerformance pre)`: contact taxi asking him if he wants to accept pre and wait for his response.

5. `void forwardPerformanceToCore(AbstractPerformance pre)`: send pre to Core with the purpose of handling it.

2.7 Selected architectural styles and patterns

2.7.1 Architectural styles

To implement the mytaxi system the following architecural styles have been used:

1. **Client-server architecture:** we decided to use this style to implement our system, customers and taxi drivers are the clients that are able to connect to a server that can execute the main functionalities of the service, such as the taxi queue menagement. This is the simplest and at the same time the safest choice because other styles may have not worked as well as the client-server architecture. For example, a peer to peer architecture

wouldn't be right for our system, because the absence of peers in a certain moment would have compromised the functionalities of the mytaxi system. Another architecture we thought of was to install some stations in the city at which the clients could connect and then each station could guarantee the communication between clients and the server. We decided not to choose this architecture because a malfunction in a single station would interrupt the functionalities of the system for some clients and this solution could be quite expensive.

2. **Layered structure:** the mytaxi system is based on a layered structure, in this way every layer that composes the system is highly indipendent. For future extensions of the system, new layers can be added to implements new functionalities or guarantee non functional requirements, and to make the system work is enough to make the new layers communicate with the old ones. The system way of working will remain the same, but with the added layers that will execute their tasks.

3. **Service oriented architecture:** our system is based on a service oriented architecture, this means that clients use the services provided by a service provider. In this way, the two type of clients (customer and taxi driver) can access only the services they are able to use (for example, customers can access the service to make request and reservation, but not the one to accept a request that can only be used by taxi drivers). Furthermore, in this way the system is extendible, because if some new functionality must be added to the system, it's sufficient to add the corrisponding service. The services provided by our system can also be sold to a third part system if it needs them.

4. **Message based point to point:** this architectural style is used to make possible the communication between the service requestor of the customer's frontend and the service provider. In fact, the customer's frontend sends a message only to the core, that has the task to process the user's request and answer him if it finds an available taxi.

2.7.2 Design patterns

1. **Core facade:** the core of our system communicates with the service provider by means of a component, that is the Core facade. In this way the access made by the service provider to the core is much simpler and all the algorithms and functionalities of the core are hidden and unaccessible from the outside.

3 Algorithm Design

3.1 Queue management algorithm

```
PROC QueueManagement(AbstractPerformance pre)
BEGIN
        Request req = pre
        Queue<Taxi> taxiQueue = getZoneQueueOfRequest(req
            )
        Buffer buffer
        Bool taxiFound = false
        Taxi taxi
        Response resp
        OfferResponse offResp
        WHILE(taxiQueue.length != 0)
                taxi = taxiQueue.pop()
                resp = contactTaxi(taxi, req)
                IF(resp == Response.REFUSE)
                THEN
                        moveTaxiToBuffer(taxi)
                ELSE
                        restoreQueue(queue, buffer)
                        sendConfirmationMessage()
                        taxiFound = true
                ENDIF
        ENDWHILE
        IF(!taxiFound)
        THEN
                FOR i FROM 1 TO 3
                        taxi = geti-
                            thNearestTaxiInAnotherZone(i)
                        IF (pre.type == reservation &
                            taxi.timeDistanceFromPoint(pre
                            .startPoint) > minutes(10) )
                                break //all taxis are
                                    further than 10
                                    minutes
                        ENDIF
                        resp = contactTaxi(taxi, pre)
                        IF(resp == Response.ACCEPT)
                                offResp =
                                    sendPerformanceOffer(
                                    taxi, ETA)
                                sendOfferResponseOfUser(
                                    taxi)
                                IF(offerResp ==
                                    OfferResponse.ACCEPT)
                                        taxi =
                                            storeAbstractPerformance
                                            (pre)
```

```
                                                taxiFound = true
                                        ENDIF
                                ENDIF
                        ENDFOR
        ENDIF
        IF(!taxiFound)
        THEN
                sendTaxiNotFoundMessage()
        ENDIF
END
```

Here a brief explanation of the main functionalities of the functions used in this piece of code follows:

- **getZoneQueueOfRequest(req)**: this function takes as input the request sent by the customer and returns the queue of all the taxis that are currently in the zone of the customer.
- **taxiQueue.pop()**: this function extracts the first taxi from the queue previously found and returns it.
- **contactTaxi(taxi, req)**: this function takes as input the taxi extracted from the queue and the request sent by the customer. It sends a message to the taxi driver who is driving the taxi extracted from the queue and returns the respons sent from the taxi driver.
- **moveTaxiToBuffer(taxi)**: if the taxi driver refuses the customer's request, his taxi (which is taken as input) is temporarily moved to a buffer, waiting to be inserted in the queue again in the new position.
- **restoreQueue(queue, buffer)**: this function takes as input the queue of taxis and the buffer that contains the taxis that have refused the request sent by the customer. When a taxi driver accepts the request, the queue of the chosen zone is rebuilt, adding to the queue the taxis that have previously refused the request (located in the buffer). In this way at the end of the execution of this function, the queue has first the taxis that have not been sent the request (if any) and then the taxis that have refused the request following the order of their refusal.
- **sendConfirmationMessage()**: this function sends a confirmation message to the customer with the code of the taxi that accepted his request.
- **geti-thNearestTaxiInAnotherZone(i)**: if the system doesn't find any available taxi in the customer's zone, it finds the nearest taxi located in a different zone. This procedure is repeated up to three times.
- **sendPerformanceOffer(taxi, ETA)**: if the system finds an available taxi outside the customer's zone, it sends a message to the customer with the code of the taxi and the estimated time of its arrival. This function then returns the response of the user (who can accept or refuse the performance).

- **sendOfferResponseOfUser(taxi)**: this function takes as input the taxi that must be notified with the decision of the customer to accept or refuse the ride.
- **storeAbstractPerformance(pre)**: this function takes as input the performance that will be made by the taxi that the function returns.
- **sendTaxiNotFoundMessage()**: if any taxi isn't found to take the customer to his destination, either in the customer zone nor in another zone, a message is sent to the customer saying that there isn't any available taxi at the moment.

4 User Interface Design

For mockups of the application see RASD document mentioned in the *Introduction* section. Mockups gives a quick overview of what a user can do with mytaxy: Basically, if the user is a customer he can submit performance request to the system, if it's a taxi driver ha can login and accept or refuse performance offer.

5 Requirements Traceability

5.1 Allow customers to make a reservation

Using the interface provided by the Performance Handler allocated in Customer Frontend a customer can easily book for a ride at the desired hour. The interaction between components of the system that pursues this requirement can be viewed at subsection *2.5.1 Request for a performance.*

5.2 Allow customers to make a request

The process for making a request it's the same of the reservation's one. The subsection *2.5.1 Request for a performance* gives and overview of components communication and interaction. The distinction between request and reservation takes act in the Prestatinon Input modules (see subsection *2.5.2 Core performance handling*), where the request is immediatly forwarded to Queue Manager component as specified in RASD document.

5.3 Allow customers to use the system via the web-app version

The second diagram in subsection *2.4 Deployment view* shows that different type of physical clients have been developed. Devices *Customer Smarthpone Browser* and *Customer Desktop Browser* allow the usage of the web-app version of mytaxy only for customers.

5.4 Allow customers to use the system via the smartphone app version

Just like in the section above, the second diagram of the deployment view gives a clear idea of what clients can be used to access the service. Devices *Customer Smarthphone App* and *Taxi Smartphone App* are utilized by user for accessing the service. Customer will only use the *Customer Smartphone App* for making every type of request and taxi drivers will use Taxi Smarthpone App for logging into system, accepting or refusing ride offers and setting their availability status.

5.5 Guarantee a fair management of taxi queues for each taxi zone

The management of the queue in a fair way is ensured by the algorithm showed in subsection *3.1 Queue management algorithm*. The same algorithm can be viewed in form of BPMN diagram in subsection 2.5.3 *Queue algorithm*.

5.6 Allow taxi drivers to use a mobile application to access to the system

This point has been already explained in section 4.4.

5.7 Allow taxi drivers to accept a request

A taxi driver can accept a request using the interface provided by the Ride Handler component contained in Taxi Frontend component. The interaction between the several components can be observed in subsection *2.5.4 Contact selected taxi*.

5.8 Allow taxi drivers to refuse a request

In the same way as explained in section 4.7 a taxi driver can refuse a request.

5.9 The service must be extendible

With the chosen SOA architecture the service can be easily extended. Also the high modularity of the system allows an easily development of new components.

6 References

6.1 Used software

For the production of this document the sofware elencated below has been used:

- Document production and layout: LyX `www.lyx.org`
- UML diagrams (component, sequence): UMLet `www.umlet.com`

- BPMN: Draw.io `www.draw.io`