OXFORD BROOKES UNIVERSITY

P00603 - Mobile and Distributed Systems Final report

Alexis Chevalier (15055343@brookes.ac.uk)

June 15, 2016

Contents

1. Introduction

This report will cover the implementation of the Vehicle Rental system. I will introduce the design choices I made during the design specification as well as the implementation choices on the distributed server and the android application.

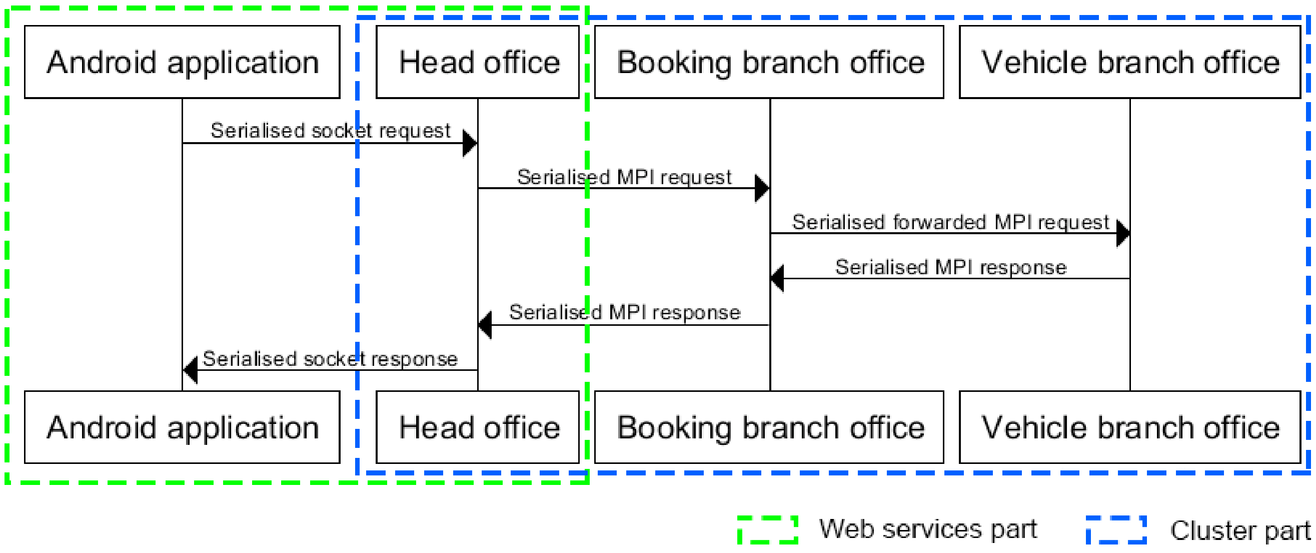
The testing of the final product will be detailed, starting with the testing plan and the results obtained during the testing phases. The configuration of the project will also be described.

Finally, I will critically discuss my work on the project, and present the strengths and the weaknesses of the actual system as a commercial product.

2. Communications design

Communication is the most important part of this project's design because it enables interaction between between all the actors of our distributed system. The figure 1 gives an overview of how the message passing is used in the most complex operation of the system, a vehicle booking request on a foreign branch.

Figure 1: Message passing systems used during a booking request.



As we can see, the request starts from the android application to the head office, this part is handled by the web service (which will be detailed in the next subsection). Then the request is transmitted to the distributed cluster, starting with the head office to a first branch office, which will host the booking. The vehicle however, is located on another branch, so we have to know if the vehicle will be available for the given dates, which is why the MPI request is forwarded to the vehicle branch, this branch will now check the vehicle availability and create the booking, in order to avoid latency.
The response must then be sent back to the branch office, then to the head office,

which will then reply to the web service request by a web service response, including the results from the MPI response.

This is a complex configuration, which requires synchronisation between the involved nodes, I am going to explain how I designed my communication protocols for both the web services and the cluster message passing with transparence and simplicity in mind.

2.1. Web services

2.1.1. Current implementation

I choose not to use a generic webservice system like SOAP, REST, or RPC in order to develop my own domain-specific system which would handle all the needs of the project without too much overhead. The system I decided to use borrows elements from those generic systems, but is also slightly lighter and doesn't use the HTTP protocol.

My system is based on TPC sockets, and uses JSON as message format notation in order to achieve a request-response process. It uses a light request enveloppe (see listing 1), composed of an operation code (integer), an authentication parameter, a current branch id and an optional message payload. The response envelope is different (see listing 2), it still includes the operation code, but also includes a status code (integer based on HTTP status codes), an optional error message and an optional response payload.

Listing 1: Web service request envelope

```
{"op_code":2, /* defined integer */
 "auth": "dXNlcjpwYXNzd29yZA==", /* base64(user:password)
    ↪ */
 "branch":"London", /* branch name */
 "serialized_object":"{\"key\":\"value\"}" /* serialised
    ↪ business object */}
```

Listing 2: Web service response envelope

```
{"op_code":2, /* defined integer */
 "status": 400, /* HTTP protocol status codes */
 "error":"Missing parameter", /* Customised error message
    ↪ */
 "serialized_object":"{\"key\":\"value\"}" /* serialised
    ↪ business object */ }
```

Synchronisation is not a problem here because when the socket receives a request, it will wait for a response before replying, the only problem could be a deadlock, but I haven't encountered a single one during all the testing process and the development period.

An important point here is to know that the whole architecture is single-threaded (because I didn't succeeded to use sockets with MPI on the HAC cluster), which means that the whole system can only handle a single request at a time and therefore isn't sensible to race conditions, however, most of the code is prepared for a switch to multithreaded environnement, I will talk more about this point in the critical evaluation.

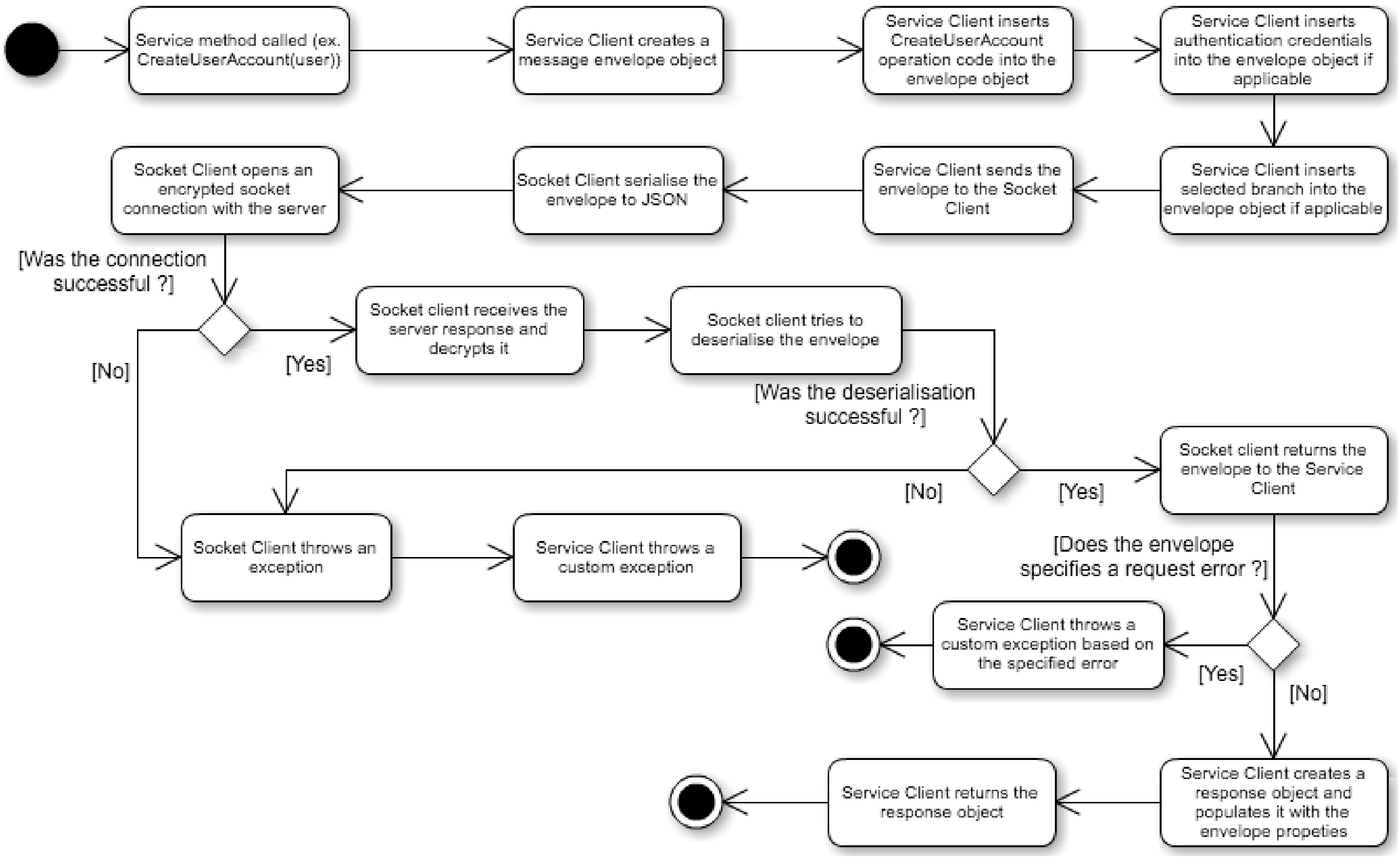
From the client's point of view, the system looks like an RPC middleware, a usage example is given in the listing 3 (The code has been voluntarily simplified and generalised), we can see that the client will transparently only call a single method with a specific object type to serve as method parameters.

Listing 3: Service client usage example

```java
try {
  //registerContract is a type shared with the server code
  UserContract createdUser =
     ↪ serviceClient.createAccount(registerContract);
} catch (ApiException e) { /* Handle errors ... */ }
```

It is not as generic as an implementation like Java RMI but it gives an appreciable abstraction when developing multiple clients. The practical processing of my protocol is described in two activity diagrams, the first one (see figure 2) explains the process on the client side.

Figure 2: Activity diagram for the web service system on the client side.



Service method called (ex. CreateUserAccount(user))
Service Client creates a message envelope object
Service Client inserts CreateUserAccount operation code into the envelope object
Service Client inserts authentication credentials into the envelope object if applicable
Service Client inserts selected branch into the envelope object if applicable
Service Client sends the envelope to the Socket Client
Socket Client serialise the envelope to JSON
Socket Client opens an encrypted socket connection with the server
[Was the connection successful ?]
[No]
[Yes]
Socket client receives the server response and decrypts it
Socket client tries to deserialise the envelope
[Was the deserialisation successful ?]
[No]
[Yes]
Socket client returns the envelope to the Service Client
Socket Client throws an exception
Service Client throws a custom exception
[Does the envelope specifies a request error ?]
Service Client throws a custom exception based on the specified error
[Yes]
[No]
Service Client creates a response object and populates it with the envelope propeties
Service Client returns the response object

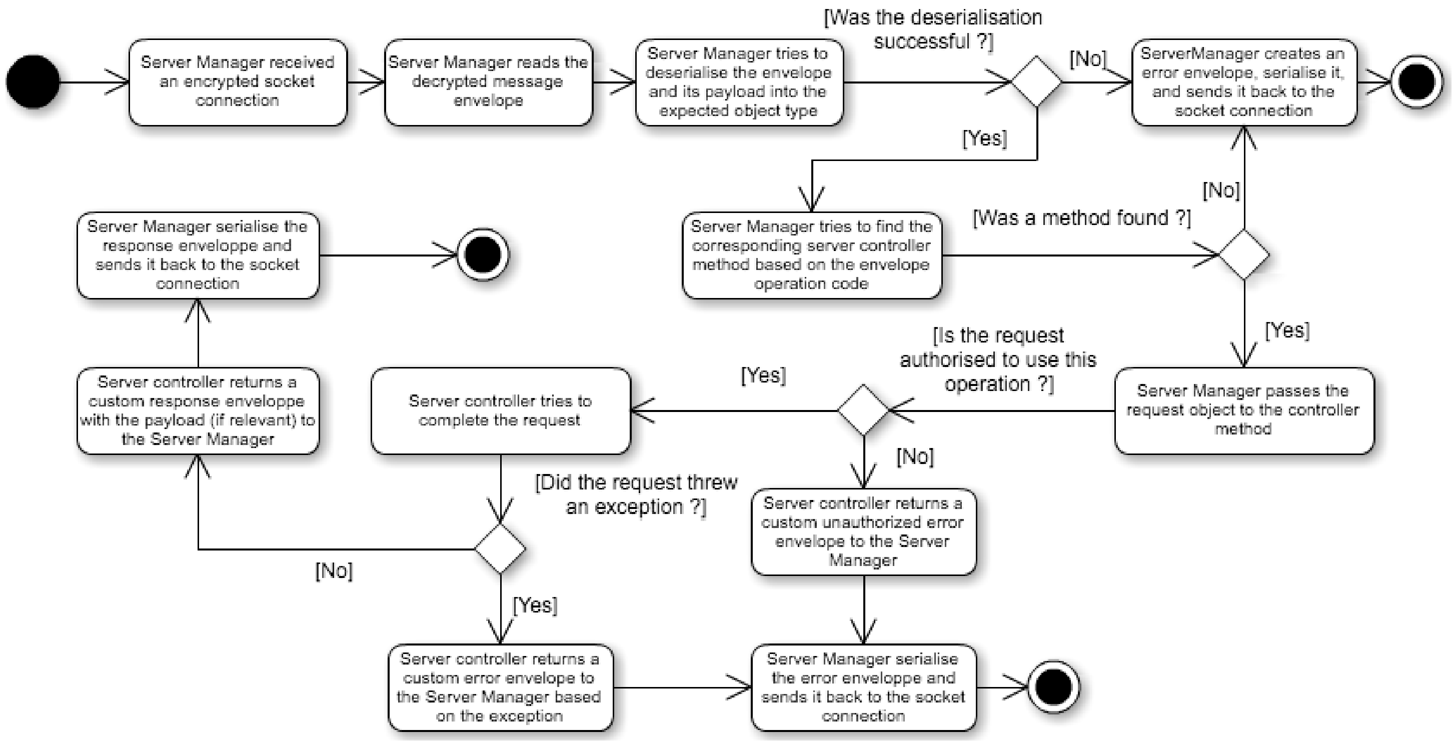
As we can see, it is a complex process, but it is entirely automatised in the API (service) client code, the goal of this system is to hide the complexity of the implementation behind a simple interface which provides a single method, accepting one or

more parameters and returning a simple type.
Even if I made a small comparison to RPC systems, I do not provide proxy objects, objects returned from the service client are completely owned and localised on the client code, the only way to apply a modification on the server is to call the methods provided by my implementation.

This system is also using some properties of REST systems, even if don't have the concept of URL or HTTP protocol in my implementation, I am using the HTTP codes to handle the response status, which is used in REST services. For example, every time I receive a status code starting with the number 2, I know that my request has been successful, on the contrary, if I receive a 403 code, I know that my authentication credentials are not authorised to use this specific operation.
Also, as the REST system, my implementation is completely stateless, even if this means that I have to send back the authentication scheme all the time, I know that I will be able to handle a server restart without any problem for the clients.

The server part is different but follows the same principles, as shown in figure 3, we can see that there is now a method-matching system, which will forward the request envelope to a previously implemented method based on the unique request code (those codes are shared between the server and the client).

Figure 3: Activity diagram for the web service system on the server side.



Server Manager received an encrypted socket connection
Server Manager reads the decrypted message envelope
Server Manager tries to deserialise the envelope and its payload into the expected object type
[Was the deserialisation successful ?]
[No]
ServerManager creates an error envelope, serialise it, and sends it back to the socket connection
[Yes]
Server Manager tries to find the corresponding server controller method based on the envelope operation code
[Was a method found ?]
[No]
[Yes]
Server Manager passes the request object to the controller method
[Is the request authorised to use this operation ?]
[Yes]
[No]
Server controller returns a custom unauthorized error envelope to the Server Manager
Server Manager serialise the error enveloppe and sends it back to the socket connection
Server controller tries to complete the request
[Did the request threw an exception ?]
[No]
[Yes]
Server controller returns a custom error envelope to the Server Manager based on the exception
Server controller returns a custom response enveloppe with the payload (if relevant) to the Server Manager
Server Manager serialise the response enveloppe and sends it back to the socket connection

This systems allows me to have clean and separate classes to differentiate the message passing implementation from the business logic, only a thin binding exists between those two parts.

As for the client, errors are handled at multiple levels of the server, custom exceptions are used in order to have typed errors, which is really easy to handle with Java.

Error codes are defined by the server depending which part of the request failed in order to give the most precise response to the client.

Finally, the table 1 summarise all the available operations of the service with some interesting properties.
The code is the integer value used to differentiate operations, safe means that the operation won't modify anything in the system, idempotent means that the request can be called many times while producing the same result (or it will return an information message), finally, Auth. level means the required authorisation level for the account.

Table 1: Available webservice operations

Operations list				
Code	Description	Safe	Idempotent	Auth. level
0	Get all the available branches	Yes	Yes	Guest
1	Search for available vehicles	Yes	Yes	Guest
2	Create user account	No	Yes	Guest
3	Book a vehicle	No	Yes	User
4	Get branch user booking list	Yes	Yes	User
5	Get user account details	Yes	Yes	User
6	Create other user account	No	Yes	Staff
7	Create or Update vehicle	No	Yes	Staff
8	Search user	Yes	Yes	Staff
9	Get the branch booking list	Yes	Yes	Staff
10	Change a booking validation state	No	Yes	Staff
11	Search a(many) vehicle(s)	Yes	Yes	Staff
12	Shutdown the system	No	Yes	Staff
13	Get future branch vehicle moves	Yes	Yes	Staff

2.1.2. Alternatives

There is many other possibilities to implement a web service such as this one but most of them (SOAP, REST, RMI) require either the HTTP protocol or a heavy XML envelope, given the fact that we had to use MPI I thought that performance was one of the key points so I didn't wanted to use a heavier protocol. That's why I decided to stay with this implementation to keep the simplicity and the lightness of the system.

In a commercial application, I should change the encryption system to use an asymmetric encryption used to determines a common symmetric key for the communication (as does HTTPS with SSL/TLS) in order to avoid bundling the secret keys inside the android application, as they could be reverse engineered and if it is the case, anyone with the keys could decrypt the communications. In that case, using HTTP could be a good idea because this is already implemented in the protocol and in the existing clients.

If a change was to come, the design of the web service client and server provides a decent level of abstraction and should allow the modification of the communication technology without much changes in the other parts of the programs.

2.2. Cluster message passing

2.2.1. Current implementation

Inside the cluster, a requirement imposed the usage of MPI (message passing interface) to serve as orchestrator for the distributed system. I decided to build a system over MPI in order to facilitate the use inside the program.

The system is very similar to the one used for the web service communications, it uses two simple envelopes for the requests and responses, which are both serialised before being sent through MPI and then deserialised after reception. A layer of encryption has been added at this point too, in order to prevent an attacker from listening to the connections from inside the cluster.

The request envelope can be seen in the listing 4, it uses composed of the same operation code that was used before in the web service. The authentication is made on the head office, so this time we have a representation of the user, serialised, instead of the authentication string. Finally, the optional request payload is still present because the branch will need the payload for some requests.

Listing 4: MPI request envelope

```
{
    //defined integer
    "op_code":2,
    //serialised user representation
    "user": "{\"id\":\"0\", \"isStaff\":true...}",
        //serialised business object
    "serialized_object":"{\"key\":\"value\"...}"
}
```

As for the response, the envelope is visible in listing 5, it still includes the operation code and the serialised object, but also a status code to describe the status of the request, and an optional error message if there was a problem.

Listing 5: MPI response envelope

```
{
    //defined integer
    "op_code":2,
    //HTTP error code
    "status": 400,
    //Custom error message
    "error": "Invalid email",
    //serialised business object
    "serialized_object":"{\"key\":\"value\"...}"
}
```

During the prototyping phase, I quickly noticed that the Java API provided by OpenMPI was not really like the usual APIs found in the Java community (I think it was too much inspired from the C/C++ API, which usually makes a large usage of pointers as 'out' parameters) and I didn't liked it very much, that's why I decided to implement a wrapper, called ClusterCommunicator.

The ClusterCommunicator simply provides two methods (sendObject and receiveObject) which makes use of MPI procedures to communicate with another branch. An example of communication is visible in the listing 6 (The code has been voluntarily simplified and generalised), we can see that only two methods are used, sendObject, with the branch ID, the operationCode, and the object to send (here a request envelope), and then receiveObject, with the branch ID, the operation code and the type of the expected response.

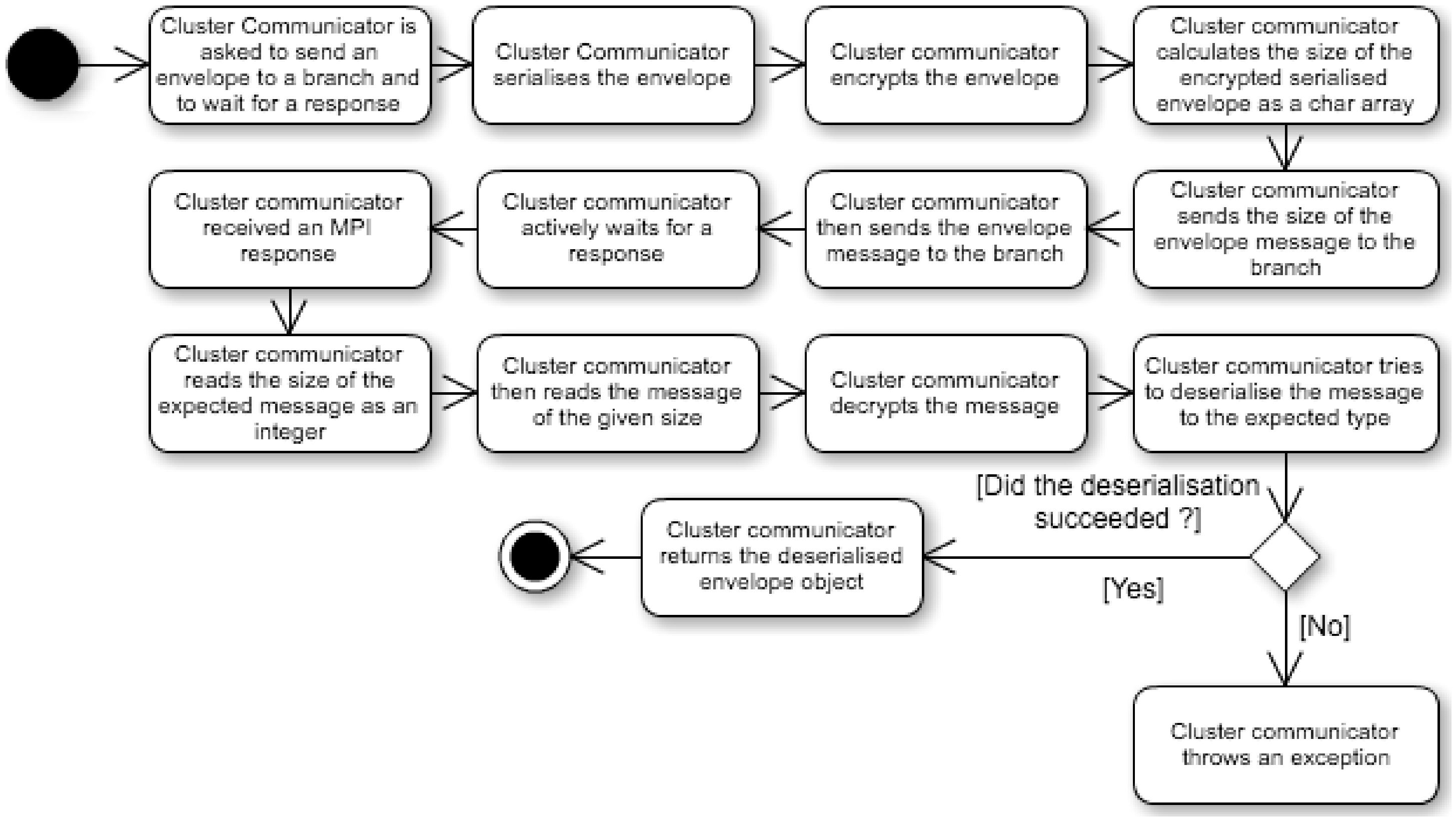
Listing 6: MPI wrapper usage example

```
try {
    //Sending the serialised envelope to the branch
    clusterCommunicator.sendObject(branchId,
      ↪ operationCode, requestEnvelope);

    //Receiving the response envelope, containing a
      ↪ payload of type Contract
    ResponseEnvelope responseEnvelope =
      ↪ clusterCommunicator.receiveObject(branchId,
      ↪ operationCode, ResponseEnvelope.class);
} catch (ClusterCommunicatorException e) { /* Handle
  ↪ errors ... */ }
```

The figure 4 shows what happens when those the previous code is used in the cluster from the calling node side.

Figure 4: Activity diagram for a standard MPI Request/Response communication in the cluster.


Cluster Communicator is asked to send an envelope to a branch and to wait for a response
Cluster Communicator serialises the envelope
Cluster communicator encrypts the envelope
Cluster communicator calculates the size of the encrypted serialised envelope as a char array
Cluster communicator sends the size of the envelope message to the branch
Cluster communicator then sends the envelope message to the branch
Cluster communicator actively waits for a response
Cluster communicator received an MPI response
Cluster communicator reads the size of the expected message as an integer
Cluster communicator then reads the message of the given size
Cluster communicator decrypts the message
Cluster communicator tries to deserialise the message to the expected type
[Did the deserialisation succeeded ?]
[Yes]
[No]
Cluster communicator returns the deserialised envelope object
Cluster communicator throws an exception

In order to send serialised and encrypted messages through MPI, we start by sending the length (in characters) of the message to the other node, then we send the actual message (as a character array). Using this system allows the other node to correctly receive the message without having to use an end of string character or other communication delimiters.

I won't describe what happens on the other node side, because it's just the opposite of this activity diagram and can be inferred from it (The cluster communicator of the second node will start by an active wait, then it will receive a request from the first node, and finally will send a message as a response to the first node).

As we can see in the code and in this report, the internal communication is less transparent than the external one, an abstraction layer is still present, but not as good as the communication layer designed for the web service. However from a commercial point of view this is less important, the internal communication needs to be secure and efficient while the external communication (if accessible by external developers) needs to be secure, efficient, and as transparent as possible to use if we want external services to interact with ours.

2.2.2. Alternatives

The actual implementation works correctly, however there is a critical problem with the lack of multithreading, this could lead to scalability issues and deadlocks in the system. Having a separate thread for each request would help to handle more request simultaneously and could allow to put a timeout on the requests to remove deadlocks after a certain amount of time. In that case, the system would have to support multiple request at the same time, this will be discussed in the software design part of the report.

2.3. Added and modified operations

Nearly every operation kept the initially proposed message passing scheme from the design report, I won't put the sequence diagrams here as it would simply be a duplicate from the first report.
However, the "Create booking operation" has changed a little bit, and the "Retrieve future vehicle moves" has been added to the system, I am going to briefly explain those modifications.

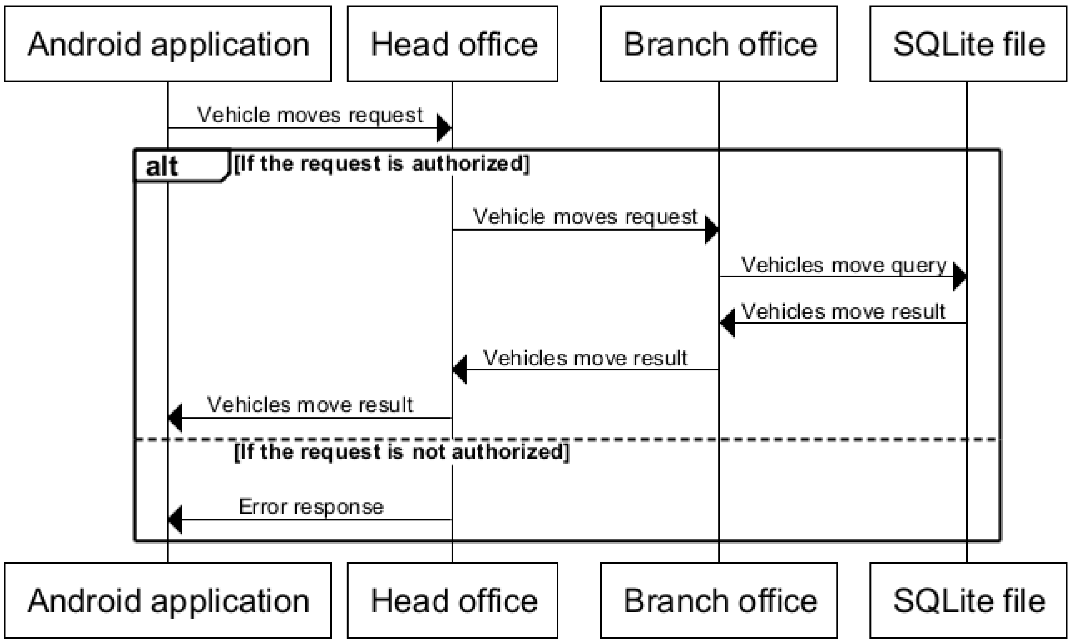
2.3.1. Retrieving future vehicles moves

This operation is completely new, it wasn't planned at all during the design proposal, but during the android application development I thought it was a good idea to display the incoming and outgoing vehicles in each branch, so that the staff members would be easily able to plan their day at work.

The sequence diagram in figure 5 shows that this is a straightforward operation, the request comes from the android application to the head office, then if it is correctly authorised it goes to the given branch office which queries the database to get the expected incoming (or outgoing, depends on the request payload) vehicles for this

branch for today and other future days.
Once the result is in the branch, it is sent back to the android application, step by step.

Figure 5: Sequence diagram for the get vehicle moves operation.

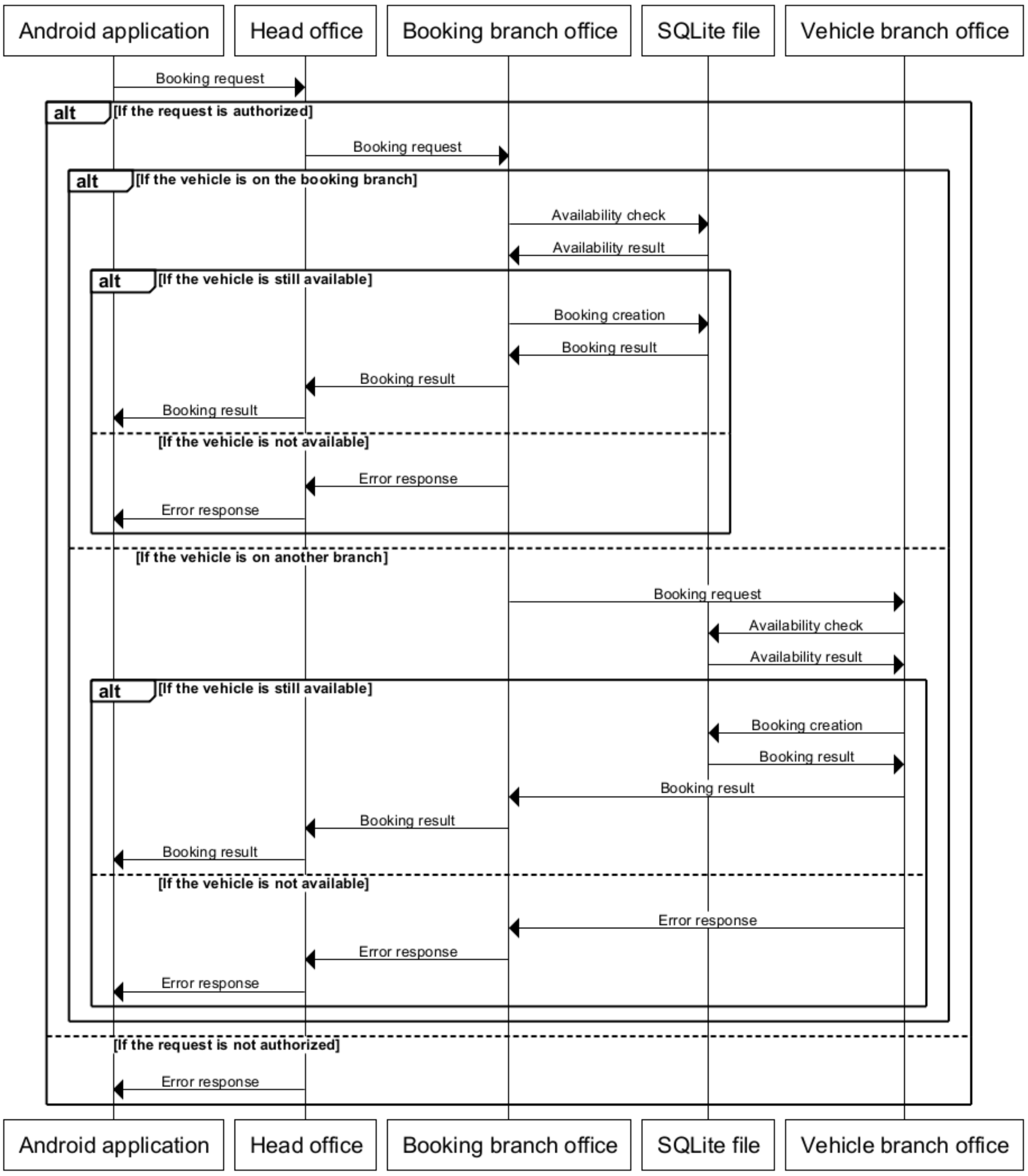


2.3.2. Booking creation

The booking creation operation was already specified into the design proposal, but failed to take into account the booking of a vehicle on a foreign branch. In the sequence diagram proposed in figure 6, we can see that another branch is added to the system, and if the vehicle is on that branch, then all the database operations are going to be done on the this branch in order to avoid latency between the validation and the booking (in a multithreaded system, an SQL transaction could be used).

This request is the most complex of the whole system, and the sequence diagram is even simplified here and doesn't take into account the potential errors if a parameter of the request is invalid.

Figure 6: Sequence diagram for the book vehicle operation.



3. Software design

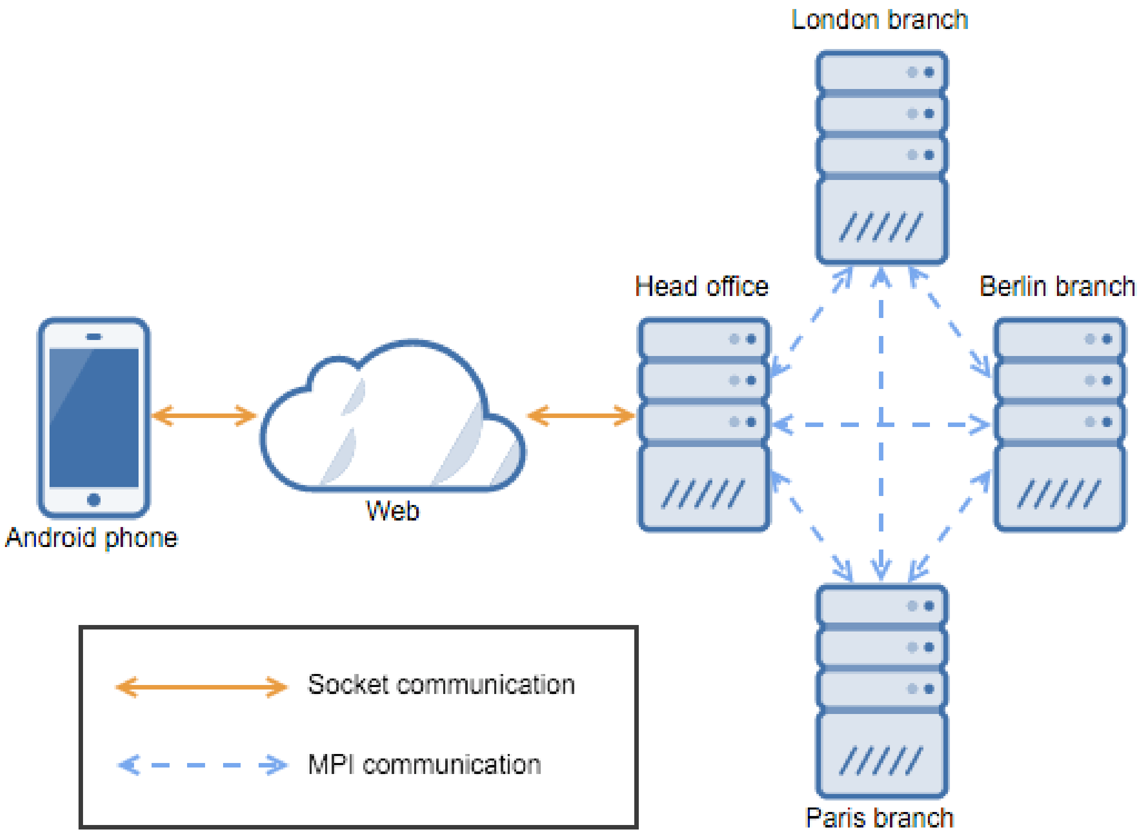
In this section I will discuss the main parts of my implementation design, starting with the distributed server design and then the android application. I will explain the global architecture and the reason for my choices.

3.1. Distributed server design

3.1.1. Distributed orchestration and lifecycle

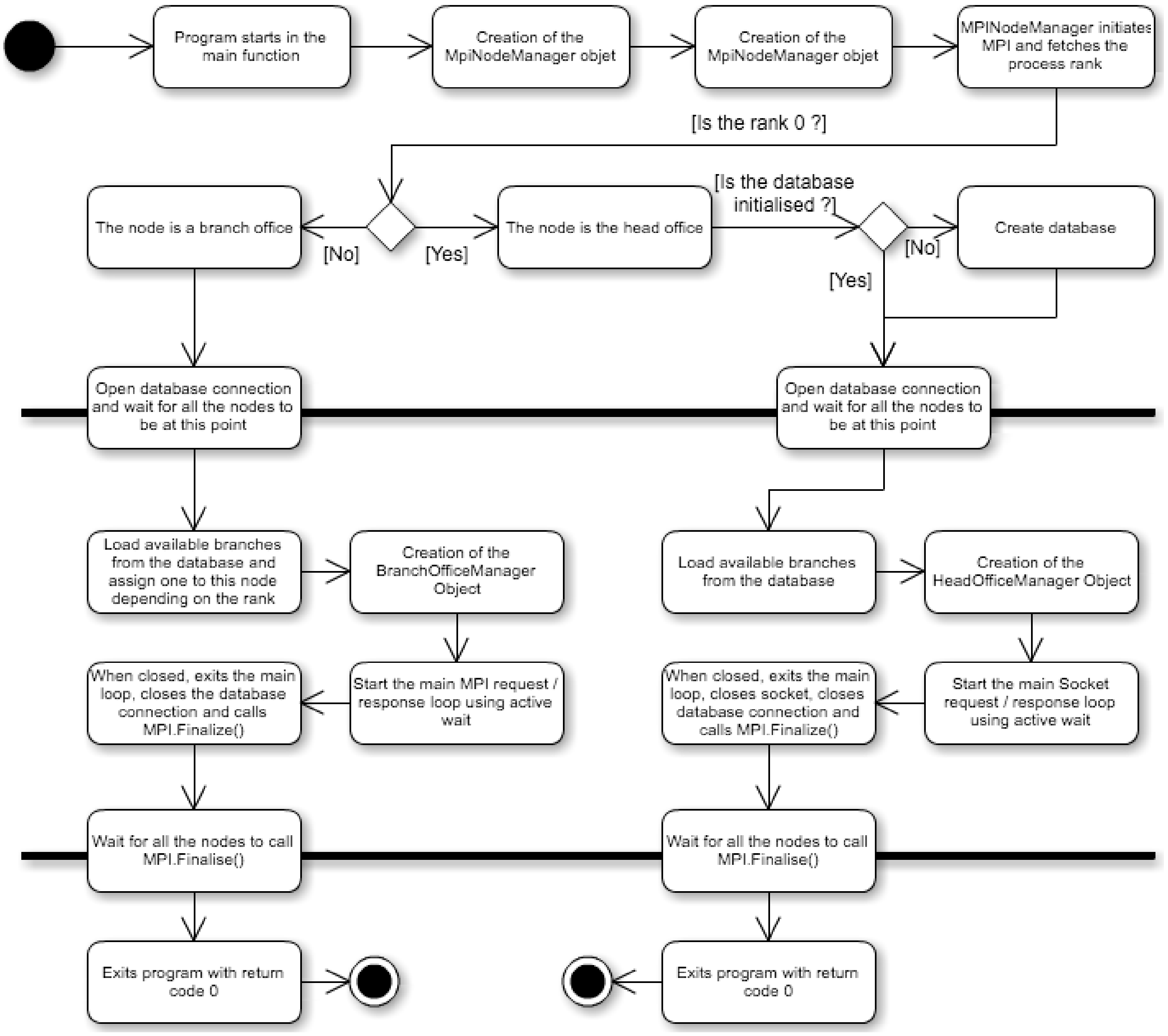
Globally speaking, the implementation of the distributed system didn't changed much since the design proposal, as shown in figure 7, there is still a cluster of different nodes which can communicate directly between each other using MPI. Aside of that, the head office offers a connection to internet through a socket.

Figure 7: General design of the system.



The lifecycle of the system wasn't deeply described in the previous report because I wasn't completely sure of the implementation using MPI. The figure 8 displays a precise and complete illustration of each node's lifecycle.
We can see that at the beginning, all the nodes share the same process, but as soon as the rank of the node is determined, two path are proposed. The first path, on the left, represents a branch node (could be considered as a slave node in a master-slave system).

Figure 8: Activity diagram picturing the lifecycle of the distributed nodes.


Program starts in the main function
Creation of the MpiNodeManager objet
Creation of the MpiNodeManager objet
MPINodeManager initiates MPI and fetches the process rank
[Is the rank 0 ?]
The node is a branch office
[No]
[Yes]
The node is the head office
[Is the database initialised ?]
[No]
[Yes]
Create database
Open database connection and wait for all the nodes to be at this point
Open database connection and wait for all the nodes to be at this point
Load available branches from the database and assign one to this node depending on the rank
Creation of the BranchOfficeManager Object
Load available branches from the database
Creation of the HeadOfficeManager Object
When closed, exits the main loop, closes the database connection and calls MPI.Finalize()
Start the main MPI request / response loop using active wait
When closed, exits the main loop, closes socket, closes database connection and calls MPI.Finalize()
Start the main Socket request / response loop using active wait
Wait for all the nodes to call MPI.Finalise()
Wait for all the nodes to call MPI.Finalise()
Exits program with return code 0
Exits program with return code 0

We can understand that this node only waits for all the other nodes to be ready (using MPI.Barrier()) because we need to wait for the head office to set up the database if its the first launch, then it loads the branches, determines which branch he should run (the branch id will be the same than the process rank) and starts the main loop, an MPI listener, waiting for requests to process. When the branch has to close, the node will properly close its database access and wait for all the other MPI nodes to finalise before going off.

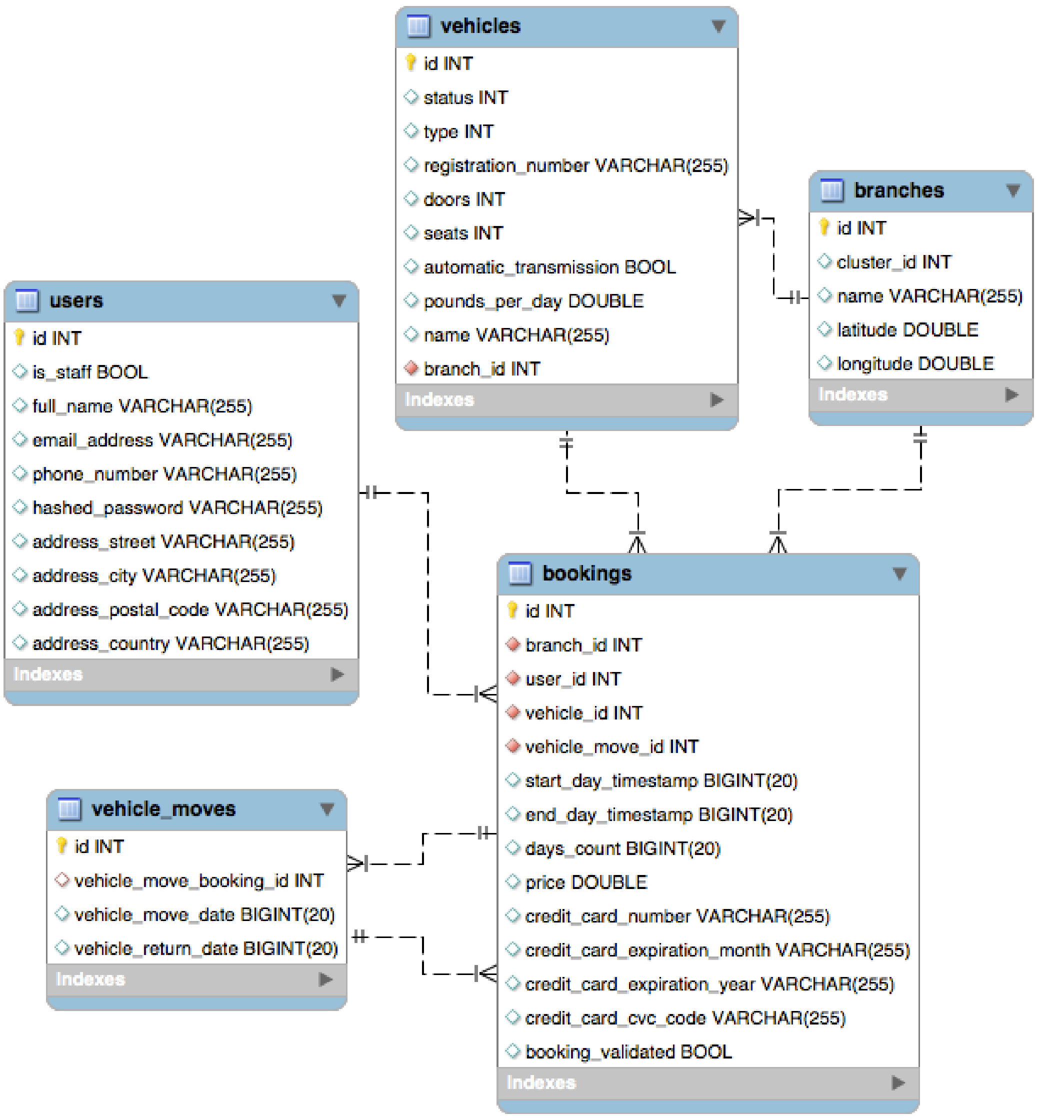
The second path, on the right, is used when the head office node (the master) is detected, it will more or less follow the same procedure, excepts that he will also initialise the database if necessary. Instead of using a MPI waiting loop it uses a socket waiting loop, and he also obviously have to close the socket properly before exiting.

The lifecycle of each node is straightforward for an application of this size, I designed it like this in order to avoid complex errors in the main program orchestration.

3.1.2. Database management

A part of the design proposal briefly introduced an UML class diagram displaying the entity scheme I was planning to use, most of this scheme was kept for the last version, but a few modifications were made. As we can see in the figure 9, a table called vehicle_moves has been added, a move exists for every booking using a vehicle on a foreign node, I use this scheme in order to efficiently find available vehicles for a given period of time with an SQL query.

Figure 9: Entity relationship diagram for the database.


vehicles
id INT
status INT
type INT
registration_number VARCHAR(255)
doors INT
seats INT
automatic_transmission BOOL
pounds_per_day DOUBLE
name VARCHAR(255)
branch_id INT
Indexes
branches
id INT
cluster_id INT
name VARCHAR(255)
latitude DOUBLE
longitude DOUBLE
Indexes
users
id INT
is_staff BOOL
full_name VARCHAR(255)
email_address VARCHAR(255)
phone_number VARCHAR(255)
hashed_password VARCHAR(255)
address_street VARCHAR(255)
address_city VARCHAR(255)
address_postal_code VARCHAR(255)
address_country VARCHAR(255)
Indexes
bookings
id INT
branch_id INT
user_id INT
vehicle_id INT
vehicle_move_id INT
start_day_timestamp BIGINT(20)
end_day_timestamp BIGINT(20)
days_count BIGINT(20)
price DOUBLE
credit_card_number VARCHAR(255)
credit_card_expiration_month VARCHAR(255)
credit_card_expiration_year VARCHAR(255)
credit_card_cvc_code VARCHAR(255)
booking_validated BOOL
Indexes
vehicle_moves
id INT
vehicle_move_booking_id INT
vehicle_move_date BIGINT(20)
vehicle_return_date BIGINT(20)
Indexes

Aside of that, a few properties were added in order to store the credit card informations for a booking for instance.

In order to fetch the available vehicles, I am using a single SQL query (visible in appendix A), involving a subquery in order to get all the vehicles that doesn't have any existing bookings in the given time period. I am using the table vehicle_moves to include the moves in my calculations directly inside the query.

The code used to generate the query can't directly use the library ORMLite (this is one of the two request that are too complex for the library), therefore I had to create a manual implementation of a query builder using a cache to avoid doing many string concatenations at every request.
The reason for this system is that SQL is not a particularly maintainable language, because it is not typed with Java and therefore makes refactoring very difficult, that's why I decided to use name constants in the program to name every table and field, which makes that whenever the java code is changed, the SQL query will be updated too.
I won't insert the code here because it is too large and not directly relevant to be included in the report, but everything is implemented in the file com/vehiclerental/-dataLayer/sqliteImplementation/VehicleDaoSqliteImpl.java.

3.1.3. Code structure

This part was already detailed in the design proposal and didn't changed much, an updated (but simplified) UML class diagram can be found in figure 10.

We can directly see the three tier layered system, starting with the data layer, composed of a base Data Access Object (DAO) interface, and its implementation for SQLite, BaseDaoSqliteImp, they provide the basic CRUD operations (Create, Update, Update, Delete) for all the entities in the system. Then comes the specialisations, with the UserDao, BookingDao, VehicleDao, BranchDao and VehicleMoveDao interfaces and their sqlite implementations, they inherit from the BaseDao in order to provide a simple access to the generic methods, but also provides their own specialised methods, for example "getUserByEmail(String email)" for the UserDao.
The DaoFactory is used in order to hide the implementation from the other layers, all that the other layers know is that the factory gives an object which is compliant to the interface, they don't care if it uses SQLite, Mysql, or a storage file in the implementation.

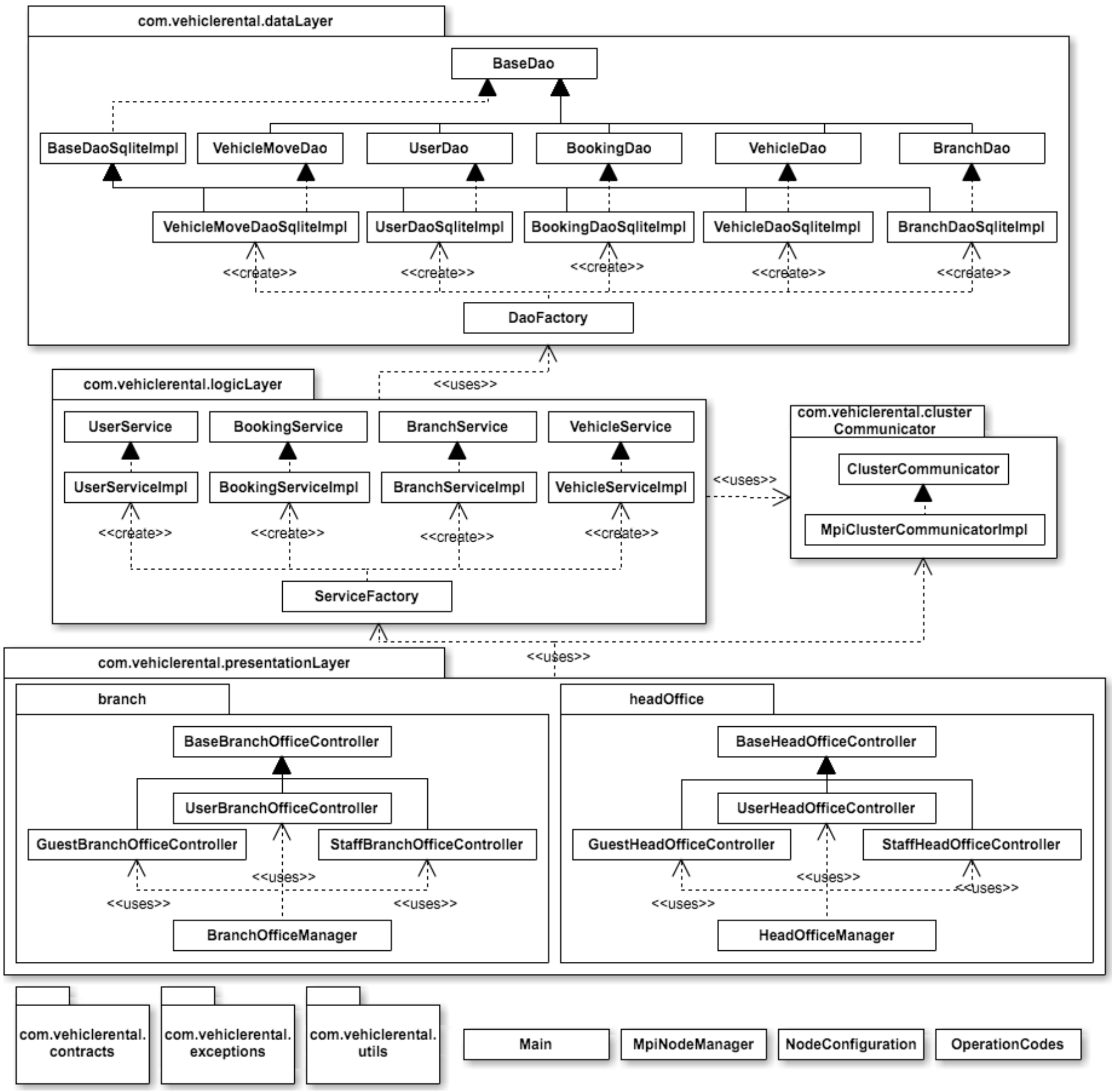
The second layer is the logic layer, composed of different services with their interfaces. This layer provides the business logic of our application, the rules that are inherited from the vehicle rental business. The goal of the interfaces here is to be able to provide stub classes in case of unit testing later on, to force a defined standard inside the application, and again, to hide the implementation from the usage.

Then, the presentation layer is used to provide an external interface for the android application (in case of the head office), or for the other branches (in case of a branch office).
The head office will use the HeadOfficeManager to run the socket server and will dispatch the requests to the Guest, User and Staff HeadOfficeControllers depending on the operation code used. All the controllers inherits from BaseHeadOffice controller in order to provide common methods (like ForwardToClusterNode, which sends an

MPI request and receives the response in a single line) to all the controllers without duplicating code.
The branch office uses the exact same system except that instead of a socket server it is an MPI listening loop and the common methods to the controllers are different.

Aside of this main structure, the code provides different helpers (To handle dates, cryptography and serialisation), the contract classes used to send information between the nodes/android application and some custom exceptions, to handle errors from the other layers of a common and typed way. The OperationCodes class is simply containing all the available operation codes for the system.
Finally, the main class initialises the system and uses MpiNodeManager and NodeConfiguration (as shown in figure 8) in order to set up the node in the cluster.

Figure 10: Simplified UML class diagram for the server.

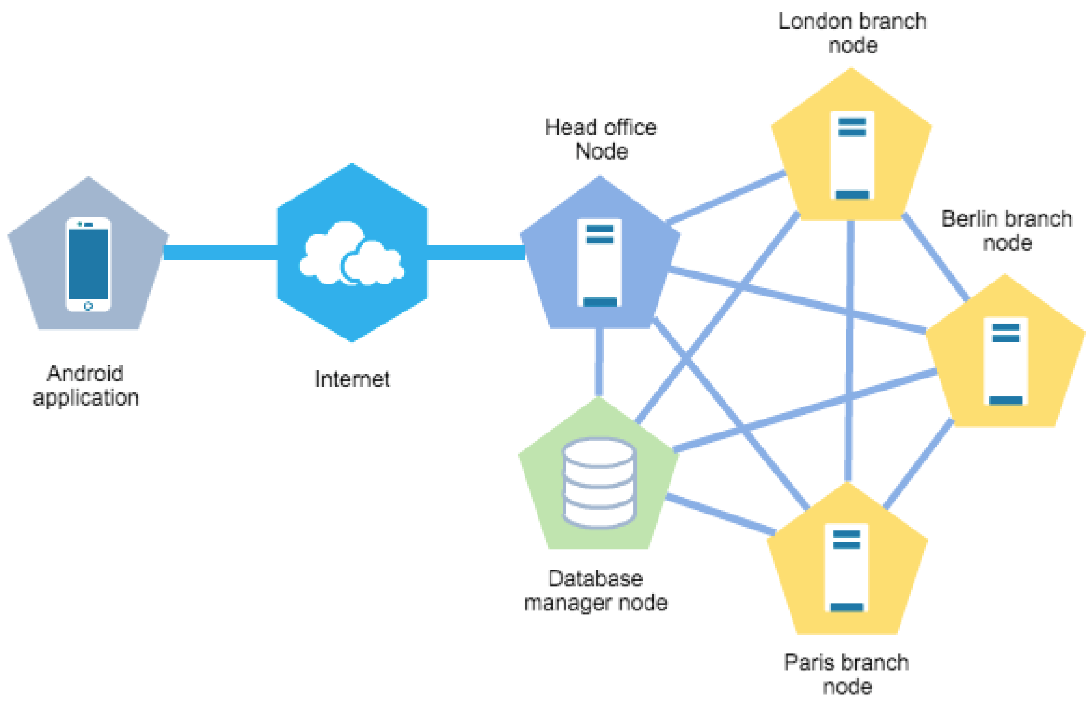


This structure has been designed with maintainability and extensibility in mind, as described the three tier architecture offer a low coupling and an ability to quickly change a single part of the code without affecting the other layers.

3.1.4. Alternatives

My initial plan for the distributed server architecture was much more complex and offered better scalability features, unfortunately my prototype wasn't working correctly with MPI on the HAC cluster. I am going to explain how I initially planned to create the architecture (visible on figure 11).

Figure 11: Improved distributed architecture.



There is two architectural problems with the actual design :

- The whole system is single threaded, which makes the system hard to put at scale if the number of concurrent requests increase
- There is no possibility to programatically stop a node and restart it

The initially planned design solved those two problems using threads and MPI.Spawn(), unfortunately, the OpenMPI version installed on the cluster didn't worked correctly with my code, and some requests were completely lost in the system, causing deadlocks. I didn't had a lot of time at the moment to search for a better implementation, that's why I switched to the current design. If I can solve this problem, then this improved architecture can be implemnted.

The threads are relatively easy to implement (the actual code should actually not be a problem if we add threads to the socket server and the MPI listening loop), the problem is that we may have concurrent MPI requests and concurrent database access.
The concurrent MPI requests can be achieved using an unique ID as request tag, this way every thread in the system could wait and expect a specific response, and the system would not need to block everything as it does at the moment.
The database problem is slightly more complex, as we are using an SQLite file, we

can't handle concurrent write access to the file, it has to be done synchronously using a task queue. The architecture proposed in figure 11 introduces a new cluster node, used as a database manager. In this architecture, this server would be the only one able to access the SQLite file, and would be multithreaded. It would use a task queue in order to treat every thread one by one, and then return the SQL response to the requesting node (it could be a branch or the head office).
Another option of course would be to use a real SQL server like MySQL or PostgreSQL if available.

Using this system increases scalability because there is now only a single bottleneck in the system, and it only waits for the result of an optimised SQL query, everything else is done asynchronously and therefore considerably increases the performance.

In order to be able to restart nodes, we need to be able to spawn them dynamically and use the name server provided by MPI. The initial prototype was able to spawn and close individual nodes, but not to use the name server, so it was really difficult to communicate between nodes because they weren't using the same MPI communicator.
Using MPI.Spawn would allow to restart nodes when desired, even to add a new branch dynamically to a running system. When used in common with the provided name server, each node would be able to register/remove itself from the cluster and therefore gives the ability to only shut down a specific branch or multiple branch while keeping the head office and the database server available.

Another distributed protocol as AMQP could be used instead of MPI in order to achieve communication between the nodes, but AMQP doesn't provides the ability to start different processes on different nodes, so I would have to implement that part manually with a bootstrap software.

Finally, in order to improve scalability, a single database could be use per node, this would reduce the bottleneck pressure on the database, but would require more cluster requests to find foreign available vehicles.

3.2. Android application design

3.2.1. Current implementation

The implementation of the android application is pretty straightforward and doesn't present any complex aspects in order to limit the size of the application. The figure 12 gives a simplified UML class diagram of the architecture. We can see that this time 3 main components are implemented.

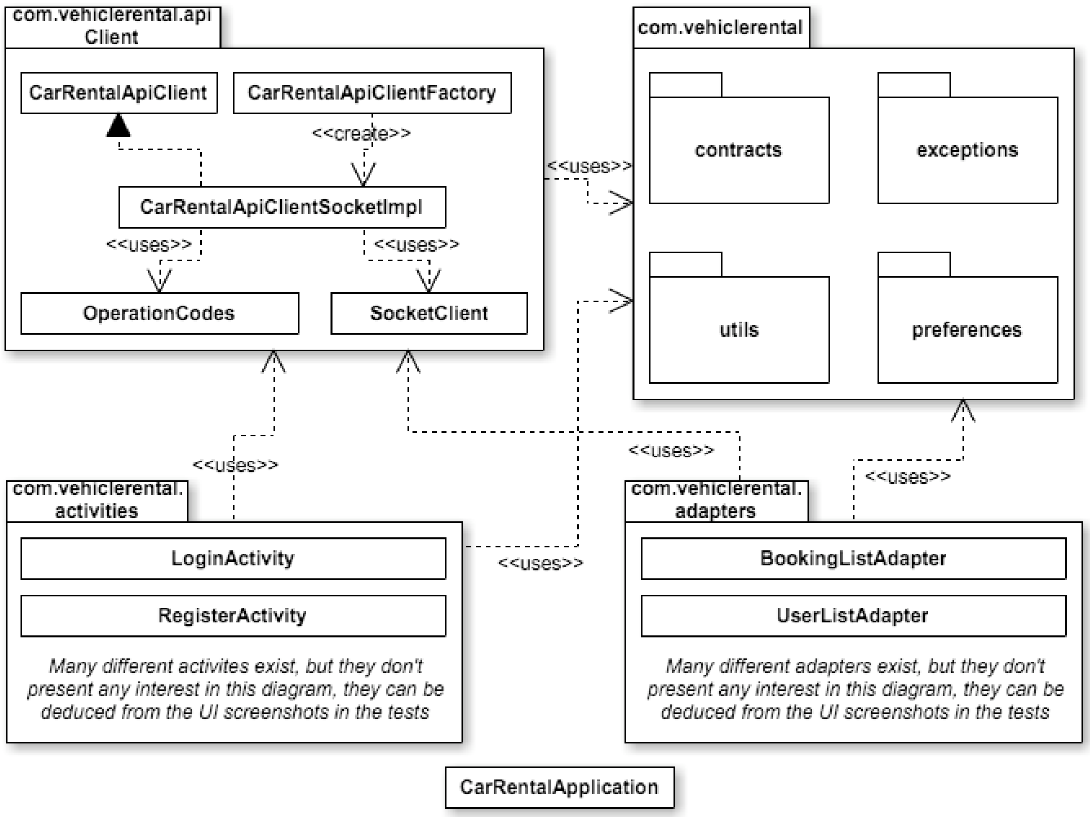
The ApiClient provides a level of abstraction through its interface in order to be able to easily change the communication protocols if a change to HTTP or to another system is needed. This also adds some transparence for the implementation of the activities and adapters.

The two main parts of the android application are the Activities and the Adapters. An activity represent a view, as you can see in the testing report, where all the views

are visible, I use many different activities. Each activity is composed of the main class, which binds all the values and event listeners to the XML layout, but also of an AsyncTask when required. AsyncTasks are the concept of threads in Android, with an abstraction layer in order to be easily able to modify the UI thread when the service call is finished.
In order to avoid a frozen screen, all the service calls are executed inside async tasks.

The adapters follow the same logic than the activities, they are used to handle lists in the application. Even if I tried to make them as generic as possible, I don't have any case where I can reuse them twice. They also provide some bindings, event listeners and async tasks when required.

Figure 12: Simplified UML class diagram for the android application.


com.vehiclerental.api Client
CarRentalApiClient
CarRentalApiClientFactory
<<create>>
CarRentalApiClientSocketImpl
<<uses>>
<<uses>>
OperationCodes
SocketClient
com.vehiclerental
contracts
exceptions
utils
preferences
<<uses>>
<<uses>>
<<uses>>
<<uses>>
<<uses>>
com.vehiclerental. activities
LoginActivity
RegisterActivity
Many different activites exist, but they don't present any interest in this diagram, they can be deduced from the UI screenshots in the tests
com.vehiclerental. adapters
BookingListAdapter
UserListAdapter
Many different adapters exist, but they don't present any interest in this diagram, they can be deduced from the UI screenshots in the tests
CarRentalApplication

There is also other packages, they are less directly related to the android application but are very useful. The contracts are the same classes that can be found in the server code, they are used to send and receive serialised information with the socket service. The custom exceptions are used to understand the errors returned from the API and display adapted error messages. The preferences are a simple wrapper to the SharedPreferences in order to provide simply typed methods to access stored user preferences like the account, the current branch or the server IP and port.

The utils package provide different helper classes, to display generic error messages in activity, to show the main application drawer, to format and use dates according to the service, to serialise and deserialise data, to handle geolocation data or to provide a direct access to some static data like vehicle types and statuses.

The CarRentalApplication provides a simple access to the application context when it is required outside of the Activities and sets up the SSL context for the secure socket.
The views and images are stored separately and I won't describe them because I only used the default layouts (Relative, Coordinator and Linear layouts), and the default UI elements, however, it is important to note that every single displayed string has been externalised in the strings.xml file of the android application, making it easy to create local versions of the app in different languages later on.

3.2.2. Alternatives

I am not experienced with the android application development, therefore I think that I may have missed some useful features in order to make the code more reusable. I especially think about fragments, I think they can help to create reusable parts of layouts that I should have used in order to display elements like a vehicle details, which is done many times in the application.

4. Testing and configuration

Before the development of the project, an initial configuration and planning phase was made. During the configuration phase I made some technological choices about the languages and tools I was going to use and started to think about my deployment system.
The testing was not my first concern because I started the project with a prototype in order to evaluate if my initial design was implementable and to have a better understanding of the requirement. Nevertheless, I decided that I wasn't going to use automated testing on the project because of my work timeline and the fact I wasn't familiar with unit and integration testing in distributed environnements using MPI as coordinator. I decided to use the requirements list, somewhat modified with my own modifications and research during the prototyping phase.
I will go in the details of those choices and tests in this part of the report.

4.1. Configuration management

4.1.1. Code versioning

Even if I was the only person working on the source code I did wanted to be able to rollback to a previous version, that is why I decided to use git as a version control system. I choose the provider Bitbucket (https://bitbucket.org/) because it is free for personal usage with private repositories. Two separate repositories were made, one for the Android application, and another one for the distributed server.

4.1.2. Documentation

The documentation of the project is composed of two documents, the design proposal submitted as part of the coursework, and this document, containing the updated elements of the design proposal, the testing and configuration reports and a critical assessment of the product I made. I will assume that the reader of the present document has access to the initial design proposal, as it may be referred in this document.

Also, all the code is documented with comments, each file disposes of a global comment, describing the purpose of the file, and each method is provided with JavaDoc comments in order to describe the expected inputs, outputs and inner logic of the method.
The initial design document has not been updated because of its proposal nature.

4.1.3. Language and tools

One of the project's requirement was to use either C, C++ or Java as the main programming language for the project, I choose to use Java for multiple reasons:

- I am much more experienced and efficient with that language than I am with the two other ones
- The Android application (developed in Java) must communicate with the server, using a common language reduces chance of having incoherences due to different platforms (Different sockets implementations, date formatting, timestamp with and without milliseconds, etc..)
- Java offers the possibility of running the server on different platforms without platform-specific code
- Finally, I share some code between the android application and the server, this would be harder with two different languages

Aside of the language, I am using diverse tools and libraries in order to avoid reimplementing existing source code (only for the parts that are outside of the project scope). Here is a list of the libraries used for the android application and/or the server:

- Gson (Apache license, V2.0) - https://github.com/google/gson - Java library used to serialise and deserialise objects to and from the JSON notation
- BouncyCastle (MPIT License) - https://www.bouncycastle.org - Widely recognised encryption java library, used to enable AES256 encryption in Java
- ORMLite (ISC License) - http://ormlite.com/ - Object relational mapping library for SQLite in java, used to reduce the maintainability impact of SQL on high level programming languages like java
- SQLite JDBC (Apache License, V2.0) - https://bitbucket.org/xerial/sqlite-jdbc - Set of classes used to communicate with an SQLite database from Java.
- OpenMPI (BSD 2 License) - https://www.open-mpi.org - Java library for the implementation of MPI used on the HAC cluster
- Butterknife (Apache License, V2.0) - http://jakewharton.github.io/butterknife/ - Android library used to easily bind the UI elements to variables in the activities
- MaterialDrawer (Apache License, V2.0) - https://github.com/mikepenz/MaterialDrawer - Android library used to easily display the main application drawer used as main menu in my application

- SmartLocation Library (MIT License) - https://github.com/mrmans0n/smart-location-lib - Android library used to get the user position with a simple wrapper of the native android tools.

Finally, some tools were used in order to achieve the development of the project, here is the main list:

- Android Studio 2.0 (Apache license, V2.0) - http://developer.android.com/sdk/index.html - IDE used to develop the android application
- GenyMotion emulator (Free personal license) - https://www.genymotion.com - Improved android emulator used to test and demonstrate the android application
- IntelliJ IDEA (Academic license) - https://www.jetbrains.com/idea/ - IDE used to develop the distributed server

Also, as the project was developed under Mac OSX 10.11.4 and run under an UNIX system, multiple UNIX tools were used (OpenSSL, OpenSSH, ps, grep, awk, kill, scp, etc...).

All the elements used for this project are correctly licensed for an academic usage.

4.1.4. Deployment

The deployment of the project was mainly a concern for the distributed environment, then android application being automatically deployed to the testing device through the IDE, there wasn't much configuration involved.

A combination of two shell scripts are used in order to deploy the software on the server easily, the first one has been designed to compile the java files on the local machine, deploy the required elements on the server, kill the previous instance of the distributed system if it was still running and start the updated version. The second script is designed to kill the currently running system.

The deployment script, displayed in the listing 7, simply makes use of unix tools to build, deploy and run the system, we can see that mpirun specifies a host file, visible in the listing 8. The hostfile is designed to dispatch the nodes used by MPI on diverse hosts. Finally, the values 15055343 (My own student ID) and 5106 (My HAC assigned port) are to be replaced by the user working deploying the software (The current configuration involves a key-based connection system to the HAC in order to avoid typing the user password every time).

Listing 7: deploy.sh

```bash
#!/usr/bin/env bash
set -e

#build
echo -e "\033[31m Building...\033[0m"
```

```
javac -d src -cp src/mpi.jar:src/bouncycastleprov-jdk15on-154.jar:src/gson-2.6.2.jar:src/sqlite-jdbc-3.8.11.2.jar:src/ormlite-core-4.49-SNAPSHOT.jar:src/ormlite-jdbc-4.49-SNAPSHOT.jar -sourcepath src src/com/vehiclerental/Main.java

#deploy classes
echo -e "\033[31m Deploying...\033[0m"
scp -r ./out/production/Server/com/* 15055343@161.73.147.225:/home/15055343/java/bin/com/
#deploy keystore
scp -r ./carrental.keystore 15055343@161.73.147.225:/home/15055343/java/
#deploy libraries
scp -r ./src/bouncycastleprov-jdk15on-154.jar 15055343@161.73.147.225:/home/15055343/java/bin/
scp -r ./src/gson-2.6.2.jar 15055343@161.73.147.225:/home/15055343/java/bin/
scp -r ./src/sqlite-jdbc-3.7.2.jar 15055343@161.73.147.225:/home/15055343/java/bin/
scp -r ./src/ormlite-core-4.49-SNAPSHOT.jar 15055343@161.73.147.225:/home/15055343/java/bin/
scp -r ./src/ormlite-jdbc-4.49-SNAPSHOT.jar 15055343@161.73.147.225:/home/15055343/java/bin/

#running
echo -e "\033[31m Running...\033[0m"
ssh 15055343@161.73.147.225 'cd /home/15055343/java && ./killjavampi.sh' || true
ssh 15055343@161.73.147.225 'cd /home/15055343/java && /shared/openmpi/bin/mpirun -n 5 --hostfile hostfile java -classpath bin:bin/bouncycastleprov-jdk15on-154.jar:bin/gson-2.6.2.jar:bin/sqlite-jdbc-3.7.2.jar:bin/ormlite-core-4.49-SNAPSHOT.jar:bin/ormlite-jdbc-4.49-SNAPSHOT.jar com.vehiclerental.Main 5106'
```

Listing 8: hostfile

```
node01 slots=1
node02 slots=1
node03 slots=1
node04 slots=1
node05 slots=1
```

The second file, the kill script (visible in listing 9), is much smaller than the deployment one, it simply finds the current process using mpirun with the given user id (15055343) using PS and Grep, and then return the process ID using awk and print.

Finally, it kills the process using the kill command. It is interesting to note that the commands are executed two times, I had a problem when only killing a single time, it looks like MPI keeps two instances running on the head node. (Killing on the main node kills on the child nodes).

Listing 9: kill.sh

```
ID=`ps aux | grep '15055343' | grep
    ↪ '[/]shared/openmpi/bin/mpirun' -m 1 | awk '{print
    ↪ $2}'`

kill -9 $ID

ID=`ps aux | grep '15055343' | grep
    ↪ '[/]shared/openmpi/bin/mpirun' -m 1 | awk '{print
    ↪ $2}'`

kill -9 $ID
```

4.2. TESTING PLAN AND RESULTS

The testing plan has been deduced from the requirements, a list of points was defined, using the following syntax: A **<user>** can (or can not) achieve a **<business value>**. Using this system helped me to produce a decent initial list of tests, introduced below.
Some other testing points were integrated to handle the remaining requirements that couldn't fit into the first syntax, or the non functional ones. The tests list is detailed here along with the results :

1. **A guest, user or staff member can set the connection settings**
   Anyone should be able to modify the server IP and Port on the application.

   Results: After testing, I can see that any guest, user or staff member can access the interface displayed in figure 15. It is either accessible from the login form, by clicking the "Change connection parameters" button (figure 13) or from the application drawer (which can be opened by swiping from the left side of the screen to the right) by clicking on the "Change server settings" link (figure 14).

   If the user wants to modify those settings, a request will be sent to the server in order to validate the new configuration, if it fails, the change will be prevented.

2. **A guest, user or staff member can choose a branch**
   Anyone should be able to pick a new branch in order to use the system.

   Results: After testing, I can see that any guest, user or staff member can access the interface displayed in figure 16. It is either accessible from the login form, by logging in, registering or browsing vehicles (figure 13) or from the application drawer by clicking on the "Change branch (<current branch>)"

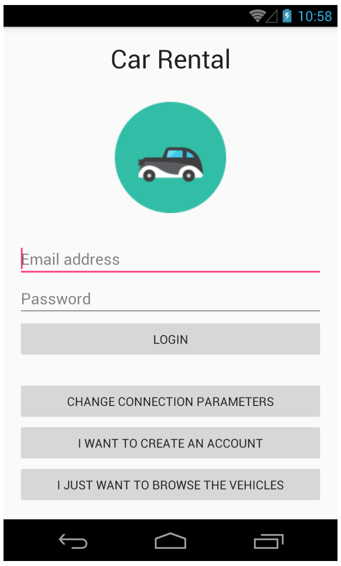

10:58
Car Rental
Email address
Password
LOGIN
CHANGE CONNECTION PARAMETERS
I WANT TO CREATE AN ACCOUNT
I JUST WANT TO BROWSE THE VEHICLES

Figure 13: Login view.

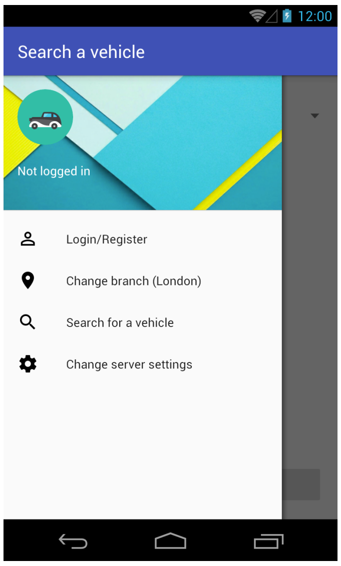
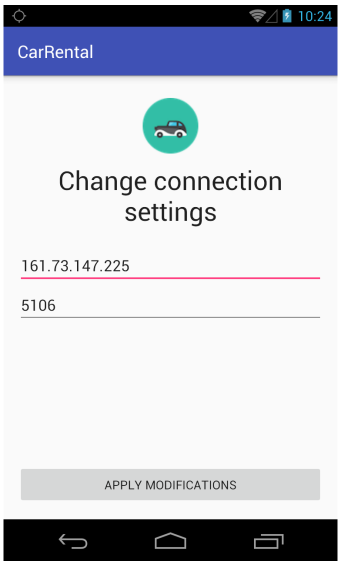


Figure 14: Guest drawer.



161.73.147.225

5106

Figure 15: Change server settings.

link (figure 14).

The user is proposed a text filterable list of the available station, and also the nearest branch if the user can be localised. Selecting a branch will bring the guest to the available vehicles search form, the user to his bookings, and the staff member to the branch bookings list.

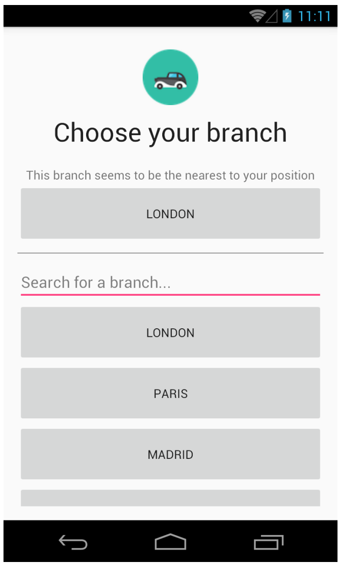
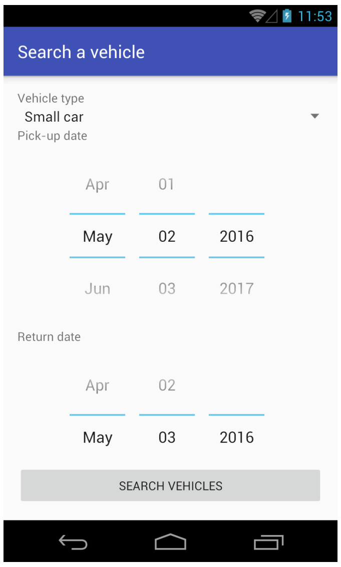


Figure 16: Change current branch.

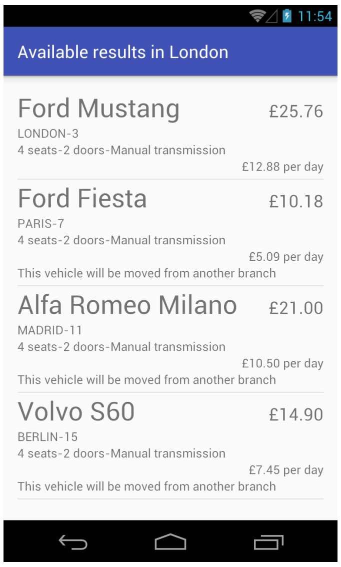


Small car

Apr 01
May 02 2016
Jun 03 2017

Apr 02
May 03 2016

Figure 17: Search vehicles form.



Ford Mustang £25.76
LONDON-3
4 seats-2 doors-Manual transmission
£12.88 per day

Ford Fiesta £10.18
PARIS-7
4 seats-2 doors-Manual transmission
£5.09 per day
This vehicle will be moved from another branch

Alfa Romeo Milano £21.00
MADRID-11
4 seats-2 doors-Manual transmission
£10.50 per day
This vehicle will be moved from another branch

Volvo S60 £14.90
BERLIN-15
4 seats-2 doors-Manual transmission
£7.45 per day
This vehicle will be moved from another branch

Figure 18: Search vehicles results.

3. **A guest or an user can search available vehicles**
   The search available vehicles features is accessible to the guests and users in order to see what they can rent.

Results: After testing, I can see that both guests and users have access to the interface visible in figure 17. Guests are directly redirected here after their branch choice, users and guests can access this page from the user drawer (figure 14) through the "Search for a vehicle" link.
When performing the search, errors will be displayed if the dates are invalid or if there is no results. Otherwise the available vehicles list will be displayed as seen in figure 18.

4. **A guest can create an account on the system**
   Every guest should be able to create an account in order to book a vehicle. He must provide informations and an unique email to do so.

   Results: After testing, I was proposed an account creation form as shown on figure 19. Once the form was filled and the button "Create account" pressed, the account has been created if there was no errors. Otherwise, an error message was displayed to guide the user. After the account creation, the user is redirected to the branch selection view.

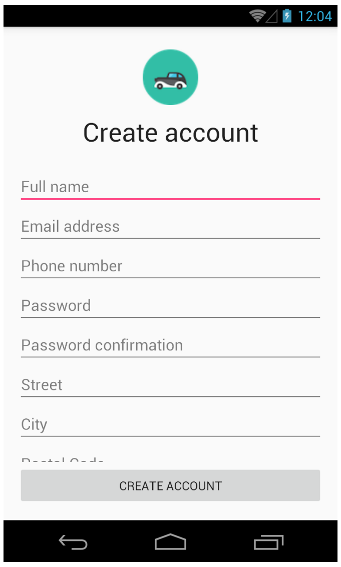


Figure 19: Register form.

12:04

Create account

Full name

Email address

Phone number

Password

Password confirmation

Street

City

CREATE ACCOUNT

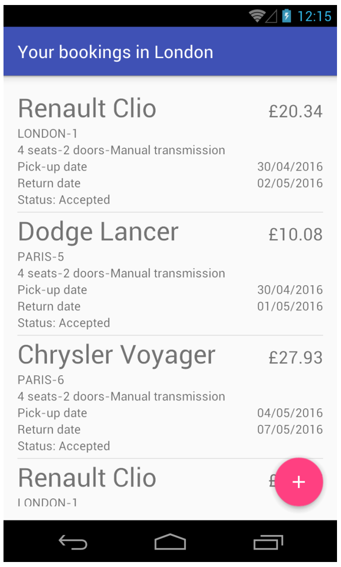


Figure 20: User bookings list.

12:15

Your bookings in London

Renault Clio £20.34
LONDON-1
4 seats-2 doors-Manual transmission
Pick-up date 30/04/2016
Return date 02/05/2016
Status: Accepted

Dodge Lancer £10.08
PARIS-5
4 seats-2 doors-Manual transmission
Pick-up date 30/04/2016
Return date 01/05/2016
Status: Accepted

Chrysler Voyager £27.93
PARIS-6
4 seats-2 doors-Manual transmission
Pick-up date 04/05/2016
Return date 07/05/2016
Status: Accepted

Renault Clio
LONDON-1

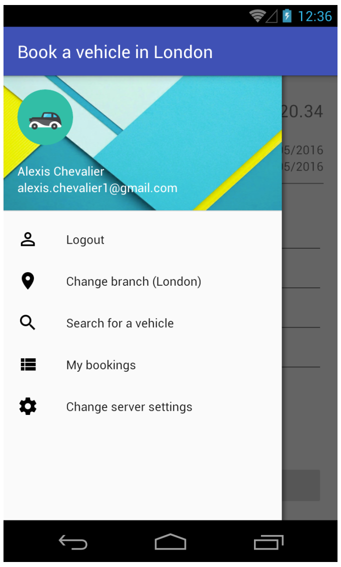


Figure 21: User drawer.

12:36

Book a vehicle in London

Alexis Chevalier
alexis.chevalier1@gmail.com

Logout

Change branch (London)

Search for a vehicle

My bookings

Change server settings

5. **A guest can login to his account in the system**
   Every guest should be able to login into the system using an existing account (user or staff account).

   Results: After testing, I can see that the login form is simple and easy to use (see figure 13). Once the form was filled and the "Login" button clicked, the user is redirected to the branch selection view if the account credentials are validated by the system. If not, an helpful error message is displayed.

6. **An user can see his bookings for the current branch**
   When the user connects to the server, he should see his booking list for the current branch.

Results: After testing, I can see that the booking list (figure 20) is displayed after the branch choice. A button on the bottom left part of the screen will bring the user to the search available vehicle view (figure 17) if he needs to make another booking. Also, as seen in the figure 21, it is possible to go back to the bookings view by clicking the "My bookings" link in the user drawer.

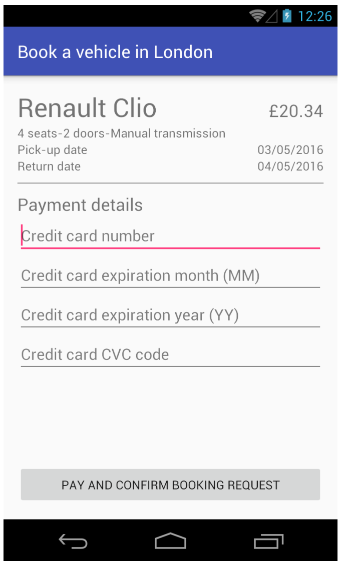
7. **An user can book an available vehicle**
   An user should be able to create a booking for an available vehicle after after searching for the vehicle.

   Results: After testing, I can see that when the user clicks on a vehicle search result, he is redirected to the create booking view (figure 22). The view displays a small summary of the requested booking and invites the user to type its payment informations. Once the form has been filled and the button "Pay and confirm booking request" has been clicked, the user is redirected to his booking list with a confirmation message, also, the new booking has been added to the list.
   If the information is invalid or if the vehicle is not available anymore, an error message will be displayed.

8. **An user can not book a busy vehicle**
   An user should not see (and should not be able to book) a vehicle considered as busy for the time period specified by the user.

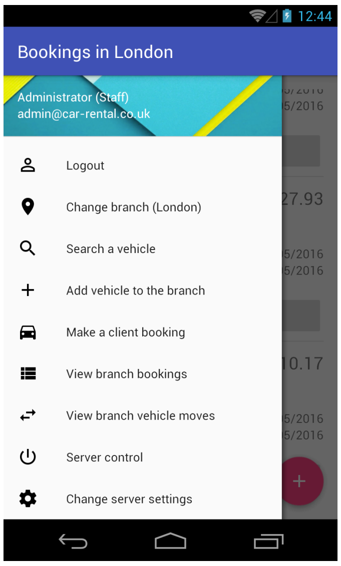
   Results: After testing, I can see that the search available vehicle feature doesn't show the vehicle previously booked for the same dates anymore (There is no figures for this test because the different vehicles are not clearly identifiable). It is therefore not possible to book an unavailable vehicle.

Figure 22: Book available vehicle.



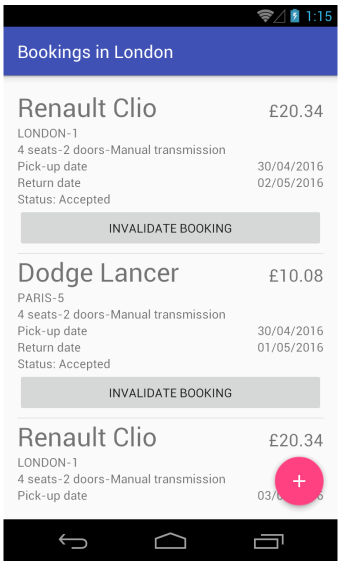
12:26
Book a vehicle in London
Renault Clio £20.34
4 seats-2 doors-Manual transmission
Pick-up date 03/05/2016
Return date 04/05/2016
Payment details
Credit card number
Credit card expiration month (MM)
Credit card expiration year (YY)
Credit card CVC code
PAY AND CONFIRM BOOKING REQUEST

Figure 23: Staff drawer.



12:44
Bookings in London
Administrator (Staff)
admin@car-rental.co.uk
Logout
Change branch (London)
Search a vehicle
Add vehicle to the branch
Make a client booking
View branch bookings
View branch vehicle moves
Server control
Change server settings

Figure 24: Branch bookings list.



1:15
Bookings in London
Renault Clio £20.34
LONDON-1
4 seats-2 doors-Manual transmission
Pick-up date 30/04/2016
Return date 02/05/2016
Status: Accepted
INVALIDATE BOOKING
Dodge Lancer £10.08
PARIS-5
4 seats-2 doors-Manual transmission
Pick-up date 30/04/2016
Return date 01/05/2016
Status: Accepted
INVALIDATE BOOKING
Renault Clio £20.34
LONDON-1
4 seats-2 doors-Manual transmission
Pick-up date 03/0

9. **An user or staff member can log out of the system**
   Once logged in and after choosing his branch, an user or staff member should

be able to log out of the system.

Results: After testing, I can see that the user drawer (figure 23) displays a "Logout" link. When this link is clicked, the user is redirected to the login view, and all the previous session settings are removed (email, password and current branch).

10. **A staff member can see all the bookings for a branch**
A staff member should be able to quickly see the current bookings in a specific branch and validate or invalidate them.

    Results: After testing, I can see that the first view visible by the staff member once he choose his branch is the booking list visible in figure 24. Each displayed booking proposes a button to either validate or invalidate a booking, depending on its current status. Once the button clicked, the request will be sent to the server and will refresh the list. If the vehicle is not available anymore because another booking has been made, an error message will be displayed.
The view also proposes a shortcut button in the bottom left of the screen to allow the staff member to quickly access the "Create booking for user" view.

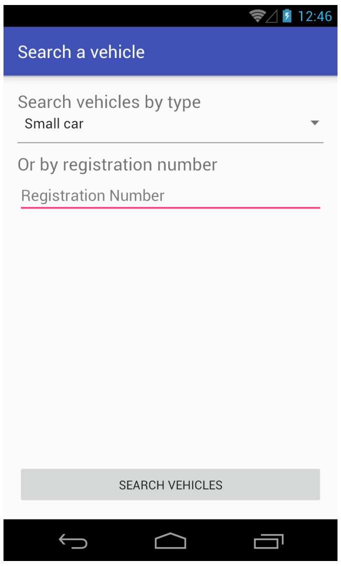
11. **A staff member can search vehicles with a registration number or vehicle type**
In order to check a vehicle status or to change it, a staff member should be able to search for a vehicle.

    Results: After testing, I can see that the staff drawer (figure 23) provides a link "Search a vehicle" which once clicked, brings to a search interface (figure 25), inviting the user to select a vehicle type or to type the exact registration number of the vehicle.
If the user just picks a vehicle type, and clicks the "Search vehicles" button, he will be redirected to a result list (figure 26), clicking on one of the results will redirect him to the update/summary view visible in figure 27.
If the user decides to search a vehicle by its registration number and that he clicks the "Search vehicles" button, he will be redirected to the same update/-summary view (figure 27), if the registration number is not in the database, he will receive an error message.

12. **A staff member can see and change a vehicle status**
In order to maintain the day to day business operations, a staff member should be able to get details about a vehicle and updating its status.

    Results: After testing, I can see that the view proposed in figure 27 provides specific details about the previously searched vehicle, and also allows the staff member to update the vehicle. If the staff member changes the status and clicks on the "Update vehicle" button, the vehicle will be updated and the user redirected to the branch bookings list. If there is an error in the updated vehicle, an error message will be displayed.

Figure 25: Search vehicle.


12:46

Search a vehicle

Search vehicles by type

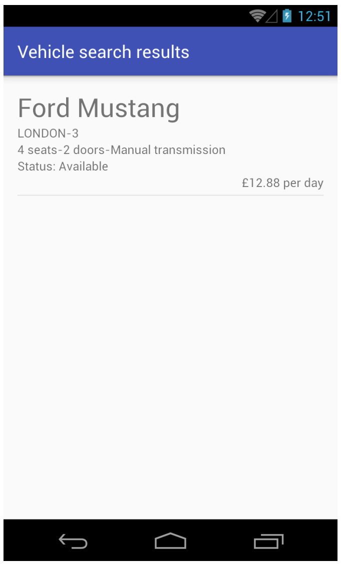
Small car

Or by registration number

Registration Number

SEARCH VEHICLES

Figure 26: Search vehicle results.


12:51

Vehicle search results

Ford Mustang

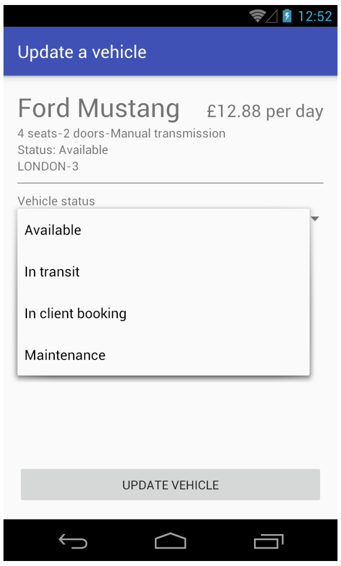
LONDON-3

4 seats-2 doors-Manual transmission

Status: Available

£12.88 per day

Figure 27: Vehicle update details.


12:52

Update a vehicle

Ford Mustang £12.88 per day

4 seats-2 doors-Manual transmission

Status: Available

LONDON-3

Vehicle status

Available

In transit

In client booking

Maintenance

UPDATE VEHICLE

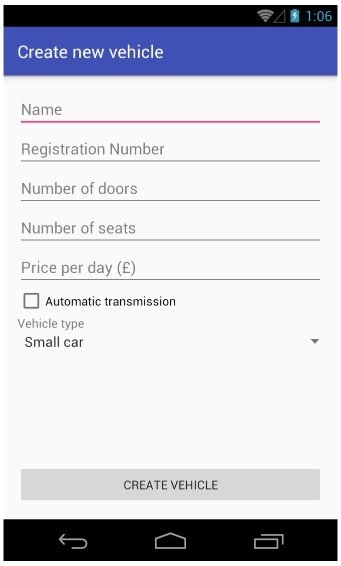
13. **A staff member can add a vehicle**
    In order to maintain the branch vehicle park, a staff member should be able to create a new vehicle.

    Results: After testing, I can see that a "Create vehicle" link is accessible from the staff drawer (figure 23). This links opens a new view as seen in figure 28, which invites the staff member to type the new vehicle details. Once the form has been filled and the "Create vehicle" button clicked, the new vehicle is added to the branch, and the user is redirected to the branch booking list with a confirmation message.
    If the provided information is invalid (duplicate registration number for example), an error message is displayed.

14. **A staff member can make a booking for an user**
    If a client comes to a physical branch, a staff member should be able to create a booking for the client from the android application.

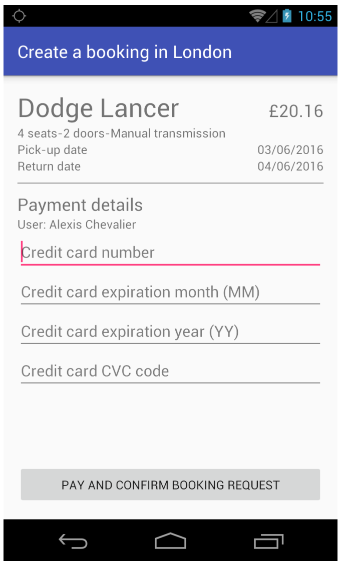
    Results: After testing, I can see that the staff member drawer (Figure 23) provides a link "Make a client booking", when the link is clicked, the user is redirected to the search available vehicles view (Figure 17). As explained in the test related to this part, the staff user is then able to search a vehicle for the given criteria. When he selects a vehicle, the system asks him to choose or create a new user (Tested separately in the next point).
    When the staff user choose an user, a summary of the booking is displayed (Figure 29) and he is invited to type the user payment details (or to let the user type them).
    When the "Pay and confirm booking request" is clicked, the information is submitted to the server, the staff user is redirected to the branch bookings list (which now contains the new booking) with a confirmation message. If an error occurs (vehicle not available anymore or invalid informations), an error mes-

Figure 28: Create vehicle.



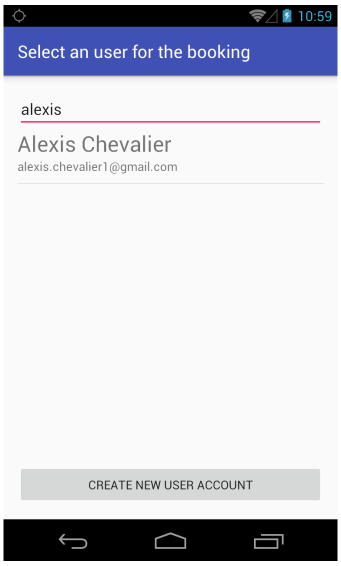
Name
Registration Number
Number of doors
Number of seats
Price per day (£)
Automatic transmission
Vehicle type
Small car
CREATE VEHICLE

Figure 29: Create booking for user.



Dodge Lancer £20.16
4 seats-2 doors-Manual transmission
Pick-up date 03/06/2016
Return date 04/06/2016
Payment details
User: Alexis Chevalier
Credit card number
Credit card expiration month (MM)
Credit card expiration year (YY)
Credit card CVC code
PAY AND CONFIRM BOOKING REQUEST

Figure 30: Choose user for booking.



alexis
Alexis Chevalier
alexis.chevalier1@gmail.com
CREATE NEW USER ACCOUNT

sage is displayed.

15. **A staff member can search an user**
In order to create a booking for another user, a staff member should be able to pick an existing user.

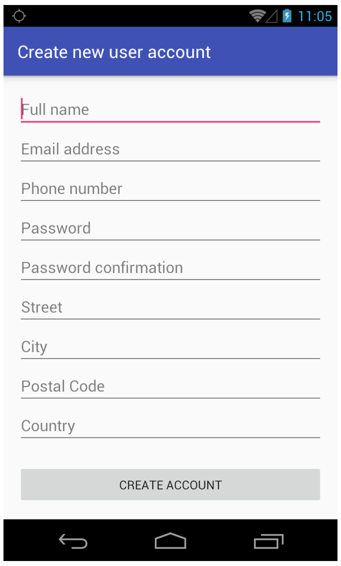
Results: After testing, I can see that during the staff booking creation procedure, the staff user is invited to select a user for the booking (he can also create a new account, but this will be tested in the next point). The search user view is represented in figure 30, it propose a text field that the staff user should fill with a part of the user name or email address (at least two characters), when the field is filled, a search request is sent to the server and displays the list of users as a result (an error message is displayed if nobody was found).
A click on one of the users brings the staff user to the booking confirmation page for this user.

16. **A staff member can create an user**
In order to create a booking for another user, a staff member should be able to create a new user account.

Results: After testing, I can see that the "pick user" view (Figure 30) discussed previously also offers a button "Create new user account". When the button is clicked, a new form is displayed (Figure 31). This form should be filled with the new user details. Once the form is filled, the staff user clicks the "Create account" button in order to continue to the booking confirmation page. If there is an error with the user creation (Existing email address for example), an error message is displayed.)

17. **A staff member can see the incoming/outgoing vehicles for a branch**
In order to facilitate the daily operations, a staff member should be able to

Figure 31: Create user account.


11:05

Create new user account

Full name

Email address

Phone number

Password

Password confirmation

Street

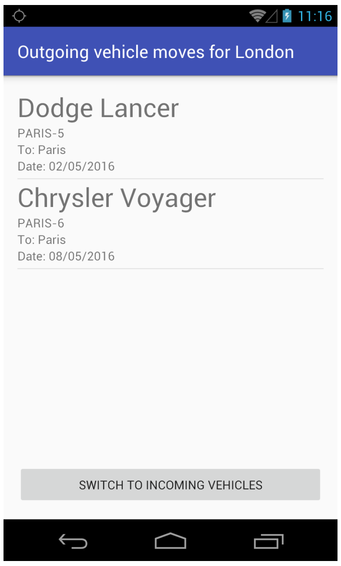
City

Postal Code

Country

CREATE ACCOUNT

Figure 32: Branch vehicles moves.


11:16

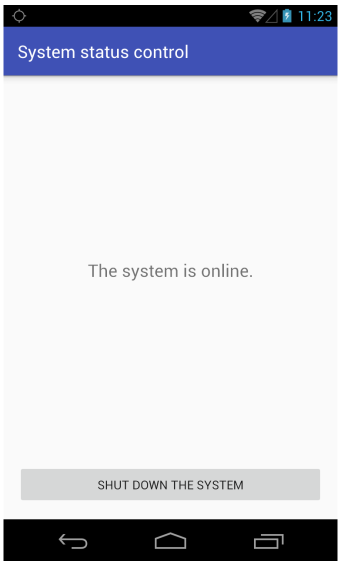
Outgoing vehicle moves for London

Dodge Lancer
PARIS-5
To: Paris
Date: 02/05/2016

Chrysler Voyager
PARIS-6
To: Paris
Date: 08/05/2016

SWITCH TO INCOMING VEHICLES

Figure 33: System shutdown.


11:23

System status control

The system is online.

SHUT DOWN THE SYSTEM

display the expected vehicle moves for the current branch.

Results: After testing, I can see that the staff member drawer (Figure 23) offers a link "Show branch vehicle moves". When this link is clicked it brings the staff user to a view (view figure 32) composed of a list displaying incoming or outgoing vehicles moves for the current branch. It is possible to switch between the incoming and outgoing moves by clicking on the corresponding button at the bottom of the view. Only the future (today included) moves are displayed in the interface.

18. **A staff member can shutdown the server**
    In order to close the system, a staff member should be able to close the whole system from the android application.

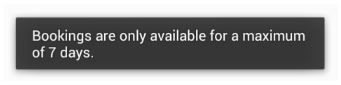
    Results: After testing, I can see that the staff member drawer (Figure 23) offers a link "Server control". Clicking on the link brings the staff user to the system control view as shown in figure 33. The staff user can see the current status of the system and can access to the "Shut down the system" button. If he clicks on the button, the system will be completely stopped and will require a manual restart.

19. **The interface must be usable**
    The interface should not be particularly well designed but should provide a simple and easy access to all the provided features.

    Results: The testing of all the available features (points 1 to 19) shows that all the interface is coherent and provide an easy way to access to all the available features through the application drawer. Also, most of the proposed controls are native to the android application. Finally, the UI mockups provided in the design report have been slightly modified but were generally respected.

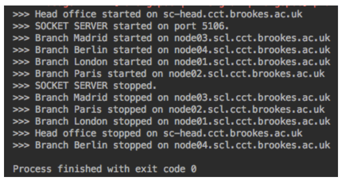
20. **Bookings should not be made for more than seven days**

    Results: A tentative was made to create a booking for 8 days and a month, both tentatives failed with the error message displayed in figure 34. It is not possible to create a booking for more than seven days.



Bookings are only available for a maximum of 7 days.

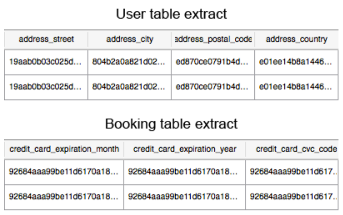
Figure 34: Invalid booking duration error.



```
>>> Head office started on sc-head.cct.brookes.ac.uk
>>> SOCKET SERVER started on port 5106.
>>> Branch Madrid started on node03.scl.cct.brookes.ac.uk
>>> Branch Berlin started on node04.scl.cct.brookes.ac.uk
>>> Branch London started on node01.scl.cct.brookes.ac.uk
>>> Branch Paris started on node02.scl.cct.brookes.ac.uk
>>> SOCKET SERVER stopped.
>>> Branch Madrid stopped on node03.scl.cct.brookes.ac.uk
>>> Branch Paris stopped on node02.scl.cct.brookes.ac.uk
>>> Branch London stopped on node01.scl.cct.brookes.ac.uk
>>> Head office stopped on sc-head.cct.brookes.ac.uk
>>> Branch Berlin stopped on node04.scl.cct.brookes.ac.uk

Process finished with exit code 0
```

Figure 35: Branch dispatching on nodes.



User table extract

address_street	address_city	address_postal_code	address_country
19aab0b03c025d...	804b2a0a821d02...	ed870ce0791b4d...	e01ee14b8a1446...
19aab0b03c025d...	804b2a0a821d02...	ed870ce0791b4d...	e01ee14b8a1446...

Booking table extract

credit_card_expiration_month	credit_card_expiration_year	credit_card_cvc_code
92684aaa99be11d6170a18...	92684aaa99be11d6170a18...	92684aaa99be11d617...
92684aaa99be11d6170a18...	92684aaa99be11d6170a18...	92684aaa99be11d617...

Figure 36: Encrypted data.

21. **The server should persist the business data**
    The data used in the system should persist after a complete system restart.

    Results: After testing, when the system is shut down and then manually restarted, all the data was exactly the same after the restart.

22. **The server must authenticate request and correctly authorise them**

    Results: This test is not really achievable without modifying the android application code because the system prevents by design an user to access an unauthorised feature. However, modifying the application allowed me to try accessing staff member features with a regular user account, an error message related to the authorisation was displayed every time and the feature was not usable.

23. **Every office must run on different cluster nodes as separate processes**

    Results: This is another test which requires higher access level than regular user or staff member. I added some logging features to the server code and tried to run it (and then stop it), the result is visible on figure 35. We can see that every branch runs on a separate node.

24. **The users should always contact the head office, which will handle the cluster communication**

    Results: After testing, it is clear that only as single IP and a single port was given through the application, therefore there is no way that the android application directly contacts another branch before contacting he head office.

25. **The branches should always handle their own vehicles and booking details**

    Results: This requires a code analysis. This has been detailed in the design report of this document and correctly implemented.

There is only a single case when a branch A creates a booking for another branch B, when the booking uses the vehicle from the branch B, in order to avoid latency between the confirmation and the booking creation.

26. **The bookings can be made with vehicles on other branch, but the vehicle will have to be moved to the booking branch and back**
    The vehicles can be borrowed from other branches, but it requires one day to bring the vehicle and one more day to return the vehicle.

    Results: After testing, when creating a booking with a foreign vehicle, the staff members can clearly see that there is a new outgoing vehicle move for the vehicle branch, and also an new incoming vehicle move for the return. Also, it is impossible to book the vehicle for one day before and after the foreign booking.

27. **Each branch should be named and correctly dispatched on the nodes**

    Results: As seen in figure 35, every branch is dispatched on a specific nodes, each node is identified by its numerical identifier and then bound to the corresponding branch ID.

28. **The shutdown facility should terminate the whole server**

    Results: After testing, we can see on figure 35 that when the shutdown facility is used, every node terminates correctly and that the whole program terminates with the exit code 0 (witch is the "No error" code). The server has cleanly exited and can be manually restarted.

29. **The critical information needs to be encrypted on the nodes**

    Results: All the user informations (except the full name and email address for indexing reasons) are encrypted in the database, as the payment informations of the bookings, the corresponding rows are visible in a database extract in figure 36.
    It is also relevant to note that the network traffic between the android application and the server is encrypted (and so is the traffic between the branch nodes using MPI).

30. **The server code should use MPI as message passing middleware**

    Results: This requires a code analysis, but the design report of this document explains how MPI is used to distribute the messages between the nodes.

31. **The server code should be implemented in C, C++ or Java**

    Results: This requires a code analysis, the code of our server has been implemented using Java (and SQL for the database purposes).

32. **The code should be maintainable, extensible and well documented**

Results: The code has been entirely commented, and the design report of this document explains how I designed the software with extensibility and maintainability in mind.

33. **The code should be easy to understand**

    Results: The code has been using oriented object features such as inheritance and interface as well as other best practices to make the design light and with a low couplage, which results in an easily understandable code. The design report will give more details about this point.

5. Critical assessment

5.1. Critical analysis of my implementation

Globally speaking, I am satisfied of my implementation, I think that I was able to produce a working solution, which respects the requirements and with a well structured and maintainable source code. However, I am disappointed not to have had taken the time to implement the improved design detailed in this report, I would have really liked to see it working.

The current implementation proposes a strong and evolutive design as I explained in this report, modifications can be easily integrated to the system thanks to the low coupling between the different layers. The security is also deeply integrated into the system, all the network communication are encrypted, from the Android application to the MPI nodes inside the cluster. The safety critical database fields are also encrypted before being written to the SQLite file.

The performance was also an important point, the webservice was designed with lightness in mind, and most of the system operations are executed in a linear time (depending on the existing amount of data concerned by the operation).

The scalability of the system however, is not really good, as explained before, the system can only handle a single request at a time, which will cause problems if the business grows quickly.

Also, external systems may find it difficult to interact with my implementation as the traffic uses an encrypted socket, when the most used standard is HTTPS.
Having used an AMQP implementation before, I find MPI inadequate for the current problem, its API doesn't provide enough abstraction in order to be used in a transparent way in a program. It clearly mets the requirements, which are being able to distribute a program between multiple remote nodes and provide communication interfaces between them, but it also provides many other features about processes synchronisation which are useless in our case. AMQP only provides a communication layer, which means that another solution would be required to start the different nodes (but that's not very hard with shell scripts), but the communication layer provides great distributed communication patterns which includes load balancing, with a decent level of abstraction which makes it really easier to integrate in a

program such as the one in this project.

5.2. Usage as a commercial application

In a commercial environnement, specific criteria will have to be taken into account. The first one would be the security, a business doesn't want to leak its consumer's data in an attack, the system does actually well in terms of security thanks to the different encryption levels (traffic and storage).

In my mind, the second one is the scalability of a system, it is really important to be able to handle the growth of a business at any time using vertical (Increase server performance) or horizontal (Increase number of servers) scalability, my system only supports vertical scalability because it would require a design change in order to have multiple files per system (or even per branch). Also, the fact that the system only supports a single request at a time is a bad point for scalability too.

Another criteria would be the ability to integrate other systems (For example an hotel booking system or traffic informations) and to let other systems integrate ours (For example an airline company which would want to offer the ability to rent a vehicle after booking a plane). In that case the system is not really ready to open its API, as explained before, an encrypted socket is not the standard, but the design makes it easy to switch to an HTTPS server using a REST API (which is a far more used standard in the industry).

There is many different criteria, but I think that those three points are the most relevant for this system, and they show that the system is more intended as a working prototype of proof of concept, but should be adapted in order to be used as a commercial solution.

Appendices

A. Search available vehicles SQL query

```sql
SELECT * FROM `vehicles` ve WHERE (
  `ve`.`branch` = 1 AND `ve`.`type` = 0
  AND NOT EXISTS (
    SELECT * FROM `bookings` bo
      LEFT JOIN `vehicle_moves` vm
      ON `vm`.`id` = `bo`.`vehicle_move_id`
    WHERE (
      `bo`.`vehicle_id` = `ve`.`id` AND
         ↪ `bo`.`booking_validated` = 1 AND
         ↪ `ve`.`status` != 3 AND (
        (  -- This statement checks that the new booking
           ↪ doesn't overlap with an exising booking in
           ↪ the middle of the desired dates
          IFNULL(`vm`.`vehicle_move_date`,
             ↪ `bo`.`start_day_timestamp`) <
             ↪ 1461931200000
          AND
          IFNULL(`vm`.`vehicle_return_date`,
             ↪ `bo`.`end_day_timestamp`) > 1462104000000
        )
        OR
        (  -- This statement checks that the new booking
           ↪ doesn't start before an existing booking
           ↪ and finish during this existing booking
          IFNULL(`vm`.`vehicle_move_date`,
             ↪ `bo`.`start_day_timestamp`) <
             ↪ 1461931200000
          AND
          IFNULL(`vm`.`vehicle_return_date`,
             ↪ `bo`.`end_day_timestamp`) <= 1462104000000
          AND
          IFNULL(`vm`.`vehicle_return_date`,
             ↪ `bo`.`end_day_timestamp`) >= 1461931200000
        )
        OR
        (   -- This statement checks that the new booking
           ↪ doesn't start during an existing booking
           ↪ and finish after this existing booking
          IFNULL(`vm`.`vehicle_move_date`,
             ↪ `bo`.`start_day_timestamp`) >=
             ↪ 1461931200000
          AND
          IFNULL(`vm`.`vehicle_return_date`, `bo`.`end_day_timestamp`) > 1462104000000
          AND
          IFNULL(`vm`.`vehicle_move_date`, `bo`.`start_day_timestamp`) <= 1462104000000
        )
        OR
        (  -- This statement checks that the new booking isn't in the middle of an existing booking
          IFNULL(`vm`.`vehicle_move_date`, `bo`.`start_day_timestamp`) >= 1461931200000
          AND
          IFNULL(`vm`.`vehicle_return_date`, `bo`.`end_day_timestamp`) <= 1462104000000
        )
      )
    )
  )
)
```