# Design and development of a car sharing application through a publish subscribe middleware

Semester Project

Scalable Computing Systems Laboratory


***Supervisor:***
Rafael Pires

***Professor:***
Anne-Marie Kermarrec

***Author:***
Ana Moreno Martinez

***Laboratory:***
Scalable Computing Systems Laboratory


June 11, 2021

# Contents

# 1 | Introduction

During the last decade, there has been a tremendous growth in platforms whose main purpose is to connect service providers and consumers with matching interests.

Apart from food delivery, skill sharing or lodging rental applications, this sharing economy business has also unveiled a new way of commuting: car sharing, an affordable and convenient alternative to conventional transportation. With it, traffic is reduced, thus decreasing environmental impact, users pay for the real use of the vehicle and there is a more efficient use of energy and resources.

According to the last mobility survey conducted by the EPFL in 2019 [1], 30% of the staff members that answered commuted by car on a daily basis, and 77% of the times the driver was alone. Nonetheless, one quarter of the drivers also stated that they would be willing to try carpooling.

Considering this, the aim of this project is to develop a car sharing mobile application targeted for the EPFL community when commuting to/from the campus. For this purpose we will use Apache Kafka [2], a system based on the producer and consumer model, that will manage the matching between client requests and available riders. In this model, producers send data to a topic and consumers subscribed to those topics receive the data. Publish/subscribe systems do not necessarily have a centralized server, therefore obviating the need of having a single company acting as intermediary.

In our Android application, drivers announce their route, departure time and date of the ride, which are stored as subscriptions for a future matching. Meanwhile, users publish their desired route, calendar and time restrictions, obtaining the riders that best match their interests. Once the user chooses the ride that fits him best, the rider will receive a notification to seal the deal.

[1] https://www.epfl.ch/campus/mobility/wp-content/uploads/2020/01/2019-Enquete-Rapport-Final-1.pdf
[2] https://kafka.apache.org/

# 2 | Architecture

In this section, the overall architecture of the application will be described, as well as the technologies used for its development.

## 2.1 Enabling Technologies

The technologies used for the construction of the project are:

- **Apache Kafka**: Distributed platform that uses the publish/subscribe message pattern to interact with real-time event drive applications, processing and storing their data. It is scalable, fault-tolerant and allows a high number of users to use it concurrently with an overall neglectable lag in its performance (will be evaluated in further steps).
- **Zookeeper**[1]: Open-source server in charge of the management of the brokers in the cluster and the scalability of the system.
- **Kafka Streams**[2]: Apache Kafka library used to process event-time stream applications. Input is pulled from one or more topics, transforming it to an output stream.
- **Spring Boot**[3]: Framework that simplifies the development of Java-based applications, due the multiple tools that it incorporates and the automatic configuration that it brings.
- **Apache Maven**[4]: Build automation tool based on the concept of a project object model (POM). Chosen over Gradle due to a better integration with Spring framework.
- **Android Studio**[5]: Integrated development environment for Google's Android operating system.
- **Firebase**[6]: Google's platform used to provide cloud services for developers. In our platform, we will use it to authenticate users with their email and password, both registering and logging them in.

[1] https://zookeeper.apache.org/
[2] https://kafka.apache.org/documentation/streams/
[3] https://spring.io/projects/spring-boot
[4] https://maven.apache.org/
[5] https://developer.android.com/
[6] https://firebase.google.com/

## 2.2 System Model

As depicted in Fig. 2.1, there are two main components in our system: server program and client program.

Due to the difficulties when trying to integrate Apache Kafka directly with Android, producers are meant to be used on the server-side, the server program has been constructed with Spring Boot. With it, we are able to create exposed RESTful end points that can be called directly from our Android application. Messages are published from the user's device to this system, and are later handler by the producer.

The workflow of the application is designed as follows:

1. First, the user is authenticated with Android's Firebase system. The user will introduce their email and password and via callback, Firebase will return the unique identifier that defines the user, authorizing the access to the platform.

2. Once the user has navigated through the screens, and has decided the location, date and hour that he desires, a HTTP POST request will be sent to the backend, with all the data needed to perform the match.

3. In the server, programs are simultaneously running. Once the data is received from the device, the producer API will push the records into either the `users` or `riders` topic within the Kafka cluster.

4. Meanwhile, the streams API will be processing both topics as they come, outputting a JSON message containing all the UUID of the possible riders for each user petition.

5. From the client, GET petitions will be made to the Consumer API controller, that will consume records from the `matches` topic and will forward them to the Android application.

6. In the application, we will start a thread to get the records sent from the consumer in the server-side. Once we have the record from our user, all the possible riders will be displayed on screen.

7. The user will then decide the ride that fits him best, sending again a request to produce a record into the topic `finalmatch`, where all the final rider-user couples will be stored.

8. Finally, a notification will be send to the rider to update him on his ride situation. Furthermore, both of them will be able to see their upcoming rides at any time in one of application's window.

Implementation details of each of these parts will be explained more in depth in the next sections.

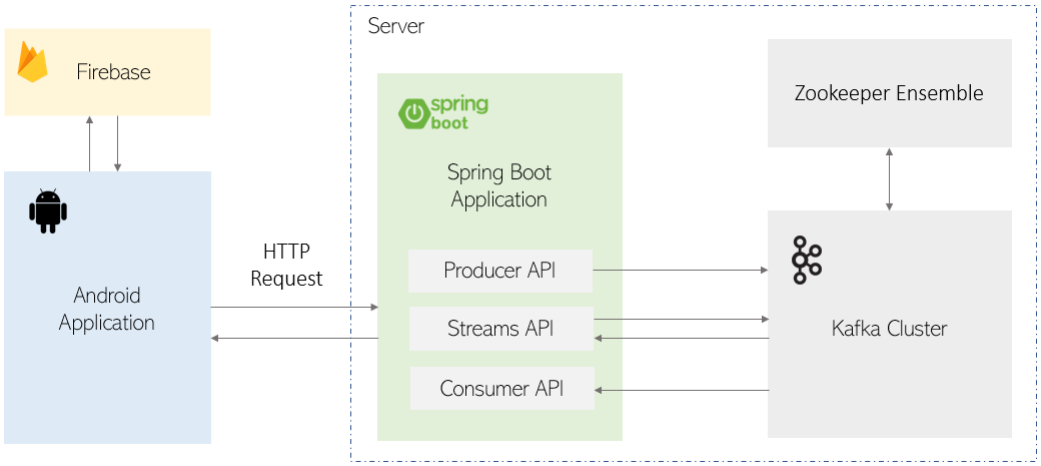


Figure 2.1: Architecture diagram of the application.

In order to develop this system, the Apache Kafka environment has been installed in the server, and topics have been created with the shell command. An uber-jar containing the Spring Boot project with all dependencies is located and running in the server.

# 3 | Server-side

As previously mentioned, the program located in the server has been developed using the Spring framework. Its main purpose is to receive petitions from the user through its RestControllers and sending the data to the cluster through the Kafka APIs.

Within this Kafka cluster, records are organized in topics. These are categories in which all the published messages are stored, splitted into partitions across multiple brokers for parallelism. There are four topics defined in our cluster. Their names and functions are presented in the following table.

| Topic | Function |
|---|---|
| *users* | List of users looking for a ride |
| *riders* | List of drivers offering a ride |
| *matches* | List of possible riders to each user |
| *finalmatch* | List of the final match rider-user |

Table 3.1: Topic names and function.

In order to construct this application, three Kafka interfaces have been used: Producer, Consumer and Streams API. Below, we will explain their configuration and deployment.

## 3.1 Producer API

The Producer API allows the application to send streams of data to topics in the Kafka brokers.

For this purpose, we employ Spring's `ProducerFactory`. This method allows the creation of Kafka Producers instances. It has a default close timeout duration of 30 seconds, which ensures that the application has enough time to send the request in case of incidence.

Additionally, we perform this produce operation by creating an instance with a `KafkaTemplate` in which the messages are sent to the respective topic via the `send` method. Ultimately, we receive a callback to check that the data was received correctly.

The configuration of the producer is the following: Bootstrap Kafka servers are located at the default localhost port 9092, while the Zookeeper processes will be located at the 2181. Acks configuration is set to all, which will guarantee that the producer gets an ack when all in-sync replias have received the record. In the case of failure, the number of retries perfromed by the client is three. Furthermore, key value is serialized with StringSerializer class and data value is

serialized with JsonSerializer class, which dictates how the Java Object should be turned into a byte array.

Partitions are assigned according to the key value. In our case, for both `users` and `riders`, the key represents the fields that take part in the matching process to find a driver to each user request. For this reason, the key value will be the date, hour and neighbourhood of the ride.

Depending on the topic, different retention periods have been defined (period for which records are stored in specific topic). For `riders` and `finalmatch` the retention period is set to 2 weeks, owing to the fact that both riders and users will be able to offer/look for a ride with this advance. On the other hand, `users` and `matches`'s period is 1 day, since neither of them are final topics, thus there is no need to retrieve the data for longer.

Regarding the content of the JSON String's that are being sent, they contain the following fields:

{"userID": StringValue, "neighbourhood": intValue, "time": StringValue, "date": StringValue, "latitude": DoubleValue, "longitude": DoubleValue }

- *userID* corresponds to the session identifier generated whenever the user logs in the application. It is used to perform the match and to retrieve the appropriate data for each request performed by the user. For each session started, a new one will be generated.
- *neighbourhood* has the value of the postal code of the address introduced.
- *date* and *time* show the schedule desired.
- *latitude* and *longitude* are the exact position from which the user will depart, and is used to calculate later on the walking time for each match proposed.

Finally, `finalmatch`'s data is mostly the same, but also adds both user and rider's location and respective identifiers. Furthermore, the *userID* field will no longer be the session id, but it will correspond to the universally unique identifier obtained with the Firebase authentication when the user registers in the platform. It remains the same for each user even when they log out, and it represents them within the application context.

## 3.2 Consumer API

The Consumer API is used to subscribe to one or more topics, pulling the stream of events produced to them.

As before, we use Spring's `ConsumerFactory` instance to implement it. It will create new consumer instances with the specified properties, those being: String deserializers for both key and value data and the identifier of the group the consumer is a part of. This consumer groups are associations of consumers connected to the same topic, and each one will receive records from different partitions. In brief, if only one consumer was in charge of getting messages from all partitions the application might not be able to keep up with the rate in which producers are sending the messages. On the other hand, thanks to the `@KafkaListener` annotation we can deploy listerners for the topic we wish to consume.

Records consumed from the topic `matches` will have the same fields as the example seen before, however it will also include an array list containing all the UUIDs of the possible riders for each user produced.

## 3.3 Streams API

The Streams API allows real-time streaming and processing data. In our case, we search for coincidences in the time, date and neighbourhood of the key-value pairs received from `users` and `riders`, pushing the coincidences into the *matches* topic.

The process followed to look for coincidences is represented in 3.1:

1. First we perform a LeftJoin, which will be key-based, left key will have to equal right key for it to be triggered. This means that, if the date, time and neighbourhood is the same for both records, they will be outputted to the resulting stream. It will also be window-based. For our system, as explained before, timestamps will have to be at least 14 days close for the messages to be joined.

2. After, a mapping transformation is carried out, changing the key value to the session identifier. This change is due to the fact that there is no longer a need to have the matching values as the key field, and it will allow us to group all the possible riders for the current request in the next step.

3. We then group the records by exisiting key. This will lead the stream to be converted into a KGroupedStream. This operation is a requisite for the aggregation of a stream, which will be our final step.

4. Lastly, we aggregate the values in order to obtain one single message with an array list containing all the possible riders found for the current session.

In all of the steps performed, we have to define explicitly the SerDes used for the resulting data, which are defined at the beggining of the class.

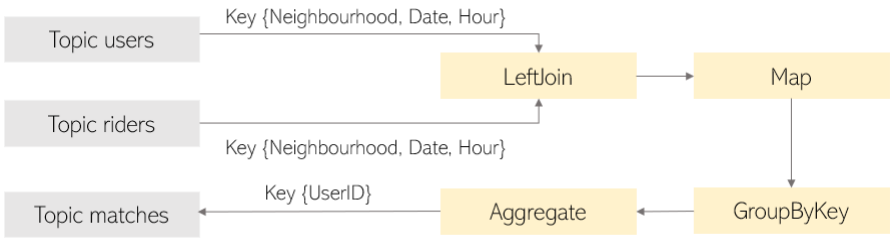


Figure 3.1: Stream diagram of the matching process.

# 4 | Client-side

The client-side application has been designed for devices working within the Android operating system.Next, we will describe the design principles that have lead to the actual application, as well as the user flow and the activities that have been created for its deployment.

## 4.1 Design principles

To develop the Android application, the following requisites have been taken into account:

- The main objective of the platform was for the user to be able to either offer or find a way to commuting to the EPFL on a daily basis.
- First, the high-fidelity wireframes of the application were designed, i.e the prototype before actually designing it in Android Studio. These were done in order to test the coherence of the interaction between the user and application flow.
- Interfaces needed to be kept simple and user-friendly, so as to provide users clear visibility of their actions in the UI.
- Possible errors must be taken into account, and clear feedback must be provided. Furthermore, the user must always be allowed to move a step back in the application.
- As soon as receiving new information from Apache Kafka, the UI will need to be updated accordingly.
- Lastly, provide confirmation in crucial steps. For instance, when creating or accepting a new ride.

## 4.2 User flow

Next we will see from the user's perspective the flow of the application. In other words, what steps he would need to follow in order to achieve his goal (finding/offering a ride). All of the screens that are going to be described are depicted in Fig. 4.1

1. First, he would encounter the login screen. Assuming he does not have an account yet, he would press the 'Sign up' textview and move directly to the Register page.
2. In this page, just by introducing the required email and password, a new account would be created. This screen also allows for user failure, for instance, if the user was to introduce an email not belonging to the EPFL domain or a password with less that 6 characters an error would trigger.

3. When moving back to the login page, the parameters would automatically appear in their respectives textviews. Therefore, just by pressing 'Login' the user would be logged in the application.

4. The main page of the platform offers four options: finding a ride, offering a ride, seeing the user's future rides or logout.

5. Whenever 'Your rides' is pressed, the user finds a non-interactive screen where the date, hour, departure address, rider's address and distance between both of them would be displayed.

6. On the other hand, if 'Find a ride' is the chosen option, the user would encounter a screen where the following parameters would have to be decided:

   (a) Initial address of the ride. When pressing the available button, a Google Maps activity would be the scenario in which the address will have to be selected. For that purposes, the user could either interact manually with the map or use the EditText at the top of the screen to write a distinctive parameter of the desired location. In the last case, the zoom would move directly to the area introduced when the lens is selected. Once, the location is set, the user can move back to the previous page, where the final location will be displayed.

   (b) The time of the ride, chosen over a time picker with 15 minutes intervals.

   (c) The date of the ride, chosen over a date picker. As discussed in previous parts, the ride can be booked 14 days in advanced.

7. When the user is satisfied with the decision, a new Google Maps activity would be displayed. In there, all the available riders obtained from the request would be represented with different markers.

8. When pressing a marker, a pop up window with all the ride details would be shown. That being, both addresses from the user and the rider, the distance between them and the planned schedule.

9. By clicking confirm the ride is created.

10. The same process would be followed if at the beginning "Offer a ride" is selected. The main difference is that instead of displaying Google Maps when clicking confirm, a pop up with the summary of the ride appears. That way, the user can check if he has made any mistakes before moving forward and proposing the new ride.

11. To conclude, 'Logout' would directly quit the user from the application and go back to the login page.

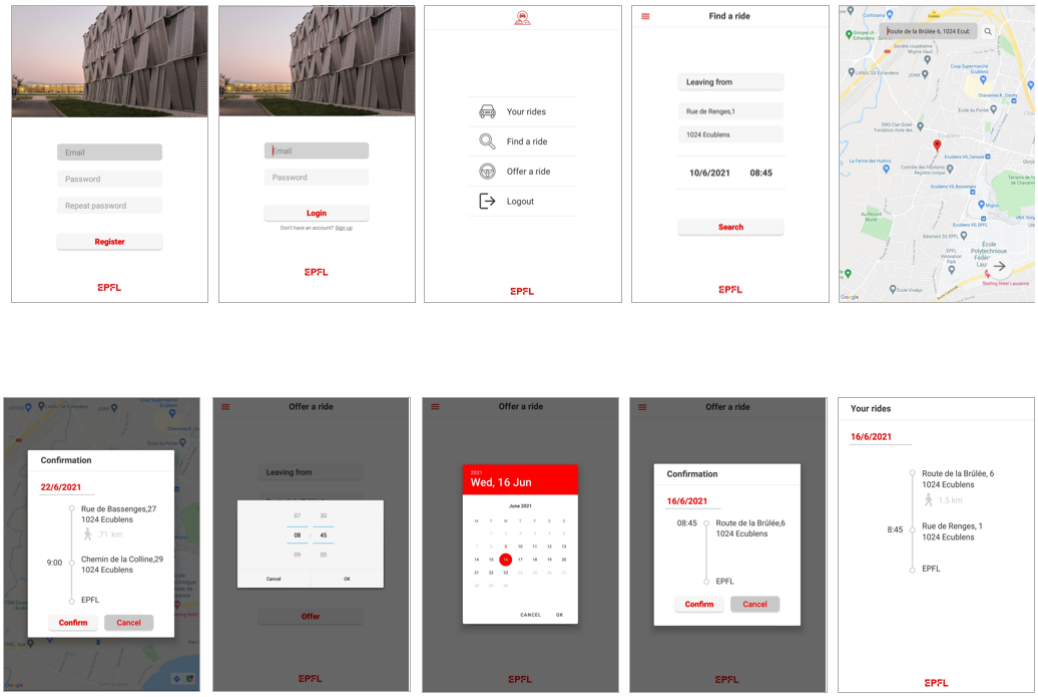


Figure 4.1: Android Application GUI.

## 4.3 Activities

All of the screen explained in the previous section are equivalent to an Activity and a layout file in the development process. Some of their main functionatilies will be explained next.

- *LoginActivity* and *RegisterActivity*: Connection with Firebase via Firebase Authentication instance, implementing the method `signInWithEmailAndPassword`. We receive a callback with a result stating if the authentication was succesful. Previously, a new Firebase database has been created, where all of the user's dependencies are stored.
- *FindRideActivity*: Implement `CustomTimePickerDialog` and `CustomDatePickerDialog` for the selection of the schedule. As all of the main functionalities are running on the main UI, both petitions are performed in threads (POST request to create a new user and petition to start the matching process). If not, the application could have an ANR error due to the waiting time for the request. After the thread is finish, the result is sent to the main process again,so that it can be displayed to the user.
- *OfferRideActivity*: Same as *FindRideActivity*, but also deploys a `AlertDialog` builder to create the pop up.
- *MapsActivity*: By installing the Google Maps SDK and configuring the necessary key, it is possible to create Google maps instances in our application. Main class performed is `geocoder`, that allows us to do multible functions, such as getting marker latitude and longitude or search for a location introduced by the user. In the *FindRideActivity* case, when performing the GET request to get the available riders, we read the consumed messages as they come, filtering them until the one from our user is received.

# 5 | Evaluation

In this chapter, we will evaluate our system. In order to do this, we have performed tests with Wrk2 becnhmarking tool and logged the latency of the requests from the actual Android application.

First, we have measured the latency and throughout obtained when performing HTTP loading tests monitored with the Wrk2 benchmarking tool[1]. The number of threads kept being changed, simulating as if different number of users were trying to access the system. For this operation, we have obtained the results represented in Fig. 5.1. The maximum throughput tested corresponds to 753 requests per second, as the computer in which it was performed did not allow to open more threads. Until that point we obtain that there are no losses. Regarding the latency, it is observed that as far as 740 request/s, it stays in the order of ms. From there, it starts scalating, reaching 3.3 s. Nonetheless, for most cases the latencies can be mostly overlooked.

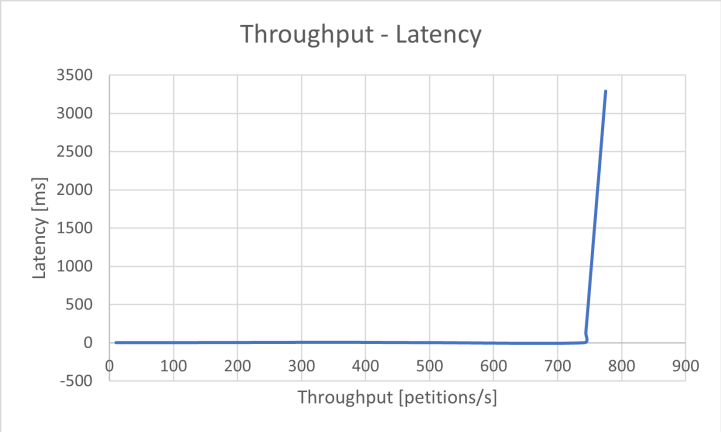


Figure 5.1: Throughput vs Latency.

Combining several tests that were being implemented at the same time, we are able to reach in a combined way a throughput of 1440 requests/second. In that case, we observe that some losses might be caused, around the percentage of 13%.

To evaluate the end-to-end latency of the system, we have run the application in the Android device under two different conditions. Initially, the time for each request has been measured, given that the user was the only one making the petitions at that time. Then, it has been measured again with HTTP loading test running from the computer. This way, we are able to obtain both the steps and the end-to-end latency of the system from the user's point of view, as it would be the one that they would experience in a real situation. The results obtained from the POST petitions show that the delay time is neglectable, as we are not getting information back and it is just measuring the time that it takes to make the petition. Nonetheless, for the

[1]https://github.com/giltene/wrk2

GET request it can be observed that there is a greater delay in the response. This delay is however, still small and almost imperceptible by the user. Regarding the end-to-end latency, it is the approximated time that an user would take from posting its request to getting the answer back (including producing, streaming and consuming processes). See table below for details.

| Condition | GET latency | End-to-End latency |
|---|---|---|
| *1 user in the system* | 0.012 s | 0.028 s |
| *Loading test in the system* | 2.51 s | 2.845 s |

Table 5.1: Latency measured from Android device.

# 6 | Conclusion

As expected, Apache Kafka is a really good tool for developing real-time streaming applications. Up to a certain number of users, it can process all of the requests without any evident lag. Furthermore, the streams API is also capable of computing the matches and forwarding them to the output topic without almost any latency.

On another note, the integration between the three main components of our platform: Android application, Spring framework and Apache Kafka also gives an remarkable result. As all of them include the needed tools for its deployment. Specially Spring Boot, which allows the communication between both sides thanks to the REST end points and the Kafka instances that it provides.

Lastly, further designing of the application could introduce more filters, that way the user gets more personified rides for his needs. This could include the price or the gender preference. Moreover, another improvement could be exploring the number of consumers in each consumer group, as well as the number of partitions deployed within the cluster, so as to outstand the performance and scalability of our server.

# Bibliography

[1] Martijn de Vos, Georgy Ishmaev, Johan Pouwelse. MATCH: A Decentralized Middleware for Fair Matchmaking In Peer-to-Peer Markets. December 7–11, 2020.