A Kafka-Based Centralized Platform for Smart Vehicle Supervising

Filipe Pires [85122], João Alegria [85048]

Software Architecture
Department of Electronics, Telecommunications and Informatics
University of Aveiro

April 21, 2020

Introduction

This report aims to describe the work developed for the second assignment of the course of 'Software Architecture', focused on a platform that collects and processes information from simulated vehicles.

The aim of the assignment was to explore the capabilities of a technology like Kafka in standard IoT solutions. The platform itself isn't meant to provide a field-tested solution to a problem or a set of problems, rather it is supposed to show how communications via Kafka can empower developers in a time that modularity is more important than ever and system components must be prepared to easily transfer data streams between each other.

So in this report we present the architecture of our solution and the Kafka-related configurations, justifying them according to what we learned and found to be most suitable for each specific use case. We also mention how the work was distributed amongst the authors.

All code developed is publicly accessible in our GitHub repository:
`https://github.com/FilipePires98/AS/`

1 IoT, Kafka and Connected Devices

Internet of Things (IoT) is becoming an increasing topic of interest among technology giants and business communities. IoT Components are interconnected devices over a network, which are embedded with sensors, software and smart applications so they can collect and exchange data with each other or with cloud / data centers.

One of the areas in which IoT is paving its way is the connected vehicles. According to Gartner predictions (*1*), by this year there should be about a quarter-billion connected vehicles on the road, which are more automated, providing new in-vehicle services such as enhanced navigation system, real-time traffic updates, weather alerts and integration with monitoring dashboards. In order to process the data generated by IoT connected vehicles, data is streamed to central processors usually located in the cloud. The collected information can be analysed and data can be extracted and transformed to the final result, which can be sent back to the vehicle or to a monitoring dashboard.

In this project we explore a hypothetical use case where communication between devices (or entities) is done through Kafka. Apache Kafka (*2*) is high-throughput distributed messaging system in which multiple producers send data to Kafka cluster and which in turn serves them to consumers. It is a distributed, partitioned, replicated commit log service. The Java application we developed and that is described in this report is a simplified version of an IoT data processing and monitoring application for connected vehicles, aimed to to explore Kafka capabilities for messaging between entities.

1.1 The Data

As having actual connected vehicles with processing units capable of collecting car data and transmitting it to other entities was out of the scope of the project, this is simulated through a simple text file containing 1 transmission (or message) per line. This data file with the name of `CAR.TXT` is placed under a specific directory and is read by our platform for processing.

In order to dynamically generate such source data, we developed a small script in Python that receives as parameters the number of cars to be simulated and the total number of messages to be generated from those cars and stored in the text file. The script is called `generateCAR.py` and is placed in the scripts package inside our project. By default, it creates 10 different cars and writes 100 messages, but these numbers can be easily modified inside the script.

We used a random-based approach to generate each aspect of a message. Unique register codes are created to represent vehicles, as well as their status and speeds. Message types are not generated with a specific pattern, but we made it far more likely to generate a message of type HEARTBEAT (see section 1.2) than of any other type.

1.2 The Messages

Messages can be of 1 of 3 types, each with a specific purpose and format:

- HEARTBEAT - the simplest type of message meant to notify the platform that the car is still connected.
  Format: | car_reg | timestamp | 00 |
- SPEED - message meant to inform the platform about the current speed of the car. This allows the system to determine whether the car is going under the speed limit or not and trigger an alarm if necessary.
  Format: | car_reg | timestamp | 01 | speed |
- STATUS - messages meant to detect whether the smart sensor detects any malfunction. This in theory could help the platform provide or suggest a solution for the malfunction to the car driver considering the entire network of connected vehicles.
  Format: | car_reg | timestamp | 02 | status |

The car_reg corresponds to the register code of each vehicle and the timestamp corresponds to the time instance when the message was created, the remaining elements are self explanatory. As we will see, each message type is treated differently both in their purpose and in the care with which their transmission is done.

```
| 45-SH-72 | 1586183268975 | 02 | OK |
| 28-MC-82 | 1586183269976 | 02 | OK |
| 42-UW-71 | 1586183272978 | 00 |
| 73-FD-20 | 1586183273979 | 00 |
| 28-MC-82 | 1586183274980 | 01 | 20 |
| 55-LZ-42 | 1586183276981 | 00 |
| 64-IY-98 | 1586183281986 | 00 |
| 45-SH-72 | 1586183286989 | 00 |
| 42-UW-71 | 1586183311005 | 00 |
| 80-DE-01 | 1586183315006 | 00 |
| 30-UU-59 | 1586183319009 | 00 |
| 78-ST-77 | 1586183324009 | 00 |
| 28-MC-82 | 1586183325011 | 02 | KO |
| 30-UU-59 | 1586183327012 | 01 | 0 |
| 73-FD-20 | 1586183328012 | 01 | 130 |
```

Example of a portion of the `CAR.TXT` file.

2 System Architecture

Modular architecture is very appealing for IoT solutions, as it offers a way to manage the complexity of a problem by breaking it down to smaller and more easily manageable modules. Plus, IoT components are constantly changing, so limiting the effects of such organic-like transformations to individual modules allows developers to keep in control of the system's growth.

Software applications are embracing distributed, real time and on-the-cloud as the new norm. They are focused on providing real service rather than just being obsessed on lists of features. This is what drives the increase in awareness for the importance of modularity in software development. And this is partly what was meant to be explored in this assignment. So in this chapter we focus on presenting the interacting entities of our vehicle supervising platform and the components that constitute their functionality.

2.1 Entities

There are 4 entities, each with its own responsibilities. These entities simulate the management of the car network, with representative features of what could be accomplished by a fully implemented version. Following is their description and a simple block diagram in Figure 1.

- CollectEntity - its role is to collect data from connected vehicles (represented by `CAR.TXT`) and produce messages to the other entities with the gathered information (through communication channels established *a priori* and explained in chapter 3).
- ReportEntity - as the name states, this entity is responsible for reporting all that is transmitted by CollectEntity, writting it to `REPORT.TXT`.
- BatchEntity - Batch's role is to make possible the computation of any relevant metrics and calculations related to the information collected; in our case it merely accomplishes the same as ReportEntity, writting results to `BATCH.TXT`.
- AlarmEntity - its role is to trigger alarms and present them to the platform user when some relevant event occurs; in our case alarms are triggered when a car surpasses the predefined speed limit of 120 Km/h; all alarms are written to `ALARM.TXT`.

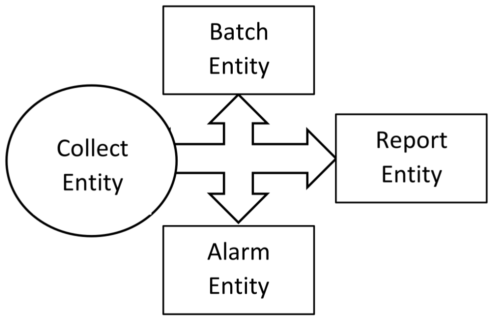


Figure 1: Basic block diagram for the platform centralized in the Collect Entity, taken from (*3*).

2.2 Components

The Java application is organized with packages. Each package contains a set of files with common natures so that managing source code is made easy. The ”data” package contains both the input (CAR.TXT) and the output data (REPORT.TXT, BATCH.TXT and ALARM.TXT). The ”entities” package contains every entity class. The ”scripts” package holds project scripts, including the Kafka initialization and deletion scripts and the CAR.TXT file generation script.

The ”message” package contains all classes related to the messages sent through Kafka; here we have the Message class, with all that was described in section 1.2, and our implementations of the serializer and deserializer classes of Message instances. Kafka has serialization classes available for non-structured data such as regular integers or strings, but we intended to send a structured message for greater usability, so implementing a serialization process for such structure was required in order to transmit data through Kafka.

Finally, the ”kafkaUtils” package provides Kafka utilities such as EntityAction. This interface, implemented by all consumer entities, ensures that such entities have means to process Kafka messages in a predefined format by defining the method *processMessage()*. This method receives the identifier of the consumer that is going to process the current message, the topic that the message belongs to and the key-value pair of the message.

The package also contains Kafka Consumers and Producers wrappers implemented by us. Kafka is explained in greater detail in chapter 3, however a simple way of understanding producers and consumers in Kafka is to see them as the components responsible for creating messages that can be sent via Kafka topics and sending them, and for subscribing to topics and receiving messages from them, respectively. By implementing our own wrappers, we gain greater control over their configurations and functionalities and facilitate our work in other components. The Producer class supports 3 message sending methods:

- *fireAndForget()* - where the producers doesn’t wait for the Kafka response in any fashion.
- *sendAsync()* - where the producer doesn’t wait for the Kafka response, but receives it asynchronously through callbacks.
- *sendSync()* - where the producer waits for a delivery confirmation before proceeding to another message

Producer also has an inner class called ProducerCallback that, as the name suggests, deals with the callbacks related to the executions of the method *sendAsync()*. The Consumer class on the other hand works as a java thread by implementing the Runnable interface. Once instantiated, it listens to the topics where it is subscribed and processes arriving messages by calling *processMessage()*. Each consumer has an instance of RebalanceListener, our implementation of Kafka’s ConsumerRebalanceListener for partition management. This listener is explained in greater detail in section 3.2.

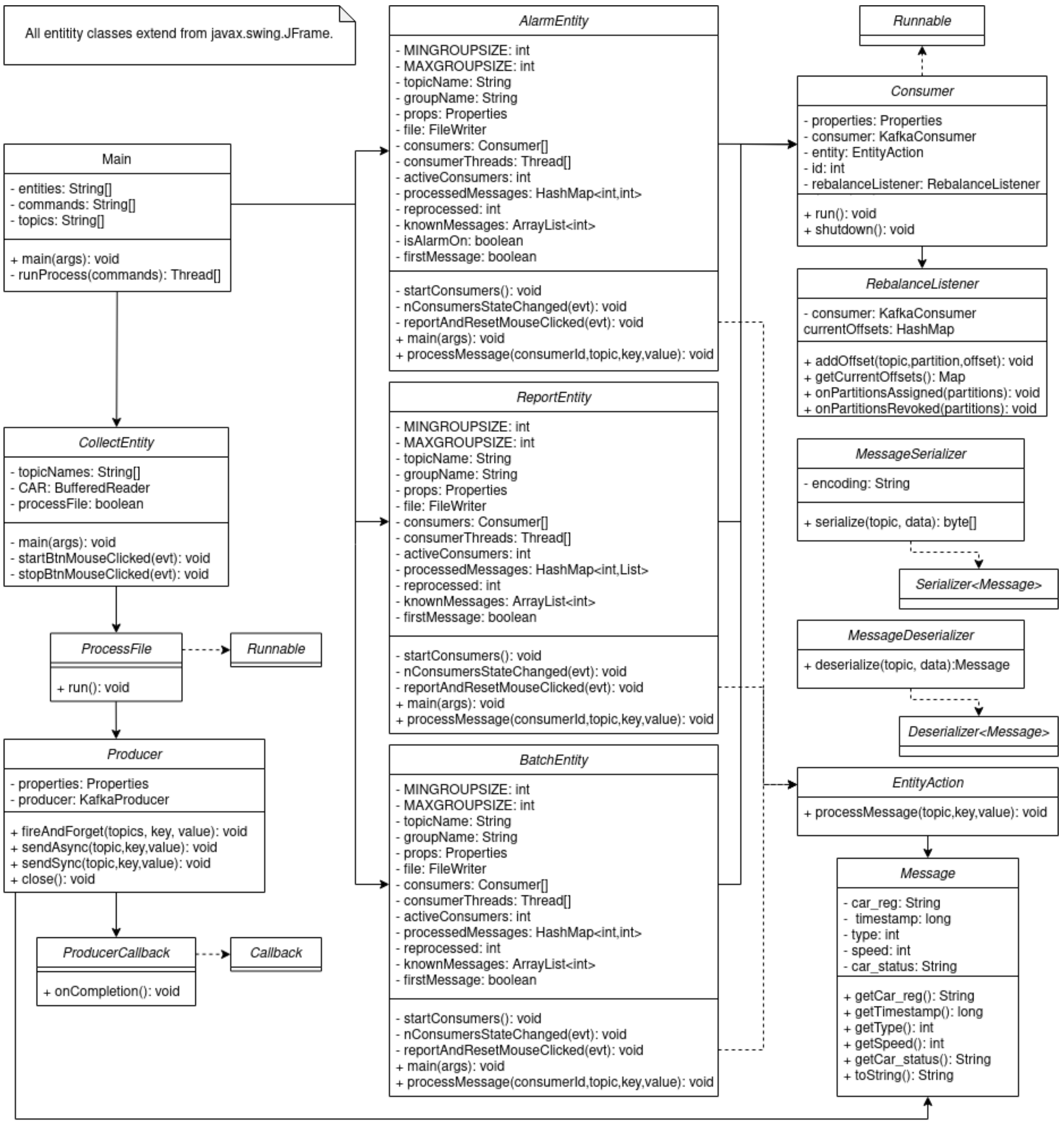


Figure 2: Java application's class diagram.

Figure 2 presents the project's class diagram with the relations between components. In order to more easily visualize interactions between entities, we designed an interaction diagram as well, present in Figure 3.

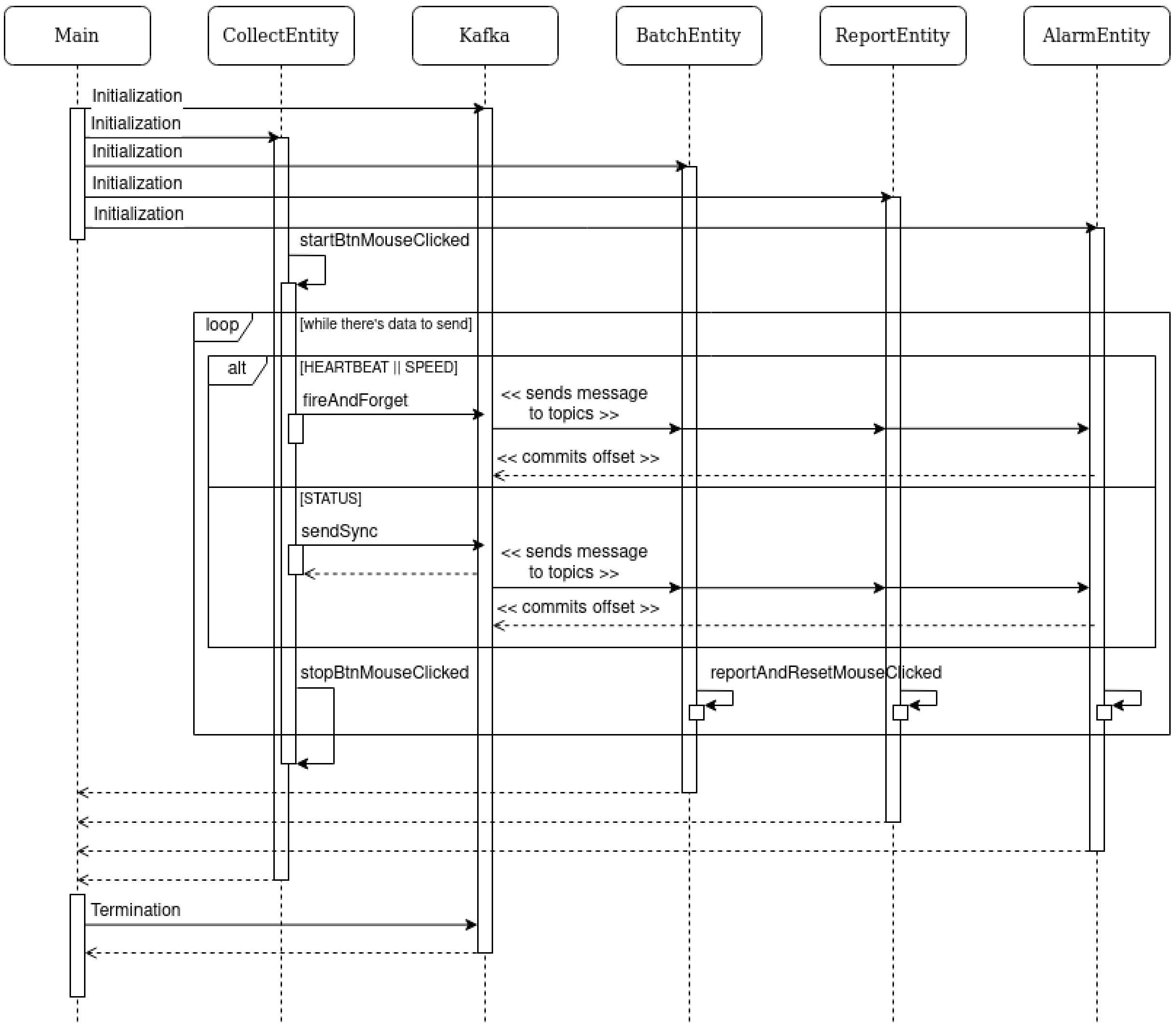
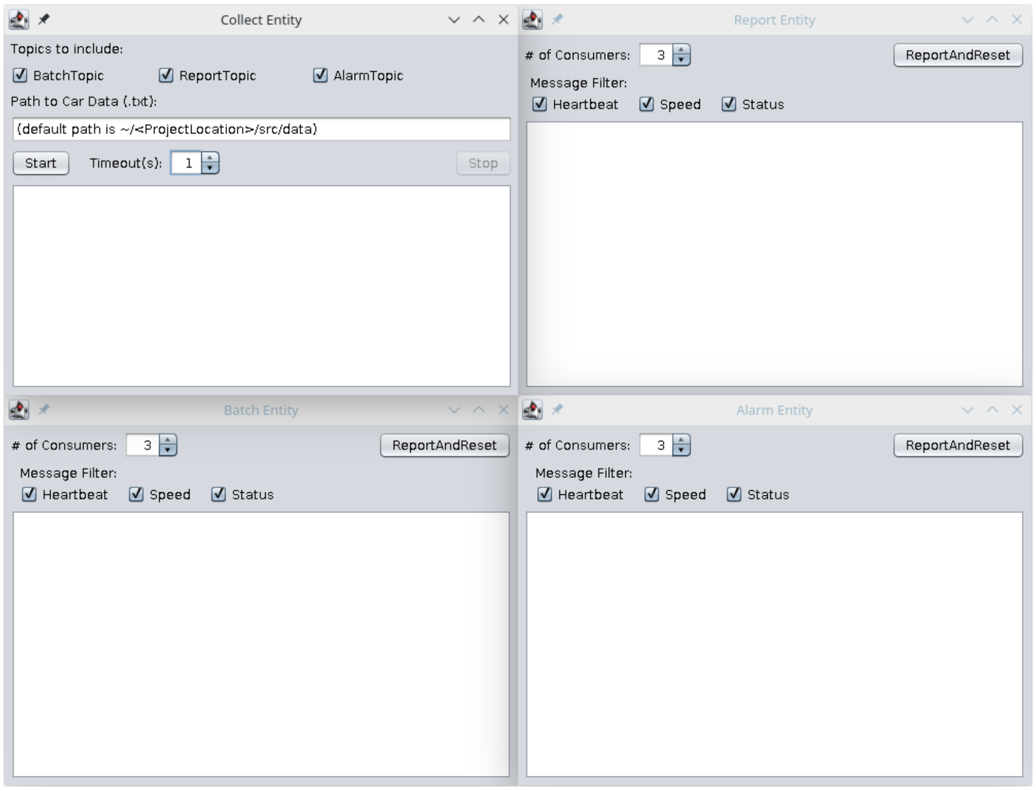


Figure 3: Entities interaction diagram.

2.3 User Interface

In any management platform there needs to be some sort of interface for users to interact with the system. Our choice was to develop a graphical user interface (GUI) for each entity using Java Swing (*4*).

All entities have a window dedicated to presenting relevant information, including processed messages status information and possible execution errors. Below we see a sequence of figures showing all GUIs in different program execution states.



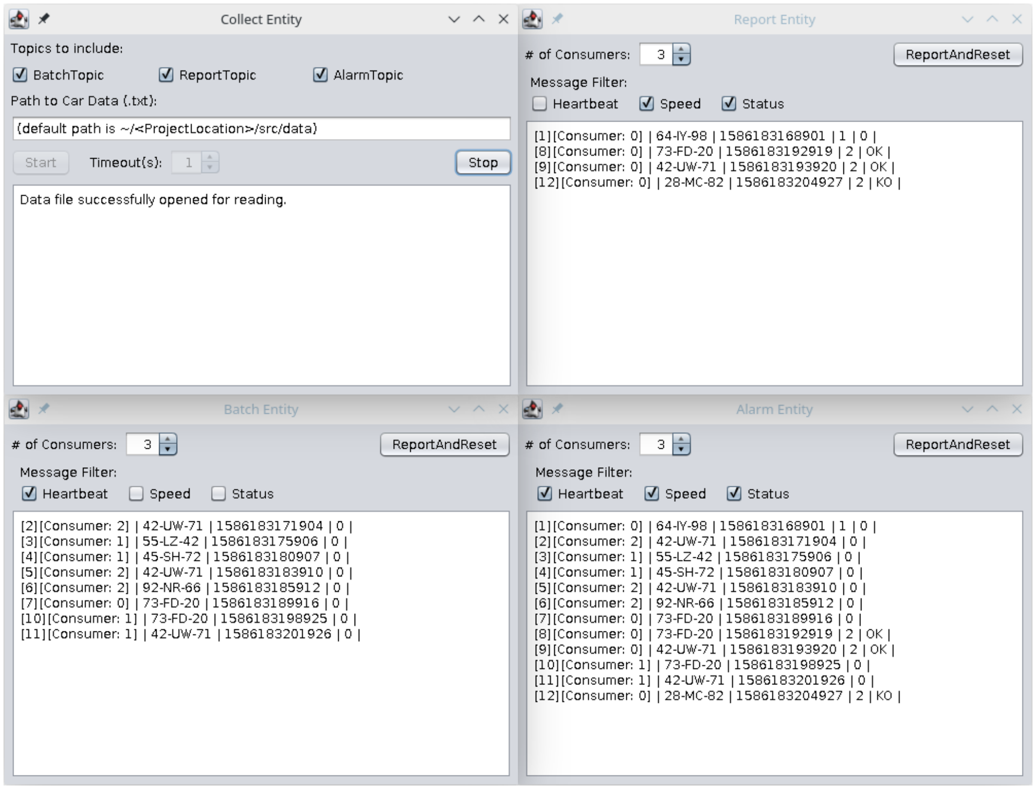
```
Collect Entity
Topics to include:
[✓] BatchTopic   [✓] ReportTopic   [✓] AlarmTopic
Path to Car Data (.txt):
(default path is ~/<ProjectLocation>/src/data)
Start   Timeout(s): 1   Stop

Report Entity
# of Consumers: 3   ReportAndReset
Message Filter:
[✓] Heartbeat   [✓] Speed   [✓] Status

Batch Entity
# of Consumers: 3   ReportAndReset
Message Filter:
[✓] Heartbeat   [✓] Speed   [✓] Status

Alarm Entity
# of Consumers: 3   ReportAndReset
Message Filter:
[✓] Heartbeat   [✓] Speed   [✓] Status
```

Figure 4: GUIs before file reading.



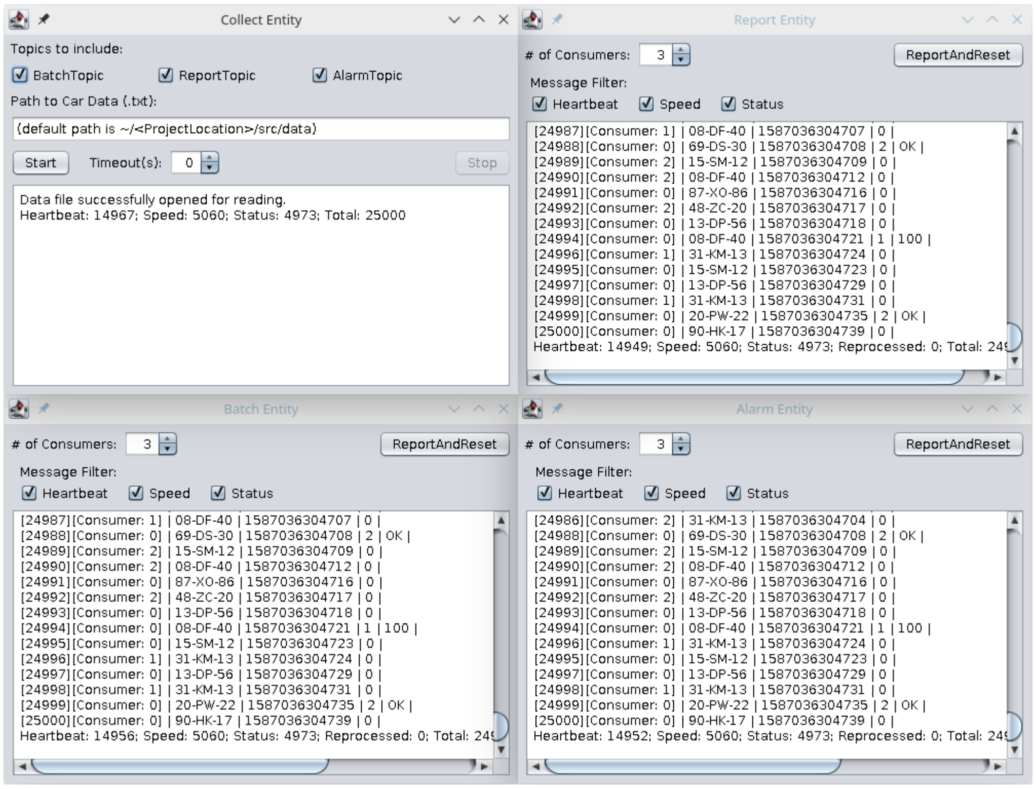
```
Collect Entity
Topics to include:
[✓] BatchTopic   [✓] ReportTopic   [✓] AlarmTopic
Path to Car Data (.txt):
(default path is ~/<ProjectLocation>/src/data)
Start   Timeout(s): 1   Stop
Data file successfully opened for reading.

Report Entity
# of Consumers: 3   ReportAndReset
Message Filter:
[ ] Heartbeat   [✓] Speed   [✓] Status
[1][Consumer: 0] | 64-IY-98 | 1586183168901 | 1 | 0 |
[8][Consumer: 0] | 73-FD-20 | 1586183192919 | 2 | OK |
[9][Consumer: 0] | 42-UW-71 | 1586183193920 | 2 | OK |
[12][Consumer: 0] | 28-MC-82 | 1586183204927 | 2 | KO |

Batch Entity
# of Consumers: 3   ReportAndReset
Message Filter:
[✓] Heartbeat   [ ] Speed   [ ] Status
[2][Consumer: 2] | 42-UW-71 | 1586183171904 | 0 |
[3][Consumer: 1] | 55-LZ-42 | 1586183175906 | 0 |
[4][Consumer: 1] | 45-SH-72 | 1586183180907 | 0 |
[5][Consumer: 2] | 42-UW-71 | 1586183183910 | 0 |
[6][Consumer: 2] | 92-NR-66 | 1586183185912 | 0 |
[7][Consumer: 0] | 73-FD-20 | 1586183189916 | 0 |
[10][Consumer: 1] | 73-FD-20 | 1586183198925 | 0 |
[11][Consumer: 1] | 42-UW-71 | 1586183201926 | 0 |

Alarm Entity
# of Consumers: 3   ReportAndReset
Message Filter:
[✓] Heartbeat   [✓] Speed   [✓] Status
[1][Consumer: 0] | 64-IY-98 | 1586183168901 | 1 | 0 |
[2][Consumer: 2] | 42-UW-71 | 1586183171904 | 0 |
[3][Consumer: 1] | 55-LZ-42 | 1586183175906 | 0 |
[4][Consumer: 1] | 45-SH-72 | 1586183180907 | 0 |
[5][Consumer: 2] | 42-UW-71 | 1586183183910 | 0 |
[6][Consumer: 2] | 92-NR-66 | 1586183185912 | 0 |
[7][Consumer: 0] | 73-FD-20 | 1586183189916 | 0 |
[8][Consumer: 0] | 73-FD-20 | 1586183192919 | 2 | OK |
[9][Consumer: 0] | 42-UW-71 | 1586183193920 | 2 | OK |
[10][Consumer: 1] | 73-FD-20 | 1586183198925 | 0 |
[11][Consumer: 1] | 42-UW-71 | 1586183201926 | 0 |
[12][Consumer: 0] | 28-MC-82 | 1586183204927 | 2 | KO |
```

Figure 5: GUIs during message transmission.



Collect Entity: BatchTopic ✓, ReportTopic ✓, AlarmTopic ✓; Path to Car Data (.txt): (default path is ~/<ProjectLocation>/src/data); Timeout(s): 0

```
Data file successfully opened for reading.
Heartbeat: 14967; Speed: 5060; Status: 4973; Total: 25000
```

Report Entity: # of Consumers: 3; Message Filter: Heartbeat ✓, Speed ✓, Status ✓

```
[24987][Consumer: 1] | 08-DF-40 | 1587036304707 | 0 |
[24988][Consumer: 0] | 69-DS-30 | 1587036304708 | 2 | OK |
[24989][Consumer: 2] | 15-SM-12 | 1587036304709 | 0 |
[24990][Consumer: 2] | 08-DF-40 | 1587036304712 | 0 |
[24991][Consumer: 0] | 87-XO-86 | 1587036304716 | 0 |
[24992][Consumer: 2] | 48-ZC-20 | 1587036304717 | 0 |
[24993][Consumer: 0] | 13-DP-56 | 1587036304718 | 0 |
[24994][Consumer: 0] | 08-DF-40 | 1587036304721 | 1 | 100 |
[24996][Consumer: 1] | 31-KM-13 | 1587036304724 | 0 |
[24995][Consumer: 0] | 15-SM-12 | 1587036304723 | 0 |
[24997][Consumer: 0] | 13-DP-56 | 1587036304729 | 0 |
[24998][Consumer: 1] | 31-KM-13 | 1587036304731 | 0 |
[24999][Consumer: 0] | 20-PW-22 | 1587036304735 | 2 | OK |
[25000][Consumer: 0] | 90-HK-17 | 1587036304739 | 0 |
Heartbeat: 14949; Speed: 5060; Status: 4973; Reprocessed: 0; Total: 24
```

Batch Entity: # of Consumers: 3; Message Filter: Heartbeat ✓, Speed ✓, Status ✓

```
[24987][Consumer: 1] | 08-DF-40 | 1587036304707 | 0 |
[24988][Consumer: 0] | 69-DS-30 | 1587036304708 | 2 | OK |
[24989][Consumer: 2] | 15-SM-12 | 1587036304709 | 0 |
[24990][Consumer: 2] | 08-DF-40 | 1587036304712 | 0 |
[24991][Consumer: 0] | 87-XO-86 | 1587036304716 | 0 |
[24992][Consumer: 2] | 48-ZC-20 | 1587036304717 | 0 |
[24993][Consumer: 0] | 13-DP-56 | 1587036304718 | 0 |
[24994][Consumer: 0] | 08-DF-40 | 1587036304721 | 1 | 100 |
[24995][Consumer: 0] | 15-SM-12 | 1587036304723 | 0 |
[24996][Consumer: 1] | 31-KM-13 | 1587036304724 | 0 |
[24997][Consumer: 0] | 13-DP-56 | 1587036304729 | 0 |
[24998][Consumer: 1] | 31-KM-13 | 1587036304731 | 0 |
[24999][Consumer: 0] | 20-PW-22 | 1587036304735 | 2 | OK |
[25000][Consumer: 0] | 90-HK-17 | 1587036304739 | 0 |
Heartbeat: 14956; Speed: 5060; Status: 4973; Reprocessed: 0; Total: 24
```

Alarm Entity: # of Consumers: 3; Message Filter: Heartbeat ✓, Speed ✓, Status ✓

```
[24986][Consumer: 2] | 31-KM-13 | 1587036304704 | 0 |
[24988][Consumer: 0] | 69-DS-30 | 1587036304708 | 2 | OK |
[24989][Consumer: 2] | 15-SM-12 | 1587036304709 | 0 |
[24990][Consumer: 2] | 08-DF-40 | 1587036304712 | 0 |
[24991][Consumer: 0] | 87-XO-86 | 1587036304716 | 0 |
[24992][Consumer: 2] | 48-ZC-20 | 1587036304717 | 0 |
[24993][Consumer: 0] | 13-DP-56 | 1587036304718 | 0 |
[24994][Consumer: 0] | 08-DF-40 | 1587036304721 | 1 | 100 |
[24996][Consumer: 1] | 31-KM-13 | 1587036304724 | 0 |
[24995][Consumer: 0] | 15-SM-12 | 1587036304723 | 0 |
[24997][Consumer: 0] | 13-DP-56 | 1587036304729 | 0 |
[24998][Consumer: 1] | 31-KM-13 | 1587036304731 | 0 |
[24999][Consumer: 0] | 20-PW-22 | 1587036304735 | 2 | OK |
[25000][Consumer: 0] | 90-HK-17 | 1587036304739 | 0 |
Heartbeat: 14952; Speed: 5060; Status: 4973; Reprocessed: 0; Total: 24
```

Figure 6: GUIs with counter reports.

The central entity, Collect, is the one with more control over the platform. In its GUI, the user can: define the path to the data file `CAR.TXT` (a standard path is defined by default, directed towards the ”data” package); select which topics shall be used, allowing the program to run even if one of the consumer entities isn't working; define a timeout variable that sets the time period between message dispatches; initiate the collecting and transmission of data by clicking the `Start` button. During execution, at any time, the user can also terminate data transmission by clicking the `Stop` button (enabled only during execution).

Batch, Report and Alarm entities all have similar interfaces with the same operations available to the user. These operations are: defining the number of consumers in the Heartbeat consumer group (see subsection 3.2.3 for further details on consumer groups); filtering which messages to present on the information window according to their type; presenting the number of messages of each type, of reprocessed messages and in total processed so far by clicking the `ReportAndReset` button (as the name suggests, these counters are set to zero every time the button is clicked).

3 Kafka Infrastructure

As a distributed streaming platform, Apache Kafka has 3 main capabilities: publish and subscribe to record streams, similarly to a message queue or messaging system; store those record streams in a fault-tolerant way; process streams of records as they occur. Its general use is for building real-time streaming data pipelines to establish communication between applications, and building real-time streaming applications to transform and react to streams of data.

For our specific case, the messaging system scenario seemed the most appropriate. As stated in their online documentation (*2*), normally these type of systems follow 2 possible models: message queueing or publish-subscribe. The queue model is characterized by a pool of consumers that read from the messaging server and each record goes to one of the consumers; this has several positive aspects such as allowing the division of the records in a balanced fashion by the registered consumers and enabling the processing rate and power to scale; on the other hand, being a queue, when a record is processed it may not be reprocessed. Comparatively, the publish-subscribe model is characterized by broadcasting each record to all registered consumers; this broadcasting feature can be seen as a positive aspect, but is counterweighted by the fact that there is no way to scale the process, since every message is sent to every consumer.

With this in mind, we established Kafka's infrastructure as the message system and the backbone for the entire platform, making use of both models metioned above. The usage of this platform was not only a constraint in the proposed assignment, but was also justified by our own personal analysis that showed that Kafka is a solution complete enough for our purposes and guarantees a number of quality metrics that were found important. It was these metrics we explored during development in order to understand their influence on the quality attributes of a system with a standard IoT architecture.

3.1 Initialization and Destruction Scripts

Obviously, to use this platform we first needed to instantiate the services and infrastructure composing Apache Kafka. To do so, we had 2 approaches: use the instantiation scripts provided with the source code available online (*2*) or use docker as a middleware framework that would enable us to abstract the instantiation process and use Kafka in a container. We chose the former option, which requested more work from our side but enabled us to instantiate the platform with more precision and control.

Manually running each necessary command was obviously out of question. The solution was to develop 2 scripts to automate both the creation and deletion of the infrastructure necessary for this project. These scripts are available together with the source code, in the package folder `scripts`.

The initialization script is the first action to be executed when the project is ran. Also, as you might guess, the deletion script is the last thing to be executed before the system is terminated (see Figure 3). In relation to *initKafka.sh*, the initialization script, it is responsible for: starting Zookeeper, the entity in charge of managing the Kafka Brokers; initializing the Kafka Brokers themselves, entities with the logic of the platform, for service providing (3 brokers are deployed); creating the necessary Kafka Topics. The final order of actions performed is:

1. Instantiate Zookeeper
2. Instantiate Kafka Brokers
3. Store process IDs of the Kafka Brokers in a auxiliary file
4. Create necessary topics for this project

When executed in the Java application's main function, the script receives 3 strings as arguments, corresponding to the names of the 3 Kafka topics required for the platform: BatchTopic, ReportTopic and AlarmTopic. Each of these has the objective of establishing communication between the CollectEntity and the BatchEntity, ReportEntity and AlarmEntity, respectively. These topics were configured with two properties in mind: the replication factor and the partition factor. For replication, each topic was set with a factor of 3 (as suggested by the 3 brokers), due to message policies explained in section 3.2. For partitioning, we defined 3 partitions per topic; decision made due to the fact that messages of type HEARTBEAT are meant to be produced in a much larger scale than those of other types, and so partitions enable a more efficient and balanced consumption.

In relations to *deleteKafka.sh*, the deletion script, its responsibility is to kill the processes previously stored in the auxiliary files and delete the logs generated by the same processes. The final order of actions performed is:

1. Fetch process IDs from the auxiliary file
2. Kill the fetched processes
3. Delete the generated logs

3.2 Topics and Constraints

Constraints are what define the behavior of the interactions with Kafka. To establish them on our infrastructure, 3 alternatives of constraint parameterization were available: either in the consumers' properties; in the producers' properties; or in the definition of properties for the topics themselves.

As we have seen in section 1.2, several message types should be considered. The imposed condition of all types being required to be sent to all topics is quite relevant since this means that the properties ensuring the proper functioning of shared resources (the Kafka topics) is dictated by the most restricting type. Other properties were defined differently for each message type, according to the predefined constraints imposed on them. Those constrains are:

- HEARTBEAT
  - can be lost
  - can be reordered
  - can be reprocessed
- SPEED
  - can be lost
  - cannot be reordered
  - cannot be reprocessed
- STATUS
  - cannot be lost
  - cannot be reordered
  - can be reprocessed

A further note needs to be made in relation to the HEARTBEAT message type and consumption strategy adopted. The fact that this message type was meant to be generated in a much larger scale in relation to the other message types, a consumer group was mandatory to be implemented to efficiently consume the massive amounts of HEARTBEAT messages. When analyzing possible strategies, 2 options stood out: either 3 different consumer groups would be instantiated per topic, with one or more consumers assigned to them to make possible a greater control over the constraints assigned to each message type; or one consumer group would be instantiated per topic with the most restricting constraints with one or more consumers assigned to it. The first strategy was not as efficient in terms of performance, since each message was going to be consumed 3 times more just to make possible the separation of messages in the 3 available types. This way, scalability is compromised, since if there were 100 messages types, 100 consumer groups would be needed and thus unnecessary overhead would be created. For that reason we implemented only one consumer group per topic which deals with all messages types in a efficient way.

3.2.1 Message Lost

Messages of type STATUS should never be lost. After doing some research, we understood that there were specific properties for Kafka Producers and Topics that allowed the assurance of message persistence. Additionally a specific Consumer behavior was taken in consideration to further assure a lossless data flow.

On the Producer side, the `acks` property, related to the way the Kafka leader considers the ”send” request complete, was the most relevant for our purposes. ”Acknowledge” has 3 options: when receiving a request to store a message, the leader immediately considers it complete, not making sure of its persistence in its logs nor requesting the acknowledgement from any of its followers; alternatively, the leader itself can assure message persistence, returning ”request complete” without requesting acknowledgement from its followers; or ”request complete” is only returned after the leader and all of its followers acknowledge the persistence of the message. The codification of these modes is made by assigning ”0” to no acknowledges, ”1” to only the leader's acknowledge and ”-1” or ”all” to the acknowledge of both leader and followers. Property `acks` was set to ”0” mode for the HEARTBEAT message type since it can be lost, and ”all” for the STATUS message type since it cannot. For the SPEED message type, although it can be lost, `acks` was set to ”all” due to another constraint (see section 3.2.3).

Additionally, for messages of types HEARTBEAT and SPEED, the *fireAndForget()* method of sending message discussed in section 2.2 was used, since the Kafka response was not important to assert if the persistance of the message was successful or not. Comparatively, for the STATUS type, *sendSync()* was used, keeping the program synchronized with Kafka by waiting for its response. This way, in case of an error, the message could be re-sent before moving on to another one to assure the correct persistance of the message in the Kafka system.

On the Topics side, the replication factor is quite important in a no-message-loss scenario, since it is mandatory that the message is stored not in a single place, but in several. Defining the replication factor to 1 would result in a single point of failure: in the event of that single machine containing critical messages crashing, all those messages would be lost since there were no backups. For that reason replication is a must in a critical scenario. We settled with a replication factor of 3, which was considered to be a satisfactory number for our context.

Lastly, on the Consumer side, we noticed that it was desirable for our system to manually commit the offsets for greater control. This option enabled controlling when to commit the processed offsets: either before the message processing or after it. To compare both possibilities, we analysed what consequences would each have and ran some tests. Committing before processing the data made possible the loss of some messages, since it can happen that the consumer fetches messages from the Kafka system, commits the offsets and immediately crashes - resulting in unprocessed and lost messages. Committing only after processing prevented this problem but brought the possibility of causing message reprocessing, since if the consumer fetches messages from the Kafka system, processes them and crashes right after, the commit is not completed and the message is reprocessed. No solution was ideal, and much was taken in consideration when implementing the configuration. However, an additional constraint discussed further ahead in section 3.2.3 made us implement this offset commit in another fashion.

3.2.2 Message Reordering

To deal with messages reordering, 2 properties on the Producer side were found along with a specific method of message sending that, through tests conducted by us, proved to be enough to handle such issue. The Producer properties were the `retries` and `max.in.flight.requests.per.connection`. The `retries` property establishes a maximum number of retries per send request. The `max.in.flight.requests.per.connection` establishes a maximum number of unacknowledged requests per connection to the Kafka leader. These properties, when combined, can minimize - or even neutralize - the reordering probability.

The values these properties have by default, which are over 2 billion for the `retries` and 5 for the `max.in.flight.requests.per.connection` allow messages to be reordered. To illustrate this, a brief example is here given: suppose that we send a message but the Kafka leader takes some time to respond and, in that time, a new connection with another message is also produced; in the case the first message response is unsuccessful and the second is successful, the first one due to the retries established will re-send the request, changing the intended order of the messages. To prevent this case, it was decided that for both the SPEED and STATUS message types (the ones where the order was mandatory) `max.in.flight.requests.per.connection` would have the value 1 assigned, neutralizing the reordering possibility.

For the message sending method, it was also considered the fact that the Kafka system was partitioned, which meant that if not specified Kafka itself distributes messages in a balanced way across all available partitions and causes the order to be lost. To neutralize that possibility, all messages of type SPEED or STATUS were forced to be produced to a specific partition, in our case, to partition 0. Recall that HEARTBEAT messages are produced in a much larger scale than the other types, so they are the ones that makes sense to divide amongst several partitions.

3.2.3 Message Reprocessing

A discovered Consumer side behavior was adopted in order to deal with message reprocessing. This behavior is explained here, along with another property configured on the Producer side.

The property configured on Kafka Producers was `enable.idempotence`, which, when enabled, assures that the producer only writes one copy of the message. Otherwise, due to retries, more than one copy can be persisted which can cause reprocessing of the message. SPEED message type was the one that required not to be reprocessed, and for that reason was the one were this property was enabled. For the correct operation of this property, other properties are affected, such as the `max.in.flight.requests.per.connection` that need to be less or equal to 5, the `retries` that need to be higher than 0 and `acks` that need to be set to "all". This last internal constraint was the reason the `acks` property was set to "all" in the SPEED message type(as mentioned in Section 3.2.1), which decreases the message loss probability but remains possible due to the *fireAndForget()* method.

Lastly, for the consumers efforts, we found that the usage of a Rebalance Listener was necessary. A Rebalance Listener is an entity which follows a Kafka defined behavior (interface), that enables the injection of code in two specific events when assigned to a consumer. Those events are the occurrence of new partitions or topics being assigned or being revoked to/from the consumer, due to either not signalling it is alive for a long time or to a new consumer being added to the group. The last event was more relevant, where partitions and topics are removed from the consumer, since that enabled committing the latest processed offsets.

Given the fact mentioned in section 3.2 and the fact that every topic may contain every message type, the order constraint of the SPEED message type meant that we needed to implement the Rebalance Listener to every consumer of every consumer group.

3.2.4 Code Configuration

For further analysis, we make available relevant code portions related to the configuration of the Kafka producers, consumers and topics. The first code snippet belongs to the Shell script `initKafka.sh`, whereas the others are Java configurations from the program's entities.

```
for var in "$@"
do

../../kafka_2.12-2.4.1/bin/kafka-topics.sh --create --bootstrap-server localhost
:9092 --replication-factor 3 --partitions 3 --topic $var

done
```

```
// Code taken from CollectEntity.java

String bootstrapServers = "localhost:9092, localhost:9093 ,localhost:9094";
Properties heartbeatProps = new Properties();
heartbeatProps.put("bootstrap.servers", bootstrapServers);
heartbeatProps.put("key.serializer", "org.apache.kafka.common.serialization.Int
egerSerializer");
heartbeatProps.put("value.serializer", MessageSerializer.class.getName());
heartbeatProps.put("ack", "0");
Properties speedProps = new Properties();
speedProps.put("bootstrap.servers", bootstrapServers);
speedProps.put("key.serializer", "org.apache.kafka.common.serialization.Integer
Serializer");
speedProps.put("value.serializer", MessageSerializer.class.getName());
speedProps.put("max.in.flight.requests.per.connection", 1);
speedProps.put("enable.idempotence", true);
speedProps.put("acks", "all");
Properties statusProps = new Properties();
statusProps.put("bootstrap.servers", bootstrapServers);
statusProps.put("key.serializer", "org.apache.kafka.common.serialization.Integer
Serializer");
statusProps.put("value.serializer", MessageSerializer.class.getName());
statusProps.put("max.in.flight.requests.per.connection", 1);
statusProps.put("acks", "all");

Producer<Integer,Message> heartbeatProducer = new Producer<>(heartbeatProps);
Producer<Integer,Message> speedProducer = new Producer<>(speedProps);
Producer<Integer,Message> statusProducer = new Producer<>(statusProps);
```

```
// Code taken from ReportEntity.java

props.put("bootstrap.servers", "localhost:9092,localhost:9093,localhost:9094");
props.put("group.id", groupName);
props.put("key.deserializer", "org.apache.kafka.common.serialization.IntegerDes
erializer");
props.put("value.deserializer", MessageDeserializer.class.getName());
props.put("enable.auto-commit", false);

String[] tmp = new String[]{topicName};
Consumer<Integer, Message> consumer;
for(int i=0; i<(Integer)nConsumers.getValue(); i++) {
  consumer = new Consumer<Integer,Message>(i, props, tmp, this);
  Thread t = new Thread(consumer); t.start();
  consumers[i] = consumer;
  consumerThreads[i] = t;
}
```

4 Additional Remarks

4.1 Documentation

Our attitude towards the developed code was to ensure it could be applied to other similar scenarios and reused in systems intended to be deployed in real scenarios. With this in mind, we took great care with regards to code readability. By maintaining a code style equal throughout the project and defining intuitive and self-explaining variable and method names, we made the code easy to understand by someone already contextualized with Kafka.

Nevertheless, we wanted to make sure this was also true to someone looking at our project for the first time, so we resorted to the well-known Javadoc (*5*) tool to manage all code documentation. Comments were also added in key points throughout the code, including the scripts.

4.2 Assignment Contributions

As the entire development phase took place in a time where on-site cooperation was not possible, we resorted to online communication platforms to debate decisions and discuss difficulties. Team scheduling allowed us to work on the project simultaneously, so no member suffered from unbalanced workloads. The dimension of the project did not appeal to the usage of repository pull requests and other synchronization tools. However, each small solution was verified and agreed by both team members.

Having said this, it is difficult to isolate what each member actually implemented, as the influence of both is present in all components. Nevertheless, one might say that each had stronger responsibilities on a set of project aspects: Filipe took care of the execution of the individual Java processes and of the Shell scripts, while João developed the Kafka-related classes such as Consumer, Producer and EntityAction; Filipe developed the Python script for generation of `CAR.TXT`, while João developed the Shell scripts for Kafka initialization and deletion; each implemented 2 entities and each wrote a portion of this report; Filipe made sure everything was coherent throughout the report and the code documentation, while João solved the most critical issues regarding the configuration of the topics. In terms of work percentage, we believe it was about 50% for each student.

Conclusions

It is only natural for an informatics engineer to eventually come in contact with Apache Kafka. The wide range of applications of such technology, its high performance in several dimensions and the ease of configuration makes it a valuable tool for several scenarios and with a variety of purposes. This is true for previous software development projects we were responsible for, both in the academical and the professional environments. Nevertheless, prior to this assignment we were never given the chance to truly explore Kafka's capabilities more deeply.

Given such opportunity, there was room for learning valuable details about Kafka. The first thing we understood was the use of different topic configurations according to their purpose and how this could have been useful in past scenarios. Also, the idea of load balancing through the use of multiple partitions within a topic was found very interesting. Having to manage different topics and different message types forced us to find flexible solutions, which was perhaps the most rewarding aspect of the assignment once our tests proved it to be working as intended.

Regarding the overall perspective over the work delivered, we believe it fulfills all of the defined requirements while ensuring some level of redundancy and providing enough resources for a correct reuse of the code in future projects. For future work, it would be interesting to more realistically simulate the smart vehicles and apply some sort of more complex computations in, for example, the Batch Entity.

References

1. Smarter With Gartner, *Staying on Track with Connected Car Security*, `https://www.gartner.com/smarterwithgartner/staying-on-track-with-connected-car-security/`, accessed in April 2020.

2. Apache Kafka, *Apache Kafka: A Distributed Streaming Platform*, `https://kafka.apache.org/`, acessed in April 2020.

3. Óscar Pereira, *SA: Practical Assignment no.2*, University of Aveiro, 2019/20.

4. Oracle, *Swing*, `https://docs.oracle.com/javase/8/docs/technotes/guides/swing/index.html`, accessed in April 2020.

5. Oracle, *Javadoc Technology*, `https://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/index.html`, accessed in April 2020.