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 14, 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.

....

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 (*2*), 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 (*3*) 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 one 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 system 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 system that the car is still connected.
  Format: | car_reg | timestamp | 00 |
- SPEED - message meant to inform the system 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 system 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

....

2.1 Entities

...

2.2 Components

...

2.3 User Interface

...

3 Kafka Infrastructure

As already mentioned, according to (*3*), Apache Kafka is a distributed streaming platform and has three 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 and process streams of records as they occur. Generally used for building real-time streaming data pipelines to establish communication between application and building real-time streaming applications to transform and react to streams of data.

Being more interest in the messaging system option, it is also stated in (*3*), that normally this type of systems follow two 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 models as several positive aspects such as allowing the division of the records in a balanced fashion by the registered consumers, 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 to this approach, but is counterweighted by the fact that there is no way to scale the process, since every message is sent to every consumer.

In this project, Kafka is used as the message system and the backbone for the entire system, making use of both models supported for the messaging. The usage of this platform was not only a constraint in the proposed project description, but was also justified by a analysis our group made over it, that showed that in fact this platform is quite complete and guarantees a number of quality metrics that were quite important and interesting, hence this project to study and deeply understand the influence of these metrics in a system with a quite standard architecture in the IOT area.

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 two approaches: use the instantiation scripts provided with the source code available at (*3*) or use docker as a middleware framework that enables us to abstract the the instantiation process and use kafka as a image. We choose the fist option, which although requesting more work from our side, it enabled us to instantiate the platform with more precision and control.

To do so, it was unfeasible to run manually each necessary command, and for that reason two scripts were developed to aid in both the creation and deletion of the infrastructure necessary for this project. These scripts are available together with the source code, in the folder `scripts`, and the initialization script is the first action to be done when the project is ran, and as expected, the deletion script is the last thing to do when the project is terminated.

In relations to *initKafka.sh*, the initialization script, it is responsible for initializing the Zookeeper, the entity in charge of managing the Kafka Brokers, initializing the Kafka Brokers themselves, entities with the logic of the platform and providing the service, in this case 3 brokers were deployed, and creating the necessary topics. The final order of actions performed is:

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

In relations to *deleteKafka.sh*, the deletion script, it is responsible for killing the processes previously stored in the auxiliary files and deleting the logs generated by the previously mentioned processes. The final order of actions performed is:

1. Fetch processes ids from the auxiliary file
2. Killing the fetched processes
3. Deleting the generated logs

3.2 Topics and Constraints

To establish constraints to the infrastructure consequently to the project, we had three methods, either in the definition of properties for the consumers, in the definition of the producers or in the definition of properties for the topics themselves.

For this project it was established that there should only exist three topics, the *BatchTopic*, *ReportTopic* and the *AlarmTopic*, each one with the objective of establishing communication between the *CollectEntity* and the *BatchEntity*, *ReportEntity* and *AlarmEntity*, respectively.

....

4 Additional Remarks

4.1 Documentation

....

4.2 Assignment Contributions

....

Conclusions

....

References

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

2. 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.

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