Cloud-based anomaly detection and visualization tool for telemetry data

Andrea Lombardi
Università degli studi di Napoli Parthenope
Naples, Italy
lombardiandrea@outlook.it

Ciro Panariello
Università degli studi di Napoli Parthenope
Naples, Italy
pana.ciro0095@gmail.com

Vincenzo Capone
Università degli studi di Napoli Parthenope
Naples, Italy
vincenzo.capone002@studenti.uniparthenope.it

Vincenzo Silvio
Università degli studi di Napoli Parthenope
Naples, Italy
vincenzosilvio@outlook.it

Abstract

In both minor motorsport categories and racing e-sports there seems to be no easily accessible tool to collect, visualize and analyze live telemetry data. Users often have to perform complex installation tasks to run these tools on their own machine, which might not be powerful enough to handle real-time data stream analysis. This work proposes a possible baseline architecture to implement real-time visualization and analysis systems for on-board car telemetry, completely based on cloud and distributed systems technologies and easily accessible through simple tools, like a web browser.

Keywords: cloud computing, telemetry data, real-time, data streams, data visualization, anomaly detection, apache spark structured streaming, apache kafka, apache zookeeper, streamlit

1 Introduction

Telemetry is an ensemble of electronic devices which records performance of engine, status of suspensions, gearbox data, fuel status, all temperature readings including tires temperature, g-forces and actuation of controls by the driver.

Telemetry data is then transmitted to a remote site where it is observed and analyzed to monitor car performance and to detect anomalous behavior. Analyzing telemetry data can give plenty of useful insights to both the driver, like how to improve his driving efficiency, and to the team, like how to optimize the setup of the vehicle, leading to an overall better race pace. Thereby, having a reliable, fast and easily accessible telemetry analysis system is definitely an advantage.

Several telemetry-related tools are available, providing off-the-shelf visualization and more-or-less complex data analysis. Typically, they're installed on the user's own machine, which might not always be capable of ingesting and processing a real-time multidimensional data stream.

Cloud Computing, 2021/2022, Parthenope University, Naples
© 2022

Therefore it might be useful to have a data stream analysis system running remotely on a cloud infrastructure and accessible by client applications. The advantage of a cloud-based system is twofold: firstly, it is advantageous for the user, who doesn't have to install any kind of software and can easily access the system, for example, through a web application. Then, developers can leverage the computational power provided by cloud systems to perform additional analysis on incoming data without making necessary for the user to have a powerful dedicated hardware.

This work proposes a possible baseline architecture to implement a real-time data visualization and analysis tool for on-board telemetry data, completely based on cloud technologies and distributed systems. The proposed system falls under the Software-as-a-Service (SaaS) paradigm and relies on Infrastructure-as-a-Service (IaaS) cloud solutions to provide hardware support to its software components.

2 System overview

The proposed system revolves around three main agents:

- A data source, which continuously produces raw telemetry records at a given frequency.
- An analysis system, which processes incoming raw data and produces analysis results as an additional data stream.
- A client application, that consumes both raw data produced by the data source and analyzed data produced by the analysis system.

These three agents appear to be strongly connected with each other, leading to a monolithic and inflexible system. We can easily organize the system in sub-modules using a publish/subscribe pattern as a middleware to indirectly connect the agents (as shown in Figure 1), making the sub-modules independent from each other and the system much more flexible. A publish/subscribe system exposes the concept of topic as a queue of messages. A client can act as a publisher (or producer), sending messages to a given topic, and/or as a subscriber (or consumer), reading messages from a given topic.

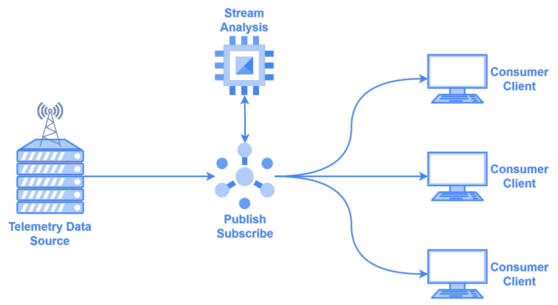

Figure 1. System architecture overview. An incoming arrow into to the publish/subscribe middleware means that the agent produces data on some topic, while an arrow going out from the middleware means that the agent consumes data from some topic. A bidirectional arrow means that the agent is both a producer and a consumer.

There are two main data streams that map on two different topics on the publish/subscribe middleware: a raw data stream, produced by the telemetry data source, and an analyzed data stream, produced by the analysis system. Both the analysis system and client applications are subscribers to the raw data topic, reading incoming telemetry records as they are produced, on that same topic, by the data source. Client applications also subscribe to the analyzed data topic, from which they fetch analysis results as they are produced by the analysis system.

Each agent only needs to know about the publish/subscribe middleware. Interaction between the agents is realized exclusively through topics and changing one agent will not imply changing any of the others.

3 System architecture and technologies

The architecture described in section 2 can be developed in several different manners. The proposed solution is based on distributed system technologies running on cloud-provided clusters. Among all the possible technologies, this work uses the following Apache frameworks: ZooKeeper, Kafka and Spark.

3.1 Apache ZooKeeper

ZooKeeper provides coordination services to distributed applications, enabling developers to focus mainly on their application logic rather than coordination. The ZooKeeper server manages a distributed shared memory structured as a hierarchy of znodes, which resembles a file system structure. Clients can connect to the ZooKeeper server and access this shared memory to coordinate and communicate.

ZooKeeper does not provide coordination primitives directly. Instead, it exposes a file system-like API that enables applications to implement their own coordination tasks using znodes.

ZooKeeper can either work as a standalone server or as a cluster ensemble. The state of a zookeeper ensemble is replicated across all servers, which together serve client requests. In cluster mode, a quorum is the minimum number of servers required to be running and available in order for ZooKeeper to work.

In this work, ZooKeeper is required for running Apache Kafka, which, just like ZooKeeper itself, can run in standalone or cluster mode. Specifically, Kafka relies on ZooKeeper to store metadata about the Kafka cluster, as well as consumer client details, to detect crashes, to implement topic discovery, and to maintain production and consumption state for topics.

3.2 Apache Kafka

Apache Kafka was developed as a publish/subscribe messaging system. Data within Kafka is stored durably, sorted, and can be read deterministically. In addition, data can be distributed and replicated within the system to provide additional protections against failures, as well as significant opportunities for scaling performance.

Data records in Kafka are categorized into topics, that are additionally broken down into several partitions hosted on different nodes.

A single Kafka server is called broker: it receives messages from producers, stores them durably into a topic and responds to consumer requests to receive produced messages. Kafka brokers are designed to operate also as part of a cluster. Within a cluster of brokers, one broker will be elected as the cluster controller, performing additional coordination operations to the usual broker functionalities.

In the proposed architecture, the publish/subscribe middleware is a single-point-of-failure for the entire system: if it breaks, the entire system stops working. With a cluster of Kafka Brokers, it is possible to distribute the publish/subscribe service across different servers, thus having a more resilient and fault-tolerant system.

3.3 Apache Spark Structured Streaming

Apache Spark is a general engine that extends and generalize the map-reduce paradigm, allowing different kinds of processing (e.g., query SQL, text processing, machine learning) in a distributed environment. Spark allows live data streams processing through a component called Spark Streaming.

An evolution of Spark Streaming is Spark Structured Streaming, which is based on the DataFrame API (instead of Spark Streaming's original DStreams API) to process data streams. DataFrames API has the benefits of having lower latency, greater throughput, and guaranteed message delivery. In short, citing Spark Structured Streaming's official guide, *"Structured Streaming provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming"*.

Input data stream is seen as an unbounded table of structured records. Each time a new records is received, a new row is appended to the unbounded table. Structured Streaming does not materialize the entire table. It reads the latest available data from the streaming data source, processes it incrementally to update the result, and then discards the source data. It only keeps around the minimal intermediate state data as required to update the result.

A query on the input will generate the "Result Table". Whenever new rows get appended to the Input Table, the result table eventually gets updated. Whenever the result table gets updated, we would want to write the changed result rows to an external sink (e.g. a Kafka topic, a file, etc.).

For the proposed architecture, Structured Streaming will continuously read data records from a raw data Kafka topic based on some triggering policy that defines when Spark will fetch and then process new records. Processing of these records generates a continuously-updating result table that contains the outcomes of the analysis. Every time the result table is updated, it is written to an output Kafka topic reserved for analysis results.

4 Setup details

The system described in section 2 relies on Infrastructure-as-a-Service solutions provided by Google Cloud. Specifically, all the previously mentioned technologies are supported by Google Cloud Dataproc Cluster services.

The following subsections show how to setup and launch the proposed system on Google Cloud Dataproc clusters. A detailed overview of this implementation is depicted in Figure 2.

4.1 ZooKeeper

In this project, ZooKeeper version 3.7.1 has been deployed on a Google Cloud Dataproc Cluster and configured to work as a three-nodes ensemble over the cluster's internal network.

By default, ZooKeeper looks for its configuration in the `zoo.cfg` file located under the conf directory in the ZooKeeper home. The following example shows a possible configuration:

```
ticktime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=20
syncLimit=5
server.1=hostnameA:2888:3888
server.2=hostnameB:2888:3888
server.3=hostnameC:2888:3888
```

The `dataDir` property specifies the directory under which the ZooKeeper node will store its data. The `clientPort` property is the port at which clients can connect to the ZooKeeper node (default port is 2181). All nodes must be specified in the format `server.ID=hostname:commPort:leaderPort` (default ports are 2888 and 3888 respectively). The same configuration must be replicated on each cluster node. Additionally, for each node, a file called `myid` must be created under the directory specified as `dataDir` and it must contain one single integer number that is the ID of the server (as specified in `server.ID`).

After ZooKeeper has been configured on all the nodes of the cluster, a ZooKeeper server can be launched on each node as part of the ZooKeeper ensemble. In order for ZooKeeper to be available to clients, at least two out of the three ZooKeeper servers must be up ad running (due to quorum policies).

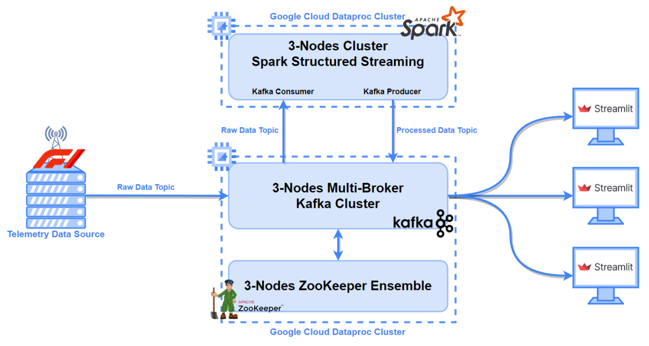

Figure 2. System architecture in details. A three-nodes Dataproc cluster runs a ZooKeeper ensemble. Each node also runs a Kafka broker that is directly connected to the ZooKeeper server running on localhost. This connection allows Kafka to operate as a cluster of three brokers. A data source continuously publishes telemetry records on a raw data topic exposed by the Kafka cluster. Another three-nodes Dataproc cluster runs Spark Structured Streaming, which reads input telemetry data from the raw data topic and writes analysis results on another topic dedicated to processed data. Clients, here implemented as a web application using Streamlit, consume both input telemetry records from the raw data topic and analysis results from the processed data topic.

4.2 Kafka

Once ZooKeeper has been configured, Apache Kafka must be installed on all the nodes of the cluster. For this project, Apache Kafka version 3.1.0 has been used.

Before launching brokers, Kafka must be properly configured on each node, specifying how clients can connect to the Kafka cluster and how Kafka brokers can connect to ZooKeeper. The `server.properties` file is the main configuration file, located in the `config` directory under the Kafka home directory (directory in which Kafka has been installed). For each cluster node the following properties must be specified in the `server.properties` file:

- Set the `broker.id` property to a unique broker ID.
- Set the `listeners` property to *PLAINTEXT://"internalIP":9092*.

- Set the `advertised.listeners` to *PLAINTEXT://"externalIP":9092.*
- Set the `log.dirs` property to a local directory for storing log files (default is `/tmp/kafka-logs`).
- Set the `zookeeper.connect` property to a comma-separated list of `internalIP:port` pairs, including internal IPs of all cluster nodes, paired with the ZooKeeper client port.

Once Kafka has been configured on all nodes and the ZooKeeper ensemble has been launched, brokers can be started and topics can be created and accessed as producers and/or consumers.

4.3 Spark

There's no need to manually install and configure Apache Spark since it is already included in Dataproc Clusters and configured to use YARN as master. From this Spark distribution, Structured Streaming and the DataFrame API can be accessed from Python.

Even though Spark and Structured Streaming APIs natively support I/O operations from and to Kafka topics, additional Kafka dependencies must be specified when submitting Python scripts to Spark with the `spark-submit` command.

5 Anomaly Detection

One of the key features of this work is the possibility of performing real-time custom data analysis. Spark, along with its Structured Streaming module, provides a great platform for high performance data analysis. Despite Spark offers ready-to-use functions, it is still up to the developer to implement the desired algorithms. Here a naive yet effective anomaly detection algorithm is provided as an example of data analysis (see Algorithm 1).

The algorithm works on a lap-by-lap basis: when a lap is completed it is able to tell whether this lap was anomalous if compared with its previous lap. Of all the telemetry attributes, this algorithm considers just car speed and engine RPMs. A lap is represented as a point in a bi-dimensional lap space where its coordinates are given by the average speed and RPMs measured during that lap.

In this space, an Euclidean distance can be measured between any two laps. If the driver is driving consistently, meaning he is constant lap after lap, two consecutive laps will have pretty much the same Speed and RPM traces and their corresponding points will be close in the aforementioned lap space. On the other hand, if a lap is very different from the previous, their corresponding points will be far in the lap space. Hence, a threshold on the Euclidean distance between a lap l and its previous lap $l-1$ can be used to easily discriminate between a regular or an anomalous lap. Euclidean distance between two consecutive laps defines an anomaly score for the most recent lap and is computed as shown in Equation 1.

Algorithm 1 The naive anomaly detection algorithm used in this project

```
$inputDF \leftarrow readStream(broker, rawDataTopic)$

for all Processing triggers do
    1. Compute aggregate mean functions over Speed and RPM, grouping $inputDF$ by lap number. Store the lap-aggregated results in the $groupedDF$ DataFrame.

    2. Compute difference between average Speed and average RPM of a given lap with its previous lap. Store the lap-by-lap differences in the $diffDF$ DataFrame.

    3. To compute Euclidean distance between two consecutive laps, compute the $l_2$ norm for each record in the $diffDF$ DataFrame over its two attributes. Store the result in the $distDF$ DataFrame.

    4. $writeStream(distDF, broker, processedDataTopic)$
end for
```

$$AS_l = \sqrt{(X_l - X_{l-1})^2 + (Y_l - Y_{l-1})^2} \qquad (1)$$

Where X_l is the average speed measured during the lap l, and Y_l is the average RPM measured during the lap l. Figure 3 shows some results obtained with the proposed algorithm on four randomly chosen Formula 1 races. There are clearly some anomalous laps that are not highlighted, but considering how simple the algorithm is, results are quite interesting. A considerable gain in performance can be obtained by tuning the anomaly detection threshold. For each of the four example races, a fixed threshold of 300 was used.

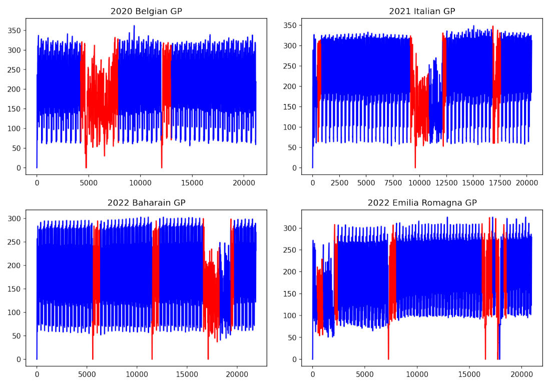

Figure 3. Anomaly detection results computed by applying the proposed algorithm on offline available data. Anomalous laps are highlighted in red. A threshold of 300 was fixed for each race.

6 Conclusions and Future developments

This work describes a baseline architecture to implement a **real-time visualization and analysis systems for on-board car telemetry**, based on distributed technologies like ZooKeeper, Kafka and Spark, and completely deployed on Google Cloud Dataproc clusters, allowing an effortless access through a web application without requiring the user to install any kind of software except the web browser.

What we do propose in this work is not a *ready for deployment* software product but rather it is just a skeleton structure around which developers can build a full cloud-based Software-as-a-Service application.

In the near future we will focus on additional developments that should be considered a must have in order to offer a ready-to-use SaaS product:

- **Multiple users support**: multiple users will be active on the system at the same time, each creating their own private client session, with their data sources and their own data streams and topics, in a completely isolated environment.
- **Multiple driver support**: each user might need to visualize the telemetry data of more than one driver. The system should therefore support multiple telemetry streams, one for each driver, between which client applications should be able to switch.
- **Additional Data analysis techniques**: having just a simple lap-based anomaly detection module might not be enough for most users. Several useful techniques might be added like diagnostic analysis, predictive analysis, etc.

Acknowledgments

The proposed solution has been implemented and tested using real telemetry data from past Formula 1 races provided by the fastf1 python APIs. The project's Github repository can be found at this link.

References

[1] Jules S. Damji, Brooke Wenig, Tathagata Das, and Denny Lee. 2020. *Learning Spark, 2nd Edition.* O'Reilly Media, Inc.

[2] Falvio Junqueira and Benjamin Reed. 2015. *ZooKeeper: Distributed process coordination.* O'Reilly Media, Inc.

[3] Gwen Shapira, Todd Palino, Rajini Sivaram, and Krit Petty. 2021. *Kafka: The Definitive Guide, 2nd Edition.* O'Reilly Media, Inc.

[4] Apache Spark. 2021. *Structured Streaming Programming Guide (3.1.2).* https://spark.apache.org/docs/3.1.2/structured-streaming-programming-guide.html