Noise data collection and analysis
Middleware Technologies for Distributed Systems project

Gibellini Federico, Grilli Francesco, Mannarino Andrea

Politecnico di Milano, A.Y. 2021-22

Contents

1 Introduction

The aim of this project is to build a system that collects noise data for every region of a country via different sources and performs cleaning, enrichment and analysis on such content.

Regions produce data in two different ways:

- Some of them provide open access to data collected by sensors that are deployed on IoT devices which sense the level of noise every 10 seconds and send either the average of the last-six-measurement window or, in case such average exceeds a certain threshold, the last six measurements themselves.
- The other regions, for which such data are unavailable, provide noise levels via simulations based on several parameters including but not limiting to the number of people and vehicles in their area and the unitary noise they are assumed to produce.

These sources send messages containing the data to a backend that should gather them all; thus, the messages should be cleaned, i.e., filtered on the basis of their content in order to remove inconsistent and irrelevant data like, for example, those that carry negative noise values due to a malfunctioning of the sensor.

After that, data should be enriched by being tagged with the name of the closest point of interest, which can be a square, a road or a monument of a specific city, and sent to the analytical section of the system that must periodically compute moving averages and some other metrics on them.

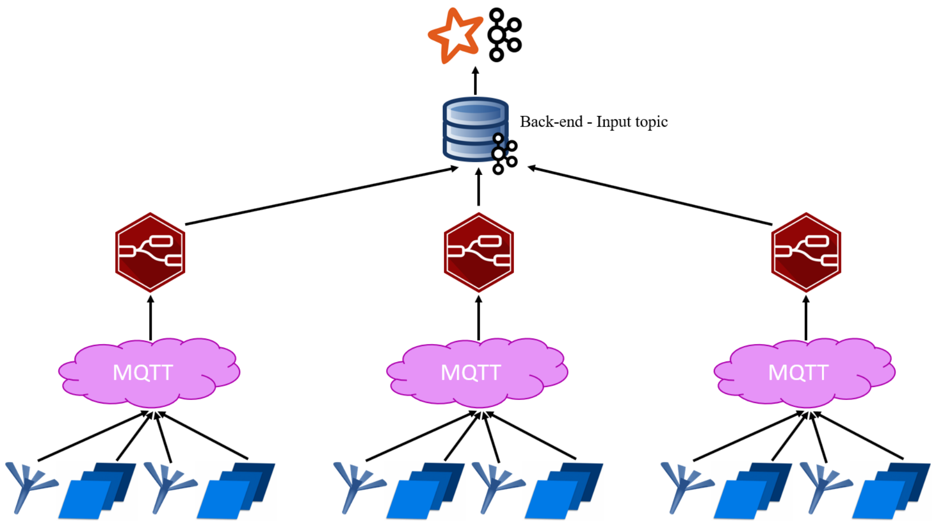

Figure 1: The system's frontend schema

2 Architecture

2.1 Overview

The designed architecture can be interpreted as divided into 3 sections:

1. The **data collection** part is the one in charge of generating all the data that should be handled in the system. It thus includes the MPI simulation and the Contiki[1] IoT sensors[2]. The components in this section interact with the next one with an MQTT-based publish-subscribe mechanism.
2. The **bridge** part consists of the Node-RED flow [3] that gathers and cleans the data coming from the source MQTT topic and publishes them on a Kafka topic.
3. The **backend**, **analytical** part is a system composed of Spark and Kafka environments to perform analysis on the cleaned data.

2.2 Detailed Description

In this section, a deeper description of the whole system behavior is provided. To support it, figures 1 and 2 have been inserted.

2.2.1 Data collection

In this part, the sensors and the simulations produce the measurement data that will be needed by the following sections. As already described, both types of sources produce messages that are published on an MQTT topic, from which they will be retrieved by the flows in the bridge section. In order to allow both types of sources to publish on the same topic, it was chosen to align the format of the sent messages. Thus, every message's payload is a JSON object that includes:

[1]Contiki part is supported by a Node-RED flow for sensor timestamp setting.

[2]Given the unavailability of real sensors, for the project demonstration a simulation of their behavior can be carried out using the Cooja simulator.

[3]Other flows present in the project are meant for sole debugging and visualization purposes.

- An array[4] of noise levels[5].
- The GPS coordinates of the point in which the data were collected.
- The timestamp at which the message is sent.

Contiki sensors The sensors are implemented using an MQTT state-machine which was modified in order to receive a configuration timestamp, working as a sort of "handshake" between the mote and the system. To obtain the timestamp, each mote has to join the network, subscribe to the MQTT "timestamp" topic and, lastly, publish a request on it. At this point, an always-active Node-RED flow receives the request and emits a reply that carries the current timestamp. Upon reception of a reply, any mote unsubscribes from the topic and sets its inner time to the just-received one. Upon collection of the noises, the sensors store the measurements in one array, which is used to compute the average and, thus, to evaluate it with respect to the threshold. Messages with the aforementioned structured are then sent on an MQTT topic to be passed to the next section.

MPI Simulation In regions where sensors are not available, data are obtained by simulating the environment of the interested area. Simulations are based on population dynamics and each noise is influenced by the presence of people and vehicles.

Message Passing Interface (MPI) is a standardized and portable message-passing technology and it is designed to work on parallel computing architectures. Thus, each simulation is parallelized by instantiating multiple cooperating processes and requires the following parameters:

- Width and length in meters of the rectangular area to simulate.
- Number of people and vehicles in the area.
- Noise in decibels produced by each person or vehicle.
- Distance in meters affected by each person or vehicle.
- Moving speed of people and vehicles.
- Time-step in seconds to inform the simulation of the amount of time available to re-compute the position of people and cars.
- Granularity in meters to identify the quantity of data to collect in order to produce a single aggregated message to send to the backend.
- Latitude and Longitude of the upper left corner of the rectangular area.

Each simulation repeatedly performs three main steps:

1. **Noise computation**: in this step, each process computes the noise contribution of a small number of people and vehicles in the entire area. Each person and vehicle is simulated independently and has an area of influence around him: here, the noise generated by each element is assumed to decrease as the distance from it increases. Each noise is then calculated by summing up all noises coming from the surrounding area and it is then used to fill a matrix that represents the whole area of the simulation.

[4]The choice of always sending an array is motivated by the aim to handle both the cases in which one and six values are sent.

[5]Measurements in decibels (dB).

2. **Position re-computation**: at this point, each process recomputes the position of each person and vehicle under its own control using simple population dynamics[6]. The tasks performed at this stage are the most complex and heavy-computational-loaded of the whole simulation. To reduce the computation, the load is divided into parallel processes in order to maximize the performance. Each process is going to simulate just a subgroup of the entire population.

3. **Noise gathering**: lastly, the master process oversees the gathering of all the matrices produced by the processes and sums them all up. Once the final matrix is computed, the master has all needed data to publish messages[7] to the MQTT topic.

2.2.2 Bridge Section

This part performs gathering, cleaning and enrichment on the data coming from the sources. The flow which is in charge of doing this obtains the messages by subscribing to the same MQTT topic on which they were published by the previous section.

The first operation that is performed after content retrieval from the topic is cleaning. During this phase, some messages are discarded; particularly, the ones that:

- Don't carry relevant noise levels, that is, the ones without noise array, with an empty one or with negative values in it.
- Don't have a timestamp or that carry a negative number for it.
- Carry illegal GPS positions; these messages are such that their latitude value is not between –90 and 90 degrees or their longitude value is not between –180 and 180 degrees. Missing-GPS-position messages are discarded as well.

After doing this, the enrichment function node associates to each clean datum its closest point of interest. In order to do this, the node is required to interact with the database in which point-of-interest information is stored. Thus, upon booting, such node initializes an array of objects that represent the points of interest and that have as fields, for each of them, its name and its GPS coordinates; the latter are needed to compute the distances from each point of measurement, whose latitude and longitude are provided within the clean message. Computed distances are beeline and obtained via the application of the haversine function[8].

Once the shortest distance has been calculated, the message undergoes a final validity check, in which the datum is kept only if the just-found distance is under a certain threshold; this method allows for the discarding of those messages that associate a noise level to a point that is too far from the closest point of interest; such measurements were reckoned not to provide a meaningful piece of information.

The result is published on the input Kafka topic, from which it will be read by the next section to perform analysis.

2.2.3 Analysis Section

This part is the one that computes the moving averages and the additional metrics. The core idea at its basis is to have multiple Spark structured streaming jobs compute a flow of subsequent aggregations starting from the input data. A schematic representation of this is provided in Figure 2. As the picture shows, it's possible to identify 3 aggregation flows in the back-end system:

[6]In this case we are using a constant speed for all people and cars without interaction between them.

[7]Position in each message is computed leveraging the last two required parameters.

[8]For more about the haversine function follow the url: `https://community.esri.com/t5/coordinate-reference-systems-blog/distance-on-a-sphere-the-haversine-formula/ba-p/902128`

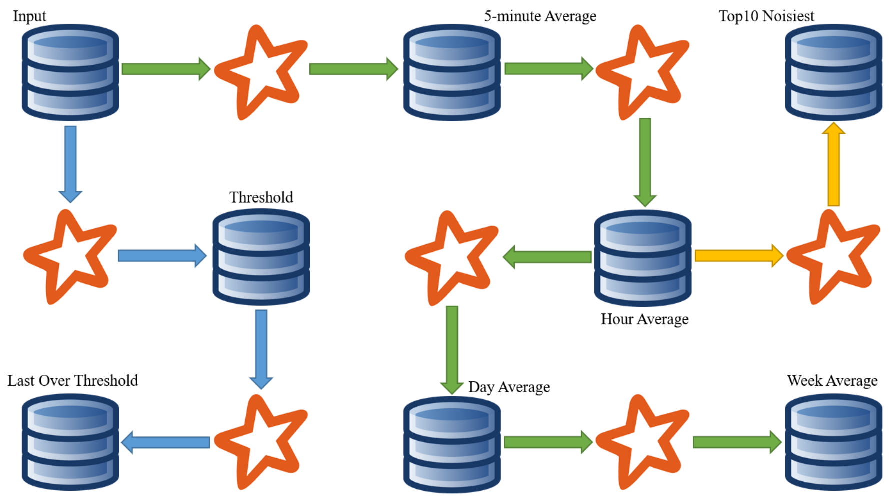

Figure 2: The system's backend schema

1. The green flow is the one that is responsible for the moving average computation. It performs consecutive averages on the very just-produced averages with increasing granularity up to a one-week aggregation.

2. The blue flow is used to perform the longest-streak query. Firstly, a Spark job selects from the Input topic all the values that are over a specific threshold[9]; then, if the timestamp associated to the incoming message is greater than the highest one associated to the same point of interest from the Last Over Threshold topic, the message itself is forwarded to the just-mentioned topic. This topic leverages Kafka's log compaction to speed up queries thanks to the periodic key-based removal of outdated messages.

3. The yellow flow performs the top 10 query for the noisiest points of interest over the last hour. To compute this query, the last hour is identified from the Hour Average topic and used to select the values with that timestamp; then, such values are sorted by descending noise and the top 10 of them are published on the Top10 Noisiest topic.

3 Design Choices

In this section, the three main design choices are presented.

3.1 The choice of performing enrichment on Node-RED instead of the backend

At the basis of this choice there was the fear of a severe bottleneck in case of a direct connection of the sources to the back end: indeed, in such a situation the input topic would have been flooded with messages that were often incomplete or useless for the computation. Instead, exploiting the edge computing paradigm, for which Node-RED is one of the core technologies, and, thus, introducing a bridge section to clean data closer to the sources, the backend is left the single responsibility of computing analytics on legit data.

[9]Threshold was assumed to be fixed and set at the launch of the Spark application.

3.2 The choice of Spark structured streaming to compute analytics

This choice was motivated by the recognition that Spark provided all the necessary functionalities to compute the requested metrics in a streaming environment, which seemed a natural assumption given the reckoning that lots of data would be injected continuously in the system. In addition to that, input data were thought to be featured by a specific structure that requires, at least, the triple <point of interest, noise value, timestamp>, which was thought to be simply handleable considering each element as a column in a relational database table.

Other alternatives have been discarded:

- Spark streaming was not considered a viable solution as it was reckoned not to be compliant with the structure of the incoming data, being it an unstructured paradigm; on top of that, this Spark approach doesn't support the handling of event time and late data, which is instead tackled by the structured streaming paradigm with the introduction of watermarks. Spark batch was refused for the same reason.
- Kafka was not reckoned to be a viable solution because, considering the structure of its topics and the way in which consumers and producers are built, it would have been particularly difficult to implement the mechanism to handle sliding windows; the same issues, then would have arisen in the approach to late data and event time. All these functionalities, instead, are already implemented in Spark which was also thought to be a better solution to work in a big data environment.
- The Akka solution was immediately discarded because it showed issues and complexities in the handling of the window averages, especially time-based ones.

3.3 The choice of Kafka to support Spark computation

Given the above reasons that motivated the introduction of Spark and Node-RED, a choice about their interface had to be made; the best solution was reckoned to be the introduction of Kafka. Before opting for that approach, all possible alternatives were taken into consideration and discarded:

- The simplest idea would likely have been the one of dumping all the cleaned data on some file; however, this solution was considered to be definitely not scalable as the writing of input files would have led to the creation of colossal documents to store on some secondary memory device, leading to a useless waste of memory resources.
- Another option could be using a socket input for Spark; however, this case would have created a bottleneck on the socket, requiring all the messages to go through a unique path (the socket, indeed).
- A non-Spark-native solution was considered as well: as Node-RED is can work easily and well with the MQTT protocol, this very protocol was thought to be an acceptable solution, because it allowed to decouple the Node-RED part from the Spark one at the price of being less Spark-fault tolerant than the Kafka solution. However, this solution encountered insurmountable difficulties in the actual implementation, as all the available libraries to provide Spark this additional network feature are outdated or apparently incompatible with the structured streaming paradigm.
- The Kafka solution was thus recognized to be the best: it allows removal of the bottleneck because messages can be sent to the input topic by several producers, it introduces decoupling between the two technologies it's meant to interface, it is more Spark-fault tolerant because it can retain messages on the topic for a certain period, and it provides also better semantics when integrated with Spark (according to Kafka documentation, it provides at-least-once semantics).