AY 2021/2022

Politecnico di Milano

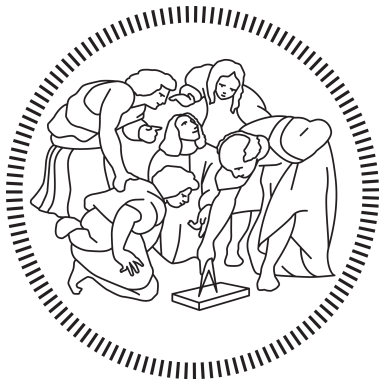
Middleware Technologies for Distributed Systems

Project report of

Simulation and Analysis of Noise Level

Students
Matteo Beltrante Marco Bendinelli Simone Berasi

Supervisors
Luca Mottola Alessandro Margara



1 Introduction

1.1 General description

The project consists in the realization of a System that studies the noise levels registered in a country. In some regions of the country it has been possible to locate IoT devices that use acoustic sensors to detect the sound level, in others these devices are not present and therefore the noises are simulated via software.
We take Italy as a reference and we divide it into two macro regions: the North and the South; in the first one the sensors are available, while the second one relies on a simulation. Each point of the two regions is represented by the geographic coordinates.

1.2 Sensor data

The IoT device applies a sliding window to the last six readings, which are detected by the sensors every ten seconds, and it calculates the average of them. If the average exceeds a certain threshold K the readings of the current window are sent to the back-end, otherwise the average is sent.
Note1: the readings are dispatched on the regular internet through static IoT devices acting as IPV6 border routers.
Note2: the IoT devices are mobile and they know their location

1.3 Simulated noise levels

The computer simulation depends on the noise generated by people and vehicles.

1.4 Back-end

1.4.1 Data cleaning and enrichment

The back-end, after receiving all the noise level data, annotated with the geographical area in which they were measured (in our implementation the Italian geographic coordinates), it performs, before storing the values, the following pre-processing steps:

- it eliminates the invalid and the incomplete readings. E.g. noise levels below the zero or missing geographical areas;
- it associates to each reading the name of the closest point of interest.

After those, it saves the device and the closest point of interest information, in a local data-set.
Note that the coordinates of the points of interest are stored in a static data-set, containing all the Italian provinces.

1.4.2 Data analysis

So, the back-end, having cleaned the data and saved it correctly, can now periodically calculate some important metrics which we see later.

2 Architecture

2.1 IoT device

The IoT devices are implemented with the Coniki-NG operating system and they use the MQTT protocol to send the readings to the back-end.
The device is composed by two processes: the mqtt_client_process and the gps_process. The mqtt_client_process consists on the implementation of a Finite State Machine that is periodically triggered by a timer: it handles the connection to the local Mosquitto broker and all the MQTT events. Moreover, it retrieves the readings by a virtual sensor that is included by the IoT device. The virtual sensor is a program that can generate different type of values within a default bound.
The messages, with topic = *iot/italy/noiseLevels/readings* and *QoS = 0* to minimize the workload of each node, are published to a local broker that uses the bridge mode to connect with a remote broker (*mqtt.neslab.it:3200*) and forwarding to it the MQTT messages. The simulation is done in Cooja OS and it is composed by two different devices: the RPL border router and the sensor devices. The RPL border router is an IoT device that is implemented for bridging the IoT network with the external internet. In order to simulate the moving ability of the devices we use the following strategy: every x seconds a different RPL Border Router's process asks Cooja for the updated coordinates of all the connected devices; a snippet of JS code written in the Cooja script editor, takes care to send the coordinates in the simulation to all the connected motes; finally, the coordinates are intercepted by the *gps_process* that checks continuously for messages received through the serial port.

2.2 Simulation

The simulation, that is done by using the **MPI** standard, takes in input the following **parameters**:

- the number of people / vehicles;
- the width and the length of the region;
- the level of noise produced by each person / vehicle;
- the distance affected by the presence of a persona / vehicle;

- the moving speed for an individual / a vehicle;
- the time step of the simulation (t).

If the correct command line arguments are not passed, some default values are used.
Note: from now on I use the word *element* to refer to both people and vehicles, and therefore the formulas that we will see are applied separately to both people and vehicles.
Before the simulation starts, each process takes over a fixed number of elements which is calculated as $number\_of\_els/number\_of\_processes$ (the remaining number is entrusted to the root process) and generates a random position and speed of each element. Now, since the simulation can begin, for an arbitrary number of times the following steps are performed:

1. each process initializes the region by setting at 0 the noise in every square meter;
2. each process computes the noise caused by people and vehicles in the region: for each spatial point in which the element generates noise (by considering the distance affected by the presence of such element), the process computes the logarithmic sum of the noise at the current point of the region and the element noise;
3. each process calculates the new position of the elements taking into account the duration of the simulation (t) and their current position and speed. The speed also changes only in the direction if an edge is hit;
4. the root process gathers together the simulated noise levels by all the processes through the *MPI_ Gather* function. So, it sums the noises of each process and prints the result into a CSV file.

2.2.1 Performance

The following tests have been run without producing the output file, because it is done by the root process only and the time needed is too big to notice great improvements from multi-processing. It is worth to mention that this code scale very well on the number of elements, not so well on the region dimensions, this because the final computation is done by the root process. Another version has been proposed, which scale well in the region dimensions too but is more dependent on network (more and bigger messages are passed around). All benchmarks have been run with the following parameters:

- **number of people = 10000;**

- **number of people = 10000;**
- **region width = 4000;**
- **region length = 4000;**
- **area affected by person = 100;**
- **area affected by vehicle = 100.**

Presented version:

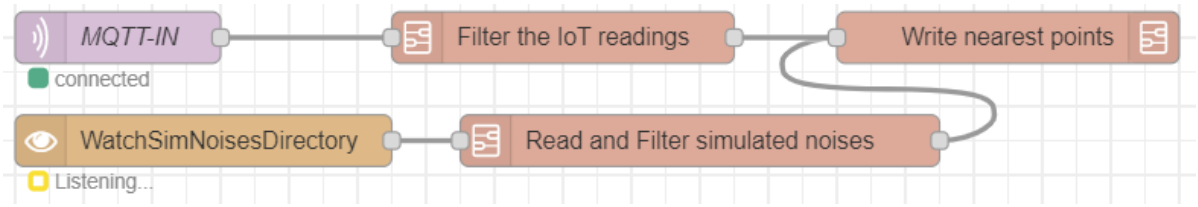
1 core	2 core local	2 core distributed	4 core local
74.87 s	47.86 s	46.69 s	42.46 s

Alternative version:

1 core	2 core local	2 core distributed	4 core local
79.21 s	47.82 s	54.11 s	29.92 s

2.3 Back-end

2.3.1 Node-RED



The node-RED flow, has shown in the image, contains the following nodes/sub-flows:

- **MQTT-IN**: it connects to the *mqtt.neslab.it:3200* broker and subscribes to messages from the topic *iot/italy/noiseLevels/readings*;
- **Filter the IoT readings**: it filters the feasible readings and it attaches to the messages the current date and time;
- **WatchSimNoisesDirectory + Read and Filter simulated noises**: it filters the values read from the CSV file generated by the simulation;
- **Write nearest points**: it is used by the 2 sub-flows to read the static data-set and to find the nearest point. It contains the following nodes/sub-flows:
  - **Read Points of Interest info sub-flow**: it reads from a static data-set the pointOfInterest,latY,longX tuples;

- **Join**: it joins the input messages ((Iot-reading || simulated noises) + points of interest information) into a single message;
- **Find nearest point function**: it calculates the distances between the points of interest and the IoT device / simulated points. It takes the minimum distance(s) and it finds the nearest point of interest(s);
- **csv node + Write file node**: it inserts into a CSV file the final tuples: longX,latY,noiseValues,nearestPoint,dateTime.

2.3.2 Apache Spark

Before computing the metrics, we create a new SparkSession and we populate a Dataset with the content of the CSV file that was previously saved by the back-end and that contains the tuples: *longitude, latitude, noise, point_of_interest, time*. Then, we create a new Dataset by replacing the *noise* column of the previous Dataset, with the average of the noise values. Now the following metrics are calculated:

- the average of the noise levels in the last hour, day and week for point of interest, by applying an appropriate filter to the rows and by exploiting the *avg* aggregate function;
- the top 10 points of interest with the highest level of noise over the last hour: we partition the data-set by point of interest, we take the highest value for each of them and we select the top 10 points;
- the point of interest with the longest streak of good noise level:
  we partition the data-set by point of interest, each partition is sorted in descending order by time and by exploiting the *row_number()* window function, for each partition we add a column with the sequential row number (seq). We filter all the values that are below T and we add again a sequential row number column (new_seq).
  Then, we remove all the tuples that have $seq! = new\_seq$ in order to reset the streak whenever the point noise exceeds T.
  E.g., if we have 5 Pisa tuples and the first one contains a value greater than K, the size of the streak is zero.
  Finally, we take the oldest date of the table and we return the streak of all the points that have such date.

3 Design Choices

- For the network protocol of the IoT devices we decided to use MQTT for its simplicity and because it works well on devices with resource constraints;

- about the simulation we used MPI in order to parallelize the computation of a compute-intensive task, i.e., the simulation of the noise levels;

- about the back-end we used Node-RED because its browser-based flow editor: it makes easy wiring together flows using a wide range of nodes. Moreover, each node implements a specific functionality that allow us not to implement it by hand;

- we chose to use Spark because we needed to process a large volume of data. In particular, we used Spark SQL instead of Spark Streaming, that might have been the obvious choice, because we also wanted to be able to calculate past metrics and because we wanted to make it possible to integrate new metrics in a simple way: our data-set can be easily queried by exploiting the features of the Spark SQL module.
  Also it is worth noting that Spark SQL is much cheaper than Spark Streaming as it does not require to be constantly running but can be executed only when needed.