Networked Software for Distributed Systems
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

Project 1: Contiki-NG, Node-RED, Apache Spark

Carlo Laurenti Argento
Gianluca Scanu
Federico Zanon

A.Y. 2023-2024

Contents

Introduction

Assignment

People roaming in a given location carry IoT devices. The devices use the radio as a proximity sensor. Every time two such devices are within the same broadcast domain, that is, at 1-hop distance from each other, the two people wearing the devices are considered to be “in contact”. When at least three people are in contact with each other, that is, each person is in contact with the other two, a “group” is formed. Formation of a group must be reported to the back-end as soon as possible. When a further person joins the group, that is, she is in contact with every other current member of the group, a notification carrying the new cardinality of the group is reported to the back-end. Likewise, when a person leaves a group, a notification carrying the updated cardinality of the group is reported to the back-end. The back-end should periodically compute statistics on every group, including i) lifetime, that is, for how long the group existed, and ii) average, maximum, and minimum cardinality so far. Optional: consider every person to be characterized by her nationality and age. The back-end should also compute:

- A 7 days moving average of participants’ age per nationality, computed for each day (for a given nationality, consider all the groups in which at least one participant is of that nationality).
- Percentage increase (with respect to the day before) of the 7 days moving average, for each day.
- Top nationalities with the highest percentage increase of the seven days moving average, for each day.

Assumptions and Guidelines

- The IoT devices may be assumed to be constantly reachable, possibly across multiple hops, from a single static IoT device that acts as a IPv6 border router, that is, you don’t need to consider cases of network partitions.
- The IoT part may be developed and tested entirely using the COOJA simulator. To simulate mobility, you may simply move around nodes manually.
- Monitoring of the group membership may be implemented with periodic functionality. The case of a person leaving a group may be handled with a timeout.

Technologies

The project uses three technologies to implement three different features:

- **Contiki-NG:**
  - Cooja is used to simulate the environment where IoT devices live.
  - Contiki-NG is used to program the behaviour of the motes living in Cooja.
- **Node-RED:**
  - Used to implement the back-end of the application.
  - Receives messages from an MQTT broker and process them.
- **Apache Spark:**
  - Used to compute statistics on collected data.
  - It uses a *.csv* file written by the back-end.

Contiki-NG

Design

The environment was developed in Cooja and motes represent people carrying the IoT devices. The simulation uses the *Constant Loss Unit-Disk Graph Model (CL-UDGM)* for the wireless propagation, with an interference range of 0 meters. This setup ensures a reliable communication between motes within the same range. According to the *Routing Protocol for Low-Power and Lossy Networks (RPL)*, all the motes are connected in a tree-shaped topology network where the root of the tree has a direct Internet access. Communication between motes occurs via MQTT events and UDP protocol. Finally, we chose to use COOJA motes due to the lack of memory size restrictions for the loaded software. The simulation involves two distinct types of motes: an `RPL BORDER-ROUTER` and an `MQTT CLIENT`.

Assumptions

- To assign a nationality and an age to each person, the mote randomly extracts these information from a finite set of possibilities and immediately report them to the back-end.
- Each group is identified by a leader elected during the creation. In fact, identifying a group by all its members can be problematic: for example if a group were divided exactly in half it would be difficult to establish which of these two halves is the original group and which is the half that has left. The presence of a leader would always allow the identification of the original group.
- Since the assignment doesn't specify how a person should leave a group, we assume that this happens if the person moves away from the communication range of the group leader. Similarly to before, this strong assumption allows us to precisely identify the person who leaves a group. Otherwise, if we had assumed that a person leaves the group as soon as he loses contact with any member, it would be difficult to establish who actually left the group: for example if two members moved away from each other but remained within the range of the other members, it would be difficult to determine who left the group and who remained.

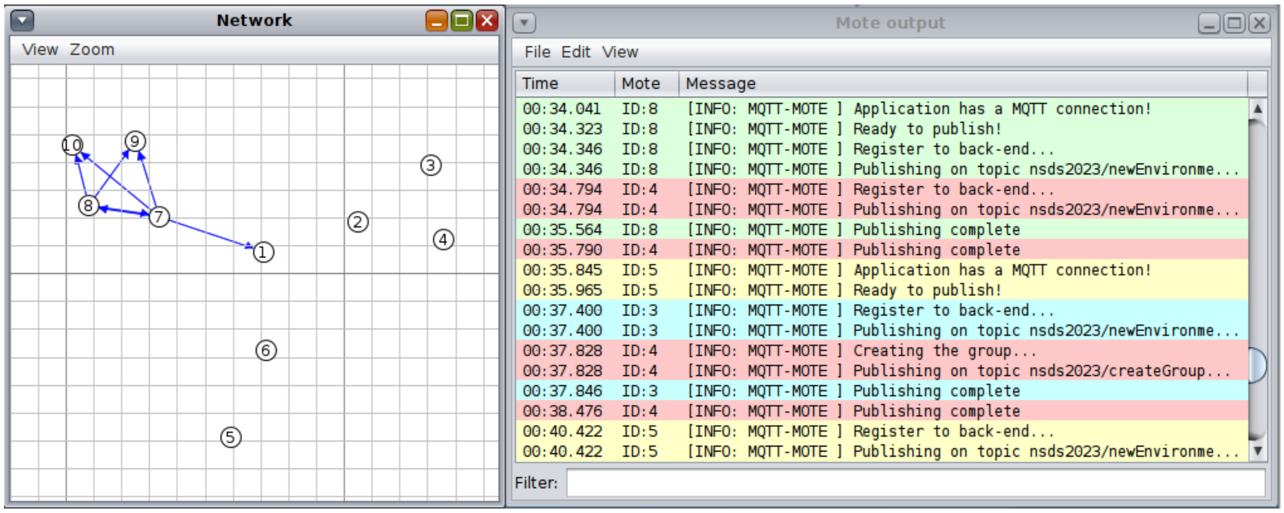
RPL Border-Router

The `RPL BORDER-ROUTER` is a specific mote that acts as a *man in the middle* to bridge the network simulated in Cooja with the external Internet. It consists of a single thread which allows communication with the external Internet and it's also configured as the root of the RPL network. This mote is required to connect the IoT devices to the MQTT broker and make them publish on MQTT topics.

MQTT Client

The `MQTT CLIENT` is the general mote which represents an IoT device in the RPL network. It's the core of the project and consists of two parallel protothreads:

- `MQTT CLIENT PROCESS`: This process consists of a Finite State Machine (FSM) which is periodically triggered by a timer. After configuring the client, it connects the local Mosquitto broker to the one hosted on the Internet and then it takes care of handling MQTT events. The complete and detailed structure of the FSM is shown in the following image:
- `GROUP PROCESS`: This process describes the management of groups by an IoT device and periodically performs different operations based on its state. Initially the mote is in `REGISTER` state and publishes its information regarding nationality and age (useful for Spark analysis) on the `nsds2023/newEnvironmentPerson` topic. Then it goes into `SEARCHING` state: in this state the mote sends a list of its neighbors via UDP to all its neighbors. The callback function performed by the motes during the reception of the UDP message checks if there are any neighbors in common between those received and its own, in case there is a neighbor in common the group can be created (consisting of itself, the sender and the neighbor in common). Then a simple leader election process begins based on the highest ID (IP address) of the identified group. Finally the elected mote publishes the new group on `nsds2023/createGroup` and enters the `LEADER` state. In this state the mote is responsible for accepting new members and checking whether all the members of its group are still among its neighbors, otherwise it communicates the change in cardinality by publishing on the topic `nsds2023/changeCardinality`. If the cardinality falls below 3, the group is eliminated and the leader returns to the `SEARCH` state. To update the table of neighbors, a timer is used that indicates the last transmission received from a neighbor. If no transmission is received within a predefined time, the mote is assumed to to be out of the communication range and is removed from the neighbors table.



```
Time       Mote  Message
00:34.041  ID:8  [INFO: MQTT-MOTE ] Application has a MQTT connection!
00:34.323  ID:8  [INFO: MQTT-MOTE ] Ready to publish!
00:34.346  ID:8  [INFO: MQTT-MOTE ] Register to back-end...
00:34.346  ID:8  [INFO: MQTT-MOTE ] Publishing on topic nsds2023/newEnvironme...
00:34.794  ID:4  [INFO: MQTT-MOTE ] Register to back-end...
00:34.794  ID:4  [INFO: MQTT-MOTE ] Publishing on topic nsds2023/newEnvironme...
00:35.564  ID:8  [INFO: MQTT-MOTE ] Publishing complete
00:35.790  ID:4  [INFO: MQTT-MOTE ] Publishing complete
00:35.845  ID:5  [INFO: MQTT-MOTE ] Application has a MQTT connection!
00:35.965  ID:5  [INFO: MQTT-MOTE ] Ready to publish!
00:37.400  ID:3  [INFO: MQTT-MOTE ] Register to back-end...
00:37.400  ID:3  [INFO: MQTT-MOTE ] Publishing on topic nsds2023/newEnvironme...
00:37.828  ID:4  [INFO: MQTT-MOTE ] Creating the group...
00:37.828  ID:4  [INFO: MQTT-MOTE ] Publishing on topic nsds2023/createGroup...
00:37.846  ID:3  [INFO: MQTT-MOTE ] Publishing complete
00:38.476  ID:4  [INFO: MQTT-MOTE ] Publishing complete
00:40.422  ID:5  [INFO: MQTT-MOTE ] Register to back-end...
00:40.422  ID:5  [INFO: MQTT-MOTE ] Publishing on topic nsds2023/newEnvironme...
```

Figure 2.1: Screenshot of an instant of the Cooja simulation

Node-RED

NodeRed is the framework which contains the backend of the project. Three different flows process MQTT messages incoming from COOJA simulation and a fourth one processes the data to be analyzed via Apache Spark. MQTT communication follow the publish and subscribe messaging pattern and each flow listen on a specific topic. The backend then computes statistics and stores all incoming information into two different files: `environment.csv` and `output.csv`. The first three flows, and their corresponding three topics are:

- **newEnvironmentPerson**: a message containing information regarding a newly spawned mote: *IP*, *nationality*, *age*. The message is processed and, after checking that the IP hasn't been already stored, is saved as a new line in the `environment.csv`.

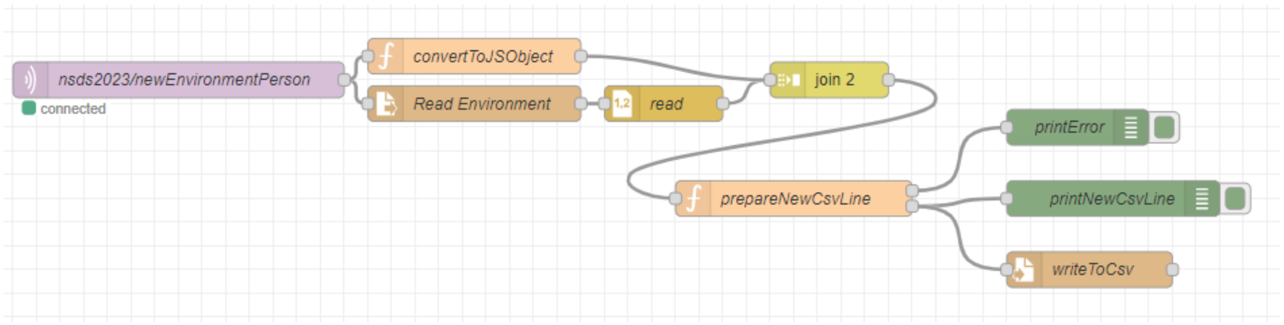


Figure 3.1: The newEnvironmentPerson flow

- **createGroups**: a message containing information on the creation of a group. It features: a *teamLeaderIP* highlighting the IP of the team leader, a *listofIPs* highlighting the IPs of the group members and an integer named *cardinality*. The message is processed and, after checking that the group hasn't been processed yet, is saved as a new line in the `output.csv`. **Computing statistics**: whenever a valid message is received on this topic, the backend also initializes the statistics of the newly formed group.

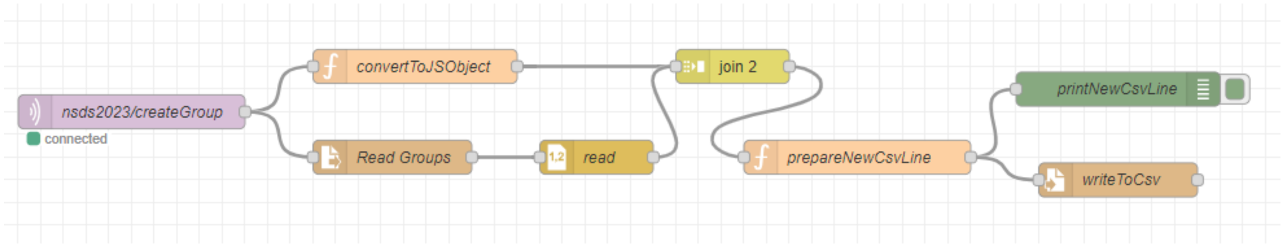


Figure 3.2: The createGroups flow

- **changeCardinality**: similarly to the previous topic, it receives a message containing three objects: *teamLeaderIP*, *listOfIPs*, *cardinality*. The cases might be two: either the group changed its old cardinality ($cardinality >= 3$) or the group has just been eliminated ($cardinality < 3$). In the first case, after checking that the group exists in the `output.csv`, the line describing that group is updated. In the second case, after checking that the group exists in the `output.csv`, a timestamp in the *dateEnded* column of the line describing that group is added and *lifetime* is computed. **Computing statistics**: whenever a valid message is received on this topic, the backend also computes and updates the statistics of the group.

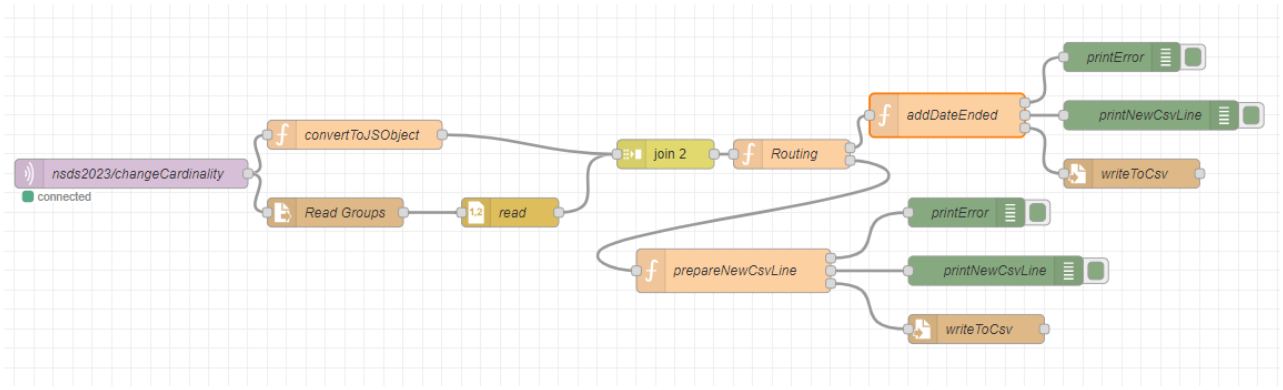


Figure 3.3: The changeCardinality flow

Lastly, there is a fourth flow that prepares the data for the Spark analysis. The flow is triggered every 24 hours, and its function is to read from `output.csv` and `environment.csv` to retrieve the age and nationality of all the motes living in the COOJA environment that are members of a group on the specific trigger date. Then, for each mote, the flow appends to another file, which we call `to_spark.csv` a new line containing the nationality and the age of each mote. In this way, this last flow prepares the data to be queried by the last technology of this project: Apache Spark.

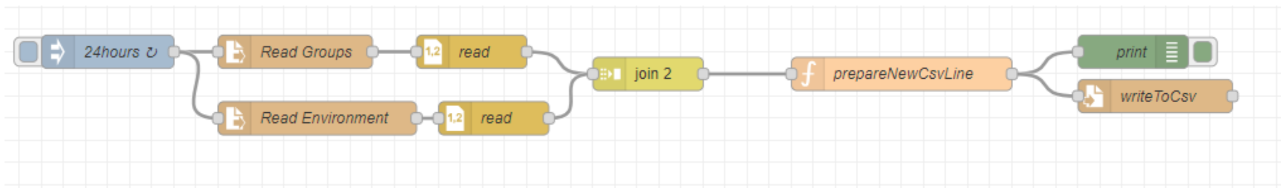


Figure 3.4: The Spark flow

Apache Spark

The optional task of the project required the back-end to also compute:

1. Compute a 7-day moving average of participants' age per nationality, computed for each day.
2. Calculate the percentage increase (with respect to the day before) of the 7-day moving average for each nationality.
3. Determine the top nationalities with the highest percentage increase of the seven days moving average for each day.

To tackle this task, we decided to use a technology known as Apache Spark, and in particular the Spark SQL engine. This engine was chosen as all three requests could be translated in a SQL query to be executed on the output dataset of the fourth NodeRed flow. The main assumption is that the backend (NodeRed) will update, every 24h, a specific `.csv` file, which will the be queried, say every few days, by the Spark engine. This assumption leads to a query being made on a static dataset, therefore justifying the usage of the Spark SQL engine, opposed to a streaming context, which could have led to the usage of the Spark Stream processing engine. As pointed out earlier, NodeRed's "Spark flow" saves the data into the `to_spark.csv` file, which is therefore the file on top of which the SQL queries will be executed.

The first step of the Spark engine is to import the `to_spark.csv` file and create the dataset `data`, which contains all the information needed for the three queries. Then the three queries can be executed.

Query 1

This query is divided into two steps. First of all, a window is created, so to consider only the dates spanning 1 week. Then, the query instead computes the rounded "moving average" of the participants, by nationality. The moving average at day D of the previous n days is defined as:

$$\mathrm{MA}_D = \frac{1}{n}\sum_{i=1}^{n} y_i$$

where y_i is the value of the series at day i, and the summation sums the values from the previous n days. Here follows the Spark code to implement such formula.

```
WindowSpec windowSpecMovingAverage = Window
            .partitionBy("nationality")
            .orderBy("date")
            .rowsBetween(-6, 0);

final Dataset<Row> movingAverage = data
            .withColumn("movingAverage", round(avg("age").over(windowSpecMovingAverage), 2))
            .drop("age")
            .orderBy("date", "nationality");
```

Query 2

This query is once again subdivided into two steps. Note that we will now operate on the `movingAverage` object directly. First, a window is created, to partition and order the data. Then, the query computes the percentage increase between the two moving averages, whenever two consecutive moving averages exist. The formula to compute the percentage increase is the following:

$$\text{Percentage Increase} = \begin{cases} \left(\frac{\text{MA}_n}{\text{MA}_{n-1}} \times 100\right) - 100 & \text{if } \text{MA}_{n-1} \text{ is not null} \\ \text{null} & \text{otherwise} \end{cases}$$

Here follows the Spark code to implement such formula.

```
 WindowSpec windowSpecPercentageIncrease = Window
             .partitionBy("nationality")
             .orderBy("date");

  Dataset<Row> percentageIncrease = movingAverage
             .withColumn("previous", lag("movingAverage", 1).over(windowSpecPercentageIncrease))
             .withColumn("percentageIncrease",
                    when(col("previous").isNotNull(),
                           (col("movingAverage").divide(col("previous")).multiply(100)).minus(100))
                           .otherwise(null))
             .withColumn("percentageIncrease", round(col("percentageIncrease"), 2))
             .drop("previous")
             .drop("movingAverage")
             .orderBy("date", "nationality");
```

Query 3

This query is once again subdivided into two steps. Note that we will now operate on the `percentageIncrease` object directly. First, a window is created, to partition and order the data. Then, with the filter method of the SQL query one extracts the top 2 nationality with the highest percentage increase, for each day.

```
WindowSpec windowSpec = Window
        .partitionBy("date")
        .orderBy(desc("percentageIncrease"));

Dataset<Row> rankedNationalities = percentageIncrease
        .withColumn("rank", rank().over(windowSpec))
        .filter(col("rank").leq(2)) // Keep only the top 2 nationalities for each day
        .drop("rank")
        .orderBy("date", "nationality")
        .orderBy(desc("percentageIncrease"));
```

Finally, the Spark engine prints to console all of the results of the queries.