UNIVERSITÀ DI PISA

University of Pisa

Laurea Magistrale (MSc) in Artificial Intelligence and Data Engineering

Project

Distributed Systems and Middleware technologies

Covid-Tracker

Academic year 2020-2021

Lorenzo Bianchi, Valerio Giannini, Alessio Serra

Github: https://github.com/ValeGian/DSMT_CovidTracker

Index

Covid-Tracker

Introduction

The project consists in the creation of a hierarchical network which allows to track data (number of positives, negatives, swabs and dead) inherent to the Covid-19 pandemic, and to elaborate them in order to produce statistical aggregations.

We provide a three layer hierarchy composed by:

1. A national node.
2. Three area nodes (north, center and south).
3. Twenty region nodes, one for each region.

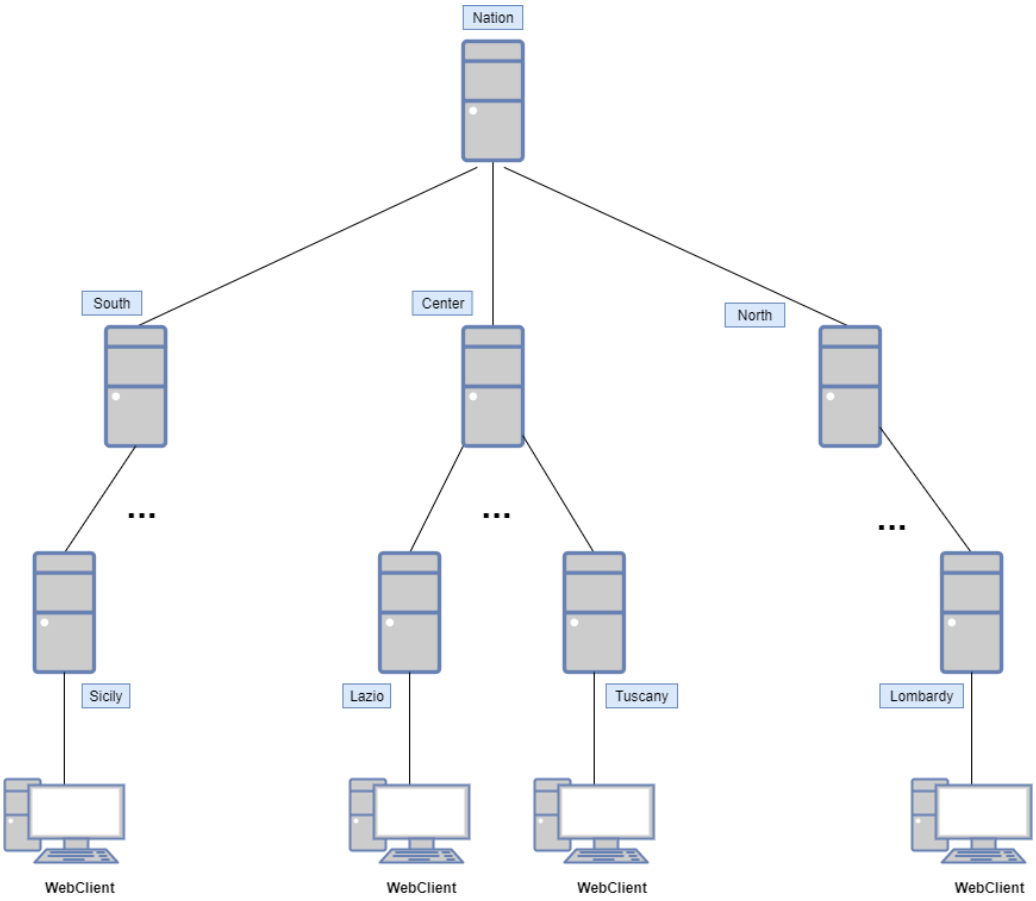

Figure 1 Architecture of the whole system.

Web-Client

A web application provides an interface to connect to one of the twenty region nodes:

- A client can connect to only one region at a time.
- A region can handle more than one connection at the same time.

After the connection, the Client can perform the following operations:

- **Add** new data log, referred to the current day, to the region to which he is connected.
  Example:
  - 19/01/2021: 4 positives, 21 negatives, 25 swabs and 4 dead.
  - 20/01/2021: 12 positives, 31 negatives, 44 swabs and 12 dead.
- **Request** data aggregation on a specified time interval regarding the data on any node (region, area or nation).
  The types of aggregation provided by the system are *average*, *sum*, *standard deviation and variance* on every type of data. **Example:**
  - I am to Tuscany and I request the average of swabs made from 20/12/2020 to 27/12/2020 on the Sicily region.
  - I am connected to Tuscany and I request the standard deviation of dead from 24/12/2020 to 04/01/2021 on the south area.

Example of Client aggregation request

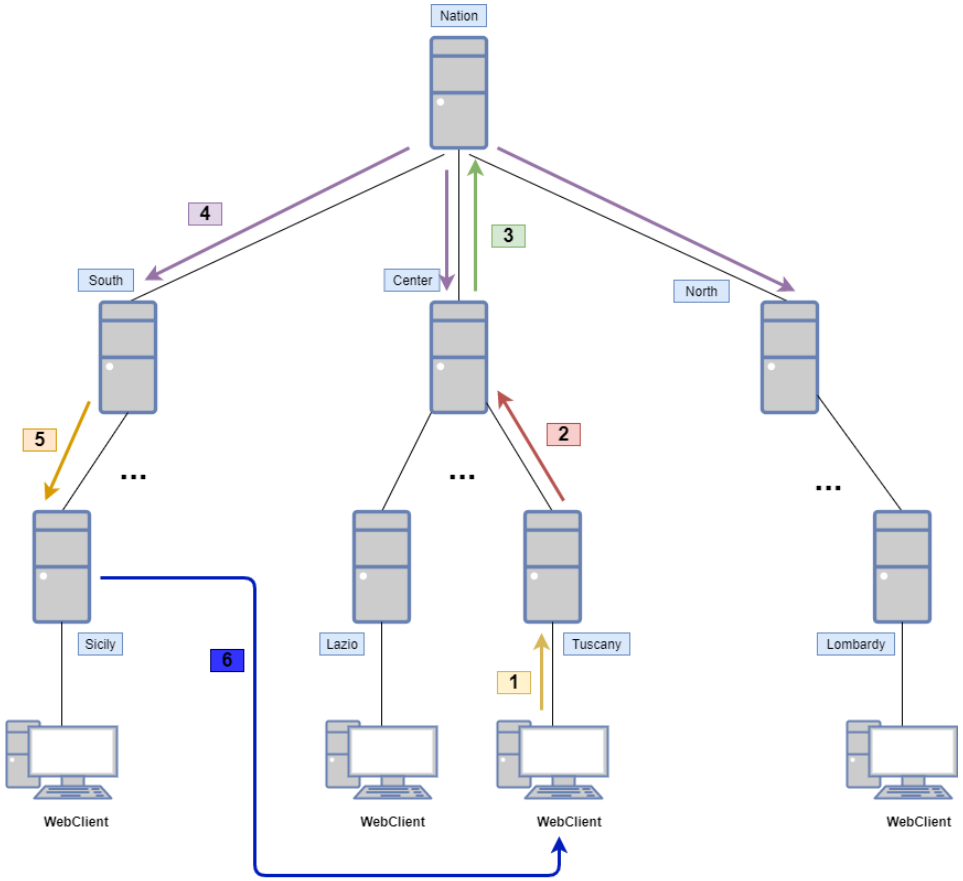

Figure 2 Example of aggregation request, from a WebClient connected to Tuscany to Sicily.

In the previous figure are shown all the steps to satisfy an aggregation request from a WebClient, connected to Tuscany, to Sicily:

1) The aggregation requested by the Client is sent to the region to which he is connected, Tuscany.
2) The Tuscany node receive the request, and since it is not referred to him, he forwards the request to his parent, the Center node.
3) The Center node checks if the request is destinated to one of his region node, since that is not the case, he forwards it to his parent, the Nation node.
4) The Nation node checks if the request is destinated to him, since that is not the case, the request is flooded to all the region nodes.
5) The Region nodes receive the request and the South node discovers that it is destinated to one of his nodes, so he forwards the request to the Sicily node. All the other just drop the message.
6) The Sicily node check if has already in memory the aggregation requested:
    a. If he does not have it, he elaborates the aggregation requests, saves the result and sends it directly to the Client who made the request.

Web-Server

A web application for each type of node provides an interface that simply prints on the screen all the messages that the node receives. In this way each node tracks all the operation requested by the client and it is possible to monitor the overall behaviour of the application.

Nation node and registry closure request

The nation node is the only one with an additional feature, that is the button to request a registry closure. Each day the nation node requests the closure of the registry to his children, that also propagate the request to the level below.

In this way it is possible to aggregate data at higher level, grouping data from the same area into just one daily report, stored by the area node, and data from different areas, stored by the nation node.

This operation automatically starts at the midnight, but can also be done manually at any time thanks to a button of the nation node interface.

Example of registry closure request

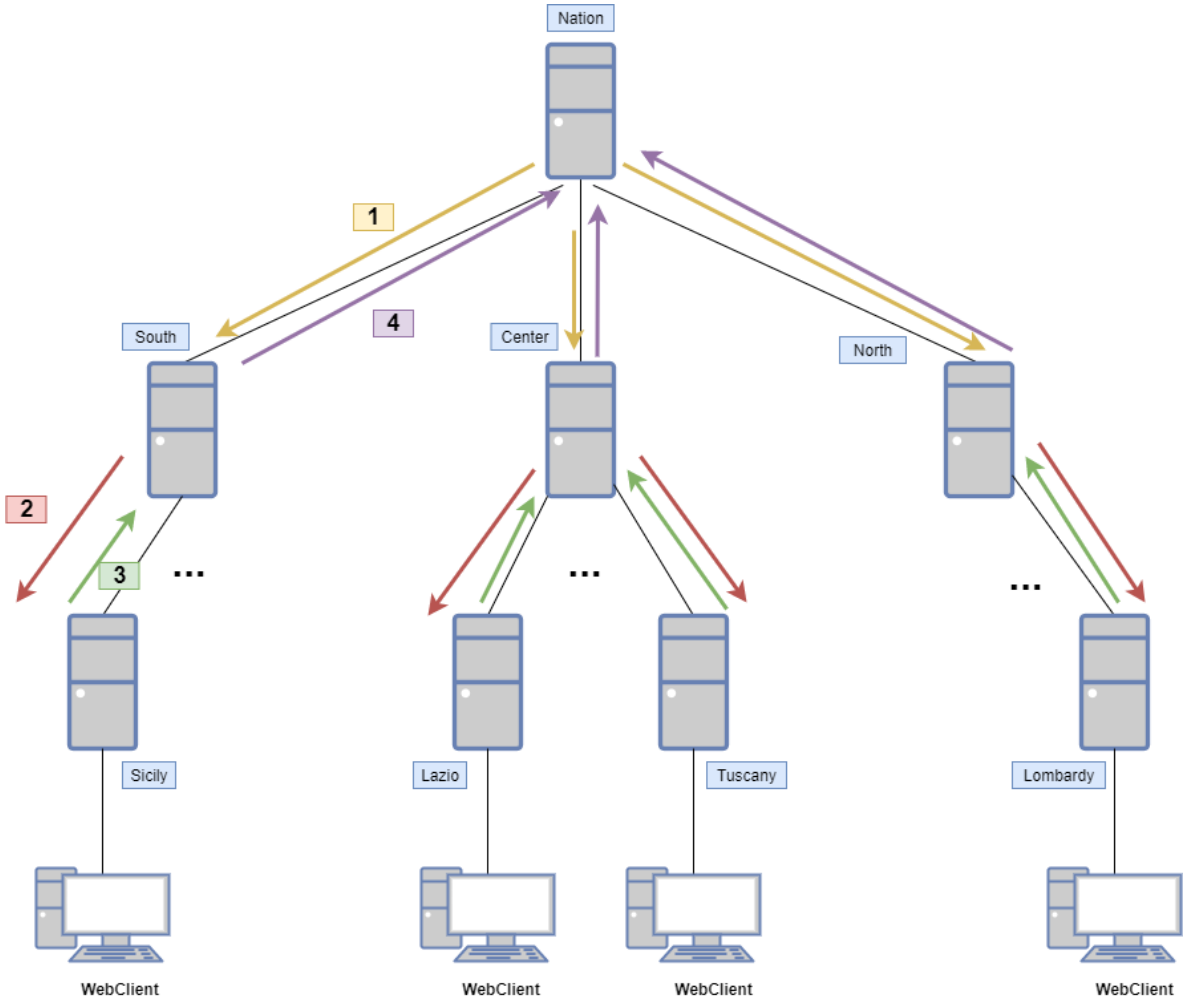

Figure 3 Request for a registry closure.

In the previous figure are shown all the steps to satisfy a registry closure request:

1) The Nation node send a registry closure request to all the Area nodes.

2) The Area nodes forward the request to their Region nodes.

3) The Region nodes elaborate the daily reports of their data summing all the log received from different clients, store the result and send it to their Area node.

4) The Area nodes aggregates all the reports received by their Regions, store the result and send it to the Nation node

5) The nation node aggregates all the reports received by the areas and store the result.

Implementation

The application is hosted on a Glassfish server (version 5.1)

Communication

For the *Communication Layer* we decided to implement a **Message Driven Communication** exploiting **JMS**; each Node consumes messages from a specific *queue*.

Because of the design choices we made, the coupling between a Node and the queue which it consumes from has to be done at *run-time*; initially we wanted to make use of **Message Driven Beans**, but setting one requires to know at *compile*-time or setting it at *run-time* through the use of "*javax.jms.MessageConsumer.setMessageListener(MessageListener listener)*" and the latter gives official problems when called by application running in the JavaEE web or EJB containers (*https://github.com/eclipse-ee4j/jms-api/issues/27*).

For this reason, we decided to make use of ***asynchronous send*** and ***synchronous receive***, dedicating a *thread* to always be waiting to consume from the setted queue.

Explaining the problem: to maintain modularity, we decided to have a *generic node* for each major node type (*nation, area* and *region*), specifying at *run-time* the specific type of node it needs to be (e.g. we can have a *generic region node* be a *Toscana node* just specifying "*Toscana*" at run-time); such node makes then use of a service provided by the *nation node* to discover the name of the queue it needs to consume from. Such service exploits a specific file which stores the whole *hierarchy* and associations between nodes and their queue.


```
"nation": "jms/nationQueue",
"north": "jms/northQueue",
"center": "jms/centerQueue",
"south": "jms/southQueue",
"valledaosta": "jms/valleAostaQueue",
"piemonte": "jms/piemonteQueue",
"liguria": "jms/liguriaQueue",
"lombardia": "jms/lombardiaQueue",
"trentinoaltoadige": "jms/trentinoAltoAdigeQueue",
"veneto": "jms/venetoQueue",
"friuliveneziagiulia": "jms/friuliVeneziaGiuliaQueue",
"emiliaromagna": "jms/emiliaRomagnaQueue",
"toscana": "jms/toscanaQueue",
"umbria": "jms/umbriaQueue",
"marche": "jms/marcheQueue",
"lazio": "jms/lazioQueue",
"abruzzo": "jms/abruzzoQueue",
"molise": "jms/moliseQueue",
"campania": "jms/campaniaQueue",
"puglia": "jms/pugliaQueue",
"basilicata": "jms/basilicataQueue",
"calabria": "jms/calabriaQueue",
"sicilia": "jms/siciliaQueue",
"sardegna": "jms/sardegnaQueue",
"regions": ["valledaosta", "piemonte", "liguria", "lombardia", "trentinoaltoadige", "veneto", "friuliveneziagiulia", "emiliaromagna"
"nationChildren": ["north", "center", "south"],
"northParent": "nation",
"northChildren": ["piemonte", "valledaosta", "liguria", "lombardia", "trentinoaltoadige", "veneto", "friuliveneziagiulia", "emiliaro
"centerParent": "nation",
"centerChildren": ["toscana", "umbria", "marche", "lazio", "sardegna"],
"southParent": "nation",
"southChildren": ["abruzzo", "molise", "campania", "puglia", "basilicata", "calabria", "sicilia"]
```


Figure 4 a snippet from ***HierarchyConnections.json***

Erlang Server

We made use of an Erlang Server to provide the "*on-node services*" needed to elaborate the *statistical aggregations* required by other nodes.

In particular, we implemented 4 types of aggregation:

- *Sum.*
- *Average.*
- *Standard Deviation.*
- *Variance.*

Each node interacts with its specific Erlang Server.

Moreover we provide the possibility to register a different Erlang node for every possible operation, in case there is the need to distribute the load among more than just one server.

Synchronization/Coordination

The Nation node at the end of every day sends a registry closure request to all the nodes: this is made using a ScheduledExecutorService, scheduling a Thread which, starting from midnight, every 24 hours sends the request to all the Area nodes. This Thread also starts a timer that, when expired, calls a function that aggregates all the daily reports received and saves the result on the database (if a daily report arrives after the timer expiration it will not be considered).

The synchronization between nodes is guaranteed thanks to the use of EJBs, that are hosted inside the EJB Container of glassfish that guarantees a correct synchronization between threads that access to the same resources.

ProducerBean: it is a Stateless Bean which provides the functions to send a message into a queue. The queue can be either a permanent queue or a temporary queue. Every Node use this bean.

RecorderBean: it is a Stateful Bean which stores a list of String representing the responses to aggregations. It has a method to return a String with all the responses that is used by the WebClient to show to the user the result of every aggregation requested.

SynchRequesterBean: it is a Stateless Bean that provides the functions to send messages synchronously: it sends an aggregation request to a specific queue, and it starts a timer when the message is sent, if the timer expires before a response is received the aggregation request is declared lost. This Bean is used by the WebClient to send aggregations.

Bean node: every node of the system has his own Stateful Bean. This Beans define the behaviour of the node when a message is received: there is a method handleMessage(Message msg) that decide the actions to perform depending on the type of the message received.
Initially this EJBs were expected to be Message-driven Bean, but due to deployment error that we have not been able to resolve we decide to use Statefuls, which in any case allow the application to have the required behaviour.

Distributed database

For the **Database**, we decided to have it *distributed* through the whole hierarchy. Every node has its own Key-Value database (LevelDB) in which it stores the daily report and the result of the aggregations (in order to respond to a request for aggregation received several times by communicating with Erlang only once).