CoronaZ: another distributed systems project

Simulating a contact tracing application in a scalable environment

Stefan Ciprian Voinea
Università degli Studi di Padova
stefanciprian.voinea@studenti.unipd.it

Stefan Vladov
Technical University of Munich
stefan.vladov@tum.de

Fabian Rensing
Paderborn University
fabian.rensing@helsinki.fi

I. Introduction

This brief paper describes *CoronaZ*, a project for the Distributed Systems course at the *University of Helsinki*.

All the code of the project is publicly available on GitHub repository [1].

This project simulates a contact tracing application where each node represents a person (or a unique device attached to someone) that send signals to each other when in range and communicate the data collected to a server using the *publish-subscribe* pattern. The server, called *broker*, can then be polled by a node called *consumer* that will send the data to a database. A front-end application then requests this data and displays the movement and the latest updates via the browser.

The idea came from simulating this kind of movements with Arduino boards capable of communicating between themselves using the *nrf24l01* and to the broker with *esp8266*. Unfortunately this was not possible given the relatively strict amount of time that each of the students involved could dedicate to the project and the waiting time to get the necessary hardware.

II. Technological choices

In this section we describe and explain why we have decided to use certain technologies rather than others.

- *Programming language*: we chose Python because it is a fast programming language for creating prototypes. Also all group members are fluent with Python;
- *MQTT Broker*: we chose *Apache Kafka* as broker for our project since it is one of the most used brokers in the market and it is has a large community that supports the project. Apache Kafka also handles well scaling and integration with other systems;
- *Containers*: we chose to use *Docker* and *docker-compose* given the easiness of constructing and spawning nodes. As explained in III, we have a `docker-compose.yml` that handles that base components such as the broker and the database;
- *Database*: we chose *MongoDB* for its high scalability and ease of use. Additionally, its dynamic schemas would allow us to be very flexible with our data model;
- *Back-end*: we chose *NodeJS* and *Express*, as it is a popular back-end for web applications, and some group members had prior experience
- *Front-end*: we chose *React* for its fast development speed and familiarity of some group members with it.

A. System requirements

The system requirements for this project are simple:

- *Docker* and *docker-compose*;
- Ubuntu or another Linux system with `jq` installed for starting the project using the `init-project.sh` bash script (partly working with `git bash` on Windows).

III. Architecture

We can divide our project in five major areas: *nodes*, *broker*, *DB consumer*, *database*, *back-end* and *front-end*.

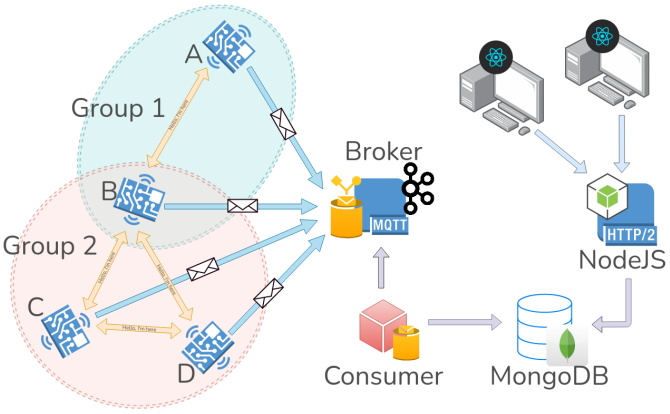

Fig. 1. Architecture of the *CoronaZ* project.

A. Nodes

This can be considered a single component since each node can be spawned separately from one another.

When each node spawns in the map it is placed on a random position. Every second the node will "*move*" in a random direction, broadcast a message telling its position and `UUID` (*Universally Unique Identifier*) and listen to the ones that are in broadcast by other nodes.

Nodes introduced into the simulation can be *infected* or *non-infected* (or *safe*). When a node is infected it will stay in his last position and will not move for a certain amount of time as specified in the parameter `infection_cooldown`, which defines the number of seconds that the node will stay in place. After this amount of time, the node will have an "*immunity period*" in which it will be able to move again and, if it gets

in contact with an infected node, it will not get infected. This time is the same as the `infection_cooldown` parameter.

In a gamification perspective, the nodes in our system are also called *humans* and *zombies*. Humans are the nodes considered *safe* while the *zombies* are the nodes that have been *infected*.

In our simulation the nodes can all connect to each other since they are all in the same network, as explained in IV, and each of them can hear the data sent in broadcast by the others. In a more realistic situation nodes would be only capable of hearing the signals of nodes nearby them, like in Fig.1 where node *A* can communicate only with node *B* and nodes *B*, *C* and *D* are nearby each other enough for them to hear their signal.

Here is an example of a message that is sent in broadcast from a node:

```
{
    "uuid":"ff0a1bda-34b9-11eb-b339 ... ",
    "position":[1, 5],
    "infected":false,
    "timestamp":"2020-12-02 16:19 ... ",
    "alive":true
}
```

After the node exceeds its lifetime in seconds, before exiting will send a last message where the value of `alive` will be `false`. This signals the front-end that this node will not have to be displayed anymore.

Each node has a unique *UUID* that is generated in Python, using the `uuid` [2]. This is created via a combination of `MAC` address and the `IP` address of the machine the script runs on and the timestamp on when the process starts.

The nodes images are built via a *Dockerfile* that uses the small Linux distribution Alpine to run the Python scripts. The nodes' python scripts are all loaded in the Docker image and the `main.py` file is executed as `CMD` when the container starts. This configuration allows to run containers with arguments as well, thus deciding if the node will start as an infected node or as a safe node.

The work of the nodes is split along four *threads*:

- main thread that starts all the other threads and controls them (main logic of the program);
- broadcasting thread in which the commands for broadcasting the message are executed;
- listening thread that waits for incoming messages;
- Kafka server connection thread that checks if the connection with the Kafka server is still up.

For connectivity the nodes use the `socket library` and will send UDP messages in broadcast to the unassigned port `4711` via a random port chosen by the library.

The sender's IP address and the port that the message has been sent from can be seen in the logs printed on screen. Setting the debugging at `INFO` level helps seeing the messages that each node send with one another and that are sent to Kafka.

The messages to be sent to the Kafka server are collected via the `get_next_broadcast_message` method and sent using the `send_message` method.

B. MQTT broker

Apache Kafka is composed by the Kafka container and the Zookeeper container. Kafka is an important part in the project since it is the *broker* in the *publish-subscribe* pattern.

Each node, after it has collected the IDs of the other nearby it, will send the list ot these IDs, with other information, to the broker. The *topic*, in our code, used by the nodes is "*coronaz*". This will contain all the information send by the nodes to the broker, which will later forward them to the consumer when it asks for them.

Here is an example of a message that is sent to the broker:

```
{
  "uuid": "5603b252-36de-11eb ... ",
  "position": [
      72,
      33
    ],
  "infected": false,
  "timestamp": "2020-12-05 09:43 ... ",
  "alive": true,
  "contacts": [
    {
      "uuid": "563eafe6-36de-11eb ... ",
      "timestamp": "2020-12-05 09:43 ... "
    },
    {
      "uuid": "56fd5d12-36de-11eb ... ",
      "timestamp": "2020-12-05 09:43 ... "
    },
    {
      "uuid": "567a64c7-36de-11eb ... ",
      "timestamp": "2020-12-05 09:43 ... "
    },
    {
      "uuid": "56b292fc-36de-11eb ... ",
      "timestamp": "2020-12-05 09:43 ... "
    },
    {
      "uuid": "575030c3-36de-11eb ... ",
      "timestamp": "2020-12-05 09:43 ... "
    },
    {
      "uuid": "563eafe6-36de-11eb ... ",
      "timestamp": "2020-12-05 09:43 ... "
    }
  ]
}
```

If a message is received from the same node twice then, the receiving node will discard the first message that arrived.

C. DB consumer

The DB consumer can be considered as a single entity since it is independent both from Kafka and from Mongo. The consumer is subscribed to the Kafka topic that contains the new messages from the nodes, in our case "*coronaz*". When a new message arrives, the consumer gets it and, every ten messages (or when a node dies), it aggregates them in a `json` that will be sent to the MongoDB database.

D. Database

As the other components, the database runs in a Docker container.

It accepts and stores incoming data from the Consumer. The database is accessed by the back-end, which will forward the data to the front-end, refreshed every second in order to always display the latest status.

Fig. 2. CoronaZ database structure.

In the database, each document is mapped with the `UUID` of the nodes as a key and the value represents the latest update of the node(from the last message).

E. Back-end

For the back-end of the project, *NodeJS* provides a single REST endpoint for the front-end. This is *GET /data*, which connects to the MongoDB database and queries all the documents containing the state of the nodes.

F. Front-end

The front-end of the project, built with *React* and *MaterialUI*, shows the evolution of the system and the simulations.

A slider allows the user to select one of the queried states from the database, which will be then visualized on the map underneath. Moreover, this will also update the statistics on the top of the page. Realism mode switches between using circles or icons.

Safe nodes are colored in blue while infected nodes are red. Dead nodes are gray.

The statistics are the following: "Total nodes", "Zombies", "Deaths" and "Dead zombies" which respectively represent the total number of nodes, number of infected nodes, number of dead nodes and number of nodes that died infected.

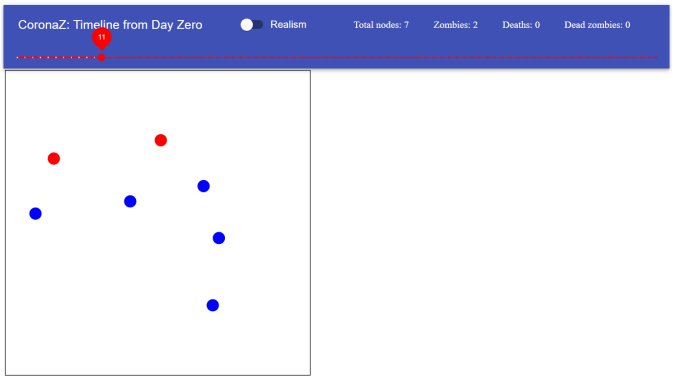

Fig. 3. CoronaZ front-end.

IV. Docker networking

Since this project utilizes Docker containers, there is a docker network called `coronaZ`, defined in the `docker-compose.yml` file, which connects all the components.

This network allows each node to get a new IP address and to have their own ports for broadcasting and listening.

Setting the network mode in another way, for example with `network mode:host` is inadvisable due to operability issues between docker for Windows and Linux, as well as creating problems with IP and port assignment.

V. Scalability and fault tolerance

We have tested the scalability and the fault tolerance of the project in the following ways:

- *unexpectedly shutting down the MongoDB database*: when MongoDB fails, the user is presented with a loading `gif` in the front-end that communicates an error happened in the system. The DB consumer will wait to send the messages until the database itself becomes reachable again;
- *unexpectedly shutting down the DB consumer*: if the DB consumer goes offline Kafka will still hold the messages in the "*coronaz*" queue until asked for them. The database will not be updated until the consumer comes back up;
- *unexpectedly shutting down the back-end*: as per the database, if the back-end fails, a loading `gif` will be presented to the user in the front-end. When it comes back up, it will serve the front-end again and the user will see the current map and simulation;
- *unexpectedly shutting down the broker*: this would make nodes not able to communicate with the rest of the system, they would only send messages among themselves. These messages will be stored in the nodes until the broker becomes available again. When the broker comes back up again the messages will contain all the contacts that have been recorded by each node, but will not contain all the movements made. The next message will contain the movement from the current position and the movements made in the downtime of the broker

will be lost. The movements are not essential since the exact movements are not relevant in a contact tracing application, the most important data comes from the contacts;
- *adding more nodes*: when a node is added it will start communicating with the other nodes already in the system and it will start sending messages to the broker;
- *unexpectedly shutting down a node*: if a node shuts down unexpectedly, the last message with the "`"alive": false`" parameter will not be sent to the server. This would cause the front-end to keep showing the node as a fixed point on the map. The rest of the system will not be affected by a node shutting down.

In general for the errors that can be felt in the front-end, such as the database not responding or the back-end server not responding, the user does not really need to know what exact component went down. This is also for security reasons in case a *malicious* user wants to find the vulnerabilities; without the specific logs it is harder for him to find the exact breaking point of the project.

Also the user does not need to concern himself with the various components which make up the whole network, he perceives the system as one entity.

VI. Simulation

To start the project we have made an `init-project.sh` script that asks the parameters with which the simulation will take place, executes the `docker-compose` (`up` and `down`) commands and manages the number of nodes by spawning as many as the user wants.

```
Database URL: http://localhost:8081/
Frontend app URL: http://localhost:3300/


Menu:
1) Add 1 non-infected node
2) Add 1 infected node
3) Add X nodes
0) Exit / Stop simulation

What option do you want to execute? 1

Adding 1 non-infected node
```

Fig. 4. CoronaZ front-end.

The script will ask for the run parameters and will set them on the file, otherwise will run with a set of defaults. The default parameters for the run are (in *json* format):

```
{
    "field_width" : 100,
    "field_height" : 100,
    "scale_factor" : 5,
    "zombie_lifetime" : 120,
    "infection_radius" : 2,
    "infection_cooldown" : 15
}
```

These parameters stand for:

- "*field_width*": width of the map;
- "*field_height*": height of the map;
- "*scale_factor*": scales the map and the nodes in order to be viewed better in the front-end;
- "*zombie_lifetime*": lifetime in seconds of the nodes;
- "*infection_radius*": the distance that the nodes can consider to infect other nodes;
- "*infection_cooldown*": "*cooldown*" period in seconds in which the nodes stand in order to "*cure*". This value is also used as an "*immunity period*" in which the node cannot get infected.

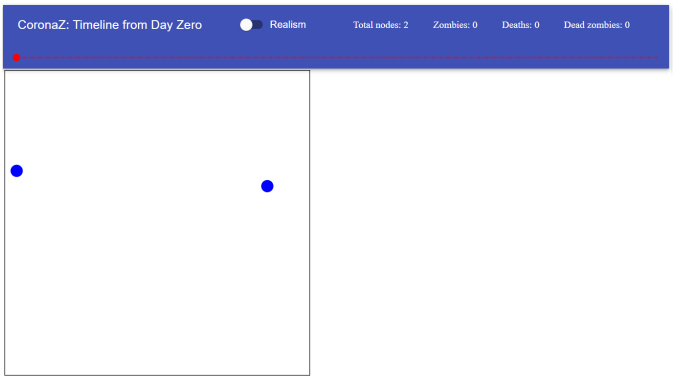

Fig. 5. CoronaZ simulation at start.

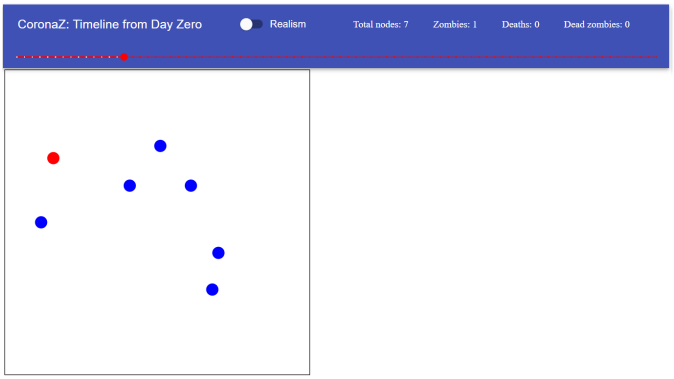

Fig. 6. CoronaZ simulation when first node is cured.

VII. Performance Evaluation

A "`simulation.pcapng`" file is present in the essay folder and contains part of a simulation for the system system running.

From this file, various message lengths can be seen, for example: the length in bytes of a message sent in broadcast by each node is *148B* (just the payload, *189B* with headers). For the messages sent to the the server by the nodes, the minimum size is *148B* (*189B* with headers), the rest depends on the number of contacts the node had.

Fig. 7. CoronaZ simulation when nodes die (from natural causes).

For a simulation with twenty nodes, each of them sends one message in broadcast a second, which means 189B. In the "*worst case scenario*" where all the nodes are in infecting range, the lenght of the message sent to the server would be *2.49KB*.

So just one node sends $189\ B + 2.49\ KB = 2.68\ KB$ each second, this would mean that for a simulation of 120 seconds, the total amount of data sent by the nodes is $2.68\ KB * 120 * 20 = 321.6\ KB * 20 = 6.43\ MB$.

Latency does not represent an obstacle in our system since each node stores the messages that need to be sent to the server and the data handled is not time sensitive as per other *real-time* applications.

Network performances for the nodes are discussed in VIII-A.

VIII. Future work

Given the nature of the project there won't be future releases but in this section we want to talk about what can be done to improve the project.

There are various improvements that can be done to make CoronaZ a more interesting and stable simulator, we will tackle them based on the areas defined in III.

A. Nodes

Nodes logic can be improved adding a more intelligent behaviour. This would result in better movements on the map and, possibly, avoiding infected nodes.

The throughput of the network could be improved by shortening the messages and choosing a different message format (like *differential updates*). This would allow not only to exchange less bytes between the nodes but especially between the nodes and the Kafka server.

B. MQTT broker

In order to allow for a better network performance, scaling the broker would be the best option. A dynamic scale rule could be created and the server replicas can be hidden behind a proxy.

This would not only help improving the performance of the network but also by adding fault tolerance in case one of the replicas fail.

C. DB consumer

As per the broker server, scaling the DB consumer node and making each node read from a separate queue or a separate server would improve network performance and fault tolerance.

D. Database

Currently the database only holds one simulation, but it can be expanded to hold multiple ones, preferably as different MongoDB collections.

As MongoDB is very configurable, one could also tweak factors such as replication.

E. Back-end

Currently the back-end only supports one end-point. An important improvement would be to allow the front-end to make intelligent queries, for example only sending the newest states, instead of all of them, to reduce bandwidth consumption.

Additionally, more complex functions such as selecting a specific simulation or user profiles can be implemented.

F. Front-end

The front-end is feature complete in terms of our project scope. A further improvement could be an option to select a specific simulation or having a user profile.

IX. Conclusions

This project represents a simulation of a contact tracing application where each node that spawns in the network communicates its position to the others in range and will send the data collected to a central server with the *publish-subscribe* pattern.

The running system implements the basic goals and functionalities requested in the given programming task sheet. As explained in V it supports scalability and fault tolerance, while the end user of the system can decide how many nodes to add, thus scaling the system *horizontally*. In VII there is the analysis of how the system performs and how much data is sent in the network during the simulation. Improvements, both in the system's architecture and it's performance are discussed in VIII. This project has these three characteristics: *naming and node discovery*, *synchronization and consistency* and *fault tolerance*.

References

[1] *CoronaZ repository:* https://github.com/cipz/CoronaZ/
[2] `uuid` — *UUID objects according to RFC 4122:* https://docs.python.org/3.8/library/uuid.html