DISTRIBUTED COMPUTING

Project 1: IoT application

Marc Cervera Rosell - 47980320C

December 2022

Grau en Enginyeria Informàtica

Contents

List of Figures

Introduction

This first project consists in create an IoT application. This application will consist in a number X of clients that will send to a MQTT broker different values, that represent temperatures. The MQTT broker will send the data to a MongoDB.
For the Kafka part, a second cloud has been created. This second cloud will receive the same data as the first one and will send it to a Kafka broker, and that will send the data to a module named "Analytics module" that has been provided by the professor.
Once the module has processed the data will return the results to the second cloud and, finally, the data is going to be stored in a MongoDB database.

Description of the components

In this section are going to be described all the components of the project. These components are the set of files conformed by docker files, python files and docker compose files.

Main Docker-compose

This ".yml" file, is a docker compose that runs the whole project.
Inside the file, are specified the number of clients that will be created.
FOllowing the structure of the file, the first thing that can be found is the docker compose version. In this case, the version 3.8 is goint go be used.
Once the docker compose version is specified, the second thing that can be found are the different services.
Inside the services, the first item that can be found is the configuration specification of the clients. The parameter, *build*, defines a configuration that is applied by Compose implementations to build Docker image from source. In other words, the parameter will go to the specified path and will search the Docker file inside that path and will build it.
The second parameter is to specify the deploying options. *mode: replicated* indicates that the service will be replicated a number X of times, and the parameter, *replicas*, indicates the number of replicas of that service. The last parameter, inside the client service, is called *networks*. This only assigns a network to the client (in this case, the network is called *asgard*, and its definition is coded at the bottom of the docker compose file).
The second service, is the first cloud. Previously, in the introduction section, I've specified that I've used two clouds to solve this practical case.
Inside the cloud service, the "new" variables, are *container_name*, *depends_on*, *ports* and *environment*. The first one (*container_name*), specifies the name of the container that contains the cloud.

The second one, (*depends_on*), specifies on which services depends the current service. In this case, the cloud depends on the database and the Kafka broker.
On the database dependency, is specified the condition *service_started*. This means that before the cloud starts running, the database service has to be started.
On the Kafka broker dependency, is specified the condition *service_healthy*. This condition specifies that the service must be running and responding.
The *ports* variable, exposes the ports where the programs listen. In this case, the port 5000 of the program listens to the custom port 5000.
Finally, the *environment* variable, shows the environment variables.
The third service is the database specification. The main difference between the database service and the ones that have been described before, is that in the database the *build* parameter is not used, because it makes no sense. Instead, the *image* parameter is used. In this parameter, the software image is going to be used, is specified.
The parameter *restart*, specifies that if, for any reason, this service fails, the service is going to be restarted.
Finally, the parameter *volumes*, shows where the data is going to be stored in case the docker program is stopped.
Until the Kafka service, there's nothing new to comment. In this service, there's a "new" parameter, that hasn't been explained before. This parameter, is one called *healthcheack*. This parameter checks whether the resource is operating normally.
Finally, in this file, the *asgard* network configuration is implemented. Firstly, the network name must be specified. Inside the network block, there's another block called IPAM. IPAM is an acronym, that stands for IP Address Management. The fact, that the *driver* parameter is configured as default, means that a network bridge is going to be used. Finally, only remains to assign an IP address to the network.

Client

The client, has a total of four dependencies. This dependencies are:

- random
- paho-mqtt
- re
- sleep

The *random* library is used for creating the ID of each client.
The *paho-mqtt* library, contains all the mandatory to work with MQTT connections. The *re* library, allows working with regular expressions. In this case, this library, is used to ignore the called special characters, such are: arrobas, underscores, colons, etc.

The last dependency is *time*. This library is used to control the time interval where the data is sent to the MQTT broker.
Once the dependencies are described, it's time to start with the methods.
The first method, is *process_file*. This method, reads the file, that is passed as a parameter, where are placed all the temperatures and put them inside a list, which is returned.
This method has been designed thinking for efficiency, because reading directly from the file all the time is not very efficient.
The second method is *data_every_10_seconds*. This method receives two parameters. The first one is the ID of the client, and the second one is the list that contains all the temperatures.
This method, will send, every ten seconds, a lecture to the broker.
This lecture will be published looking like: *ClientID;lecture;datetime* not without firs specifying in which topic has to be performed the *publish* operation.

Dockerfile

As the project is implemented with Python, firstly the Python image has to he specified. This specification is done with the parameter *FROM*. The second "bloc" consists in specifying the files that are going to be run. This is done with the parameter *COPY*. Lastly, only remains running the file. This is done with the parameter *CMD*. This is only the docker syntax for the command line execution command.
There's one particularity in the file execution part. As it can be seen, is specified the parameter "-u". This is for not creating buffer.

MQTT broker

The used broker, is the broker used in the class examples. The "mosquito" broker.
The client will send the messages to the mosquito broker and the cloud will receive it to store the data in the database.

Cloud

The first used cloud has a total of five dependencies. These dependencies are:

- datetime
- paho-mqtt
- pymongo
- kafka
- pickle

The *datetime* dependency allows manipulating dates and times.
The second dependency is *paho-mqtt*. This dependency has been explained previously.
The next dependency is *pymongo* that has been explained previously too.
The third dependency is *kafka*. This, is a Python client for the Apache Kafka distributed stream processing system.
The last dependency is *pickle*. This module implements binary protocols for serializing and deserializing a Python object structure.
This Python file, has three methods. The first one is called *connect_to_mongo*. This method establishes connection to a MongoDB database and gets the database and the collection. If the connection is not performed successfully, a message is sent reporting the error. One more point to comment is that the connection to the database is performed using a username and a password. This method does not receive any parameters.
The second method of the file is *save_to_db*. This method saves the data received from the MQTT broker to the database. If the insertion is not possible, a message is displayed reporting the error. The aforementioned data, from this method's point of view, is received as a parameter.
The last method is *on_message*. This method receives the data from that arrives from the broker that looks like, *ClientID;Lecture;datetime*, splits it up and does two tasks. For one site the data is saved to the MongoDB database and for the other site, the data is sent to the Kafka broker that subsequently is going to be sent to the analytics module, provided by the professor.
In this point, it can exist a confusion, because two methods have been referred regarding saving the data. The following description will clear the possible confusion:

- *save_to_db* $\longrightarrow$ Performs the insertion operation.
- *on_message* $\longrightarrow$ After splitting the data, calls the *save_to_db* method.

Another matter that must be cleared, is the declaration of the *on_message* method. As it can be seen in the code, the method has three parameters, but only one is used. This is because the library defines the function in this way.

Dockerfile

This docker file has the same explanations as the client's one, but conformed for the cloud Python file.

Database

The used database to resolve the proposed practical case is a MongoDB database.
The database is called "temperatures" and has two different collections. The first collection, is called "temperatures" too and the second collection is named "analytics_results".

In the "temperatures" are placed all the temperatures together with the identifier of the client that has sent that lecture. In other words, this collection saves the lectures of each client alongside the ID of that client.
The second collection, saves the results of the "analytics_module".

Kafka

The Kafka used image is *docker.io/bitnami/kafka:3.3*.
The cloud assumes the message producer role. The cloud will send the messages to the Kafka broker, and the Kafka broker will send the messages to the analytics module. The results of this module (provided by the professor), will be returned to the Kafka broker in another topic and, finally, will be sent to a second cloud, that assumes the message consumer role. This second cloud is going to be explained afterwards.

Analytical module

As this module has been provided by the professor, there's nothing to comment except one aspect. The *bootstrap_servers* IP, has been changed to my Kafka IP.
The results of this module, are returned to the Kafka broker inside the topic *analytics_results*.

Cloud 2

The first aspect to comment about this second cloud are the dependencies. It has a total of six dependencies, that are:

- paho-mqtt
- pymongo
- kafka
- pandas
- pickle

As it can be seen, all the dependencies but one have been explained. The dependency that hasn't been explained is *pandas*. This library, is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool. The methods are the same as in the first cloud. That is, the methods *connect_to_mongo* and *save_to_db* perform the same task as in the first cloud.
The only one difference between this cloud and the first one is located in the *__main__*

method. Inside this method, a *KafkaConsumer* is created in the same as in the professor's Python script.

Main design decisions

Using MongoDB

Compared to normal collections, storing time series data in time series collections improves query efficiency and reduces the disk usage for time series data and secondary indexes.
Time series collections use an underlying columnar storage format and store data in time-order with an automatically created clustered index. The columnar storage format provides the following benefits:

- Reduced complexity for working with time series data.
- Improved query efficiency.
- Reduced disk usage.
- Reduced I/O for read operations.
- Increased WiredTiger cache usage.

Using two clouds instead of one

This is done thinking in fault tolerance. That is, if one cloud fails, the other one can keep working. For example; let's imagine that the second cloud (the one that saves the results of the analytics module), for any reason, fails. The fact of having two clouds allow keeping receiving messages from the MQTT broker.

Not reading all the time from the .csv

Reading all the time directly from the csv file is not efficient because, consumes a lot of resources of the computer. In order to economize the consumption of resources, instead of reading all the time the csv, the file with the data is only red once. All the data is introduced inside a list and the lectures are picked from this list.

How to run the project

In order to run the whole project, there are few commands to execute in the terminal:

1. docker compose build

- This command will build the docker images for the whole project. This means that will download and install all the dependencies and will build all the docker images from all the docker files.
- This command has to be executed only the first time that the project is executed or when there are some changes in the code.

2. docker compose up
    - This command will run the entire project, once the docker images are build.
3. docker compose down
    - This command will stop the entire project.