UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Cloud Computing

Docker Health Monitoring System

Project Documentation

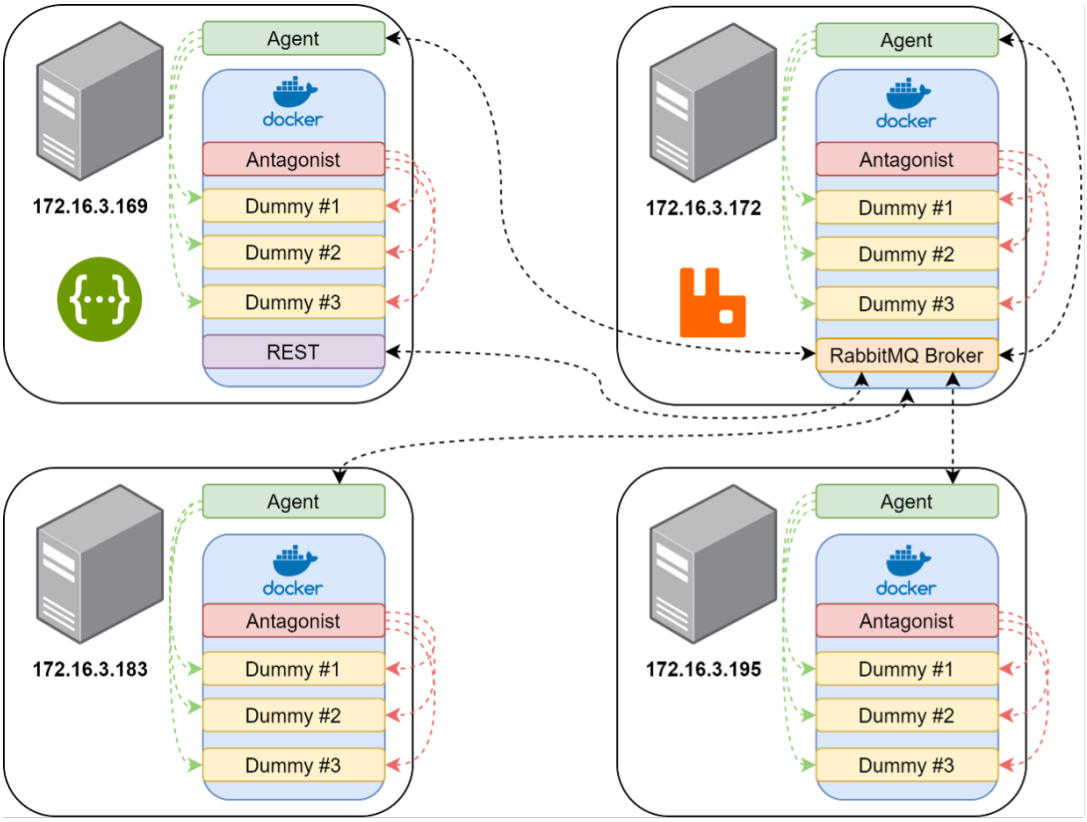
TEAM MEMBERS:
Federica Baldi
Daniele Cioffo
Edoardo Fazzari
Mirco Ramo

Academic Year: 2020/2021

Contents

1 — Introduction

The purpose of this project is to develop a system to monitor the health of a set of Docker hosts. The structure of such a system is as shown in the figure.



On each Docker host runs an **agent**, a software whose task is to periodically check the wellbeing of containers running on the same host. Specifically, every x seconds the agent pings all containers in the *"to be monitored"* list and calculates the experienced packet loss. If a container is down or if it is experiencing packet loss greater than a certain threshold, the agent must destroy it and automatically restart it.
The list of containers to be monitored and the threshold are set by the system administrator, who communicates with the system via a **REST interface**. This interface is exposed by a control module running on one of the Docker hosts (in our case, on the machine with IP address 172.16.3.169).
In addition, one of the Docker hosts (in our case, the machine with IP address 172.16.3.172) exposes the **RabbitMQ broker service** as a container, thus allowing the various components (the agents and the swagger server to communicate.
To test the proper functioning of the system we have:

- created "dummy" containers running on each Docker host that will be monitored by each of the agents
- developed the **antagonist**, a small program running on each Docker host that periodically stops containers or artificially causes packet losses

2 — Agent

As already mentioned, the agent's main task is to monitor the health of containers running on its same machine by pinging them periodically. The agent is nothing more than a Python program that can communicate with the Docker daemon thanks to the "docker" Python library[1].
In order to talk to the Docker daemon, first of all a client is instantiated through the command *from_env()*. Then, the agent will start its periodic check.
Specifically, every *INTERVAL_BETWEEN_PINGS* seconds, for each containers name in the *MONITORED_LIST* the agent performs the following actions:

- It tries to retrieve the container object from its name. If the container does not exist on that particular host, the agent skips the next steps.
- It retrieves the IP address of the container within the bridge network. If the container has no IP address assigned, it means that it is not running, so the agent restarts it and skips the next steps.
- It pings the IP address of the container by sending 5 packets and it calculates the percentage of packet loss. If this percentage is higher than the *THRESHOLD* value, then the agent restarts the container; otherwise, it assumes that everything is working properly.

In addition to the above actions, the agent also writes to a log file every time it performs a check or some action, so that it is easier for us to debug.
Moreover, in order to communicate with the other containers of the Health Monitoring System, the agent implements a **communication module** through which it receives the administration commands coming from the REST interface and forwarded by the relative **swagger server**, and performs the relative actions on all the monitored containers in the targeted hosts. (Of course every agent instance controls the host in which it runs).
To be precise, the **communication module subscribes** to different topics, corresponding to the possible commands that can be received by the administrator, and when a message is consumed, the body is decoded in order to perform the action. Those topics are:

1. **threshold** (publisher: Swagger server, subscribers: agent instances, body: new_value): asks to every receiver agent to change the allowed rate of packet loss for every monitored container
2. **list_request** (publisher: Swagger server, subscribers: agent instances, body: N/D): asks to every receiver agent the status of all the containers running in the target host, in terms of hostname, container name, operational status (running or exited) and monitoring status (monitored or non-monitored)
3. **actives** publisher: Swagger server, subscribers: agent instances, body: tuple(str hostname, str name, boolean new_status): asks for a specific container to be added to (new_status=True) or removed from (new_status=False) the monitored list. Only the relative agent instance performs the action, the others discard the packet.
4. **list_response** (publishers: agent instances, subscriber: Swagger server, body: json_str

   $\{hostname : \backslash hostname", name : \backslash name", status : \backslash running/exited", monitored : "true/false"\}$

   ): response provided to list_request, in which every agent instances report the status of the container running on the relative host

[1] https://docker-py.readthedocs.io/en/stable/

Since the RabbitMQ consume operation is blocking, in order not to stale the agent we adopted the following solutions:

- The communication module and the "pure agent" one run on **two different python threads**, in this way the administration command can still be received while the periodic check is performed
- The shared context is made only by two data structures, i.e., *MONITORED_LIST* and *THRESHOLD* integer; every possible concurrent access to them has been avoided using the python explicit locking
- In order to minimize the lock time, instead of maintaining the lock for the entire agent's read access to the shared variables, they are locally copied every time a periodic check is about to start, keeping the lock itself just for the copy time.

3 — REST Interface

This REST APIs were generated using *Swagger Editor* and the functionalities implemented are:

- **GET**:

  containers/ : Retrieve the list of all containers[2] with their informations. The *response* is a list of containers containing for each one its hostname, its name, its status and if it is monitored or not.

- **POST**:

  containers/{hostname}/{containerName} : Monitor container specified in the path

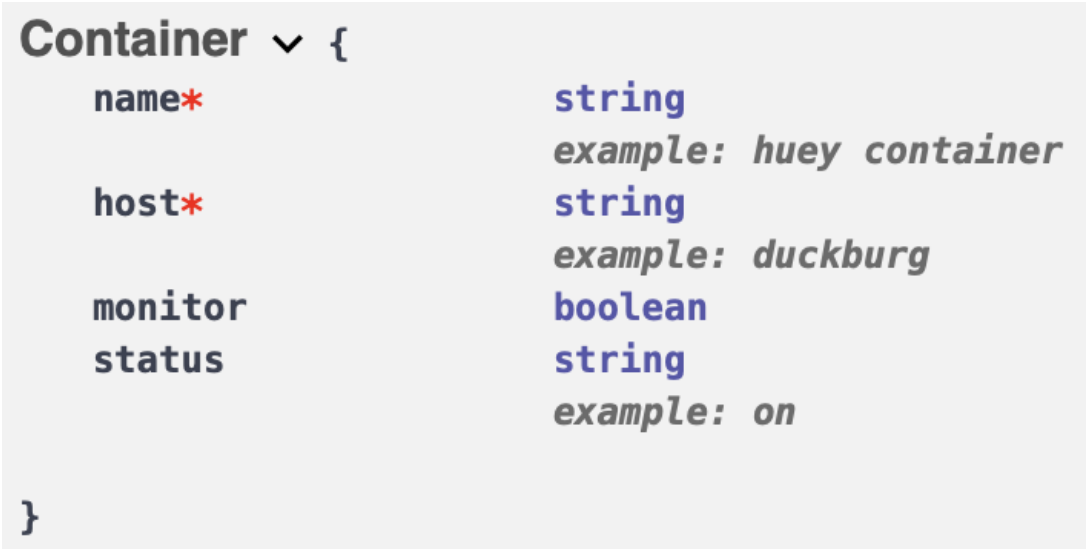
- **DELETE**:

  containers/{hostname}/{containerName} : Unmonitor container specified in the path

- **PUT**:

  /threshold : Update the packet loss threshold used in the Agent. The value is specified in the body of the request and its type is *double*.

The REST Interface makes use also of a model for specifying the structure of a Container, which is the following[3]:



[2]All containers that are active. The REST interface knows the total number of actives hosts, but if some of them are down the GET operation works retrieving only the ones that respond within a lap of time equal to 4 seconds. If all of the hosts answer to the request in a shorter time, the REST interface stops wait for answers and send the list of all containers back to the user

[3]the asterisk indicates that the attribute must be set

4 — Antagonist

In order to test the health monitoring system a small program (the antagonist) has been developed to stop containers or cause some artificial packet loss. More precisely, the antagonist was implemented in two different modules.
One of the two modules is implemented in each host on a docker container, starting from a customized Ubuntu image. In this container there is the python interpreter which is used to execute the antagonist.py file. This first module periodically attacks one of the local docker containers, simulating a stop of that container (leveraging the Docker API for python [4]).
The second module has been implemented inside each dummy containers, which we will discuss in detail in the next chapter. This module allows you to simulate network problems on a certain container, through the use of the Netem library [5]. The simulated packet loss percentage is periodically changed, through the use of this command: *tc qdisc add dev eth0 root netem loss XX%*, with XX the packet loss value, generated randomly from 20% to 100%.

[4] `https://docker-py.readthedocs.io/en/stable`
[5] `https://wiki.linuxfoundation.org/networking/netem`

5 — Dummy Container

As already anticipated, in order to test the proper functioning of the system we created containers on each Docker host. These "dummy" containers are built from a custom image we created specifically for this project. In particular, it is an Ubuntu image that includes a Python interpreter and that runs two Python scripts at startup.
One of the two scripts is the actual dummy, and it is trivially an infinite loop that periodically prints messages specifying the time at which it was started and when it is being executed.
The other, as explained in the previous section, is the part of the antagonist that is responsible for creating artificial problems on the network.