University of Pisa

MSc in Artificial Intelligence and Data Engineering

Cloud Computing

Health Monitoring System for Docker Hosts

Students:
Giada Beccari
Nicola Bicchielli
Lorenzo Biondi
Francesco Campilongo

Professor:
Prof. Carlo Vallati

Academic Year 2020-2021

Contents

Chapter 1

Introduction

The aim of this project is to develop a health monitoring system for a set of Docker hosts. The system is composed by 4 different virtual machines: a **node manager** and three **compute nodes**. The IP address of the 4 virtual machines are:

- *node manager*: 172.16.3.170
- *compute nodes*: 172.16.3.171, 172.16.3.182 and 172.16.3.209

The architecture of the system is shown in figure 1.1.

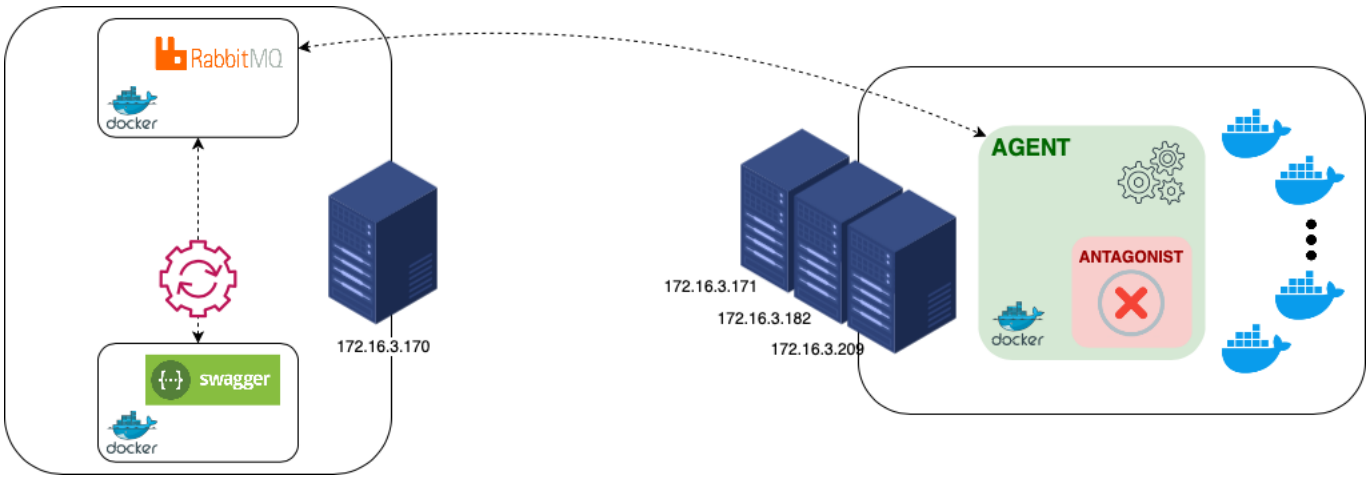

Figure 1.1: The system architecture

The node manager hosts two Docker containers: the **RabbitMQ broker** and the **Swagger REST interface**. The first enables comunication among the different components of the system, while the latter is used to retrieve the status of containers currently running in the platform and set some configuration parameters via RESTful APIs. Regarding the three compute nodes, each one of them hosts a Docker container running the **Agent** and a number of "dummy" Docker containers. The agent has to monitor the wellbeing of the local containers, checking if they are up and if they are experiencing some packet loss by sending periodic pings. If the packet loss for a certain container is higher than a specified threshold, or the monitored container has been stopped, the agent restarts it automatically. The **Antagonist** is executed inside the agent container and it is responsible to stop containers or cause some artificial packet loss on each Docker host, in order to test the health monitoring system. To communicate with the Agents running on the compute nodes, the REST interface exploits a **python manager** that handles and forwards the requests.

Chapter 2

Agent

The agent module in our project has the duty to check the status of local containers that the users chose to monitor. This control is performed periodically by sending "ping" messages to the targeted containers to monitor their health. If the percentage of packets lost is higher than a certain threshold or the container is not running anymore, the agent will restart it. The agent uses the "python docker" library to communicate with Docker.

To function correctly, the agent uses different parameters that will be described in the following:

- *Threshold*: the maximum acceptable percentage of packet loss.
- *Ping retries*: number of ping messages sent periodically to the containers to assess their packet loss.
- *Monitoring period* : the time interval the agents waits between monitoring cycles.

The agent maintains a list of containers that it has to monitor. Every *Monitoring period* seconds, a function "monitor" is executed to obtain the following status information from the listed containers:

- *Running*: a boolean which indicates whether a container is executing or not.
- *Started at*: the timestamp at which the container was launched the last time.
- *Restart Count*: the number of times the container has been restarted.
- *Image*: the image used to deploy the container.
- *IP*: the IP address assigned to the container. This is the address that is used for the ping messages.

Once this information has been collected, the agent first checks the status of the containers. If it is not running, it is restarted. Otherwise, an exchange of ping messages is carried out. If the pings fail, or the packet loss is greater than *Threshold*, the target container is restarted. The main functionalities offered by the agent are:

- *add_container*: adds a container to the list of monitored ones.
- *remove_container*: removes a container from the list of monitored ones.
- *set_threshold*: updates the "Threshold" value previously described.
- *set_ping_retries*: updates the "Ping retries" value previously described.
- *set_monitoring_period*: updates the "Monitoring period" value previously described.

Finally, other functionalities are called by the manager (REST interface) to obtain information regarding the current situation of the system:

- *get_configuration*: returns the values of the 3 parameters on the specific Docker host.
- *get_monitored_container_status*: returns the information regarding a specific monitored container.
- *get_all_monitored_containers_status*: returns the information regarding all the monitored containers.
- *get_all_containers*: returns the information regarding all the hosted containers (including those which are currently not monitored).

The agent communicates with the REST interface through a queuing system "RabbitMQ" which will be described in the following paragraphs.

Chapter 3

Antagonist

To stress the health monitoring system, a small program (the antagonist) has been run in the same Docker container as the one of the agent, on each host of the system (except for the node manager).

The antagonist has two main functionalities:

- to stop a random container on the host, with a probability of 20%, using the *stop()* command of the Docker library. The antagonist makes sure to not stop its own container.
- to introduce packet loss in the communication between the agent and each monitored container, with a probability of 30%. The packet loss is randomly chosen in the interval between 10 and 40 %. To simulate this behaviour we exploited the following Netem commands:
  - *tc qdisc del dev eth0 root*
  - *tc qdisc add dev eth0 root netem loss XX%* , where XX is the percentage of packet loss we want to introduce.

Chapter 4

RabbitMQ

In this project, we used RabbitMQ to implement a message queueing system. This service was deployed as a Docker container and exposed by the node manager (with an IP address of 172.16.3.170). All the other nodes could publish and receive messages exploiting different topics. In particular, different actors were involved in the communication: a manager and several agents. The manager ran on the same host of the REST interface and provided the APIs for the forwarding of REST requests to agents. To do this, we decided to use two different kinds of queues:

- *general queues*, which are used for messages that must be consumed by all the agents. For example, the request to retrieve the list of all the containers in the system must trigger the execution of the *get_all_containers* function on all agents. When an answer is expected, the manager must collect and merge all the responses before returning the result to the REST interface. All the agents must respond within a certain time interval or else an error or a partial result is returned by the manager. In order to match the responses to the request that triggered them, we exploited a unique token. The token is sent with the request and embedded in the corresponding responses. The topics binded to this kind of queue are "set_threshold", "set_ping_retries", "set_monitoring_period", "all_containers_status", "container_list", "config", "status_response", "containers_list_response", "config_response".
- *personal queues* that can be exploited to send a command only to a specific agent. This is possible thanks to agent-specific topics, composed by the name of the host and the actual topic name. For example, if we consider a host "datanode1", the topics that concern it are "datanode1add_container", "datanode1remove_container" and "datanode1container_status". These topics are binded to the personal queue used by datanode1.

A brief description for each possible operation and the associated topics was included in the following:

Description	**Request topic**	**Response topic**
Request to retrieve the status of *all* the containers in the system	container_list	containers_list_response
Request to retrieve the status of all the *monitored* containers in the system	all_containers_status	status_response
Requests the current configuration of all agents	config	config_response
Updates the value for the maximum packet loss allowed on *all* agents	set_threshold	/
Updates the number of ping retries on *all* agents	set_ping_retries	/
Updates the monitoring period on *all* agents	set_monitoring_period	/
Requests the status of a specific container running on *one* agent	*hostname*container_status	status_response
Adds a container to the monitored containers of *one* agent	*hostname*add_container	/
Removes a container from the monitored containers of *one* agent	*hostname*remove_container	/

Chapter 5

REST interface

The REST interface has been created using **Swagger Editor** and it exposes the following functionalities:

Method	**Endpoint**	**Description**	**Result**
GET	/container	Requests the list of all the active containers in the system	String (hostname and container name) list
GET	/container/{name}	Requests the status of a container. The name identifies the container and the agent.	*Container*
POST	/container/{name}	Adds a container to the monitored containers of an agent. The name identifies the container and the agent.	
DELETE	/container/{name}	Removes a container from the list of moniter containers of an agent. The name identifies the container and the agent.	
GET	/container/status	Requests the status of all the monitored containers in the system	*Container* list
GET	/config	Requests the current configuration of all agents, considering *threshold*, *ping-retries* and *monitoring period*	*Configuration* list
PUT	/config	Updates the configuration of all the agents. The new configuration is passed in the request body	

The objects *Container* and *Configuration* are defined as follows:

Container {

name*	string
monitored	boolean
running	boolean
started_at	string
restart_count	integer($int32)
image	string
ip	string
packet-loss	string

}

Config {

hostname	string
threshold	number($double)
ping-retries	integer($int32)
monitoring-period	integer($int32)

}