Università di Pisa

Master Degree in Artificial Intelligence and Data Engineering

Cloud Computing

Health Monitoring System

Tommaso Amarante
Pietro Calabrese
Francesco Marabotto
Edoardo Morucci
Enrico Nello

Contents

1 Introduction **1**

2 Requirements **3**
2.1 Functional Requirements 3
2.2 Non Functional Requirements 3

3 Architecture **5**

4 Implementation **7**
4.1 Agent 7
4.2 Controller 9
4.2.1 REST Interface 9
4.3 Antagonist 11
4.4 RabbitMQ 12
4.5 Architecturally Significant Use Cases 13

Chapter 1

Introduction

Containers are the industry standard for hosting applications. The benefits they offer to cloud-based microservices are infinite and have allowed organizations, both large and small, to deploy hundreds and thousands of containers to power their applications. At same time a cloud-service provider company, could have many physical machine that host a large amount of container, and often they are widely distributed over various geographic areas.
A typical company deploying its applications in a containerized manner could have anywhere from a few to thousands of containers working at any given time. Containers running complex configurations can be dynamically deployed and removed depending on the scale and load expected. Scaling poses challenges in tracking their performance issues and overall health on an on-going basis.
It's why monitoring the performance of containerized applications to ensure application continuity is essential. Monitoring and alerting becomes effective through analyzing metrics, obtained from many sources such as host and daemon logs, and monitoring agents installed on each node.
Moreover, it's very important to have a Monitor Docker Containers application because the health of an organization's containerized applications directly impacts the efficacy of its business. Monitoring application performance ensures that both the containerized applications and the infrastructure are always at optimum levels.
Also, monitoring historical-data and CPU usage are useful to recognize trends that lead to recurring issues or bottlenecks. Use these metrics to forecast resource needs more accurately, as it will lead to better resource allocations and deployments. On-going monitoring keeps app performance at its peak. It helps you detect and solve problems early on, so you can be proactive. You can avoid risks at the production level. Monitor the whole environment so you can implement changes safely.
Monitoring your containers in real time is essential to ensure peak app performance. When it comes to Docker containers, however, monitoring helps you to:

- Detect and solve issues early and proactively to avoid risks in production
- Implement changes safely as the entire environment is monitored
- Fine-tune applications to deliver improved performance and better user experience
- Optimize resource allocation

The difficulties of monitoring Docker containers revolves around containerized applications, isolated within containers, and with resources allocated dynamically. As container images become increasingly complicated with patches and updates, it is crucial to choose Docker monitoring tools that are robust and allow them to be deployed quickly across many thousands of nodes.

Chapter 2

Requirements

2.1 Functional Requirements

For the monitoring system developed by us, the following modules and the following requirements for each module have been identified.

- **Agent Module**: it is a module that is run on each host and takes care of checking the status of the containers present on the machine, sending periodic tests, and checking that these are above a target threshold; if this does not happen it destroys and restarts the container automatically. Furthermore, it interfaces with the RabbitMQ broker to manage communication with the Controller module
- **Controller Module**: it is a module that, through a REST interface, exposes all the container control functions to the system administrator. In detail, it allows you to retrieve the list that are running on the platform, set the thresholds for packet loss, select for which containers the monitoring system is active, remove a container from the monitoring system
- **Antagonist module**: it is a module that deals with creating system behaviors in a real environment. In particular, it randomly stops containers and simulates packet losses and delays in the network

2.2 Non Functional Requirements

In large scale containerized environments, this is only possible through dedicated cloud native monitoring tools, and must be automated.
Failure to achieve observability can result in:

- Poor visibility and operational challenges—without observability, it is difficult for developers and operations tasks to understand what is running and how it is performing. Maintaining applications, meeting SLAs and troubleshooting can become very challenging
- Scalability challenges—the ability to rapidly scale out applications or microservice instances on demand is an important requirement for containerized environments. However, observability is the only way to gauge demand and user experience.

Scaling up too late can result in poor performance, scalability issues, and outages; scaling down too late results in wasted resources and money

Chapter 3

Architecture

The aim of this project is to offer a lightweight solution for monitoring Docker containers running over a set of clients. The overall architecture consists in a pool of five (arbitrary) agents and a single controller:

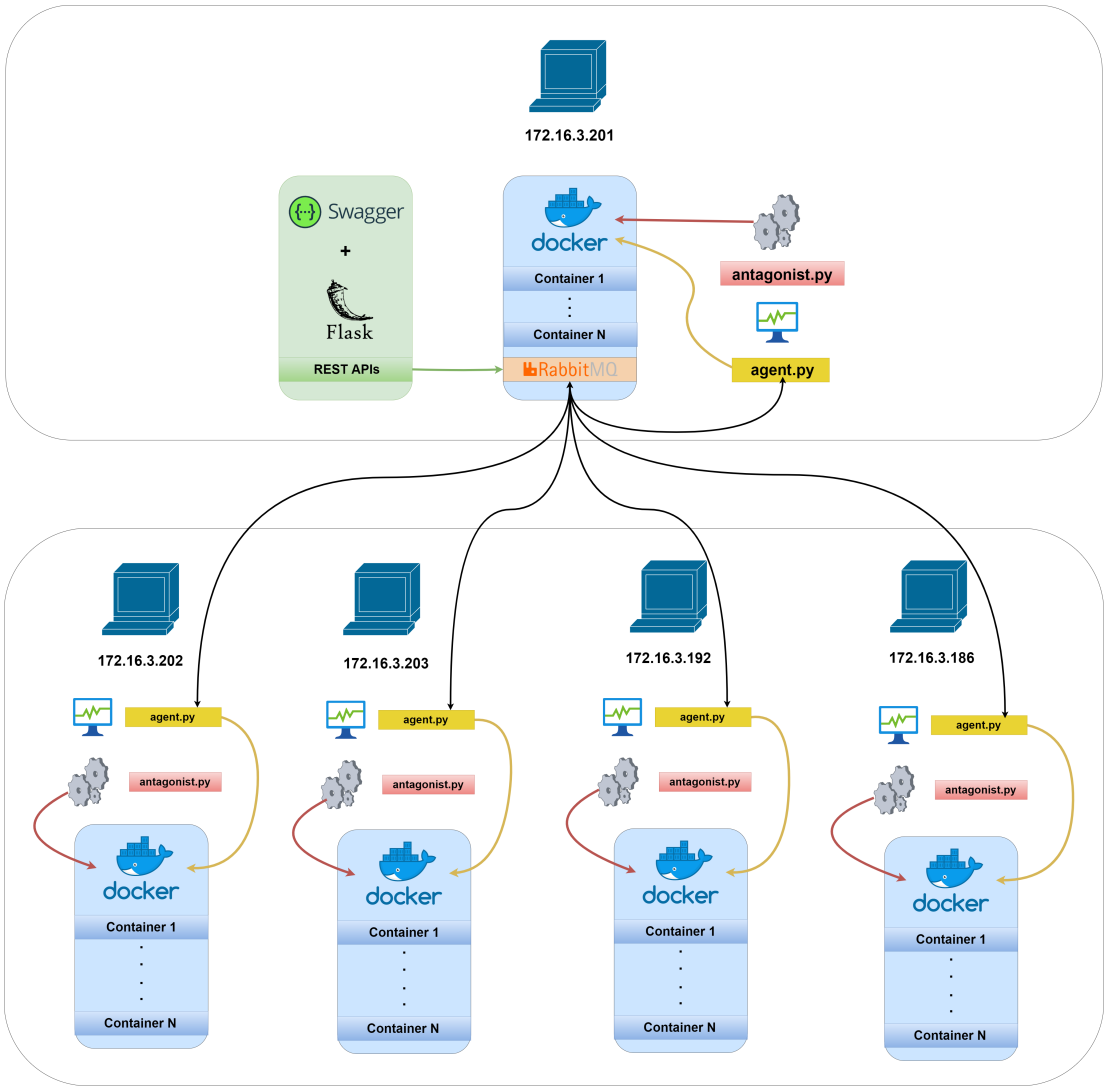

Figure 3.1: Architecture Scheme

In our case, the addresses of the involved actors are the following:

1. Controller [172.16.3.201]
2. Agent 1 [172.16.3.201]
3. Agent 2 [172.16.3.186]
4. Agent 3 [172.16.3.192]
5. Agent 4 [172.16.3.203]
6. Agent 5 [172.16.3.202]

The agent software module runs on each host and performs periodic pings to every container, from a list of monitored ones, to check their wellbeing. If a container is down or experiences a packet loss higher than a given threshold, the agent must destroy it and then restart it. Both the containers to be monitored and the packet loss threshold are provided by the system administrator via REST APIs.
The APIs were generated by Swagger Editor and exposed by the Controller module combined with Python Flask Server. The REST interactive panel for the system administrator is available at:

$$\boldsymbol{http://[IP\_of\_Controller\_Node]:8080/ui} \tag{3.1}$$

The controller is also responsible for implementing the RabbitMQ Broker Service as a Docker container, to let the other agents communicate asynchronously with the controller. Indeed, all the commands sent out by the system administrator through the APIs and the replies from the Agents are handled by a Message Queue System.
An "**Antagonist**" program, written in Python, runs in background on each host to introduce a given percentage of packet loss and some delays on the Docker Network Interface; it can also stop some containers randomly.

Chapter 4

Implementation

This section describes the implementation choices adopted for the development of the Health Monitoring System.

4.1 Agent

The "Agent" module installed on the various hosts is responsible for managing the ping of the containers distributed on the machine, periodically sending the updated list of containers currently running and finally interacting with the controller module via the REST APIs.
Since the three operations necessary for the correct functioning of the module are blocking (for some commands) and concurrent, they are performed on three different threads. The module has a "Config.ini" configuration file, which contains all the configuration settings useful for correct operation; in particular, the following fields must be set:

- **my_ip**, the IP address of the machine on which the "Agent" module is running
- **threshold**, which corresponds to the default threshold set at the time of execution
- **broker_ip** e **broker_port**, which respectively indicate the IP address of the machine on which the RabbitMQ broker runs and the port on which it is listening

The "Agent" module consists of 3 main submodules:

- **ContainerManagement**, inside which there is a module for container management and a module for making ICMP requests and responses on the various containers
- **RabbitInterface**, which deals with managing communications with the RabbitMQ broker
- The file **main.py**, which exposes the "Agent" module externally

To interface with Docker containers, the Python "DockerSDK" library was used. For ICMP requests, the Python "PingParsing" library was used, which allowed us not to have to expose this request to the outside through a call to the subprocess module, but to be able to use the ping command directly inside of the code and consequently

manage the packet loss rate like a normal dictionary.
As for the ContainerManagement submodule, we can find two files inside it called **Handler.py** and **Pinger.py**.
A global variable monitorized_containers has been created in the **Handler.py** file, which contains the list of containers that the system administrator has decided to monitor through the controller. The functions that are part of this file are:

- *update_treshold(new_treshold)*: this function updates the value of the threshold variable
- *kill_and_restart_container(target_container)*: this function destroys a container that has packet loss above the threshold and restarts it
- *add_to_monitorized(target_container_id)*: this function adds a container to the monitorized_containers list
- *stop_monitorizing_container(target_container_id)*: this function removes a container from the monitorized_containers list
- *start_monitoring_all_containers()*: this function starts the monitoring of all running containers deployed on the machine
- *stop_monitoring_all_containers()*: this function stops the monitoring of all running containers deployed on the machine
- *get_active_containers()*: this function retrieves the running container list and sends it to the controller

The **Pinger.py** file has the following function inside:

- *ping_containers*: this function pings the containers in the monitorized_containers list every n seconds

Finally, it should be noted that the functions that write to the variables perform a lock, to handle concurrency problems; when these problems do not exist, that is when the functions read the values of the variables, the lock function is not used.

4.2 Controller

The "Controller" module has two files of interest: containers_controller.py, located in the controller/swagger_server/controllers submodule, and main.py, located in the controller/swagger_server submodule.
The first of the two files, container_controllers.py, defines the following functions, associated with the REST APIs:

- *get_containers_list*: This function retrieves the list of active containers on the platform, taking data from the global variable also used by consumer_Rabbit
- *monitor_all_containers_on_host(host)*: This function sends the command to a specific agent to add all containers to the monitored container list
- *stop_monitorizing_all_containers_on_host(host)*: This function sends the command to a specific agent to remove all containers to the monitored container list
- *set_monitored_container(host, containerID)*: This function sends the command to a specific agent to add a specific container to the monitored container list
- *stop_monitorizing_container(host, containerID)*: This function sends the command to a specific agent to remove a specific container to the monitored container list
- *change_treshold(treshold_value)*: This function sends the command to all agents to update the treshold for packet loss with a specific value

The __main__.py file has been modified with the introduction of a new thread that asynchronously executes the Rabbit consumer for container list updates. By default, it still runs on another thread of the swagger server.

4.2.1 REST Interface

The "Controller" module oversees exposing a simple API interface for monitoring and interact with Docker Containers.
The available commands with the respective endpoint are listed below:

- **GET**
  - *monitoring/*: retrieve information about the list of containers up and running on each host. The response consists in host ip, container id, container name, local container ip address, timestamp of the last update
- **PUT**
  - *monitoring/treshold/treshold_value*: changes treshold value for packet loss
  - *monitoring/host*: this method allows to add all the docker containers on a specific host in the list of monitored ones. It requires to specify the host IP value
  - *monitoring/host/container_ID*: this method allows to add a specific docker container on a specific host in the list of monitored ones. It requires to specify the host IP value on which the container is running and the container ID to be monitored

- **DELETE**
  - *monitoring/host*: stop monitorizing all running containers on a given host. You need to specify the IP of the host on which you want to stop monitorizing the containers
  - *monitoring/host/container_ID*: stop monitorizing a specific Docker container running on a certain host. You need to specify the IP of the host and the ID of the container that you don't want to monitor anymore

4.3 Antagonist

To test the health monitoring system, a small program was developed, called "antagonist", which aims to artificially introduce delays in communication and packet loss; for the implementation, the Netem library was used. It can also stop some containers randomly, if they are not contained in a list of containers that cannot be stopped (RabbitMQ).
This program runs on each health monitoring system host. The percentage of packet loss and the delay in communication can be entered manually, by changing simple parameters within the program:

- *add*: to launch the command for the first time
- *change*: for any subsequent changes in delay or packet loss
- *del*: to remove previously inserted delay or packet loss

4.4 RabbitMQ

In this project we implemented some of the RabbitMQ functionalities for message-queueing to let the various components communicate. We have been exploiting Pika library, which is a pure-Python implementation of the AMQP protocol including RabbitMQ's extensions.
Specifically, the following Exchange types have been used for different porpouses:

1. **Fanout Exchange**: required for broadcasting messages (commands) sent out from the REST APIs to all the agents that are bound to the "containers" queue

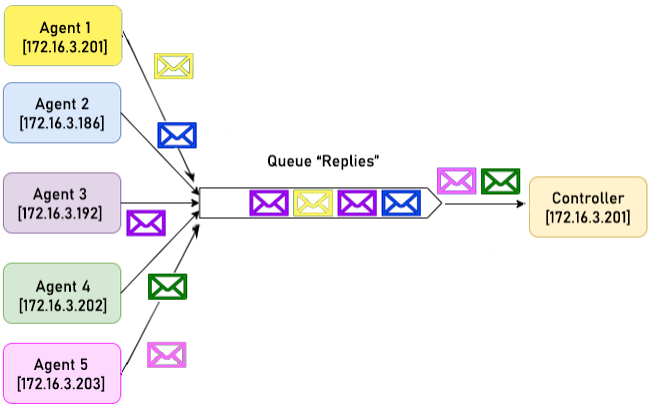

Figure 4.1: Unicast Queue

2. **Direct Exchange**: required for delivering messages from all the agents that periodically (every 20 seconds) send to the controller their updated list of active containers. A direct exchange is ideal for the unicast routing of messages: in our case this scenario occurs after the system administrator invokes the GET method to know which are the containers up and running on every host. This method returns the value of the variable active_containers_list, which by the way is filled every time the controller receives an update message from the direct exchange queue

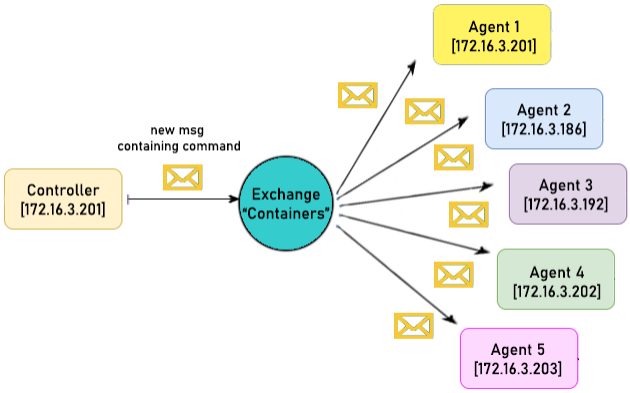

Figure 4.2: Fanout Queue

4.5 Architecturally Significant Use Cases

In this paragraph are reported the sequence diagrams of the GET and PUT methods in the attempt to clarify the interaction between modules and software components. The other implemented methods share the same seuquence diagram of the PUT.

- **GET**:

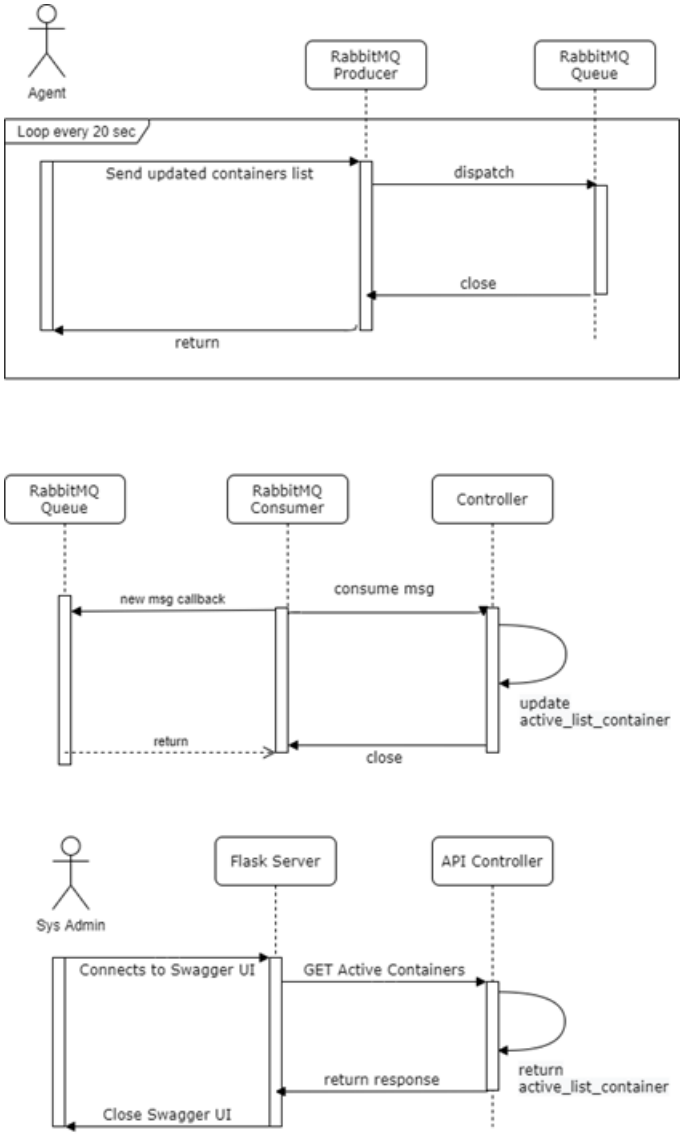

Figure 4.3: Direct Exchange

As described in the diagram, the Agent module sends every twenty seconds a message through RabbitMQ, containing the updated list of its own active docker containers. The controller module consumes the message in the callback method and if it spots any differences from the last update received, it updates the value of the global variable list_containers.
The GET method is simply going to print the state of that global variable

list_containers.
This way, we were able to implement the actual Space & Time Uncoupling between agents and controller

- **PUT**:

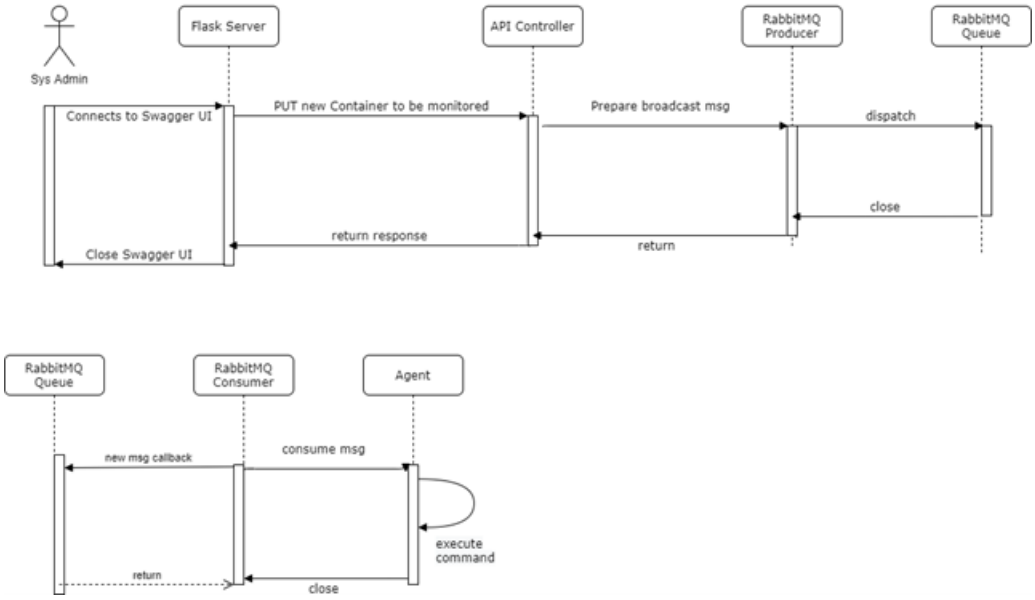

Figure 4.4: Fanout Queue

The PUT method behaves different from the GET; as shown above, this kind of commands are sent broadcast to all the agents connected to the “Containers” queue, so that they can independently consume it and execute.