UNIVERSITÁ DI PISA
ENGEGNERIA DELL'INFORMAZIONE

Health Monitor System for Docker Containers

Project Report

Barsanti Nicola
Casu Pereira de Sousa Bruno Augusto
Fregosi Frederico
Lemmi Laura
Martino Giuseppe

Group: Lost (in the Cloud)

876II CLOUD COMPUTING

Pisa, July 2021

Contents

Introduction

The project for the Cloud Computing course aims the development of a Health Monitor System for a Docker based platform. The system then must detect faults and connection issues from the containers running on the platform, and if detected, the monitor system must act and reboot the faulty container. For the project, the monitor system will also provide a REST interface, in order to display to the user information on the wellbeing of the containers and updates from the monitoring, as well as some control functions to change internal monitoring parameters of the system. These activities will then be divided in two subsystems, one responsible for the local assessment and communication with the containers, defined as the Manager, and one responsible for the control, interface and communication of the system, defined as the Controller.

With this architecture, the monitor system will have a dedicated control over the containers by designating a Manager to each Virtual Machine (Docker Host) of the simulated cloud environment, combined with an overseer that operate the communication and issues commands to the monitoring units.

The Managers will then be responsible to retrieve a list of all the active containers on the platform, send periodic probes (TCP ping protocol) and account for packet loss, executing a restart process if needed. The Controllers on the other hand will gather the information from the Mangers, issue commands and display the data on the REST interface, while the intelligent monitoring is active.

For the communication between the submodules, a RabbitMQ message queue system will be used, connecting all the submodules deployed in the platform. In this way the commands issued by the user in the REST interface will be transmitted to the Controller that will furthermore send the parameters to the Managers in the platform. In addition, the communication system will be used to transmit the monitoring data from the Managers to the Controller, that will update the interface, again using the queues.

With this implementation, the user will be able to survey the active containers, and set the parameters for the monitor system, such as the maximum value for the loss of packets in the connection and add containers to an ignored list (remove the unit from the supervision of the system).

To test the developed system an Antagonist program was developed to insert an artificial packet loss and to randomly disable some containers on the platform. The attacker was also programed to perform the intrusions with a random time interval and duration, making distributed and unpredictable attacks during the test run.

With the proper testing procedures, the efficiency of the Health Monitor System and its ability to keep the containers running will be estimated, characterizing the platform availability. The implemented submodules, the communication system and the server (executing the REST interface) were based in a set of Python scripts, which were executed on the provided Virtual Machine environment. The schematics of the interface developed are available in the following link: <172.16.3.167:800> (the page provides guidelines on how to operate the system and a preview of the user interface, to access the link, it is necessary to be connected to VPN of UNIPI).

Health Monitoring System architecture

In order to follow the design instructions for the Health Monitoring System (HMS) development, the activities performed by the system were divided into two separate submodules, they are the Manager and the Controller. Those submodules then must provide all the features expected for the HMS operation, such as: measurement of the packet loss occurring in the container communication and restarting all the components when a fault is detected. In the context of the project, a fault is characterized when the HMS detects a packet loss greater than a preset threshold or if a container was incorrectly disabled (attacked). In addition to the basic monitoring and correction protocols of the system, a control module must also be designed to provide the user the information of the active containers on the platform and to set the threshold parameter for the packet loss.

The functionalities of the HMS are expected to be executed in an environment composed of 5 Virtual Machines (deployed on the same network). On each of these VMs, a Docker host will be running some idle containers, and it is desired that the HMS oversees the operation of all the running containers.

For the project, a particular design was defined, and for each VM a dedicated container supervisor will be deployed. This submodule is then defined as the HMS Manager. As previously mentioned, its basic function is to list and verify the active containers on the platform, detecting any inappropriate shutdowns. Also, the local Manager must probe the active containers using an Internet Message Control Protocol (ICMP), also known as the Ping command, and account for the packet loss. In case of a detected fault, the submodule will interact with the Docker engine requesting the restart of the container where the error was spotted.

The proposed design architecture also implements a general control and interface system, that will oversee the multiple Managers deployed on the VMs. This submodule is defined as the Controller, and it must establish a communication protocol to issue commands to the Mangers and retrieve the updates from the monitoring. In this way the information is displayed to the user by means of a web interface, combined with the option to set the Managers packet loss threshold and its monitored containers. A schematic of the placement of the submodules on the platform is illustrated in Figure 1:

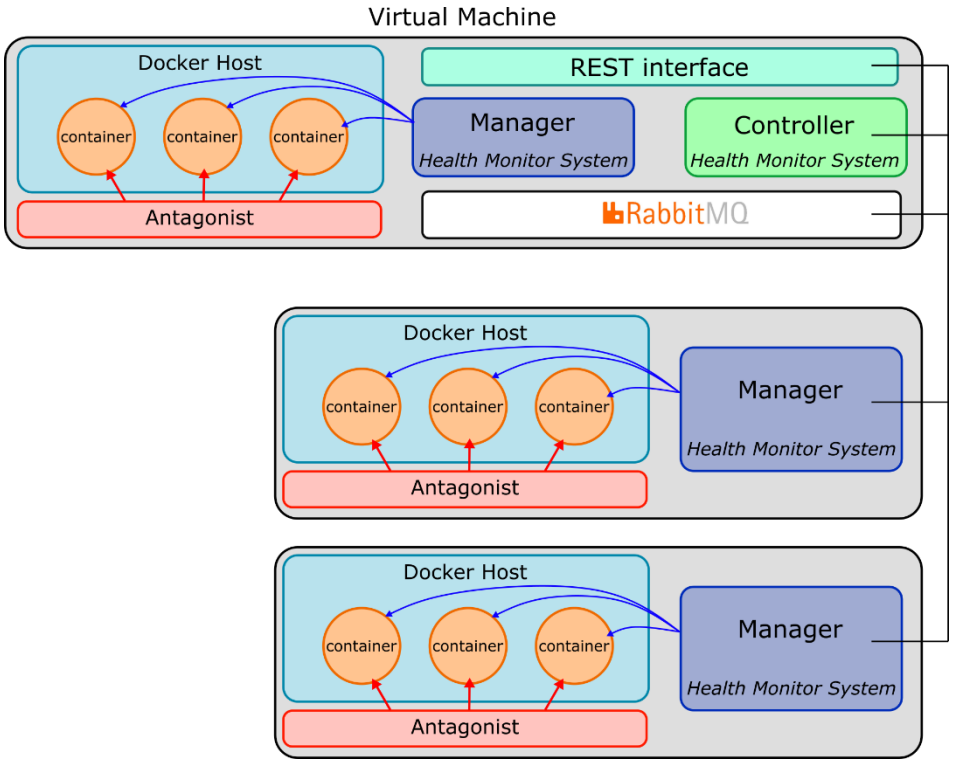

Figure 1 – Health Monitoring System architecture

From the image it is noticeable that a few extra components will also be part of the platform. For the communication between the submodules a RabbitMQ message broker will be used, in order to transmit the commands and data from the deployed submodules that composes the HMS using several message queues.

Also, the other unit present on the schematic is the antagonist agent, used for the testing and validation of the System. Both these structures will be further presented on the report.

Manager description

The first submodule to be detailed is the Docker Manager, or HMS Manager. The main goal for this module it is that it can interact with the containers and check their status and network connection. The software was based on a Python script, implementing a thread that maintains the monitoring during the run time. This software is deployed as the HMS is started on the Docker host and immediately begins its operation on the active containers.

The thread then uses the Docker python library to interact with the running containers and the Docker engine. With the APIs from this library, it is possible to list all containers and execute commands, like restart, check for status and even deploy new containers. Using these functions, the Manager then implements a monitoring routine based on a loop interaction, illustrated in Figure 2:

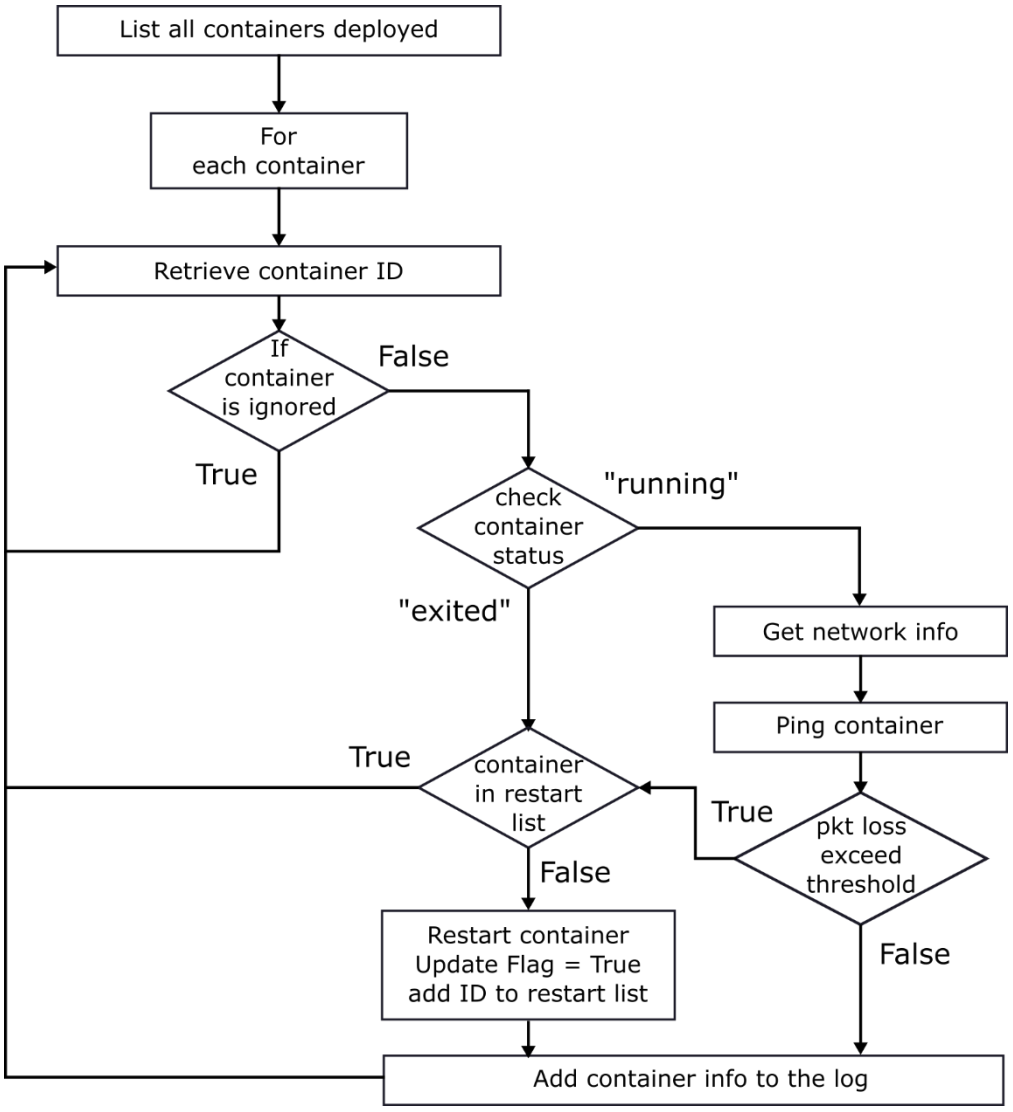

Figure 2 – Manager monitoring algorithm

This loop basically consists in a sequence of checks, and if any error is detected the thread issues a restart command to the Docker engine, that stops the container and re-deploy it. To keep the system consistent, a restarted list is used to add any container that is current in the process of restarting. This list then checked before the execution of the restart, as a container may be still in the middle of the process. If the ID is contained n the list the HMS skips the restart protocol. At the beginning of the checks, any container that is running is removed from the list, as in this case the restart process has already finished.

All these activities are logged by the thread, saving the information of every interaction in a Python list. After the verification of all container IDs listed at the beginning of the execution, the Manager check if the Update Flag is set, and if it is true the module then sends the Controller an update message using the queue.

After the update message is issued, the Manager is then ready to send the last monitoring log, with the information of all running containers, that are: the container image, its IPv4 address, the packet loss calculated and the HMS status (in case of an ignored container, the information displayed is only the short ID and the HMS status as “ignored”). The transmission of the monitoring log will be executed when the Manager receives a ‘give content’ command issued by the Controller, characterizing the synchronous operation of the HMS.

The Manager can also receive other commands from the Controller interface, allowing the configuration of the monitoring parameters, such as which containers are monitored (by adding or removing the ID from the ignored list) and the threshold value (percentage of maximum packet loss until a restart is issued for the container). These interactions are illustrated on the following images:

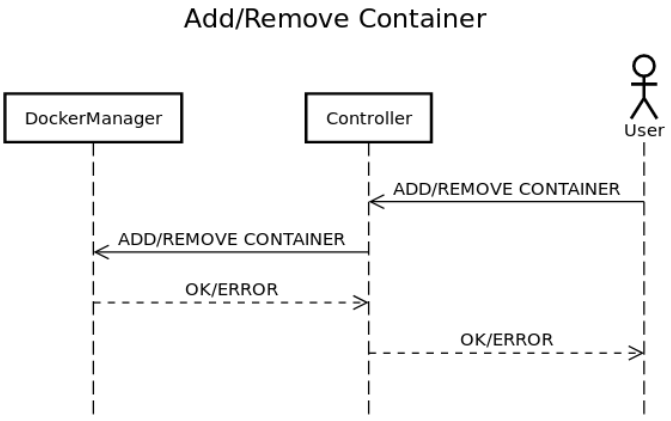

Figure 3 – Add/remove container from the ignored list message protocol

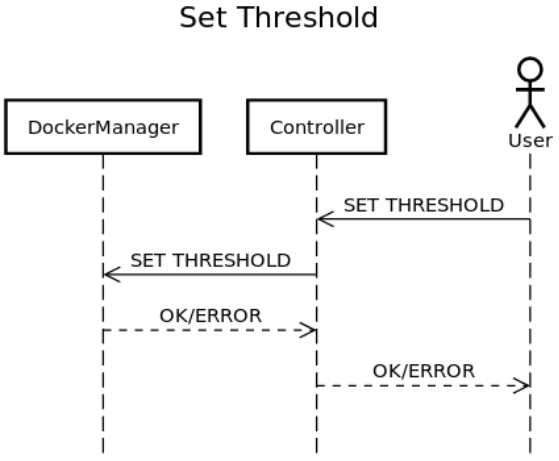

Figure 4 – Set packet loss threshold value message protocol

From the schematics it is noticeable that as the Manager receives the command it tries to execute and implement the received parameters. In case of an inconsistency, missing value or incorrect ID, for example, the Manager informs the Controller the error. If no inconsistencies are detected, an OK message is sent back, and the monitoring process continues to be executed.

Another important consideration for the proper management of the monitoring is to be able to send and receive the ICMP messages from the containers. For this implementation, the Manager program must be able to establish a connection with the Docker network bridge, allowing the containers running inside of it to be accessed from the external unit. This design was preferred due to the possibility of an attack on the HMS Manager, as if it were created inside the Docker network (inside a container, for example), the whole system would be vulnerable to the antagonist interferences.

Therefore, the Manager was kept in a separate module form the Docker network, and because of this, the bridge must be accessed from the Host IP. For this configuration, a second library had to be used on the program, and it is the *icmplib* version 3.0. From the available APIs it is possible to retrieve the Host IP address and it network configuration. With this information, the Manager software then uses the Ping command to access the containers. This command sends several packets with a defined interval.

For the project, the configuration of the probe messages was set on the following manner:

```
ping(ip_ctr, count=20, interval=0.01, timeout=0.1)
```

The ‘ip_ctr’ variable is the IPv4 address of the target container to be monitored, and it is retrieved by using the Docker Python API equivalent to the inspect network bridge command. The other parameters represent the number of packets sent, the interval in seconds between the packets, and the maximum timeout value for the response (in seconds), respectively. The interconnection between the HMS Manager and the Docker daemon and network are illustrated in Figure 5:

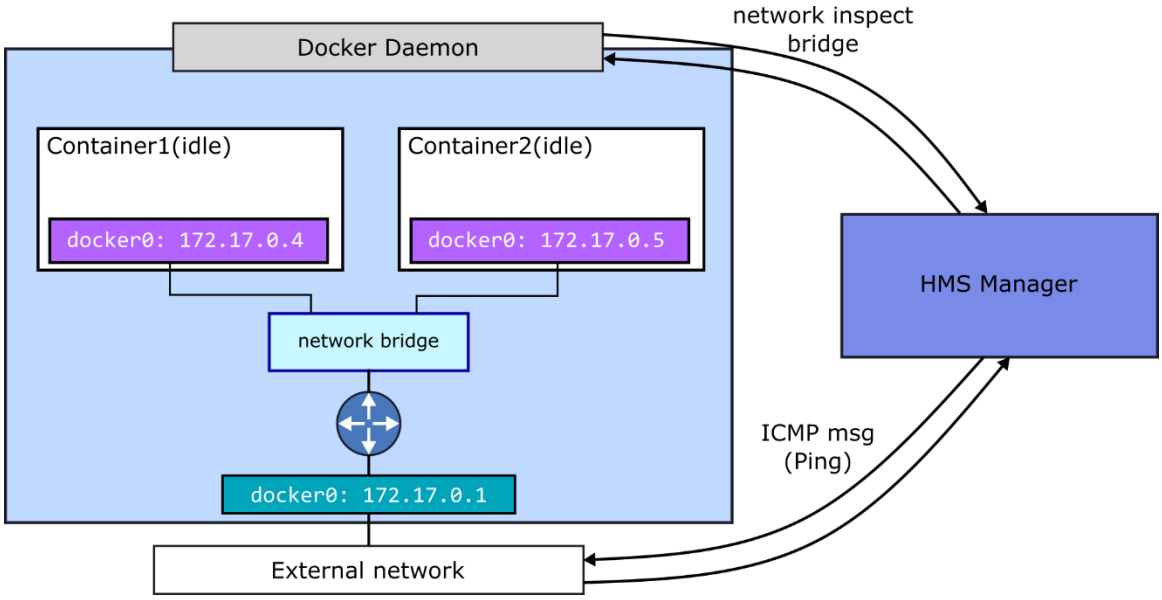

Figure 5 – Manager connections with the Docker Host

With the previous configuration the supervision of the system is guarantied by the Manager, as it can interact with the daemon and retrieve information of the containers, as well as the TCP parameters to perform the Ping command. This structure then establishes the basic operation of the monitoring system.

On Figure 6 a preliminary test is shown, demonstrating the operation of the Manager. On this scenario a total of four containers were running on the platform, and the local manager was able to properly Ping all container without any loss (the packet loss percentage is calculated as the return value from the Ping API contains the number of packets sent back by the target address).

Figure 6 – Manager test on four containers running

```
root@brunocasu-desktop:/home/brunocasu/Documents/CC/proj# python3 hmsmanager.py
begin test
Manager IP [ 127.0.1.1 ]
<Container: 8db94990d2> IP addr: [ 172.17.0.5 ] Pkt Loss: 0.0 % True Active
<Container: 5c43514eb0> IP addr: [ 172.17.0.4 ] Pkt Loss: 0.0 % True Active
<Container: 5385dc0e57> IP addr: [ 172.17.0.3 ] Pkt Loss: 0.0 % True Active
<Container: 8f18eb2007> IP addr: [ 172.17.0.2 ] Pkt Loss: 0.0 % True Active
end test
root@brunocasu-desktop:/home/brunocasu/Documents/CC/proj#
```

Controller description

After the configuration of the active agent of the monitoring system (the Docker Manager), the next submodule to be detailed is the Controller. The idea for the controller is that it will manager all the communication from the module that interacts with the Docker engine and the user/administrator. This overseer then must handle the requests from the user and the updates from the Managers, enabling the configuration and customization of the monitoring system, as well as the display of the status of the platform (availability and communication bandwidth).

The program for the Controller then executes a thread that initializes the monitor system by deploying the Manager and maintaining a supervision over the platform. The loop holds until any updates are issues, as the manager detects the faults. Considering the division of the system, the Controller must also keep track of the events on the monitoring system. This is done by gathering the received logs from the managers, and further displaying them to the user interface.

To efficiently retrieve the information from the monitoring units, the Controller will receive the updates in an asynchronous manner, that is immediately when the Managers finishes its monitoring routine (considering also that some action on the system was performed). From the asynchronous messages the Controller will then schedule a time to request the updates and receive the actual logs. For this communication, a synchronous message protocol will be executed. The next diagram illustrates this process:

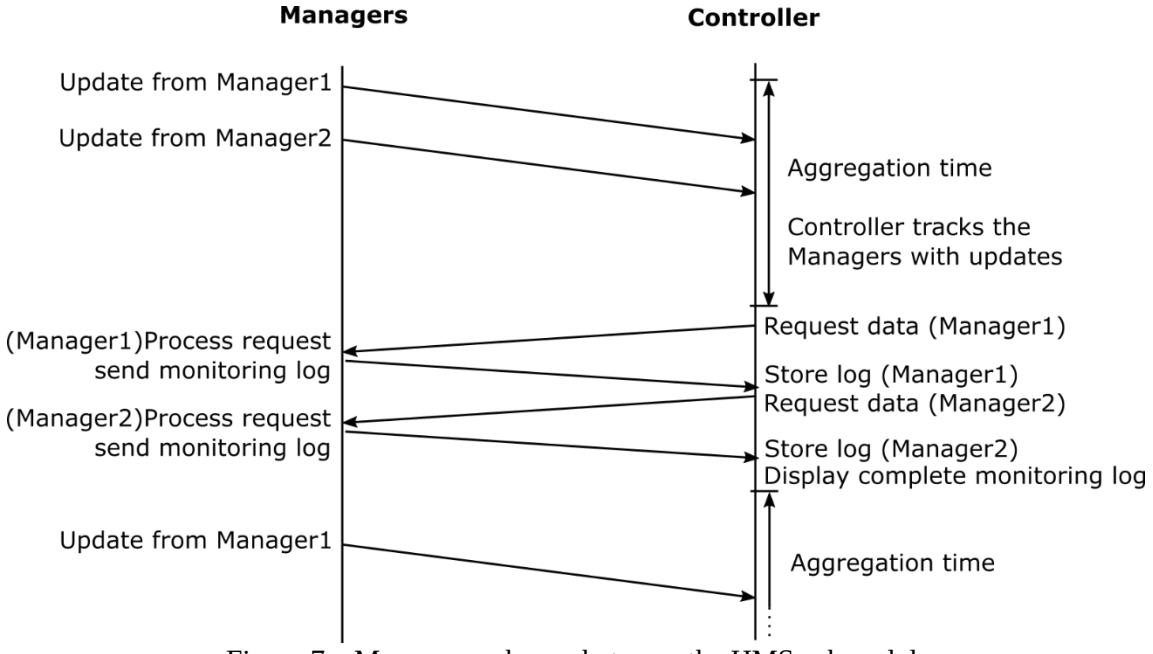

Figure 7 – Message exchange between the HMS submodules

From Figure 7 it is noticeable that the updates from the Managers can arrive at any given time, thus characterizing the asynchronous behavior previously mentioned. Also, the image shows the defined aggregation time that is kept before gathering the update requests to only then issue the synchronous commands for the Manager to transmit their new logs.

For a whole view of the Controller operations, a schematic is shown in Figure 8. From the main thread the controller maintains its standby status until an update message arrives. From there the Controller verifies all the pending Managers to have an update an performs the request process. At the end, the submodule executes the commands to display the information to the user.

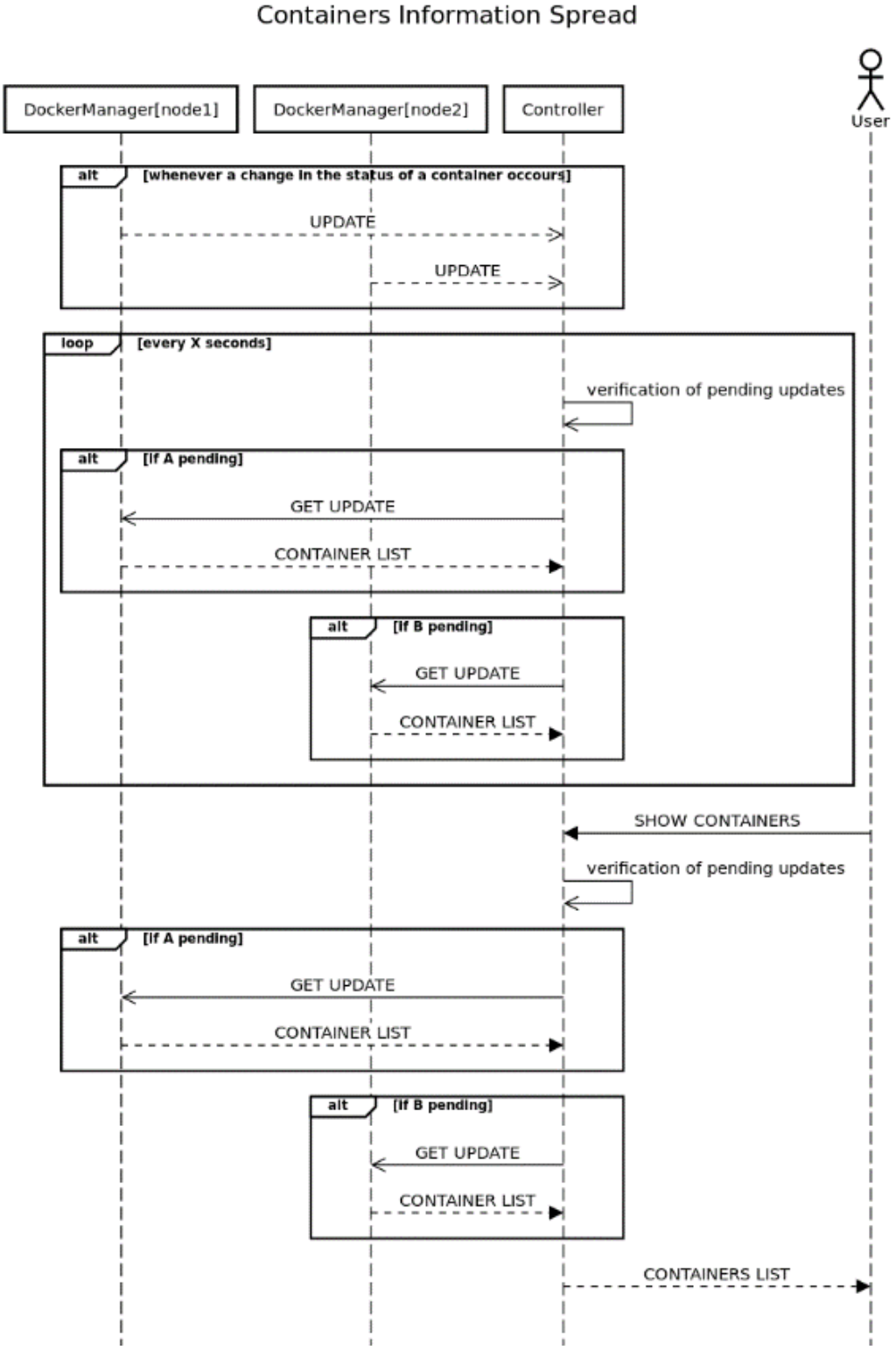

Figure 8 – Controller thread schematic

In addition to the supervision, gathering and displaying the information from the Managers, the Controller must also deploy the Managers on the platform. Once active, those units will send a heartbeat signal as an indication to the Controller if the module is offline or if its active. Figure 9 illustrates this secondary process executed by the Controller:

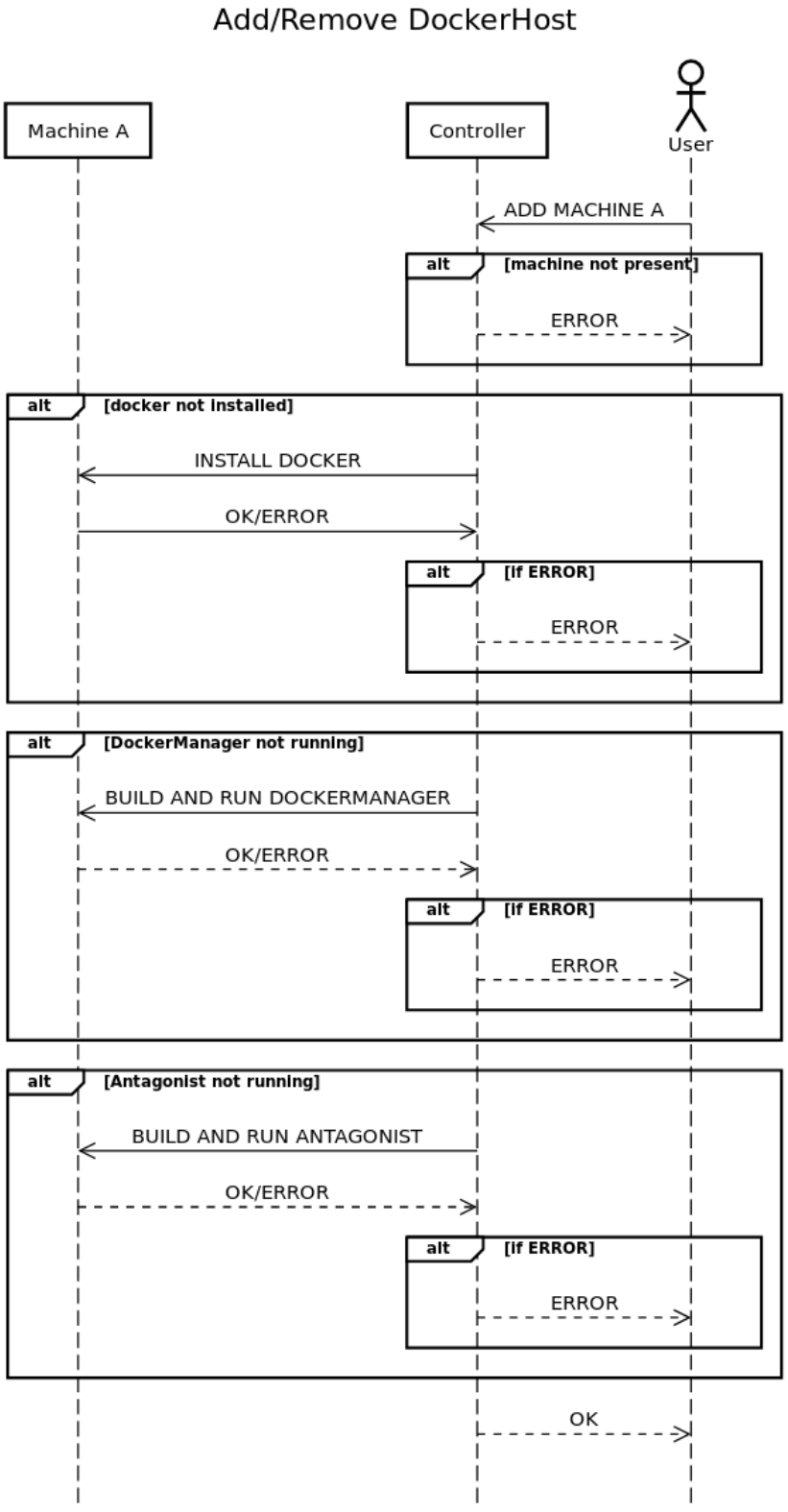

Figure 9 – Manager submodule deployment schematic

REST Interface

Complementing the activities executed by the system Controller, a REST interface was deployed to provide the user/administrator a visual and manageable interface with the HMS. In agreement with the project specifications, the interface must then expose all the functionalities of the running HMS. This is executed from a deployed swagger-enabled Flask server, as the system is launched on the platform.

The server will be deployed as an independent component on the VM where the HMS Controller is operating, being executed form a python script as well. As the monitoring system is launched inside the machine, the modules for a swagger server will be executed deploying the server with the REST APIs used to manage and control the HMS.

From this build, the Controller will exchange information with this module in the form of .json files, as the corresponding messages will be converted to the format and sent using the message queue already available and running on the platform (the deployment of the RabbitMQ system will be prior to the execution of the server)

. The information of the containers from Managers will be forwarded to the server using the Controller as the intermediate agent. This allows the HMS to evaluate the bandwidth used to make the monitoring information available for the user, as the web interface is updated. The placement and connections between the mentioned modules are exemplified in Figure 10.

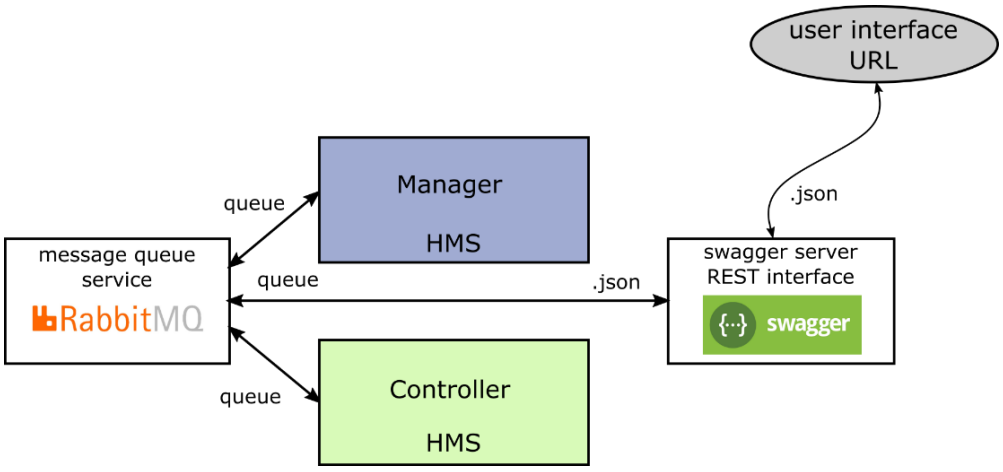

Figure 10 – Manager submodule deployment schematic

The server will then make available for the administrator the commands necessary to configure the HMS parameters and to make available the status of the platform, including the list of active containers. The interface will use the standard command definition from the REST API, as it will execute the following methods for the container's management:

1. GET/containers – to retrieve the list of running containers that are monitored by the system with their respective status from the monitoring system
2. GET/containers/{address} – Retrieve information of all containers from a single Docker Host
3. GET/containers/{address}/{containerID} – retrieve the information for a single container in a target Docker Host
4. PUT/containers/{address}/{containerID} – to add a container to the monitoring system supervision (allows the HMS to take actions and monitor the parameters for this unit)
5. DELETE//containers/{address}/{containerID} – to remove a specific container from the monitoring system (add it to the ignored list)

In addition to the container display, the REST interface is also used to issue commands for the Managers, to set the monitoring parameters. The commands also make use of the methods provided by the interface. The selected commands then trigger the Controller algorithm, that retransmit the parameters to the Managers on the platform. Also, the controller of the running managers on the platform can be set by the interface. The following commands were implemented in the swagger.yaml configuration file (in addition to the previously mentioned):

1. POST/containers – Set the threshold value for the container connection maximum packet loss in all managed Docker Hosts (each local Manager will use this parameter to restart every container that has a higher packet loss detected)
2. POST/containers/{address} – Set the threshold value for a specific Docker Host from the platform (only the target local Manager unit will have its parameters altered)
3. PUT/containers/{address} – Add a Docker Host to the HMS supervision (also creates a local manager to the Unit)
4. DELETE/containers/{address} – Remove a Docker Host from the HMS supervision

The interface for some these commands is illustrated in the following figures, as well as the web interface available for the user. Noticeably all the commands have a protocol to detect any incorrect commands sent by the interface. In this case, the Controller must identify the inconsistency and reply with the correspondent error message.

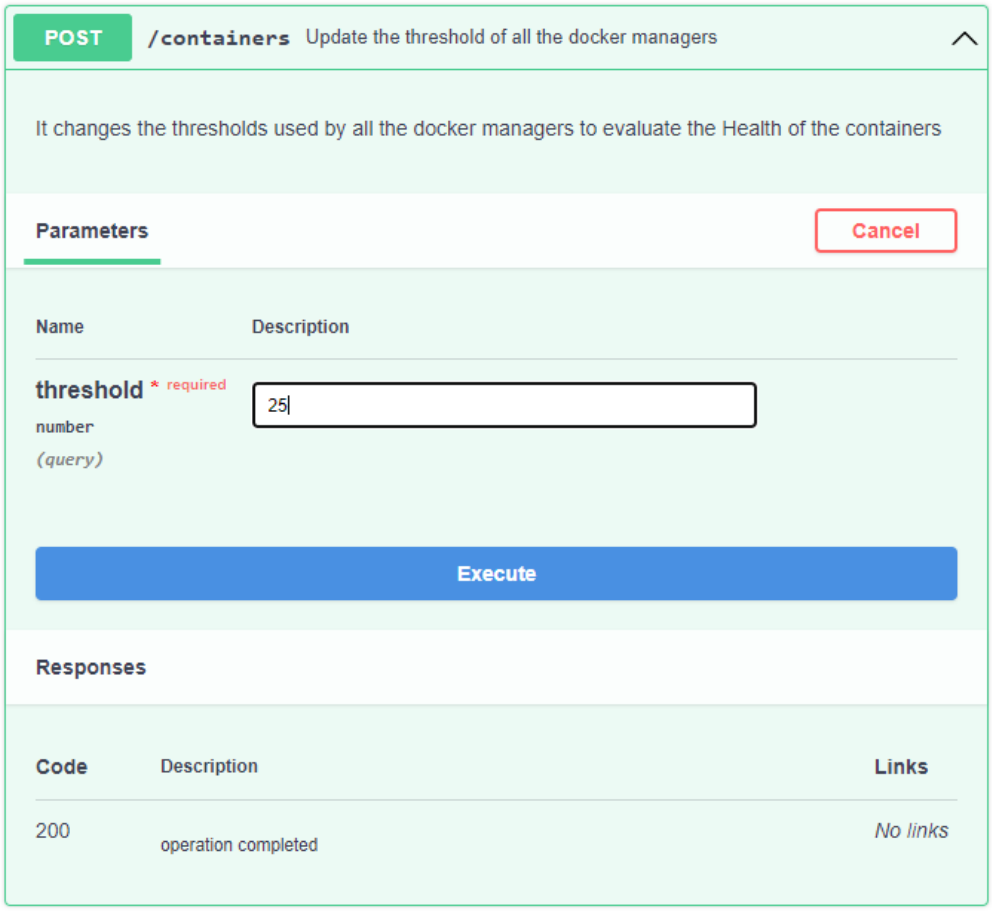

Figure 11 – REST API to set threshold value for all Docker Hosts

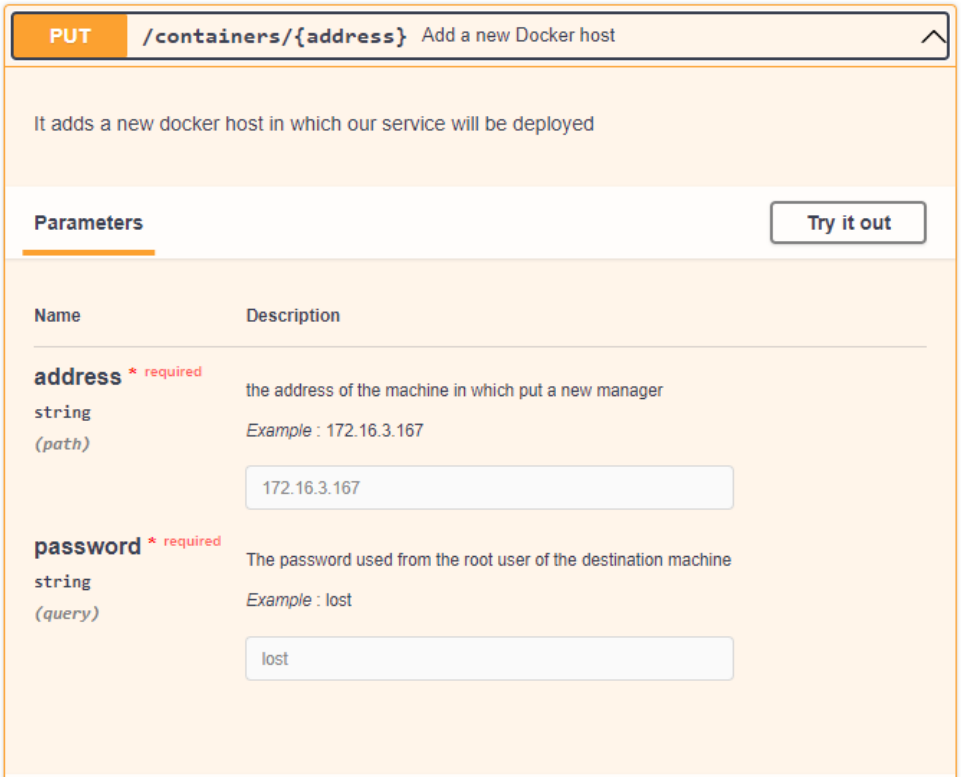

Figure 12 – REST API to add a new Docker Host to the HMS supervision

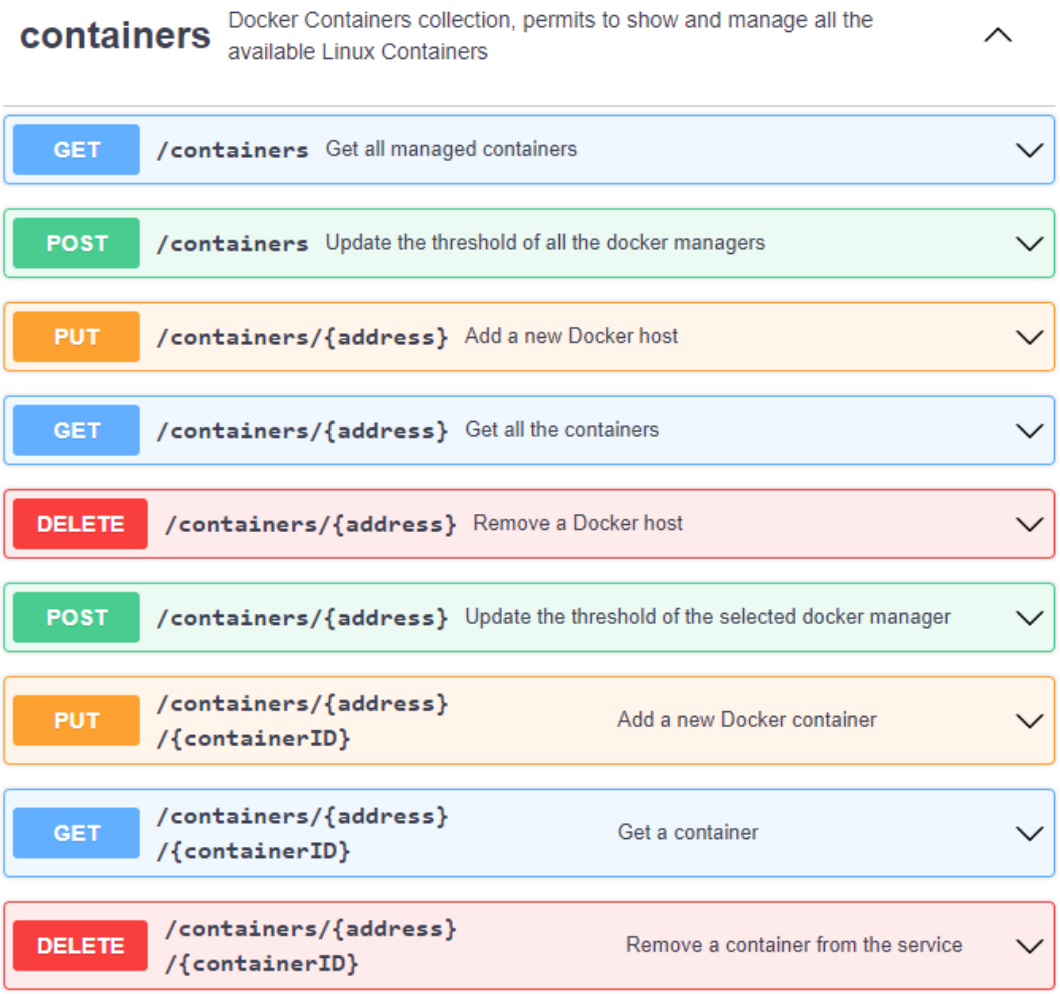

Figure 13 – REST API to add a new Docker Host to the HMS supervision

Subsystem's communication

The communication between the submodules that configures the HMS will use a message queue design, in order to maintain the consistency of the information sent from the Managers to the Controller. This design is useful when using multiple modules deployed in a platform, as more than one component may try to send a message to the system Controller at the same time, causing some sort of conflict in the messages received.

For the design of the HMS, the RabbitMQ software together with the PIKA Python library will be used to organize the data sent from the submodules, as they provide a collection of APIs to send and receive messages from a set of defined queues. From the RabbitMQ architecture several keys will be used to bind each submodule to a particular queue, allowing the Controller to receive simultaneously and in an asynchronous form the update messages from the several Managers deployed and to send the configuration commands.

When executing the HMS, each deployed subsystem will execute a rabbit.py program and will create its channel, allowing other entities from the system to connect and communicate with every instance of the system.

The multiple Managers will then receive the parameters and commands using this interface combined with a simple messaging format used to describe the functions and data parameters from the messages. Also, from the REST interface, the user will be able to send the desired commands to configure the Managers. This information will be handled by the Controller that will make use of the message queue system to forward the commands. As an important remark, the contents of the message will be encoded in a .json format, accordingly with the interface provided by the Flask server and the Python programs deployed (all the commands and data will be converted to this format before been inserted in the queues).

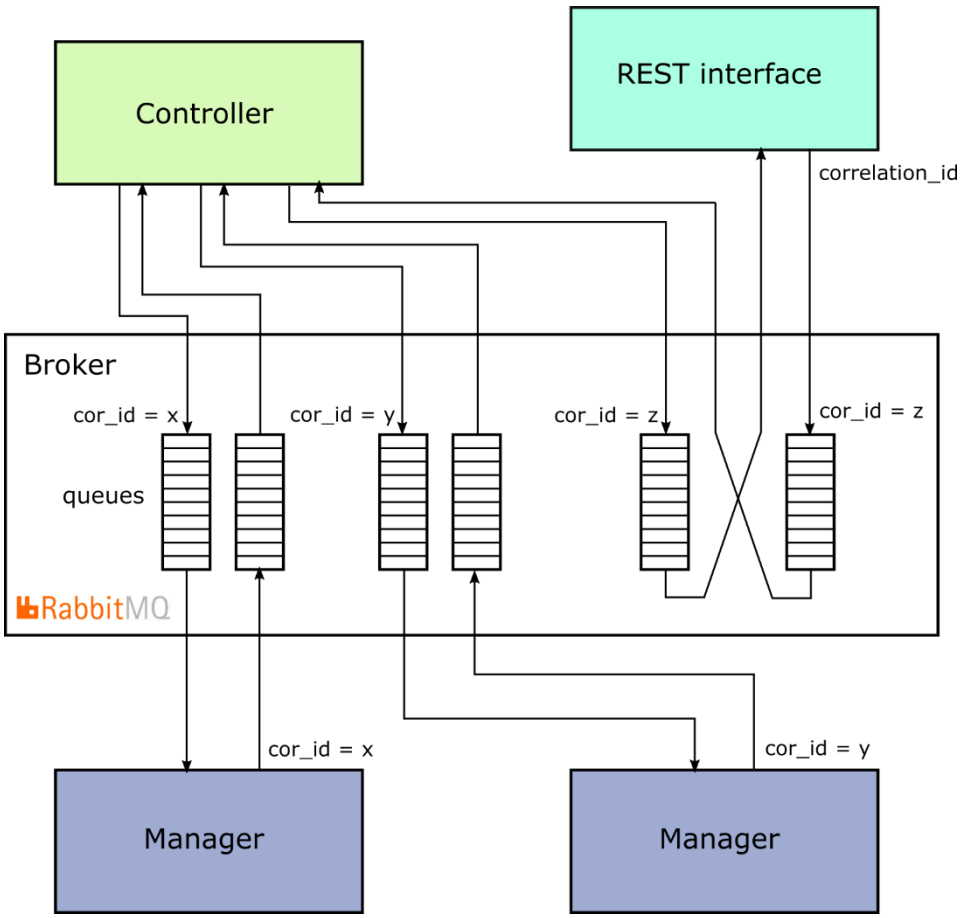

Figure 14 – HMS communication architecture design

Considering the design of the submodules and the HMS communication requirements, the message queue system had to be implemented both ways for each unit, that is, the Managers and the Controller will need to send and receive data using the queues (both must act as consumers and publishers when needed). Also, by using a property of RabbitMQ, the messages inserted in the queues will be associated to a correlation ID (cor_id, in the figure). This improves the overall system communication consistency, considering the multiple submodules that are simultaneously operating. The ID will then be used by the components to identify the requests and responses sent on the monitoring operation.

For this mutual operation, a python class was created, aggregating a set of useful commands to create the message interface on the components of the HMS, as well as connecting to the broker and the queue assignment.

In addition to the class created, a message format was also developed to facilitate the interpreting of commands by the Managers. On this format, a command definition is sent, followed by a configuration parameter. Due to the reduced number of functionalities used for the HMS (according to the specifications), the contents of the messages were simple and easily handled by the components. The information transmitted is in the form of .json files and converted to a Python dictionary when processed inside the submodules. The following structure represents the message format (already in the library format) used in the configuration of the threshold value for the Managers:

```
self._rabbit.send_manager_unicast({
                    'command' : 'container_threshold',
                    'address' : message['address'],
                    'threshold' : message['threshold']
                    }, message['address'])
```

With the interpreting of these parameters, the Manager can internally change its configuration to the value sent by the user. In the same way, when the Controller whishes to retrieve the monitoring data a from a Manager (using the synchronous message protocol) a similar structure is used:

```
self._rabbit.send_manager_unicast({ “command" : "give_content" }, dock['address'])
```

The Manager then replies with the monitoring log in the form of a Python list, using the following command:

```
return {'command': 'ok', 'address': self._manager_ip, 'content': self._monitor_log}
```

Using this message system, a few tests were conducted in order to check for any inconsistencies from the messages, as many issues may occur when the Controller tries to identify from which Manager the message are coming from (in a scenario where multiple faults are detected in different Managers). To improve efficiency and consistency in the HMS, the hostname (IPv4 addr) of the Managers were added to the message structure, as it made the proper identification of the many submodules on the platform easier.

From these preliminary tests, the basic functionalities of the system were evaluated, demonstrating its efficiency in actively prompting the restarts for the faulty containers (forcefully disabled by issuing commands to the Docker daemon) and sending the update messages to the Controller using the developed messaging queue system.

System Antagonist

The last component that will be detailed is the Antagonist. The objective of this module is to test the HMS developed by introducing an artificial packet loss into the containers network, as well as shutting down some of the containers in the platform.

This program will use the Docker Python library to interact with the local Docker engine. The attacker will then run a process where it retrieves a list of running containers, and during the offensive, it will randomly issue a stop command to the daemon using the obtained IDs, disabling the target container to simulate a fault.

A second library will be used to create an emulated network environment and introduce TCP packet loss in the docker bridge where the containers are connected. With this attack any packet coming from the exterior net and passing to the bridge is susceptible to the attack, as the emulation will randomly drop a defined percentage of packets.

The library used in this type of attack is the ‘netem’, and its functions are executed in the operational system (in the case of the VM used, the OS is an ubuntu distribution). The network emulation command used to introduce the packet loss is the following:

```
os.system("tc qdisc add dev docker0 parent 1:2 handle 11: netem loss " +
                                                    str(self._get_packet_loss())+"%")
```

The parameters inserted in the example command are the target network to be attacked, in this case the docker0 network (access bridge inside the Docker Host), the filter selected (1:2, meaning the insertion of packet loss), and the percentage of packets to be dropped on the target container. Noticeably, the command is executed from the Python program, using a defined method to executed shell commands from the script. This was particularly useful to make the Antagonists an independent agent that runs in parallel with the other submodules inside the VM where it is deployed.

Using the defined attack parameters, the Antagonist program follows a particular algorithm to perform the intrusion. This process then enables a better randomization of the attacks, approximating its behavior to a real situation, where the attacks are unpredictable and with different characteristics over time.

The algorithm used is represented in Figure 15, and it runs a loop with some preset parameters. Noticeably, the execution follows a set of configurations that guide the program to execute different patterns of attacks during the time were the Antagonist is running.

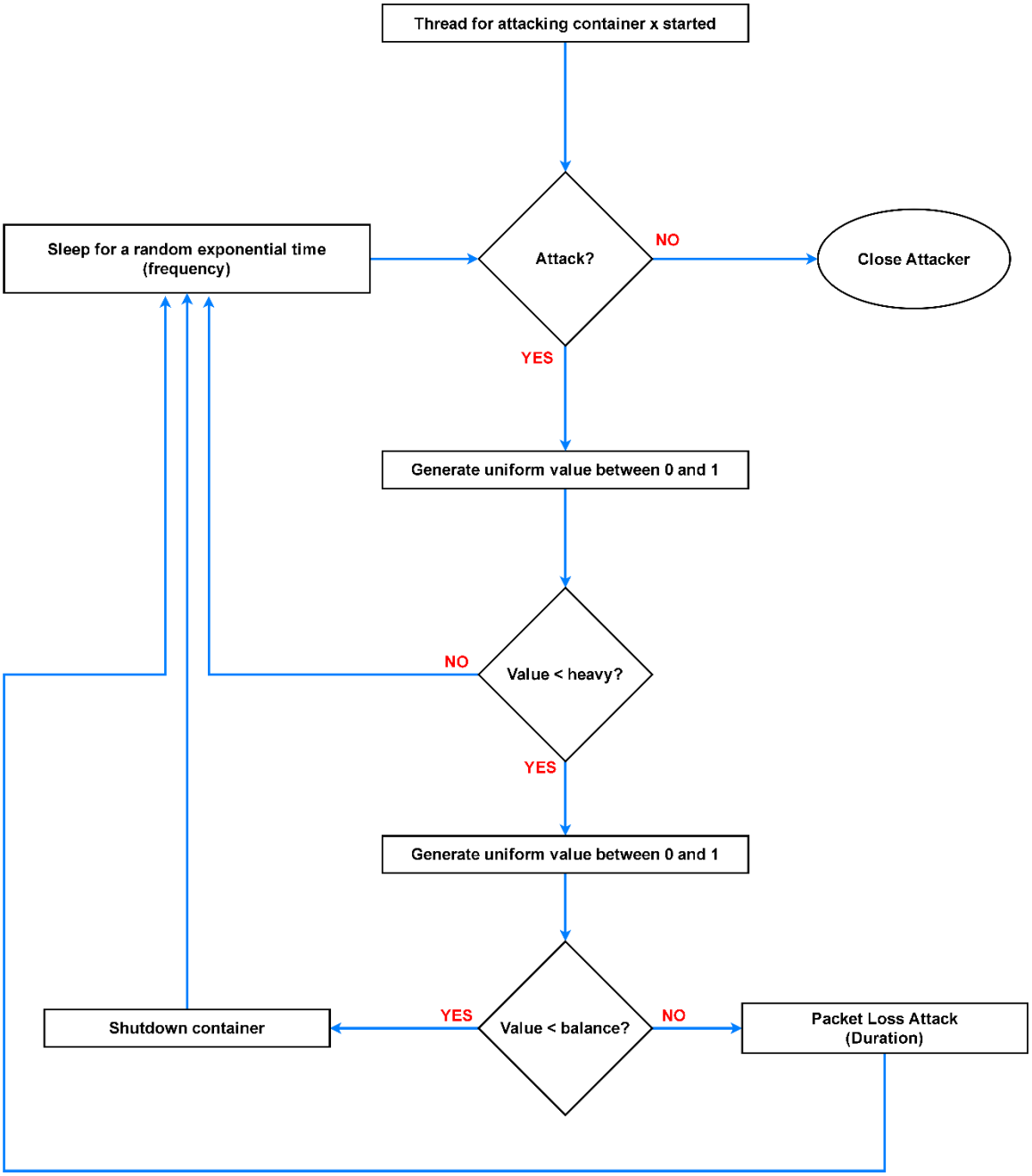

Figure 15 – The Antagonist program algorithm

In order to obtain the attack configuration parameters, which are used in the algorithm execution, the following format was used to set the intensity and duration of the intrusion performed by the Antagonist (these parameters are retrieved by the class defined):

- 'name' : # name of the test
- 'command' : 'conf_antagonist' # standard format for the Antagonist command
- 'loss' : # packet loss value to be introduced (numerical mean value for the gaussian distribution used to set the loss in the emulated network)
- 'balance' : # defined probability (in percentage) for a shutdown type of attack; (100 – balance) = probability of a packet loss type of attack
- 'heavy' : # probability (percentage) of the execution of the attack
- 'frequency' : # period (in seconds) between an attack attempt
- 'duration' : # duration of the inserted packet loss intrusion (seconds)

Some fields of this structure contain the numerical values for the parameters that will be used in the generation of the attack by the Antagonist algorithm. As a remark, the Antagonist will be executed from the Controller interface, and during the test, the program will repeatedly try to execute an attack on a target container listed from the platform, based on the pre-configured parameters as previously defined.

Antagonist test results

In order to evaluate the behavior of the HMS in face of an attack, four test scenarios were defined in the project using the Antagonist to execute the intrusions. They are based in two features, the duration and the intensity of the attack. In Figure 15 a visual representation of the performed tests is represented, demonstrating the behavior of each configuration used:

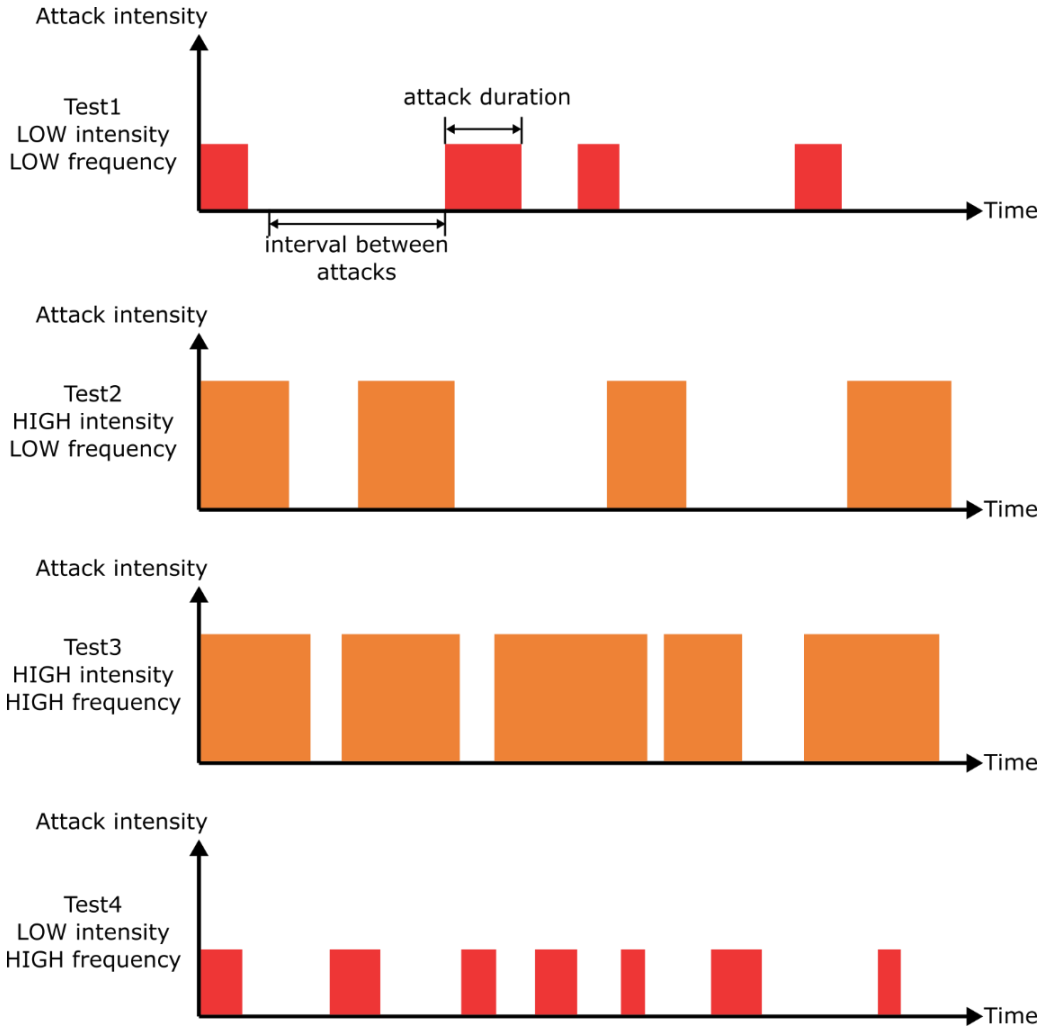

Figure 16 – Test scenarios visual representation

Also, for each scenario, the aggregation time for the requests generated by the Manager were changed. The aggregation time influences in the amount of gathered updates sent by the Managers, and correspondent number of logs sent to the interface. The tested aggregation times were 0,1 s, 1 s and 2,5 s, considering that for higher aggregation times the system starts to lose its sensibility to the changes in the platform (containers being restarted). As a result, the best performance was obtained with a relatively low aggregation time (less than 1 second to retain all requests and start issuing the ‘give content’ command to the Managers).

The battery of tests ran on the platform by executing each configuration of parameters 30 times, with a period of 5 minutes, adding to a total of 30 hours of continuous operation of the monitoring system. From the results it is possible to evaluate the overall performance of the HMS and how the availability of the platform was affected with the intrusions, as well as the bandwidth use within the different configurations. The parameters for the attacks were set in the following manner:

```
# low attack low frequency
'name'      : 'test_low_low',
'command'   : 'conf_antagonist',
'loss'      : 80,
'balance'   : 70,
'heavy'     : 25,
'frequency' : 1,
'duration'  : 5
```

```
# low attack high frequency
'name'      : 'test_low_high',
'command'   : 'conf_antagonist',
'loss'      : 80,
'balance'   : 70,
'heavy'     : 25,
'frequency' : 0.1,
'duration'  : 5
```

```
# heavy attack low frequency
'name'      : 'test_high_low',
'command'   : 'conf_antagonist',
'loss'      : 80,
'balance'   : 70,
'heavy'     : 90,
'frequency' : 1,
'duration'  : 20
```

```
# heavy attack high frequency
'name'      : 'test_high_high',
'command'   : 'conf_antagonist',
'loss'      : 80,
'balance'   : 70,
'heavy'     : 90,
'frequency' : 0.1,
'duration'  : 20
```

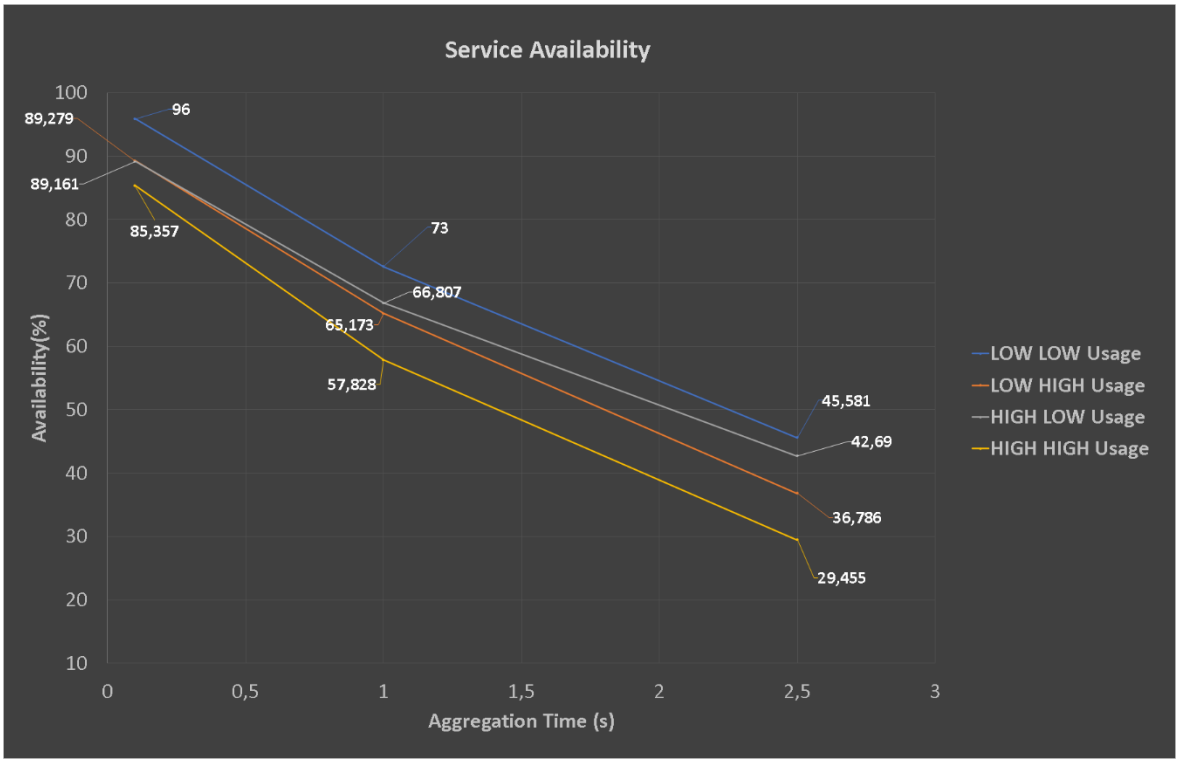

Figure 17 – Platform Availability results from the tests using the HMS and the Antagonist

The availability results show the percentage of time that all the active containers remain operational, and, therefore, describes the HMS efficiency in keeping the container running and the platform idle service working. From the results shown in Figure 16 it is possible to conclude that the HMS was able to actively correct the faults in the system keeping a reasonable availability for the implemented service.

In face of heavier attacks, the availability had a decrease, despite the monitoring and correction protocols implanted. However, the levels were not off the expected, as in some cases the disabling of simultaneous container will require a longer time to effectively reboot all faulty units, reducing the platform availability.

From the results it is also noticeable that for higher aggregation times, the availability was compromised, and had worse results when compared with the operation using a low aggregation time.

Comparing the results with the observed bandwidth (that is the number of update messages transmitted in each period) the conclusions were that for a higher aggregation time, the reduced amount of traffic had a positive impact in the system bandwidth. However, for high intensity attacks the overall gain in performance was reduced, indicating that this method did not scaled well with the intensity of the attacks.

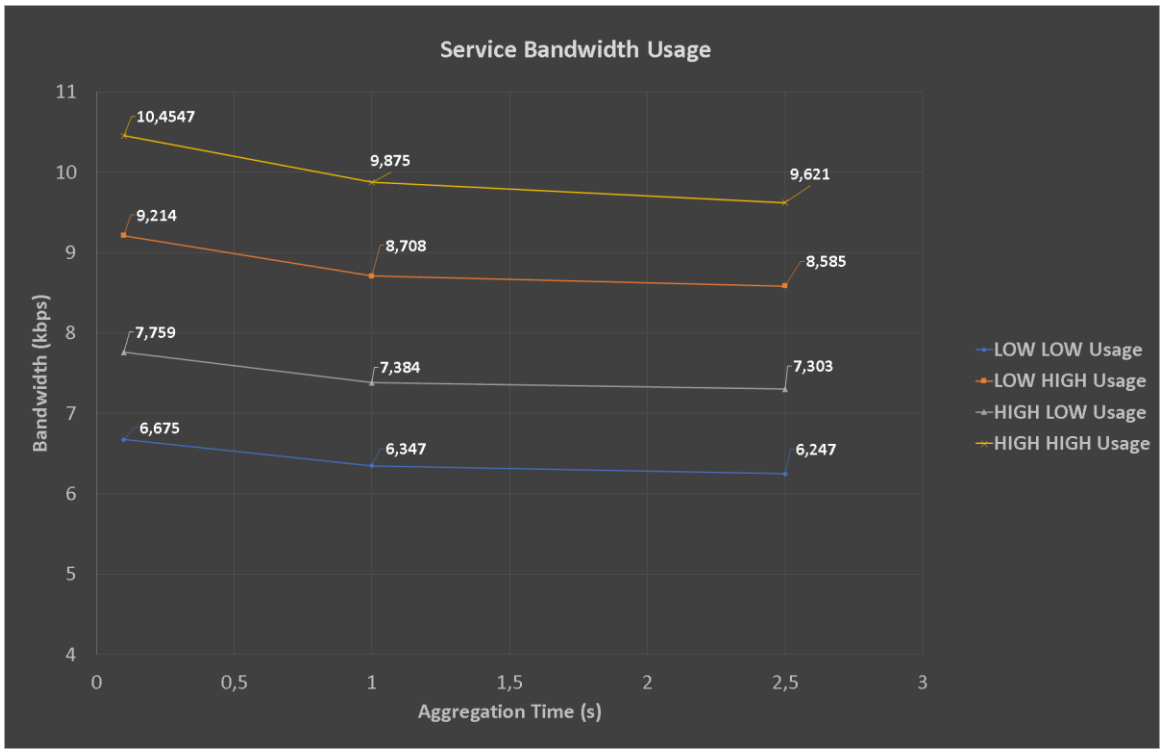

Figure 18 – Platform Bandwidth results from the tests using the HMS and the Antagonist

Another important observation is that for low aggregation time, a larger number of peaks in the transmission were detected, as a result of the higher sensibility of the system regarding simultaneous and recurrent updates (for example, a packet loss attack tends to produce lot of updates, as for percentage change a new update is raised). This had a negative influence on the communication channel, in spite the better availability observed in the previous results for low values of aggregation time.

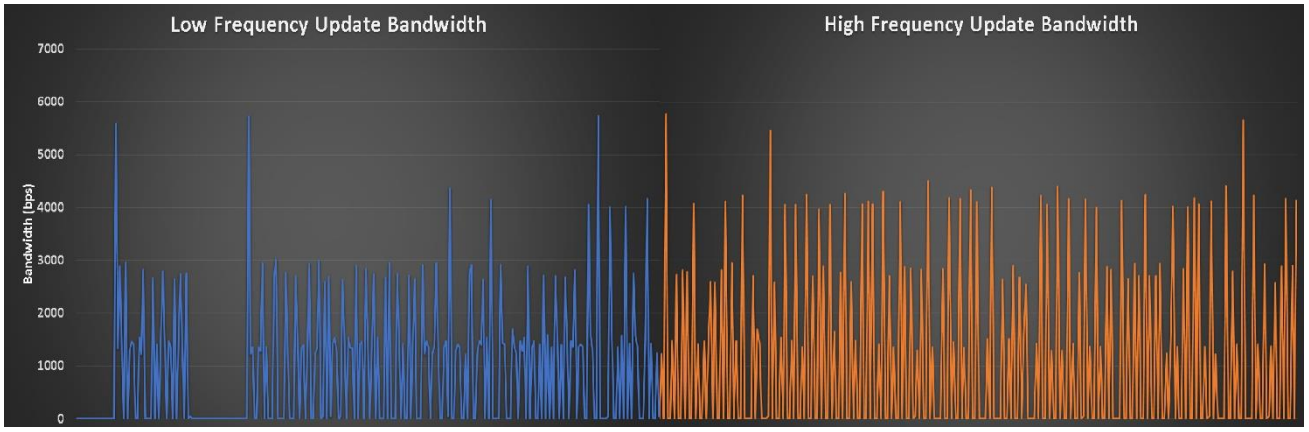

Figure 19 – Platform Bandwidth peaks in the tests

Conclusions

From the results of the tests performed it is possible to conclude that the Docker container Health monitoring system develop was able to detect the faults in the system caused by the Antagonist and effectively correct the issues. The measured system availability observed from the tests demonstrated the efficiency of the HMS as the program was able to dynamically restart the faulty containers keeping all the active units available for the platform.

The many python classes defined in the HMS were also thoroughly tested and many inconsistencies were removed, as the message queue implemented had a significant complexity, as well as the proper issuing of the synchronous and asynchronous messages exchanged within the modules deployed. With the proper testing and tunning, the queues were organized, and the communication worked as expected, including the issuing the commands from the REST interface to the Managers.

The overall system was then designed in a way that it is possible customize the monitoring parameters, as well as the aggregation time of the platform, making it a flexible tool to be used in a container-based environment where different types of attacks might occur. In addition, the design allows for a quick integration and ready to use interface for the control of the platform.

For future improvements, it is desirable to increase the system scalability, by adding a more reliable messaging system, as well as a consistent Controller interface to manage a higher number of updates and information. In addition, more utilities can be added in the designed tool in order to prover other useful information about the container status, such as the memory usage evaluation or even if any container has experienced a code bug and its software is not operational (like a watchdog functionality).