Appendix B: Technical documentation

INTRODUCTION

Welcome to the technical documentation guide of the cloud orchestration project source code. This document will guide you through how to maintain and develop the system source code along with technical aspects of important pieces of the system developed.

To start with, it is not necessary to have an environment like the one used in the implementation. In the case of arranging it by yourself you can see Appendix C: SETUP GUIDE for the complete steps to configure the master and worker nodes with its dependencies, so the steps in section 'SIMULATE THE ENVIRONMENT' will not be necessary.

In general, the first step to take is to clone the source code repository in the $HOME directory of the environment where the project will be developed, in the designated master node.

```
cd ~
git clone https://github.com/llopisga/cloud-orchestration.git
```

It is highly recommended to be cloned in this location, in case you have to clone it in another directory for personal reasons, you must edit the main path in the configuration file that will be seen later.

OVERVIEW

The technical documentation guide has the following content structure:

1. Simulate the environment
2. Source Code
    a. Model Orchestration
    b. Platform Orchestration
3. Provision Deploy
    a. Antidote Selfmedicate
    b. The Littlest Jupyterhub

1. SIMULATE THE ENVIRONMENT

In order to simulate the environment, to simulate the web form requests you must modify a base request that is included in the application source code.

```
vi
/home/$USER/cloud-orchestration/TM/thesisraul/public_html/requests/T
Mrequest-a4aaecdb66cf4b53883c3ff1496572ff.yaml
```

The request has the YAML format presented in the next piece of code, which must be manually modified to perform tests on different requests that may be generated in the real web form as previously mentioned.

```
---
requestor_id: raul.llopis
timestamp_start: 01-05-2020 22:49:01
firstname: Raul
surname: Llopis
request_UUID: a4aaecdb66cf4b53883c3ff1496572ff
requestor_email: raullg8@gmail.com
home_institute: EPSA
home_department: DCOM
teacher_role: ldteacher
operation_mode:
  create:
    Undefined: true
machine_size: small
lesson_type: it
lesson_topic: OperatingSystems
home_directories: mche
users:
  user1: Lucia
  user2: Anna
  user3: Sebastian
  user4: John
user_cooperation_mode:
  isolated:
additional_services:
  - web: "true"
persisted_items:
  what:
  - user_data
```

Next, we must go to the configuration file that is loaded in the Python files each time it is orchestrated, in this file we observe the following fields.

```
vi /home/$USER/cloud-orchestration/code/conf/config.yaml
  PROJECT_PATH: "/home/USERNAME/cloud-orchestration"
  TM_PATH: "/var/www/thesisraul/public_html/requests"
  SMTP_SERVER: "smtp.gmail.com"
  SMTP_PORT: 587
  NOTIFY_ADMIN: "raullg8@gmail.com"
```

There is only a variable that is not modified is the PROJECT_PATH, unless you clone the repository in another location The USERNAME variable is replaced by the user who executes the code automatically at runtime, so it must not be replaced manually. Regarding the TM_PATH variable, this is where the requests folder is located, in this case we must replace the default string with the following, changing the home user with yours:

```
TM_PATH:
"/home/raulloga/cloud-orchestration/TM/thesisraul/public_html/reques
ts"
```

For the other variables, the SMTP server is specified with its port, in this case the one provided by gmail has been used. In addition, the mail of an administrator who will be notified in case of errors in the execution of the code or on the system must be placed in the 'NOTIFY_ADMIN' value field.

Then, you must adapt the number of nodes you have in your system, in the case of working on the same machine you must go to repositories/inventory/nodes and leave only one node with the data of your system. The most important thing here is the IP address which should be changed by the IP loopback, typically 127.0.0.1, of your machine or the static IP.

```
node01:
  network:
    fqdn: node01.mche.edu.pl
    domain: mche.edu.pl
    ip: 127.0.0.1
```

Also, you must add this line in /etc/hosts:in order to set a local DNS for the nodes.

```
127.0.0.1    node01
```

Regarding the execution of the code, once these steps have been performed, you must change the current directory to the folder where the source code resides, from which you will execute the following command.

```
cd ~/cloud-orchestrator/code
python3 orchestrator.py <UUID>
```

The code 'orchestrator.py' is called with Python 3, passing as the argument the UUID of the service that we are going to process, this <UUID> must be changed by the one you want to process every time.

After generating the number of requests you desire, the following command should be entered from the directory code/, and the service orchestrator will start and complete if there is any service generated from the other model.

```
python3 service.py
```

2. SOURCE CODE

A. *MODEL ORCHESTRATION*

I. orchestrator.py

Next, I proceed to explain what are the most important points to understand about the source code. The main starting point is the orchestrator.py file, which contains the three models that are processed in each request. First, the variables that will be passed to the PROJECT_PATH and TM_PATH models are introduced, they are loaded from the configuration YAML file and the UUID is gathered from the first argument passed to the program.

Next, the variables are defined regarding the SMTP service authentication to send the errors that may happen during the execution, for this a try-catch is implemented in the main method which in throwing an exception will be received by the 'logging' module and this will send it by calling the errorEmail class in the file emailClient.py.

For each file, a main class has been defined in which the code functions are called in the desired order and the steps that the code takes can be logged. These logs are stored in *~/cloud-orchestration/logs/* with the service's UUID, the format of the logs is as follows.

```
2020-05-26 22:52:50,064 - pim - Generating PIM Model
```

Next, each of the main classes belonging to the modules pim.py, feasibles.py and psm.py are executed in the same order.

II. pim.py

In this model, dictionary type variables are defined to save the loaded data from YAML files. A constructor is defined in the conversionPIM class which includes the variables passed from the main code.

```
TM_DATA = dict()           # Handles TM request data in a dictionary
PIM_DATA = dict()          # Handles all PIM data
CONVERSION_DATA = dict()   # Handles conversion policy data

class conversionPIM:
    def __init__(self, UUID, TM_PATH, PROJECT_PATH):
        self.UUID = UUID
        self.TM_PATH = TM_PATH
        self.PROJECT_PATH = PROJECT_PATH
```

Next, the 'openTM' function is responsible for storing the most important data for program execution in variables. Then, the function 'openConversion' checks in the templates repository the templates that start with the pattern 'PIM_' and for each of them it is opened in reading mode and the elements that are later saved in a variable are parsed.

```
listFiles = [f for f in os.listdir(templatesPath) if
f.startswith(pattern)]
for pim in listFiles:
    with open(templatesPath + pim, 'r') as conversionfile:
        [CONVERSION_DATA.update({key: value}) for key, value in
yaml.full_load(conversionfile).items()]
```

Then, the 'generatePIM' function is in charge of checking if the mode of operation is 'delete' or different, in the case of being deleted, simply change the mode of operation to delete. In the creation or modification process, the size of the TM machine is first converted to values understandable by the system. Then, set the privileges based on the type of teacher who has requested the service. And finally, the network and firewall data is loaded. All of this data is dumped along with the TM data to a PIM file.

```
for key in CONVERSION_DATA:
    for x in CONVERSION_DATA[key]:
        if (x == lessontype):
            systemDict = dict(CONVERSION_DATA[key][x][machinesize])
        elif (x == cooperationmode):
            privilegesDict = dict(CONVERSION_DATA[key][x][teacher])
```

III. feasibles.py

In this model the PSMs and nodes feasible for the system are generated, this program follows the same role as 'pim.py'. Global variables are defined first and then the class with its constructor. In the main main method, the process logger is defined and the condition to carry out different actions in case the service is eliminated.

In general, the previously generated PIM model is first loaded with the most important data stored in the corresponding variables. Then, in the case of service creation mode, the PSM inventory is processed first.

It is consulted in the repository for each inventory that eimpice by the pattern 'PSM_' is loaded individually. PSMs are selected that have lessons based on the type and topic of the lesson that has been taken. All the PSMs that meet this condition are stored in an array that is then processed to gather the URL of the lesson and the requirements of the virtual machine of the specific platform.

```
for psm in listFiles:
    with open(inventoryPath + psm, 'r') as conversionfile:
        convert = yaml.full_load(conversionfile)
        for key, value in convert.items():
            INVENTORY_PSM_DATA.update({key: value})
            for k in value:
                for x in value[k]:
                    if (x == lessontype):
                        for y in value[k][x]:
                            if (y == lessontopic):
              # Selects feasibles PSM based on Lesson Type and Topic
                                feasibles.append(key)
```

Next, the repository of Feasible nodes is consulted in which, following the file pattern 'node_', the specifications of each node are checked. For each specification that meets the requirements, one is added to the counter, at the end the values are saved in a node and result dictionary. Nodes that have more than four conditions fulfilled are added as Feasible nodes to the final dictionary.

```
for node in listFiles:
    with open(inventoryPath + node, 'r') as conversionfile:
        convert = yaml.full_load(conversionfile)
        for key, value in convert.items():
            INVENTORY_NODES_DATA.update({key: value})
            if (value['specs']['vcpu'] > vcpu):
                counter += 1
            if (value['specs']['ram'] > ram):
                counter += 1
            if (value['specs']['os'] == opsys):
                counter += 1
            if (value['specs']['vt'] == vt):
                counter += 1
            if (value['specs']['storage'] > storage):
                counter += 1
            matching.update({key: counter})
feasiblesNodes = {}
for i in matching:
    if (matching[i] >= 4):
        feasiblesNodes.update({i: INVENTORY_NODES_DATA[i]})
```

Finally, the PIM data is dumped along with the PSM and Feasible Node data.

IV. psm.py

This model is where the application of policies on the data processed so far happens. In this model, it is necessary to notify the teacher of the status of the requested service, so the SMTP server data is loaded in variables. As in the previous models, the class and its constructor are defined, then the previously generated feasible PSM model is opened and the most important data is saved.

Next, the three main policies of the system are loaded by processing their content and saving it in dictionary variables for more efficient handling.

First, the screening policy is applied by checking whether the defined policies are followed. It is verified that the defined start date is greater than or equal to the current one by means of a function 'time_difference' which calculates the difference in days and compares it to the value specified in the policy. In the case of being indefinite, it is established that the start date will be one day from the moment it is executed.

```
for key in SCREENING_DATA:
    if (key == 'min-time-before-start'):
        try:
            if (duration != 'undefined'):
                if (self.time_difference(start_date) <
SCREENING_DATA[key]):
                    NOTIFY_USER.append(
                        'The start date must be greater than or
equal to ' + str(SCREENING_DATA[key]) + ' day.')
            else:
                tmp = datetime.now() + \
                    timedelta(days=SCREENING_DATA[key])
                start_date = tmp.strftime('%d-%m-%Y')
        except:
            pass
```

Then the minimums and maximums of the policy are checked. These fields are the number of users, the number of groups and the duration established in the request. IF this is indefinite, the minimum established in the policy is defined and in the same way we proceed with the maximums. At the end, the duration of the service is calculated using a random number of days between the minimum and maximum value.

If any policy has not been complied with, the string will be saved with the corresponding error in an array, the channel will be checked later if it is empty or contains any non-compliance.

Next, the selection policy is applied in which the feasible PSM is checked first. The policy specifies for each PSM an array of sizes and types of lessons they accept, these data are related to the properties of the particular PSM. For example, in this implementation Antidote is oriented to IT lessons and Jupyter can instead be oriented to any of the topics that have been specified.

```yaml
---
psm:
  antidote:
    size: [small, medium]
    type: [it]
  jupyter:
    size: [medium, large]
    type: [it, biology, maths, physics]
```

For this reason, the code includes for each PSM the size and type of lesson specified with those specified in the policy. These are stored in a dictionary with PSM key and value the number of matches. For all the PSM, it is chosen which is the PSM that obtains more coincidences with the policy and finally one of them is chosen.

```python
for psm in feasiblesPSM:
    for key in SELECTION_DATA:
        if (key == 'psm'):
            for rule in SELECTION_DATA[key]:
                if (rule == psm):
                    for x in SELECTION_DATA[key][rule]:
                        for i in SELECTION_DATA[key][rule][x]:
                            if i in PIMarray:
                                counter += 1
    matchPSM.update({psm: counter})
    counter = 0
bestPSM = max(matchPSM, key=matchPSM.get)
#print('Selected PSM is ' + bestPSM)
```

In the case of the nodes, a more dynamic policy pattern is followed, for each node in the generated feasibles, two dynamic parameters are checked using a 'nodesHealth' function: CPU Usage and RAM Available. For each node these two parameters are saved and returned as a result. Next, the node is saved to a dictionary as a key and the RAM value in MB available. At the end the node with the most available RAM memory is selected.

```
for node in feasiblesNodes:
    nodeDict = self.nodesHealth(node)
    if node in nodeDict:
        print("Node " + node + " is down.")
    else:
        matchNode.update({node: nodeDict['MemAvailable']})

bestNode = max(matchNode, key=matchNode.get)
#print('Selected Node is ' + bestNode)
```

Next, the validation of the data of the Screening policy is applied. As previously mentioned, if there is a warning in the 'NOTIFY_USER' array, they are added to the body of the email to inform the user and are sent to the teacher to warn them that they must modify the request and the program will end its execution. In the event that it complies with the policies and the operating method is not ‘delete’, a message will be sent notifying that the service has been successfully registered in the system and is waiting to be built. In the case of the service with operation mode delete, the corresponding mail will be sent, proceeding to generate the following preempt.

The sending of the mail is done through the ‘emailClient’ module, calling the senEmail class which acts as an interface where the SMTP server values and the data to send are passed.

Next, the Schedule policy is applied, which is based on the preempt priority theory. First, all the preempted files that currently exist in the system corresponding to the active services or marked to be deleted are loaded. The priority of each one is gathered and it is checked which one has the highest priority of the set in order to assign an even higher priority than this one. When I refer to high, it will be processed later than all the previous ones due to the priority queue.

```
pattern = 'preempt-'
listFiles = [f for f in os.listdir(
    schedulePath) if f.startswith(pattern)]
priorities = []
try:
    for p in listFiles:
        with open(schedulePath + p, 'r') as conversionfile:
            convert = yaml.full_load(conversionfile)
            for key, value in convert.items():
                for x in value:
                    if (x == 'priority'):
                        priorities.append(value[x])
    priority = max(priorities) + 1
except:
    priority = 0
```

Then the preempt corresponding to the current service is created, with the following data template.

```
preempt:
  arrival_time: DD-MM-YYYY HH:MM:SS
  destination: nodeXX
  priority: Y
  start_date: DD-MM-YYYY
  state: ready
  uuid: UUID
```

To end the process, the feasibles are updated to those selected in the main dictionary and the new YAML file is dumped with the final PSM data.

B. PLATFORM ORCHESTRATION

On the other hand, the code corresponding to the platform orchestrator is presented, which has a main class in charge of providing the logic for N Python programs corresponding to the N PSMs that are in the system. These are located in / code / PSM / and follow the template that we will see later.

I. service.py

To run this program in a production environment, a Crontab could be configured to run every day at the specified time. This program is designed to run once and the creation of all platforms are orchestrated in the same execution.

First, all variables are defined with the values loaded from the configuration file on the data of the SMTP server to which errors will be sent, if any, exactly as in the main orchestrator. Then the main directories where the preempts, the final PSM requests and the Python programs corresponding to the PSMs are defined. When the main function of the code starts to execute, it makes a call to the function 'checkDuration()' which is mainly in charge of creating preempt files for services that are about to end.

For each PSM in the production requests directory, the duration and the start date specified at the time are extracted. The calculations are made to check if the current time is the same as the start date plus the duration, in which case the same PSM file modifies the operation mode field and is set to delete. In addition, the preempt corresponding to this service is generated with the same logic as in a creation process to determine the corresponding priority.

Then, when all the active services have been verified, each preempt that exists at that moment is added to the priority queue. In the priority queue two values are added, the UUID of the service along with its priority.

```
pattern = 'preempt-'
listFiles = [f for f in os.listdir(
    PREEMPT_PATH) if f.startswith(pattern)]
q = PriorityQueue()
for p in listFiles:
    with open(PREEMPT_PATH + p, 'r') as conversionfile:
        convert = yaml.full_load(conversionfile)
        for key, value in convert.items():
            for x in value:
                if (x == 'start_date'):
                    start = datetime.strptime(value[x], '%d-%m-%Y')
                    now = datetime.now()
                    diff = start - now
                elif (x == 'priority'):
                    priority = value[x]
                elif (x == 'uuid'):
                    UUID = value[x]
    try:
        if (diff.days < 1):
            q.put((priority, UUID))
    except:
        pass
```

When all the preempts have been placed in the priority queue, the main program loop is executed, which while the queue is not empty, the priority queue is emptied based on the priority. For each one, open your PSM file and gather the values that are necessary to create the service, these are: Operation Mode, PSM Name, Virtualization Technology and Destination Node IP.

Next, a very important function is used for the logic of this program, it is presented below.

```
# Importing PSM Python file as a module
psm = importlib.import_module(PSM_NAME, package=None)
```

For the specific PSM its code is imported as a module, which facilitates the execution of the program because each one simply loads the module it needs. Otherwise, all modules should be loaded once and mapped, so this option is very handy for this solution.

Once the code is imported, depending on the mode of operation, its creation or destruction will be executed. For the creation two classes of the code are executed, these are code and deploy. When processed, in the end the preempt is destroyed and more PSM continues to be processed until it ends.

```
try:
    if (op_mode != 'delete'):
        logger.warning('Executing PSM code module of ' + PSM_NAME)
        process1 = psm.code(UUID, PROJECT_PATH,
                            PSM_NAME, VIRT_TECH, PSM_DATA)
        process1.main()
        logger.warning('Code modified and ready to be deployed')
        logger.warning(
            'Deploying with specific virtualization technology')
        process2 = psm.deploy(UUID, PROJECT_PATH, DEST_NODE,
PSM_DATA)
        process2.main()
        logger.warning('Finished deploying')
    else:
        deleteps = psm.destroy(UUID, PROJECT_PATH, DEST_NODE,
PSM_DATA)
        deleteps.main()
except Exception:
    logger.exception('Unhandled Exception')
```

Once the processing of all the PSMs is finished, it is ready to reorder the priority of all the preempts that remain to be executed in the system. First, the values of the current preempts are loaded into a dictionary variable which will contain the priority and the service's UUID.

```
try:
    # Re-arrange current preempt files if there are gaps between
priorities
    gaps = [ele for ele in range(
        max(priorities)+1) if ele not in priorities]
    # If there are gaps between the priorities numbers
    if (gaps):
        priorities.sort()
        arranged = {}
        tmp = list(range(len(priorities)))
        for x in priorities: # Rearrange the priorities
            new = tmp[0]
            del tmp[0]
            arranged.update({new: data[x]})
    [...] # Change priority to its file
except:
    pass
```

II. PSM template code

The same defined structure is followed for each PSM template. There must be three classes: code, deploy and destroy. Each one will be called in service.py depending on the mode of operation. There will be a variable to save the PSM data and another to save the data that is needed to configure the code and the service.

2.1. Code

The code generation class follows an almost identical structure for all PSMs. First, the function 'setVariables ()' is executed, which obtains the important data for the service and saves it in the dictionary with the string that will be replaced in the file and the value to be replaced.

```
ANTIDOTE_DATA.update(
    {'ANTIDOTE_VMNAME': self.PSM_DATA['request_UUID']})
ANTIDOTE_DATA.update(
    {'ANTIDOTE_MEMORY':
self.PSM_DATA['service'][self.PSM_NAME][self.VIRT_TECH]['memory']})
[ . . . ]
```

Then, the function 'checkAvailablePorts ()' is executed which is based on the range of ports specified in PIM with which it will be verified by the socket module that ports are free and not used to be assigned.

```
a_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
port_range = 1000
for x in range(startp, stopp):
    location = (dstServer, x)
    result_of_check = a_socket.connect_ex(location)
    if not result_of_check == 0:
        ANTIDOTE_DATA.update({'ANTIDOTE_PORT': x})
        break
```

Next, the 'copyPSMcode()' function is executed, which copies the source code found in ~/cloud-orchestration/PSM/repositories/<PSM>, if there is any lesson to incorporate into the PSM it will be downloaded using 'git clone' and all the source code will be copied to the following location ~/cloud-orchestration/production/deploy/<PSM>.

Finally, the function 'replaceVariables()' is executed which replaces the objective data of the PSM in the source code, for this the files to be modified are specified in an array and these are opened in writing mode to replace the necessary data.

```
ANTIDOTE_DIR = str(DEPLOY_PATH) + 'antidote-selfmedicate/'
arrayOfFiles = ['antidote-config.yml', 'selfmedicate.sh']
        for file in arrayOfFiles:
            fin = open(ANTIDOTE_DIR + file, "rt")
            data = fin.read()
            for i, j in ANTIDOTE_DATA.items():
                data = data.replace(i, str(j))
                fin.close()
                fin = open(ANTIDOTE_DIR + file, "wt")
                fin.write(data)
            fin.close()
```

2.2. Deploy

The service deployment class aims to execute the code previously generated on the destination server. To do this, the copyFile () function is called first, which does exactly what the function says. Once copied to the destination server under the $ HOME directory, the file that all PSMs must have to interact with the command line using the Bash language is executed.

```
command = "/bin/bash " + DEPLOY_PATH + "provision.sh " + DEPLOY_PATH
+ " >>" + DEPLOY_PATH + "logs/output.txt 2>&1"
stdout, stderr = Popen(['ssh', SSH_NODE, command]).communicate()
print(stdout)
```

This 'provision.sh' file is called from a function which will wait for it to finish. Once finished, it means that the service has been completed, then the teacher is notified that the service is ready by calling the sendEmail program with the sendComplete class which is specific for this case because it will attach the file with the generated data as an attachment by the PSM so that the teacher can access and have the instructions.

2.3. Delete

This class is designed to be implemented in future work, although it can be seen how the destruction of the service is planned. First, depending on the virtualization, the service is located to obtain the results of the students and save them on an SFTP server. Then, the service is stopped and destroyed in the virtualization platform, in addition to removing the directory with the data of the service on the destination server. Finally, the teacher is notified that the service has been removed and where they can consult the saved data.

3. PROVISION DEPLOY

This section explains how the generic provisioning script works for all existing and future PSMs which is called by the Python code of the specific PSM. This provision file is written in Bash to have maximum efficiency when executing the commands in the Linux terminal for bringing up the service. All provision files must have this code structure:

```
#!/bin/bash
set -e
trap 'last_command=$current_command; current_command=$BASH_COMMAND' DEBUG
trap 'echo "\"${last_command}\" command filed with exit code $?."' EXIT

[... SPECIFIC DEPLOY CODE ...]

cd $DEPLOYPATH
echo "The service with UUID $UUID, is ready at URL antidote-local:$PORT."
>> completed.txt

[... Specific instructions ...]
```

It begins by explicitly defining that the Bash language is used from its binary, then the program is marked to exit immediately if a command exits with a non-zero status, that is, when an error occurs in the execution of a command. Next, the specific code for the service deploy is specified along with the variables that gather the arguments passed to the program. Finally, change the directory to the parent where the deploy is made and a file 'completed.txt' is generated with the necessary information and instructions.

Regarding the specific PSM code in the provision file, the code used for both PSM implemented in the final project will be commented.

A. ANTIDOTE SELFMEDICATE

Next, the Antidote provisioning code is going to be discussed.

```
#!/bin/bash
set -e
trap 'last_command=$current_command; current_command=$BASH_COMMAND' DEBUG
trap 'echo "\"${last_command}\" command filed with exit code $?."' EXIT
```

```
# The main variables are assigned using the arguments of the code call
VMPROVIDER=$1
DEPLOYPATH=$2
UUID=$3
PORT=$4

# Change directory to the antidote-selfmedicate path
cd $DEPLOYPATH

# In this implementation, libvirt has not been introduced, although work
has been done so the code is left.

if [ "$VMPROVIDER" == "virtualbox" ]; then
    # If the provider is Virtualbox

    # Check if Virtualbox and vagrant are installed. Abort if not.
    command -v VBoxManage >/dev/null 2>&1 || { echo >&2 "I require
VBoxManage but it's not installed.  Aborting."; exit 1; }
    command -v vagrant >/dev/null 2>&1 || { echo >&2 "I require vagrant but
it's not installed.  Aborting."; exit 1; }

    # Change directory to antidote selfmedicate and install needed plugins.
    cd ./antidote-selfmedicate

    vagrant plugin install vagrant-vbguest
    vagrant plugin install vagrant-hostsupdater

elif [ "$VMPROVIDER" == "libvirt" ]; then
    # If the provider is Libvirt

    # Check if Libvirt and vagrant are installed. Abort if not.
    command -v libvirtd >/dev/null 2>&1 || { echo >&2 "I require libvirtd
but it's not installed.  Aborting."; exit 1; }
    command -v vagrant >/dev/null 2>&1 || { echo >&2 "I require vagrant but
it's not installed.  Aborting."; exit 1; }

    # Change directory to antidote selfmedicate and install needed plugins.
    cd ./antidote-selfmedicate

    vagrant plugin install vagrant-libvirt
    vagrant plugin install vagrant-hostsupdater

fi

# Be sure to be in Antidote dir and perform the deployment of the Antidote
code
cd ./antidote-selfmedicate
vagrant up

# Add designed port to firewall exception
```

```
sudo firewall-cmd --permanent --add-port=$PORT/tcp
sudo firewall-cmd --reload

# Change working directory to the parent deploy directory
cd $DEPLOYPATH

# Create the instructions for the teacher if the service deployment
succeeds
echo "The service with UUID $UUID, is ready at URL antidote-local:$PORT."
>> completed.txt
echo "Before accessing it, add this line:" >> completed.txt
echo "Linux: gedit /etc/hosts" >> completed.txt
echo "Windows: Open with Administrator privileges
c:\Windows\System32\Drivers\etc\hosts" >> completed.txt
echo "" >> completed.txt
echo "In any case add this line:" >> completed.txt
echo "$(eval "hostname -i")    antidote-local" >> completed.txt
```

Although the previous script is the most important, variables are also edited in the following. First, in *antidote-selfmedicate/antidote-config.yml*, the variables are replaced in the execution of the code class which will be loaded in the execution of 'vagrant up' in Vagrantfile for the configuration.

```
---
version: "0.6.0"
# Customizable configuration options are provided here for the vagrant up
command. This is used in conjunction with the vagrantfile.  Any changes in
this file require vagrant reload --provision
vm_config:
  vmname: ANTIDOTE_VMNAME
  memory: ANTIDOTE_MEMORY
  cores: ANTIDOTE_CPUS
  provider: ANTIDOTE_PROVIDER
  port: ANTIDOTE_PORT
  porttmp: ANTIDOTE_TMP
  ssh: ANTIDOTE_SSH
```

Finally, in *antidote-selfmedicate/selfmedicate.sh*, three variables responsible for configuring the platform within Virtualbox are also replaced in code class.

```
[...]

SELFMEDICATE_PORT=ANTIDOTE_PORT
CPUS=${CPUS:=ANTIDOTE_CPUS}
MEMORY=${MEMORY:=ANTIDOTE_MEMORY}
[...]
```

B. THE LITTLEST JUPYTERHUB

Finally, the Jupyter provisioning code is going to be discussed.

```
#!/bin/bash
# Steps:
#
http://tljh.jupyter.org/en/latest/contributing/dev-setup.html?highlight=dev

# exit when any command fails
set -e

trap 'last_command=$current_command; current_command=$BASH_COMMAND' DEBUG
trap 'echo "\"${last_command}\" command filed with exit code $?."' EXIT



# The main variables are assigned using the arguments of the code call
DEPLOYPATH=$1
LESSON=$2
TYPE=$3

# Change directory to the antidote-selfmedicate path
cd $DEPLOYPATH

# If the main Jupyter code is not present, then clone it
if [ ! -d "the-littlest-jupyterhub" ]
then
    git clone https://github.com/jupyterhub/the-littlest-jupyterhub.git
fi
# Jump to the Jupyter code directory
cd the-littlest-jupyterhub/

# Build the docker image and run it with variables
docker build -t tljh-systemd . -f integration-tests/Dockerfile
docker run --privileged --detach --name=JUPYTER_VMNAME --publish
JUPYTER_PORT:80 --mount type=bind,source=$(pwd),target=/srv/src
tljh-systemd
# After being created, first execute the installation and then set
resources limits
docker exec -t JUPYTER_VMNAME python3 /srv/src/bootstrap/bootstrap.py
--admin admin:JUPYTER_PASSWORD
docker exec -t JUPYTER_VMNAME sudo tljh-config set limits.memory
JUPYTER_MEMORYM
docker exec -t JUPYTER_VMNAME sudo tljh-config set limits.cpu JUPYTER_CPUS

# For N users requested, create them in the system
for i in {1..JUPYTER_USERS}
do
    docker exec -t JUPYTER_VMNAME sudo tljh-config add-item users.allowed
user$i
```

```
done

# Future work: All info in tljh/configurer.py
# Add collaboration modes
#if groups:
#   docker exec -t JUPYTER_VMNAME sudo tljh-config set
users.extra_user_groups.group1 user1

# Reload all applied config in TLJH
docker exec -t JUPYTER_VMNAME sudo tljh-config reload

# Clone the selected lesson in the shared folder under /home
docker exec -t JUPYTER_VMNAME git clone $LESSON /home/shared/$TYPE

# Add designed port to firewall exception
sudo firewall-cmd --permanent --add-port=JUPYTER_PORT/tcp
sudo firewall-cmd --reload

# Change working directory to the parent deploy directory
cd $DEPLOYPATH

# Create the instructions for the teacher if the service deployment
succeeds
echo "The service with UUID JUPYTER_VMNAME, is ready at URL $(eval
"hostname -i"):JUPYTER_PORT." >> completed.txt
echo "Administrator credentials:" >> completed.txt
echo "Username: admin" >> completed.txt
echo "Password: JUPYTER_PASSWORD" >> completed.txt
```