TÉCNICO LISBOA

AGISIT 19/20	ASSIGNMENT	Number:	1
Project Create a Lab		Issue Date:	December 30, 2019
Container orchestration - Docker Swarm		Due Date:	18 Dez 2019
Authors: 84727, 84750, 84766		Revision:	1.0

1 Introduction

In this project we are going to explore the use of Container technologies together with nginx[1] webservers. A Container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another[2]. This project will make use of Docker, a set of Platform-as-a-Service (PaaS) products that use Operating System-level virtualization. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Containers isolate software from its environment and ensure that it works uniformly despite differences, for instance, between development and staging. The experiments in this project will create a load balanced nginx web service in a cluster. The Docker Swarm Tool will be used to provide native clustering functionality for the Docker containers, turning the group of Docker engines into a single virtual Docker service engine.

To create the virtual environment (see Figure 1), which will include one virtual machine with Ansible[3] on it to act as the Cluster Manager, and two additional virtual machines to act as Worker nodes of the Cluster, we'll use Vagrant[4]. Vagrant allows to create Infrastructure as Service (IaaS) clouds on a desktop/laptop machine. Vagrant acts as wrapper around virtualization software, speeding up many of the tasks associated with setting up, tearing down, and sharing Virtual Machines and/or Containers. Vagrant is an invaluable tool for managing local sandboxes because it creates disposable multi-node environments, as self contained systems, used for development, testing and gaining experience with versions of Operating Systems, Applications, Tools or configurations, and then throw them away. These sandbox systems can also be used to reproduce ticket related issues, performance problems, or test client-server interactions. Vagrant uses a definition file (named Vagrantfile) to define compute nodes (one or more Virtual Machines or Containers), networking, and storage for an environment that can be created and run with a local virtualization tool. Because the definition is contained in an externalized file, as with any infrastructure defined "as code", a Vagrant environment is reproducible. The entire environment can be torn down (even destroyed), and then rebuilt with a single command (for that Vagrantfile).

Ansible is used to automate many IT needs such as Cloud provisioning, Configuration management, Application deployment, Intra-service Orchestration, etc. Ansible models an IT infrastructure by describing how all systems inter-relate. Ansible is "agent-less", meaning that it does not uses agents in the infrastructure nodes, making it easy to deploy – and most importantly, it uses a very simple language called YAML (Yet Another Markup Language), in the form of Ansible Playbooks. Ansible Playbooks are special text files describing the automation jobs in plain English. Ansible works by connecting

[1]https://www.nginx.com/
[2]https://www.docker.com/resources/what-container
[3]https://www.ansible.com/
[4]https://www.vagrantup.com/

to the Infrastructure Nodes and pushing out small programs, called “Ansible Modules” to them. These programs are written to be resource models of the desired state of the system. Ansible executes those modules (over SSH by default), and automatically removes them when finished. Ansible is typically installed on a “management machine”, which in this case, will be the Swarm Manager node. In the context of this project, we'll only use it to share configuration files among the swarm nodes.

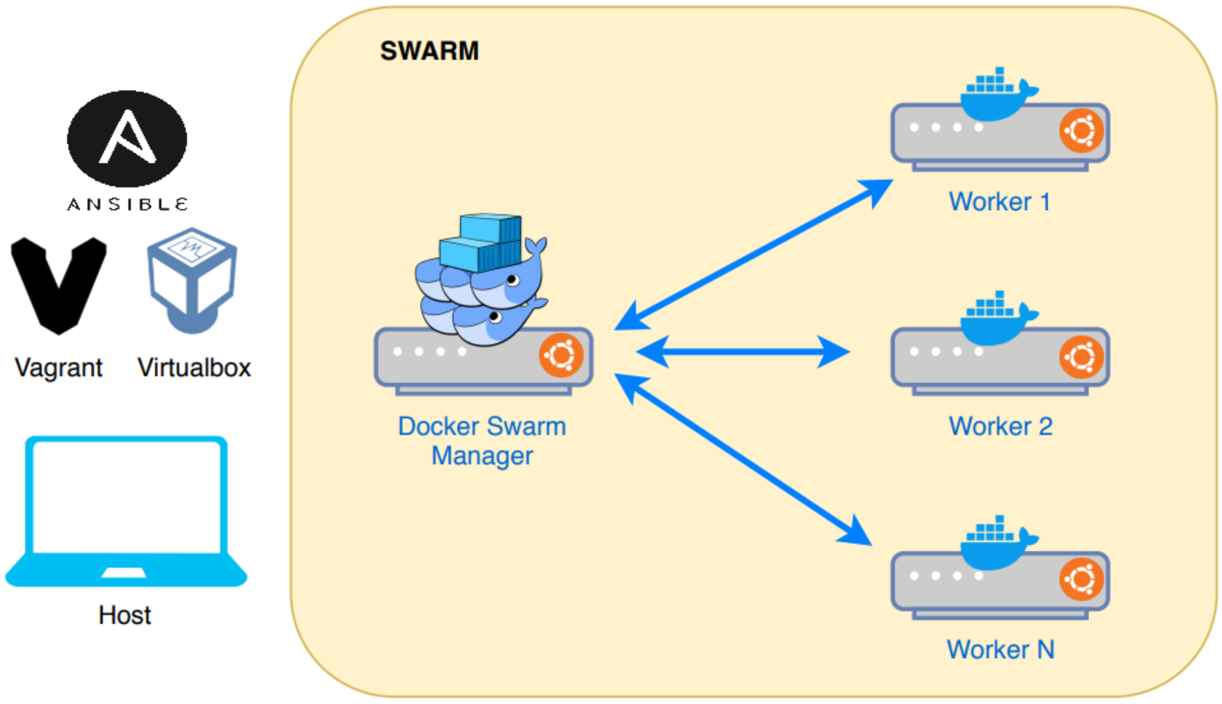

Figure 1: The Docker Swarm Environment

2 Preliminary Notes

A video demonstrating the execution of this lab in a Linux environment is available on the following link: `https://youtu.be/477DzQ_tgbU` .

From this point on, we'll assume that you already have Vagrant and VirtualBox installed/configured on your system.

Before proceeding you should verify if you have a “clean” environment, i.e., no Virtual Machine “instances” running (using precious resources in your system), or inconsistent instances in Vagrant and Virtualbox. For that purpose run the `vagrant global-status` command and observe the results (as in the following example):

```
:~$ vagrant global-status
id      name    provider   state    directory
--------------------------------------------------------------------
28fb48a mininet virtualbox poweroff /Users/x/Projects/mininet
f0ccec2 web1    virtualbox running  /Users/x/Projects/multinode
f09c279 web2    virtualbox running  /Users/x/Projects/multinode
```

In the above example, you can observe that there are three Virtual Machines instantiated, being the first “mininet”, which is powered off, but two “web” servers still running. It is advisable to halt VMs if running, and then clean and destroy VMs not related with this project.

If you use Windows System, make sure you will use Git-Bash terminal during the execution of this experiment.

Note: Avoid copying text strings from the command line examples or configurations in this document, as pasting them into your system or files may introduce/modify some characters, leading to errors or inadequate results.

3 Setting Up the Vagrant/Swarm Environment

To start the experiments, download the *.zip* of the content present in `https://github.com/santos-samuel/dockerswarm-vagrant` and uncompress it.

Verify that you will end up with a folder named dockerswarm-vagrant-master with the following file structure:

```
.
|-- files
|   |-- ansible.cfg
|   |-- config_files
|   |   '-- daemon.json
|   |-- inventory.ini
|   |-- nginx_example
|   |   |-- custom_nginx_image.tar
|   |   '-- docker-compose.yml
|   |-- ssh-addkey.yml
|   '-- update_docker_config.yml
|-- provision_manager.sh
|-- provision_worker.sh
'-- Vagrantfile
```

Start by inspecting the Vagrantfile already present in the downloaded content. If you want, you can change the ”numworkers” variable to increase the number of nodes that we'll be part of the swarm cluster. In this report we'll assume that there are only 2 worker nodes. Furthermore, verify if the IPs given to the machines (lines 15 and 19) are not in conflict with addresses that you may already be using. If you change the manager IP, make sure that you update the `config_files/daemon.json` file accordingly and that whenever you see us doing commands with the manager IP you replace with your manager IP.

We will use for this environment the box named "ubuntu/trusty64", which you should already have in your system from previous AGISIT labs, and so there is no need to download it.

Note that we will create a Swarm Manager node, called manager, and two worker nodes, called worker1 and worker2. By inspecting the `Vagrantfile`, you can see that the Swarm Manager node is being declared in the block starting in line 54. Note that, for the manager node, we are uploading and synchronizing the "files" folder, which comes with the downloaded zip; we are uploading the `provision_manager.sh` file into the manager node, which will make it execute as soon as the machine is booted; and that we are creating the swarm "on-the-fly" and saving the token linked with the swarm so that the worker machines can later join it. The worker nodes are being declared in the block starting in line 80. Note that, for the worker nodes, we are uploading the `provision_worker.sh` file, which will make it execute as soon as the machines are booted, and make them join the swarm created by the manager node, by executing the instruction in line 93.

The `provision_manager.sh` file will install Docker and Ansible on the manager node. Furthermore, it will add entries to the `/etc/hosts` file of the Swarm Manager Node, which will allow us to have name resolution to be used by Ansible for the hosts defined in the inventory `inventory.ini` for this infrastructure.

3.1 Launching the Vagrant/Swarm Environment

Before launching the Vagrant environment, run `vagrant status` to see that there are three virtual machines defined, i.e., a Swarm Manager node, and two Worker nodes.

```
~/dockerswarm-vagrant-master $ vagrant status
Current machine states:

manager                   not created (virtualbox)
worker1                   not created (virtualbox)
worker2                   not created (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```

The listed machines are all in the "not created" state, so let's fire them up by running the following command:

```
~/dockerswarm-vagrant-master $ AUTO_START_SWARM=true vagrant up
```

This command will launch the machines into the Vagrant environment; execute the respective provision post-install scripts, which will install Docker in all the VMs and Ansible into the Swarm Manager node; and configure the Swarm Environment, as explained in the previous section.

After it's completed, we can use *vagrant status* to confirm that the machines were launched. Also note that the token file was created inside the dockerswarm-vagrant-master folder, which was used for the worker nodes to join the swarm.

As soon as the machines are created and joined the Swarm we can login to the Manager Node using the following command:

```
~/dockerswarm-vagrant-master $ vagrant ssh manager
```

3.2 Ensuring SSH Connectivity Between Nodes

Now that you verified that the environment was successfully launched you are in conditions to proceed. When using an Automation Engine such as Ansible, the connectivity to remote machines from the Management Node is done via ssh (Secure Shell). One issue that we need to deal with, is that, if the Management Node has not yet connected to a machine via SSH, you will be prompted to verify the ssh authenticity of the remote node.

There are several ways of dealing with this issue:

1. Populate known hosts file: Use ssh-keyscan command to populate the file at the Management Node, with keys of the machines from the environment; - this option still needs password authentication.

2. Establishing a SSH Trust. - secure and appropriate for automated processes.

3. Turn off the authentication: via a configuration in the ssh config file; - insecure and dangerous!

It is obvious that option 3 is not an option!

Let's start by running `ssh-keyscan` for the Swarm Manager machine, manager, as well as for worker1 and worker2, and then pipe the output into `.ssh/known_hosts` file:

```
vagrant@manager:~/$ ssh-keyscan manager worker1 worker2 >> ~/.ssh/known_hosts
# manager SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.13
# manager SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.13
# worker1 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.13
# worker1 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.13
# worker2 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.13
# worker2 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.13
```

You can now verify that the known hosts file in the Management Node contains a bunch of keys from the remote machines:

```
vagrant@manager:~$ cat ~/.ssh/known_hosts
```

The hosts are now “known” at the Manager Node, and Ansible can remotely execute commands, however, as there is no SSH trust established, it will always ask for a password when executing remote commands.

To establish a “password less” but secure access to the infrastructure nodes – for a “continuous integration system” – in order to deploy automated application updates on a frequent basis, we need to establish a SSH trust, between the Manager Node and the other nodes of the infrastructure. We will do that using Ansible itself via a Playbook.

You can verify that on the Manager Node, in the .ssh directory (of the home user), there is no public RSA key:

```
vagrant@manager:~$ ls -la .ssh/
total 12
drwx------ 2 vagrant vagrant 4096 Dec 11 14:21 .
drwxr-xr-x 6 vagrant vagrant 4096 Dec 11 14:14 ..
-rw------- 1 vagrant vagrant  466 Dec 11 12:39 authorized_keys
-rw-rw-r-- 1 vagrant vagrant 1674 Dec 11 14:21 known_hosts
```

Let’s then create one key by running `ssh-keygen -t`, and specify the type of key we want to create, which will be RSA, and then tell how long a key we want with parameter -b. After that, verify that the keys have been created, by listing the contents of .ssh.

Please hit ENTER to the prompts, and do not enter a password.

```
vagrant@manager:~$ ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter file in which to save the key (/home/vagrant/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/vagrant/.ssh/id_rsa.
Your public key has been saved in /home/vagrant/.ssh/id_rsa.pub.
The key fingerprint is:
15:5c:45:63:a9:fd:61:fa:57:25:10:34:1a:8a:d9:fb vagrant@manager
The key's randomart image is:
+--[ RSA 2048]----+
|       + ..+.o.. |
|      o o o  +   |
|         o  . oo.|
|        S     ooo|
|         .   .  o|
|          E   . .|
|               ..|
|                .|
+-----------------+
```

You can now verify that in the .ssh directory (of the home user), there is a public RSA key:

```
vagrant@manager:~$ ls -la .ssh/
total 20
drwx------ 2 vagrant vagrant 4096 Dec 11 14:10 .
drwxr-xr-x 5 vagrant vagrant 4096 Dec 11 12:40 ..
-rw------- 1 vagrant vagrant  466 Dec 11 12:39 authorized_keys
-rw------- 1 vagrant vagrant 1675 Dec 11 14:10 id_rsa
-rw-r--r-- 1 vagrant vagrant  397 Dec 11 14:10 id_rsa.pub
-rw-rw-r-- 1 vagrant vagrant 1674 Dec 11 14:05 known_hosts
```

Now that we have fulfilled the requirements of generating a local RSA public key for the Vagrant user, let's deploy it to our remote client machines. We do this by running the command `ansible-playbook`, and then the Playbook name, in this case `sshaddkey.yml`.

Looking at the `sshaddkey.yml`, inside the *files* folder, you can see that we have only the hosts play, and the task to deploy an authorized ssh key onto a remote machine, which will allow Ansible to connect without using a password.

We also need to add the `--ask-pass` option, since we do not have "password less" login configured yet in those remote machines. The password to use is *vagrant*.

```
vagrant@manager:~/files$ ansible-playbook ssh-addkey.yml --ask-pass
SSH password:
[WARNING]: Found both group and host with same name: manager


PLAY [all] **********************************************************************
```

```
TASK [install ssh key] *********************************************************
changed: [worker1]
changed: [worker2]
changed: [manager]

PLAY RECAP *********************************************************************
manager                    : ok=1    changed=1    unreachable=0    failed=0
↪  skipped=0    rescued=0    ignored=0
worker1                    : ok=1    changed=1    unreachable=0    failed=0
↪  skipped=0    rescued=0    ignored=0
worker2                    : ok=1    changed=1    unreachable=0    failed=0
↪  skipped=0    rescued=0    ignored=0
```

The Playbook response should say that it changed three nodes (manager, worker1 and worker2) to deploy the authorized key from the Management Node.

Let's now run an ad-hoc Ansible command, targeting all nodes, saying that we want to use the ping module to verify that everything works as expected and without a password.

```
vagrant@manager:~/files$ ansible all -m ping
[WARNING]: Found both group and host with same name: manager

worker2 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
manager | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
worker1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
```

4 Playing Around with Docker Swarm

Docker Swarm is a clustering solution, turning multiple physical (or virtual) hosts into a cluster, which practically behaves as a single Docker host. The Docker Swarm additionally provides tools and mechanisms to easily scale the Containers in the swarm and create managed services with automatic load balancing to the exposed ports.

In this project, we will use a Docker Image called `nginx` available in Docker Hub (https://hub.docker.com/_/nginx), a simple webserver that only displays a default nginx webpage. Then, we'll see how to replicate the containers across the swarm and update the image being used with the rolling update feature.

4.1 Creating a Nginx Web Service in the Swarm

Firstly, we can check that the swarm is structured as intended with the command *docker node ls*:

```
vagrant@manager:~/files$ docker node ls
ID                            HOSTNAME STATUS  AVAILABILITY  MANAGER STATUS
cn4jfeziis8k1ovjy045cntan     worker1  Ready   Active
q8k1443vd3tvqkcg9i3u2pwkr     worker2  Ready   Active
rlkh43vux05p9fblq137ykqrr *   manager  Ready   Active        Leader
```

The cluster right now isn't doing anything, because we still haven't launched any service, as we can verify by running the command below, so let's launch one.

```
vagrant@manager:~/files$ docker service ls
ID       NAME        MODE          REPLICAS      IMAGE       PORTS
```

To start our service we'll use *docker stack deploy*[5]. Docker Stack (see Figure 2) sits at a higher level than Docker containers and helps to manage the orchestration of multiple containers across several machines. Docker Stack is run across a Docker Swarm.

Stacks allow for multiple services, which are containers distributed across a swarm, to be deployed and grouped logically. The services run in a Stack can be configured to run several replicas, which are clones of the underlying container.

[5]https://docs.docker.com/engine/reference/commandline/stack_deploy/

The deployment of many inter-communicating microservices is where Docker Stack comes in. The stack is configured using a docker-compose file. This is a YAML file written in a domain-specific language to specify Docker services, containers and networks. Given a compose file, it is as simple as one command to deploy the stack across an entire swarm of Docker nodes.

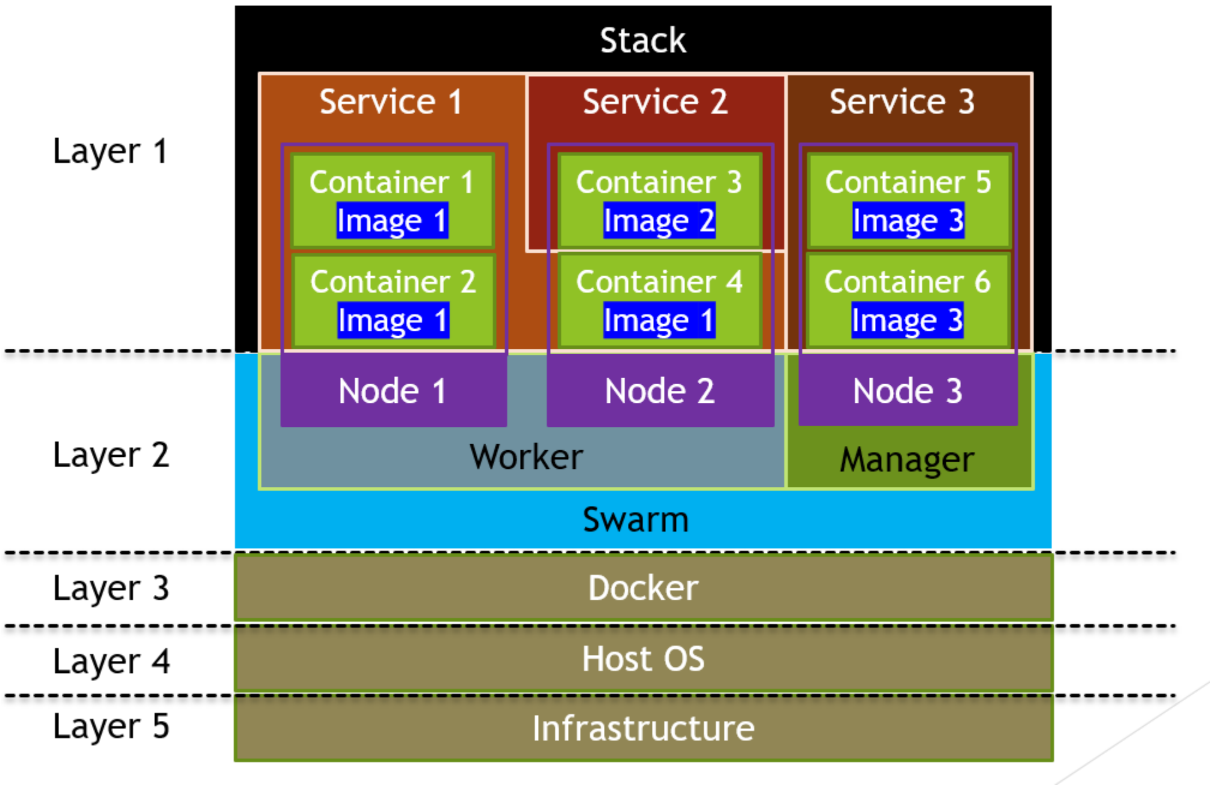

Figure 2: Docker stack architecture

Inside the *nginx_example* directory, there is a file named *docker-compose.yml*. This file is responsible for deploying a service named *webserver*. This webserver will create one replica of a container that uses a nginx image and will expose port 80.

To launch the service, run the following command:

```
vagrant@manager:~/files/nginx_example$ docker stack deploy -c docker-compose.yml
↪  nginx_sample
Ignoring unsupported options: restart

Creating network nginx_sample_default
Creating service nginx_sample_webserver
```

After that, we can check the list of the services deployed by running the command below. The service that we just deployed must appear in that list:

```
vagrant@manager:~/files/nginx_example$ docker service ls
ID                  NAME                     MODE                REPLICAS
↪   IMAGE               PORTS
nzfvtqtwp22b        nginx_sample_webserver   replicated          1/1
↪   nginx:alpine        *:80->80/tcp
```

Furthermore, we can use the *docker service ps* command to list the tasks of one or more services.

```
vagrant@manager:~/files/nginx_example$ docker service ps nginx_sample_webserver
ID                  NAME                       IMAGE               NODE
↪   DESIRED STATE       CURRENT STATE               ERROR               PORTS
jwmg8qda0ad2        nginx_sample_webserver.1   nginx:alpine        manager
↪   Running             Running about a minute ago
```

From the output of the command, we can see that the container was deployed in the manager node. Open a window on your browser and go to: `http://192.168.56.2/`. You should see the default web page of the nginx image that we used for this service. You can also curl the address and see the output below:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
```

```
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Note that the container could have been deployed in any node of the swarm. Furthermore, note that, although the container was deployed in the manager node (in our case), if we try to curl, or search in the browser, the IPs of the workers, which are not running any container, we also get the webpage of the service that is running in the master node.

Right now, the service is running but only one container was deployed and we have 2 spare nodes. So, we are going to scale up the service, for example to 4 additional web servers, with the command *docker service scale nginx_sample_webserver=4*.

```
vagrant@manager:~/files/nginx_example$ docker service scale nginx_sample_webserver=4
nginx_sample_webserver scaled to 4
vagrant@manager:~/files/nginx_example$ docker service ls
ID                  NAME                     MODE                REPLICAS
↪   IMAGE               PORTS
nzfvtqtwp22b        nginx_sample_webserver   replicated          4/4
↪   nginx:alpine        *:80->80/tcp
```

We can see that we now have 4 containers running in the swarm. If you run `docker service ps nginx_sample` you will also see that they were distributed among the swarm nodes.

4.2 Create and Update Nginx websersers

As the default web page of this image is simple and boring, let's update it for something cooler.

In the *nginx_example* folder we have a file named *custom_nginx_image.tar*. This is a docker image created by us from the previous nginx image.

Let's start by loading our custom image to the manager node by doing:

```
vagrant@manager:~/files/nginx_example$ docker load < custom_nginx_image.tar
1e3287c226eb: Loading layer [==================================================>]
↪   10.24kB/10.24kB
Loaded image: nginx_moo:latest
```

AGISIT 19/20	**ASSIGNMENT**	**Number:**	1
Project Create a Lab		**Issue Date:**	December 30, 2019
Container orchestration - Docker Swarm		**Due Date:**	18 Dez 2019
Authors: 84727, 84750, 84766		**Revision:**	1.0

You should see the new image (nginx_moo) present in the list of images known by the manager node:

```
vagrant@manager:~/files/nginx_example$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED              SIZE
nginx_moo           latest              053c62e7c2c6        About a minute ago
↪  21.5MB
nginx               alpine              a624d888d69f        3 weeks ago
↪  21.5MB
```

We already have the image that we want to use in our rolling update, but before updating the running containers, we first have to share this image among the nodes of the swarm, because currently, only the manager node has the image.

To share the image among the nodes we'll use, for simplicity purposes, a local registry. A better way of sharing the image would be to upload it in Docker Hub, for instance.

While it's highly recommended to secure a registry using a TLS certificate issued by a known CA, we chose to use a registry over an unencrypted HTTP connection because we know we are in an isolated testing environment.

To configure Docker to entirely disregard security for the registry we'll create to share the image, we have to edit the */etc/docker/daemon.json* file in all the nodes. We'll use an *ansible-playbook* to copy/replace our file to the nodes of the swarm. There is a file named *update_docker_config.yml* inside the *files* folder that does exactly that. Because we'll change docker configurations, there is also a handler that will restart the docker service in all the swarm's nodes.

Run the playbook with the following command:

```
vagrant@manager:~/files$ ansible-playbook update_docker_config.yml
[WARNING]: Found both group and host with same name: manager



PLAY [all]
↪  ***********************************************************************************************

TASK [Update docker config]
↪  ***********************************************************************************************
changed: [worker1]
changed: [manager]
changed: [worker2]

RUNNING HANDLER [restart docker]
↪  ***********************************************************************************************
changed: [worker1]
```

```
changed: [worker2]
changed: [manager]

PLAY RECAP
↪ ****************************************************************************************
manager                    : ok=2    changed=2    unreachable=0    failed=0
↪ skipped=0    rescued=0    ignored=0
worker1                    : ok=2    changed=2    unreachable=0    failed=0
↪ skipped=0    rescued=0    ignored=0
worker2                    : ok=2    changed=2    unreachable=0    failed=0
↪ skipped=0    rescued=0    ignored=0
```

Because we restarted the docker service in all the nodes, the service was temporarily unavailable, as you can see by running `docker service ps nginx_sample_webserver`. To avoid putting the whole system down at the same time, it would also be possible to do the restart in a serialized way, with some delay between, using ansible features. By inspecting the output of the command, you can see that all the nodes failed because of the error `"task: non-zero exit (0): No s..."` but then recovered and are running again.

```
vagrant@manager:~/files$ docker service ps nginx_sample_webserver
ID                  NAME                          IMAGE               NODE
↪ DESIRED STATE       CURRENT STATE                ERROR
↪ PORTS
xevt5ewf4oeh        nginx_sample_webserver.1      nginx:alpine        worker1
↪ Running             Running about a minute ago
x6i1ykrv0tko         \_ nginx_sample_webserver.1  nginx:alpine        worker1
↪ Shutdown            Failed about a minute ago    "task: non-zero exit (0): No
↪ s..."
iqil2ahfcbd4        nginx_sample_webserver.2      nginx:alpine        manager
↪ Running             Running about a minute ago
8e54p7rzuzn7         \_ nginx_sample_webserver.2  nginx:alpine        manager
↪ Shutdown            Failed about a minute ago    "task: non-zero exit (0): No
↪ s..."
p79g3m70pai5        nginx_sample_webserver.3      nginx:alpine        manager
↪ Running             Running about a minute ago
guzn813yl3xf         \_ nginx_sample_webserver.3  nginx:alpine        manager
↪ Shutdown            Failed about a minute ago    "task: non-zero exit (0): No
↪ s..."
l0o6v6t8uguu        nginx_sample_webserver.4      nginx:alpine        worker2
↪ Running             Running about a minute ago
skhytsbi8uz8         \_ nginx_sample_webserver.4  nginx:alpine        worker2
↪ Shutdown            Failed about a minute ago    "task: non-zero exit (0): No
↪ s..."
```

Now that all the nodes are disregarding security for the registry, we'll create a local registry by running the following command:

```
vagrant@manager:~/files$ docker run -d -p 5000:5000 --restart=always --name registry
↪  registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry
c87736221ed0: Pull complete
1cc8e0bb44df: Pull complete
54d33bcb37f5: Pull complete
e8afc091c171: Pull complete
b4541f6d3db6: Pull complete
Digest: sha256:8004747f1e8cd820a148fb7499d71a76d45ff66bac6a29129bfdbfdc0154d146
Status: Downloaded newer image for registry:2
d0dad1a21d1ce0202adbd107a791080ccd4d817eef470252970fb57ac7ae314b
```

After pressing 'Enter', Docker downloads the 'registry' image from Docker Hub online repository, and creates a container running it, providing an easy way to store and distribute Docker images through the swarm.

At this point, if you run `docker ps` in the terminal, you'll see that a new container was instantiated in the manager node that is running the registry image.

Let's now tag our custom image to create a repository with the full registry location and push the new repository to its home location by running the following commands:

```
vagrant@manager:~/files$ docker tag nginx_moo 192.168.56.2:5000/nginx_moo

vagrant@manager:~/files$ docker push 192.168.56.2:5000/nginx_moo
The push refers to a repository [192.168.56.2:5000/nginx_moo]
864cd660d6df: Pushed
f4cef7054e83: Pushed
77cae8ab23bf: Pushed
latest: digest:
↪  sha256:13778581ee762dd05c2e252bfa429b0777f166d78d9029c54fc92e90ccb54062 size: 946
```

Now that we published the image we want to use in the update in the local registry, the worker nodes will be able to pull it from there, even though they don't have the image in their docker images list.

To do the rolling update run the following command:

```
vagrant@manager:~/files$ docker service update --image 192.168.56.2:5000/nginx_moo
↪  --mount-add
↪  type=bind,source=/etc/hostname,destination=/usr/share/nginx/html/host-hostname.txt,
↪  readonly=true nginx_sample_webserver
nginx_sample_webserver
Since --detach=false was not specified, tasks will be updated in the background.
In a future release, --detach=false will become the default.
```

The `--mount-add` flag in the command is simply copying the `/etc/hostname` file to each container that is updated.

After a while, if you go to your browser again and search `http://192.168.56.2/` you should see a new page reflecting the usage of this new image.

Also notice that, when refreshing the page several times using shift+f5, you see that different nodes are attending the requests. This is the load balancing of Docker Swarm in action, which uses the Raft Consensus Algorithm[6] to distribute the load across the swarm containers.

Note: shift+f5 is used to refresh ignoring browser cache. Most browsers have it, for example chromium doesn't, so you may need to open the address in an incognito window, and to refresh the page close the incognito window and open a new incognito window again. We believe that browser's cache will always give you the same page if you don't do this process.

4.3 Finishing the experiment

In order to finish this lab experiment, first we are going to remove the service with the command:

```
vagrant@manager:~$ docker service rm nginx_sample_webserver
nginx_sample_webserver
```

Then, we will check that it was indeed removed:

```
vagrant@manager:~$ docker service inspect nginx_sample_webserver
[]
Status: Error: no such service: nginx_sample_webserver, Code: 1
```

Now that the service is no longer running, we can logout from the manager node and stop the machines with *vagrant halt*. In order to confirm that they are all stopped, run *vagrant global-status* and confirm that the status is "powered off".

If you won't use them anymore you can destroy these machines.

[6] https://raft.github.io