Instituto Superior Técnico

Advanced Network Security and Architectures

Project Report:

Kubernetes & Docker Swarm

Félix Saraiva – 98752

2022

Table of Contents:

Table of Figures:

1. Kubernetes networking

The aim of this section is to explore Kubernetes and Kubernetes networking. The laboratory topics analysed are:

I. Kubernetes control plane
II. Deploying a single pod
III. Deploying a single pod through a YAML manifest
IV. Deploying a ReplicaSet
V. Deploying a ClusterIP service
VI. Kubernetes DNS
VII. Deploying a NodePort service

Network Topology:

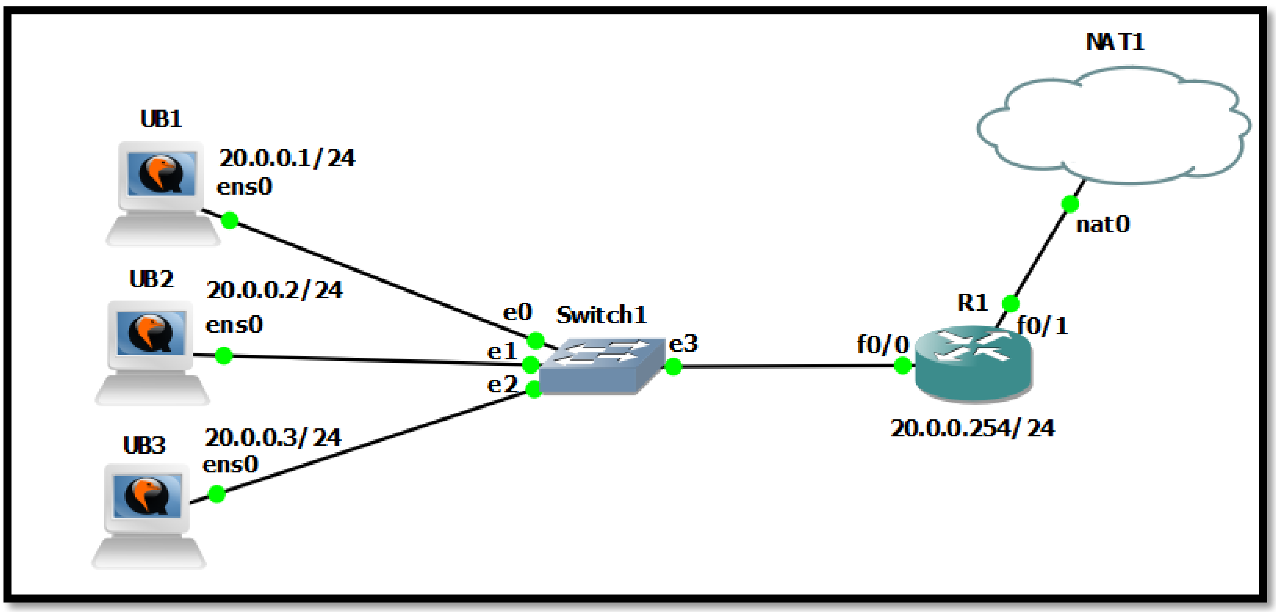


Figure 1 - Kubernetes networking network topology

Throughout this first part of the project, the study will focus on the network presented in Figure 1. Applied in this architecture are three Ubuntu 18.04.3 VMs, simulating Kubernetes nodes, one switch and one Cisco 3725 Router (R1).

The setup of the Ubuntu servers consisted of the installation of **Docker** and **Kubeadm**, the configuration of each machine's interfaces, and a **Kubernetes cluster**. These configurations are provided in Annex A.

1.1. Kubernetes control plane

A Kubernetes cluster is composed of a set of worker machines, called nodes, that run containerised applications. These worker nodes host the Pods, which are the components of the application workload.

This study starts by analysing the exchanged traffic between the worker nodes and the master node. (Kubernetes, 2022)

The Control Plane:

In Kubernetes, the control plane manages the cluster's worker nodes and Pods. In our topology and in production environments, the control plane runs across multiple machines, providing fault tolerance and high availability.

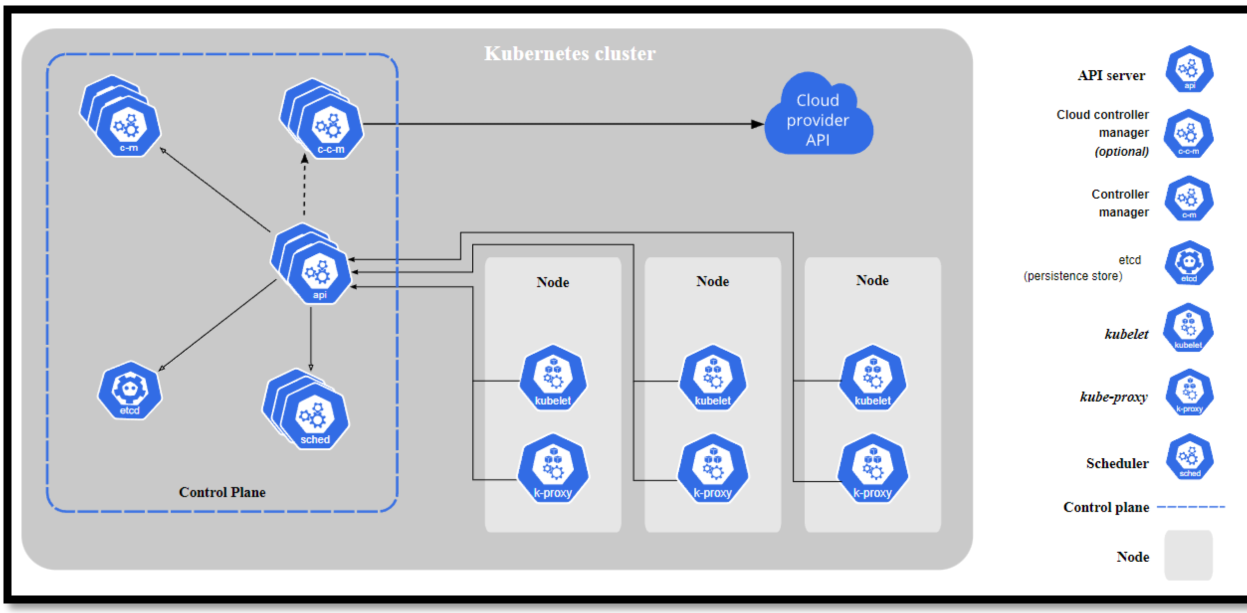


Figure 2 - Kubernetes cluster components (Kubernetes, 2022)

When running Kubernetes in an environment with strict network boundaries, it is helpful to be aware of the ports and protocols used by Kubernetes components. Table 1 shows the default ports and protocols used by the control plane with the reminder that all default ports can be overridden. (Kubernetes , 2022)

Table 1 - Control plane Ports and Protocols (Kubernetes , 2022)

Protocol	Direction	Port Range	Service	Used By
TCP	Inbound	6443	Kubernetes API server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10259	kube-scheduler	Self
TCP	Inbound	10257	kube-controller-manager	Self

In the following Figure 3 and Figure 4, it is possible to confirm two packets between the master and the worker nodes using port 6443 with Kubernetes API server information.

The packets from the master to the worker nodes use TLS1.2, while the packets from the worker nodes to the master use TCP.

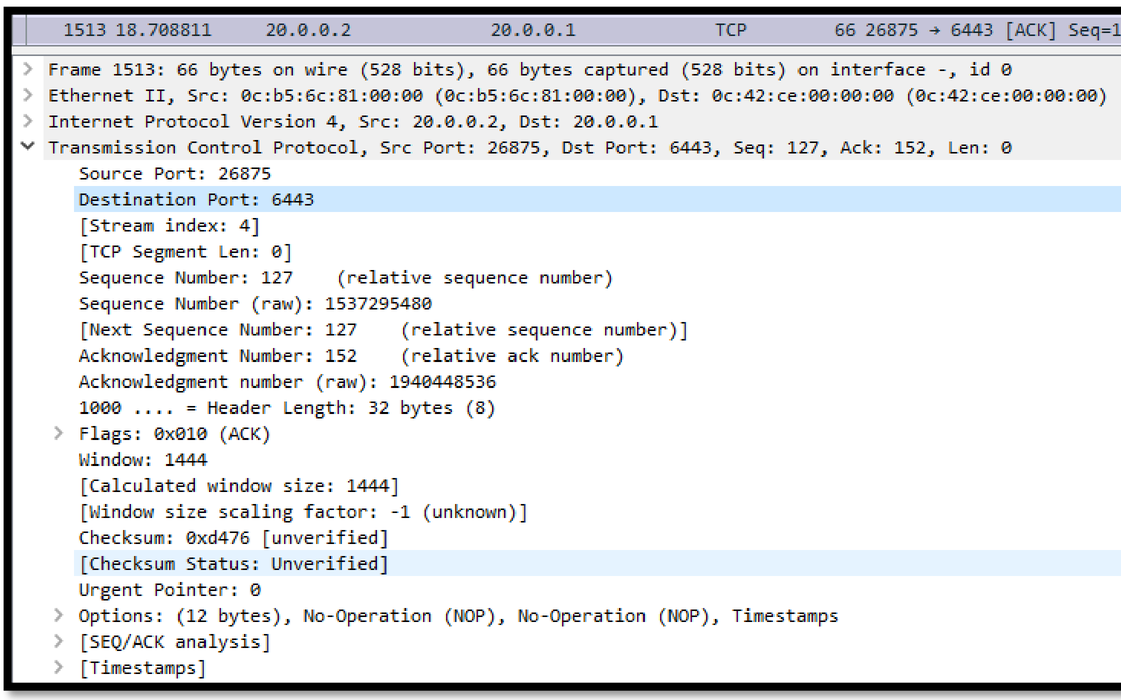


Figure 3 - Control plane packet from worker (UB1) to the master (UB2)

```
314 2.978017     20.0.0.1          20.0.0.2          TLSv1.2   560 Application Data

> Frame 314: 560 bytes on wire (4480 bits), 560 bytes captured (4480 bits) on interface -, id 0
> Ethernet II, Src: 0c:42:ce:00:00:00 (0c:42:ce:00:00:00), Dst: 0c:b5:6c:81:00:00 (0c:b5:6c:81:00:00)
> Internet Protocol Version 4, Src: 20.0.0.1, Dst: 20.0.0.2
v Transmission Control Protocol, Src Port: 6443, Dst Port: 48926, Seq: 126, Ack: 535, Len: 494
    Source Port: 6443
    Destination Port: 48926
```

Figure 4 - Control plane packet from master (UB1) to the worker (UB2)

1.2. Deploying a single pod

Pods are the smallest deployable units of computing that can be created and managed in Kubernetes. A Pod is a group of one or more containers with shared storage and network resources and a specification for how to run the containers.

In terms of Docker concepts, a Pod is similar to a group of Docker containers with shared namespaces and shared filesystem volumes. (Kubernetes, 2022)

This section presents the deployment of a single pod with an nginx container. The complete configuration can be found in Annex A.

Proof of Concept:

1. We create the pod named **mypod** with a nginx image. The pod is initially in a ContainerCreating state and then changes to a Running state. The created pod can be seen Running in Figure 5, with the IP address of 10.32.0.2 and located in node ub3.

```
root@UB1:/home/gns3# kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS        AGE   IP          NODE   NOMINATED NODE   READINESS GATES
mypod   1/1     Running   1 (2m52s ago)   50m   10.32.0.2   ub3    <none>           <none>
root@UB1:/home/gns3#
```

Figure 5 - mypod IP address and node

2. To explore the Pod, a shell is created. In Figure 6, that same shell can be observed along with some commands demonstrating some of the Pod's information.

```
root@UB1:/home/gns3# kubectl exec -it mypod -- /bin/bash
root@mypod:/# ls
bin   docker-entrypoint.d   home   media  proc  sbin  tmp
boot  docker-entrypoint.sh  lib    mnt    root  srv   usr
dev   etc                   lib64  opt    run   sys   var
root@mypod:/# cat /etc/hosts
# Kubernetes-managed hosts file.
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.32.0.2       mypod
root@mypod:/# cat /etc/hostname
mypod
root@mypod:/# cat /usr/share/nginx/html/index.html
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

Figure 6 - Inside mypod shell

3. To prove that the Pods' service is operational, curl is installed, and a GET request is sent to the nginx server. Figure 7 shows a successful message from that same request.

```
root@mypod:/# curl http://localhost/
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
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>
```

Figure 7 - GET request to the nginx server from mypod

1.3. Deploying a single pod through a YAML manifest

The most common way of deploying a Pod is through a YAML manifest. A manifest specifies the desired state of an object that Kubernetes will maintain when that same manifest is applied. Each configuration file can contain multiple manifests.

This section presents the deployment of the same pod of the previous section, 1.2, but using a YAML manifest.

Proof of Concept:

1. This procedure starts with the creation of a .yaml file specifying all the Pod's specifications. The contents of the file are shown in Figure 8.

```
  GNU nano 2.9.3                              pod_def.yaml

apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - image: nginx
    name: mycont
```

Figure 8 - mypod YAML Manifest

2. Now the command **kubectl create -f pod_def.yaml** is executed and the Pod is created as shown in Figure 9.

```
root@UB1:/home/gns3# kubectl create -f pod_def.yaml
pod/mypod created
root@UB1:/home/gns3# kubectl describe pods mypod
Name:         mypod
Namespace:    default
Priority:     0
Node:         ub3/20.0.0.3
Start Time:   Mon, 20 Jun 2022 18:11:33 +0000
Labels:       <none>
Annotations:  <none>
Status:       Running
IP:           10.32.0.2
IPs:
  IP:  10.32.0.2
```

Figure 9 - Pod creation and Information

3. If the same command is executed twice, an error is displayed. So, it can be concluded that running the Pod creation command several times is not the correct way to generate replicas of a specific Pod, as seen in Figure 10.

```
root@UB1:/home/gns3# kubectl create -f pod_def.yaml
Error from server (AlreadyExists): error when creating "pod_def.yaml": pods "mypod" already exists
root@UB1:/home/gns3#
```

Figure 10 - Repeating the creation command

4. The choice of the node where the Pod is created has been done automatically so far. To manually deploy the Pod in a specific node a new entry called **nodeName** must be added under **specs** on the YAML manifest. The Pod is now created again but now with the specification that it should be created on node UB2, as shown in Figure 11 and Figure 12.

```
  GNU nano 2.9.3                     pod_def.yaml

apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  nodeName: ub2
  containers:
  - image: nginx
    name: mycont
```

Figure 11 - Modified YAML Manifest with manual node choice

```
root@UB1:/home/gns3# kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP          NODE   NOMINATED NODE   READINESS GATES
mypod   1/1     Running   0          43s   10.38.0.1   ub2    <none>           <none>
root@UB1:/home/gns3#
```

Figure 12 - mypod created in node UB2

1.4. Deploying a ReplicaSet

A ReplicaSet aims to maintain a stable set of replica Pods running at a given time.

A ReplicaSet is defined with fields including a selector that specifies how to identify Pods, several replicas and a Pod template specifying the data of new Pods it should create to meet the number of replicas required. (Kubernetes, 2022)

This section demonstrates the deployment of five replicas of an nginx pod using a YAML manifest of a ReplicaSet. The complete configuration can be found in Annex A.

Proof of Concept:

1. The generation of five replicas of a nginx pod will be done through a YAML manifest of a ReplicaSet. The respective file can be seen in Figure 13.

```
GNU nano 2.9.3                                   rep_def1.yaml

apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myrep
spec:
  replicas: 5
  selector:
    matchLabels:
      app: mynginx
  template:
    metadata:
      labels:
        app: mynginx
    spec:
      containers:
      - name: mycont
        image: nginx
```

Figure 13 - ReplicaSet YAML Manifest

2. After the creation of the manifest, the set of replicas is created. The result and the main characteristics of the ReplicaSet can be seen in Figure 14, and the IP addresses plus the assigned nodes of each replica in Figure 15.

```
root@UB1:/home/gns3# kubectl create -f rep_def1.yaml
replicaset.apps/myrep created
root@UB1:/home/gns3# kubectl get replicasets -o wide
NAME     DESIRED   CURRENT   READY   AGE   CONTAINERS   IMAGES   SELECTOR
myrep    5         5         0       13s   mycont       nginx    app=mynginx
```

Figure 14 - ReplicaSet creation and Pods characteristics

```
root@UB1:/home/gns3# kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP          NODE   NOMINATED NODE   READINESS GATES
myrep-9bk2f   1/1     Running   0          87s   10.38.0.3   ub2    <none>           <none>
myrep-hc76x   1/1     Running   0          87s   10.32.0.3   ub3    <none>           <none>
myrep-jhc28   1/1     Running   0          87s   10.38.0.2   ub2    <none>           <none>
myrep-lsmz7   1/1     Running   0          87s   10.38.0.1   ub2    <none>           <none>
myrep-xphp8   1/1     Running   0          87s   10.32.0.2   ub3    <none>           <none>
root@UB1:/home/gns3#
```

Figure 15 - Pod's IP addresses and assigned nodes

3. Unlike when a user directly creates Pods, a ReplicaSet replaces Pods that are deleted or terminated for any reason, such as node failure or disruptive node maintenance, as a kernel upgrade. For this reason, it is recommended to use a ReplicaSet even if the application requires only a single Pod. Figure 16 shows the deletion of a Pod from the ReplicaSet that is immediately recreated in a different node, so it can be concluded that the pod that replaces the deleted is not always placed in the same node.

```
root@UB1:/home/gns3# kubectl delete pods myrep-9bk2f
pod "myrep-9bk2f" deleted
root@UB1:/home/gns3# kubectl get pods -o wide
NAME          READY   STATUS              RESTARTS   AGE    IP          NODE   NOMINATED NODE   READINESS GATES
myrep-hc76x   1/1     Running             0          3m1s   10.32.0.3   ub3    <none>           <none>
myrep-jhc28   1/1     Running             0          3m1s   10.38.0.2   ub2    <none>           <none>
myrep-lsmz7   1/1     Running             0          3m1s   10.38.0.1   ub2    <none>           <none>
myrep-n5hjv   0/1     ContainerCreating   0          6s     <none>      ub3    <none>           <none>
myrep-xphp8   1/1     Running             0          3m1s   10.32.0.2   ub3    <none>           <none>
root@UB1:/home/gns3#
```

Figure 16 - Pod deletion and replacement

4. Scaling a ReplicaSet to create more replicas is very straightforward. Using the command **kubectl scale replicasets myrep --replicas=7** two more replicas are immediately created, as shown in Figure 17.

```
root@UB1:/home/gns3# kubectl get pods -o wide
NAME          READY   STATUS              RESTARTS   AGE     IP          NODE   NOMINATED NODE   READINESS GATES
myrep-4vxhs   0/1     ContainerCreating   0          3s      <none>      ub2    <none>           <none>
myrep-bkfcc   0/1     ContainerCreating   0          3s      <none>      ub2    <none>           <none>
myrep-hc76x   1/1     Running             0          4m10s   10.32.0.3   ub3    <none>           <none>
myrep-jhc28   1/1     Running             0          4m10s   10.38.0.2   ub2    <none>           <none>
myrep-lsmz7   1/1     Running             0          4m10s   10.38.0.1   ub2    <none>           <none>
myrep-n5hjv   1/1     Running             0          75s     10.32.0.4   ub3    <none>           <none>
myrep-xphp8   1/1     Running             0          4m10s   10.32.0.2   ub3    <none>           <none>
root@UB1:/home/gns3#
```

Figure 17 - Scaling the ReplicaSet to seven Pods

5. When managing a ReplicaSet, there is no need to delete one Pod at a time. As shown in Figure 18, one can delete all Pods associated with a specific ReplicaSet.

```
root@UB1:/home/gns3# kubectl delete replicasets myrep
replicaset.apps "myrep" deleted
root@UB1:/home/gns3# kubectl get pods -o wide
NAME          READY   STATUS        RESTARTS   AGE     IP          NODE   NOMINATED NODE   READINESS GATES
myrep-4vxhs   1/1     Terminating   0          47s     10.38.0.4   ub2    <none>           <none>
myrep-bkfcc   1/1     Terminating   0          47s     10.38.0.3   ub2    <none>           <none>
myrep-hc76x   1/1     Terminating   0          4m54s   10.32.0.3   ub3    <none>           <none>
myrep-jhc28   1/1     Terminating   0          4m54s   10.38.0.2   ub2    <none>           <none>
myrep-lsmz7   1/1     Terminating   0          4m54s   10.38.0.1   ub2    <none>           <none>
myrep-n5hjv   1/1     Terminating   0          119s    10.32.0.4   ub3    <none>           <none>
myrep-xphp8   1/1     Terminating   0          4m54s   10.32.0.2   ub3    <none>           <none>
root@UB1:/home/gns3#
```

Figure 18 - Deleting the entire ReplicaSet

1.5. Deploying a ClusterIP service

Kubernetes provide Pods with their own IP addresses and a single DNS name for a set of Pods and can load-balance across them.

In Kubernetes, a Service is an abstraction that defines a logical set of Pods and a policy by which to access them. This kind of pattern is also sometimes used to apply a microservice.

There are several types of services in Kubernetes, the one applied in this exercise is called a **ClusterIP service**. This is a default service type, often used in Kubernetes. (Kubernetes , 2022)

Imagine deploying a microservices application in a cluster where a Pod runs a microservice container. That Pod will use a determined port number for that container and get an IP address from a Worker Node's range of IP addresses. With that port number and IP address the Pod can provide access to the microservice container. When creating a ReplicaSet for that specific microservice container the replica Pod opens the same port number but with a different IP address, just as demonstrated in Figure 19. The IP address of this replica may belong to a different range of IP addresses if it is created in a different worker node.

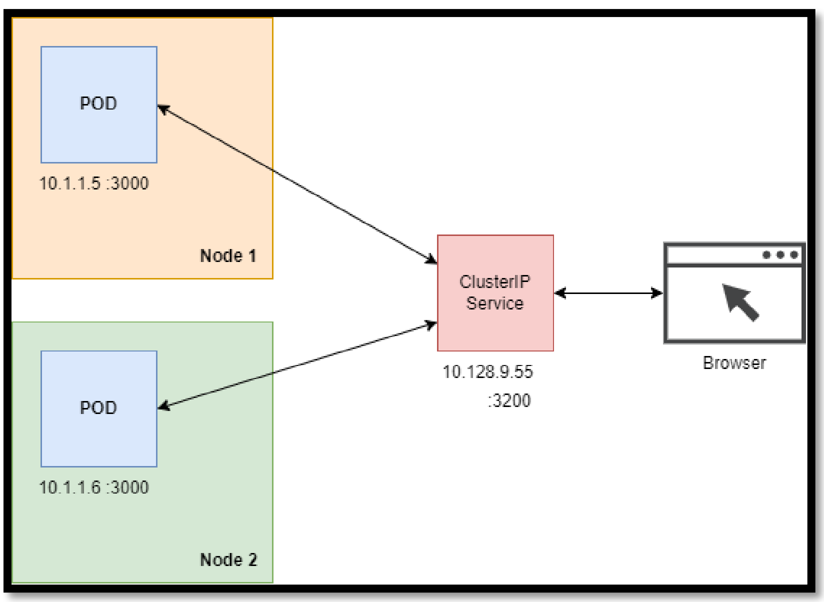


Figure 19 - ClusterIP service diagram

In the above situation, a ClusterIP service abstracts the ReplicaSet created for the microservice. When requests are made to that specific microservices, for example through a browser, a ClusterIP service with a certain port number and IP address handles those requests through the replicas. (Nana, 2020)

Proof of Concept:

This laboratory exercise demonstrates the deployment of a ClusterIP Service using a simple *nginxdemos/hello* application and a ReplicaSet. The complete configuration of this lab can be found in Annex A.

1. The first step is to create a Pod with curl installed in UB1. This Pod will be used to test the load balancing service. To generate this Pod, we create a Pod with a simple nginx image and then enter the Pod to install curl.

2. Then a ReplicaSet is created with three nginxdemos/hello image Pods. The replicas and the Pod assembled in the previous step can be seen in Figure 20.

```
root@UB1:/home/gns3# kubectl get pods -o wide
NAME            READY   STATUS    RESTARTS   AGE   IP          NODE   NOMINATED NODE   READINESS GATES
mycurlpod       1/1     Running   0          32m   10.32.0.2   ub2    <none>           <none>
myrep2-8tpqj    1/1     Running   0          22m   10.32.0.3   ub2    <none>           <none>
myrep2-flpp6    1/1     Running   0          26m   10.40.0.1   ub3    <none>           <none>
myrep2-mf7t9    1/1     Running   0          22m   10.40.0.4   ub3    <none>           <none>
```

Figure 20 - Replica and Curl Pods

3. Then follows the deployment of the ClusterIP service, where it is specified that the pods containing the **app: hello** will be handled by the ClusterIP. Now with the ReplicaSet and ClusterIP service created, we scale the ReplicaSet from three to five replicas. The information regarding the created service can be found in Figure 21, which presents the service's IP address, Selector, the number of Endpoints and more.

```
root@UB1:/home/gns3# kubectl describe svc mysvc1
Name:              mysvc1
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=myhello
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.105.112.62
IPs:               10.105.112.62
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.32.0.3:80,10.32.0.4:80,10.40.0.1:80 + 2 more...
Session Affinity:  None
Events:            <none>
root@UB1:/home/gns3#
```

Figure 21 - ClusterIP service description

4. Using the command **curl -s http://10.105.112.62 | grep name** from inside the curl equipped pod created earlier, we receive a response indicating the name of the pod that was contacted by curl. Repeating this process reveals no patterns; therefore, the load balancing algorithm is random, as illustrated in Figure 22.

```
root@UB1:/home/gns3# kubectl get services -o wide
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE    SELECTOR
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP   27h    <none>
mysvc1       ClusterIP   10.105.112.62   <none>        80/TCP    5m53s  app=myhello
root@UB1:/home/gns3# kubectl exec -it mycurlpod -- /bin/bash
root@mycurlpod:/# curl -s http://10.105.112.62 | grep name
<p><span>Server name:</span> <span>myrep2-8tpqj</span></p>
root@mycurlpod:/# curl -s http://10.105.112.62 | grep name
<p><span>Server name:</span> <span>myrep2-flpp6</span></p>
root@mycurlpod:/# curl -s http://10.105.112.62 | grep name
<p><span>Server name:</span> <span>myrep2-flpp6</span></p>
root@mycurlpod:/# curl -s http://10.105.112.62 | grep name
<p><span>Server name:</span> <span>myrep2-8tpqj</span></p>
root@mycurlpod:/# curl -s http://10.105.112.62 | grep name
<p><span>Server name:</span> <span>myrep2-flpp6</span></p>
root@mycurlpod:/# curl -s http://10.105.112.62 | grep name
<p><span>Server name:</span> <span>myrep2-qg6xf</span></p>
root@mycurlpod:/# curl -s http://10.105.112.62 | grep name
<p><span>Server name:</span> <span>myrep2-8tpqj</span></p>
root@mycurlpod:/# curl -s http://10.105.112.62 | grep name
<p><span>Server name:</span> <span>myrep2-flpp6</span></p>
root@mycurlpod:/# curl -s http://10.105.112.62 | grep name
<p><span>Server name:</span> <span>myrep2-mf7t9</span></p>
root@mycurlpod:/# curl -s http://10.105.112.62 | grep name
<p><span>Server name:</span> <span>myrep2-qg6xf</span></p>
root@mycurlpod:/# curl -s http://10.105.112.62 | grep name
<p><span>Server name:</span> <span>myrep2-8tpqj</span></p>
root@mycurlpod:/# exit
exit
root@UB1:/home/gns3#
```

Figure 22 - Pings from curl Pod to ClusterIP service

5. Now from one replica Pod, we ping another replica pod, and we can see in Figure 23 that the ping is successful. However, the same is not observed, in Figure 24, when we try to ping the ClusterIP service; this is because the Cluster IP is an abstraction of the service provided by the pods and does not reply to requests; it merely forwards them to a randomly chosen pod.

```
root@UB1:/home/gns3# kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE     IP          NODE   NOMINATED NODE   READINESS GATES
mycurlpod      1/1     Running   0          19m     10.32.0.2   ub2    <none>           <none>
myrep2-8tpqj   1/1     Running   0          9m52s   10.32.0.3   ub2    <none>           <none>
myrep2-flpp6   1/1     Running   0          13m     10.40.0.1   ub3    <none>           <none>
myrep2-mf7t9   1/1     Running   0          9m52s   10.40.0.4   ub3    <none>           <none>
myrep2-qg6xf   1/1     Running   0          10m     10.32.0.4   ub2    <none>           <none>
myrep2-z4mhq   1/1     Running   0          13m     10.40.0.3   ub3    <none>           <none>
root@UB1:/home/gns3# kubectl exec myrep2-8tpqj ping 10.40.0.3
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl exec [POD
ead.
PING 10.40.0.3 (10.40.0.3): 56 data bytes
64 bytes from 10.40.0.3: seq=0 ttl=64 time=5.682 ms
64 bytes from 10.40.0.3: seq=1 ttl=64 time=0.668 ms
64 bytes from 10.40.0.3: seq=2 ttl=64 time=0.758 ms
64 bytes from 10.40.0.3: seq=3 ttl=64 time=0.684 ms
64 bytes from 10.40.0.3: seq=4 ttl=64 time=0.977 ms
64 bytes from 10.40.0.3: seq=5 ttl=64 time=0.915 ms
^C
root@UB1:/home/gns3#
```

Figure 23 - Ping from one replica pod to another

```
root@UB1:/home/gns3# kubectl exec myrep2-8tpqj ping 10.105.112.62
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in
ead.
PING 10.105.112.62 (10.105.112.62): 56 data bytes
^C
root@UB1:/home/gns3#
```

Figure 24 - Ping from one replica pod to the ClusterIP service

1.6. Kubernetes DNS

In Kubernetes the DNS system schedules a DNS Pod and Service on the cluster and configures the kubelets to inform individual containers to use the DNS Service's IP to resolve DNS names. (Kubernetes , 2022)

Every Service Defined in the cluster is assigned a DNS name, including the DNS server itself. The complete configuration can be consulted in Annex A.

Proof of Concept:

1. To learn more about how DNS works in a Kubernetes cluster, we start by analysing the *kube-system* namespace. In Figure 25, we can see two DNS servers with IP addresses: 10.40.0.2 and 10.38.0.1. Figure 26 shows that the DNS servers are gathered in a ClusterIP service named **kube-dns**.

```
root@UB1:/home/gns3# kubectl get pods -n kube-system -o wide
NAME                          READY   STATUS    RESTARTS       AGE    IP          NODE   NOMINATED NODE   READINESS GATES
coredns-6d4b75cb6d-f8chl      1/1     Running   1 (110m ago)   120m   10.40.0.2   ub3    <none>           <none>
coredns-6d4b75cb6d-vsw84      1/1     Running   3 (96m ago)    120m   10.38.0.1   ub1    <none>           <none>
etcd-ub1                      1/1     Running   14 (96m ago)   28h    20.0.0.1    ub1    <none>           <none>
kube-apiserver-ub1            1/1     Running   13 (96m ago)   28h    20.0.0.1    ub1    <none>           <none>
kube-controller-manager-ub1   1/1     Running   9 (96m ago)    28h    20.0.0.1    ub1    <none>           <none>
kube-proxy-9864s              1/1     Running   4 (110m ago)   28h    20.0.0.3    ub3    <none>           <none>
kube-proxy-hvpmr              1/1     Running   6 (96m ago)    28h    20.0.0.1    ub1    <none>           <none>
kube-proxy-ksp5m              1/1     Running   3 (110m ago)   28h    20.0.0.2    ub2    <none>           <none>
kube-scheduler-ub1            1/1     Running   9 (96m ago)    28h    20.0.0.1    ub1    <none>           <none>
weave-net-26gmr               2/2     Running   7 (110m ago)   28h    20.0.0.2    ub2    <none>           <none>
weave-net-5ljzk               2/2     Running   13 (95m ago)   28h    20.0.0.1    ub1    <none>           <none>
weave-net-n4lzq               2/2     Running   9 (110m ago)   28h    20.0.0.3    ub3    <none>           <none>
root@UB1:/home/gns3#
```

Figure 25 - Pods in the kube-system namespace

```
root@UB1:/home/gns3# kubectl get service -n kube-system
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   28h
root@UB1:/home/gns3#
```

Figure 26 - DNS ClusterIP service

2. By consulting the *resolv.conf* file in several Pods, we can conclude that no Pod uses the IP address of a DNS server directly; they instead use the IP address of the ClusterIP service, **kube-dns**, that handles the DNS requests for the available DNS servers. Such a search can be verified in Figure 27.

```
root@UB1:/home/gns3# kubectl exec myrep2-8tpqj -- cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
root@UB1:/home/gns3# kubectl exec myrep2-flpp6 -- cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
root@UB1:/home/gns3# kubectl exec myrep2-z4mhq -- cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
root@UB1:/home/gns3#
```

Figure 27 - IP address of the DNS server on multiple Pods

3. To test this DNS service, we access the service mysvc1 using the service's DNS name. Using the commands presented below, it is possible to conclude that all these different commands work because they represent a smaller section of the search parameter shown in Figure 27 defined in the DNS server. As previously confirmed, all Pods use it.

```
#Access the service mysvc1 using the service's DNS name
kubectl exec mycurlpod -- curl -s http://mysvc1
kubectl exec mycurlpod -- curl -s http://kubectl exec mycurlpod -- curl -s http://mysvc1.kube-system
kubectl exec mycurlpod -- curl -s http://kubectl exec mycurlpod -- curl -s http://mysvc1.kube-system.svc
kubectl exec mycurlpod -- curl -s http://kubectl exec mycurlpod -- curl -s http://mysvc1.kube-system.svc.cluster.local
```

1.7. Deploying a NodePort service

The NodePort service allows the connection of external users to the cluster.

When creating a Service of type NodePort, Kubernetes provides a **nodePort** value. Then the Service is accessible by using the IP address of any node along with the **nodePort** value. (Google , 2022)

Proof of Concept:

1. A NodePort service is created using a YAML file selecting an app: myhello service and a nodePort: 30123. The service's IP address and port number can be seen in Figure 28.

```
root@UB1:/home/gns3# kubectl get svc -o wide
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE   SELECTOR
kubernetes   ClusterIP   10.96.0.1     <none>        443/TCP        45h   <none>
mysvc2       NodePort    10.96.1.251   <none>        80:30123/TCP   8s    app=myhello
```

Figure 28 - NodePort Service

2. Then the ReplicaSet is scaled to three replicas, as we can see in Figure 29, and a webterm and Toolbox are connected to the network. Using the IP address of one of the worker nodes and the port number provided by the NodePort service, we can confirm the service's success in Figure 30.

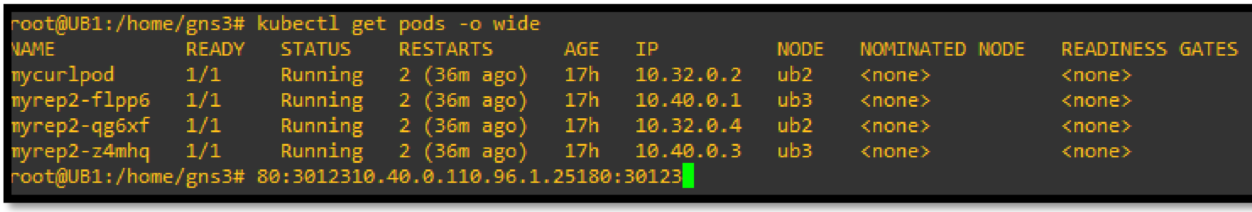


Figure 29 - Replica Pods rescaled

Figure 30 - Webterm browser HTTP request

3. By using the Toolbox to access the containers, we can grep the name of the Pod used in each call. By doing this, it is possible to conclude that the type of load balancing algorithm is random since no pattern was identified in a consecutive experience, showed in Figure 31.

```
100  7217    0  7217    0     0  201k      0 --:--:-- --:--:-- --:--:--  207k
<p><span>Server name:</span> <span>myrep2-flpp6</span></p>
root@Toolbox-1:~# curl http://20.0.0.2:30123 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0  370k      0 --:--:-- --:--:-- --:--:--  370k
<p><span>Server name:</span> <span>myrep2-qg6xf</span></p>
root@Toolbox-1:~# curl http://20.0.0.2:30123 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0  306k      0 --:--:-- --:--:-- --:--:--  440k
<p><span>Server name:</span> <span>myrep2-flpp6</span></p>
root@Toolbox-1:~# curl http://20.0.0.2:30123 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0 1409k      0 --:--:-- --:--:-- --:--:-- 1409k
<p><span>Server name:</span> <span>myrep2-qg6xf</span></p>
root@Toolbox-1:~# curl http://20.0.0.2:30123 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0  271k      0 --:--:-- --:--:-- --:--:--  271k
<p><span>Server name:</span> <span>myrep2-z4mhq</span></p>
root@Toolbox-1:~# curl http://20.0.0.2:30123 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0  783k      0 --:--:-- --:--:-- --:--:--  880k
<p><span>Server name:</span> <span>myrep2-z4mhq</span></p>
root@Toolbox-1:~# curl http://20.0.0.2:30123 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0  783k      0 --:--:-- --:--:-- --:--:-- 1761k
<p><span>Server name:</span> <span>myrep2-z4mhq</span></p>
root@Toolbox-1:~# curl http://20.0.0.2:30123 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0  414k      0 --:--:-- --:--:-- --:--:--  414k
<p><span>Server name:</span> <span>myrep2-z4mhq</span></p>
root@Toolbox-1:~#
```

Figure 31 - Consecutive HTTP requests to UB2

4. By analysing the flow of HTTP messages in a Wireshark capture, it is possible to conclude that when a message arrives at a node but is assigned to a Pod on another node, the node contacted is the one that sends the HTTP response. Such an experiment is demonstrated in Figure 32, where an HTTP request is forwarded to UB2 but gets assigned to Pod myrep2-z4mhq located in UB3. In Figure 33, we can see that although assigned to a Pod on a different host, it is still UB2 that responds.

```
root@Toolbox-1:~# curl http://20.0.0.2:30123 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0  704k      0 --:--:-- --:--:-- --:--:--  704k
<p><span>Server name:</span> <span>myrep2-z4mhq</span></p>
root@Toolbox-1:~#
```

Figure 32 - HTTP request to UB2 assigned to Pod at UB3

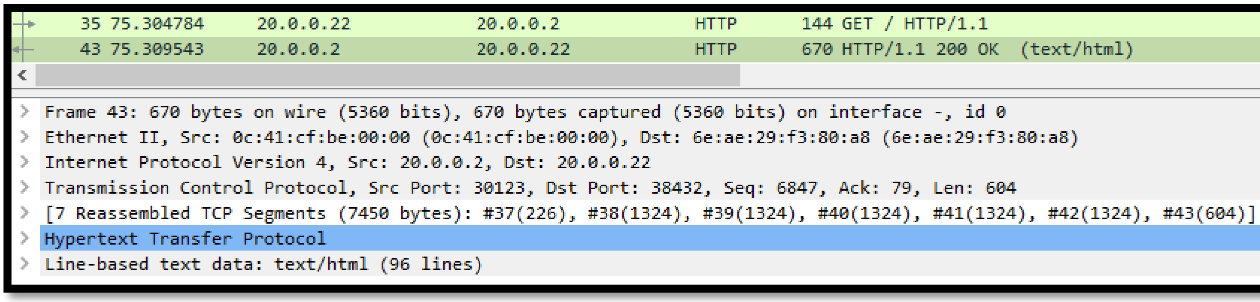


Figure 33 - HTTP message flow Request to UB2

2. Docker Swarm

This section aims to explore Docker technology, in particular comparing a service's deployment in swarm versus Kubernetes. The laboratory topics analysed are:

I. Deploying a Service in Docker Swarm

2.1. <u>Deploying a Service in Docker Swarm</u>

The objective of this exercise is to compare the deployment of services between a docker swarm and a Kubernetes cluster.

Docker swarm makes a cluster orchestration more manageable and allows the opportunity to publish ports for services. Usually, the only way to access containers is through their host machine's IP address, but in a swarm, all nodes participate in an ingress routing mesh.

<u>Swarm mode Routing Mesh</u>

Docker Engine swarm mode eases the process of publishing ports for services to make them available to resources outside the swarm. The routing mesh enables each node in the swarm to accept connections on published ports for any service in the swarm, even if no task is running on the node. (Docker, 2022)

The routing mesh routes all incoming requests to published ports on available nodes to an active container.

The ingress mesh uses the following ports:

- Port 7946 TCP/UDP for container network discovery.
- Port 4789 UDP for the container ingress network.

Network Architecture

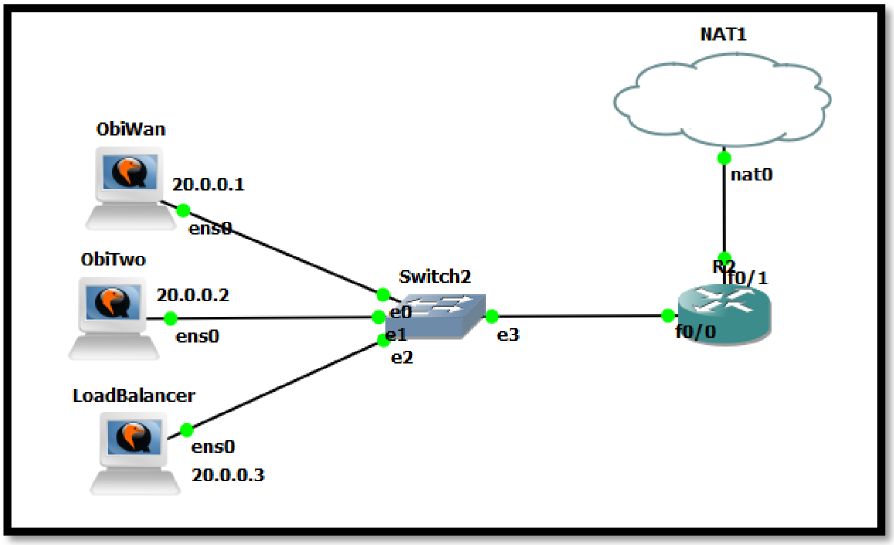


Figure 34 - Docker swarm architecture

In this architecture, we use one docker swarm environment with ObiWan and ObiTwo and a separate Load Balancer machine; all are Ubuntu 18.04 machines. To connect them to the internet, we use NAT, a Cisco 3725 router and a standard Switch. The complete configuration can be found in Annex A.

Proof of Concept:

To better understand this technology, we demonstrate the publication of a simple nginxdemos/hello web service using a Docker swarm and a Load Balancer.

1. First, we enable the docker swarm for two of the three nodes presented in Figure 34, ObiWan as the manager and ObiTwo as the worker. Then start the web host nginxdemos/hello service in two replicas and publish them to external port 8080, shown in Figure 35.

```
root@ObiWan:/home/gns3# docker service ls
ID             NAME      MODE         REPLICAS   IMAGE                      PORTS
pp1rbsvejvcg   backend   replicated   2/2        nginxdemos/hello:latest    *:8080->80/tcp
```

Figure 35 - Docker services in Docker Swarm

2. For consistency, the load balancer is deployed on a single node outside the swarm as shown in Figure 36. As all the published services are available through any of the swarm nodes thanks to the ingress routing, the load balancer can be set to use the swarm private IP addresses without concern about which node is hosting what service.

```
root@LB1:/home/gns3# docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
xghfnklhf5crq5buk0qdmgqaa *   LB1        Ready     Active         Leader           20.10.7
```

Figure 36 - Load Balancer deployment

3. Then we create the load balancing container using the configuration file presented in Figure 37. As we can see, the file contains the IP addresses of both swarm nodes, and it is specified to publish the container to port 80.

```
  GNU nano 2.9.3            /data/loadbalancer/default.conf

server {
   listen 80;
   location / {
      proxy_pass http://backend;
   }
}
upstream backend {
   server 20.0.0.1:8080;
   server 20.0.0.2:8080;
}
```

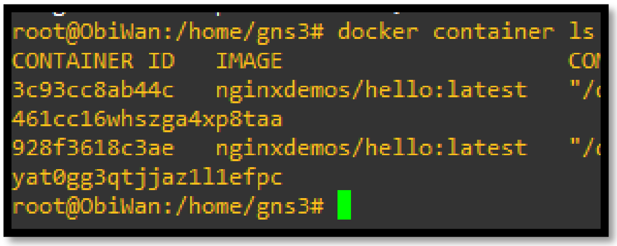
Figure 37 - Load balancing container configuration file

4. Now to test the load balancer we attach a Webterm to the network and execute an HTTP GET request from its web browser, using the IP of the load balancer and as shown in Figure 38 the connection to the web service is successful.

Figure 38 - HTTP GET Request from Webterm browser

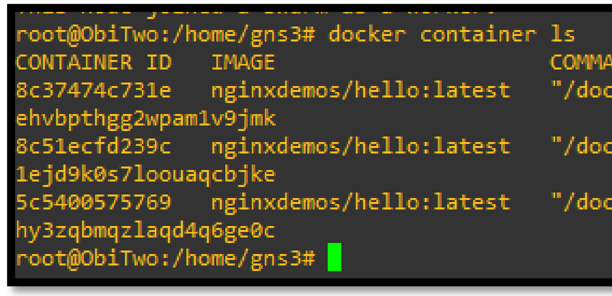
5. To better understand the type of load balancing method used in this solution we will execute multiple HTTP Get requests to the load balancer. For such a purpose we first, identify each container and where they are located; we can see the containers created in

node ObiWan and ObiTwo in Figure 40 and Figure 39, respectively. We can now conclude that the container used in the previous web browser request is located in ObiTwo.



```
root@ObiWan:/home/gns3# docker container ls
CONTAINER ID   IMAGE                     CO
3c93cc8ab44c   nginxdemos/hello:latest   "/
461cc16whszga4xp8taa
928f3618c3ae   nginxdemos/hello:latest   "/
yat0gg3qtjjaz1l1efpc
root@ObiWan:/home/gns3#
```

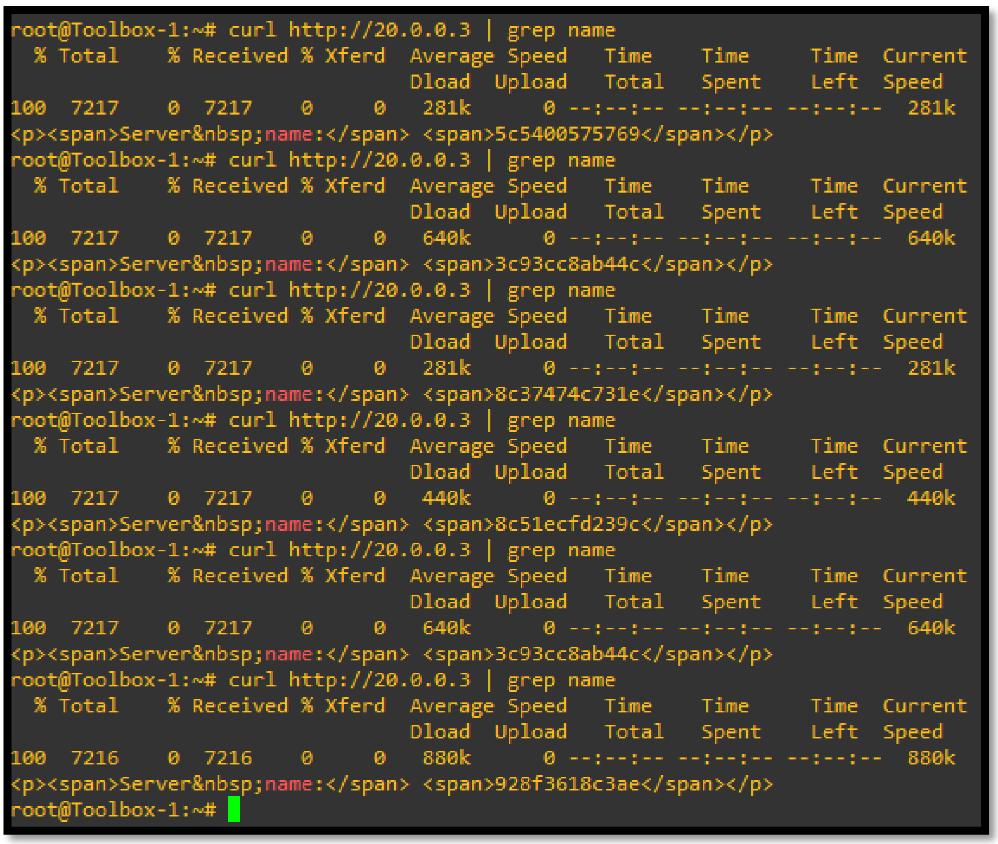
Figure 40 - Containers in ObiWan



```
root@ObiTwo:/home/gns3# docker container ls
CONTAINER ID   IMAGE                     COMMA
8c37474c731e   nginxdemos/hello:latest   "/doc
ehvbpthgg2wpam1v9jmk
8c51ecfd239c   nginxdemos/hello:latest   "/doc
1ejd9k0s7loouaqcbjke
5c5400575769   nginxdemos/hello:latest   "/doc
hy3zqbmqzlaqd4q6ge0c
root@ObiTwo:/home/gns3#
```

Figure 39 - Containers in ObiTwo

6. Analysing the successive requests presented in Figure 41, we can conclude that the load balancing algorithm is merely random with no pattern identified.



```
root@Toolbox-1:~# curl http://20.0.0.3 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0   281k      0 --:--:-- --:--:-- --:--:--  281k
<p><span>Server name:</span> <span>5c5400575769</span></p>
root@Toolbox-1:~# curl http://20.0.0.3 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0   640k      0 --:--:-- --:--:-- --:--:--  640k
<p><span>Server name:</span> <span>3c93cc8ab44c</span></p>
root@Toolbox-1:~# curl http://20.0.0.3 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0   281k      0 --:--:-- --:--:-- --:--:--  281k
<p><span>Server name:</span> <span>8c37474c731e</span></p>
root@Toolbox-1:~# curl http://20.0.0.3 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0   440k      0 --:--:-- --:--:-- --:--:--  440k
<p><span>Server name:</span> <span>8c51ecfd239c</span></p>
root@Toolbox-1:~# curl http://20.0.0.3 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7217    0  7217    0     0   640k      0 --:--:-- --:--:-- --:--:--  640k
<p><span>Server name:</span> <span>3c93cc8ab44c</span></p>
root@Toolbox-1:~# curl http://20.0.0.3 | grep name
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7216    0  7216    0     0   880k      0 --:--:-- --:--:-- --:--:--  880k
<p><span>Server name:</span> <span>928f3618c3ae</span></p>
root@Toolbox-1:~#
```

Figure 41 - Successive HTTP requests from Toolbox

References:

Docker. (2022). *Use swarm mode routing mesh*. Retrieved from Docker Documentation: https://docs.docker.com/engine/swarm/ingress/

Google . (2022). *Services*. Retrieved from Google Clound: https://cloud.google.com/kubernetes-engine/docs/concepts/service

Kubernetes . (2022, May 30). *DNS for Services and Pods*. Retrieved from Kubernetes Documentation: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

Kubernetes . (2022, May 09). *Ports and Protocols*. Retrieved from Kubernetes Documentation: https://kubernetes.io/docs/reference/ports-and-protocols/

Kubernetes . (2022, May 31). *Service*. Retrieved from Kubernetes Documentation: https://kubernetes.io/docs/concepts/services-networking/service/

Kubernetes. (2022, April 30). *Kubernetes Components*. Retrieved from Kubernetes: https://kubernetes.io/docs/concepts/overview/components/

Kubernetes. (2022, January 10). *Pods*. Retrieved from Kubernetes Documentation: https://kubernetes.io/docs/concepts/workloads/pods/#:~:text=Pods%20are%20the%20smallest%20deployable,how%20to%20run%20the%20containers.

Kubernetes. (2022, June 07). *ReplicaSet*. Retrieved from Kubernetes Documentation: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

Nana, T. w. (2020, October 28). *Kubernetes Services explained | ClusterIP vs NodePort vs LoadBalancer vs Headless Service*. Retrieved from YouTube: https://www.youtube.com/watch?v=T4Z7visMM4E

Annex A: Configurations

Docker and Kubernetes installation in GNS3:

```
R1 Configuration:

conf t
int f0/0
ip add 20.0.0.254 255.255.255.0
ip nat inside
no shut

int f0/1
ip add dhcp
ip nat outside
no shut
exit
ip nat inside source list 1 interface FastEthernet0/1 overload
end

conf t
access-list 1 permit 20.0.0.0 0.0.0.255
end

####################################################################
UB1 Configuration:

hostnamectl set-hostname UB1
exec bash
nano /etc/netplan/50-cloud-init.yaml

network:
  version: 2
  renderer: networkd
  ethernets:
    ens3:
      addresses:
        - 20.0.0.1/24
      gateway4: 20.0.0.254
      nameservers:
        addresses:
          - 8.8.8.8

netplan apply
reboot

#Docker installation
apt-get update
apt install docker.io
reboot
systemctl status docker
sudo apt-get update && sudo apt-get upgrade

#Kubeadm installation
nano /etc/fstab
reboot

nano /lib/systemd/system/docker.service
ExecStart=/usr/bin/dockerd --exec-opt native.cgroupdriver=systemd
systemctl status docker

apt-get update
```

```
apt-get install -y apt-transport-https ca-certificates curl
curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg https://packages.cloud.google.com/apt/doc/apt-key.gpg
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc/apt/sources.list.d/kubernetes.list

apt-get update
apt-get install -y kubelet kubeadm kubectl
apt-mark hold kubelet kubeadm kubectl

kubeadm config images pull

#Docker images for labs
docker pull nginx
nano app.js
const http = require('http');
const os = require('os');
console.log("Kubia server starting...");
var handler = function(request, response) {
 console.log("Received request from " + request.connection.remoteAddress);
 response.writeHead(200);
 response.end("You've hit " + os.hostname() + "\n");
};
var www = http.createServer(handler);
www.listen(8080);

nano Dockerfile
FROM node:7
ADD app.js /app.js
ENTRYPOINT ["node", "app.js"]

docker build -t kubia .
docker image ls

#Kubernetes cluster
kubeadm init --apiserver-advertise-address=20.0.0.1 --pod-network-cidr=10.32.0.0/12

mkdir -p $HOME/.kube
cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
chown $(id -u):$(id -g) $HOME/.kube/config

kubectl apply -f https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n');

kubectl get pods -A
kubectl get nodes
#####################################################################
Ubuntu 2 Configuration:

hostnamectl set-hostname UB2
nano /etc/netplan/50-cloud-init.yaml

network:
  version: 2
  renderer: networkd
  ethernets:
    ens3:
      addresses:
        - 20.0.0.2/24
      gateway4: 20.0.0.254
      nameservers:
        addresses:
          - 8.8.8.8
netplan apply
```

```
reboot


#Join worker
kubeadm join 20.0.0.1:6443 --token vjrxi0.noa2wizoz107dsxu --discovery-token-ca-cert-hash sha256:b44227d48388f9d8c7edceacaa27a825141d7dd9055d642bbd9784c0371bf74f

#####################################################################
Ubuntu 3 Configuration:

hostnamectl set-hostname UB3
nano /etc/netplan/50-cloud-init.yaml

network:
  version: 2
  renderer: networkd
  ethernets:
    ens3:
      addresses:
        - 20.0.0.3/24
      gateway4: 20.0.0.254
      nameservers:
        addresses:
          - 8.8.8.8

netplan apply
reboot

#Join worker
kubeadm join 20.0.0.1:6443 --token vjrxi0.noa2wizoz107dsxu --discovery-token-ca-cert-hash sha256:b44227d48388f9d8c7edceacaa27a825141d7dd9055d642bbd9784c0371bf74f
```

Deploying a single pod

```
sudo /etc/init.d/networking restart
apt-get update
docker pull nginx

kubectl get pods -A
kubectl run mypod --image=nginx
kubectl get pods

#Identify the IP address assigned to the pod and the node the pod was assigned to
kubectl get pods -o wide
kubectl describe pods mypod * | grep Node * | grep IP

#Explore the pod interior
kubectl exec -it mypod -- /bin/bash
ls
cat /etc/hosts
cat /etc/hostname
cat /usr/share/nginx/html/index.html

#Install curl and send a GET request to the nginx server
apt-get update
apt-get install curl

curl http://localhost/
exit
```

Deploying a ReplicaSet

```
#Create Replicas Manifest
nano rep_def1.yaml

apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myrep
spec:
  replicas: 5
  selector:
    matchLabels:
      app: mynginx
  template:
    metadata:
      labels:
        app: mynginx
    spec:
      containers:
      - name: mycont
        image: nginx

#Create Replicas
kubectl create -f rep_def1.yaml
kubectl get replicasets -o wide
kubectl get pods -o wide

#Delete one pod
kubectl delete pods myrep-9bk2f

#Scale the ReplicaSet to seven pods
kubectl scale replicasets myrep --replicas=7
kubectl get pods -o wide

#Delete the replica set
kubectl delete replicasets myrep

```

Deploying a ClusterIP service

```
#Create a pod that runs curl:

kubectl run mypod --image=nginx
kubectl get pods -A
kubectl exec -it mycurlpod -- /bin/bash
apt update
apt install curl
curl --version

kubectl delete pod mycurlpod

#Create the ReplicaSet
nano rep_def1.yaml

apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myrep2
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myhello
  template:
    metadata:
      labels:
        app: myhello
    spec:
      containers:
      - name: mycont
        image: nginxdemos/hello

kubectl create -f rep_def1.yaml

kubectl get replicasets -o wide
kubectl get pods -A
kubectl get nodes

kubectl delete replicasets myrep2

#Deploy the ClusterIP service
nano svc_def1.yaml

apiVersion: v1
kind: Service
metadata:
  name: mysvc1
spec:
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: myhello

kubectl create -f svc_def1.yaml

kubectl get services -o wide
kubectl describe svc mysvc1
kubectl get endpoints mysvc1

kubectl delete services mysvc1
```

```

#Rescale the ReplicaSet to 5 pods
kubectl scale rs myrep2 --replicas=5
kubectl get pods -o wide

#Type of load balancing algorithm is being used
kubectl get pods
kubectl exec -it mycurlpod -- /bin/bash
curl -s http://10.105.112.62 | grep name

#Pings
kubectl get pods -o wide
kubectl exec myrep2-8tpqj ping 10.40.0.3
kubectl exec myrep2-8tpqj ping 10.105.112.62
```

Kubernetes DNS

```
#Analyze the pods of the kube-system namespace
kubectl get pods -n kube-system -o wide

#Check that the DNS servers are gathered in a ClusterIP service
kubectl get service -n kube-system
kubectl get endpoints -n kube-system kube-dns

#Look for the IP address of the DNS server
kubectl exec <pod name> -- cat /etc/resolv.conf

#Access the service mysvc1 using the service's DNS name
kubectl exec mycurlpod -- curl -s http://mysvc1
kubectl exec mycurlpod -- curl -s http://kubectl exec mycurlpod -- curl -s http://mysvc1.kube-system
kubectl exec mycurlpod -- curl -s http://kubectl exec mycurlpod -- curl -s http://mysvc1.kube-system.svc
kubectl exec mycurlpod -- curl -s http://kubectl exec mycurlpod -- curl -s http://mysvc1.kube-system.svc.cluster.local

#Delete the service
kubectl delete svc mysvc1
```

Deploying a Service in Docker Swarm

```
#Initiate Cluster manager
docker swarm init --advertise-addr 20.0.0.1

#Join worker node
docker swarm join --token SWMTKN-1-1nku8t8e8g3bckl4fzu7hb2ovjcutg2jbl48qzfw9321tkh1uv-29z31vn96me8je9w7rug1n88j 20.0.0.1:2377

#Create Replica service
docker service create --name backend --replicas 2 --publish 8080:80 nginxdemos/hello
docker service scale backend=5

#Setup Load Balancer
docker swarm init --advertise-addr 20.0.0.3
mkdir -p /data/loadbalancer

nano /data/loadbalancer/default.conf

server {
   listen 80;
   location / {
      proxy_pass http://backend;
   }
}
upstream backend {
   server 20.0.0.1:8080;
   server 20.0.0.2:8080;
}

docker service create --name loadbalancer --mount type=bind,source=/data/loadbalancer,target=/etc/nginx/conf.d --publish 80:80 nginx

#Test Load balancer
http://20.0.0.3
curl http://<LoadBalancer IP address> | grep name

```