CO515: Advanced Computer Networks: Selected Topics - Lab 02

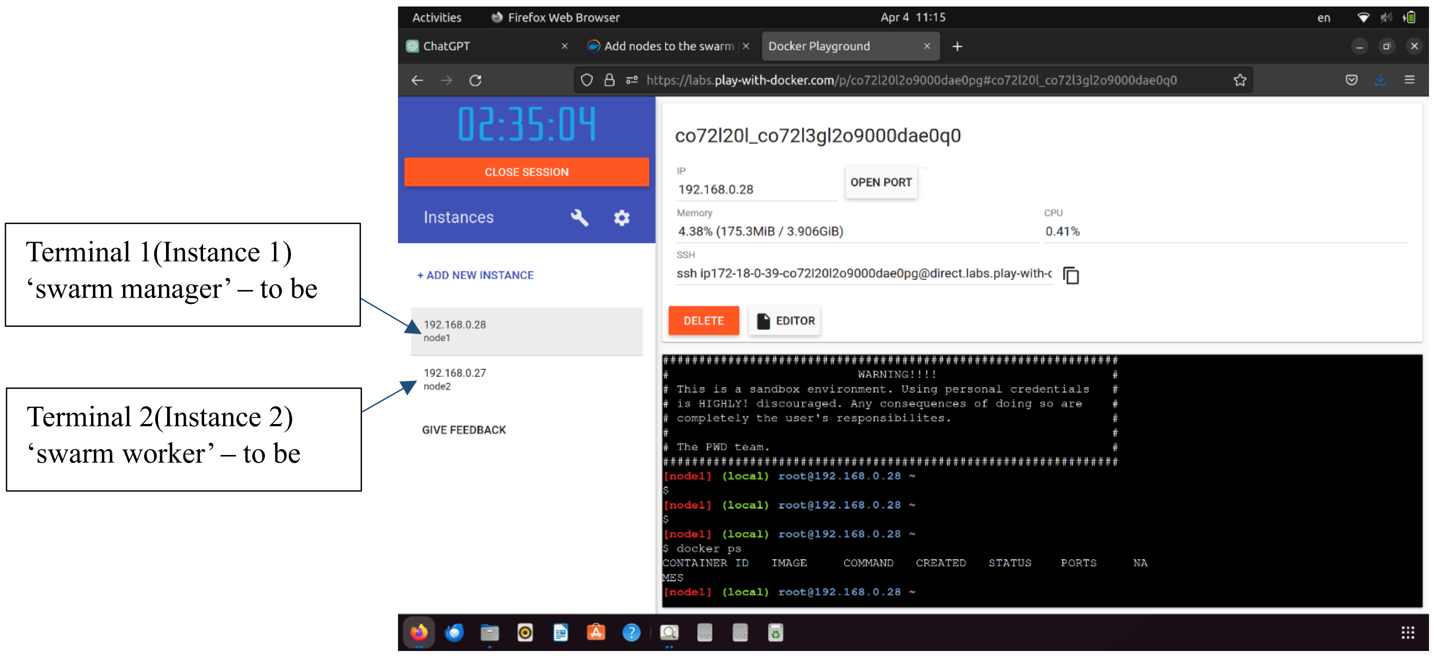
E/19/446: Wijerathna I.M.K.D.I.

04/04/2024

Lab Task 01

1. In this step you'll initialize a new Swarm, join a single worker node, and verify the operations worked.

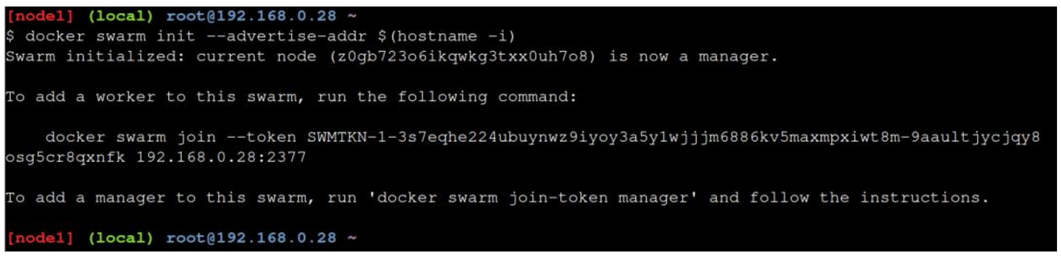
For this lab [Play with Docker (play-with-docker.com)](play-with-docker.com) platform was used since it is easier to create a swarm network using it.


Terminal 1(Instance 1) 'swarm manager' – to be
Terminal 2(Instance 2) 'swarm worker' – to be
Activities
Firefox Web Browser
Apr 4 11:15
ChatGPT
Add nodes to the swarm
Docker Playground
https://labs.play-with-docker.com/p/co72l20l2o9000dae0pg#co72l20l_co72l3gl2o9000dae0q0
02:35:04
CLOSE SESSION
Instances
+ ADD NEW INSTANCE
192.168.0.28 node1
192.168.0.27 node2
GIVE FEEDBACK
co72l20l_co72l3gl2o9000dae0q0
IP 192.168.0.28
OPEN PORT
Memory 4.38% (175.3MiB / 3.906GiB)
CPU 0.41%
SSH ssh ip172-18-0-39-co72l20l2o9000dae0pg@direct.labs.play-with-c
DELETE
EDITOR

```
##################################################################
#                        WARNING!!!!                             #
# This is a sandbox environment. Using personal credentials      #
# is HIGHLY! discouraged. Any consequences of doing so are       #
# completely the user's responsibilites.                         #
#                                                                #
# The PWD team.                                                  #
##################################################################
[node1] (local) root@192.168.0.28 ~
$
[node1] (local) root@192.168.0.28 ~
$
[node1] (local) root@192.168.0.28 ~
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NA
MES
[node1] (local) root@192.168.0.28 ~
```

Figure 1: Docker Play Ground Platform Setup

1.1. Run "docker swarm init --advertise-addr $(hostname -i) "



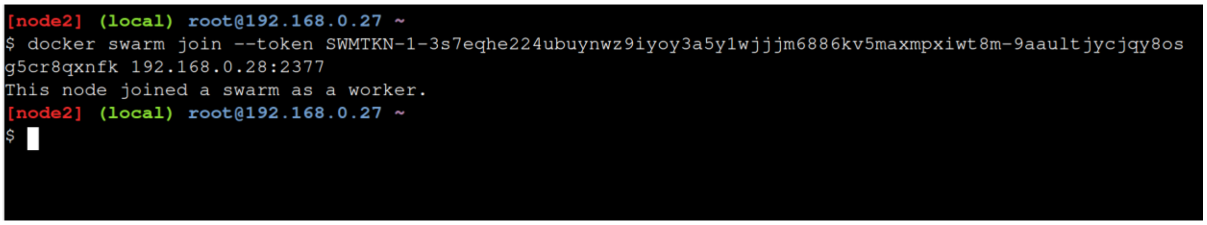
```
[node1] (local) root@192.168.0.28 ~
$ docker swarm init --advertise-addr $(hostname -i)
Swarm initialized: current node (z0gb723o6ikqwkg3txx0uh7o8) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-3s7eqhe224ubuynwz9iyoy3a5y1wjjjm6886kv5maxmpxiwt8m-9aaultjycjqy8
osg5cr8qxnfk 192.168.0.28:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

[node1] (local) root@192.168.0.28 ~
```

This command sets up a Docker swarm on the current node, designates it as a manager, and provides instructions for adding other nodes to the swarm as workers or managers.

- *"docker swarm init": This part of the command initializes a Docker swarm.*

- *"--advertise-addr $(hostname -i)": This flag specifies the address to advertise to other nodes in the swarm. In here, it's using the output of 'hostname -i' to dynamically get the IP address of the current node.*

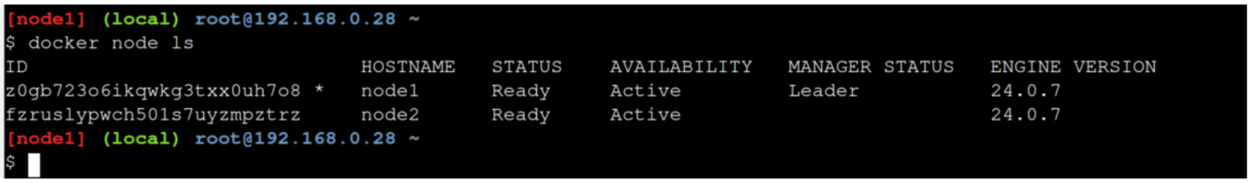
1.2. In the first terminal copy the entire "docker swarm join ..." command that is displayed as part of the output from your terminal output. Then, paste the copied command into the second terminal.



```
[node2] (local) root@192.168.0.27 ~
$ docker swarm join --token SWMTKN-1-3s7eqhe224ubuynwz9iyoy3a5y1wjjjm6886kv5maxmpxiwt8m-9aaultjycjqy8os
g5cr8qxnfk 192.168.0.28:2377
This node joined a swarm as a worker.
[node2] (local) root@192.168.0.27 ~
$ 
```

This command successfully joined the 2nd node as a worker to the swarm network which was created by node 1.

1.3. Run a "docker node ls" to verify that both nodes are part of the Swarm. Check whether both nodes have joined the Swarm and are ready and active



```
[node1] (local) root@192.168.0.28 ~
$ docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
z0gb723o6ikqwkg3txx0uh7o8 *   node1      Ready     Active         Leader           24.0.7
fzruslypwch501s7uyzmpztrz     node2      Ready     Active                          24.0.7
[node1] (local) root@192.168.0.28 ~
$ 
```

This command is used to list the nodes within the Docker swarm network. It lists down following information about each node:

- *ID: The unique identifier for each node in the swarm network.*
- *HOSTNAME: The hostname of the node.*
- *STATUS: The current status of the node. In this case, "Ready" indicates that the node is ready to accept tasks from the swarm.*
- *AVAILABILITY: Indicates the availability of the node. "Active" means the node is available for scheduling tasks.*
- *MANAGER STATUS: Indicates whether the node is a manager or a worker in the swarm. "Leader" indicates that the node is the leader manager in the swarm.*
- *ENGINE VERSION: The version of the Docker Engine running on the node.*

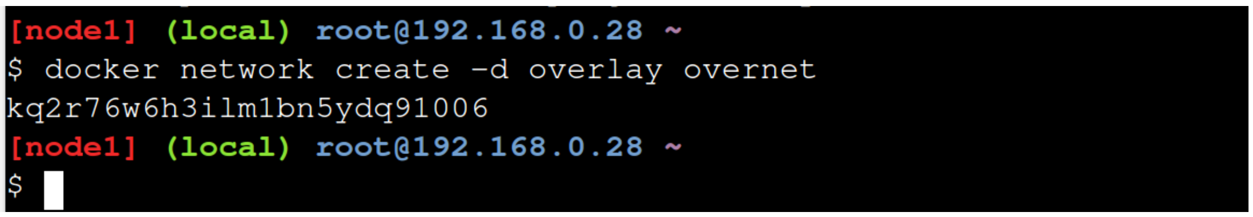
In the obtained output:

i. *The first node (ID: z0gb723o6ikqwkg3txx0uh7o8) is named "node1". It's marked with an asterisk (*), indicating that it's the current node where the command was executed. It's also marked as the leader/ manager node of the swarm.*

ii. *The second node (ID: fzruslypwch501s7uyzmpztrz) is named "node2". It's marked as a worker node because it doesn't have the "Leader" status under "MANAGER STATUS".*

2. Create an overlay network

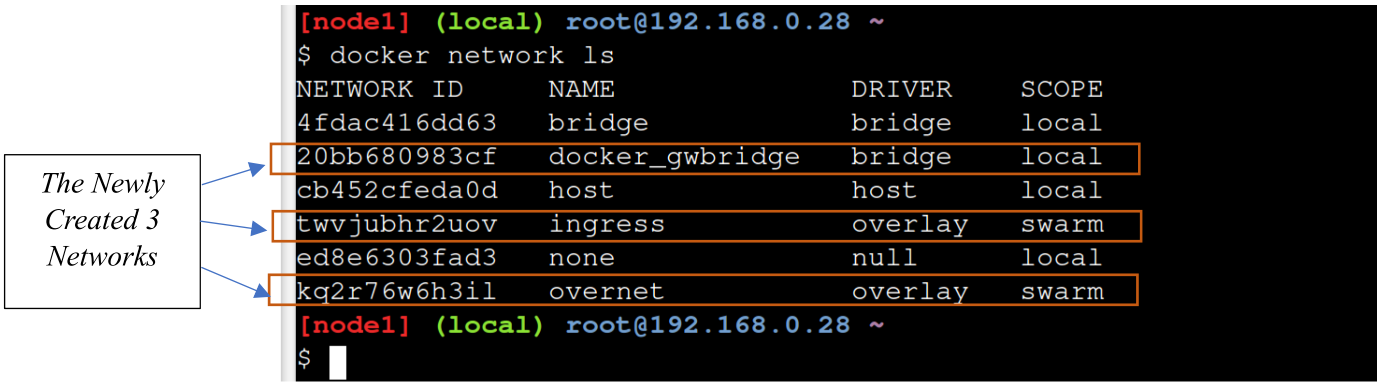
2.1. Create a new overlay network called "overnet" by running docker network create -d overlay overnet



```
[node1] (local) root@192.168.0.28 ~
$ docker network create -d overlay overnet
kq2r76w6h3ilm1bn5ydq91006
[node1] (local) root@192.168.0.28 ~
$ 
```

This command creates an overlay network named "overnet" using the overlay driver, which is the suitable driver-type for Docker Swarm deployments.

2.2. Use the "docker network ls" command to verify the network was created successfully. Notice how the new "overnet" network is associated with the overlay driver and is scoped to the entire Swarm. NOTE: The other new networks (ingress and docker_gwbridge) were created automatically when the Swarm cluster was created.



```
[node1] (local) root@192.168.0.28 ~
$ docker network ls
NETWORK ID     NAME              DRIVER    SCOPE
4fdac416dd63   bridge            bridge    local
20bb680983cf   docker_gwbridge   bridge    local
cb452cfeda0d   host              host      local
twvjubhr2uov   ingress           overlay   swarm
ed8e6303fad3   none              null      local
kq2r76w6h3il   overnet           overlay   swarm
[node1] (local) root@192.168.0.28 ~
$ 
```

This command outputs networks available in the docker environment with following details tabulated.

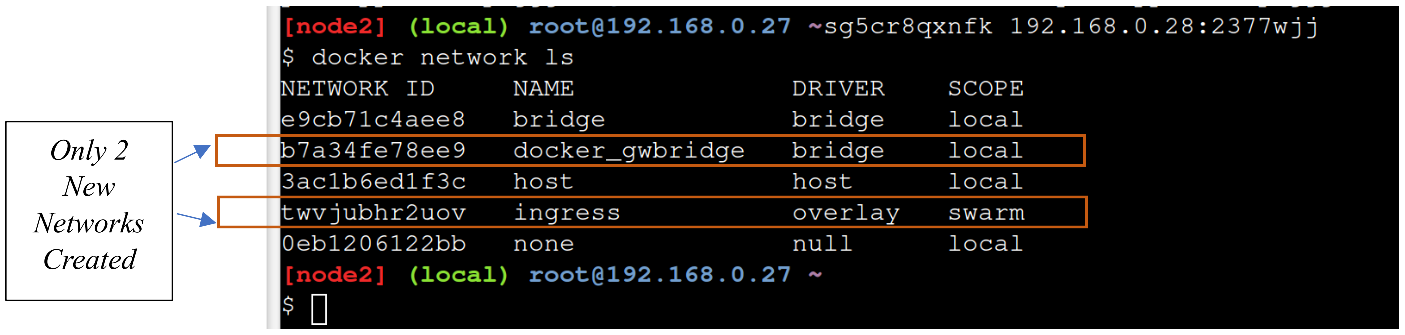
- *NETWORK ID: A unique identifier for each network created within Docker.*
- *NAME: The name of the network.*
- *DRIVER: The driver used by the network.*
- *SCOPE: The scope of the network, which indicates where it is available.*

The Newly Created 3 networks:

- *'overnet' Network: created from the 'create -d overlay overnet' command. Intended for communication between services across different nodes in the Docker Swarm. The scope is swarm, indicating it's available across all nodes in the swarm.*

- *'docker_gwbridge' and 'ingress' Networks were created when the swarm was initialized. The 'docker_gwbridge' network is used for communication between the docker container and the docker host. 'ingress' network is used for routing ingress traffic to services inside the swarm.*

2.3. Run the same "docker network ls" command from the second terminal.



```
[node2] (local) root@192.168.0.27 ~sg5cr8qxnfk 192.168.0.28:2377wjj
$ docker network ls
NETWORK ID     NAME              DRIVER    SCOPE
e9cb71c4aee8   bridge            bridge    local
b7a34fe78ee9   docker_gwbridge   bridge    local
3ac1b6ed1f3c   host              host      local
twvjubhr2uov   ingress           overlay   swarm
0eb1206122bb   none              null      local
[node2] (local) root@192.168.0.27 ~
$ 
```

These 2 new networks were created at the moment which the node joined the swarm. But since "node 2" is not currently running any task/service on the 'overnet' network it does not have 'overnet' under it's list of networks-connected.

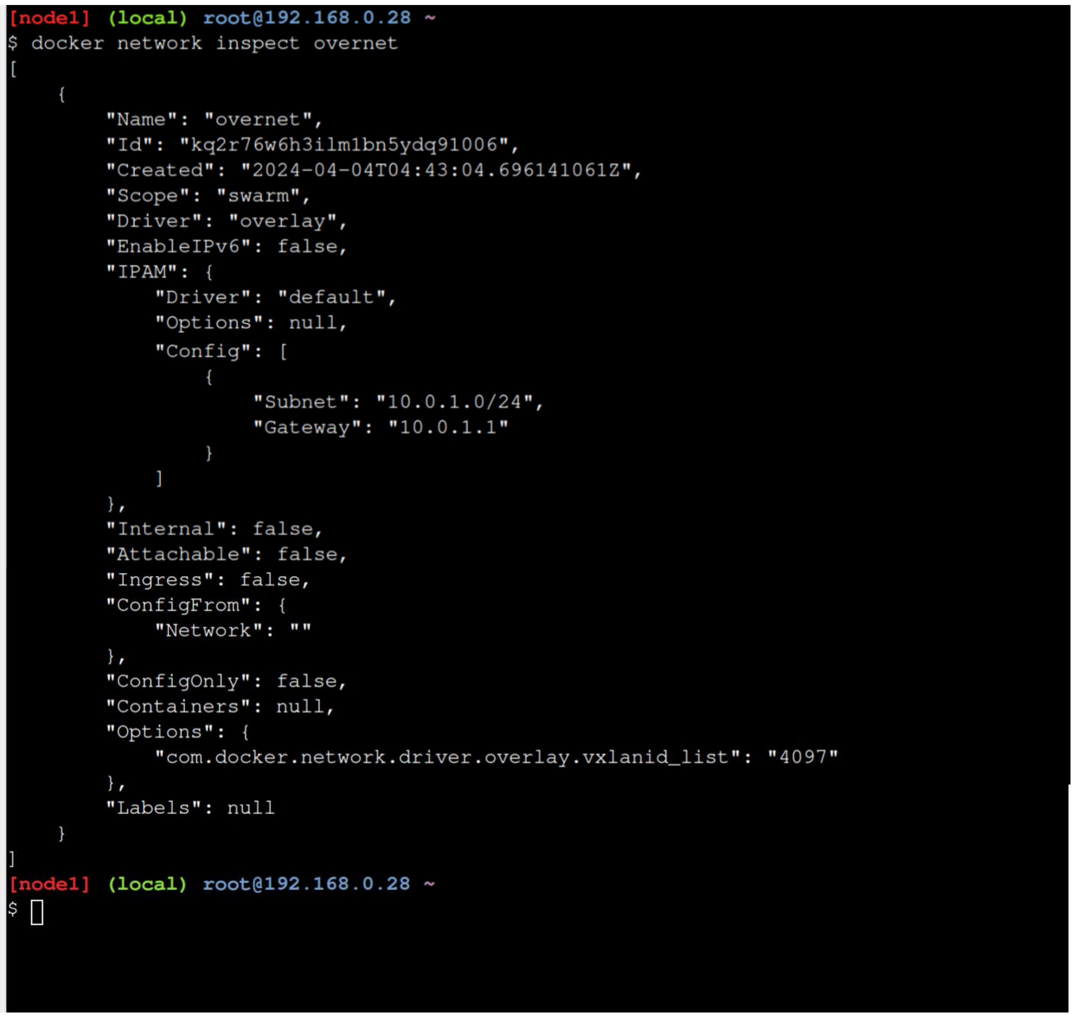
It can also be observed that the ingress network-id is the same as the network-id in manager node's ingress network. This is because, the ingress network is common to all the nodes which belong to a particular swarm network. Moreover, this network facilitates services deployed in the swarm to be accessed from outside the swarm.

2.4. Use the docker network inspect command to view more detailed information about the "overnet" network. Run this command from the first terminal

This command provides detailed information about the "overnet" network. It provided the following information:

- *Name: "overnet" - The name of the network.*
- *Id: "kq2r76w6h3ilm1bn5ydq91006" - The unique identifier of the network.*
- *Created: "2024-04-04T04:43:04.696141061Z" - The timestamp indicating when the network was created.*
- *Scope: "swarm" - The scope of the network. In this case, it's a swarm-scoped network.*
- *Driver: "overlay" - The driver used by the network. Overlay networks enable communication between services running on different nodes within the Docker Swarm.*
- *IPAM: This section provides information about IP address management for the network. It specifies the subnet and gateway for the network.*
- *Subnet: "10.0.1.0/24" - The subnet range used for assigning IP addresses to containers within the network.*
- *Gateway: "10.0.1.1" - The gateway IP address for the network.*
- *Internal: false - Indicates whether the network is an internal network. In this case, it's not an internal network.*
- *Attachable: false - Indicates whether containers outside the swarm can attach to this network. In this case, it's not attachable.*

- *Ingress: false - Indicates whether this network is the ingress network. Ingress networks are used for routing external traffic into the swarm. In this case, it's not the ingress network.*
- *Options: This section provides additional options set for the network. In this case, it specifies the VXLAN ID list for the overlay driver.*
- *Containers: null - This field would list the containers connected to the network, but it's null since there are no containers connected to the network at the moment.*
- *Labels: null - This field would contain any user-defined labels assigned to the network, but it's null, indicating no labels are assigned.*

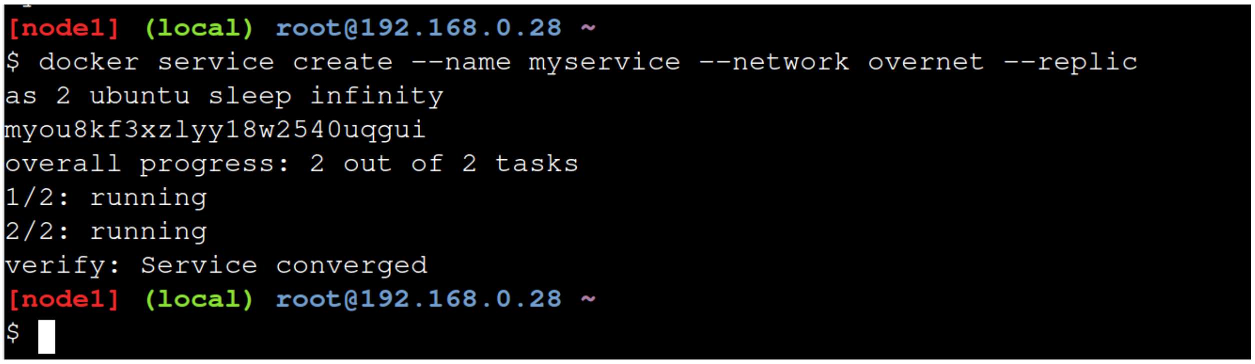


```
[node1] (local) root@192.168.0.28 ~
$ docker network inspect overnet
[
    {
        "Name": "overnet",
        "Id": "kq2r76w6h3ilm1bn5ydq91006",
        "Created": "2024-04-04T04:43:04.696141061Z",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.0.1.0/24",
                    "Gateway": "10.0.1.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": null,
        "Options": {
            "com.docker.network.driver.overlay.vxlanid_list": "4097"
        },
        "Labels": null
    }
]
[node1] (local) root@192.168.0.28 ~
$ 
```

Figure 2: Output of 'docker network inspect overnet' (regarding part 2.4)

3. Create a service. Now that we have a Swarm initialized and an overlay network, it's time to create a service that uses the network.

3.1. Execute the following command from the first terminal to create a new service called myservice on the overnet network with two tasks/replicas. "docker service create --name myservice --network overnet --replicas 2 ubuntu sleep infinity"



```
[node1] (local) root@192.168.0.28 ~
$ docker service create --name myservice --network overnet --replic
as 2 ubuntu sleep infinity
myou8kf3xzlyy18w2540uqgui
overall progress: 2 out of 2 tasks
1/2: running
2/2: running
verify: Service converged
[node1] (local) root@192.168.0.28 ~
$
```

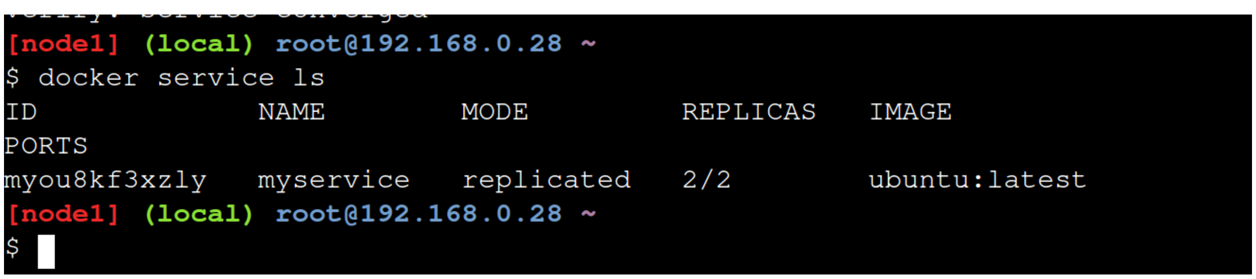
This command shows that both tasks (replicas) of the service have been created and are running. Finally, it verifies that the service has converged, meaning it has reached the desired state (2 replicas running) and is stable.

The command breakdown:

- *--name myservice: Specifies the name of the service as "myservice".*
- *--network overnet: Specifies that the service should be connected to the "overnet" network.*
- *--replicas 2: Specifies that the service should have 2 replicas, meaning two instances of the service will be created.*
- *ubuntu: Specifies the Docker image to use for the service, which in this case is the official Ubuntu image*
- *sleep infinity: Specifies that "sleep infinity" command runs when the container starts, which keeps the container running indefinitely.*

The output myou8kf3xzlyy18w2540uqgui is the ID of the newly created service.

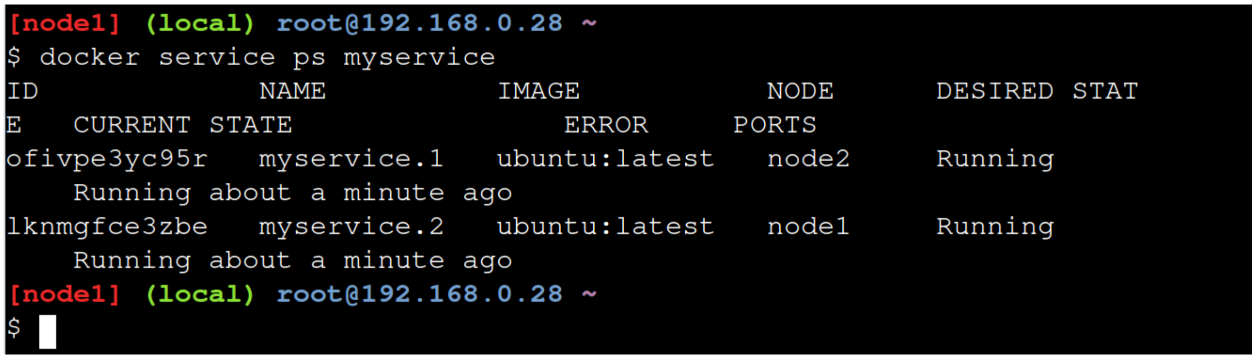
3.2. Verify that the service is created and both replicas are up by running "docker service ls". Examine the output.



```
[node1] (local) root@192.168.0.28 ~
$ docker service ls
ID             NAME        MODE         REPLICAS   IMAGE
PORTS
myou8kf3xzly   myservice   replicated   2/2        ubuntu:latest
[node1] (local) root@192.168.0.28 ~
$
```

This shows that a service named "myservice" has ID "myou8kf3xzly" and is running in "replicated" mode with 2 replicas, as indicated by "2/2" under REPLICAS. Both use ubuntu:latest image.

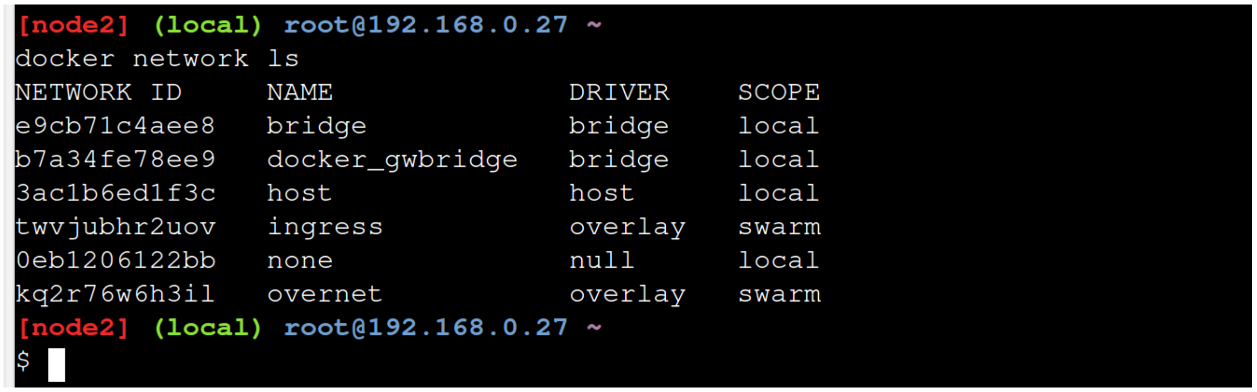
3.3. Verify that a single task (replica) is running on each of the two nodes in the Swarm by running "docker service ps myservice".



```
[node1] (local) root@192.168.0.28 ~
$ docker service ps myservice
ID              NAME            IMAGE           NODE       DESIRED STAT
E    CURRENT STATE                  ERROR      PORTS
ofivpe3yc95r    myservice.1     ubuntu:latest   node2      Running
     Running about a minute ago
lknmgfce3zbe    myservice.2     ubuntu:latest   node1      Running
     Running about a minute ago
[node1] (local) root@192.168.0.28 ~
$ 
```

This command lists the tasks/replicas associated with the specified Docker service. And this states that both the replicas are running without errors.

3.4. Now that the second node is running a task on the "overnet" network it will be able to see the "overnet" network. Lets run docker network ls from the second terminal to verify this.



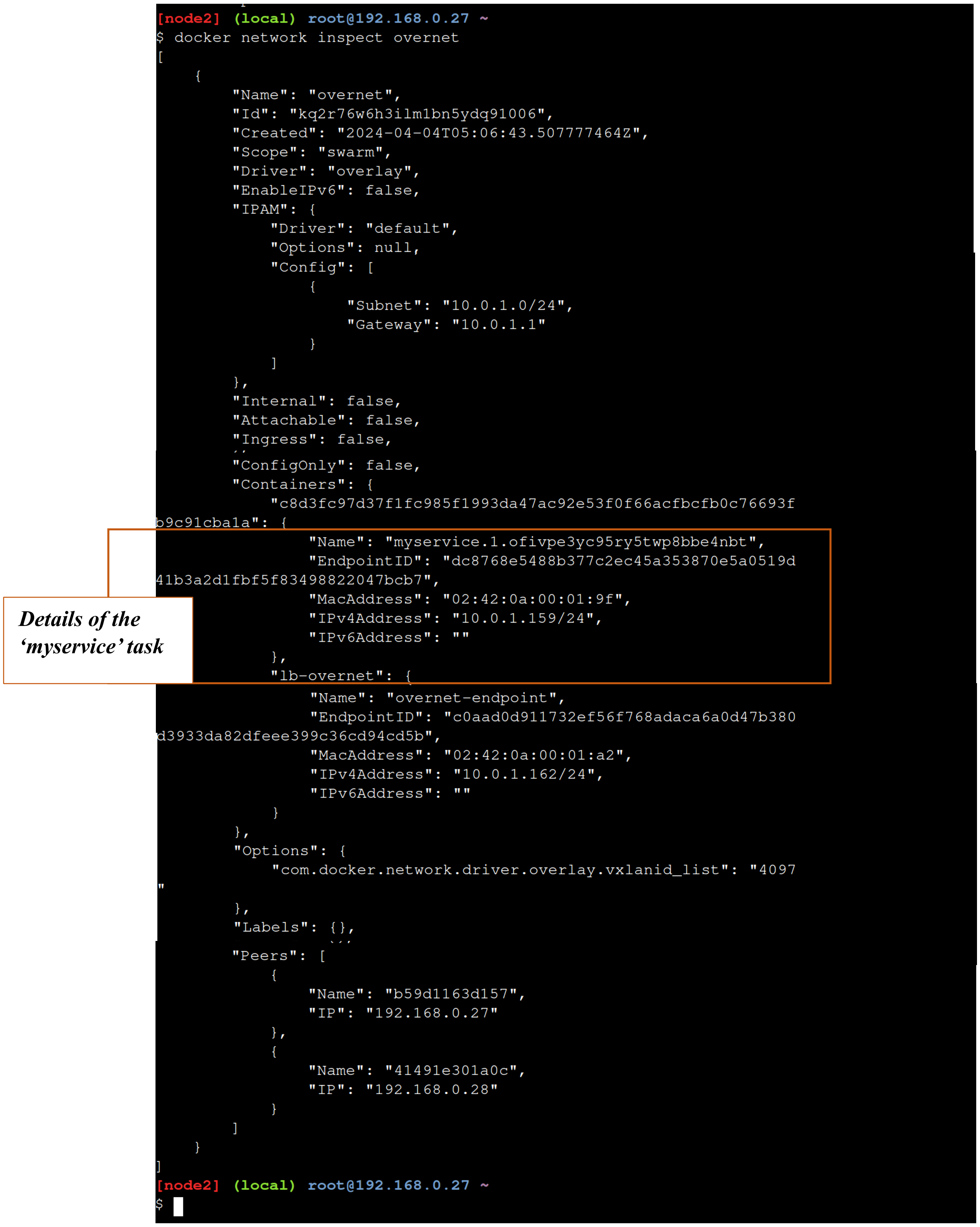
```
[node2] (local) root@192.168.0.27 ~
docker network ls
NETWORK ID      NAME              DRIVER    SCOPE
e9cb71c4aee8    bridge            bridge    local
b7a34fe78ee9    docker_gwbridge   bridge    local
3ac1b6ed1f3c    host              host      local
twvjubhr2uov    ingress           overlay   swarm
0eb1206122bb    none              null      local
kq2r76w6h3il    overnet           overlay   swarm
[node2] (local) root@192.168.0.27 ~
$ 
```

The "overnet" network is now displaying in the output of "docker network ls" on node2 because a task associated with a service running on node2 has been connected to the "overnet" network.

3.5. Use docker network inspect overnet on the second terminal to get more detailed information about the "overnet" network and obtain the IP address of the task running on the second terminal.

According to the output of this command the obtained details of the 'myservice' task (IP address, MAC address) are highlighted below.

- *IP Address of the task: 10.0.1.159/24*

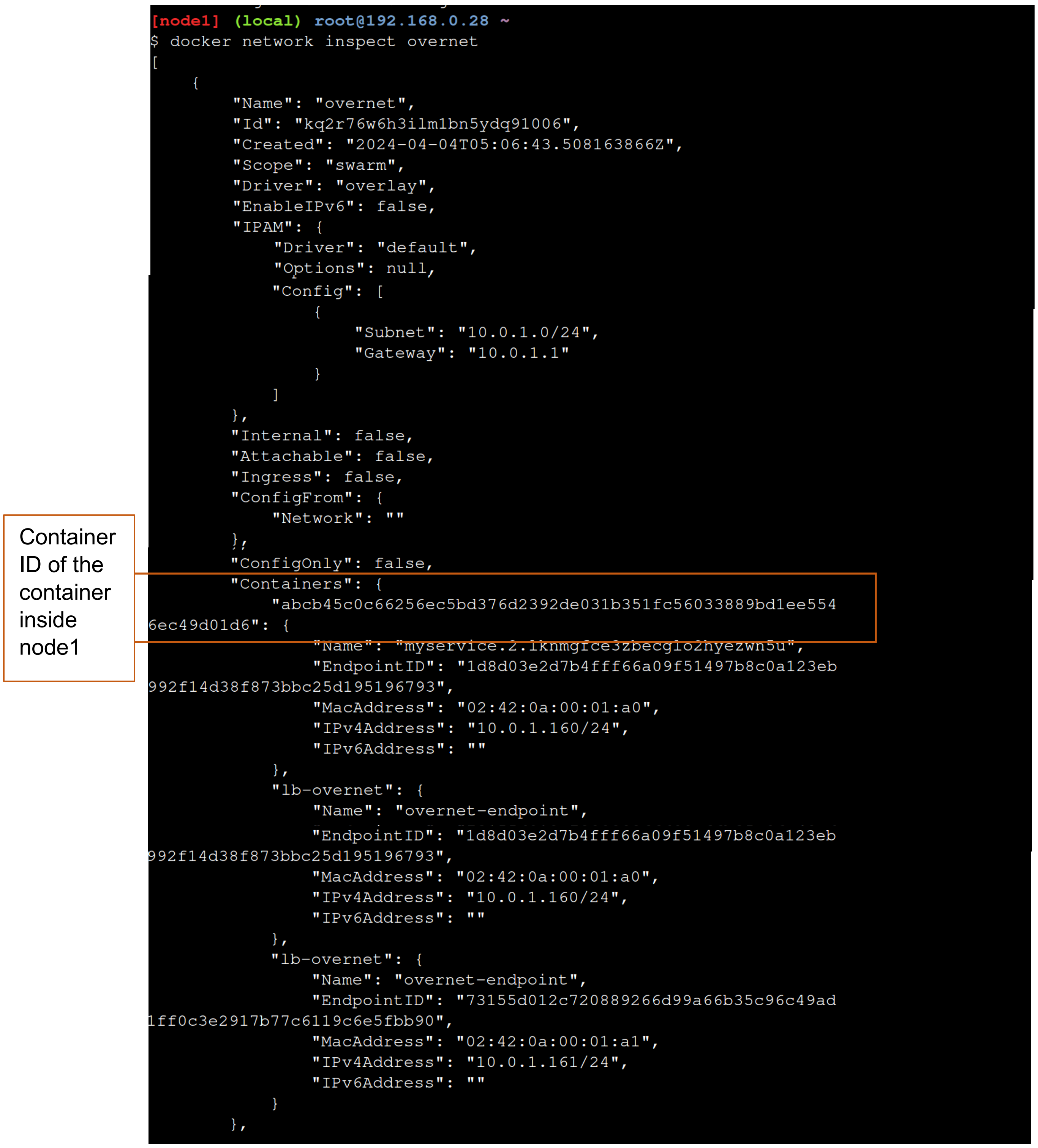


```
[node2] (local) root@192.168.0.27 ~
$ docker network inspect overnet
[
    {
        "Name": "overnet",
        "Id": "kq2r76w6h3ilm1bn5ydq91006",
        "Created": "2024-04-04T05:06:43.507777464Z",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.0.1.0/24",
                    "Gateway": "10.0.1.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigOnly": false,
        "Containers": {
            "c8d3fc97d37f1fc985f1993da47ac92e53f0f66acfbcfb0c76693f
b9c91cba1a": {
                "Name": "myservice.1.ofivpe3yc95ry5twp8bbe4nbt",
                "EndpointID": "dc8768e5488b377c2ec45a353870e5a0519d
41b3a2d1fbf5f83498822047bcb7",
                "MacAddress": "02:42:0a:00:01:9f",
                "IPv4Address": "10.0.1.159/24",
                "IPv6Address": ""
            },
            "lb-overnet": {
                "Name": "overnet-endpoint",
                "EndpointID": "c0aad0d911732ef56f768adaca6a0d47b380
d3933da82dfeee399c36cd94cd5b",
                "MacAddress": "02:42:0a:00:01:a2",
                "IPv4Address": "10.0.1.162/24",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.driver.overlay.vxlanid_list": "4097
"
        },
        "Labels": {},
        "Peers": [
            {
                "Name": "b59d1163d157",
                "IP": "192.168.0.27"
            },
            {
                "Name": "41491e301a0c",
                "IP": "192.168.0.28"
            }
        ]
    }
]
[node2] (local) root@192.168.0.27 ~
$
```

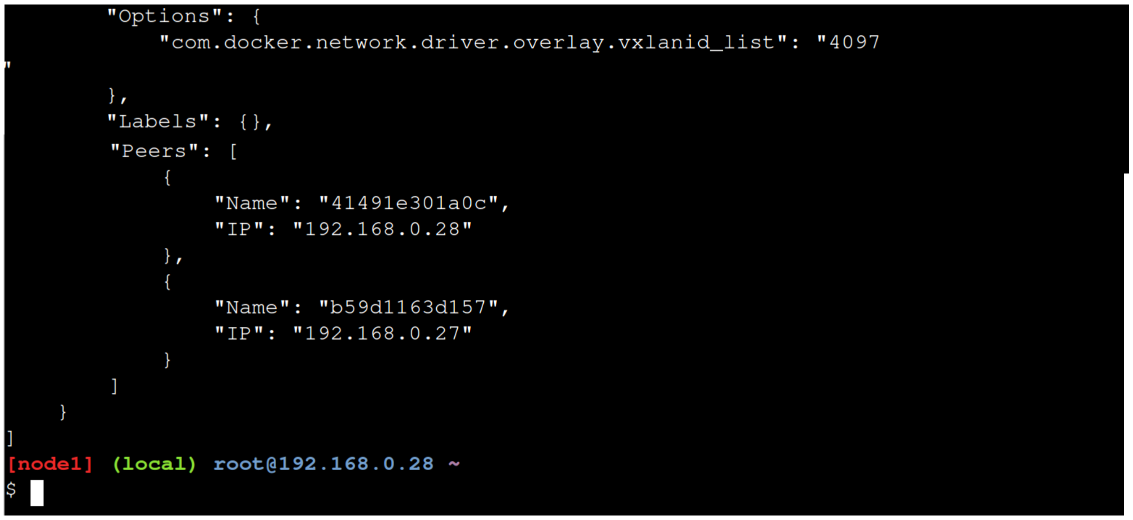
Figure 3: Output of 'docker network inspect overnet' command on terminal-2

4. Test the network. To complete this step you will need the IP address of the service task running on node2 that you saw in the previous step

4.1. Execute the "docker network inspect overnet" command from the first terminal.



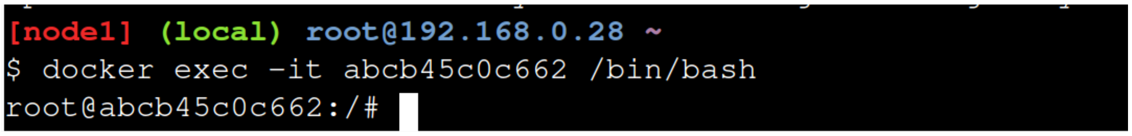
```
[node1] (local) root@192.168.0.28 ~
$ docker network inspect overnet
[
    {
        "Name": "overnet",
        "Id": "kq2r76w6h3ilm1bn5ydq91006",
        "Created": "2024-04-04T05:06:43.508163866Z",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.0.1.0/24",
                    "Gateway": "10.0.1.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "abcb45c0c66256ec5bd376d2392de031b351fc56033889bd1ee554
6ec49d01d6": {
                "Name": "myservice.2.1knmgfce3zbecglo2hyezwn5u",
                "EndpointID": "1d8d03e2d7b4fff66a09f51497b8c0a123eb
992f14d38f873bbc25d195196793",
                "MacAddress": "02:42:0a:00:01:a0",
                "IPv4Address": "10.0.1.160/24",
                "IPv6Address": ""
            },
            "lb-overnet": {
                "Name": "overnet-endpoint",
                "EndpointID": "1d8d03e2d7b4fff66a09f51497b8c0a123eb
992f14d38f873bbc25d195196793",
                "MacAddress": "02:42:0a:00:01:a0",
                "IPv4Address": "10.0.1.160/24",
                "IPv6Address": ""
            },
            "lb-overnet": {
                "Name": "overnet-endpoint",
                "EndpointID": "73155d012c720889266d99a66b35c96c49ad
1ff0c3e2917b77c6119c6e5fbb90",
                "MacAddress": "02:42:0a:00:01:a1",
                "IPv4Address": "10.0.1.161/24",
                "IPv6Address": ""
            }
        },
```



```
        "Options": {
            "com.docker.network.driver.overlay.vxlanid_list": "4097
"
        },
        "Labels": {},
        "Peers": [
            {
                "Name": "41491e301a0c",
                "IP": "192.168.0.28"
            },
            {
                "Name": "b59d1163d157",
                "IP": "192.168.0.27"
            }
        ]
    }
]
[node1] (local) root@192.168.0.28 ~
$
```

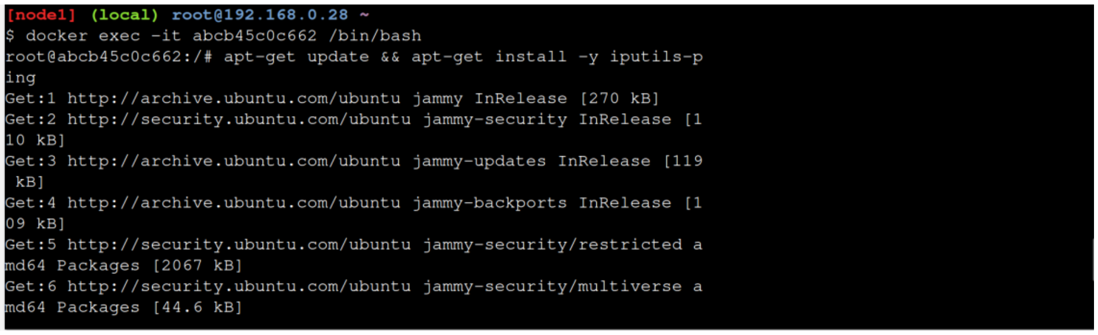
Figure 4: Output Received at node1 for the command "docker network inspect overnet"

4.2. Log on to the service task. Run “docker exec -it /bin/bash”



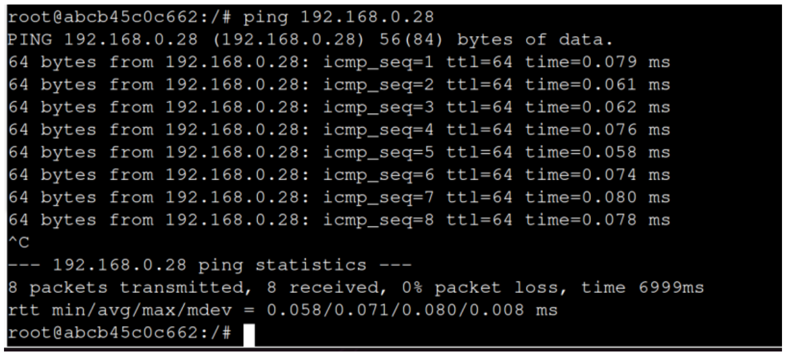
```
[node1] (local) root@192.168.0.28 ~
$ docker exec -it abcb45c0c662 /bin/bash
root@abcb45c0c662:/#
```

4.3. Install the ping command and ping the service task running on the second node used in step 3 above from the “docker network inspect overnet” command.



```
[node1] (local) root@192.168.0.28 ~
$ docker exec -it abcb45c0c662 /bin/bash
root@abcb45c0c662:/# apt-get update && apt-get install -y iputils-p
ing
Get:1 http://archive.ubuntu.com/ubuntu jammy InRelease [270 kB]
Get:2 http://security.ubuntu.com/ubuntu jammy-security InRelease [1
10 kB]
Get:3 http://archive.ubuntu.com/ubuntu jammy-updates InRelease [119
 kB]
Get:4 http://archive.ubuntu.com/ubuntu jammy-backports InRelease [1
09 kB]
Get:5 http://security.ubuntu.com/ubuntu jammy-security/restricted a
md64 Packages [2067 kB]
Get:6 http://security.ubuntu.com/ubuntu jammy-security/multiverse a
md64 Packages [44.6 kB]
```

4.4. Ping to the ip address obtained in step 3 above

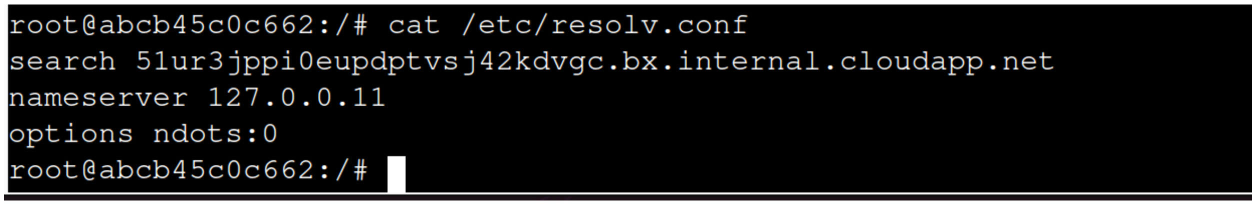


```
root@abcb45c0c662:/# ping 192.168.0.28
PING 192.168.0.28 (192.168.0.28) 56(84) bytes of data.
64 bytes from 192.168.0.28: icmp_seq=1 ttl=64 time=0.079 ms
64 bytes from 192.168.0.28: icmp_seq=2 ttl=64 time=0.061 ms
64 bytes from 192.168.0.28: icmp_seq=3 ttl=64 time=0.062 ms
64 bytes from 192.168.0.28: icmp_seq=4 ttl=64 time=0.076 ms
64 bytes from 192.168.0.28: icmp_seq=5 ttl=64 time=0.058 ms
64 bytes from 192.168.0.28: icmp_seq=6 ttl=64 time=0.074 ms
64 bytes from 192.168.0.28: icmp_seq=7 ttl=64 time=0.080 ms
64 bytes from 192.168.0.28: icmp_seq=8 ttl=64 time=0.078 ms
^C
--- 192.168.0.28 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 6999ms
rtt min/avg/max/mdev = 0.058/0.071/0.080/0.008 ms
root@abcb45c0c662:/#
```

Hence, it can be observed that the ‘node1’ is connected to the service task running on the overnet network.

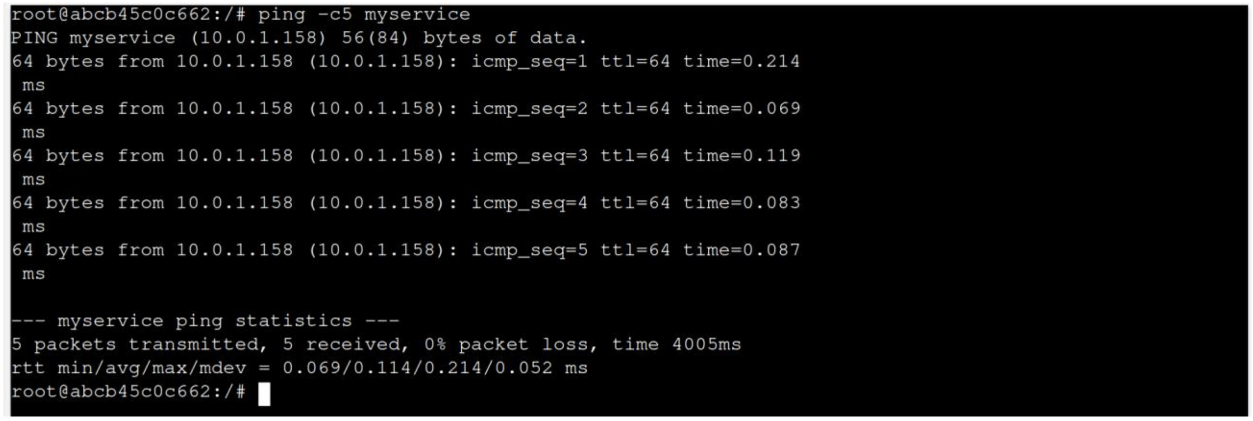
5. Test service discovery

5.1. Run 'cat /etc/resolv.conf' from inside of the container.



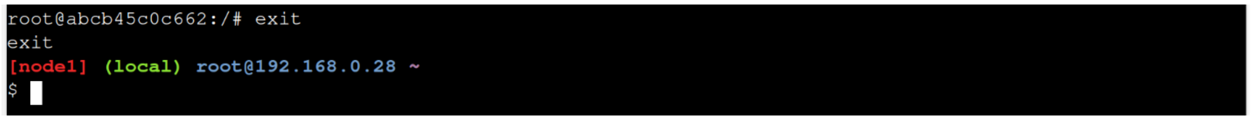
```
root@abcb45c0c662:/# cat /etc/resolv.conf
search 51ur3jppi0eupdptvsj42kdvgc.bx.internal.cloudapp.net
nameserver 127.0.0.11
options ndots:0
root@abcb45c0c662:/#
```

5.2. Try and ping the "myservice" name from within the container by running "ping -c5 myservice" Obtain the returning ip address. Next let's verify that this address is the virtual IP (VIP) assigned to the myservice service.



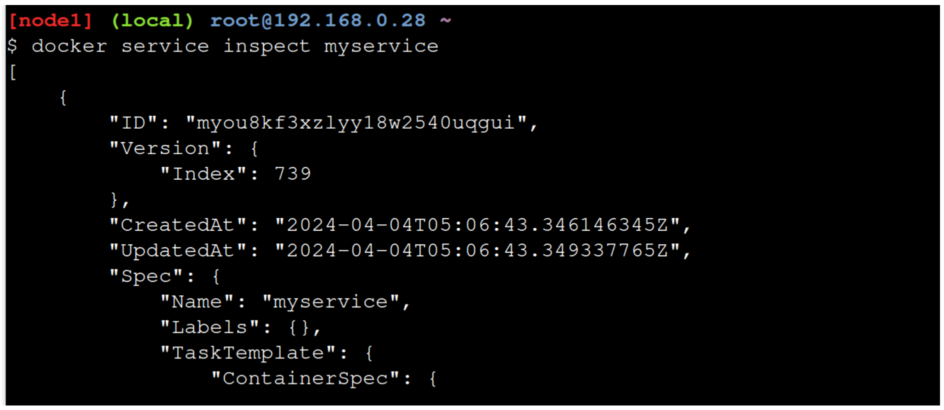
```
root@abcb45c0c662:/# ping -c5 myservice
PING myservice (10.0.1.158) 56(84) bytes of data.
64 bytes from 10.0.1.158 (10.0.1.158): icmp_seq=1 ttl=64 time=0.214
 ms
64 bytes from 10.0.1.158 (10.0.1.158): icmp_seq=2 ttl=64 time=0.069
 ms
64 bytes from 10.0.1.158 (10.0.1.158): icmp_seq=3 ttl=64 time=0.119
 ms
64 bytes from 10.0.1.158 (10.0.1.158): icmp_seq=4 ttl=64 time=0.083
 ms
64 bytes from 10.0.1.158 (10.0.1.158): icmp_seq=5 ttl=64 time=0.087
 ms

--- myservice ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 0.069/0.114/0.214/0.052 ms
root@abcb45c0c662:/#
```

5.3. Type the exit command to leave the exec container session and return to the shell prompt of your Docker host.



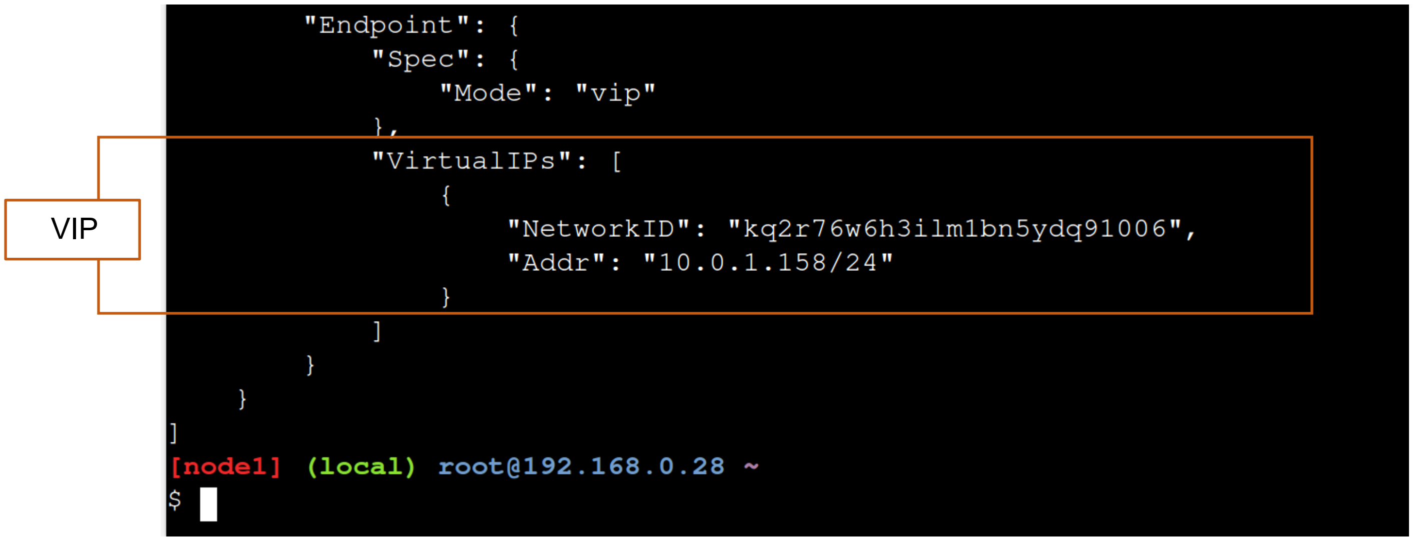
```
root@abcb45c0c662:/# exit
exit
[node1] (local) root@192.168.0.28 ~
$
```

5.4. Inspect the configuration of the "myservice" service by running "docker service inspect myservice". Lets verify that the VIP value matches the value returned by the previous ping -c5 myservice command.



```
[node1] (local) root@192.168.0.28 ~
$ docker service inspect myservice
[
    {
        "ID": "myou8kf3xzlyy18w2540uqgui",
        "Version": {
            "Index": 739
        },
        "CreatedAt": "2024-04-04T05:06:43.346146345Z",
        "UpdatedAt": "2024-04-04T05:06:43.349337765Z",
        "Spec": {
            "Name": "myservice",
            "Labels": {},
            "TaskTemplate": {
                "ContainerSpec": {
```

[Truncated Output]



```
        "Endpoint": {
            "Spec": {
                "Mode": "vip"
            },
            "VirtualIPs": [
                {
                    "NetworkID": "kq2r76w6h3ilm1bn5ydq91006",
                    "Addr": "10.0.1.158/24"
                }
            ]
        }
    }
]
[node1] (local) root@192.168.0.28 ~
$
```

5.5. Does the VIP listed above match the value returned by the ping -c5 myservice command.?

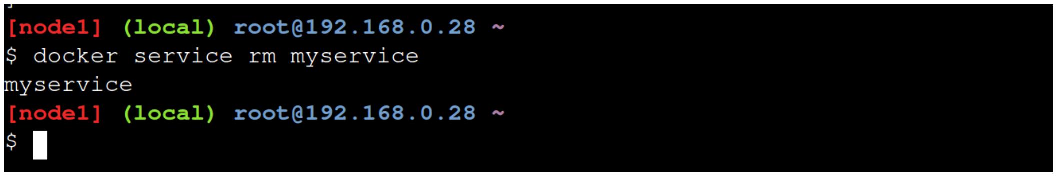
The IP value retuned by the previous ping 'myservice' command ***: 10.0.1.158***

The VIP value in the inspect 'myservice' command output: ***10.0.1.158***

Therefore, it can be observed that both the IPs refer to the same.

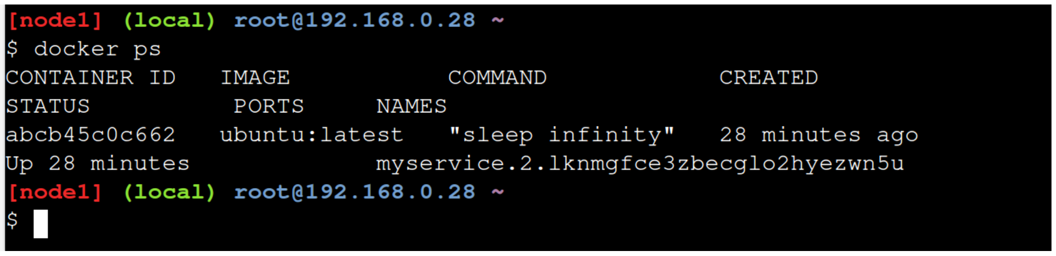
Cleaning Up

1. Execute the 'docker service rm myservice' command to remove the service called myservice.



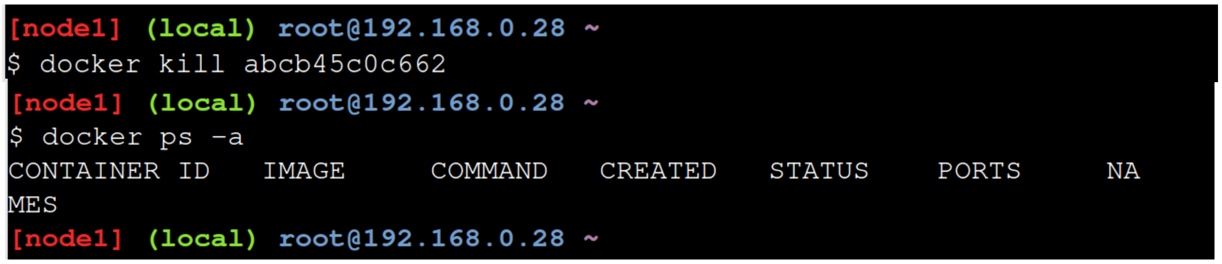
```
[node1] (local) root@192.168.0.28 ~
$ docker service rm myservice
myservice
[node1] (local) root@192.168.0.28 ~
$
```

2. Execute the 'docker ps' command to get a list of running containers.



```
[node1] (local) root@192.168.0.28 ~
$ docker ps
CONTAINER ID   IMAGE           COMMAND              CREATED
STATUS          PORTS      NAMES
abcb45c0c662   ubuntu:latest   "sleep infinity"     28 minutes ago
Up 28 minutes              myservice.2.lknmgfce3zbecglo2hyezwn5u
[node1] (local) root@192.168.0.28 ~
$
```

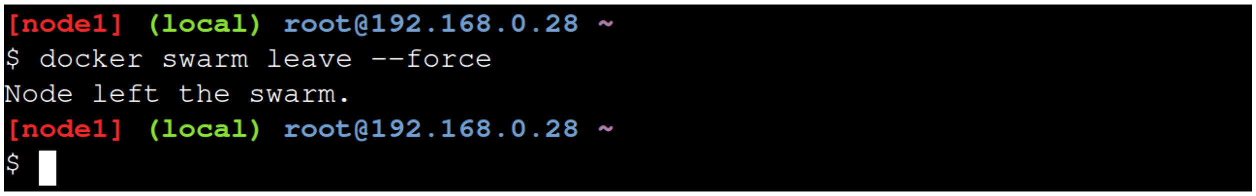
3. You can use the 'docker kill' command to kill the ubuntu and nginx containers we started at the beginning.



```
[node1] (local) root@192.168.0.28 ~
$ docker kill abcb45c0c662
[node1] (local) root@192.168.0.28 ~
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND    CREATED    STATUS     PORTS     NA
MES
[node1] (local) root@192.168.0.28 ~
```

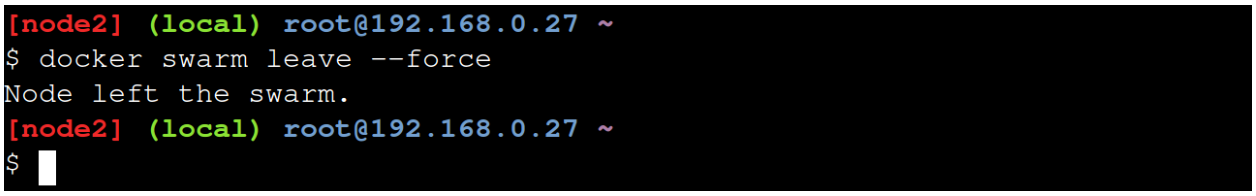
The docker container has been successfully deleted.

4. Run docker 'swarm leave –force' on node1.



```
[node1] (local) root@192.168.0.28 ~
$ docker swarm leave --force
Node left the swarm.
[node1] (local) root@192.168.0.28 ~
$
```

5. Run docker 'swarm leave –force' on node2



```
[node2] (local) root@192.168.0.27 ~
$ docker swarm leave --force
Node left the swarm.
[node2] (local) root@192.168.0.27 ~
$
```