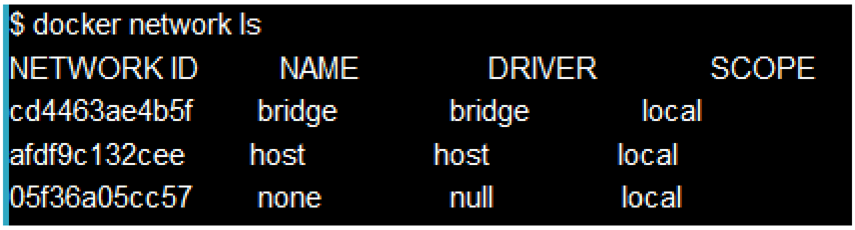
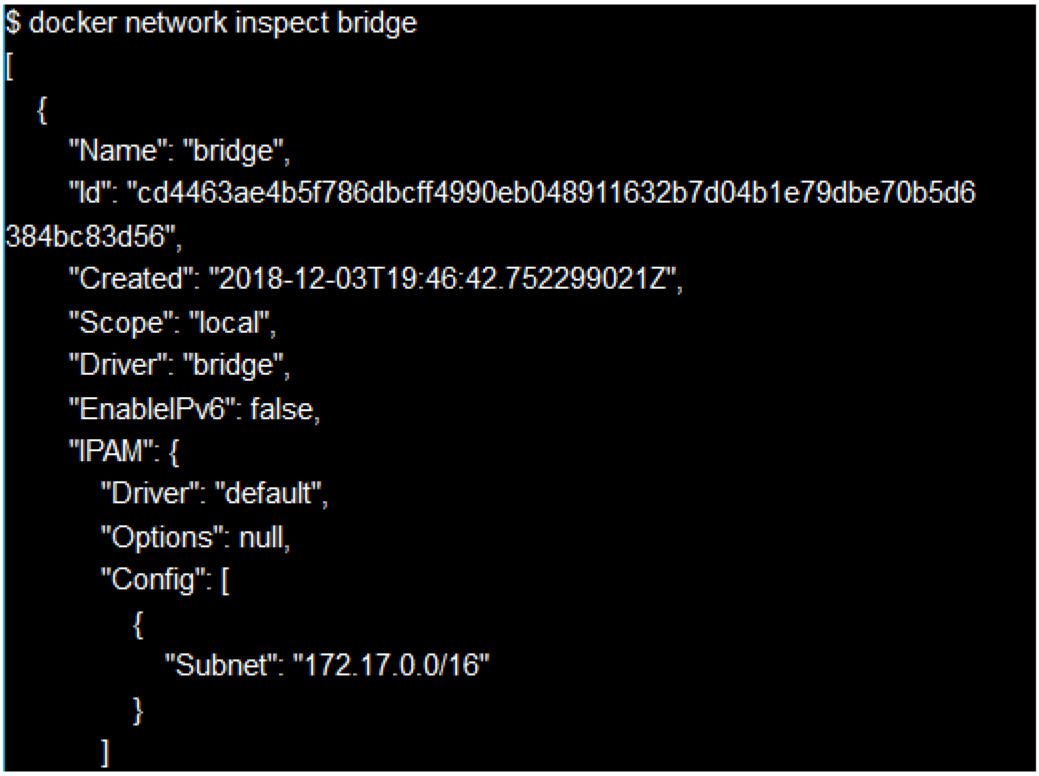
Sprawozdanie

Piotr Bogdanowicz

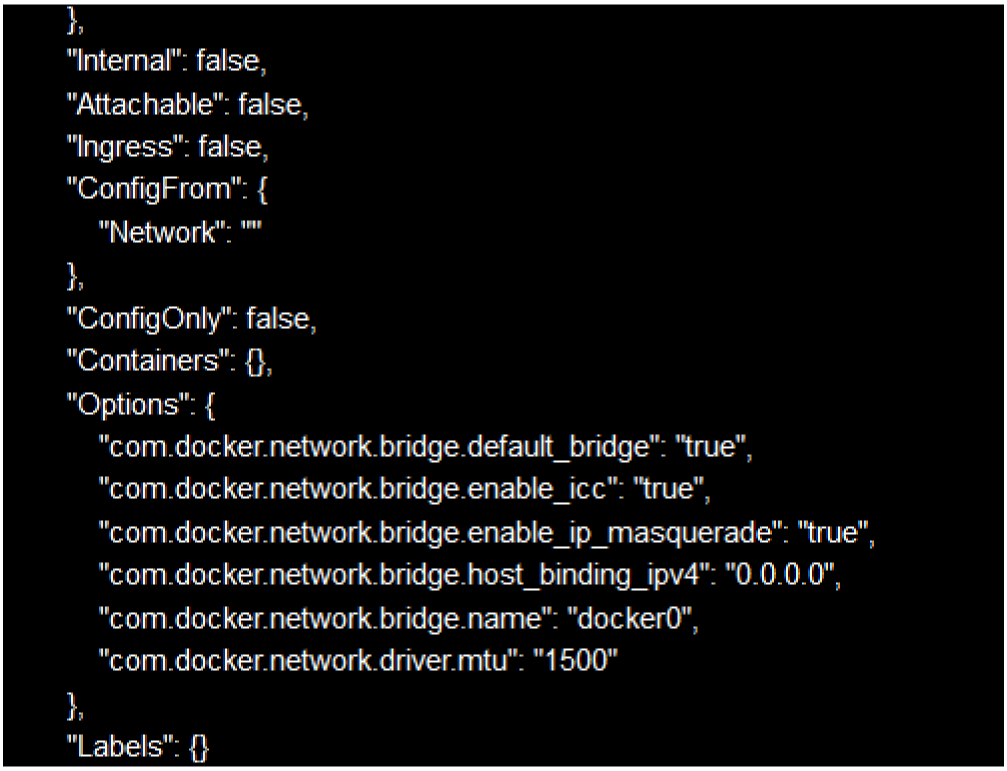
Section #1



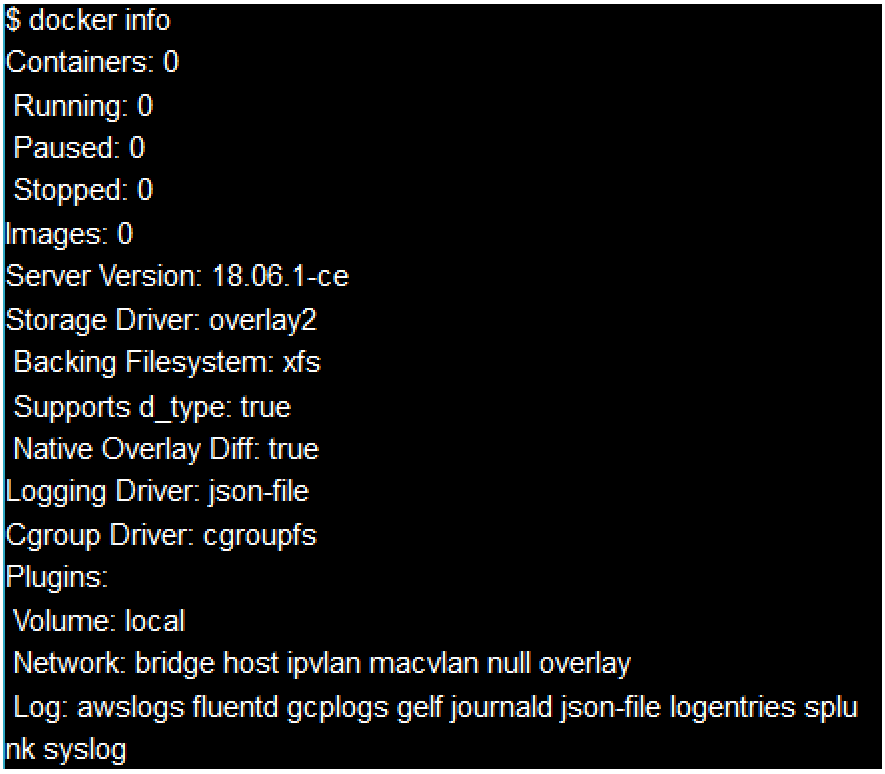
```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
cd4463ae4b5f        bridge              bridge              local
afdf9c132cee        host                host                local
05f36a05cc57        none                null                local
```



```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "cd4463ae4b5f786dbcff4990eb048911632b7d04b1e79dbe70b5d6
384bc83d56",
        "Created": "2018-12-03T19:46:42.752299021Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16"
                }
            ]
```

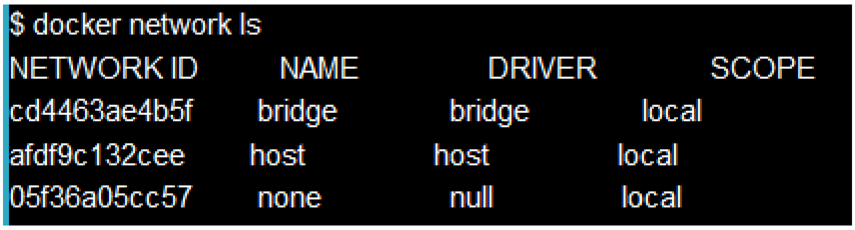


```
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
```

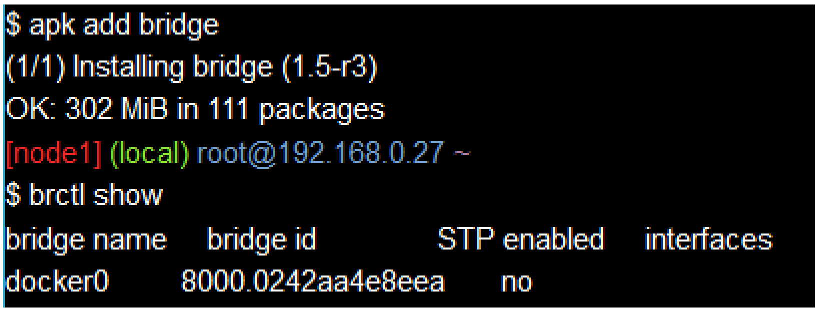


```
$ docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 18.06.1-ce
Storage Driver: overlay2
 Backing Filesystem: xfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host ipvlan macvlan null overlay
 Log: awslogs fluentd gcplogs gelf journald json-file logentries splu
nk syslog
```

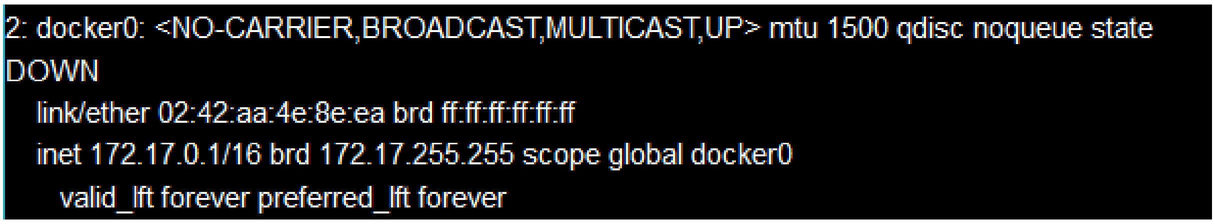
Section #2



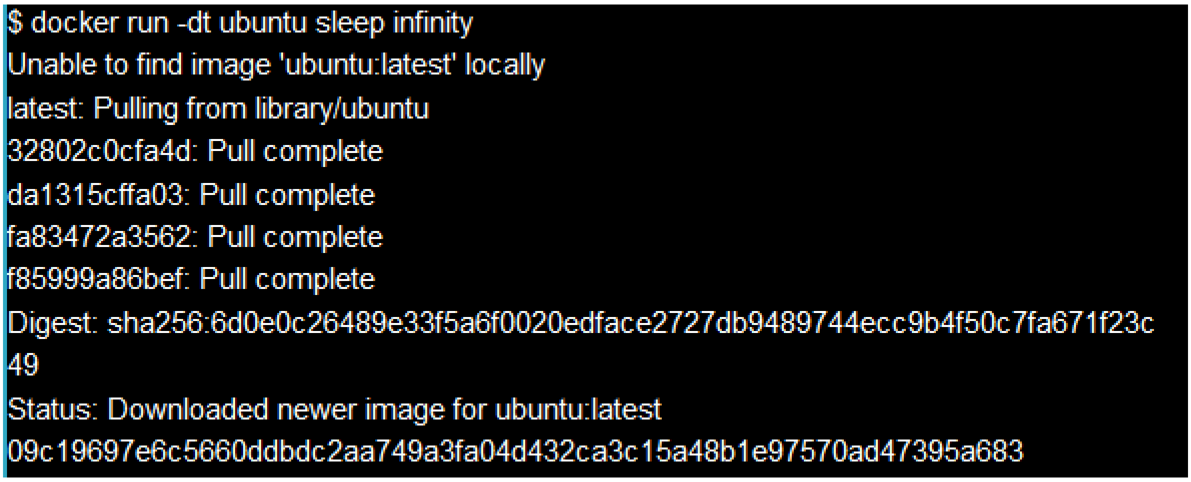
```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
cd4463ae4b5f        bridge              bridge              local
afdf9c132cee        host                host                local
05f36a05cc57        none                null                local
```



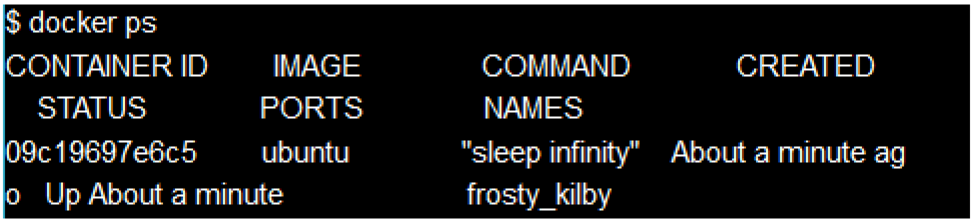
```
$ apk add bridge
(1/1) Installing bridge (1.5-r3)
OK: 302 MiB in 111 packages
[node1] (local) root@192.168.0.27 ~
$ brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242aa4e8eea       no
```



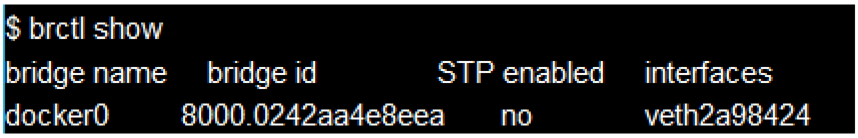
```
2: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
DOWN
    link/ether 02:42:aa:4e:8e:ea brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
```



```
$ docker run -dt ubuntu sleep infinity
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
32802c0cfa4d: Pull complete
da1315cffa03: Pull complete
fa83472a3562: Pull complete
f85999a86bef: Pull complete
Digest: sha256:6d0e0c26489e33f5a6f0020edface2727db9489744ecc9b4f50c7fa671f23c
49
Status: Downloaded newer image for ubuntu:latest
09c19697e6c5660ddbdc2aa749a3fa04d432ca3c15a48b1e97570ad47395a683
```

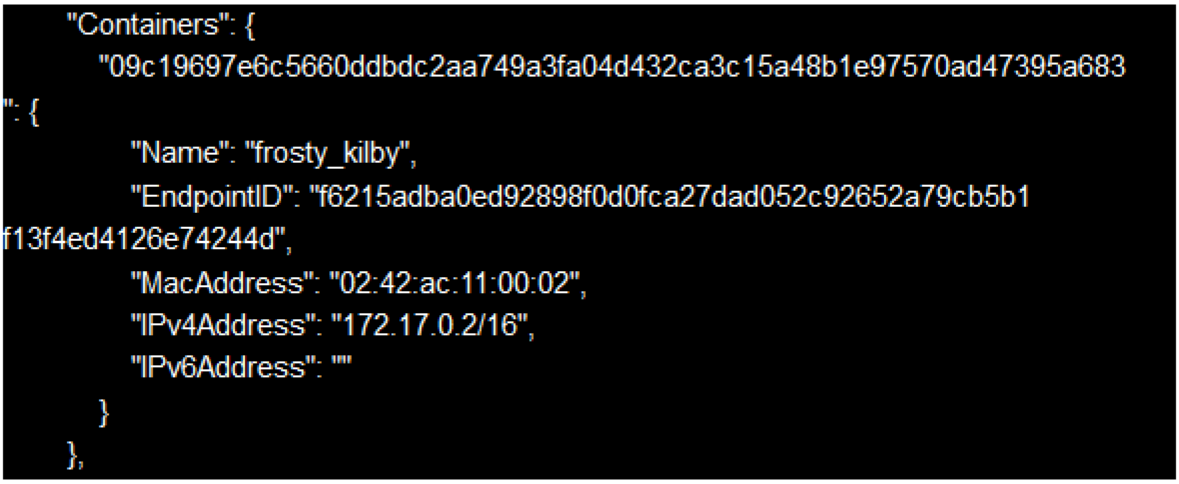


```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
    STATUS              PORTS               NAMES
09c19697e6c5        ubuntu              "sleep infinity"    About a minute ag
o   Up About a minute                       frosty_kilby
```

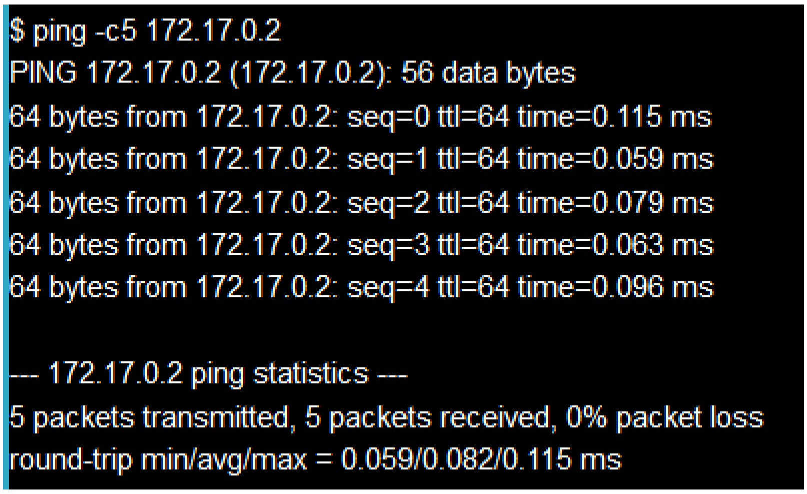


```
$ brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242aa4e8eea       no              veth2a98424
```

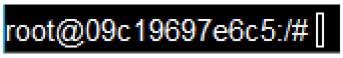
For docker0:



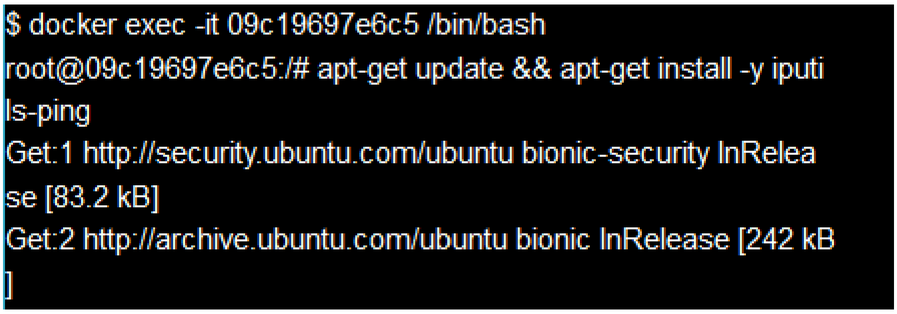
```
    "Containers": {
        "09c19697e6c5660ddbdc2aa749a3fa04d432ca3c15a48b1e97570ad47395a683
": {
            "Name": "frosty_kilby",
            "EndpointID": "f6215adba0ed92898f0d0fca27dad052c92652a79cb5b1
f13f4ed4126e74244d",
            "MacAddress": "02:42:ac:11:00:02",
            "IPv4Address": "172.17.0.2/16",
            "IPv6Address": ""
        }
    },
```



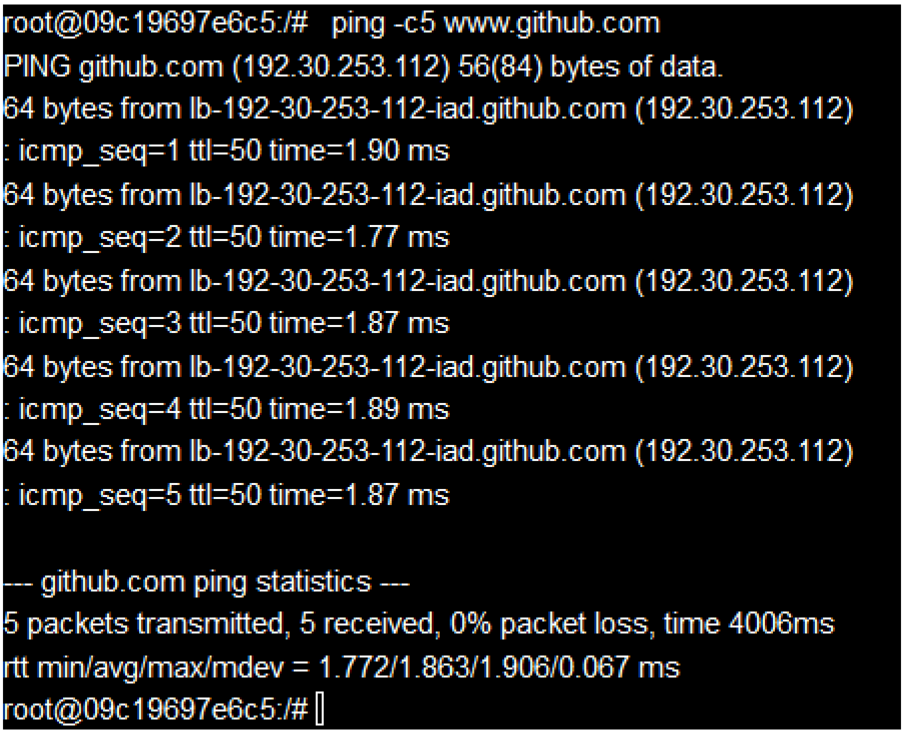
```
$ ping -c5 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.115 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.059 ms
64 bytes from 172.17.0.2: seq=2 ttl=64 time=0.079 ms
64 bytes from 172.17.0.2: seq=3 ttl=64 time=0.063 ms
64 bytes from 172.17.0.2: seq=4 ttl=64 time=0.096 ms

--- 172.17.0.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.059/0.082/0.115 ms
```



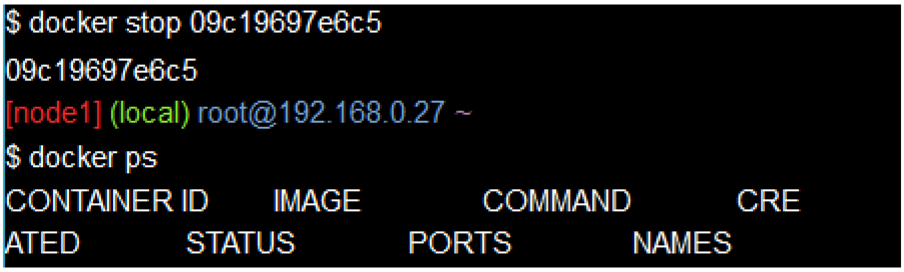
```
root@09c19697e6c5:/#
```



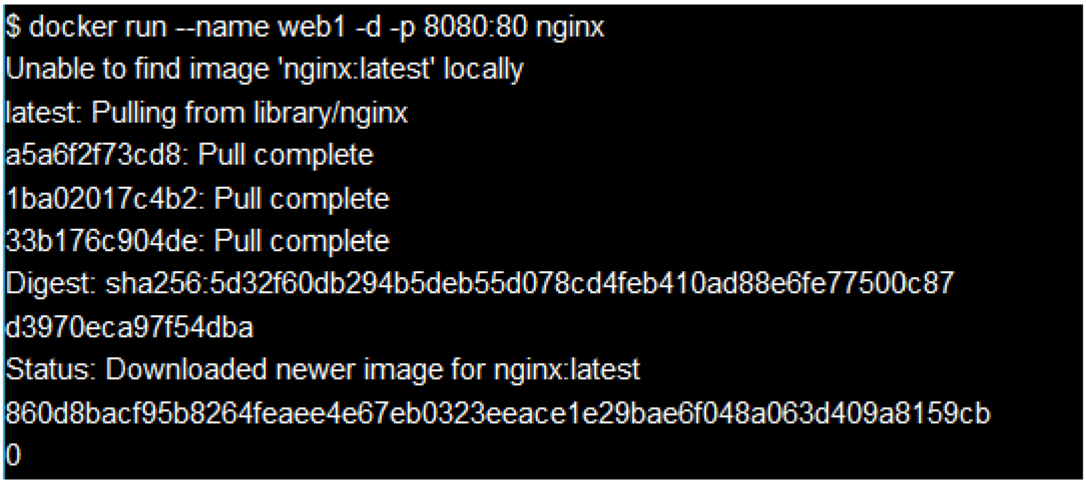
```
$ docker exec -it 09c19697e6c5 /bin/bash
root@09c19697e6c5:/# apt-get update && apt-get install -y iputi
ls-ping
Get:1 http://security.ubuntu.com/ubuntu bionic-security InRelea
se [83.2 kB]
Get:2 http://archive.ubuntu.com/ubuntu bionic InRelease [242 kB
]
```



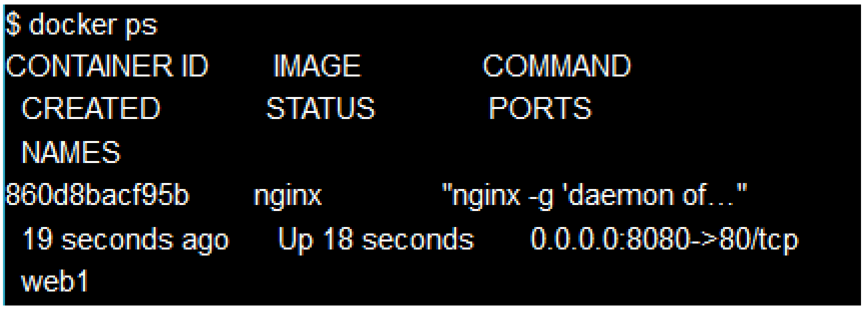
```
root@09c19697e6c5:/#   ping -c5 www.github.com
PING github.com (192.30.253.112) 56(84) bytes of data.
64 bytes from lb-192-30-253-112-iad.github.com (192.30.253.112)
: icmp_seq=1 ttl=50 time=1.90 ms
64 bytes from lb-192-30-253-112-iad.github.com (192.30.253.112)
: icmp_seq=2 ttl=50 time=1.77 ms
64 bytes from lb-192-30-253-112-iad.github.com (192.30.253.112)
: icmp_seq=3 ttl=50 time=1.87 ms
64 bytes from lb-192-30-253-112-iad.github.com (192.30.253.112)
: icmp_seq=4 ttl=50 time=1.89 ms
64 bytes from lb-192-30-253-112-iad.github.com (192.30.253.112)
: icmp_seq=5 ttl=50 time=1.87 ms

--- github.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 1.772/1.863/1.906/0.067 ms
root@09c19697e6c5:/# 
```



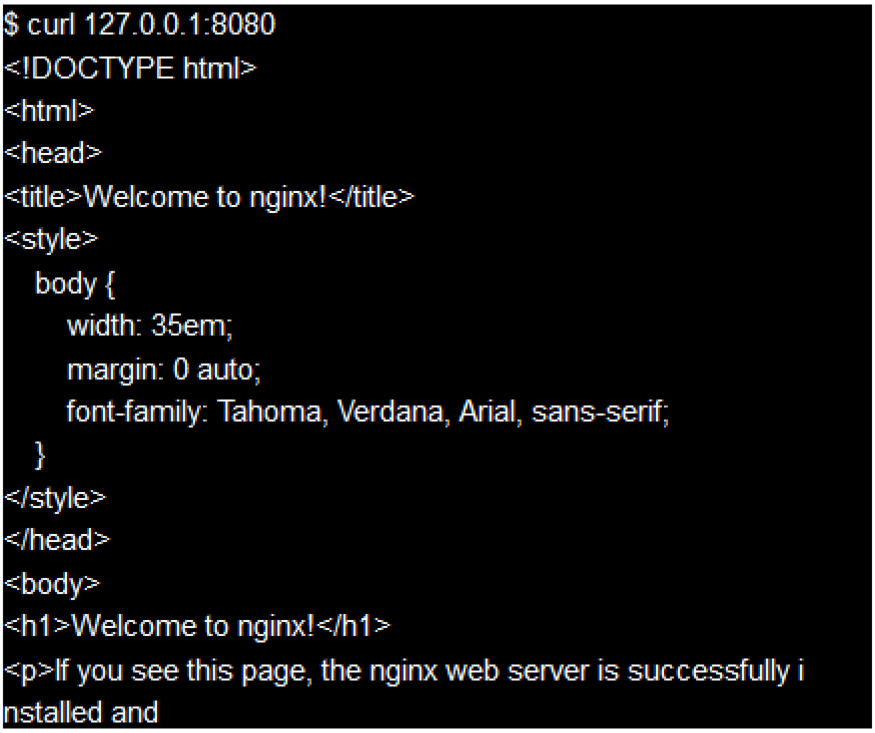
```
$ docker stop 09c19697e6c5
09c19697e6c5
[node1] (local) root@192.168.0.27 ~
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CRE
ATED             STATUS              PORTS               NAMES
```



```
$ docker run --name web1 -d -p 8080:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
a5a6f2f73cd8: Pull complete
1ba02017c4b2: Pull complete
33b176c904de: Pull complete
Digest: sha256:5d32f60db294b5deb55d078cd4feb410ad88e6fe77500c87
d3970eca97f54dba
Status: Downloaded newer image for nginx:latest
860d8bacf95b8264feaee4e67eb0323eeace1e29bae6f048a063d409a8159cb
0
```

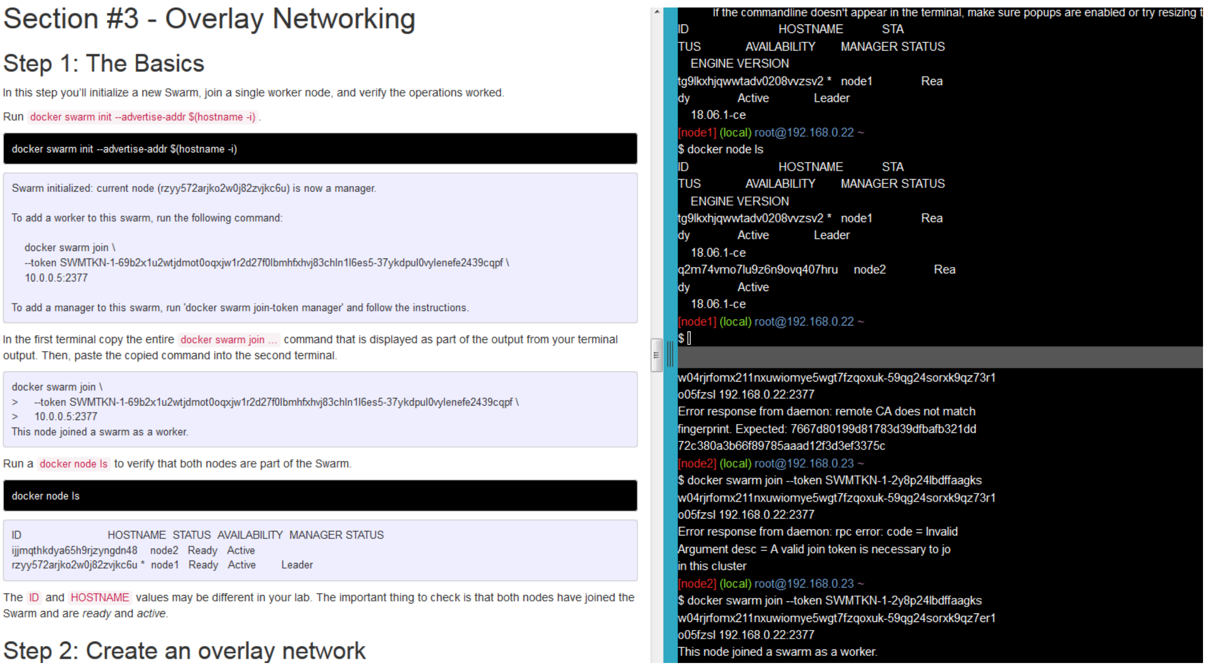


```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
  CREATED             STATUS              PORTS
  NAMES
860d8bacf95b        nginx               "nginx -g 'daemon of…"
  19 seconds ago      Up 18 seconds       0.0.0.0:8080->80/tcp
  web1
```



```
$ curl 127.0.0.1:8080
<!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 i
nstalled and
```

Section #3



Section #3 - Overlay Networking

Step 1: The Basics

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

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

```
docker swarm init --advertise-addr $(hostname -i)
```

```
Swarm initialized: current node (rzyy572arjko2w0j82zvjkc6u) is now a manager.

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

    docker swarm join \
    --token SWMTKN-1-69b2x1u2wtjdmot0oqxjw1r2d27f0lbmhfxhvj83chln1l6es5-37ykdpul0vylenefe2439cqpf \
    10.0.0.5:2377

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

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.

```
docker swarm join \
>     --token SWMTKN-1-69b2x1u2wtjdmot0oqxjw1r2d27f0lbmhfxhvj83chln1l6es5-37ykdpul0vylenefe2439cqpf \
>     10.0.0.5:2377
This node joined a swarm as a worker.
```

Run a `docker node ls` to verify that both nodes are part of the Swarm.

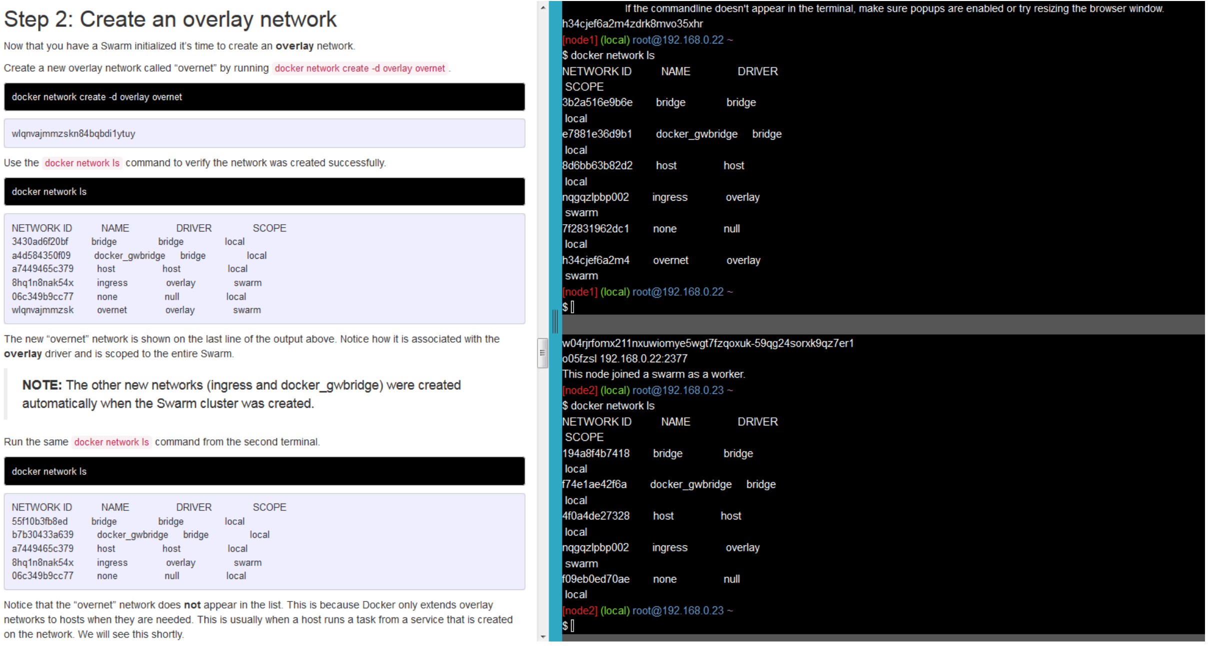
```
docker node ls
```

```
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
ijjmqthkdya65h9rjzyngdn48    node2   Ready   Active
rzyy572arjko2w0j82zvjkc6u *  node1   Ready   Active        Leader
```

The `ID` and `HOSTNAME` values may be different in your lab. The important thing to check is that both nodes have joined the Swarm and are *ready* and *active*.

Step 2: Create an overlay network

```
     If the commandline doesn't appear in the terminal, make sure popups are enabled or try resizing t
ID                          HOSTNAME            STA
TUS            AVAILABILITY        MANAGER STATUS
   ENGINE VERSION
tg9lkxhjqwwtadv0208vvzsv2 *   node1              Rea
dy             Active              Leader
   18.06.1-ce
[node1] (local) root@192.168.0.22 ~
$ docker node ls
ID                          HOSTNAME            STA
TUS            AVAILABILITY        MANAGER STATUS
   ENGINE VERSION
tg9lkxhjqwwtadv0208vvzsv2 *   node1              Rea
dy             Active              Leader
   18.06.1-ce
q2m74vmo7lu9z6n9ovq407hru     node2              Rea
dy             Active
   18.06.1-ce
[node1] (local) root@192.168.0.22 ~
$ []

w04rjrfomx211nxuwiomye5wgt7fzqoxuk-59qg24sorxk9qz73r1
o05fzsl 192.168.0.22:2377
Error response from daemon: remote CA does not match
fingerprint. Expected: 7667d80199d81783d39dfbafb321dd
72c380a3b66f89785aaad12f3d3ef3375c
[node2] (local) root@192.168.0.23 ~
$ docker swarm join --token SWMTKN-1-2y8p24lbdffaagks
w04rjrfomx211nxuwiomye5wgt7fzqoxuk-59qg24sorxk9qz73r1
o05fzsl 192.168.0.22:2377
Error response from daemon: rpc error: code = Invalid
Argument desc = A valid join token is necessary to jo
in this cluster
[node2] (local) root@192.168.0.23 ~
$ docker swarm join --token SWMTKN-1-2y8p24lbdffaagks
w04rjrfomx211nxuwiomye5wgt7fzqoxuk-59qg24sorxk9qz7er1
o05fzsl 192.168.0.22:2377
This node joined a swarm as a worker.
```



Step 2: Create an overlay network

Now that you have a Swarm initialized it's time to create an **overlay** network.

Create a new overlay network called “overnet” by running `docker network create -d overlay overnet` .

```
docker network create -d overlay overnet
```

```
wlqnvajmmzskn84bqbdi1ytuy
```

Use the `docker network ls` command to verify the network was created successfully.

```
docker network ls
```

```
NETWORK ID        NAME              DRIVER            SCOPE
3430ad6f20bf      bridge            bridge            local
a4d584350f09      docker_gwbridge   bridge            local
a7449465c379      host              host              local
8hq1n8nak54x      ingress           overlay           swarm
06c349b9cc77      none              null              local
wlqnvajmmzsk      overnet           overlay           swarm
```

The new “overnet” network is shown on the last line of the output above. Notice how it 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.

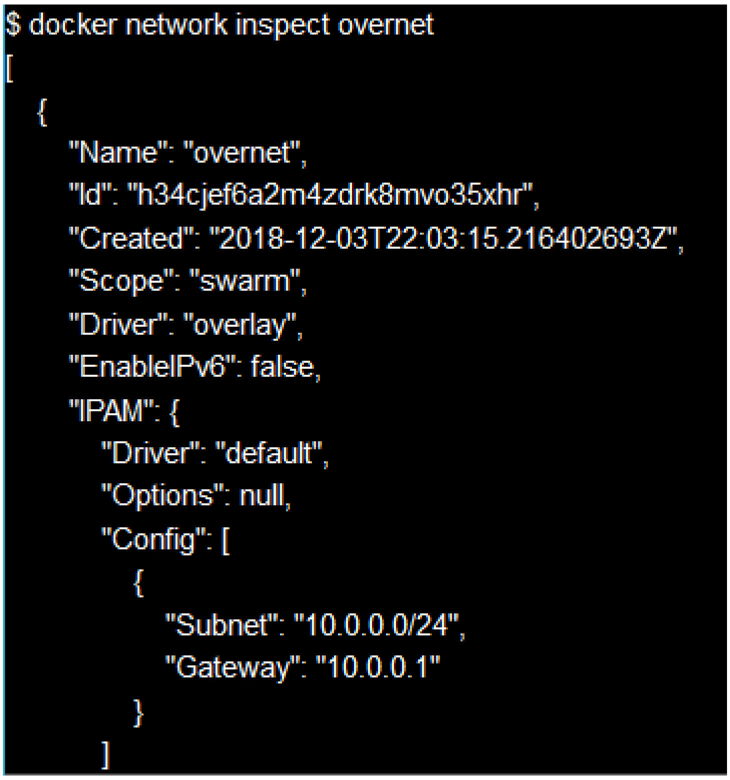
Run the same `docker network ls` command from the second terminal.

```
docker network ls
```

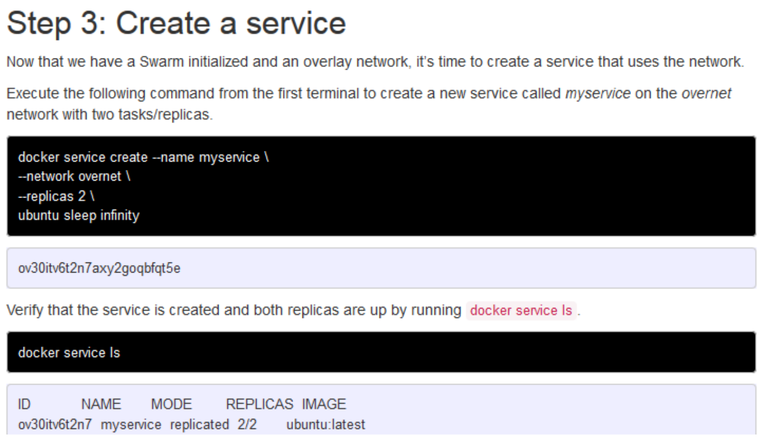
```
NETWORK ID        NAME              DRIVER            SCOPE
55f10b3fb8ed      bridge            bridge            local
b7b30433a639      docker_gwbridge   bridge            local
a7449465c379      host              host              local
8hq1n8nak54x      ingress           overlay           swarm
06c349b9cc77      none              null              local
```

Notice that the “overnet” network does **not** appear in the list. This is because Docker only extends overlay networks to hosts when they are needed. This is usually when a host runs a task from a service that is created on the network. We will see this shortly.

```
          If the commandline doesn't appear in the terminal, make sure popups are enabled or try resizing the browser window.
h34cjef6a2m4zdrk8mvo35xhr
[node1] (local) root@192.168.0.22 ~
$ docker network ls
NETWORK ID          NAME                DRIVER
 SCOPE
3b2a516e9b6e        bridge              bridge
 local
e7881e36d9b1        docker_gwbridge     bridge
 local
8d6bb63b82d2        host                host
 local
nqgqzlpbp002        ingress             overlay
 swarm
7f2831962dc1        none                null
 local
h34cjef6a2m4        overnet             overlay
 swarm
[node1] (local) root@192.168.0.22 ~
$ []

w04rjrfomx211nxuwiomye5wgt7fzqoxuk-59qg24sorxk9qz7er1
o05fzsl 192.168.0.22:2377
This node joined a swarm as a worker.
[node2] (local) root@192.168.0.23 ~
$ docker network ls
NETWORK ID          NAME                DRIVER
 SCOPE
194a8f4b7418        bridge              bridge
 local
f74e1ae42f6a        docker_gwbridge     bridge
 local
4f0a4de27328        host                host
 local
nqgqzlpbp002        ingress             overlay
 swarm
f09eb0ed70ae        none                null
 local
[node2] (local) root@192.168.0.23 ~
$ []
```



```
$ docker network inspect overnet
[
    {
        "Name": "overnet",
        "Id": "h34cjef6a2m4zdrk8mvo35xhr",
        "Created": "2018-12-03T22:03:15.216402693Z",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.0.0.0/24",
                    "Gateway": "10.0.0.1"
                }
            ]
```



Step 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.

Execute the following command from the first terminal to create a new service called *myservice* on the *overnet* network with two tasks/replicas.

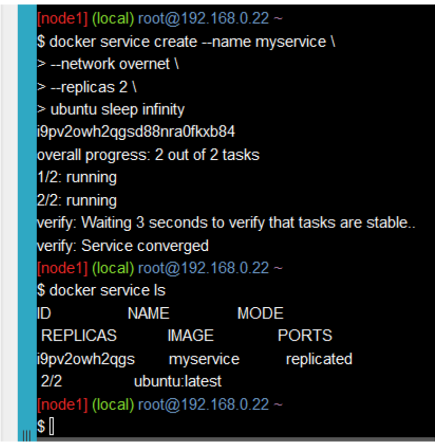
```
docker service create --name myservice \
--network overnet \
--replicas 2 \
ubuntu sleep infinity
```

```
ov30itv6t2n7axy2goqbfqt5e
```

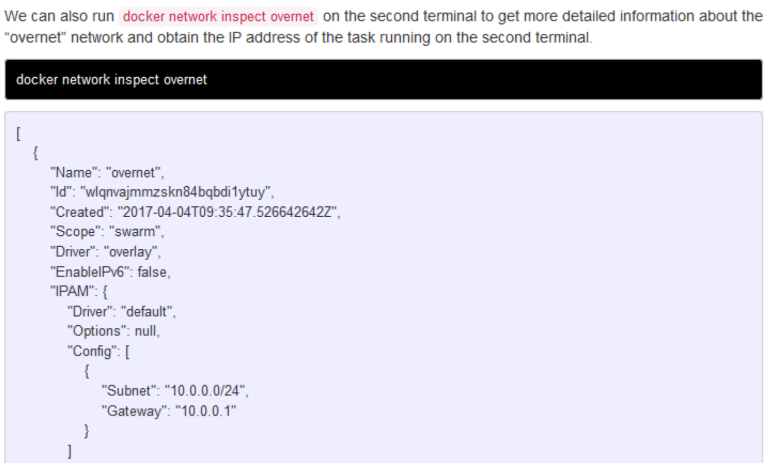
Verify that the service is created and both replicas are up by running `docker service ls`.

```
docker service ls
```

```
ID           NAME       MODE        REPLICAS  IMAGE
ov30itv6t2n7  myservice  replicated  2/2       ubuntu:latest
```



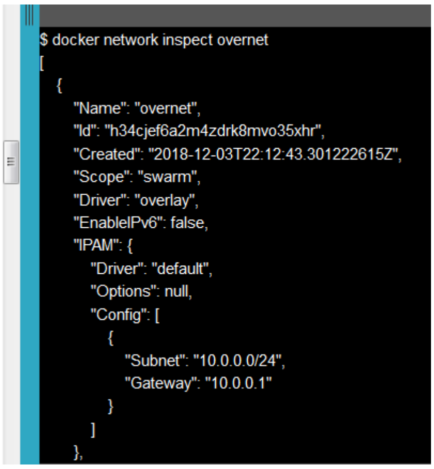
```
[node1] (local) root@192.168.0.22 ~
$ docker service create --name myservice \
> --network overnet \
> --replicas 2 \
> ubuntu sleep infinity
i9pv2owh2qgsd88nra0fkxb84
overall progress: 2 out of 2 tasks
1/2: running
2/2: running
verify: Waiting 3 seconds to verify that tasks are stable..
verify: Service converged
[node1] (local) root@192.168.0.22 ~
$ docker service ls
ID                  NAME                MODE
 REPLICAS            IMAGE               PORTS
i9pv2owh2qgs        myservice           replicated
 2/2                 ubuntu:latest
[node1] (local) root@192.168.0.22 ~
$
```



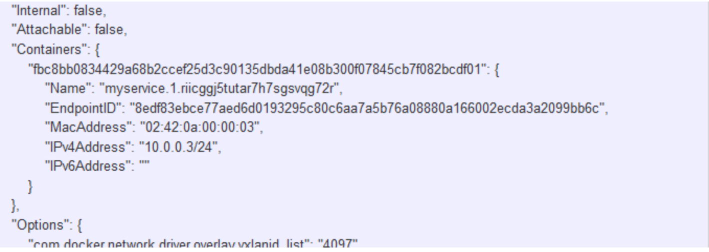
We can also run `docker network inspect overnet` on the second terminal to get more detailed information about the "overnet" network and obtain the IP address of the task running on the second terminal.

```
docker network inspect overnet
```

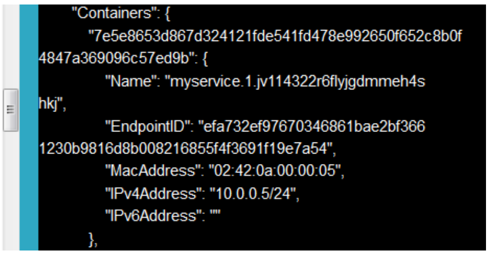
```
[
    {
        "Name": "overnet",
        "Id": "wlqnvajmmzskn84bqbdi1ytuy",
        "Created": "2017-04-04T09:35:47.526642642Z",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.0.0.0/24",
                    "Gateway": "10.0.0.1"
                }
            ]
```



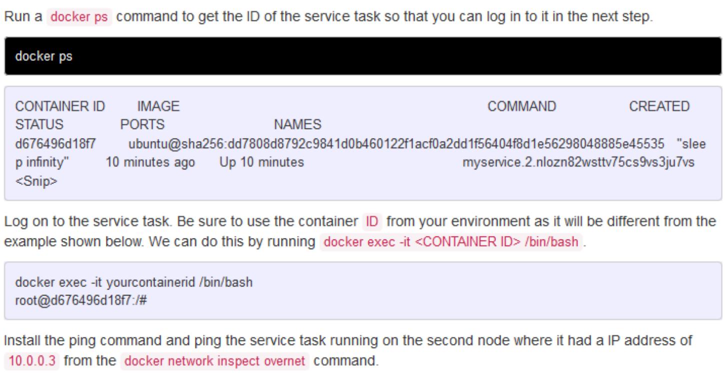
```
$ docker network inspect overnet
[
    {
        "Name": "overnet",
        "Id": "h34cjef6a2m4zdrk8mvo35xhr",
        "Created": "2018-12-03T22:12:43.301222615Z",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.0.0.0/24",
                    "Gateway": "10.0.0.1"
                }
            ]
        },
```



```
"Internal": false,
"Attachable": false,
"Containers": {
    "fbc8bb0834429a68b2ccef25d3c90135dbda41e08b300f07845cb7f082bcdf01": {
        "Name": "myservice.1.riicggj5tutar7h7sgsvqg72r",
        "EndpointID": "8edf83ebce77aed6d0193295c80c6aa7a5b76a08880a166002ecda3a2099bb6c",
        "MacAddress": "02:42:0a:00:00:03",
        "IPv4Address": "10.0.0.3/24",
        "IPv6Address": ""
    }
},
"Options": {
    "com.docker.network.driver.overlay.vxlanid_list": "4097"
```



```
    "Containers": {
        "7e5e8653d867d324121fde541fd478e992650f652c8b0f
4847a369096c57ed9b": {
            "Name": "myservice.1.jv114322r6flyjgdmmeh4s
hkj",
            "EndpointID": "efa732ef97670346861bae2bf366
1230b9816d8b008216855f4f3691f19e7a54",
            "MacAddress": "02:42:0a:00:00:05",
            "IPv4Address": "10.0.0.5/24",
            "IPv6Address": ""
        },
```



Run a docker ps command to get the ID of the service task so that you can log in to it in the next step.

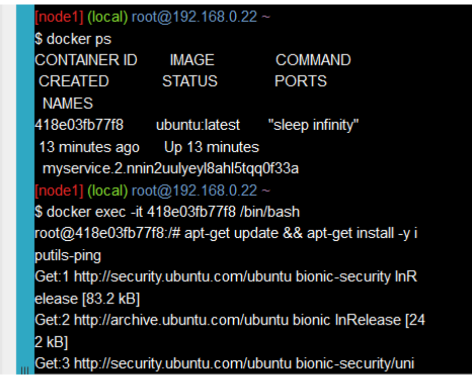
```
docker ps
```

```
CONTAINER ID        IMAGE                                                                                     COMMAND             CREATED
STATUS              PORTS                      NAMES
d676496d18f7        ubuntu@sha256:dd7808d8792c9841d0b460122f1acf0a2dd1f56404f8d1e56298048885e45535   "slee
p infinity"         10 minutes ago        Up 10 minutes                             myservice.2.nlozn82wsttv75cs9vs3ju7vs
<Snip>
```

Log on to the service task. Be sure to use the container ID from your environment as it will be different from the example shown below. We can do this by running docker exec -it <CONTAINER ID> /bin/bash.

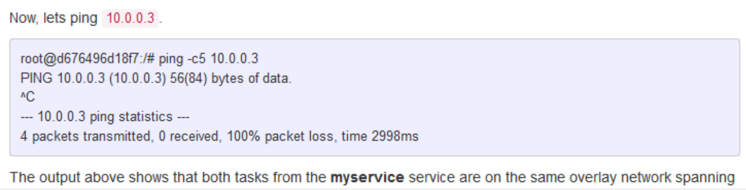
```
docker exec -it yourcontainerid /bin/bash
root@d676496d18f7:/#
```

Install the ping command and ping the service task running on the second node where it had a IP address of 10.0.0.3 from the docker network inspect overnet command.



```
[node1] (local) root@192.168.0.22 ~
$ docker ps
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
  NAMES
418e03fb77f8        ubuntu:latest       "sleep infinity"
 13 minutes ago      Up 13 minutes
  myservice.2.nnin2uulyeyl8ahl5tqq0f33a
[node1] (local) root@192.168.0.22 ~
$ docker exec -it 418e03fb77f8 /bin/bash
root@418e03fb77f8:/# apt-get update && apt-get install -y i
putils-ping
Get:1 http://security.ubuntu.com/ubuntu bionic-security InR
elease [83.2 kB]
Get:2 http://archive.ubuntu.com/ubuntu bionic InRelease [24
2 kB]
Get:3 http://security.ubuntu.com/ubuntu bionic-security/uni
```

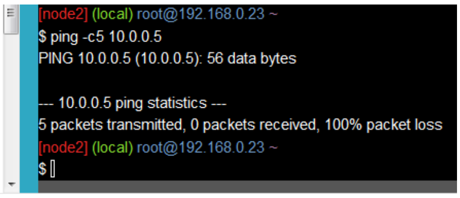
Tutaj takie dwie wersje, bo nie wiem, o który terminal im chodziło. Pingowanie zarówno 10.0.0.5 jak i 10.0.0.6 z wnętrza przechodzi jak na drugim obrazku, choć wskazują roota wewnątrz. Z zewnątrz ping wyglada jak w rozwiązaniu, na obrazku pierwszym.



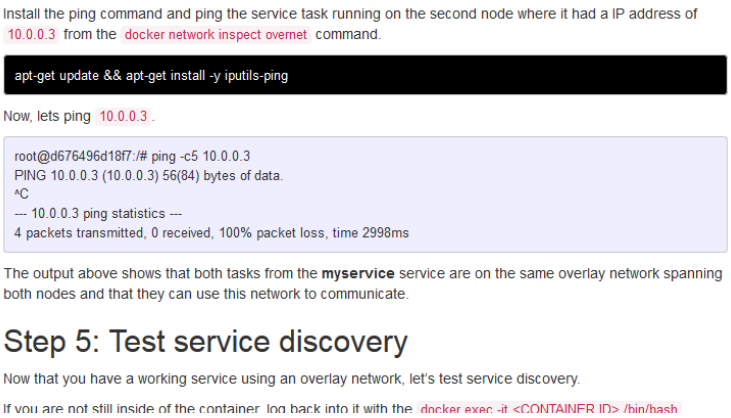
Now, lets ping 10.0.0.3.

```
root@d676496d18f7:/# ping -c5 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
^C
--- 10.0.0.3 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 2998ms
```

The output above shows that both tasks from the **myservice** service are on the same overlay network spanning



```
[node2] (local) root@192.168.0.23 ~
$ ping -c5 10.0.0.5
PING 10.0.0.5 (10.0.0.5): 56 data bytes

--- 10.0.0.5 ping statistics ---
5 packets transmitted, 0 packets received, 100% packet loss
[node2] (local) root@192.168.0.23 ~
$ ▯
```



Install the ping command and ping the service task running on the second node where it had a IP address of 10.0.0.3 from the docker network inspect overnet command.

```
apt-get update && apt-get install -y iputils-ping
```

Now, lets ping 10.0.0.3.

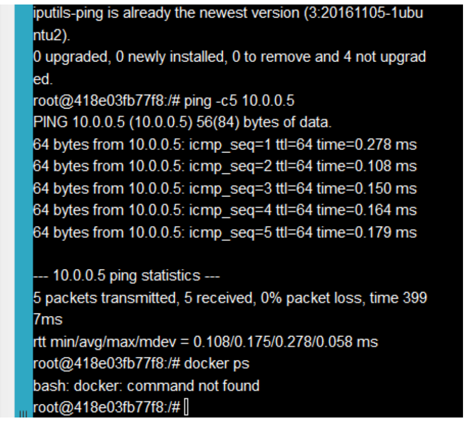
```
root@d676496d18f7:/# ping -c5 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
^C
--- 10.0.0.3 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 2998ms
```

The output above shows that both tasks from the **myservice** service are on the same overlay network spanning both nodes and that they can use this network to communicate.

Step 5: Test service discovery

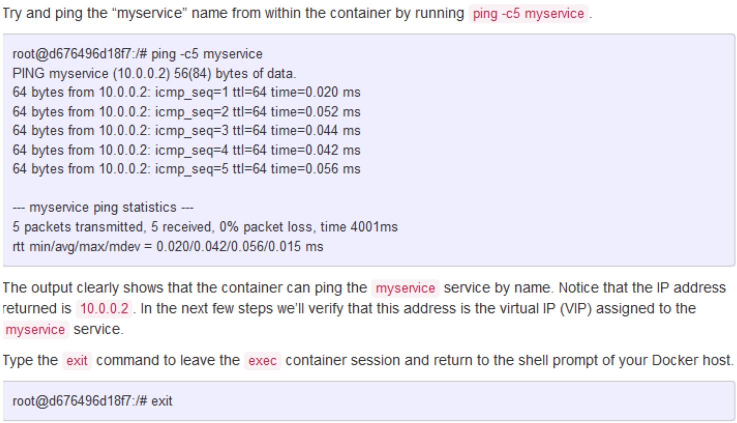
Now that you have a working service using an overlay network, let's test service discovery.

If you are not still inside of the container, log back into it with the docker exec -it <CONTAINER ID> /bin/bash



```
iputils-ping is already the newest version (3:20161105-1ubu
ntu2).
0 upgraded, 0 newly installed, 0 to remove and 4 not upgrad
ed.
root@418e03fb77f8:/# ping -c5 10.0.0.5
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
64 bytes from 10.0.0.5: icmp_seq=1 ttl=64 time=0.278 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=64 time=0.108 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=64 time=0.150 ms
64 bytes from 10.0.0.5: icmp_seq=4 ttl=64 time=0.164 ms
64 bytes from 10.0.0.5: icmp_seq=5 ttl=64 time=0.179 ms

--- 10.0.0.5 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 399
7ms
rtt min/avg/max/mdev = 0.108/0.175/0.278/0.058 ms
root@418e03fb77f8:/# docker ps
bash: docker: command not found
root@418e03fb77f8:/# ▯
```

Step 5



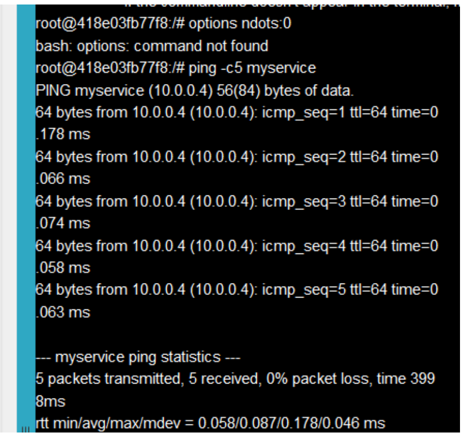
Try and ping the “myservice” name from within the container by running `ping -c5 myservice`.

```
root@d676496d18f7:/# ping -c5 myservice
PING myservice (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.020 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.052 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.044 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.042 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.056 ms

--- myservice ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4001ms
rtt min/avg/max/mdev = 0.020/0.042/0.056/0.015 ms
```

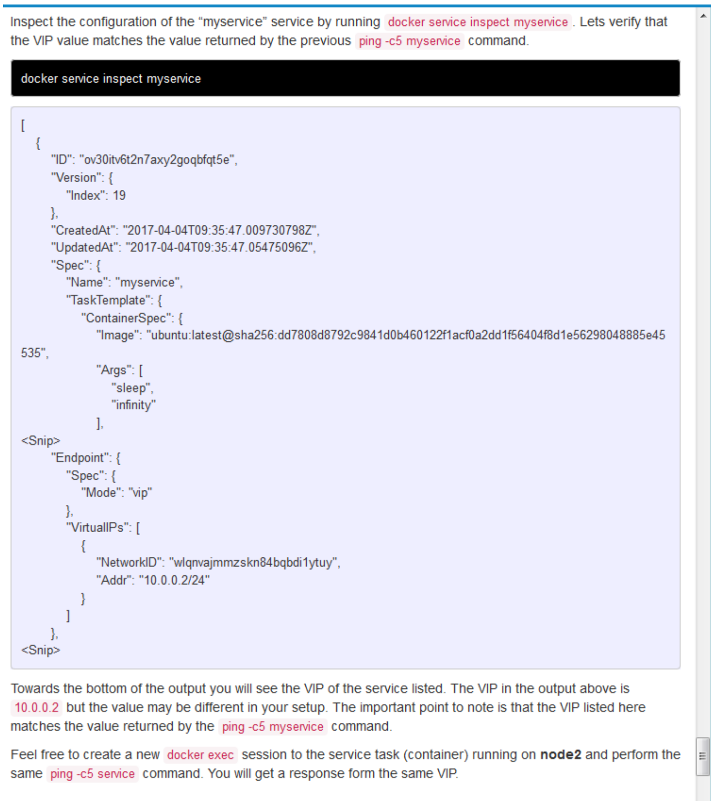
The output clearly shows that the container can ping the `myservice` service by name. Notice that the IP address returned is `10.0.0.2`. In the next few steps we’ll verify that this address is the virtual IP (VIP) assigned to the `myservice` service.

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

```
root@d676496d18f7:/# exit
```



```
root@418e03fb77f8:/# options ndots:0
bash: options: command not found
root@418e03fb77f8:/# ping -c5 myservice
PING myservice (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4 (10.0.0.4): icmp_seq=1 ttl=64 time=0
.178 ms
64 bytes from 10.0.0.4 (10.0.0.4): icmp_seq=2 ttl=64 time=0
.066 ms
64 bytes from 10.0.0.4 (10.0.0.4): icmp_seq=3 ttl=64 time=0
.074 ms
64 bytes from 10.0.0.4 (10.0.0.4): icmp_seq=4 ttl=64 time=0
.058 ms
64 bytes from 10.0.0.4 (10.0.0.4): icmp_seq=5 ttl=64 time=0
.063 ms

--- myservice ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 399
8ms
rtt min/avg/max/mdev = 0.058/0.087/0.178/0.046 ms
```



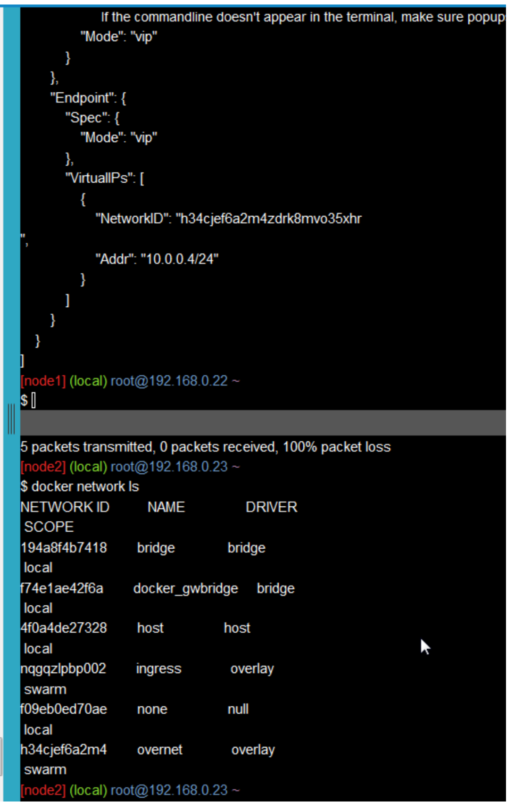
Inspect the configuration of the “myservice” service by running `docker service inspect myservice`. Lets verify that the VIP value matches the value returned by the previous `ping -c5 myservice` command.

```
docker service inspect myservice
```

```
[
    {
        "ID": "ov30itv6t2n7axy2goqbfqt5e",
        "Version": {
            "Index": 19
        },
        "CreatedAt": "2017-04-04T09:35:47.009730798Z",
        "UpdatedAt": "2017-04-04T09:35:47.05475096Z",
        "Spec": {
            "Name": "myservice",
            "TaskTemplate": {
                "ContainerSpec": {
                    "Image": "ubuntu:latest@sha256:dd7808d8792c9841d0b460122f1acf0a2dd1f56404f8d1e56298048885e45
535",
                    "Args": [
                        "sleep",
                        "infinity"
                    ],
<Snip>
        "Endpoint": {
            "Spec": {
                "Mode": "vip"
            },
            "VirtualIPs": [
                {
                    "NetworkID": "wlqnvajmmzskn84bqbdi1ytuy",
                    "Addr": "10.0.0.2/24"
                }
            ]
        },
<Snip>
```

Towards the bottom of the output you will see the VIP of the service listed. The VIP in the output above is `10.0.0.2` but the value may be different in your setup. The important point to note is that the VIP listed here matches the value returned by the `ping -c5 myservice` command.

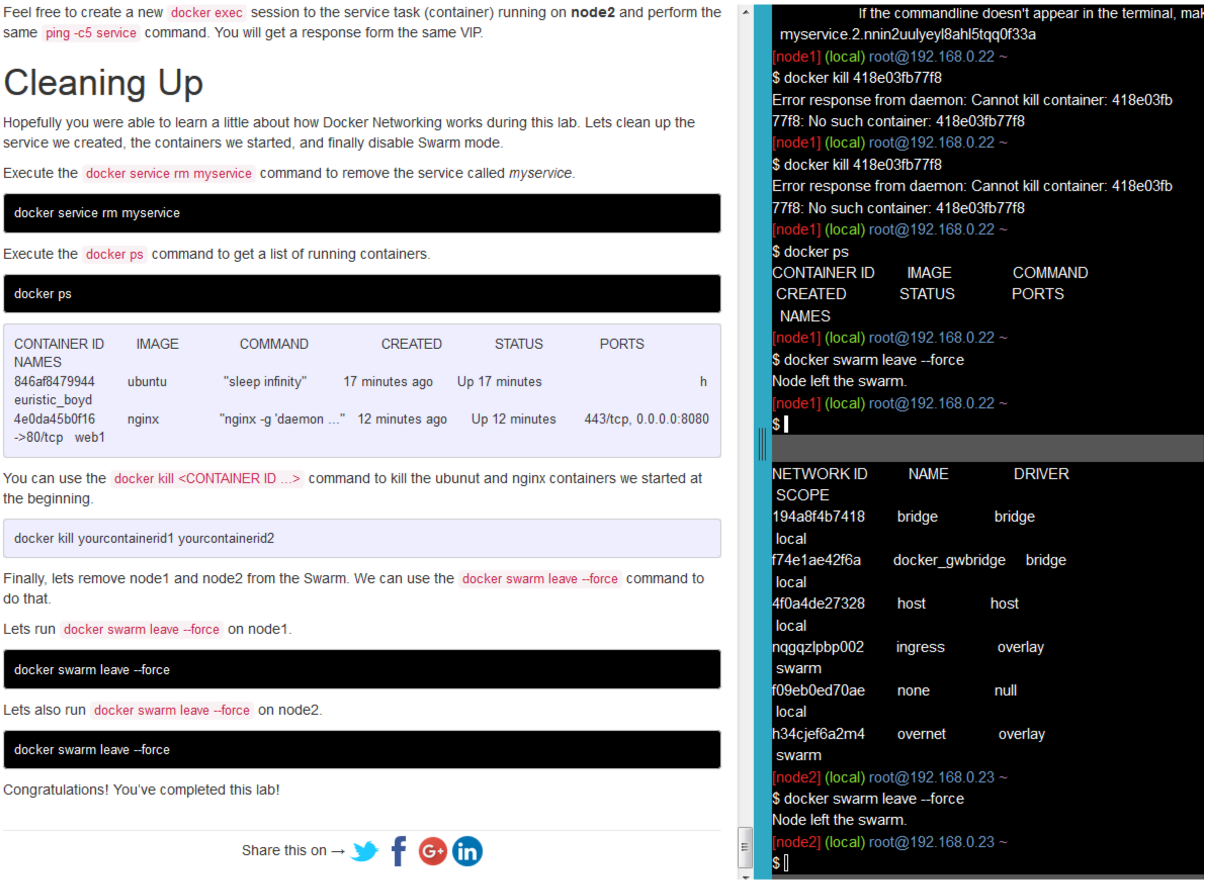
Feel free to create a new `docker exec` session to the service task (container) running on **node2** and perform the same `ping -c5 service` command. You will get a response form the same VIP.



```
            If the commandline doesn't appear in the terminal, make sure popup
            "Mode": "vip"
        }
    },
    "Endpoint": {
        "Spec": {
            "Mode": "vip"
        },
        "VirtualIPs": [
            {
                "NetworkID": "h34cjef6a2m4zdrk8mvo35xhr
",
                "Addr": "10.0.0.4/24"
            }
        ]
    }
  }
]
[node1] (local) root@192.168.0.22 ~
$ ▯
```

```
5 packets transmitted, 0 packets received, 100% packet loss
[node2] (local) root@192.168.0.23 ~
$ docker network ls
NETWORK ID        NAME              DRIVER
 SCOPE
194a8f4b7418      bridge            bridge
 local
f74e1ae42f6a      docker_gwbridge   bridge
 local
4f0a4de27328      host              host
 local
nqgqzlpbp002      ingress           overlay
 swarm
f09eb0ed70ae      none              null
 local
h34cjef6a2m4      overnet           overlay
 swarm
[node2] (local) root@192.168.0.23 ~
```

Cleaning Up



Feel free to create a new docker exec session to the service task (container) running on **node2** and perform the same ping -c5 service command. You will get a response form the same VIP.

Cleaning Up

Hopefully you were able to learn a little about how Docker Networking works during this lab. Lets clean up the service we created, the containers we started, and finally disable Swarm mode.

Execute the docker service rm myservice command to remove the service called *myservice*.

```
docker service rm myservice
```

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

```
docker ps
```

```
CONTAINER ID     IMAGE          COMMAND              CREATED          STATUS           PORTS
NAMES
846af8479944     ubuntu         "sleep infinity"     17 minutes ago   Up 17 minutes                     h
euristic_boyd
4e0da45b0f16     nginx          "nginx -g 'daemon ..."   12 minutes ago   Up 12 minutes   443/tcp, 0.0.0.0:8080
->80/tcp   web1
```

You can use the docker kill <CONTAINER ID ...> command to kill the ubunut and nginx containers we started at the beginning.

```
docker kill yourcontainerid1 yourcontainerid2
```

Finally, lets remove node1 and node2 from the Swarm. We can use the docker swarm leave --force command to do that.

Lets run docker swarm leave --force on node1.

```
docker swarm leave --force
```

Lets also run docker swarm leave --force on node2.

```
docker swarm leave --force
```

Congratulations! You've completed this lab!

Share this on →

```
            If the commandline doesn't appear in the terminal, mak
 myservice.2.nnin2uulyeyl8ahl5tqq0f33a
[node1] (local) root@192.168.0.22 ~
$ docker kill 418e03fb77f8
Error response from daemon: Cannot kill container: 418e03fb
77f8: No such container: 418e03fb77f8
[node1] (local) root@192.168.0.22 ~
$ docker kill 418e03fb77f8
Error response from daemon: Cannot kill container: 418e03fb
77f8: No such container: 418e03fb77f8
[node1] (local) root@192.168.0.22 ~
$ docker ps
CONTAINER ID      IMAGE          COMMAND
 CREATED          STATUS         PORTS
  NAMES
[node1] (local) root@192.168.0.22 ~
$ docker swarm leave --force
Node left the swarm.
[node1] (local) root@192.168.0.22 ~
$
```

```
NETWORK ID        NAME            DRIVER
 SCOPE
194a8f4b7418      bridge          bridge
 local
f74e1ae42f6a      docker_gwbridge    bridge
 local
4f0a4de27328      host            host
 local
nqgqzlpbp002      ingress         overlay
 swarm
f09eb0ed70ae      none            null
 local
h34cjef6a2m4      overnet         overlay
 swarm
[node2] (local) root@192.168.0.23 ~
$ docker swarm leave --force
Node left the swarm.
[node2] (local) root@192.168.0.23 ~
$
```