**COMP 4964 – Lab 4**

**Docker Networking**

Information

Container Network Model (CNM) design documents:

<https://github.com/docker/libnetwork/blob/master/docs/design.md>

Libnetwork – de facto implementation of the CNM

Compare to CNI (Container Network Interface) model - Kubernetes

Drivers – network-specific detail for different native drivers like bridge, IPVLAN, etc.

*Note: The actual lab starts at the top of Page 28 below.*

System Check

This part of the lab was done in an AWS CloudShell environment.

(Hint: to simplify the prompt: PS1='\u:\W\$ ')

Within a CloudShell terminal window, navigate to the /var/run directory.

Run “sudo ls -al dock\*”

A screenshot of a computer

Description automatically generated

There is file called docker.sock. This file is owned by root.

Run “cat /etc/group | grep docker”

The result is: docker:x:997:cloudshell-user

A black background with white text

Description automatically generated

This means that cloudshell-user is part of the docker group. This is good; otherwise, we would have to add the cloudshell-user to the docker group.

Check the options for using the docker networking command.

“docker network”

A screenshot of a computer

Description automatically generated

Which docker networks are installed? “docker network ls”

This shows three container networks. They are created automatically as part of the docker install.

A computer screen with white text

Description automatically generated

The NETWORK ID and NAME columns are arbitrary. We are really interested in the type of driver. The SCOPE column: values are either local or swarm cluster (multi-host). Multi-host means the network can span an entire swarm cluster. Local means single-host. Swarm means multi-host.

To drill deeper into a network, do this: “docker network inspect **bridge**” [or inspect host or local]

This is a partial screenshot:

A screen shot of a computer

Description automatically generated

Things to notice: Subnet CIDR range (172.17.0.0/16) and Gateway IP (172.17.0.1).

Remember, the gateway IP is our route to the Internet (outside world).

Shows that no Containers {} are attached to the network.

Note under Options: “com.docker.network.bridge.name”: “docker0” (close to the bottom above). This shows the connection to the host (CloudShell Linux environment.)

What is docker0? Where did it come from?

Run “ifconfig”. Remember: We ran the inspect command on the bridge driver.

A screenshot of a computer

Description automatically generated

“Bridged” means we’re attaching to the local NIC card.

What if you run “docker network inspect **host**”?

A computer screen shot of a computer

Description automatically generated

What does this information tell us?

Run “docker info”

A black background with white text

Description automatically generated

Shows a list of the network plugins: “Network: bridge host ipvlan macvlan null overlay”

These are the types of networks that this version of Docker supports in this Linux environment. (This information will be different on Windows and Mac.)

Single-Host Networking (with bridge driver)

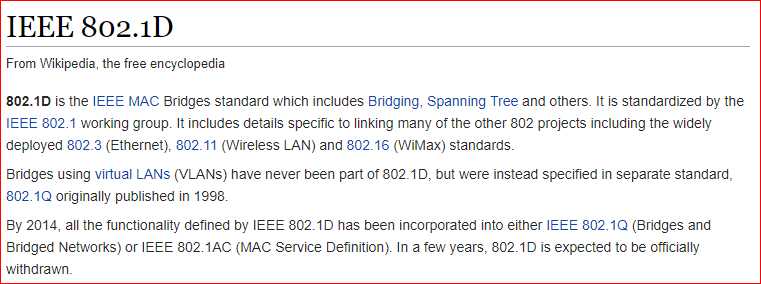
We’re going to create a user-defined bridged network on a Docker host

On Windows, it would be the NAT driver, not bridged.

Bridged networked are single-host and are scoped locally.

A bridge network creates an 802.1D Bridge on your Linux host.

This is also called a virtual switch or vswitch.



A Docker bridge driver on a Linux system uses the Linux bridge in the Linux kernel. This makes it run fast.

The “docker network create” command with the bridge driver creates a local bridge entirely in software on your Docker host.

Let’s create the bridge and plug containers into it. This is like creating a private subnet in AWS.

Run: “docker network create -d bridge --subnet 10.0.0.0/24 ps-bridge”

* Two hyphens in front of subnet
* The last parameter is the network name. You create/invent that name.
* You might want to type this command manually because of Word metadata.

The command returns the network ID.

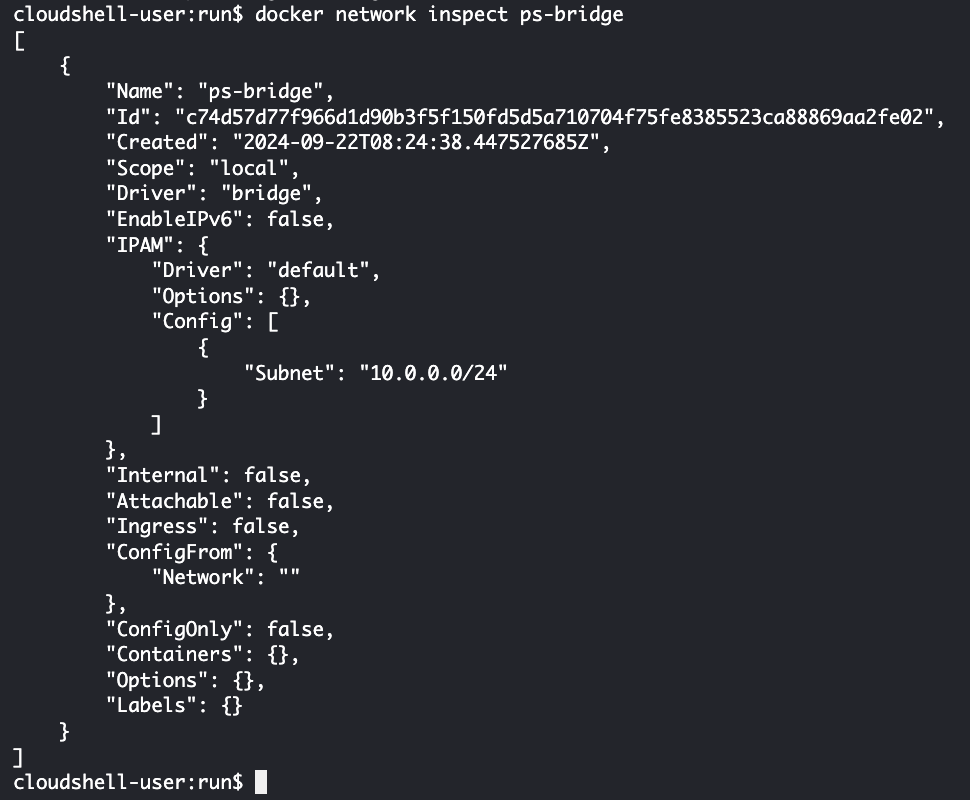


Run “docker network ls”, our new subnet “ps-bridge” is now listed.

A black screen with white text

Description automatically generated

Run “docker network inspect ps-bridge” to show the details of our subnet.



What do you see when you run “ifconfig”?

A screen shot of a computer

Description automatically generated

The “docker0” interface is the default bridged network.

Said another way, the Docker bridge network is attached to the Linux docker0 driver. Our new network (ps-bridge) is attached to the Linux “br-c74d57d77f96” interface.

Note: Our two bridge networks will have different IP subnet ranges. We can see that from the ifconfig command. Run “ip link show” Notice this is very similar to running ifconfig. Notice there is no carrier. That means these are virtual NICs.

A computer screen shot of white text

Description automatically generated

How do we attach containers to these networks? After all, that is the entire purpose of this exercise! We want to create a series of Docker containers running on the same subnet, like AWS instances, and communicating with each other.

*We can simultaneously create an image and run a container from that image!*

Run “docker run -dt --name **c1** --network ps-bridge alpine sleep 1d” [sleep one day]

* One hyphen in front of dt
* Two hyphens in front of name and network

A computer screen with white text

Description automatically generated

What is Alpine? Version of Linux? How do I find out? Hint: Look in Docker Hub.

Run “docker images” and “docker ps”

A black background with white text

Description automatically generated

I have both an image and a running container.

Create a second container and put it on the *same* network:

“docker run -dt --name **c2** --network ps-bridge alpine sleep 1d” [sleep one day]

How many images do we have? How many containers? Why?

Run “docker images” and “docker ps” to check your understanding.

A black background with white text

Description automatically generated

See the container names in the far-right column.

Run “docker network inspect ps-bridge”

A computer screen shot of a number

Description automatically generated

This shows two containers both with MAC and IP addresses. Make a note of the IP addresses.

Are the two containers on the same subnet?

Should the two containers ping each other? Let’s find out.

Run “docker exec -it **c1** sh” [Run container c1 in executive mode]

You are now inside the container.

Then run “ip a” to check the IP address of the container.

A computer screen shot of a computer program

Description automatically generated

The IP address is 10.0.0.2.

*In CloudShell, open another terminal window*. Run “docker exec -it **c2** sh”.

You are now inside the second container. Run “ip a” to see it’s IP address (10.0.0.3)

A computer screen shot of white text

Description automatically generated

From the first container, ping the second container by name or ip address.

“ping 10.0.0.3” or “ping c2”.

Why can we ping by name? Because…Every Docker engine has an embedded DNS service.

From the second container, try pinging the first container. Then, Exit out of the two containers, c1 and c2. You can leave the two containers running for now.

But, we want containers on a bridged network (ps-bridge) to be accessible from outside our network. To do that, we need to publish a container’s service on the host’s port. This is not recommended or used much, but we want to illustrate the idea.

Start by finding a Docker image on Docker Hub with Apache already installed. I found **eboraas/apache**. Look at how the Docker file was built.

<https://hub.docker.com/r/eboraas/apache/dockerfile>

A screenshot of a computer

Description automatically generated

Two important things about this image: Port 80 is exposed and it starts Apache HTTP server when a container is run from the image. We will pull this image and run a container using it now. We are putting it on our network.

So, start another container with a web server:

“docker run -d --name **web1** --network ps-bridge -p 5000:80 eboraas/apache”

A computer screen shot of a computer

Description automatically generated

Install elinks on *CloudShell “sudo yum install elinks”*

*Run elinks with* the public IP address of the *CloudShell instance* plus the port 5000.

How to figure out your CloudShell public IP? Hint: ifconfig

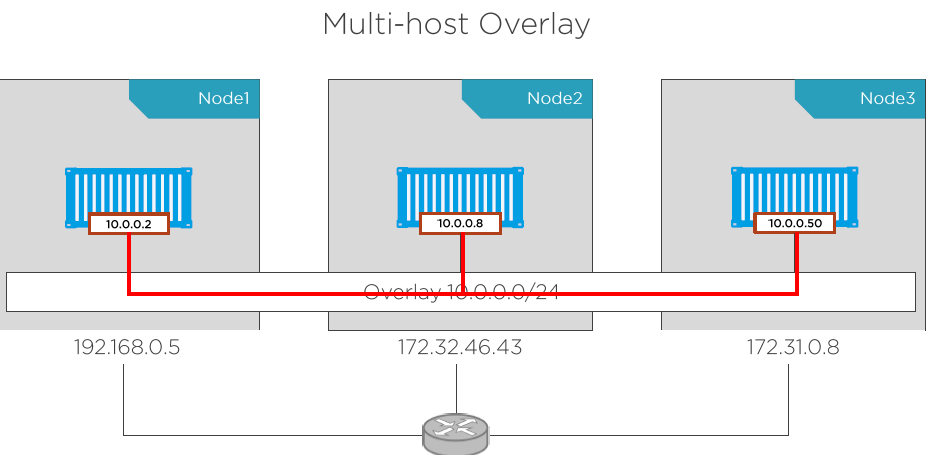
A screenshot of a computer

Description automatically generated

Did you open port 5000 on your security group? (Not port 80!)

This is clunky because we are using port 5000. This is not scalable. What do we mean by that? No other services can use port 5000 because we have attached it to our container. This is OK for use on a single host, but that’s not the real world.

Multi-Host Overlay Networking



In the above diagram, think of each node as a different AWS instance with a public and private IP address. Each is running a Docker container. But the containers think they are all running on the same subnet. We are building an overlay network, with a single CIDR block, that spans multiple hosts. Then, the containers running on different hosts can all be on the same subnet. No port mapping, etc., is required.

A VXLAN tunnel (the overlay network) is connecting the hosts (instances). The underlay network is a router on a broadcast network. It is the network containing the instances.

Configuring AWS

From the AWS Console, search and select the VPC.

A screenshot of a computer

Description automatically generated

Be careful with networking! Do *NOT* delete your Default VPC or subnets!

Select Create VPC from the VPC Dashboard.

A screenshot of a computer

Description automatically generated

On the left, select “VPC and more” [default]

A screenshot of a computer

Description automatically generated

Scroll down, make the settings as follow:

IPv4 CIDR block: 192.168.0.0/**16**

Number of Availability Zones (AZs): 1

Open **Customize AZs** section, and then

First availability zone: us-west-2a

A screenshot of a computer

Description automatically generated

Continue making settings:

Number of public subnets: 1

Number of private subnets: 0

Public subnet CIDR block in us-west-2a: 192.168.16.0/20

A screenshot of a computer

Description automatically generated

NAT gateways: None

VPC endpoints: S3 Gateway

A screenshot of a computer

Description automatically generated

Before confirming with “Create VPC”, double check in Preview section as:

A screenshot of a chat

Description automatically generated

Click “Create VPC” button, and then “View VPC”.

Back to VPC dashboard, you should now see two VPCs.

After changing the new VPC name to “Docker”, you should see:

A screenshot of a computer

Description automatically generated

Next, perform *VPC peering* using your existing default VPC and the new, Docker VPC. Follow the instructions for peering in the same region.

<http://docs.aws.amazon.com/AmazonVPC/latest/PeeringGuide/create-vpc-peering-connection.html#create-vpc-peering-connection-local>

Make the Default VPC the Requestor and new, Docker VPC as the Accepter.

A screenshot of a computer

Description automatically generated

…

A screenshot of a login box

Description automatically generated

Click Create Peering Connection.

A close-up of a sign

Description automatically generated

Next, select the new connection and then click the Actions menu and Accept request

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

The status should show Active:

This means that instances created on the two different networks can ping each other even though they are not on the same subnet.

Next, update the route tables: <http://docs.aws.amazon.com/AmazonVPC/latest/PeeringGuide/vpc-peering-routing.html>

*You must do this in both directions.* In other words, you select your Default network and a route through the Peer Connection to the new VPC (Docker). Then, select your new VPC (Docker) and create a route through the peer connection to the Default VPC.

In the screenshots below, I have selected the routing table for the Default VPC and added a route to the Docker VPC. Remember: The route goes through the peering connection.

Graphical user interface, text, application

Description automatically generated

Make sure the Status is “Active” and NOT “Blackhole.” What would Blackhole mean?

In these screenshots below, I have selected the *main* Docker routing table and added a route to the Default VPC. Note: The CIDR range for you Default VPC may be different than 172.31.0.0/16. You will have to check that first.

In both cases the Target for the new route is the Peering connection (pcx-).

Graphical user interface, text, application, email

Description automatically generated

You must create new Security Group (SG) to be used for this Docker networking architecture. You must create a new SG *in each VPC*. You can name security group “Docker-SG-1” and “Docker-SG-2”. They should be identical. *For simplicity, create the SG to allow all inbound traffic from Anywhere*. [The underlay networking connects on ports 2789 (udp), 7946 (tcp/udp) and 2377 (tcp). If we used just those ports and something didn’t work, it might be difficult to diagnose for this lab.]

Graphical user interface, text, application

Description automatically generated

Create two Ubuntu, instances, one in the default VPC and one in the new VPC (Docker).

A screenshot of a computer

Description automatically generated

Make sure they are in the *same* Availability Zone. Make sure they both get public and private IP addresses. Make sure they use one of the new Security Groups you created previously. Add your Key Pair to each instance. You can tag them as Docker Manager and Docker Worker.

A screenshot of a computer

Description automatically generated

**Before** installing anything on the instances, make sure the instances can ping each other on their *private* IP addresses! (SSH into each instance. The username is ubuntu, not ec2-user.) This means the network peering is working. (We rock!) For myself, one instance has a 34.x.x.x IP address and the other has a 35.x.x.x address. This means that they are on different networks.

I logged into each instance. They have different private IP addresses. They can ping each other.

A screenshot of a computer

Description automatically generated A screenshot of a computer code

Description automatically generated

Install Docker and Docker Swarm

Install Docker and Docker Swarm on each Ubuntu instance. All you’re going to do is to create a two-node swarm with a single manager and a single worker.

First, run “sudo apt update -y”.

Next, Run “sudo apt install docker.io”. [*don’t* add a ‘-y’ at the end]

This will install Docker.

Docker Swarm is installed, by default, as part of this installation.

Verify that Docker is installed. Run “docker --version”.

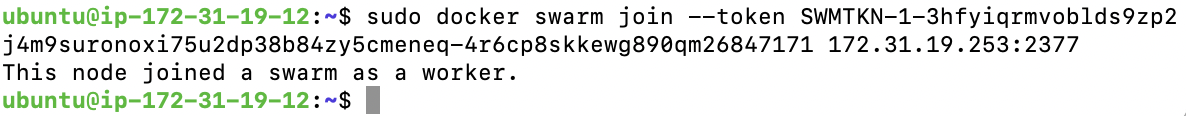
You are going to use Swarm Mode.

On the Docker Manager instance, run “sudo docker swarm init”. This node (instance) will be the manager. The command will return the command you need to run on the other node for it to join the swarm as a worker. See the line beginning “docker swam join”.

A screen shot of a computer

Description automatically generated

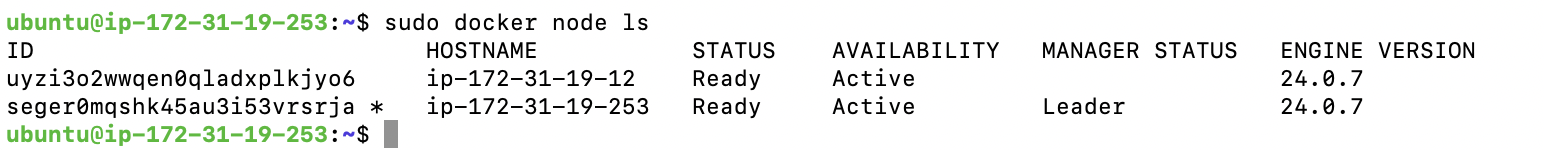
Go to the other instance (Worker) and paste the “docker swarm join –token ” stuff. Add “sudo” in front of the command as well. Don’t forget to include the IP address and the port# at the end of the command!



It should say “This node joined a swarm as a worker.”

Go back to the manager node, run “sudo docker node ls”

The manager node is the Leader.



Notice the IP addresses.

Now, list the networks on the Manager:

Run “sudo docker network ls”

A screenshot of a computer

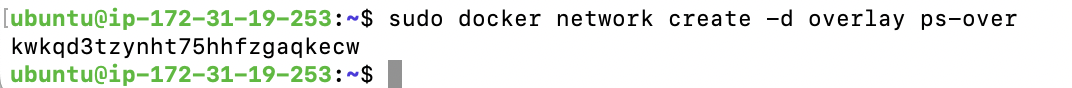
Description automatically generated

We should see the “docker\_gwbridge” network has been added.

It is like a default gateway to get to the outside world.

We also see “ingress” which is an overlay network used for swarm balancing. But you need to create your own overlay network. Why are we doing this? You are going to create a container on each node. Each container must be on the same subnet so they can see each other! That is the purpose of the overlay network, a separate network for the containers.

Make sure you are on Node 1, the Manager node. Run “sudo docker network create -d overlay ps-over” [the last param is the name we give the network]



Then, run “sudo docker network ls”

A screenshot of a computer code

Description automatically generated

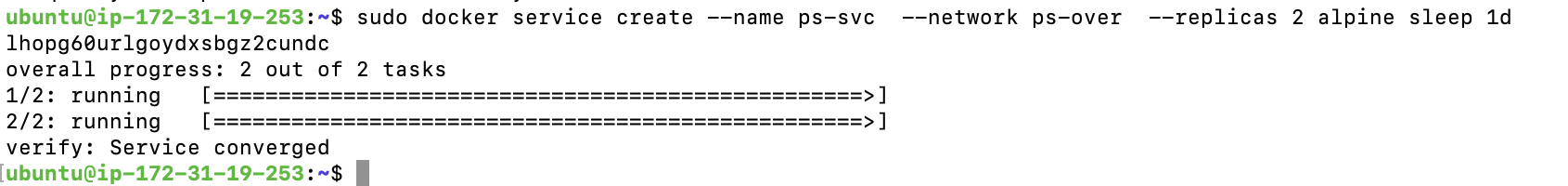
Our network is scoped to the swarm which means it should be available on both nodes.

But, when you do “sudo docker network ls” on the second node (Worker node), you won’t see the new ps-over network. Why not? It will only appear when a container is created on the second node that needs to use the ps-over network and then it will magically appear.

To create a container on the 2nd node (Worker node), start a **service** on node 1 (Manager).

On node 1 (Manager node):

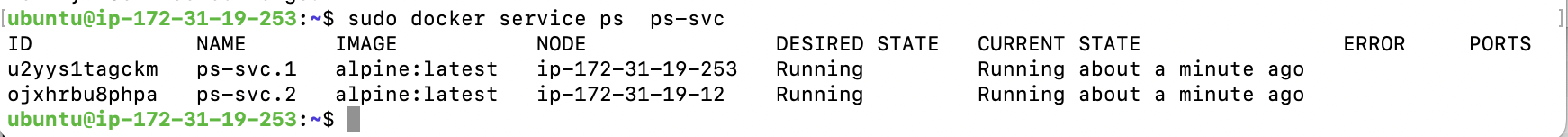
Run “sudo docker service create --name ps-svc --network ps-over --replicas **2** alpine sleep 1d” [sleep one day] Replicas are containers. Swarm will automatically do load balancing and create one container on each node.



Still on Manager node, run “sudo docker service ps ps-svc”

This shows tasks or replicas running in the service ps-svc.

Look at the Node column. It shows the IP address where each container is running.



On each node, run “sudo docker ps” You should see one container on each node.

How did the second container get launched and running on the worker node?

On the manager node, run “sudo docker network inspect ps-over”

Notice the subnet configuration and default gateway.

A screenshot of a computer code

Description automatically generated

Notice that only one container is attached to the network. Why is that? Notice the IP address of the container.

A computer code with numbers and letters

Description automatically generated

You can see the Peers or nodes that are part of the swarm.

A screenshot of a computer code

Description automatically generated

How do I know which instance (node) the container is running on? First, notice each container has long name. On each node, run “sudo docker ps”. This will display the name of the container on that node.

Go to node 2 (Worker node) and run “sudo docker network ls”. The ps-over will now be shown.

On node 2, run “sudo docker network inspect ps-over”

Now, let’s see if we can ping the container on node 1 (manager) over the overlay network

On node 2 (Worker), grab the name of the container and “exec” into it.

Here’s my container Name on my Worker node:

A computer code with numbers and letters

Description automatically generated

Run “sudo docker exec -it ps-svc.2.ojxhrbu8phpa01jewx2f6vscr  **sh**” [don’t forget sh on the end, and don’t forget to replace the name in blue with your own container Name]

Now, ping the IP address inside the container on Node 1 (manager):

A screenshot of a computer program

Description automatically generated

You can go to Node 1 (Manager) and do the same thing if you wish. “Exec” into the container and ping the container on node 2 (Worker).

Once you get all this working, and understand everything as well, then stop these instances. You will be able to terminate them at the very end of this lab.

Integrating Containers with Existing Workloads (networks)

Network Services

Bringing applications and networks together.

How do I put containers under a service? (We did this above.)

* *Service discovery – important to microservices*
* Port-based routing with routing mesh
* App-aware routing with the HTTP routing mesh

Makes building and managing containerized applications easier.

* Service discovery is about finding services
* Modern applications -> gluing together microservices
* Service discovery is automatic with Docker

DNS service discovery in Docker

* Every container gets a DNS resolver (not a server)
* Listens on port 53 at 127.0.0.1 on every container
* Forwards requests to DNS server on the local Docker host machine. If it can’t resolve it, it sends it to the default gateway (www).

Service discovery is what is called network scope, but the containers must be on the same container network. This is called “network-scoped discovery.”

Virtual IP-based load balancing: Every Docker service gets a virtual IP

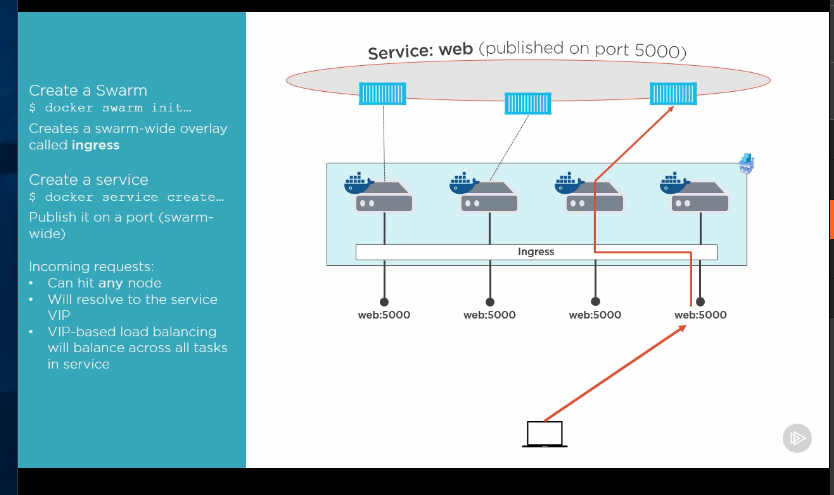
A service may have many containers under it. Each service has one or many tasks (containers). Every task (containers) gets an IP address. All the task IPs get grouped together under a service-wide VIP. So, if the VIP gets multiple requests, they are round-robined across the tasks. This is internal load balancing on the same network. What about outside requests? Next section….

The Routing Mesh

*Port-based routing with the routing mesh.*

Port-based routing, Layer 4 (transport layer). This is good for swarm services, but it is limited. Transport/Layer 4 routing mesh balances outside requests.

What happens to requests that hit nodes not running a task from the service?



[Note to instructor: Start the Routing Mesh video at location 3:00 or so.]

Lab 4

The lab is based on four Docker machines running on the same network. (It doesn’t make any difference which VPC you choose.) One instance will be the Manager node and the others will be Worker nodes.

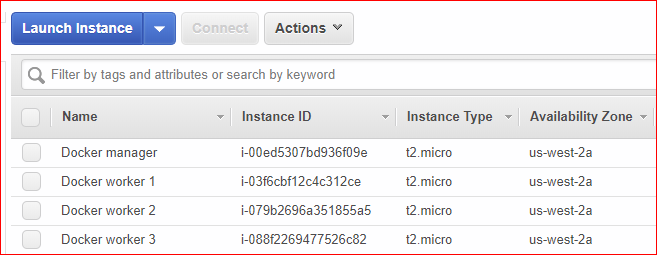
Launch four (4) Ubuntu instances at the same time. Add this to Advanced Details and spin up the four (4) nodes:

#include https://get.docker.com

Graphical user interface, application

Description automatically generated

Make sure they all launch in the same Availability Zone.



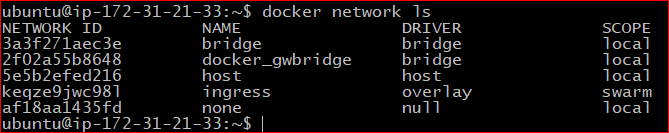
After the instances are running, you can run “sudo usermod -a -G docker ubuntu” on each node or simply run “sudo su” on all instances. (If you run “usermod”, you will have to disconnect and reconnect to your instance).

Verify Docker has been installed on each instance by “docker --version”.

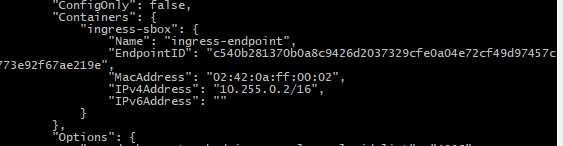
Pick one instance as the manager and run “docker swarm init”

Copy the “docker swarm join” line from the manager and paste it into the other three (3) instances so that they all join the swarm. (If this doesn’t work for you, it is because you didn’t read the bottom of page 23).

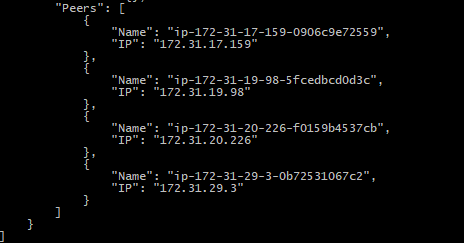
Run “docker network ls” on the manager. The ingress network is system-wide and instantly available on all nodes and the scope is swarm.



Run “docker inspect ingress”

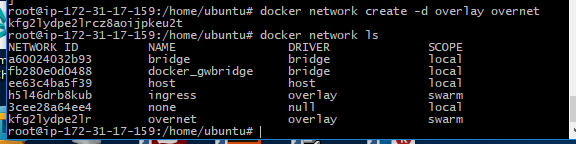


The ingress-endpoint sits on every node. It is a system container. It is used by swarm for routing mesh operations. The container, however, is different on every node.



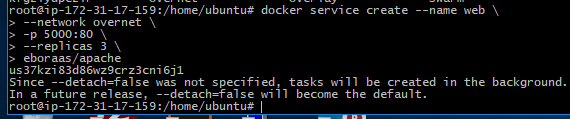
Next, create an overlay to run a service on the manager node. Call it “overnet”.

Run “docker network create -d overlay overnet”



Create a service with three (3) containers running on the swarm:

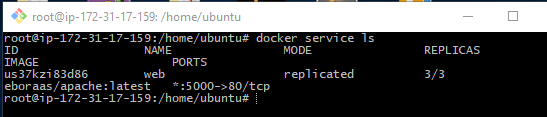
“docker service create --name web --network overnet -p 5000:80 --replicas 3 **fred2030/myapache**” [This should be your Docker repository from the previous lab!!!]



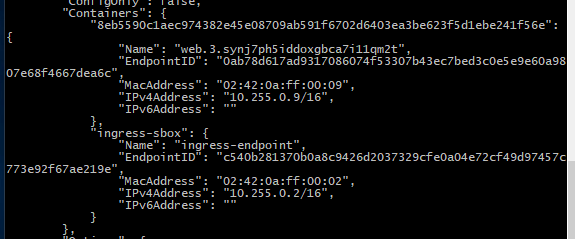
What is going to be running on each node (instance)? How can we test this?

Run “docker ps” on each instance. Is a container running?

Run “docker service ls” on the manager node.



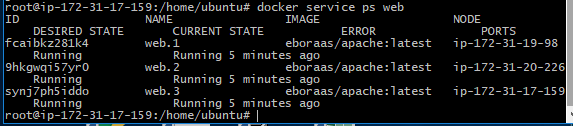
Run “docker network inspect ingress”



There is now a container (task) from the web service. Every time you create a service with a published port, every task or container in the service also gets attached to the ingress network for routing and load balancing.

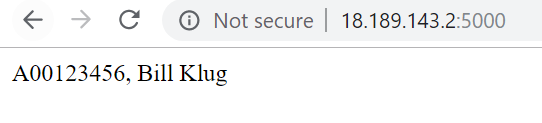
We have four nodes in a swarm and three nodes in a service.

“docker service ps web”



This shows which nodes doesn’t have a task. In my case, node 4 wasn’t running anything.

How do I test if everything is working? Get the public IP address of one of the nodes (EC instances) running a container. Enter the IP address plus the port number, e.g. <http://18.189.143.2:5000>, in your web browser. You should see this



It works! The ingress network routing mesh makes sure the request hits the service, not the individual instance.

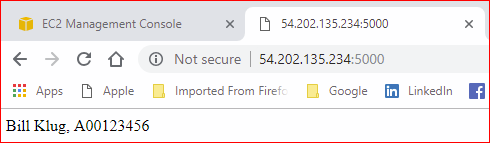
This is port-based routing, Layer 4 (transport layer). This is good for swarm services. But it is limited: You can’t have two services listening on the same port.

Lab Deliverables

Send an email to your instructor (frederic\_guo@bcit.ca). In the email, include your four (4) public IP addresses of your Ubuntu, EC2 instances.

I will enter this in my browser: <IP Address>:5000 to verify your work.

I should see a screen with your name (or initials) and A00 number:



I will respond to your email and ask you to terminate your instances.

[Bonus Topic] The HTTP Routing Mesh (HRM)

We aren’t doing to do this because it costs money!

*App-aware routing with the HTTP routing mesh (HRM)*

This is application, Layer/7 routing.

It builds on top of port-base/layer 4 (transport layer) routing mesh.

Remember the limitation of port-based routing mesh? You only a single service on any swarm cluster can listen on particular network port, i.e., you can’t have two services listening on the same port. Because it operates at layer 4, it is not aware of what happens higher up the stack, like layer 7

HRM requires **Docker Datacenter** – See AWS Quick Start for Docker Datacenter

HRM is a commercial offering – it costs money.