Domain 4: Networking (15% of exam)

Create a Docker bridge network for a developer to use for their containers

1. Create a user defined bridge network.

```
# docker network create --driver bridge mynetwork
```

NOTE: "--driver bridge" can be skipped since the default driver is bridge. One can specify gateway, subnet, ip range to allocate from etc

CLI Reference: docker network create

Troubleshoot container and engine logs to understand a connectivity issue between containers

nicolaka/netshoot is an image that is specifically used to access the network name space of any other container for troubleshooting. The image comes with a lot of network troubleshooting tools installed. The following command runs the netshoot container and provides access to the network namespace of another container.

```
# docker run -it --rm --network container:<container_name> nicolaka/netshoot
NOTE: This provides access to network namespace of <container_name> via netshoot
container. From here tools like nslookup, dig etc can be used to check
connectivity and DNS resolution.

The following commands provide useful information for troubleshooting connectivity
issues.

# docker network inspect -v <network-name-of-failing-network>
NOTE: Get list of containers using a specific network, along with GW and Subnet
details.

# docker service ps <service-name-of-service-with-issues>
NOTE: Get list of tasks the node on which it is running, for failing service.

# docker inspect --type task <task_id>
NOTE: inspect tasks of service that failed

# journalctl -u docker --no-pager --since "YYYY-MM-DD 00:00" --until "YYYY-MM-DD
00:00"
NOTE: check daemon (engine) logs on hosts where failure is observed, for the
duration of failure

One can also access the ingress network's namespace by running the following
command.
# docker run -it --rm -v /var/run/docker/netns:/netns --privileged=true
nicolaka/netshoot nsenter --net=/netns/ingress_sbox sh
```

```
NOTE: ingress network is the overlay network that connects multiple nodes in a
swarm. Requests can be sent to any of the nodes in swarm (even if that node is not
running that task) and the requests are forwarded automatically to containers.
```

Publish a port so that an application is accessible externally

As discussed in the earlier section "Add networks, publish ports" under "Domain 1: Orchestration" (which has a document of its own available as a separate download), ports can be published. Example,

docker run -dt -p 127.0.0.1:443:443/tcp alpine
NOTE: publishes TCP port 443 on docker host ip 127.0.0.1

docker run -dt -p 443:443/tcp alpine
NOTE: publishes TCP port 443 on docker host on all IPv4 addresses. i.e 0.0.0.0.

Instead of -p one can also use '--publish published=443,target=443,protocol=tcp'

docker container port <container-name>
NOTE: this tells what ports are mapped for a specific container to which IP on host.

Iptables and docker: Every time a port is published, docker automatically implements iptable rules to provide network isolation. Docker creates two chains on docker host, namely "DOCKER" and "DOCKER-USER". Any manipulation on docker host should be done only to the DOCKER-USER chain. For example, By default all external IPs are allowed to access docker host for container access. To restrict this behavior custom rules can be added to "DOCKER-USER" chain.

Identify which IP and port a container is externally accessible on

The following commands can be used to identify port mapping details of a container.

docker container port <container-name>
NOTE: this provides list of all container ports and its corresponding host ports, host IP and protocol.

docker container port <container-name> 443/tcp
NOTE: this tells to which port, host IP and protocol the container port 443/tcp is mapped to.

Describe the different types and use cases for the built-in network drivers

"Bridge" network is primarily used for networking between instances running on the same docker daemon host. This private network is created automatically when docker is started and becomes the default network for new containers, unless a different network is specified. Default bridge network does not provide DNS. Containers can ping each other using IP only. To overcome this, user defined bridge networks can be created and used.

NOTE: A user defined bridge network can be created and attached to multiple containers. Containers sharing the user defined bridge expose all ports with each other and need not be published.

2. Create a user defined bridge network.

```
# docker network create --driver bridge mynetwork
```

NOTE: "--driver bridge" can be skipped since the default driver is bridge. One can specify gateway, subnet, ip range to allocate from etc

CLI Reference: docker network create

3. Connect a container to an existing network

```
# docker network connect mynetwork somecontainer
```

CLI Reference: docker network connect

4. Disconnect a container to an existing network

```
# docker network disconnect mynetwork somecontainer
```

CLI Reference: docker network disconnect

5. Inspect a network

```
# docker network inspect mynetwork
```

CLI Reference: docker network inspect

6. List all networks

```
# docker network ls
```

CLI Reference: docker network ls

7. Remove all unused networks

```
# docker network prune
```

CLI Reference: docker network prune

8. Remove a network

```
# docker network rm mynetwork
```

CLI Reference: docker network rm

"overlay" is a type of network driver which is used for communication between containers or swarm tasks running across multiple docker daemon hosts. An overlay network can be created by passing flag "--driver overlay" when creating a network.

1. Create an overlay network

```
# docker network create --driver overlay myOverlayNetwork
```

2. Create an overlay network with encryption enabled

```
# docker network create --driver overlay --opt encrypted
 myOverlayNetwork
```

NOTE: when creating a network, if "--attachable" flag is passed to it, the created network can be used to connect to containers or swarm services from other docker daemon hosts.

NOTE: when initiating a swarm or join node to swarm, data traffic and control traffic uses the same network in that node. This can be separated by passing flag "--datapath-addr <data-traffic-ip>" and "--advertise-addr <control-ip>"

"macvlan" is a type of network driver which assigns a MAC address to each container's virtual interface. This is useful when legacy applications require to communicate directly with physical networks. "macvlan" mimics that kind of setup in docker. For this to work one physical network, subnet and gateway needs to be designated from docker host. Also, the network equipment should support promiscuous mode to support tagging of multiple MAC addresses to single interface.

1. Create a macvlan network

```
# docker network create --driver macvlan myOverlayNetwork
```

Understand the Container Network Model and how it interfaces with the Docker engine and network and IPAM drivers

Some important terminologies to know for DCA about CNM.

Sandbox – An entity that holds a container's network stack namely container's network interfaces, DNS and routing table.

Endpoint – Endpoint connects sandbox to a network. This way the container service is separated from the actual underlying network. This enables portability, where in container service can run regardless of the type of underlying network.

Network – A collection of endpoints connected with each other.

Network drivers can be native drivers (bridge, overlay, macvlan, none) or remote drivers provided by docker community which is available as plugins.

IPAM Driver – This is the inbuilt driver that assigns IPs to containers from the private bridge network that gets created on docker host.

When you run “docker network ls” the networks will also identify itself as “local” or “swarm” to identify its scope. Local scope is for networks that connects containers of a host, whereas swarm scope is for networks that connects multiple docker hosts. The network ID for a swarm scope network is the same on all nodes that uses that network.

Configure Docker to use external DNS

External DNS can be configured either when starting a container/service or it can also be added as default to /etc/docker/daemon.json

Set DNS while starting container, Example

```
# docker run --dns 8.8.8.8 -it centos /bin/bash
```

Set dns, dns options and dns search in a service, Example

```
# docker service create --entrypoint "ping google.com" --replicas 2 --name
myvm --dns 8.8.8.8 --dns 4.4.4.4 --dns-option timeout:5 --dns-search
unixutils.com  centos:latest
```

Set default dns globally at daemon level. (If DNS is not set manually like in above commands, this is used)

```
# Update the following in /etc/docker/daemon.json and restart docker
service.

{
    "dns": ["8.8.8.8", "4.4.4.4"]
}
```

Use Docker to load balance HTTP/HTTPs traffic to an application (Configure L7 load balancing with Docker EE)

Below is an example of how nginx can be used to setup LB. In the following example we have 3 containers, all running inginx. ‘app1’ and ‘app2’ uses nginx base image and simply serves a html file containing the hostname of the container. The 3rd container “nginx” acts as a load balancer. Our aim is that every time we access the load balancer, ‘app1’ and ‘app2’ should serve its content in a round robin fashion. We will of course bring the services together using docker-compose.

```
├── app1
│   └── Dockerfile
├── app2
│   └── Dockerfile
├── docker-compose.yaml
└── nginx
    ├── Dockerfile
```

```
    └── nginx.conf

3 directories, 5 files

# cat app1/Dockerfile
FROM nginx
RUN echo `hostname` > /usr/share/nginx/html/index.html

# cat app2/Dockerfile
FROM nginx
RUN echo `hostname` > /usr/share/nginx/html/index.html

# cat nginx/Dockerfile
FROM nginx
RUN echo > /etc/nginx/nginx.conf
COPY nginx.conf /etc/nginx/nginx.conf

# cat nginx/nginx.conf
http {
upstream backend_hosts {
    server app1;
    server app2;
}
server {
    listen 80;
    server_name localhost;
    location / {
        proxy_pass http://backend_hosts;
    }
}
}
events { }

# cat docker-compose.yaml
version: '3'
services:
  app1:
    build: ./app1
  app2:
    build: ./app2
  nginx:
    build: ./nginx
    ports:
    - "8080:80"
    depends_on:
    - app1
    - app2
```

```
Every time a curl request is sent to load balancer, it automatically load-balances
the request to app1 and app2 in a round robin fashion.

Example, curl -X GET http://[any-of-your-swarm-node]:8080
```

The above is an example of Swarm Mode routing mesh (L4) where in regardless of which swarm node you're accessing and regardless of weather a container for the service you're trying to reach is on that node, the swarm node automatically redirects your request to the container on any node that is running it (purely based on port you're trying to connect to)

UCP of Docker EE extends this feature with an L7 http/https routing mesh, providing the following features.

1. Access services based on domain names. By accessing the domain name, you are redirected served by one of the containers from swarm providing the service mapped to domain name
2. Every time the service is accessed load balancing happens where in a different container is brought in to serve the request.

The above features can be enabled by accessing admin settings in ucp > enable http routing mesh. Once this is done, a service and a network in the name "ucp-hrm" is created. The service redirects http/https requests to the corresponding service and the network is used for routing mesh communication.

Understand and describe the types of traffic that flow between the Docker engine, registry, and UCP controllers

ucp-controller: This is a component of UCP (Universal Control Pane) that runs the UCP web server (GUI for managing docker clusters) available as a part of Docker Enterprise Edition. Ucp-controller runs only on UCP manager nodes.

UCP can be accessed by CLI and the GUI that it exposes. UCP listens for traffic from the following.

Internal: Traffic from outside the cluster via end-user interaction.
External: Traffic arrives from other hosts of same cluster.
Self: Traffic from the same host

Recommended read: Ports used by UCP

The two main aspects of UCP is service discovery and load balancing:

Embedded DNS that provides name resolution functions only for containers connected to same network. For containers not on same network, containers cannot resolve each

other's DNS. In this case, docker engine forwards the DNS query to default DNS server for name resolution.

External load balancing: When ports are published in a swarm, the ports are publised on every node in cluster automatically and the node forwards the traffic to a node that has the service running on it. This is done using the swarm mode routing mesh over the ingress overlay network. This works at L4 transport layer.

L7 Load balancing: UCP will automatically route requests that contain the "*Host*:" in its http header, automatically to the virtual IP of the corresponding services. For this, the service should be created with label *com.docker.lb.hosts*. This enables hostname-based routing.

Recommended read: UCP Service Discovery and Load Balancing

registry: This refers to the DTR or Docker trusted registry used to manage Docker Images via a GUI, available as a part of Docker Enterprise Edition. DTR also allows high availability and load balancing by means of having multiple replicas of the DTR instance. Communication between DTR services happen through an overlay network called *dtr-ol*.

Recommended read: DTR architecture

Deploy a service on a Docker overlay network

NOTE: A network can be created and be used in a service, using the “docker network create” command. Following is an example of how one can create a network and use it in a service or standalone container. We have already discussed creation of different types of network including overlay type, in the previous section. In this section we will create an overlay network that is encrypted and attachable (i.e the network can be used not only in swarm services but also attached to standalone containers)

1. Create an attachable and encrypted overlay network

```
$ docker network create -d overlay --opt encrypted --attachable mynetwork
NOTE: --opt encrypted enabled encryption and –attachable enables the
network to be attached to standalone containers as well as swarm services.
```

2. Create a swarm service with the previously created network

```
$ docker service create --name myservice --network mynetwork --replicas 2
nginx
```

3. Run a standalone container and provide access to the previously created network

```
$ docker run -dt --name myapp --network mynetwork nginx
```

Describe the difference between "host" and "ingress" port publishing mode (Host, Ingress)

Note: the difference in publishing ports in host mode and ingress mode is that, with ingress mode, the port is published on every node participating in swarm cluster. This means, request can be sent to any node in swarm regardless of whether the task is running on a node. The node can redirect the request to one that has the task running, via routing mesh.

Whereas in host mode, the port is published on the nodes in which the tasks are scheduled. Also, the container listens directly on the node's interface on the specified port. This means, unless we have a service

1. Create a service and publish port in ingress mode

```
$ docker service create --name mysvc --publish
published=8080,target=80,mode=ingress nginx
```

NOTE: port 8080 is published on nodes in swarm and traffic is forwarded automatically to nodes running tasks exposing port 80

2. Create a service and publish port in host mode

```
$ docker service create --name mysvc --publish
published=8080,target=80,mode=host nginx
```

NOTE: port 8080 is published only on the nodes in which the tasks are scheduled. One can pass --global to the command so that a task will be placed on every single node in swarm. This way we don't end up not knowing which nodes has the services runningto accept requests on the published port.

Another drawback with this is that a node can run only a single replica or task for that service since it is listening directly on host's interface.