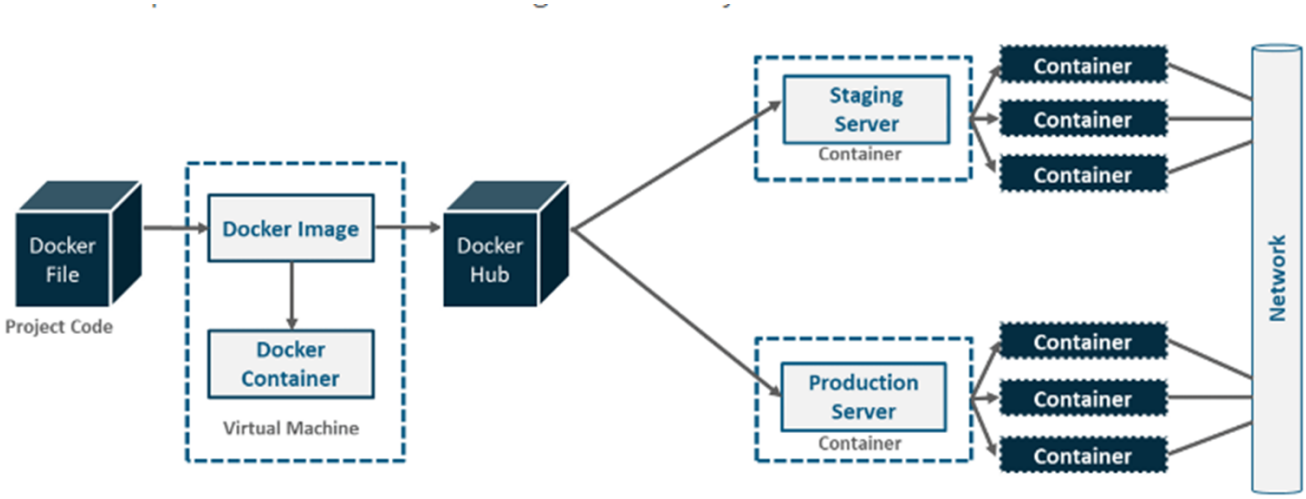
What is Docker Networking?

Docker networking is essential for **container communication**. Different network drivers cater to various scenarios, from single-host communication to multi-host setups. Understanding these networking options helps you design and manage containerized applications effectively.



Explain Basics of Conatiner Networking?

Container Networking Basics:

- Containers on the same network can communicate using IP addresses.
- Containers on different networks or hosts need to expose ports for external communication.
- Ports are exposed using the `-p` flag when running a container.
- The host's IP address is usually used to access exposed ports.

How can we Inspecting Container Networks?

You can inspect a container's networks using the `docker inspect` command. This provides detailed information about the container's networking, IP addresses, network settings, and more.

What is docker0 in terms of Docker Networking?

When Docker is installed, a **default bridge network named docker0 is created**. Each new Docker container is automatically attached to this network, unless a custom network is specified.

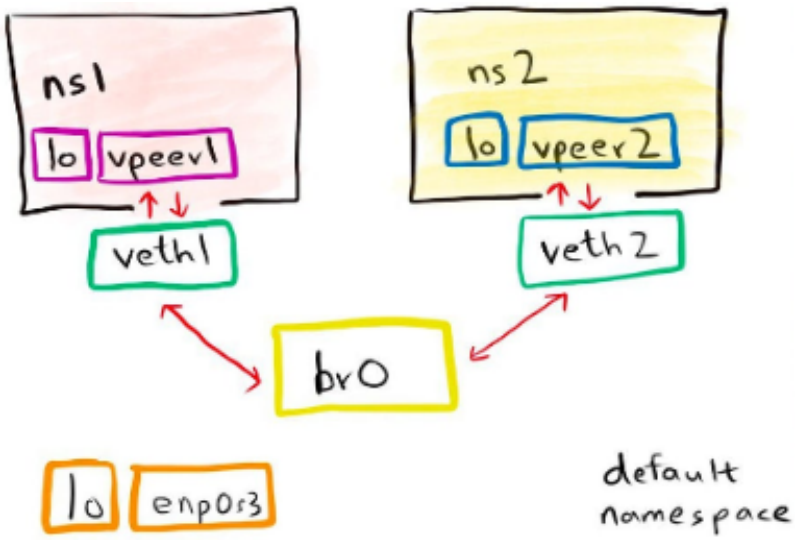
Besides docker0, two other networks get created automatically by Docker:

- **host** (no isolation between host and containers on this network, to the outside world they are on the same network)

- **none** (attached containers run on container-specific network stack)

What is veth?

In Docker networking, a **veth pair** is a key component that facilitates **communication between containers and the host machine**. It's a **virtual Ethernet** pair consisting of two virtual network interfaces that are connected together. One end of the veth pair is placed inside the container's network namespace, while the other end resides in the host's network namespace. This arrangement allows data to flow between the container and the host, enabling network communication.



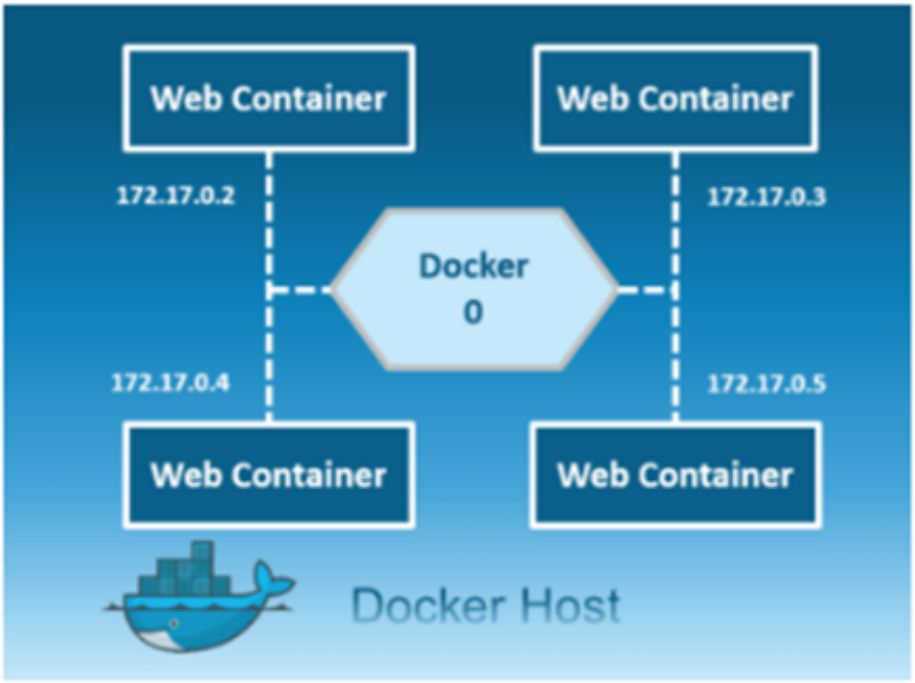
- Docker uses veth pairs to connect the network stack of a container to the network stack of the Docker host.
- A veth pair consists of two virtual interfaces – one sits inside the container network namespace and the other sits on the host.
- Packets sent via one interface are received on the other interface, allowing communication between namespaces.
- One end of the veth pair is attached to the Docker bridge, while the other end is placed inside the container namespace.
- Docker automatically creates and configures veth interfaces when new containers are created.
- Each container has its own veth pair to provide an isolated networking stack connected to the Docker host.
- veth interfaces enable containers to have unique MAC addresses and IPs even though they share the host network stack.
- veth provides high performance packet transport between namespaces since it avoids intermediate hops.
- Overall, veth interfaces are fundamental to how Docker constructs network isolation for containers on the host IP stack.

Explain different types of Networking in Docker?

1. Bridge Network

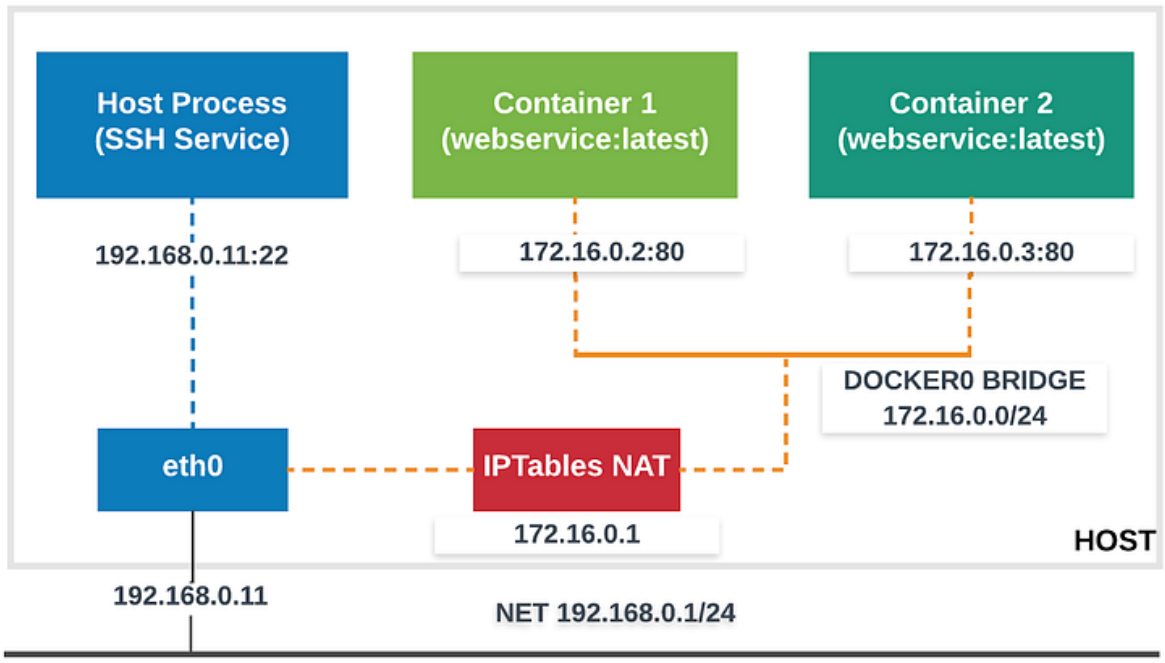
The bridge network is a private default internal network created by docker on the host. So, all containers get an internal IP address and these containers can access each other, using this internal IP. It can not communicate with name of the containers

- The default and most common network driver is the **bridge network**. When a container is created without specifying a network, it's connected to the default bridge network.
- Containers on the same bridge network can communicate using container names as hostnames.
- Containers on different bridge networks or the host's network are isolated from each other.
- This is suitable for single-host scenarios where containers need to communicate with each other.



- **Bridge Network:** A bridge network is a private internal network created by Docker on a host machine. Containers connected to the same bridge network can communicate with each other using their container names as hostnames.
- **Default Bridge:** When you install Docker, it creates a default bridge network named `bridge`. Containers that are started without explicitly specifying a network will automatically connect to the default bridge network.
- **Isolation:** Each bridge network is isolated from other bridge networks and the host's network interfaces. This isolation helps prevent conflicts and unintended communication between containers and between containers and the host.

- **IP Addressing:** Containers on a bridge network are assigned IP addresses from a private IP address range. These addresses are managed by Docker and are not directly accessible from outside the bridge network.
- **DNS Resolution:** Docker provides DNS resolution within the bridge network. Containers can address each other using their container names as hostnames. Docker's built-in DNS server resolves these names to the appropriate IP addresses.
- **Container-to-Container Communication:** Containers connected to the same bridge network can communicate with each other using their container names as hostnames. For example, if you have two containers named `web` and `db` on the same bridge network, the `web` container can communicate with the `db` container using the hostname `db`.
- **Host-to-Container Communication:** Containers on a bridge network can also communicate with the host machine. Docker forwards traffic from the container's IP address to the host's IP address.
- **Exposing Ports:** To allow external traffic to access services within containers, you need to expose and map container ports to host ports. This is typically done using the `-p` or `--publish` option when starting a container.
- **Creating Custom Bridge Networks:** You can create your own custom bridge networks in Docker to isolate groups of containers. This is particularly useful when you want to segment different parts



- **Creating a Custom Bridge Network:**

To create a custom bridge network named `my_network`, you can use the following command:

```
docker network create my_network
```

Connecting Containers to a Custom Bridge Network:

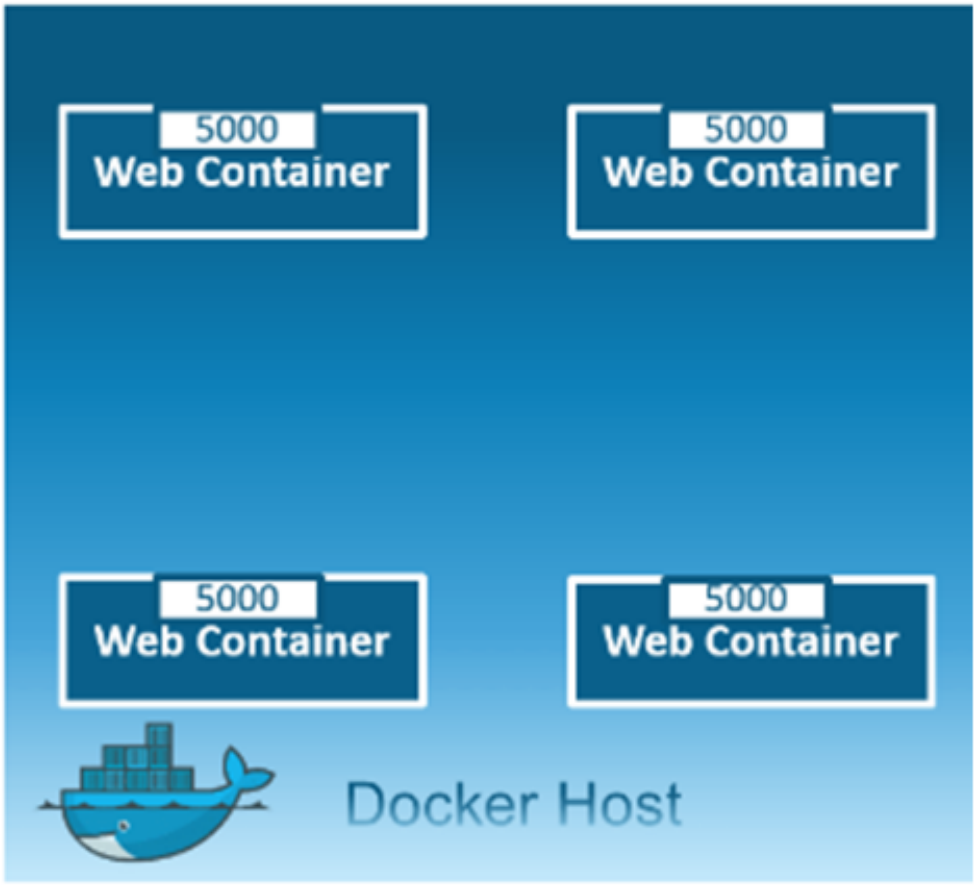
When starting a container, you can specify the network using the `--network` option:

```
docker run --network my_network --name container1 -d image1
docker run --network my_network --name container2 -d image2
```

2.Host Network

This driver removes the network isolation between the docker host and the docker containers to use the host's networking directly. So with this, you will not be able to run multiple web containers on the same host, on the same port as the port is now common to all containers in the host network

- Using the **host network** driver, containers share the host's networking stack. They use the host's IP address and have full access to the host's network interfaces.
- Offers better performance but less isolation, as containers can directly access the host's network.



- **Isolation vs. Host Networking:**

By default, Docker containers are isolated from each other and from the host system using network namespaces. Each container has its own virtual network stack, including its own IP address and port space. This isolation ensures that containers do not interfere with each other or with the host's network.

However, in some scenarios, you might want to bypass this isolation and allow a container to directly use the host's network stack. This can be useful for applications that need direct access to the host's network interfaces or for improving network performance.

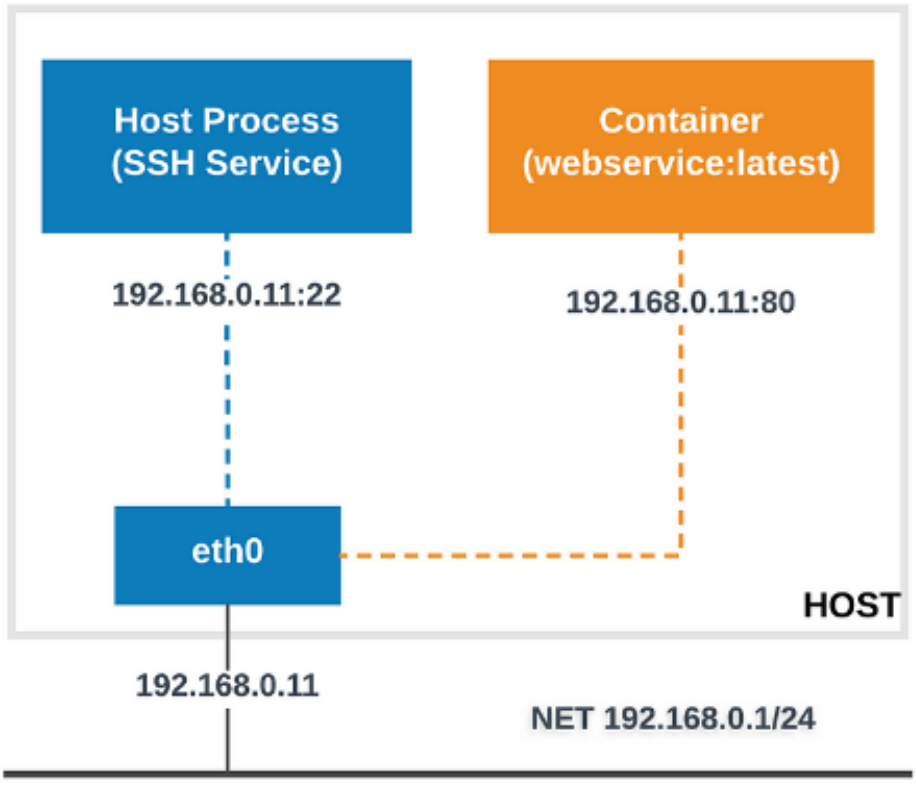
How Host Networking Works?

When a container is started in Host Networking mode, it shares the same network namespace as the host. This means that the container has access to the same network interfaces, IP addresses, and ports as the host system. Any network service running within the container can listen on the same ports that are available on the host.

In Host Networking mode, Docker does not perform any Network Address Translation (NAT) for container traffic. This results in better network performance because packets are not being translated and routed through an additional layer of NAT.

Advantages of Host Networking

- **Performance:** Containers in Host Networking mode can achieve better network performance compared to other networking modes, as they directly use the host's networking stack without NAT overhead.
- **Simplified Configuration:** Applications that require specific ports to be available on the host can benefit from Host Networking mode, as there's no need to map ports between the container and host.
- **Limitations and Considerations:**
- **Security:** Host Networking mode reduces the level of network isolation between the container and the host. This can be a security concern, especially if the containerized application is not properly secured.
- **Port Conflicts:** Since the container shares the same network namespace as the host, any port conflicts between the container and host services can arise. You need to ensure that there are no port clashes to avoid unexpected behavior.
- **Compatibility:** Host Networking mode might not be supported in all environments or with all



How to Use Host Networking?

To start a container in Host Networking mode, you can use the `--network host` option with the `docker run` command:

```
docker run --network host <image_name>
```

For example, if you're using a web server in a container and you want it to be accessible on the same port as the host (e.g., port 80), you can use the following command:

```
docker run --network host -d -p 80:80 <image_name>
```

What is the Use Cases for Host Networking ?

Host Networking mode is typically used in scenarios where direct access to the host's network stack is required. Some use cases include:

- Networking tools or monitoring agents that need access to network interfaces and hardware.
- Applications that require low-latency communication and high network throughput.
- Situations where network port conflicts must be avoided.

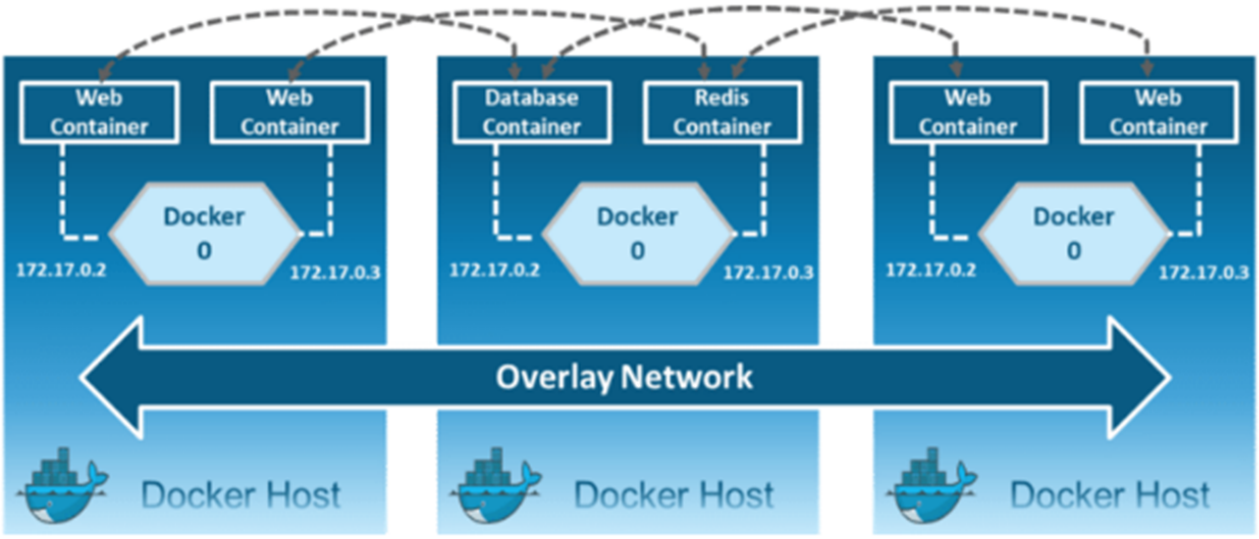
Security Considerations

Keep in mind that Host Networking mode reduces network isolation between containers and the host, which can potentially expose sensitive services on the host to the container. This can increase the risk of security breaches. Therefore, Host Networking should be used judiciously and with appropriate security measures in place.

3. Overlay Network

Creates an internal private network that spans across all the nodes participating in the swarm cluster. So, Overlay networks facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker Daemons

- The **overlay network** driver is used in Docker swarm mode for multi-host communication.
- Containers in different swarm nodes can communicate as if they're on the same network, enabling cross-host communication.



Docker overlay networking is a feature provided by Docker that enables communication between containers across multiple Docker hosts. It's particularly useful in scenarios where you have a swarm of Docker nodes working together as a cluster. Overlay networks allow containers to communicate with each other as if they were on the same network, regardless of the physical hosts they are running on. This is essential for building distributed and scalable applications.

Here's a detailed breakdown of Docker overlay networking:

- **Overlay Networks:**

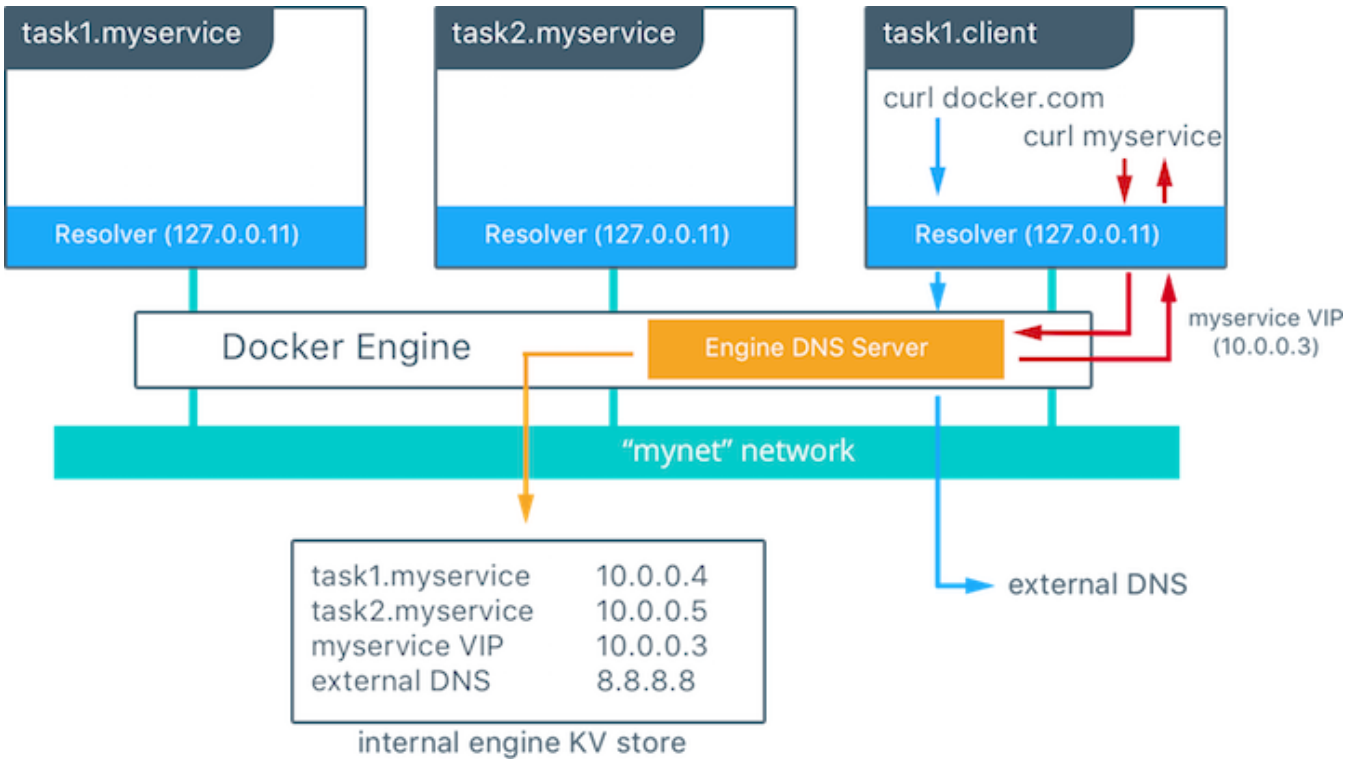
An overlay network is a virtual network that spans multiple Docker nodes, enabling secure communication between containers on different hosts. Overlay networks are an essential component of Docker Swarm mode, which provides built-in orchestration and clustering features. Overlay networks are created on top of the physical network infrastructure and provide a consistent networking model for containers.

- **Docker Swarm Mode:**

Docker Swarm mode allows you to create and manage a cluster of Docker nodes, turning them into a single, cohesive unit called a swarm. Within a swarm, you can create services that define the desired state of an application, including the number of replicas, the network configuration, and other settings. Overlay networks are used to connect these services together.

- **Key Concepts:**

- **Services:** A service defines the tasks to be executed on the cluster, such as running a container. Services can be scaled up or down to distribute the load across the swarm.
- **Tasks:** A task represents a running container associated with a service. Each task is scheduled to run on a specific node within the swarm.
- **Networks:** Overlay networks provide communication between services and tasks across multiple nodes. Containers in different services can communicate with each other using overlay networks.



How Overlay Networks Work?

When you create an overlay network, Docker sets up a VXLAN (Virtual Extensible LAN) tunnel between the nodes in the swarm. VXLAN encapsulates Ethernet frames inside UDP packets, allowing them to be transmitted over the underlying network. Each container in the overlay network is assigned an IP address, and Docker's routing mesh ensures that requests sent to a service IP are properly routed to the correct task, regardless of its location in the swarm.

- **Creating an Overlay Network:**

You can create an overlay network using the `docker network create` command, specifying the `--driver overlay` option. For example:

```
docker network create --driver overlay my-overlay-network
```

- **Connecting Services:**

When you create a service, you can specify which overlay network it should be connected to. All tasks associated with that service will then join the specified network. For example:

```
docker service create --name my-service --network my-overlay-network my-imag
e
```

What are Benefits?

- **Scalability:** Overlay networks enable seamless communication between containers regardless of their physical location, making it easier to scale your application.
- **Load Balancing:** Docker's routing mesh ensures that requests to a service are distributed to the appropriate task, improving load distribution.
- **Service Discovery:** Overlay networks simplify service discovery by allowing you to reference services by name instead of IP addresses.
- **Isolation:** Overlay networks provide isolation between different services, preventing unauthorized access.

What are Use Cases of Overlay Network?

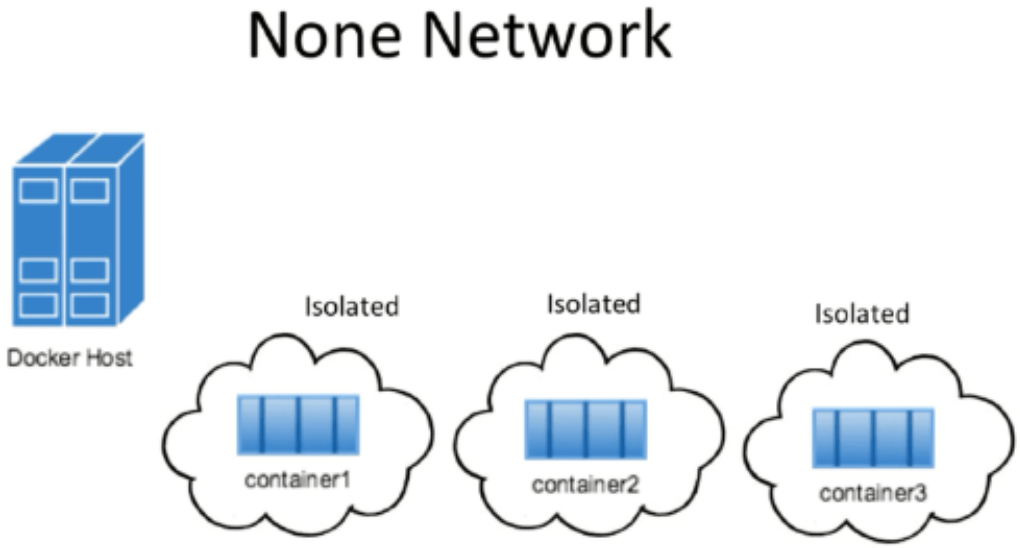
Overlay networks are particularly useful for:

- Microservices architectures: Applications composed of multiple services that need to communicate with each other.
- Stateful applications: Applications that require persistent data storage and need to communicate across different nodes.
- High availability: Services that need to be replicated across multiple nodes for improved availability and fault tolerance.
- **Considerations:**
- Overlay networks introduce some networking overhead due to encapsulation and tunneling.
- Network segmentation and security considerations are important when using overlay networks in a production environment.

4. None Network:

In Docker, networking is a critical component that enables communication and interaction between containers and external networks. Docker provides various networking options to facilitate seamless communication between containers, services, and the external world. One of the networking modes offered by Docker is the "None" network mode. Let's delve into the details of the "None" networking mode in Docker.

- The **none network** driver isolates the container completely from networking. Containers on this network have no external connectivity.



- **None Networking Mode:**

When you run a container in the "None" networking mode, Docker isolates the container from all networking interfaces, including the host's network and other containers. In this mode, the container is effectively disconnected from any network. This can be useful for scenarios where you want to run a container in complete isolation, without any network connectivity.

- **Key Characteristics:**

- **No Network Connectivity:** Containers in the "None" network mode are unable to communicate with the external world, including other containers and the host's network interfaces.

- **Loopback Interface:** Containers in the "None" network mode have only the loopback interface (127.0.0.1) available to them. This means that processes running inside the container can only communicate with themselves.

- **Isolation:** The "None" network mode provides a high degree of isolation for the container. It is suitable for scenarios where you want to restrict network access entirely.

What are the Use Cases?

The "None" networking mode can be useful in specific scenarios:

1. **Security Isolation:** When you want to ensure maximum security and isolation for a container, you can run it in the "None" network mode. This prevents any network communication from or to the container.

2. **Debugging and Testing:** If you are debugging or testing a specific application or software component, isolating it from the network can help prevent any unintended interactions that might interfere with your debugging process.

3. **Container as a Process:** Sometimes, you might want to run a container as a standalone process that doesn't require networking. In such cases, the "None" network mode can be beneficial.

- **Example:**

To run a container in the "None" networking mode, you can use the `--network=none` flag when starting the container. For instance:

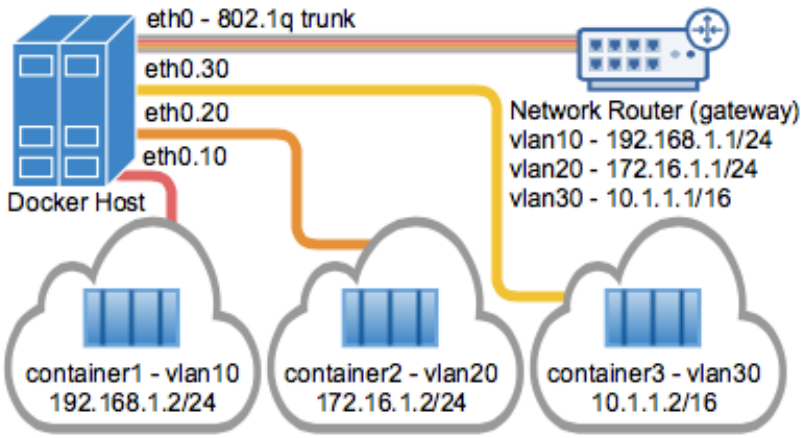
```
docker run --network=none -it alpine
```

This command starts an Alpine Linux container in the "None" network mode. Inside the container, you'll find that network-related commands won't work, and the container won't have any connectivity.

Note: If you need network connectivity for certain tasks within the container, such as installing

4. Macvlan Network :

Docker Macvlan network is a type of Docker network driver that allows you to connect Docker containers directly to a physical network interface on the host machine. This means that containers connected to a Macvlan network will have their own unique MAC address and IP address, and they will be able to communicate with other devices on the physical network, just like any other physical machine.



Macvlan networks are typically used for applications that need to be directly connected to the physical network, such as network monitoring tools, network appliances, and legacy applications. They can also be used to create isolated networks for containers, or to allow containers to communicate with each other without having to go through the Docker host.

To create a Macvlan network, you need to specify the name of the network, the parent interface, the subnet, and the gateway. The parent interface is the physical network interface on the host machine that the containers will be connected to. The subnet and gateway are the network settings for the containers.

How to create a Macvlan network?

```
docker network create --driver macvlan --subnet 192.168.1.0/24 --gateway 19
2.168.1.1 my-macvlan
```

This command will create a Macvlan network named `my-macvlan` with a subnet of `192.168.1.0/24` and a gateway of `192.168.1.1` . The parent interface for this network will be the default network interface on the host machine.

Once you have created a Macvlan network, you can create containers that are connected to it. To do this, you need to specify the

`-network` flag when you create the container.

For example, the following command will create a container named `my-container` that is connected to the `my-macvlan` network:

```
docker run --network my-macvlan my-image
```

The container will be assigned its own unique MAC address and IP address, and it will be able to communicate with other devices on the physical network.

What are the advantages of using Docker Macvlan networks?

- Containers connected to a Macvlan network have their own unique MAC address and IP address, which allows them to communicate with other devices on the physical network just like any other physical machine.
- Macvlan networks are isolated from each other, which means that containers on different Macvlan networks cannot see each other. This can be useful for creating security zones or for isolating different types of applications.
- Macvlan networks can be used to create bridged networks, which means that containers on different Macvlan networks can communicate with each other. This can be useful for applications that need to communicate with each other, but that need to be isolated from the physical network.

Mention some disadvantages of using Docker Macvlan networks?

- Macvlan networks can be more complex to set up than other types of Docker networks.
- Macvlan networks can be less efficient than other types of Docker networks, as they require more resources on the host machine.
- Macvlan networks are not supported on all platforms.

Overall, Docker Macvlan networks are a powerful tool that can be used to connect Docker containers directly to physical networks. They can be used for a variety of applications, including network monitoring, network appliances, and legacy applications. However, they can be more complex to set up

To check network drivers :

docker network ls

```
[root@ip-172-31-42-6 ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c7ed8a4fe6a3        bridge              bridge              local
9f29cd076e3e        host                host                local
82acd3f2020d        none                null                local
```

To check driver details :

docker inspect bridge

```
[root@ip-172-31-42-6 ~]# docker inspect bridge
[
    {
        "Name": "bridge",
        "Id": "c7ed8a4fe6a3b8bc0ce711cc192388999683aba5b52973ef5fe1439826e5ac4f",
        "Created": "2019-06-28T03:17:59.5023952332",
        "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": {
            "19813fa5da9e6747a4888c2d009da814224e24b6f73a1d97c21c7ae1c38eb4cd": {
                "Name": "ubunut-ctr2",
                "EndpointID": "76db907e898cc6b428002f1509e181575b424e6a9815d59472b4972f147e8ad6",
```

To check container details :

docker inspect ubunut-ctr1

```
[root@ip-172-31-42-6 ~]# docker inspect  ubunut-ctr1
[
    {
        "Id": "ee366634a9310bcb34c9519c8a0f296ee02eeb2a723c6288404f27dae941286b",
        "Created": "2019-06-28T14:58:50.4672898522",
        "Path": "/bin/bash",
        "Args": [],
        "State": {
            "Status": "running",
```

Network driver details of the container:

```
"NetworkSettings": {
    "Bridge": "",
    "SandboxID": "1a70877766c1c5f1afd5663010b1893605852cd72b9e283c60ac2c4bd276efaf",
    "HairpinMode": false,
    "LinkLocalIPv6Address": "",
    "LinkLocalIPv6PrefixLen": 0,
    "Ports": {}
```

docker network inspect bridge

```
"ConfigOnly": false,
"Containers": {
    "19813fa5da9e6747a4888c2d009da814224e24b6f73a1d97c21c7ae1c38eb4cd": {
        "Name": "ubunut-ctr2",
        "EndpointID": "76db907e898cc6b428002f1509e181575b424e6a9815d59472b4972f147e8ad6",
        "MacAddress": "02:42:ac:11:00:04",
        "IPv4Address": "172.17.0.4/16",
        "IPv6Address": ""
    }
```

```
"ee366634a9310bcb34c9519c8a0f296ee02eeb2a723c6288404f27dae941286b": {
    "Name": "ubunut-ctr1",
    "EndpointID": "31d3684a05516cce27aa7d84b458300887b549a6ecfb70f00e4df222b19c22d8",
    "MacAddress": "02:42:ac:11:00:03",
    "IPv4Address": "172.17.0.3/16",
    "IPv6Address": ""
}
```

Connect to one of the container :

docker network inspect bridge

```
[root@ip-172-31-42-6 ~]# docker attach ubunut-ctr1
root@ee366634a931:/#
```

Ping the other container with IP Address:

By default ping is not avialable in ubuntu container, install ping

apt-get update

apt-get install iputils-ping

```
root@ee366634a931:/# ping 172.17.0.4
PING 172.17.0.4 (172.17.0.4) 56(84) bytes of data.
64 bytes from 172.17.0.4: icmp_seq=1 ttl=255 time=0.076 ms
64 bytes from 172.17.0.4: icmp_seq=2 ttl=255 time=0.052 ms
64 bytes from 172.17.0.4: icmp_seq=3 ttl=255 time=0.054 ms
64 bytes from 172.17.0.4: icmp_seq=4 ttl=255 time=0.054 ms
^C
```

Ping the other container with name:

```
root@ee366634a931:/# ping ubunut-ctr2
ping: ubunut-ctr2: Name or service not known
```

Create Network :

docker network create --driver=bridge custom-network

```
[root@ip-172-31-42-6 ~]# docker network create --driver=bridge custom-network
69a579f4b2e0756ee1f9dd273387131fe1745eb021859ddaeba6a21674751ce0
```

Check the network

```
[root@ip-172-31-42-6 ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c7ed8a4fe6a3        bridge              bridge              local
69a579f4b2e0        custom-network      bridge              local
9f29cd076e3e        host                host                local
82acd3f2020d        none                null                local
```

Create container with custom network

docker run -dit --name=ubuntu-ctr3 --network=custom-network ubuntu

docker run -dit --name=ubuntu-ctr4 --network=custom-network ubuntu

```
[root@ip-172-31-42-6 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
d482051d201c        ubuntu              "/bin/bash"         2 seconds ago       Up 1 second                             ubuntu-ctr4
8f9791ef0843        ubuntu              "/bin/bash"         59 seconds ago      Up 58 seconds                           ubuntu-ctr3
```

docker inspect custom-network


```
    "ConfigOnly": false,
    "Containers": {
        "8f9791ef0843adc1f0bbd05e3e15d7903f8746363d82f3918f443a7003ebac39": {
            "Name": "ubuntu-ctr3",
            "EndpointID": "bb2444c34a23b2d8bbacf746f5f6737dce45affb8ea3690498ee8a6fb3dbbac1",
            "MacAddress": "02:42:ac:12:00:02",
            "IPv4Address": "172.18.0.2/16",
            "IPv6Address": ""
        },
        "d482051d201c3adbd4eef89323c2bb6a77aee9812131b84030169f1d51a171e5": {
            "Name": "ubuntu-ctr4",
            "EndpointID": "b01ce4b26fcf311b3ec7327e9f154da6833a0bf65082e77da8e1373342ea625d",
            "MacAddress": "02:42:ac:12:00:03",
            "IPv4Address": "172.18.0.3/16",
            "IPv6Address": ""
        }
```


Login to one container and ping other one with IP Address and name

```
root@8f9791ef0843:/# ping 172.18.0.3
PING 172.18.0.3 (172.18.0.3) 56(84) bytes of data.
64 bytes from 172.18.0.3: icmp_seq=1 ttl=255 time=0.073 ms
64 bytes from 172.18.0.3: icmp_seq=2 ttl=255 time=0.060 ms
64 bytes from 172.18.0.3: icmp_seq=3 ttl=255 time=0.059 ms
```

```
root@8f9791ef0843:/# ping ubuntu-ctr4
PING ubuntu-ctr4 (172.18.0.3) 56(84) bytes of data.
64 bytes from ubuntu-ctr4.custom-network (172.18.0.3): icmp_seq=1 ttl=255 time=0.046 ms
64 bytes from ubuntu-ctr4.custom-network (172.18.0.3): icmp_seq=2 ttl=255 time=0.061 ms
64 bytes from ubuntu-ctr4.custom-network (172.18.0.3): icmp_seq=3 ttl=255 time=0.058 ms
```

Change the Network driver for existing containers

Step 1 : Disconnect from existing driver : docker network disconnect bridge 26febb6f4867
Step 2 : Connect to new driver : docker network connect custom-network 26febb6f4867

Note: It is possible only for bridge networks not for others (two different bridge networks)

Remove one or more networks

docker network rm NETWORK

Building Docker Image for Tomcat with Specified War file from Nexus

The Dockerfile that includes the `tomcat-users.xml` configuration and builds the Docker image:

1. **Create a Directory Structure:** Start by organizing your files in a directory structure. Create a directory named `tomcat-config` in your project directory. Inside this directory, place your `tomcat-users.xml` file.

   Your directory structure should look like this:

   ```
   your-project-directory/
   ├── Dockerfile
   ├── tomcat-config/
   │   └── tomcat-users.xml
   ```

2. **Create the Dockerfile:** Create a file named `Dockerfile` in your project directory with the following content:

```
# Use the official Tomcat image as the base
FROM tomcat:9.0

# Set the maintainer label
LABEL maintainer="Prudhvi Vardhan"

# Copy tomcat-users.xml to the appropriate location
COPY ./tomcat-config/tomcat-users.xml $CATALINA_HOME/conf/

# Download and add the specified WAR file from Nexus
ADD http://nexus-url/path/to/your/war-file.war $CATALINA_HOME/webapps/

#Expose Tomcat port
EXPOSE 8080

#Start Tomcat
CMD ["catalina.sh", "run"]
```

Replace `http://nexus-url/path/to/your/war-file.war` with the actual URL of your WAR

```
-- Prudhvi Vardhan (LinkedIn)
```