Networking overview

Container networking refers to the ability for containers to connect to and communicate with each other, or to non-Docker workloads.

Containers have networking enabled by default, and they can make outgoing connections. A container has no information about what kind of network it's attached to, or whether their peers are also Docker workloads or not. A container only sees a network interface with an IP address, a gateway, a routing table, DNS services, and other networking details. That is, unless the container uses the `none` network driver.

Network Drivers:

- Docker supports various network drivers, allowing users to choose the most suitable one for their requirements.
- Examples of network drivers include bridge, overlay, host, macvlan, and ipvlan.

Networking Commands:

- Common Docker networking commands include `docker network create`, `docker network ls`, and `docker network inspect`.
- These commands are used to create, list, and inspect Docker networks, respectively.

Container DNS:

Container DNS, also known as container name resolution, is a feature in Docker that provides automatic DNS resolution between containers. This means that instead of relying on IP addresses, containers can use each other's names as hostnames to communicate within the same Docker network. This feature simplifies container communication and reduces the need to hardcode IP addresses.

Here's how Container DNS works:

1. **Automatic DNS Resolution:**

- When you create a Docker container within a network, Docker automatically assigns a unique name to the container. This name is then registered with Docker's embedded DNS server.

2. **Container Name as Hostname:**
   - Containers can use each other's names as hostnames to communicate. For example, if you have a container named "webapp" and another named "database," the "webapp" container can reach the "database" container using the hostname "database" instead of an IP address.

Example Usage:

- Assume you have two containers, one running a web application and another running a database. The web application container can connect to the database container using its name:
  Bash

```
# Connecting from the web application container to the database container
$ curl http://database:3306
```

3. When Container DNS Works:

- Container DNS works seamlessly when containers are part of the same Docker network (not working with default bridge network, working with custom bridge network). As long as containers are within the same network, they can use each other's names for communication.

4. Network Isolation:

- Container DNS is particularly useful in scenarios where you want to provide network isolation between different services or components of an application. Containers on the same network can communicate with each other using names, but they are isolated from containers in other networks.

5. When Container DNS Might Not Work:

- If containers are not part of the same Docker network, container DNS may not work. Communication via container names relies on Docker's internal DNS service, which is specific to the Docker network.

6. External DNS Resolution:

- Container DNS is mainly designed for communication within a Docker network. If you need to resolve external domain names or communicate with resources outside the Docker environment, you might need additional configurations or rely on the host machine's DNS settings.

7. Custom Bridge Networks:

- When using custom bridge networks, containers can communicate with each other using names within the same network. However, containers in different custom bridge networks might not resolve each other's names.

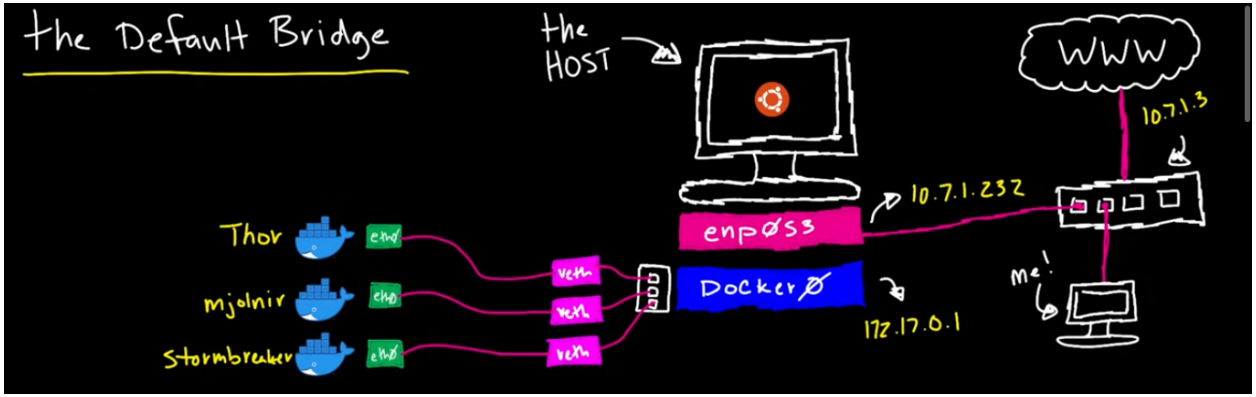
In summary, Container DNS is a convenient feature for facilitating communication between Docker containers within the same network by using container names as hostnames.

Types of Docker Networks :

Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

1. **Default Bridge Network**
2. **Custom Bridge Network (We can use container name as a dns)**
3. **Host Network: (Uses Host Network)**
4. **None Network:**
5. **Macvlan**
6. **Ipvlan**
7. **Overlay (Suitable for multi host docker like docker swarm)**

1. Default Bridge Network :



- **Description:**
  - The default network mode for Docker containers.
  - Provides communication between containers running on the same host.
  - Each container in a bridge network gets its own IP address.
  - Containers will not communicate using each other name or dns
  - Require to bind host port with container port to enable external access to the services
- **Use Cases:**
  - Running multiple containers on a single host that need to communicate with each other.

Current State :

```
docker ps  -a
ip a
```

```
vishal@vishal-HP-245-G8:~$ docker ps  -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
vishal@vishal-HP-245-G8:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eno1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc fq_codel state DOWN group default qlen 1000
    link/ether e0:70:ea:e2:26:4e brd ff:ff:ff:ff:ff:ff
    altname enp1s0
3: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether f8:89:d2:50:96:49 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.8/24 brd 192.168.1.255 scope global dynamic noprefixroute wlp2s0
       valid_lft 68048sec preferred_lft 68048sec
    inet6 fe80::94a2:1cab:31e:696f/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:42:48:92:af 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
```

The `docker0` bridge network is a virtual network interface that is created on the host machine during the installation of Docker. It acts as a bridge between the containers running on the host, allowing them to communicate with each other. The `docker0` interface is a part of the host's network stack.

```
vishal@vishal-HP-245-G8:~$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
bc4acb59c3dd   bridge    bridge    local
415f6c9bf0cf   host      host      local
5c5e42e405f7   none      null      local
vishal@vishal-HP-245-G8:~$
```

```
docker run -itd --rm --name container1  nicolaka/netshoot
docker run -itd --rm --name container2  nicolaka/netshoot
```

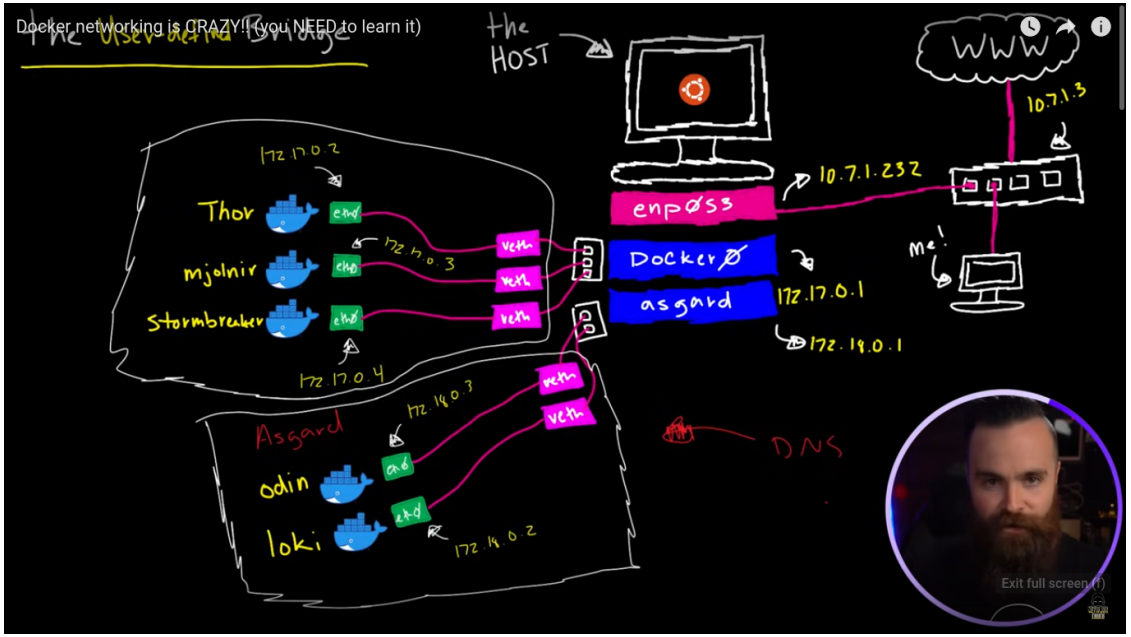
```
vishal@vishal-HP-245-G8:~$ docker run -itd --rm --name container1  nicolaka/netshoot
Unable to find image 'nicolaka/netshoot:latest' locally
latest: Pulling from nicolaka/netshoot
8a49fdb3b6a5: Pull complete
f08cc7654b42: Pull complete
bacdb080ad6d: Pull complete
df75a2676b1d: Pull complete
d30ac41fb6a9: Pull complete
3f3eebe79603: Pull complete
086410b5650d: Pull complete
4f4fb700ef54: Pull complete
5a7fe97d184f: Pull complete
a6d1b2d7a50e: Pull complete
599ae1c27c63: Pull complete
dd5e50b27eb9: Pull complete
2681a5bf3176: Pull complete
2517e0a2f862: Pull complete
7b5061a1528d: Pull complete
Digest: sha256:a7c92e1a2fb9287576a16e107166fee7f9925e15d2c1a683dbb1f4370ba9bfe8
Status: Downloaded newer image for nicolaka/netshoot:latest
0fd9b02ca78a3ff786adfa17ba4db49007ba30f94bbc4495b80d4d89e55f40b4
vishal@vishal-HP-245-G8:~$ ^C
vishal@vishal-HP-245-G8:~$ docker run -itd --rm --name container2  nicolaka/netshoot
4617f36635034c826068f65a84675a373d6a06586bc87730655523d0cdeb1a40
vishal@vishal-HP-245-G8:~$
```

```
ip a
```

```
    inet6 fe80::94a2:1cab:31e:696f/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:42:48:92:af 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
    inet6 fe80::42:42ff:fe48:92af/64 scope link
       valid_lft forever preferred_lft forever
8: veth310fdb4@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP gro
    link/ether 16:c8:2c:06:65:4b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::14c8:2cff:fe06:654b/64 scope link
       valid_lft forever preferred_lft forever
10: vethb0dcb15@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP gr
    link/ether 42:3d:db:7d:62:ee brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::403d:dbff:fe7d:62ee/64 scope link
       valid_lft forever preferred_lft forever
vishal@vishal-HP-245-G8:~$ bridge link
8: veth310fdb4@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master docker0 state forwarding priority
10: vethb0dcb15@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master docker0 state forwarding priority
vishal@vishal-HP-245-G8:~$
```

In Docker, the **veth** interface is typically connected to the Docker bridge network. The **Docker bridge network** is a virtual bridge that facilitates communication between containers on the same host. When a container is started, a pair of **veth** interfaces is created—one end inside the container's network namespace, and the other end in the host's network namespace.

2. **Custom Bridge Networks:**
    - **Description:**
        - Created using the `docker network create` command with the `bridge` driver.
        - Provides isolation between containers on different networks.
        - Containers within a custom bridge network can communicate with each other using container names or IP addresses.
        - Require to bind host port with container port to enable external access to the services
    - **Use Cases:**
        - Deploying applications with multiple components that need to communicate while maintaining network isolation.



We can Create Custom Networks. In above image we have created different network. So , Container Running in custom Bridge Network , Generated separate Veth that will communicate with custom Bridge Nework or User Defined Bridge Network.

Custom Bridge Network Provide Isolation from default bridge network or any other Network as its Custom Space.

```
docker network ls
docker network create -d bridge custom_bridge    .... -d = driver ; bridge = driver name
```

```
vishal@vishal-HP-245-G8:~$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
bc4acb59c3dd   bridge    bridge    local
415f6c9bf0cf   host      host      local
5c5e42e405f7   none      null      local
vishal@vishal-HP-245-G8:~$
vishal@vishal-HP-245-G8:~$ docker network create -d bridge custom_bridge
beb9bb4a4cfd8b63ad56f0dd73366ad33382a5bff1b56b4ffc8253fb948e409e
vishal@vishal-HP-245-G8:~$
vishal@vishal-HP-245-G8:~$ docker network ls
NETWORK ID     NAME            DRIVER    SCOPE
bc4acb59c3dd   bridge          bridge    local
beb9bb4a4cfd   custom_bridge   bridge    local
415f6c9bf0cf   host            host      local
5c5e42e405f7   none            null      local
vishal@vishal-HP-245-G8:~$
```

Container 1 ,2 running on default bridge network , you can also check the ip assign from **docker 0 eth**

```
ip a | grep docker0
docker inspect container1
```

```
vishal@vishal-HP-245-G8:~$ ip a | grep docker0
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
8: veth310fdb4@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
10: vethb0dcb15@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
vishal@vishal-HP-245-G8:~$
```

```
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID": "bc4acb59c3dd18d5d12a8ee126da2bee887589164b38b8f017d195cef384ccbf",
                    "EndpointID": "e4bb7819110285154dff5f1ff777992daa815ca9e646bd94fed78655534a94fb",
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
```

Run Container 3,4 on custom bridge network that we were created :

```
docker network ls
docker run -itd --rm --network custom_bridge --name container3  nicolaka/netshoot
docker run -itd --rm --network custom_bridge --name container4  nicolaka/netshoot
```

```
vishal@vishal-HP-245-G8:~$ docker network ls
NETWORK ID     NAME       DRIVER    SCOPE
bc4acb59c3dd   bridge     bridge    local
415f6c9bf0cf   host       host      local
5c5e42e405f7   none       null      local
vishal@vishal-HP-245-G8:~$
```

```
vishal@vishal-HP-245-G8:~$ docker run -itd --rm --network custom_bridge --name container3  nicolaka/netshoot
bcb5c8ca7a8d4a3a29533697299c6d817742e5377f1b27ee1c2f721d57dcddf7
vishal@vishal-HP-245-G8:~$ docker run -itd --rm --network custom_bridge --name container4  nicolaka/netshoot
1f9ac243ae4c414527dc9c8083896b1291490a7e69b0972c7cbe618b5d44f4cf
```

When we Check container 3,4 addresses , **br-beb9bb4a4cfd** it is custom bridge that we were created

```
ip a
bridge link
```

```
       valid_lft forever preferred_lft forever
12: br-beb9bb4a4cfd: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:58:1f:4d:1d brd ff:ff:ff:ff:ff:ff
    inet 172.19.0.1/16 brd 172.19.255.255 scope global br-beb9bb4a4cfd
       valid_lft forever preferred_lft forever
    inet6 fe80::42:58ff:fe1f:4d1d/64 scope link
       valid_lft forever preferred_lft forever
14: vetha4190d0@if13: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-beb9bb4a4cfd state UP group default
    link/ether ca:5b:da:16:d9:18 brd ff:ff:ff:ff:ff:ff link-netnsid 2
    inet6 fe80::c85b:daff:fe16:d918/64 scope link
       valid_lft forever preferred_lft forever
16: veth557a90d@if15: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-beb9bb4a4cfd state UP group default
    link/ether ca:84:a9:b7:4d:82 brd ff:ff:ff:ff:ff:ff link-netnsid 3
    inet6 fe80::c884:a9ff:feb7:4d82/64 scope link
       valid_lft forever preferred_lft forever
vishal@vishal-HP-245-G8:~$ bridge link
8: veth310fdb4@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master docker0 state forwarding priority 32 cost 2
10: vethb0dcb15@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master docker0 state forwarding priority 32 cost 2
14: vetha4190d0@if13: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br-beb9bb4a4cfd state forwarding priority 32 cost 2
16: veth557a90d@if15: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br-beb9bb4a4cfd state forwarding priority 32 cost 2
vishal@vishal-HP-245-G8:~$
```

```
docker inspect container3
```

```
                "Aliases": [
                    "bcb5c8ca7a8d"
                ],
                "NetworkID": "beb9bb4a4cfd8b63ad56f0dd73366ad33382a5bff1b56b4ffc8253fb948e409e",
                "EndpointID": "ccb122b4bd7aaee9b0658f98185a08b1a14bed736e584a352f5a9ff3160e4031",
                "Gateway": "172.19.0.1",
                "IPAddress": "172.19.0.2",
                "IPPrefixLen": 16,
                "IPv6Gateway": "",
                "GlobalIPv6Address": "",
                "GlobalIPv6PrefixLen": 0,
                "MacAddress": "02:42:ac:13:00:02",
```

As you can see, it has ip address from newly created custom bridge network.

If I ping container 3 from container 4 , it will work over container ip or container name so called dns ! .

But if I ping container 1 from container 4, then it will not work , because container 1,2 are in different bridge network i.e., default bridge network

```
vishal@vishal-HP-245-G8:~$ docker exec -it container4 /bin/bash
1f9ac243ae4c:~# ping container3
PING container3 (172.19.0.2) 56(84) bytes of data.
64 bytes from container3.custom_bridge (172.19.0.2): icmp_seq=1 ttl=64 time=0.233 ms
64 bytes from container3.custom_bridge (172.19.0.2): icmp_seq=2 ttl=64 time=0.098 ms
^Z
[1]+  Stopped                 ping container3
1f9ac243ae4c:~# ping container1
ping: container1: Try again
1f9ac243ae4c:~#
```

3. **Host Network:**
    - **Description:**
        - Shares the network namespace with the host, bypassing Docker's network isolation.
        - Containers on the host network can directly access ports on the host.
    - **Use Cases:**
        - Situations where you want containers to have the same network namespace as the host for better performance.

```
docker network ls
docker run -itd --rm --name host-5 --network host nginx
ip a
```

```
vishal@vishal-HP-245-G8:~$ docker network ls
NETWORK ID     NAME            DRIVER    SCOPE
bc4acb59c3dd   bridge          bridge    local
beb9bb4a4cfd   custom_bridge   bridge    local
415f6c9bf0cf   host            host      local
5c5e42e405f7   none            null      local
vishal@vishal-HP-245-G8:~$ docker run -itd --rm --name host-5 --network host nginx
cf3eb0e051b9902460ca83e1b0875beb20c8bd12df168d1a8407e1b24165cc00
vishal@vishal-HP-245-G8:~$
```

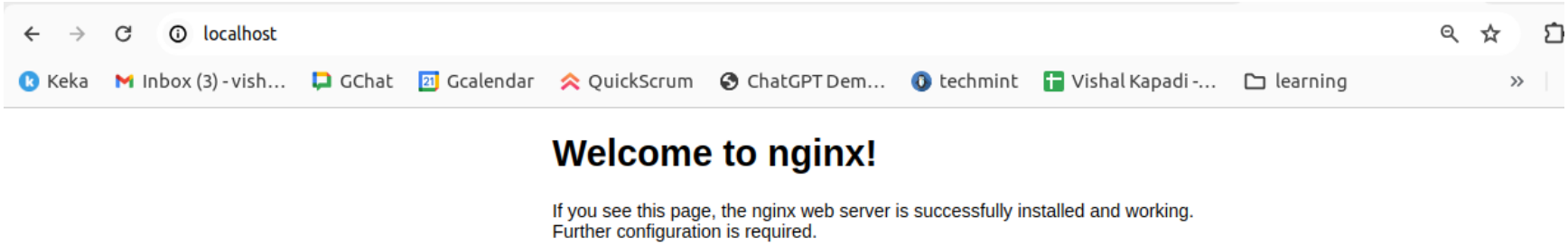
Now Check container (host-5) ip :

```
docker inspect host-5
```

```
            "MacAddress": "",
            "Networks": {
                "host": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID": "415f6c9bf0cfe9ddd9478398df7e0356c227f58154657ccc329315fb7708557f",
                    "EndpointID": "aefd7ca41be69e42edbff01049e4d2548e4442c112cf534808993ed94a05ac00",
                    "Gateway": "",
                    "IPAddress": "",
                    "IPPrefixLen": 0,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "",
                    "DriverOpts": null
                }
```


It is bind with host eth not with docker0 eth.

In host network , container treated as a just like normal application on host machine. We dont need to expose its port. We will have direct access to the external as its using same interface as host.



Nginx running on default port 80 , so I can able to access nginx service running inside container (host-5) without exposing its port. By default due to host network it has gain access to the host Eth .If container want to access 80 then it can directly access host 80.

4. None Network:

- **Description:**
  - Containers in this mode have no network interfaces.
  - Containers are completely isolated from the network.
  - Useful when a container doesn't need network access.
- **Use Cases:**
  - Isolating a container from the network entirely.
  - Security-sensitive environments where network access is not required.

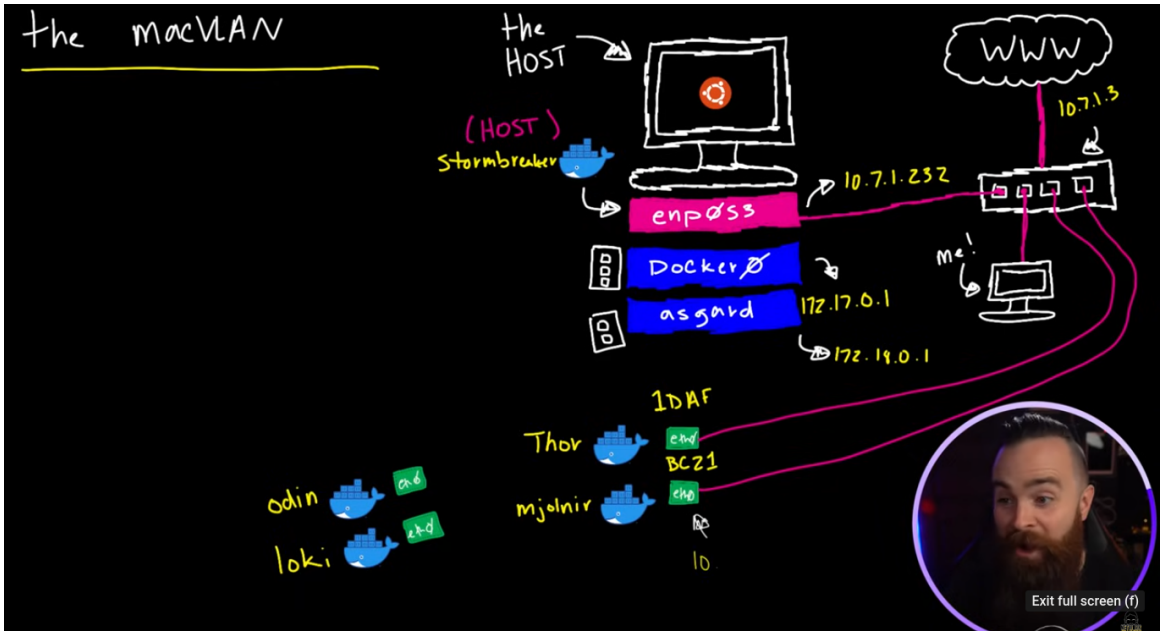
5. Overlay Network (Suitable for multiple docker hosts):

- **Description:**
  - Essential for container orchestration systems like Docker Swarm and Kubernetes.
  - Enables communication between containers running on different hosts in a cluster.
  - Docker uses the VXLAN protocol to create overlay networks.
- **Use Cases:**
  - Deploying and managing containerized applications in a distributed and scalable environment.

6. Macvlan:

- **Description:**
  - Containers in a `macvlan` network are integrated directly into the host's physical network.
  - They can communicate as if they were physical devices on the same network.
  - Containers in a `macvlan` network can be assigned IP addresses on the same subnet as the physical network.
  - Unlike the default bridge network, where containers are assigned IP addresses from a private subnet, `macvlan` containers use IPs from the physical network.
  - This provides more direct visibility and interaction with other devices on the same subnet.
- **Use Cases:**
  - `macvlan` networks are useful when containers need direct integration into the physical network, especially for scenarios where containers behave like physical devices.

- Common use cases include scenarios where containers need to interact with devices or services that rely on MAC address-based access controls.



In above image thor and mjolnir container directly connected to the router i.e, physical network.

Command to Create a Macvlan Network:

- Use the following command to create a `macvlan` network:
  bash

```
docker network create -d macvlan \
  --subnet=<subnet> \
  --gateway=<gateway> \
  --ip-range=<ip-range> \
  -o parent=<parent-interface> \
```

```
  <network-name>
```

- Replace placeholders with your specific values.

Command to Run Container on Macvlan Network:

- Use the following command to run a container on a `macvlan` network:
  bash

```
docker run -d --network <network-name> --name <container-name> <image-name>
```

Replace placeholders with your specific network, container, and image names.

7. Ipvlan:

IPvlan is a network driver in Docker that offers a unique approach to container networking. It provides high performance, simplicity, and advanced networking features.

Key Characteristics:

- **Directly attaches containers to a physical network interface:** This eliminates the overhead of a Linux bridge, leading to reduced latency and better throughput.
- **Leverages lightweight Linux kernel features:** It uses IPVLAN L2 or L3 modes for efficient traffic handling.
- **Two modes of operation:**
  - **L2 mode (default):** Containers share the same MAC address as the parent interface, appearing as virtual interfaces on the same subnet.
  - **L3 mode:** Containers have their own unique IP addresses, allowing for separate subnets and routing within the host.

Advantages:

- **Performance**: Bypass of bridge overhead for lower latency and higher throughput.
- **Simplicity**: Fewer networking layers for easier configuration and management.
- **External access**: Containers can be directly accessed from the external network without port mapping.

Use Cases :

- `ipvlan` is useful when you want containers to be part of the same IP subnet as the host but with individual MAC and IP addresses.
- It provides direct access to the host network, allowing containers to communicate with other devices on the same subnet.
- Suitable for scenarios where containers need to share the host's network configuration for simplicity or compatibility.

Create an IPvlan Network:

- Use the `docker network create` command with the `--driver ipvlan` option to create an `ipvlan` network.
- Specify the parent network interface using the `-o parent=<parent-interface>` option.

- Choose a name for your `ipvlan` network.

Example:

```
docker network create -d ipvlan \
  --subnet=<subnet> \
  -o parent=<parent-interface> \
  <network-name>
```

Run a Container on IPvlan Network:

- Start a Docker container and connect it to the `ipvlan` network.
- Use the `--network` option to specify the `ipvlan` network by name.
- Replace placeholders with your specific network, container, and image names.

Example:

```
docker run -d --network <network-name> --name <container-name> <image-name>
```

==========

\==========

Credits:

- Few Images Taken From : NetworkChuck's youtube Video.
- Docker Networking Image Taken from : ostechnix
- Other things done by myself (vishalk17)

\==========

\========== ⭐ Connect With me ⭐ ========== 👉 Telegram Acc : https://t.me/vishalk17

👉 Telegram DevOps Channel : https://t.me/vishalk17_devops (Interview Qs and Other DevOps Material)

👉 Telegram DevOps Discussion Group : https://t.me/devops_discussion

👉 Linkedin: https://www.linkedin.com/in/vishal-kapadi/

👉 My Github Acc : https://github.com/vishalk17
👉 My Github DevOps Repo : https://github.com/vishalk17/devops (pdf Notes and source code)

👉 My YouTube Channel : https://www.youtube.com/@vishalk17

==========