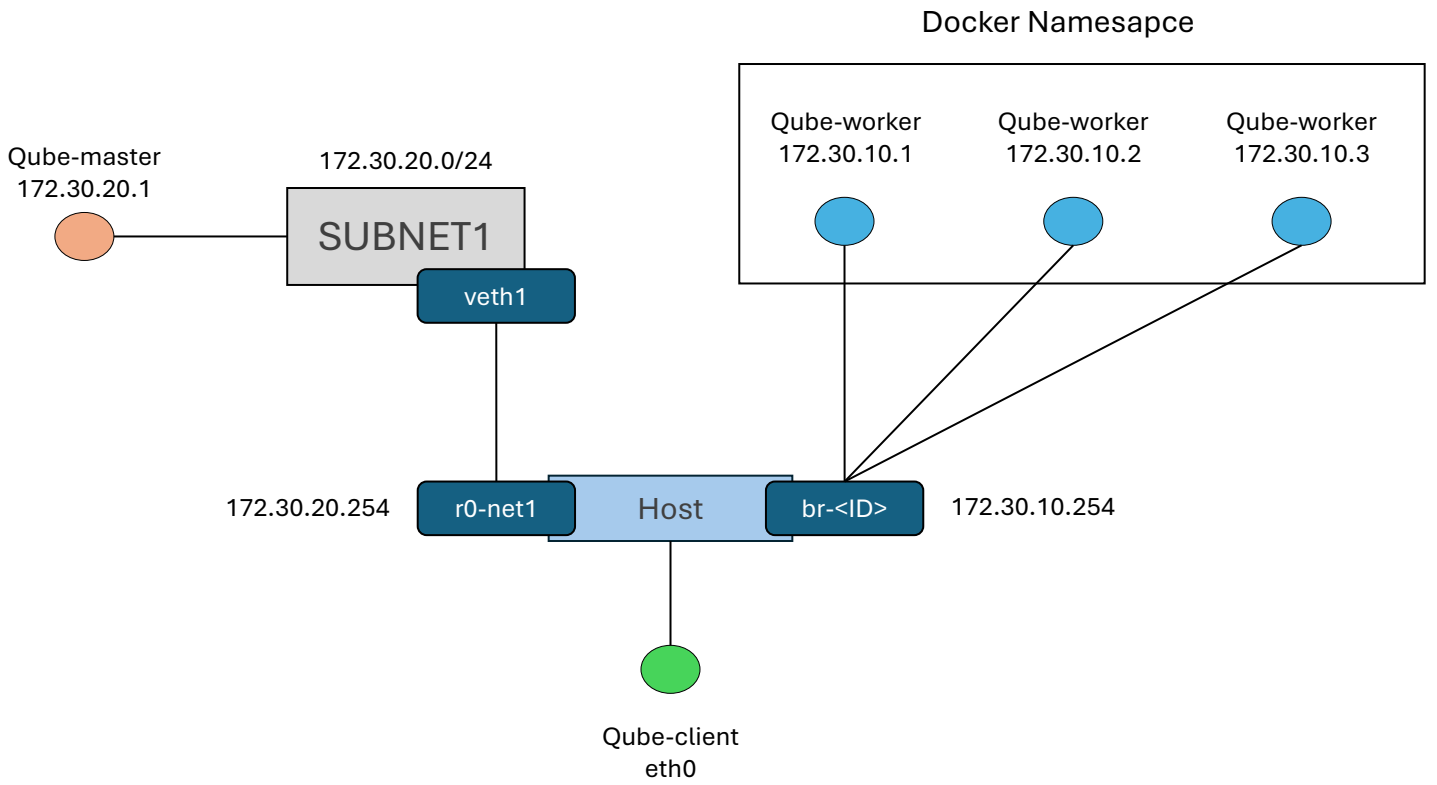
Network Configuration Example



Network Configuration Example

- In the previous slide a possible network configuration has been presented
- The idea is to have the *qube-manager* (master) inside a different subnet as well as all the different *qube-worker*
- Although, it is not needed to test if the tool is working or not, since it should work independently from the network environment
- Finally, also the *qube-client* can be either in the same network of the manager or in a different subnet
  - In this case it is in a different subnet, the root namespace that will have different IP addresses
- Let's see how it is possible to setup such a network configuration
- These are the most important steps
  1. **Creation of the three docker containers with correct subnet and gateway configuration**
  2. **Creation of the net1 network namespace and virtual ethernet pair**
  3. **Setup routing and iptables rules to allow traffic from net1 to the docker network**

Network Configuration Example Step [1] – Docker Compose

- The first step is to create the three containers
- We will use *docker compose* utility to easily setup three containers sharing the same image and subnet
- The content of the *docker-compose.yml* file will look like the following

```
version: '3.8'
name: disqube-example
services:
  qube-client:
    image: alpine:latest
    command: tail -f /dev/null
    networks:
      - qubenet
    deploy:
      replicas: 3
networks:
  qubenet:
    driver: bridge
    ipam:
      config:
        - subnet: 172.30.10.0/24
          gateway: 172.30.10.254
```

Once the *docker-compose* file has been saved, it is possible to run docker compose with the command:

```
docker compose up -d
```

```
[+] Building 0.0s (0/0)                                    docker:default
[+] Running 3/3
 ✔ Container disqube-example-qube-client-3  Created                  0.1s
 ✔ Container disqube-example-qube-client-1  Created                  0.1s
 ✔ Container disqube-example-qube-client-2  Created                  0.1s
Attaching to disqube-example-qube-client-1, disqube-example-qube-clien
t-2, disqube-example-qube-client-3
```

Network Configuration Example Step [1] – Docker Compose

- We can easily see that there are now three running container

```
CONTAINER ID   IMAGE           COMMAND                  CREATED              STATUS              PORTS     NAMES
aa0d6f217fc7   alpine:latest   "tail -f /dev/null"      About a minute ago   Up About a minute             disqube-example-qube-client-3
26141f8365d4   alpine:latest   "tail -f /dev/null"      About a minute ago   Up About a minute             disqube-example-qube-client-1
3ee95465f21a   alpine:latest   "tail -f /dev/null"      About a minute ago   Up About a minute             disqube-example-qube-client-2
```

- Inspecting the custom network created when running *docker compose* we can see:
  - There are three containers attached to the network
  - Ip addresses of the three container

```
NETWORK ID     NAME                          DRIVER    SCOPE
da80334dc33f   bridge                        bridge    local
9655792f1145   disqube-example_qubenet       bridge    local
7d40cf305b09   host                          host      local
32ee6360c74f   none                          null      local
```

```
└─$ docker network inspect disqube-example_qubenet | \
pipe> jq '.[] | .Containers[] | "\(.Name) \(.IPv4Address)"'
"disqube-example-qube-client-1 172.30.10.2/24"
"disqube-example-qube-client-2 172.30.10.3/24"
"disqube-example-qube-client-3 172.30.10.1/24"
```

Network Configuration Example Step [1] – Docker Compose

- Another important aspect is what docker creates on the docker host, network interface-level
- We already know that there is a default docker bridge «installed» called *docker0*
- However, when creating a custom network, based on bridge, a new bridge interface is created
- Moreover, for each new container created attached on that network a virtual ethernet pair is created
  - Obviously, each host-end veth pair the bridge network is set as *master*
  - So that, all containers belonging to the same subnet, can communicate directly
- Doing a simple *ip a* command will show this setting

```
27: br-9655792f1145: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:91:4a:15:fd brd ff:ff:ff:ff:ff:ff
    inet 172.30.10.254/24 brd 172.30.10.255 scope global br-9655792f1145
       valid_lft forever preferred_lft forever
    inet6 fe80::42:91ff:fe4a:15fd/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
29: veth5e63776@if28: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-9655792f1145 state UP group default
    link/ether ee:75:7e:38:23:3e brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::ec75:7eff:fe38:233e/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
31: vethed84162@if30: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-9655792f1145 state UP group default
    link/ether ce:41:16:ed:55:fe brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::cc41:16ff:feed:55fe/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
33: vethb36bd2d@if32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-9655792f1145 state UP group default
    link/ether fa:5f:5d:88:ba:0e brd ff:ff:ff:ff:ff:ff link-netnsid 2
    inet6 fe80::f85f:5dff:fe88:ba0e/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
```

Network Configuration Example Step [2/3] – Custom Network Namespace

- After creating the docker containers, the following step is to create the *qube-master* environment
- In order to make this node beloging to a different subnet, a custom network namespace and veth-pair will be created
- In general, it is the same procedure accomplished by docker when running a container
  - Each container has its own network namespace and veth pair connecting the namespace to the root one and to the bridge interface
- Notice that, in order to obtain our goal, all the following operations must be run as *sudo* user
- The following script will create the namespace, veth-pair, set up routing and iptable rules (not really needed)

```
ip netns add net1
ip link add veth1 type veth peer name r0-net1
ip link set veth1 netns net1
```

```
ip netns exec net1 ip addr add 172.30.20.1/24 dev veth1
ip netns exec net1 ip link set veth1 up
ip netns exec net1 ip link set lo up
```

```
ip link addr add 172.30.20.254/24 dev r0-net1
ip link set r0-net1 up
ip netns exec net1 ip route add default via 172.30.10.254
```

```
sysctl –w net.ipv4.ip_forward=1
iptables –A FORWARD –i r0-net1 –o br-9655792f1145 –j ACCEPT
iptables –t nat –A POSTROUTING –s 172.30.20.0/24 ! –o r0-net1 –j MASQUERADE
```

On the root namespace, this sequence of commands create the network namespace net1 and the veth-pair (veth1, r0-net1) and assign one of the end of the virtual cable, i.e., veth1, to the custom network namespace net1.

This sequence of commands assign to veth1 inside the net1 namespace an IP address of the subnet 172.30.20.0/24 and brings both veth1 and loopback interfaces up and running.

This sequence of command sets the ip address of r0-net1 acting as a gateway for veth1 in net1, brings it up and set the default route required for packets generated from veth1 to reach the root namespace, the host.

This sequence of command sets the IPv4 forwarding on the host, so that it can applies the FORWARD chain in iptables, and add iptables rules for successfully reaching the docker containers behind the bridge interface.

Network Configuration Example Step [2/3] – Custom Network Namespace

- After running the script we now should be able to perform three kind of communications:
  1. Container-to-Host Communication (and vice-versa)
  2. Host-to-net1 namespace Communication (and vice-versa)
  3. Container-to-net1 namespace Communication (and vice-versa)
- It might seem that the third type of communication comes directly from the first two
- It is not always true. It depends on the default policy of the FORWARD chain
  - If it is to DROP any packet not maching any of the other conditions this communication will not happen
  - If it is to ACCEPT, then we can also avoid inserting the iptables roules
- In general, adding explicit iptable rules I think it is a good practice, making the communication transparent to the user
- At this point the network setup is done and we can start these three entity communicate.