

# Containerization (Contd.)

**Simplifying *Application Deployment* with Containers**

# Docker - Networking

# Multi-Container Apps

- The power of Docker is realized when you compose your apps out of multiple containers.
- Docker provides two features:
  1. Docker Networking – **A framework for complex network namespace setup**
  2. Docker Compose -
    - a) **Allows specification and setup of multi-container applications, and**
    - b) **Establishes the network communication between them**

# Docker Networking

- By default Docker comes with the following networks:

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER
7fca4eb8c647	bridge	bridge
9f904ee27bf5	none	null
cf03ee007fb4	host	host

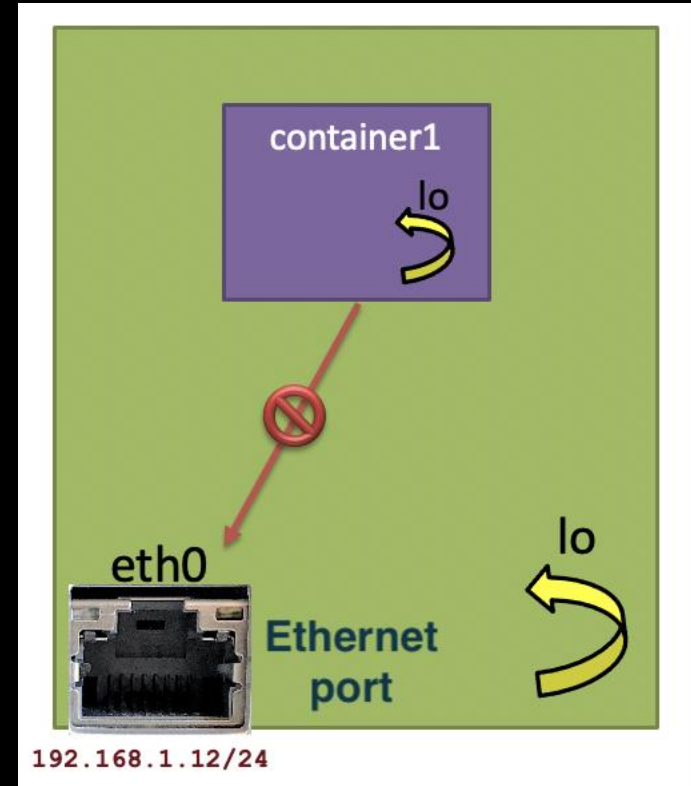
- Network Driver (Types):
  1. None - Its **container has no networking capabilities**
  2. Host - Its **container is not placed in a new network namespace.**
  3. **Bridge (default) - container is attached to a “bridge” (virtual bridge interface).**

# The “none” Network

```
$ docker run \
```

```
--network=none ...
```

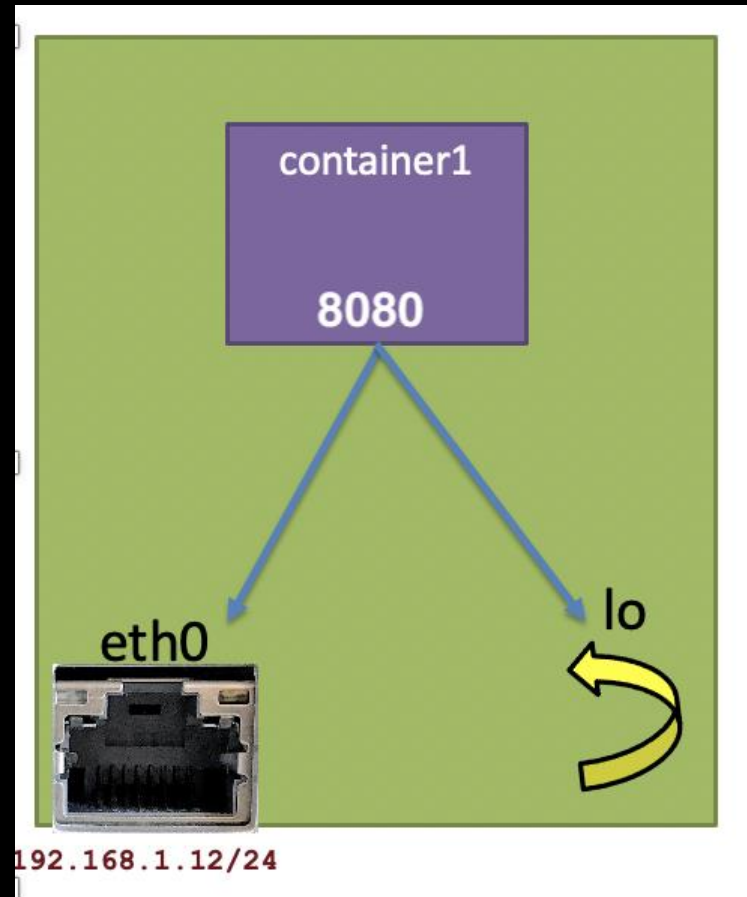
- **Container is in a new network namespace.**
- **Local loopback is enabled.**
- **No other network interfaces are supplied.**



# The “host” Network

```
$ docker run \  
  --network=host ...
```

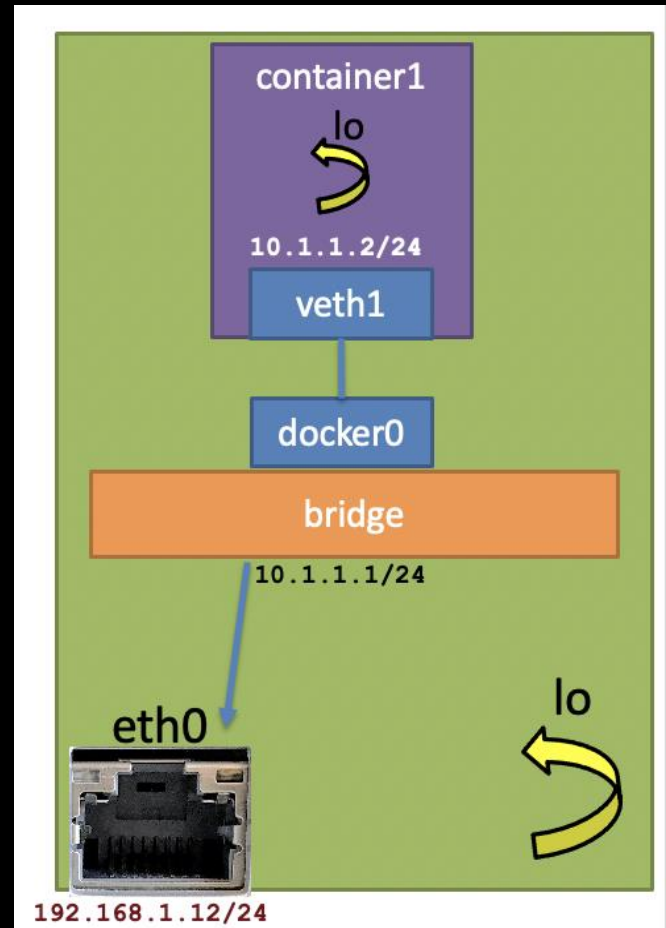
- **Container is in the host's network namespace (i.e. no new network namespace created).**
- **Processes inside container have same access to network resources as those outside.**
- **Like adding a service to the host.**



# A “bridge” Network

```
$ docker run -it --rm  
  --network=bridge ...
```

- **Default network type.**
- **Assigned a subnet IP address range.**
- **Sets up with NAT with a virtual bridge (bridge0) as a gateway.**
- **This enables a container to access the outside network**
- **Each container has a virtual ethernet i/f.**



# Docker Network

- There can be multiple bridge networks.
  - Each assigned a different IP subnet range

```
mybridge1
● diarmuidoconnor ~ ✓ docker network create mybridge1 bash at
4018c1071c50409865d00dc2e47f44698933c993bcc2a14cb06e825f4274a05b
● diarmuidoconnor ~ ✓ docker network create mybridge2
6763c4d030c210722e125396756ff10162eb16e752dd089f81312841564a1820
● diarmuidoconnor ~ ✓ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
60964a6b734c    bridge    bridge       local
8537f98578cd    host      host         local
4018c1071c50    mybridge1 bridge       local
6763c4d030c2    mybridge2 bridge       local
a72c3511f663    none      null         local
● diarmuidoconnor ~ ✓ docker network remove mybridge1    in ba
mybridge1
○ diarmuidoconnor ~ ✓
```

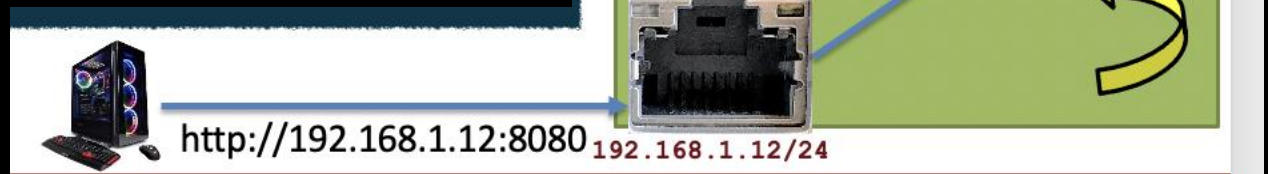
- Inspect a network:
  - \$ docker inspect network <net\_name>
  - Returns Lots of JSON-formatted information.



# Port Forwarding.

- Forwarding external socket requests to a container.
- Requested port and container port can be different.
- Ex.: Container1 is a web server listening on port 80.

Client sends request to host port 8080  
→ Host forwards requests on to container's port 80 → The request is received by a docker-proxy process → Forwarded to the container port 80.



# Docker – Building images

# Build A Docker Image.

- The normal way to create images is through a *Dockerfile* build description.

1. Create a Dockerfile, e.g.

```
FROM nginx
```

```
COPY index.html /usr/share/nginx/html/
```

2. Build the image and give it a name, e.g.

```
$ docker build -t my-nginx . (. current folder)
```

- Note:

1. Place Dockerfile in project's base folder
2. All paths are relative to the Dockerfile
3. Each command in the Dockerfile creates a new temporary container
4. Every creation step is cached, so repeated builds are fast.

# DockerFile.

- FROM
  - Sets the Base Image for the newly created image
  - e.g. FROM nginx:15:04
- COPY - copy files from the project to to the image, i.e. COPY <src>...<dest>
  - Source can contain wildcards
  - If dest does not exist it will be created
  - Example:  
COPY service.config /etc/service/  
COPY service.config /etc/service/myconfig.cfg  
COPY \*.config /etc/service/  
COPY cfg/ /etc/service/

# DockerFile.

- CMD - specifies the default start-up command to execute at container runtime.
  - Form: CMD ["executable","param1","param2"]
  - Example: CMD ["nginx", "-g", "daemon off;"]
  - If supplied, the docker run arguments overwrite those of the CMD.  
docker run image executable params....
- RUN <command> - execute command(s) inside the container during build the process.
  - It is common to tie related commands together into one RUN command, using shell magic.  
RUN apt-get update && \  
apt-get install -y ca-certificates && \  
rm -rf /var/lib/apt/lists/\*

# DockerFile.

- ENV - sets environment variables which are present during container build and remain existent in the image. Form:

ENV <key> <value>

ENV <key>=<value> ...

- They can be overwritten at container runtime with the -e option:

\$ docker run -e key=value my\_image

e.g. \$ docker run \

-e message='The answer is' -e answer=42 \

ubuntu \

bash -c 'echo \$message \$answer'

The answer is 42

# DockerFile.

- ADD - can do the same as COPY with the following additions:
  1. If src is an URL, the file is downloaded, e.g.  
ADD <https://download.elasticsearch.org/elasticsearch/elasticsearch-1.4.4.tar.gz> /es/  
RUN cd /es && tar xvfz elasticsearch-1.4.4.tar.gz
  2. If src is a local tar archive, it will be extracted to destination, e.g.  
ADD configs.tar.gz /etc/service/
- WORKDIR path - set path as default directory inside the container, e.g.  
WORKDIR /usr/app  
COPY src . # Copy src folder to /usr/app/src

# Designing good tests

- **Pre-requisite: A clear specification of the *required* behaviour (functionality) of the target.**
  - We test the specification (What), not the implementation (How).
- **Write tests in terms of: the expectations of the target's behavior for a given circumstance and specific inputs.**
  - e.g. I expect to get result X when the input is Y and the database has data Z.
- **Test Target:**
  - Unit testing– Function / Class.
  - System/Acceptance testing – End-to-end application.