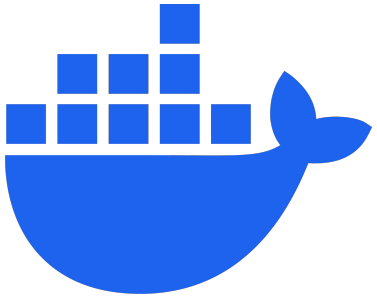
Deep dive into containers (Docker, K8s)

CS695 – Topics in Virtualization and Cloud Computing

Debojeet Das



Assignment 3's Conductor

Conductor container management tool had the following features:

- Ability to create only debian container images.
- Running simple containers without any cgroup capabilities.
- Allows only basic network functionalities.

What if you want to build and containerize your own applications which requires custom libraries or more functionalities?

[Image Credit - https://dev.to/ben/meme-monday-1o5g#comment-22ekf]

Docker Terminologies

- Dockerfile: (Like source code) List of instructions to build an image
- Docker image: (Like compiled binary)
- Docker container: (Like running process) Runtime instance of an image
- Docker registry: (Like GitHub) Repository or store of images
- Docker engine: The docker daemon process running on the host which manages images and containers

```
$ docker info
```

Docker is a server-client application. The docker engine (server) implements the container management and exposes HTTP API for communication which is used by docker CLI (client).

Docker Terminologies

```
Client: Docker Engine - Community
 Version:    26.0.0
 Context:    default
 Debug Mode: false
 Plugins:
  buildx: Docker Buildx (Docker Inc.)
    Version:  v0.13.1
    Path:     /usr/libexec/docker/cli-plugins/docker-buildx
  compose: Docker Compose (Docker Inc.)
    Version:  v2.25.0
    Path:     /usr/libexec/docker/cli-plugins/docker-compose
```

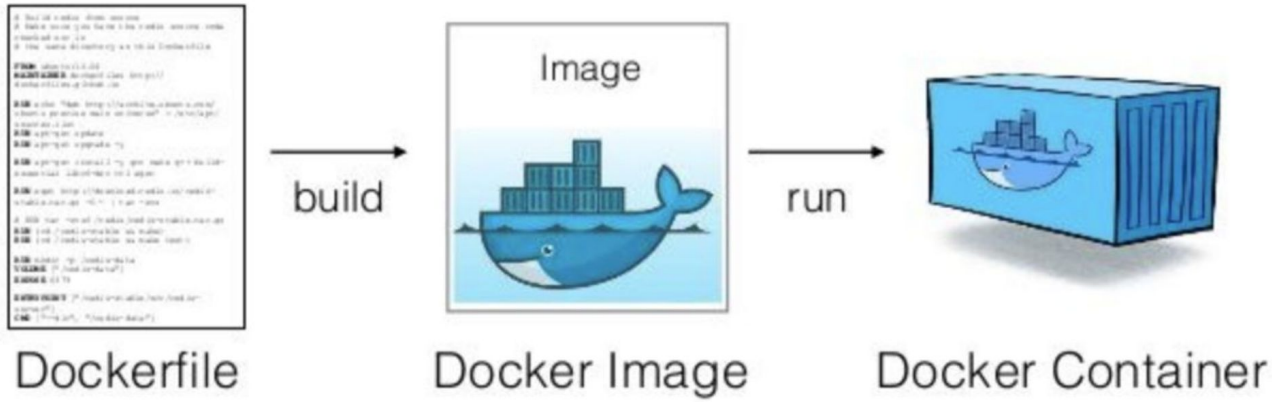
Client Details

```
Server:
 Containers: 2
  Running: 1
  Paused: 0
  Stopped: 1
 Images: 13
 Server Version: 26.0.0
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Using metacopy: false
  Native Overlay Diff: true
  userxattr: false
 Logging Driver: json-file
 Cgroup Driver: systemd
 Cgroup Version: 2
```

Server Details

Cgroup version 2 is being used here

Docker Images



Container images can either be built locally or “pulled” from a registry (which was built by someone).

Let’s try to run a container by pulling a docker image first.

We will use a docker image based on Alpine Linux with a complete package index and only 5 MB in size!

```
$ docker pull alpine:3.18
```

```
$ docker image inspect alpine:3.18
```

[image taken from ACM India Winter School on "Full-stack Networking (FSN)"]

[image registry - https://hub.docker.com/]

Docker containers and its hidden details

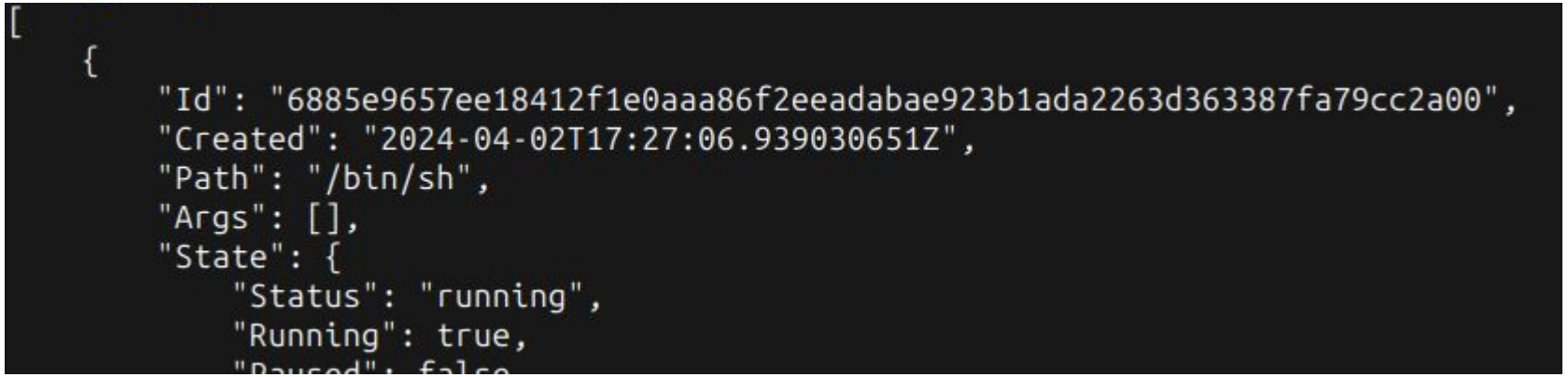
Let’s run a container and understand its internals!

```
$ docker run -it [--name <container-name>] alpine:3.18
```

```
$ docker ps
```

```
CONTAINER ID  IMAGE          COMMAND       CREATED             STATUS         PORTS     NAMES
6885e9657ee1  alpine:3.18    "/bin/sh"     23 minutes ago      Up 23 minutes            test
```

```
$ docker inspect <CONTAINER ID>or<NAME>
```



Docker containers and its hidden details

Interesting details

```
$ docker inspect <CONTAINER ID>or<NAME>
```

```
"ResolvConfPath": "/var/lib/docker/containers/6885e9657ee18412f1e0aaa86f2eeadabae923b1ada2263d363387fa79cc2a00/resolv.conf",
"HostnamePath": "/var/lib/docker/containers/6885e9657ee18412f1e0aaa86f2eeadabae923b1ada2263d363387fa79cc2a00/hostname",
"HostsPath": "/var/lib/docker/containers/6885e9657ee18412f1e0aaa86f2eeadabae923b1ada2263d363387fa79cc2a00/hosts",
```

System configuration files

```
"NetworkSettings": {
    "Bridge": "",
    "SandboxID": "99e89b1463600362f0d686af8a4984f4c4c5c0194d8d35c78e4bbee8a7a6fa09",
    "SandboxKey": "/var/run/docker/netns/99e89b146360",
```

Network settings

network namespace inode
(can be linked to /var/run/netns for netns usage)

Docker containers and its hidden details

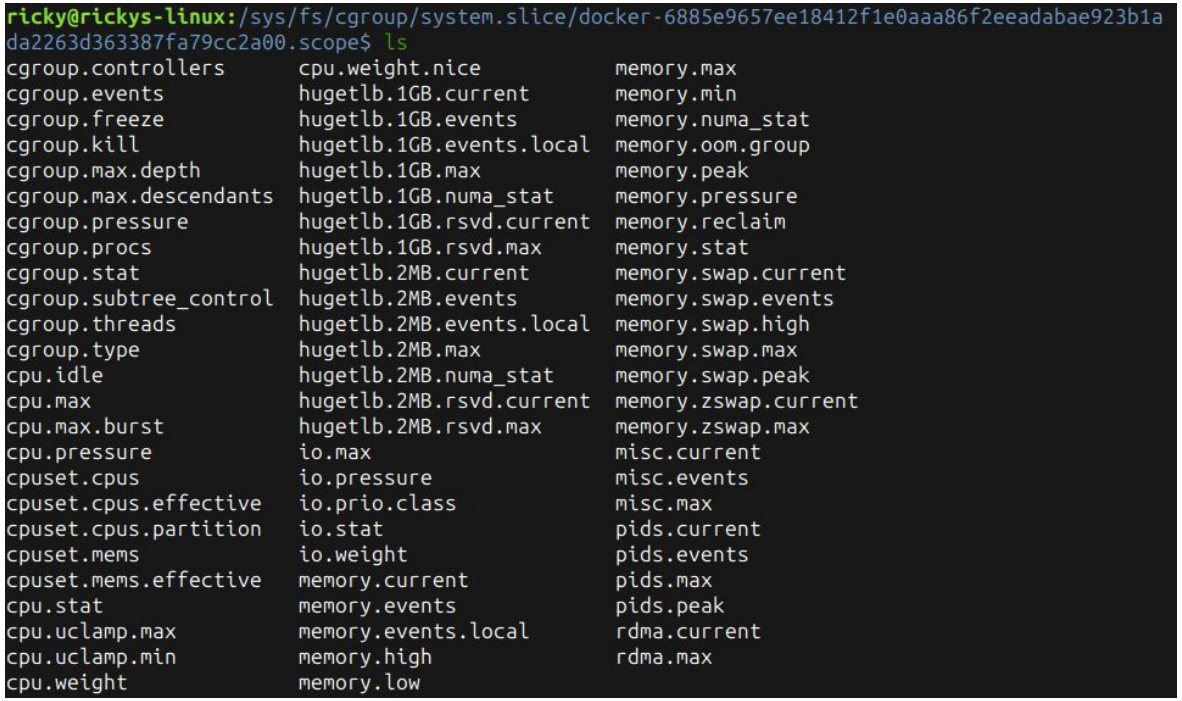
cgroup

```
$ cd /sys/fs/cgroup/cpu/docker/<container-id>
```

For cgroup v1

```
$ cd /sys/fs/cgroup/system.slice/docker-<container-id>.scope
```

For cgroup v2



Conductor to Docker - Commands

Conductor	Docker
`conductor.sh build <image-name>`	`docker build -t <image-name> <dockerfile>`
`conductor.sh images`	`docker images`
`conductor.sh rmi <image-name>`	`docker rmi <image-name>`
`./conductor.sh run <image-name> <container-name>` `./conductor.sh addnetwork <container-name> -i`	`docker run -it --name <container-name> <image-name>`
`./conductor.sh ps`	`docker ps`
`./conductor.sh stop <container-name>`	`docker stop <container-name>` `docker rm <container-name>`
`./conductor.sh exec <container-name> -- <command>`	`docker exec -it <container-name> <command>`
`./condunctor.sh run <image-name> <container-name>` `./conductor.sh addnetwork <container-name> -e 8080-80 -i`	`docker run -it --name <container-name> -p 8080:80 <image-name>`

Major difference between Assignment 3's Conductor and Docker

Conductor is a bash script based tool whereas docker is a server-client application. The docker engine (server) implements the container management and exposes HTTP API for communication which is used by docker CLI (client).

e.g. e.g. `docker ps` is `GET /containers/json`

Docker containers and its hidden details

Let's kill the container

```
$ docker rm <name>
$ docker stop <name>
$ docker inspect <name>
$ docker rm <name>
$ docker inspect <name>
```

```
Error response from daemon: You cannot
remove a running container e036efae...
Stop the container before attempting
removal or force remove
```

```
"State": {
    "Status": "exited",
    "Running": false,
    "Paused": false,
    "Restarting": false,
}
```

```
[]
Error: No such object: <name>
```

Docker containers and its hidden details

We saw the container in running status and exited status. What is this status?
Container is an instance of an image with a process running. Status is the state of that process.

1. Created - Not started, no CPU or memory is used.
   a. Using `docker create`
2. Running - Process is running
3. Exited - Process terminates, no CPU or memory used
   a. Naturally - ML training job
   b. Manually - `docker stop`
   c. Error - code panic
4. Restarting `docker run --restart=always centos:7 sleep 5`
   a. By default if command finishes, container exits. But, if restart policy is always, container restarts
5. Paused - Process is suspended, CPU is released, memory is consumed
   a. `docker pause <name>`
   b. `docker unpause <name>` - resumes the container from where it stopped
6. Removing - In the process of being removed
   a. `docker rm`

DIY:
Try running docker stats command in each of these status to examine the CPU and memory usage

[image taken from ACM India Winter School on "Full-stack Networking (FSN)"]

Docker Build

Build is a key part of container software development life cycle allowing us to package and bundle our code and ship it anywhere

Pulling vs building an image

When to pull?

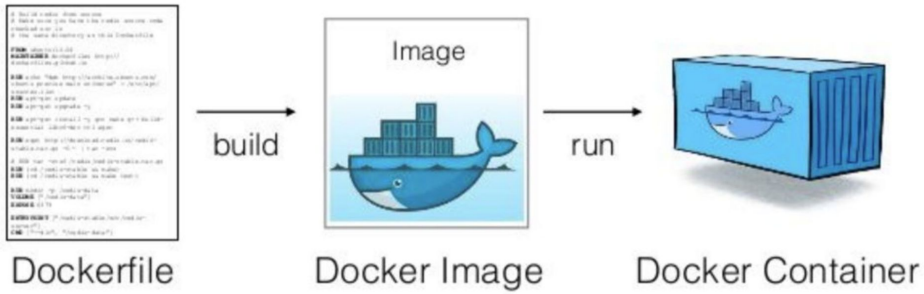
- When using someone's created image:
  - If you want to play with python, get a python image
- To access your own created image
  - Create image, push it to a registry and the pull it from elsewhere

When to build?

- To create an environment/recipe for sharing/running deterministically
- For creating any application for running on the cloud

Docker Build

Build is a key part of container software development life cycle allowing us to package and bundle our code and ship it anywhere




```
FROM memcached:1.6
RUN apt update && apt install -y stress-ng
```

Networking experiments

```
FROM ubuntu:jammy
# Install golang
RUN cd downloads \
    && wget https://go.dev/dl/go1.19.1.linux-amd64.tar.gz \
    && tar -C /usr/local -xvf go1.19.1.linux-amd64.tar.gz
ENV PATH $PATH:/usr/local/go/bin
```

k8s network simulator

Docker Build

Basic build commands

Command	Description
FROM *image* | scratch	Use a pre-existing docker image as a base image for the build
COPY *path dst*	Add files to the image. Copy from *path* in host into container at *dst*
RUN *args…*	Run arbitrary commands inside the container
WORKDIR *path*	Set the default working directory
ENV *name value*	Set an environment variable
ENTRYPOINT/CMD [“executable”, “param1”, “param2”]	Set the command to execute (when the container starts)

Task - Create Docker image

Goal: Create a “Hello World” application for container using Flask running on ubuntu.

1. Create `hello.py` with the following lines:
2. To run this in baremetal you will need to install python3 and flask and then run the application.

Similarly the Dockerfile should create a container image, which has all the dependencies installed and that automatically starts the application.

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"
```

Task - Create Docker image

```
FROM ubuntu:22.04

# install app dependencies


# copy the flask app


# final configuration and running the application
```

Task - Create Docker image

```
FROM ubuntu:22.04

# install app dependencies
RUN apt-get update && apt-get install -y python3 python3-pip
RUN pip install flask==3.0.*

# copy the flask app
COPY hello.py /

# final configuration and running the application
ENV FLASK_APP=hello
CMD ["flask", "run", "--host", "0.0.0.0", "--port", "8000"]
```

Task - Create Docker image

```
FROM ubuntu:22.04

# install app dependencies
RUN apt-get update && apt-get install -y python3 python3-pip
RUN pip install flask==3.0.*

# copy the flask app
COPY hello.py /

# final configuration and running the application with exposed port
ENV FLASK_APP=hello
EXPOSE 8000
CMD ["flask", "run", "--host", "0.0.0.0", "--port", "8000"]
```

Let’s build and run your container

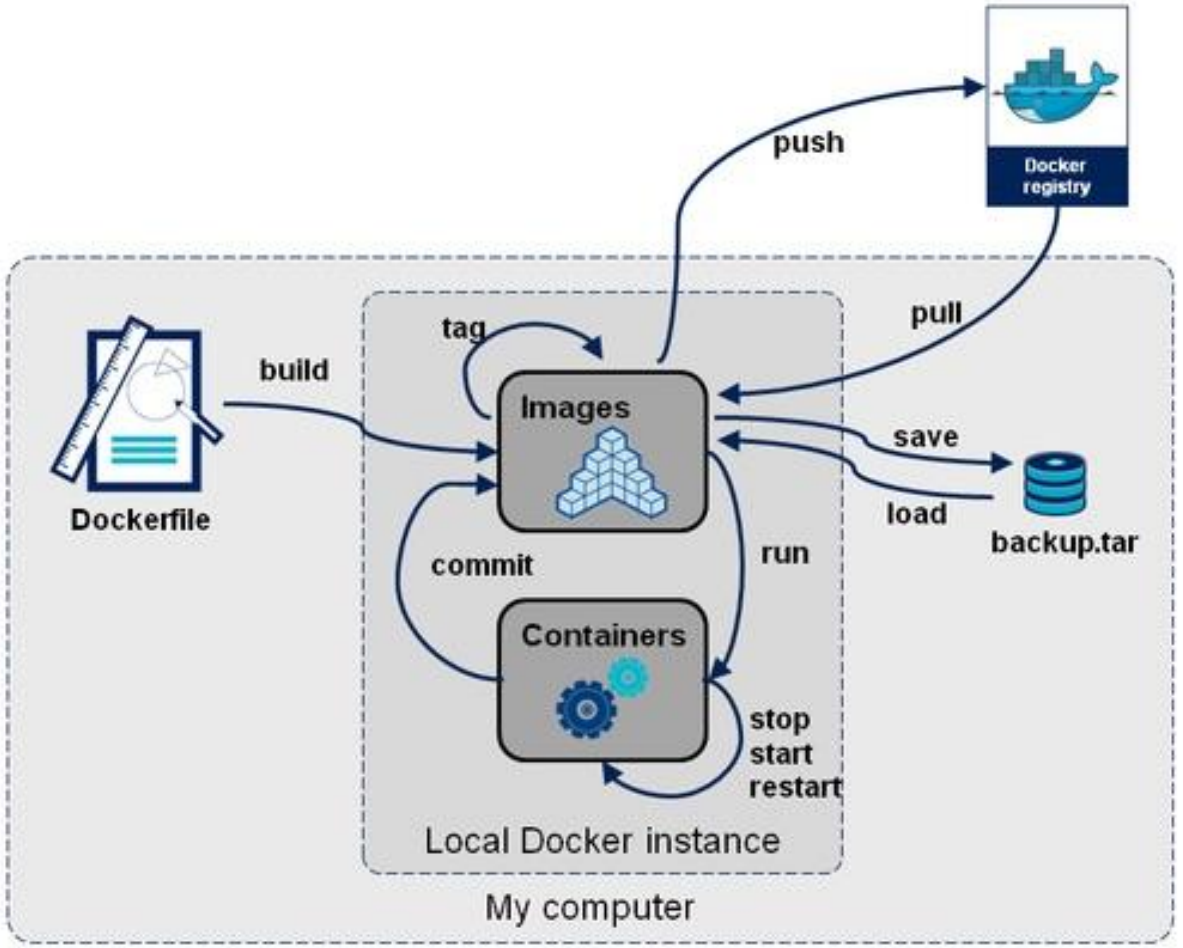
1. Build the docker image
   $ docker build -t test:latest .
2. See if the image is present and run it.
   $ docker images
   $ docker run -p 127.0.0.1:8000:8000 test:latest

Go to terminal and do curl to 127.0.0.1:8000 to see the application in action.

If you have docker hub account you can push the image just like git.

1. Docker login to registry
   $ docker login --username username
2. Rename/Tag your image
   $ docker tag my-image username/my-repo
3. Push the image
   $ docker push username/my-repo

The whole story



[image credit - https://blog.octo.com/docker-registry-first-steps]

Docker Compose

Docker Compose is a tool for defining and running multi-container applications. (Just like task 4)

You specify multiple docker containers and it brings them all up.

It sets up a single network for your entire application, all containers join them and can reach each other on this network.

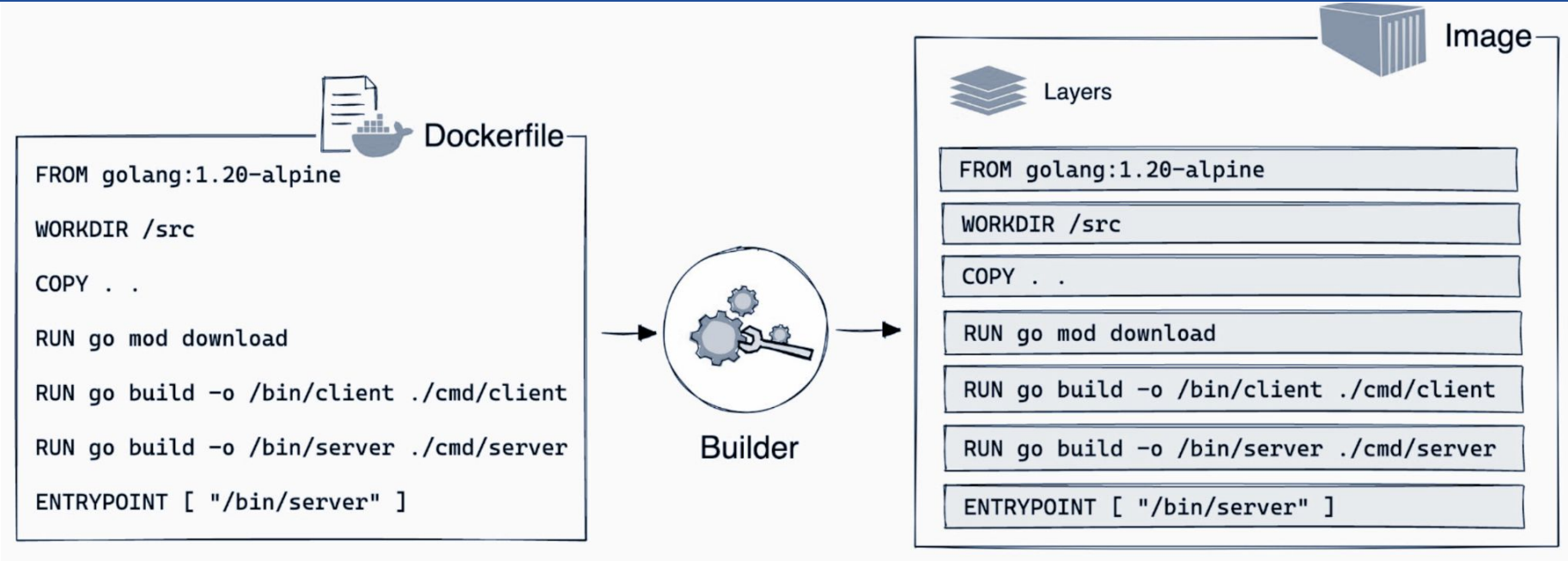
Checkout a simple example - https://docs.docker.com/compose/gettingstarted/

```
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

```
$docker compose up
```

(to setup container deployments specified in the docker compose yaml file)

Docker Internals - Layers



Docker image is built as a series of layers, each layer represents a line in the Dockerfile.

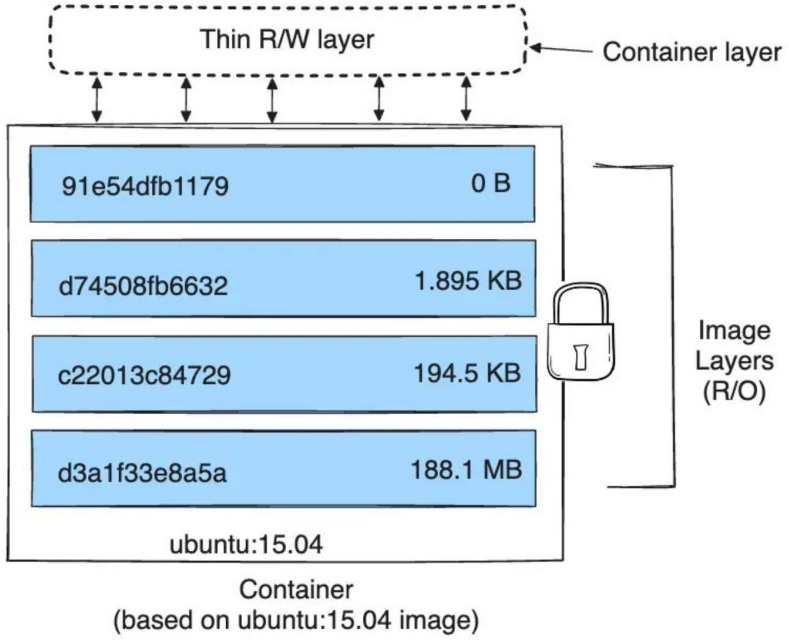
- Every command that modifies the filesystem is a new layer.
- The layer only captures the diff from the previous layer.
- Layers are shared across different images.

[image taken from ACM India Winter School on "Full-stack Networking (FSN)"]

Docker Internals - Layers

Docker image is built as a series of layers, each layer represents a line in the Dockerfile.

- When we run a container, a new writable layer (called container layer) is created. Other layers are read-only.
- The difference between a container and an image is in this writable layer
- When a container is deleted, the container layer is also deleted
- Multiple containers can share the same base image and have their own state in the container layer



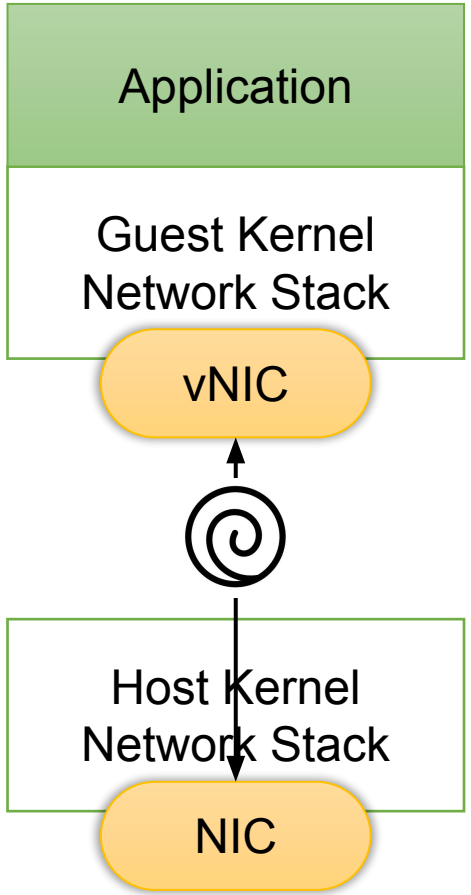
[image taken from ACM India Winter School on "Full-stack Networking (FSN)"]

Network virtualization

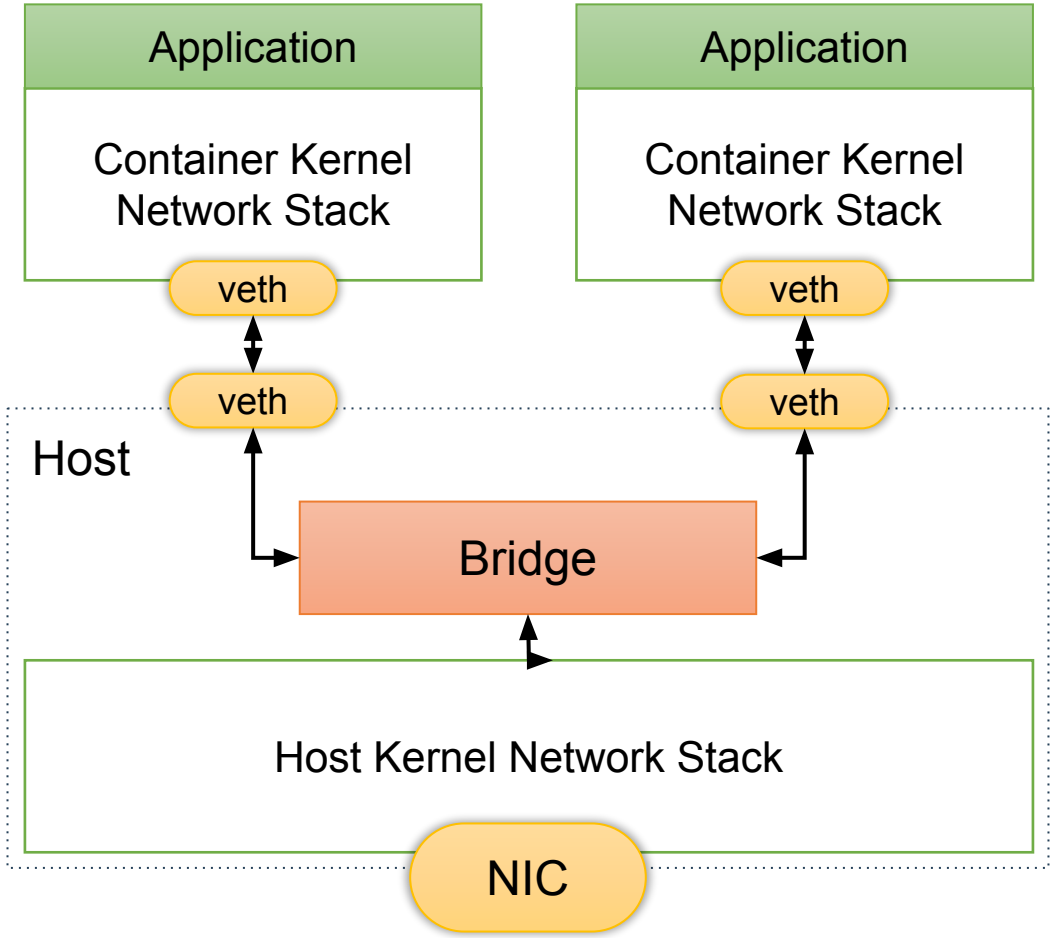
Network virtualization techniques is required to connect different VMs / Containers with each other as well as other hosts.

Types of Communication possible:

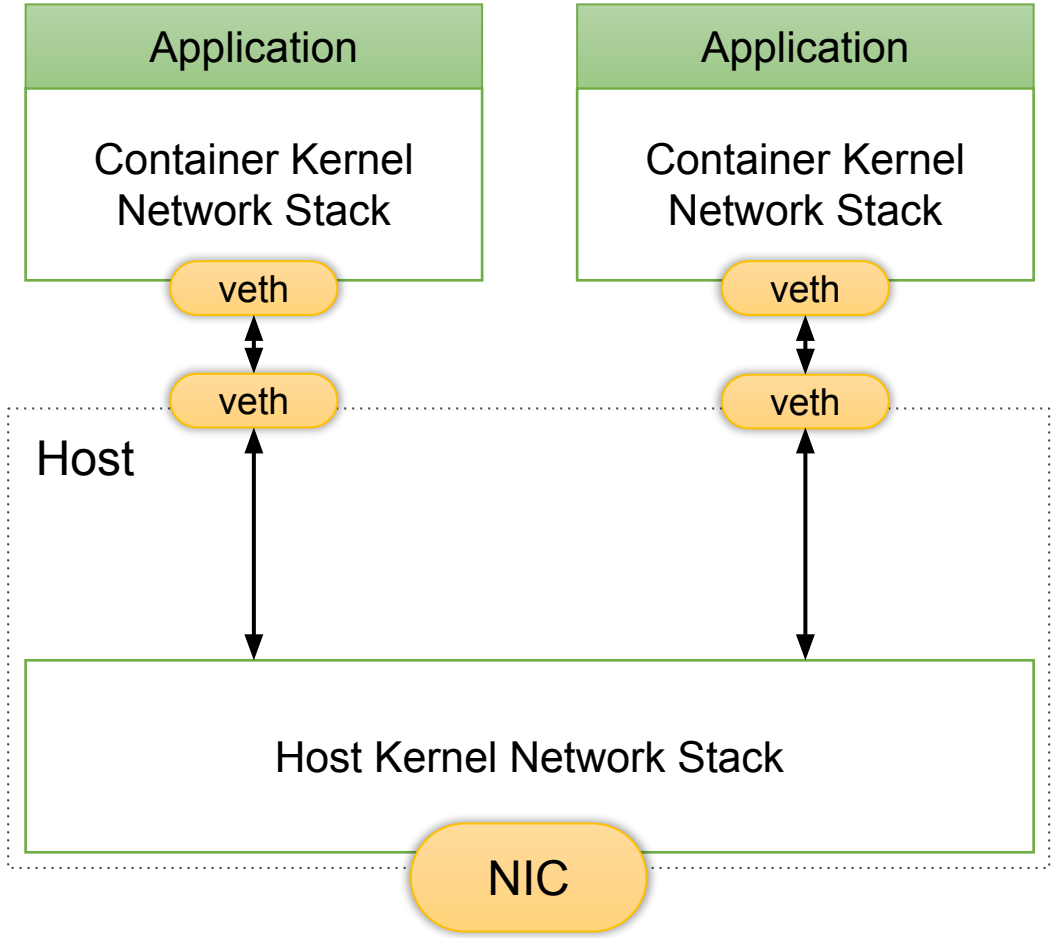
- Intra-Host Communication
- Inter-Host Communication



Intra-Host Communication

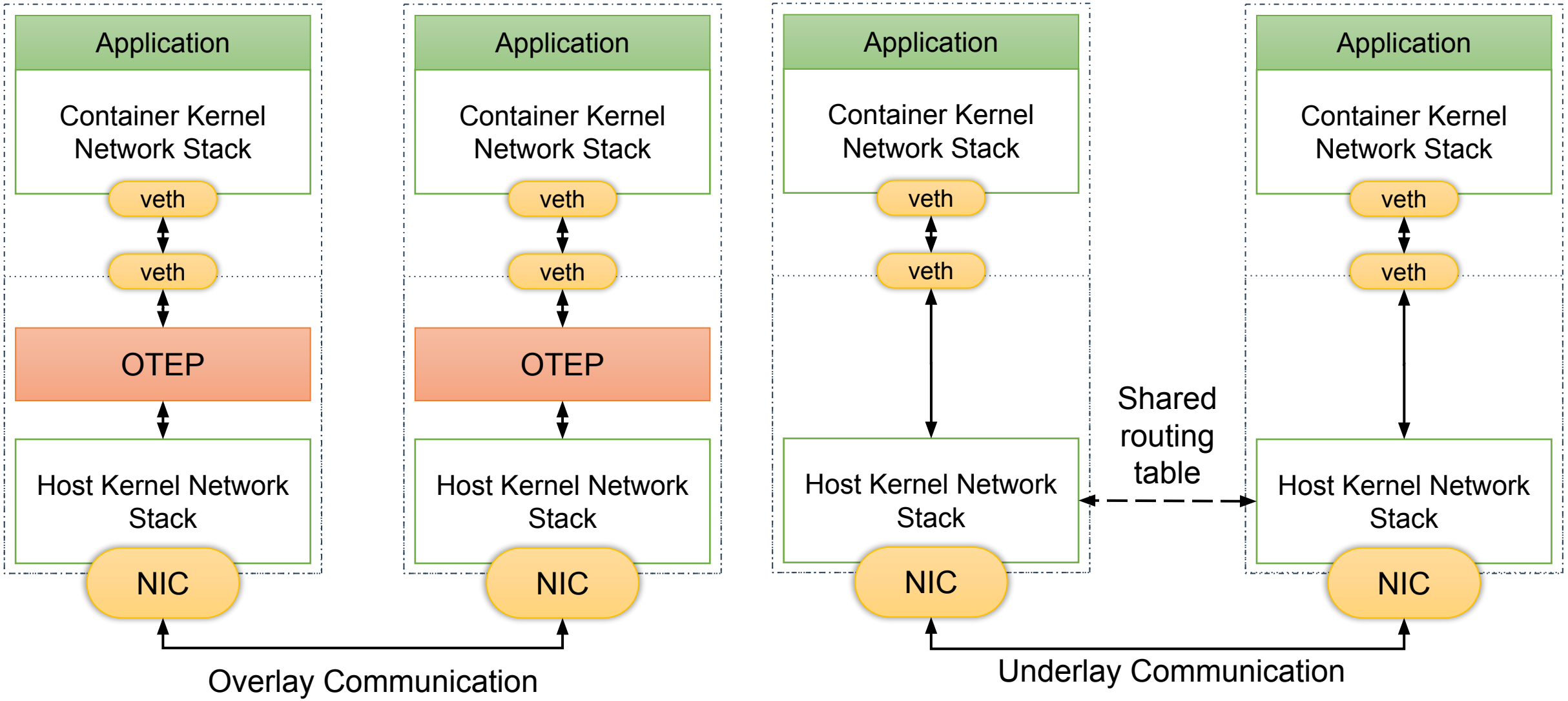


Layer 2 Forwarding Method

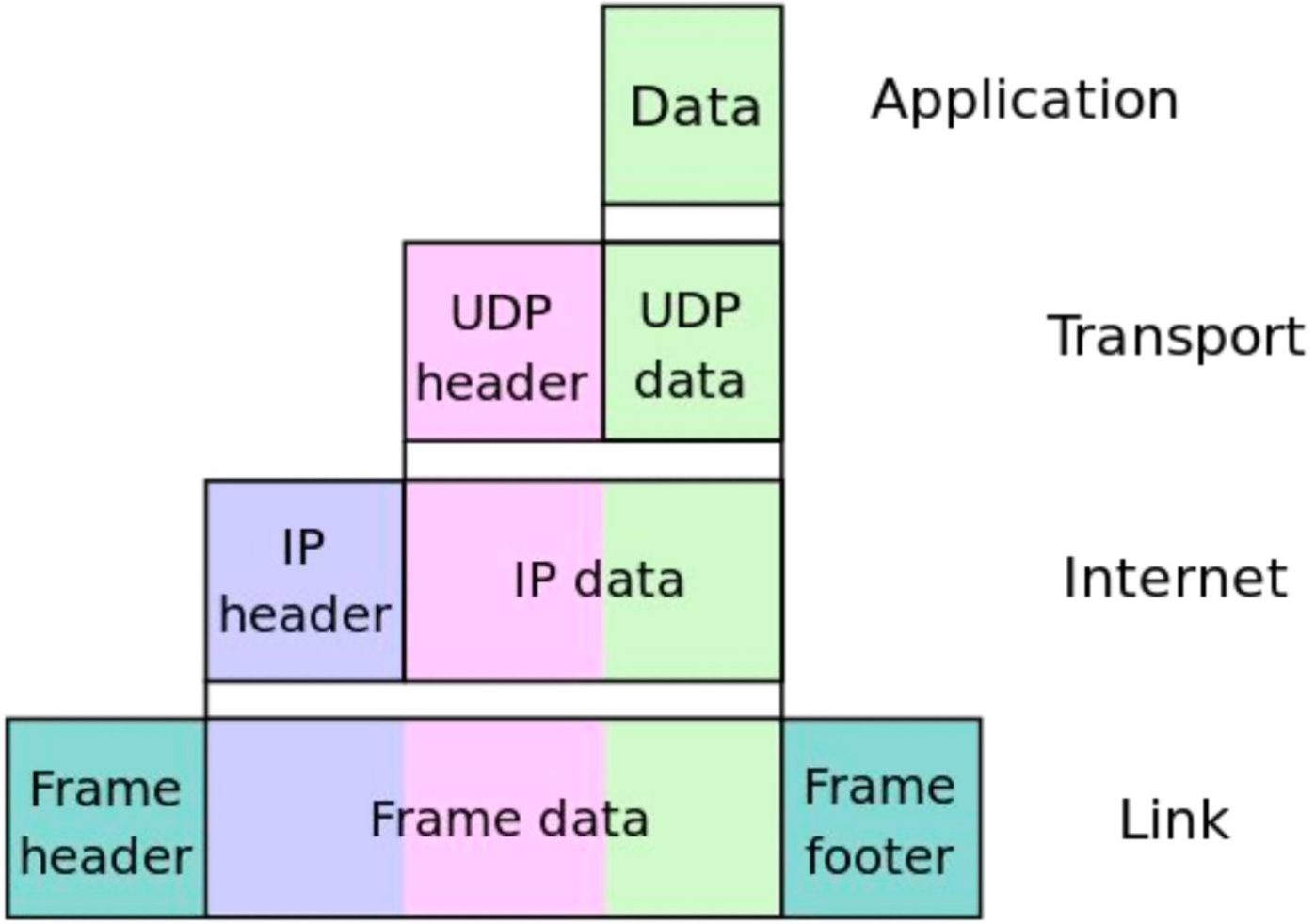


Layer 3 Routing Method

Inter-Host Communication



Overlay communication - Tunneling



[image taken from ACM India Winter School on "Full-stack Networking (FSN)"]

Overlay communication - Tunneling

[image taken from ACM India Winter School on "Full-stack Networking (FSN)"]

Docker Internals - Networking

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

1. **Bridge:** The default network driver. If you don't specify a driver, this is the type of network you are creating.
2. **Host:** Remove network isolation between the container and the Docker host, and use the host's networking directly.
3. **Overlay:** Overlay networks connect multiple Docker daemons together and enable Swarm services and containers to communicate across nodes. This strategy removes the need to do OS-level routing.
4. **IPvlan:** IPvlan networks give users total control over both IPv4 and IPv6 addressing. The VLAN driver builds on top of that in giving operators complete control of layer 2 VLAN tagging and even IPvlan L3 routing for users interested in underlay network integration.
5. **Macvlan:** Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses.
6. **None:** Completely isolate a container from the host and other containers. none is not available for Swarm services. See None network driver.

How does docker work in Windows and Mac OS?

[image credit: https://twitter.com/sidpalas/status/1638558418830233600]