Docker & Containerization

Jutlandia - Thomas & Rasmus Nov. 2025

Motivation

The problems that containers are trying to solve

“It works on my machine…”

Common problems in software development:

- Different development environments
- Dependency conflicts
- Isolation
- Inconsistent deployment environments
- Difficult to reproduce bugs
- Complex setup procedures

Containers solve these problems by packaging applications in containers

Agenda

- Motivation
  - Learning objectives
  - What is a container
- Installation
- The basics
- What Docker actually does
- Docker images
- Docker volumes
- Docker compose
- Docker network
- Advanced & extra

Learning Objectives

After this session, you will be able to:

- Understand the basics of containerization
- Use basic Docker commands to manage containers
- Understand the Linux primitives behind containers
- Pull, run, and interact with containerized applications
- Manage container lifecycles effectively
- Learn to
- Appreciate what Docker abstracts away from you

What is a Container?

A container is a lightweight, standalone, executable package that includes everything needed to run a piece of software

Includes:

- Code
- Runtime
- System tools
- System libraries

Key principle: Isolation without the overhead of a full virtual machine

Why use containers?

So with the context of what a container is this is the extended motivation:

- **Consistency:** Same environment from dev to production
- **Isolation:** Applications don’t interfere with each other
- **Portability:** Run anywhere that supports containers
- **Efficiency:** Lightweight and fast to start
- **Scalability:** Easy to replicate and orchestrate

How to install Docker

Windows

Docker Desktop:

- Download Docker Desktop
- Run the installer
- During installation, ensure WSL 2 is selected as the backend
- Restart your computer when prompted
- Launch Docker Desktop

Prerequisites for Windows:

- WSL 2 enabled (Docker Desktop will help you set this up)
- Virtualization enabled in BIOS

Mac

Docker Desktop (recommended):

- Download Docker Desktop
- Open the .dmg file and drag Docker to Applications
- Launch Docker from Applications
- Follow the setup wizard

Maybe able to use Homebrew

Suggested to look into **podman** as this usually works better on mac

Linux

Use your package mamanger to install Docker engine.

See more here: https://docs.docker.com/engine/install/ubuntu/

(Search install docker linux)

You are already on linux you got this

Docker Basics

Basic Docker Commands

1. `docker pull <image_name>`: Downloads an image from a registry (e.g., Docker Hub).
2. `docker run <image_name>`: Runs a command in a new container.
3. `docker ps`: Lists running containers.
4. `docker ps -a`: Lists all containers (running and stopped).
5. `docker exec -it <container_id/name> <command>`: Executes a command inside a running container.
6. `docker stop <container_id/name>`: Stops a running container.
7. `docker start <container_id/name>`: Starts a stopped container.
8. `docker rm <container_id/name>`: Removes a container.
9. `docker rmi <image_id/name>`: Removes an image.
10. `docker run -it kalilinux/kali-rolling bash`

Docker Basics: Pull

```
docker pull <image-name>:<tag>
```

Examples:

```
docker pull ubuntu:22.04
docker pull hello-world:latest
docker pull python:3.11-slim
```

- Downloads container images from registry - default Docker Hub
- Images are stored locally in Docker's image cache. See local images using: `docker images`
- Tags specify versions (`:latest` is default if omitted)

Docker Basics: Run

```
docker run [OPTIONS] IMAGE [COMMAND]
```

Example:

```
docker run hello-world
```

- Docker pulls (if needed) and run the specified image
- Without command the image will run predefined entry
- Command to run custom command in new container based on image
- Add options to specify how to run iamge

Docker Basics: Run with options

Common options:

```
-d                      # Run in detached mode (background)
-it                     # Interactive with terminal
-p 8080:80              # Port mapping (host:container)
--name mycontainer      # Give container a name
--rm                    # Remove container when it stops
-e VAR=value            # Environment variable
```

Example:

```
docker run -d -p 8080:80 --name webserver nginx
```

Docker Basics: ps

```
docker ps      # Show running containers
docker ps -a   # Show all containers (including stopped)
```

Output shows:

- Container ID
- Image used
- Command running
- Creation time
- Status
- Ports

- Container name

Docker Basics: Exec

```
docker exec [OPTIONS] CONTAINER COMMAND
```

Most common usage - interactive shell:

```
docker exec -it mycontainer bash
docker exec -it mycontainer sh # if no bash
```

Other examples:

```
docker exec mycontainer ls /app
docker exec mycontainer cat /var/log/app.log
```

Note: Container must be running to use exec

Container Lifecycle Management

Control state of container:

```
docker start <container>    # Start a stopped container
docker stop <container>     # Gracefully stop (SIGTERM)
docker restart <container>  # Stop and start
docker kill <container>     # Force stop (SIGKILL)
docker pause <container>    # Pause all processes
docker unpause <container>  # Resume paused container
```

Container Lifecycle Management

Cleanup:

```
docker rm <container>          # Remove stopped container
docker rm -f <container>       # Force remove (stops first)
docker container prune         # Remove stopped containers
docker rmi <image_id/name>     # Delete image for container
```

Other Useful Lifecycle Commands

Information about containers:

```
docker logs <container>          # View logs
docker logs -f <container>       # Follow logs (tail)
docker inspect <container>       # Detailed info (JSON)
docker stats                     # Live resource usage
docker top <container>           # Running processes
```

Copy files:

```
docker cp file.txt container:/path
docker cp container:/path/file.txt ./
```

Docker Basics DEMO

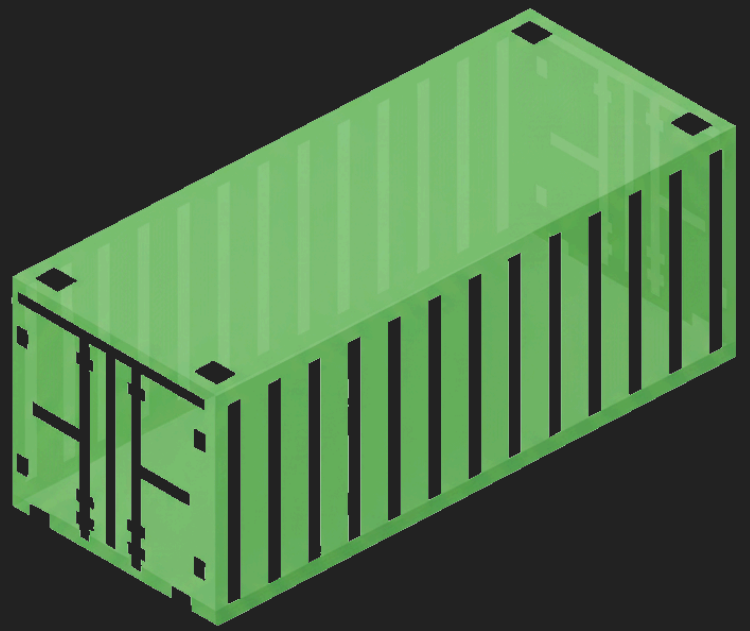
What Docker Actually Does

The Building Blocks of Containers

Components

Docker is a **wrapper** around Linux kernel features:

- **Namespaces** - Isolation (what you can see)
- **cgroups** - Resource limits (what you can use)
- **rootfs** - Filesystem isolation (what you can access)



Linux Namespaces

Namespaces provide **isolation** - separate instances of global resources Types of namespaces:

- **PID** - Process IDs (processes can't see outside)
- **NET** - Network stack (own IP, ports, routing)
- **MNT** - Mount points (filesystem view)
- **UTS** - Hostname and domain name
- **IPC** - Inter-process communication
- **USER** - User and group IDs
- **CGROUP** - cgroup root directory

Each container runs in its own set of namespaces

Linux Namespaces: unshare command

```
$ man unshare
unshare(1)                  User Commands                  UNSHARE(1)

NAME
       unshare - run program in new namespaces

DESCRIPTION
The unshare command creates new namespaces (as specified by
the command-line options described below) and then executes
the specified program.
```

Linux Namespaces: Manual Creation Examples

Create a new namespace manually:

```
# Create new PID and mount namespace
unshare --pid --mount --fork bash
mount -t proc proc /proc
# Inside this namespace:
ps aux  # Only shows processes in this namespace

# Create UTS namespace (hostname)
unshare --uts bash
hostname container-host
hostname  # Shows new hostname
exit
hostname  # Back to original
```

Network Namespaces

```
$ man network_namespaces
network_namespaces(7)  Miscellaneous Information Manual  network_namespaces(7)
NAME
       network_namespaces - overview of Linux network namespaces
DESCRIPTION
       Network namespaces provide isolation of the system resources associated
       with networking: network devices, IPv4 and IPv6 protocol stacks...
```

Network Namespaces

```
$ man network_namespaces
network_namespaces(7)  Miscellaneous Information Manual  network_namespaces(7)
NAME
       network_namespaces - overview of Linux network namespaces
DESCRIPTION
       Network namespaces provide isolation of the system resources associated
       with networking: network devices, IPv4 and IPv6 protocol stacks...
```

Network namespaces (using iproute2)

```
$ ip netns add jutlandia
$ ip netns list
$ ip netns exec jutlandia ip link
$ ip netns del jutlandia
```

Linux cgroups (Control Groups)

cgroups **limit and account** for resource usage:

- CPU time
- Memory usage
- Disk I/O
- Network bandwidth
- Device access

Prevents containers from consuming all host resources

cgroups

```
$ man cgroups
cgroups(7)            Miscellaneous Information Manual            cgroups(7)

NAME
       cgroups - Linux control groups

DESCRIPTION
Linux kernel feature which allow processes to be organized into hierarchical groups
whose usage of various types of resources can then be limited and monitored.
```

cgroups: Manual Setup

```
# Create a new cgroup
sudo mkdir /sys/fs/cgroup/demo_container

# Set memory limit to 1MB
echo 1048576 | sudo tee /sys/fs/cgroup/demo_container/memory.max

# Add current shell to this cgroup
echo $$ | sudo tee /sys/fs/cgroup/demo_container/cgroup.procs

# Now try allocating memory beyond the limit
stress-ng --vm 1 --vm-bytes 150M --vm-keep --timeout 10s

# Remove yourself from the cgroup first
echo $$ | sudo tee /sys/fs/cgroup/cgroup.procs

# Then remove the cgroup
sudo rmdir /sys/fs/cgroup/demo_container

#libcgroup version does the same as above
$ cgcreate -g cpu,memory:/mygroup
```

Root Filesystem (rootfs)

Each container has its own **root filesystem**:

- Looks like a complete Linux filesystem
- Contains `/bin`, `/etc`, `/usr`, etc.
- Isolated from host filesystem
- Created from container image layers

The container process sees this as `/` (root) On the host, it's just a directory somewhere

Rootfs

```
$ man chroot
CHROOT(1)                        User Commands                        CHROOT(1)
NAME
       chroot - run command or interactive shell with special root directory
SYNOPSIS
       chroot [OPTION] NEWROOT [COMMAND [ARG]...]
       chroot OPTION
```

Creating a Simple rootfs

```
# Create a minimal rootfs
mkdir -p myroot/{bin,lib,lib64,proc}
# Copy bash and its dependencies
cp /bin/bash myroot/bin/
ldd /bin/bash  # Find required libraries
# Copy those libraries to myroot/lib/
# Use chroot to change root
sudo chroot myroot /bin/bash
# Now your root is myroot/
ls /  # Shows bin, lib, proc
# You're isolated in this filesystem!
```

This is the basis of container filesystem isolation

Recap

When you run `$docker run`:

1. **Creates namespaces** (PID, NET, MNT, UTS, IPC, USER)
2. **Sets up cgroups** with resource limits
3. **Prepares rootfs** from image layers
4. **Changes root** to container's filesystem
5. **Configures network** (virtual ethernet pair)
6. **Executes** the container's entrypoint/command

DEMO Custom script

Packaging a simple Python application

Simple Python App : main.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run(host="0.0.0.0", debug=True, port=80)
```

Simple Python App : requirements.txt

```
flask
```

Simple Python App : .Dockerignore

```
*__pycache__/*
.env
```

Simple Python App : files

```
static/
 - index.html
 - jutlandia.jpg
.Dockerignore
.env
main.py
requirements.txt
Dockerfile
```

Simple Python App : Dockerfile

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

Build it

```
docker build -t my_app .
```

Simple Python App 2-Stage : Dockerfile

```
FROM python:2.7-alpine as base
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
RUN apk add --update postgresql-dev gcc musl-dev linux-
headers

RUN pip install wheel && pip wheel . --wheel-dir=/app/
wheels
```

```
FROM python:2.7-alpine
COPY --from=base /app /app
WORKDIR /app
RUN pip install --no-index --find-links=/app/wheels -r
requirements.txt
COPY . .
CMD ["python", "app.py"]
```

Docker volumes

```
docker run -it --rm my_app bash
```

```
docker run -it --rm -v ./host/data:/app/data my_app bash
```

```
cd ./host/data ; wget https://jutlandia.club/static/favicon.ico
```

Docker Compose

Docker Compose keywords

1. image
2. ports
3. volumes
4. networks
5. environment
6. depends_on
7. restart
8. build
9. container_name
10. labels
11. healthcheck
12. devices
13. extra_hosts
14. security_opt
15. env_file
16. ulimits
17. deploy

Example 2 Compose:

```
caddy:
  image: caddy
  cap_add:
    - NET_ADMIN
  ports:
   - "80:80" # HTTP
   - "443:443" # HTTPS
   - "443:443/udp" # QUIC
  volumes:
    - ./caddy/:/etc/caddy/
  networks:
    - backend
  restart: always
```

```
app:
  image: myapp
  networks:
    - backend
  restart: always
  volumes:
   - ./data:/data
```

Docker Network

Network types:

- bridge
- host
- none
- overlay
- ipvlan
- macvlan

Bridge

```
ip a
  1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    inet 127.0.0.1/8 scope host lo

  2: enp10s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    link/ether xx:xx:xx:xx:xx:xx brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.2/24 brd 192.168.1.255 scope global dynamic

  3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether da:89:10:7d:f4:93 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
```

Network Bridges in linux

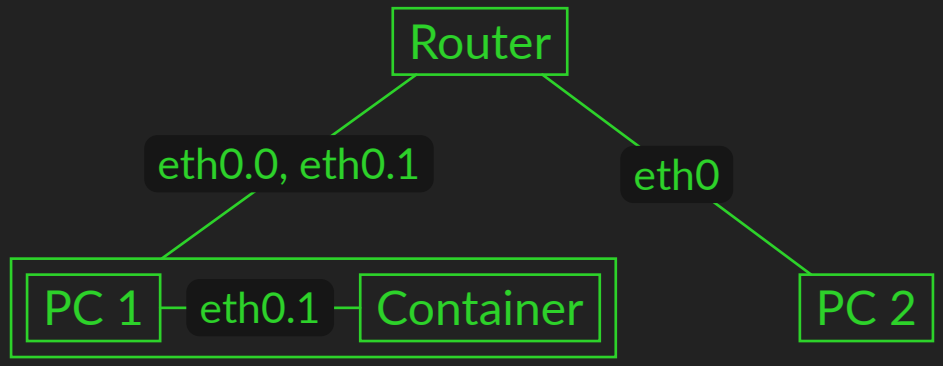
Virtual network switch

```
$ ip link add name br0 type bridge
$ ip addr set dev br0 up
$ ip link set enp0 master br0
$ ip link set enp1 master br0
```

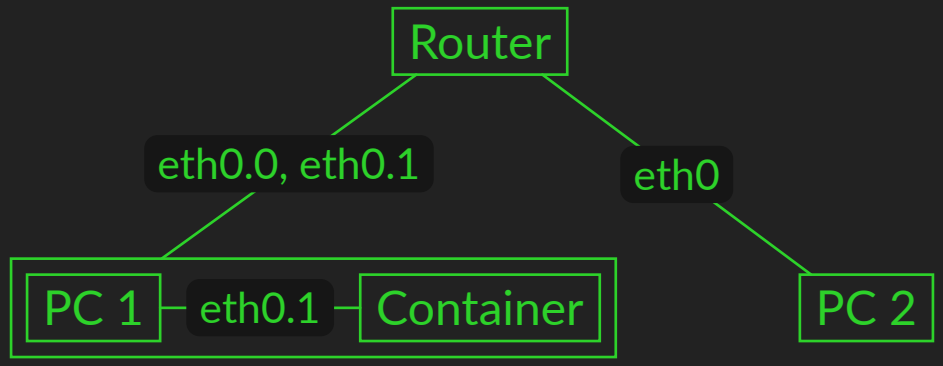
Docker bridge network

```
$ bridge
  13: xxx
  15: vethbxx@enp10s0: master docker0 state forwarding
priority 32 cost 2
  16: xxx
```

macvlan



macvlan




```
2: enp10s0.10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
  link/ether 56:ec:62:0d:26:1a brd ff:ff:ff:ff:ff:ff
  inet 192.168.1.2/24 brd 192.168.1.255 scope global dynamic
3: enp10s0.20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
  link/ether b5:f1:eb:8c:c4:44 brd ff:ff:ff:ff:ff:ff
  inet 192.168.1.3/24 brd 192.168.1.255 scope global dynamic
```

overlay

- Used for Docker Swarm
- Virtual network between hosts
- Uses VXLAN tunneling
- A containers can connect as if they were on the same network.
  - e.g. `$ exec app bash -c ping db`

bind host ip

```
services:
  app:
    image: my_app
    ports:
      - "100.101.125.25:8080:80"
```

- Allows binding to lan/wan/vpn only

ports and expose

```
services:
  app:
    image: my_app
    ports:
      - "100.101.125.25:8080:80"
    expose:
      - "80"
```

- Ports binds ports on the host to the container
- Expose allows inter-container communication

Advanced Compose: Build

```yaml
app:
  build: "./python-app/"
  image: myapp
  container_name: my_app
  depends_on:
    - db
  networks:
    - backend
```

```yaml
db:
  image: postgres:15
  networks:
    - backend

networks:
   backend:
```

build
depends_on

Advanced Compose: Build

```yaml
app:
  build: "./python-app/"
  image: myapp
  container_name: my_app
  depends_on:
    - db
  networks:
    - backend
```

```yaml
db:
  image: postgres:15
  networks:
    - backend

networks:
   backend:
```

build
depends_on

```
$ docker compose up -d --build app
```

Advanced Compose: Healthcheck

```
app:
  image: myapp
  depends_on:
    db:
      condition: service_healthy
      restart: true
  networks:
    - backend
  healthcheck:
    test: ["CMD", "curl", "-f", "http://
localhost"]
    interval: 30s
    timeout: 10s
    retries: 3
    start_period: 40s
```

```
db:
  image: postgres:15
  networks:
    - backend

networks:
   backend:
```

Advanced Compose: Healthcheck

New keywords:

```
healthcheck
depends_on (health)
```

1. Arbitrary commands to check container health
2. Lifecycle management e.g. -> auto restart

Common Errors

- Apt/Apk/yum/zyp/..
- Bash/sh

Extras

Local CI (quick and dirty)

```
#!/bin/bash
docker buildx build --platform linux/amd64 -t myrepo/
myimage:latest --push .
ssh remote 'cd deployments/myapp && docker compose up -d
myapp --force-recreate --pull always --quiet-pull'
#EOF
```

compose watch [1]

- Missing the important parts
  - Testing, A/B, Green-Blue deployment, rollbacks

Compose GPU

```yaml
jupyter:
  image: intel/intel-extension-for-pytorch:2.8.10-xpu-pip-jupyter
  devices:
    - "/dev/dri:/dev/dri"
    # /card0
    # /renderD128
```

```yaml
jupyter:
  image: nvidia/cuda:12.1.1-runtime-ubuntu22.04
  deploy:
    resources:
      reservations:
        devices:
          - driver: nvidia
            count: all
            capabilities: [gpu]
```

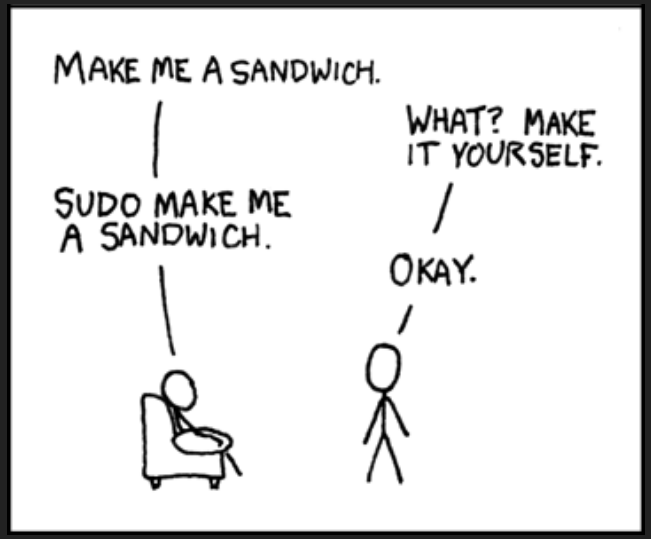
- Example 1 AMD/Intel GPU container
- Example 2 Nvidia GPU container (Nvidia container toolkit)s
- Rapidly developing (TPU, SR-IoV etc.)

Priveliged Containers

```
services:
  app:
    image: myapp:latest
    privileged: true
```

```
docker run --privileged -it
myapp:latest bash
```

```
docker run --cap-add
NET_RAW ...
```



XKCD Sandwhich [2]

Docker Model Runner

LLM side cars - DMR example [3]

```yaml
services:
  agents:
    image: example_image:latest
    models:
      qwen3:
```

```yaml
models:
  qwen3:
    model: ai/qwen3
    context_size: 8192
    runtime_flags:
      - --no-prefill-assistant
```

- Not very mature
- Clean interface
- Abstract, hard to debug, simple to deploy

Nix builds [4], [5]

```
{ pkgs ? import <nixpkgs> {} }:
pkgs.dockerTools.buildImage {
  name = "myapp";
  tag = "latest";
  contents = [
    pkgs.python39
    pkgs.numpy
    pkgs.requests
  ];
  config = {
    Cmd = [ "python3" "/app/main.py" ];
  };
}
```

One liners

```
docker buildx prune
docker image prune
runningImages=$(docker ps --format {{.Image}})

docker images --format "{{.ID}} {{.Repository}}:{{.Tag}}" |
grep -v "$runningImages"
```

Container runtimes

- Kubernetes
- ECS
- Fargate

...

Android emulation [6]

```
docker run -it \
  --device /dev/kvm \
  -v /tmp/.X11-unix:/tmp/.X11-unix \
  -e "DISPLAY=${DISPLAY:-:0.0}" \
  -p 5555:5555 \
  sickcodes/dock-droid:latest
```

mininet/containernet, routing, queuing

https://containernet.github.io

```
from mininet.net import Containernet
from mininet.node import Controller
from mininet.cli import CLI
from mininet.link import TCLink
from mininet.log import info, setLogLevel
net = Containernet(controller=Controller)
net.addController('c0')
d1 = net.addDocker('d1', ip='10.0.0.251',
dimage="ubuntu:trusty")
d2 = net.addDocker('d2', ip='10.0.0.252',
dimage="ubuntu:trusty")
s1 = net.addSwitch('s1')
s2 = net.addSwitch('s2')
net.addLink(d1, s1)
net.addLink(s1, s2, cls=TCLink,
delay='100ms', bw=1)
net.addLink(s2, d2)
net.start()
net.ping([d1, d2])
CLI(net)
net.stop()
```

```yaml
services:
  m1:
    image: satellite
    container_name: m1
    command: sleep infinity
    networks:
      - link_a
    privileged: true

  m2:
    image: satellite
    container_name: m2
    command: sleep infinity
    networks:
      - link_a
      - link_b
    privileged: true

  m3:
    image: satellite
    container_name: m3
    command: sleep infinity
    networks:
      - link_b
    privileged: true

networks:
  link_a:
  link_b:
```

```
(m1) $ ip route replace 192.168.0.3 via
172.27.0.2
(m2) $ tc qdisc add dev eth_a root netem delay
10ms rate 1mbit
(m2) $ tc qdisc add dev eth_b root netem delay
10ms rate 1mbit
(m3) $ ip route replace 192.168.0.1 via
172.27.0.2
```

container network emulation

- link_a, link_b networks, with subnets. All twice removed nodes in the network graph need routing tables (ip route add ..)
- First container must not have a tc rule on its interface for correct TCP behavior
- using tc/netem inside containers (a lot of caveats (e.g. tcp small queues))
- Can be added to all containers

Podman

- Daemonless
- Rootless
- API compatible with Docker
- Kubernetes style composition (pods vs services)

WIP: Ʉ₴฿ / ł₱ ł₦ ĐØ₵₭ɆⱤ

```
services:
  app:
    image: ubuntu:20.04
    command: ["usbip", "attach", "-r", "192.168.1.2", "--bus-id=0"]
    #devices: usb host side
    priveliged: true
```

Vagrant docker

Linux Desktop in docker (linux server images) (VNC/Wayland sockets)

Advanced:

```
docker buildx create --name crossbuilder
docker buildx use crossbuilder
docker buildx inspect --bootstrap

docker buildx build --platform linux/amd64 -t github.com/
user/repo/image_name --push
```

Bibliography

[1] Docker Inc., “Docker Compose Watch.” [Online]. Available: https://docs.docker.com/compose/how-tos/file-watch/

[2] Randall Munroe, “xkcd sandwhich.” [Online]. Available: https://xkcd.com/149/

[3] Docker Inc., “compose for agents.” [Online]. Available: https://github.com/docker/compose-for-agents/blob/main/crew-ai/compose.yaml

[4] xeiaso, “Nix is a better Docker image builder than Docker's image builder.” [Online]. Available: https://xeiaso.net/talks/2024/nix-docker-build/

[5] Mithcell Hasimoto, “Using Nix with Dockerfiles.” [Online]. Available: https://mitchellh.com/writing/nix-with-dockerfiles

[6] “DockDroid - Android in Docker.” [Online]. Available: https://github.com/sickcodes/dock-droid

[7] Docker Inc., “Docker Compose Samples overview.” [Online]. Available: https://docs.docker.com/reference/samples/