



OST

Eastern Switzerland
University of Applied Sciences

Distributed Systems (DSy)

Docker

Thomas Bocek

02.03.2025

Learning Goals

- Lecture 4 (Docker)
 - How does docker work (container implementation)?
 - Best practices
 - What is docker-compose, and how to run multiple services
 - How to use it in your challenge task

Docker Examples

- Install docker [[ubuntu](#), [Mac](#), [Windows](#)]
 - `docker run hello-world`
 - Fetches the hello world example from [docker hub](#)
 - No version provided – latest
 - [Docker Hub](#): container image repository
 - Community / official
 - [Alpine](#)
 - `docker save hello-world -o test.tar`
 - `tar xf test.tar`
 - `tar xf cdccdf50922d90e847e097347de49119be0f17c18b4a2d98da9919fa5884479d/layer.tar`
 - `./hello`
- See your installed images
 - `docker images / docker images -a`
 - `docker rmi hello-world / docker rmi fce289e99eb9`
 - `docker ps -a`
 - `docker rm 913edc5c90c4`
- GUI: e.g., [Docker Desktop](#)

Details

- **Bocker**: Docker implemented in around 100 lines of bash
 - Requirements: btrfs-progs, curl, iproute2, iptables, libcgroup-tools, util-linux, coreutils
- FS Virtualization
 - **OverlayFS**: **union filesystem**, “combines multiple different underlying mount points into one”
- Dockerfile:
 - docker build . -t test
 - docker run test
 - docker save test:latest > test.tar

Dockerfile:

```
FROM alpine
ADD hello.sh .
ENTRYPOINT ["sh", "hello.sh"]
```

hello.sh:

```
#!/bin/sh
echo "Hallo"
```

- 2+ Layers
 - Alpine, with **BusyBox**, **1MB**, libc (musl), crypto, ssl, etc.
 - hello.sh
- Add a new layer
 - If input does not change, docker layer is kept - cached

OverlayFS

- Example

- The lower directory can be read-only or could be an overlay itself
- The upper directory is normally writable
- The workdir is used to prepare files as they are switched between the layers.

```
cd /tmp
mkdir lower upper workdir overlay
```

```
sudo mount -t overlay -o \
lowerdir=/tmp/lower,\
upperdir=/tmp/upper,\
workdir=/tmp/workdir \
none /tmp/overlay
```

- Read only

- How to remove data in read-only lowerdir
 - Mark as deleted in upperdir

```
cd /tmp
mkdir lower upper workdir overlay
```

```
sudo mount -t overlay -o \
lowerdir=/tmp/lower1:/tmp/lower2 /tmp/overlay
```

```
cd /tmp
mkdir lower upper workdir overlay
```

```
sudo mount -t overlay -o \
lowerdir=/tmp/lower1:/tmp/lower2,\
upperdir=/tmp/upper,\
workdir=/tmp/workdir \
none /tmp/overlay
```

Cgroups

- **control groups**: limits, isolates, prioritization of CPU, memory, disk I/O, network

```
ls /sys/fs/cgroup
```

```
sudo apt install cgroup-tools / yay -S libcgroup-git
```

```
cgcreate -g cpu:red  
cgcreate -g cpu:blue
```

```
echo -n "20" > /sys/fs/cgroup/blue/cpu.weight  
echo -n "80" > /sys/fs/cgroup/red/cpu.weight
```

```
cgexec -g cpu:blue bash  
cgexec -g cpu:red bash
```

```
sha256sum /dev/urandom #does not work?  
taskset -c 0 sha256sum /dev/urandom
```

- Install tools
- Create two groups
 - Assign 20% of CPU and 80% of CPU
- Execute bash → test CPU
- **Resource control with docker**

```
docker run \  
--name=low_prio \  
--cpuset-cpus=0 \  
--cpu-shares=20 \  
alpine sha256sum /dev/urandom
```

```
docker run \  
--name=high_prio \  
--cpuset-cpus=0 \  
--cpu-shares=80 \  
alpine sha256sum /dev/urandom
```

Separate Networks

- Linux Network Namespaces
 - provide isolation of the system resources associated with networking [source]

```
ip netns add testnet
ip netns list
```

- Create virtual ethernet connection

```
ip link add veth0 type veth peer name veth1 netns testnet
ip link list #?
ip netns exec testnet <cmd>
```

- Configure network

```
ip addr add 10.1.1.1/24 dev veth0
ip netns exec testnet ip addr add 10.1.1.2/24 dev veth1
ip link set dev veth0 up
ip netns exec testnet ip link set dev veth1 up
```

- Run server

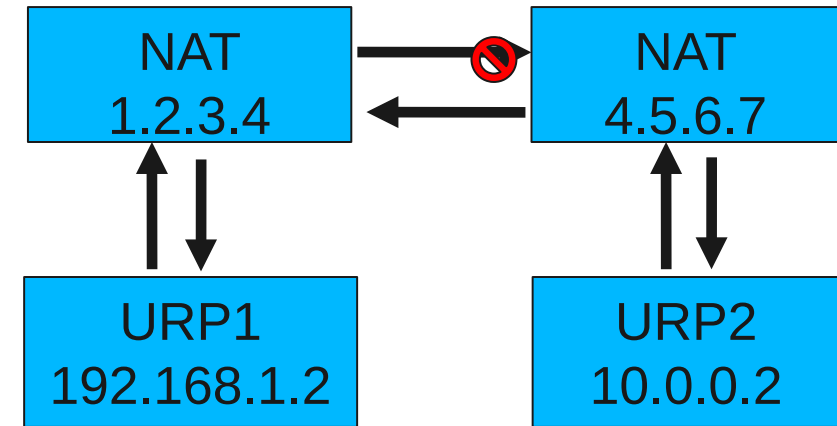
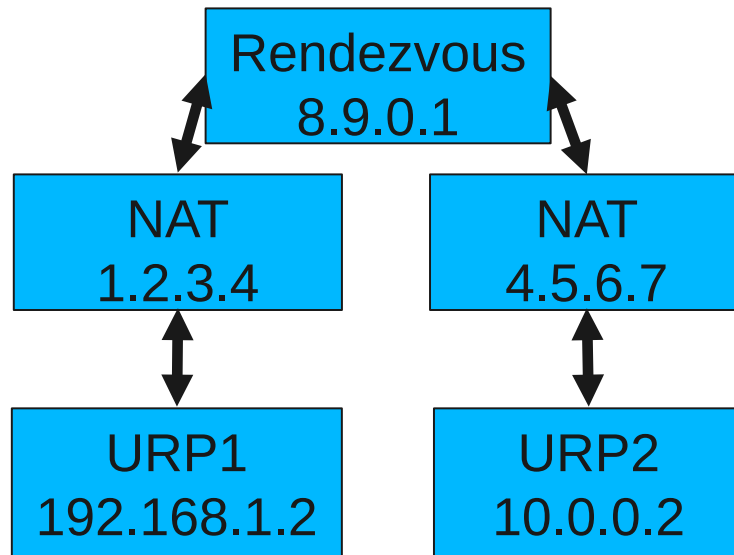
```
ip netns exec testnet nc -l -p 8000
```

- Server can be contacted
- How to connect to outside?
 - E.g. layer 3

```
echo 1 > /proc/sys/net/ipv4/ip_forward
ip netns exec testnet ip route add default via 10.1.1.1 dev veth1
iptables -t nat -A POSTROUTING -s 10.1.1.0/24 -o enp6s0 -j MASQUERADE
iptables -A FORWARD -j ACCEPT #open up wide...
```

Connectivity, Security, and Robustness

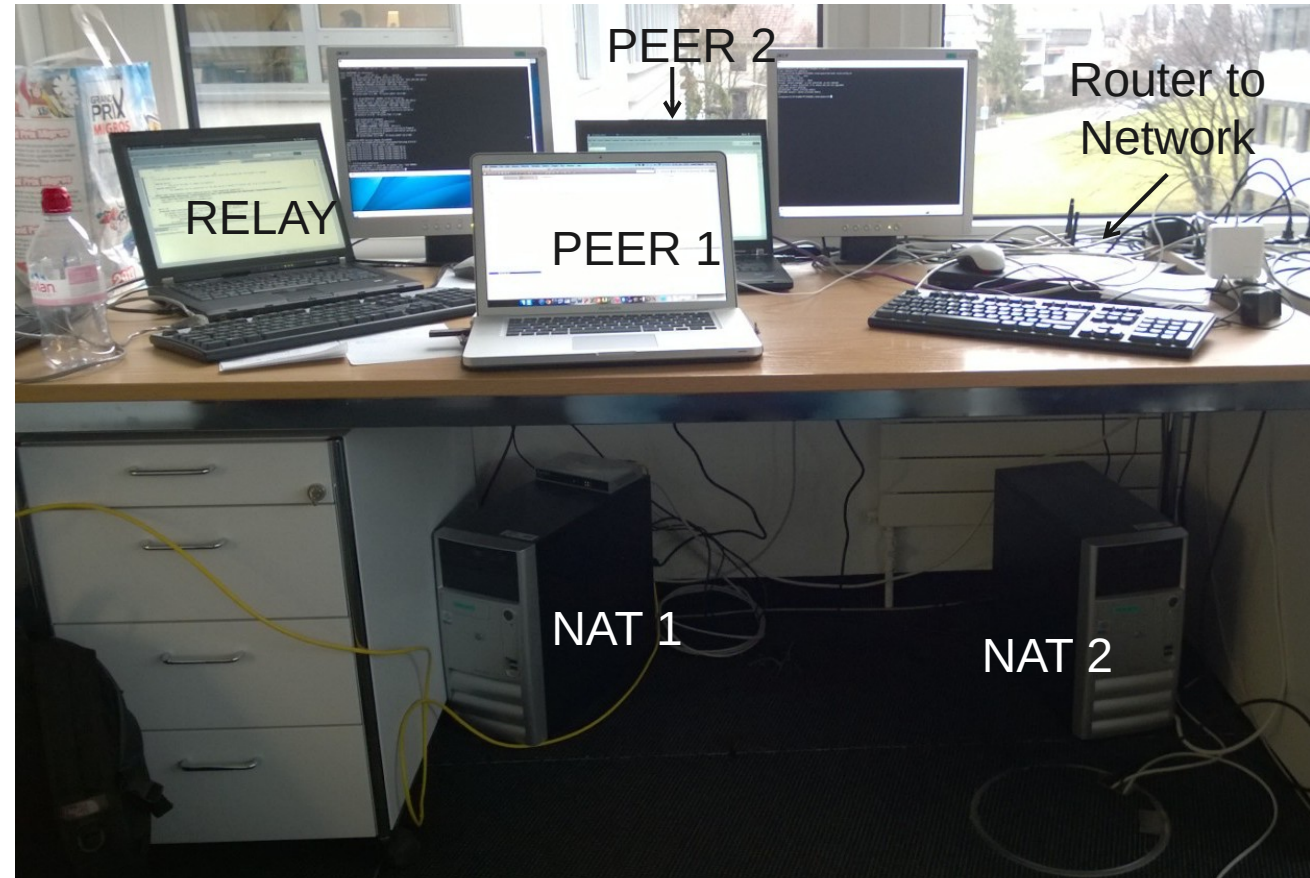
- Hole punching
 - URP1 got 4.5.6.7:5000, URP2 got 1.2.3.4:4000
 - Unreachable peer 1 request to NAT 4.5.6.7, will fail – no mapping, however, unreachable peer 1 creates mapping with that request
 - Unreachable peer 2 sends request to unreachable peer 1 (1.2.3.4:4000) success!



Mapping for NAT 1.2.3.4 (Unreachable peer 1)			
192.168.1.2:4000	4.5.6.7:5000	4.5.6.7:5000	1.2.3.4:4000
Mapping for NAT 4.5.6.7 (Unreachable peer 2)			
10.0.0.2:5000	1.2.3.4:4000	1.2.3.4:4000	4.5.6.7:5000

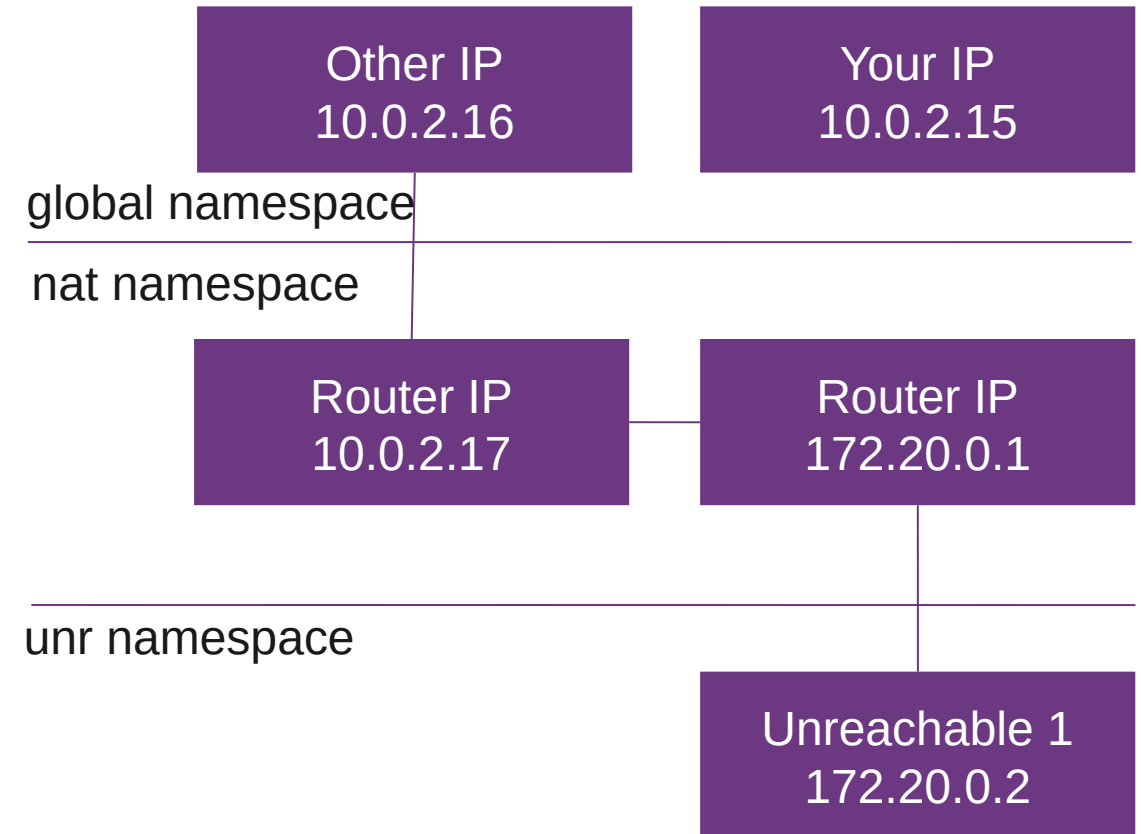
Connectivity, Security, and Robustness

- P2P / Hole Punching Development (in the old days)
- Currently: network namespaces (since Linux 2.6.24)



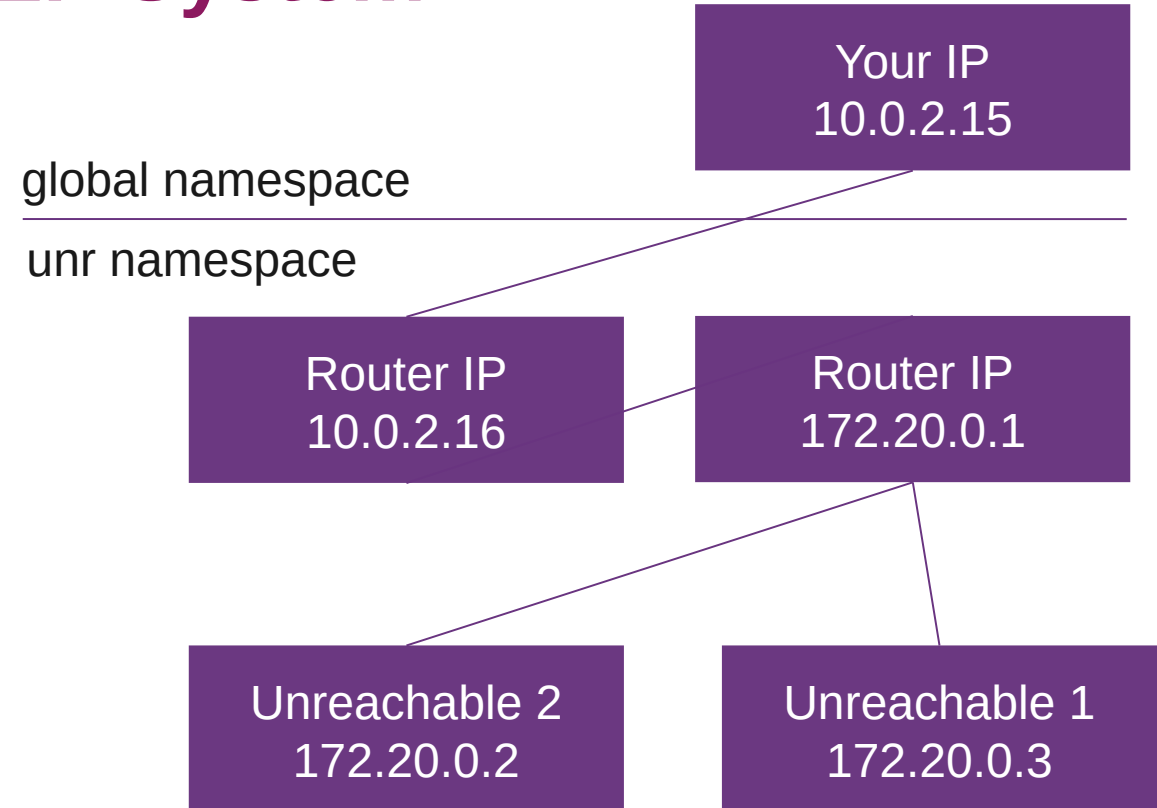
Make your own Testbed for P2P System

- veth - Virtual Ethernet Device
 - Tunnels between network namespaces
 - `ip netns add unr / ip netns list`
 - `ip link add nat_lan type veth peer name nat_wan`
 - `ip link set nat_lan netns unr`
 - `ip address add 10.0.2.16/24 dev nat_wan`
 - `ip link set nat_wan up`
 - `ifconfig / ping`
 - `ip netns exec unr ip address add 172.20.0.1/24 dev nat_lan`
 - `ip netns exec unr ip link set nat_lan up`



Make your own Testbed for P2P System

- Setup 2 unreachable peers
 - `ip netns exec unr ip link add unr1 type dummy`
 - `ip netns exec unr ip address add 172.20.0.2/24 dev unr1`
 - `ip netns exec unr ip link set unr1 up`
 - `ip netns exec unr ifconfig`
 - `ip netns exec unr ip link set lo up`
 - `ip netns exec unr route add default gw 172.20.0.1`
 - `ip route add 172.20.0.1 dev nat_wan`



Docker Compose

- Docker Compose to deploy multiple containers
 - E.g, load balancer, services, DB
 - Configure your services
 - Lightweight orchestration
 - Start service depending on others (e.g., Postgres)

```
#docker-compose.yml
services:
  server1:
    build: .
  client:
    image: alpine
    entrypoint: >
      sh -c "sleep 3 && echo hallo | nc server1 8081"
```

- Dockerfile
 - Build your binary with a Dockerfile - [example](#)
 - Create our own image with a Dockerfile
 - keep images small
 - Multi-stage builds, copy required files
 - Remove cache, as docker is caching aggressively

```
#Dockerfile
FROM golang:alpine AS builder
WORKDIR /build
COPY server.go .
RUN go build server.go

FROM alpine
WORKDIR /app
COPY --from=builder /build/server .
ENTRYPOINT ["/server"]
```

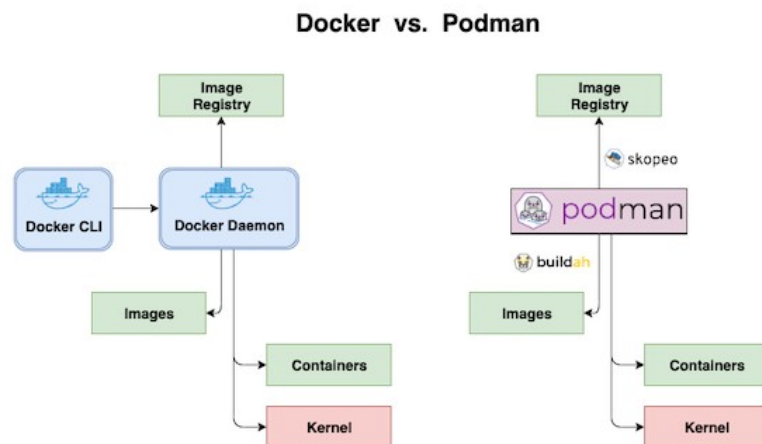
Docker Security

- Best practices [\[link\]](#)
 - Keep images small, up to date / attack surface
 - Use tiny runtimes – [alpine](#) / [distroless](#), avoid big images Ubuntu/Arch/Debian/Fedora
 - My preference alpine (switched from distroless to alpine due to convenience)
 - Check your image for vulnerabilities, [snyk](#), [trivy](#)
 - Do not expose the Docker daemon socket (even to the containers)
 - But traefik...?
- Set a user – do not run as root
 - Needs a bit more configuration, but good advice
- Limit capabilities (Grant only specific capabilities, needed by a container)
 - Seems overkill
- Disable inter-container communication
 - Ok in prod mode, not in dev mode
- Limit resources (memory, CPU, file descriptors, processes, restarts)
 - Good for production → docker does not have a hard memory limit. Docker kills process that goes over limit, no pressure for GC if not docker aware
- Set filesystem and volumes to read-only
 - Very good advice

Docker Security

- Lint the Dockerfile at build time
 - Use an IDE
- Run Docker in root-less mode
 - Architectur better suited for root-less: podman
[link] [intro] → no daemon → **systemd**
- Set the logging level to at least INFO
 - Logging practices

- Think of Your Audience
- Use Logging Libraries
- Use a Suitable Log Level
 - TRACE level: this is a code smell if used in production. This should be used during development to track bugs
 - DEBUG level: log at this level about anything that happens in the program.
 - INFO level: log at this level all actions that are user-driven, or system specific (timers)
 - NOTICE level: this will certainly be the level at which the program will run when in production.



Logging

- WARN level: log at this level all events that could potentially become an error.
- ERROR level: log every error condition at this level.
- FATAL level: too bad, it's doomsday.
- Set the log level per environment. The DEV environment can run in the level DEBUG, the LOCAL environment in the level TRACE, while TEST/STAGE and PROD should run with the level INFO.
- Use Meaningful Messages
- Log in JSON – structured logging
 - Keep the Log Structure Consistent
- Avoid Logging Sensitive Information
 - **Never ever log sensitive data in non local environments. Only do this temporarily in local environments**
- Secret management
 - Use SaaS solutions / e.g., github secrets. Cloud-based: HashiCorp Vault, AWS Secrets Manager or the GCP Secret Manager, or simplified: git secret [[link](#)], [[link](#)], [[link](#)]