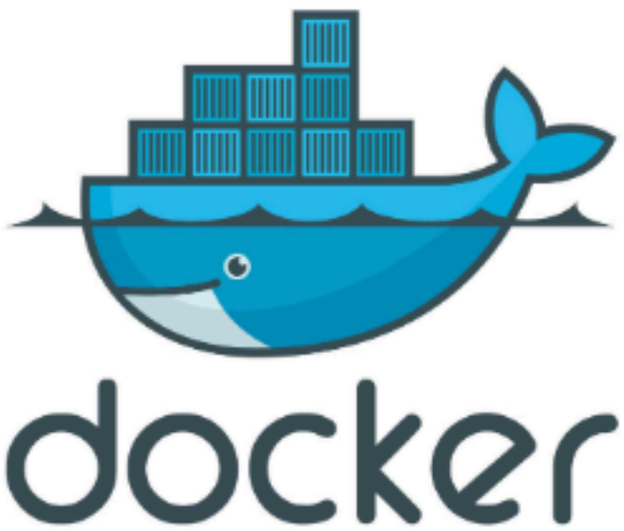
Docker: Zero to Hero

l.sharifi@urmia.ac.ir



old logo

new logo

History

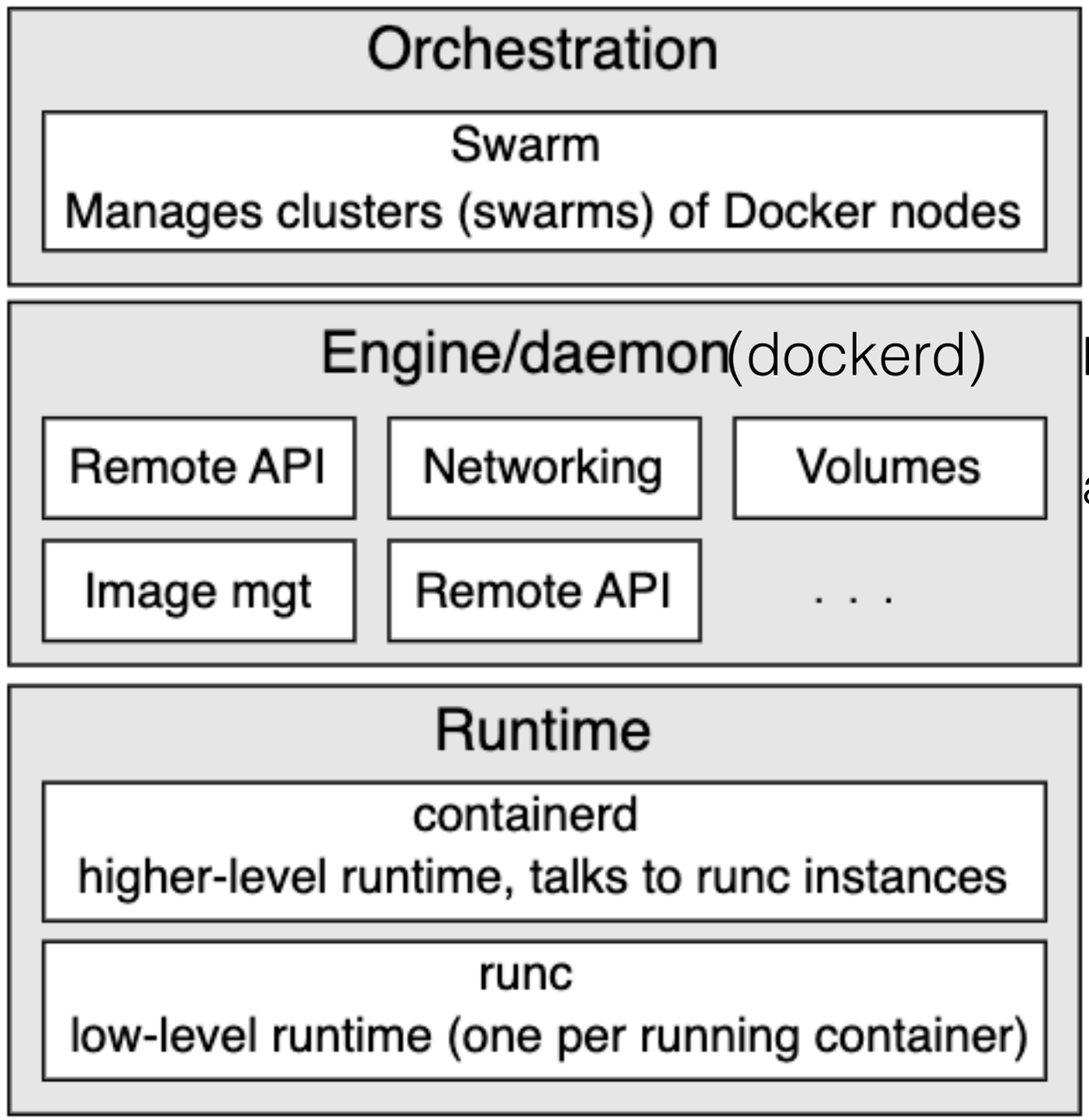
- Various tools built from Moby open source project
- Docker Inc.
  - Solomon Hykes is the founder
  - First company name idotCloud (PaaS)
  - In 2013 changed name to Docker: comes from British expression dock worker.

Introduction

- Docker facilitates:
    - Create (Build)
    - Distribute (Release)
    - Run
- Runs Linux containers on windows: Hyper V or Windows subsystem on Linux (WSL)

Docker Architecture

Docker Architecture



Sometimes the set of Daemon + runc+ cintainerd is called ENGINE

Provides easy to use standard interface that abstracts the lower level.

Typically:

- One containerd
- A runs for each container

runc is a reference implementation of OpenContainers Initiative (OCI) runtime space.
It is a CLI wrapper for Libcontainer to create containers.
It is an interface with underlying OS.

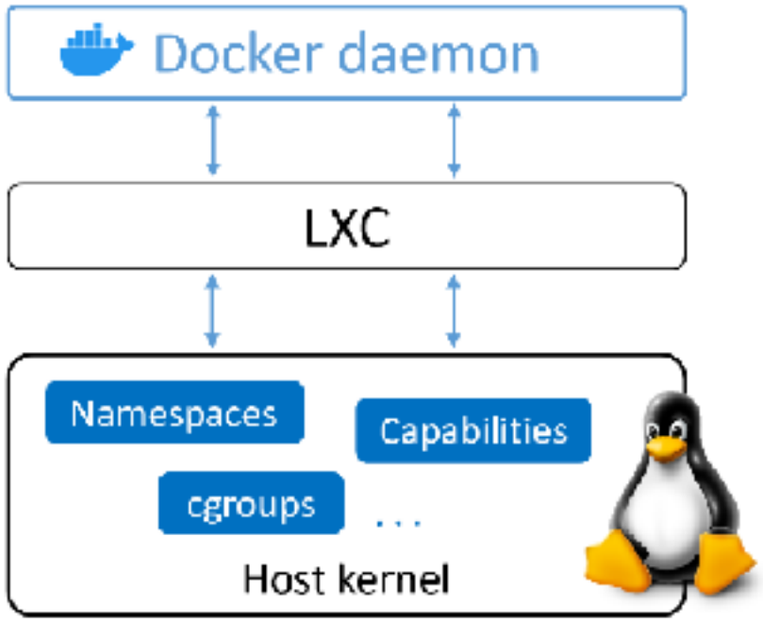
Containerd

- First developed by docker then donated to CNCF
- Cloud Native Computing Foundation (CNCF) Project
- It manages the entire lifecycle of a container:
  - Pulling images
  - Creating network interface
  - Managing lower level runc instances

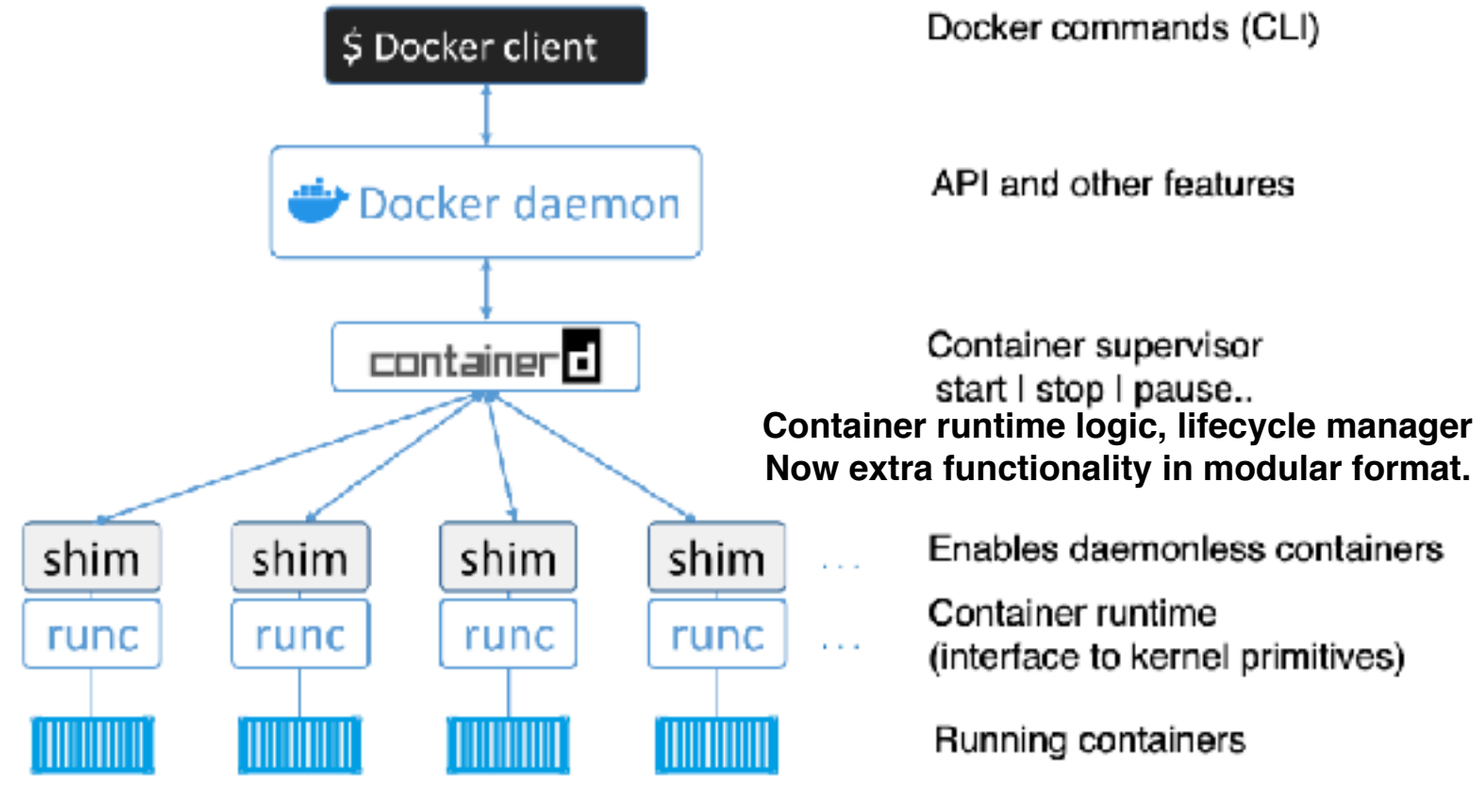
Open Container Initiative (OCI)

- A governance council responsible for standardising the low-level fundamental components of container infrastructure
    - image format (image-spec)
    - container runtime (runtime-spec)
- Initiated by CoreOS company
    - Created an open standard appc by redHat then acquired by IBM
- Docker 1.11 on follow OCI runtime-spec

Docker Engine



Old Engine
(Before Docker 0.9)
Monolitic

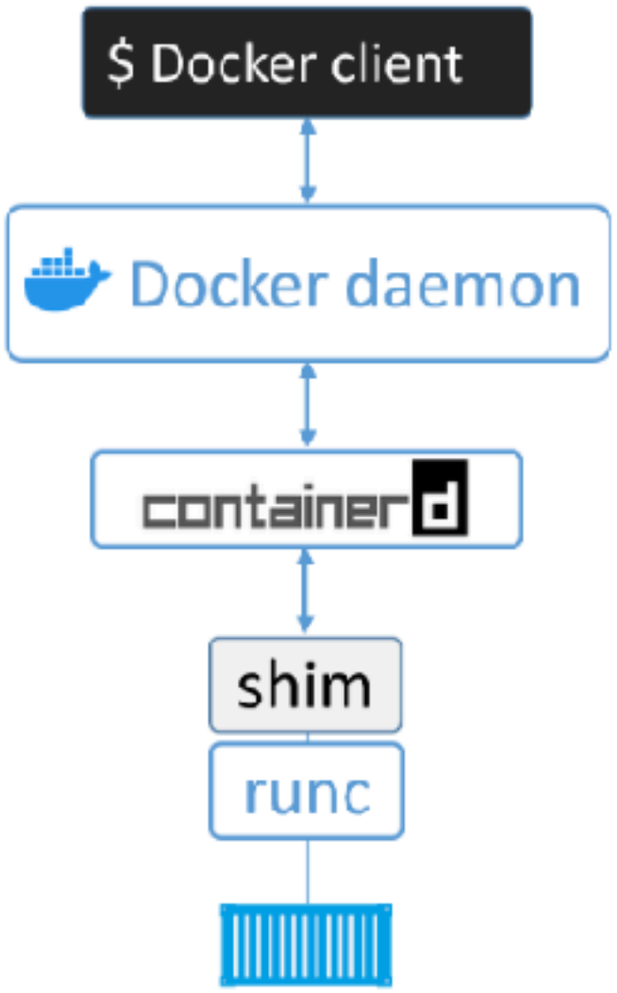


High-level view of the current Docker engine architecture

Modular
(Breaking and refactoring the Daemon)

Start a new container

- $ docker container run - - name ctr1 -it imagename:latest sh
- Docker client converts command into appropriate API payload and POSTs them to the API endpoint exposed by the docker daemon.
    - API can be exposed on local host or network
    - **Linux:** /var/run.docker.sock
    - **Windows**: \pipe\docker_engine



Issue `**docker container run**` command to **Docker API** exposed by Docker daemon

Receive instruction at API endpoint.
Instruct **containerd** (via gRPC API) to start new container based on OCI bundle and ID provided

Receive instruction to create containers
Instruct **runc** to create container.

- **shim:**
- **Keeping any STDIN and STDOUT streams open;**
- **when daemon requested => container doesn't terminated due to pipe being closed.**
- **Reports the container's exit status back to the daemon.**
- **Responsible for daemonless container**

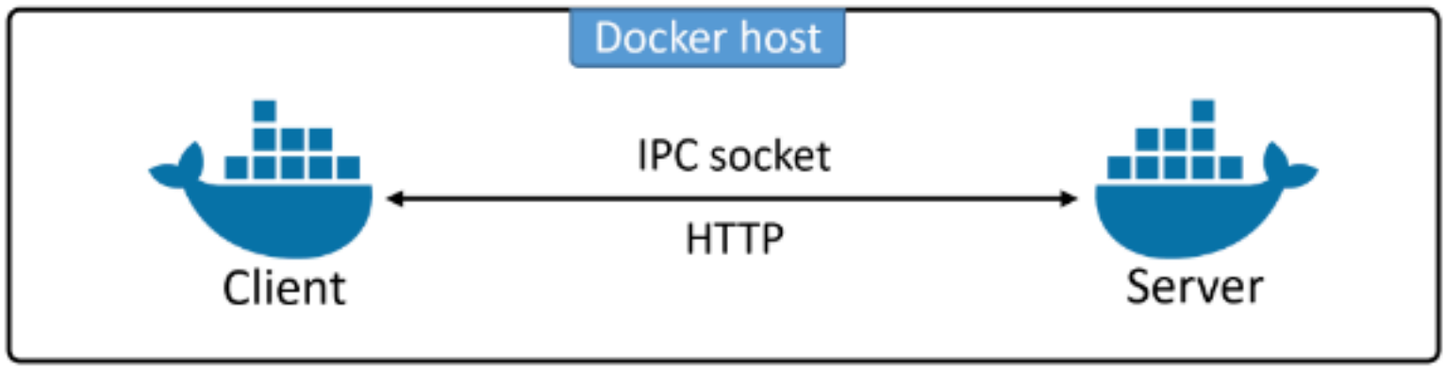
Build and start container
runc exits after container starts
shim becomes container's parent process

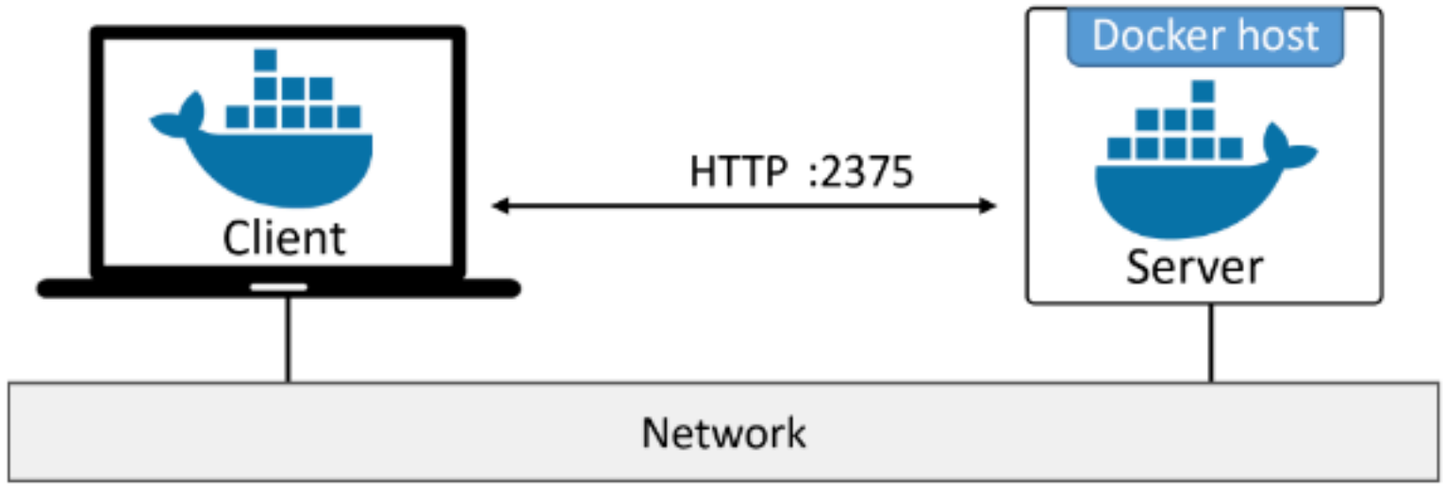
Starting a new container

docker client —> API server (docker daemon) —> Image repo —> docker hub
command start container Search pull image
containerd/ runc

Daemonless container: Entire container runtime is decoupled from the docker daemon.
Makes it possible to maintain and upgrade on docker daemon without impacting running containers.

Docker Security

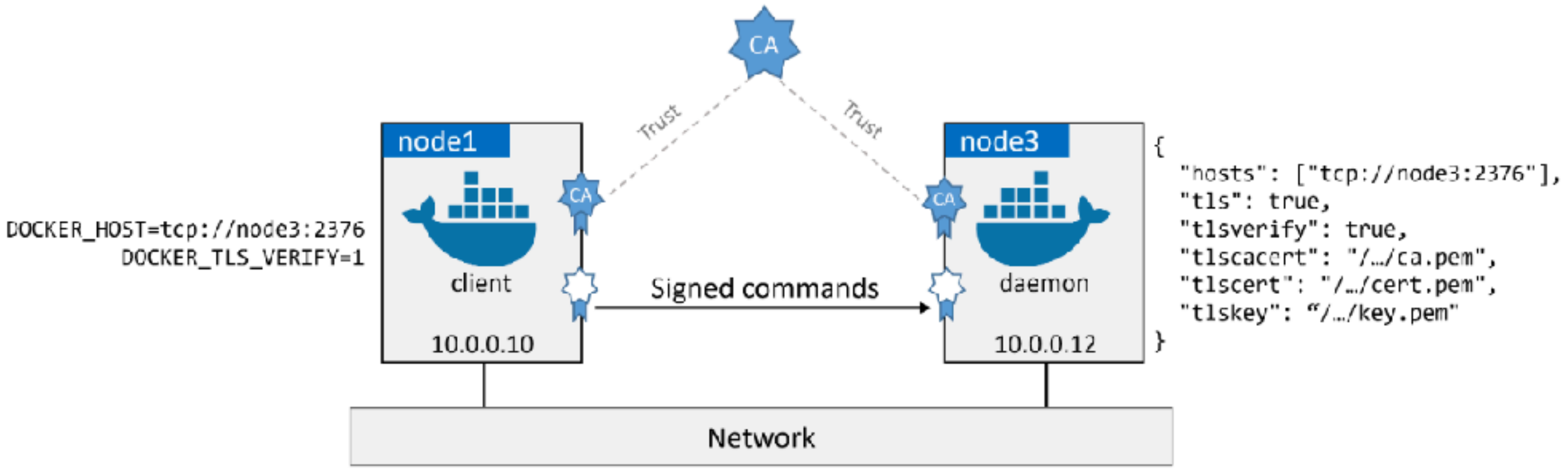




Network communication in Docker

Use TLS and Certificate Authority (CA) for more security in production area.

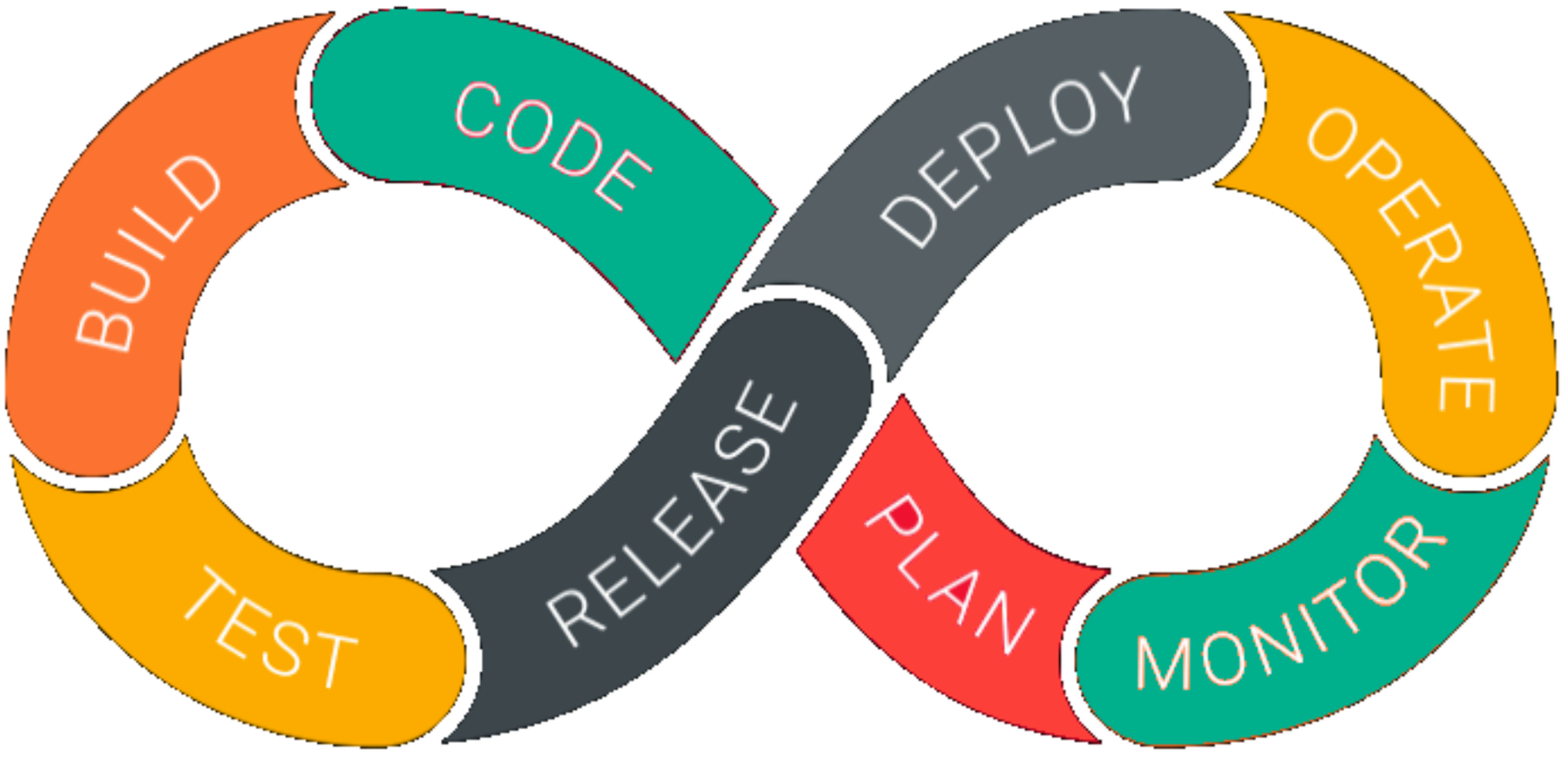
Docker with TLS config



Useful links

- https://www.docker.com/products/docker-desktop/
- https://docs.docker.com/desktop/
- https://play-with-docker.com

DevOps Cycle



Docker

- Dev
    - Image (build-time construct)
        - $ docker image …
    - Compose
        - $ docker compose …
- Ops
    - Container (runtime-construct)
        - $ docker container …
    - Network
        - $ docker network …
    - Stack
        - $ docker stack …

Docker commands

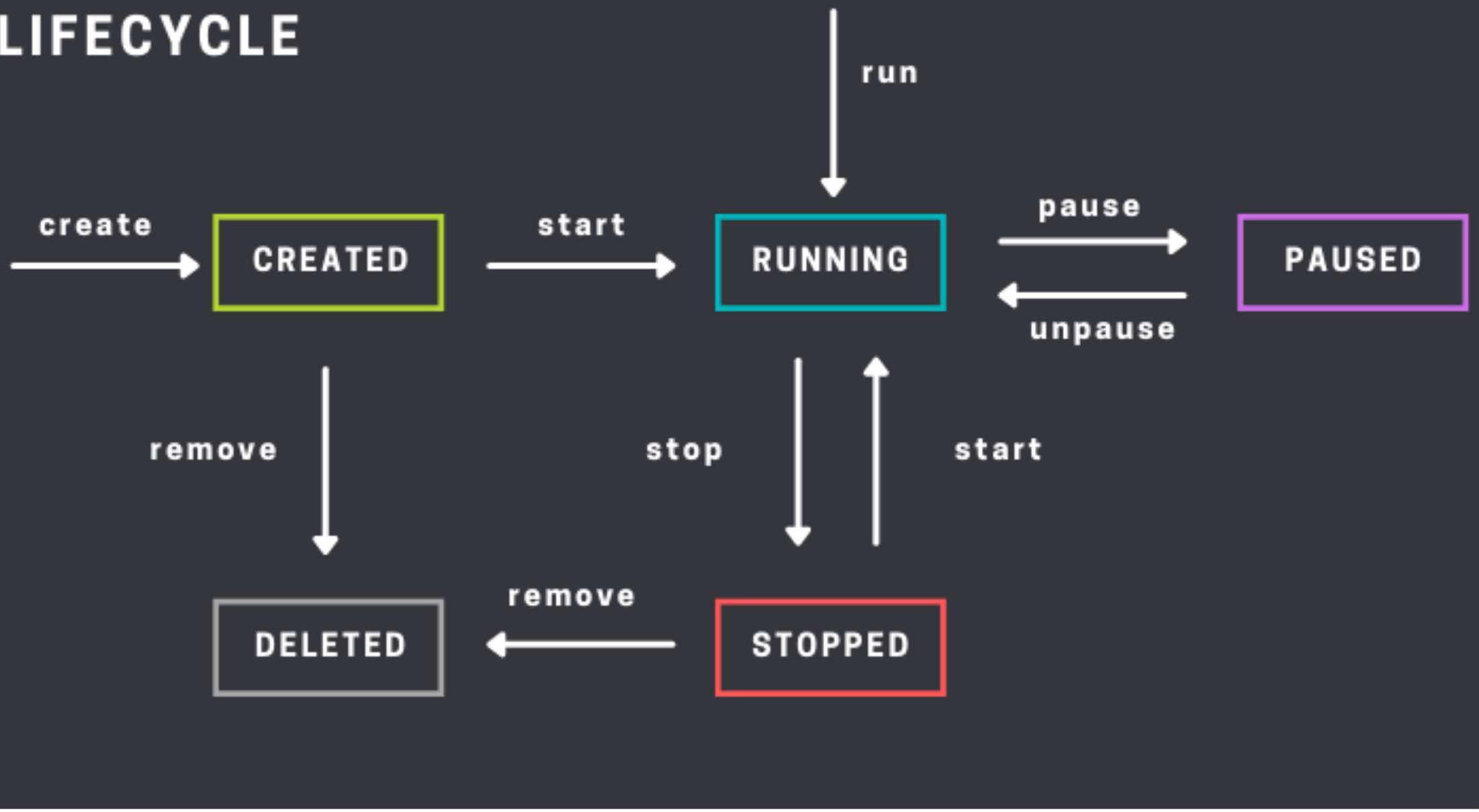
- Check if the service is running
  - systemd: $ systemctl is-active docker
  - non-systemd: $service docker status
  - Windows Powershell: > Get-service docker

Container

- Is a logical construct within OS
  - Independent packages
  - Independent NameSpace (Isolation)
  - Lightweight VMs

DOCKER CONTAINER LIFECYCLE

LINUX HANDBOOK



Docker Commands

- You can directly attach the container running in detached mode to the shell using
  - $docker attach <container id>.
    - But you will not see any output as no process in the container is printing to stdout of the container.
    - WARNING: If you accidentally press ctrl+c or any kill signal to get back to the shell, the application will stop.
- docker logs command to interact with the detached containers.
  - $docker logs <CONTAINER ID> >output.log

Container Commands

- **docker container run** is the command used to start new containers. In its simplest form, it accepts an image and a command as arguments. The image is used to create the container, and the command is the application the container will run when it starts. This example will start an Ubuntu container in the foreground and tell it to run the Bash shell: docker container run -it ubuntu /bin/bash. -d flag is to run container in the background, i.e. daemon mode. -p flag sets the host port: container port.
    - When you don't see the port in $docker ps, it means the port is in use.
    - lost -i TCP:<port> prints all the services using the specific port
    - restart policies: $docker container run —name example -it —restart always sh
        - always: always restarts after kill
        - unless-stopped: restart unless it is stopped with exit 0
        - on-failed: restarts if exit with non-zero code
- **Ctrl-PQ** will detach your shell from the terminal of a container and leave the container running (UP) in the background.
- **docker container ls** lists all containers in the running (UP) state. If you add the -a flag, you will also see containers in the stopped (Exited) state.
- **docker container exec** runs a new process inside of a running container. It's useful for attaching the shell of your Docker host to a terminal inside of a running container. This command will start a new Bash shell inside of a running container and connect to it: docker container exec -it <container-name or container-id> bash. For this to work, the image used to create the container must include the Bash shell.
- **docker container stop** will stop a running container and put it in the Exited (0) state. It does this by issuing a SIGTERM to the process with PID 1 inside of the container. If the process has not cleaned up and stopped within 10 seconds, a SIGKILL will be issued to forcibly stop the container. docker container stop accepts container IDs and container names as arguments.
- **docker container start** will restart a stopped (Exited) container. You can give docker container start the name or ID of a container.
- **docker container rm** will delete a stopped container. You can specify containers by name or ID. It is recommended that you stop a container with the docker container stop command before deleting it with docker container rm.
- **docker container inspect** will show you detailed configuration and runtime information about a container. It accepts container names and container IDs as its main argument.

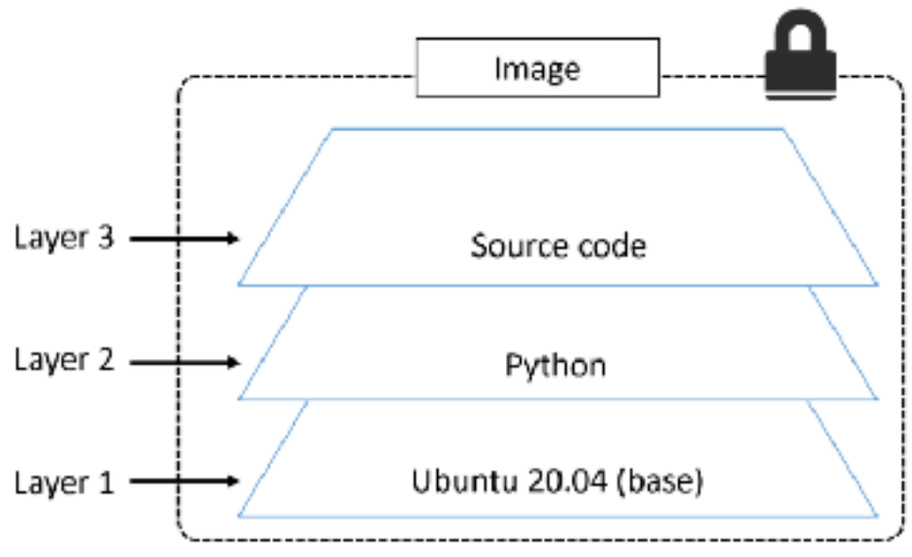
Docker Image

Image

- build-time construct
    - Tiny images (no kernel): 5MB
    - Normal Images: 40 MB
    - Windows Images: GB
- local image repo:
    - Linux: /var/lib/docker/<storage-driver>
    - Windows: c:\ProgramData\Docker\windowsfilter
- Official and unofficial repos
- Microsoft repos: mcr.microsoft.com

Image Layers

- $ docker image inspect : to get layer info
- Multiple images can and do share layers. This leads to efficiencies in space and performance.



A Docker image with 3 layers: Ubuntu, Python, and source code

Squash the image

- Making single layer image
    - $ docker image build - -squash
- Suitable for images with too many layers
- Downsides:
    - Storage inefficiency
    - Large pull and push
    - No shared underlying layer with other images

Digest

- Docker 1.10 introduced a content-addressable storage model. As part of this model, all images get a cryptographic content hash (Digest).

- $ docker image pull <image-name> - -digests

Distribution Hash

- each layer also gets something called a distribution hash.
    - This is a hash of the compressed version of the layer and is included with every layer pushed or pulled to a registry.
    - This can then be used to verify that the layer arrived without being tampered with.
    - As well as providing a cryptographically verifiable way to verify image and layer integrity, it also avoids ID **collisions** that could occur if image and layer IDs were randomly generated.

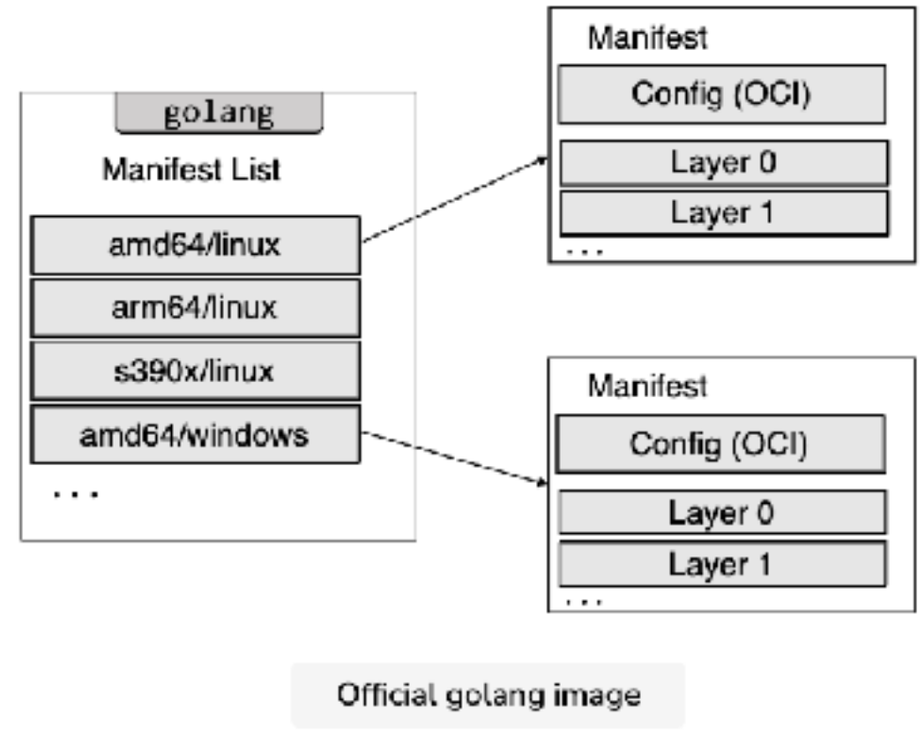
Multi-architecture Images

- A single image tag supporting multiple platforms and architectures.

- To make this happen, the Registry API supports two important constructs:

- Manifest lists: a list of architectures supported by a particular image tag.

- Manifests: Each supported architecture then has its own manifest that details the layers that make it up.



Manifest

- $ docker manifest inspect <imagename> | grep ‘architecture\|os'
- You can create your own builds for different platforms and architectures with docker buildx, and then use docker manifest create to create your own manifest lists.
  - $ docker buildx build --platform linux/arm/v7 -t myimage:arm-v7 .
  - $ docker buildx build --platform <archname> -t <imagename>
  - NOTE: buildx is an experimental feature and requires experimental=true setting in your ~/.docker/config.json file.

```
{
  "experimental": true
}
```

```
$ export DOCKER_CLI_EXPERIMENTAL=enabled
```

Image Commands

- docker image pull is the command to download images. We pull images from repositories inside of remote registries. By default, images will be pulled from repositories on Docker Hub. -a pull all versions, it doesn't work with multi-version and musty-architecture images.
    - $ docker image pull <repo/containername:tag> e.g. gcr.io/google-container
- docker image push <image name> default registry as per docker info i.e.index.docker.io/ repository per docker image/imagename:tag
- docker image inspect is a thing of beauty! It gives you all of the glorious details of an image—layer data and metadata.
- docker manifest inspect allows you to inspect the manifest list of any image stored on Docker Hub. This will show the manifest list for the redis image: docker manifest inspect redis.
- docker buildx is a Docker CLI plugin that extends the Docker CLI to support multi-arch builds.
- docker image rm is the command to delete images. This command shows how to delete the alpine:latest image: docker image rm alpine:latest. You cannot delete an image that is associated with a container in the running (Up) or stopped (Exited) states.
    - Delete all images: $docker image rm $(docker image ls -q) -f
- docker image history shows all the commands used to build the image

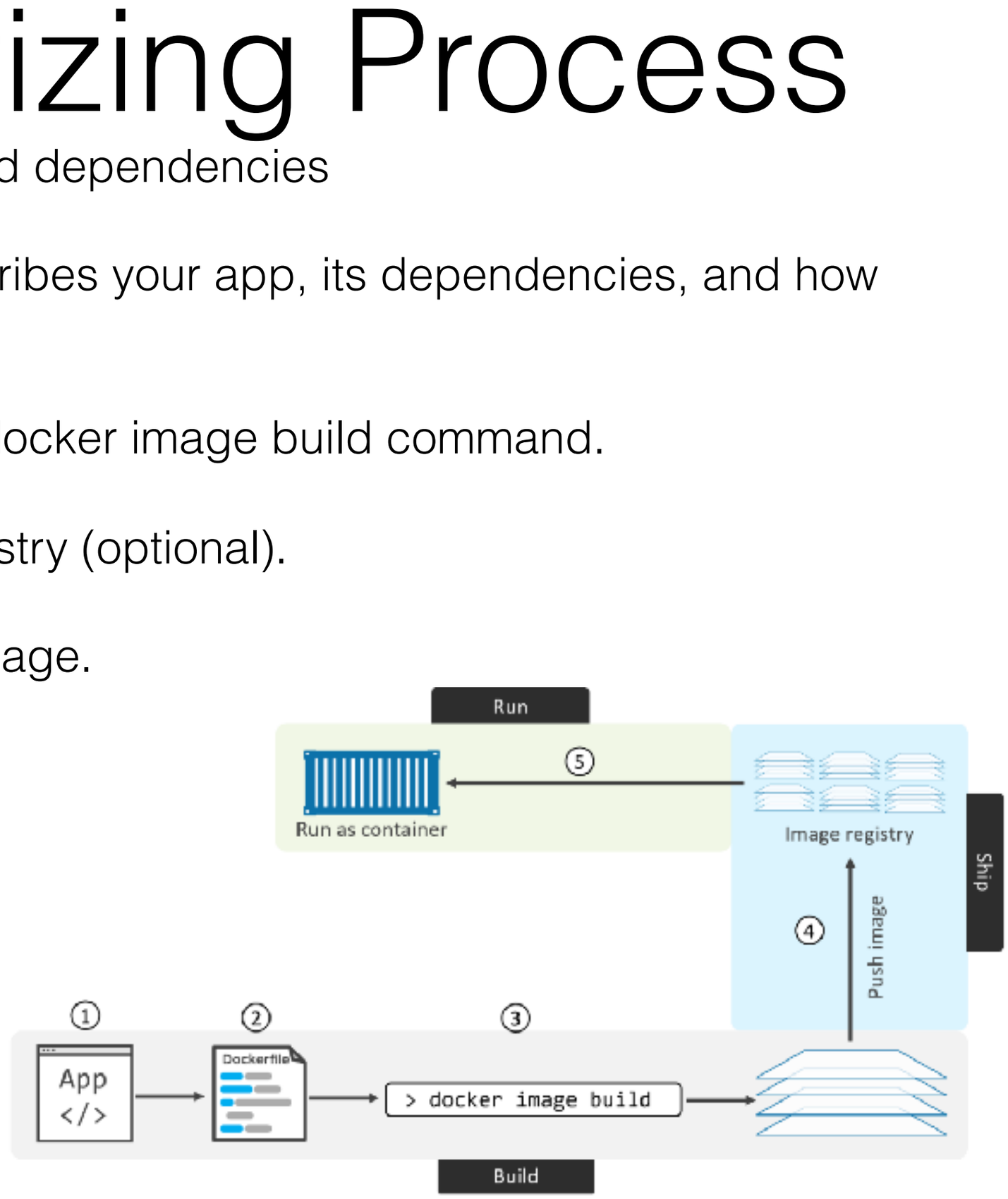
Image Commands…

- docker image ls lists all of the images stored in your Docker host's local image cache.
  - To see the SHA256 digests of images add the --digests flag.
- - - filter flag, e.g. $docker image ls - - filter dangling = true, (filter list includes: dangling, before, since, label).
  - $ docker image ls - -filter = reference= "*.latest"
  - $ docker image ls —format "{{.Size}}" or "{{.Repository}}: {{.Tag}}:{{.Size}}"
  - $ docker search <image name> —filter "is-official= true" "is_automated=true" - -limit 100

Containerizing an App

Containerizing Process

- Start with your application and dependencies
- Create a Dockerfile that describes your app, its dependencies, and how to run it.
- Feed the Dockerfile into the docker image build command.
- Push the new image to a registry (optional).
- Run the container from the image.



Dockerfile ×

```dockerfile
FROM python:3.9-rc-buster

# Setting up Docker environment
WORKDIR /code
# Export env variables.
ENV FLASK_APP app.py
ENV FLASK_RUN_HOST 0.0.0.0
###


#Copy requirements file from current directory to file in
#containers code directory we have just created.
COPY requirements.txt requirements.txt

#Run and install all required modules in container
RUN pip3 install -r requirements.txt

#Copy current directory files to containers code directory
COPY . .

#RUN app.
CMD ["flask", "run"]
```

Dockerfile

Note that FROM, COPY and RUN add new layer to image.

```
FROM python:3.9-rc-buster

# Setting up Docker environment
WORKDIR ./code  # set the working dir in the image
# Export env variables.
ENV FLASK_APP app.py
ENV FLASK_RUN_HOST 0.0.0.0
###


#Copy requirements file from current directory to file in
#containers code directory we have just created.
COPY requirements.txt requirements.txt

#Run and install all required modules and apps in container
RUN pip3 install -r requirements.txt

#Copy current directory files to containers code directory
COPY . .

EXPOSE 8080

#RUN app.
CMD ["flask", “run”]   #or
ENTRYPOINT ["flask", “run"]
```

Best Practice

- Each RUN adds a new layer to image
    - agains the lightweightness goal
        - Include multiple commands as part of a single RUN using &&, \
- To keep image lightweight in Linux
    - apt-get install no-install-recommends
        - Only installs main dependencies

Docker Build Cache

- The $docker image build process iterates through a Dockerfile one line at a time starting from the top.
- For each instruction, Docker looks to see if it already has an image layer for that instruction in its cache.
    - If yes: cache hit; it uses that layer —> huge speedup in the build process
    - else: cache miss; it builds a new layer from the instruction
- NOTE: The operation of invalidating the cache invalidates it for the reminder of the build.
- Ignoring the entire cache during build
        - $ docker image build —no-cache = true
- If the instruction doesn't change but the content, e.g. COPY ./src, where the content of src file has changed, comparing the file checksum update can be detected.

Containerizing an App

- **docker image build** is the command that reads a Dockerfile and containerizes an application. The -t flag tags the image, and the -f flag lets you specify the name and location of the Dockerfile. With the -f flag, it is possible to use a Dockerfile with an arbitrary name and in an arbitrary location. The build context is where your application files exist, and this can be a directory on your local Docker host or a remote Git repo.

- docker image tag <current tag> <new tag>

- The FROM instruction in a Dockerfile specifies the base image for the new image you will build. It is usually the first instruction in a Dockerfile and a best-practice is to use images from official repos on this line.

- The RUN instruction in a Dockerfile allows you to run commands inside the image. Each RUN instruction creates a single new layer.

- The COPY instruction in a Dockerfile adds files into the image as a new layer. It is common to use the COPY instruction to copy your application code into an image.

- The EXPOSE instruction in a Dockerfile documents the network port that the application uses.

- The ENTRYPOINT instruction in a Dockerfile sets the default application to run when the image is started as a container.

- Other Dockerfile instructions include LABEL, ENV, ONBUILD, HEALTHCHECK, CMD, VOLUME and more.

Note that FROM, ADD, COPY and RUN add new layer to image.

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Do it yourself!

- Pull the Python 3.8 image and keep track of the number of layers it has fetched.

- Create a Docker volume named app_files.

- Create a container named ‘first_container’ from the Python 3.8 image with the app_files volume attached to it.

- Write a Python program named ‘current_time.py’ to display the current timestamp inside the app_files volume.

- Create another container named ‘second_container’ from the Python 3.8 image and update the Python script located in the ‘app_files’ volume to print the current date.

- https://www.educative.io/courses/working-with-containers-docker-docker-compose/B8jr3MLVy8k

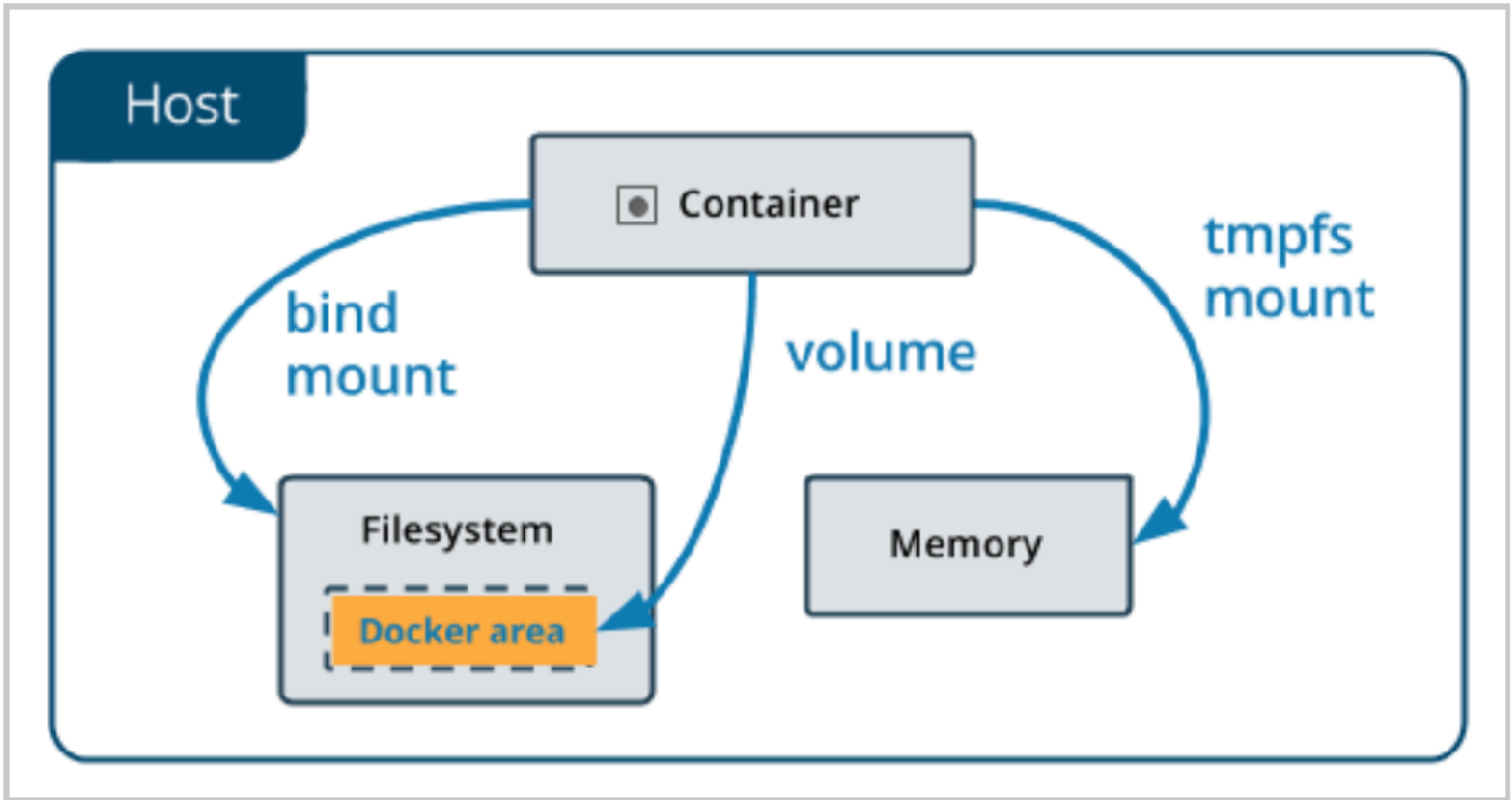
Command	Action
`docker run [image_name]`	Create an instance of the image that is a container. The flag `-d` or `--detach` is used run a container in background mode.
`docker ps`	Lists all running containers. `-a` option will list stopped and running both
`docker inspect [container_name]`	Provides all info about the container
`docker stop [container_name]`	Stops the running container
`docker kill [container_name]`	Kills(stops) the container and removes the container from the system
`docker rmi [image/s]`	Removes the provided image
`docker images`	Lists all images on the system
`docker exec [-it]`	Executes command in a Docker container
`docker system`	Gets the Docker system information such as memory usage and housekeeping stuff
`docker system prune`	This command will save you from getting the "N memory left" nightmare with production system

Data Management for Containers

Data Management

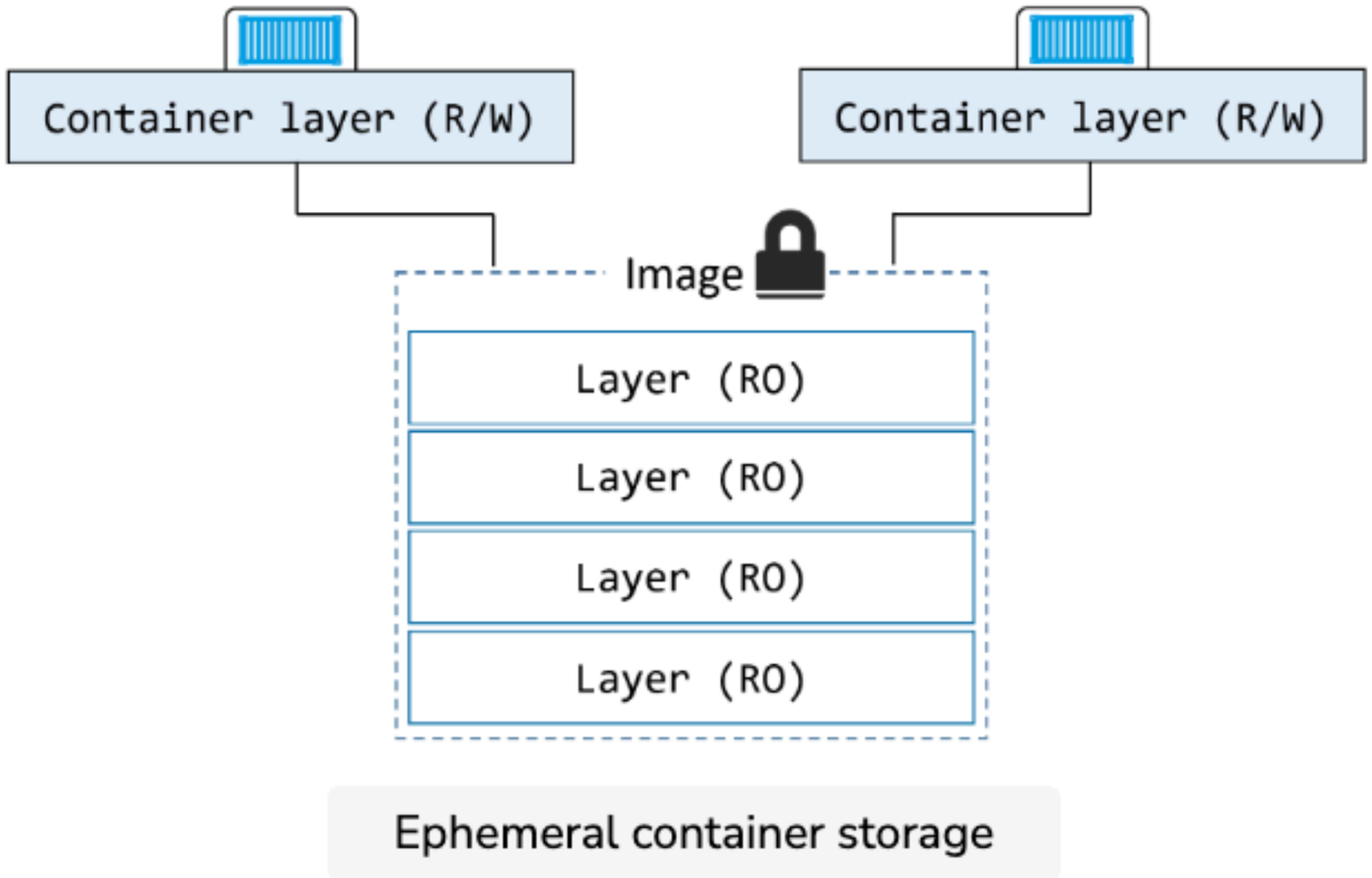
- Whenever a container is created, a file system is also created with it, which is a default Linux filesystem. Although Docker shares the OS's kernel, there is a separation between file systems.

Persistent vs Non-persistent



Non-persistent data store

Read-write Filesystems



Every Docker container is created by adding a thin read-write layer on top of the read-only image it's based on.

Storage Drivers

- This writable layer of local storage is managed on every Docker host by a storage driver (not to be confused with a volume driver).

- If you're running Docker in production on Linux, you'll need to make sure you match the right storage driver with the Linux distribution on your Docker host.

    - Use the following list as a guide:

        - **Red Hat Enterprise Linux:** Use the overlay2 driver with modern versions of RHEL running Docker 17.06 or higher. Use the devicemapper driver with older versions. This applies to Oracle Linux and other Red Hat related upstream and downstream distros.

        - **Ubuntu:** Use the overlay2 or aufs drivers. If you're using a Linux 4.x kernel or higher you should go with overlay2.

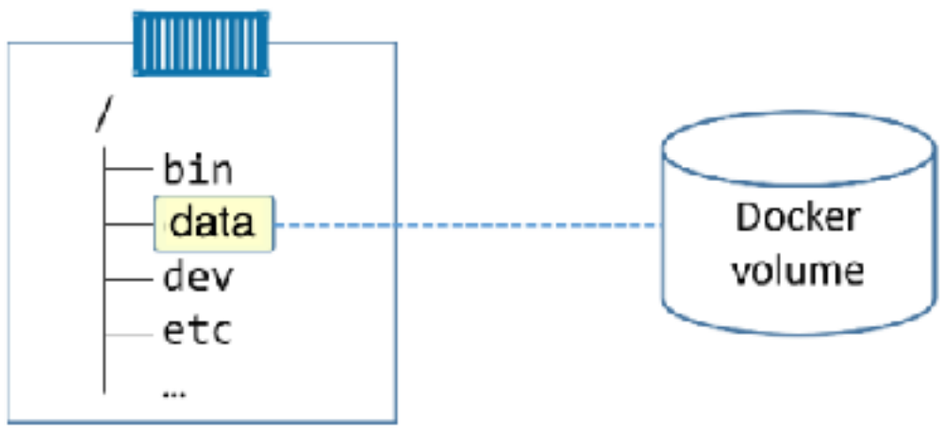
        - **SUSE Linux Enterprise Server:** Use the btrfs storage driver.

- **Windows:** Windows only has one driver and it is configured by default.

Persistent data store

Volumes

- Volumes are independent objects that are not tied to the lifecycle of a container.
- Volumes can be mapped to specialized external storage systems.
- Volumes enable multiple containers on different Docker hosts to access and share the same data.
- At a high level, you create a volume, then you create a container and mount the volume into it. The volume is mounted into a directory in the container's filesystem, and anything written to that directory is stored in the volume. If you delete the container, the volume and its data will still exist.



High-level view of volumes and containers

Related Commands

- docker volume --help: to get the volume help
- docker volume create: to create a new volume
- docker volume inspect: to inspect the created volume
- docker run -v: to mount a volume

Bind mount

- docker run -v: to mount a volume
  - $docker run -it -v <absolute_path>:<folder path or new folder name> date_project:1.0
    - docker run -it -v /Users/Desktop/:/desktop date_project:1.0
  - To mount the file system as read-only, use ro flag.
    - $docker run -it -v <absolute_path>:<folder path or new folder name>:ro date_project:1.0

- **Limitation:dependent on the host's file system. If a folder is accidentally deleted from the host, Docker can't do anything.**

Volumes

- Volumes are created in Docker space.
- Using volumes, we can share data in different containers.
- Volumes are more reliable than bind mounts.
- $docker volume create <volume_name> -d <driver name, default is local>
    - If you don’t provide the volume name, Docker will assign a random unique one.
    - also possible to deploy volumes via Dockerfiles using the VOLUME instruction. The format is VOLUME <container-mount-point>. Interestingly, you cannot specify a directory on the host when defining a volume in a Dockerfile. This is because host directories are different depending on what OS your Docker host is running – it could break your builds if you specified a directory on a Docker host that doesn’t exist. As a result, defining a volume in a Dockerfile requires you to specify host directories at deploy-time.
- docker container run -dit --name leila --mount source=vol_docker,target=/vol alpine
    - NOTE: if the vol_docker doesn’t exist it creates this volume

Commands

- docker volume create creates new volumes. By default, volumes are created with the local driver, but you can use the -d flag to specify a different driver.

- docker volume ls lists all volumes on the local Docker host.

- docker volume inspect shows detailed volume information. Use this command to see many interesting volume properties, including where a volume exists in the Docker host's filesystem.

- docker volume prune deletes all volumes that are not in use by a container or service replica. Use with caution!

- docker volume rm deletes specific volumes that are not in use.

- docker plugin install installs new volume plugins from Docker Hub.

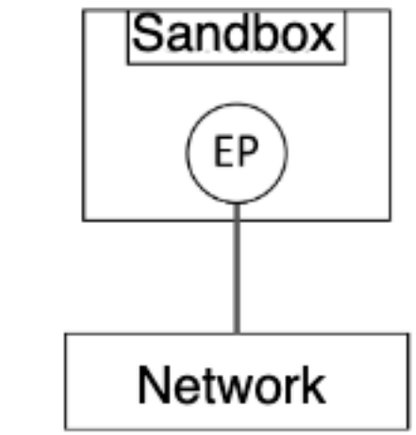
- docker plugin ls lists all plugins installed on a Docker host.

Solving Space issue

- $docker system df
    - #This will print space used by Docker in a human-readable format.
- $docker system prune
    - #This will prompt you to notify what it should remove.
- docker container prune, docker images prune, docker network prune

Networking

Docker networking comprises three major components:



CNM

Design
(DNA)

libnetwork

Primitives
(Control & Management plane)

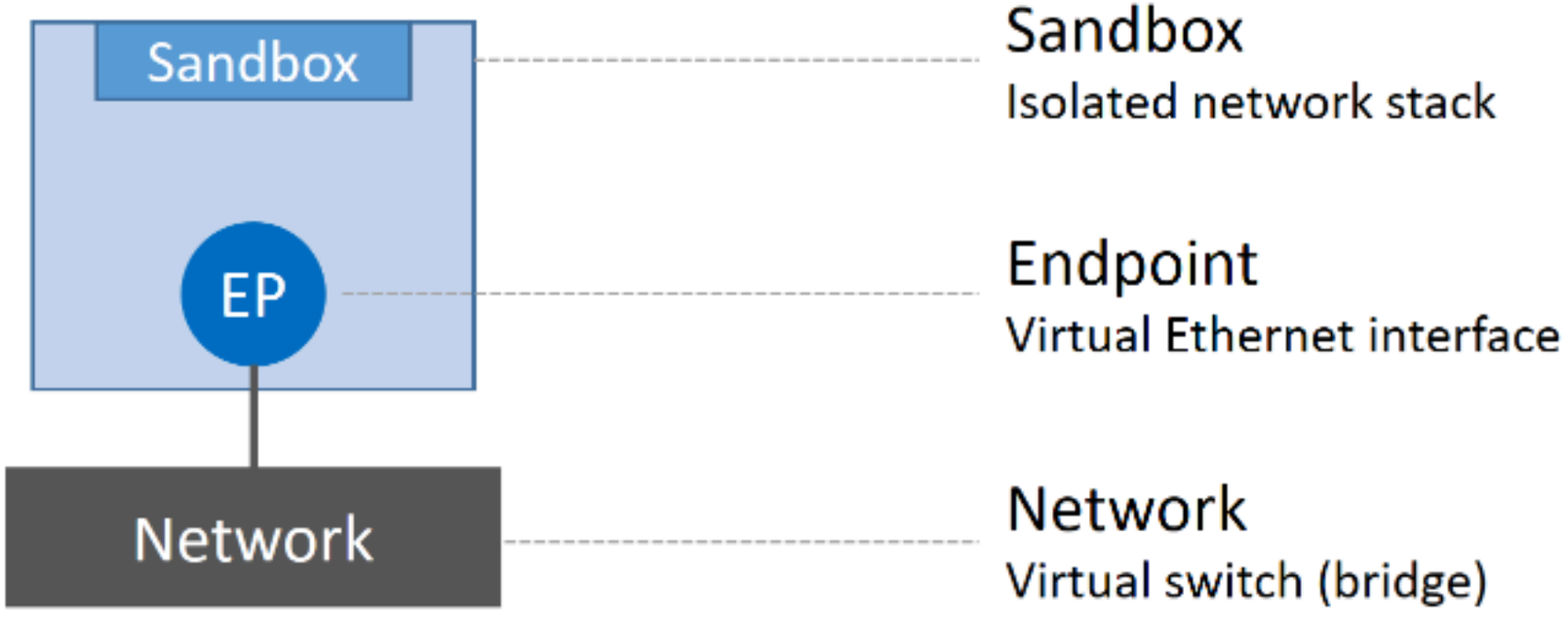
Drivers

Networks
(Data plane)

Container Network Management (CNM)

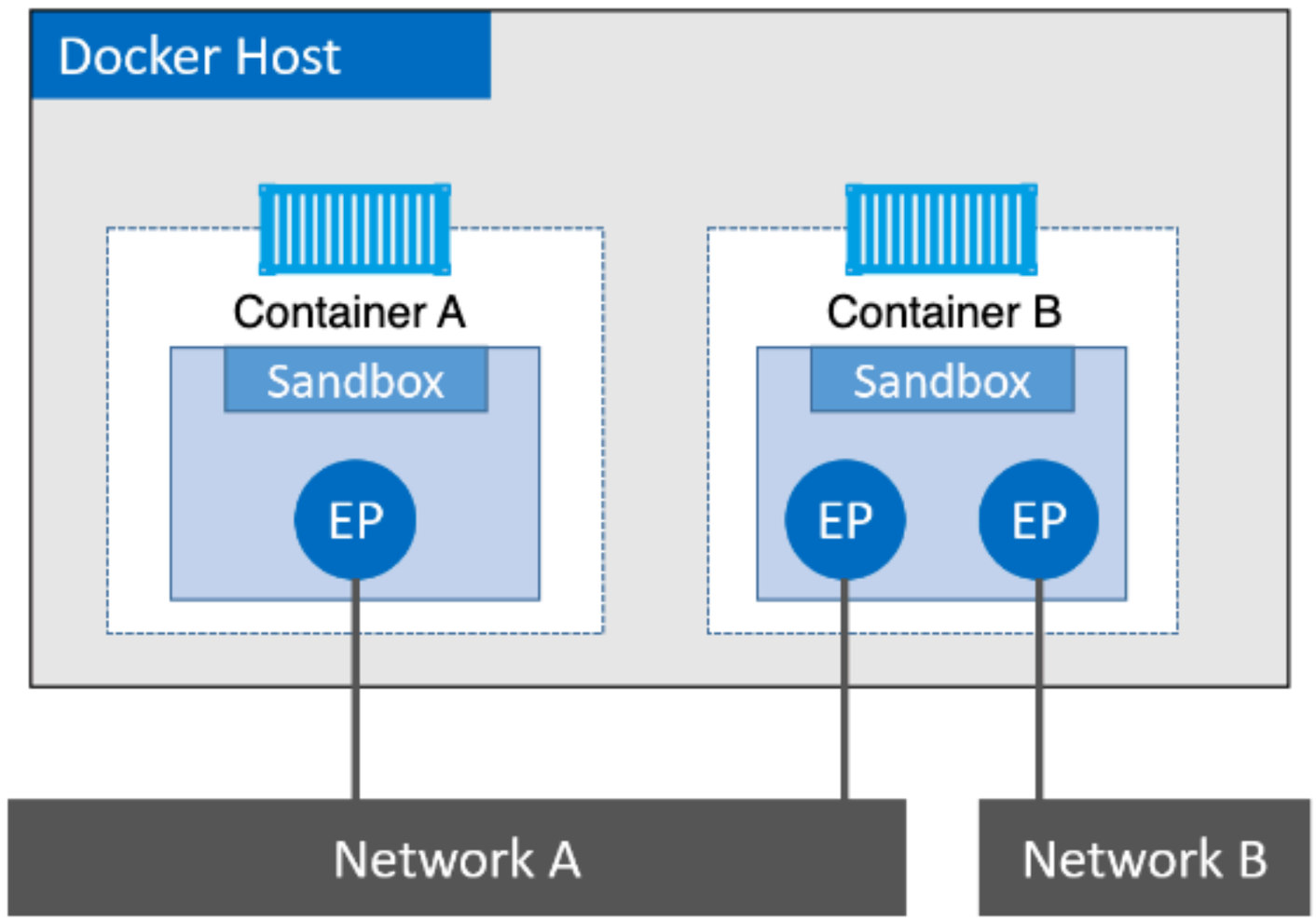
- The CNM is the design specification. It outlines the fundamental building blocks of a Docker network.

- **libnetwork** is a real-world implementation (in Go lang) of the CNM, and is used by Docker.



The Container Network Model

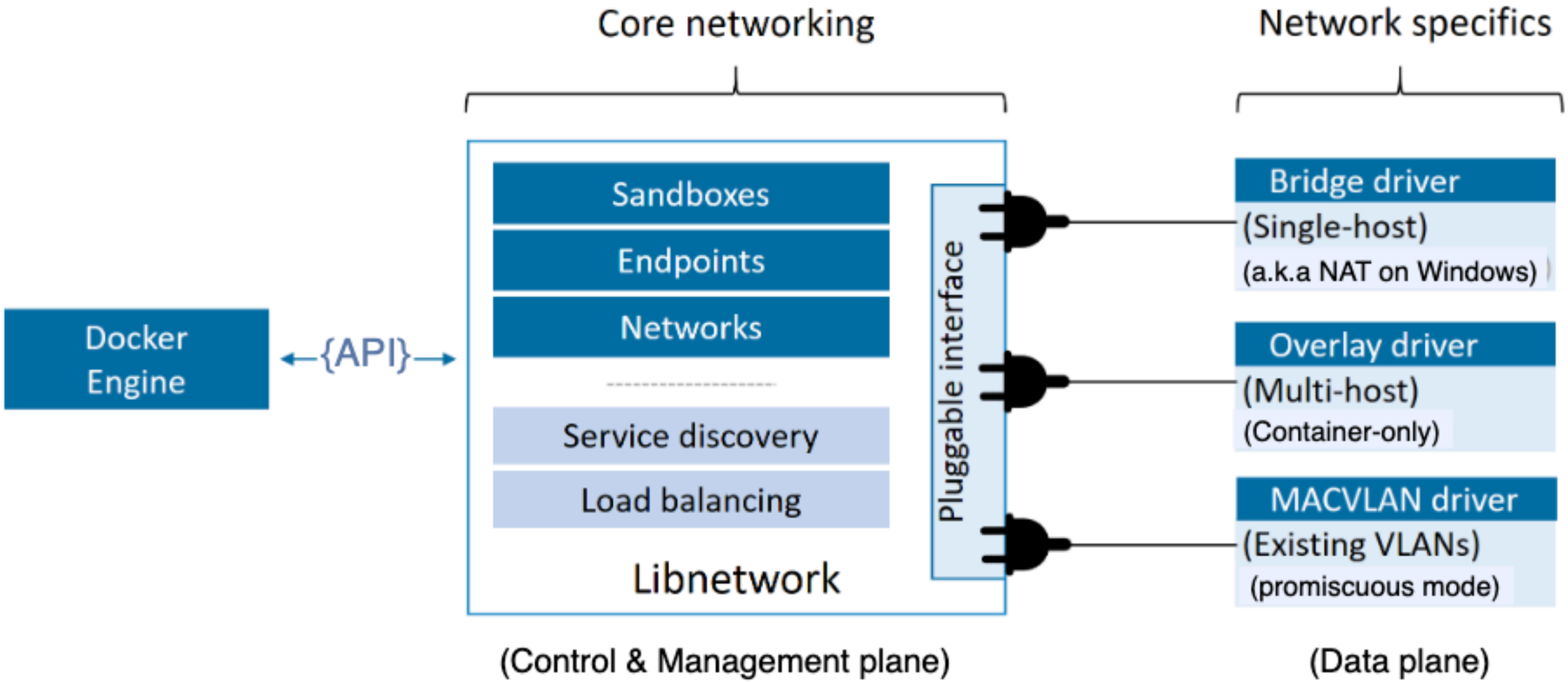
- A **sandbox** is an isolated network stack. It includes; Ethernet interfaces, ports, routing tables, and DNS config.
- **Endpoints** are virtual network interfaces (E.g. veth). Like normal network interfaces, they're responsible for making connections. In the case of the CNM, it's the job of the endpoint to connect a sandbox to a network.
- **Networks** are a software implementation of a switch (802.1d bridge). As such, they group together and isolate a collection of endpoints that need to communicate.



CNM components and containers with a Docker host

Although Container A and Container B are running on the same host, their network stacks are completely isolated at the OS-level via the sandboxes.

Driver



Relationship between control, management, and data planes

Docker ships with several built-in drivers, known as native drivers or local drivers. On Linux, they include bridge, overlay, and macvlan. On Windows, they include nat, overlay, transparent, and l2bridge.

3rd-parties can also write Docker network drivers known as remote driver or plugin.

Networks

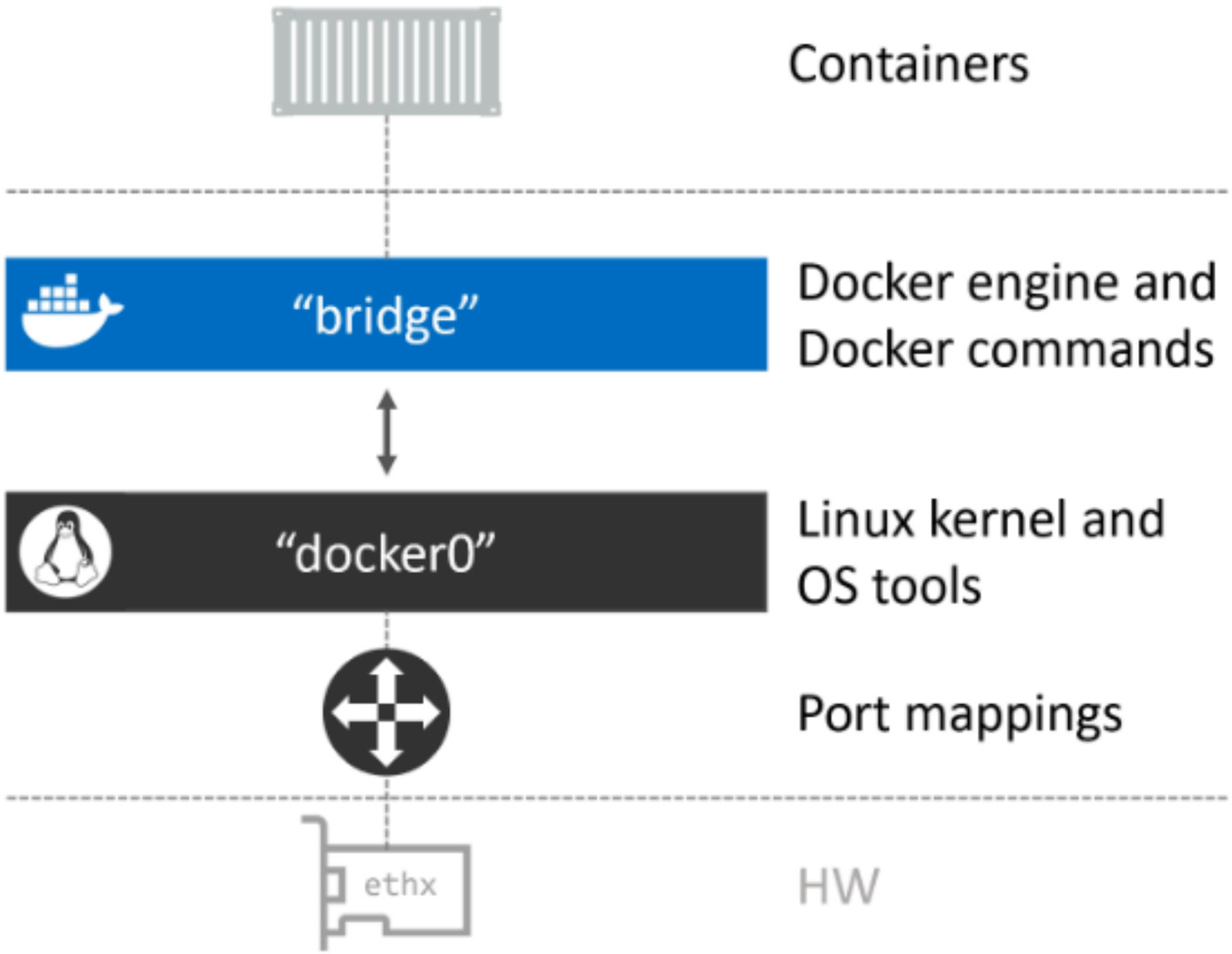
- $docker network ls
- When you initially install Docker, the platform automatically configures three different networks that are named none, host, and bridge.

- The none and host networks cannot be removed, they're part of the network stack in Docker, but not useful to network administrators since they have no external interfaces to configure.

- Admins can configure the bridge network, also known as the Docker0 network. This network automatically creates an IP subnet and gateway.

- All containers that belong to this network are part of the same subnet, so communication between containers in this network can occur via IP addressing.
    - NOTE: A drawback of the default bridge network is that automatic service discovery of containers using DNS is not supported. Therefore, if you want containers that belong to the default network to be able to talk to each other, you must use the --link option to statically allow communications to occur. Additionally, communication requires port forwarding between containers.

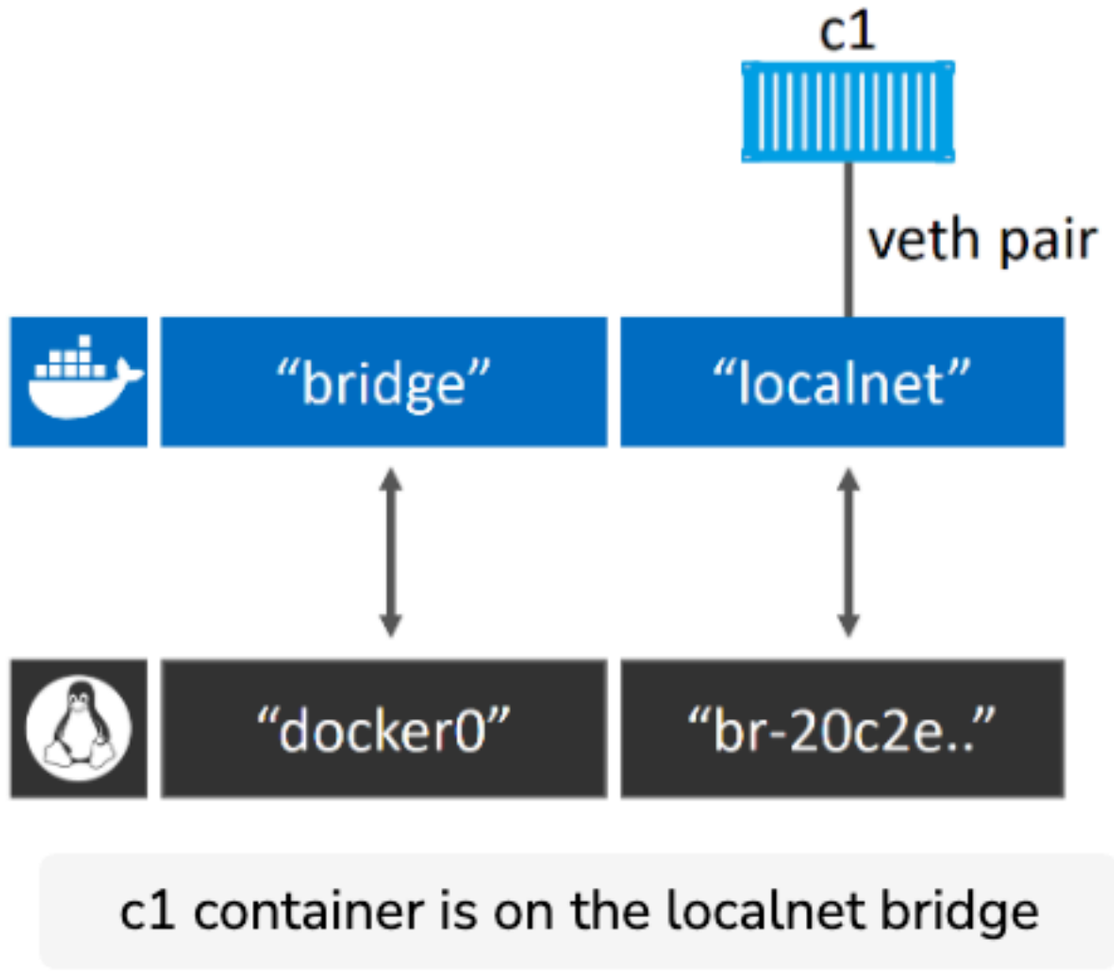
Bridge



The "bridge" network maps to the "docker0" Linux bridge in the host's kernel, which can be mapped back to an Ethernet interface on the host via port mappings.

Bridge Example

- $ docker network create -d bridge localnet
- $ apt-get install bridge-utils
- $ bctl show
- $ docker container run -d --name c1 --network localnet alpine sleep 1d
- $ docker network inspect localnet --format '{{json .Containers}}'
- $ docker container run -it --name c2 --network localnet alpine sh
- $ ping c1



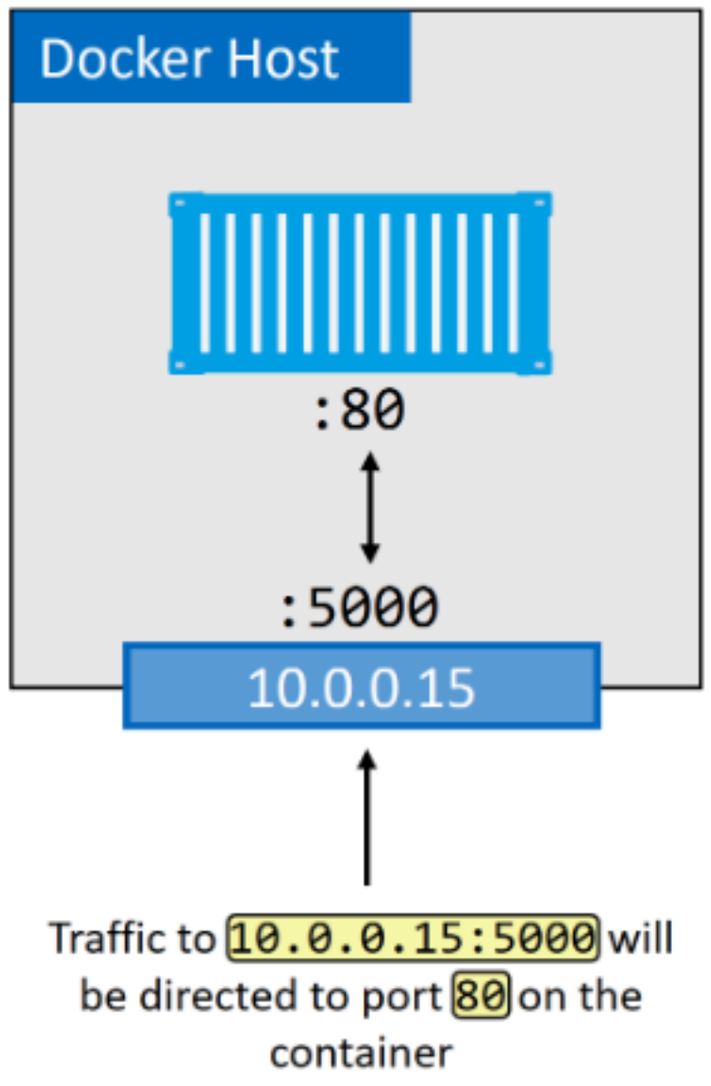
Port Mapping

- $ docker container run -d --name web --network localnet -publish 5000:80 nginx

- $ docker port web

  - 80/tcp -> 0.0.0.0:5000

  - $ docker network create -d overlay

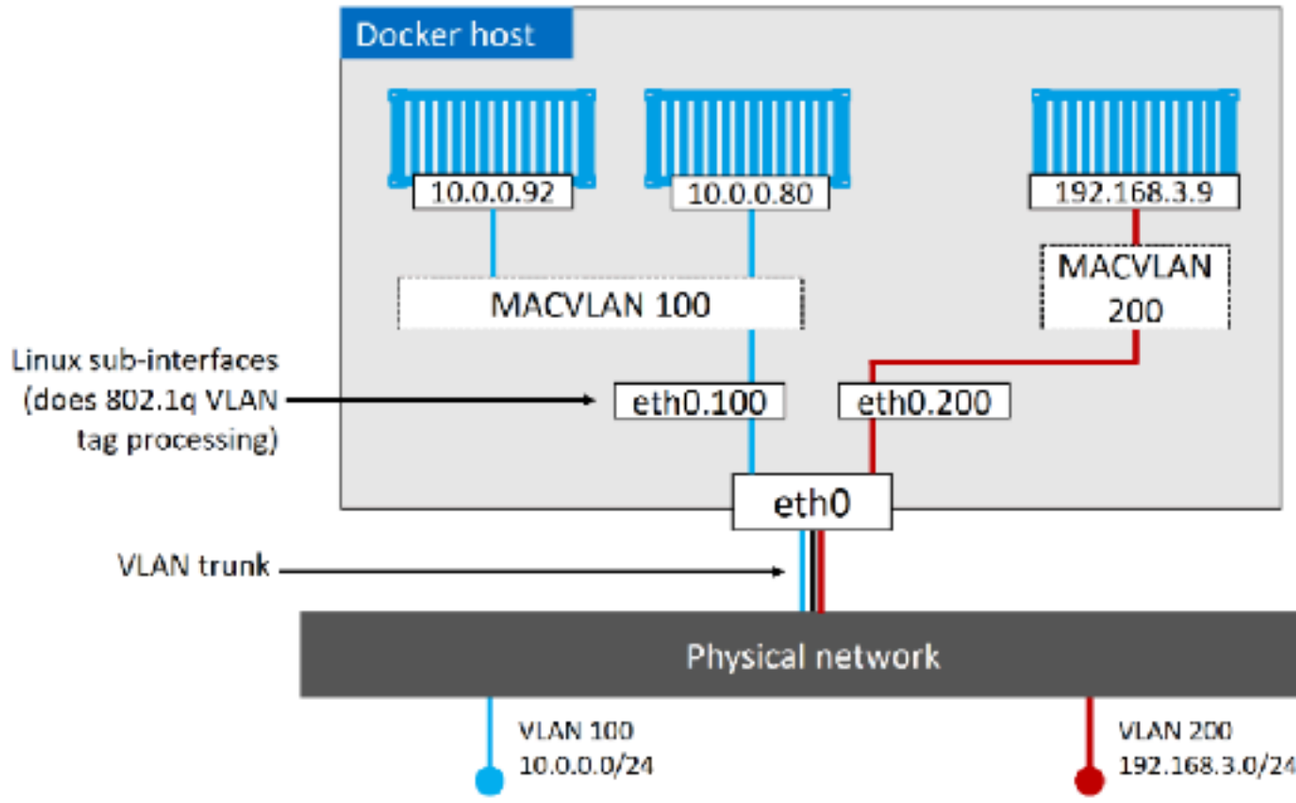


High level flow of port mapping

Mapping ports like this works, but it's clunky and doesn't scale. For example, only a single container can bind to any port on the host. This means no other containers on that host will be able to bind to port 5000. This is one of the reasons that single-host bridge networks are only useful for local development and very small applications.

Connecting to Existing Networks

- $ docker container run -d --name mactainer1 --network macvlan100 alpine sleep 1d



Creating multiple macvlan networks

```
$ docker network create -d macvlan \
  --subnet=10.0.0.0/24 \
  --ip-range=10.0.0.0/25 \
  --gateway=10.0.0.1 \
  -o parent=eth0.100 \
  macvlan100
```

Service Discovery

- Step 1: The ping c2 command invokes the local DNS resolver to resolve the name “c2” to an IP address. All Docker containers have a local DNS resolver.

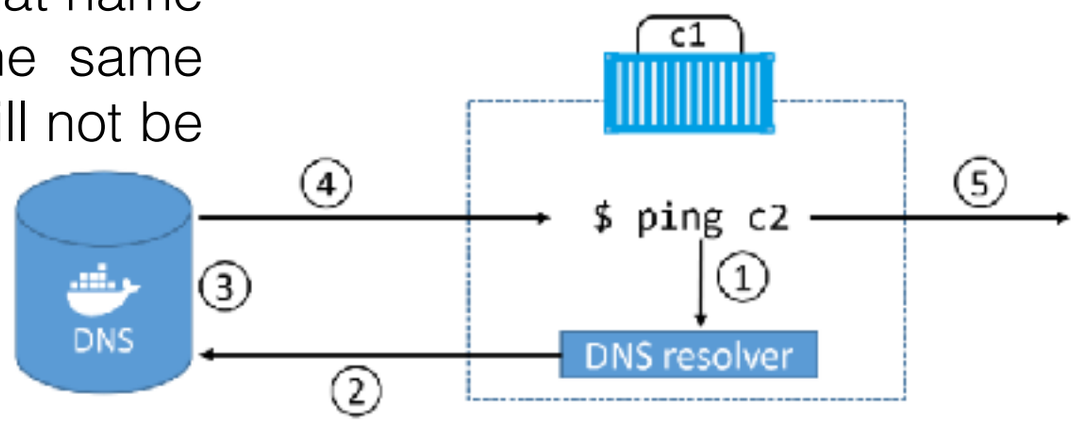
- Step 2: If the local resolver doesn’t have an IP address for “c2” in its local cache, it initiates a recursive query to the Docker DNS server. The local resolver is pre-configured to know how to reach the Docker DNS server.

- Step 3: The Docker DNS server holds name-to-IP mappings for all containers created with the --name or --net-alias flags. This means it knows the IP address of container “c2”.

- Step 4: The DNS server returns the IP address of “c2” to the local resolver in “c1”. It does this because the two containers are on the same network; if they were on different networks this would not work.

- Step 5: The ping command issues the ICMP echo request packets to the IP address of “c2”.

NOTE: service discovery is network-scoped. This means that name resolution only works for containers and Services on the same network. If two containers are on different networks, they will not be able to resolve each other.



c1 pinging c2 by name

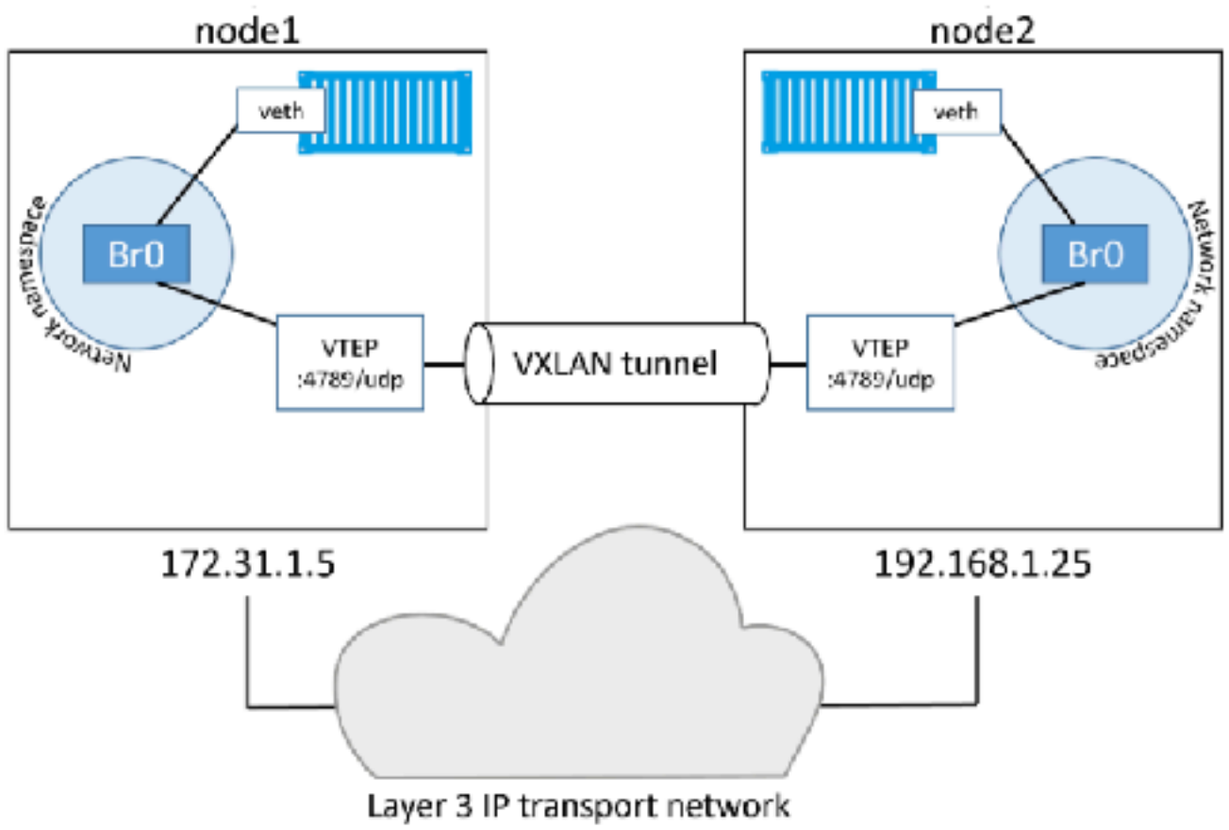
DNS

- It's possible to configure Swarm services and standalone containers with customized DNS options.
- For example, the --dns flag lets you specify a list of custom DNS servers to use in case the embedded Docker DNS server cannot resolve a query.
    - This is common when querying the names of services outside of Docker.
    - You can also use the --dns-search flag to add custom search domains for queries against unqualified names (i.e. when the query is not a fully qualified domain name).
    - On Linux, these all work by adding entries to the /etc/resolv.conf file inside every container.
- The following example will start a new standalone container and add the infamous 8.8.8.8 Google DNS server, as well as nigelpoulton.com as search domain to append to unqualified queries.
    - $ docker container run -it --name c1 --dns=8.8.8.8 --dns-search=nigelpoulton.com alpine sh

Network Commands

- Docker networking has its own docker network sub-command. The main commands include:
    - docker network ls lists all networks on the local Docker host.
    - docker network create creates new Docker networks.
        - By default, it creates them with the nat driver on Windows and the bridge driver on Linux.
        - You can specify the driver (type of network) with the -d flag. $ docker network create -d overlay overnet will create a new overlay network called overnet with the native Docker overlay driver.
    - docker network inspect provides detailed configuration information about a Docker network.
    - docker network prune deletes all unused networks on a Docker host.
    - docker network rm deletes specific networks on a Docker host.

Overlay Network



Topology for two containers to communicate over VXLAN overlay network

$ docker network create --subnet=10.1.1.0/24 --subnet=11.1.1.0/24 -d overlay prod-net
This would result in two virtual switches, Br0 and Br1, being created inside the sandbox, and routing happens by default.

Docker Compose

https://docs.docker.com/compose/reference/
https://github.com/compose-spec/compose-spec

This is aimed at creating an open standard for defining multi-container **cloud-native** apps. The ultimate aim is to greatly simplify the **code-to-cloud** process.

History

- In the beginning, there was **Fig**. It was a powerful tool, created by a company called Orchard, and it was the best way to manage multi-container Docker apps. It was a Python tool that sat on top of Docker and let you define entire multi-container apps in a single YAML file. You could then deploy and manage the lifecycle of the app with the fig command-line tool. Behind the scenes, Fig would read the YAML file and use Docker to deploy and manage the app via the Docker API.

Docker Compose

- Docker-compose is a tool that combines and runs multiple containers of interrelated services with a single command. It is a tool to define all the application dependencies in one place and let the Docker take care of running all of those services in just one simple command $docker-compose up

Docker Compose .yml file

```
version: '3'
services:
  web:
    # Path to dockerfile.
    # '.' represents the current directory in which
    # docker-compose.yml is present.
    build: .
     context: ./db
     dockerfile: Dockerfile-db
    # Mapping of container port to host

    ports:
      - "5000:5000"
    # Mount volume
    volumes:
      - "/usercode/:/code"

    # Link database container to app container
    # for reachability.
    links:
      - "database:backenddb"
    depends_on:
      - database

database:

    # image to fetch from docker hub
    image: mysql/mysql-server:5.7

    # Environment variables for startup script
    # container will use these variables
    # to start the container with these defined variables.
     env_file:
      - ./.env
    environment:
      - "MYSQL_ROOT_PASSWORD=root"
      - "MYSQL_USER=testuser"
      - "MYSQL_PASSWORD=admin123"
      - "MYSQL_DATABASE=backend"
    # Mount init.sql file to automatically run
    # and create tables for us.
    # everything in docker-entrypoint-initdb.d folder
    # is executed as soon as container is up nd running.
    volumes:
      - "/usercode/db/init.sql:/docker-entrypoint-initdb.d/init.sql"
```

.yml file sections

version '3': Like other software, docker-compose also started with version 1.0. At the time of writing this course, the current latest version of Compose file is 3.7. We have specified the version of Compose file we will be using and Docker will provide the features accordingly.
Compose versions are backward compatible, hence it is recommended to use the latest version.

services: The services section defines all the docker images required and need to be built for the application to work. In short, it's the collection of all different components of the application that are dependent on each other.
We have two services namely, web and database. In Compose version 3, we can have multiple containers of the same service as well.
We will see that in the next section, but if you are curious, you can check here under the deploy section in the compose file.

web: The name web is the name of our Flask app service. It can be anything. Docker Compose will create containers with this name.

build: This clause specifies the Dockerfile location. '.' represents the current directory where the docker-compose.yml file is located and Dockerfile is used to build an image and run the container from it. We can also provide the absolute path to Dockerfile instead of the current working directory symbol.
By default, docker-compose looks for a file named Dockerfile. Dockerfiles can have any name. It's just a file without any extension. We can override the default behaviour using dockerfile:'custom-name' directive inside the build section.

ports: The ports clause is used to map the container ports to the host machine's port. It creates a tunnel from the specified container port to the provided host machine's port.
This is the same as using the -p 5000:5000 option to map the container's 5000 port to the host machine's 5000 port while running the container using the docker run command.

volumes: This is the same as the -v option used to mount disks in docker run command. Here, we are attaching our code files directory to the container's /code directory so that we don't have to rebuild the images for every change in the files.
This will also help in auto-reloading the server when running in debug mode.

links: Links literally link one service to another. In the bridge network, we have to specify which container should be accessible to what container using a link to the respective containers.
Here, we are linking the database container to the web container, so that our web container can reach the database in the bridge network.

depends_on: This is used to inform docker-compose about all the dependencies of a service. Docker-compose will then start dependencies first and the main service after.

image: If we don't have a Dockerfile and want to run a service directly using an already built docker image, then specify the image location using the 'image' clause. Compose will pull the image and fork a container from it.

environment: Any environment variable that should be present in the container can be created using the environment clause. This does the same work as the -e argument in the docker run command while running a container.

Docker compose commands

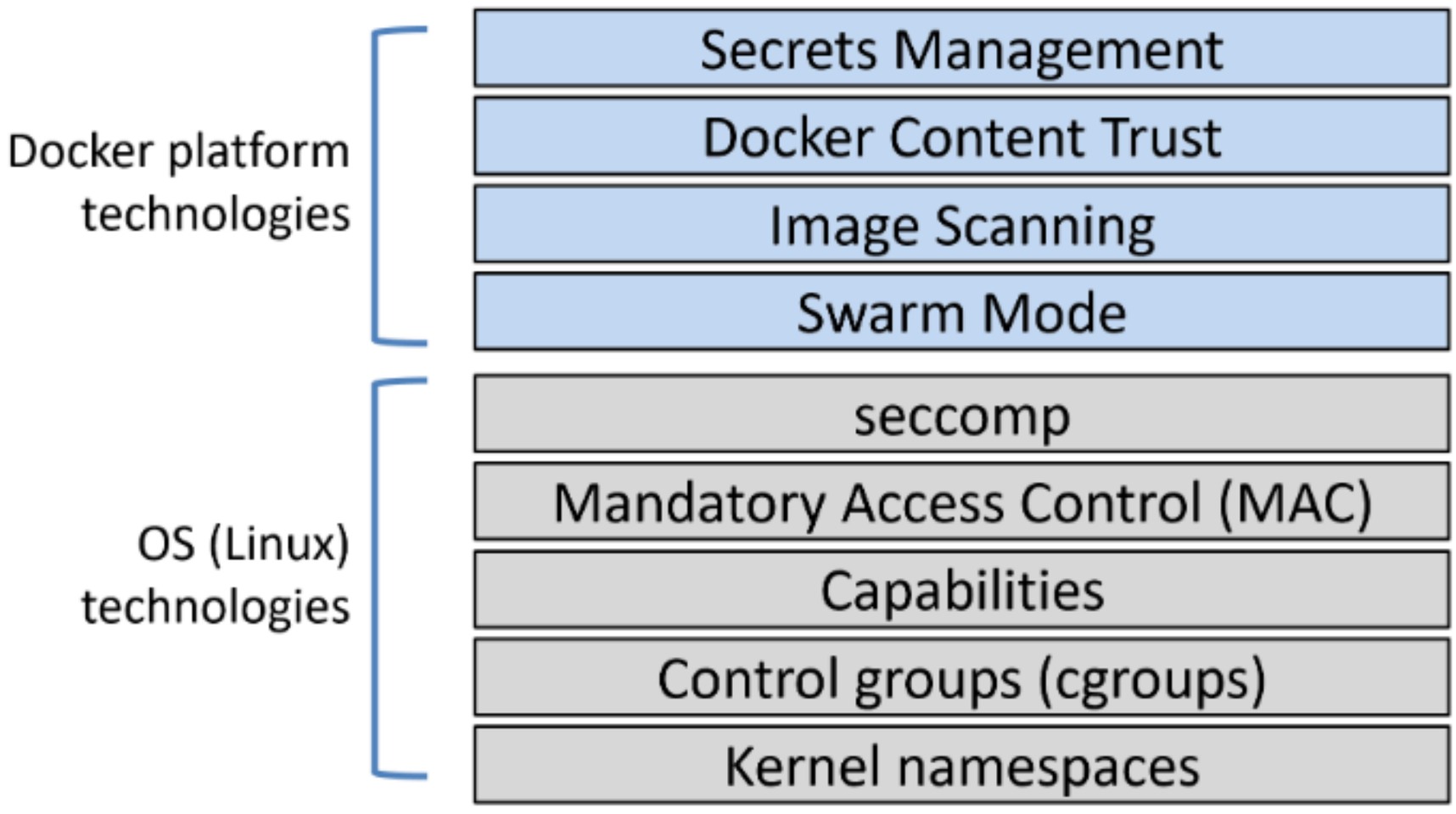
- https://docs.docker.com/compose/reference/
- $ docker-compose —help
- $ docker-compose up/ps/top/stop/restart/down/rm
- $ docker-compose build
    - This command builds images of the mentioned services in the docker-compose.yml file for which a Dockerfile is provided.
- $docker-compose images
    - This command lists images built using the current docker-compose file.
- $docker-compose run
    - Creates containers from images built for the services mentioned in the compose file. It runs a specific service provided as an argument to the command. it will start all the dependent services and then, the mentioned service. **It will start all the dependent services and then, the mentioned service.**
- $docker-compose up
    - This does the job of the docker-compose build and docker-compose run commands. It initially builds the images if they are not located locally and then starts the containers. If images are already built, it will fork the container directly. We can force it to rebuild the image by adding a --build argument.
    - $ docker-compose up &

Environment Variable

- Priority of .env variables
- There are multiple ways we can use the environment variables in Docker, which includes,
    - ENV in Dockerfile
    - environment keyword in the docker-compose.yml file
    - -e option from the command line
- When you set the same environment variable in multiple files, here's the priority used by Compose to choose which value to use:
    - Compose file
    - Shell environment variables
    - Environment file
    - Dockerfile
    - Variable is not defined
- So, if we define a variable in the Compose file in the - environment section and also in a .env file, Compose will consider the variable declared in the Compose file from -environment section.
- Sometimes, this might be the solution to any unexpected behavior of the application when multiple environment variables are used.

Security

Security Technologies in Docker



Security-related technologies

Security in Docker

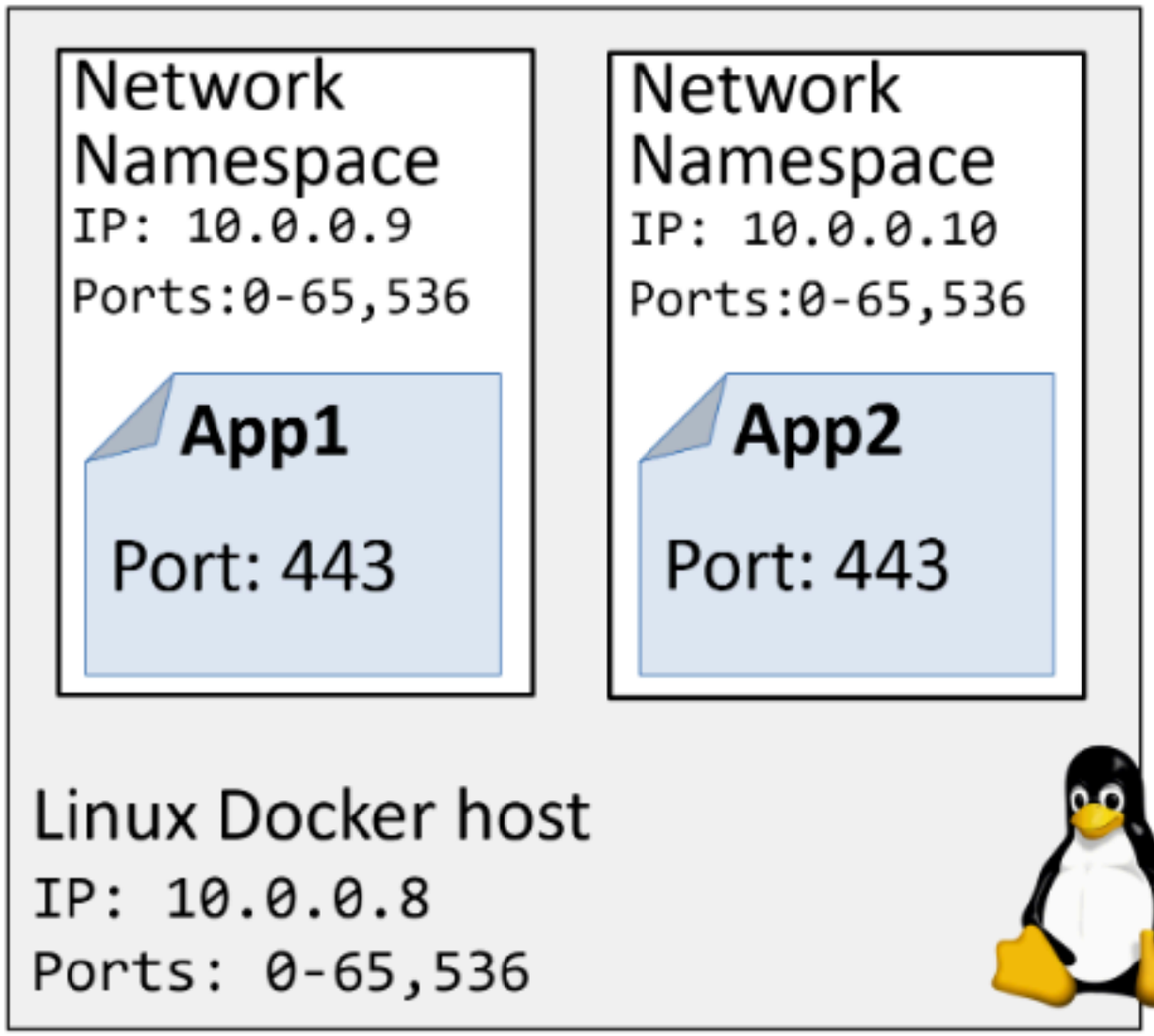
- Docker Swarm Mode is secure by default. You get all of the following with zero configuration required: cryptographic node IDs, mutual authentication, automatic CA configuration, automatic certificate rotation, encrypted cluster store, encrypted networks, and more.

- Docker Content Trust (DCT) lets you sign your images and verify the integrity and publisher of images you consume.

- Image security scanning analyses images, detects known vulnerabilities, and provides detailed reports.

- Docker secrets are a way to securely share sensitive data and are first-class objects in Docker. They're stored in the encrypted cluster store, encrypted in-flight when delivered to containers, stored in in-memory file systems when in use, and operate a least-privilege model.

Namespace



Two web server applications running on a single host

Docker’s usage of namespace

- namespaces is used for isolation
- **Process ID namespace:** Docker uses the pid namespace to provide isolated process trees for each container. This means every container gets its own PID 1. PID namespaces also mean that one container cannot see or access the process tree of other containers. Nor can it see or access the process tree of the host it’s running on.
- **Network namespace:** Docker uses the net namespace to provide each container its own isolated network stack. This stack includes; interfaces, IP addresses, port ranges, and routing tables. For example, every container gets its own eth0 interface with its own unique IP and range of ports.
- **Mount namespace:** Every container gets its own unique isolated root (/) filesystem. This means every container can have its own /etc, /var, /dev, and other important file system constructs. Processes inside of a container cannot access the mount namespace of the Linux host or other containers — they can only see and access their own isolated filesystem.
- **Inter-process Communication namespace:** Docker uses the ipc namespace for shared memory access within a container. It also isolates the container from shared memory outside of the container.
- **User namespace:** Docker lets you use user namespaces to map users inside of a container to different users on the Linux host. A common example is mapping a container’s root user to a non-root user on the Linux host.
- **UTS namespace:** Docker uses the uts namespace to provide each container with its own hostname.

Control Groups (Cgroups)

Cgroups are about setting limits!

Capabilities

- CAP_CHOWN: Lets you change file ownership
- CAP_NET_BIND_SERVICE: Lets you bind a socket to low numbered network ports
- CAP_SETUID: Lets you elevate the privilege level of a process
- CAP_SYS_BOOT: Lets you reboot the system.

Least Privilege policy, add the capabilities as needed

Risks

- Host machine access
    - Since containers use the host's kernel as a shared kernel for running processes, a compromised container kernel can exploit or attack the entire host system.
- Container breakouts
    - If somehow, a user is able to escape the container namespace, it will be able to interact with the other processes on the host and can stop or kill the processes.
- Max resource utilization
    - Sometimes a container uses all the resources of the host machine if it is not restricted. This will force other services to halt and stop the execution.
- Attack using untrusted images
    - Docker allows us to run all images present on Docker Hub as well as a local build. So, when an image from an untrusted source is run on the machine, the attacker's malicious program may get access to the kernel or steal all the data present in the container and mounted volumes.

Security Best Practices

- Container lifecycle management
    - Through the container lifecycle management process, we establish a strong foundation for the review process of creating, updating, and deleting a container. This takes care of all the security measures at the start while creating a container. When a container is updated instead of reviewing only the updated layer, we should review all layers again.
- Information management
    - Never push any sensitive data like passwords, ssh keys, tokens, certificates in an image. It should be encrypted and kept inside a secret manager. Access to these secrets should be explicitly provided to the services and only when they are running.
- No root access
    - A container should never be run with root-level access. A role-based access control system will reduce the possibility of accidental access to other processes running in the same namespace. Many of the organizations restrict access using the active directory and provide appropriate access based on user roles. In general, we can use Linux's inbuilt commands to create a temporary non-root user on the fly.
- Trusted image source
    - Check the authenticity of every image pulled from Docker Hub. Use Docker Content Trust to check the authenticity of the Image. Docker Content Trust is a new feature incorporated into Docker 1.8. It is disabled by default, but once enabled, allows you to verify the integrity, authenticity, and publication date of all Docker images from the Docker Hub registry. Enable Docker content trust using export DOCKER_CONTENT_TRUST=1
- Limit resource usage
    - Assign a predefined resource usage policy for a container. This is supported only in swarm mode.
- Cautionary use of Docker socket
    - In the visualizer service, we have mounted Docker socket /var/run/docker.sock on the container. Bind mounting the Docker daemon socket gives a lot of power to a container as it can control the daemon. It must be used with caution and only with containers we can trust. There are a lot of third-party tools that demand this socket to be mounted while using their service. You should verify such services with Docker Content Trust and vulnerability management processes before using them.
- We mounted the Docker socket on the visualizer service container as the visualizer image is Docker's official image and it can be verified by Docker Content Trust and Docker bench for Security.
- Mounting volumes as read only
- One of the best practices is to mount the host filesystem as read-only if there is no data being saved by the container. We can do that by simply using a ro flag for mounts using -v argument or readonly with the --mount argument.

Loging

Daemon Logs

- On **Windows** systems, the daemon logs are stored under ~AppData\Local\Docker, and you can view them in the Windows Event Viewer.
- On **Linux**, it depends what init system you're using.
  - If you're running a systemd, the logs will go to journald and you can view them with the journalctl -u docker.service command.
  - If you're not running systemd you should look under the following locations:
    - Ubuntu systems running upstart: /var/log/upstart/docker.log
    - RHEL-based systems: /var/log/messages
    - Debian: /var/log/daemon.log

Managing the log outputs

- You can also tell Docker how verbose you want daemon logging to be.
  - Edit the daemon config file (daemon.json) so that **debug** is set to **true** and log-level is set to one of the following:
    - debug: The most verbose option
    - info: The default value and second-most verbose option
    - warn: Third most verbose option
    - error: Fourth most verbose option
    - fatal: Least verbose option

```
{
  <Snip>
  "debug":true,
  "log-level":"debug",
  <Snip>
}
```

Container logs

- Logs from standalone containers can be viewed with
    - $ docker container logs
- Swarm service logs can be viewed with
    - $ docker service logs
- However, Docker supports lots of logging drivers, and they don't all work with the docker logs command.

Logging drivers

- Docker host has a default logging driver and configuration for containers. Some of the drivers include:
    - json-file (default)
    - journald (only works on Linux hosts running systemd)
    - syslog
    - splunk
    - gelf
- json-file and journald are probably the easiest to configure, and they both work with the docker logs and docker service logs commands. The format of the commands is docker logs <container-name> and docker service logs <service-name>.
- If you're using other logging drivers you can view logs using the third-party platform's native tools.
- The following snippet from a daemon.json shows a Docker host configured to use syslog.

```
{
  "log-driver": "syslog"
}
```

Logging (cont.)

- You can configure an individual container, or service, to start with a particular logging driver with the --log-driver and --log-opts flags. These will override anything set in daemon.json.

- Container logs work on the premise that your application is running as PID 1 inside the container and sending logs to STDOUT, and errors to STDERR. The logging driver then forwards these “logs” to the locations configured via the logging driver.

- If your application logs to a file, it’s possible to use a symlink to redirect log-file writes to STDOUT and STDERR.

- The following is an example of running the docker logs command against a container called “vantage-db” configured to use the json-file logging driver.

    - $ docker logs vantage-db

Orchestration

Orchestration

- Managing this cluster of containers effectively is called container orchestration.
- Three major tools for container orchestration
    - Docker swarm, Kubernetes, and Apache Mesos.