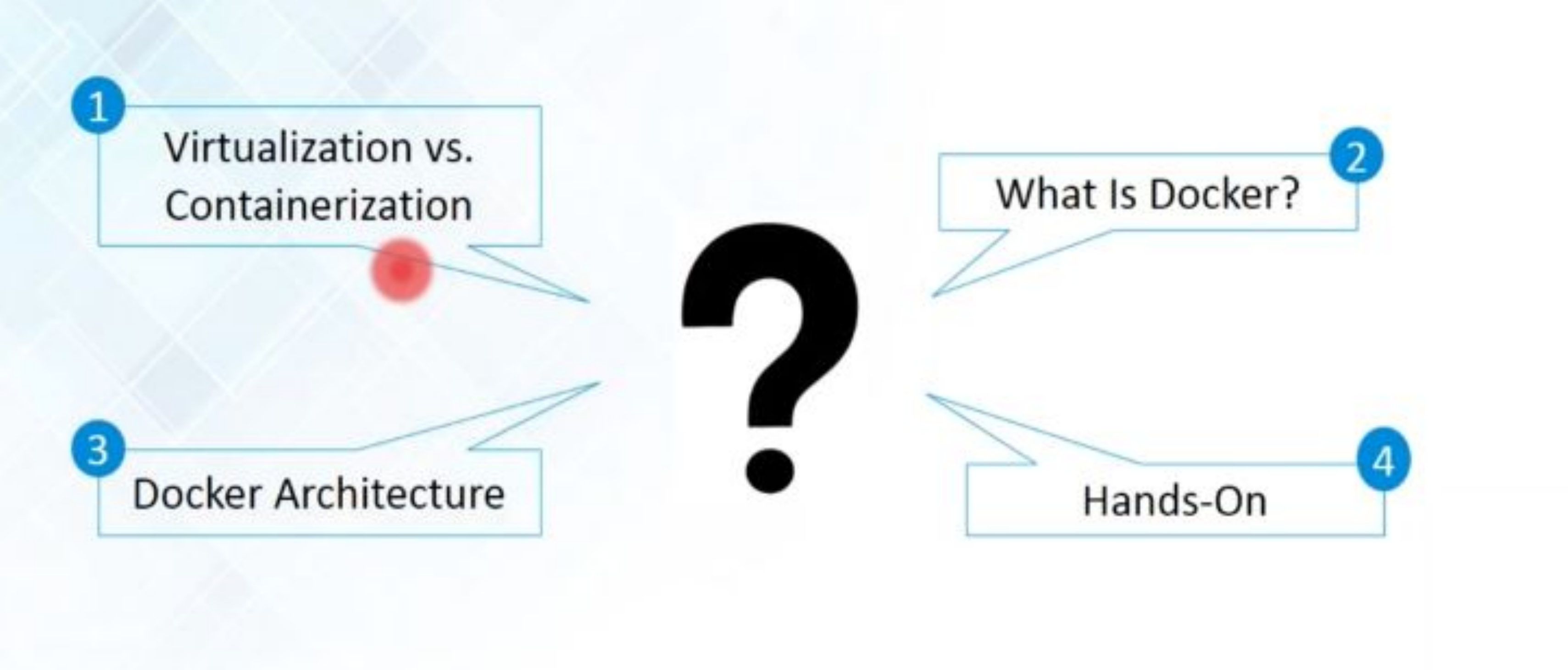
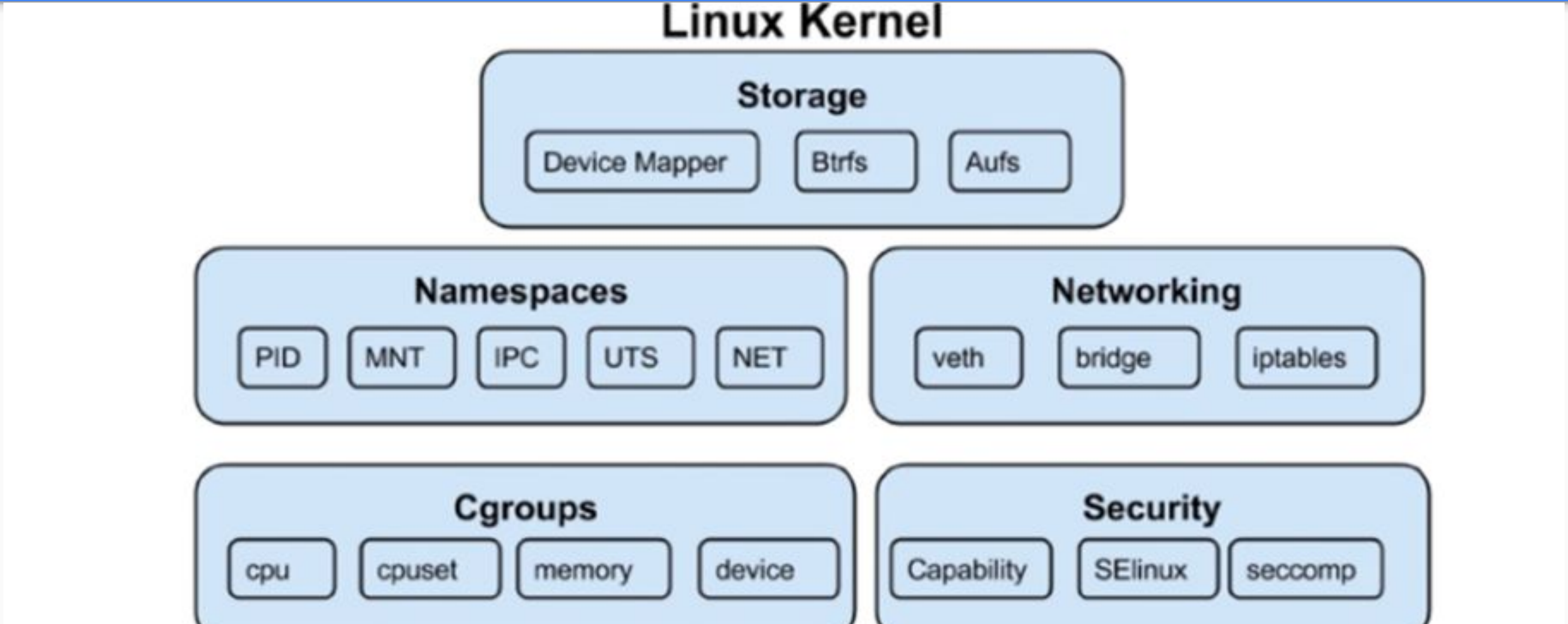
Docker



1
Virtualization vs. Containerization
2
What Is Docker?
3
Docker Architecture
4
Hands-On
?

Docker Internals



Docker Internals

- Container are method of operating system virtualization that allow you to run application and its dependencies in resource isolated process.
- Containers allows you to easily package an application code, configurations and dependencies into easy to use building blocks and that application can be deployed quickly and consistently regardless of deployment environment

Docker Internals

Linux kernel features that create the walls between container and other processes running on the host.

To understand Containers We have to start with **Linux Cgroup and Namespace & Union File System.**

Linux Namespace - Wrap a set of system resources and present them to a process to make it look like they are dedicated to that process.

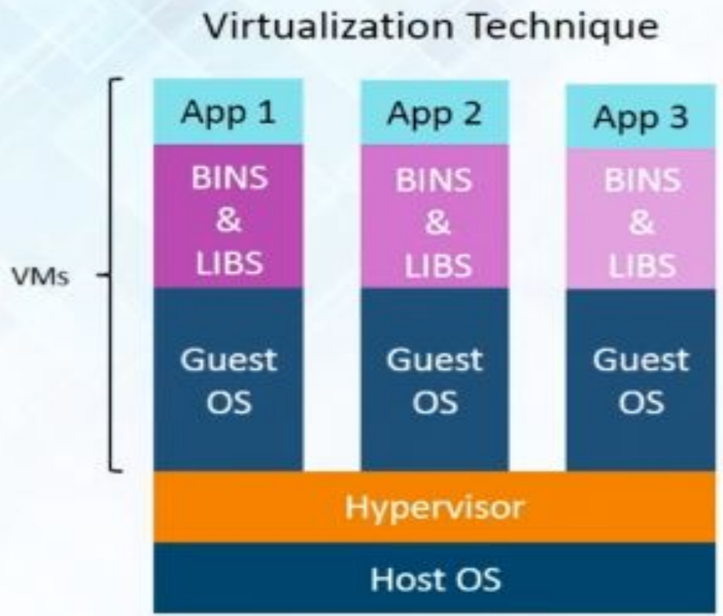
Docker Internals

Linux Cgroup - Governs the isolation and usage of system resources such as cpu and memory for group of process.E.g. - If you have a application that takes up lot of cpu cycles and memory such as scientific computing application you can put the application in a cgroup to limit a CPU and memory usage.

Docker Internals

- **Namespaces** deal with resource isolation for single process
- **Cgroup** manages resources for group of processes

Virtualization



Advantages

- Multiple OS In Same Machine
- Easy Maintenance & Recovery
- Lower Total Cost Of Ownership

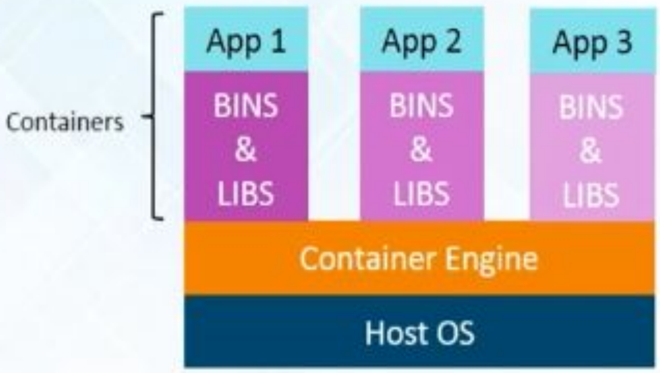
Disadvantages

- Multiple VMs Lead To Unstable Performance
- Hypervisors Are Not As Efficient As Host OS
- Long Boot-Up Process (Approx. 1 Minute)

Containerization

Note: Containerization Is Just Virtualization At The OS Level

Containerization Technique

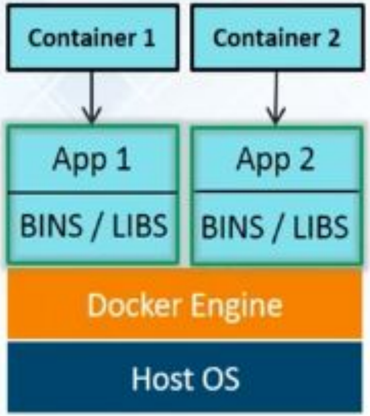


Advantages Over Virtualization

- Containers On Same OS Kernel Are Lighter & Smaller
- Better Resource Utilization Compared To VMs
- Short Boot-Up Process ($1/20^{th}$ of a second)

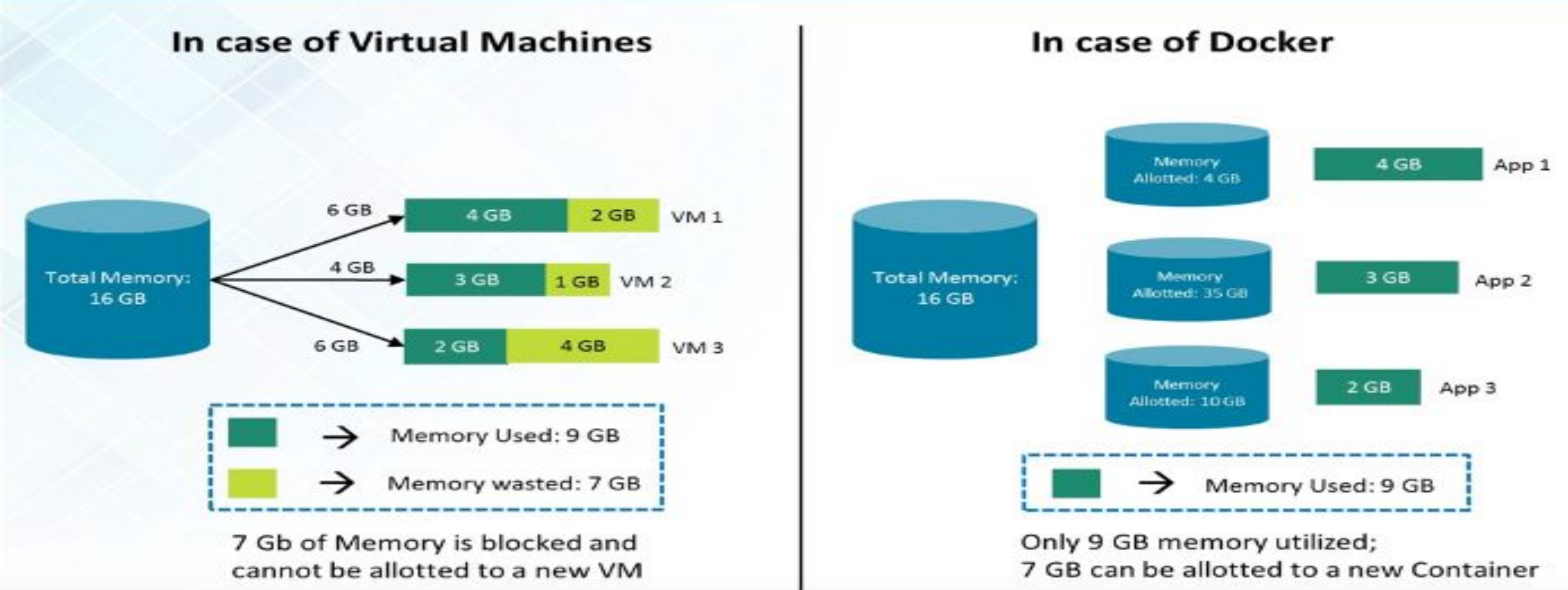
Docker Info

Docker is a Containerization platform which packages your application and all its dependencies together in the form of Containers so as to ensure that your application works seamlessly in any environment be it Development or Test or Production.

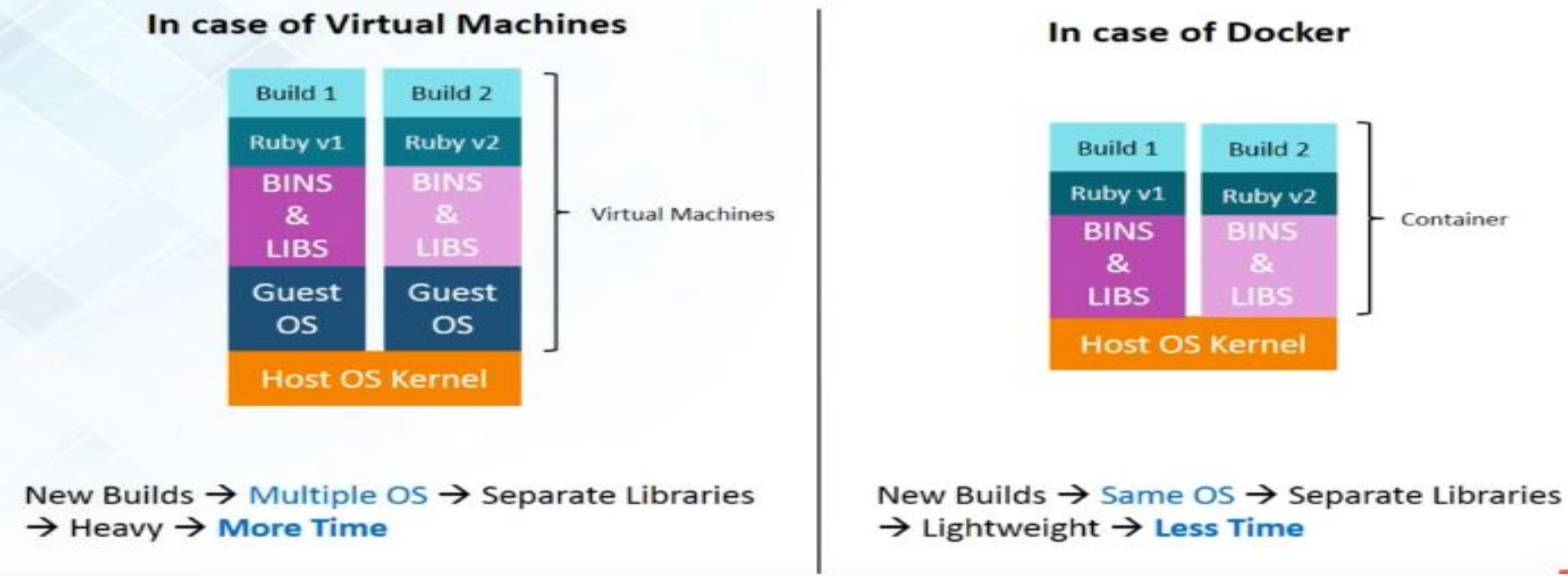


Docker Info

Docker Info


In case of Virtual Machines
Total Memory: 16 GB
6 GB
4 GB
6 GB
4 GB
2 GB
VM 1
3 GB
1 GB
VM 2
2 GB
4 GB
VM 3
Memory Used: 9 GB
Memory wasted: 7 GB
7 Gb of Memory is blocked and cannot be allotted to a new VM
In case of Docker
Total Memory: 16 GB
Memory Allotted: 4 GB
4 GB
App 1
Memory Allotted: 35 GB
3 GB
App 2
Memory Allotted: 10 GB
2 GB
App 3
Memory Used: 9 GB
Only 9 GB memory utilized;
7 GB can be allotted to a new Container

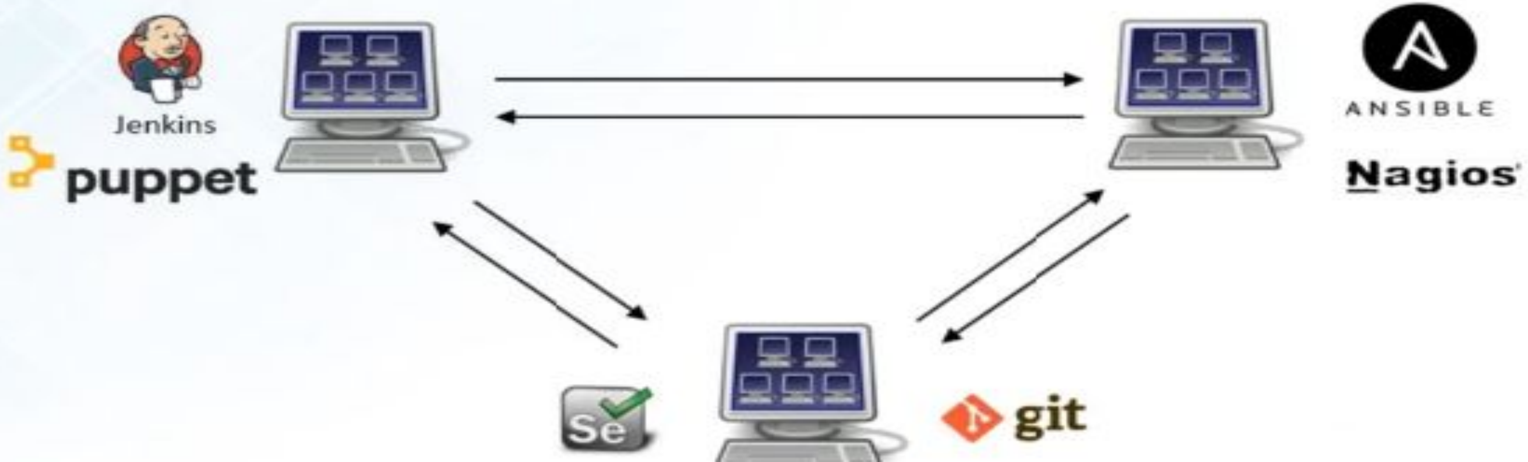
Docker Info


In case of Virtual Machines
Build 1
Build 2
Ruby v1
Ruby v2
BINS & LIBS
BINS & LIBS
Guest OS
Guest OS
Host OS Kernel
Virtual Machines
New Builds → Multiple OS → Separate Libraries → Heavy → More Time
In case of Docker
Build 1
Build 2
Ruby v1
Ruby v2
BINS & LIBS
BINS & LIBS
Host OS Kernel
Container
New Builds → Same OS → Separate Libraries → Lightweight → Less Time

Integration in VMs

Integration In Virtual Machines Is Possible, But:

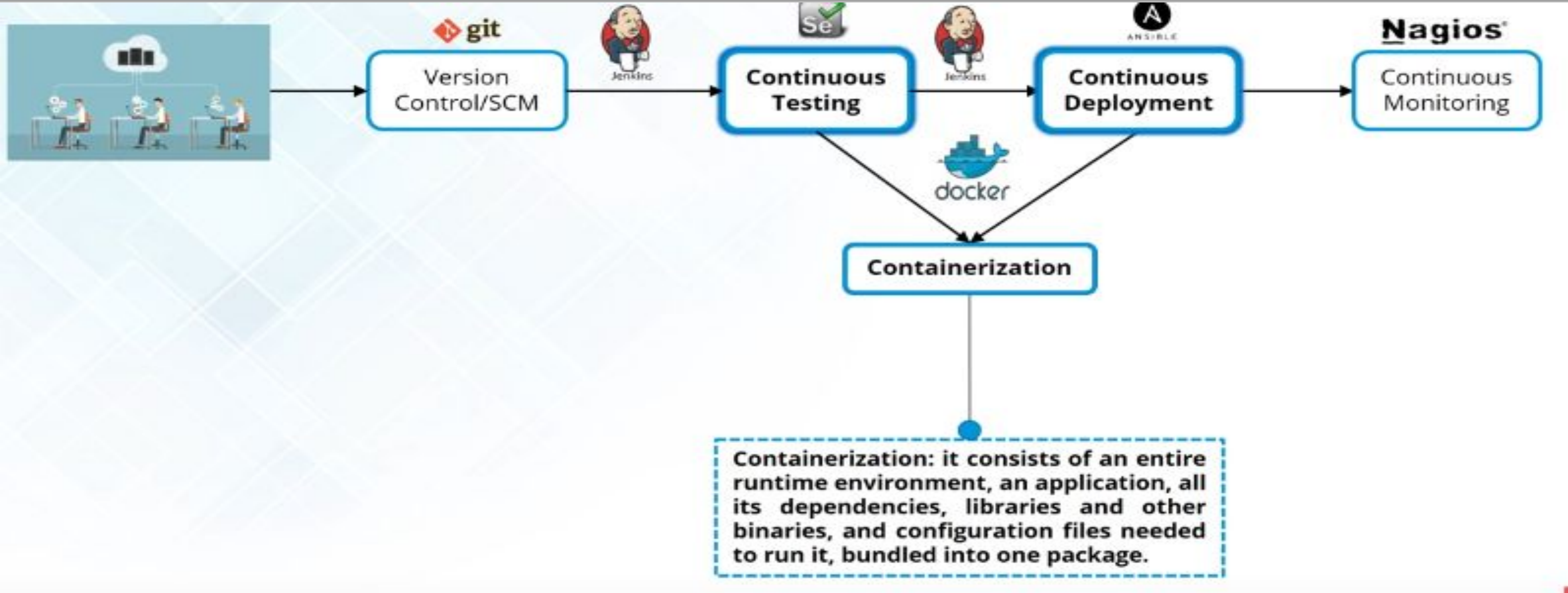
- Costly Due To Infrastructure Requirements
- Not Easily Scalable



Who Can Use Docker

Docker is designed to benefit both Developers and System Administrators, making it a part of many DevOps toolchains.

How is Docker Used In Devops



Docker Images & Containers

Docker Images

Docker Containers

- Read Only Template Used To Create Containers
- Built By Docker Users
- Stored In Docker Hub Or Your Local Registry

- Isolated Application Platform
- Contains Everything Needed To Run The Application
- Built From One Or More Images

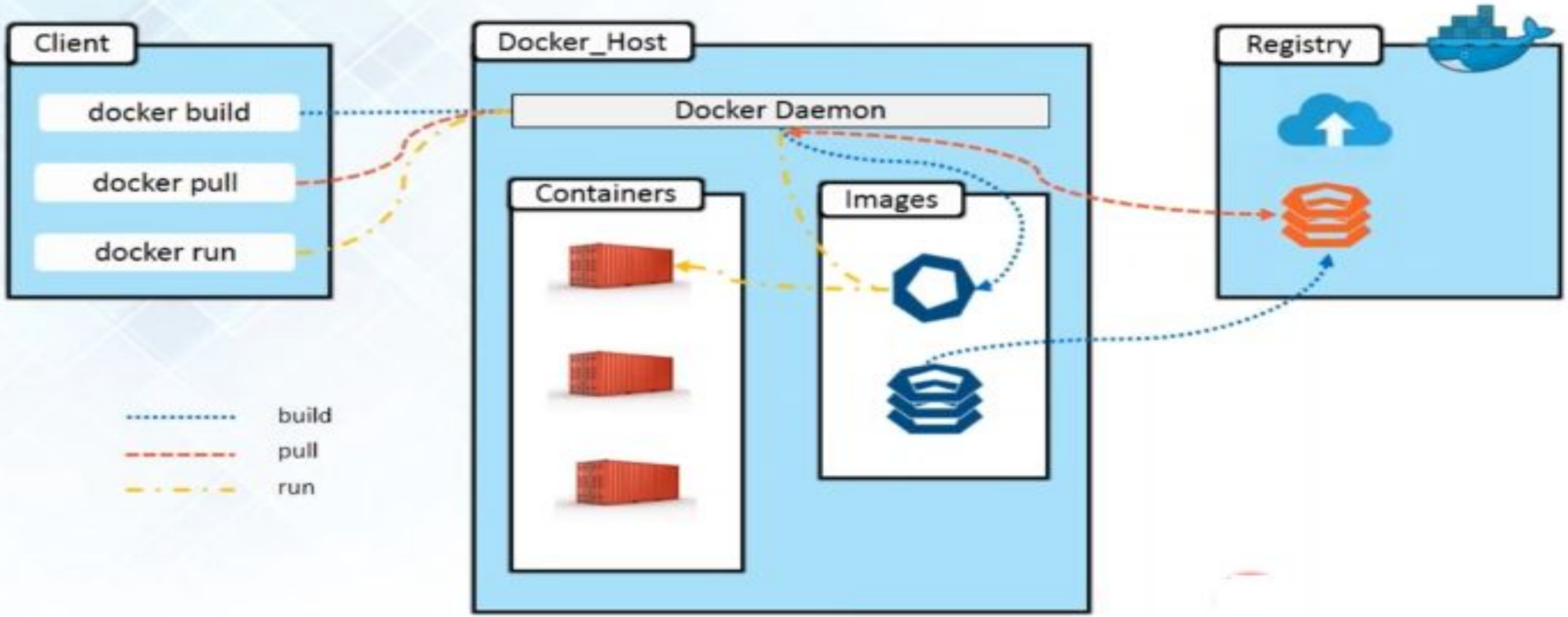
Docker Registry

- Docker Registry is a storage component for Docker Images
- We can store the Images in either Public / Private repositories
- Docker Hub is Docker's very own cloud repository

Why Use Docker Registries?

- Control where your images are being stored
- Integrate image storage with your in-house development workflow

Docker Architecture



Basic Docker Commands

To Pull a Docker image from the Docker hub, we can use the command:

$ docker pull <image-name:tag>

To Run that image, we can use the command:

$ docker run <image-name:tag> or **$ docker run <image-id>**

To list down all the images in our system, we can give the command:

$ docker images

To list down all the running containers, we can use the command:

$ docker ps

To list down all the containers (even if they are not running), we can use the command:

$ docker ps -a

Building Images

- Images are comprised of multiple layers
- Each layer in an Image is an Image of its own
- They comprise of a Base Image layer which is read-only
- Any changes made to an Image are saved as layers on top of the Base Image layer
- Containers are generated by running the Image layers which are stacked one above the other

Single Image Layer

Multiple Image Layers

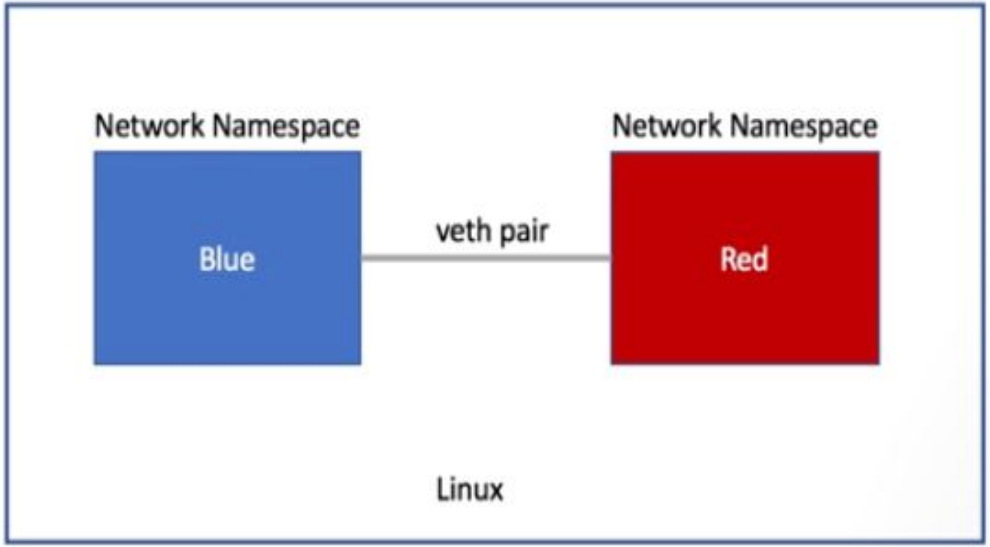
Docker File Instructions

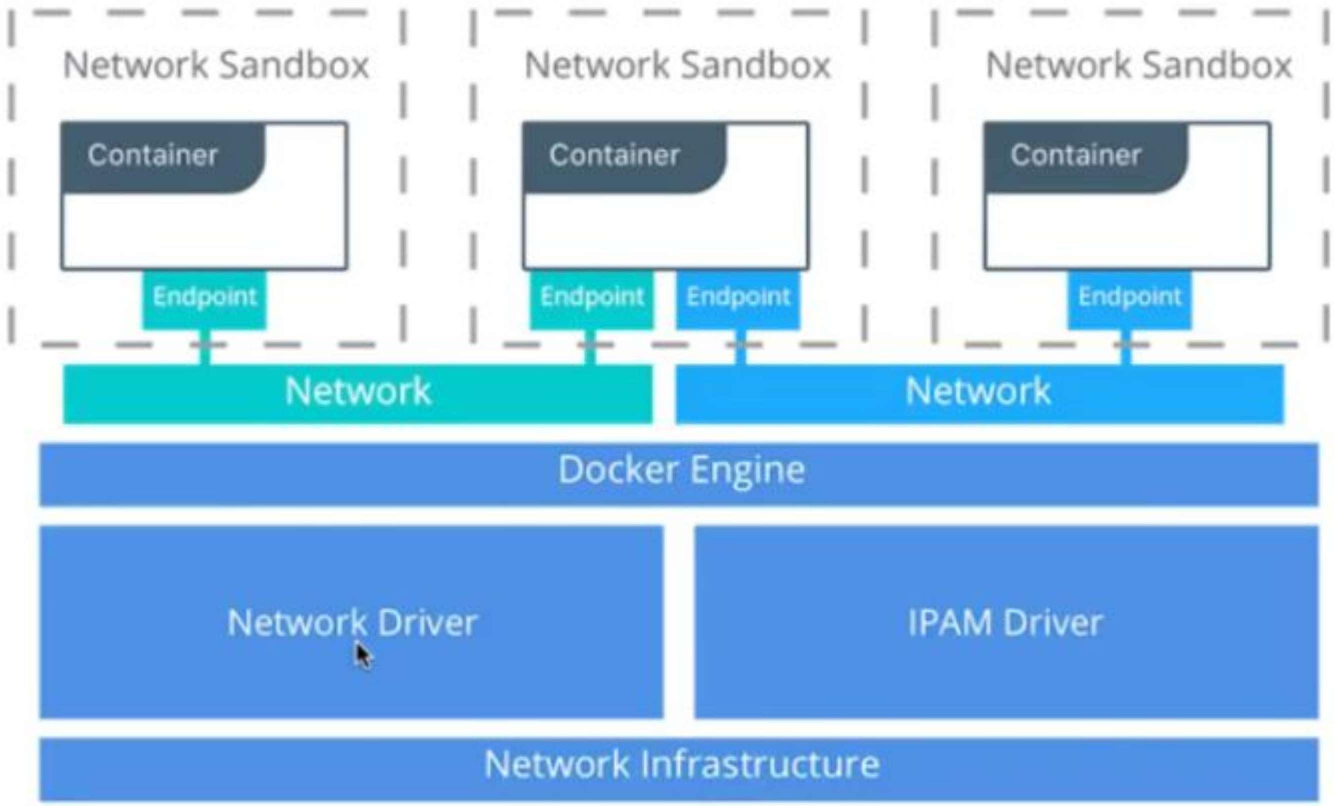
https://docs.docker.com/engine/reference/builder/#environment-replacement

Instruction	Comments
FROM <image>:<tag>	Tells Docker what will be the base for the image being created
ADD	Copies the files from the source on the host into the Docker image's own filesystem at the desired destination. Command is not only about copying files from the local filesystem--you can use it to get the file from the network.
COPY	It will copy new files or directories from <source path> and adds them to the filesystem of the container at the path <destination path>.
CMD / ENTRYPOINT	CMD ["executable","parameter1","parameter2"] his is the so-called exec form CMD command parameter1 parameter2 This a shell form of the instruction
RUN	Instruction will execute any commands in a new layer on top of the current image and then commit the results
EXPOSE	Instruction informs Docker that the container listens on the specified network ports at runtime.

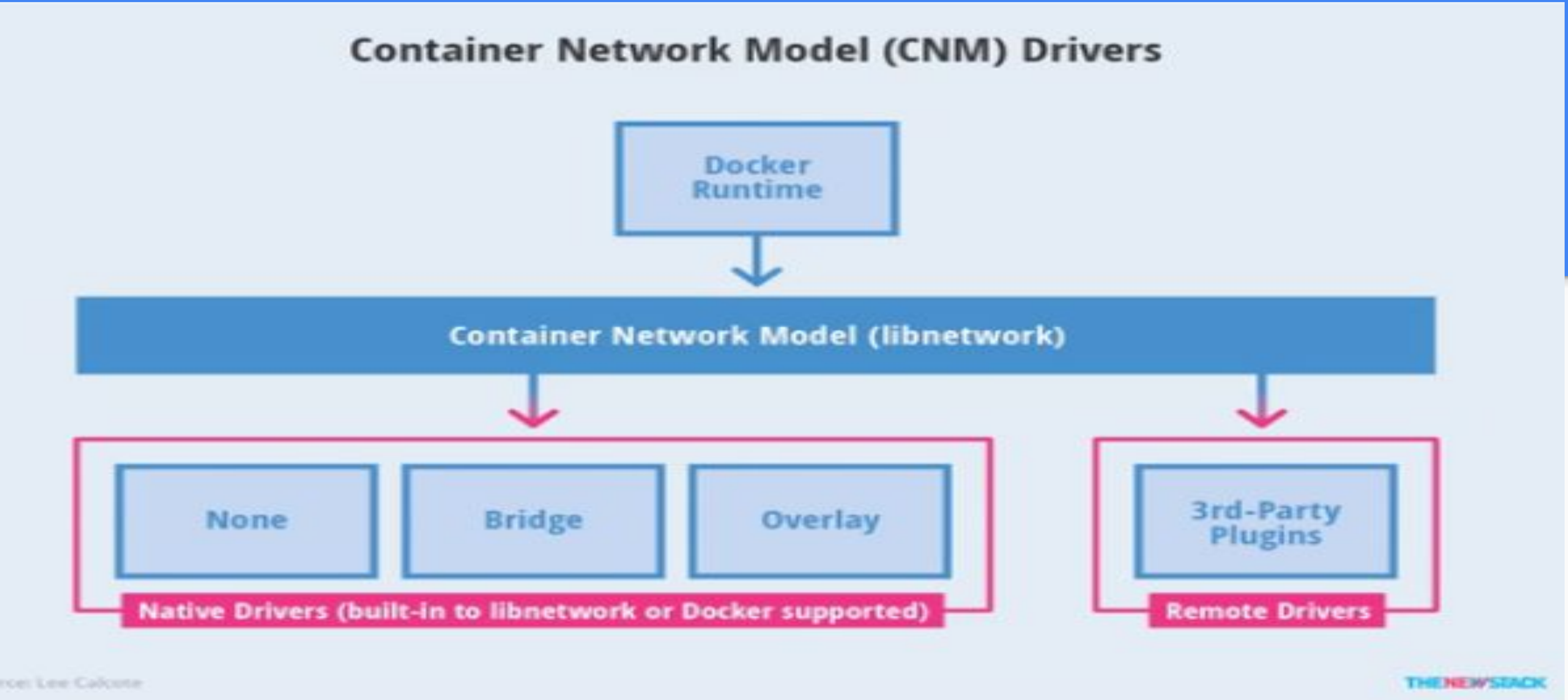
Docker Networking: Linux Network Namespace

- → Network interface
- → Routing table
- → Firewall rules

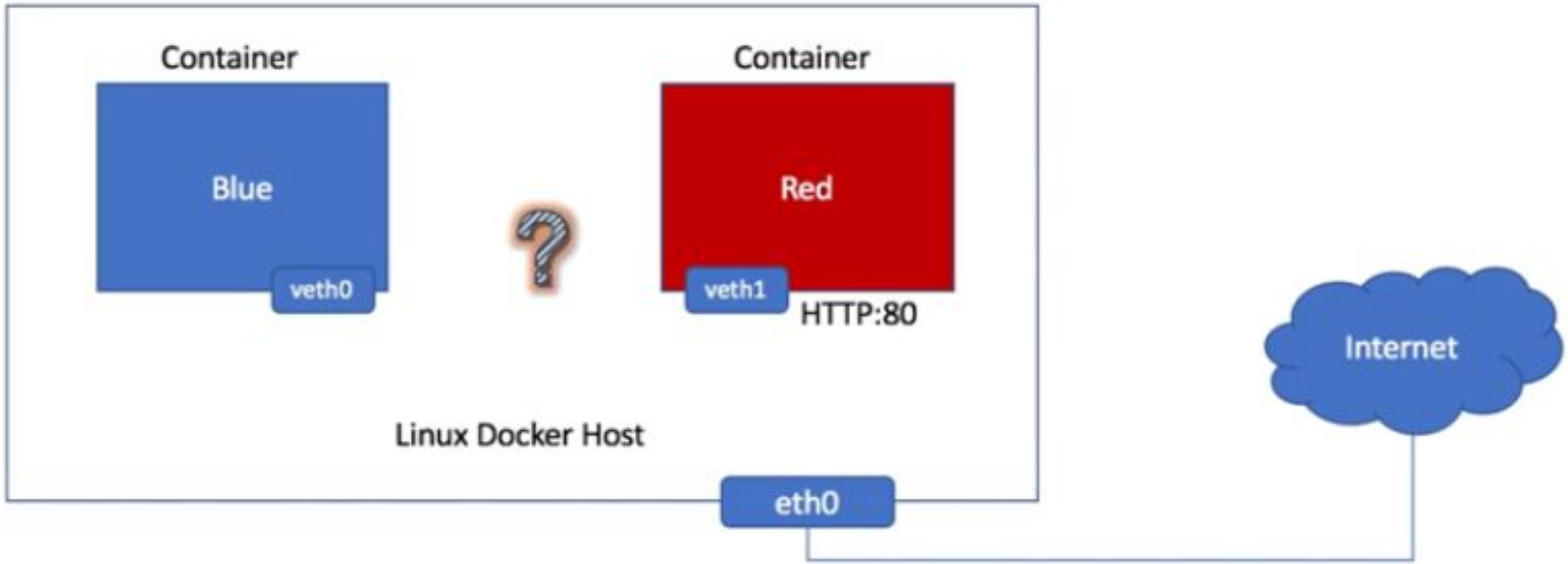




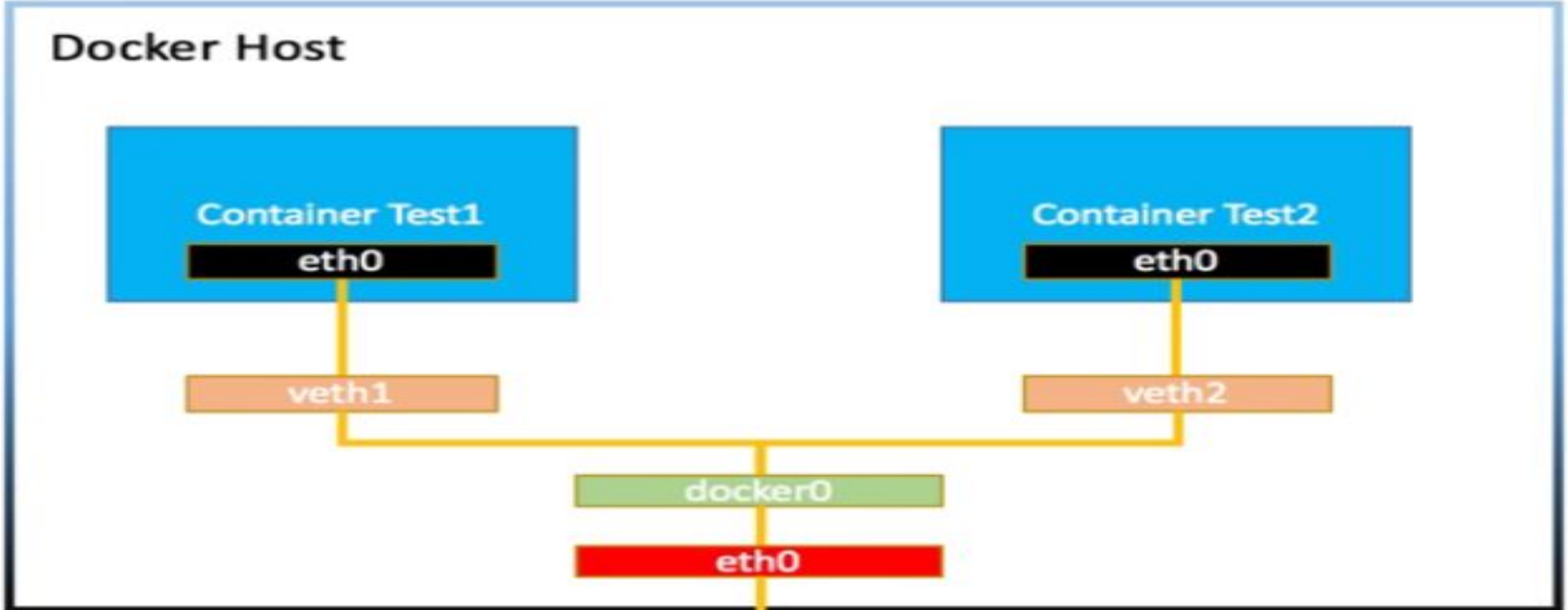
https://github.com/docker/libnetwork/blob/master/docs/design.md


Container Network Model (CNM) Drivers
Docker Runtime
Container Network Model (libnetwork)
None
Bridge
Overlay
Native Drivers (built-in to libnetwork or Docker supported)
3rd-Party Plugins
Remote Drivers

- → How two different containers in the same host communicate with each other?
- → How the container communicate with the outside of Linux host (Internet)?
- → How access container from the local docker host and outside of the docker host?

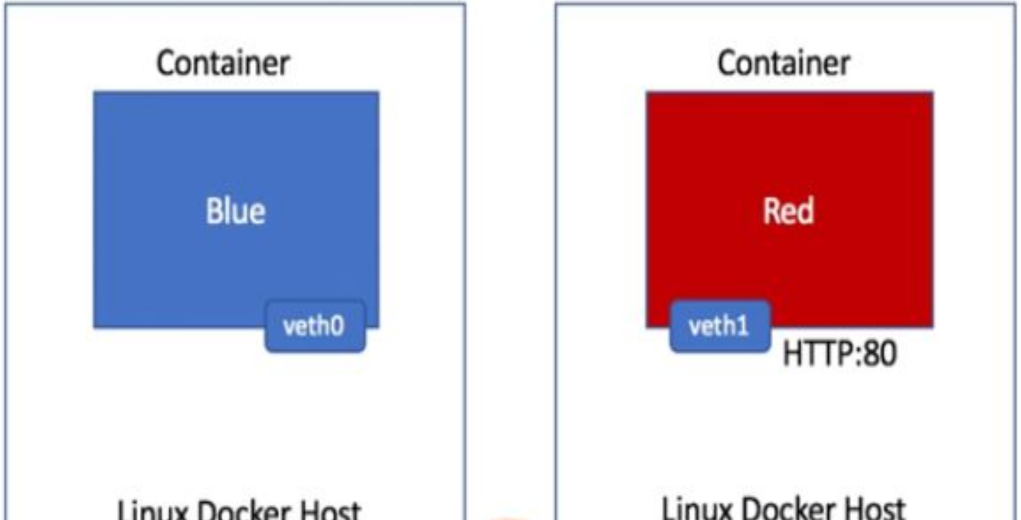


Solution: Linux Bridge and Iptables



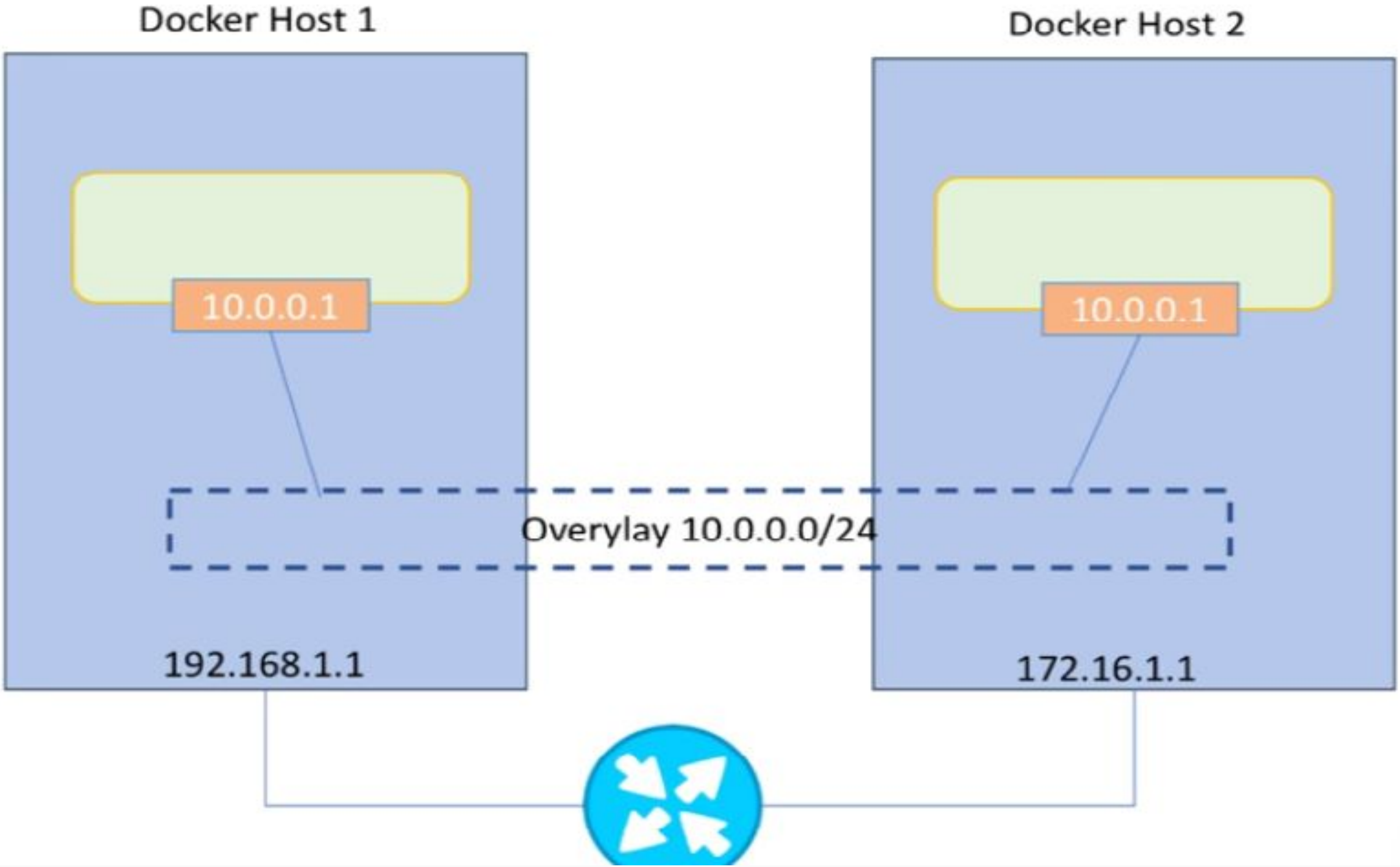
Multi Host Container Networking

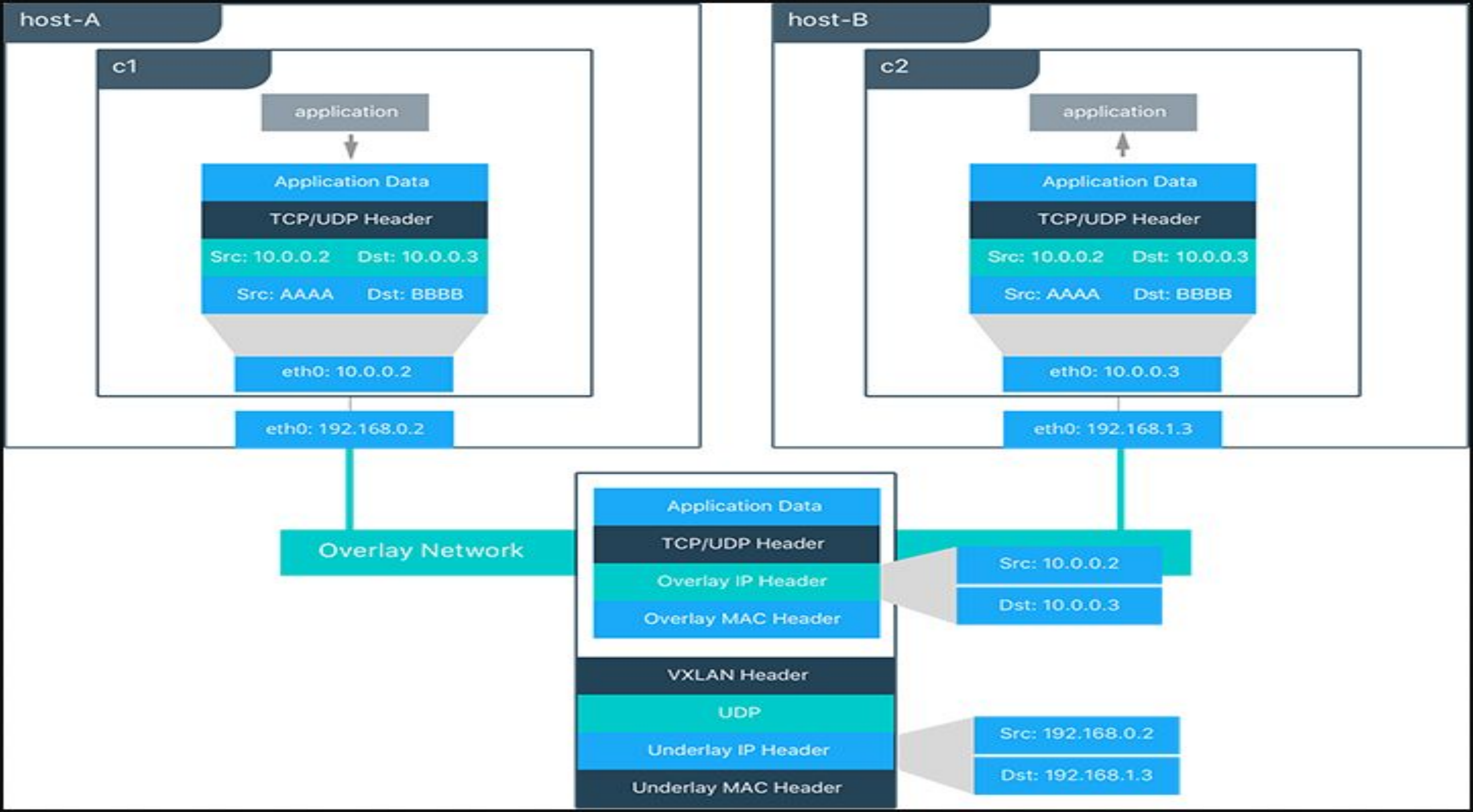
- ➔ How two containers located on different docker host communicate with each other?



Solution: Tunnel and Routing

- Tunnel
  - Docker build-in overlay network: VXLAN
  - OVS: VXLAN or GRE
  - Flannel: VXLAN or UDP
  - Weave: VXLAN or UDP
- Routing
  - Calico: Layer 3 routing based on BGP
  - Contiv: Layer 3 routing based on BGP


Docker Host 1
Docker Host 2
10.0.0.1
10.0.0.1
Overylay 10.0.0.0/24
192.168.1.1
172.16.1.1

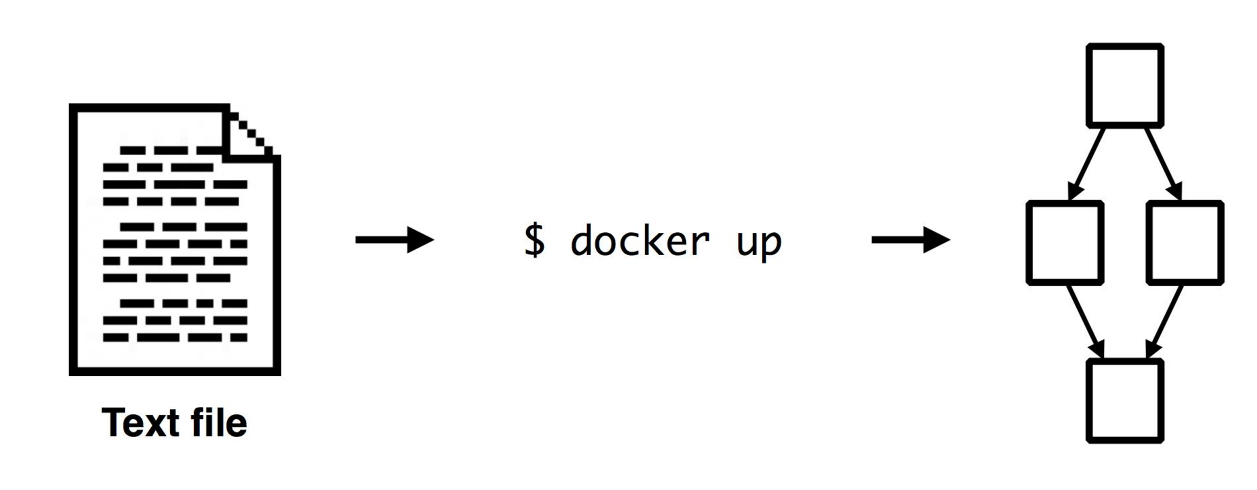

host-A
c1
application
Application Data
TCP/UDP Header
Src: 10.0.0.2
Dst: 10.0.0.3
Src: AAAA
Dst: BBBB
eth0: 10.0.0.2
eth0: 192.168.0.2
host-B
c2
application
Application Data
TCP/UDP Header
Src: 10.0.0.2
Dst: 10.0.0.3
Src: AAAA
Dst: BBBB
eth0: 10.0.0.3
eth0: 192.168.1.3
Overlay Network
Application Data
TCP/UDP Header
Overlay IP Header
Overlay MAC Header
Src: 10.0.0.2
Dst: 10.0.0.3
VXLAN Header
UDP
Underlay IP Header
Underlay MAC Header
Src: 192.168.0.2
Dst: 192.168.1.3

Docker Compose

Multi-container apps are a hassle

- Build images from Dockerfiles
- Pull images from the Hub or a private registry Configure and create containers
- Start and stop containers

Docker Compose



Overview

- Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your application’s services. Then, using a single command, you create and start all the services from your configuration.

app.py

```python
from flask import Flask
from redis import Redis
import os
import socket

app = Flask(__name__)
redis = Redis(host=os.environ.get('REDIS_HOST', 'redis'), port=6379)

@app.route('/')
def hello():
   redis.incr('hits')
   return 'Hello Container World! I have been seen %s times and my hostname is %s.\n' % (redis.get('hits'),socket.gethostname())

if __name__ == "__main__":
   app.run(host="0.0.0.0", port=5000, debug=True)
```

requirements.txt

flask

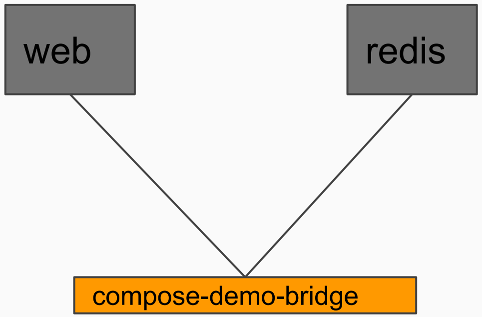
redis

Dockerfile

```
FROM python:2.7
MAINTAINER Peng Xiao "xiaoquwl@gmail.com"
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
EXPOSE 5000
CMD [ "python", "app.py" ]
```

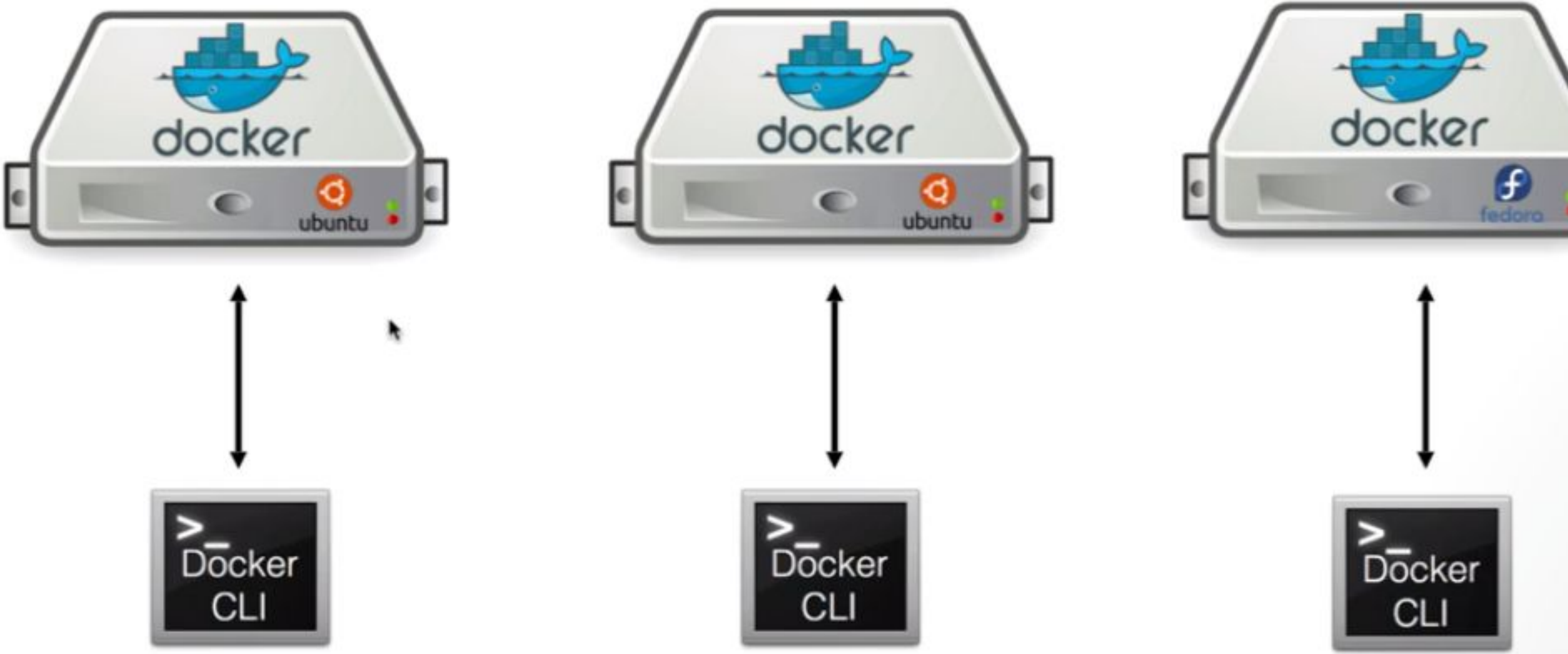
docker-compose.yml

```yaml
version: "2"

services:
  web:
    build: .
    ports:
      - "80:5000"
    links:
      - redis
    networks:
      - compose-demo-bridge

  redis:
    image: redis
    ports: ["6379"]
    networks:
      - compose-demo-bridge


networks:
  compose-demo-bridge:
```

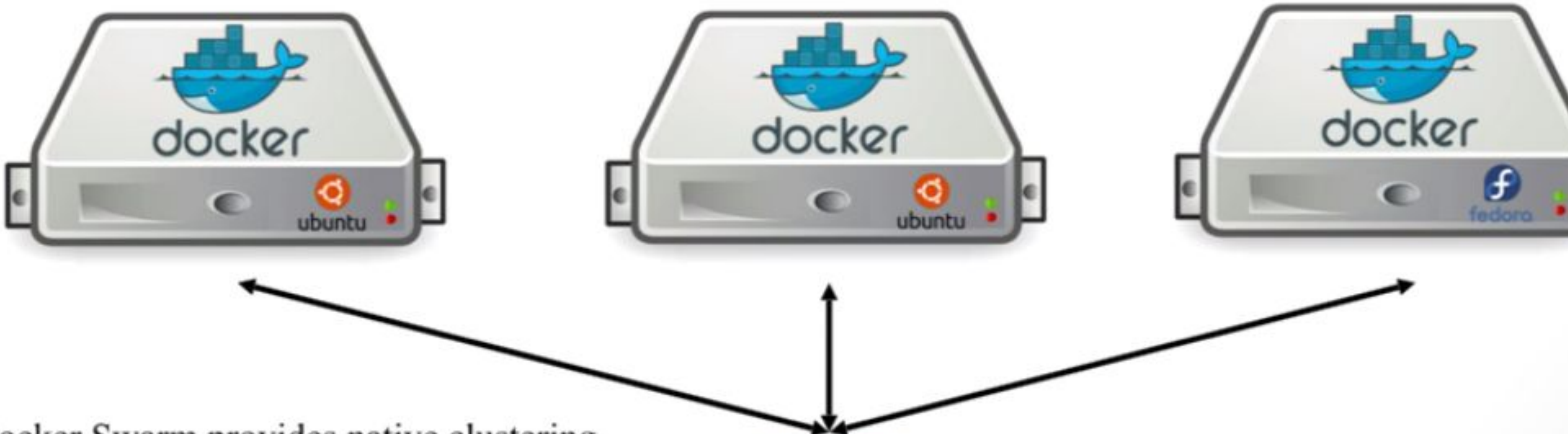


Docker Swarm

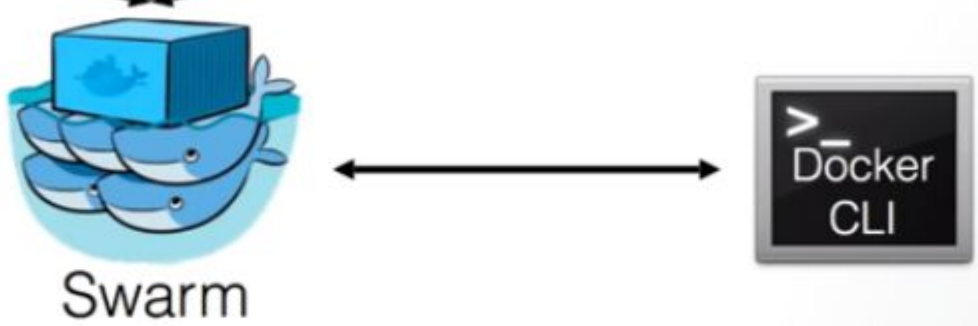
Before Docker Swarm



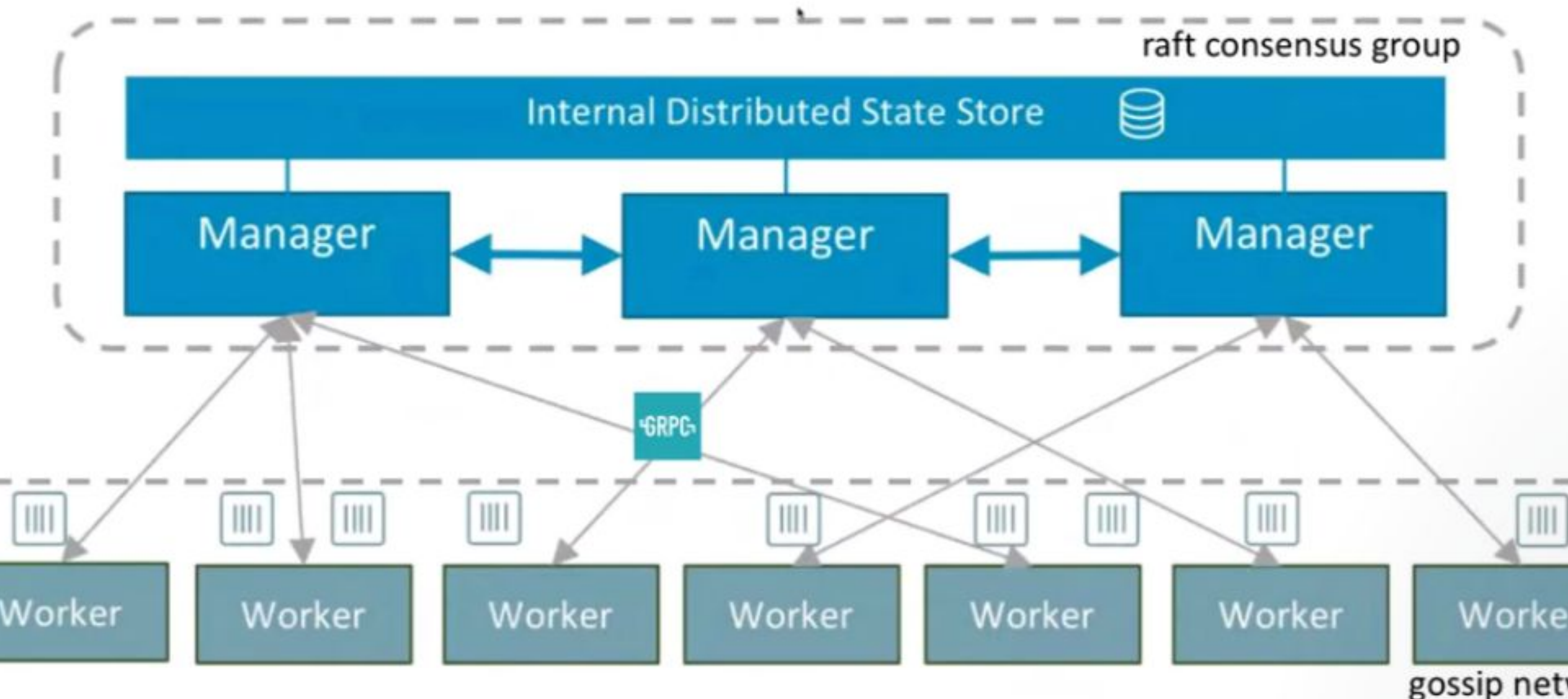
With Docker Swarm



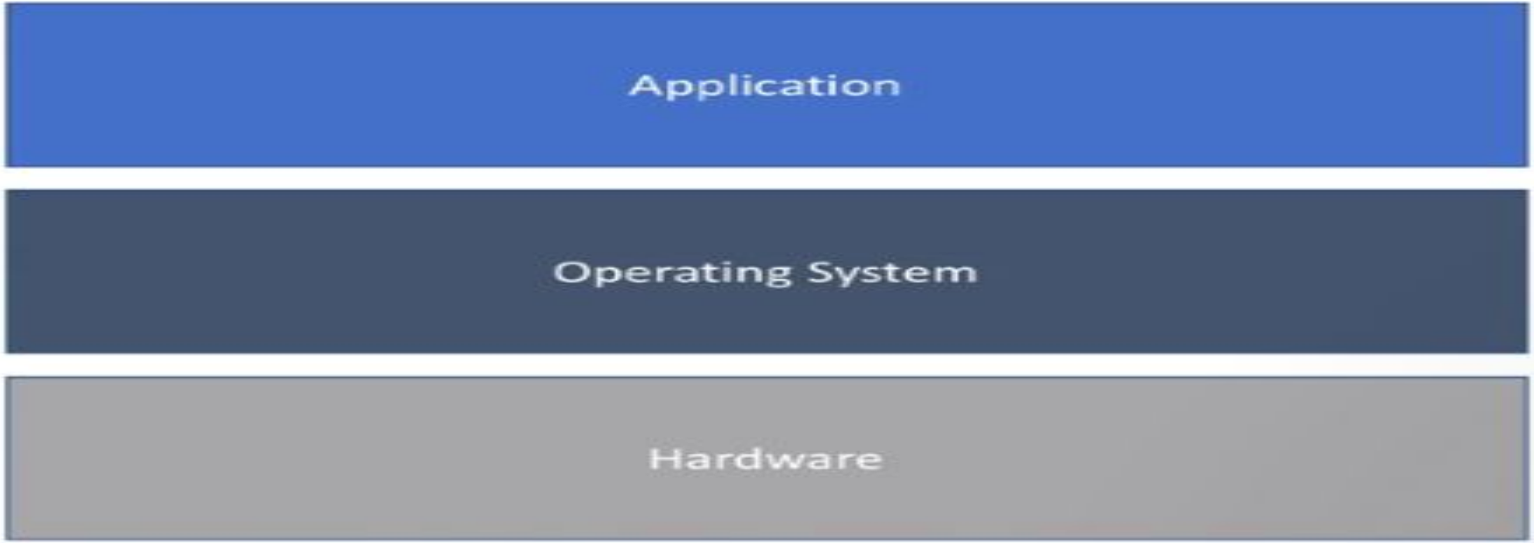
ocker Swarm provides native clustering
pabilities to turn a group of Docker
ngines into a single, virtual Docker Engine.



warm mode cluster architecture

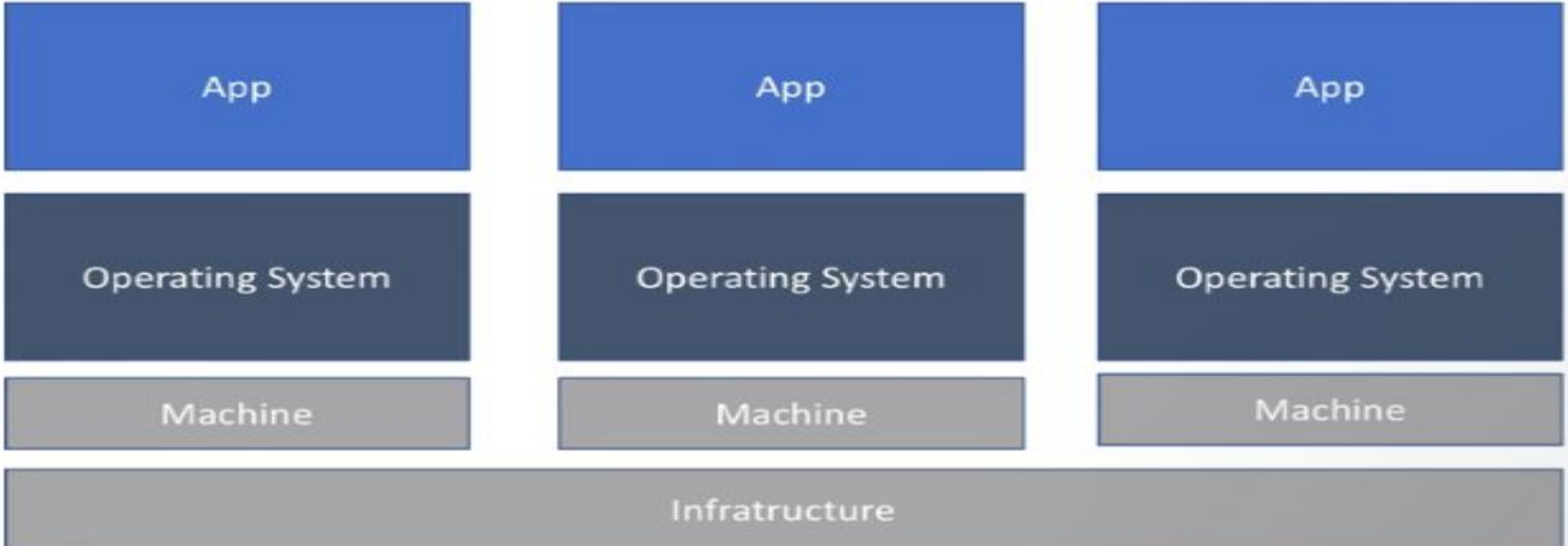


In the very Beginning...

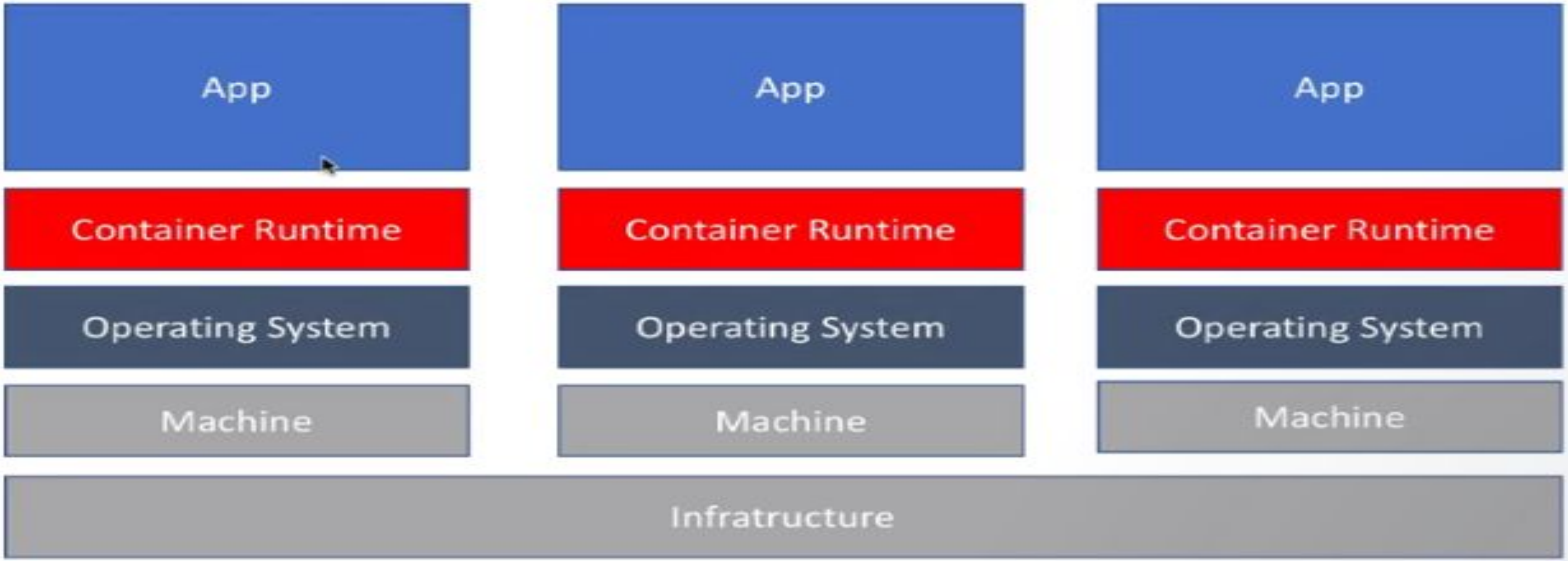


Scale & High Availability

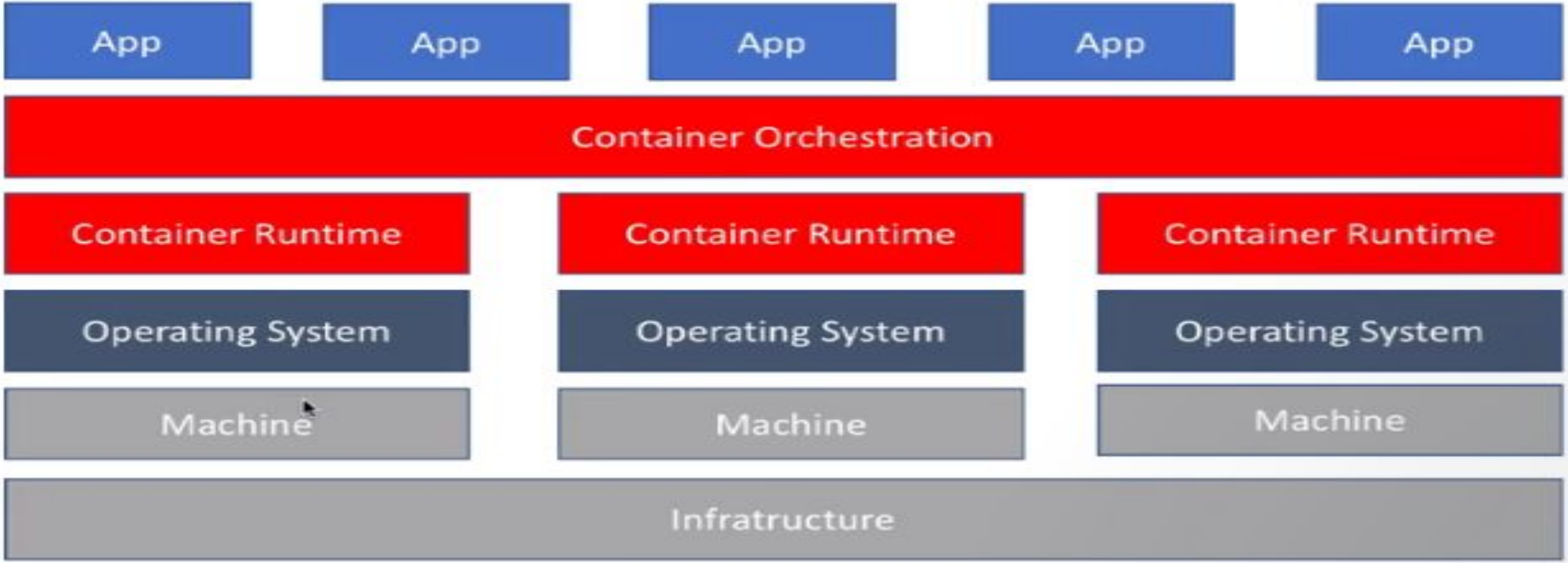
Hardware Virtualization



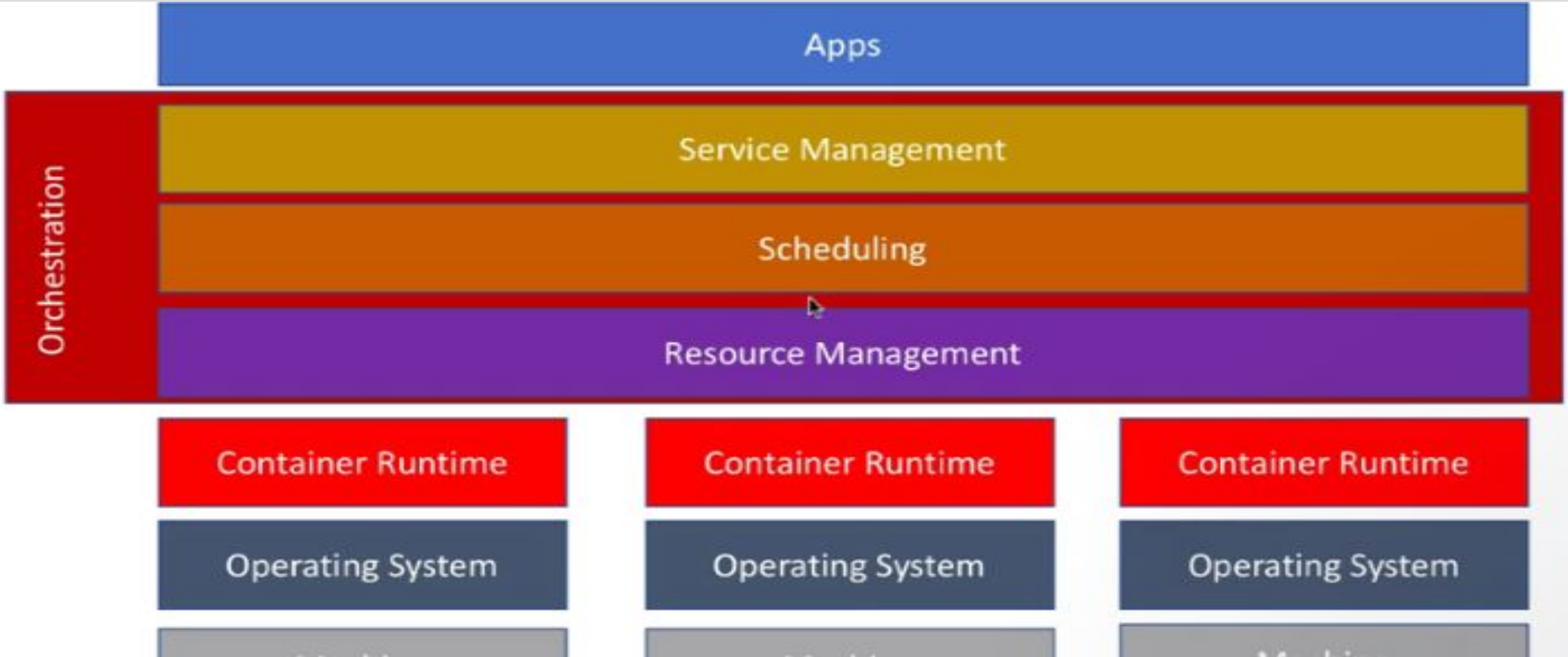
Containerized



Container Orchestration



Container Orchestration



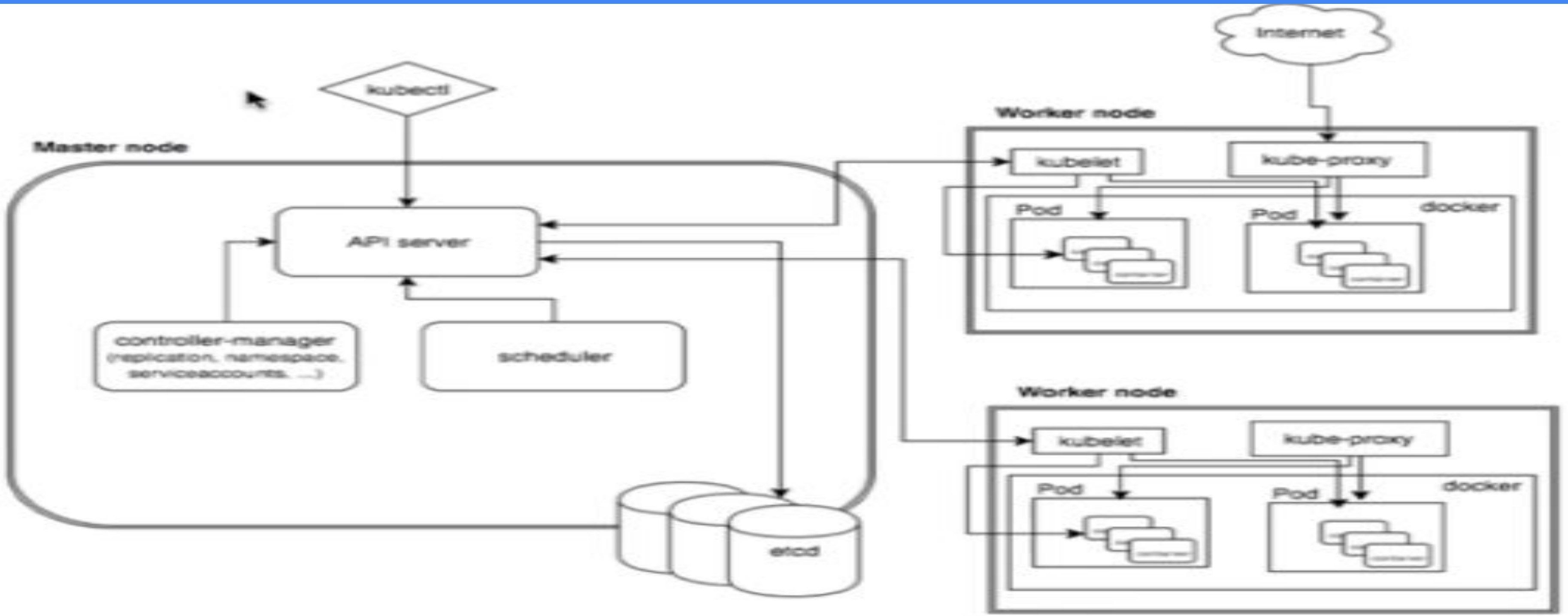
Container Orchestration

- Schedule Containers to physical/virtual machines
- Restart containers if they stop
- Provide private container network
- Scale up and Scale down
- Service Discovery

Container Orchestration war

Kubernetes

- Greek for “Helmsman”; also the root of the word “governor” and “cybernetic”
  - Orchestrator for containers
  - Builds on Docker containers
    - Also supporting other container technologies
  - Multi-cloud and bare-metal environments
  - Inspired and informed by Google’s experiences and internal systems
  - **100% Open Source**, written in **Go**.
  - Release 1.0 21th July 2015



Internet
kubectl
Master node
API server
controller-manager
(replication, namespace, serviceaccounts, ...)
scheduler
etcd
Worker node
kubelet
kube-proxy
Pod
Pod
docker
Worker node
kubelet
kube-proxy
Pod
Pod
docker

Kubernetes Deploymnet

Kubernetes Pod

- Pods- Pods are smallest deployable unit of computing that can be created and managed in kubernetes
- It is group of 1 or more containers. A pods are always co-located and co-scheduled and run in shared context
- Shared context of pod is a set of linux namespaces and cgroups, same thing that isolated docker containers.
- Containers within a pods share an same p address and pod space and find each other using localhost.
- They can communicate with each other using standard IPC communication.

Kubernetes Replication Controller

- It ensures that specified number of pods/replicas are running at any point of time.
- In other words replication controller make sure that pods or homogeneous set pod s are always up and available.
- If there are too many pods it will kill excess one and if there are less pods it will create new one to maintain that state.

Deployment

- It provides updates for pods and replica sets .
- Replica sets is the next generation of replication controller.
- You only need to describe a desired state in a deployment object
- And Deployment controller will change actual state to desired state.
- You can define a Deployments to create new resources or replace existing ones by new ones.