Cloud Systems

Chapter 3: Containers

Dr Lauritz Thamsen
lauritz.thamsen@glasgow.ac.uk
School of Computing Science
University of Glasgow

First Half: Lecture Chapters

Cloud Resource Management:

1. Cloud Computing Intro
2. Virtual Machines
3. **Containers**
4. Cloud Infrastructure Management
5. Cloud Sustainability

Outline of Chapter 3: Containers & Container Mngt.

3.1 Intro to Containerization

3.2 Linux Containment Features

3.3 Container Technologies (like LXC and Docker)

– short break (10-15 minutes) –

3.4 Containers vs. VMs

3.5 Container Orchestration

3.6 Orchestration Systems (like Kubernetes)

Outline of Chapter 3: Containers & Container Mngt.

3.1 Intro to Containerization

3.2 Linux Containment Features

3.3 Container Technologies (like LXC and Docker)

– short break (10-15 minutes) –

3.4 Containers vs. VMs

3.5 Container Orchestration

3.6 Orchestration Systems (like Kubernetes)

Containerization

- Also known as OS-Level Virtualization
- Motivation: System virtualization comes with too much overhead, large images, and long boot times
- Idea: Do not create *virtual machines*, instead…
  - Reuse operating system kernel but *isolate applications*
  - Virtualize access to resources used by processes
    - File system
    - Devices
    - Network
    - Other processes
    - …

History of OS-Level Virtualization

- Unix v7 **`chroot`** system call (1979)
  - Allows setting the file system root for processes
  - Unix philosophy: (almost) everything is accessed through the file system → (almost) everything can be virtualized though **`chroot`**
- Wave of container technologies in early 2000s
  - FreeBSD Jails, Linux VServer, Solaris Zones, …
  - Different degrees of isolation, different tool chains
  - Mostly specific to Linux distributions
  - Not widely popular, mainly used by sys admins in large companies

History of OS-Level Virtualization

- LXC (2008)
  - User space tools for accessing Linux kernel process isolation features (e.g. *cgroups* and *namespaces*)
- Docker (2013)
  - Most widely used container technology and ecosystem
- Rocket (CoreOS, 2014)
  - Alternative to Docker, more focused on security and standardization
- LXD (Canonical, 2014)
  - Image-based, focused on entire Linux distributions
- Ubuntu Snap (Canonical, 2018)
  - Universal packaging format for Linux distros

Outline of Chapter 3: Containers & Container Mngt.

3.1 Intro to Containerization

3.2 Linux Containment Features

3.3 Container Technologies (like LXC and Docker)

– short break (10-15 minutes) –

3.4 Containers vs. VMs

3.5 Container Orchestration

3.6 Orchestration Systems (like Kubernetes)

Linux Kernel Features for OS-Level Virtualization

- Linux kernel contains numerous mechanisms for process isolation:
  - `chroot` system call
  - Namespaces
  - Capabilities
  - cgroups
  - SELinux
  - `seccomp`
  - ...
- Partially overlapping functionality, yet different configuration approaches

chroot

- Oldest mechanism for process isolation
- System call that changes the root directory (“/”) of the calling process
- Since “everything is a file” in Linux, many aspects of the underlying system can be virtualized
- System calls, networking, etc. remain unchanged

```
/
├── bin
├── boot
├── dev
├── etc
├── home
├── container-root
│   ├── bin
│   ├── boot
│   ├── dev
│   ├── etc
│   ├── home
│   ├── lib
│   ├── proc
│   …
├── lib
├── proc
…
```

View on the host system

```
/
├── bin
├── boot
├── dev
├── etc
├── home
├── lib
├── proc
…
```

View in the chrooted process

Linux Namespaces

- Provide separate views on kernel resources for processes in different namespaces

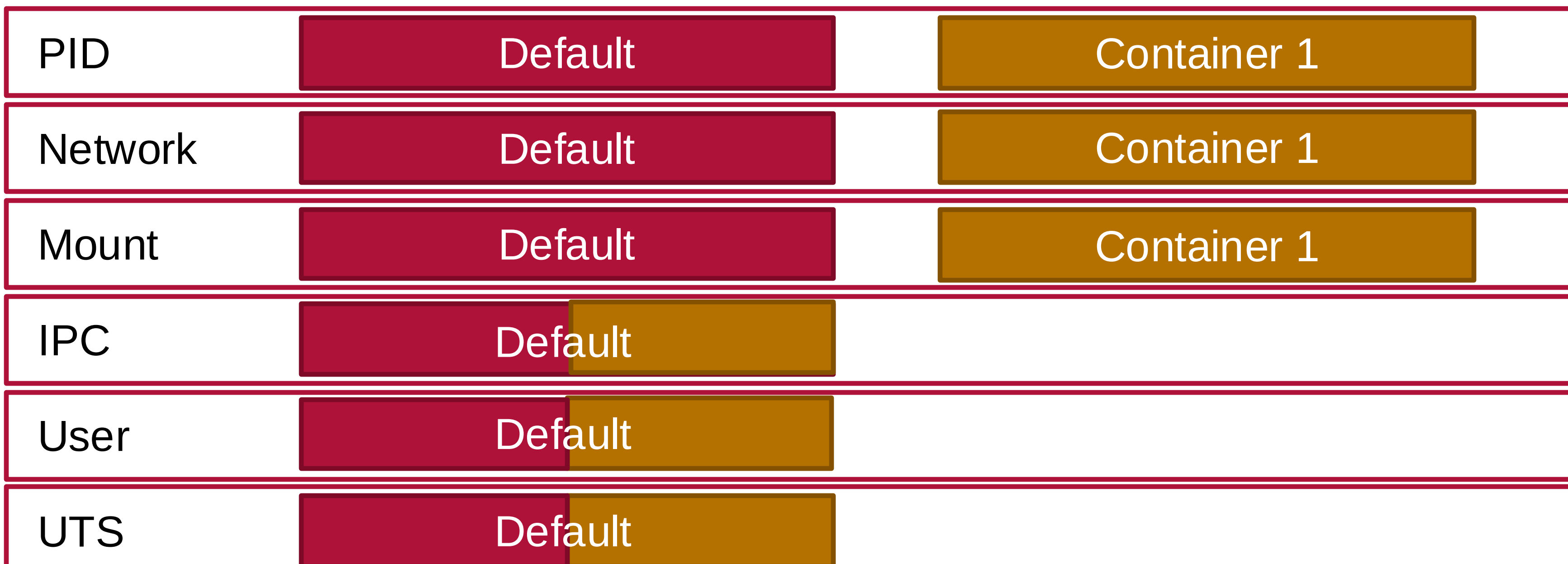
- Resources that can be isolated:
  - PID (processes)
  - IPC (inter-process communication)
  - Network (devices, protocol stacks, firewalls, ports, …)
  - Mount (mount points, file system structure)
  - User (users and groups)
  - UTS (hostname and domain name)

Linux Namespaces Example

- Container 1: Share users and hostname, but isolate processes, network, and mount points

Namespaces used by processes running in Container 1

Namespaces used by other processes

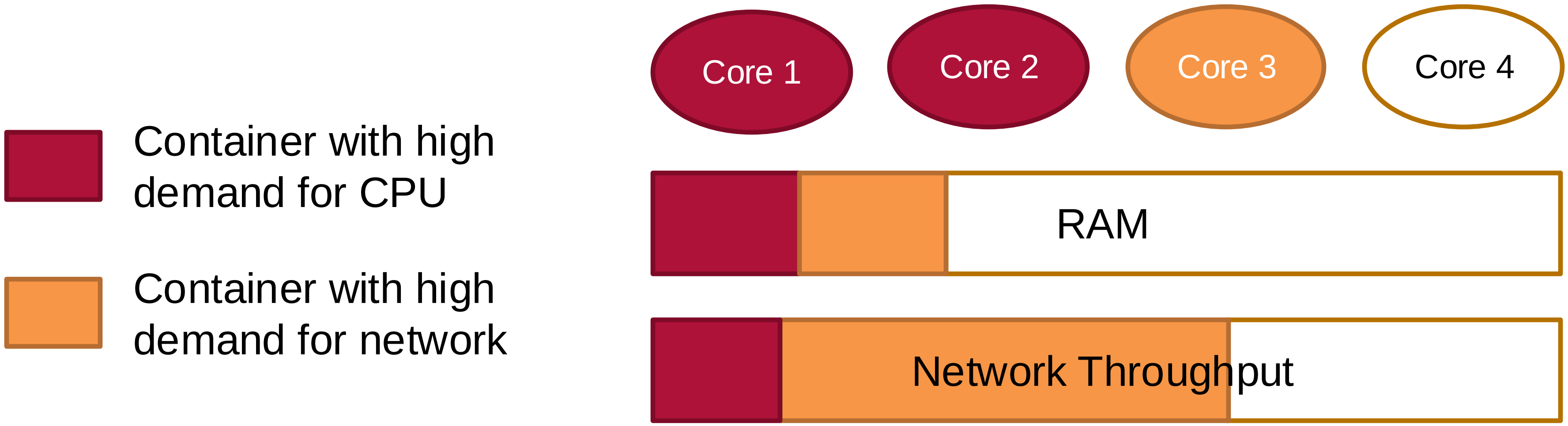


Linux Capabilities

- Traditionally, super user (root) is the only one to perform administrative tasks on a Linux system
- Linux capabilities allow to assign selected rights to new processes
- Long list of possible capabilities, including
  - many system calls,
  - device access,
  - file system modifications,
  - …

Linux cgroups (Control Groups)

- Allow definition of resource usage constraints for parts of the process tree
- Used for limiting CPU, RAM, network, and disk usage for containers



Linux Security Policies

- SELinux, AppArmor
  - Optional kernel modules for security
  - Allow rule-based confinement of processes to limit their access to certain files and devices only
  - SELinux: complex, but powerful rule definitions
  - AppArmor: simpler security profiles

- Seccomp policies
  - `seccomp` system call puts the process into a secure computation mode, where only selected system calls are allowed

Outline of Chapter 3: Containers & Container Mngt.

3.1 Intro to Containerization

3.2 Linux Containment Features

3.3 Container Technologies (like LXC and Docker)

– short break (10-15 minutes) –

3.4 Containers vs. VMs

3.5 Container Orchestration

3.6 Orchestration Systems (like Kubernetes)

Examples for Container Technologies

- LXC: First container technology widely adopted

- Docker: Currently most widely used container platform and eco system

LXC Containers

- LXC: Library and userspace tools (scripts) to access kernel process isolation features
- No daemon, instead containers and their file systems are accessible via `/var/lib/lxc`

Language Bindings
(Python, Lua, Go, Python, Ruby, …)

Userspace tools
`lxc-create, lxc-start, lxc-stop, …`

`liblxc`

Kernel features:
Namespaces, capabilities, chroot, cgroups,
AppArmor, SELinux, seccomp

LXC Container Creation

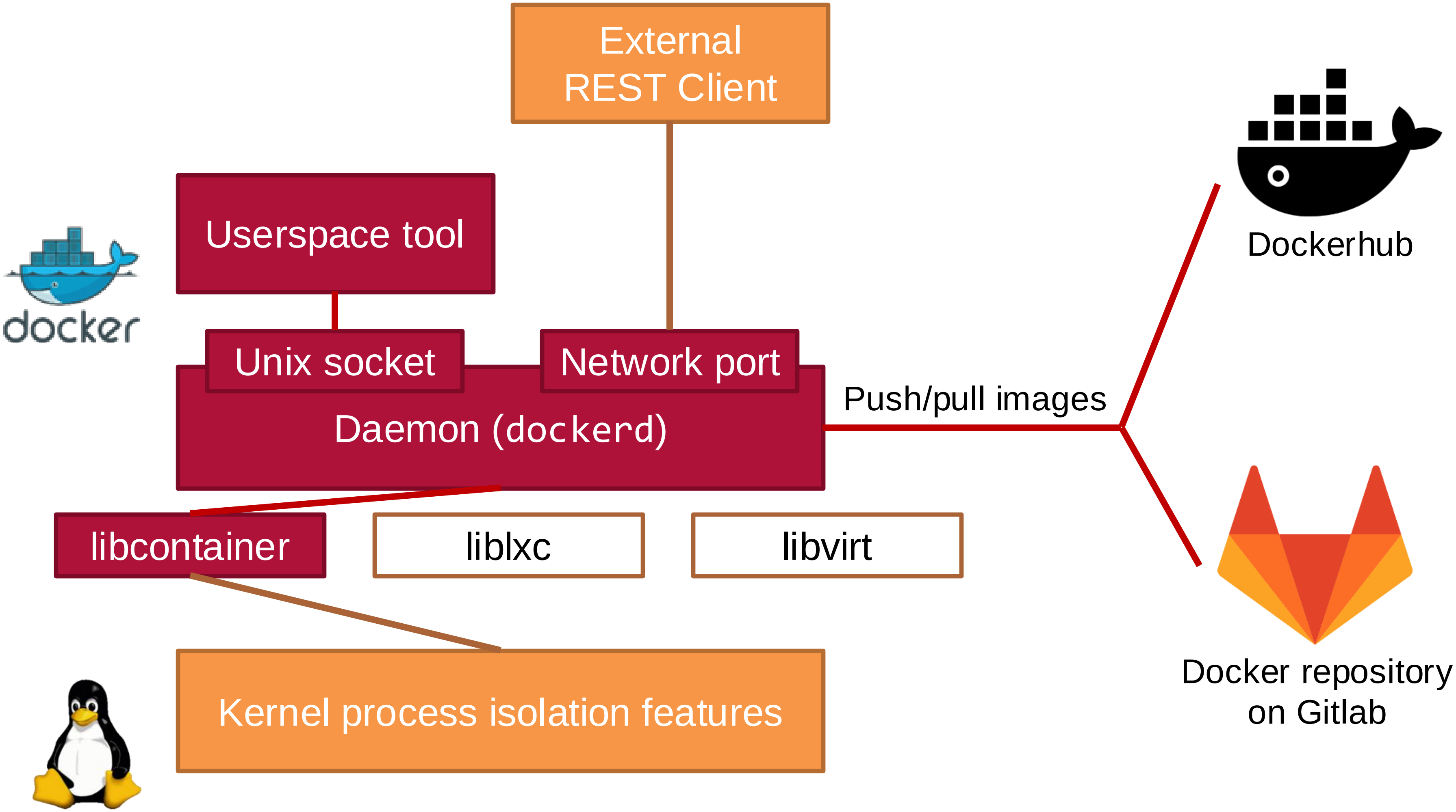
- Container Creation
  1. Container directory allocated under `/var/lib/lxc`
  2. "Template script" executed to populate the file system of the container
  3. Process is spawned and `chrooted` into its file system
  4. Process isolation is configured
  5. Main application process is spawned, all child processes will inherit isolation context
- LXC does not rely on an image format
- Not much infrastructure for sharing images

Docker

- Most popular container technology at the moment

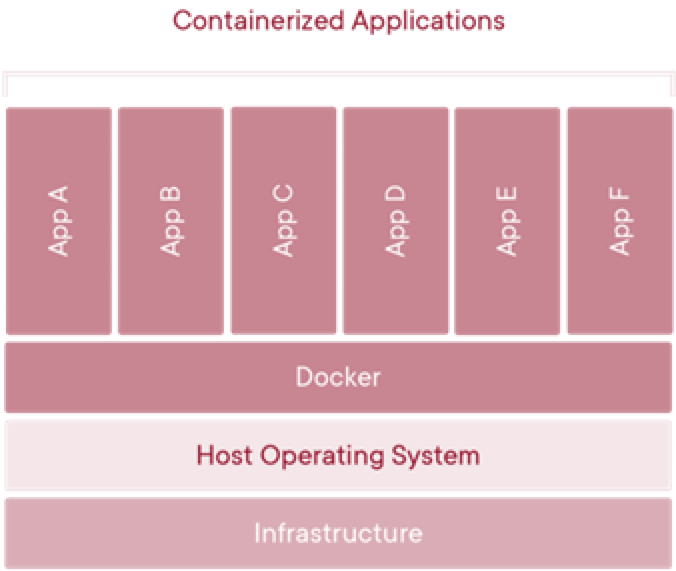
- Docker includes:
  - Daemon and user space tools for managing local containers and images
  - Hierarchical image format
  - Build tool for images, called *Dockerfiles*
  - Private and public repositories for sharing images (e.g. *Dockerhub*)
  - Internal and external tools for automatic orchestration of container infrastructures

Docker Components



Docker Instances, Images, and Files

- Multiple containers share a single OS kernel, isolated by kernel containment features



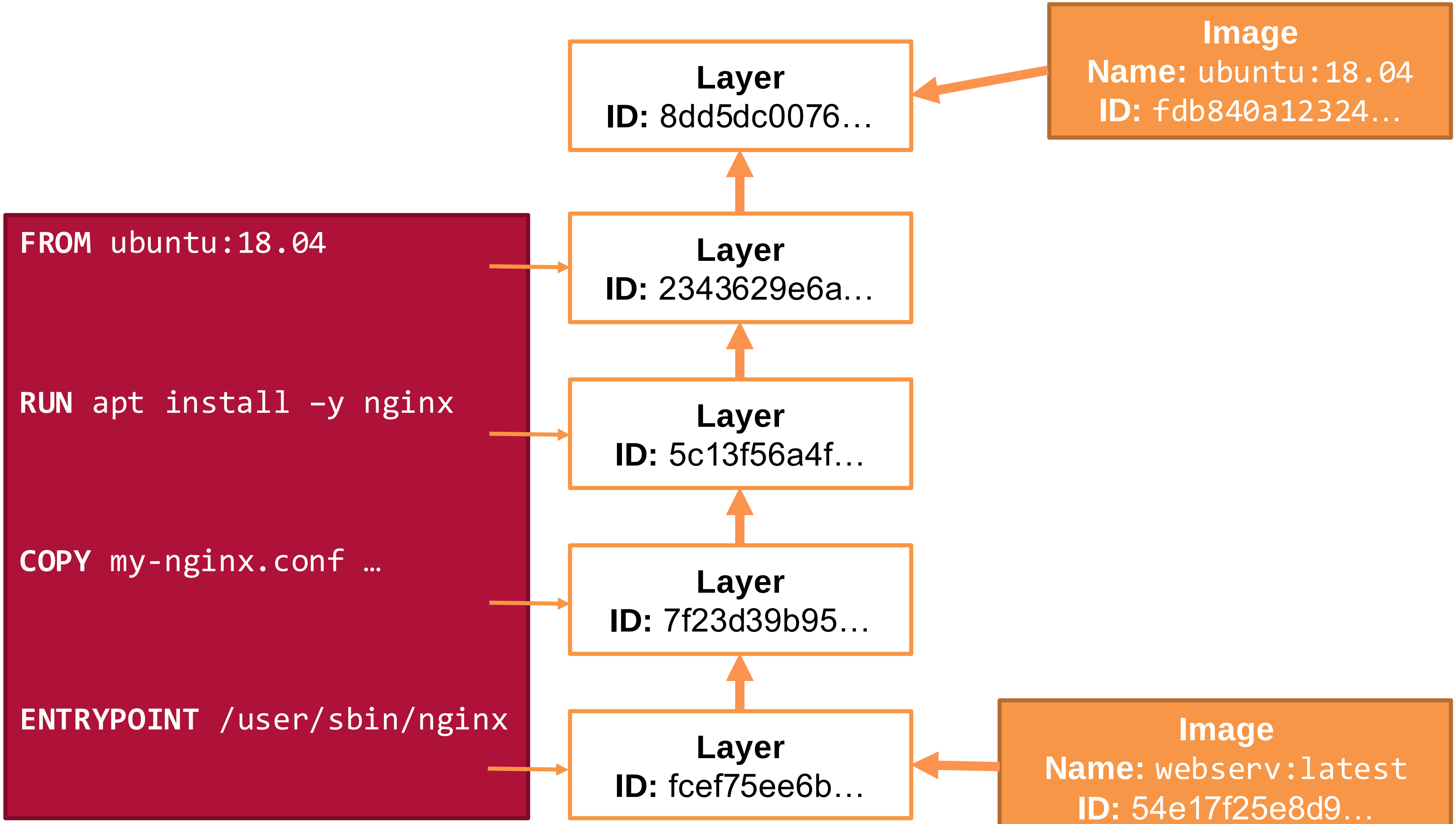
- Container instance: running an isolated process, started from an image
- Docker image: snapshot of a container, packaging the application, its dependencies, and its configuration
- Dockerfile: instructions for creating images

Dockerfiles

- Automatic creation of container images
- Text file with commands that modify files or change configuration
- Intermediate steps can be cached, resulting in reduced image build times

```
FROM ubuntu:18.04
RUN apt install -y nginx
COPY my-nginx.conf /etc/nginx/nginx.conf
ENTRYPOINT /user/sbin/nginx
```

Docker Image Format



Docker Images

- Docker images contain:
  - List of layers
  - ID (Hash of layer IDs and other config data)
  - Configuration data: ports, mounts, env. variables, …
  - Meta data (creation time, author, history, …)
- Each image layer contains:
  - ID of the layer (hash of all files) and ID of parent layer
  - Layer files (tarball): Files that were added/changed on this layer, relative to parent layer
  - Command that was used to create the layer

Another Dockerfiles Example

- Creating docker images from configuration files

- Example: Dockerfile for an Express web server image using NodeJS

```
$ docker build --tag express .

$ docker run --rm express
```

```
# Create image based on the official Node 6 image from dockerhub
FROM node:9

# Create a dir where our app will be placed
RUN mkdir -p /usr/src/app

# Change dir so that our commands run inside this new dir
WORKDIR /usr/src/app

# Copy dependency definitions
COPY package.json /usr/src/app

# Install dependecies
RUN npm install

# Get all the code needed to run the app
COPY . /usr/src/app

# Expose the port the app runs in
EXPOSE 3000

# Serve the app
CMD ["npm", "start"]
```

Outline of Chapter 3: Containers & Container Mngt.

3.1 Intro to Containerization

3.2 Linux Containment Features

3.3 Container Technologies (like LXC and Docker)

– short break (10-15 minutes) –

3.4 Containers vs. VMs

3.5 Container Orchestration

3.6 Orchestration Systems (like Kubernetes)

Outline of Chapter 3: Containers & Container Mngt.

3.1 Intro to Containerization

3.2 Linux Containment Features

3.3 Container Technologies (like LXC and Docker)

– short break (10-15 minutes) –

3.4 Containers vs. VMs

3.5 Container Orchestration

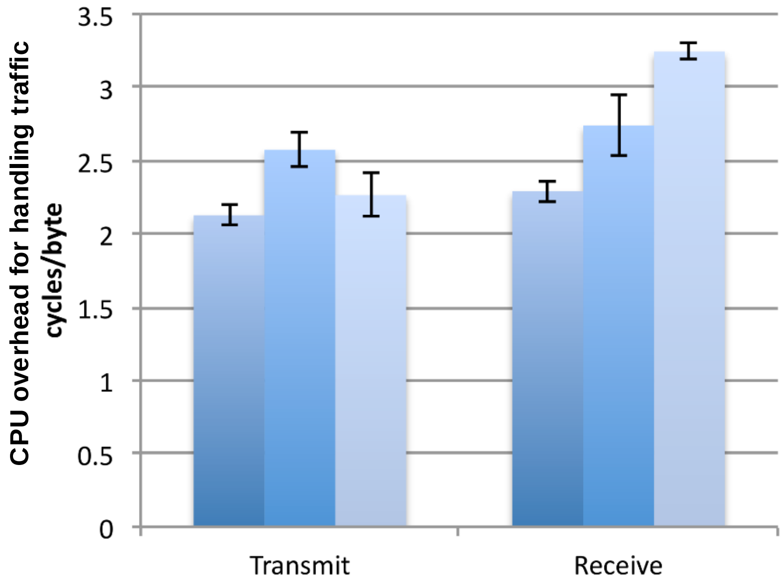
3.6 Orchestration Systems (like Kubernetes)

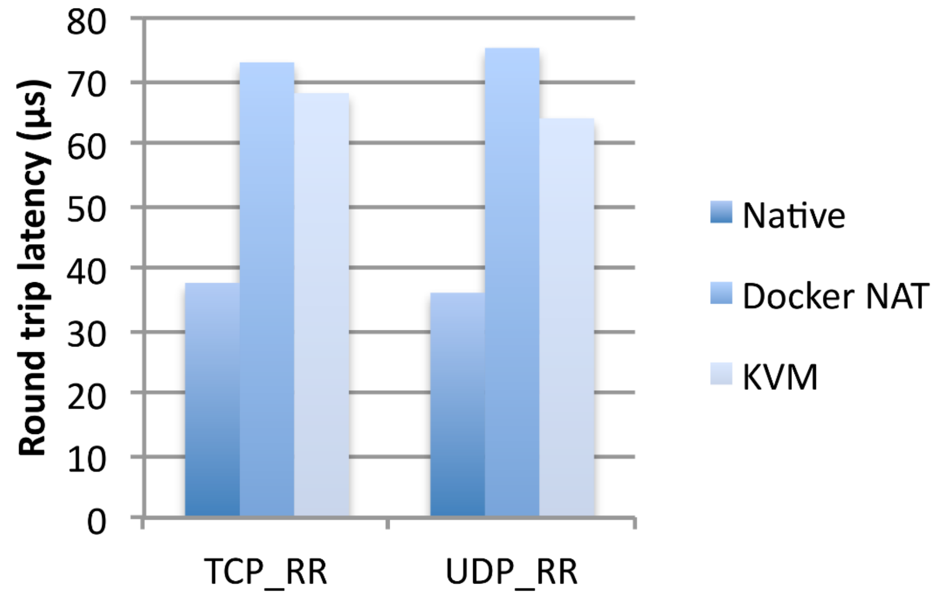
Containers vs VMs [19]: Performance (CPU & RAM)

Workload	Native	Docker	KVM
Linpack (GFLOPS)	290.8 [±1.13]	290.9 [±0.98]	284.2 [±1.45]
Memory (Random Access, GIOps/s)	0.0126 [±0.00029]	0.0124 [±0.00044]	0.0125 [±0.00032]
Memory (Sequential Access, GB/s)	45.8 [±0.21]	45.6 [±0.55]	45.0 [±0.19]

- VMs and container introduce low CPU and memory access overhead
  - Prerequisite: exposing cache topology and CPU acceleration features (e.g. NUMA, FPUs, SSE)

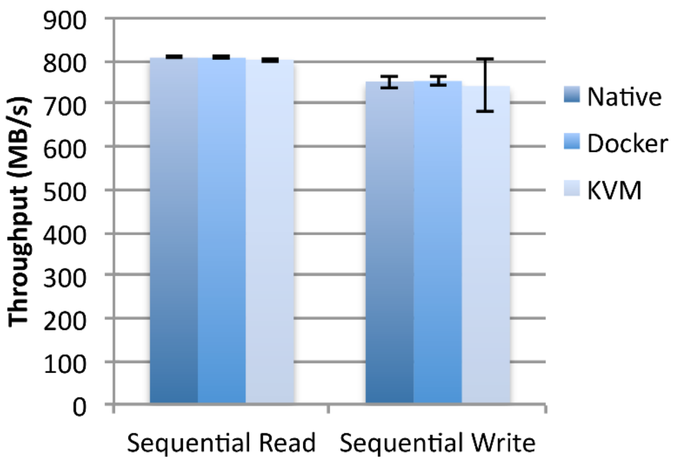
Containers vs VMs [19]: Performance (Network)

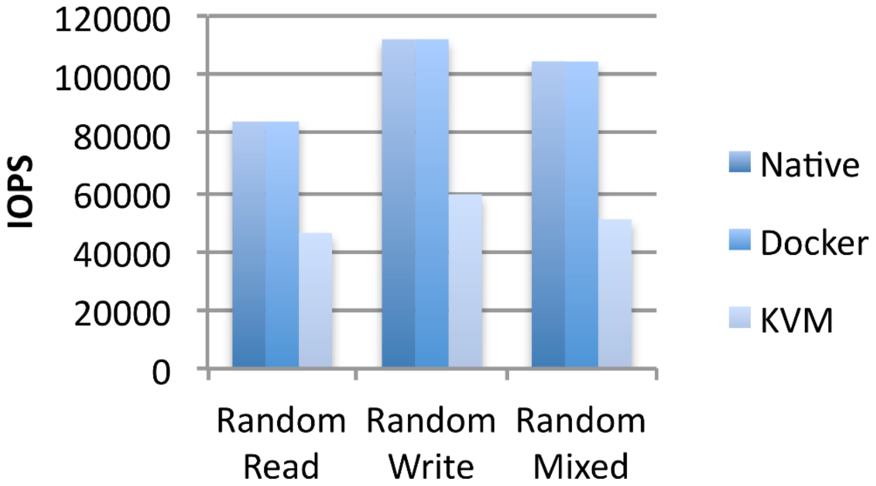




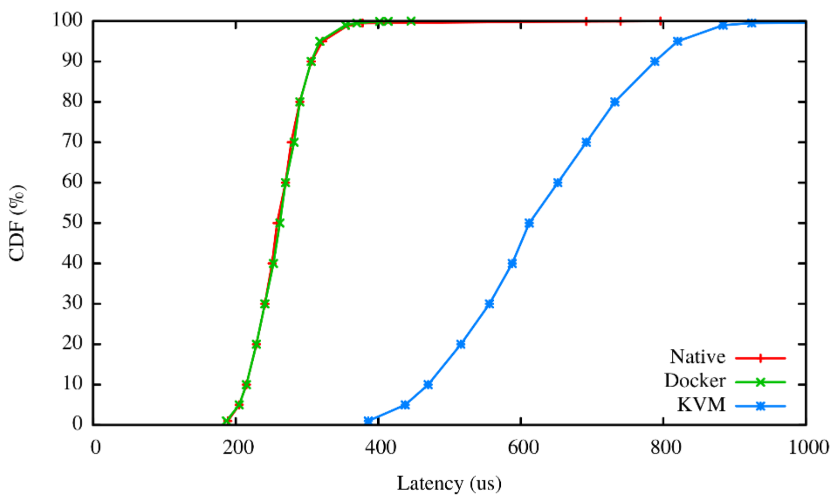
- Low CPU overhead per packet
- Reasons for latency increase:
  - NAT in Docker networking
  - Virtual network device in KVM (NAT might further increase latency)

Containers vs VMs [19]: Performance (Disk IO)





- Similar throughput
- Penalty for latency and IOPs due to virtual IO device



Containers vs VMs: Image Size & Boot Time

- Virtual machine images usually larger than container images (contain entire OS)
- Image caching on execution hosts for both VMs and containers
- Boot time of VMs can be orders of magnitude longer than container startup

Containers vs VMs: Isolation & Security

- Containers share the host OS kernel
  - Vulnerable to Linux kernel bugs
  - Denial-of-Service attacks: Resource usage, system calls, and context switches of one container can starve others
  - Kernel parameters cannot be tuned to workload

- VMM orders of magnitude less code than OS kernels → more exploits discovered and fixed?

Outline of Chapter 3: Containers & Container Mngt.

Container Analogy

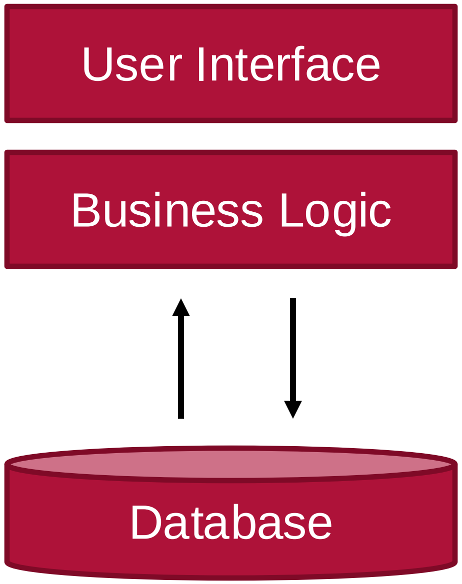
- Containers
  - Standardized
  - Easy to move
  - Isolated
  - Many containers fit hosts

- Dependency management: libs and config bundled with the application to run it everywhere → build your app in a container, test the container locally or your staging environment, then ship the container

Container Composition

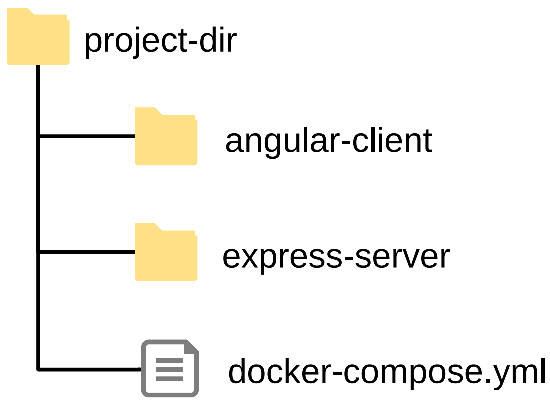
- Applications typically consist of multiple connected components intended to run as a single service

- e.g. Web Application
  - Front-end framework
  - Back-end framework
  - Persistent data store



Docker Compose

- Tool for defining and running multi-container Docker applications on one node

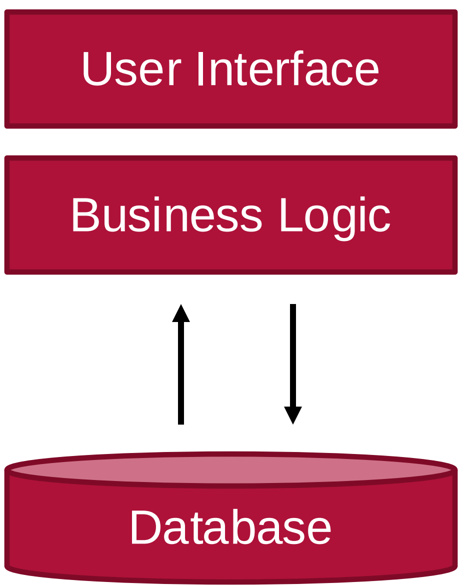



```
$ docker compose up
```

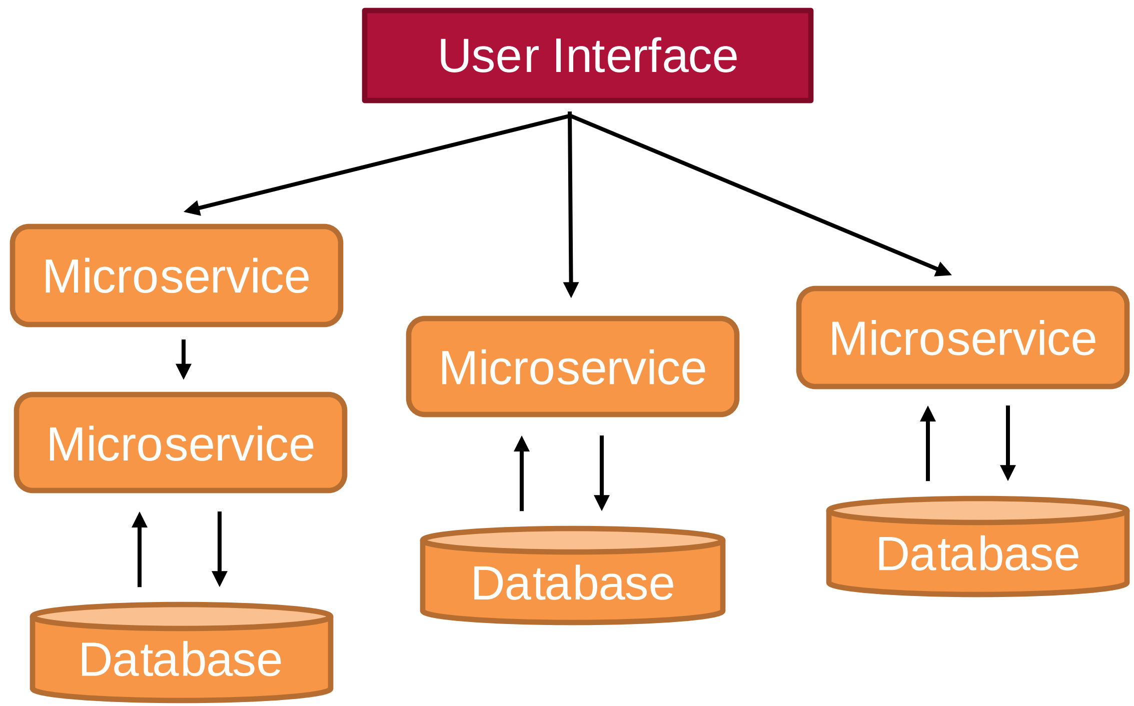
```yaml
version: '2' # specify docker-compose version

# Define the services/containers to be run
services:
  angular:
    build: angular-client # Dockerfile directory
    ports:
      - "4200:4200" # port forwarding

  express:
    build: express-server # Dockerfile directory
    ports:
      - "3000:3000" # port forwarding

  database:
    image: mongo # image to build container from
    ports:
      - "27017:27017" # port forwarding
```

Microservices (1/2)

Monolith

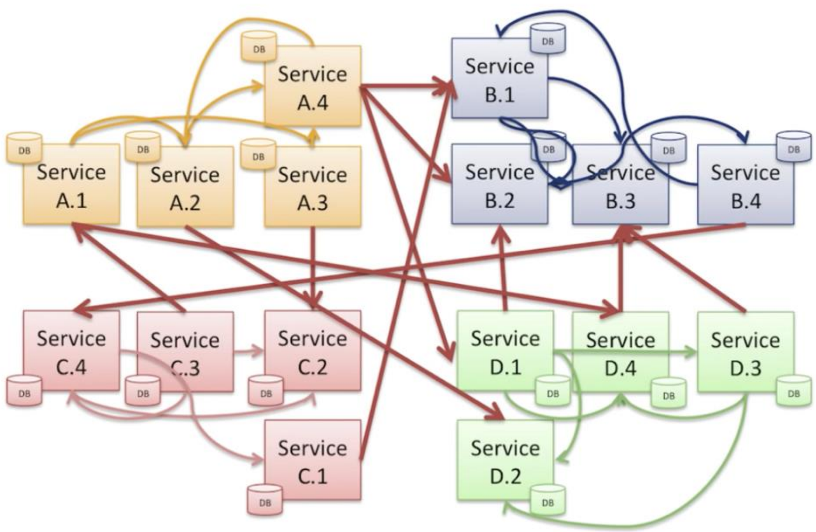


Microservices



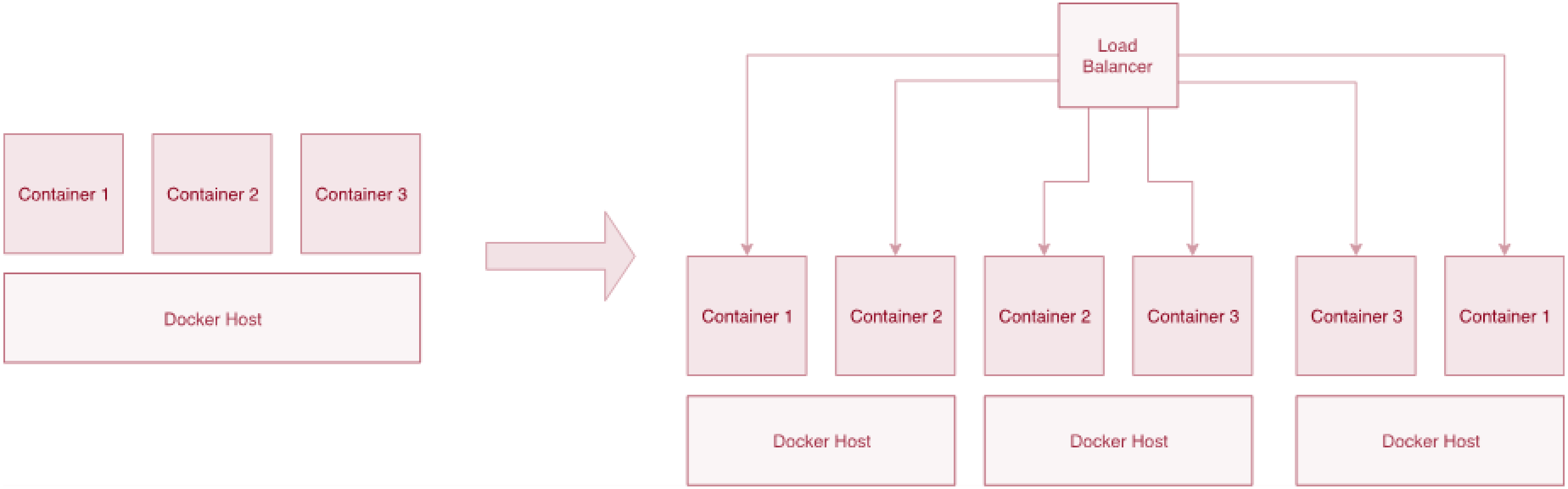
Microservices (2/2)

- Advantages
  - Independent development
  - Small teams
  - Fault isolation
  - Scalable
- Disadvantages
  - Overhead (duplicated tech, e.g. databases)
  - Complexity of services and networking



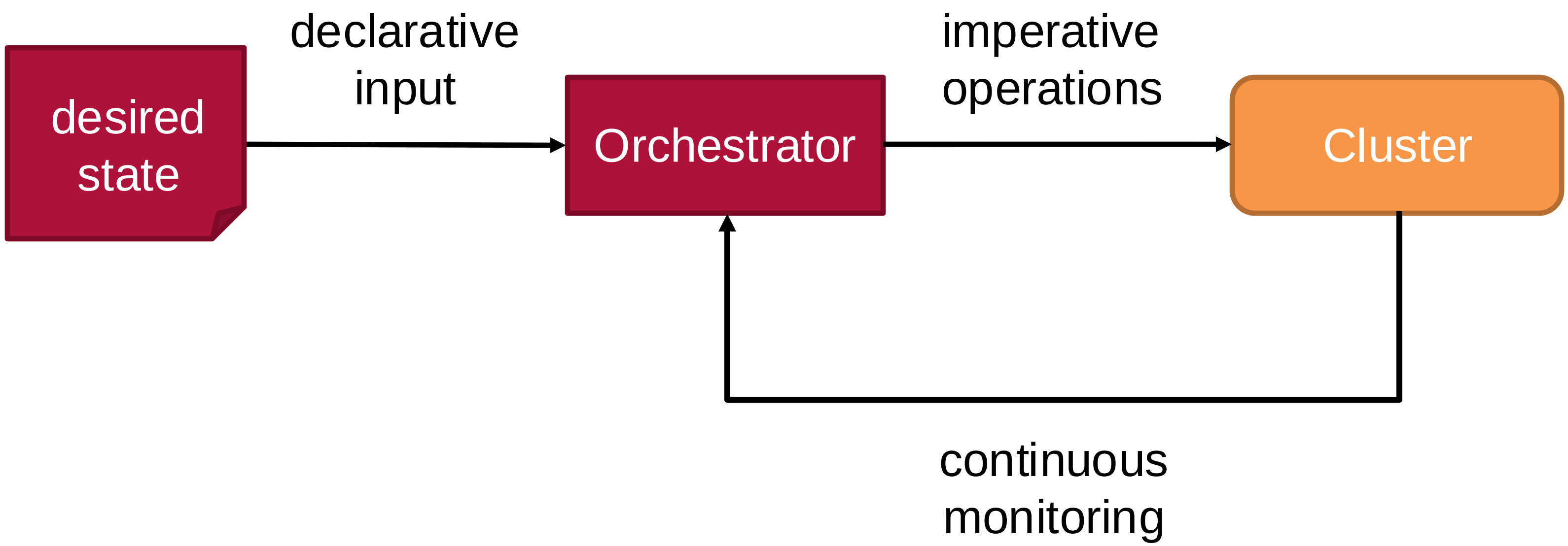
Docker Cluster

- Running an application on a cluster of docker hosts for fault tolerance and scalability



Resource Orchestration

- Orchestration tools: Control systems for clusters



Container Orchestration

Container Orchestration	Distributed container management
Container Runtime	Local container management
Infrastructure	Container-agnostic infrastructure

- Provisioning and deployment of containers
- Configuration and networking of containers
- Replication and availability of containers
- Monitoring of replicated containers
- Load balancing across replicated containers

Outline of Chapter 3: Containers & Container Mngt.

3.1 Intro to Containerization

3.2 Linux Containment Features

3.3 Container Technologies (like LXC and Docker)

– short break (10-15 minutes) –

3.4 Containers vs. VMs

3.5 Container Orchestration

3.6 Orchestration Systems (like Kubernetes)

Kubernetes

- Greek for “helmsman” or “pilot”, abbreviated ”k8s”

- Open-sourced by Google in 2014
- Version 1.0 released in July 2015

- Google and the Linux Foundation created the Cloud Native Computing Foundation (CNCF) to make Kubernetes an open standard

- Google Kubernetes Engine (GKE), Azure Container Service (AKS), AWS Elastic Container Service (EKS)

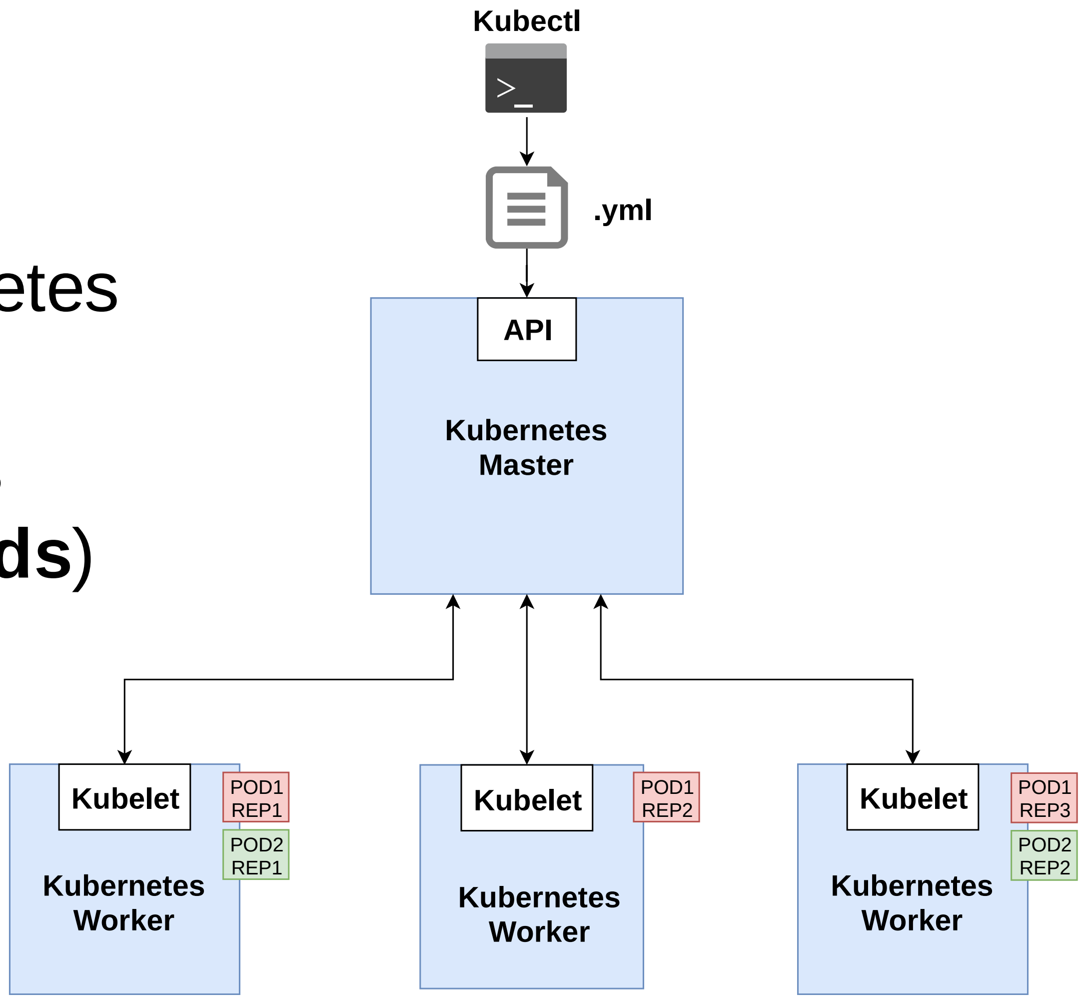
What Does Kubernetes Do for You?

- Automation engine for cluster management, based on a declarative description of the desired cluster state

- Deployment, monitoring, recovery, and scaling
  - Instantiating sets of containers
  - Connecting containers through agreed interfaces
  - Exposing services to machines outside of the cluster
  - Monitoring, logging, and re-starting/scheduling
  - Dynamically scaling the cluster

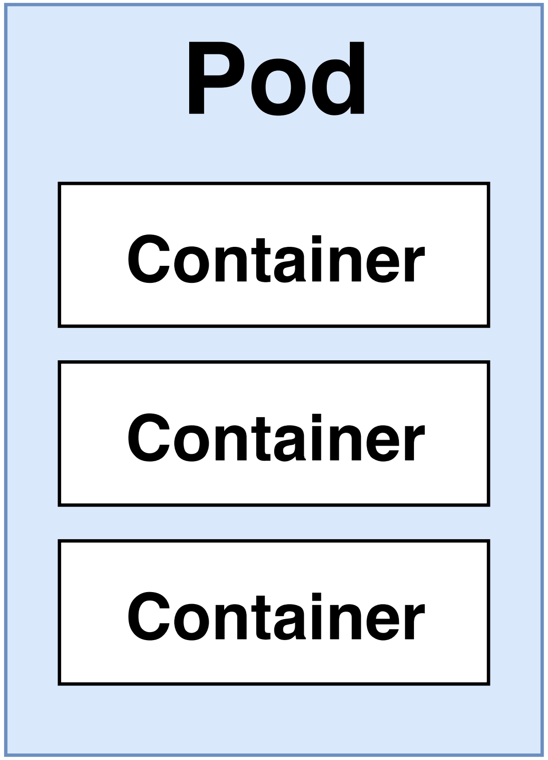
Kubernetes Cluster

- Collection of nodes (either bare-metal or virtual machines), managed by Kubernetes
- Runs groups of replicated containers (which are called **Pods**)



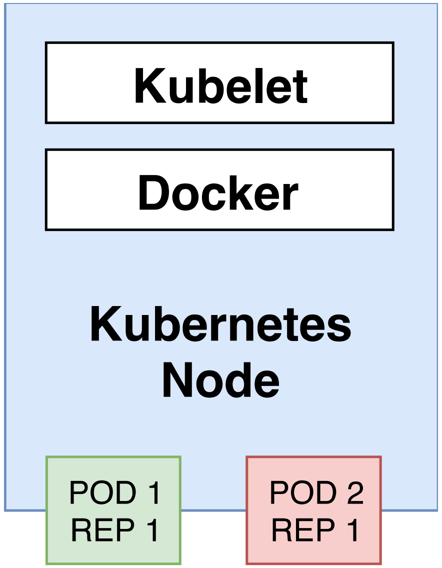
Kubernetes Pods

- **Pods** consist of
  - Group of containers
  - Container configurations
  - Shared storage
- Containers in a pod
  - are scheduled together
  - are guaranteed to be on the same node
- **Replicas** of pods



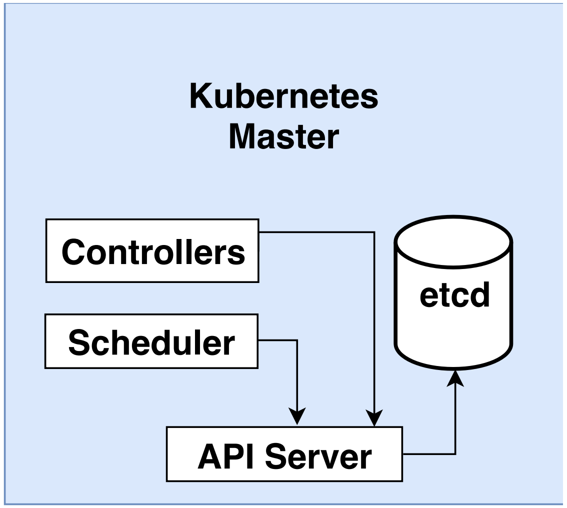
Kubernetes Nodes

- Each worker node in the cluster runs two processes:
  - **`kubelet`**: agent that communicates with the Kubernetes master (master → cluster nodes)
  - **`kube-proxy`**: makes defined services available on each node (cluster nodes → master)



Kubernetes Master

- Collection of processes managing the cluster state on a single node of the cluster

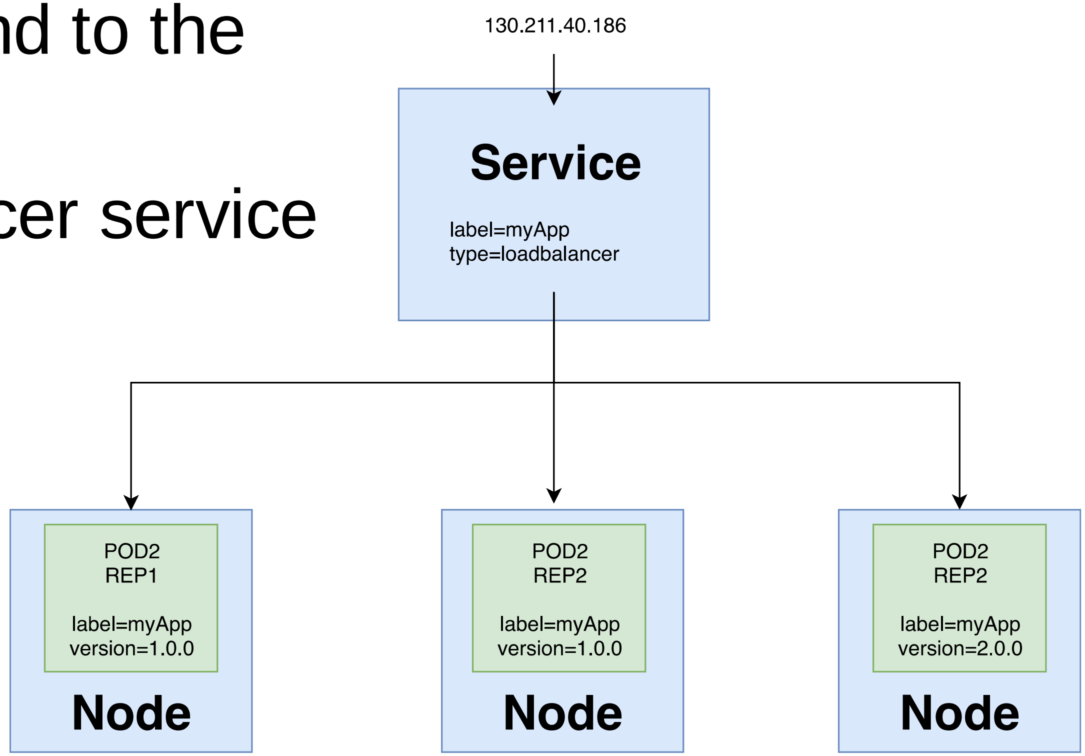


- **Controllers**, e.g. replication and scaling controllers
- **Scheduler**: places pods based on resource requirements, hardware and software constraints, data locality, deadlines…
- **etcd**: reliable distributed key-value store, used for the cluster state

Kubernetes Services

- **Services** expose functionality of pods in the cluster and to the outside
- Example: load balancer service

```
apiVersion: v1
kind: Service
metadata:
  name: myApp-service
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    # accessible externally
    nodePort: 80
    # port in Pod
    targetPort: 8080
  selector:
    app: myApp
```



Kubernetes Cluster State

- Kubernetes allows to specify the state of pods in a cluster and then manages the cluster accordingly
  - **Deployments** for stateless pods:
    - Do not preserve pod state and do not need permanent storage and IDs for networking
    - Used e.g. for web server pods (like Nginx and Apache)
  - **StatefulSets** for stateful pods:
    - Keep track of pod state by saving it to storage and pods communicate using persistent unique IDs
    - Used e.g. for database pods (like MongoDB)

Kubernetes Deployments

- Specify the **deployment** of a stateless pod, which k8s then establishes (and re-establishes)

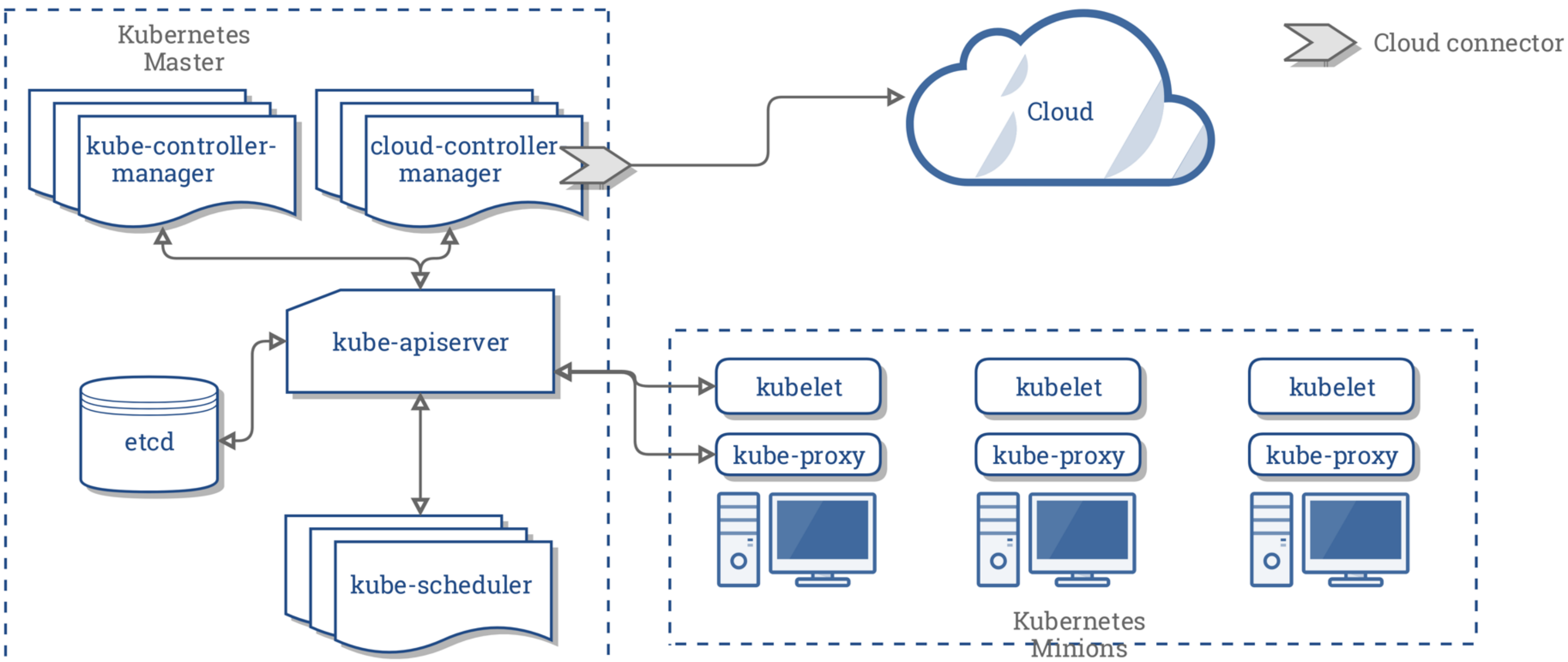
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.4
        ports:
        - containerPort: 80
```

```
$ kubectl create -f nginx-deployment.yaml
```

```
$ kubectl get deployments
```

```
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           18s
```

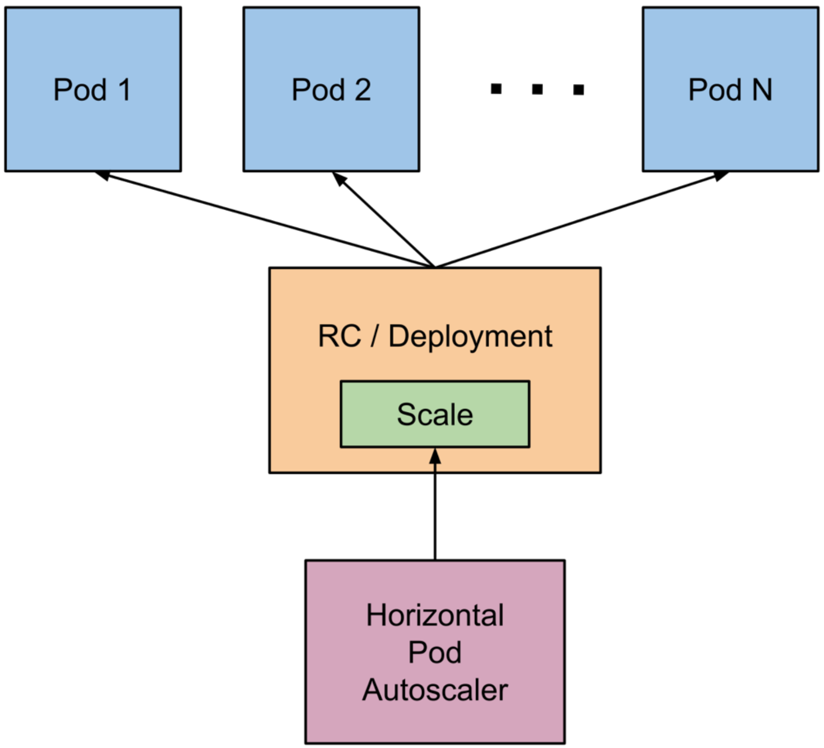
Kubernetes
Horizontal Pod Autoscaler



Kubernetes
Horizontal Pod Autoscaler

- Automatically scales the number of pods using the **replication controller**
- Based on
  - CPU utilization or custom metrics
  - Target value for the metric
- Autoscaler checks metrics regularly (e.g. every 30 sec)
- Scales the number of replicas to optimize the metric towards the target value

Kubernetes Horizontal Pod Autoscaler



Kubernetes Horizontal Pod Autoscaler

- Number of replicas is scaled by comparing an averaged current metric value to target value
- No scaling if metrics are within a tolerance (e.g. <= 0.1)

```
desiredReplicas =
  ⌈ currentReplicas * (currentMetricValue/desiredMetricValue) ⌉
```

❼This assumes linear scaling!

- Autoscaler scales to the highest number of desired replicas in a sliding window of 5 minutes
  - Quick responses to more load, but reduces *thrashing*

Sometimes Metrics are not Available

- Ignore pods that are currently being shut down
- Normally running pods with missing metrics
  - If result without these would be to scale out: assume metric to be 0
  - If result without these would be to scale in: assume metric to be 1
- Assume metric to be 0 for pods that are not yet ready

“Dampens” the scaling

Outline of Chapter 3: Containers & Container Mngt.

Summary: Containers

- Containers provide an alternative or supplement to VMs for isolating individual applications
- Containerization using OS kernel containment features (e.g. chroot, namespaces, cgroups)
- Docker adds hierarchical images, image registries, and Dockerfiles
- Container orchestration to manage distributed, replicated multi-container applications

References

[19] W. Felter, A. Ferreira, R. Rajamony, J. Rubio: “An updated performance comparison of virtual machines and Linux containers”, Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software, 2015.

- Real-world systems discussed:
  - https://linuxcontainers.org/
  - https://canonical.com/lxd
  - https://www.docker.com/
  - https://docs.docker.com/compose/
  - https://kubernetes.io/