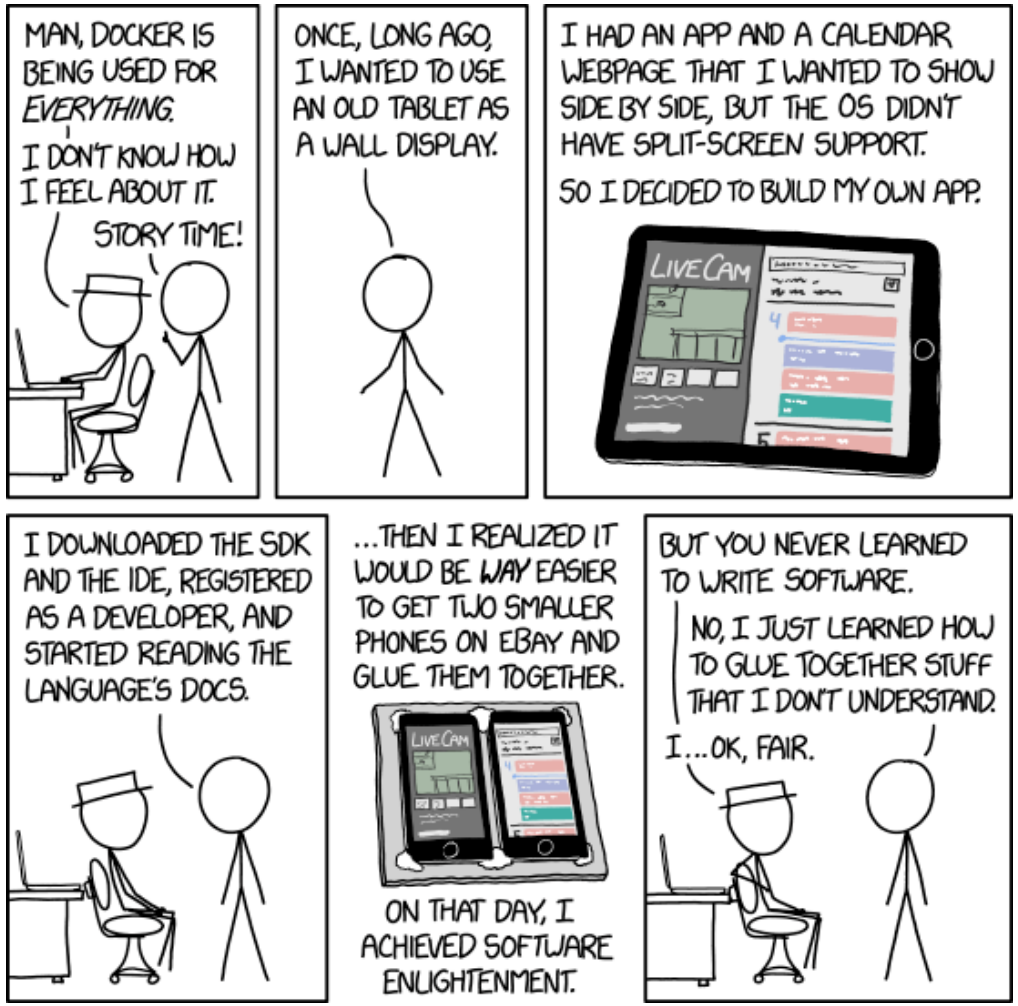

# Operation System Security

## 10 – Kapselung

SECURNITE

# OSSEC 10 - Kapselung

- Kapselung unter IT Security Aspekten
- Kapselung von Applikationen
- Erstellung von gekapselten Applikationen
- Verteilen von gekapselten Applikationen




SECURNITE


# Kapselung unter IT Security Aspekten

# Kapselung

## Trennung von Daten und Software

- **Typischer Trugschluss**: Malware in **Quarantäne** = **Gefahr gebannt**

- **Achtung**: Malware könnte bereits **Hintertüren** geöffnet und **Code** nachgeladen haben

- **Kapselung** als neuer Ansatz
  - **Schirmt** z.B. Daten **ab**
  - **Annahme**: IT-Umgebung ist grundsätzlich **feindlich** und **gefährlich**

# Kapselung

## Trennung von Daten und Software

- Man geht grundsätzlich davon aus, dass auf **„feindlichen“ Systemen** gearbeitet wird (infiziert und von Angreifern kontrolliert)
- Lösungen **schützen wertvollen**, vertrauenswürdigen **Inhalt**
- Statt einzelne Geräte zu schützen, werden alle **geschäftskritischen Anwendungen** und **Daten** in einem abgesicherten Container isoliert

# Kapselung

## Trennung von Daten und Software

- Eignet sich besonders für
  - „business critical“ **Applikationen**
  - „business critical“ **Daten**
  - Externe und interne **Kommunikation** mit Partnern, Lieferanten, Patienten, Kunden oder Mitarbeitern in Unternehmen und Behörden

- **Beispiel**: Abkapseln eines Warenwirtschaftssystems

# Containment Lösungen

«von der Stange»

- Für **browserbasierte** und **lokale Anwendungen** verfügbar

- Schutz vor **Keylogging**
  - Tastatureingaben werden vor Übergabe an den Tastaturtreiber abgefangen und verschlüsselt
  - Eingaben werden erst in der sicheren Box entschlüsselt und an die gekapselte Applikation übergeben

- Schutz vor **Auslesen** des **Arbeitsspeichers** durch «Anti Memory Scraping»
  - Externe Applikationen erhalten keinen Zugang zum Container Memory

# Containment Lösungen

«von der Stange»

- Schutz vor **Man-in-the-Middle** Attacken
  - SSL Zertifikate werden gegen integrierte Datenbanken geprüft
  - Für ungültige Zertifikate wird der Zugriff verhindert

- Schutz vor **Remote-Takeover** Attacken (Ransomware)
  - Angriff wird erkannt und Zugriff auf die Anwendung blockiert


# Kapselung von Applikationen

# Kapselung

Compile once, run anywhere

- **Ziel** kommerzieller Anbieter: Anwendung für **mehrere** (Linux-)**Distributionen vertreiben**

- **Möglichkeiten** für Linux:
  - **Nativ** für Distribution übersetzen und eigene **Repositories** betreiben
  - **Applikation** vom Betriebssystem **abkapseln**

- **Herausforderungen**
  - Hoher **Aufwand** zum Erstellen portabler „Binary-only“- Anwendungen
  - **Statisches Linken → Problem** für Applikationen mit **verschiedenen Programmiersprachen**

# Schwache Kapselung

## Beispiel Linux

- Benötigte **Bibliotheken** werden **gemeinsam** mit **Applikation** ausgeliefert
- **Bibliotheken** können z.B. mit LD_PRELOAD durch den **dynamischen Lader** *ld-linux.so* vor Ausführen des Programms geladen werden
- Programm **sieht** nur die **„richtigen“ Bibliotheken**

# Schwache Kapselung

## Nachteile

- **Aufwand** des Extrahierens der richtigen Bibliotheken
- **Abhängigkeiten** der verwendeten libc von **Kernel** und **Loader** des Zielsystems
- **Kompliziert** und **fehleranfälliger**, wenn Applikation aus **mehreren** ausführbaren **Einheiten** besteht

# Komplette Kapselung

- ermöglicht **Hardwarevirtualisierung**
- **Abhängigkeiten** werden an den Virtualisierer **ausgelagert**
- **Applikation** sieht nur ihre eigene **Ausführungsumgebung** und bringt **eigenen Kernel** mit

# Komplette Kapselung

## Nachteile

- **Overhead** der Virtualisierung
- Zusätzlicher **Speicherbedarf** durch die Auslieferung eines kompletten Betriebssystems
- „passende“ **Virtualisierungsumgebung** muss vorgehalten werden

# Container

- **Kapselung** erfolgt auf Ebene von **Namespaces** und **Control Groups** innerhalb des Linux-Kernels
- Applikationen werden in Form eines **Images** aus einem **Repository** bezogen
- **Repositories** können auch **lokal** erstellt werden
- Alle benötigten **Bibliotheken** sowie weitere erforderliche **Systemanwendungen** finden sich innerhalb des Image (**selbstkonsistent**)

# Container

- Gekapselte Anwendung verwendet **Kernel** des **Hosts**

- **Abhängigkeiten** nur auf **zwei Ebenen**:
  - Die enthaltenen **Systembibliotheken** müssen zum **Betriebssystemkernel** passen und dürfen keine dort nicht vorhandenen Systemaufrufe verwenden
  - Hostsystem muss **Container-Umgebung** (z.B. Docker) bereitstellen (Docker-Pakete im Repository, kompatibler Kernel)

# Container

- **Entwicklung** der Systembibliotheken verläuft **konservativ** → **Abhängigkeiten** vom Hostsystem werden an Container **ausgelagert**

- In Containern können auch **Entwicklungswerkzeuge** installiert sein → eine **Anwendung** kann in derselben Umgebung **übersetzt** werden, in der sie später **laufen** soll

- **Auslieferung** als (Docker-)**Image** (so „schlank“ wie möglich)

# Gruppenübung (30 Minuten)

Docker

- **Docker** gehört zu den meist genutzten **Container** Lösungen. Dennoch zeigte eine Studie, dass 80% der zertifizierten Docker-Images **schwere Sicherheitslücken** haben

- **Recherchieren** Sie
  - Welche **Kategorien** bei der **Klassifizierung** von Docker Images angewendet werden und was sie ausmacht.
  - Welche **Schichten** der Container **Stack** (Grafik) enthält.
  - Welche **Massnahmen** für das **Härten** von Docker Images getroffen werden sollten. Gehen Sie dabei auf die unterschiedlichen Schichten ein.
- **Präsentieren** Sie Ihre **Ergebnisse** (max. **5 Minuten**)

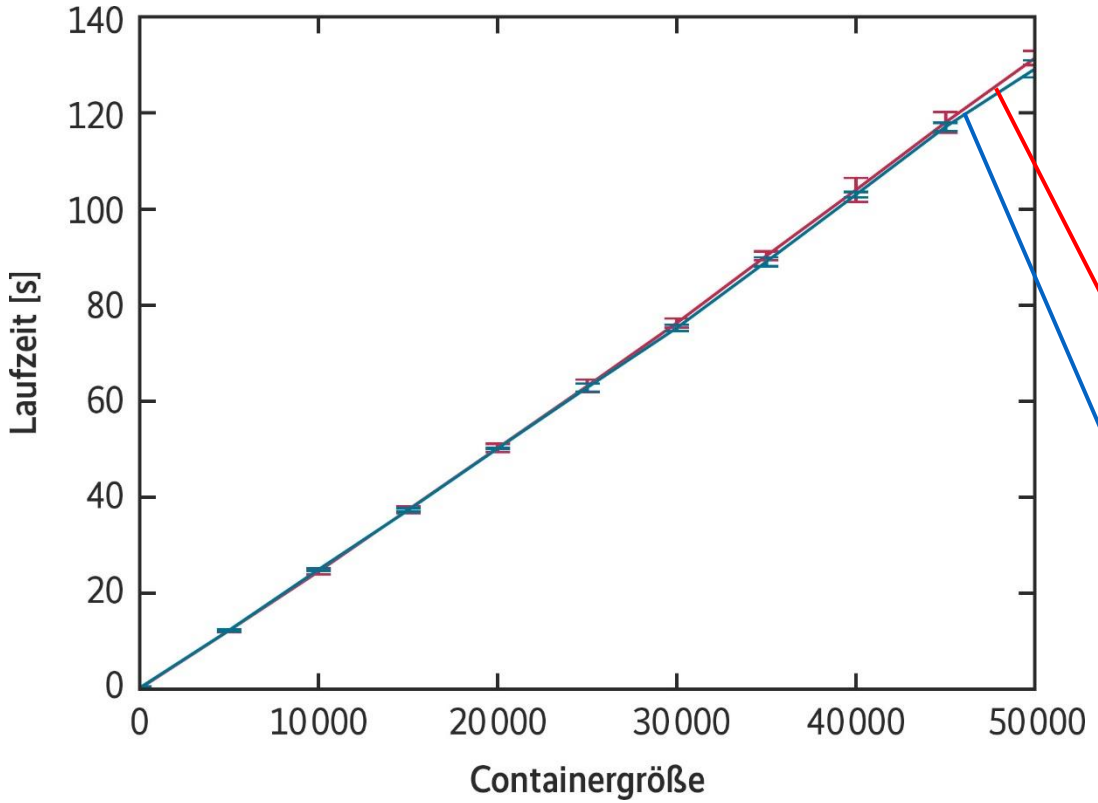
# Container

## Beispiel

- **xServer** der Karlsruher Planung Transport und Verkehr (PTV) Group
  - Server-Applikation
  - Stellt **geografische Webservices** bereit
  - Wird zur **Entwicklung** von **Routing-Anwendungen** mit Verwendungsschwerpunkt in der **Logistik** eingesetzt
  - Läuft üblicherweise auf einem **Backend-Server** (Windows oder Linux, eventuell in einem Container)
  - Erhält Anfragen über **Web-Frontend** (getrennt entwickelt)
  - Besondere **Anforderungen** an Hardware hinsichtlich **CPU-Auslastung**, **Netzwerk-Traffic** und (überwiegend lesendem) Zugriff auf umfangreiche Datenbestände

# Container

## Beispiel




https://www.heise.de/select/ix/2016/5/1462093953284843

- **Forschungsfrage**: Welchen **Performance-Overhead** zieht Einsatz im Docker-Container nach sich?
- Grafik für 50 Clients
  - Rot: **native Performance** einer Benchmark-Applikation
  - Blau: **Docker**-basierter Server
- **Ergebnis**: Kaum Unterschied

# Container

## Beispiel

- **Hintergrund**
  - **CPU-lastige Anwendungen: kein Overhead**, da Docker-Applikation nativ auf dem Host-Kernel läuft
  - **Netzwerklastige Szenarien**: **Overhead** zu erwarten, da meist mit Network Address Translation (NAT) gearbeitet wird
  - **Disk-Performance** spielt eine **geringere Rolle**, da Docker auf **Verzeichnisse** des **Hostsystems** zugreifen kann
  - **Overlay-Filesystem** hat nur **geringen Einfluss**, da sich Applikation nach dem ersten Laden im **Hauptspeicher** befindet, umfangreiche Datenbestände können **außerhalb** des Overlay-Filesystems **gelagert** werden


# Erstellung von gekapselten Applikationen

# Best Practices

## Für Docker

- **Einfluss** der Container-Anbindung auf die Applikation **individuell testen**
- **Native GUI-Ausgaben** in Containern sind nur mit einigem Aufwand zu erreichen
  - Remote-Display (X11-Socket in Container „hineinreichen“)
  - Zugang zu eigenem Grafik-Device
- **Nutzer** der Anwendungen von Details von Docker **abschirmen**
- Anwendungen in Docker Images sollten **„stateless“** sein: **keine**, für spätere Aufrufe notwendigen, **Daten** innerhalb des Containers **speichern**

SECURNITE

# Best Practices

## Für Docker

- **Bedarfs-Applikationen**:
  - Docker **Kommandozeilenargumente** an die im Container gestartete Anwendung „**durchreichen**“
  - Docker hat Angewohnheit, viele **Container-Dateien** auf Festplatte des **Hosts** zu speichern → Skript sollte nach der Ausführung „**aufräumen**“
  - Anwendung auf PC **installieren** (z.B. mit **Bordmitteln** jeder Linux-Distribution)

# Best Practices

## Für Docker

- **Allgemeine Applikationen**:
  - Image in **native Linux-Paketformate** verpacken (z.B. Debian/RPM)
  - Benötigte **Dateien automatisiert** in vorgesehene Verzeichnisse **kopieren**, werden vom Skript gefunden und Docker-Container beim Start **zur Verfügung gestellt**
  - Nach Entpacken der „Payload“ können **Skripte ausgeführt** werden
    - Docker-Image kann in **lokalen Docker-Store** des PCs geladen werden → **Kein** eigenes **Repository** für Docker-Images **nötig**
    - Post-Removal-Skript kann **Images/Container** aus lokalem Store **entfernen**, wenn Nutzer das Paket entfernen möchte

# Best Practices

## Für Docker

- **Allgemeine Applikationen:**
  - **Docker** selbst kann **als Abhängigkeit** mitgegeben werden → beim Installieren des Anwendungspakets merkt apt oder rpm, dass Docker nicht vorhanden ist und sorgt für Installation
  - **Upgrades** lassen sich über Mechanismen von RPM und Debian realisieren (neuer Docker Container wird geladen)
  - Persistente **Datenspeicherung** muss im **Host-Filesystem** erfolgen
  - Für **grosse** Docker **Images**:
    - Image **nicht** mit dem RPM- /Debian-Paket **ausliefern**
    - Stattdessen mit **Post-Install-Skript** der Pakete das Image aus **externem Docker-Repository** laden

# Best Practices

## Für Docker

- **Allgemeine Applikationen**:
  - Ordnung im Image durch **Init-Skripte**
    - **Starten/Beenden/Reload** der Applikation
    - **Auslesen** von **Konfigurationsdateien**/**Umgebungsvariablen**
    - **Achtung**: `systemd` wird aus Sicherheitsgründen geblockt → andere Init Systeme verwenden, z.B. my-init

# Best Practices

## Für Docker

- **Server-Applikationen:**
  - **Starten** der Applikation mit Starten des **Host Systems** → **Integration** in das **Init-System** des Host Systems
  - Docker beim Booten mit **passenden Argumenten** aufrufen
  - Um **Netzwerkanbindung** kümmern (Angabe des internen und externen Ports der Applikation beim Start des Containers)
  - Benötigte **Dateien** in Container **einblenden**, so dass es diese als Teil seines lokalen Dateisystems sieht
  - Auch **Shutdown** des Docker-Containers im Falle des **Herunterfahrens** des **Hosts** steuern

# Best Practices

## Für Docker

- **Server-Applikationen:**
  - **Abhängigkeit** vom Zielsystem
  - Start-, Reload- und Shutdown-Prozeduren **kapseln**
  - Dafür sorgen, dass z.B. Upstart / systemd **Skripte korrekt aufrufen** → Einbindung z.B. durch RPM- / Debian-Paket im Zuge von **Post-Install-Skripten**

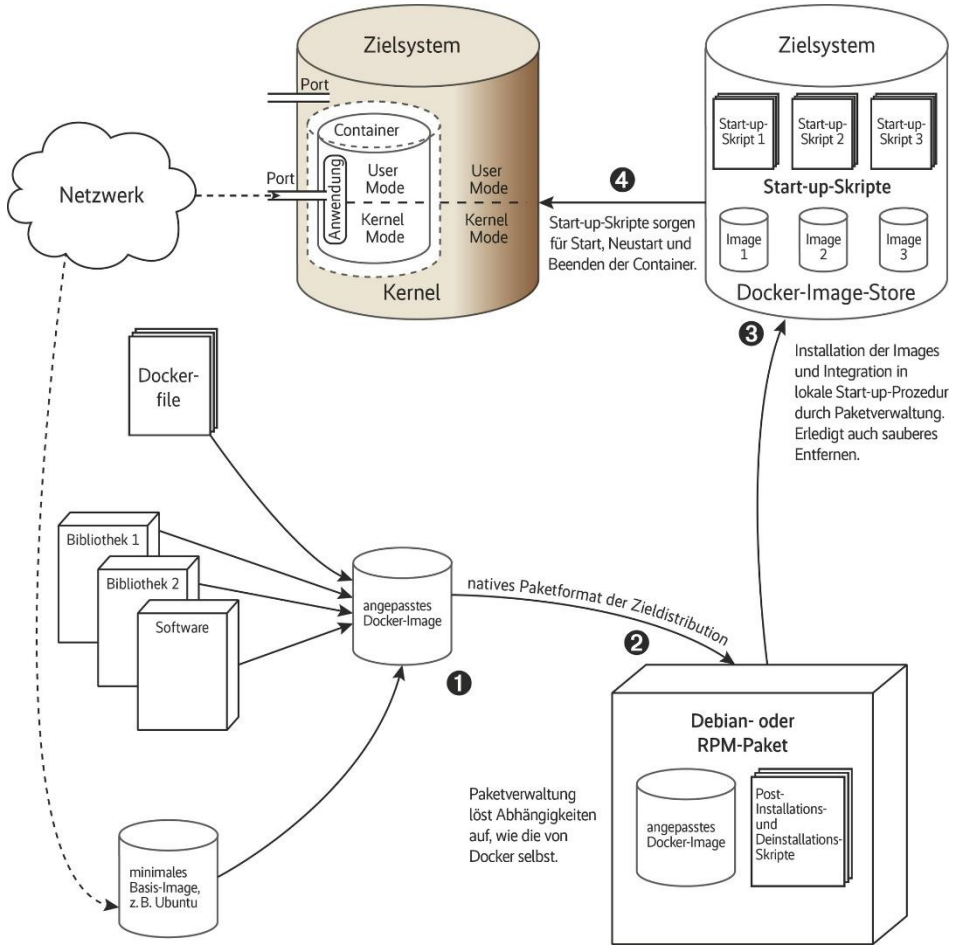
# Gruppenübung (10 Minuten)

Docker

- Container Images sollten **„stateless“** sein.
  - Diskutieren Sie **warum** das in der Realität häufig nicht umgesetzt werden kann bzw. wird
  - Gehen Sie auf konkrete **Beispiele** ein
  - Wie bewerten Sie die Tatsache, dass Images in den konkreten Fällen **nicht „stateless“** sind?
  - Welche Anwendungsfälle für **wirklich „stateless“** Images fallen Ihnen ein?

# Docker

## Ablauf




1. **Erstellung** eines Docker-Image für die Anwendung
2. **Integration** in Debian- oder RPM-Paket
3. **Einbindung** in Init-System des Hosts, **Paketverwaltung** installiert fehlende Pakete bei Bedarf nach
4. Init-System sorgt für reibungslosen **Start** und **Shutdown** des Containers

https://www.heise.de/select/ix/2016/5/1462093953284843

# Docker

## Vorteile

- Nutzer müssen lediglich einige **wenige Kommandos** zur **Installation** von Paketen kennen
- Installation ist **minimalinvasiv**
- **Abhängigkeiten** der Applikation vom Zielsystem sind auf Systemaufrufe **reduziert**
- Anwendung bringt ihre **eigenen Bibliotheken** mit
- Hervorragende **Performance** möglich

# Docker

## Herausforderungen

- **GUI-Ausgaben** Docker-basierter Anwendungen sind **aufwändig** umzusetzen

- **Geschwindigkeit –** Anwendungen im Docker-Container erreichen nicht dieselbe Geschwindigkeit wie die direkte Benutzung von Betriebssystem und Hardware. Wenn Geschwindigkeit entscheiden ist, muss das berücksichtigt werden.

- **Permanente Speicherung** - Docker zerstört alle Daten, die sich im Container befinden, nachdem der Container heruntergefahren wurde. Dauerhafte Daten müssen aus dem Container auf zusätzliche Speichermedien heruntergeladen oder durch Docker-Volumes verwaltet werden.

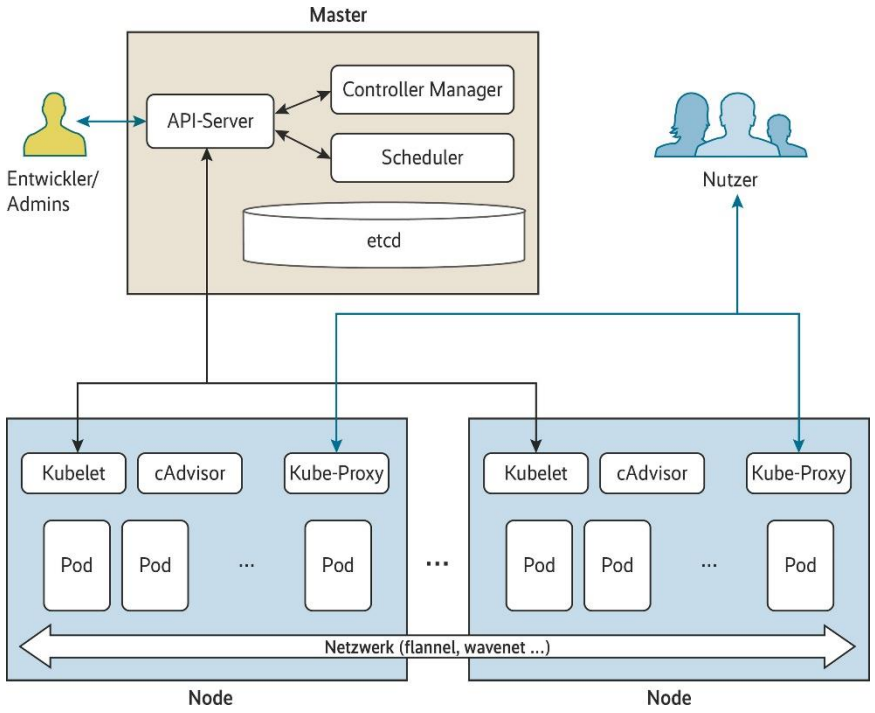

# Verteilen von gekapselten Applikationen

# Container verteilen und verwalten

- **Kubernetes**
  - Googles Container Verwaltung
  - Open source
  - Hilft, containerisierte Anwendungen auf einzelnen Knoten zu verteilen
  - Dienste werden in **Pods** oder **Deployments** zusammengefasst
  - Von aussen sehen Anwender deren **Zusammenspiel** als eine Applikation
  - Anwendungen werden in **Template-Dateien** im YAML- oder JSON-Format beschrieben
  - besteht aus **Diensten**, die auf die Hosts der Installation verteilt sind

# Kubernetes

## Master

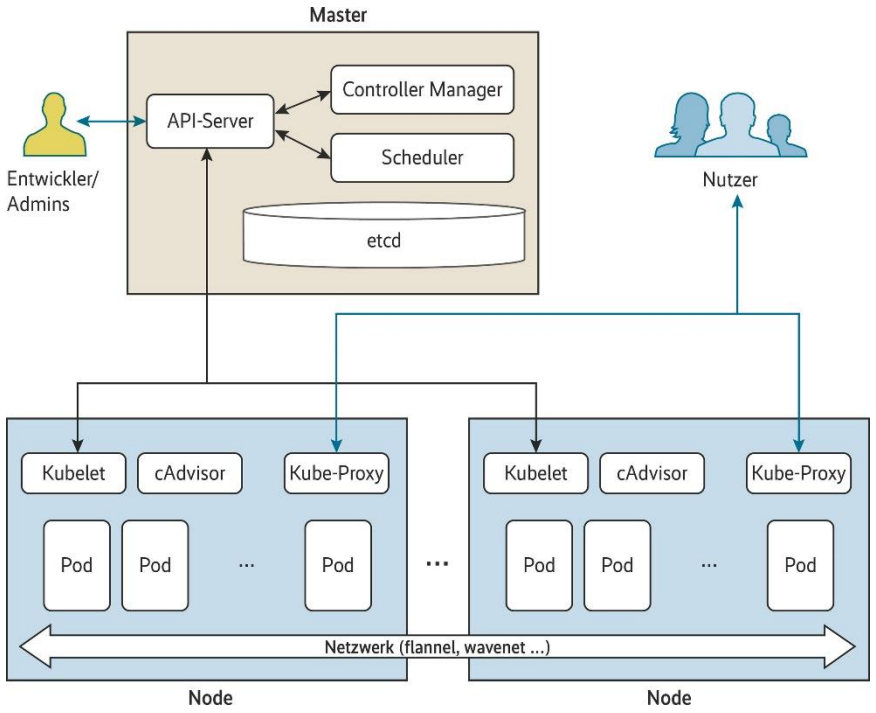



https://www.heise.de/select/ix/2017/7/1499457702012615

- **Mehrere** zentrale **Kubernetes-Dienste**
- Für den Betrieb der Plattform unbedingt **notwendig**
- Schnittstelle zum Nutzer = **API-Server**
  - von aussen **Anfragen** an Kubernetes senden
  - leitet Anfrage an „Controller Manager Service“ weiter

# Kubernetes

## Master



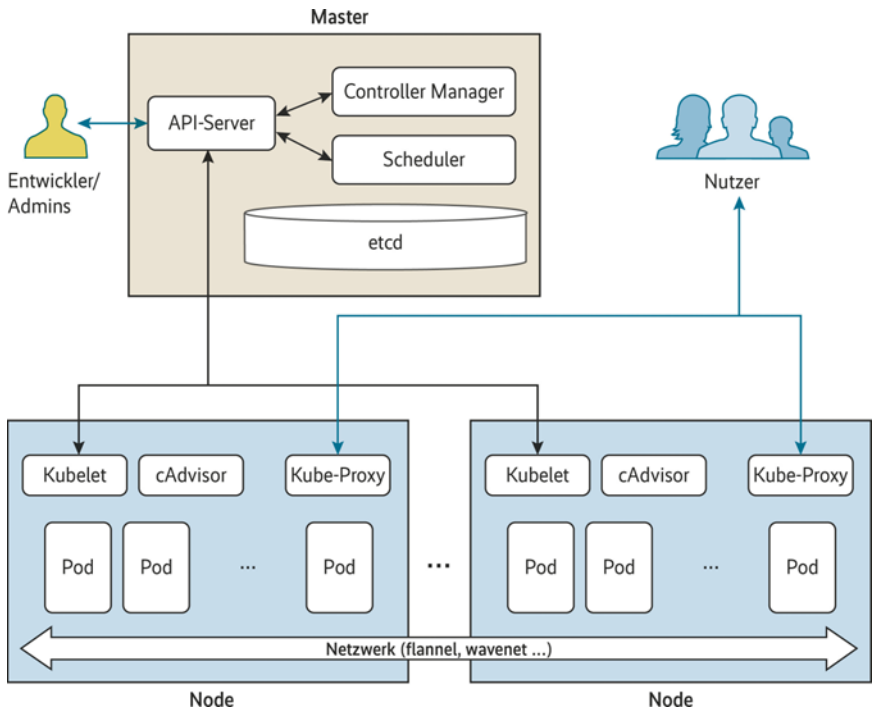

https://www.heise.de/select/ix/2017/7/1499457702012615

- **Controller Manager Service**
  - Kern Service
  - Kümmert sich um **Wartungsaufgaben**
  - Sorgt dafür, dass **Befehle ordnungsgemäss umgesetzt** werden

- **Scheduler Service**
  - Weiss, welche **Knoten** zum Setup gehören
  - Legt fest, **wo** neue Container starten

# Kubernetes

## Nodes



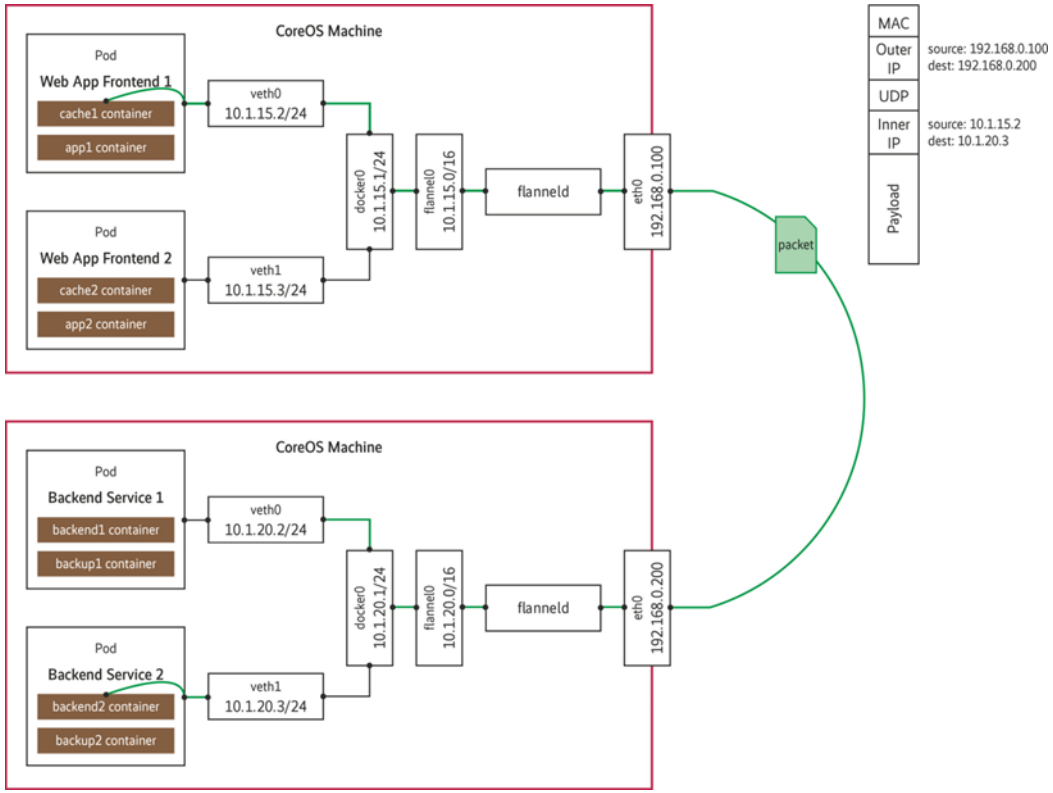

https://www.heise.de/select/ix/2017/7/1499457702012615

- Hosts, auf denen **Container** laufen

- **2 Komponenten**
  - **Kubelet:** Kubernetes-Agent, der auf dem Host die **Arbeiten** durchführt, die ihm die Master-Dienste vorgeben
  - **Network-Fabric flannel:** sorgt dafür, dass Container in einem eigenen **Subnetz** starten

# Kubernetes

## Nodes - Netzwerk

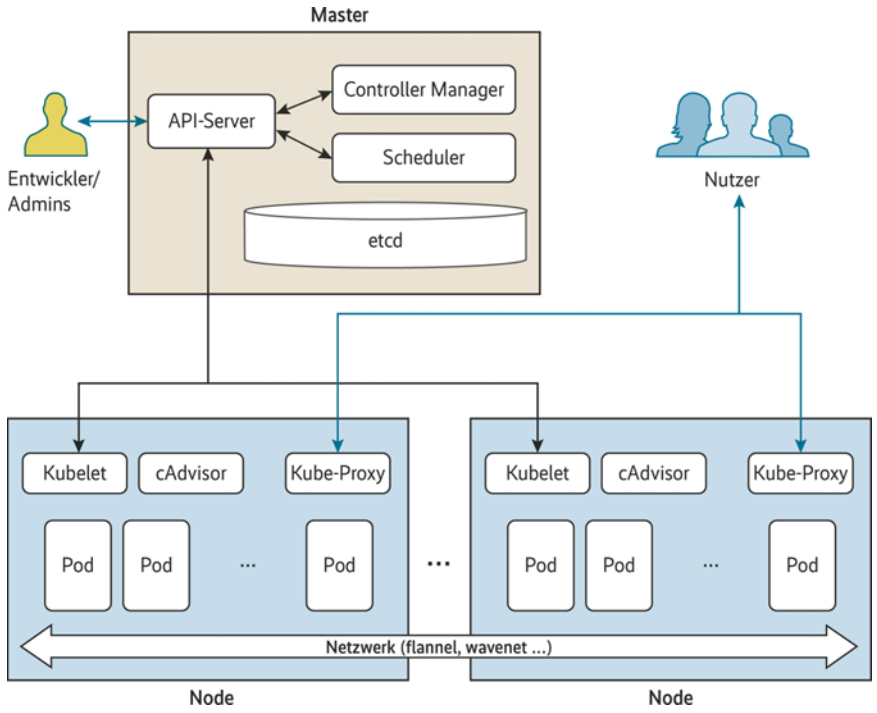



https://www.heise.de/select/ix/2017/7/1499457702012615

- Detaillierte Ansicht eines Netzwerks mit Network-Fabric **flannel**
  - Verbindung zwischen zwei **Nodes**, mit jeweils zwei **Pods**

# Kubernetes

## Nodes




https://www.heise.de/select/ix/2017/7/1499457702012615

- Nodes enthalten Kubernetes-eigene **Proxy-Server (Kube-Proxy)**
  - Stellen sicher, dass Container von aussen für Nutzer erreichbar sind
  - Integriertes Load-Balancing

# Kubernetes

## Applikationen und Pods

- **Applikation**: alle Komponenten, die zu einer in Mikroarchitektur verfassten Applikation gehören

- **Pods:**
  - kleinste **Verwaltungseinheit**
  - Beinhaltet **Container**, die die Dienste einer **Applikation** ausmachen
  - **Applikation** kann auf mehrere Pods **verteilt** sein

# Kubernetes

## Konfiguration

- **Aufwand** für das Setup eines RZ-tauglichen Clusters ist **hoch**
- **Tools** zum **teil-automatisierten Setup** verfügbar, z.B. kubeadm
- **Für Experten:** Vorgefertigte Playbooks und Rollen für definierte Setups verfügbar, z.B. für Ansible

# Docker verteilen und verwalten

- **Docker Swarm**
  - Spezielle **Betriebsart** des Docker-Daemon, in dem der sich nicht mehr nur um einen einzelnen Computer (Host oder Knoten) kümmert
  - **Zwei** Sorten von **Knoten**: **Manager** und **Arbeiter**
  - **Minimaler Schwarm** = ein Knoten, der sowohl Manager, als auch Arbeiterrolle in sich vereint
  - Mehrere Manager & Arbeiter ermöglichen **Redundanz**
  - Stellt Funktionen bereit, um **Services** und **Stacks** auf die **Knoten** eines **Clusters** zu **verteilen**
  - Stellt **virtuelle Netze** zur Verfügung
  - Bringt **keinen** gemeinsamen **Datenspeicher** mit

# Docker Swarm

## Manager

- Mindestens einer, übernimmt **Koordination**
- Ihm stehen **beliebig viele Arbeiterknoten** zur Seite
- Entscheidet, **welcher Knoten welche Aufgabe** übernimmt
- Achtet darauf, dass **Anforderungen** erfüllt werden
- **Startet** und **beendet** Container auf Knoten
- Kann selbst **Arbeitsaufträge** annehmen

SECURNITE

# Docker Swarm

## Container

- Sollte möglichst nur **einen Prozess** enthalten
- Bekommt **standardmässig** eine **eigene IP-Adresse** aus privatem Netz → Eingehende **Netzwerkanfragen** laufen meist **über Host**
- Daten werden in **Volumes** gespeichert (auf dem Docker-Host abgelegte **Verzeichnisbäume**)

# Docker Swarm

## Tasks und Services

- **Task** = Laufender Container

- **Service**
  - **Task** + **Beschreibung**, wie **häufig** ein Schwarm sie ausführen soll
  - Kann **global** sein (**jeder** Knoten im Schwarm führt **einen** Task aus (Manager und Arbeiter))
  - Kann **repliziert** sein (Beschreibung bestimmt, **wie viele** Knoten im Schwarm **einen** Task des Service ausführen)
  - **Kleinste Einheit** aus Sicht eines Schwarms

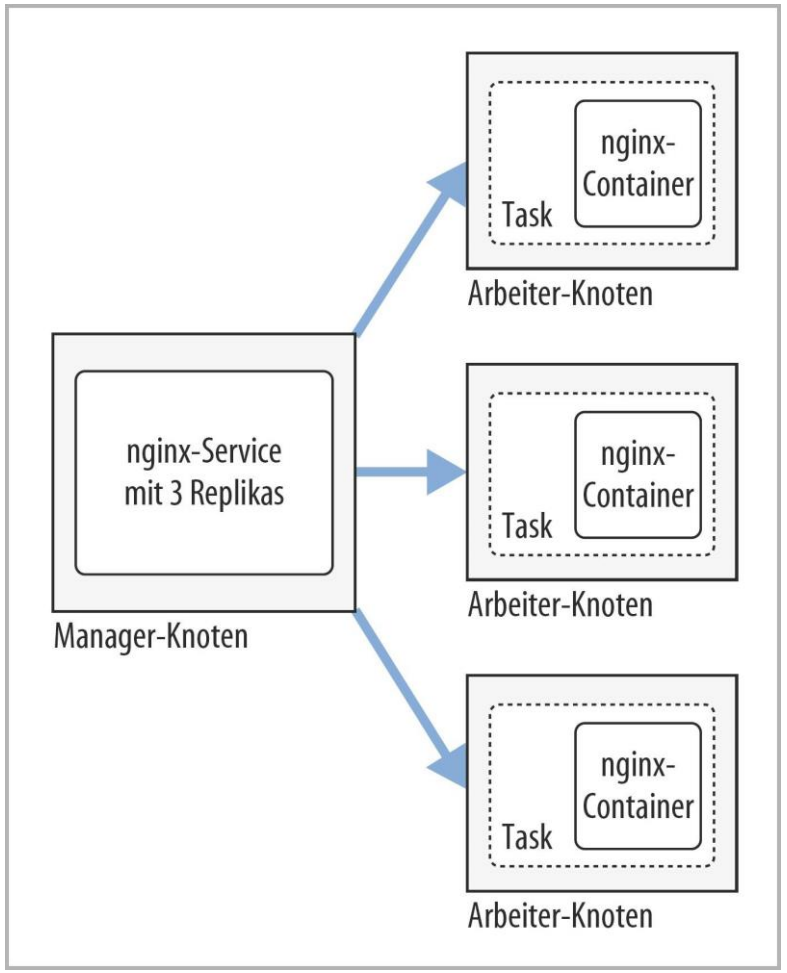
# Docker Swarm

## Stack

- **Mehrere Services** (z.B. Webserver, Datenbank, Anwendungsserver)
- In sich **geschlossene** Anwendung
- Vergleichbar mit in Docker-Compose zusammengestellten Arrangement
- **Beschreibung** enthält Hinweise im Hinblick auf die **Anzahl** der **Container-Instanzen**, die der Schwarm ausführen soll

# Docker Swarm

## Beispiel

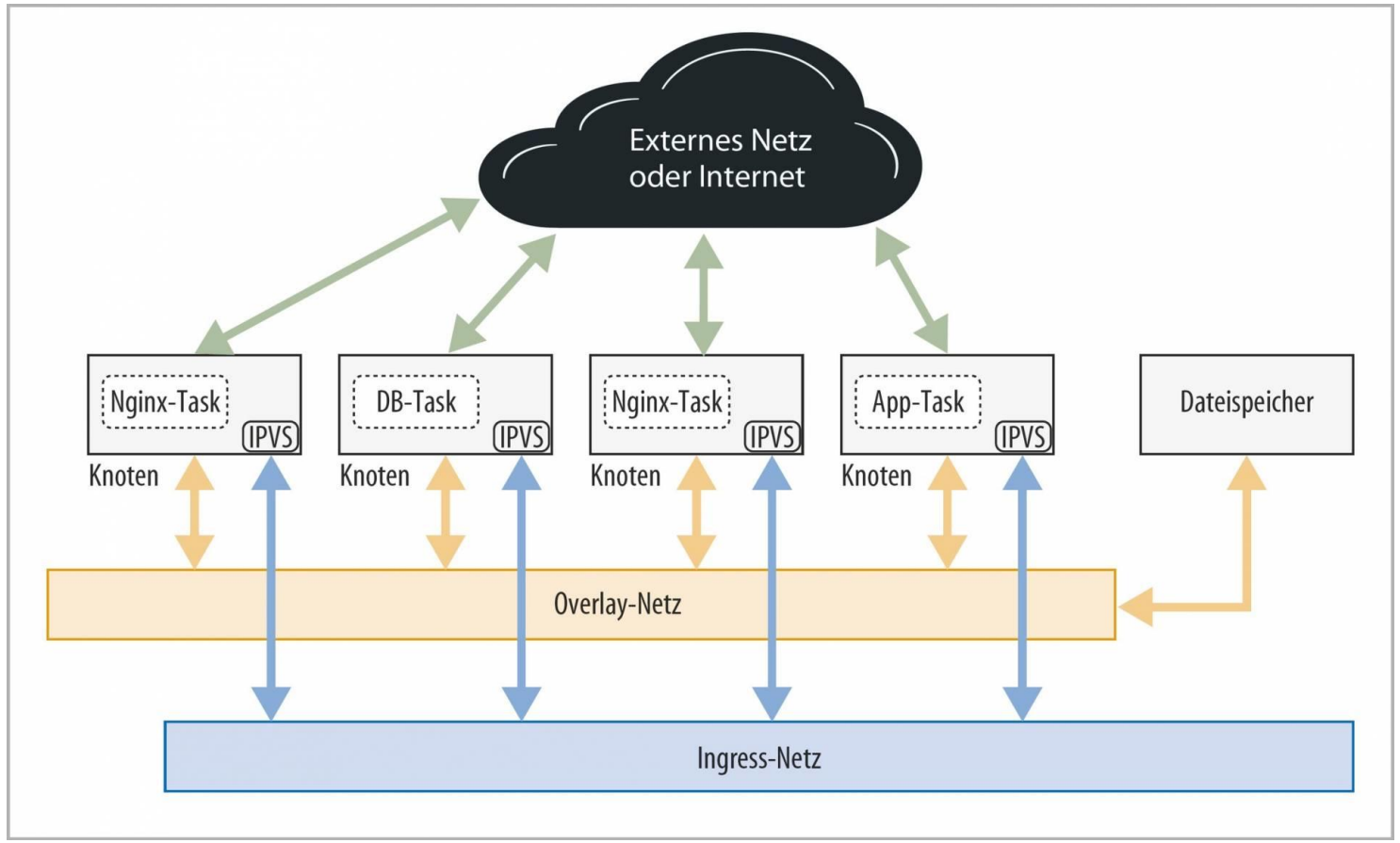



- Beim Swarm-Manager **drei Nginx-Webserver anfordern**
- Manager **verteilt** diese **auf freie Knoten**, indem er dort als **Task** einen Nginx-Container startet

https://www.heise.de/ratgeber/Docker-Swarm-Container-verteilen-und-verwalten-4313862.html

# Docker Swarm

## Virtuelle Netze




https://www.heise.de/ratgeber/Docker-Swarm-Container-verteilen-und-verwalten-4313862.html

- Mehrere, auf einem System laufende Container können **kommunizieren**
- **Beispiel**: Webanwendung in einem Container, Datenbank in einem anderen

# Docker Swarm

## Virtuelle Netze

- **Ingress-Netzwerk:**
  - vermittelt **Netzwerkanwendungen** im Schwarm an **Aussenwelt**
  - **Beispiel**: Schwarm stellt **nginx-Webserver** als Service auf **Port 8008** bereit → ist an jeder externen **IP-Adresse** aller Knoten auf diesem Port **erreichbar**
  - Kombinierbar mit **Load Balancern**

# Docker Swarm

## Virtuelle Netze

- **Overlay-Netz:**
  - Zur **internen Kommunikation** für Services, die gemeinsam **eine Anwendung** bilden
  - **Stacks** richten oft **eigene** Overlay-Netze ein → **mehrere Stacks** in einem Schwarm **kommen** sich **nicht** in die **Quere**
  - Daten werden standardmässig **unverschlüsselt** übertragen

# Docker Swarm

## Konfiguration

- Initiale **Konfiguration**
  - Wird **beim Errichten** eines Schwarms **im Hintergrund** gemacht
  - **Knoten** werden miteinander **verknüpft**
  - **Zertifikate** werden **generiert**
  - **Netzwerkschnittstellen** und **Brücken** werden **erstellt**
  - IPv4-**Firewall-Regeln** werden **eingerichtet** und später beim Freigeben von Ports für Services **erweitert**

- **Laufzeit Konfiguration**
  - **Einfach** hoch und runter **skalierbar**

# Docker Swarm

## Inbetriebnahme

- **Inbetriebnahme** des Swarm-Modus mit:

```
docker swarm init --advertise-addr 2xx.2x.2xx.x
```

- Richtet System als **Manager** ein

- Gibt **Kommandozeilenbefehl** aus, um dem Schwarm einen **Knoten** auf einem anderen PC hinzuzufügen

- **IP-Adresse:** unter welcher seiner Adressen gibt der Knoten Dienste für die Allgemeinheit frei (Eintrittspunkt für **Ingress-Netzwerk**)

# Docker Swarm

## Inbetriebnahme

- **Hinzufügen** von Knoten mit:

```
docker swarm join --token <Token> 2xx.2x.2xx.x:2377 --
                  advertise-addr 1xx.2xx.x.3
```

- Option --advertise-addr ist für **NAT** Adressen gedacht

- **Firewall** muss **Ports** freischalten und bei verschlüsselten Overlay-Netzen Encapsulating Security Payload (**ESP**) Pakete passieren lassen

# Gruppenübung (15 Minuten)

Containerorchestrierung

- **Kubernetes** und **Docker Swarm** sind zwei der wichtigsten Akteure in der Containerorchestrierung. Beide bieten jeweils **Vor-** und **Nachteile**. Stellen Sie eine **tabellarische Übersicht** über die Vor- und Nachteile zusammen. Gehen Sie z.B. auf diese Aspekte ein
  - Administrationsmöglichkeiten (Überwachung, Verwaltung)
  - Skalierbarkeit und Verfügbarkeit

- **Präsentieren** Sie Ihr Ergebnis (**max. 5 Minuten**)

# Lastausgleich

- Load Balancing (Lastausgleich) ist eine Technik zum gleichmäßigen **Verteilen** von **Workloads** auf Servern oder andere Rechenressourcen, um die **Effizienz**, **Zuverlässigkeit** und **Kapazität** des Netzwerks zu optimieren

- Umfangreiche Berechnungen oder große Mengen von Anfragen werden auf mehrere **parallel** arbeitende Systeme **verteilt** mit dem Ziel, ihre gesamte Verarbeitung effizienter zu gestalten

- Neben der Maximierung der Netzwerkkapazität und -leistung bietet Load Balancing auch eine **Ausfallsicherheit**. Fällt ein Node aus, leitet ein Load Balancer seine Workloads sofort auf ein anders System um und fängt damit die Auswirkungen auf die Endbenutzer ab

# Lastausgleich

- **Kubernetes**: Pods werden über einen Dienst exponiert, der als Load Balancer innerhalb des Clusters verwendet werden kann. Hierfür muss ein Ingress-Controller als Dienst manuell konfiguriert werden um den Lastenausgleich zu ermöglichen

- **Swarm**: Load Balancing ist in Docker Swarm automatisch integriert. Alle Container in einer Gruppe interagieren mit einem gemeinsamen Netzwerk, das Verbindungen von jedem Knoten zu jedem Container ermöglicht.

# Lastausgleich Swarm

- Docker Swarm **Load Balancer** läuft auf **jedem Knoten** und kann die Lastverteilung für alle Container auf allen Hosts des Clusters übernehmen

- Alle Container in einer Gruppe interagieren mit einem gemeinsamen Netzwerk, das Verbindungen von jedem Knoten zu jedem Container ermöglicht.

- Diese Funktion ist in Kubernetes nicht integriert und muss manuell hinzugefügt/ konfiguriert werden