



Fakultät Informatik

Softwaretechnik und Medieninformatik

Bachelorthesis

Evaluierung verschiedener Container-Technologien

Corvin Schapöhler

751301

Semester 2018

Firma: NovaTec GmbH

Betreuer: Dipl.-Ing. Matthias Haeussler

Erstprüfer: Prof. Dr.-Ing. Dipl.-Inform. Kai Warendorf

Zweitprüfer: Prof. Dr. Dipl.-Inform. Dominik Schoop

Ehrenwörtliche Erklärung

Hiermit versichere ich, Corvin Schapöhler, dass ich die vorliegende Bachelorarbeit mit dem Titel „Evaluierung verschiedener Container-Technologien“ selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebene Literatur und Hilfsmittel verwendet habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht in gleicher oder anderer Form als Prüfungsleistung vorgelegt worden.

Stuttgart, 15. Mai 2018

Ort, Datum

Corvin Schapöhler

Kurzfassung

Stichwörter: *Container, Docker, Cloud Native, OCI, Linux, rkt, Evaluation*

Abstract

Keywords: *Container, Docker, Cloud Native, OCI, Linux, rkt, Evaluation*

Inhaltsverzeichnis

Ehrenwörtliche Erklärung	I
Kurzfassung	II
Abstract	II
1 Einleitung	1
1.1 Motivation	1
1.2 Aufbau der Arbeit	1
2 Grundlagen	3
2.1 Standards	4
2.2 Funktionsweise	6
2.3 Eigene Implementierung	10
3 Geschichte	17
3.1 Container-Engines	17
3.2 Aktuelle Probleme	21
3.3 Zusammenfassung	22
4 Container-Runtimes	24
4.1 Vorgehen	24
4.2 Docker Stack	25
5 Letzte Inhaltsseite	27
Abbildungsverzeichnis	A
Tabellenverzeichnis	B
Listings	C
Glossar	D

Akronyme

E

Literatur

F

1 Einleitung

1.1 Motivation

Die Welt wird immer stärker vernetzt. Durch den Drang, Anwendungen für viele Nutzer zugänglich zu machen besteht der Bedarf an Cloud-Diensten wie Amazon Web Services. Eine dabei immer wieder auftretende Schwierigkeit ist es, die Skalierbarkeit der Services zu gewährleisten. Selbst wenn viele Nutzer gleichzeitig auf einen Service zugreifen, darf dieser nicht unter der Last zusammenbrechen.

Bis vor einigen Jahren wurde diese Skalierbarkeit durch Virtuelle Maschinen (VMs) gewährleistet. Doch neben großem Konfigurationsaufwand haben VMs auch einen großen Footprint und sind für viele Anwendungen zu ineffizient. Eine Lösung für dieses Problem stellen Container dar.

Diese Arbeit gibt einen Einblick in das Thema Container-Virtualisierung und beantwortet die Fragen, wie sich Docker als führende Technologie durchsetzen konnte, wie sich andere Technologien im Vergleich zu Docker schlagen und was die Zukunft in Form von Serverless-Technologien mit Bezug zu Containern bereithält.

1.2 Aufbau der Arbeit

Zu Beginn der Arbeit werden benötigte Grundlagen der Technologie erläutert. Dabei werden bestehende Container-Standards betrachtet und alle benötigten Kernel-Funktionen erklärt, die in Container-Runtimes Verwendung finden. Um einen besseren Einblick in die Technologie zu geben wird gezeigt, wie man mit Bash-Befehlen ohne Container-Runtime einen Prozess von einem Host-OS

trennt. Dabei wird darauf eingegangen, wie eine eigene Dateihierarchie isoliert werden kann, wie Namespaces dabei helfen Funktionen des Linux-Kernels zu virtualisieren und wie der isolierte Prozess sicherer ausgeführt werden kann.

Im Anschluss wird die Frage beantwortet, wie Docker die populärste Container-Technologie wurde. Dazu wird die Geschichte betrachtet und Probleme einzelner Technologien aufgezeigt. Zudem wird gezeigt, wie Innovation durch die Vereinfachung von Schnittstellen entstehen kann.

Vergleich Docker vs the World

Aktuelle Probleme, Orchestrierung

Serverless, wenn Zeit reicht

2 Grundlagen

Container werden häufig als leichtgewichtige VMs beschrieben. Dies ist allerdings nicht richtig. Wie in Abbildung 1 zu erkennen, virtualisieren Container kein vollständiges Betriebssystem (*Operating System*) (OS), sondern lediglich das benötigte Dateisystem. Dabei wird der Kernel des Hosts nicht virtualisiert, sondern mitverwendet. Dies macht Container deutlich leichtgewichtiger als VMs, isoliert allerdings weniger umfangreich als diese.

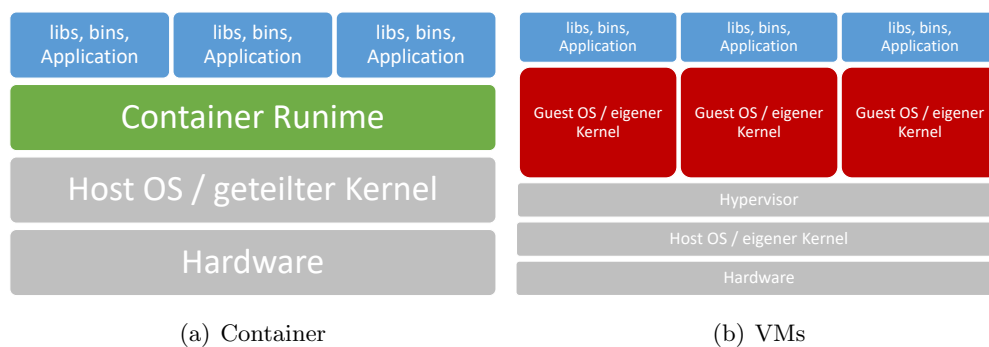


Abbildung 1: Container Isolation im Vergleich zu VMs

Dieses Kapitel behandelt alle benötigten Grundlagen, die zur Isolation eines Prozesses benötigt werden. Es werden vorhandene Standards wie die Open Container Initiative (OCI) und benötigte Systemcalls wie Change Root (chroot) näher erläutert. Zudem wird beschrieben, wie die Isolation, die Container bieten, durch Systemmittel des Linux-Kernels selber erreicht werden kann.

2.1 Standards

Durch die immer größere Verwendung von Containern und die Verbreitung verschiedener Container-Runtimes ist die Standardisierung eine wichtige Aufgabe. Folgend werden Standardisierungsprojekte aufgezählt, die bestehenden Spezifikationen erläutert und aktuelle Aufgaben der Projekte näher betrachtet.

2.1.1 App Container

App Container (appc) ist ein Standard, der viele Aspekte innerhalb der Container-Landschaft behandelt. Dabei lag die Hauptaufgabe darin, eine Laufzeitumgebung wie auch das Image-Format und die Verbreitung von Images zu spezifizieren. Seit 2016 wird das Projekt nicht mehr aktiv weiterentwickelt, da mit der Gründung der OCI ein größeres Standardisierungsprojekt entstand. Bestandteile der appc wurden von der OCI übernommen und dienen als Vorlage für die Spezifikation dieser.

2.1.2 Open Container Initiative

Die OCI ist eine Initiative, die seit 2015 unter der Linux Foundation agiert. Das Ziel der OCI ist es, einen offenen Standard für Container zu schaffen, so dass die Wahl der Container-Laufzeitumgebung nicht mehr zu Inkompatibilität führt. Dabei liegt der Fokus auf eine einfache, schlanke Implementierung (Open Container Initiative, 2018).

Die OCI arbeitet aktuell an zwei Spezifikationen. Die runtime-spec standardisiert die Laufzeitumgebung von Containern. Dabei wird festgelegt, welche Konfiguration, Prinzipien und Schnittstellen Laufzeitumgebungen stellen müssen. Um die Umsetzung der runtime-spec zu fördern, stellt die OCI eine beispielhafte Implementierung durch runC. Das zweite Projekt der OCI ist die image-spec.

Dieses versucht einen Standard für Images zu definieren. Dabei plant die OCI nicht, vorhandene Image-Formate zu ersetzen, sondern auf diesen aufzubauen und sie zu erweitern (Open Container Initiative, 2018).

Wie in Tabelle 1 zu sehen, wurden einige Konzepte des appc-Projekts in die OCI übernommen. Vor allem die Image-Spezifikation wurde durch die Mitarbeit ehemaliger appc-Maintainer gefördert. Allerdings sind einige Projekte noch nicht übernommen worden. So gibt es keine OCI Spezifikation für die Verbreitung von Images, eines der meistgenutzten Features verschiedener Container-Runtimes. Um die Weiterentwicklung an solchen Projekten zu fördern wurden einige in die Cloud Native Computing Foundation (CNCF) übernommen (Polvi, 2015).

	Standard		Container Runtime	
	OCI	appc	Docker	rkt
Container Image	✗	✓	OCI image-spec	appc Image Format
Image Verbreitung	✗	✓	Docker Registry	appc Discovery Spec
Lokales Speicherformat	✓	✗	keine Spezifikation	keine Spezifikation
Runtime	✓	✓	runC	appc runtime Spec

Tabelle 1: Standards OCI und AppC im Vergleich (Polvi, 2015)

2.1.3 Cloud Native Computing Foundation

Die CNCF beschäftigt sich im Gegensatz zur OCI nicht nur mit Containern, sondern der kompletten Cloud-Native-Landschaft (CNCF, 2018). Projekte wie Kubernetes (K8) und Prometheus werden durch die CNCF weiterentwickelt und publiziert. Da der Cloud-Native Entwicklungsprozess von Containern getragen wird, spielen Technologien wie containerd und rkt eine entscheidende Rolle für die CNCF und sind ein großer Teil der Cloud-Native-Landscape, wie in Abbildung 2 gezeigt. Neben Container-Runtimes beinhaltet die CNCF auch Projekte zur Orchestrierung von Containern, Logging und Monitoring dieser, wie auch Spezifikationen, zum Beispiel die TUF, eine Spezifikation die standardisiert, wie Softwarepakete upgedatet werden sollen (CNCF, 2017).

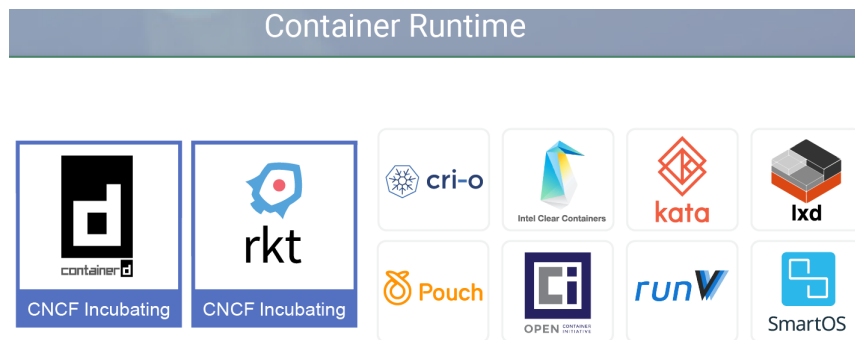


Abbildung 2: CNCF Container Runtime Landschaft (CNCF, 2018)

2.2 Funktionsweise

Container isolieren einzelne Prozesse durch verschiedene Kernel-Technologien, die im Folgenden erklärt werden sollen.

2.2.1 Change Root

Chroot ist ein Unix Systemaufruf, der es erlaubt einen Prozess in einem anderen Wurzelverzeichnis auszuführen (McGrath, 2017). Daraus folgt, dass der Prozess in einer eigenen Verzeichnisstruktur arbeitet und keine Dateien des Host-OS ändern kann. Chroot erlaubt somit die Isolierung des Dateisystems, die Container nutzen.

2.2.2 Control Groups

Control groups (cgroups) dienen dazu, Systemressourcen für einzelne Prozesse zu limitieren. Cgroups sind anders als chroot kein Unix-Feature sondern Teil des Linux-Kernels. Im OS sind cgroups als Dateihierarchie repräsentiert. Das gesamte cgroup-Dateisystem ist unter `/sys/fs/cgroup/` zu finden. Cgroups stellen zur Steuerung verschiedene Controller zur Verfügung.

Controller	Ressource
io	Zugriff und Nutzung von Block Geräten wie Festplatten
memory	Monitoring und Beschränken des Arbeitsspeichers
pids	Limitierung der Anzahl an Unterprozessen
perf_event	Erlaubt Performance Monitoring der Prozesse
rdma	Zugriffe über RDMA limitieren oder sperren
cpu	CPU-Zyklen und maximale CPU-Bandwidth

Tabelle 2: Cgroups-Controller und deren Verwendung (S. H. M. Kerrisk, 2018)

2.2.3 Namespaces

Namespaces abstrahieren einzelne Bereiche des OS. Sie werden genutzt, um globale Ressourcen zu virtualisieren. Ein Namespace kapselt dabei einzelne Ressourcen. Veränderungen an diesen sind für alle Prozesse innerhalb desselben Namespaces sichtbar, allerdings außerhalb dieses unsichtbar (Biederman, 2017).

Namespace	Ressource
Cgroup	Cgroup-Dateisystem
IPC	System V IPC, POSIX Nachrichten
Network	Netzwerk Geräte, Stacks, Ports, ...
Mount	Mount Punkte
PID	Prozess IDs
User	Nutzer und Gruppen IDs
UTS	Hostnamen und Domännennamen

Tabelle 3: Linux Namespaces und verbundene Ressourcen (Biederman, 2017)

2.2.4 Mounting

Durch die Isolation eines Prozesses und der Bedingung, das Container unveränderlich sein sollen, stellt sich die Frage, wie man Containern dynamische Inhalte aus dem Host-System zur Verfügung stellt. Dies ist vor allem wichtig, wenn bei Veränderung der Umgebung nicht den Container neu gestartet werden soll. Sollte zum Beispiel eine neue Datei durch einen Webserver zur Verfügung gestellt werden, möchte man nicht den Container neu starten. Die Lösung dieses Problems ist der Unix-Systembefehl `mount`.

Mit diesem Befehl wird eine beliebige Dateihierarchie an eine andere Stelle des Dateibaums angeheftet. Durch dieses vorgehen kann man Ordner vom Host-System für das mit `chroot` isolierte Dateisystem des Containers zugänglich machen. Dabei ist zu beachten, dass es sich bei dem gemounteten Ordner nicht um einen symbolischen Link handelt. Diese könnten durch den Aufruf von `chroot` nicht mehr aufgelöst werden, wie in Abbildung 3 zu sehen.

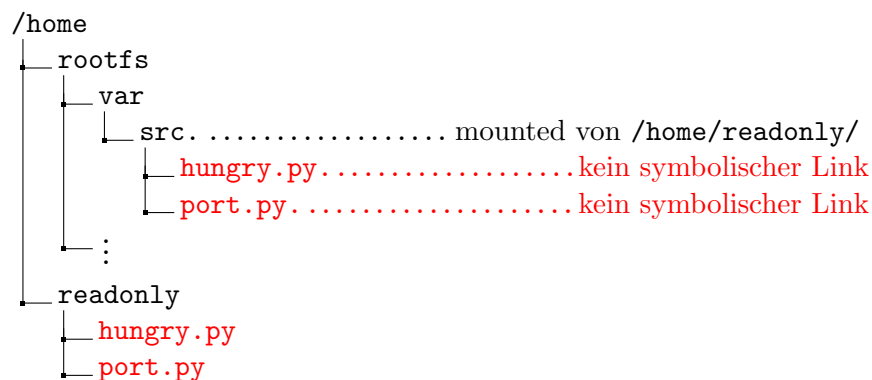


Abbildung 3: Auszug aus Dateisystem mit gemounteten Dateien

2.2.5 Netzwerk

Einen weiteren Aspekt, den Container vom Host-OS isolieren ist das Netzwerk. Dabei kommen virtuelle Ethernet-Adapter zum Einsatz. Diese erlauben es, ein unabhängiges Netzwerk zu erzeugen. Ein Adapter der virtuellen Ethernet-Verbindung wird dabei der Process Identifier (PID) des Containers zugewiesen, das andere dem Host. Zusätzlich wird der Network-Namespace genutzt um eine vollständige Isolation des Netzwerks zu erhalten.

2.2.6 Sicherheit

"Docker is about running random code downloaded from the Internet and running it as root"

—Dan Walsh (Red Hat)

Container haben ein großes Problem. Alle genannten Kernel-Features müssen als Nutzer `root` ausgeführt werden. Dadurch haben die gestarteten Prozesse häufig Berechtigungen, die es erlauben würden, aus der Isolierung des Containers auszubrechen. Um dies zu verhindern, können verschiedene Sicherheitskonzepte verwendet werden.

Das leichteste dieser Konzepte sind Capabilities. Jedem Prozess, sowie jeder Datei kann eine Liste an Capabilities zugeordnet oder genommen werden. Dabei können einzelnen Dateien beispielsweise die Rechte genommen werden, auf Port 80 zu hören. Auch viele Systemaufrufe können über Capabilities gewährt oder verwehrt werden. Ein anderes Konzept ist die Implementation eines *Mandatory Access Control*-Systems wie SELinux oder AppArmor. Diese Implementationen sind granularer als Capabilities, allerdings mit einem höheren Konfigurationsaufwand verbunden.

Capability	Systemaufruf	Erklärung
CAP_CHOWN	chown	Ändern des Owners einer Datei
CAP_KILL	kill	Beenden eines Prozesses
CAP_CHROOT	chroot	Wechseln des Root Directories
CAP_NET_BIND_SERVICE		Binden eines Prozesses auf Ports <1024
CAP_SYS_TIME	stime, settimeofday	Setzen der Systemzeit
CAP_SYS_ADMIN	mount, umount, setns, pipe, syslog, ...	Alles, was in keine andere Kategorie passt

Tabelle 4: Einige Capabilities (M. Kerrisk, 2018)

2.2.7 Container unter Windows

Viele Kernel-Features des Linux-Kernels erlauben eine Isolation und wurden teilweise spezifisch für diese entwickelt (Biederman, 2017). Seit 2016 können spezifisch Docker-Container auch unter Windows genutzt werden. Dabei trennt Microsoft Container in zwei verschiedene Isolationen auf.

Windows Server Container 2016 sind ein nativer Windows Ansatz zur Isolation eines Prozesses. Dabei werden, wie bereits unter Linux Systemen, verschiedene Kernel-Technologien verwendet, um einen einzelnen Prozess zu isolieren (siehe Abbildung 4). Der andere Ansatz sind Hyper-V Container. Diese sind eher VMs als Containers. Dabei ist der größte Unterschied, das Hyper-V Container eine minimale Installation eines Windows Betriebssystems nutzen, um auf diesem einzelne Isolationen zu erstellen, während Windows Server Container ein gemeinsames OS nutzen, nämlich Windows Server.

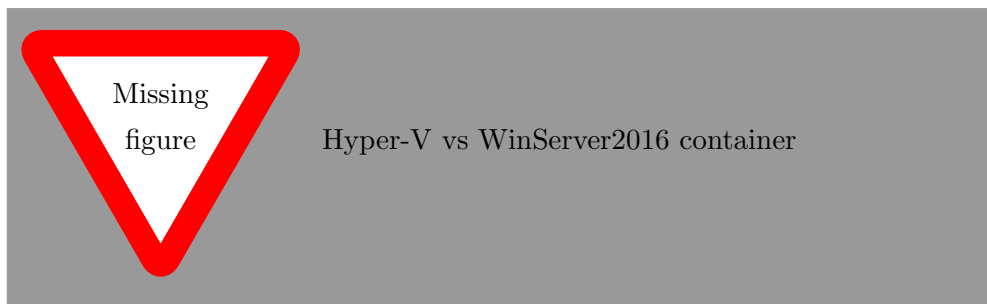


Abbildung 4: Unterschiede der Isolation bei Containern unter Windows

2.3 Eigene Implementierung

Um die in Abschnitt 2.2 erläuterten Kernel-Features näher zu beleuchten wird folgend gezeigt, wie ein Prozess isoliert vom Host-System ausführen kann. Dabei wird darauf eingegangen, wie ein tarball durch das Tool buildroot erstellt werden kann, was am Beispiel der Python-Runtime gezeigt wird. Dieser lässt sich folgend in Container-Runtimes wie rkt importieren. Im Folgenden wird dieser mithilfe der Kernel-Funktionen aus Abschnitt 2.2 isoliert. Das Ergebnis ist ein Dateisystem innerhalb eines Host-OS, indem eine Instanz der Python-Runtime isoliert und ohne Root-Berechtigungen ausgeführt wird.

2.3.1 Erstellen eines tarballs

Bei einem tarball handelt es sich um ein komprimiertes Dateisystem. Dabei wird ein vollwertiges Dateisystem stark komprimiert, um es leichter zu versenden oder zusichern. Das Erzeugen eines tarballs ist durch das Tool `buildroot` einfach. Buildroot ist ein Unix-Tool, welches zur Erstellung von minimalistischen Linux-Distributionen für Embedded-Systems entworfen wurde. Es erlaubt aber auch, nur ein Dateisystem zu erzeugen, ohne Kernel oder Init-System. Dies ist entscheidend, da bei Containern der bestehende Kernel des Host-OS mitbenutzt wird (*siehe Abbildung 1*). Das Init-System, welches dazu dient, neue Prozesse zu starten, wird in einem Container ebenfalls nicht benötigt, da diese nur einen Prozess ausführen. Diese Features von buildroot erlauben es, ein Image für Container zu erzeugen.

Zudem erlaubt es buildroot, einzelne Bibliotheken, wie zum Beispiel die Python-Runtime, beim Build-Prozess der Distribution zu integrieren. Nach dem Einstellen der benötigten Bibliotheken und dem deaktivieren der, für Container, unnötigen Features erzeugt Buildroot eine Config-Datei, die alle Änderungen beinhaltet. Durch das Starten des Build-Prozesses mit dem Befehl `make` wird die gewünschte Linux-Distribution erstellt. Am Ende dieses Prozesses liegt im Ordner `/buildroot/out/images/` das gewünschte Dateisystem `rootfs.tar`.

2.3.2 Isolieren der Python-Runtime

In Abschnitt 2.3.1 wurde ein Dateisystem mit der Python-Runtime erstellt. Dieses muss nun isoliert, ein Pythonprogramm in das Dateisystem gemounted und ausgeführt werden.

Der erstellte tarball wird durch folgende Bash-Befehle entpackt. Durch den Aufruf aus Listing 1 entsteht die in Abbildung 5 gezeigte Dateistruktur.

Um einen Prozess mit dem Wurzelverzeichnis `/home/rootfs/` auszuführen, ist lediglich der folgende Aufruf nötig.

Durch diesen wird ein Webserver auf Adresse `http://0.0.0.0:8000` ausgeführt, der alle ihm zugänglichen Dateien zum Download bereitstellt. Beim Aufrufen dieser Adresse erkennt man, dass der Webserver nur Zugriff auf die in


```
cd /home
mkdir rootfs
cp rootfs.tar rootfs/
cd rootfs
sudo tar xvf rootfs.tar
sudo rm rootfs.tar
```

Listing 1: Entpacken des buildroot tarballs nach /home/rootfs

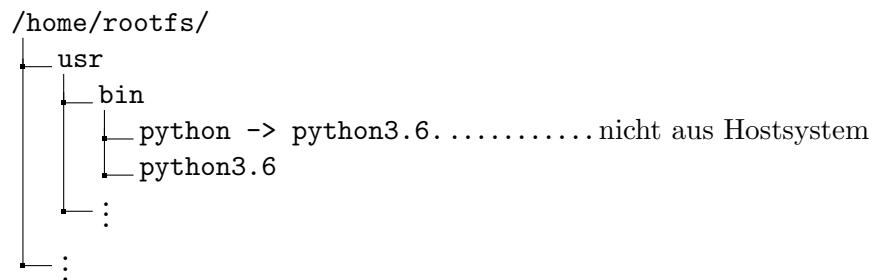


Abbildung 5: Dateibaum nach entpacken der `rootfs.tar`

`/home/rootfs/` liegenden Dateien hat, wie in Abbildung 6 gezeigt.

```
sudo chroot rootfs /usr/bin/python3.6 -m http.server
```

Listing 2: Shell-Command um Webserver mit definierter Wurzel zu starten

Directory listing for /

- [bin/](#)
- [dev/](#)
- [etc/](#)
- [lib/](#)
- [lib64/](#)
- [linuxrc/](#)
- [media/](#)
- [mnt/](#)
- [opt/](#)
- [proc/](#)
- [root/](#)
- [run/](#)
- [sbin/](#)
- [sys/](#)
- [tmp/](#)
- [usr/](#)
- [var/](#)

Abbildung 6: Python Webserver mit festgesetztem Root-Verzeichnis

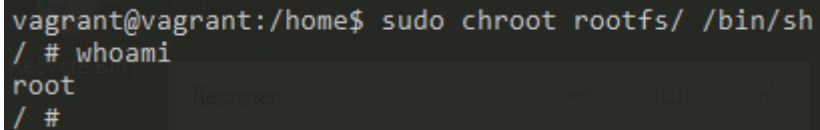
Um weiterhin Zugriff auf dynamische Inhalte aus dem Host-System zu haben, kann man, wie in Abschnitt 2.2.4 erklärt, entsprechende Verzeichnisse in das neue rootfs des Prozesses mounten.

Durch die Dateitrennung und den Aufruf von `chroot` tritt allerdings ein großes Problem auf. Der Python-Webserver wird mit erhöhten Rechten ausgeführt, da diese für den Aufruf von `chroot` benötigt werden (siehe Abbildung 7).

Dies führt zu vielen Problemen. Ein Prozess, der nur auf diese Weise isoliert wird, könnte beispielsweise Prozesse auf dem Hostsystem mit `kill <pid>` be-

```
nsenter --mount=/proc/<PID isolierter Prozess>/ns/mnt \
mount --bind -o ro \
      $PWD/readonly \
      $PWD/rootfs/var/src
```

Listing 3: Mounten von Verzeichnis /readonly/ zu /rootfs/var/src/

A terminal window showing a user at a vagrant machine. The user runs 'sudo chroot rootfs/ /bin/sh'. The prompt changes from '\$' to '#', and the user's identity changes from 'vagrant' to 'root'.

```
vagrant@vagrant:/home$ sudo chroot rootfs/ /bin/sh
/ # whoami
root
/ #
```

Abbildung 7: Root-Eskalation durch Aufruf von `sudo chroot`

enden. Die Lösung dieses Problems sind die in Abschnitt 2.2.3 beschriebenen namespaces.

Um alle Prozesse des Hostsystems vor dem Container zu verstecken, muss der PID-Namespace des Container-Prozesses neu gemounted werden.

```
sudo unshare -p --mount-proc=$PWD/rootfs/proc -f chroot
→ rootfs /bin/sh
```

Listing 4: Remount des PID-Namespaces und Chroot einer Shell

Beim Aufruf von „ps aux“ wird nur noch der Prozess `/bin/sh` angezeigt, der die PID 1 bekommen hat. Dieses Vorgehen löst allerdings nicht die Ursache des Problems. Der gestartete Prozess läuft auch weiterhin unter dem Nutzer `root`. Ein auf diese Weise isolierter Prozess, kann zum Beispiel auf Port 80 hören. Um diese Berechtigungen zu entfernen werden die in Abschnitt 2.2.6 angesprochenen Capabilities verwendet.

Durch den folgenden Aufruf hat, die in `rootfs` gestartete `/bin/bash` nicht mehr die Möglichkeit, auf niedrigere Ports, wie Port 80 zu hören.

Um vollständige Isolation des Containers zu erreichen, müssen Systemressour-

```
capsh --drop=cap_net_bind_service --chroot=rootfs/ --
```

Listing 5: Entfernen der Capability um auf Port 80 zu hören

cen, wie Arbeitsspeicher oder CPU-Zyklen limitiert werden. Dazu dienen die in Abschnitt 2.2.2 beschriebenen cgroups.

Um eine cgroup zu erstellen, muss ein Ordner unterhalb des Wurzelverzeichnis erstellt werden. Um einen Prozess einer cgroup zuzuordnen, wird die PID des Prozesses in die Datei `/sys/fs/cgroup/CONTROLLER/CGROUPNAME/tasks` geschrieben.

```
mkdir /sys/fs/cgroup/memory/container  
echo $ContainerPID > /sys/fs/cgroup/memory/container/tasks
```

Listing 6: Erzeugen einer memory cgroup namens container

Um festzusetzen, wie viel Arbeitsspeicher der isolierte Prozess nutzen darf, kann man innerhalb der cgroup einzelne Limits festlegen.

```
echo "0" >  
→ /sys/fs/cgroup/memory/container/memory.swappiness  
echo "100000000" >  
→ /sys/fs/cgroup/memory/container/memory.limit_in_bytes
```

Listing 7: Limitieren des Arbeitsspeichers und Memory-Swap deaktivieren

Um zu testen, ob die Zuweisung funktioniert und durch den isolierten Prozess maximal ~100Mb Arbeitsspeicher belegt werden können, kann folgendes Python Programm aus Listing 8 ausgeführt werden. Abbildung 8 zeigt die Ausgabe des Prozesses, der sich in der cgroup container befindet.

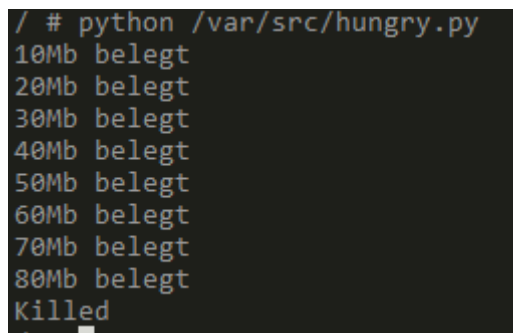
```
#hungry.py - Eating up memory in 10Mb blocks
import time

TEN_MEGABYTE = 10000000

f = open("/dev/urandom", "rb")
data = bytearray()
i = 0

while True:
    data.extend(f.read(TEN_MEGABYTE))
    i += 1
    print("%dMb belegt" % (i*10,))
    time.sleep(1)
```

Listing 8: Python Programm hungry.py um Arbeitsspeicher zu verbrauchen



```
/ # python /var/src/hungry.py
10Mb belegt
20Mb belegt
30Mb belegt
40Mb belegt
50Mb belegt
60Mb belegt
70Mb belegt
80Mb belegt
Killed
```

Abbildung 8: Ausgabe des Pythonprogramms hungry.py

3 Geschichte

Um die Frage zu beantworten, wie Docker die führende Container-Technologie wurde, wird im Folgenden die Geschichte dieser betrachtet. Dabei liegt der Schwerpunkt auf Lösungen und Probleme der Technologien.

3.1 Container-Engines

Wie in Tabelle 6 zu sehen, sind Container keine neue Erfindung. Bereits 2008 wurde die erste volle Implementierung einer Container-Runtime mit LXC veröffentlicht. Im folgenden Abschnitt wird ein Blick auf Container-Runtimes in der Vergangenheit geworfen und die Frage geklärt, wie Docker so erfolgreich wurde. Dazu werden Probleme älterer Container-Engines und Lösungen, die jüngere Schritte mit sich bringen, erklärt.

3.1.1 Vor LXC: Isolation mit Kernel-Patches

Bereits 1979 wurde mit chroot die erste, noch heute notwendige, Funktion veröffentlicht. Diese kam mit dem Betriebssystem Unix V7 und erlaubte es erstmals, verschiedene Prozesse in unterschiedliche Dateisystemen zu trennen. Der Systemaufruf wurde 1982 in BSD hinzugefügt (Osnat, 2018). 20 Jahre später rücken isolierbare Prozesse durch FreeBSD Jails in den Mittelpunkt. Diese erweitern das Chroot-Konzept, indem nicht nur das Dateisystem vom Host-System getrennt wird, sondern auch Hostnamen, IP-Adressen und die Nutzerverwaltung (The FreeBSD Documentation Project, 2018).

Neben FreeBSD Jails wurde bereits 2001 Linux VServer veröffentlicht, eine Software, die ähnlich wie Jails eine Isolation des Dateisystems und auch der Netzwerkadresse erlaubt. Entgegen der aktuell noch weiterentwickelten Jails

war der letzte stabile Release von VServer 2008 (Pötzl, 2011). Der größte Nachteil des VServers waren die benötigten Kernel-Patches, die benötigt wurden, um die Isolierung zu gewährleisten.

In den folgenden Jahren wurden immer mehr Lösungen zur Isolation von Prozessen veröffentlicht, darunter Oracles Solaris Containers, das auf Zonen im Betriebssystem setzt und Open VZ, welches wie VServer, einen gepatchten Linux-Kernel benötigt, aber auch Ressourcen isolieren kann. Der größte Nachteil, denn alle diese Technologien haben, ist die unzureichende und komplizierte Virtualisierung einzelner Prozesse. Zudem muss man Funktionen, die zur Isolation benötigt werden, nachpatchen. Dies führte 2006 dazu, dass Entwickler von Google eine bessere Lösung entwickelten, Process Containers. Diese erlaubten ohne Patches eine einfache Verwendung und Isolation einzelner Ressourcen. Im Jahr 2007 wurden Process Container unter dem Namen cgroups in den Linux-Kernel gemerged und liefern seitdem das Fundament für aktuelle Container-Technologien.

3.1.2 LXC: Erste Schritte in Container-Runtimes

Mit Linux Containers (LXC) kam 2008 die erste vollwertige Implementation, die alleine mit dem nativen Kernel des Linux-OS funktioniert. Damit löst LXC eins der größten Probleme der vorherigen Lösungen, indem kein gepackter Kernel benötigt wird.

Feature	Verwendung
Namespaces	Trennung unterschiedlicher Systemkomponenten zur Virtualisierung
Apparmor und SELinux	Mandatory Access Control
Seccomp	Sicherheitsprofile, sperrt Systemaufrufe des isolierten Prozesses
chroot und pivot_root	Isolation des Dateisystems und Nutzernamespace-Mapping
Capabilities	Entfernen von einzelnen Systemrechten
CGroups	Verwalten der Systemressourcen

Tabelle 5: Von LXC genutzte Kernel-Features (Graber, 2018)

Ein weiterer Vorteil den LXC gegenüber älteren Technologien besitzt, ist die Einfachheit in der Benutzung. Im Gegensatz zu Linux VServern liefert LXC eine Konfigurationsdatei, vorgefertigte Seccomp oder SELinux Profile und ein Command Line Interface (CLI). Diese erlauben es durch einfache Bash-Kommandos Container Prozesse in Containern zu isolieren. Dieser Vorteil ist allerdings auch ein großer Nachteil an LXC. LXC ist sehr systemnah und lässt sich durch die verwendeten Kernel-Features nur auf Linux-Systemen nutzen. Zudem ist die Konfiguration eines Containers weitreichender und komplexer als bei aktuellen Container-Runtimes.

3.1.3 CF Warden: Innovation durch Vereinfachung

2011 wurde die erste kommerziellen Container-Runtime mit Cloud Foundry (CF) Warden veröffentlicht. Diese basierte zu Beginn auf LXC und erweiterte diese mit eigenen Funktionen, wie einer REST API zur Steuerung und Verwaltung eines Container Clusters. Dabei setzt CF Warden auf eine Client-Server-Architektur.

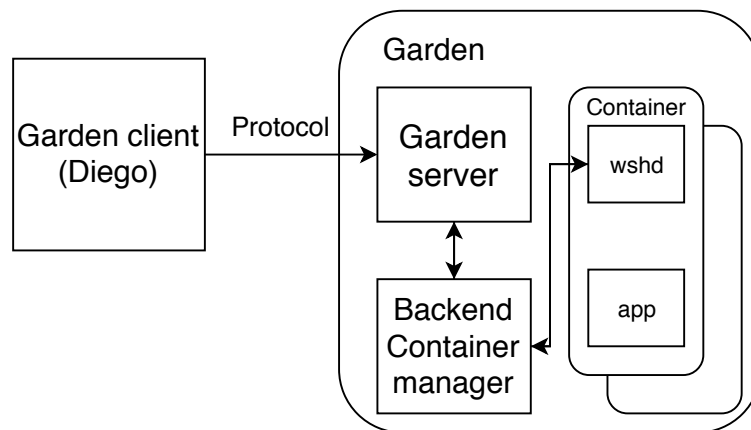


Abbildung 9: Client-Server-Architektur von CF Warden (Fedzkovich, 2016)

Im Gegensatz zu LXC ist CF Warden nicht mehr an Linux gebunden, sondern entkoppelt. Durch die Verwendung einer Bibliothek ist es möglich, Warden auch unter Windows Systemen zu verwenden. Warden kann als Container-Runtime

für CF-Cluster genutzt werden, ist dort allerdings durch Garden ersetzt worden.

3.1.4 Let me contain that for you (LMCTFY) und Go: Googles Einfluss auf Container

CF Warden wird nahezu ausschließlich innerhalb eines CF-Clusters genutzt. Eine lokale Nutzung ist dabei nicht der Schwerpunkt. Für den Entwicklungsprozess und für die Nutzung außerhalb einer CF Umgebung hat Google 2013 LMCTFY veröffentlicht. Dieser Service bildet Googles Container-Stack ab. LMCTFY wird nicht aktiv weiterentwickelt, die Kernkonzepte wurden in libcontainer übernommen und von Docker weiter entwickelt.

Eine weitere wichtige Entwicklung von Google, neben cgroups und LMCTFY, ist die Sprache Go. Diese findet in allen Container-Runtimes Verwendung und ist eine der wichtigsten Entwicklungen der letzten 10 Jahre für Container. Dabei sind die Geschwindigkeit und Flexibilität von Go die wichtigsten Aspekte. Kompilierte Go-Programme laufen auf nahezu allen Systemen. Dabei sind die ausführbaren Binaries durch statisches Binden unabhängig von installierten Bibliotheken auf dem System und ist dabei effizienter als andere Sprachen.

3.1.5 Docker: Ecosystem, Runtime und Software as a Service (SaaS) in einem

Im Jahr 2013 kam der größte Durchbruch für Container. Mit dem Release der Software Docker explodierten Container in Popularität und Nutzung (siehe Abbildung 10).

Wie auch CF Warden setzte Docker zu Beginn auf LXC. Der größte Unterschied zu bestehenden Container-Runtimes ist aber das Angebot als Ecosystem. Docker erlaubt es als erster Anbieter, Images aus dem Internet zu nutzen und bietet mit Docker Hub eine Plattform, die es sehr einfach ermöglicht, neue SaaS-Angebote für Kunden zugänglich zu machen. Zudem bietet Docker eine deutlich vereinfachte Form der Konfiguration mit Dockerfiles an. Durch die einfache Nutzung mittels CLI und der Möglichkeit, seine Services als Image

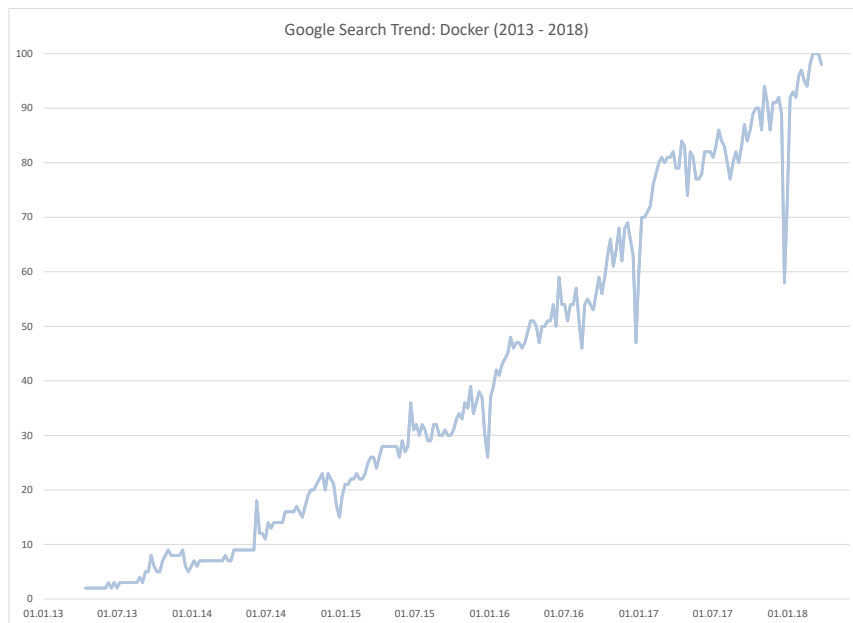


Abbildung 10: Google Search Trend für Docker (Google, 2018)

bereitzustellen gelang es Docker, unangefochten die meistgenutzte Container-Software zu werden.

3.2 Aktuelle Probleme

Durch die rasant ansteigende Popularität von Docker und Containern im Allgemeinen werden auch Probleme mit der Technologie ersichtlich. So startete Google im Jahr 2014 jede Woche mehr als 2 Milliarden Container (Beda, 2014). Durch die Menge an Container ist es eine zunehmende Herausforderung, diese zu verwalten. Seit 2014 befindet sich aus diesem Grund das Container-Orchestrierungstool K8 in Entwicklung. Neben der Herausforderung der Orchestrierung löst K8 auch das automatische Deployment von Container-

Applikationen und das Skalieren bei entsprechender Nutzlast.

Zudem wird zunehmend die Bedeutung der Sicherheit innerhalb von Containern ersichtlich. Dies wurde vor allem 2016 durch den Security-Exploit dirtyCOW zur Schau gestellt, durch den unprivilegierte Nutzer ihre Rechte eskalieren konnten (Oester, 2016). Durch diese Verlagerung werden Container-Runtimes wie Rocket (rkt), die über den gesamten Entwicklungsprozess Sicherheit versprechen, interessanter.

3.3 Zusammenfassung

Tabelle 6 gibt einen Rückblick auf die Geschichte der Container-Technologie. Dabei wird der zeitliche Hergang einzelner Funktionen und Runtimes in Bezug gestellt und aktuelle Themen aufgezeigt.

Benötigte Technologien	
1979	• Unix V7 mit chroot
1998	• SELinux
	• AppArmor
1999	• Linux Capabilities
2000	• FreeBSD Jails
2001	• Linux VServer
2002	• Linux namespaces
2004	• Solaris Container
2005	• Open VZ
2006	• Google Process Container
2007	• Process Container in Linux Kernel als cgroups
2012	• Erste stabile Version der Sprache Go
Container-Runtimes	
2008	• LXC
2011	• CF Warden
2013	• LMCTFY
	• CF Graden, Umstieg auf Go
2014	• Appc Spezifikation Release
	• rkt
2015	• LXD
	• runC
2016	• Windows Containers
	• CF Guardian Release, Support für runC
2017	• containerd v1.0.0
	• cri-o
Entwicklung Container-Ecosystem	
2011	• Initialer Release CF
2013	• Release Docker, erstes Container Ecosystem
2014	• Entwicklung K8 startet
2015	• Gründung CNCF und OCI
	• Docker Swarm
2016	• "Dirty Cow" → Container Sicherheit
	• Apache Mesos v1.0.0 Release
2017	• Übernahme rkt und containerd in CNCF
	• Release OCI runtime-spec und image-spec

Tabelle 6: Timeline Container-Technologien (Osnat, 2018)

4 Container-Runtimes

Container-Runtimes sind das Herz eines jeden Container-Angebots. Sie laden benötigte Images herunter und instanziiieren übergebene Prozesse in Containern. In diesem Kapitel werden verschiedene Runtimes miteinander verglichen und veranschaulicht, wie Runtimes neben Docker für spezielle Anforderungen geeignet sind.

4.1 Vorgehen

Um verschiedene Runtimes zu vergleichen wurde eine eigene Anwendung mit drei Microservices implementiert. Dabei wurden, wie in Abbildung 11 zu sehen, verschiedene Technologien verwendet, um zu prüfen, wie die getesteten Container-Runtimes mit diesen Umgehen. Diese wurde im Folgenden mit verschiedenen Runtimes bereitgestellt.

Um verschiedene Runtimes objektiv zu bewerten wurden dabei die folgenden Kriterien untersucht.

- Installations- und Konfigurationsaufwand
- Ease of Use
- Orchestrierung
- Sicherheit
- Aufnahme in bestehende Ecosysteme

Dabei wurde für jede Runtime dieselbe Ausgangssituation, eine virtuelle Maschine mit installiertem Ubuntu Xenial, gewählt.

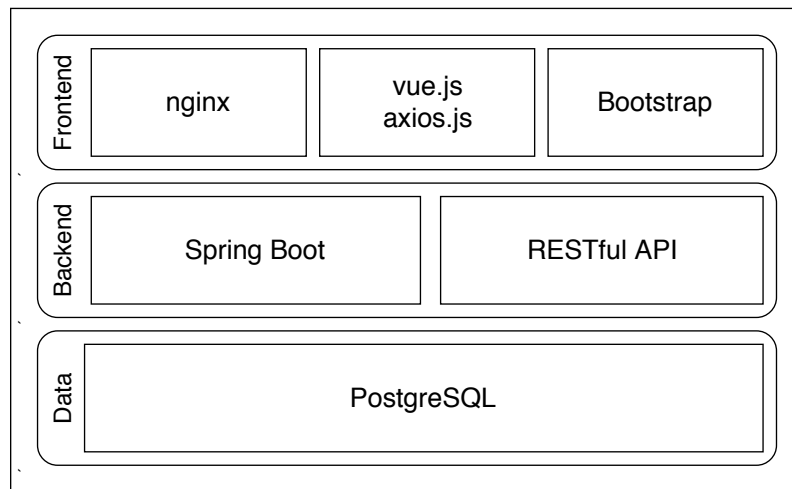


Abbildung 11: Beispielhafte Darstellung einer Micorservice-Architektur

4.2 Docker Stack

Docker ist de facto der Standard der Container-Runtimes. Dabei verwendet Docker als Runtime intern die unter der CNCF veröffentlichte Runtime containerd, die wiederum ein Aufsatz zur standardisierten Runtime runC ist (Siehe Abbildung 12).

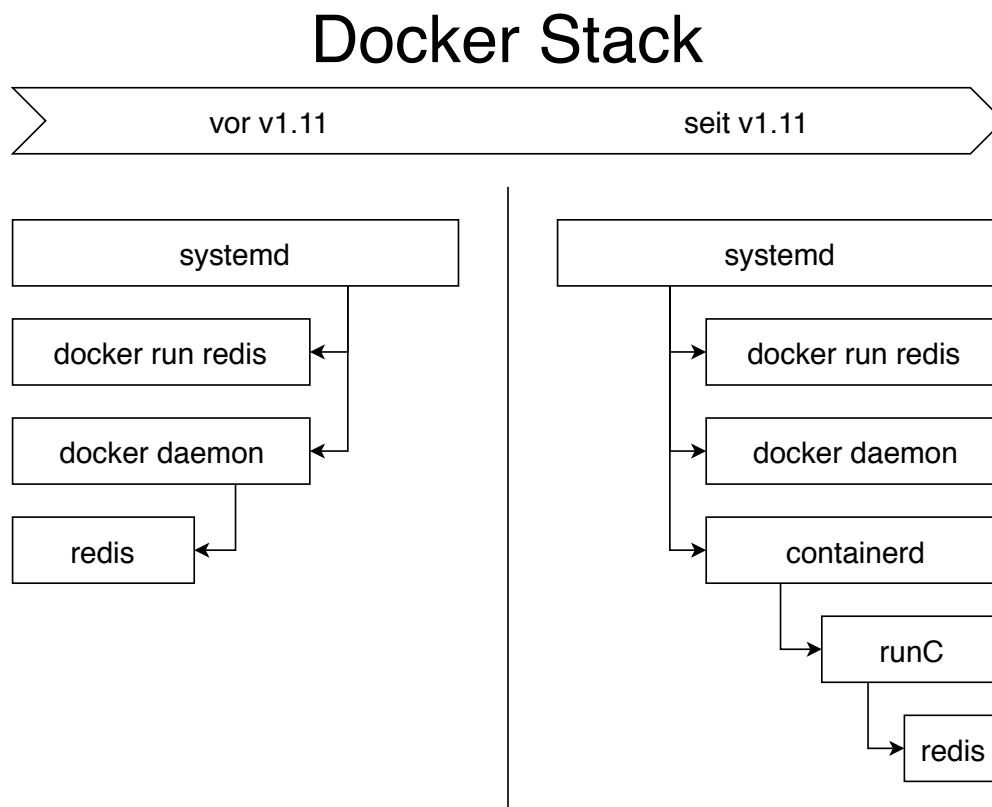


Abbildung 12: Docker Stack im Laufe der Zeit (Galeiras, 2017).

5 Letzte Inhaltsseite

Abbildungsverzeichnis

1	Container Isolation im Vergleich zu VMs	3
2	CNCF Container Runtime Landschaft (CNCF, 2018)	6
3	Auszug aus Dateisystem mit gemounteten Dateien	8
4	Unterschiede der Isolation bei Containern unter Windows . . .	10
5	Dateibaum nach entpacken der <code>rootfs.tar</code>	12
6	Python Webserver mit festgesetztem Root-Verzeichnis	13
7	Root-Eskalation durch Aufruf von <code>sudo chroot</code>	14
8	Ausgabe des Pythonprogramms <code>hungry.py</code>	16
9	Client-Server-Architektur von CF Warden (Fedzkovich, 2016) .	19
10	Google Search Trend für Docker (Google, 2018)	21
11	Beispielhafte Darstellung einer Micorservice-Architektur	25
12	Docker Stack im Laufe der Zeit (Galeiras, 2017).	26

Tabellenverzeichnis

1	Standards OCI und AppC im Vergleich (Polvi, 2015)	5
2	Cgroups-Controller und deren Verwendung (S. H. M. Kerrisk, 2018)	7
3	Linux Namespaces und verbundene Ressourcen (Biederman, 2017)	7
4	Einige Capabilities (M. Kerrisk, 2018)	9
5	Von LXC genutzte Kernel-Features (Graber, 2018)	18
6	Timeline Container-Technologien (Osnat, 2018)	23

Listings

1	Entpacken des buildroot tarballs nach /home/rootfs	12
2	Shell-Command um Webserver mit definierter Wurzel zu starten	13
3	Mounten von Verzeichnis /readonly/ zu /rootfs/var/src/ .	14
4	Remount des PID-Namespaces und Chroot einer Shell	14
5	Entfernen der Capability um auf Port 80 zu hören	15
6	Erzeugen einer memory cgroup namens container	15
7	Limitieren des Arbeitsspeichers und Memory-Swap deaktivieren	15
8	Python Programm hungry.py um Arbeitsspeicher zu verbrauchen	16

Glossar

Bash Bash ist eine freie, umfangreiche Shell, die in den meisten Unix-Systemen Standard ist.

Cloud-Native Cloud-Native Anwendungen sind spezifisch für Cloud-Architekturen entwickelt. Sie spalten meistens große Funktionalitäten in kleine Micro-services, die mittels API miteinander kommunizieren und sind somit skalierbar, loose gekoppelt und ausfallsicherer als Full-Client Anwendungen.

Image Ein Container Image ist eine Datei, die spezifiziert, wie ein Container von der Laufzeitumgebung ausgeführt werden soll.

Akronyme

appc App Container.

CF Cloud Foundry.

cgroup control group.

chroot Change Root.

CLI Command Line Interface.

CNCF Cloud Native Computing Foundation.

K8 Kubernetes.

LMCTFY Let me contain that for you.

LXC Linux Containers.

OCI Open Container Initiative.

OS Betriebssystem (*Operating System*).

PID Process Identifier.

rkt Rocket.

SaaS Software as a Service.

VM Virtuelle Maschine.

Literatur

- ARAVENA, Ricardo, 2018. What's Up With All The Container Runtimes. In: *What's Up With All The Container Runtimes. KubeCon Europe 2018* [online] [besucht am 2018-05-03]. Abgerufen unter: <http://sched.co/Dqtw>.
- BEDA, Joe, 2014. Containers at Scale. In: *Containers at Scale. GlueCon 2014* [online] [besucht am 2018-04-11]. Abgerufen unter: <https://speakerdeck.com/jbeda/containers-at-scale>.
- BIEDERMAN, Michael Kerrisk; Eric W., 2017. *namespaces(7) - Linux Manual Page*.
- CHIANG, Eric, 2017. *Containers from Scratch*. Auch verfügbar unter: <https://ericchiang.github.io/post/containers-from-scratch/>.
- CNCF, 2017. *Cloud Native Computing Foundation*. Auch verfügbar unter: <https://www.cncf.io/>.
- CNCF, 2018. *CNCF Cloud Native Interactive Landscape*. Auch verfügbar unter: <https://landscape.cncf.io/>.
- CORE OS, 2017. *rkt 1.29.0 Documentation*. Auch verfügbar unter: <https://coreos.com/rkt/docs/latest/>.
- CREQUY, Simone Gotti; Luca Bruno; Iago López Galeiras; Derek Gonyeo; Alban, 2018. *App Container*. Auch verfügbar unter: <https://github.com/appc>.
- DOCKER INC., 2018a. *Docker Documentation*. Auch verfügbar unter: <https://docs.docker.com/>.
- DOCKER INC., 2018b. *Docker security*. Auch verfügbar unter: <https://docs.docker.com/engine/security/security/>.
- FEDZKOVICH, Victoria, 2016. *Cloud Foundry's Garden: Back Ends, Container Security, and Debugging*. Auch verfügbar unter: <https://www.altoros.com/blog/cloud-foundry-garden-back-ends-container-security-and-debugging-oss-cf/>.

- GALEIRAS, Iago López, 2017. *rkt vs other projects* [online] [besucht am 2018-05-15]. Abgerufen unter: <https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html>.
- GOOGLE, 2018. *Docker - Erkunden - Google Trends*. Auch verfügbar unter: <https://trends.google.de/trends/explore?date=today%205-y&q=Docker>.
- GRABER, Daniel Lezcano; Christian Brauner; Serge Hallyn; Stéphane, 2018. *lxc(7) - Linux manual page* [online] [besucht am 2018-04-09]. Abgerufen unter: <https://linuxcontainers.org/lxc/manpages/man7/lxc.7.html>.
- HARRINGTON, Brian "Readbeard", 2015. *Building minimal Containers: Getting Weird with Containers*. Auch verfügbar unter: https://github.com/brianredbeard/minimal_containers.
- HEO, Tejun, 2015. *Control Group v2*.
- KERRISK, Michael, 2018. *capabilities(7) - Linux manual page*.
- KERRISK, Serge Hallyn; Michael, 2018. *cgroups(7) - Linux Manual Page*.
- KNULST, Cornell, 2016. *Deep dive into Windows Server Containers and Docker*. Auch verfügbar unter: <http://blog.xebia.com/deep-dive-into-windows-server-containers-and-docker-part-1-why-should-we-care/>.
- MAS, Raphaël Hertzog; Roland, 2015. *The Debian Administrator's Handbook, Debian Jessie from Discovery to Mastery*. Freexian. ISBN 9791091414043.
- MATTHIAS, Karl; KANE, Sean P., 2015. *Docker: Up & Running: Shipping Reliable Containers in Production*. O'Reilly Media. ISBN 978-1-491-91757-2.
- MCGRATH, Roland, 2017. *chroot(1) - Linux Manual Page*.
- OESTER, Phil, 2016. *Dirty COW (CVE-2016-5195)*. Auch verfügbar unter: <https://dirtycow.ninja/>.
- OPEN CONTAINER INITIATIVE, 2018. *Open Container Initiative*. Auch verfügbar unter: <https://www.opencontainers.org/>.
- OSNAT, Rani, 2018. *A Brief History of Containers: From the 1970s to 2017*. Auch verfügbar unter: <https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016>.
- PETAZZONI, Jérôme, 2013. *Create Lightweight Containers with Buildroot*. Auch verfügbar unter: <https://blog.docker.com/2013/06/create-light-weight-docker-containers-buildroot/>.

- POLVI, Alex, 2015. *Making Sense of Container Standards and Foundations: OCI, CNCF, appc and rkt*. Auch verfügbar unter: <https://coreos.com/blog/making-sense-of-standards.html>.
- PÖTZL, Herbert, 2011. *Paper - Linux-VServer*.
- ROUMELIOTIS, Rachel, 2016. The Open Container Essentials Video Collection. In: *The Open Container Essentials Video Collection*. O'Reilly Open-Source Conference. ISBN 9781491968253.
- RUSSINOVICH, Mark, 2015. *Containers in Windows Server, Hyper-V and Azure*. Hrsg. von MECHANIC, Microsoft. Auch verfügbar unter: https://www.youtube.com/watch?v=YoA_MM1GPRc.
- THE FREEBSD DOCUMENTATION PROJECT, 2018. *FreeBSD Handbook* [online] [besucht am 2018-04-09]. Abgerufen unter: https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html.
- ZAK, Karel, 2015. *mount(8) - Linux Manual Page*.

Todo list

Vergleich Docker vs the World	2
Aktuelle Probleme, Orchestrierung	2
Serverless, wenn zeit reicht	2
Figure: Hyper-V vs WinServer2016 container	10