Grundlagen Docker

Christopher J. Ruwe

23. Januar 2020

1 Namespaces und Control Groups

Namespaces

Ein *Namespace* [1] ist ein Mechanismus, einer Prozessgruppe von $\{1 \dots p\} \mid p \in \mathbb{N}$ Prozessen eine Kapselung einer (globalen) Resource eines Systems (hier Betriebssystem) zur Verfügung zu stellen. Bei Betriebssystem-Namespaces wird dies vom Betriebssystem-*kernel*, konkret durch bestimmte *in-kernel* Datenstrukturen, vorgenommen.

[1] *Linux Programmer's Manual: namespaces(7) - Overview of Linux Namespaces.* (2018). http://man7.org/linux/man-pages/man7/namespaces.7.html. Linux man-pages Project.

Innerhalb dieser Kapselung haben die Prozesse einen aus Innen-Sicht exklusiven Zugriff auf die gekapselte Resource. Bei der gekapselten Resource kann es sich beispielsweise um eine *process ID* (PID) Hierarchie, eine separate *inter-process communication* (IPC) Domäne oder um eine separate Abbildung *user-* und *group-IDs* auf Berechtigungen handeln.

Damit erlaubt diese Kapselung eine quasi-Virtualisierung einer Resource durch den *kernel*, obwohl aus Außensicht die Resource von allen Prozessen (und damit Prozeßgruppen) des Systems geteilt wird.

Diese Prozess-Isolierung schließt so eine Lücke zwischen klassischen *time-sharing systems* und Virtualisierung durch einen Hypervisor.

Bei *time-sharing* Systemen wie beispielsweise Unix oder Linux wird Prozessen durch den *kernel* Speicher und Rechenzeit zugeteilt. Die Prozesse sind wechselseitig „sichtbar" und der Zugriff auf System-Ressourcen wird zwar durch den kernel gesteuert, ist grundsätzlich aber durch jeden Prozeß des Systems möglich.

Bei Virtualisierung durch einen Hypervisor werden die Ressourcen eines Computer-Systems verschiedenen Betriebssystem-Instanzen zugeteilt. Diese Instanzen haben alle eine separaten *kernel* und teilen wiederum ihren Prozessen die virtualisierten Ressourcen so wie auf einem nicht-virtualisierten System auch zu.

Formen der Prozess-Isolierung/-Virtualisierung.

time-sharing systems → Betriebssystem-Namespaces → Hypervisor I/II.

Prozess-Isolierung trennt durch *namespaces* stärker als klassische *time-sharing* Systeme. Alle isolierten Prozesse teilen sich jedoch den selben *kernel*, so daß weniger stark getrennt wird als durch Hypervisor-Virtualisierung. Die Trennung ist allerdings für sehr viele Anwendungsfälle hinreichend stark.

Seit den späten 90igern ist Prozess-Isolierung durch *namespaces* im produktiven Einsatz. Die bekanntesten Varianten sind FreeBSD Jails und Solaris Zones. Der Linux-*kernel* stellt beginnend mit den späten *kerneln* der 2.6er-Reihe eine Reihe von namespaces zur Verfügung.[2]

[2] `/proc/$$/ns`, wobei `$$` den aktuellen Prozess bezeichnet. Analog können namespaces für andere Prozesse unter `/proc` unter den jeweiligen PIDs eingeseehen werden.

Die unter Linux bekannten Systeme zur Isolation von Prozessen LXC, LXD und auch docker nutzen eben diese Namespaces zur Kapselung von kompletten Betriebssystemen (ohne *kernel*, LXC, LXD) oder „lediglich“ von Applikationen (docker, rkt).

Ein PID-*namespace* [3] isoliert *process* IDs (1 . . . `$(sysctl kernel.pid_max)`), so daß Prozesse in verschiedenen PID-*namespaces* aus Innen-Sicht, nicht aus Außensicht, kollisionsfrei die gleiche PID haben.

Der erste Prozess innerhalb eines solchen *namespaces* erhält die PID 1 analog zum klassischen `init`-Prozeß und wird zum *parent*-Prozeß aller nachfolgend aus diesem Prozeß ge*fork*ten Prozesse. Damit ist dieser Prozeß auch verantwortlich, gegebenenfalls Signale der *child*-Prozesse zu *reapen*. Wird dieser Prozeß terminiert, werden alle *child*-Prozesse in diesem *namespace* durch den *kernel* mit `SIGKILL` terminiert.

PID-*namespaces* können bis zu einer Tiefe von 32 in sich gekapselt (*nested*) werden. Die *process* ID von Prozessen ist für alle Prozesse innerhalb des namespaces und für alle Prozesse in „Vorgänger“-*namespaces* sichtbar. Deshalb können Operationen wie das senden eines Signals oder das „Einsteigen“ in einen Namespace „abwärts“, aber nicht „aufwärts“ erfolgen.

[3] *Linux Programmer's Manual: pid_namespaces(7) - Overview of Linux PID Namespaces.* (2017). http://man7.org/linux/man-pages/man7/pid_namespaces.7.html. Linux man-pages Project.

Beware, here be dragons: Nicht jede Applikation eignet sich, *parent process* zu sein. Startet man beispielsweise einen Jenkins CI-Server mit PID 1 **und** *forkt* Jenkins aus der JVM Prozesse wie `make` oder `sh`, weil nicht-Java kompiliert wird **und** terminiert dieser Prozeß $\neq 0$, dann wird das geworfene Signal von der JVM oft nicht korrekt gefangen und die JVM terminiert. Hierzu benötigt man dann einen `init`-*Stub* wie `tini` oder startet den Container mit `docker --init`, damit wenigstens Signale sauber gesammelt (*reaping*) werden.

Ein *network namespace* [4] isoliert Systemressourcen wie Netzwerk-*devices*, IPv4/6 *stacks*, *routing*-Tabellen, *firewall*-Regeln und den Konfigurationen des `/proc/$$/net`-Verzeichnisses.

Ein physisches Netzwerk-*device* ist genau einem *network-namespace* zugeordnet. Virtuelle *ethernet-devices* werden immer als Paare in zwei verschiedenen *network-namespaces* erzeugt und können so als *tunnel* oder *bridge* dienen. Mit einem *veth* in einem virtuellen und einem physischen *namespace* beispielsweise kann über den Host ein namespace Zugriff auf angeschlossene Netzwerke des Hosts erhalten.

[4] *Linux Programmer's Manual: network_namespaces(7) - Overview of Linux Network Namespaces.* (2018). http://man7.org/linux/man-pages/man7/network_namespaces.7.html. Linux man-pages Project.

Ein *inter-process communication* (IPC) *namespace* [5] isoliert SysV IPC Domänen, innerhalb derer Prozesse mit Signalen, *shared memory* und POSIX *message queues* kommunizieren können.

IPC *namespaces* erlauben damit, „Kommunikations-Verbünde“ von Prozessen zu erzeugen und, umgekehrt, Kommunikation zwischen Prozessen einzuschränken.

[5] *Linux Programmer's Manual: svipc(7) - System V interprocess communication mechanisms.* (2016). http://man7.org/linux/man-pages/man7/svipc.7.html. Linux man-pages Project.; *Linux Programmer's Manual: mq_overview(7) - Overview of POSIX message queues.* (2017). http://man7.org/linux/man-pages/man7/mq_overview.7.html. Linux man-pages Project.

mount namespaces [6] virtualisieren die Liste der *mount points*, die für Prozesse innerhalb dieses *namespaces* sichtbar sind. Ein *mount-namespace* gehört immer zu einem *user-namespace*.

Wird ein Prozess erzeugt, ist die initiale Liste der *mount points* die Liste des *parent* Prozesses. Nachfolgende `mount`- oder `umount`-Operationen innerhalb des *namespaces* verändern dann standardmäßig nur noch die Liste des Prozesses im jeweiligen *namespace*.

Es ist möglich, *mount-namespaces* mit verschiedenen Prozessgruppen zu teilen; in diesem Spezialfall können solche Operationen im *child-namespace* die *mounts* im *parent-namespace* verändern.

[6] *Linux Programmer's Manual: mount_namespaces(7) - Overview of Linux Mount Namespaces.* (2018). http://man7.org/linux/man-pages/man7/mount_namespaces.7.html. Linux man-pages Project.

`/proc/$$/{mounts,mountinfo,mountstat}`

Unix *time-sharing system namespaces* erzeugen separate *Identifier* für `hostname` und `domainname`. Prozesse innerhalb eines UTS *namespaces* haben damit eigene Sichten auf die Datenstruktur `utsname`, die von `uname` zurückgegeben wird und gestatten „eigene" System-Namen für derart isolierte Prozesse.

User namespaces [7] erlauben separate Abbildungen von *user* und *group* IDs und damit Datei-Berechtigungen sowie *capabilities* [8] .

Damit können UID/GID eines Prozesses aus Innen- und Außensicht verschieden sein. Im häufigsten Fall kann so ein nicht-privilegierter *user* (außen) Berechtigungen eines privilegierten users (`uid == 0`) innerhalb eines *namespaces* erhalten.

Wie schon PID-*namespaces* können *user namespaces* bis zu einer Tiefe von 32 in sich gekapselt (*nested*) werden.

[7] *Linux Programmer's Manual: user_namespaces(7) - Overview of Linux User Namespaces.* (2018). http://man7.org/linux/man-pages/man7/user_namespaces.7.html. Linux man-pages Project.

[8] *Linux Programmer's Manual: capabilities(7) - Overview of Linux Capabilities.* (2018). http://man7.org/linux/man-pages/man7/capabilities.7.html. Linux man-pages Project.

Control group namespaces [9] kapseln die Sicht auf die spezifischen *limits* und *quotas*, die dem Prozeß auf verschiedene Ressourcen-Typen gesetzt sind.[10]

Sie fügen der Prozess-Isolierung keine in dem Sinne „neue" Funktion hinzu. Sie sind dennoch erforderlich, um zu verhindern, daß ein isolierter Prozeß über den Umweg seiner Sicht auf die Ressourcenkontrolle eine Sicht auf das kapselnde System erhält oder die cgroups seines *parent*-Prozesses steuern kann.

[9] *Linux Programmer's Manual: cgroup_namespaces(7) - Overview of Linux Cgroup Namespaces.* (2017). http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html. Linux man-pages Project.

[10] `/proc/$$/{cgroup,mountinfo}`

Control Groups

Ein *Control Group* (*cgroup*) [11] ist ein Mechanismus, für eine Prozessgruppe von $\{1 \ldots p\} \mid p \in \mathbb{N}$ Prozessen den „Konsum" von Systemressourcen wie *block device I/O*, CPU *time-slices* zu limitieren oder zu quotieren. So lassen sich Prozessgruppen Anteile dieser Ressourcen zuordnen. *cgroups* gehören immer zu einem bestimmten *kernel* Sub-System, im Wesentlichen CPU, *memory*, *block* und *network I/O*.

[11] *Linux Programmer's Manual: cgroups(7) - Linux Control Groups.* (2018). http://man7.org/linux/man-pages/man7/cgroups.7.html. Linux man-pages Project.

Mehrere CPU-cgroup-controller steuern den Konsum der Rechenleistung des Gesamtsystems:

Einer cgroup kann mit `CONFIG_CGROUP_SCHED` ein Minimal-Anteil an der CPU-Leistung eines Systems zugesichert werden, der dann, wenn das Rechner-System unter Last steht, für den Gebrauch der cgroup reserviert ist. Umgekehrt kann eine Obergrenze konfiguriert werden, die auch dann, wenn das System nicht belastet ist, nicht überschritten wird.

Ausgesteuert über `CONFIG_CGROUP_CPUACCT` kann die CPU-Nutzung der Prozesse einer cgroup abgerechnet werden. Unterschieden wird hier zwischen `user` (Zeit im *user-mode*) und `system` (Zeit im *kernel-mode*).

Mit `CONFIG_CPUSETS` können Prozesse „hart“ an spezifische Instanzen von Prozessoren, Prozessorkernen oder NUMA-Instanzen gebunden werden.

Über den *cgroup* `memory`-*controller* kann der Konsum von *process* und *kernel-memory* sowie *swap* einer Prozessgruppe ausgesteuert werden. Der `hugetlb`-*controller* steuert die Nutzung von *huge pages* aus.

`net_prio` steuert Prioritäten (*per network-device*) aus. `net_cls` flaggt Pakete, die die cgroup verlassen, mit einer class-ID, die für firewall-Regeln oder für *traffic shaping* verwendet werden können.

`blkio` steuert den Bandbreiten-Verbrauch der übergebenen *block devices* durch entweder *time-slices* oder durch obere Grenzen der konsumierten Bandbreite.

Andere, schlechter zu klassifizierende *cgroup*-controller steuern dii Erzeugung (`mknod`) oder den Zugriff auf *devices* (`devices`), die Fähigkeit zum `suspend`/`restore` (`freezer`), die Fähigkeit, Monitoring-*probes* auf Prozesse zu setzen (`perf_event`) und begrenzen die maximale Anzahl der Prozesse (`pids`)

Nutzung von Namespaces und Cgroups, Unterschiede zu anderen

Für das GNU/Linux-Betriebssystem existieren verschiedene Implementierungen von dort „Container“ genannten Mechanismen zur Isolation von Prozessen. Sie unterscheiden sich zum einen in der Konfigurierbarkeit der genutzten Isolations-Primitiven als auch im Anwendungsprofil.

`lxc`/`lxd` [12] container dienen der Provisionierung vollständiger Betriebssysteminstanzen (ohne kernel, `/sbin/init` hat PID 1) und erlauben Konfiguration der Paramter von namespace/cgroup-Isolation (bspw. *capabilities*).

Docker [13] hingegen gestattet, die Isolierung aus verschiedenen *namespace*-Typen sehr flexibel zusammenzusetzen: Verschiedenen docker-container-Instanzen können bspw einen namespace wie den network-namespace oder den IPC-namespace teilen (so genutzt in Kubernetes) oder auch die Trennung vom Host-System ganz aufzuheben und beispielsweise den network-namespace des Hosts mitzunutzen (so genutzt für das `kubelet` einer Kubernetes *node*).

[12] *lxc(7) - Linux Containers.* (2019). https://linuxcontainers.org/lxc/manpages//man7/lxc.7.html. linux-containers.org; *LXD - System Container Manager.* (2019). https://lxd.readthedocs.io/en/latest/. linux-containers.org

[13] Docker Inc. (2018). Docker Documentation. https://docs.docker.com/

Aus anderen Unixoiden Systemen sind unter Umständen Prozess-Isolations-Mechanismen wie FreeBSD Jails [14] oder Solaris Zones [15] bekannt. Für diese sind die zu nutzenden *namespace*-Primitiven fest. Es besteht keine Möglichkeit, dies zu konfigurieren und beispielsweise mehrere Zonen die selbe IPC-Kommunikations-Domäne gemeinsam zu nutzen und darüber Information auszutauschen.

Sowohl Jails als auch Zonen verfolgen einen anderen Einsatzzweck als docker-container. Dies stellt keinen Makel dar, ist aber für den Vergleich zu berücksichtigen. docker verfolgte von Anfang an den Zweck, verteilte Systeme auch aus eng anstatt nur lose gekoppelten Komponenten zu ermöglichen. Jails und Zonen sollten immer in sich geschlossene Gesamtsysteme bereitstellen und allenfalls über das Netzwerk zu verteilten Systemen zusammengefügt werden.

[14] Kamp, P.-H. & Watson, R. N. M. (2000). Jails: Confining the Omnipotent Root. In *In Proc. 2nd Intl. SANE Conference*; *FreeBSD System Manager's Manual - jail(2)*. (2012). FreeBSD Project; *FreeBSD System Manager's Manual - jail(8)*. (2012). FreeBSD Project

[15] Price, D. & Tucker, A. (2004). Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of 18th Large Installation System Administration Conference (LISA)* (S. 241–254); *Oracle Solaris 11.4 Reference Library - zones(7)*. (2016). https://docs.oracle.com/cd/E88353_01/html/E37853/zones-7.html. Oracle, Inc.

docker

`docker` zu betreiben bedeutet, Applikations-Container zu betreiben. Ein Applikations-Container ist ein durch die beschriebenen `namespace`-Mechanismen gekapselter Prozeß, wo das zugehörige Programm als eine *appliance*, als ein Image transportiert wird. Ein solches Image enthält alle Dateien (Programmtext und Bibliotheken), die zum Betrieb des Programmes benötigt werden. Oft enthalten Images allerdings erheblich mehr.

Streng genommen müssen lediglich der Programmtext der Applikation, alle dynamisch gebundenen libraries und die Konfiguration vorliegen.

```
ldd /usr/bin/emacs25 | tr -d '\\r' | sort
/lib64/ld-linux-x86-64.so.2 (0x00007ffa1de9a000)
libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007ffa19476000)
libasound.so.2 => /usr/lib/x86_64-linux-gnu/libasound.so.2 (0x00007ffa1a430000)
libatk-1.0.so.0 => /usr/lib/x86_64-linux-gnu/libatk-1.0.so.0 (0x00007ffa15ea6000)
<...>
/usr/bin/emacs25
```

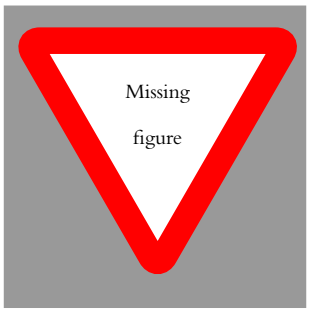
Listing 1: Auflösen der Bibliotheken, gegen die eine *binary* dynamisch gebunden wurde.

Ein Image kann man sich dann als ein komprimiertes Archiv analog einer `.tar.gz` Datei vorstellen, das alle benötigten Dateien enthält. Dieses Image wird dann auf der Maschine, auf der der Container instanziiert werden soll, entpackt. Um die Menge der im Image transportierten Dateien zu reduzieren und damit die Übersichtlichkeit zu erhöhen, ist im „idealen" docker-container nur ein einziges, statisch gebundenes *binary* enthalten.

In der Realität jedoch findet man in images oft vollständige Betriebssysteme (ohne *kernel*), die allenfalls in einer minimalen Variante vorliegen (`debian:jessie-slim`). Es ist unbestritten erheblich einfacher und weniger fehleranfällig, Programme und deren Abhängigkeiten mit dem Paketmanager einer Betriebssystem-Distribution zu installieren. Abhängigkeiten manuell aufzulösen und anderweitig in das Image zu kopieren stellt eine in vielen Fällen zu vermeidende Re-Implementierung dar.

Vor Instanziierung eines Containers wird das Image auf das Zielsystem kopiert und entpackt. Bei Instanziierung wird hieraus eine Kopie erstellt und wird diese Kopie als Wurzelverzeichnis in den mount-namespace des Containers gelegt. Die konkrete Form der Kopie wird durch die Wahl des Storage für den docker-daemon, den *graph-volume driver* bestimmt.

Wählt man `vfs` als Storage, handelt es sich um eine tatsächliche Kopie. Dies ist eine schreib-intensive und deshalb langsame Methode, die für produktive Umgebungen nach Möglichkeit vermieden wird. Die Treiber `overlay` und `overlay2` ermöglichen es, das Wurzelverzeichnis des Containers aus einem oder mehreren `overlay-mounts` zusammenzusetzen. Dies ist erheblich weniger schreib-intensiv und daher um ein vielfaches performanter. Mit dem `devicemapper` kann man analog aus einem *logical volume manager* logische volumes konsumieren. Docker-Treiber für moderne Dateisysteme wie `ZFS` oder `btrfs` erlauben es, Images als separates Dateisystem anzulegen und im Bedarfsfall einen *snapshot* zu *clonen*.



Layered Images.

Eine spezielle Eigenschaft von Images rechtfertigt diesen „Aufwand“. Images können gestapelt werden. Die Idee dabei ist, gemeinsame, „basische“ Anteile der paketierten appliance (Applikation + Libraries) als gemeinsame, identische Grundlage und damit das korrespondierende Image für mehr als eine appliance zu verwenden. Erst Differenzen im Aufbau von Appliances verlangen dann eigene, separate Images.

Dieser Aufbau läßt sich bei den `overlay`-Treibern schön beobachten:

Listing 2: Aufbau eines Docker-Images aus mehreren *Layern*.

```
docker inspect $SOMEDOCKER \
| jq '.[].GraphDriver'
```

zeigt in Listing 3 die Schichtung eines Containers aus dem `nginx:latest` image, in Listing 4 die erheblich klarere, inhaltlich aber gleiche Ansicht bei Nutzung von ZFS. Storage wird hier als dataset in einer Art „verketteten Liste“ von *snapshots* und *clones* aufgebaut.

Die als `Lower Dir` angegebenen Verzeichnisse sind in absteigender Folge ”übereinander geschichtet”und unter dem Pfad `MergedDir` liegt das vollständig aufgebaute Wurzelverzeichnis der Appliance.

Listing 3: Aufbau der Storage-Schicht eines Docker-Containern bei `overlay2`. Auszug, kein legales JSON, dafür lesbar.

```
"GraphDriver": {
  "Data": {
  "LowerDir": "/var/lib/docker/overlay2/e6daa<...>-init/diff
              :/var/lib/docker/overlay2/efe96<...>/diff
              :/var/lib/docker/overlay2/01f84<...>/diff
              :/var/lib/docker/overlay2/aca66<...>/diff
              :/var/lib/docker/overlay2/cc55a<...>/diff
              :/var/lib/docker/overlay2/35a77<...>/diff
              :/var/lib/docker/overlay2/1f882<...>/diff
              :/var/lib/docker/overlay2/6e29c<...>/diff
              :/var/lib/docker/overlay2/522d6<...>/diff
              :/var/lib/docker/overlay2/d2d57<...>/diff
              :/var/lib/docker/overlay2/a3e53<...>/diff
              :/var/lib/docker/overlay2/17304<...>/diff
              :/var/lib/docker/overlay2/67639<...>/diff
              :/var/lib/docker/overlay2/dc8ad<...>/diff
              :/var/lib/docker/overlay2/7c9b8<...>/diff
              :/var/lib/docker/overlay2/44c5f<...>/diff
              :/var/lib/docker/overlay2/bf38e<...>/diff
              :/var/lib/docker/overlay2/e68db<...>/diff
              :/var/lib/docker/overlay2/ac010<...>/diff",
  "MergedDir": "/var/lib/docker/overlay2/e6daa<...>/merged",
  "UpperDir": "/var/lib/docker/overlay2/e6daaf<...>/diff",
  "WorkDir": "/var/lib/docker/overlay2/e6daa<...>/work"
  },
  "Name": "overlay2"
}
```

Listing 4: Aufbau der Storage-Schicht eines Docker-Containern bei `zfs`.

```
{
  "Data": {
    "Dataset": "rpool/var/lib/docker/647eb<...>",
    "Mountpoint": "/var/lib/docker/zfs/graph/647eb8<...>"
  },
  "Name": "zfs"
}
```

2 *Übungen mit dem* `docker` *CLI*

Einige wenige praktischen Übungen sollten dies weiter vertiefen. Im ersten Schritt werden wir aus `Dockerfile`'s Images bauen und damit Programme als Appliance paketieren und ausführen. Im zweiten Schritt werden wir kurz auf das Management des docker hosts und von Storage und Networking eingehen.

Images erzeugen

Ausgangspunkt eines builds ist üblicherweise eine `Dockerfile`. Diese Datei enthält in einer auf das Wesentliche beschränkten Syntax `batch`-ähnliche Aufrufe, wie die *layer*, aus denen letztendlich die Appliance aufgebaut wird, zu befüllen sind.

Dies Anweisungen in Listing 5 bauen auf einem Basis-Image auf (gegeben durch das statement `FROM:`) auf, setzen eine Umgebungsvariable (`ENV`), die den Paketmanager in den nicht-interaktiven Modus versetzt und installieren einige Pakete. Weil jedes Statement für sich ein separates *layer* erzeugt, und jedes *layer* für sich atomar abgeschlossen ist, löscht man nach Aufrufen des Paketmanagers die Caches. [16] Anderenfalls blieben die Caches bestehen und würden durch alle nachfolgenden Build-Schritte, die auf diesem Image aufbauen, mitgeschleppt.

```
FROM debian:stretch

ENV DEBIAN_FRONTEND=noninteractive
RUN \
    apt update\
 && apt upgrade --yes \
 && apt install \
      --no-install-recommends \
      --yes \
        apt-transport-https \
        ca-certificates \
        curl \
        gnupg \
        less \
        procps \
 && apt clean \
 && apt autoremove \
 && rm -R /var/lib/apt/lists/* \
 && rm -R /var/cache/apt
```

Listing 5: Aufbau eines Docker-Images aus einer Basis zzgl. der Installation von Tools.

[16] In gut ausgebauten Build-Umgebungen wie einer CI/CD-Pipeline würde man vor jedem Build die `Dockerfiles` mit einem sogenannten `linter` [17] prüfen.Damit werden `Dockerfiles` einer statischen Code-Analyse unterworfen, die bekannte Fehlerquellen (wie nicht-Löschen der Caches) moniert und daraus einen Fehler-Report erzeugt.

[17] `github.com/hadolint`. (2018). Haskell Dockerfile Linter. https://github.com/hadolint

[18] Namespaces zu erzeugen erfordert root-Rechte.

Aus einer solchen `Dockerfile` wird das Image wie in Listing 6 erzeugt.[18]

```
sudo docker build \
  -f Dockerfile \
  -t [<dockerregistry>/][<organization>/]<imagename>:[<version>|latest] \
  <context_root_directory>
```

Listing 6: Image Bau

Die flags `-f` und `-t` erklären sich aus dem Beispiel. Es wird immer der sogenannte *build context*, oft `$PWD` übergeben. Aus diesem *build context* bezieht der `dockerd` Dateien, die während des *builds* durch Primitiven wie `COPY` und `ADD` verwendet werden. Mit dem Kommando `docker tag -t <THETAG> <THEIMAGEHASH>` kann der Name eines Images im Nachhinein festgelegt werden.

In einer „leeren“ Installation wir der Befehl `docker images` (alte Form) oder `docker image ls` eine leere Ergebnismenge liefern. Nach erfolgreichem Build sollte das Image mit dem als Argument von `-t` angegebenen *tag* angezeigt werden.

Weitere Feinheiten des *builds* werden im Kurs über CI/CD Pipelines behandelt.

```
FROM scratch

COPY <binary> /
COPY <config> /config.yaml

CMD ["/binary", "-c", "/config.yaml"]
```

Listing 7: Image Bau `FROM: scratch`

Listing 7 zeigt die „reine Lehre“ (Leere?) für den Aufbau eines docker images. In einer wirklichen Micro-Service-Architektur befindet sich in

einem Docker-Container nur eine einzelnes *binary* (notwendigerweise dann statisch gebunden) und einige Konfigurationsdateien, pfiffigerweise vielleicht noch `/etc/passwd`, die Wurzelzertifikate und die *timezone* Definitionen.

Das Basis-Image `scratch` ist dabei ein „besonders", nämlich leeres Image. So erzeugt man mit docker kleine und damit leicht transportable Images. Als ein Nebeneffekt erleichtert dies die Verwundbarkeits-Analyse ganz erheblich.

Alternativ zu der Methode `FROM: scratch` lässt sich mit dem Aufruf von `docker import` ein Image aus einem tar-ball erzeugen. Der Inhalt des tar-balls entspricht dann dem Inhalt des Wurzelverzeichnisses. So könnte ein docker-image alternativ wie in Listing 8 erstellt werden.[19] Weiterhin alternativ können Images auch mit `docker save` und `docker load` oder `docker export` in tar-balls importiert und exportiert werden. `docker commit` erzeugt neue

```
WHEN=$(date +%s)
FAMILIY=ubuntu
DISTRO=bionic
TARGET=/${HOME}/images/${FAMILY}/${DISTRO}/$WHEN
TAG="${FAMILY}-${DISTRO}-${WHEN}"

sudo debootstrap \
  --merged-usr \
  --variant=minbase \
  --verbose ${TARGET} \
  https://<aptproxy>:<port>/repository/apt-${DISTRO}

sudo tar \
  -cPf $TAG.tar \
  -C $TARGET \
  ${PWD}

sudo docker import \
  $TAG.tar \
  $TAG
```

Listing 8: Bau eines Images aus einem tar-ball

Images aus den Änderungen in einem Container. Dies wird im Aufbau von Images aus `Dockerfiles` durch den `dockerd` angewandt: Es wird aus der Basis ein Container erzeugt, im Container das Kommando aus der Dockerfile ausgeführt, committed und beendet. Dann folgt das nächste Kommando analog.[22]

Container zu erzeugen, drin rumzurühren und dann manuell zu committen ist für mich persönlich ein Hinweis, daß *application state* nicht sauber isoliert ist oder die Installation einer Applikation interaktive und daher schwer zu reproduzieren ist. Beides halte ich für *A Bad Thing*™.

Mit dem Kommando `docker history` lässt sich wie in Listing 9 gezeigt nachvollziehen, wie ein Image entstanden ist. Das Image für `docker-compose`[23] wurde offenbar von Linux Alpine [24] abgeleitet, in das danach das *binary*[25] kopiert wurde. Jede Zeile entspricht einem *layer* (Erinnerung: Listing 3) und man kann erkennen, daß jede Zeile einer `Dockerfile` zu einem separaten *layer* führt. Im Gegensatz dazu hat ein Image, das direkt aus einem tar-ball erzeugt wurde (`deboostrap` uä), nur ein *layer*.

[19] Als Alternative zu eigenen Scripten könnte `hashicorp/packer` [20] oder ein dediziertes, prinzipiell gleich funktionierendes Build-Tool wie das `docker-maven-plugin` [20] verwendet werden.

[20] `hashicorp/packer`. (2019). Hashicorp Packer - Docker Builder. https://www.packer.io/docs/builders/docker.html

[21] `github.com/fabric8io/docker-maven-plugin`. (2019). Maven plugin for running and creating Docker images. https://github.com/fabric8io/docker-maven-plugin

[22] Während eines *builds* kann man mit `watch sudo docker container ls` oder `watch sudo docker ps` beobachten, wie Container erzeugt und wieder beendet werden.

[23] `docker-compose` dient dem deklarativen Erzeugen von Kompositen aus mehreren Docker-Container und wird später (??) behandelt.

[24] Alpine Linux Development Team. (2019). Alpine Linux. Small. Simple.Secure. https://alpinelinux.org

[25] Python, paketiert mit pyinstaller [26] , *dynamically linked, stripped*.

[26] Pyinstaller Development Team. (2019). Pyinstaller. https://pyinstaller.org

Listing 9: Untersuchung des Aufbaus eines Images

```
docker history docker/compose:1.21.2

IMAGE CREATED CREATED BY SIZE
86bf85b58bc8 9 months ago /bin/sh -c #(nop)  ENTRYPOINT [”docker-compo…”] 0B
<missing> 9 months ago /bin/sh -c #(nop) COPY file:04bb859a7ea360f4… 10.9MB
<missing> 10 months ago /bin/sh -c apk update && apk add --no-cache … 29.8MB
<missing> 10 months ago /bin/sh -c #(nop)  ENV DOCKERBINS_SHA=1270dc… 0B
<missing> 10 months ago /bin/sh -c #(nop)  ENV GLIBC=2.27-r0 0B
<missing> 12 months ago /bin/sh -c #(nop)  CMD [”/bin/sh”]              0B
<missing> 12 months ago /bin/sh -c #(nop) ADD file:6edc55fb54ec9fc36… 3.96MB
```

Images transportieren

Funktional wäre es wohl unschädlich, aber üblicherweise möchte man nicht vor jeder Benutzung Images bauen. Stattdessen nutzt man sogenannte docker registries. Diese registries können privat [27] oder öffentlich sein, verschiedene Firmen bieten diesbezüglich *SaaS*-Produkte an[28] und eigentlich alle Cloud-Provider haben Vergleichbares im Portfolio.[31]

Docker Images werden explizit (`docker pull nginx`) oder implizit (`docker run nginx`) gezogen. Mit der ersten Form werden die *layer* von der gewählten Docker Registry heruntergeladen und auf der lokalen Maschine gespeichert. Gleichzeitig wird, weil durch die fehlende Versionsangabe implizit `nginx:latest` verlangt wird, geprüft ob ein „jüngeres“ `:latest` verfügbar ist und dann, wenn, das Update durchgeführt. Mit der zweiten Form werden Images nur dann, wenn sie lokal nicht vorliegen, gezogen, ein implizites Update findet also nicht statt.[32]

Wir erinnern uns an die Form des Namens eines Docker-Images aus dem Aufruf eines Builds (Listing 6, Seite8):

```
(-t): [<docker-registry>/][<organization>/]<imagename>:[<version>|latest]
```

`<docker-registry>` ist optional, entfällt dieser Teil des Arguments, wird angenommen, daß von der docker-registry hub.docker.io gezogen wird. Entfällt die `<organization>`, wird implizit eine spezielle, „*top-level*“ Organisation angenommen. Wird die Version nicht übergeben, wird automatisch `:latest` gezogen. Damit ist `docker pull nginx` (*at the time of writing*) eine Kurzform von `docker pull docker.io/nginx:1.15.8` Umgekehrt wird mit dem Aufruf von

```
docker push my.company.org/mydepartment/myimage:latest
docker push my.company.org/mydepartment/myimage:v1.2.3
```

ein Image in eine Docker Registry hochgeladen.

Images können mit `docker rmi <IMAGETAG>|<IMAGEHASH>` oder analog mit `docker image rm` gelöscht werden. Mit `docker search` können Images, allerdings nur von `docker.io` über die Kommandozeile gesucht werden. In aller Regel ist davon abzuraten, binaries ohne Herkunftsnachweis zu verwenden. Im Regelfall finden sich über die Webinterfaces der Registries Verweise auf das Quellcode-Repository[33], wo die `Dockerfiles` verwaltet und gepflegt werden. Dies erlaubt zum einen Kontrolle, zum anderen auch niederschwellige Möglichkeiten der Anpassung für den eigenen Bedarf.

[27] Private Registries verlangen einen login vor der Nutzung, `docker login` und `docker logout`.

[28] Die bekanntesten sind der sogenannte Docker Hub [29] und quay.io. [30]

[29] Docker Inc. (2019c). dockerhub. Build and Ship any Application Anywhere. https://hub.docker.com

[30] CoreOS, Inc. (2019). Quay [builds, analyzes, distributes] your container images. https://quay.io

[31] Elastic Container Registry bei Amazon AWS, Azure Container Registry bei Microsoft und Google Container Registry bei Google. Einfallslos-langweilige Namen sind ein sicheres Kennzeichen des *mainstreams*.

[32] Ausführliche Dokumentation: *docker pull(1) - Pull an image or a repository from a registry.* (2018). https://docs.docker.com/engine/reference/commandline/pull/. Docker Inc.

[33] zB `github.com/docker-library`. (2019). Docker Official Images. https://github.com/docker-library/official-images

Container erzeugen und bedienen

Linux container und damit auch docker container nutzen die Isolations-Mechanismen des kernels (namespaces), um einen Prozess oder eine Prozessgruppe bezüglich der Nutzung spezifischer Ressourcen vom Rest des Systems zu trennen. Ein kein-Prozeß kann nicht in namespaces gestellt werden, weswegen zur Erzeugung eines Containers ersteinmal ein Prozeß erzeugt werden muß.

Ein in einem container gekapselter Prozeß wird mit

```
docker run <imagename>[:<imageversion>] [program]
```

erzeugt. Dabei wird das Programm `[program]`, wenn es übergeben wurde, gestartet, sonst das Programm unter `CMD`. Existiert ein `ENTRYPOINT` oder wurde docker mit

```
docker run \
  [--entrypoint <path>] \
  <imagename>[:<imageversion>] [program]
```

aufgerufen, wird stattdessen dieses Programm gestartet.

Der sogenannte `ENTRYPOINT` dient dem Start *vorbereitender* Programme, die „Hilfstätigkeiten" wie das Setzen von *permissions* uÄ ausführen. Üblicherweise ruft der `ENTRYPOINT` danach das „eigentliche" Programm auf, oft, nicht eine `bash` als parent-Prozeß stehen zu lassen, mit dem Aufruf von `exec`.

Es haben sich hier mittlerweile verschiedene Geschmäcker herausgebildet, es kommt auch vor, daß einige mit `ENTRYPOINT` das „eigentliche" Programm starten.

Möchte man zu Test-Zwecken bspw. ein aktuelles Ubuntu starten, könnte man mit

```
docker run \
  --interactive \
  --tty \
  ubuntu:18.04 /bin/bash
```

eine `/bin/bash` in einem Container startet. Alle *binaries* würden dann in einem aus Ubuntu abgeleiteten Image bezogen. Hätte man lieber ein Image aus der RedHat-Familie, setze man statt `ubuntu:18.04` `centos:7.5` oder `fedora:27` ein.

Der Container könnte auch ohne `-tty` und `--interactive` gestartet werden. [34] Dies wäre allerdings für den Fall einer `/bin/bash` wenig sinnvoll: Das Flag `--interactive` hält `STDIN` offen und `--tty` klebt `STDIN` des containers an ein (Pseudo)-Terminal. Ohne `--tty` kann man den Container starten und sogar extrem rudimentäre Eingaben (`ls` *anyone*) starten, Signale über das Terminal zu senden (`C^c`) ist aber nicht möglich (der container muß danach mit `docker rm -vf <containerId>` terminiert werden).

[34] Docker Inc. (2019b). Docker run reference. https://docs.docker.com/engine/reference/run/

Ohne `--interactive` kann man den container ebenfalls starten, man sieht dann sogar den Eingabe-Prompt. Eingaben kann man wenigstens auf das Terminal leiten, allein, es nützt nichts: Sie gelangen im container nicht zur Ausführung. Erst die Kombination von `--interactive --tty` erlaubt eine „normale" Interaktion mit der gestartet shell. Um einen nicht-interaktiv zu nutzenden Container wie `nginx` zu starten, benötigt man beides nicht.

Ruft man `docker ps -a`, läßt man sich alle Container inklusive aller terminierten Container anzeigen. Container nach Abschluß stehen zu lassen hat Nebeneffekte, die man bei sorgsamen Betrieb besser vermeidet: Das Verzeichnis mit den gemergten binaries bleibt stehen und der Container kann wieder gestartet werden. Beides bricht mit dem Prinzip, daß container keinen persistenten Zustand haben sollen. Auch wenn bei Verwendung von aktuellen storage Mechanismen wie `overlay(2), btrfs, zfs` oder `devicemapper` kein zusätzlicher Plattenplatz konsumiert wird, verdreckt dies dennoch das System.

Kommt der treiber `vfs` zum Einsatz, dann handelt es sich um Voll-Kopien und dann *wird* Platz konsumiert. Dies ist allerdings ein Spezialfall, aus Performanz-Gründen wird `vfs` produktiv nicht verwendet.

Weil man das Löschen terminierter Container meistens dann doch vergisst, ruft man docker sinnvollerweise mit

```
docker run \
--name mybeautifulshellcontainer
--interactive \
--rm \
--tty \
ubuntu:18.04 /bin/bash
```

wodurch der Container nach Abschluss automatisch gelöscht wird. Containern Namen zu geben erhöht, ohne funktional erforderlich zu sein, die Übersicht ganz erheblich.

Möchte man nicht alle terminierten Container einzeln löschen, empfiehlt sich der Aufruf von `docker container prune`, der ohne weitere Unterscheidung alle terminierten Container abräumt.

Benötigt man für container Persistenz, so empfiehlt es sich, dies entweder in separaten docker-volumes vorzuhalten oder mit sogenannten *bind-mount*, `--volume=[=[[HOST-DIR:]CONTAINER-DIR[:OPTIONS]]]` persistenten Speicher vom docker-host im container verfügbar zu halten. Beliebig viele `--volume` Parameter können übergeben werden, es ist jedoch darauf zu achten, nicht durch in der Reihenfolge später übergebene frühere zu überschreiben.

```
docker run \
  --interactive \
  --rm \
  --tty \
  --volume=<onhost>:/<indocker>
ubuntu:18.04 /bin/bash
```

Mit den Optionen `:rw` (default) oder `:ro` kann ausgesteuert werden, ob das Volume *read-write* oder *read-only* gemounted werden soll. Weitere *tunables* sind in der Dokumentation [35] oder der man-page beschrieben.

[35] Docker Inc. (2019d). Manage data in Docker. https://docs.docker.com/storage/volumes/

Ohne Netzwerkanbindung kann Software nicht gut genutzt werden. Einen webserver wie nginx von Netzwerk abgekoppelt zu nutzen ist wenig sinnvoll. Startet man bspw. nginx ohne gesonderte Parametrisierung der Konnektivität,

```
docker run \
  --rm \
  --volume=<somepath>/wwwdata:/var/www/html \
  nginx
```

kann man bspw. mit `docker inspect` und `jq` dessen Anbindung untersuchen.

```
docker inspect <name|id> \
| jq '.'
docker inspect <name|id> \
| jq '.[].NetworkSettings.IPAddress'
```

Ungeachtet des Spezial-Themas empfehle ich jedem dringend die weitere Vertiefung des Programmes `jq` [36] , der manpage [36] und dem ausführlichen Manual. [36]

[36] `github.com/stedolan/jq`. (2019). Command-line JSON processor. https://github.com/stedonlan/jq

[37] *jq - Command-line JSON Processor(1)*. (2018). http://manpages.ubuntu.com/manpages/bionic/man1/jq.1.html. Canonical - Ubutu Manpage Repository

[38] Stephen Dolan. (2019). *jq Manual (development version)*. https://stedolan.github.io/jq/manual/

Mit der so ermittelten IP kann man tatsächlich unter Port 80 zugreifen. Auf Hosts Dienste mit RFC1918 [39] Adressen zu exponieren, die uU in nicht in den Unternehmens-Kontext integrierten Netzen stehen, wird allerdings wenig zielführend sein. Für die Öffentlichkeit bestimmte Dienste kann man so erst recht nicht betreiben.

[39] Moskowitz, R., Karrenberg, D., Rekhter, Y., Lear, E. & de Groot, G. J. (1996). Address Allocation for Private Internets. https://rfc-editor.org/rfc/rfc1918.txt

Statt dessen exponiert man mit dem sogenannten *port-forwarding* Ports im Container auf dem Host. Wird `docker` mit dem Parameter `--publish` gerufen,

```
docker run \
   --publish <HOSTPORT>:<CONTAINERPORT> \
   --rm nginx
```

dann wird üblicherweise mit `iptables` wie in Listing 10 gezeigt der host port auf die IP des containers geNATed. Die Container-Configuration ist mit Listing 11 gezeigt.

```
   sudo iptables-save \
2  | grep -E '8888|192.168.66.4|docker'

4  -A POSTROUTING -s 192.168.66.0/24 ! -o docker0 -j MASQUERADE
   -A POSTROUTING -s 192.168.66.4/32 -d 192.168.66.4/32 -p tcp -m tcp --dport 80 -j
   ↪  MASQUERADE
6  -A DOCKER -i docker0 -j RETURN
   -A DOCKER ! -i docker0 -p tcp -m tcp --dport 8888 -j DNAT --to-destination
   ↪  192.168.66.4:80
8  -A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
   -A FORWARD -o docker0 -j DOCKER
10 -A FORWARD -i docker0 ! -o docker0 -j ACCEPT
   -A FORWARD -i docker0 -o docker0 -j ACCEPT
12 -A DOCKER -d 192.168.66.4/32 ! -i docker0 -o docker0 -p tcp -m tcp --dport 80 -j
   ↪  ACCEPT
   -A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
14 -A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
```

Listing 10: Nutzung von `iptables` durch Docker.

Die Struktur des JSON aus Listing 11 zeigt es bereits: `NetworkSettings.Ports` referenziert ein Dictionary mehrerer Werte. `docker run` kann nämlich mehrere `--publish`-Parameter verarbeiten.[40]

[40] Warum `.Ports.<no>/<proto>` ebenfalls ein array referenziert, sei dem geneigten Leser anhand der Networking Dokumention [41] zur Übung empfohlen.

[41] Docker Inc. (2019a). Container Networking. https://docs.docker.com/config/containers/container-networking/

```
  docker inspect <name|id> \
| jq '.'
  docker inspect <NGINXREVERSEPROXY> \
| jq '.[].NetworkSettings.Ports'

{
  "80/tcp": null,
  "8441/tcp": [
    {
      "HostIp": "0.0.0.0",
      "HostPort": "8441"
    }
  ],
  "8442/tcp": [
    {
      "HostIp": "0.0.0.0",
      "HostPort": "8442"
    }
  ],
  "8443/tcp": [
    {
      "HostIp": "0.0.0.0",
      "HostPort": "8443"
    }
  ]
}
```

Listing 11: Nutzung von `iptables` durch Docker.

Literatur

Literatur

Alpine Linux Development Team. (2019). Alpine Linux. Small. Simple.Secure. https://alpinelinux.org

CoreOS, Inc. (2019). Quay [builds, analyzes, distributes] your container images. https://quay.io

Docker Inc. (2018). Docker Documentation. https://docs.docker.com/

Docker Inc. (2019a). Container Networking. https://docs.docker.com/config/containers/container-networking/

Docker Inc. (2019b). Docker run reference. https://docs.docker.com/engine/reference/run/

Docker Inc. (2019c). dockerhub. Build and Ship any Application Anywhere. https://hub.docker.com

Docker Inc. (2019d). Manage data in Docker. https://docs.docker.com/storage/volumes/

docker pull(1) - Pull an image or a repository from a registry. (2018). https://docs.docker.com/engine/reference/commandline/pull/. Docker Inc.

FreeBSD System Manager's Manual - jail(2). (2012). FreeBSD Project.

FreeBSD System Manager's Manual - jail(8). (2012). FreeBSD Project.

`github.com/docker-library`. (2019). Docker Official Images. https://github.com/docker-library/official-images

`github.com/fabric8io/docker-maven-plugin`. (2019). Maven plugin for running and creating Docker images. https://github.com/fabric8io/docker-maven-plugin

`github.com/hadolint`. (2018). Haskell Dockerfile Linter. https://github.com/hadolint

`github.com/stedolan/jq`. (2019). Command-line JSON processor. https://github.com/stedonlan/jq

Gregg, B. (2013). Virtualization Performance: Zones, KVM, Xen. http://dtrace.org/blogs/brendan/2013/01/11/virtualization-performance-zones-kvm-xen/.

Gritton, J. (2010). Running the New FreeBSD Jails. http://2010.eurobsdcon.org/fileadmin/fe_user/jamie/qvo7qI.pdf. EuroBSDCon.

`hashicorp/packer`. (2019). Hashicorp Packer - Docker Builder. https://www.packer.io/docs/builders/docker.html

jq - Command-line JSON Processor(1). (2018). http://manpages.ubuntu.com/manpages/bionic/man1/jq.1.html. Canonical - Ubutu Manpage Repository.

Kamp, P.-H. & Watson, R. N. M. (2000). Jails: Confining the Omnipotent Root. In *In Proc. 2nd Intl. SANE Conference*.

Linux Programmer's Manual: capabilities(7) - Overview of Linux Capabilities. (2018). http://man7.org/linux/man-pages/man7/capabilities.7.html. Linux man-pages Project.

Linux Programmer's Manual: cgroup_namespaces(7) - Overview of Linux Cgroup Namespaces. (2017). http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html. Linux man-pages Project.

Linux Programmer's Manual: cgroups(7) - Linux Control Groups. (2018). http://man7.org/linux/man-pages/man7/cgroups.7.html. Linux man-pages Project.

Linux Programmer's Manual: clone(2) - Create a Child Process. (2017). http://man7.org/linux/man-pages/man2/clone.2.html. Linux man-pages Project.

Linux Programmer's Manual: mount_namespaces(7) - Overview of Linux Mount Namespaces. (2018). http://man7.org/linux/man-pages/man7/mount_namespaces.7.html. Linux man-pages Project.

Linux Programmer's Manual: mq_overview(7) - Overview of POSIX message queues. (2017). http://man7.org/linux/man-pages/man7/mq_overview.7.html. Linux man-pages Project.

Linux Programmer's Manual: namespaces(7) - Overview of Linux Namespaces. (2018). http://man7.org/linux/man-pages/man7/namespaces.7.html. Linux man-pages Project.

Linux Programmer's Manual: network_namespaces(7) - Overview of Linux Network Namespaces. (2018). http://man7.org/linux/man-pages/man7/network_namespaces.7.html. Linux man-pages Project.

Linux Programmer's Manual: pid_namespaces(7) - Overview of Linux PID Namespaces. (2017). http://man7.org/linux/man-pages/man7/pid_namespaces.7.html. Linux man-pages Project.

Linux Programmer's Manual: setns(2) - Reassociate Thread with a Namespace. (2017). http://man7.org/linux/man-pages/man2/setns.2.html. Linux man-pages Project.

Linux Programmer's Manual: svipc(7) - System V interprocess communication mechanisms. (2016). http://man7.org/linux/man-pages/man7/svipc.7.html. Linux man-pages Project.

Linux Programmer's Manual: unshare(2) - Disassociate Parts of the Process Execution Context. (2018). http://man7.org/linux/man-pages/man2/unshare.2.html. Linux man-pages Project.

Linux Programmer's Manual: user_namespaces(7) - Overview of Linux User Namespaces. (2018). http://man7.org/linux/man-pages/man7/user_namespaces.7.html. Linux man-pages Project.

Linux Programmer's Manual: veth(4) - Virtual Ethernet Device. (2018). http://man7.org/linux/man-pages/man4/veth.4.html. Linux man-pages Project.

Linux User Commands: nsenter(1) - Run Program with Namespaces of Other Processes. (2013). http://man7.org/linux/man-pages/man1/nsenter.1.html. Linux man-pages Project.

Linux User Commands: unshare(1) - Run Program with some Namespaces Unshared from Parent. (2016). http://man7.org/linux/man-pages/man1/unshare.1.html. Linux man-pages Project.

lxc(7) - Linux Containers. (2019). https://linuxcontainers.org/lxc/manpages//man7/lxc.7.html. linux-containers.org.

LXD - System Container Manager. (2019). https://lxd.readthedocs.io/en/latest/. linux-containers.org.

Moskowitz, R., Karrenberg, D., Rekhter, Y., Lear, E. & de Groot, G. J. (1996). Address Allocation for Private Internets. https://rfc-editor.org/rfc/rfc1918.txt

Oracle Solaris 11.4 Reference Library - zones(7). (2016). https://docs.oracle.com/cd/E88353_01/html/E37853/zones-7.html. Oracle, Inc.

Price, D. & Tucker, A. (2004). Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of 18th Large Installation System Administration Conference (LISA)* (S. 241–254).

Pyinstaller Development Team. (2019). Pyinstaller. https://pyinstaller.org

Schenkeveld, P. (2010). Building Servers with NanoBSD, ZFS and Jails. In *Proceedings of AsiaBSDCon 2010*. http://2010.asiabsdcon.org/papers/abc2010-P4A-paper.pdf. Tokyo University of Science, Tokyo, Japan.

Stephen Dolan. (2019). *jq Manual (development version)*. https://stedolan.github.io/jq/manual/.

Teske, D. (2013). VIMAGE Jails on FreeBSD-8. http://devinteske.com/vimage-jails-on-freebsd-8.

Tripathi, S., Droux, N., Belgaied, K. & Khare, S. (2009). Crossbow Virtual Wire: Network in a Box. In *Proceedings of the 23rd Conference on Large Installation System Administration (LISA)* (S. 4–4). LISA'09. Baltimore, MD: USENIX Association. http://dl.acm.org/citation.cfm?id=1855698.1855702

Tripathi, S., Droux, N., Srinivasan, T. & Belgaied, K. (2009). Crossbow: From Hardware Virtualized NICs to Virtualized Networks. In *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures* (S. 53–62). VISA '09. Barcelona, Spain: ACM. doi:10.1145/1592648.1592658. http://doi.acm.org/10.1145/1592648.1592658

Zeeb, B. A. (2010). FreeBSD Jails - Notes Jotted on the Prison Wall. http://2010.eurobsdcon.org/fileadmin/fe_user/bz/ZDZZ8M.pdf. presented at EuroBSDCon 2010, Karlsruhe, Germany.

Zeeb, B. & Watson, R. (2010). The New VVorld – FreeBSD Jail Based Virtualization. http://www.bsdcan.org/2010/schedule/attachments/130_2010-bz-the-new-vvorld.pdf. presented at BSDCan, Ottawa, Canada.