Anpassung eines Rechnerverbundes an experimentspezifische Anforderungen mithilfe container-basierter Virtualisierung

Bachelorarbeit in Physik

von

William Ma

angefertigt im

III. Physikalischen Institut B

der

Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen

bei

Prof. Dr. Achim Stahl

20. September 2016

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 3

2 Docker **5**
- 2.1 Einführung . . . 5
- 2.2 Architektur . . . 6
- 2.3 Installation . . . 14
- 2.4 Nutzung . . . 14
- 2.5 Benchmark . . . 18
- 2.6 Sicherheit . . . 23

3 Testumgebung **29**
- 3.1 Wrapper (Schnittstelle) . . . 30
- 3.2 Installation und Konfiguration . . . 32
- 3.3 Testlauf . . . 36

4 HTCondor **37**
- 4.1 Einführung . . . 37
- 4.2 Docker . . . 38
- 4.3 Wrapper . . . 39
- 4.4 Testlauf . . . 39

5 Zusammenfassung **41**

A Monte-Carlo-Simulationen **43**

B Dockerfiles **44**

C Docker Compose **46**

Abkürzungsverzeichnis

API Application Programming Interface – Programmierschnittstelle

CERN Conseil Européen pour la Recherche Nucléaire – die Europäische Organisation für Kernforschung

CentOS Community Enterprise Operating System

CMS Compact Muon Solenoid

CMSSW CMS Software

GiB Ein Gibibyte – 1 GiB = 2^{30} Byte = 1.073.741.824 Byte

HTTP Hypertext Transfer Protocol

LDAP Lightweight Directory Access Protocol

LHC Large Hadron Collider

LXC Linux-Container

RHEL Red Hat Enterprise Linux

SL Scientific Linux

SLC Scientific Linux Cern

TLS Transport Layer Security

VM Virtuelle Maschine

WLCG Worldwide LHC Computer Grid

Kapitel 1

Einleitung

Am Teilchenbeschleuniger LHC am CERN werden jährlich 30 Petabyte[1] an Daten erzeugt, die nur mithilfe von Computer effektiv ausgewertet werden können. Obwohl lediglich ein Millionstel der Daten[2] weiterverarbeitet werden, werden trotzdem diese gewaltigen Datenmengen angesammelt. Diese müssen nicht nur gespeichert werden, sondern sollen auch den Wissenschaftlern weltweit schnell und im vollen Umfang zur Verfügung stehen.

Das CERN hat nicht die benötigte Rechenleistung bzw. finanziellen Mittel, um diese Aufgabe alleine zu bewältigen, weshalb das Worldwide LHC Computer Grid (WLCG) ins Leben gerufen wurde. Das WLCG ist ein globales Rechnernetz bestehend aus diversen Rechenzentren, die ihre Rechenleistung freigeben. Dabei wird das WLCG in „Tiers" von 0 bis 3 gegliedert, die verschiedene Ebenen der Verarbeitung und Analyse darstellen. Zuerst werden alle Rohdaten im CERN Data Centre (Tier 0) vorselektiert und gespeichert, um danach an die 13 Tier-1-Rechenzentren [3] im weltweiten Grid verteilt zu werden. Dort werden die Daten sofort verarbeiten oder werden dem Tier 2, bestehend aus Rechenzentren an Universitäten und wissenschaftlichen Einrichtungen, bereitgestellt. Die Tier-2-Zentren können einen Bruchteile der Daten mit ihrer Rechenleistung genauer analysieren. Dedizierte Computercluster, die zwar auch an das Grid angebunden sind, aber einen eingeschränkten Nutzerkreis haben, werden dem Tier 3 zugeordnet. [4]

Auch die RWTH Aachen stellt ein Tier-2-Rechenzentrum und Tier-3-Netz zur Verfügung, welche u. a. für die Datenauswertung des CMS-Experimentes eingesetzt werden. Momentan läuft an der RWTH auf den Tier-2/3-Zentren und den meisten Arbeitsplatzrechnern das Linux-Betriebssystem *Scientific Linux 6.8 (SL6)*[1]. Die Linux-Distributionen[2] Scientific Linux wurde, wie das

[1] `https://www.scientificlinux.org/`

[2] Linux-Distributionen bestehen aus dem Linux-Kernel (s. Abschnitt 2.6) und einer bestimmten Komposition von Softwarepaketen.

ebenfalls bekannte Betriebssystem CentOS[3], von der Distribution *Red Hat Enterprise Linux* (RHEL)[4] abgeleitet.

Es gibt auch eine vom CERN angepasste Distribution mit dem Namen Scientific Linux Cern (SLC), die sich nur geringfügig von Scientific Linux durch spezielle Konfigurationen für die Integration in die CERN-Rechnerumgebung unterscheidet. Zukünftig wird am CERN statt Scientific Linux das Betriebssystem CentOS unterstützt, welches dann den Namen *CentOS CERN* (CC) trägt.

Einerseits besteht das Interesse auf eine aktuellere Version umsteigen, um von den neuen Paketen und Funktionalitäten zu profitieren, andererseits möchte man eine von den Wissenschaftler für die Experimente validierte Distribution, wie z. B. SLC6 für das CMS-Experiment, benutzen. Zur Validierung werden diverse Kompatibilitätstests auf der verwendeten Distribution durchgeführt und geprüft, ob Berechnungen in den Analysen konsistent mit vorherigen Resultaten sind. Bei einer Betriebssystemaktualisierung werden viele Pakete aktualisiert, die nicht zwangsläufig mit der verwendeten Analysesoftware kompatibel sind, weshalb man zusätzlich die vorhandenen Programme zur Datenanalyse und dessen Abhängigkeiten mit dem neuen Betriebssystem abstimmen müsste. Alternativ kann eine aktuelle Linux-Distribution genutzt werden und gleichzeitig mithilfe einer Betriebssystemvirtualisierung eine validierte ältere Distribution mit gewünschter Laufzeitumgebung bereitgestellt werden. Zur Virtualisierung von Betriebssystemen ist die Betriebssystemvirtualisierung eine spezielle Art, die auf der Ebene des Betriebssystems stattfindet. Eines der führenden Virtualisierungsprogrammen dieser Art ist *Docker*[5].

Diese Arbeit bietet Einblicke in die Betriebssystemvirtualisierung mit *Docker* und dessen Funktionsweise. Es wird untersucht, inwiefern *Docker* für wissenschaftliche Berechnungen, speziell in einem Rechnernetz mit mehreren Nutzern, geeignet ist und welche Probleme es zu lösen gilt. Außerdem wird ein besonderes Augenmerk auf die Sicherheit und die Benutzerfreundlichkeit gelegt.

Zu Testzwecken wurde *Docker* im Rechnernetz des III. Physikalischen Instituts der RWTH eingerichtet und erste Testläufe darin absolviert.

[3] `https://www.centos.org/`

[4] `https://www.redhat.com/de/`

[5] `https://www.docker.com/`

1.1 Motivation

Mit *Docker* gelingt eine effiziente und einfache Virtualisierung vieler Betriebssysteme. Dagegen können virtuelle Maschinen (VMs) solche System nur mit hohen Leistungseinbußen [6, 7] virtualisieren. Also muss man sich zwischen einer starken Isolation oder hohen Effizienz entscheiden, hier zwischen VMs oder Betriebssystemvirtualisierung. Durch restriktive Sicherheitsrichtlinien können die Risiken einer Betriebssystemvirtualisierung für den Benutzer und Hostsystem auf ein akzeptables Maß gesenkt werden. Doch eine geringe Effizienz ist ein kritisches Problem für rechenintensive Analyseaufgaben, was gegen die Verwendung von VMs spricht. Ein weiteres Problem der VMs ist, dass die zugewiesenen Ressourcen wie z. B. Prozessorzeit und Arbeitsspeicher nicht dynamisch vergeben werden können, sondern immer einen festen Anteil blockieren.

Die vielversprechendsten Anwendungsfälle einer container-basierten Betriebssystemvirtualisierung für wissenschaftliche Zwecke sind, erstens die erwähnte Virtualisierung von validierten Distributionen, zweitens eine größere Vielfalt an Distributionen oder Programmen im Rechnernetz und drittens eine einfachere Nutzung externer opportunistischer Rechenressourcen.

1. Momentan sind einige Wissenschaftler, wie am CMS-Experiment, durch die verwendeten Analyseprogramme aus Kompatibilitätsgründen an SLC6 gebunden. Dieses Problem muss gelöst werden, weil die Betriebssysteme in absehbarer Zeit aktualisieren werden müssen. Die Analysesoftware auf das aktuelle Betriebssystem anzupassen, ist mit hohem Aufwand verbunden, der bei jeder weiteren Aktualisierung des Betriebssystems wieder anfällt. Dies ist viel umständlicher, als das validierte System über eine Virtualisierung weiterhin zu verwenden. VMs sind wegen der niedrigen Effizienz, insbesondere in den Größenordnungen der Rechenleistung, die für die Datenanalyse benötigt werden, keine Alternative.
   Am CERN existieren bereits Entwicklungen in Richtung Containerlösungen wie *Docker* [8, 9]. Auch Wissenschaftler am CMS-Experiment haben bereits angefangen *Docker* als Alternative zu evaluieren [10].

2. VMs sind in der Lage alle Betriebssysteme zu virtualisieren, die auf der Rechnerhardware lauffähig sind, wie z. B. Windows auf einem Linux Hostsystem und bieten deshalb mehr Auswahl als eine Betriebssystemvirtualisierung. Dies wird durch die Virtualisierung des kompletten Gastbetriebssystems, einschließlich des Kernels, realisiert, was zur Folge hat, dass das Hochfahren eines Betriebssystemes länger dauert als

es bei der Verwendung einer Betriebssystemvirtualisierung der Fall ist. Immerhin ermöglicht eine container-basierten Betriebssystemvirtualisierung die Verwendung mehrerer Linux-Distributionen, statt lediglich einer.

3. Durch die Flexibilität einer Virtualisierung kann ungenutzte Rechenleistung in verschiedenen Rechenzentren genutzt werden. Eine manuelle Installation eines Betriebssystems und dessen Konfiguration an die experimentspezifische Anforderungen durch die Administratoren der Rechenzentren wird auf wenig Zustimmung stoßen. Genauso werden auch keine Administratorrechte gewährt, damit der Nutzer die Konfiguration übernehmen kann. Eine container-basierte Betriebssystemvirtualisierung wäre eine effiziente Alternative, um schnell ein Betriebssystem ohne Administratorrechte an bestimmte Anforderungen anzupassen. Dies setzt voraus, dass entsprechende Virtualisierungsprogramme von den Administratoren bereitgestellt werden. Doch bisher besteht bei den Administratoren verschiedener Rechenzentren eine gewisse Skepsis gegenüber Betriebssystemvirtualisierungen aufgrund der Sicherheitsbedenken. VMs können eine starke Isolation gewährleisten, aber nur auf Kosten der Effizienz. Es wird eine zufriedenstellende Isolation von Container angestrebt, um die Akzeptanz für dieses Verfahrens zu gewinnen, damit diese opportunistischen Rechenressourcen nicht ungenutzt bleiben.

Kapitel 2

Docker

2.1 Einführung

Die Idee der container-basierten Virtualisierung ist nicht neu[1], doch diese Technologie bekam erst durch *Docker* breite Aufmerksamkeit. *Docker* vereint die vorhandenen Funktionalitäten zur Betriebssystemvirtualisierung und übernimmt damit viele Aufgaben, die zum Betreiben und zur Isolation der Container erforderlich sind. Die einfache Bedienung von *Docker* und simple Verwaltung von virtualisierten Betriebssystemen, die es bietet, hat maßgeblich zu dessen steigenden Beliebtheit beigetragen.

Man unterscheidet das Hostsystem und das Gastsystem. Das Hostsystem nimmt die Virtualisierung eines Betriebssystems vor, welches als das Gastsystem bezeichnet wird. Herkömmlich virtuelle Maschinen, wie z. B. VMware Player[2] oder VirtualBox[3], nehmen eine komplette Virtualisierung vor, die das komplette Betriebsystem und damit den Kernel (s. Abschnitt 2.6) umfasst. Im Gegensatz dazu wird bei der Betriebssystemvirtualisierung nur der Kernel des Hostsystems gebraucht, der Prozessgruppen gegeneinander in Container isoliert und weitere Pakete nach Bedarf hinzugefügt. Die Tatsache, dass alle Prozesse direkt auf dem Betriebssytem des Hostsystems laufen, ist ein Grund für die wesentlich höhere Effizienz im Vergleich zu VMs [6, 7]. Allerdings kann durch eine container-basierten Betriebssystemvirtualisierung im Gegensatz zur VM nur der Kernel des Hostsystems benutzt werden. Wenn lediglich ein Betriebssystem mit der Vorgängerversion des verfügbaren Kernels, z. B. eine ältere Distribution, virtualisiert werden soll, ist dies keine problematische Einschränkung. Dennoch birgt diese hardwarenahe Virtua-

[1] Linux Container wurden mit dem Kernel der Version 2.6.29 eingeführt, welche im Jahr 2009 veröffentlicht wurde.

[2] `http://www.vmware.com/de/products/player.html`

[3] `http://www.virtualbox.org/`

lisierung ein gewisses Sicherheitsrisiko, welches in jedem Fall berücksichtigt werden muss, damit Benutzer von Container nicht auf fremde Container oder sogar das Hostsystem übergreifen können. [11]

2.2 Architektur

Docker wird getrennt in *Docker Client* und *Docker Daemon*. Ein Benutzer kann alle Befehle zur Virtualisierung über den *Docker Client* an den *Docker Daemon* senden. Dieser *Docker* Dienst (*Docker Daemon*) setzt alle notwendigen Schritte zur Virtualisierung um, wie die Verwaltung der Systemabbilder (*images*) und die Erzeugung eines *Docker Containers*, welche in folgenden Unterabschnitten behandelt werden. *Docker Client* und *Docker Daemon* müssen nicht zwangsläufig auf demselben Rechner laufen, trotzdem ist der *Docker Client* in der Installation von *Docker* standardmäßig enthalten.

Docker hat ursprünglich externe Containerlösungen wie *libvirt* oder *LXC* genutzt, um mit dem Kernel zu interagieren. Seit der *Docker* Version 0.9 ist die Schnittstelle *libcontainer* der *Docker*-Entwickler der neue Standard, wodurch ohne Umwege direkt auf die Kernel-API zugegriffen werden kann (s. Abb. 2.1). Zwei für *Docker* wichtige Funktionen des Kernels sind *Namespaces* und *cgroups*[4] (s. jeweils Abschnitt 2.6).

SELinux fügt eine weitere Ebene der Zugriffskontrolle ein, die interessant wird, wenn man privilegierte Container braucht. Dieser Fall wird hier nicht betrachtet, da der Benutzer eines Containers, wie bisher im hiesigen Rechnernetz, keine administrativen Rechte benötigt, um die Laufzeitumgebung sinnvoll nutzen zu können. Weiterhin soll der Nutzer nur seine bisherigen Rechte haben, wenn er z. B. im Container auf von außen eingebundene Daten zugreift.

Das virtualisierte Betriebssystem wird in einem Image zusammengefasst, das in einem privatem oder öffentlichem *Docker* Registry (siehe unten) gespeichert und bereitgestellt werden.

[4]Abk. für *control groups*, welche ab der Linux-Kernel-Version 2.6.24 unterstützt werden.

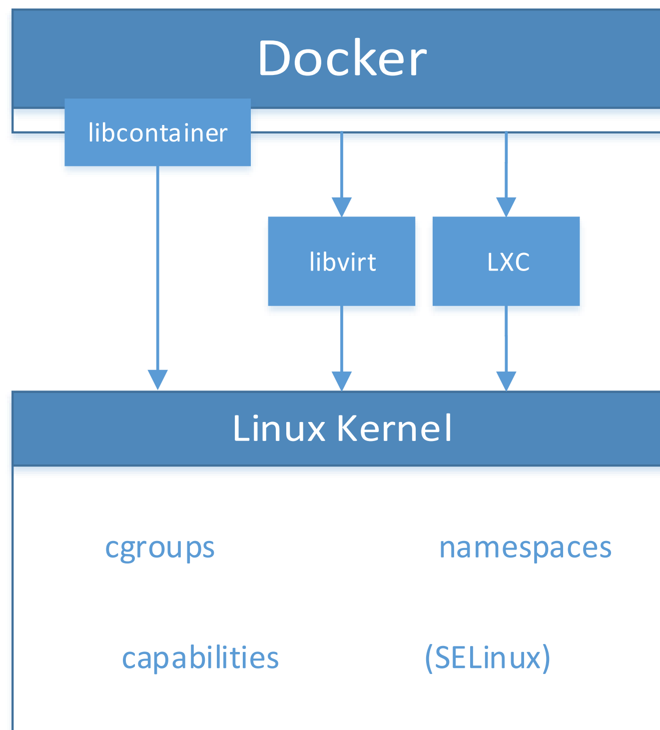


Abbildung 2.1: Schnittstelle zwischen *Docker* und Kernel [12]

Docker Images

Das zu virtualisierende Betriebssystem einschließlich aller Pakete und Konfigurationen sind in Abbildern, auch Images genannt, enthalten, die Grundlage sind, um eine Virtualisierung vorzunehmen. Also ermöglicht ein Image einen bestimmten Zustandes einer Umgebung exakt zu rekonstruiert. Unter *Docker* gibt es auch entsprechende Images, um einen Container mit dem virtualisiertem Betriebssystem zu starten. *Docker Images* können entweder aus einem Registry (siehe unten) geladen werden oder lokal erstellt werden. Diese bestehen aus verschiedenen Layers (Ebenen), die intern mithilfe von *UnionFS*[5] überlagert werden. Es gibt Image Layers, die schreibgeschützt sind, und Container Layers (siehe unten). Ein Image basiert immer auf einem Grundimage (base) und kann weitere Layer haben, die nur die Änderungen, z. B. Installation von Paketen, im Vergleich zum vorherigen Layer speichern. Alle Layer bis auf das Container Layer werden durch einen eindeutigen *content hash*

[5]Union File System: `https://www.gnu.org/software/hurd/hurd/translator/unionfs.html`

identifiziert. Einerseits hat dies den Vorteil, dass nach dem Übertragen eines Images die Integrität überprüft werden kann und andrerseits kann ein Layer zwischen verschiednen Images geteilt werden, die gleiche Layers benutzen. Der eindeutige Hash ermöglicht *Docker* bereits vorhandene Layer zu erkennen und wiederzuverwenden. Dies verhindert das identische Layer mehrfach gespeichert werden und trägt dazu bei, Speicherplatz zu sparen.

Änderungen, die innerhalb eines Containers vorgenommen wurden, können nur vom Hostsystem aus über den Befehl `docker commit` (s. Abschnitt 2.4) persistent gemacht werden. Dabei wird beim Aufruf eines `commit`-Befehls das gegenwärtige Container Layer gespeichert und dem Image zugeordnet. Für kleine Modifikationen ist dieser Befehl praktisch, aber für umfangreiche Images ist diese Methode nicht ratsam.

Erstens werden konkrete Änderungen, z. B. ausgeführte Befehle und Installationen, nicht dokumentiert, d. h., dass eine Rekonstruktion des Images im Nachhinein schwierig sein wird, wenn keine anderweitigen manuelle Dokumentation erstellt wurde.

Zweitens werden kontinuierlich Änderungen und Paketaktualisierungen eingespielt, die jedes Mal ein neues Layer erzeugt, welche *immer* zusätzlichen Speicherplatz einnimmt.

Drittens kann dies dazu führen, dass eine Vielzahl von Layers erstellt werden, die bei Dateizugriffen alle berücksichtigt werden müssen und sich damit die Zugriffszeiten erhöhen.

Stattdessen sollte man Images über sogenannte *dockerfiles* definieren. Ein *dockerfile* ist eine Textdatei mit verschiedenen Befehlen, die vom Docker-Dienst (*docker daemon*) auf einen Container angewendet werden, um daraus ein Image zu erstellt. Also kann, statt Gigabyte große Images hochzuladen, ein *dockerfile* verteilt werden, welches dem Nutzer zur Generierung des Images dient. Dies vereinfacht das Teilen und Verbreiten von bestimmten Konfigurationen, wie z. B. die besonderen Anforderungen vom CERN an die verwendeten Betriebssysteme.

Dockerfile-Befehle	
`FROM`	setzt das Grundimage
`ADD`	fügt Verzeichnis oder Datei hinzu
`RUN`	führt einen Befehl aus
`ENV`	setzt Umgebungsvariablen
`ENTRYPOINT`	startet Programm, wenn Container gestartet wird

Mit dem Befehl

```
docker build -t REGISTRY/IMAGENAME:VERSION PFAD
```

wird im Verzeichnis PFAD nach einem *dockerfile* gesucht, womit das Image erstellt wird. Das Argument `-t` vergibt diesem Image sofort einen Namen (tag). Üblicherweise wird das Registry dem Imagenamen und der Version vorangestellt, wie es im obigen Beispiel gezeigt ist.

In *dockerfiles* wird für jeden *RUN*-Ausdruck ein neues Layer erstellt. Da die Layer wiederverwendet werden können, wird bei jeder Änderung nur das Layer zur dazugehörigen Zeile und nachfolgende Befehle neu ausgeführt, statt das komplette Image neu zu erstellen. Also können in darauffolgenden Builds diese Layer aus dem Zwischenspeicher geladen werden. Einige Befehle, wie z. B. zur Aktualisierungen, möchte man eventuell trotzdem neu ausführen lassen. Dafür muss zum Befehl `docker build` das Argument `--no-cache` übergeben werden.

Man sollte beachten, dass nachfolgende Löschvorgänge das Image nicht mehr verkleinern, wenn diese in einem separaten *RUN*-Aufruf und damit in einem anderen Layer durchgeführt werden. Ein kleineres Image erhält man nur, wenn alle *RUN*-Ausdrücke zu einem zusammengefasst werden. Beispielsweise ist ein Image, welches mit dem *dockerfile*

```
FROM ubuntu:16.04
RUN apt-get update -y
RUN apt-get install -y vim
RUN apt-get purge -y vim
RUN apt-get autoremove -y
RUN apt-get clean all
# Dateigröße: 222 MB
```

erstellt wurde, größer als mit den folgenden Befehlen, obwohl keine Löschbefehle ausgeführt werden:

```
FROM ubuntu:16.04
RUN apt-get update -y
RUN apt-get install -y vim
# Dateigröße: 221,3 MB
```

Im ersten Fall wurden zusätzliche Layer hinzugefügt, die trotz der Löschvorgänge alle geladen werden müssen. Damit das Löschen die Imagegröße verringert und eine geringere Zahl an Layers generiert werden, fasst man die *RUN*-Ausdrücke zusammen:

```
FROM ubuntu:16.04
RUN apt-get update -y && apt-get install -y vim && \
    apt-get purge -y vim && apt-get autoremove -y && \
    apt-get clean all
# Dateigröße: 165,1 MB
```

Es konnten damit die Dateigröße des modifizierte *cern/slc6* (s. Anhang B) Images von 1780 MB auf 892,7 MB fast halbieren werden. Außerdem wurden *Dockerfiles* für Fedora und Ubuntu erstellt[6].

Docker Container

Ein Container isoliert Prozesse oder Prozessgruppen gegen das Hostsystem und andere Container. Bei der Betriebssystemvirtualisierung wird diese Isolation der Container vom Kernel des Betriebssystems vorgenommen (s. Abschnitt 2.6). Jeder Container erzeugt einen eigenen Prozess, der allen weiteren Prozesse, die im Container gestartet werden, übergeordnet ist. Mit einem Prozessmanager wie `htop`[7] können die Container und deren Subprozesse im Hostsystem überwacht werden. Wenn der Benutzer eines Containers mit `docker -u TESTUSER:TESTGRUPPE [...]` spezifiziert wurde, wird dieser und nicht der Administrator, der den *Docker Client* aufgerufen hat, entsprechend als Besitzer dieses Prozesses festgelegt. Innerhalb eines Containers wird gewährleistet, dass nur eigene Prozesse, also keine vom Hostsystem oder anderen Container, sichtbar sind. Durch eine restriktive Sicherheitseinstellung soll jeder Container nur Zugriff auf seine Prozesse und Daten haben. Ein Ausnahme sind externe Verzeichnisse, die in Container eingebunden werden. Die Nutzer der jeweiligen Container können entsprechend ihrer Zugriffsrechte auf dem Hostsystem die externen Daten lesen oder ändern. In den Abschnitten 2.6 und 3.1 werden diese Einstellungen genauer erläutert und einige Beispiele vorgeführt.

Ein *Docker Container* hat zusätzlich zum fundamentalen *Image Layer* ein

[6]Die Betriebssysteme sind angepasst an die Umgebung des Rechnernetzes des III. Physikalischen Instituts der RWTH.

[7]`http://hisham.hm/htop/`

sogenanntes *Container Layer* (s. Abb. 2.2). Diesem wird im Gegensatz zu den *Image Layer* nur eine zufällige UUID[8] zugeordnet. Während das zugrundeliegende Image von den Containern geteilt wird, werden für jeden Container alle Schreibvorgänge in einem eigenen *Container Layer* gespeichert.

Anfangs greifen alle Container auf dieselben Daten des Images zu. Erst bei Schreibvorgängen auf die Daten wird eine Kopie des Datensatzes für den Container angelegt (*copy-on-write*). Also bleiben alle Image Layer des Grundimages zu jedem Zeitpunkt unverändert.

Wenn ein Container entfernt wird, wird das *Container Layer* mit allen Änderungen, die nicht vorher mit einem *commit*-Befehl gespeichert wurden, verworfen. Dies betrifft keine Verzeichnisse, die aus dem Hostsystem in den Container eingebunden wurden, da die Änderungen außerhalb des Containers berücksichtigt und ebenda gespeichert werden.

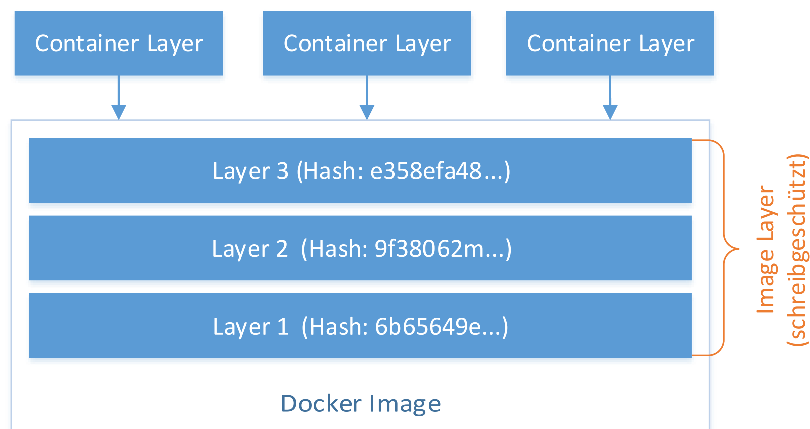


Abbildung 2.2: Container und Image Layers [13]

Docker Registry

Selbsterstellte *Docker Images* werden lokal gespeichert und sind deshalb nur auf einem Rechner vorhanden. Möchte man die Images verteilen kann ein *Docker Registry* genutzt werden. Die grundlegenden Aufgaben eines Registries sind das zentrale Speichern und die Bereitstellung der Images. Hierbei kann auf ein öffentliches Registry, wie das *Docker Hub*, zugegriffen werden oder ein eigenes privates Registry erstellt werden. *Docker Hub* ist ein Registry der *Docker*-Entwickler, welches nur für öffentliche Images kostenlos

[8] `https://de.wikipedia.org/wiki/Universally_Unique_Identifier`

ist. Dort können die gängigsten Linux-Distributionen gefunden werden und viele Komplettsystemen mit Anwendungen, wie z. B. einem Webserver, sind dort veröffentlicht. Images, die nicht öffentlich zugänglich sein sollen, sind auf *Docker Hub* kostenpflichtigen[9].

Alternativ kann ein eigenes privates Registry erstellt werden, um eine Auswahl eigener Images zu verwalten und nur im lokalen Netzwerk zu verteilen. Die *Docker*-Entwickler stellen ein Image, namens *registry*, bereit, womit ein Registry erstellt werden kann, welches in einem Container läuft. Im Folgenden werden die Container für das Registry mit *Docker Compose* organisiert, aber der Vollständigkeit halber wird das manuelle Vorgehen beschrieben.

Mit dem Befehl

```
docker run -d -p 5000:5000 --name registry registry:2.1
```

kann ein Container gestartet werden, der das Registry der Version 2.1 startet und nach außen auf den Port 5000 hört.

Intern speichert das Registry alle Images, genauer die Layer, in dem Ordner */var/lib/registry*. Es ist wartungsfreundlicher ein Verzeichnis im Hostsystems in den Container zu genau diesem Pfad einzubinden, damit die Images stattdessen dort gespeichert werden.

Der Zugriff auf das Registry lässt sich mit einer einfachen HTTP-Authentifizierung, welche später in abgeänderter Form verwendet wird, beschränken. Hier werden Benutzer des Registries und deren Passwörter in einer Textdatei *htpasswd* gespeichert. Die Verbindung zum Registry wird zusätzlich mit einer TLS Verschlüsselung abgesichert. Alle Dateien, inklusive der erzeugten oder vorhandenen TLS Zertifikate, müssen in den Container eingebunden werden und bestimmte Umgebungsvariablen gesetzt werden,[10] damit die Dateien ausgelesen werden [14]:

```
docker run -d \
    -p 5000:5000 \
    --restart=always \
    --name registry \
    -v /certs:/certs \
    -v /storage/registry:/var/lib/registry \
```

[9] `https://hub.docker.com/account/billing-plans/`

[10] Die Pfade müssen entsprechend ersetzt werden.

```
    -v /auth/htpasswd:/auth/htpasswd \
    -e "REGISTRY_AUTH=htpasswd" \
    -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
    -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
    -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
    -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
    registry:2.1
```

Docker Compose

Durch die Flexibilität der *Docker Container* bietet es sich an, Applikationen in einzelne Container aufzuteilen. Hier sind komplexe Konstellationen von mehreren Programmen gemeint, die eine Einheit bilden. Ein einfaches Beispiel ist ein Webserver in einem Container, der sich mit einer Datenbank in einem anderen Container verbindet. Diese Modularität können sich Entwickler zu Nutze machen, um mehrere Systeme in Form von Container virtualisieren können, welche schnell gestartet werden können. In einer Produktionsumgebung ist es auch sinnvoll, verschiedene Komponenten zu trennen, da ein Fehler in einem Container nicht das komplette System mit anderen Applikationen beeinträchtigt.

Wenn man ein komplexes System in verschiedene Container unterteilen möchte, sollte *Docker Compose*[11] genutzt werden. Damit lassen sich mehrere Container in einer Textdatei namens *docker-compose.yml* vordefinieren, die dann mithilfe von *Docker Compose* administriert und verwaltet werden können.

Die Container können dann mit

```
docker-compose up
```

gestartet werden. Nützliche Argumente sind `-d`, das die Container im Hintergrund als Dienst startet, und `-f`, welches eine Datei abweichend von der Standarddatei *docker-compose.yml* festlegt.

[11] https://docs.docker.com/compose/

2.3 Installation

Zur Verwendung von *Docker* wird eine Linux-Kernel-Version ab 3.10 empfohlen, die ab RHEL 7, CentOS 7 oder SL 7 ausgeliefert wird.[12] *Docker* ist auch auf RHEL 6.5 lauffähig, da Red Hat die notwendigen Änderungen an diesem Kernel[13] vorgenommen hat [15]. Auf andere Betriebssysteme, wie Windows oder OS X, müssen virtuelle Maschinen verwendet werden, wodurch man wiederum die Leistungseinschränkungen in Kauf nehmen muss.

Zuallererst wird das Paket *docker-engine* aus dem offiziellen[14] Repository[15] mit dem Paketmanager *yum* bezogen und installiert:

```
yum install docker-engine
```

Zusätzliche Informationen findet man in der Dokumentation von *Docker* [16, 17]. Danach kann der Docker-Dienst (*docker daemon*) mit `service docker start` gestartet werden. Der Administrator kann in der Kommandozeile über den *Docker Client* `docker` diesen Dienst ansprechen und darüber Container starten oder Images verwalten.

2.4 Nutzung

Dieser Abschnitt soll nicht das Handbuch für die Kommandozeile von *Docker* ersetzen, stattdessen werden nur grundlegende Befehle eingeführt. Eine Übersicht aller Befehle und ihrer Funktionen findet man in der Dokumentation[16] oder mit dem Befehl `docker --help`. Einige Kommandos nehmen noch weitere Optionen und Argumente an, welche mit `docker KOMMANDO --help` angezeigt werden können.

Der Programmaufruf von *docker* ist wie folgt aufgebaut:

```
docker [OPTIONEN] KOMMANDO [ARGUMENTE]
```

[12] CentOS ist im Grunde ein RHEL ohne den kostenpflichtigen Support von Red Hat. Das für wissenschaftliche Analysen optimierte SL basiert ebenfalls auf RHEL. Deshalb wird in dieser Arbeit nur Scientific Linux angesprochen, aber es ist alles direkt übertragbar auf CentOS und RHEL oder größtenteils auf andere Linux-Distributionen.

[13] Kernel-Version 2.6.32-431

[14] `https://yum.dockerproject.org/repo/main/centos/7/`

[15] Ein Repository liegt meist auf einem externen Server, der Pakete oder Programme verwaltet und diese ausliefert.

[16] `https://docs.docker.com/engine/reference/commandline/`

In den folgenden Tabellen sind einige nützlich Befehle aufgelistet:

Start eines Containers `docker run ... IMAGE`	
... `-it` ...	öffnet ein interaktives Terminal
... `-e` ...	setzt Umgebungsvariablen im Container
... `-v` ...	bindet Verzeichnisse in den Container ein (z. B. `docker run -v /home/:/home/`)
... `-w` ...	setzt das Arbeitsverzeichnis
... `-u` ...	definiert Benutzer und Gruppe
... `--name` ...	benennt einen Container um

Verwaltung von Container `docker` ...	
... `ps`	zeigt aktive Container an (`docker ps -a` zeigt alle Container)
... `rm CONTAINER`	entfernt einen oder mehrere Container
... `stop CONTAINER`	hält einen oder mehrere Container an
... `start CONTAINER`	startet gestoppte(n) Container
... `attach CONTAINER`	verbindet sich mit einem Container

Verwaltung von Images `docker ...`	
... `build PFAD`	erstellt das Image aus einem *dockerfile*
... `commit CONTAINER`	speichert die vorgenommenen Änderungen
... `images`	zeigt lokal vorhandene Images
... `tag IMAGE IMAGENAME`	ändert `tag` des Images
... `pull IMAGE`	lädt Image herunter
... `push IMAGE`	lädt Image auf ein Registry
... `rmi IMAGE`	entfernt Image

Administration `docker ... CONTAINER`	
... `stats` ...	zeigt den Ressourcenverbrauch (CPU, RAM, I/O) an
... `logs` ...	zeigt die Ausgaben an (STDOUT und STDERR)
... `inspect` ...	zeigt alle eingestellten Parameter eines Containers

Netzwerke

Jeder Container, genauer jeder *Namespace* (s. Abschnitt 2.6), hat seine eigenen virtuellen Netzwerkschnittstellen[17] (s. Abb. 2.3). Auf dem Hostsystem wird bei der Installtion von *Docker* die virtuelle Netzwerkschnittstelle *docker0* eingerichtet, die direkt mit den Netzwerkschnittstellen des Hostsystems verbunden ist. Über weitere virtuelle Netzwerkschnittstellen kann nun ein Verbindung zwischen *docker0* und den Netzwerkschnittstellen der Container hergestellt werden. Jedem dieser virtuellen Netzwerkschnittstellen[18] wird automatisch eine eigene MAC- und IP-Adresse zugewiesen.

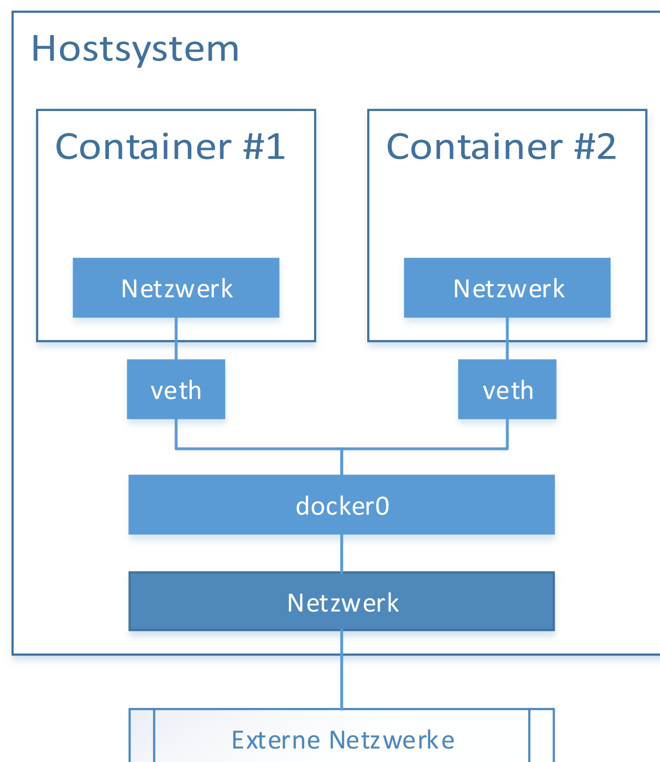


Abbildung 2.3: Netzwerkschnittstellen

Es gibt drei von *Docker* vordefinierte Netzwerke, die mittels `docker run --net NETZWERK` eingestellt werden können. Der Befehl `docker network ls`

[17] Virtuelle Netzwerkschnittstellen sind, anders als physische Netzwerkschnittstellen, nur in der Software realisiert.

[18] Auf dem Hostsystem können diese virtuellen Netzwerkschnittstellen an ihrer Bezeichnungen, die mit „veth“ beginnen, erkannt werden.

zeigt alle verfügbaren Netzwerke, welche anfänglich nur die Netzwerke *bridge*, *none* und *host* sind.

Standardmäßig werden alle Container in das Netzwerk *bridge* hinzugefügt, welches mit dem virtuelle Netzwerkschnittstelle *docker0* verbunden ist. Darüber hinaus können auch alle Netzwerkschnittstellen des Hostsystems dem Container bereitgestellt werden, wenn das Netzwerk *host* ausgewählt wird. Mit dieser Einstellung können grafische Oberflächen im Container mit dem X-Server des Hostsystems kommunizieren.

Das Netzwerk *none* bietet keine Netzwerkschnittstellen des Hostsystems dem Container an, weshalb dann keine Verbindung zu anderen Container oder externen Netzwerken möglich ist. Diese Option ist nützlichen, wenn keine weiteren Verbindungen benötigt werden oder der Anwendungen im Container komplett gegen andere externe Netzwerke isoliert werden sollen.

Alle Container in einem gemeinsamen Netzwerk, mit Ausnahme des Netzwerkes *none*, sind in der Lage untereinander zu kommunizieren, wenn die Adresse des Zieles bekannt ist. Möchte man die Container direkt miteinander verbinden, so muss dies vom *docker daemon* durchgeführt werden, weil er im Gegensatz zu den Container einen Überblick über alle Netzwerkschnittstellen hat. Dies kann während der Erstellung eines Containers explizit über das Argument `--link` angegeben werden, z. B.:

```
docker run --link ANDERER_CONTAINER IMAGENAME
```

Außerdem kann ein Port mit dem Argument `-p` vom Hostsystem auf einen bestimmten Port des Containers abgebildet werden. Zum Beispiel werden durch den Befehl

```
docker run -p 80:8001 IMAGENAME
```

alle anfragen auf den Port 80 des Hostsystems auf den Port 8001 im Container umgeleitet.

OpenLDAP

OpenLDAP ist eine freie Implementierung vom Lightweight Directory Access Protocol (LDAP), welches auch im Rechnernetz des III. Physikalischen Instituts der RWTH verwendet wird. LDAP ist ein Protokoll, womit Nutzerinformationen von einem LDAP-Server angefragt werden können. Unter anderem

werden Benutzerkonten mit den dazugehörigen Benutzer- und Gruppenidentifikatoren (UID und GID) gespeichert. Der Nutzer kann sich an seinem Arbeitsplatzrechner über den LDAP-Server anmelden, um seine Identifikatoren (UID und GID) zu setzen, sodass er Zugriff auf seine Dateien, wie das Homeverzeichnis[19], erhält.

Wenn die benutzerdefinierten Konfigurationen zusätzlich auf einem zentralen Server liegen, kann mit der Zuhilfenahme von LDAP ein homogenes Rechnernetz aufbauen, wo der Nutzer auf jedem dieser Rechner die gleiche Einstellung und Umgebung vorfindet. Dadurch muss der Benutzer nicht auf jedem Computer, den er nutzen möchte, die Konfigurationen neu einrichten. Diese Vorteile möchte man bei der Verwendung von Container bewahren, weshalb OpenLDAP auch innerhalb eines Containers konfiguriert werden muss.

Im SLC6 Container kann der LDAP-Client mithilfe des Paketes *authconfig-gtk* konfiguriert werden:

```
authconfig --enableldap --enableldapauth \
    --ldapserver="ldaps://ldap.physik.rwth-aachen.de/" \
    --ldapbasedn="dc=physik,dc=rwth-aachen,dc=de" --update
```

Wenn auf dem Hostsystem ein LDAP-Client aktiv ist, werden Konflikte mit dem LDAP-Client im Container entstehen. Beide Clients werden versuchen den nslcd-Socket[20] mit der gleichen Adresse zu beanspruchen. Dies kann man beheben, indem man den Socket des Hostsystems im Container verwendet, also den Socket aus */var/run/nslcd/socket* in den Container einbindet. Außerdem müssen die Zertifikate für LDAP (hier in */etc/openldap/cacerts*) im Container vorhanden sein.

Die Konfiguration des LDAP-Clients für die Images Fedora 24 und Ubuntu 16.04 sind im Anhang (s. Anhang B) zu finden.

2.5 Benchmark

Ein entscheidender Vorteil von *Docker* gegenüber der vollständigen Virtualisierung ist die hohe Effizienz. Bei den enormen Datenmengen und rechenintensiven Aufgaben möchte man den Leistungsverlust durch eine Virtualisierung nicht zuletzt aus finanziellen Gründen minimieren. Die Tests messen die Prozessorleistung, den Datendurchsatz des lokalen Datenträgers und die

[19] Im hiesigen Netzwerk sind die Homeverzeichnisse zentral auf einem Server gespeichert.

[20] nslcd ist ein Dienst, der entsprechend den Konfigurationen die LDAP-Anfragen an den Server sendet (s. `http://linux.die.net/man/8/nslcd`.)

Netzwerklatenz. Eine effiziente Nutzung des Prozessors ist wichtig, da viele wissenschaftlichen Analysen primär rechenintensiv sind. Oft muss während dieser Analysen Dateien, z. B. Ergebnisse, lokal gelesen oder geschrieben werden, weshalb über einen Test der Datendurchsatz verglichen wird. Außerdem wird die Netzwerklatenz bestimmt, weil die Daten der Experimente selten auf dem lokalen Rechner liegen, sondern auf externen Server.

Der Testrechner wird von zwei AMD Opteron™ 6172 Prozessoren angetrieben, die jeweils 12 Kerne mit einer Taktung von 2,1 GHz haben. Der Arbeitsspeicher setzt sich aus acht DDR3 1333 MHz Riegeln mit jeweils 8 GiB Kapazität zusammen.

Dieser Abschnitt fasst die diversen Leistungstests zusammen, dem die Container und das native System unterzogen wurden.

Prozessor

Zur Leistungsbestimmung des Prozessors wird *Phoronix Test Suite*[21] genutzt. *Phoronix Test Suite* übernimmt sowohl die Installation der Tests als auch deren Durchführung. Es bietet eine Vielzahl verschiedener Tests[22], die automatisch bezogen werden können.

Alle Tests können automatisch nacheinander im *batch*-Modus ausgeführt werden. Dieser Modus muss vorher mit `phoronix-test-suite batch-setup` eingerichtet und mit `phoronix-test-suite batch-benchmark ...` aufgerufen werden. Wenn der *batch*-Modus so konfiguriert wurde, dass nicht alle Tests gestartet werden, können diese während der Eingabeaufforderung gewählt werden. Dann können beispielsweise im Testpaket *scimark2* die untergeordnete Tests gestartet werden.

Die folgenden Tests wurden mit *Phoronix Test Suite* ausgeführt:

- C-Ray (c-ray)
- Gzip (compress-gzip)
- Scimark2: Monte Carlo (scimark2)
- Scimark2: FFT (scimark2)

C-Ray ist ein allgemeiner Test zur Bestimmung der Prozessorleistung über „Raytracing“ (dt. Strahlenverfolgung). Dieses Verfahren wird zwar primär in 3D-Anwendungen gebraucht, doch hat es sich auch als Benchmark etabliert.

[21] `http://www.phoronix-test-suite.com/`
[22] `http://openbenchmarking.org/tests/pts`

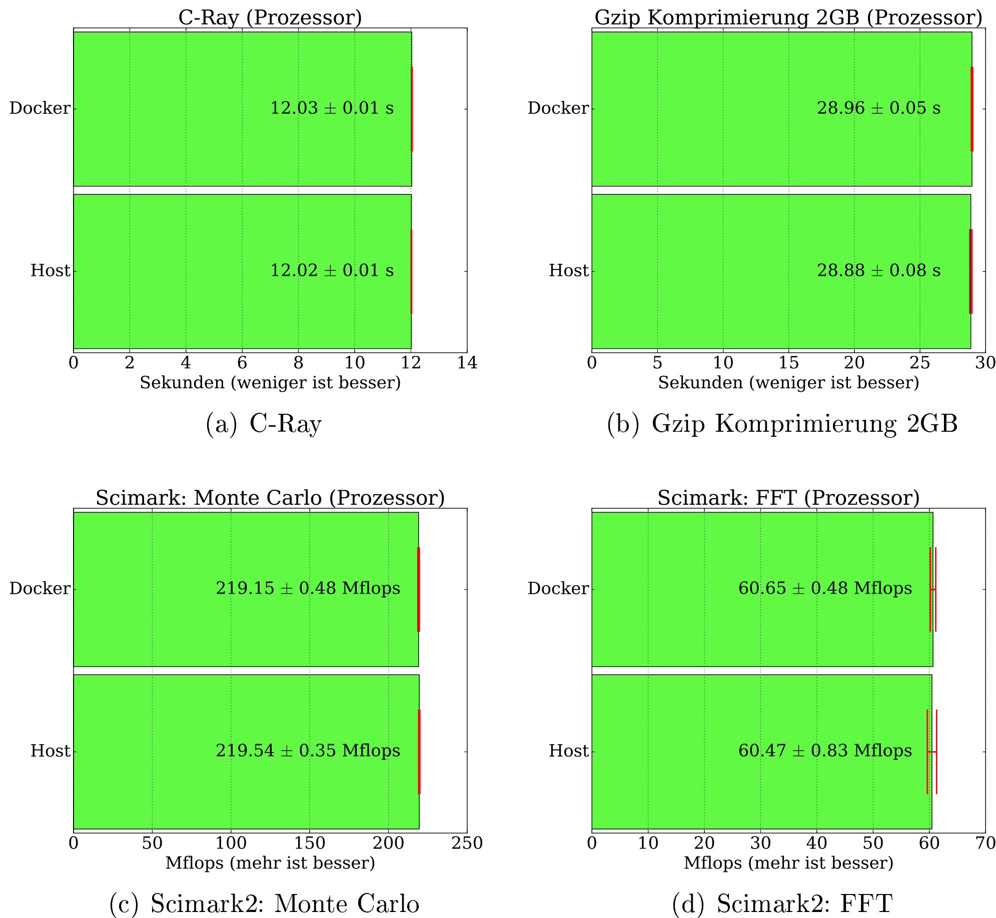
Gzip ist ein Dateiformat, das von dem gleichnamigem Komprimierungsprogramm *gzip* verwendet wird. Komprimierung, z. B. in das Format *gzip*, wird zur Verringerung der lokalen Datengröße und bei Datenübertragungen über ein Netzwerk eingesetzt. Eine Komprimierung eignet sich ebenfalls gut für die Messung der Leistung des Prozessors, weil sie größtenteils diesen belastet.

Das Paket *scimark2* liefert speziell angepasste Leistungstests für wissenschaftliche Analysen und numerische Berechnungen, wie eine Monte-Carlo Simulation und eine FFT[23]. Die Monte-Carlo-Methode (s. Anhang A) ist ein wichtiges Werkzeug in der Teilchenphysik und findet Anwendung in viele Berechnungen und Simulationen. Der entsprechende Test benutzt die Monte-Carlo Methode, um π zu bestimmen.

Es wurde sichergestellt, dass diese Tests (s. Abb. 2.4) lediglich den Prozessor belasteten ohne andere Komponenten wie Datenträger oder Arbeitsspeicher komplett auszulasten, um eine mögliche Verzerrung der Ergebnisse zu verhindern.

---

[23]fast Fourier transform, dt. schnelle Fourier-Transformation



(a) C-Ray

(b) Gzip Komprimierung 2GB

(c) Scimark2: Monte Carlo

(d) Scimark2: FFT

Abbildung 2.4: Tests zur Prozessorleistung

Die *Phoronix Test Suite* führt automatisch drei Durchläufe durch und gibt den Fehler auf den Mittelwert an. Nach längerer Messung (zehn Tests à drei Durchläufe) kann jedoch festgestellt werden, dass der Mittelwert stärker abweicht, weshalb der Fehler aus der kompletten Messreihe manuell berechnet wurde. Durchgehend zeigen die Benchmarkergebnisse, dass der *Docker Container* nicht signifikant langsamer ist.

I/O-Leistung

Diese Tests bestimmen den Datendurchsatz der lokalen Festplatte, also die Lese- und Schreibleistung (I/O performance). Dafür wurde das Benchmark-Programm *sysbench*[24] benutzt. Zur Vorbereitung werden insgesamt 1024 MB

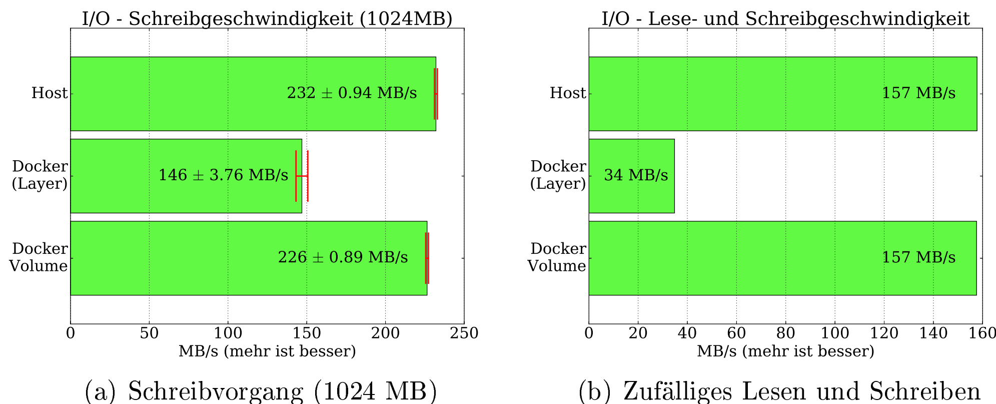
[24] `https://github.com/akopytov/sysbench`

an Daten geschrieben, wobei sie in 128 Dateien mit einer Größe von je 8MB aufgeteilt werden. Das Testsystem benutzt das Dateisystem XFS[25] und die Festplatten sind in einem RAID-1-System[26].

Die erste Messreihe (s. Abb. 2.5(a)) bestand aus sequentiellen Schreibvorgängen von jeweils 128 Dateien. Dieser wurde auf dem Host und im *Docker Container* durchgeführt, wobei im Container zusätzlich Schreibvorgänge in ein eingebundenes Verzeichnisse (*Docker Volume*) getestet wurde. Die 128 Dateien wurden in jeder Umgebung für diese Testreihe zehnmal neu erstellt.

In der zweiten Messreihe (s. Abb. 2.5(b)) wurden auf zufälligen Speicherpositionen gelesen und geschrieben. Die I/O-Operationen bestehen zu 60% aus Lese- und zu 40% aus Schreibvorgängen. Der Datendurchsatz wurde jeweils über einen Zeitraum von 600 s aufgenommen und gemittelt.

Während der Tests wurde sichergestellt, dass der Prozessor nicht komplett ausgelastet wurde. Beim Starten einiger Tests kann die Prozessorauslastung kurzzeitig auf bis zu 85% hochschnellen, doch nähert sie sich im Laufe des Tests nicht einer Auslastung von 100%.



(a) Schreibvorgang (1024 MB)

(b) Zufälliges Lesen und Schreiben

Abbildung 2.5: Tests zur I/O-Leistung

Zuallererst kann man sehen, dass der Datendurchsatz im Hostsystems und eingebundenen Verzeichnis im Container ähnlich hoch sind. In der ersten Testreihe ist der Schreibvorgang in das eingebundene Verzeichnis im Container nur $2,6 \pm 0,6\%$ langsamer als auf dem Hostsystem und in der zweiten Reihe gibt es keinen Unterschied. Dazu ist anzumerken, dass in der zweiten Messreihe, zumindest auf den angegebenen Stellen, keine Schwankung beobachtet werden konnte.

[25]`https://de.wikipedia.org/wiki/XFS_(Dateisystem)`

[26]`https://de.wikipedia.org/wiki/RAID#RAID_1`

Allerdings ist deutlich zu sehen, dass ein Schreib- oder Lesevorgang in den Container und damit in das Container Layer wesentlich langsamer ist. Der große Leistungsunterschied ist damit zu erklären, dass *Docker* alle Schreibvorgänge aufzeichnet, damit alle Veränderungen zwischen den *Layers* (s. Abschnitt 2.2) rekonstruiert werden können. Besonders in der zweiten Messreihe, wo über eine Million zufällige Schreibvorgänge im Container Layer vorgenommen werden, wird dies besonders deutlich. Stattdessen kann man für schreibintensive Aufgaben ein Verzeichnis über ein *Docker Volumes* in den Container einbinden, welches eine höhere Leistung erreicht und damit geeigneter ist.

Netzwerklatenz

Die Netzwerklatenz zwischen Rechner ist die Dauer zwischen der Netzwerkanfrage und dessen Antwort. Es wird das Programm *ping*[27] benutzt, um eine kurze Anfrage an die Zieladressen zu senden.

Als Ziel wurde der DNS-Server von Google und von der RWTH festgelegt. Sowohl im Container als auch auf dem Hostsystem wurden 1000 Anfragen abgesendet und die Latenzen gemittelt.

Ping	Google DNS (8.8.8.8)		RWTH DNS (134.130.4.1)	
	Host	Docker	Host	Docker
Latenz in ms	10,309 $\pm$ 0,030	10,381 $\pm$ 0,041	0,330 $\pm$ 0,045	0,431 $\pm$ 0,036
min in ms	10,2	10,3	0,231	0,322
max in ms	10,4	10,5	0,579	0,805

Tabelle 2.1: Vergleich der Netzwerklatenz

Die Netzwerklatenz ist im Container nur etwa eine Zehntelmillisekunde höher als auf dem Host. Diese Differenz ist nicht signifikant, da der Unterschied lediglich 1,4 σ (Google DNS) bzw. 1,8 σ (RWTH DNS) entspricht. In den meisten Anwendungsbereichen ist ohnehin eine zusätzliche Netzwerklatenz im Submillisekundenbereich vernachlässigbar.

2.6 Sicherheit

Es gibt diverse Maßnahmen und Konfigurationen, um die Sicherheit zu erhöhen und die Angriffsfläche zu minimiert. In diesem Abschnitt werden einige

[27] `http://linux.die.net/man/8/ping`

wichtige Komponenten des Linux-Kernels wie *Namespaces*, *Cgroups* und *Capabilities* untersucht. [18] Später (s. Abschnitt 3.1) werden diese Maßnahmen durch einen Wrapper[28] durchgesetzt.

Kernel

Der Kernel hat direkten Zugriff auf die Hardware und ist somit der fundamentale Softwarebestandteil des Betriebssystems. Programme werden vom Kernel gestartet oder beendet und können über ihn Ressourcen, wie z. B. Prozessorzeit, Zugriffsrechte und Arbeitsspeicher, anfragen. Das Einhalten der Ressourcen- und Zugriffsbeschränkungen ist einer der Aufgaben des Kernels, weshalb er insbesondere bei der Betriebssystemvirtualisierung eine zentrale Rolle einnimmt. Wichtige Funktionalitäten, wie *Namespaces*, *Cgroups* oder *Capabilities*, werden auch vom Kernel bereitgestellt. Bei der containerbasierten Virtualisierung wird lediglich der Kernel des Hostsystems genutzt, weshalb es besonders bei der Betriebssystemvirtualisierung wichtig ist, den Kernel aktuell zu halten, damit die Mechanismen zur Isolation der Container korrekt funktionieren.

Namespaces

Eine wichtige Komponente für Container sind sogenannte *Namespaces* des Linux-Betriebssystems. Durch *Namespaces* kann eine wichtige Funktion der Container, die Isolation von Prozessen und Prozessgruppen, umgesetzt werden. Es gibt verschiedene Arten von *Namespaces*, die jeweils Prozesse auf der Ebene des Betriebssystems voneinander trennen kann. Jeder Prozess kann in einem eigenen *Namespace* laufen und dabei auf die globalen Ressourcen des Systems zugreifen, ohne dass der Prozess in einem *Namespace* andere Prozesse in einem andern *Namespace* sehen kann. Dadurch kann dieser Prozess nicht auf andere *Namespaces* zugreifen oder andere Prozesse beeinflussen. Eine Betriebssystemvirtualisierung nutzt diese Funktion, um die Container in ihrem eigenen Satz von *Namespaces* zu isolieren. *Docker* als container-basierte Virtualisierungssoftware macht ebenfalls von *Namespaces* Gebrauch, allerdings übernimmt *Docker* die Erstellung und Verwaltung der *Namespaces*, wodurch dies nicht mehr manuell vom Benutzer eines Containers getan werden muss. Im Folgenden werden einige *Namespaces* vorgestellt, die essentiell für die Isolation von Container sind.

PID Namespace – Prozesse unter Linux erhalten einen numerischen und

[28] Ein Wrapper bedeutet übersetzt „Verpackung“ und hat die Funktion mit einem anderen Programm auf eine vorbestimmte Weise zu interagieren.

eindeutigen Prozess-Identifikator (PID). Ein fundamentale Prozess unter Linux ist der *init*-Prozess, welcher beim Starten als erstes gestartet wird und die PID-Nummer 1 trägt. Alle weitere Prozesse, die gestartet werden, werden diesem untergeordnet. Wenn ein Container von *docker* gestartet wird, wird ein sogenannter *PID Namespace* erstellt, welcher die PIDs gegeneinander trennt, sodass dieselbe PID in verschiedenen *Namespaces* verwendet werden kann. Der erste Prozess in einem Container, der nicht zwangsläufig ein *init*-Prozess sein muss[29], hat in seinem *PID Namespace* ebenfalls die PID-Nummer 1. Im globalen *PID Namespace* ist dieser Prozess nicht als PID 1 erkennbar, stattdessen wurde ihm global eine andere PID zugewiesen. Alle Prozesse, die in diesem Container gestartet werden, sind diesem Prozess untergeordnet, daher muss in einem Container ein Prozess gestartet werden. Wird dieser Prozess geschlossen, werden alle untergeordneten Prozesse und damit der Container beendet. Auf dem Hostsystem bleiben alle Prozesse und *Namespaces* sichtbar[30], während Prozesse in einem untergeordnetem *Namespace* lediglich diesen und dessen Prozesse einsehen können.

User Namespace – Vergleichbar zur PID hat jeder Benutzer eine eindeutige Identifikationsnummer, die UID. Das Betriebssystem überprüft bei den Prozessen die UID des Besitzers und entscheidet welche Aktionen dieser Prozess durchführen und auf welche Daten zugegriffen werden darf. Wenn ein *Docker Container* gestartet wird, besitzt der Prozess in diesem Container die UID des Ausführenden[31]. Standardgemäß ist dies der Administrator mit der UID-Nummer 0, die der Prozess im Container hat. Dies bedeutet, dass der Prozess, falls er Zugriff auf das Hostsystem erlangt, ebenfalls Administratorrechte hätte. Um dies zu verhindern, kann *Docker* einen *User Namespace*[32] für den Container erzeugen, welches auf dem Hostsystem die UID auf eine andere Nummer ungleich Null abbildet, während der Prozess im Container aus dessen Sicht die UID 0 hat, also Administrator ist. Auf dem Hostsystem hat er keine Administratorrechte.

Network Namespace – Jeder Prozess in einem bestimmten *Network Namespace* hat seine eigenen virtuellen Netzwerkschnittstellen. Wenn ein Prozess eines Container dem globalen *Network Namespace* des Hostsystems

[29]Einer der Stärken einer Betriebssystemvirtualisierung ist, dass kein komplettes Betriebssystem gestartet werden muss, sondern die gewünschten Programme sich direkt ausführen lassen.

[30]Diese können mit einem Prozessmanager, wie *htop*, angezeigt werden

[31]Falls ein *Docker Container* mit dem Argument `-u` erstellt wird, wird die angegebene UID verwendet.

[32]Unter SL7 müssen *User Namespaces* im Kernel manuell aktiviert werden. Im Weiteren werden keine *User Namespaces* benutzt, weil alle Container so gestartet werden, dass man innerhalb dieser kein Administrator ist.

zugeordnet ist, kann er beispielsweise einen Port belegen, sodass keine anderen Prozesse diesen mehr benutzen können, obwohl sie sich gegenseitig nicht sehen können. Deshalb erzeugt *docker* für jeden neuen Container einen eigenen *Network Namespace*, damit sie sich nicht gegenseitig behindern. Zu Beginn sind die Netzwerkschnittstellen im *Network Namespace* nicht aktiv. Um eine Verbindung mit einem anderen *Network Namespace* aufzubauen, erzeugt *docker* automatisch eine weitere virtuellen Netzwerkschnittstellen, die mit *docker0* verbunden wird. Damit erhält der Container Zugriff auf das Netzwerk des Hostsystems.

Mount Namespace – In diesem *Namespace* kann ein Prozess ein eigenes Wurzelverzeichnis und private Einbindungen (mounts) haben, die Verzeichnisse in anderen *Namespaces* nicht beeinflussen. Also kann z. B. jeder *Namespace* ein eigenes privates */tmp* Verzeichnis anlegen. [19, 20]

Control groups

Die Aufgabenbereiche von *cgroups* sind die Ressourcenüberwachung und Ressourcenkontrolle von Prozessen oder Prozessgruppen durch bestimmte Parameter. Im Falle einer Betriebssystemvirtualisierung sind sie erforderlich, damit jeder Container bzw. die dazugehörige Prozessgruppe nur seine zugewiesenen Ressourcen nutzen kann. Für Systemressourcen wie CPU, Arbeitsspeicher, Netzwerk- und Datenträgerzugriffe gibt es eigene *cgroups* Untersysteme[33]. Jeder Prozess kann jeweils eine *cgroup* in mehreren Untersystemen haben, also eine *cgroup* für jede Systemressource. Innerhalb dieser Untersystemen sind die *cgroups* hierarchisch geordnet, das heißt, dass alle Parameter zur entsprechenden Systemressource an untergeordnete *cgorups* weitergegeben werden[34]. Nachdem eine *cgroups* erstellt wurde, kann ein Prozess dieser Gruppe zugeordnet werden und alle Parameter zur Ressourcenbeschränkung werden durchgesetzt. Die gemessene Ressourcennutzung ist auch in diesen *cgroups* zu finden. Alle Werkzeuge, um mit *cgroups* zu arbeiten, erhält man mit dem Paket *libcgroup-tools*. Praktischerweise werden *cgroups* automatisch von *Docker* hinzugefügt und verwaltet. [21, 22]

Capabilities

Jeder Prozess hat einen Satz von Privilegien, um bestimmte Aktionen auszuführen, die vom System überwacht werden. Linux unterteilt seit Kernel-

[33] Diese findet man im Pseudo-Dateisystem unter */sys/fs/cgroup* oder bei älteren Distributionen unter */cgroup*.

[34] Wenn manuell in einer Systemressource unter */sys/fs/cgroup* ein Ordner erstellt wird, hat man eine neue *cgroup* hinzugefügt und alle Parameter werden übernommen.

Version 2.2 die Privilegien in eine Vielzahl an *Capabilities*[35], z. B. ein Setuid-Bit auf eine Datei zu setzen (CAP_SETUID), Netzwerkschnittstellen des Systems zu kontrollieren (CAP_NET_ADMIN) oder einen Neustart durchzuführen (CAP_SYS_BOOT). Jedes dieser Privilegien kann nach Bedarf an Prozesse vergeben werden. Somit können für Prozesse Privilegien, falls diese benötigt werden, einzeln freigegeben werden, statt diesem alle Administratorrechte, beispielsweise über das Setuid-Bit, zu gewähren.

Docker bietet die Möglichkeit, *Capabilities* für einzelne Container beim Starten festzulegen. *Capabilities* können über den Parameter `--cap-add` hinzugefügt und mit `--cap-drop` entfernt werden, z. B.:

```
docker run --cap-add=CAP_NET_ADMIN \
           --cap-drop=CAP_SYS_BOOT [...]
```

Mit der *Capability* (CAP_NET_ADMIN) kann der Container einen Port unter 1024 belegen. Container ohne diese *Capability* können dies nicht tun, auch wenn Administrator im Container sind.

Falls man nur wenige *Capabilities* gewähren möchte, kann man alle *Capabilities* entziehen und anschließend die gewünschten hinzufügen:

```
docker run --cap-drop=ALL \
           --cap-add=CAP_NET_ADMIN [...]
```

Selbst wenn es einem Nutzer gelingen sollte, im Container Administratorrechte zu erlangen, wird er keine Administratorrechte auf dem Hostsystem haben, weil das Administratorkonto im Container nun weniger privilegiert ist als auf dem Hostsystem. [23, 24]

Obwohl später jeder Container als normaler unprivilegierter Nutzer gestartet werden soll, wird ein Wrapper (s. Abschnitt 3.1) genutzt, um sicherheitshalber alle *Capabilities* zu entziehen und durch die Angabe des *docker*-Argument `--security-opt=no-new-privileges`[36] wird diesem Container die Möglichkeit genommen *Capabilties* wieder zu erlangen. In diesem Anwendungsfall werden ohnehin keine besonderen Privilegien benötigt.

[35] `http://man7.org/linux/man-pages/man7/capabilities.7.html`

[36] Dieser Funktion wird seit der Linux-Kernel-Version 3.5 unterstützt und verhindert zusätzlich, dass ein Prozess im Container durch ein Setuid-Bit höhere Rechte bekommt.

Ressourcenbeschränkung

Beim Start eines *Docker Containers* können Beschränkungen als Argumente[37] für `--ulimit` übergeben werden:

```
docker run --ulimit=[] -it IMAGENAME
```

Die Standardeinstellung beschränkt die maximale Anzahl der parallel laufenden Prozesse (nproc) und der geöffneten Dateien (nofile) auf 1048576 pro Nutzer. Diese werden jeweils auf 1024 limitiert, was bei normaler Nutzung genügen sollte. Damit soll verhindert werden, dass ein Nutzer zu viele Ressourcen verbraucht und somit auch andere Container beeinträchtigt. Außerdem werden *Core dumps* deaktiviert, indem die Core-Dump-Dateigröße (core) auf Null gesetzt wird.

Explizit lautet der Befehl, um die Hard und Soft Limits[38] zu setzen wie folgt:

```
docker run --ulimit nproc=1024:1024 \
           --ulimit nofile=1024:1024 \
           --ulimit core=0 -it IMAGENAME
```

[37] Eine komplette Liste der Argumente findet man unter `http://linux.die.net/man/5/limits.conf`.

[38] Soft Limits („weiche Begrenzungen“) können vom Prozess selbst auferlegt und bis zum Hard Limit („harte Begrenzung“) beliebig verändert werden. Hard Limits werden vom Administrator gesetzt und können nicht überschritten werden.

Kapitel 3

Testumgebung

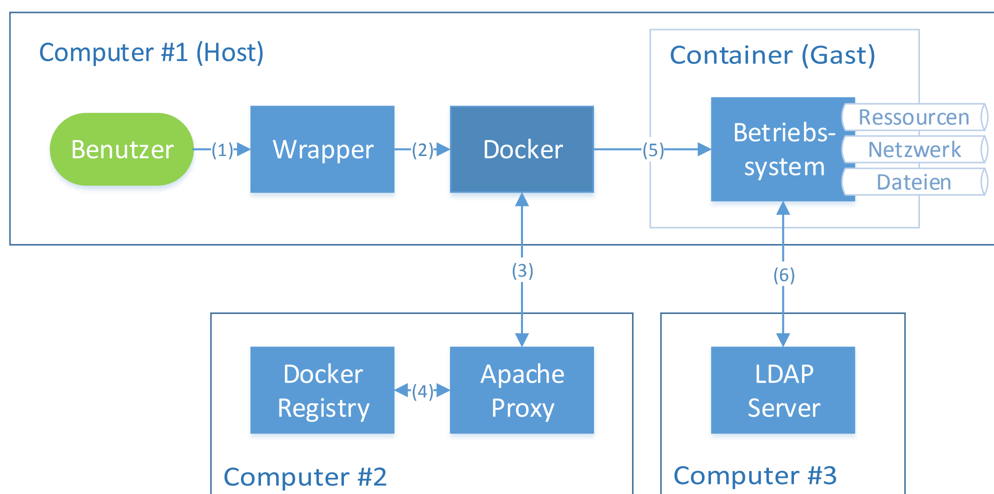


Abbildung 3.1: Testumgebung

In Abbildung 3.1 wird eine mögliche Konstellation in einem Rechnernetz skizziert, um *Docker* zu nutzen. Zu sehen ist eines der mehreren Hostsysteme im Netzwerk, ein zentrales Registry und ein zentraler LDAP-Server. Zuallererst ist der Nutzer mit einem der Hostsysteme verbunden und kann dort einen Wrapper ausführen (1), der automatisch *Docker* mit bestimmten Argumenten aufruft (2). Als Nächstes bezieht *Docker* das Image vom Registry, falls es nicht bereits lokal vorhanden ist. *Docker* authentifiziert sich beim *Apache*-Proxy (3) und lädt dann das Image herunter (4). Nun kann ein Container mithilfe des Images erstellt werden (5). Im Container werden die Benutzerinformationen von einem LDAP-Server erlangt (6). Schließlich kann

der Nutzer innerhalb des Containers auf das Netzwerk des Hostsystems und eingebundene Verzeichnisse, wie sein Homeverzeichnis, zugreifen.

3.1 Wrapper (Schnittstelle)

Ziel ist es eine möglichst benutzerfreundliche Oberfläche zu bieten, damit der Nutzer mit geringem Aufwand seine Arbeit wie gewohnt durchführen kann. Um die sicherheitsrelevanten Einstellungen durchzusetzen, wurde im Rahmen dieser Arbeit ein Wrapper für *docker* entwickelt. Dieser übernimmt das Übergeben der entsprechenden Argumente beim Aufruf von *Docker* und bindet die benötigten Daten und Verzeichnisse ein. Weil *Docker* Administratorrechte benötigt, aber der Benutzer diese nicht haben soll, wird das Setuid-Bit durch den Administrator (root) auf diesen Wrapper gesetzt. Dadurch wird der Wrapper zusätzlich zu den Benutzerrechten mit administrativen Rechten gestartet und damit *Docker* gestartet. So ist die Nutzung von *Docker* als normaler Benutzer möglich. Ein Setzen des Setuid-Bits direkt auf *Docker* würde zu viele Sicherheitsrisiken bergen, da beliebige Argumente an *Docker* übergeben werden könnten. Wenn der Nutzer direkte Kontrolle über den *Docker Client* hätte, könnte er einen Container mit Administratorrechten starten und Verzeichnisse einbinden, wodurch er auf dem Hostsystem beliebige Dateien verändern könnte.

Außerdem wurde kein Shellskript als Wrapper erstellt, sondern ein Programm in der Programmiersprache C++ geschrieben, weil ein auf ein Shellskript gesetztes Setuid-Bit aus Sicherheitsgründen vom Betriebssystem nicht berücksichtigt wird.

Im Verzeichnis, wo das *makefile* liegt, wird das Programm mit `make` kompiliert und anschließend mit `make install` konfiguriert. Das Programm `dock`[1] nimmt als erstes Argument den Imagenamen an. Dieses Argument ist auf alphanumerische Zeichen und auf Unterstrich (_), Bindestrich (-) sowie Doppelpunkt (:) beschränkt. Als zweites Argument kann optional ein Programmpfad angegeben werden, der im Container aufgerufen wird. Zusätzlich liest der Wrapper einige Umgebungsvariablen aus, die ebenfalls geprüft werden.

[1] `https://git.rwth-aachen.de/william.ma/docker/tree/master/dock`

Ein *Docker Container* kann dann über den Wrapper mit dem kurzen Befehl gestartet werden:

```
dock image:version
```

Im Hintergrund wird, beispielsweise für den Aufruf von `dock slc6`, ein entsprechender Befehl ausgeführt, der die in den vorangegangenen Abschnitten diskutierten Optionen nach den Anforderungen im hiesigen Rechnernetz setzt:

```
/bin/docker --config=/etc/dock/intern run \
  --name uid1001_ma_1473847244801 \
  --cap-drop=all \
  --security-opt=no-new-privileges \
  --net host \
  --ulimit nofile=1024:1024 \
  --ulimit nproc=1024:1024 \
  --ulimit core=0 \
  -v /etc/openldap/cacerts:/etc/openldap/cacerts:ro \
  -v /var/run/nslcd/socket:/var/run/nslcd/socket:ro \
  -v /.automount:/.automount:slave \
  -v /cvmfs:/cvmfs:slave \
  -v /home:/home \
  -v /net:/net \
  -v /tmp:/tmp \
  -e DISPLAY=localhost:17.0 \
  -e SHELL=/bin/zsh \
  -e HOME=/home/ma -e USER=ma \
  -e LANG=en_US.UTF-8 \
  -u 1001:1001 \
  -w=/home/ma \
  -it grid-tmp5:5043/slc6 /bin/zsh
```

Alle Konfigurationsdateien liegen in */etc/dock/*, worauf der normalen Benutzer nur Leserechte[2] hat.

Datei	Funktion
registry.conf	gibt das Registry an, wovon alle Images bezogen werden. Beispiel: `docker.io`
ulimit.conf	setzt Ressourcenbegrenzungen (Soft- und Hard-Limits) für alle Container. Beispiel: `nproc=1024:1024`
volumes.conf	bindet Verzeichnisse vom Hostsystem in den Container ein. Damit im Container nur lesend auf das Verzeichnis zugegriffen werden kann, muss man `:ro` anfügen. Bei Pfaden, die im Hostsystem per *automount* eingebunden werden, muss `:slave` angehängt werden. Beispiel: `/host/pfad:/pfad/im/container:ro`
volumes.d	imagespezifisches Einbinden von Verzeichnissen kann in diesem Ordner eingestellt werden. Die Konfigurationsdateien werden nach den Images benannt. Falls keine Konfigurationsdatei vorhanden ist, wird auf *volumes.conf* zurückgegriffen.

Tabelle 3.1: Konfiguration von `dock`

3.2 Installation und Konfiguration

In diesem Abschnitt wird ein Server (*grid-tmp5*) als privates Registry eingerichtet und ein Client konfiguriert. Der Client soll Images über den Wrapper (s. Abschnitt 3.1) vom privaten Registry beziehen können. Außerdem sollen nur autorisierte Benutzer Images erstellen und auf das Registry laden können, während alle anderen Nutzer Images ohne Authentifizierung herunterladen können.

Die Palette der Images wird auf einige wenige beschränkt[3], die möglichst die gleiche Laufzeitumgebung bieten sollen, die hier momentan im Rechnernetz vorhanden ist. Durch die geringere Zahl der Images kann sichergestellt werden, dass diese durch den Administrator wartbar bleiben.

[2]Die Konfigurationsdateien gehören dem Administrator und die Zugriffsbeschränkungen sind gesetzt auf: -rw-r--r-- (0o644).

[3]Im Anhang (s. Anhang B) sind *Dockerfiles* für die Distributionen SLC6, Fedora und Ubuntu zu finden.

Als Erstes wird der Server behandelt, wo zuallererst *Docker* installiert wird (s. Abschnitt 2.3). Die Authentifizierung soll über den Webserver *Apache*[4] als Proxy[5] erfolgen. Vor der Konfiguration ist ein Ordner *conf* erforderlich, welcher ebenfalls erstellt wird. Die Einstellungen des Webservers werden dann in der Datei *conf/httpd.conf* [6] angegeben. Zusätzlich wird *TLS*[7] genutzt, um den Datenverkehr zu verschlüsseln. Dafür werden die Schlüssel mithilfe des Paketes *openssl* erstellt:

```
openssl req -newkey rsa:4096 -nodes -sha256 \
  -keyout conf/domain.key -x509 -days 365 \
  -out conf/domain.crt
```

Als nächstes werden die Benutzerkonten und Gruppen in der Datei *conf/httpd.htpasswd* eingerichtet. Es wird eine Administratorkonto, welches der Gruppe *pusher* angehört, und ein Benutzer *readuser* erzeugt, der nur lesend auf das Registry zugreifen kann:

```
docker run --entrypoint htpasswd httpd:2.4 \
  -Bbn admin ADMIN_PASSWORT > conf/httpd.htpasswd
docker run --entrypoint htpasswd httpd:2.4 \
  -Bbn readuser NUTZER_PASSWORT >> conf/httpd.htpasswd
echo "pusher: admin" > conf/httpd.groups
```

Hier wird das *Docker Image* für *Apache* geladen, um `htpasswd` auszuführen, um die Passwortdatei *conf/httpd.htpasswd* zu erstellen. Der Administrator *admin* mit dem Passwort *ADMIN_PASSWORT* und der Nutzer *readuser* mit dem Passwort *NUTZER_PASSWORT* werden hinzugefügt. Anschließend wird der Administrator der Gruppe *pusher* hinzugefügt.

Der Webserver und das Registry werden in separaten Container laufen, wofür in einer Datei namens *docker-compose.yml* die Container entsprechend definiert (s. Anhang C) werden. Für den Webserver wird das benötigte Verzeichnis *conf* in den Container eingebunden, sodass die Konfigurationsdatei `httpd.conf` und darin definierte Dateien vorhanden sind. Der Hostname vom Registry wird noch entsprechend gesetzt (hier *grid-tmp5*) und der Port

[4] `https://httpd.apache.org/`

[5] Ein Proxy dient als Vermittler zwischen einem Client und einem Server, der Anfragen annimmt und sie dann mit der eigenen Adresse an den Server sendet.

[6] `https://git.rwth-aachen.de/william.ma/docker/tree/master/registry/conf`

[7] Transport Layer Security: Verschlüsselungsprotokoll zur Datenübertragung

auf 5043 gesetzt. Das Registry soll das offizielle Image *registry*[8] benutzen und später auf Port 5000 hören. Außerdem wird noch ein benutzerdefinierter Pfad in */var/lib/registry* im Container eingebunden, wo später die Images automatisch gespeichert werden. [25] Schließlich kann beides mit

```
docker-compose up -d
```

als Dienst gestart werden.

Auf der Seite des Clients wird *Docker* auch installiert. Der Wrapper wird installiert und entsprechend eingerichtet (s. Abschnitt 3.1). Der Inhalt der Datei *registry.conf* wird mit der Adresse des privaten Registries *grid-tmp5:5043* ersetzt. In der nur für den Administrator lesbaren Datei *config.json* werden die Anmeldeinformationen gespeichert und sollte wie folgt aussehen:

```
{
    "auths": {
        "grid-tmp5:5043": {
            "auth": "ANMELDUNG"
        }
    }
}
```

Hierbei ist die Zeichenkette `ANMELDUNG` mit dem Base64 kodierten[9] Anmeldedaten von *readuser* zu ersetzen. Zu beachten ist, dass hier nur der Benutzer mit Leserechten eingetragen ist, da Benutzername und Passwort nicht verschlüsselt sind, sondern lediglich über das Base64 Verfahren kodiert sind. Unter Linux kann man dies mit dem Programm `base64` und `printf` durchführen, welches die kodierte Zeichenkette ausgibt:

```
printf "BENUTZERNAME:PASSWORT"| base64
```

Es ist wichtig, dass `printf` statt `echo` aufgerufen wird, weil Letzteres einen Zeilenumbruch einfügt, was zu einer falsch kodierten Zeichenkette führt. Der Administrator verwendet statt des Wrappers direkt den *Docker Client*, um

[8] https://github.com/docker/distribution/tree/master/registry
[9] https://de.wikipedia.org/wiki/Base64

Images in das Registry zu laden. Um sich anzumelden, benutzt man

```
docker login
```

und gibt die vorher festgelegten Anmeldedaten des Administratorkontos an. Da das Registry eine TLS-Verbindung nutzt, muss das entsprechende Zertifikat *domain.crt* vom Server auf den Client unter dem Namen */etc/docker/grid-tmp5:5043/ca.crt* gespeichert und danach der *Docker* Dienst neu gestartet werden.

Zum Testen kann ein Image aus *Docker Hub*, z. B. *busybox*, bezogen werden:

```
docker pull busybox
```

Über das Argument `tag` kann das Image entsprechend benannt und das Registry angegeben werden:

```
docker tag busybox grid-tmp5:5043/busybox
```

Anschließend kann mit dem Argument `push` das Image in das private Registry mit der Adresse *grid-tmp5:5043* geladen werden:

```
docker push grid-tmp5:5043/busybox
```

Von nun an wird das Image automatisch vom Registry heruntergeladen, wenn der Nutzer einen Container mit

```
dock busybox
```

startet, falls es nicht lokal verfügbar ist. Im letzten Befehl muss das Registry nicht nochmal angegeben werden, weil der Wrapper dieses aus der Datei */etc/dock/registry.conf* entnimmt. Zum Herunterladen eines Images aus dem Registry ist keine manuelle Anmeldung erforderlich, weil die Anmeldeinformationen aus der Datei */etc/dock/intern/config.json* geladen werden.

3.3 Testlauf

Der erste Testlauf wurde von einem Wissenschaftler des III. Physikalischen Instituts, Lehrstuhl A durchgeführt. Dabei wurde auf SL7 verifiziert, dass die Ausführung von CMSSW[10] Programmen scheitert.

Nachdem ein Container mit dem Betriebssystem SLC6 mithilfe des Wrappers über `dock slc6` gestartet wurde, konnten die Programmen von CMSSW problemlos gestartet werden. Die ersten Ergebnisse decken sich mit vorherigen Resultaten, die auf Rechnern mit SLC6 ohne Virtualisierungstechniken erzielt wurden.

[10]CMSSW ist ein Framework (dt. Rahmenstruktur), welches Programme und Module enthält, die für das CMS-Experiment eingesetzt werden.

Kapitel 4

HTCondor

4.1 Einführung

HTCondor[1] wurde mit dem Ziel entwickelt, ungenutzte Rechenkapazitäten der vielen Arbeitsplatzrechner auszunutzen. Es verbindet die Rechner des Netzwerkes zu einem Cluster, und daraufhin kann jeder Benutzer Jobdateien[2] mit dem Befehl `condor_submit` an einen sogenannten *Scheduler* schicken, der automatisch die Jobs im Cluster verteilt. Alle Jobs werden auf dem jeweiligen Rechner sequentiell abgearbeitet, weshalb man hier auch von einer Stapelverarbeitung (*batch processing*) spricht. Der Fokus liegt, wie der Name andeutet, auf einen möglichst hohen Durchsatz (high throughput), also hier auf einer möglichst hohen Anzahl bearbeiteter Jobs in einem relativ langen Zeitraum, wie Wochen oder Monate. Im Gegensatz dazu steht das Hochleistungsrechnen, wo Aufgaben zeitnah und schnell unter Aufwendung hoher Rechenleistung bearbeitet werden.

Häufig wird HTCondor so eingestellt, dass nur Jobs bearbeitet werden, wenn niemand am jeweiligen Rechner aktiv arbeitet, also über eine gewisse Zeitspanne keine Benutzereingaben getätigt wurden oder die Prozessorauslastung gering ist. Falls während eines laufenden Jobs die Benutzeraktivität am Rechner aufgenommen wird, werden alle Jobs auf diesem Rechner angehalten. Ist die Laufzeitumgebung *Standard Universe*[3] gesetzt, kann HTCondor beim Anhalten eines Jobs einen Kontrollpunkt (checkpoint) vom Bearbeitungszustand erstellen und wartet bis der Rechner wieder untätig ist oder verschiebt den Job auf einen anderen verfügbaren Rechner im HTCondor-

[1] `https://research.cs.wisc.edu/htcondor/`

[2] `https://research.cs.wisc.edu/htcondor/manual/v7.8/2_5Submitting_Job.html`

[3] Ein *Universe* ist eine von HTCondor vordefinierte Laufzeitumgebung in der die Jobs ausgeführt werden.

Cluster. In einem anderen *Universe* kann der Jobs nur pausiert werden oder er muss komplett abgebrochen und auf einem anderen Rechner neu gestartet werden.

Jeder Rechner kann im HTCondor-Cluster mit einem ClassAd seine Anforderungen an die Jobs und seine angebotenen Ressourcen, wie z. B. CPU, Arbeitsspeicher oder Betriebssystem, verkünden. Mit ClassAds lassen sich anspruchsvolle Regelwerke realisieren, mit denen die Rechner angeben können, welche Jobs bearbeitet und welche abgewiesen werden sollen. Damit kann in einem heterogenen Rechnernetz gewährleistet werden, dass ein Job auf einem lauffähigen System bearbeitet wird. Außerdem können mithilfe von ClassAds die Ressourcen für jeden Job beschränkt werden. Jeder Job erzeugt ebenfalls ein ClassAd, der Anforderungen an den Rechner und Präferenzen definiert. Dieser gibt an, welche Ressourcen der Job benötigt und welche er wünscht. HTCondor vergleicht alle ClassAds miteinander und versucht, die Anforderungen eines Jobs bestmöglichst mit den vorhandenen Ressorcen abzustimmen. [26]

4.2 Docker

Ab der HTCondor Version 8.3.6 kann auch *Docker*, wenn es auf dem System installiert ist, genutzt werden, um Jobs zu bearbeiten. Dafür muss in der eingesendeten Jobdatei *Docker* als *Condor Universe* gesetzt werden und ein *Docker Image* angegeben werden, z. B.:

```
universe = docker
docker_image = ubuntu
executable = /PFAD/ZUM/PROGRAMM
queue
```

Wenn dies im Job angegeben ist, wird HTCondor automatisch einen *Docker Container* starten, in dem der Job ausgeführt wird. In der Konfigurationsdatei von HTCondor */etc/condor/condor_ config.local* kann global *Docker* als verwendetes *Universe* gesetzt werden:

```
DEFAULT_UNIVERSE = docker
```

Desweiteren lassen sich hier Verzeichnisse angeben, die von *Docker* eingebunden werden sollen, z. B.:

```
DOCKER_VOLUMES = myfolder, myreadfolder
DOCKER_VOLUME_myfolder = /path/to/myfolder:/to/folder
DOCKER_VOLUME_myreadfolder = /path/myfolder2:/folder2:ro
DOCKER_MOUNT_VOLUMES = myfolder, myreadfolder
```

4.3 Wrapper

Das *Docker Images* bzw. der Ausdruck *docker_image* kann nicht global voreingestellt werden, weshalb ein Wrapper für `condor_submit` eingesetzt wird. Er soll lediglich diesen Ausdruck in der Jobdatei setzen, falls er nicht vorhanden ist. Wenn der Ausdruck *docker_image* definiert wird, soll das angegebene *Docker Image* benutzt werden. Ansonsten soll sich der Wrapper wie `condor_submit` bedienen lassen. Dies lässt sich einfach mit einem Shellskript realisieren:

```
#!/bin/bash
/usr/bin/condor_submit \
  "docker_image=grid-tmp5:5043/slc6" "$@"
```

Den Wrapper kann man beispielsweise unter */usr/local/bin/condor_submit* abspeichern. Beim Ausführen eines Programms in der Konsole werden, wenn nicht der komplette Pfad zum Programm angegeben wurde, verschiedene Verzeichnisse in der Umgebungsvariable PATH abgesucht. Normalerweise ist in dieser Umgebungsvariable der Pfad */usr/local/bin* vor */usr/bin*, wodurch der Wrapper */usr/local/bin/condor_submit* und nicht */usr/bin/condor_submit* ausgeführt wird. Außerdem sollte sichergestellt werden, dass der Wrapper von allen Nutzern gestartet werden kann, aber nicht bearbeitet werden kann.

4.4 Testlauf

Im zweiten Testlauf wurde *HTCondor* und *Docker* auf zwei Rechnern mit SL7 eingerichtet. Diese bildeten ein *HTCondor*-Cluster. *HTCondor* wurde so konfiguriert, dass es *Docker* benutzt, und mit dem Wrapper das SL6-Image

ausgewählt. Es wurden verschiedene Jobs abgeschickt, die geprüft haben, dass *HTCondor* einen *Docker Container* mit SLC6 kreieren lässt und dieser funktionstüchtig ist.

Kapitel 5

Zusammenfassung

Durch besondere experimentspezifische Anforderungen an das Betriebssystem muss man nach Lösungsansätzen suchen, die diesen gerecht werden. Diese Arbeit hat allgemein den Ansatz über die container-basierte Virtualisierung mithilfe von *Docker* untersucht und ein Anwendung im Rechnernetz des III. Physikalischen Instituts der RWTH erprobt.

Im Vergleich zu anderen Virtualisierungslösungen, wie z. B. virtuelle Maschinen, bietet *Docker* eine höhere Effizienz, die vergleichbar mit dem Hostsystem ist und in diversen Leistungstests bestätigt werden konnte. Die Isolation kann durch verschiedene restriktive Maßnahmen, wie z. B. die Ausführung des Prozesses als normaler Benutzer, verbessert werden. Mit den vorgestellten Maßnahmen kann ein zufriedenstellendes Maß an Sicherheit gewährleistet werden, ohne den bisherigen Arbeitsablauf des Benutzers einzuschränken oder stark zu verändern. Über die zwei Wrapper wurden die Maßnahmen umgesetzt und die Bedienung stark vereinfacht. *Docker* wurde im Rahmen dieser Arbeit lediglich auf Testrechner eingerichtet. In näherer Zukunft könnte *Docker* auf einem größeren Teil des Rechnernetzes des Instituts eingesetzt werden.

Wenn man sich entschieden hat, eine Virtualisierungslösung für experimentspezifische Anforderungen zu nutzen, hat man diese bisher als eine Notwendigkeit angesehen. Welchen Mehrwert diese Technologie schaffen wird, wird sich in den nächsten Jahren herausstellen. Langfristig könnte eine Betriebssystemvirtualisierung, wie *Docker*, helfen eine homogene Umgebungen zwischen verschiedenen Rechenzentren aufzubauen. Dadurch können die Zusammenarbeit vereinfacht und opportunistische Rechenressourcen besser genutzt werden.

Danksagung

Ich möchte mich bei meinem Betreuer Dr. Andreas Nowack für das mehrmaliges Korrekturlesen meiner Arbeit, konstruktive Verbesserungsvorschläge, viele von ihm beantwortete Fragen und zielführenden Hilfestellungen bei aufgetretenen Problemen herzlich bedanken.
Außerdem danke ich Tobias Pook aus dem III. Physikalischen Institut, Lehrstuhl A für den Test des Wrappers und der CMS Analyseprogramme im *Docker Container*.
Zusätzlich möchte ich dem III. Physikalischen Institut B und insbesondere Prof. Dr. Stahl danken, dass mir diese Bachelorarbeit ermöglicht wurde.

Anhang A

Monte-Carlo-Simulationen

Monte-Carlo Simulationen sind ein nützliches Werkzeug, um komplexe Berechnungen durch simulierte Zufallsexperimente zu vereinfachen. Eine wichtige Komponente für die simulierten Zufallsexperimente ist der Zufallsgenerator. Dieser sollte eine hohe Güte aufweisen, also möglichst zufällige Zahlen erzeugen können. Beispielsweise können dafür physikalische Prozesse wie Spannungsschwankungen durch thermisches Rauschen oder radioaktive Zerfälle ausgenutzt werden. Doch diese Quellen haben den Nachteil, dass sie nicht beliebig schnell abgefragt werden können, ohne die Güte zu beeinträchtigen. Deshalb werden stattdessen sogenannte Pseudozufallszahlengeneratoren verwendet. Die damit generierten Zahlen wirken zwar zufällig, doch die generierte Zahlenfolge ist vollkommen deterministisch. Solange sich die Zahlenfolge nicht zu schnell wiederholt und die Güte hoch ist, muss dies kein Nachteil sein, denn ein deterministischer Zufallsgenerator erleichtert es, Ergebnisse verschiedener Programme zu prüfen. Außerdem können Pseudozufallszahlengeneratoren Zufallszahlen, im Rahmen der Rechengeschwindigkeit, beliebig schnell erzeugen.

In der Physik existieren viele Probleme, die nicht analytisch lösbar sind, weshalb numerische Lösungsmethoden, wie die Monte-Carlo-Simulation, angewendet werden müssen. Auch, wenn analytische Lösungen vorhanden sind, ist es manchmal sinnvoll das Problem numerisch zu lösen, wenn die analytische Lösung wesentlich aufwändiger ist. Beispielsweise werden in der Teilchenphysik numerische Berechnungen oft eingesetzt, um Erwartungen zu einem Modell zu ermitteln und diese mit dem realen Experiment zu prüfen. Man versucht mit entsprechender Rechenleistung in die Größenordnung der Anzahl der Ereignisse im Experiment oder sogar darüberhinaus zu gelangen, um genauere Werte aus der Simulation zu bestimmen. Die Komplexität des Problems verlagert sich auf die möglichst effiziente und hinreichend realitätsnahen Simulation des untersuchten Phänomens. [27]

Anhang B

Dockerfiles

Die folgenden *Dockerfiles* können mit `docker build` verwendet werden, um die erwähnten *Docker Images* zu erstellen (s. Abschnitt 2.2). Alle Konfigurationsdateien und *Dockerfiles* sind unter `https://git.rwth-aachen.de/william.ma/docker/tree/master/dockerfiles` zu finden.

SLC6:

```
FROM cern/slc6-base

RUN yum update -y && yum install -y HEP_OSlibs_SL6 wget \
 rpm-build which git screen e2fsprogs \
 tcsh sh tcsh csh zsh mksh ksh dash \
 tcl tk e2fsprogs perl-ExtUtils-Embed compat-libstdc++-33 \
 libXmu e2fsprogs-libs libXpm java-1.7.0-openjdk bc\
 zip perl-libwww-perl svn cvs readline-devel vim \
 openldap openldap-clients nss-pam-ldapd sssd-ldap \
 authconfig-gtk && \
 authconfig --enableldap --enableldaptls --enableldapauth \
   --ldapserver="ldaps://ldap.physik.rwth-aachen.de/" \
   --ldapbasedn="dc=physik,dc=rwth-aachen,dc=de" --update
```

Fedora 24:

```
FROM fedora:24

RUN dnf update -y && dnf install -y wget which git screen \
 zip svn cvs hostname \
 tcsh tcsh csh zsh mksh ksh dash \
 openldap openldap-clients nss-pam-ldapd authconfig-gtk \
 sssd-ldap && \
 authconfig --enableldap --enableldaptls --enableldapauth \
   --ldapserver="ldaps://ldap.physik.rwth-aachen.de/" \
   --ldapbasedn="dc=physik,dc=rwth-aachen,dc=de" --update
ADD conf/etc /etc
```

Ubuntu 16.04:

```
FROM ubuntu:16.04

ENV DEBIAN_FRONTEND="noninteractive"
RUN apt-get update -y && \
 apt-get install -y libnss-ldap libpam-ldap nscd portmap
ADD conf/etc /etc

RUN apt-get install -y libclhep-dev cvs git libapache2-svn \
 libwww-perl screen subversion wget zip \
 openjdk-9-jdk-headless \
 csh tcsh zsh mksh ksh dash \
 tcl tk bc e2fsprogs vim
```

Anhang C

Docker Compose

Mithilfe dieser Datei kann *Docker Compose* ein *Docker Registry* starten (s. Abschnitt 3.2). Konfigurationsdateien und *Dockerfiles* sind unter `https://git.rwth-aachen.de/william.ma/docker/tree/master/registry` zu finden.

docker-compose.yml:

```
apache:
  image: httpd:2.4
  hostname: grid-tmp5
  ports:
    - 5043:5043
  links:
    - registry:registry
  volumes:
    - ./conf:/usr/local/apache2/conf

registry:
  image: registry:2
  ports:
    - 127.0.0.1:5000:5000
  volumes:
    - /storage/registry:/var/lib/registry
```

Literaturverzeichnis

[1] *CERN Computing*, https://home.cern/about/computing (Stand: 14. September 2016).

[2] Helmut Satz et al., *Großforschung in neuen Dimensionen*, Springer-Verlag Berlin Heidelberg 2016, S. 99

[3] *WLCG REBUS: Topology*, https://gstat-wlcg.cern.ch/apps/topology/ (Stand: 29. Juni 2016).

[4] *The Grid: A system of tiers* https://home.cern/about/computing/grid-system-tiers (Stand: 19. Juli 2016)

[5] *About CERN*, https://home.cern/about, (Stand: 29. Juni 2016).

[6] Stephen Soltesz et al., *Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors*, https://docs.google.com/viewer?a=v&pid=sites&srcid=bGFyY2VzLnVlY2UuYnJ8bWFyY2lhbHxneDo0NTY5YmI0ZDk5MThkMGM4 (Stand: 26. August 2016).

[7] Wes Felter et al., *IBM Research Report-An Updated Performance Comparison of Virtual Machines and Linux Containers*, https://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf (Stand: 4. Juli 2016).

[8] Dario Berzano et al., *Lightweight scheduling of elastic analysis containers in a competitive cloud environment: a Docked Analysis Facility for ALICE*, http://iopscience.iop.org/article/10.1088/1742-6596/664/2/022005/pdf (Stand: 15. September 2016).

[9] Hamid Gadirov et al., *Container technology for the Upgrade of the ATLAS Trigger and Data Acquisition (TDAQ) system*, http://cds.cern.ch/record/2209132/files/SS_report_hgadirov.pdf (Stand: 15. September 2016).

[10] Giulio Eulisse et al., *Containerization of CMS Applications with Docker*, http://cds.cern.ch/record/2144862/files/PoS(ISGC2015)007.pdf (Stand: 15. September 2016).

[11] *Understanding Docker*, https://docs.docker.com/engine/understanding-docker/ (Stand: 14. September 2016).

[12] *Docker Linux interfaces*, https://commons.wikimedia.org/wiki/File:Docker-linux-interfaces.svg (Stand: 14. September 2016).

[13] *Docker—Sharing Layers*, https://docs.docker.com/engine/userguide/storagedriver/images/sharing-layers.jpg (Stand: 18. September 2016).

[14] *Authenticating proxy with apache*, https://docs.docker.com/registry/deploying/ (Stand: 26. Juli 2016).

[15] *Docker Installation*, https://docs.docker.com/v1.5/installation/rhel/ (Stand: 30. Juni 2016).

[16] *Installation on CentOS*, https://docs.docker.com/engine/installation/linux/centos/ (Stand: 19. Juli 2016).

[17] *Docker Documentation*, https://docs.docker.com/ (Stand: 17. September 2016).

[18] *Docker security*, https://docs.docker.com/engine/security/security/ (Stand: 14. September 2016).

[19] Michael Kerrisk, *Namespaces—overview of Linux namespaces*, http://man7.org/linux/man-pages/man7/namespaces.7.html (Stand: 5. September 2016).

[20] Eric W. Biederman, *Multiple Instances of the Global Linux Namespaces*, https://www.landley.net/kdocs/ols/2006/ols2006v1-pages-101-112.pdf (Stand: 14. September 2016).

[21] Serge Hallyn, *cgroups—Linux control groups*, http://man7.org/linux/man-pages/man7/cgroups.7.html (Stand: 14. September 2016).

[22] Jérôme Petazzon, *Gathering LXC and Docker containers metrics*, https://blog.docker.com/2013/10/gathering-lxc-docker-containers-metrics/ (Stand: 14. September 2016).

[23] Michael Kerrisk, *Linux capabilities*, http://man7.org/linux/man-pages/man7/capabilities.7.html (Stand: 1. Juli 2016).

[24] *Runtime privilege and Linux capabilities*, https://docs.docker.com/engine/reference/run/#/runtime-privilege-and-linux-capabilities (Stand: 29. Juli 2016).

[25] *Authenticating proxy with apache*, https://docs.docker.com/registry/recipes/apache/ (Stand: 26. Juli 2016).

[26] Todd Tannenbaum et al., *Condor—A Distributed Job Scheduler*, https://research.cs.wisc.edu/htcondor/doc/beowulf-chapter-rev1.pdf (Stand: 14. September 2016).

[27] Gerhard Bohm und Günter Zech, *Introduction to Statistics and Data Analysis for Physicists*, http://www-library.desy.de/preparch/books/vstatmp_engl.pdf (Stand: 8. August 2016).

Eidesstattliche Versicherung

Ma, William — Name, Vorname

331127 — Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

Anpassung eines Rechnerverbundes an experimentspezifische Anforderungen mithilfe container-basierter Virtualisierung

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 20.09.2016

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen, 20.09.2016

Ort, Datum

Unterschrift