Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Timo Feddersen

Evaluierung unterschiedlicher Persistenzlösungen für Docker

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Timo Feddersen

Evaluierung unterschiedlicher Persistenzlösungen für Docker

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 29. September 2016

Timo Feddersen

Thema der Arbeit
Evaluierung unterschiedlicher Persistenzlösungen für Docker

Stichworte
Docker, Volume, Volume-Plugin, Persistenz, VM, virtuelle Maschine

Kurzzusammenfassung
Das Ziel dieser Bachelorarbeit ist es, einen Vergleich der gebräuchlichsten Docker-Volume-Plugins zu erstellen. Dazu werden zunächst die am weitest verbreiteten Volume-Plugins ausgewählt und deren angebotenen Features sowie deren grundlegende Verwendung an einem übergreifenden Beispiel demonstriert. Anhand eines Uni-Projektes wird aufgezeigt wie man für gegebene Anforderungen zu einer geeigneten Volume-Plugin-Auswahl gelangt. Nach der Pluginauswahl wird für dieses Projekt ein Persistenzkonzept erarbeitet und dieses anschließend praktisch umgesetzt.

Timo Feddersen

Title of the paper
Evaluation of different persistence solutions for Docker

Keywords
Docker, Volume, Volume-Plugin, persistence, VM, Virtual machine

Abstract
The purpose of this thesis is to examine and compare the most widely used volume plugins for Docker. To this end certain prevalent plugins are chosen and analysed for their features. Their basic usage is then demonstrated by an example. Based on a college research project it is then shown how, given specific needs and criteria, one can get to choose an appropriate volume plugin. Finally a persistence concept for this project is developed and implemented.

Inhalt

1 Einleitung

> *„Docker is an open platform to build, ship and run distributed applications anywhere“*
> - Docker, Inc.[1]

Docker erblickte 2013 das Licht der Welt und ist mittlerweile innerhalb der IT-Branche in aller Munde. Docker verspricht dabei wie klassische virtuelle Maschinen Plattformunabhängigkeit zu schaffen, allerdings ohne dabei Kompromisse im Bezug auf Performanceverlust und Ressourcenverbrauch zu machen. Docker ermöglicht es Entwicklern ihre Software inklusive aller Abhängigkeiten in einen Docker-Container-Image zu verpacken und so auf jeder Plattform, welche Docker unterstützt, laufen zu lassen.

Klassische virtuelle Maschinen haben keine Schwierigkeit im Umgang mit Programmen die Daten persistent vorhalten müssen. Sie sind darauf ausgelegt dauerhaft zu existieren und haben einen gekapselten Zugriff auf den lokalen Speicher. Docker-Container hingegen sind für eine kurze Lebensdauer entworfen und können nicht ohne weiteres Daten persistent halten. Mit den Docker-Volumes existiert ein grundlegender Mechanismus um Containerdaten auch über die Lebenszeit des Containers hinaus zu erhalten.

Diese Volumes bieten allerdings nur lokale beschränkte Dateioperationen. Durch die Einführung von Docker-Volume-Plugins, wurde es Drittanbietern ermöglicht, die grundlegenden Volume-Möglichkeiten von Docker zu erweitern. Mittlerweile existieren eine ganze Reihe dieser Plugins. Sie ermöglichen eine Reihe von Zusatzfunktionen wie beispielsweise Snapshots von Volumes oder Cloud-Speicherdienste als Backend.

[1] https://www.docker.com/what-docker

1.1 Zielsetzung

Im Rahmen dieser Bachelorarbeit wird untersucht, welche Persistenz-Möglichkeiten heutzutage durch Docker-Volumes-Plugins existieren. Auch der grundlegende Gebrauch der einzelnen Plugins soll erläutert werden. Zusätzlich soll anhand eines praktischen Fallbeispiels aufgezeigt werden, wie eine geeignete Pluginauswahl erfolgt und deren praktische Anwendung aussieht.

1.2 Aufbau der Arbeit

In Kapitel 2 werden die benötigten Grundlagen für diese Arbeit behandelt. Zu Beginn wird die Technik der virtuellen Maschinen erläutert. Aufbauend darauf wird anschließend auf die allgemeine Funktionsweise von Docker eingegangen.

Im Kapitel 3 werden die einzelnen Lösungsmöglichkeiten vorgestellt und deren Verwendung anhand eines übergreifenden Beispiels demonstriert. Am Ende dieses Kapitels werden die einzelnen Lösungsansätze gegenübergestellt und deren Möglichkeiten verglichen.

Im Kapitel 4 wird für ein Forschungsprojekt anhand der zuvor gewonnen Erkenntnisse und gegebenen Anforderungen eine geeignete Lösung umgesetzt und anschließend bewertet.

Das abschließende Kapitel 5 fasst die Arbeit zusammen und gibt einen Ausblick auf die weitere Entwicklung der Techniken sowie Möglichkeiten weiterführender Arbeiten.

2 Grundlagen

Dieses Kapitel bildet für diese Arbeit die essentiellen fachlichen Grundlagen. Dazu gehören, neben der allgemeinen Funktion und Nutzen von virtuellen Maschinen, vor allem auch die Grundlagen von Docker und inwiefern es sich von der klassischen Virtualisierung unterscheidet.

2.1 Virtuelle Maschinen

2.1.1 Was sind virtuelle Maschinen?

Eine virtuelle Maschine (kurz VM) ist ein simulierter PC der in einer abgeschotteten Umgebung auf einem realen PC läuft. Die virtuelle Maschine besteht in erster Linie aus der virtuellen Hardware nämlich Grafikkarte, CPUs, Speicher, Netzwerkkarten und Festplatten. Zusätzlich kann die virtuelle Hardware um optische Laufwerke oder USB-Sticks erweitert werden. Auch ein virtuelles BIOS wird simuliert. In einer virtuellen Maschine läuft ein vollständiges Betriebssystem welches Gastbetriebssystem genannt wird. Das Gastbetriebssystem bedarf keinerlei Anpassungen, da es durch die Hardware-Abstraktion aus seiner Sicht auf einer vollständig dedizierten Hardware läuft. Das System auf welcher die virtuelle Maschine läuft wird Hostsystem genannt.(vgl. Zimmer, 2012)

2.1.2 Welche Vorteile haben virtuelle Maschinen?

Ressourcenoptimierung - Durch den Hypervisor ist es nun möglich, dass mehrere Betriebssysteme zeitgleich auf einer Hardware parallel laufen. Durch diese Flexibilität lassen sich komplexe Testumgebungen auf einem einzigen Rechner aufbauen. Die Anzahl der physischen Server lässt sich dadurch verringern, was zu Ersparnissen in Strom, Platz und Anschaffungen führt. Die vorhandenen physikalischen Ressourcen führen durch die Verwendung von mehreren VMs zu einer besseren Auslastung. (vgl. Ahnert, 2006)

Hardwareunabhängigkeit - Ein weiterer Vorteil, ist das einfache kopieren von VMs. Sie lassen sich beispielsweise einfach vorkonfigurieren um sie dann schließlich an Kunden zu verteilen. Durch den Hypervisor ist die VM auch komplett Hardware unabhängig, da für sie immer die gleiche virtuelle Hardware vorhanden ist. (vgl. Ahnert, 2006)

Isolation von Anwendungen - Anwendungen können isoliert von dem Hostsystem in einer eigenen virtuellen Maschine laufen.(vgl. Ahnert, 2006)

Backup Management - Durch sogenannte Snapshots lassen sich virtuelle Systemplatten sichern und Wiederherstellungspunkte setzten. Damit lässt sich jederzeit zu den zuvor gesicherten Systemzuständen zurückspringen. Dies ist hilfreich bei eventuellen Störungen des Systems oder auch um gefahrlos testweise Veränderungen am System vorzunehmen. (vgl. Ahnert, 2006)

Legacy-Support - Virtualisierung bietet einen Ausweg aus dem Dilemma der Pflege althergebrachter Software. Nicht selten ist Software seit vielen Jahren unverändert im Einsatz, die nur auf einer bestimmten ebenfalls veralteten Hardware oder nur unter einem bestimmten Betriebssystem läuft. Vielfach kommt eine Portierung dieser Software nicht in Frage, da der Hersteller das Produkt nicht mehr pflegt und/oder bestimmte Hardware-Komponenten nicht mehr existieren. Mit der Hilfe von Virtualisierung lässt sich eine Umgebung simulieren, so dass auch ältere Software lauffähig bleibt.

Auslastung der Server - Häufig sind physikalische Server mit einer einzelnen Aufgabe nicht voll ausgelastet. Durch die Verwendung von mehreren VMs auf einem Server lässt sich die Auslastung des Servers erhöhen.

2.1.3 Gibt es auch Nachteile?

Neben all den Vorteilen von VMs existieren auch einige Nachteile.

Know-how - Neben dem Wissen über die einzelnen Betriebssysteme kommt mit der Virtualisierung eine weitere Komplexitität hinzu. Die virtuelle Infrastruktur will geplant, aufgebaut, betrieben und aktualisiert werden.(vgl. Zimmer, 2012)

Single Point of Failure - Wird eine komplexe IT-Infrastruktur innerhalb einer VM aufgebaut und stürzt diese ab, ist damit die gesamte IT-Infrastruktur betroffen.

Perfomance - Auch wenn die Hersteller von VM Lösungen dies immer weiter optimieren, existieren durch die Virtualisierung immer noch einen Performance-Nachteil gegenüber einem nativ laufenden Betriebssystem. Dies muss bei der Planung komplexer Infrastruktur beachtet werden.

2.1.4 Hypervisor

Die Maschinenbefehle des Gastsystems müssen an das Hostsystem weitergeleitet und gegebenenfalls übersetzt werden. Diese abstrahierende Schicht stellt der Hypervisor auch *Virtual Maschine Monitor* (kurz VMM) genannt bereit. Er kümmert sich um die Zuordnung der Hardwareressourcen zu den virtuellen Systemen. Durch den Hypervisor ist es auch möglich, dass völlig unterschiedliche Betriebsysteme auf einem Hostsystem parallel laufen. Das Hostsystem kann dabei beispielsweise ein Microsoft Windows System sein und die Gastsysteme sind Linux Distributionen. Es existieren zwei grundlegende Hypervisor Arten, welche sich durch ihre Architektur unterscheiden.

Typ-1-Hypervisor

Der Hypervisor vom Typ 1 läuft direkt auf der Hardware des Trägersystems und hat als einziger die Kontrolle über die Hardware, um den darauf laufenden virtuellen Systemen die notwendigen Ressourcen zuzuteilen und diese zu kontrollieren. Da hier der Hypervisor direkt auf der Hardware läuft, ist dieser performanter als ein Typ-2-Hypervisor. (vgl. Zimmer, 2012)

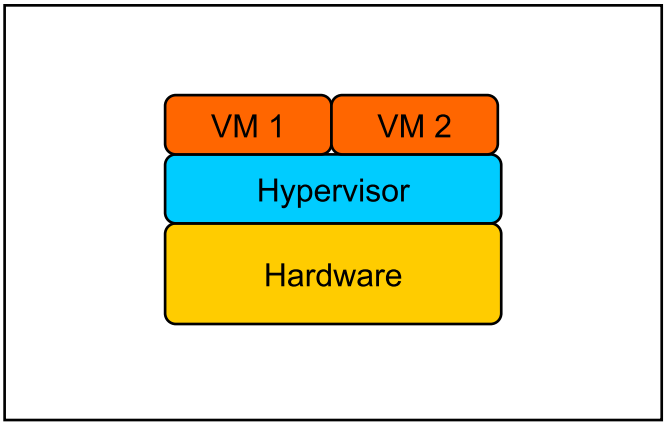


Abbildung 2.1: Typ-1-Hypervisor

Typ-2-Hypervisor

Beim Typ-2-Hypervisor läuft der Hypervisor als ein normales Benutzerprogramm auf einem bereits installierten Betriebssystem. Auch dieser stellt dem Gastsystem Ressorcen des Hostsystemes zur Verfügung. Aus der Sicht eines Gastsystems macht es dabei keinen Unterschied, ob es auf einem Typ-1 oder Typ-2 Hypervisor läuft. Dabei ist allerdings zu beachten, dass ein Typ-2-Hypervisor weniger Ressourcen den Gastsystemen zur Verfügung stellen kann als ein Typ-1 Hypervisor auf identischer Hardware, da ja bereits das Hostbetriebssystem Ressourcen für sich beansprucht.(vgl. Zimmer, 2012)

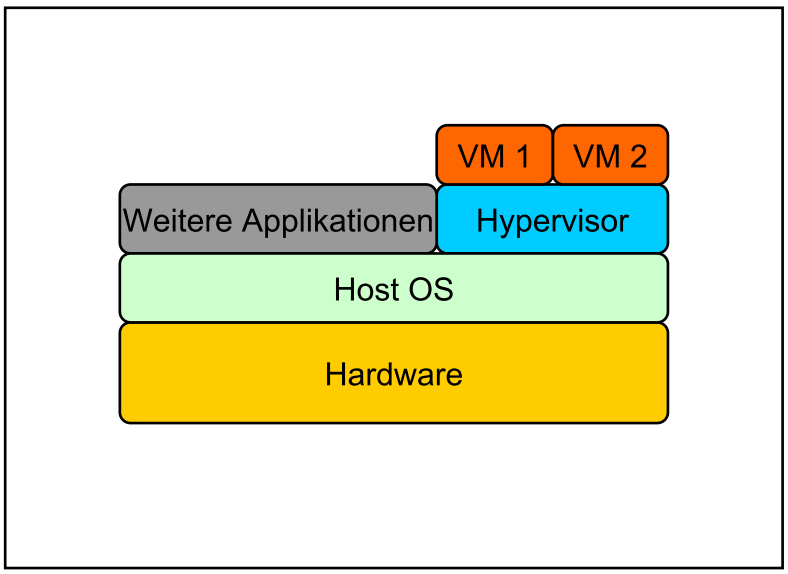


Abbildung 2.2: Typ-2-Hypervisor

2.2 Dateisysteme

Im Zuge dieser Arbeit werden verschiedenartige Dateisysteme verwendet, deren grundlegende Funktionsweise in diesem Abschnitt erläutert wird. Grundsätzlich sei gesagt, dass Dateisysteme eine Schnittstelle zwischen dem Betriebssystem und den Partitionen auf Datenträgern bilden. Sie sind dafür zuständig wann, wo und wie die Daten gespeichert werden.

2.2.1 NFS

Das Network File System (NFS) ist ein UNIX-Netzwerkprotokoll welches den Zugriff von Dateien über ein Netzwerk ermöglicht. NFS hat eine Server Client Architektur. Der NFS Server stellt Teile seines eigenen Dateisystem als NFS-Freigabe zur Verfügung. Clients haben nun die Möglichkeit auf Dateien/Ordner innerhalb dieser Freigabe zuzugreifen. Daten können damit an einem zentralen Ort gespeichert und von den Clients über das Netzwerk aufgerufen werden.(vgl. Mandl, 2014)

2.2.2 GlusterFS

GlusterFS ist ein verteiltes Dateisystem, welches Speicherkapazitäten von mehreren Servern zu einem einzigen virtuellen Laufwerk zusammenfasst. Dieses kann anschließend wie ein herkömmliches Dateisystem über NFS oder Fuse auf einem Client eingebunden werden. Während des Betriebes lassen sich jederzeit weitere Server hinzufügen oder vorhandene aus dem Speicherpool abziehen. Die Arbeitsweise von GlusterFS erinnert an eine Art "Netzwerk-RAID", und in der Tat erkennt man bei der Einrichtung viele RAID-Konzepte wieder. Es verteilt auf Wunsch die Nutz- und Metadaten geschickt auf alle Server, es ist somit fehlertolerant und hochverfügbar.(vgl. Schuermann, 2016)

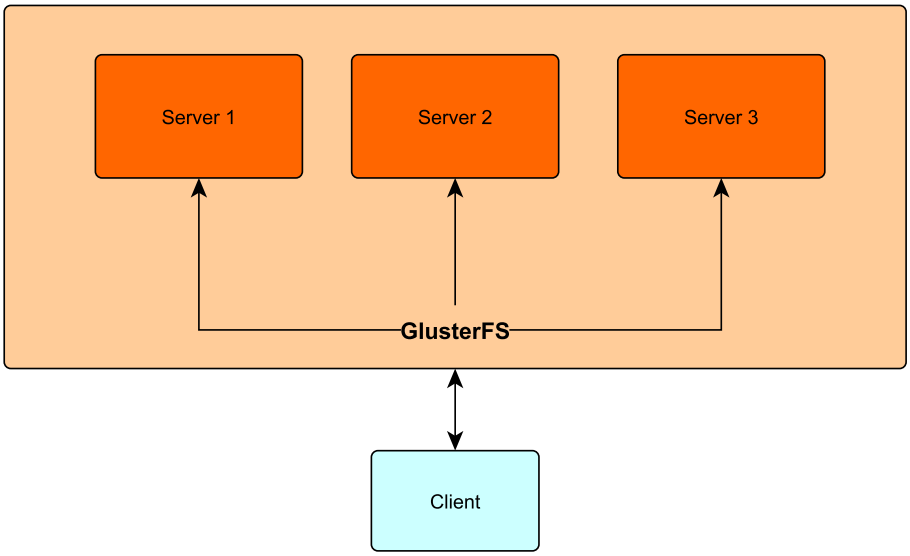


Abbildung 2.3: Ein GlusterFS Aufbau mit drei Servern und einem Client

2.2.3 Stapelbare Dateisysteme

Bei stapelbaren Dateisystemen werden einzelne Schichten aufeinander gelegt. Jede dieser Schichten wird dabei getrennt erstellt. Bekannt ist dieses Verfahren von so genannten Live-CDs wie beispielsweise Knoppix, bei denen ein Betriebssystem von einer CD gestartet wird. Um während der Benutzung von Knoppix Daten temporär im Dateisystem zu speichern, wird eine zweite beschreibbare Schicht über das System gelegt. Auch Docker verwendet für die Image-Speicherung stapelbare Dateisysteme (u.a AUFS, btrfs). Für jede Änderung und Festschreibung

eines Images, wird eine neue *readonly* Schicht angelegt. Bei Lesevorgängen hangelt sich Docker durch die Kette von Images, bis es die gewünschte Datei gefunden hat. Bei dem Start eines Containers wird auf dem vorhandenen Image-Stack eine neue beschreibbare Schicht für den Betrieb des Containers hinzugefügt. Da der Container keine Schreibrechte auf das Image hat aus welchem er erstellt wurde, gehen alle Containerdaten nach dem beenden des Containers mit ihm verloren.

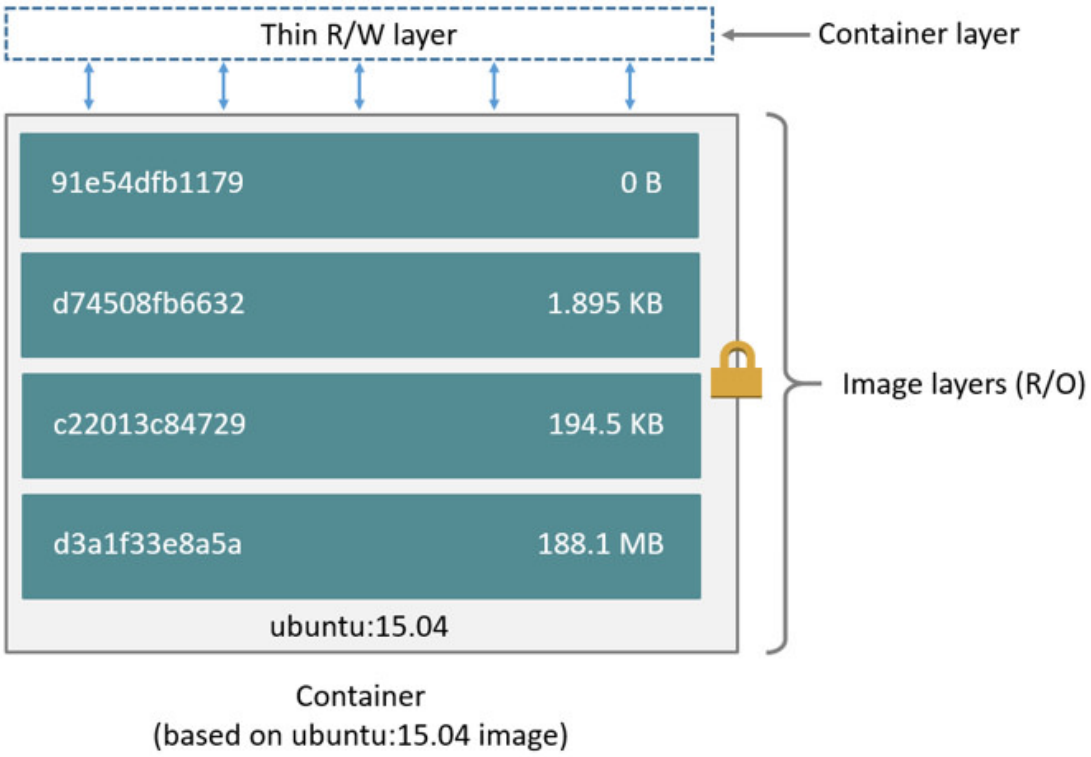


Abbildung 2.4: Stapelbares Dateisystem
Quelle: https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/

2.3 Docker

Ein großer Nachteil der System-Virtualisierung mittels Hypervisor ist, dass jede VM als komplettes System gebootet wird und damit auch viele Systemressourcen beansprucht. Docker hingegen verfolgt einen anderen Ansatz. Dabei werden Linux-Techniken wie Cgroups und Namespaces eingesetzt die es erlauben, sogenannte Container isoliert von anderen Prozessen zu verwenden.

Hierzu wurde anfangs noch, als Interface zu diesen Diensten, das sogenannte Linux-Container (LXC) verwendet. Ab Docker-Version 0.9[1] wird statt LXC allerdings eine eigene Container-Bibliothek namens libcontainer verwendet. LXC wurde mit der Linux-Kernel Version 2.6.29 eingeführt. Veröffentlicht wurde diese Kernel Version am 23.03.2009. Die Technik ist also nicht neu, allerdings bietet Docker ein komplettes Framework um diese Linux-Container benutzerfreundlich zu verwenden.

Im Unterschied zu einer Virtualisierung mittels Hypervisor, bei welchem eine Abstraktion zur Hardware-Ebene stattfindet und jedes Gastsystem ein eigenen Kernel hat, teilen sich hierbei die Linux-Container gemeinsam den Kernel des Gastsystems. Es findet dabei zwar eine Isolierung gegenüber anderen Prozessen statt, aber keine Virtualisierung eines gesamten Systems. Neben dem Vorteil der Schonung von Ressourcen, starten Linux Container auch noch wesentlich schneller als VMs. Die Hardware und Betriebssystemunabhängigkeit bleibt dabei wie bei klassischen VMs erhalten.

Somit sind Programme, die innerhalb eines Docker-Container-Images erstellt werden, auf jeder Plattform welche Docker unterstützt lauffähig. Für Entwickler fallen somit Softwareanpassungen innerhalb von heterogenen Umgebungen weg.

2.3.1 Docker-Image und Docker-Container

Genauso wie virtuelle Maschinen auf Images basieren, basieren auch Docker-Container auf Images. Ein Docker-Container stellt dabei eine Instanz eines Docker-Images da und es können beliebig viele Container-Instanzen eines Images gestartet werden. Ein Docker-Image verwendet dabei ein stapelbares Dateisystem 2.2.3. Ein Docker-Image wird somit, ausgehend von einem Basis-Image, schichtweise aufgebaut.

[1] https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/

Abbildung 2.5 zeigt ein Beispiel dazu. Ausgehend von einem Ubuntu-Basis-Image, werden jeweils noch Emacs und MongoDB hinzugefügt. Jede dieser Änderungen fügt eine weitere Schicht im Image hinzu, welches anschließend als Container instanziiert werden kann.

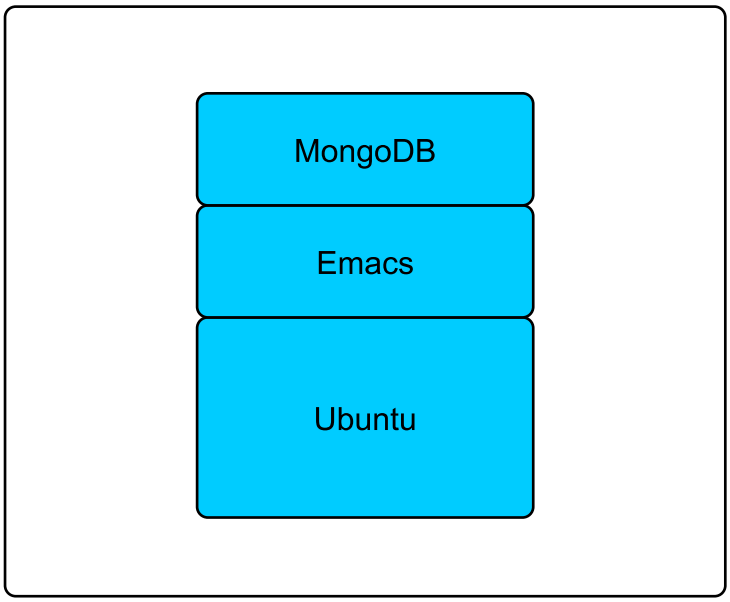


Abbildung 2.5: Image Schichten

2.3.2 Docker-Engine

Das Herzstück von Docker bildet die Docker-Engine. Diese besteht aus dem Docker-Daemon und dem Docker-Client. Der Docker-Daemon ist für das Verwalten der Container zuständig. Er startet diese, kann sie stoppen und auch wieder löschen. Außerdem streamt der Docker-Daemon den Output der Container zurück an den aufrufenden Docker-Client. Mit dem Docker-Client wird die Steuerung des Docker-Daemon über ein Command-line interface (CLI) angeboten. Die Kommunikation findet dabei über eine REST Schnittstelle, welche der Docker-Daemon anbietet, statt. (Siehe Abbildung 2.6)

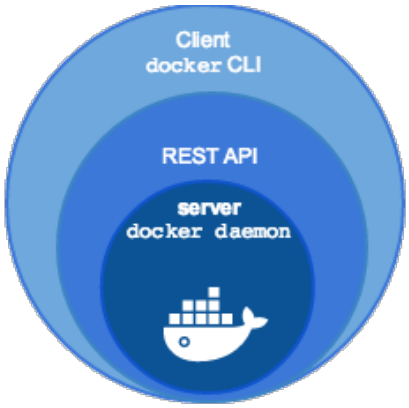


Abbildung 2.6: Client-Daemon REST API
Quelle: http://www.ctan.org/tex-archive/macros/latex/contrib/mwe

2.3.3 Docker-Machine

Docker-Machine ist ein Tool welches es ermöglicht die Docker-Engine bereitzustellen. Es hat dabei zwei Haupteinsatzzwecke. Auf Windows- und Mac-Systemen gibt es, neben der mittlerweile erhältlichen nativen Docker Unterstützung, auch noch den Betrieb innerhalb der Oracle VirtualBox. Mit der Hilfe der Docker-Machine, lassen sich so beliebig viele lokale virtuelle Docker-Hosts erstellen.

Ein weiterer Einsatzzweck ist das Installieren der Docker-Engine auf Remote-Machinen in Cloud-Umgebungen wie beispielsweise *Amazon Web Services*. Somit kann das Herunterladen und Installieren der Docker-Engine auf den Hostsystemen automatisiert werden.

2.3.4 Dockerfile

Docker-Images lassen sich erweitern indem ein Container davon gestartet wird, alle gewünschten Änderungen vorgenommen und anschließend diese Änderungen festgeschrieben werden. Beispielsweise möchte man ein Ubuntu-Image noch um die Software *Nodejs* erweitern. Dieser manuelle Prozess lässt sich durch ein Dockerfile automatisieren. Das Dockerfile ist lediglich eine Textdatei. Docker bietet eine DSL (Domain Specific Language) an, mit welcher sich die Image-Aktionen steuern lassen. Das Image wird dann mit dem Dockerfile und dem Docker Befehl *docker build* erstellt.

```
FROM ubuntu:trusty
RUN sudo apt-get install -y nodejs
RUN sudo apt-get install -y npm
```

Listing 2.1: Dockerfile-Beispiel für die Installation von Nodejs und NPM auf einem Ubuntu Grundimage

2.3.5 Docker-Compose

Verteilte Anwendungen können aus einer großen Anzahl von Docker-Containern bestehen. Anstatt diese alle einzeln manuell zu starten, bietet Docker-Compose die Möglichkeit, dieses Vorgehen scriptgesteuert zu verwalten. Nach dem definieren der Services in einer YAML-Datei, lassen sich die Container beispielsweise mit dem Befehl *docker-compose up* starten.

```
version: '2'
services:
  web:
    build: .
    ports:
    - "5000:5000"
    links:
    - redis
  redis:
    image: redis
```

Listing 2.2: Docker-Compose YAML-Datei Beispiel

2.3.6 Docker-Registry

Docker-Images können zentral in einer sogenannten Registry abgelegt werden. Docker selbst bietet ein eigene Cloud-Registry namens *Docker Hub* an. Es existieren aber auch eine Reihe von Drittanbietern, die eine Registry für Docker-Images anbieten. Die Registry bietet, neben dem Zugriff auf das Image, auch noch eine Versionsverwaltung für das Image an.

2.3.7 Docker-Swarm

Mit Docker-Swarm ist es möglich, mehrere Docker-Host zu einem einzigen logischem Docker Cluster zusammenzufassen. Da Docker-Swarm die Standard-Docker API verwendet, kann

auf diesen Cluster weiterhin mit den gebräuchlichen Docker Tools zugegriffen werden. Zur späteren Identifizierung der einzelnen Docker-Host, werden diese bei einem Discovery-Service registriert. Durch eine Reihe von Filter und Verteilungsstrategien wird bestimmt, welche Container, unter welchen Umständen, auf welchen Hosts instanziiert werden.

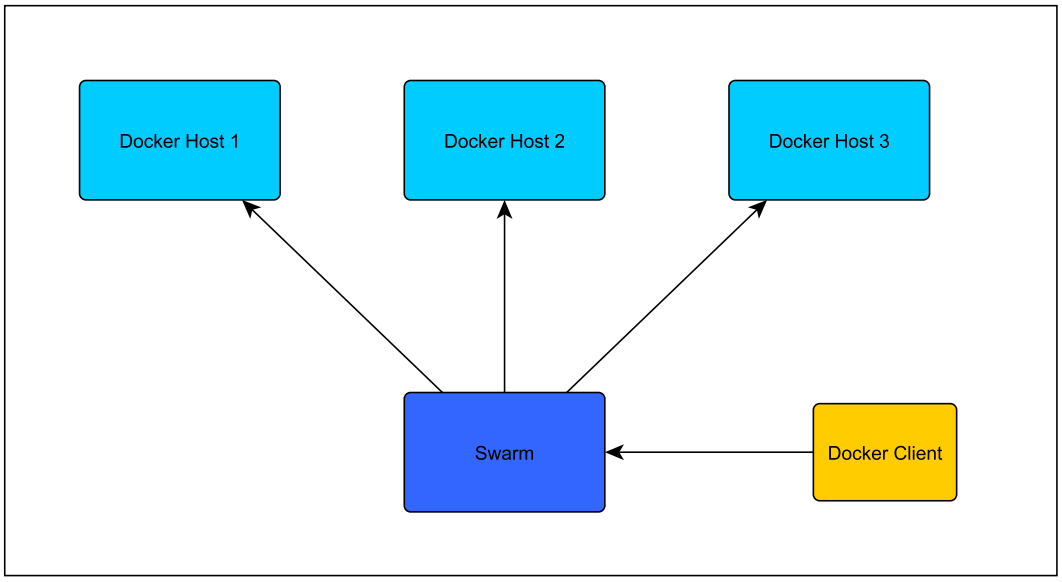


Abbildung 2.7: Aufbau eines Docker-Swarm

2.3.8 Docker Volume

Wenn ein Docker-Container gelöscht wird, sind alle Daten die während der Laufzeit verändert wurden verloren. Gegen diese Problematik wurden Docker-Volumes eingeführt. Mit den Volumes lassen sich einzelne Pfade des Container-Dateisystems als Mountpunkt von dem Host-Dateisystem mounten.

Beispielsweise läuft in dem Container eine Datenbank, welche ihre Logdaten in dem Pfad /var/log speichert. Nun kann ein Pfad auf dem Hostsystem, auf diesen Container-Pfad gemountet werden, so dass alle Daten die nach /var/log gespeichert werden, außerhalb des Container Dateisystems landen. Die Daten bleiben so auch nach dem löschen eines Containers erhalten. Es wird also sinnbildlich ein Loch in den Container gebohrt, um teilweise Daten hinaus leiten zu können.

Wird ein neuer Container instanziiert, kann ein bereits vorhandenes Volume angehängt werden. Auch die gleichzeitige Verwendung von mehreren Containern auf ein Volume ist möglich. In dem Listing 2.3 wird ein Container mit einer Redis-Datenbank gestartet und als Volume-Mountpunkt das Verzeichnis */data* angegeben. Hierbei wurde allerdings kein Pfad des Hostsystems angegeben, wovon dieses Verzeichnis gemountet wird. In diesem Fall wird automatisch ein Pfad innerhalb des Hostsystems gemountet. In der Regel befinden sich diese Verzeichnisse unter dem Pfad */var/lib/docker/volumes/...*. Die Verwendung eines bestimmtes Hostpfades findet sich im Listing 2.4

```
1 docker run -d -v /data redis
```

Listing 2.3: Volume Beispiel 1

```
1 docker run -d -v /home/redis/data:/data redis
```

Listing 2.4: Volume Beispiel 2

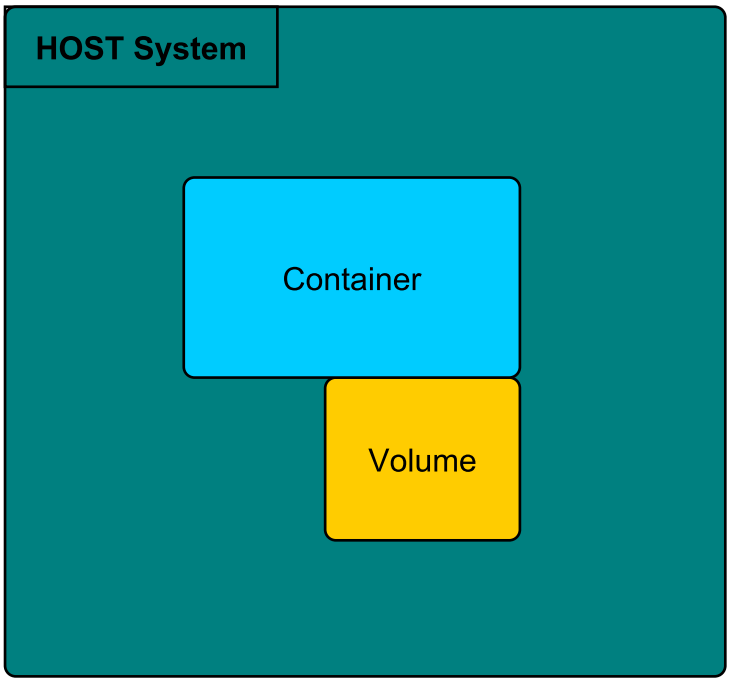


Abbildung 2.8: Ein Container mit Volume

Volume-Plugin

Um die Funktionalitäten von Docker erweitern zu können, wurden mit der Version 1.8.0 die Engine-Plugins eingeführt. Über eine angebotene API können somit die Netzwerk- und Volumemöglichkeiten erweitert werden. Dies macht den Anwender unabhängiger von der Docker-Entwicklung und ermöglicht es ihm, eigene Volume-Plugins zu verwenden bzw. zu entwickeln. In dieser Arbeit werden eine Reihe dieser Erweiterungen auf deren Möglichkeiten hin untersucht, verglichen und ausgewertet.

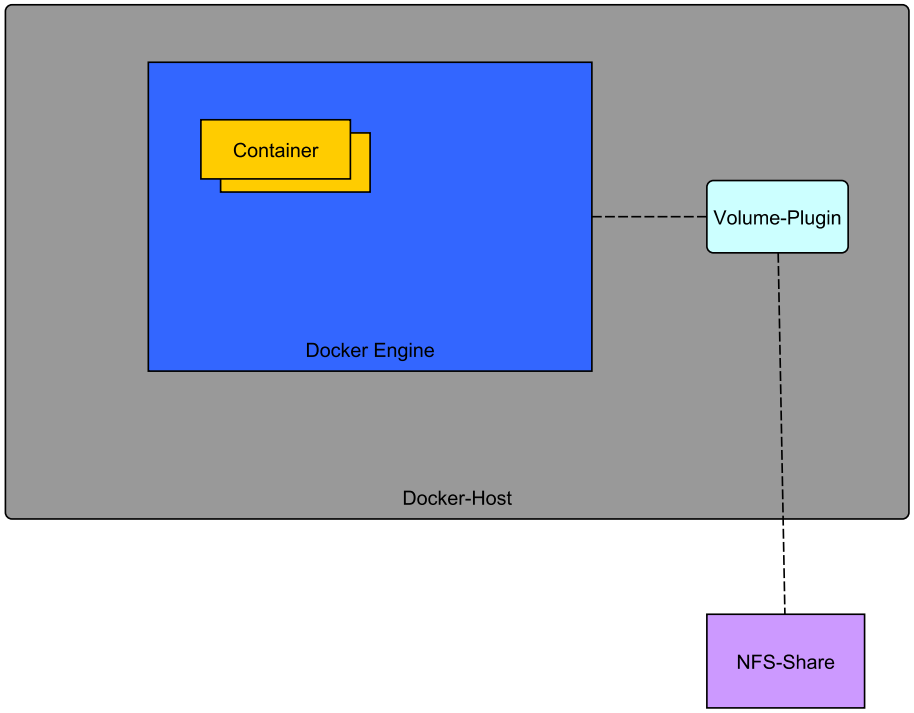


Abbildung 2.9: Ein NFS-Volumeplugin

3 Persistenzlösungen im Überblick

Im Grundlagen Kapitel wurden mit den Docker-Volumes bereits Möglichkeiten aufgezeigt, mit deren Hilfe Daten von Docker-Containern persistent gespeichert werden können. Die in Docker integrierte Volume-Lösung, stößt allerdings sobald das Volume außerhalb des lokalen Dateisystems liegt, schnell an seine Grenzen. Auch die gleichzeitige Verwendung von mehreren Host Systemen auf ein zentrales Volume bietet die native Lösung nicht. Mit der Docker-Version 1.8 wurden die Engine-Plugins eingeführt. Diese ermöglichen Drittanbietern, dass entwickeln von eigenen Docker-Plugins. Mittlerweile existieren einige Volume-Plugins welche es ermöglichen, Docker-Volumes auch über Systemgrenzen hinweg zu verwenden oder bieten weitere Zusatzmöglichkeiten wie unter anderem die Versionierung von Volumes durch Snapshots.

3.1 Marktübersicht

Nachfolgend eine Übersicht von Drittanbieterlösungen, welche den Funktionsumfang von Docker, im Bezug auf das Volume-Management erweitern. Beschrieben wird deren allgemeiner Aufbau sowie der Funktionsumfang. Um einen Einblick in die Benutzung zu geben, wird mit allen Lösungen ein kapitelübergreifendes Beispielszenario (siehe Abschnitt 3.1.1) umgesetzt. Für die Vergleichbarkeit der einzelnen Plugins werden zunächst folgende Feature-Kategorien eingeführt:

- **Unterstützte Backends** Hiermit wird aufgezeigt auf welchen Speicher-Backends sich die Volumes ablegen lassen.
- **Volume-Operationen** Die Volume-Operationen geben an, welche Möglichkeiten das Plugin für die Verwendung der Volumes bietet. Dies sind beispielsweise das Erstellen, Löschen sowie das Anlegen von Backups.
- **Multihost-Volumes** Die Multihostfähigkeit bietet die Möglichkeit das ein Volume gleichzeitig von mehreren Docker-Deamons eingebunden werden kann. Beispielsweise

kann so ein Volume, welches von einem Docker-Daemon auf einem Host A eingebunden ist, gleichzeitig von einem Docker-Daemon auf Host B verwendet werden.

- **Persistenzerhaltendes Verschieben** Diese Eigenschaft zeigt an, ob ein Container beim verschieben auf ein anderes Hostsystem ein existierendes Volume ohne Persistenzverlust erneut einbinden kann.

Für alle in diesem Kapitel aufgenommenen Plugins werden diese Feature-Kategorien erfasst. Im späteren Abschnitt Feature-Vergleich (siehe Abschnitt 3.2), folgt anschließend eine tabellarische Feature-Gegenüberstellung der einzelnen Lösungen. Dies soll eine schnelle Übersicht bieten, was die Plugin-Auswahl für gegebene Anforderungen erleichtert.

3.1.1 Beispielszenario

In diesem Beispielszenario existieren zwei Hostsysteme. Im folgenden HostA und HostB genannt. Auf HostA läuft eine Redis[1] Docker-Container-Instanz. Diese Instanz verwendet zur Persistenz ein Volume. Dieses Volume liegt je nach Lösung entweder direkt auf dem HostA oder befindet sich an einem Ort außerhalb des Hostsystems. Anschließend wird nun auf HostA der Container gestoppt und eine Redis-Instanz auf HostB gestartet. Nun soll nun ein Zugriff auf das Volume welches vorher von der Redis-Instanz auf HostA verwendet wurde, von der neuen Redis-Instanz auf HostB ermöglicht werden. Es wird jeweils davon ausgegangen das die Hostsysteme und der gegebenenfalls darunterliegende Cluster bereits fertig eingerichtet sind.

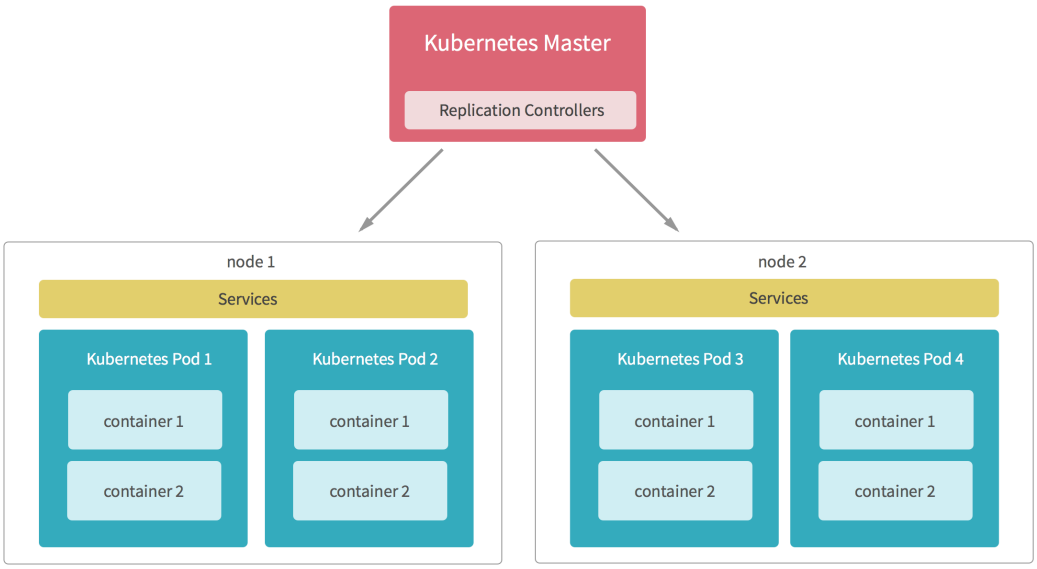
3.1.2 Kubernetes

Kubernetes ist ein von Google entwickeltes Container-Orchestrierung-Tool. Es ermöglicht wie Docker-Swarm (siehe Abschnitt 2.3.7) unter anderem eine transparente Bündelung von mehren Docker-Hosts. Der Cluster selbst kann dabei sowohl auf lokalen Maschinen betrieben werden, als auch auf Cloud Infrastructure as a Service [2] (IaaS) Anbietern. Ein Cluster besteht aus einem Master und den Arbeiter-Knoten welche auch Minions genannt werden. Auf dem Master Knoten laufen mehrere Services die für die Steuerung und Koordination zwischen den einzelnen Knoten zuständig sind. Die Container werden dabei in sogenannten Pods erstellt. Ein Pod ist eine logische Gruppe von Containern. Kubernetes stellt sicher, dass die Container eines Pods

[1] Redis ist eine Schlüssel-Wert-in-Memory-Datenbank

[2] Infrastructure as a Service (IaaS) ist ein Bereitstellungsmodell, mit dessen Hilfe ein Unternehmen die benötigte IT-Infrastruktur wie Storage, Hardware, Server und Netzwerkkomponenten mietet wenn diese benötigt wird.

immer gemeinsam auf einem Knoten des Clusters ausgeführt werden. Zur Benutzersteuerung des Clusters bietet Kubernetes sowohl ein Web UI als auch ein CLI namens *kubectl* an.



Quelle: https://cldup.com/YgsLg7gM2L.png

Abbildung 3.1: Kubernetes Architektur

Persistenz Kubernetes bietet eine native Unterstützung für eine Reihe von Speicher-Backends (Siehe Abbildung 3.2) an. Die Konfiguration der Volumes geschieht dabei direkt in den Konfigurationsdateien der Pods. Neben der direkten Verwendung von einzelnen Speicher-Backends, bietet Kubernetes mit den *Persitent Volumes* ein zweites Storage Konzept an. Dabei werden dem Cluster zunächst sogenannte PersistentVolume (PV) als Ressource zur Verfügung gestellt. Dies sind mögliche Kapazitäten von den Storage-Backends wie beispielsweise eine 100 GB NFS-Freigabe.

Der Cluster Benutzer kann anschließend mit einem PersistentVolumeClaim (PVC) Ressourcenanfragen mit einer gewünschten Speichergröße an den Cluster senden. Sobald es eine Übereinstimmung zwischen dem Claim und einem PV gibt, kann das Volume gemountet werden. Der Claim stellt damit eine Abstraktion zwischen dem Speicher und dem Bedarf da. Die Claim Anforderungen an das PV müssen dabei mindestens erfüllt sein. Die Kapazitäten des PV dürfen allerdings dabei auch darüber hinausgehen. Für die PV's werden zurzeit folgende Speicher-Backends unterstützt:

Backend
GCEPersistentDisk
AWSElasticBlockStore
NFS
iSCSI
RBD (Ceph Block Device)
Glusterfs
Host
emptyDir
gitRepo
AzureFileVolume
vsphereVirtualDisk

Abbildung 3.2: Kubernetes unterstützte Speicher-Backends

Dabei ist zu beachten, dass die Hostvariante nur auf Clustern mit einem Knoten unterstützt wird. Multi-Knoten Cluster werden bei dieser Variante nicht unterstützt. Eine weitere Besonderheit ist das EmptyDir-Volume. Dieses Volume wird, sobald einem Pod ein Knoten zugewiesen wurde, auf diesem automatisch erzeugt. Alle Container innerhalb des Pods haben Schreib-/Lesezugriff auf dieses Volume. Wird der Pod auf einen anderen Knoten verschoben, wird das Volume gelöscht.

Eine Featureübersicht von Kubernetes:

Tab. 3.1: Featureübersicht Kubernetes

Unterstützte Backends	siehe Abbildung 3.2
Volume-Operationen	Erstellen, Löschen
Multihost-Volumes	Nein
Persitenzerhaltendes Verschieben	Ja

Kubernetes im Einsatz Die Konfiguration eines PV geschieht über eine YAML-Datei. Mit *kubectl* kann auf der Basis dieser YAML-Datei anschließend ein PV erzeugt werden. Das Beispielszenario (siehe Absatz 3.1.1) wird mit dem Konzept der abstrakten Persitent-Volumes umgesetzt. Dabei kommt ein NFS-Share zum Einsatz. Dabei wird davon ausgegangen das der Cluster mit einem Master und den zwei Knoten A und B bereits existiert. Das Beispiel lässt sich allerdings in ähnlicher Weise auf die weiteren Speicher-Backends übertragen. Im ersten Schritt wird ein PV erzeugt und anschließend ein PVC, welches dann in der Pod-Konfiguration

verwendet wird. In dem Cluster existiert noch ein Storage-Server (IP: 192.168.50.135) welcher eine NFS-Freigabe anbietet. Die folgenden Befehle werden auf dem Master Knoten ausgeführt.

Zunächst wird ein 50 Gigabyte NFS-Persistent-Volume auf dem Storage-Server mit der IP 192.168.50.135 angelegt.

```
1  apiVersion: v1
2  kind: PersistentVolume
3  metadata:
4    name: pv0003
5  spec:
6    capacity:
7      storage: 50Gi
8    accessModes:
9      - ReadWriteMany
10   nfs:
11     path: /tmp
12     server: 192.168.50.135
```

Listing 3.1: nfs-pv.yaml

```
1 $ kubectl create -f nfs-pv.yaml
```

Mit den Access-Modes werden die Zugriffsrechte, sowie die Anzahl der Knoten die gleichzeitig das Volume einbinden können, festgelegt. Dabei sind folgende Access-Modes möglich.

Tab. 3.2: Volume Zugriffsmodus

ReadWriteOnce	Ein Knoten mit Schreib-Leserechten
ReadOnlyMany	Beliebig viele Knoten mit Leserechten
ReadWriteMany	Beliebig viele Knoten mit Schreib-Leserechten

Allerdings bieten nicht alle Speicher-Backends auch alle Modi an. Nachfolgend ein Überblick welche Modi jeweils unterstützt werden.

Der Claim fordert ein Storage an, welches die Anforderungen *ReadWriteMany* Modi und mindestens 3 Gigabyte Speicher erfüllt.

```
1 kind: PersistentVolumeClaim
2 apiVersion: v1
3 metadata:
4   name: myclaim-1
5 spec:
```

Tab. 3.3: Volume Zugriffsmodi der Speicher-Backends

Volume	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWSElasticBlockStore	x	-	-
Azure	x	x	x
cephFS	x	x	x
Cinder	x	-	-
FC	x	x	-
FlexVolume	x	x	-
GlusterFS	x	x	x
HostPath	x	-	-
iSCSI	x	x	-
NFS	x	x	x
RDB	x	x	-
VsphereVolume	x	-	-

```
  accessModes:
    - ReadWriteMany
  resources:
    requests:
storage: 3Gi
```

Listing 3.2: claim.yaml

```
$ kubectl create -f claim.yaml
```

Nun können zwei voneinander unabhängige Pods gestartet werden. Die beiden Host des Clusters haben bei der Erstellung ein nodeLabel mit *NodeA* bzw. *NodeB* erhalten. Mit der Hilfe des nodeSelectors wird nun sichergestellt, dass die Pods auf unterschiedlichen Host-Systemen laufen. Da das Volume nun nicht direkt auf dem Host A liegt, bleiben alle Daten des Volumes nach dem Entfernen des Pods auf dem Host A erhalten und weiterhin für den Pod B zugänglich.

```
kind: Pod
apiVersion: v1
metadata:
  name: podA
spec:
  containers:
    - name: myRedis
      image: dockerfile/redis
      volumeMounts:
```

```
      - mountPath: "/var/www/html"
        name: mypd
    nodeSelector:
       nodeLabel: NodeA
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim-1
```

Listing 3.3: podA.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: podB
spec:
  containers:
    - name: myRedis
      image: dockerfile/redis
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
    nodeSelector:
       nodeLabel: NodeB
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim-1
```

Listing 3.4: podB.yaml

```
$ kubectl create -f podA.yaml
$ kubectl delete pods -l name=podA
$ kubectl create -f podB.yaml
```

Listing 3.5: PV, PVC und Pod Erzeugung

3.1.3 Flocker

Eines der größten Projekte, die sich dem Thema Docker-Volume-Management angenommen haben, ist Flocker. Flocker ist ein von ClusterHQ [3] bereitgestellter Open-Source-Containerdaten-Volumemanager. Die einzelnen Docker-Hosts werden dabei in einem Cluster verwaltet. Unterstützt werden zurzeit Docker-Swarm, Kubernetes und Mesos. Die Docker-Volumes werden in einem Block-Storage-Backend abgelegt. Die Hauptkomponenten von Flocker bestehen aus dem Flocker-Control-Service, Flocker-Agents und Flocker-Docker-Plugin.

Flocker-Control-Service Der Flocker-Control-Service bildet das Herzstück von Flocker und läuft dabei auf einem Knoten im Cluster. Es ermöglicht die Konfiguration und Überwachung des Cluster Zustandes. Der Control-Service nimmt Anfragen über die Flocker REST[4] API entgegen. Sobald der Control-Service Anfragen entgegennimmt, sendet er diese an die betroffenen Flocker Agents weiter, welche wiederum ihren aktuellen Status an den Control-Service zurücksenden.(vgl. Flocker-A, 2016)

Flocker Agents Auf jedem Docker-Host im Cluster läuft ein Flocker Agent, welcher zentral durch den Flocker-Control-Service Knoten gesteuert wird. Diese stellen sicher, dass der Zustand des Clusters dem der Konfiguration entspricht. Zu diesem Zweck durchlaufen diese wiederholt folgende Schritte:

1. Überprüfen des lokalen Zustandes für den der Agent zuständig ist.[5]
2. Den Control-Service über den lokalen Zustand informieren.[5]
3. Berechnung die Aktionen die nötig sind, um den lokalen Zustand mit der gewünschten Konfiguration übereinzustimmen.[5]
4. Ausführung diese Aktionen.[5]
5. Starten der Schleife von vorne.[5]

Ein Beispiel wäre, dass der Control-Service den Agenten darüber informiert, dass auf dem Knoten ein Volume namens A existieren sollte. Wenn der Agent nun feststellt, dass auf dem

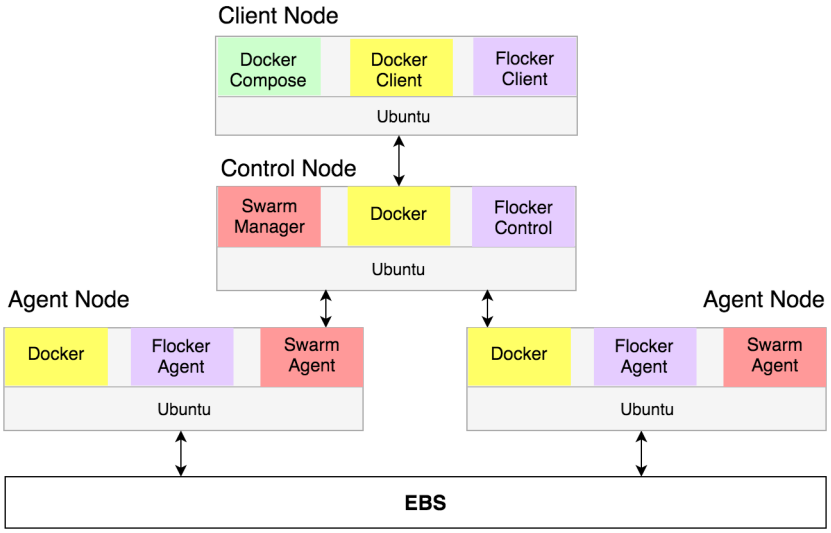
[3] https://clusterhq.com/

[4] REST(Representational State Transfer) bezeichnet ein Programmierparadigma für verteilte Systeme. Dabei wird ein per Netzwerk erreichbarer Endpunkt mit HTTP Request angesprochen.

Knoten dieses Volume nicht existiert, teilt er dies dem Control-Service mit und erstellt anschließend dieses Volume. Im Anschluss daran informiert der Agent den Control-Service über die erfolgreiche Erstellung.[5]

Flocker Docker-Plugin Das Flocker Docker-Plugin stellt die Verbindung zu den Docker-Volume-Features her. Es stellt sicher, dass Container mit benannten Volumes Zugriff auf die Daten haben, unabängig davon auf welchem Server sich diese Volumes befinden. Es existieren dabei drei Hauptfälle die das Plugin behandelt:[5]

1. Wenn das Volume noch nicht in dem Flocker Cluster existiert, wird es auf dem Host erstellt auf welchem es angefordert wurde[5]
2. Falls das Volume bereits auf einem anderen Host existiert, wird es verschoben bevor der Container gestartet wird.[5]
3. Existiert das Volume bereits auf dem Host, wird der Container umgehend gestartet.[5]

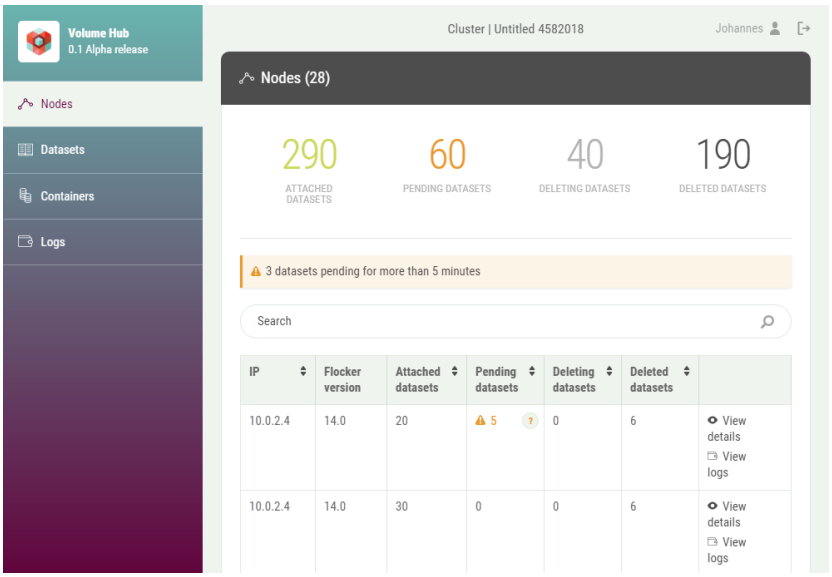


Quelle:https://clusterhq.com/flocker/introduction/

Abbildung 3.3: Flocker Übersicht

Volume Hub Mit dem Volume-Hub, stellt Flocker eine grafische Weboberfläche zur Verfügung, welches aktuelle Informationen zu dem Cluster anzeigt. Siehe Abbildung 3.4.

[5]https://docs.clusterhq.com/en/latest/flocker-features/architecture.html



Quelle:https://clusterhq.com/flocker/introduction/

Abbildung 3.4: Volume Hub

Eine Featureübersicht von Flocker:

Tab. 3.4: Featureübersicht Flocker

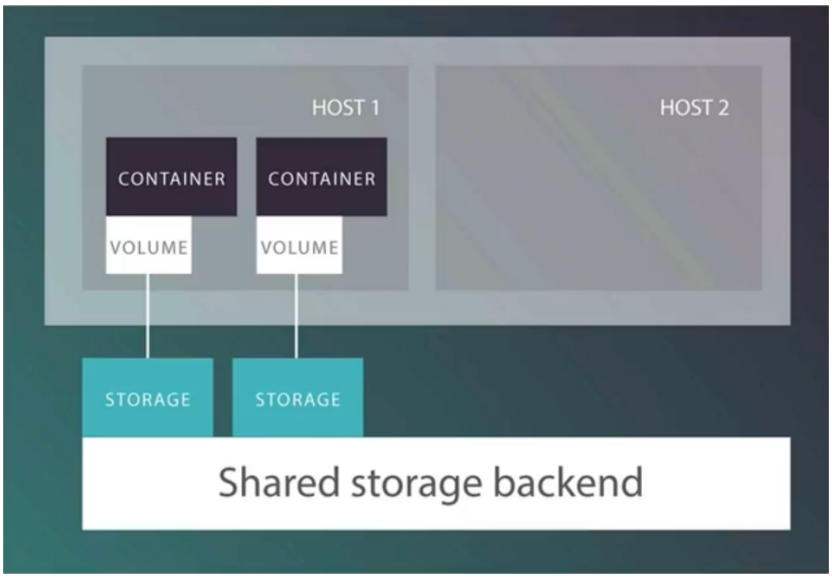
Unterstützte Backends	Amazon AWS, Google GCE, OpenStack BDB
Volume-Operationen	Erstellen, Löschen
Multihost-Volumes	Nein
Persitenzerhaltendes Verschieben	Ja

Flocker Persistenz Docker Container können wie gewohnt über den *docker run* Befehl direkt auf einem Knoten erstellt werden. Auch die Erstellung mit Docker-Compose ist möglich. Dabei kümmert sich dann der darunterliegende Cluster um die Verteilung auf die Knoten. Flocker wird dabei als Volume-Driver angegeben. Neben der impliziten Verwendung von Flocker durch das Volume-Plugin, bietet Flocker auch noch mit *flockerctl* ein CLI (Command Line Interface) zur expliziten Steuerung des Flocker-Control-Service an.

Erstellte Docker-Volumes werden nicht auf dem jeweiligen Hostsystem erstellt, sondern auf dem gemeinsam verwendetet Block Storage-Backend-System. Dabei kann aus einer ganzen Reihe von Systemen wie z. B. Amazon EBS, Cinder und vielen weiteren gewählt werden.[6] Wird

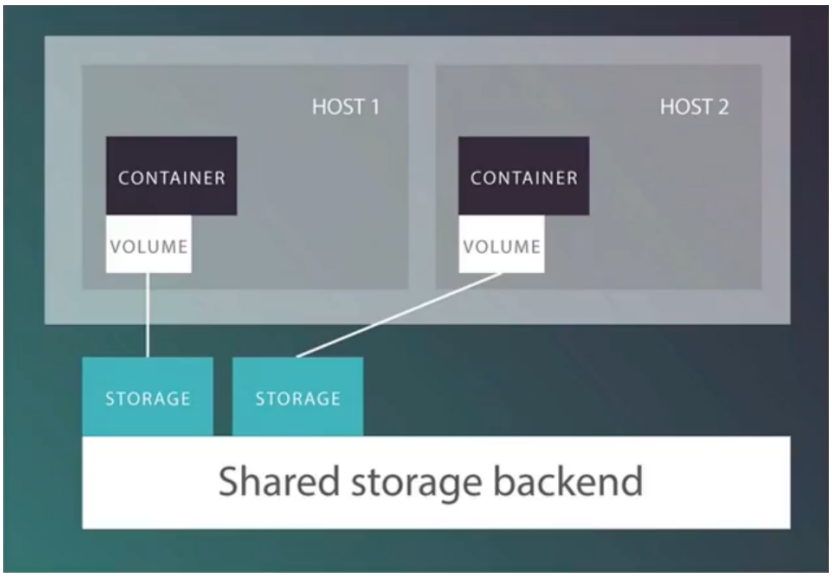
[6]https://docs.clusterhq.com/en/latest/flocker-features/storage-backends.html

nun ein Container mit einem Volume von Host 1 auf Host 2 verschoben, registriert Flocker dies und löst das Volume von dem Container auf Host 1 und bindet es anschließend auf dem Host 2 wieder ein. Flocker orientiert sich dabei an dem Namen des Containers der mit dem Parameter *–name* bei der Container-Instanziierung übergeben wird,



Quelle:https://www.youtube.com/watch?v=39wmAaUT2Y4

Abbildung 3.5: Shared Block Storage vorher



Quelle:https://www.youtube.com/watch?v=39wmAaUT2Y4

Abbildung 3.6: Shared Block Storage nachher

Flocker im Einsatz Für die Umsetzung des Beispielszenarios (siehe Absatz 3.1.1) wird in diesem Fall Docker-Compose verwendet. Es wird davon ausgegangen, dass ein Cluster mit Flocker bereits konfiguriert, sowie zwei Knoten bereits erstellt wurden. Zunächst wird eine Compose-Datei für den Redis-Container angelegt (siehe Listing 3.6).

```
1  version: "2"
2
3  volumes:
4     redis-data:
5        driver: "flocker"
6        driver_opts:
7          size: "10GiB"
8  services:
9     redis:
10      image: redis
11      ports:
12         -  "5432:5432"
13      environment:
14         - "constraint:flocker-node==1"
15      network_mode: "bridge"
16      volumes:
```

```
17          - 'postgres:/data'
```

Listing 3.6: flocker-redis-nodeA

Es wird ein Volume mit dem Namen *redis-data* angegeben. Sollte ein Volume mit diesem Namen noch nicht existieren, wird dieses von Flocker automatisch im dahinter liegenden Backend erstellt. Zu dem wird mit dem *Constraint* Parameter der Knoten auf welchem der Redis-Container instanziiert wird, explizit gewählt. Nun kann der Container per Docker-Compose gestartet werden.

```
1 $ docker-compose -f flocker-redis-node1.yml up -d
```

Listing 3.7: flocker-redis-node1-start

Um abschließend die Funktionalität zu testen, ob das Volume bei einem Hostwechsel automatisch wieder zu Verfügung steht, wird nun der Container auf dem Host 1 beendet und auf dem Host 2 neu gestartet. In der *flocker-redis-node2.yml* Compose-Datei, wurde lediglich der Constraint so weit angepasst, dass der Container auf dem zweiten Host startet.

```
1 $ docker-compose -f flocker-redis-node1.yml down -d
2 $ docker-compose -f flocker-redis-node2.yml up -d
```

Listing 3.8: flocker-redis-node2-start

Flocker prüft nun erneut ob ein Volume mit dem Namen *redis-data* bereits existiert. Da dies der Fall ist, wird das bereits vorhandene Volume zum mounten für den Container genutzt. Somit sind nun alle Daten, die bisher durch den Container im ersten Host erstellt wurden, erneut verfügbar.

3.1.4 Convoy

Convoy ist ein von der Firma Rancher Labs entwickeltes Docker-Volume-Plugin, welches als Backend das lokale Dateisystem, Virtual File System(VFS)/Network File System(NFS) sowie Amazon Elastic Block Store(EBS) ermöglicht. Die Container Management Plattform *Rancher* (siehe Abschnitt 3.1.5) verwendet Convoy als Docker-Volume-Plugin. Die Verwendung von Convoy ist allerdings auch unabhängig von Rancher möglich. Neben dem üblichen Erstellen und Löschen von Volumes, bietet das Convoy-Plugin auch die Möglichkeit Snapshots und Backups von Volumes zu erstellen und diese zur späteren Migration zu verwenden.

Eine Featureübersicht von Convoy:

Tab. 3.5: Featureübersicht Convoy

Unterstützte Backends	Lokal, NFS, Amazon EBS
Volume-Operationen	Erstellen, Löschen, Backup
Multihost-Volumes	Ja
Persitenzerhaltendes Verschieben	Ja

Convoy im Einsatz Zur Verwendung von Convoy muss zunächst der Convoy-Daemon gestartet werden. Anschließend können Volumes sowohl über das Convoy CLI als auch das Docker-CLI erstellt und verwaltet werden. Wird ein Volume beim *docker run* Befehl angegeben und dieses existiert nicht, wird dieses automatisch erstellt. Um das Beispielszenario (siehe Absatz 3.1.1) umzusetzen, wird in diesem Fall ein NFS-Share als Backend genutzt. Es wird davon ausgegangen, dass neben den zwei Hostsystemen zusätzlich ein Storage-Server, welcher den NFS-Share zur Verfügung stellt, existiert. Zunächst wird der Convoy-Daemon unter der Angabe des NFS Pfades auf jedem Host gestartet. Anschließend wird der Redis-Container mit einem Volume namens *redis-data* gestartet.

```
$ sudo convoy daemon --drivers nfs --driver-opts nfs.path=<nfs>
$ sudo docker run -it -v redis-data:/test --volume-driver=convoy redis
```

Listing 3.9: Convoy Deamonstart und Redis-Instanziierung

Wird nun der Container auf dem Host A beendet, kann unter der Angabe desselben Volumes auf dem Host B, das Volume neu gemountet werden. Alternativ wird hier die Verwendung von Snapshots demonstriert. Die Snapshots können erstellt, als Backup gespeichert und anschließend auf einem neuem Host als Volume migriert werden. Der Backup Befehl gibt als Return-Wert eine Backup-URL des Speicherortes zurück. Diese Backup-URL wird für die spätere Wiederherstellung des Backups mit *Convoy create –backup <backupURL>* benötigt.

```
$ sudo convoy snapshot create redis-data --name snap1redis-data
$ sudo convoy backup create snap1redis-data --dest nfs://meineBackups/

nfs://meineBackups/?backup=7a07c344-be75-4e55-87c8-81dc2a0\
9e8c0\u0026volume=f9fae323-fbd2-4a89-b3b2-0700b5860e88

#Login auf Host B
$
$ ssh HostB
$ sudo convoy create redis-data --backup nfs://meineBackups/?backup\
=7a07c344-be75-4e55-\87c8-81dc2a09e8c0\u0026volume=f9fae323-fbd2-4a89-\
```

```
b3b2-0700b5860e88
$
$ sudo docker run -it -v redis-data:/test --volume-driver=convoy redis
```

Listing 3.10: Convoy Snapshot Erstellung und Wiederherstellung

3.1.5 Rancher

Aufbauend auf dem Convoy-Plugin, wurde von der Firma Rancher Labs, das Open-Source-Projekt Rancher entwickelt. Rancher ist eine Container-Management Plattform. Sie ermöglicht das Bereitstellen von Container in Clustern von *Docker-Swarm*, *Kubernetes*, *Mesos* oder mit Rancher Labs eigener Cluster Lösung *Cattle*. Für die Verwaltung der Containerlandschaft bietet Rancher ein grafisches Webinterface an (Rancher UI). Aber auch die Verwendung der nativen Docker-CLI ist jederzeit möglich. Unter anderem bringt die Plattform einen integrierten Load Balancer mit. Rancher bietet auch eine verteilte DNS-basierte Service Discovery inklusive Health Checks an. Als Storage-Service wird schließlich Convoy (siehe Abschnitt 3.1.4) eingesetzt.

Zur Einrichtung von Rancher, wird auf einem Host der Rancher Server als Container gestartet.

```
sudo docker run -d --restart=always -p 8080:8080 rancher/server
```

Listing 3.11: Rancher Server start

Nach dem Starten des Rancher Servers, wird auf dem Port 8080 des Host, das Rancher UI bereitgestellt. Über dieses Interface können nun Docker-Hostsysteme hinzugefügt, sowie alle weiteren Konfigurationen vom Rancher vorgenommen werden.

Rancher-Compose Rancher Compose erweitert die Funktionen die das native Docker-Compose bietet. Es verwendet die gleichen YAML Dateien, und startet die Applikation. Zusätzlich kann noch eine rancher-compose.yml Datei definiert werden. In diese werden Rancher spezifische Einstellungen wie unter anderem die Containerskalierung, Loadblancer Konfigurationen oder auch Health-Checks konfiguriert.

```
mywordpress:
  scale: 2
wordpresslb:
  scale: 1
  load_balancer_config:
```

```
6     haproxy_config: {}
7   health_check:
8     port: 42
9     interval: 2000
10    unhealthy_threshold: 3
11    healthy_threshold: 2
12    response_timeout: 2000
13 database:
14   scale: 1
```

Listing 3.12: rancher-compose.yml Beispiel

Rancher-Katalog Der Rancher-Katalog bietet eine Reihe von Template Applikationen an, welche auf Knopfdruck deployed werden können. Ein Template besteht aus vorkonfigurierten docker-compse.yml und rancher-compse.yml Dateien. Unter anderem wird dort auch das Storage-Plugin Convoy angeboten.

Rancher Persistenz Rancher bietet als Speicher-Backend lediglich Convoy NFS an. Wird ein Container instanziiert kümmert sich Convoy um die Verwaltung und der Container erhält von jedem Hostsystem Zugriff auf die Volumes. Zur Verwendung von NFS-Shares wird zunächst ein Storage-Pool benötigt. Zu diesem Zweck stellt Rancher im Katalog das Template *Convoy NFS* zur Verfügung. Dieses Template benötigt als Eingaben den NFS Server und Mountpoint. Nach der erfolgreichen Erstellung wird innerhalb des Rancher UI ein Storage-Pool für die NFS Freigabe angelegt. Bei der Erstellung von Containern gelten folgende Regeln:

- Wenn als Volume-Driver der Storage-Pool Name angeben wird, wird das Volume auf einem der dem Storage-Pool zugehörigen Hostsystemen erstellt.
- Existiert der Volumename bereits in Rancher, wird unabhängig davon ob ein Volume-Driver angeben ist, der Container auf einem Hostsystem des Storage-Pools gestartet, welcher Zugriff auf das Volume hat.
- Wenn kein Volume-Driver angeben ist und das Volume noch nicht in Ranger existiert, wird ein lokales benanntes Volume angelegt.

Eine Featureübersicht von Rancher:

Tab. 3.6: Featureübersicht Rancher

Unterstützte Backends	NFS
Volume-Operationen	Erstellen, Löschen, Backup
Multihost-Volumes	Ja
Persitenzerhaltendes Verschieben	Ja

Rancher im Einsatz Um das Beispielszenario umzusetzen wird als Storage-Backend ein Convoy-NFS-Share verwendet. Es wird ein NFS-Storage-Pool mit dem Namen *storage* angelegt. Der Redis-Container kann nun über das Rancher UI angelegt werden. Aber auch die Erstellung über eine Docker-Compose-Datei ist möglich.

```
1 test:
2   tty: true
3   image: redis
4   stdin_open: true
5   volumes:
6   - redis-Vol:/data
7   volume_driver: storage
```

Listing 3.13: docker-compose.yml Rancher Beispiel

Mit dem Befehl *docker compose up* kann nun der Container innerhalb des Clusters instanziiert werden. Wird anschließend der Container innerhalb des Clusters auf einen anderen Host verschoben, wird aufgrund des identischen Volumenamen der Zugriff auf das Volume weiterhin ermöglicht.

3.1.6 REX-Ray Plugin

REX-Ray ist ein von EMCcode entwickeltes Docker-Volume-Plugin. Es bildet eine Abstraktionsschicht zwischen den Speicher-Backends und den Container Plattformen. Hierdurch wird die Verwendung der verschiedenen Speicher-Backends mit einheitlichen Kommandos ermöglicht. Es werden zurzeit folgende Speicher-Backends unterstützt:

Die Auswahl des Speicher-Backends wird nach der Installation von REX-Ray in einer YAML Datei konfiguriert. Anschließend wird REX-Ray als Service gestartet. Eine beispielhafte Konfiguration für ein Amazon-EC2-Backend wird in Listing dargestellt 3.14. Anschließend wird der REX-Ray Service mit dem Befehl *rexray service start* gestartet.

```
1 rexray:
```

Anbieter	Plattform
Amazon EC2	EBS
Google Computer Engine	Disk
Open Stack	Cinder
Rackspace	Cinder
EMC	ScaleIO, XtremeIO, VMAX, Isilon
Virtual Box	Virtual Media

Abbildung 3.7: Von REX-Ray unterstützte Speicher-Backends

```
  storageDrivers:
  - ec2
aws:
  accessKey: MyAccessKey
  secretKey: MySecretKey
```

Listing 3.14: REX-Ray EC2 Konfiguration

Wird ein Volume von einem Container gemountet und wird dieser Container anschließend auf einen neuen Host verschoben, wird dasselbe Volume automatisch wieder gemountet. Allerdings ist kein Multi-Host-Zugriff auf ein Volume möglich.

Eine Featureübersicht von REX-Ray:

Tab. 3.7: Featureübersicht REX-Ray

Unterstützte Backends	EMC ScaleIO, Isilon Oracle, VirtualBox Virtual Media
Volume-Operationen	Erstellen, Löschen
Multihost-Volumes	nein
Persitenzerhaltendes Verschieben	Ja

REX-Ray im Einsatz Zur Umsetzung des Beispielszenarios, wird das in der Einführung konfigurierte EC2-Backend verwendet. Nun wird zunächst auf dem Host A per Docker-CLI ein Volume mit dem Namen *redis-data* erstellt. Anschließend wird eine Redis-Instanz unter der Angabe des Volumes gestartet. Der Container kann nun gestoppt werden und auf dem Host B kann das zuvor verwendete Volume wieder verwendet werden.(siehe Listing 3.15)

```
# Host A
$ docker volume create --driver=rexray --name=redis
$ docker run -d --volume-driver=rexray -v redis-data:/data --name redis redis
```

```
$ docker stop redis
$
$
$
$ ssh HostB
$ docker run -d --volume-driver=rexray -v redis-data:/data redis
```

Listing 3.15: REX-Ray Verwendung

3.1.7 Marathon

Marathon ist eine Container-Orchestrierungs-Plattform, welche auf der Cluster-Lösung Mesos[7] aufbaut. Mesos ist ein Apache Projekt und stellt wie Docker-Swarm (siehe Abschnitt 2.3.7) und Kubernetes (siehe Abschnitt 3.1.2) ein Cluster-System da. Auch dieses Cluster-System besteht aus einer Reihe von Knoten welche von einem Master-Knoten gesteuert werden.

Marathon bietet zwei Speicherkonzepte an. Da wäre zum einen die *Local Persistent Volumes*. Bei diesen handelt es sich um Volumes die auf den Knoten des Clusters angelegt werden. Der Speicherort ist dabei jeweils das lokale Dateisystem. Diese Volumes sind allerdings damit an den Knoten gebunden. Container müssen somit zur Verwendung der Volumes auf diesem Knoten instanziiert werden.

Mit den *External Persistent Volumes* bietet Marathon auch die Verwendung von Drittanbieter Volume-Plugins an. Diese können durch die Angabe des Volume-Drivers innerhalb der Konfigurationsdateien verwendet werden.

Eine Featureübersicht von Marathon:

Tab. 3.8: Featureübersicht Marathon

Unterstützte Backends	Je nach gewählten Volume-Plugin
Volume-Operationen	Je nach gewählten Volume-Plugin
Multihost-Volumes	Je nach gewählten Volume-Plugin
Persitenzerhaltendes Verschieben	Je nach gewählten Volume-Plugin

Marathon im Einsatz Um das Beispielszenario (siehe Absatz 3.1.1) umzusetzen, werden in diesem Fall die *External Persistent Volumes* verwendet. Als Volume-Plugin kommt das schon

[7] http://mesos.apache.org/

im vorherigen Abschnitt eingeführte Rex-Ray Plugin (siehe Absatz 3.1.6) zum Einsatz. Es wird davon ausgegangen, dass der Cluster mit einem Master und den Knoten A und B bereits erstellt ist. Auch ein Volume mit dem Namen *redis-data* wurde mit dem Rex-Ray Plugin bereits erstellt. Zum Erstellen einer Redis-Instanz wird zunächst folgende Konfigurationsdatei angelegt.

```
1  {
2   "id": "redis",
3   "container": {
4     "docker": {
5        "image": "redis",
6         "network": "BRIDGE",
7         "portMappings": [
8            { "containerPort": 80, "hostPort": 0, "protocol": "tcp"}
9         ],
10        "parameters": [
11            { "key": "volume-driver", "value": "rexray" },
12            { "key": "volume", "value": "redis-data:/data/www" }
13            ]
14      }
15  },
16  "cpus": 0.2,
17  "mem": 32.0,
18  "instances": 1
19 }
```

Listing 3.16: marathon-redis-config

Im Listing 3.16 in den Zeilen 11 und 12 werden zuerst das Volume-Plugin rexray deklariert und anschließend der Container Dateipfad */data/www* dem bereits existierenden Volume *redis-data* zugeordnet. Diese Konfigurationsdatei kann nun über einen HTTP Post Request an die REST Schnittstelle des Mesos/Marathon Master Server gesendet werden, woraufhin umgehend eine Redis-Instanz auf einem zufälligem Knoten des Clusters gestartet wird. Wird nun der Knoten auf welchem die Redis-Instanz läuft heruntergefahren, registriert dies der Marathon Service und startet automatisch auf einem anderen Knoten ein neue Redis-Instanz. Die neue Instanz verweist weiterhin auf das Volume *redis-data* und hat damit auch Zugriff auf dessen Daten.

3.1.8 Blockbridge

Blockbridge bietet eine umfassende Softwarelösung für elastischen Speicher an. Speicher von mehreren Servern kann gebündelt und innerhalb des erstellten Blockbridge-Netzwerk zur Verfügung gestellt werden. Neben Sicherheit und Skalierbarkeit, wird auch die Automatisierung bei der Bereitstellung angeboten. Blockbridge-Speicher lässt sich außer mit einem grafischen Frontend auch über verschiedene APIs steuern und beispielsweise in OpenStack integrieren. (vgl. AdminMagazin, 2016)

Für den Zugriff auf den Blockbridge-Speicher, bietet der Blockbridge-Entwickler ein passendes Docker-Plugin an. Es werden dabei verschiedene Volume-Arten unterstützt.

- **Autovol** ist der Standardfall und hat ein Verhalten wie es bei Docker-Volumes üblich ist.
- **Autoclone**-Volumes kopieren zunächst von einem angebeben Snapshot die Daten als Basis. Neue Daten werden dann in diesen kopierten Snapshot-Klon geschrieben. Somit bleiben die Originaldaten von den Änderungen unberührt.
- **Snappy** stellt eine Weiterentwicklung von einem Autovol-Volume dar. Es erstellt in einem festgelegten Intervall automatisch Snapshots von dem Volume. Somit besitzt man eine Zugriffsmöglichkeit auf historische Datenzustände. Die Zeitspanne des Intervalls kann der Benutzer frei konfigurieren.

Eine Übersicht über die sonstigen Features des Blockbridge-Docker-Plugin:

Tab. 3.9: Featureübersicht Blockbridge

Unterstützte Backends	Blockbridge
Volume-Operationen	Erstellen, Löschen, Automatische Backups
Multihost-Volumes	Ja
Persitenzerhaltendes Verschieben	Ja

Blockbridge im Einsatz Für die Umsetzung des Beispielszenarios (siehe Absatz 3.1.1) , werden in diesem Fall Autovol-Volumes verwendet. Es wird davon ausgegangen, dass ein Blockbridge-Speicher-Cluster bereits existiert. Die Volumeerstellung kann dabei sowohl explizit über *docker volume create*, als auch implizit über den *docker run* Befehl erfolgen. Allerdings sind nur bei der expliziten Erstellung weitere Optionen wie zum Beispiel die Größe des Volumes oder die zwanghafte Verwendung eines SSD Laufwerk im Backend möglich.

Der Redis-Container wird auf einem Docker-Host A unter der Verwendung eines Volumes mit dem Namen *Data* gestartet. Der Container wird beendet und anschließend auf einem Host B unter der Angabe des selben Volume-Namen neu gestartet. Durch die Multihost-Fähigkeit wird sichergestellt, dass das selbe Volume wiederverwendet wird und die Daten somit erhalten bleiben.

```
#Host A
$ docker run --volume-driver blockbridge -name redistest1 -v data:/data redis
$ docker stop redistest
$
$ ssh HostB
$
$ docker run --volume-driver blockbridge -name redistest2 -v data:/data redis
```

Listing 3.17: Blockbridge-Verwendung

3.1.9 Netshare

Netshare[8] ist ein von ContainX entwickeltes Docker-Volume-Plugin. Es unterstützt dabei NFS 3/4, CIFS/Samba sowie EFS. Das Plugin spricht dabei direkt die Docker API an und bietet keine zusätzlichen Funktionen wie zum Beispiel Snapshots.

Eine Featureübersicht von Netshare:

Tab. 3.10: Featureübersicht Netshare

Unterstützte Backends	NFS, EFS
Volume-Operationen	Erstellen, Löschen
Multihost-Volumes	Ja
Persitenzerhaltendes Verschieben	Ja

Netshare im Einsatz Das Plugin wird zunächst auf den betreffenden Host als Service gestartet. Zur Umsetzung des Beispielszenarios wird ein vorhandener NFS-Share auf dem Storage-Server mit der IP: 192.168.178.2 verwendet. Beim Start wird das gewünschte Backend mit angeben (*nfs*,*cifs*,*efs*). Anschließend kann das Plugin sowohl implizit über *docker run* als auch explizit mit *docker volume* verwendet werden.

[8] http://netshare.containx.io/

```
$ #Service start
$ docker-volume-netshare nfs
$
$
$ docker run -it --volume-driver=nfs -v 192.168.178.2/data:/data redis
```

Listing 3.18: Netshare-Verwendung

Anschließend kann der Container auf dem Host A beendet und unter der Angabe desselben Volumes Pfades auf dem Host B wieder gestartet werden. Auch der gleichzeitige Zugriff von mehreren Hosts auf ein Volume ist damit möglich.

3.1.10 GlusterFS-Plugin

Das GlusterFS[9] Plugin ermöglicht das Speichern von Volumes in einem vorhandenen GlusterFS-Share. Die genaue Funktion des GlusterFS Dateisystem wurde bereits in den Grundlagen (siehe Abschnitt 2.2.2) dieser Arbeit erläutert. Es bietet neben dem Erstellen und Löschen von Volumes keine zusätzlichen Funktionalitäten an. Die GlusterFS-Volumes werden nicht durch das Anlegen eines Docker-Volumes automatisch mit erstellt, sondern müssen bereits im Vorwege manuell erstellt worden sein. Mit dem Projekt *glusterfs-rest*[10], existiert allerdings eine Erweiterung die auch die Möglichkeit einer automatischen GlusterFS-Volumegenerierung ermöglicht.

Eine Featureübersicht von GlusterFS:

Tab. 3.11: Featureübersicht GlusterFS

Unterstützte Backends	GlusterFS
Volume-Operationen	Erstellen, Löschen
Multihost-Volumes	Ja
Persitenzerhaltendes Verschieben	Ja

GlusterFS-Plugin im Einsatz Um das Beispielszenario umzusetzen, wurde bereits ein GlusterFS über die Host A und B angelegt. Auch ein GlusterFS-Volume mit dem Namen *redis-data* wurde bereits erstellt. Zunächst muss der Service unter der Angabe der Gluster Knoten gestartet werden:

[9] https://github.com/calavera/docker-volume-glusterfs
[10] https://github.com/aravindavk/glusterfs-rest

```
$ docker-volume-glusterfs -servers HostA:HostB:
```

Listing 3.19: GlusterFS-Plugin Service-Start

Anschließend kann der Redis-Container unter der Angabe des GlusterFS-Plugins auf dem Host A gestartet werden.

```
$ docker run --volume-driver glusterfs --volume redis-data:/data redis
```

Listing 3.20: GlusterFS Container-Start

Alle weiteren Container, welche auf Hostsystemen innerhalb des GlusterFS Clusters gestartet werden, haben Zugriff auf das Volume. Auch eine gleichzeitige Verwendung eines Volumes von mehreren Host ist möglich. Um auch eine automatische Volumeerstellung zu ermöglichen, wird die gluster-rest Erweiterung auf einem Knoten des GlusterFS Clusters installiert. Beim Service Start muss dieser Knoten sowie ein Basis Pfad für zukünftige Volumes angeben werden. Die erstellten Volumes werden nach deren Erstellung, auf alle im *-Server* Flag angegeben Knoten repliziert.

```
$ docker-volume-glusterfs -servers HostA:HostB \
    -rest http://HostA:9000 -gfs-base /var/lib/gluster/volumes
```

Listing 3.21: GlusterFS Service Start mit Extension

3.1.11 Horcrux Volume-Plugin

Das Horcrux[11] ist ein Docker-Volume-Projekt, welches sich auf den Umgang von großen zentralen Daten wie zum Beispiel einer SQL Datenbank konzentriert. Als Backend für die Daten unterstützt Horcrux zurzeit AWS S3, SCP, Lokal sowie Minio[12]. Die Daten werden von Horcrux in kleine Chunks umgewandelt. Werden nun vom Client Daten angefordert, muss nicht die gesamte Datenbank übertragen werden, sondern nur die benötigten Chunks. Die bereits heruntergeladenen Daten verweilen in einem Cache und bei einem erneuten Zugriff auf die gleichen Chunks werden die Daten des Caches verwendet.

Das Datenvolume wird lokal mit FUSE (**F**ilesystem in **USE**rspace) eingebunden. FUSE ist eine UNIX-Technologie und damit ist Horcurx nicht auf Windows verfügbar, sondern bietet nur eine Unterstützung für Linux und OSX Systeme an. Der Benutzer arbeitet auf einer lokalen Kopie der Daten und verändert damit nicht die Daten des zentralen Repositorys. Die Entwicklung des

[11] https://github.com/muthu-r/horcrux
[12] https://minio.io/

Plugins ist noch in einer frühen Entwicklungsphase und erst in der Zukunft (Stand 11/2016) soll es möglich sein dass Horcrux versionsverwaltende Funktionen anbietet. So soll es ermöglicht werden, dass lokale Veränderungen mit einem Kommentar mit dem Zentralen Repository synchronisiert werden.

Ebenso soll es in der Zukunft ermöglicht werden, alle historischen Versionen des zentralen Repositorys zu durchsuchen und bestimmte Versionen lokal zu verwenden. Ein Plugin welches diese Features bereits umsetzt ist das dvol (siehe Abschnitt 3.1.12) Plugin.

Eine Featureübersicht von Horcrux:

Tab. 3.12: Featureübersicht Horcrux

Unterstützte Backends	Lokal, SCP, S3, Minio
Volume-Operationen	Erstellen, Löschen
Multihost-Volumes	Ja
Persitenzerhaltendes Verschieben	Nein

Horcrux im Einsatz

Horcrux besteht aus zwei Programmteilen:

- horcrux-cli: Generiert eine horcrux Version der Daten
- horcrux-dv: Ein Docker-Volume-Plugin

Um das Beispielszenario umzusetzen wird davon ausgegangen, das neben den zwei Hostsystemen A und B, auch noch ein Storage-Server vorhanden ist, auf welchem die Daten abgelegt werden. Zunächst muss eine Horcrux-Version der Redis Daten erzeugt werden. Dies wird mit dem Kommandozeilen-Tool *horcrux-cli* bewerkstelligt.

```
$#                    Name   Quelle          Ziel
$ horcrux-cli generate redis /redis/data      ./horcrux-redis-data
```

Listing 3.22: Horcrux-Daten-Generierung

Im Anschluss daran wird auf den Docker-Hosts das Volume-Plugin gestartet.

```
$ .\horcrux-dv
```

Listing 3.23: Horcrux Volume-Plugin-Start

Die implizite Volumeerstellung wird nicht unterstützt, somit müssen die Volumes wie in Listing 3.24 dargestellt, erst mit dem Befehl *docker volume create* erstellt werden. Dabei wird die Quelle, wo sich die von Horcrux generierten Daten befinden, angegeben. Anschließend kann der Container unter der Angabe des Volumenamens instanziiert werden.

```
$ docker volume create --name v1 -d horcrux -o --name=redis \
-o --access=scp:///username@testPC:/usr/test/horcrux-redis-data

$ docker run -it --name testContainer -v v1:/data
```

Listing 3.24: Horcrux Volume-Plugin Start

Das Beispielszenario lässt sich zurzeit mit diesem Plugin nicht zufriedenstellend umsetzen, da die Veränderungen in der Redis-Datenbank, die auf dem Host A auftreten, auf dem Host B nicht zum Tragen kommen würden.

3.1.12 dvol

Das von ClusterHQ entwickelte Docker-Volume-Plugin *dvol*, ermöglicht versionsverwaltende Operationen auf ein Docker-Volume. Es lassen sich von dem Volume Entwicklungszweige (Branch) erzeugen und auf diese können Dateiveränderungen festgeschrieben (Commit) werden. Es wird eine Historie der Commits angelegt, wodurch es auch ermöglicht wird, zu früheren Commits zurückzuspringen (Reset). Dies ermöglicht es Entwicklern beispielsweise, zum Testen einer Datenbank, immer wieder zu einem genau definierten Punkt zurückzuspringenn.

Eine Featureübersicht von dvol:

Tab. 3.13: Featureübersicht dvol

Unterstützte Backends	Lokal
Volume-Operationen	Erstellen, Löschen, Commit, Checkout, Reset
Multihost-Volumes	–
Persitenzerhaltendes Verschieben	–

dvol im Einsatz Eine konkrete Anwendung zu dem Beispielszenario lässt sich, da der Fokus des Plugins auf einem ganz anderen Bereich liegt, in diesem Falle nicht herstellen. Daher hier eine allgemeine Demonstration der Verwendung des Plugins.

Nach der Installation des dvol Plugins wird zunächst ein Redis-Container durch Docker-Compose instanziiert.

```
redis:
 image: redis
 volumes:
   - "redis_data:/data"
 volume_driver: dvol
```

Listing 3.25: dvolredis.yml

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

Listing 3.26: dvol Redis Start

Nun kann beispielsweise ein neuer Branch namens *dev* angelegt werden. Auf diesen wird ein Commit *cleanState* angelegt. Nun könnte man sich vorstellen das einige Veränderungen in der Datenbank vorgenommen werden. Mit einem *reset* kann anschließend wieder zu dem Commit *cleanState* zurückgesprungen werden.

```
$ dvol checkout -b dev
$ dvol commit -m "clean state"
commit id: 6a51fc9353934911a9af8b9e36b6c097a1353e72

# Nach einigen DB Veränderungen
$
$ dvol reset --hard  6a51fc9353934911a9af8b9e36b6c097a1353e72
$
```

Listing 3.27: dvol Verwendung

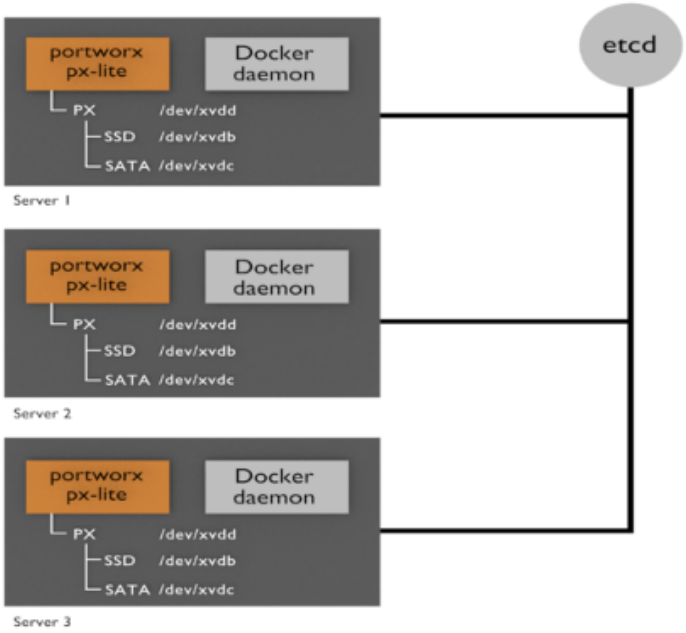
3.1.13 PX-Developer

PX-Developer ist ein von Portworx[13] entwickeltes Docker-Volume-Plugin. Dieses ermöglicht ähnlich wie GlusterFS2.2.2 vorhanden Speicher in einem Cluster von Knoten transparent zu bündeln und verfügbar zu machen. Volumes die auf einem der beteiligten Knoten des Clusters erstellt werden, werden automatisch durch PX-Developer auf alle weiteren Knoten repliziert. Dadurch wird eine hohe Verfügbarkeit sichergestellt.

[13] http://portworx.com/

Die Knoten des Clusters können sowohl dedizierte Linux Server sein, als auch Server von IaaS (Infrastructure as a Service) Anbietern wie beispielsweise Amazon AWS. Ebenso ist die Verwendung in Kubernetes Pods möglich. Die Volumeerstellung kann dabei implizit durch das Docker-CLI oder alternativ auch durch das von Portworx bereitgestellte Tool *pxctl* erfolgen. Die Entwicklung ist noch in einem Beta Stadium und die maximale Anzahl von Knoten im Cluster ist zurzeit noch auf 3 beschränkt.

Zum Betrieb von PX-Developer, wird auf jedem der Knoten im Cluster ein PX-Developer Container gestartet. Zusätzlich wird noch eine Key/Value Datenbank zur Volume-Verwaltung benötigt. PX-Developer setzt zu diesem Zweck eine vorhandene etcd[14] Datenbank voraus. Alle Knoten des Clusters müssen Zugang zu dieser Datenbank erhalten.



Quelle: https://raw.githubusercontent.com/portworx/px-dev/master/images/cluster.png

Abbildung 3.8: PX-Developer Übersicht

PX-Control-Tool Mit dem PX-Control-Tool kurz pxctl, bietet Portworx ein CLI an um den erstellten Cluster zu verwalten. Das pxctl bietet immer eine globale Sicht auf den gesamten Cluster. Somit lassen sich unter anderem die verwendeten Festplattenkapazitäten der einzelnen

[14] https://coreos.com/etcd/

Knoten, aber auch den freien und bereits verbrauchten Platz des gesamten Clusters anzeigen. Zusätzlich bietet es die Möglichkeit die Volumes und deren Snapshots zu verwalten.

```
# pxctl help
NAME:
   pxctl - px cli

USAGE:
   pxctl [global options] command [command options] [arguments...]

VERSION:
   0.4.3-9da1bcd

COMMANDS:
   status       Show status summary
   volume, v    Manage volumes
   snap, s      Manage volume snapshots
   cluster, c   Manage the cluster
   container    Display containers in the cluster
   service, sv  Service mode utilities
   host         Attach volumes to the host
   eula         Show license agreement
   help, h      Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --json, -j           output in json
   --color              output with color coding
   --raw, -r            raw CLI output for instrumentation
   --help, -h           show help
   --version, -v        print the version
```

Listing 3.28: PX-Control-Tool Befehlsübersicht

Eine Featureübersicht von PX-Developer:

Tab. 3.14: Featureübersicht PX-Developer

Unterstützte Backends	Erstellter PX-Developer Cluster
Volume-Operationen	Erstellen, Löschen
Multihost-Volumes	Ja
Persitenzerhaltendes Verschieben	Ja

PX-Developer im Einsatz Für die Umsetzung des Beispielszenarios wird auch an dieser Stelle davon ausgegangen, dass der Cluster inklusive der etcd Datenbank bereits vorkonfiguriert ist. Die Volumes lassen sich über den *docker volume create* Befehl erstellen. Über zusätzliche Optionen lassen sich wie in diesem Beispiel auch das Dateiformat und die Größe des Volumes angeben. Anschließend ist es von jedem Knoten des Clusters aus möglich das Volume gleichzeitig zu verwenden.

```
# Host A
$ docker volume create --driver=pxd --name=redis-data \
 --opt format=ext4 --opt size=1G
$
$ docker run -v redis-data:/data --name redisHostA redis


# Host B
$ docker run -v redis-data:/data --name redisHostB redis
$
```

Listing 3.29: Volumeerstellung über Docker-CLI

3.1.14 Azure-Docker-Volumedriver

Dieses Volume-Plugin[15] ermöglicht die Verwendung von Azure Storage Ressourcen. Die Migration eines Containers von einem Host zu einem anderen, sowie die gleichzeitige Verwendung eines Volumes von mehreren Containern auf unterschiedlichen Hostsystemen wird damit ermöglicht.

Eine Featureübersicht vom Azure-Docker-Volumedriver:

Tab. 3.15: Featureübersicht Azure-Docker-Volumedriver

Unterstützte Backends	Azure File Storage
Volume-Operationen	Erstellen, Löschen
Multihost-Volumes	Ja
Persitenzerhaltendes Verschieben	Ja

Azure-Docker-Volume-Driver im Einsatz Nach der Installation des Plugins können die Volumes direkt über das Docker-CLI erstellt werden. Dabei ist sowohl die explizite Erstellung

[15] https://github.com/Azure/azurefile-dockervolumedriver

mit dem Befehl *docker volume create* als auch die implizite Erstellung mittels des *docker run -v* Befehls möglich. Für die Umsetzung des Beispielszenarios wird zunächst auf dem Host A ein Volume für eine Redis-Instanz erstellt, diese anschließend zunächst auf Host A eingebunden und schließlich auf dem Host B von einer weiteren Redis-Instanz verwendet.

```
# Host A
$ docker volume create --name redis-data -d azurefile -o share=azureShare
$ docker run -v redis-data:/data redis


# Host B
$ docker run -v redis-data:/data redis
$
```

Listing 3.30: Volume-Verwendung über das Docker-CLI

3.2 Featurevergleich

Um einen schnellen Überblick über die Features der einzelnen Plugins zu ermöglichen, folgt an dieser Stelle eine Übersichtstabelle, welche die Features der Plugins gegenüberstellt (siehe Abbildung 3.9).

Plugin	Backends	Volume Operationen	Multihost Volumes	Persistenzerhaltendes Verschieben
Convoy	Device Mapper Network File System(NFS) Amazon Elastic Block Store(EBS)	Erstellen, Löschen, Backup	ja	ja
Flocker	Amazon AWS Google GCE OpenStack Block Device	Erstellen, Löschen	Nein	ja
Rancher	Local Network File System(NFS)	Erstellen, Löschen	Ja	ja
Rex-Ray	EMC ScaleIO, Isilon Oracle VirtualBox Virtual Media	Erstellen, Löschen	Nein	ja
Mesos/ Marathon	Abhängig vom gewählten Volume Plugin	Abhängig vom gewählten Volume-Plugin	Abhängig vom gewählten Volume-Plugin	Abhängig vom gewählten Volume-Plugin
Blockbridge	Blockbridge storage	Erstellen, Löschen, Backup	Ja	ja
Netsare	NFS, EFS	Erstellen, Löschen	ja	ja
Horcrux	Lokal, SCP, S3, Minio	Erstellen, Löschen	ja	nein
portworx	Lokaler Speicher wird im Cluster verteilt	Erstellen, Löschen, Backup	ja	ja
Kubernetes	GCP, AWS, NFS, ISCSI, Gluster, CEPH, Cinder, Flocker	Erstellen, Löschen	Nein	ja
glusterfs	GlusterFS	Erstellen, Löschen	ja	ja
Azure	Azure File Storage	Erstellen, Löschen	ja	ja
dvol	Lokal	Erstellen, Löschen	Nein	Nein

Abbildung 3.9: Featurevergleich

3.3 Zusammenfassung

Dieses Kapitel bietet einen Überblick der heutzutage am Markt erhältlichen Volume-Plugins. Es zeigt sich, dass die Einführung der Engine-Plugins und die damit einhergehenden entwickelten Volume-Plugins, zu einer sinnvollen Weiterentwicklung der nativen Docker-Persistenz-Möglichkeiten geführt hat. Mittlerweile werden die gebräuchlichsten Speicher-Backends unterstützt und mit Plugins wie dvol (siehe Abschnitt 3.1.12) werden auch neuartige Speicher-Konzepte eingeführt.

4 Prakische Umsetzung einer Persistenzlösung

In dem vorherigen Kapitel wurde eine Übersicht über die aktuell verfügbaren Persistenzlösungen aufgezeigt. Um den Prozess von gegebenen Anforderungen hin zu einer geeigneten Persistenzlösung zu demonstrieren, wird in diesem Kapitel für ein reales Projekt eine geeignete Persistenzlösung entwickelt. Zunächst wird dafür anhand der Anforderungen ein geeignetes Volume-Plug-in ausgewählt. Anschließend wird für das Projekt ein Konzept zur Docker-Persistenz erarbeitet, dieses praktisch umgesetzt und abschließend ein Fazit gezogen. Für diese Demonstration diente ein Projekt, welches zurzeit an der Universität HAW Hamburg im Informatik Departement mit dem Namen *MARS* in der Entwicklung ist.

4.1 MARS Projekt

Das MARS (Multi Agent Research and Simulation) Projekt ist eine Forschungsgruppe an der HAW Hamburg im Informatik Departement. Diese entwickelt ein Multi Agenten Simulationssystem für die interdisziplinäre Entwicklung.

> *„Die MARS-Gruppe beschäftigt sich mit komplexen Fragestellungen und Problemfeldern des Alltags, z.B. Auswirkungen der globalen Klimaveränderung, Ausbreitung von Infektionskrankheiten oder dem menschlichen Verhalten in Extremsituationen. Zielsetzung ist es, gemeinsam mit den jeweiligen Fachexperten, effiziente Software-Werkzeuge für die Modellbildung, die Simulation, die Auswertung und Visualisierung zu entwickeln. Dabei stehen der Praxisbezug sowie eine exzellente wissenschaftliche Ausrichtung im Fokus.“* - MARS Group[1]

Das Ziel soll eine Art MSaas (Modeling and Simulation as a Service) darstellen. Mit den bereitgestellten grafische Werkzeugen zur Erstellung von Agentenlogik wird es Domänenexperten

[1] http://mars-group.mars.haw-hamburg.de/framework/

wie zum Beispiel Biologen, Ökologen oder Medizinern ermöglicht eigene Modelle zu designen. (vgl. MARS2016, 2016)

4.1.1 Infrastruktur

Die einzelnen Bestandteile der MARS Software werden jeweils als Docker-Container innerhalb eines Docker Swarm Clusters gestartet. Der Docker-Swarm Cluster (siehe Abschnitt 2.3.7) besteht zurzeit aus 5 Mac Pro Knoten. Auf diesen läuft als Basis ein Ubuntu System, auf welchem wiederum für die Virtualisierung ein Linux KVM Typ-2-Hypervisor (siehe Abschnitt 2.1.4) verwendet wird. Auf einer der virtuellen Maschinen des jeweiligen Knoten, läuft ein System mit einem Docker-Daemon. Als gemeinsamer Speicher steht ein NAS Gerät, welches über 1 Terabyte SSD sowie 40 Terabyte HDD verfügt, zur Verfügung. Die Verwaltung wie beispielsweise das starten und stoppen der Container geschieht über Docker-Compose (siehe Abschnitt 2.3.5).

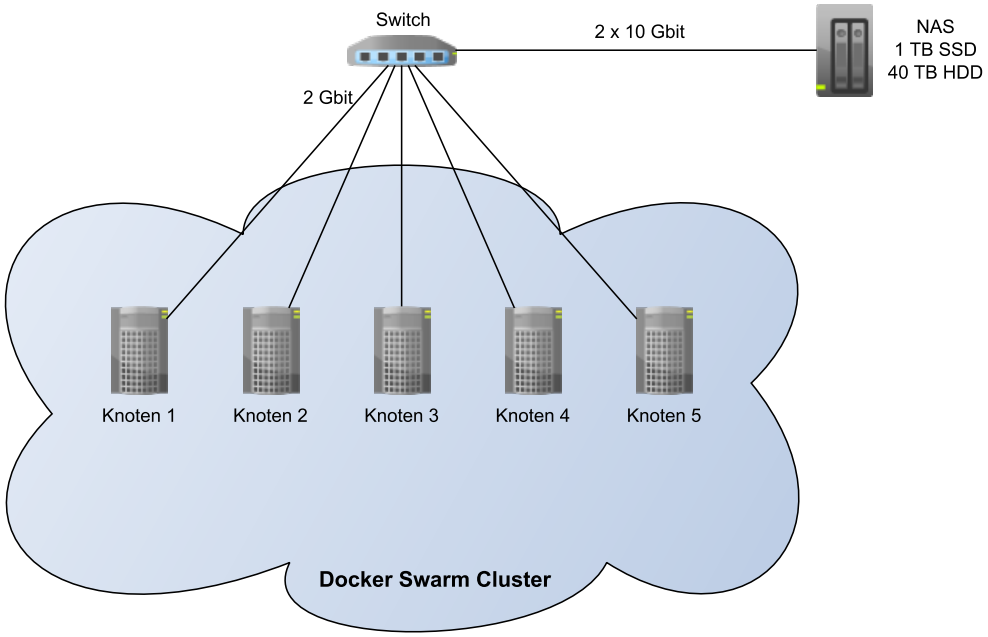


Abbildung 4.1: Projekt MARS Architektur

4.1.2 Architektur

Die Mars-Software verwendet eine Microservice-Architektur. Bei diesem Architekturkonzept wird die Software in einzelne modulare Dienste aufgeteilt. Zur Kommunikation zwischen den Diensten müssen diese jeweils eine Schnittstelle anbieten. Dazu nutzen Microservices Protokolle, die lose Kopplung unterstützen, wie beispielsweise REST oder Messaging-Lösungen. Microservices können in unterschiedlichen Technologien implementiert sein. Es gibt keine Einschränkung auf eine bestimmte Programmiersprache oder Plattform. (vgl. Wolff, 2015)

Die einzelnen Dienste sind in einen Docker-Image verpackt, welche in dem Docker-Swarm-Cluster mit der Hilfe von Docker-Compose als Container gestartet werden. Die MARS-Software umfasst zum Zeitpunkt des erstellen dieser Bachelorarbeit 22 verschiedene Container-Images. Dies sind unter anderem Dienste zur Servicesuche, Lastverteilung, Datenbanken, fachliche Komponenten sowie auch des Frontends.

4.2 Anforderungsanalyse

Das MARS Gesamtsystem besteht aus einer ganzen Reihe von Docker-Containern. Viele dieser besitzen zur persistenten Speicherung der Daten lokale Volumes. Da die Container innerhalb eines Swarm Clusters, bei jedem Start durch die Cluster-Verteilungsstrategie auf einem anderen Knoten als beim vorherigem Start landen können, ist der Zugriff auf die vorherigen Daten nicht immer gewährleistet. Aus diesem Grund wurde für das Projekt die Verwendung eines NFS fähigen Docker-Volume-Plug-in angedacht. Die Speicherung soll so direkt auf dem zentralen NAS stattfinden. Dadurch wäre eine freie Zuordnung der Container auf die Knoten, bei gleichzeitiger Datenerhaltung gewährleistet. Das Plugin soll zudem dafür sorgen, dass ein Volume welches durch einen Container auf einem Knoten A erstellt wurde, automatisch bei dem Start des Containers auf einem Knoten B eingebunden wird. Auch eine Backup/Snapshot Möglichkeit der Volumes war eine Anforderung an das Plugin. Der bereits verwendete Docker-Swarm soll weiterhin im Einsatz bleiben. Nachfolgend sind die einzelnen Punkte der Anforderung nochmals aufgelistet.

1. Zentrale Speicherung von Volumes in einem NFS-Share
2. Automatisches einbinden der Volumes beim Verschieben der Container zwischen den Knoten
3. Backup/Snapshot Möglichkeiten für Volumes

4. Die Weiterverwendung von Docker-Swarm ist möglich
5. Ein schlankes Plugin und keine große allumfassende Plattform

4.3 Persistenzauswahl

Zur Eingrenzung der möglichen Volume-Plugins wurde zunächst die Übersichtstabelle aus dem Featurevergleich verwendet (siehe Abschnitt 3.2). Daraus ergibt sich, dass folgende Plugins grundsätzlich NFS-Shares unterstützen:

- Rancher (siehe Abschnitt 3.1.5)
- Kubernetes (siehe Abschnitt 3.1.2)
- Netshare (siehe Abschnitt 3.1.9)
- Convoy (siehe Abschnitt 3.1.4)

Rancher bringt neben Volume-Funktionen noch eine ganze Reihe weiterer Funktionen mit und stellt eine komplette Managementplattform dar. Innerhalb des MARS Projektes war allerdings die Verwendung einer schlanken Lösung gewünscht, womit Rancher ausschied.

Tab. 4.1: Rancher Anforderungsübereinstimmung

Zentrale Speicherung von Volumes in einem NFS-Share	X
Automatisches einbinden der Volumes beim Verschieben der Container zwischen den Knoten	X
Backup/Snapshot Möglichkeiten für Volumes	X
Die Weiterverwendung von Docker-Swarm ist möglich	X
Eine schlanke Plugin Lösung	

Da weiterhin der Docker-Swarm verwendet werden sollte, schied auch Kubernetes aus, da Kubernetes eine eigene Clusterlösung darstellt. Desweiteren stellt Kubernetes keine schlanke Lösung dar und bietet auch keine implizite Backup/Snapshot Möglichkeit an.

Tab. 4.2: Kubernetes Anforderungsübereinstimmung

Zentrale Speicherung von Volumes in einem NFS-Share	X
Automatisches einbinden der Volumes beim Verschieben der Container zwischen den Knoten	X
Backup/Snapshot Möglichkeiten für Volumes	
Die Weiterverwendung von Docker-Swarm ist möglich	
Eine schlanke Plugin-Lösung	

Netshare als reines Volume-Plugin erfüllt die meisten Anforderungen, allerdings bietet Netshare keine Möglichkeiten für Backups/Snapshots von Volumes an.

Tab. 4.3: Netshare Anforderungsübereinstimmung

Zentrale Speicherung von Volumes in einem NFS-Share	X
Automatisches einbinden der Volumes beim Verschieben der Container zwischen den Knoten	X
Backup/Snapshot Möglichkeiten für Volumes	
Die Weiterverwendung von Docker-Swarm ist möglich	X
Eine schlanke Plugin-Lösung	X

Convoy als die letzte Lösung dieser Liste bietet, wie Netshare, die einfache Verwendung von NFS-Shares und bietet zusätzlich noch die Möglichkeit der Erstellung von Backups/Snapshots von Volumes. Somit wurde das Volume-Plug-in Convoy für die Umsetzung der Anforderungen ausgewählt.

Tab. 4.4: Convoy Anforderungsübereinstimmung

Zentrale Speicherung von Volumes in einem NFS-Share	X
Automatisches einbinden der Volumes beim Verschieben der Container zwischen den Knoten	X
Backup/Snapshot Möglichkeiten für Volumes	X
Die Weiterverwendung von Docker-Swarm ist möglich	X
Eine schlanke Plugin-Lösung	X

4.4 Umsetzung

Für die Umsetzung wurde zunächst auf dem NAS-Speicher ein NFS-Share-Ordner mit dem Namen *Volume* freigegeben. Auf allen Knoten des Clusters wurde anschließend dieser NFS Pfad unter dem Pfad /mnt/convoynfs eingehängt. Anschließend wurde auf allen Knoten des Clusters das Convoy-Plug-in installiert. Dazu wurde zunächst die letzte verfügbare Version heruntergeladen, das Paket entpackt und die Binärdateien in die Systempfade kopiert. Abschließend wurde in der Konfiguration von Docker das Convoy-Plug-in bekannt gemacht.

```
$ wget https://github.com/rancher/convoy/releases/download/v0.5.0-rc1/
 convoy.tar.gz
$ tar xvf convoy.tar.gz
$ sudo cp convoy/convoy convoy/convoy-pdata_tools /usr/local/bin/
$ sudo mkdir -p /etc/docker/plugins/
$ sudo bash -c 'echo "unix:///var/run/convoy/convoy.sock" >
/etc/docker/plugins/convoy.spec'
```

Listing 4.1: Convoy Installation

Um sicherzustellend das der Convoy-Daemon bei jedem Systemstart korrekt gestartet wird, wurde ein *Ubuntu systemd Service* erstellt.

> *„systemd ist ein System- und Sitzungs-Manager (Init-System), der für die Verwaltung aller auf dem System laufenden Dienste über die gesamte Betriebszeit des Rechners, vom Startvorgang bis zum Herunterfahren, zuständig ist. Prozesse werden dabei immer (soweit möglich) parallel gestartet, um den Bootvorgang möglichst kurz zu halten.“* [2]

Ein systemd Service wird über eine Textdatei konfiguriert. Der Convoy-Service sorgt zunächst dafür, dass auf den Docker Service gewartet wird. Sobald dieser gestartet ist, wird der Convoy-Daemon unter der NFS-Pfadangabe */mnt/convoynfs* gestartet. Convoy wird anschließend alle erstellten Volumes innerhalb dieses Ordners ablegen.

```
[Unit]
Description=Convoy Daemon
Requires=docker.service

[Service]
ExecStartPre=/usr/bin/sudo rm -r -f /var/lib/rancher
```

[2] https://wiki.ubuntuusers.de/systemd/

```
ExecStart=/usr/bin/sudo convoy daemon --drivers vfs
 --driver-opts vfs.path=/mnt/convoynfs

[Install]
WantedBy=multi-user.target
```

Listing 4.2: Convoy systemd Service Konfigurationsdatei

Die bereits vorhandene Docker-Compose Datei musste lediglich um die Angabe des Convoy-Volume-Drivers und Volume-Namen ergänzt werden.

```
volumes:
  geoserver-data:
    driver: convoy


geoserver:
   image: artifactory.mars.haw-hamburg.de:5002/geoserver_master:latest
   networks:
       - marscloud
   volumes:
       - geoserver-data:/opt/geoserver/data_dir/
```

Listing 4.3: Ein Auszug der angepassten Compose Datei

4.5 Lösungsbewertung

Durch die zentrale Speicherung war es nun möglich, dass Container ohne Persistenzverlust zwischen den Swarm Knoten wechseln konnten. Auch die Möglichkeit von Backups und Snapshots waren durch Convoy gegeben. Beachtet werden sollte, dass durch die Verwendung eines Netzwerkspeichers, je nach verfügbarer Netzwerkleistung, ein Performancenachteil gegenüber einer lokalen Speicherung entstehen kann. In diesem Einsatzszenario gab es in diesem Bereich allerdings keine Einschränkung. Es zeigte sich das die Umstellung auf ein Volume-Plug-in innerhalb eines bestehenden Systems, im Falle von Convoy, relativ einfach möglich ist. Dies liegt vor allem an dem modularen Konzept des Plug-in-Mechanismus. Für die Anwendungen innerhalb des Containers zeigt sich eine Änderung des Volume-Plugins komplett transparent und bedarf an dieser Stelle keiner Anpassung.

5 Schlussbetrachtung

5.1 Zusammenfassung

Diese Arbeit hat sich mit dem Thema der Docker-Persistenz beschäftigt. Es wurde untersucht, inwiefern sich Docker in Bezug auf die Persistenz-Möglichkeit durch Plugins erweitern lässt. Dazu wurde zunächst eine Übersicht der zurzeit gebräuchlichen verfügbaren Docker-Volume-Plugins erstellt.

Dabei ergab sich, dass mittlerweile eine ganze Reihe von Plugins existieren, welche die Docker-Persistenz-Möglichkeiten erweitern. Um die einzelnen Plugins miteinander vergleichen zu können wurden zunächst einige Feature-Kategorien eingeführt. Diese waren beispielsweise welche Speicher-Backends die Plugins unterstützen und welche Operationen anschließend mit diesen Volumes getätigt werden können. Die Eigenschaften der Plugins wurden den Feature-Kategorien zugeordnet und tabellarisch vergleichbar gegenüber dargestellt.

Es stellte sich heraus, dass mittlerweile für die gebräuchlichsten Speicher-Backends Plugins existieren. Dies umfasst zum einen typische Backends wie NFS und Cloud-Speicheranbieter, aber auch verteilte Dateisysteme wie beispielsweise GlusterFS werden unterstützt.

Ein Teil der Plugins sind dabei recht schlanke Lösungen und können unabhängig verwendet werden. Andere Plugins hingegen, wie beispielsweise bei Kubernetes, sind innerhalb einer komplexen Docker-Containerverwaltung integriert und erweitern die Docker-Möglichkeiten über die reine Volume-Verwaltung hinaus.

Die praktische Verwendung der verschiedenen Plugins gestaltetet sich, dank des Docker-Plugin-Mechanismus, welches für die Volume-Pugins eine API anbietet, problemlos. In der Regel bedarf es, neben der Installation des Plugins auf dem betreffenden System, nur die Angabe des Plugins-Namen während der Verwendung der Container. Die Erstellung der Volumes geschieht dabei meist implizit über den Docker-Client. Manche Plugins bringen daneben zusätzlich auch noch ein eigenes CLI/GUI zur Steuerung mit.

Im späteren Verlauf dieser Arbeit wurde schließlich für ein Forschungsprojekt an der HAW Hamburg ein Docker-Persistenzkonzept erstellt. Dieses basiert auf dem Docker-Volume-Plugin *Convoy*. Dabei wurden zunächst die Anforderungen der Forschungsgruppe aufgenommen. Anschließend konnte anhand der im Kapitel 3 gewonnen Erkenntnisse das Plugin *Convoy* als das geeignetste ausgewählt werden. Testweise wurde abschließend diese Lösung umgesetzt und dokumentiert wie das System für die Verwendung mit *Convoy* konfiguriert wurde.

Insgesamt lässt sich hieraus der Schluss ziehen, dass durch die Einführung der Docker-Engine-Plugin-API und der damit entwickelten Docker-Volume-Plugins, die Persistenz-Möglichkeiten von Docker umfassend erweitert wurden. Durch das Plugin-System besteht nicht mehr die Abhängigkeit von den Docker-Entwicklern, dass diese gewünschte Speicher-Backends und weitere Features implementieren. Die Vielzahl der mittlerweile existierenden Plugins zeigt, dass die Möglichkeit der Eigenentwicklung gerne in Anspruch genommen wird.

5.2 Ausblick

Im Kapitel 4 wurde anhand des HAW MARS-Projekts ein geeignetes Persistenz-Konzept entwickelt und testweise umgesetzt. Da sich das MARS-Projekt noch in der laufenden Entwicklung befindet, ändern sich auch gelegentlich die technischen Anforderungen und Gegebenheiten. Ursprünglich haben eine Reihe von Containern Volumes zur Speicherung der Daten benötigt. Im Laufe dieser Bachelorarbeit haben sich allerdings diese Anforderungen geändert.

Mittlerweile existiert ein MongoDB-Datenbank-Cluster, welcher eine API anbietet, so dass alle Container ihre produzierten Daten in diesem Datenbank-Cluster speichern können. Der Datenbank-Cluster besteht dabei aus mehreren MongoDB-Instanzen, welche selbst auch als Container innerhalb des Swarm-Clusters gestartet werden. Dabei stellen allerdings bestimmte Bedingungen sicher, dass ein MongoDB-Container immer wieder auf demselben Knoten gestartet wird. Somit wird es für die Datenbank ermöglicht, dass das lokale Dateisystem des Knoten ohne Persistenzverlust als Volume-Speicherpunkt verwendet werden kann. Als Backuplösung wird ein eigenes Programm entwickelt, welches über die angebotenen Backup-API des MongoDB-Clusters die Daten sichert.

Unter diesen Voraussetzungen können die gesamten Projektanforderungen mit den nativen Docker-Möglichkeiten erfüllt werden und es ist kein weiteres Docker-Plugin nötig. Somit kam die ursprüngliche entwickelte Lösung mit *Convoy* schlussendlich nicht zum Einsatz. Es ist

allerdings nicht auszuschließen, dass in der Zukunft durch eine weitere Umstrukturierung des Projektes wieder eine Plugin-Unterstützung für die Persistenz des Projektes benötigt wird.

Anhang

Abbildungen

Listings

Literaturverzeichnis

[AdminMagazin 2016] : *Admin Magazin*. 2016. – URL http://www.admin-magazin.de/News/Blockbridge-bietet-iSCSI-mit-TLS-Verschluesselung. – Zugriffsdatum: 2016-9-19

[Flocker-A 2016] : *Flocker HP*. 2016. – URL https://docs.clusterhq.com/en/latest/flocker-features/architecture.html. – Zugriffsdatum: 2016-5-27

[MARS2016 2016] : *MARS HAW*. 2016. – URL http://mars-group.mars.haw-hamburg.de/. – Zugriffsdatum: 2016-5-8

[Ahnert 2006] Ahnert, Sven: *Virtuelle Maschinen mit VMware und Microsoft. Fuer Entwicklung, Schulung, Test und Produktion.* Addison-Wesley Verlag, 2006. – ISBN 3827323746

[Mandl 2014] Mandl, Peter: *Grundkurs Betriebssysteme: Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation, Virtualisierung (German Edition)*. Springer Vieweg, 2014. – ISBN 3658062177

[Schuermann 2016] Schuermann, Tim: *Das verteilte Dateisystem GlusterFS aufsetzen*. 2016. – URL http://www.admin-magazin.de/Das-Heft/2012/02/Das-verteilte-Dateisystem-GlusterFS-aufsetzen-und-verwalten. – Zugriffsdatum: 2016-8-10

[Wolff 2015] Wolff, Eberhard: *Microservices: Grundlagen flexibler Softwarearchitekturen*. dpunkt.verlag, 2015. – ISBN 3864903130

[Zimmer 2012] Zimmer, Dennis: *VMware vSphere 5: Das umfassende Handbuch*. Galileo Press GmbH, 2012. – ISBN 383621847X

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 29. September 2016 Timo Feddersen