Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Integration von Big-Data-Systemen in Docker und Kubernetes

Marvin Elsen und Denis Wagner

Teamprojekt

Betreuer: Prof. Dr. Christoph Schmitz

Trier, 13.08.2018

Aufteilung

Autor	**Kapitel**
Denis Wagner	Kurzfassung
Denis Wagner	Einleitung und Problemstellung (S. 1 - 2)
Denis Wagner	Docker (S. 3 - 7)
Marvin Elsen	Kubernetes (S. 7 - 14)
Marvin Elsen	Aufsetzen eines orchestrierten Cassandraclusters (S. 15 - 20)
Denis Wagner	Aufsetzen von HDFS (Einleitung)
Denis Wagner	Aufsetzen von HDFS (S. 21 - 28)
Denis Wagner	Aufsetzen von YARN (Einleitung) (S. 28)
Denis Wagner	Bereitstellung eines Resourcemanagers (S. 28)
Denis Wagner	Bereitstellung und Integration von Nodemanagern (S. 29)
Marvin Elsen	Einführen eines History-Servers (S. 29 - 30)
Marvin Elsen	Verbinden von Resource- und Nodemanager (S. 30 - 31)
Marvin Elsen	Zusammenfassung (S. 32 - 34)

Tabelle 0.1. Die Aufteilung der Arbeit nach Kapitel/Seitenzahlen

Kurzfassung

In dieser Arbeit werden Big-Data-Systeme auf ihre Integrierbarkeit in die Container- und Orchestrierungssysteme Docker und Kubernetes untersucht. Große Infrastrukturen, wozu auch Big-Data-Systeme zählen, besitzen häufig einen hohen Konfigurations- und Verwaltungsaufwand. Dies bezieht sich schon alleine auf das Aufsetzen eines statischen Clusters. Diesen noch beliebig skalieren zu können erscheint nahezu utopisch, da diese Infrastrukturen häufig viele Abhängigkeiten untereinander besitzen, die kaum manuell beherrschbar sind. *Docker* und *Kubernetes* bieten jedoch einen Lösungsansatz, um diese Probleme effizient zu automatisieren. Die Untersuchung dieser Probleme ergab, dass, trotz der Kurzlebigkeit von *Pods*, dennoch ein verteilter Zustand gehalten werden kann. Mit Hilfe von *StatefulSets* können Pods einzeln identifiziert und in ihrer Anzahl flexibel skaliert werden. *Services* gruppieren netzwerktechnisch zusammenhängende Pods und erlaubt es ihnen untereinander zu kommunizieren. Es stellte sich heraus, dass der daraus entstehende Zustand sehr dynamisch über *Persistent-Volume-Claims* auf den dazugehörigen *Persistent-Volumes* gespeichert werden kann. Dieser Mechanismus für die Speicherung der Daten erfüllt somit auch die Anforderungen an einen beliebig skalierbaren Cluster.

Inhaltsverzeichnis

1

Einleitung und Problemstellung

Die meisten Programme und Systeme werden statisch auf realen Maschinen installiert und betrieben, manuell verwaltet und sind meist nur sehr umständlich erweiterbar. Dies sollte jedoch nicht so sein. In einer rasant wachsenden Gesellschaft mit sich ständig ändernden Anforderungen an Programme und Systeme ist die herkömmliche, manuelle Verwaltung kaum umsetzbar, oder gar unmöglich. Die Verwendung von virtuellen Maschinen hat in den letzten Jahren viele dieser Probleme lösen können und das Denken in der IT-Branche wurde von Grund auf verändert. Mit diesen Werkzeugen wurde eine dynamische Bereitstellung und Ersetzung von Systemen möglich, wodurch diese einfacher erweitert werden können. Jedoch kommen mit dem Verschwinden von alten Problemen neue hinzu. Da ein ganzes Betriebssystem virtualisiert werden muss, entsteht ein großer Mehraufwand (engl. *overhead*), was die Verwendung für kleinere Programme ungeeignet macht. Darüber hinaus wird auch das Grundproblem nicht gelöst: Eine virtuelle Maschine muss nach wie vor manuell aufgesetzt und installiert werden. Die beste Lösung wäre nun ein schlankes System, welches vereinzelte Programme oder auch ganze Programmstrukturen mit wenig Overhead abkapselt, sich selbst verwaltet und, wenn erforderlich, auf verschiedene Maschinen dynamisch verteilen kann. Das gesuchte System ist die Kombination aus *Docker*, einer Container-Plattform, die das schlanke Abkapseln (oder auch *containerisieren*) von Programmen ermöglicht [Inc18c], und *Kubernetes*, einem Containerorchenstrierungssystem, welches die Verwaltung von Docker-Containern über mehrere Maschinen übernimmt [Aut18f].

Big-Data benötigt eine enorme Menge an Rechenleistung, um die hohe Anzahl an Daten verarbeiten zu können [Mic18], die, wie zum Beispiel bei Hadoop, von mehreren parallel arbeitenden Maschinen stammt [Fou18a]. Solche Systeme bestehen häufig aus mehreren kleinen Programmen und Services, die die Infrastruktur des Systems ausmachen und ebenfalls verteilt betrieben werden müssen. Je mehr Systeme gleichzeitig laufen, desto höher ist jedoch die Wahrscheinlichkeit, dass eines von ihnen abstürzt oder oder einen Defekt erleidet. Bei einem System, bei dem die Einzelteile sehr stark voneinander abhängen, kann dies fatal sein und sogar zum Versagen des gesamten Clusters führen. Kubernetes nimmt sich mit Hilfe von Docker genau diesem Problem an. Docker und Kubernetes haben die Aufgabe, Programmstrukturen zu containerisieren, automatisiert auf viele Maschinen eines Clusters zu replizieren und auch zu verwalten. Sollte ein Container abstürzen oder

nicht mehr ordnungsgemäß funktionieren, so ersetzt Kubernetes ihn durch einen neuen und sorgt dafür, dass die Last der übrigen Container gleichmäßig auf alle Maschinen aufgeteilt wird [Say17].

In dieser Arbeit soll die Frage geklärt werden, ob zustandsbehaftete Big-Data-Systeme für Kubernetes und Docker geeignet sind. Der Fokus liegt dabei auf der NoSQL-Datenbank *Cassandra* und dem verteilten Berechnungsframework *Hadoop*, die sich vor allem dadurch auszeichnen, einen verteilten Zustand zwischen mehreren Maschinen zu halten. Dieser verteilte Zustand und die daraus resultierende Abhängigkeit der einzelnen Bestandteile kann das Aufsetzen oder die Erweiterung dieser Systeme erschweren. Jedoch könnte Kubernetes und Docker nicht nur für die Aufrechterhaltung und Verwaltung eines bereits bestehenden Systems, sondern auch für das schnelle und einfache Bereitstellen (engl. *deployment*) von neuen Instanzen verwendet werden. Einen Hadoop-Cluster aufzusetzen kann unter Umständen viel Zeit in Anspruch nehmen. Gerade wenn für Lern- oder Testzwecke, beispielsweise in einer Universität, häufig neue und kleine Cluster benötigt werden, ist ein hoher Zeitaufwand denkbar unangenehm. Diese beiden Probleme werden in dieser Arbeit analysiert und umgesetzt.

2

Verwendete Software

2.1 Docker

Hierbei handelt es sich um eine *Container-Plattform* (nach [MG17] auch *Container-Management-System*), die es erlaubt, Programme oder ganze Programmstrukturen abgekapselt betreiben zu können [Inc18c]. Eine Container-Platform ist eine Ansammlung von Technologien, um beispielsweise Sicherheit, Steuerung, Automation, Support und Zertifizierung für den gesamten Lebenszyklus einer containerisierten Anwendung sicherzustellen [Inc18c].

2.1.1 Docker als Lösung

Ein häufig in der Softwareentwicklung auftretendes Problem ist, dass die Entwicklungsumgebung nicht der produktiven Umgebung entspricht, was auch gerne als *works on my machine problem* (dt. Funktioniert-Auf-Meiner-Maschine-Problem) bezeichnet wird [MG17]. Zum Beispiel muss für einen Webauftritt der Webserver eine bestimmte Konfiguration aufweisen. Sollte sich im Entwicklungsprozess eine Einstellung aufgrund von erweiterten Anforderungen ändern, muss diese auch in der produktiven Umgebung geändert werden. Aber auch die restliche Umgebung muss im Einklang zur Produktivumgebung sein, wie zum Beispiel Zugriffsrechte. [MG17]. Dies kommt natürlich zustande, da jeder Entwickler seine eigene Art zu Arbeiten und persönliche Präferenzen besitzt [MG17]. Docker löst dieses Problem, indem die entwickelte Software und zugehörige Libraries, Konfigurationen und andere Abhängigkeiten (engl. *dependencies*) in einem Docker-Container abgekapselt und in die Produktivumgebung ausgeliefert wird [Kro17]. Die containerisierte Anwendung besitzt innerhalb des Containers überall die gleiche Infrastruktur. Dabei kann die Software unter Windows, macOS oder Linux entwickelt und containerisiert werden. Docker stellt sicher, dass die Container auf jedem System ordnungsgemäß funktionieren, sofern die verwendete Docker-Version kompatibel ist. Somit kann jeder Entwickler in seiner gewünschten IDE und dem gewohnten Arbeitsfluss bleiben [MG17].

Auch aus der Sicht eines Operators, der für die ordnungsgemäße Funktion eines bereits laufenden Systems zuständig ist, kann es weitere Probleme geben [oLS18]. Wenn zum Beispiel verschiedene Applikationen auf mehreren Servern verteilt sind

und nun entschieden wird, dass zwei Applikationen auf demselben Server laufen sollen. Wenn in dem Softwarestack für die erste Applikation aufgrund von Abhängigkeiten jedoch unterschiedliche Versionen vorhanden sind, als die zweite Applikation benötigt, muss häufig zu sehr umständlichen Mitteln zugegriffen werden, was auch nicht selten eine gesamte Umstrukturierung der Applikationen mit sich bringt [MG17].

2.1.2 Unterschied zu virtuellen Maschinen

Der Vorteil von Systemen auf virtuellen Maschinen (im Folgenden *VM* genannt) im Gegensatz zu fest installierten Programmen auf einer dedizierten Maschine liegt auf der Hand: VMs sind modular. Wie Abbildung 2.1 zeigt, sind auf der dedizierten Maschine drei Anwendungen installiert die sich alle denselben orangenen Softwarestack teilen müssen [MG17], wobei die VMs zwei unterschiedliche Softwarestacks (grün und orange) besitzen und somit individuellere Einstellungen vorgenommen werden können [MG17]. In der Abbildung ist auch zu erkennen, dass die Lösung mit

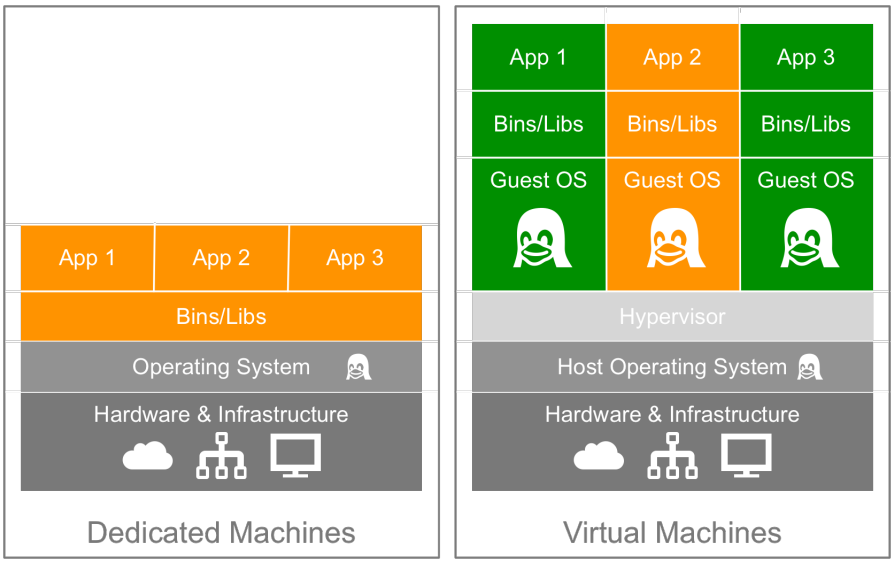

Abb. 2.1. Vergleich der Bereitstellung von Anwendungen zwischen einer dedizierten Maschine (links) und virtuellen Maschinen (rechts) mit unterschiedlichen Softwarestacks [MG17]

virtuellen Maschinen deutlich mehr Schichten aufweist. Dies stammt daher, dass eine VM zusätzlich einen Hypervisor auf dem Host-Betriebssystem und ein eigenes Gast-Betriebssystem benötigt. Dies macht sich sowohl in einer erhöhten Speicherplatznutzung als auch in einer geringeren Performance des Systems bemerkbar. Die Abbildung 2.2 zeigt die Bereitstellung der gleichen drei Anwendungen mit Hilfe von Docker, die ebenfalls zwei unterschiedliche Softwarestacks (grün und orange) verwenden. Auf dem Host-Betriebssystem liegt Docker als zusätzliche Schicht, um die Container zu verwalten, ähnlich wie der Hypervisor die VMs verwaltet [Kro17]. Der

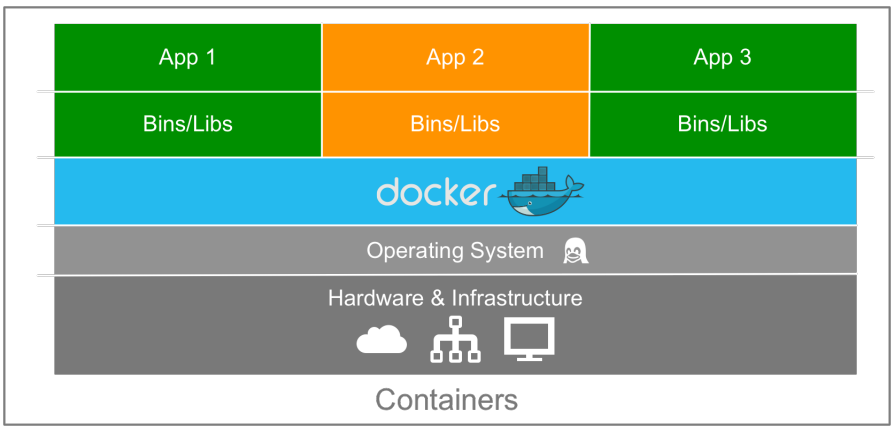

Abb. 2.2. Visualisierung der Bereitstellung von Anwendungen mit Docker [MG17]

Unterschied ist aber, dass keine Emulation von Hardware, wie es bei einem Hypervisor der Fall ist, stattfindet [Kro17]. Dies ist auch gar nicht erforderlich, da Docker den Linux-Kernel des Host-Betriebssystem verwendet. [Kro17]. Somit ist es unerheblich welches auf Linux basierende Betriebssystem der Host aufweist. Innerhalb eines Docker-Containers kann also fast jede Linux-Distribution benutzt werden, ohne das gesamte Betriebssystem samt Kernel installieren zu müssen [Kro17]. Nur der User-Space muss im Container installiert werden, dies schließt zum Beispiel den Paket-Manager mit ein [MG17]. Dies resultiert in einer schlankeren Infrastruktur mit deutlich weniger Overhead, was sowohl geringere Speicherplatznutzung als auch erhöhte Performance mit sich bringt [Kro17].

2.1.3 Aufbau eines Containers

In diesem Unterkapitel wird auf die Einzelheiten eines Docker-Containers eingegangen. Dies beinhaltet das Erarbeiten eines konkretes Beispiels, wie solche Container mit Docker erstellt werden können und weiterführende Erläuterungen, die die Vorteile und das Grundkonzept von Docker erklären sollen.

Images

Ein Docker-Container wird aus einem Image erzeugt [Kro17]. Dabei handelt es sich um eine Vorlage, die als Grundlage zur Erstellung eines Containers dient [Kro17]. Jedes Image erbt von einem Basis-Image, zum Beispiel dem offiziellen Ubuntu-Image, das als grundlegendes Betriebssystem verwendet werden kann [Kro17]. Darauf können nun weitere Konfigurationen vorgenommen und Software installiert werden. Docker erstellt Images aus einer Serie von Instruktionen, die in einer *Dockerfile* festgelegt werden [Kro17]. Eine Dockerfile kann nach [MG17] wie folgt aussehen:

```
1  FROM alpine:latest
2  LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
3  LABEL description="This example Dockerfile installs NGINX."
4  RUN apk add --update nginx && \
5      rm -rf /var/cache/apk/* && \
6      mkdir -p /tmp/nginx/
7  COPY files/nginx.conf /etc/nginx/nginx.conf
8  COPY files/default.conf /etc/nginx/conf.d/default.conf
9  ADD files/html.tar.gz /usr/share/nginx/
10 EXPOSE 80/tcp
11 ENTRYPOINT ["nginx"]
12 CMD ["-g", "daemon off;"]
```

Das Basis-Image, hier Alpine Linux, wird mit dem Schlüsselbegriff `FROM` angegeben [MG17]. Dieses Basis-Image wird, falls nicht lokal vorhanden, von Docker Hub heruntergeladen [Inc18b]. Mit `LABEL` werden Meta-Informationen angegeben, welche das zukünftige Image beschreiben sollen [Inc18b]. Durch `RUN` werden Kommandos ausgeführt, wie in diesem Beispiel der Paket-Manager, welcher den Webserver NGINX installiert [MG17]. `COPY` kopiert neue Dateien oder einen Ordner (erstes Argument) und fügt sie dem Dateisystem des Containers im angegebenen Pfad (zweites Argument) hinzu [Inc18b]. Der Schlüsselbegriff `ADD` besitzt ein ähnliches Verhalten wie `COPY`. Wenn mit `ADD` ein Archiv kopiert werden soll, wird dieses extrahiert, sofern es eines der bekannten Formate aufweist: identity, gzip, bzip2 oder xz [Inc18b]. Des Weiteren kann `ADD` auch URLs entgegen nehmen, um Dateien aus einer entfernten Quelle herunterzuladen und in dem Container ablegen zu können [Inc18b]. `EXPOSE` informiert Docker, dass der angegebene Port zur Laufzeit geöffnet werden soll [Inc18b]. Dieser Port wird nicht an das Host-System gebunden, sondern ermöglicht es nur, Programme innerhalb des Containers ansprechen zu können [MG17]. Mit `ENTRYPOINT` kann ein Container wie ein ausführbares Programm konfiguriert werden [Inc18b]. In diesem Beispiel wird `nginx` ausgeführt. Wird dieser Container nun wie folgt mit zusätzlichen Argumenten gestartet:

```
docker container run -d --name nginx dockerfile-example -v
```

dann wird `nginx` mit dem Argument `-v` aufgerufen: `nginx -v` [MG17]. Die Instruktion `CMD` kann ebenfalls Programme und Kommandos ausführen (ähnlich wie `RUN`). `CMD` wird allerdings auch dafür verwendet, vordefinierte Argumente anzugeben, die von `ENTRYPOINT` benutzt werden [Inc18b]. Wird also kein zusätzliches Argument bei dem Start des Containers angegeben, so wird in diesem Beispiel `nginx` nicht parameterlos aufgerufen, sondern nach [MG17] wie folgt:

```
nginx -g daemon off;
```

Weitere wichtige Instruktionen sind `USER`, `WORKDIR` und `ENV`. Mit Hilfe von `USER` kann der zu verwendende Benutzer gesetzt werden, mit welchem die Kommandos in dem Container ausgeführt werden [MG17]. `WORKDIR` gibt das Verzeichnis an, in dem die Instuktionen ausgeführt werden sollen [Inc18b]. Die Instruktion `ENV` setzt eine Umgebungsvariable, sowohl zum Zeitpunkt der Image-Erzeugung als auch bei dem Start des Containers [MG17].

Layer

Ein Image besteht aus einer Anzahl an Layern (deut. *Schichten*) [MG17]. Ein Layer ist eine Modifikation eines Images, welcher durch Instruktionen in der Dockerfile repräsentiert wird [Inc18a]. Layer werden sequentiell dem Basis-Image hinzugefügt [Inc18a]. Wird ein Image aktualisiert oder neu erstellt (engl. *rebuilt*), so werden die neuen Instruktionen als neuer Layer an das Image angefügt und veränderte Layer erneuert und aktualisiert [Inc18a]. Die Größe des finalen Images resultiert aus der Größe aller beinhalteten Layer [Inc18a]. Aufgrund der Eigenschaft, dass sie aufeinander aufbauen, sind Docker-Images wiederverwendbarer und leichtgewichtiger als virtuelle Maschinen [Kro17].

2.2 Kubernetes

Kubernetes ist ein, in der Programmiersprache *Go* implementiertes, quelloffenes System zur Orchestrierung von containerisierten Applikationen [Aut18f] [Fou18a]. Das Projekt wurde zunächst im Jahr 2014 von dem Unternehmen *Google* ins Leben gerufen, wird nun jedoch von der *Cloud Native Computing Foundation (CNCF)*, die sich unter anderem aus Google und der *Linux Foundation* zusammensetzt, gesteuert [Met15] [Fou17]. Die Hauptfunktionen Kubernetes sind insbesondere die automatische Bereitstellung (engl. *deployment*), horizontale Skalierung und Verwaltung verteilter Anwendungen innerhalb eines Clusters aus Maschinen (Knoten) [Aut18f]. Zu diesem Zweck werden die einzelnen Container der Applikation in logische Einheiten gruppiert, um die Organisation zu vereinfachen [Aut18f]. Zusätzlich übernimmt Kubernetes die ausbalancierte Verteilung der einzelnen Komponenten der Anwendung (engl. *load-balancing*) auf den verschiedenen Knoten des Clusters, sowie deren Überwachung (engl. *monitoring*) [Say17]. Die Knoten eines Clusters können entweder physikalische Rechner oder virtuelle Maschinen darstellen [Say17]. Sollten Container der Anwendung nicht mehr reagieren oder ausfallen, so stellt Kubernetes sicher, dass diese voll funktionsfähig neu gestartet werden [Say17]. Grundsätzlich bleibt, ohne Mehraufwand für den Kubernetesadministrator, der Zustand der Applikation dabei nicht vorhanden, aufgrund dessen sich hier insbesondere die Verwendung zustandsloser Programme anbietet [Say17].

2.2.1 Aufbau

In diesem Unterkapitel werden die Kernkomponenten eines Kubernetesclusters definiert, soweit diese für das Verständnis der Ausarbeitung von Relevanz sind.

Cluster und Nodes

Unter einem *Cluster* wird eine Ansammlung von Knoten (engl. *nodes*) verstanden [Say17]. Ein Knoten ist ein einzelner Rechner (engl. *host*) [Say17]. Dieser kann entweder eine virtuelle oder physikalische Maschine sein, welche dem Cluster Rechenleistung, Netzwerkressourcen und Speicherplatz zur Verfügung stellt [Say17].

Auf jedem dieser *Nodes* laufen die Programme `kubelet` zum Kommunizieren mit Kubernetes API-Server, um unter anderem neue Applikationen zu starten, die vom *Scheduler* eingereiht werden und `kubeproxy`, welches für das Load-Balancing zuständig ist und den eingehenden Netzwerkverkehr an die entsprechenden *Pods* auf den Knoten verteilt [Bai17].

Zu jedem Cluster gehört zudem ein *Master* [Say17]. Auf diesem Rechner laufen unter anderem der API-Server, der Scheduler und der *Replication-Controller* [Bai17] [Say17]. Grundsätzlich existiert nur ein Master pro Cluster, jedoch können zum Zweck einer erhöhten Verfügbarkeit Master auch repliziert werden [Say17]. Das Kommandozeilenwerkzeug `kubectl`, welches verwendet wird, um einen Kubernetescluster zu konfigurieren, zu überwachen und zu steuern, kommuniziert ausschließlich mit der REST-Schnittstelle des API-Servers auf dem Master [Bai17].

Pods

Pods stellen die kleinste Arbeitseinheit in einem Kubernetessystem dar [Say17]. Jeder Pod besteht aus einem oder mehreren verwandten Containern und läuft auf genau einem Knoten [Say17] [Bai17]. Diese Container teilen sich denselben IP- und Port-Adressraum und können somit über `localhost` miteinander kommunizieren [Say17]. Es ist auch möglich, den Containern eines Pods gemeinsamen Speicher des Knotens zur Verfügung zu stellen [Say17]. Hierbei ist wichtig anzumerken, dass es sich bei Pods um kurzlebige Entitäten handelt, welche willkürlich verworfen und ersetzt werden können [Say17]. Bei diesem Vorgang geht grundsätzlich jeglicher Zustand verloren [Say17].

Üblicherweise werden Pods automatisch über *Deployments* oder *StatefulSets* erstellt, wie in dem Abschnitten 2.2.1 (Abschnitt Deployments) und 2.2.1 (Abschnitt StatefulSets) genauer beschrieben wird. Die folgende Vorlage nach [Aut18e] zeigt, wie Pods durch eine YAML-Vorlage direkt definiert werden können:

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: busybox-pod
5    spec:
6      containers:
7      - name: busybox-container
8        image: busybox
9        command: ['sh', '-c', 'echo Hello Kubernetes!']
```

Das in Zeile acht angegebene Dockerimage lädt Kubernetes automatisch, sofern keine andere Repository angegeben ist, von Dockerhub herunter [Aut18c].

Labels

Labels sind Schlüssel-Wert-Paare, die dazu verwendet werden, um zusammenhängende oder gleiche Einheiten im Kubernetessystem zu gruppieren und nach ihnen zu filtern [Bai17]. Häufig werden Pods mit Labels versehen, damit diese von dynamischen Entitäten, wie Services, als Gruppe identifiziert werden können [Say17].

Zwischen den Objekten und Labels herrscht eine n-zu-n-Beziehung: Jedes Objekt kann mit einem oder mehreren Labels versehen werden und zu jedem Label können beliebig viele Objekte gehören [Say17]. Der Name des Schlüssels sowie des Wertes kann frei gewählt werden, unterliegen jedoch gewissen Zeichenrestriktionen, die im Rahmen dieser Ausarbeitung nicht weiter behandelt werden [Say17].

Deployments

Ein Deployment bzw. eine Deploymentkonfigurationsdatei weist Kubernetes an, wie Pods einer Applikation erstellt und aktualisiert werden [Aut18j]. Der Master scheduled die angegebenen Instanzen automatisch auf individuelle Nodes des Clusters und erstellt einen *Deployment-Controller* [Aut18j]. Dieser überwacht durchgehend die Anwendung und ersetzt Pods, die zum Beispiel beim Ausfall eines Knotens nicht mehr erreichbar sind [Aut18j].

Eine Deploymentkonfigurationsdatei kann wie folgt über YAML definiert werden (Vorlage nach [Aut18a]):

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5    labels:
6      app: nginx
7    spec:
8      replicas: 3
9      selector:
10       matchLabels:
11         app: nginx
12     template:
13       metadata:
14         labels:
15           app: nginx
16       spec:
17         containers:
18         - name: nginx
19           image: nginx:1.7.9
20           ports:
21           - containerPort: 80
```

In den Zeilen neun bis elf wird ein Label verwendet, um festzulegen, welche Pods zu diesem Deployment gehören und vom Controller überwacht werden müssen. In der achten Zeile wird über das Schlüsselwort `replicas` angegeben, wie viele Pods gleichzeitig zu jedem Zeitpunkt existieren sollen. Diese Zahl kann während der Laufzeit in- bzw. dekrementiert werden. Ab der Zeile zwölf wird definiert, wie die Pods des Deployments generiert werden sollen.

Services

Pods sind kurzlebige Objekte und obwohl jedem Pod eine einzigartige IP-Adresse durch Kubernetes zugewiesen wird, kann nicht sichergestellt werden, dass, sollte

dieser ausfallen, einem neuen wieder dieselbe Adresse zugewiesen wird [Aut18g]. Ein *Service* stellt eine Schnittstelle nach außen für gleichartige Pods dar, die über Labels identifiziert werden [Say17]. Services dienen dazu, eine Anwendung, die in Kubernetes ausgeführt wird, für den Endbenutzer oder für andere Entwickler zu abstrahieren und den Netzwerkzugriff auf die Applikation nach außen oder auch intern zu vereinfachen [Aut18g] [Bai17]. Je nach Typ wird einem Service zum Beispiel eine feste, statische IP-Adresse zugewiesen [Aut18g]. Netzwerkverkehr, der an einen Service gerichtet ist, wird von diesem entweder gleichmäßig verteilt (engl. *load-balanced*) oder an einen zufälligen zugehörigen Pod weitergeleitet [Bai17]. Ein Entwickler würde also nur gegen den Service und nicht gegen individuelle Pods entwickeln. Sollten neue Pods hinzukommen oder alte Pods nicht mehr erreichbar sein, so wird derjenige, der den Service benutzt, nicht darüber informiert und muss keine Veränderungen an seinem Quelltext vornehmen.

Im Rahmen dieser Ausarbeitung sind die folgenden Servicetypen von Bedeutung:

- **ClusterIP:** Der voreingestellte Servicetyp. Dieser Service ist nur innerhalb des Clusters, entweder über eine zufällig von Kubernetes zugewiesenen oder selbst konfigurierbaren IP-Adresse, erreichbar [Aut18g].
- **NodePort**: Dieser Service ist auch von außerhalb des Clusters über jeden Knoten ansprechbar. Für diesen muss ein Port konfiguriert werden, auf welchem der Service auf jedem Knoten auf eingehende Verbindung wartet [Aut18g]. Dieser kann daraufhin über `NodeIP:NodePort` angesprochen werden [Aut18g].
- **Headless**: Headless Services sind jene, denen eine `ClusterIP` von `None` vergeben wird [Aut18g]. Diese verfügen weder über Load-Balancing noch über eine zugeordnete IP-Adresse und werden insbesondere im Zusammenhang mit StatefulSets verwendet, damit Pods sich über dynamischer Namensauflösung (DNS) finden lassen [Aut18g].

Sollen Services und ihre Pods auch über einen Namen adressierbar sein, so muss ein DNS Cluster-Addon für Kubernetes installiert werden [Aut18g]. Da im Rahmen dieser Ausarbeitung das Entwicklungswerkzeug `minikube` verwendet wurde, auf dem das DNS Addon vorinstalliert ist, wird nachfolgend angenommen, dass dieses vorhanden ist.

Ein Service, der den in Kapitel 2.2.1 erstellten Nginx Server extern über Knoten zugreifbar macht, kann wie folgt über YAML definiert werden (Vorlage nach [Aut18g]):

```yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  type: NodePort
  ports:
  - name: http
    protocol: TCP
    port: 80
```

StatefulSets

StatefulSets sind Deployments, welche besondere Eigenschaften zum Verwalten von zustandsbehafteten Anwendungen bereitstellen [Aut18h]. So wird jedem Pod ein fester einzigartiger Hostname sowie eine aufsteigende Ordnungszahl zugewiesen [Say17]. Auch kann einem Pod persistenter Speicher über *Persistent Volumes* zugewiesen werden [Say17]. Wie bei einem Deployment enthalten die verschiedenen Pods dieselben Container, sind jedoch nicht untereinander austauschbar [Aut18h]. Jeder Pod behält seine eindeutige Kennung über das Rescheduling hinaus bei [Aut18h]. Damit die Zuweisung von eindeutigen Hostnamen stattfinden kann und die Pods miteinander kommunizieren können, wird ein Headless-Service für jedes StatefulSet benötigt, wie im Kapitel 2.2.1 angesprochen. Dieser kann wie folgt über YAML definiert werden (Vorlage nach [Aut18h]):

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
```

Hierbei ist insbesondere darauf zu achten, dass das Attribut `clusterIP` auf `None` gesetzt und dass dem Service, wie in Zeile vier zu sehen, ein Name zugewiesen wird, auf den später das StatefulSet referenzieren kann.

Eine StatefulSet kann wie folgt über YAML definiert werden (Vorlage nach [Aut18h]):

```
1  apiVersion: apps/v1
2  kind: StatefulSet
3  metadata:
4    name: web
5  spec:
6    selector:
7      matchLabels:
8        app: nginx
9    serviceName: "nginx-service"
10   replicas: 3
11   template:
12     metadata:
13       labels:
14         app: nginx
15     spec:
16       containers:
17       - name: nginx
18         image: nginx:1.7.9
19         ports:
20         - containerPort: 80
21           name: web
```

Die einzigen Unterschiede zu dem, im Kapitel 2.2.1 erstellten, Deployment, sind die Art des Objektes in Zeile zwei, sowie der in Zeile neun angegebene Name des Headless-Services.

Jedem der drei, von dem StatefulSet erzeugten, Pods wird eine Ordinalzahl von 0 bis 2 zugewiesen [Aut18h]. Der Hostname setzt sich aus `StatefulSetName-Ordinalzahl` zusammen, also zum Beispiel `web-0` und `web-1` [Aut18h]. Mithilfe des Headless-Service ist es ebenfalls möglich, die einzelnen Pods über folgenden FQDN anzusprechen: `HostName.ServiceName.NamespaceName.svc.cluster.local`, beziehungsweise konkret: `web-0.nginx-service.default.svc.cluster.local` [Aut18h].

Volumes

Die Dateien eines Containers sind flüchtig und obwohl Pods bei einem Absturz automatisch erneut gestartet werden, gehen ihre Daten verloren [Aut18k]. Ein weiteres Problem ist, dass die Daten nur innerhalb eines Containers zugreifbar und nach außen, sprich für andere Container desselben Pods, nicht sichtbar sind [Aut18k]. Sollen nun Dateien zwischen mehreren Containern eines Pod geteilt werden, finden Kubernetes-Volumes Verwendung [Aut18k] [Say17]. Diese funktionieren grundsätzlich ähnlich, wie die Volumes Dockers, haben jedoch keine direkte Verbindung mit diesen [Say17].

Kubernetes bietet eine Vielfalt an Typen von Volumes an, von denen nur `hostPath`, sowie `Persistent Volumes` im Rahmen dieser Ausarbeitung von Bedeutung sind. Ein Volume vom Typ `hostPath` hängt eine Datei oder ein Verzeichnis des Dateisystems des Hostknotens ein, auf dem der Pod läuft [Aut18k]. Ein Vorteil vom Typ `hostPath` ist, dass die Daten auch weiterhin persistent auf dem Knoten vorhanden bleiben, sollte der dazugehörige Pod abstürzen. Die Nachteile sind, dass, sollte ein Knoten den Cluster verlassen, die Daten dennoch verloren gehen und dass nach dem Absturz eines Pods oder Containers nicht mehr eindeutig

festgestellt werden kann, welche Daten welcher Entität zugeordnet waren [Say17]. Weiterhin existiert das Problem, dass, sollte ein Pod nach einem Neustart auf einem anderen Host laufen, er seine geschriebenen Daten nicht mehr finden kann, da diese sich auf einer anderen Maschine befinden [Say17].

Eine solches Volume kann wie folgt über YAML definiert werden (Vorlage nach [Aut18k]):

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: test-pd
5  spec:
6    containers:
7     - name: nginx
8       image: nginx:1.7.9
9       volumeMounts:
10      - name: web-volume
11        mountPath: "/var/www"
12   volumes:
13   - name: web-volume
14     hostPath:
15       path: "/data"
16       type: DirectoryOrCreate
```

In den Zeilen neun bis elf wird definiert, welches Volume in welchem Verzeichnis innerhalb des Containers eingehangen werden soll. Die Zeilen zwölf bis sechzehn beschreiben das Volume selbst und welcher Ordner auf dem Knoten verwendet werden soll.

Persistent-Volumes sind Volumes, die nicht durch eine Definition eines Pods erzeugt werden, sondern welche durch einen Administrator des Kubernetesclusters bereitgestellt und durch Persistent-Volume-Claims von Pods angefordert werden können [Bai17]. Persistent-Volume-Claims geben an, welche Voraussetzungen und Anforderungen ein Pod an Speicher stellt [Bai17]. Existiert ein Persistent Volume, welches diesen entspricht, so wird es dem Pod zugewiesen [Bai17]. In Verbindung mit einem StatefulSet, welches jedem Pod eine eindeutige ID zuweist, wird das Persistent Volume an einen Pod gebunden, sodass dieser auch, wenn er rescheduled wird, weiterhin seinen Datenstand behält, unabhängig davon, auf welchem Knoten er sich befindet.

2.2.2 Beispiel Cluster

Abschließend soll an der Grafik 2.3 visuell dargestellt werden, wie die Topologie eines Beispielclusters aussehen könnte. Der Cluster setzt sich aus vier Maschinen zusammen: drei Knoten und einem Master. Auf diesem befinden sich die drei Anwendungen A, B und C, zu denen jeweils ein Deployment existiert. Die Anwendung A hat einen Replikationsfaktor von Zwei und jeder der Pods besteht aus jeweils zwei Containern. Das Deployment B fordert mindestens drei Pods, in welchen jeweils ein Container und Volume erzeugt werden. Die letzte Applikation C verfügt über genau ein Replikat mit einem Container. Für A und C existiert jeweils ein

ClusterIP-Service, welcher die Pods anhand eines Labels mithilfe eines Labelselectors selektiert. In der Abbildung sind ebenfalls die verschiedenen IP-Adressen der Objekte zu sehen, um ein grobes Bild davon zu erzeugen, wie die Netzwerkinfrastruktur Kubernetes aufgebaut ist.

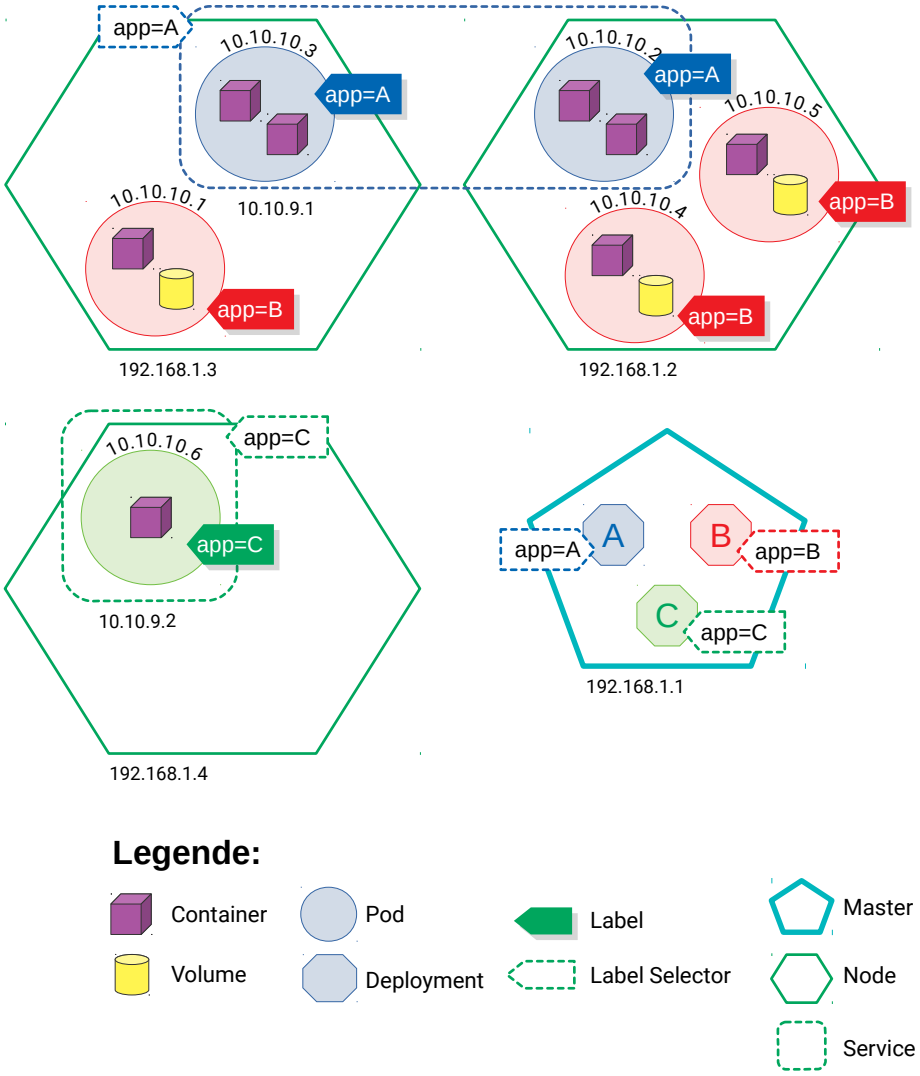

Abb. 2.3. Topologie eines möglichen Kubernetesclusters, bestehend aus drei Knoten, einem Master, drei Deployments, zwei Services und sechs Pods. Angelehnt an [Aut18i].

3

Problemanalyse und Ergebnisse

In diesem Kapitel wird auf unsere Projektarbeit und ihre Ergebnisse eingegangen. Es werden einzelne Problemstellungen der Big-Data-Technologien im Zusammenhang mit Docker und Kubernetes konkretisiert und eine mögliche Lösung erarbeitet. Des Weiteren werden Einzelheiten von Cassandra und Hadoop erläutert, welche die Problemstellungen betreffen.

3.1 Aufsetzen eines orchestrierten Cassandraclusters

Apache Cassandra ist ein freies, quell-offenes, verteiltes *NoSQL* Datenbankmanagementsystem [Dat18]. Verglichen mit relationalen Systemen, verwendet Cassandra sogenannte *Wide-Columns*, um Datensätze zu speichern und zu verwalten [Dat18].

3.1.1 Cassandra im Docker Container

Installation

Die erste Fragestellung ist, wie Cassandra in einem Dockercontainer installiert werden kann. Grundsätzlich wäre der Ablauf, das Basisimage einer Distribution zu verwenden, um über den spezifischen Paketmanager die Anwendung zu installieren. Aufgrund dessen, dass nur wenige Distributionen Cassandra über ihre eigenen Repositories ausliefern, ist dieser Lösungsansatz nicht ohne Umwege möglich.

Die offizielle Webseite Cassandras stellt Paketrepositories und Schlüssel für Distributionen, die auf Debian basieren, bereit, um das Datenbanksystem über den Paketmanager `apt-get` installieren zu können [Fou18b]. Als Basisimage für den Container kann also das offizielle Ubuntu-Dockerimage, welches bereits viele Abhängigkeiten Cassandras abdeckt und das Programm `apt-get` zur Verfügung stellt, verwendet werden.

```
1 FROM ubuntu:16.04
2
3 RUN apt-get update && apt-get install -y \
4 curl \
5 && echo "deb http://www.apache.org/dist/cassandra/debian 36x main" \
6 | tee -a /etc/apt/sources.list.d/cassandra.sources.list \
7 && curl https://www.apache.org/dist/cassandra/KEYS | apt-key add - \
8 && apt-get install -y cassandra \
```

Zunächst wird die offizielle Cassandra-Repository in den Zeilen fünf bis sechs eingebunden und in Zeile acht wird Cassandra installiert.

Netzwerkkommunikation

Cassandra ist ein Programm, welches über das Netzwerk mit anderen Instanzen kommuniziert. Dazu verwendet das Datenbanksystem die in Tabelle 3.1 aufgeführten TCP Ports. Im Rahmen dieser Arbeit sind nur die Ports 9042 für Anfragen an die Datenbank und 7000 für die interne Clusterkommunikation *(Gossip)* von Interesse.

Port	**Beschreibung**
7000	Cluster Communication
7001	Cluster Communication (SSL)
7199	JMX (Java Management Extension)
9042	CQL (native protocol)

Tabelle 3.1. Die von Cassandra verwendeten Ports

Nun ist die Anwendung in dem Container, kann jedoch noch nicht von der Außenwelt über das Netzwerk erreicht werden. Die folgende Zeile im Dockerfile legt die Ports nach außen offen:

```
1 # Cassandra Installation hier ...
2
3 EXPOSE 7000 7001 7199 9042
```

Cassandras Voreinstellungen sind so konfiguriert, dass die Ports an die lokale IP-Adresse `127.0.0.1` *(localhost)* gebunden werden. Dies führt zu dem Problem, dass die Instanz weiterhin von außen nicht ansprechbar ist. Eine Lösung stellt dar, die Einträge `rpc_address` und `listen_address` in der Konfigurationsdatei `/etc/cassandra/cassandra.yaml` auf die externe IP-Adresse des Containers zu setzen. Die folgenden Zeilen Shellskript der Datei `start.sh` ersetzen mithilfe regulärer Ausdrücke die entsprechenden Stellen in der Konfigurationsdatei und starten Cassandra im Container:

```
#!/usr/bin/env bash

IP=$(hostname --ip-address)

sed -i -e "s/^rpc_address.*/rpc_address: $IP/" /etc/cassandra/cassandra.yaml
sed -i -e "s/^listen_address.*/listen_address: $IP/" \
/etc/cassandra/cassandra.yaml

exec cassandra -Rf
```

Damit die Skriptdatei ordnungsgemäß beim Start des Containers ausgeführt wird, müssen die folgenden Zeilen zum Dockerfile hinzugefügt werden:

```
# Installation und Portfreigabe hier ...

USER root
ADD start.sh /usr/bin/start_cas.sh
ENTRYPOINT sh /usr/bin/start_cas.sh
```

Clusterbildung

Mit den bereits vorgenommenen Einstellungen besteht das Problem, dass jeder Cassandraknoten sich in seinem eigenen Cassandracluster befindet. Dies beruht darauf, dass der Seedknoten nicht explizit angegeben wird und implizit jeder Knoten sich selbst als Seed ansieht. Dies resultiert darin, dass jeder Container einen neuen Cluster nur für sich erstellt. Unter dem Seed versteht man einen Knoten eines bereits existierenden Clusters, der als erster Ansprechpartner für einen neuen Knoten dient, um ihm Informationen über den Cluster zu senden.

Über den folgenden Zusatz kann die Skriptdatei `start.sh` erweitert werden, um entweder das erste übergebene Argument oder die eigene IP-Adresse für den Seedknoten zu verwenden:

```
# Austauschen der rpc_address und listen_address hier ...

SEED=$IP
if [ -n "$1" ]
then
  SEED=$1
fi
sed -i -e "s/- seeds: \"127\.0\.0\.1\"/- seeds: \"$SEED\"/" \
/etc/cassandra/cassandra.yaml

exec cassandra -Rf
```

Angenommen auf einer Maschine mit der IP-Adresse 192.168.2.2 würde bereits ein Cassandracontainer laufen, dann könnte mit dem folgenden Kommando ein weiterer Knoten zu diesem Cluster hinzugefügt werden:

```
docker run -d cassandra 192.168.2.2
```

Der Cassandracontainer ist nun soweit einsatzbereit, besitzt jedoch noch das folgende Problem: Die Daten der einzelnen Cassandrainstanzen gehen bei einem Absturz oder Neustart der Container verloren. Das Dockerfile könnte jedoch so

angepasst werden, dass dem Container über Dockervolumes persistenter Festplattenspeicher des Hostsystems zugewiesen würde. Dies wird im Rahmen dieser Ausarbeitungen allerdings nicht ferner spezifiziert, da in einem späteren Kapitel gezeigt wird, wie Kubernetes Volumefunktionen für genau diesen Fall verwendet werden können.

3.1.2 Cassandra auf Kubernetes

Dieses Unterkapitel befasst sich mit der Fragestellung, wie ein containerisiertes Cassandra nun auf Kubernetes ausgeführt werden kann.

Bei dem Datenbanksystem handelt es sich um eine zustandsbehaftete Anwendung, welche Daten auf ein persistentes Speichermedium schreibt. Im Falle Cassandras ist dies in der Regel die Festplatte. Neu erstellten Pods der Applikation muss es möglich sein, einen weiteren Pod, der als Seedknoten dient, ausfindig zu machen. Für genau diesen Zweck bieten sich hier die im Kapitel 2.2.1 besprochenen StatefulSets an, die jedem Pod einen einzigartigen Hostnamen zuweisen.

Netzwerkkommunikation

Im Kapitel 3.1.1 wurde besprochen, wie ein Container netzwerkfähig gemacht werden kann. Durch Kubernetes kommt eine zusätzliche Abstraktionsschicht zum Netzwerk und somit auch ein weiteres Problem hinzu. Die Frage ist nun, wie die Container, welche sich jeweils in verschiedenen Pods befinden, miteinander kommunizieren können. Da Pods, die sich im selben Cluster befinden, bereits ohne zusätzliche Services Nachrichten austauschen können, könnte angenommen werden, dass ein einfaches Deployment ausreichend wäre. In Anbetracht dessen, dass die Cassandraknoten jedoch einen ausgewählten Seed benötigen, muss die IP-Adresse eines Pods statisch vergeben werden können.

Um nun auch den Cassandracluster von außen ansprechen zu können, wird ein Service benötigt, der von außen sichtbar ist. Zu diesem Zwecke wird ein Service vom Typ `NodeIP` verwendet, der auf jedem Knoten des Kubernetesclusters eingehende Verbindungen abhört.

```
1 # Service Metadaten deklaration hier ...
2 # ...
3 spec:
4   type: NodePort
5   ports:
6   - port: 9042
7   selector:
8     app: cassandra
```

Dieser externe Service bindet einen zufälligen freien Port der Hostmaschine an den Port 9042 der Pods. Eine Anfrage, die an einen Knoten des Kubernetesclusters an diesen zufälligen Port gestellt wird, wird an eine der Cassandrainstanzen weitergeleitet. Dabei wird nicht sichergestellt, dass es sich um einen Pod handelt, der auf der angesprochenen Maschine läuft. Angenommen ein Kubernetescluster besteht

aus zwei Knoten mit den IP-Adressen 192.168.2.2 und 192.168.2.3 und der Service bindet sich an den Port 30242. Wird nun eine CQL Anfrage an 192.168.2.2:30242 gestellt, könnte diese zum Beispiel an den Pod auf der Maschine 192.168.2.3 weitergeleitet werden.

Bereitstellen der Container

Dieses Unterkapitel beschäftigt sich mit der Frage, wie nun die Cassandracontainer auf Kubernetes verteilt werden können. Wie in dem Kapitel 2.2.1 besprochen, bieten sich StatefulSets an, um zustandsbehaftete Anwendungen bereitzustellen. Dem Erstellen des StatefulSets für Cassandra bedarf es nur weniger außerordentlicher Features und orientiert sich stark an dem im Kapitel 2.2.1 eingeführten Gerüst. Dafür werden zunächst die bekannten Schlüssel-Wert-Paare, wie der Name des Headless-Services (dessen Definition hier nicht explizit aufgeführt wird), die Labels, der Replikationsfaktor, die Containerkonfiguration etc. festgelegt. Wichtig zu beachten ist hierbei, dass die entsprechenden Ports Cassandras aufgeführt werden, damit diese über den Pod zugreifbar sind. Da hier mehrere Ports angegeben sind, muss diesen jeweils auch ein Name zugewiesen werden. Der Einfachheit halber wurde hier das offizielle, statt dem im Kapitel 3.1.1 erstellten Dockerimage verwendet.

```
1  # Aufbau wie im StatefulSet Geruest ...
2  # ...
3      spec:
4        containers:
5        - name: cassandra
6          image: cassandra
7          ports:
8          - containerPort: 7000
9            name: intra-node
10         - containerPort: 7001
11           name: tls-intra-node
12         - containerPort: 7199
13           name: jmx
14         - containerPort: 9042
15           name: cql
```

Über das Schlüsselwort `resources` kann über Kubernetes festgelegt werden, welche und wie viele Ressourcen ein Container initial anfragt und maximal verwenden darf. Diese folgenden Einstellungen wurden verwendet, um die leistungsschwachen Rechner der Testumgebung nicht zu stark auszulasten.

```
1  # Aufbau wie im StatefulSet Geruest ...
2  # ...
3            resources:
4              limits:
5                cpu: "500m"
6                memory: 1Gi
7              requests:
8                cpu: "500m"
9                memory: 1Gi
```

Aufgrund dessen, dass hier von dem offiziellen Dockerimage Gebrauch gemacht wird, muss hier zusätzlich eine Umgebungsvariable für den Container definiert werden, welche den Hostnamen des Seedknotens angibt. Hier wird der Vorteil eines StatefulSets sichtbar: Der erste Pod, den Kubernetes startet, besitzt immer den Hostnamen `cassandra-0.cassandra.default.svc.cluster.local`. Dies wird durch die eindeutige Ordinalzahl, die von dem StatefulSet verwaltet wird, sichergestellt. Somit wird garantiert, dass der Seedknoten existiert und sich die Cassandrainstanzen im selben Cluster befinden, sofern dieser zum dem gegebenen Zeitpunkt temporär nicht unerreichbar ist. Der Umgebungsvariablenname `CASSANDRA_SEEDS` wird durch das Dockerimage vorgegeben.

```
1  # Aufbau wie im StatefulSet Geruest ...
2  # ...
3            env:
4            - name: CASSANDRA_SEEDS
5              value: "cassandra-0.cassandra.default.svc.cluster.local"
```

Aufsetzen des Clusters

Um nun einen neuen Cassandracluster zu starten, muss lediglich der Befehl

```
kubectl create -f cassandra.yaml
```

über die Kommandozeile aufgerufen werden. Sollen Knoten dem Cluster hinzugefügt oder entfernt werden, so wird der Befehl

```
kubectl scale -f cassandra.yaml ANZAHL_PODS
```

verwendet, wobei `ANZAHL_PODS` angibt, wie viele Pods nach Ausführen des Programms bestehen sollen. Anhand dieses Anwendungsbeispiels werden die Vorteile Dockers und Kubernetes direkt ersichtlich: Der initiale hohe Konfigurationsaufwand wird damit belohnt, dass das Erstellen, Vergrößern und Verkleinern eines Clusters umso einfacher ist.

3.2 Aufsetzen eines orchestrierten Hadoopclusters

Bei *Apache Hadoop* handelt es sich um ein verteiltes Berechnungsframework, welches sich aus dem verteilten Dateisystem *HDFS* und dem Ressourcen-Verwaltungssystem *YARN* zusammensetzt [Fou18c]. Gemeinsam bilden sie das Gerüst für *MapReduce*, ein Programmierparadigma, welches die parallele Verarbeitung großer Datenmengen erlaubt [Fou18c][IBM17]. Ein Programm (auch *Job* genannt), nach dem MapReduce Schema, spaltet sich in einen *Map-* (dt. *Abbilden*) und einen *Reduce-* (dt. *Reduzieren*) Teil auf [IBM17]. Der Map-Teil konvertiert eine Menge von Daten in eine Menge aus Tupeln (Schlüssel-Wert-Paare) [IBM17]. Diese Tupel-Menge gilt als Eingabe des Reduce-Teils [IBM17]. Dieser reduziert die Tupel-Menge nun zu einer kleineren Tupel-Menge [IBM17].

MapReduce ist ein stark skalierbarer Mechanismus, wodurch es sich anbietet, viele Maschinen für eine Aufgabe einzusetzen [IBM17]. Die Idee ist nun, genau diese Skalierbarkeit mit Kubernetes zu vereinfachen und noch robuster gegen Ausfälle zu machen.

Es soll an dieser Stelle angemerkt werden, dass alle verwendeten Docker-Images auf den Dateien des Docker-Hub-Benutzers *uHopper* basieren.

3.2.1 Aufsetzen von HDFS

Das HDFS (*Hadoop Distributed File System* [Fou18c]) ist ein verteiltes Dateisystem, welches entwickelt wurde, um handelsübliche und „billige“ Server für große Cluster verwenden zu können [IBM18]. Das HDFS weist außerdem eine hohe Fehlertoleranz und einen hohen Datendurchsatz auf, was vor allem für Big-Data besonders von Vorteil ist [Fou13][Fou18c]. Das Dateisystem besitzt eine Master-Slave-Architektur, welche sich in einen Namenode (Master) und viele Datanodes (Slaves) unterteilt.

Bereitstellung eines Namenodes

Der Namenode ist die Master-Instanz für das HDFS und verwaltet Namensräume des Dateisystems und reguliert Datenzugriffe von Clients [Fou13].

Ein HDFS-Cluster benötigt nur einen Namenode. Um dies in Kubernetes zu realisieren, muss sichergestellt werden, dass auch nur ein Namenode-Container bereitgestellt wird. Um dies zu erreichen, wurden die folgenden drei Herangehensweisen in Betracht gezogen:

- StatefulSet
- Deployment
- Einzelner Pod

Ein StatefulSet und ein Deployment haben die Eigenschaft den Replikationsfaktor einzustellen und somit die Anzahl der zu startenden Namenode-Pods zu regulieren. Der Nachteil ist, dass im Nachhinein der Replikationsfaktor einfach geändert werden kann. Dies kann im laufenden Betrieb unter Umständen leicht zu Fehlern

führen. Die dritte Möglichkeit einzelne Pods zu definieren, die einen Namenode-Container enthalten, besitzt diesen Nachteil nicht, da Pods selbst nicht skalierbar sind. Dadurch kann sichergestellt werden, dass keine überschüssigen Namenode-Instanzen existieren. Der entscheidende Nachteil wiederum ist, dass abgestürzte Einzel-Pods nicht wieder neu gestartet werden, was die Verfügbarkeit in einer produktiven Umgebung belasten könnte. Dieses Problem haben StatefulSets und Deployments jedoch nicht, da sie ihre Pods versuchen am Laufen zu halten und Fehler der Container zu erkennen. Im Gegensatz zu Deployments nummerieren StatefulSets ihre Pods, was den Umgang mit diesem Typ der Bereitstellung vereinfacht. Aus diesen Gründen bietet sich das StatefulSet am ehesten an. Der Replikationsfaktor kann durch `spec.replicas` auf `1` gesetzt werden, um den oben beschriebenen Effekt zu erreichen.

Um das Namenode-Image verwenden zu können, muss eine Umgebungsvariable (engl. *environment variable*) namens `CLUSTER_NAME` und dem Wert `test-cluster` angegeben werden. Diese dient zur Benennung des Clusters nach der initialen Formatierung des Dateisystems. Dafür kann die Eigenschaft `spec.template.spec.containers.env` mit einer Liste aus `name` und `value` gesetzt werden. Eine alternative Vorgehensweise wäre, die Umgebungsvariable direkt in der Dockerfile oder in einem Skript im Container anzugeben. Jedoch müsste bei jeder Änderung das Image neu erzeugt und auf Docker-Hub hochgeladen werden, was aus Sicht der Wartbarkeit und Konfigurierbarkeit nicht praktikabel ist. Die Konfigurationsdatei könnte so aussehen:

```
1  apiVersion: apps/v1
2  kind: StatefulSet
3  metadata:
4    name: hdfs-namenode
5      labels:
6        app: hdfs-namenode
7  spec:
8    replicas: 1
9    selector:
10     matchLabels:
11       app: hdfs-namenode
12   template:
13     metadata:
14       labels:
15         app: hdfs-namenode
16     spec:
17       containers:
18       - name: hdfs-namenode
19         image: fronepi/hdfs-namenode
20         imagePullPolicy: Always
21         env:
22         - name: CLUSTER_NAME
23           value: "test-cluster"
```

Bereitstellung von Datanodes

Datanodes verwalten den Speicher der Knoten auf denen sie laufen [Fou13] und bilden somit den Hauptbestandteil des HDFS-Clusters für die Datenspeicherung.

Im Gegensatz zu Namenodes kann es beliebig viele Datanodes innerhalb eines HDFS-Clusters geben. Somit wird hier ein leicht skalierbarer Typ von Bereitstellung benötigt. Dafür wurden auch hier StatefulSets und Deployments als mögliche Lösung in Betracht gezogen. Beide Bereitstellungstypen bieten Skalierung an. Und da einzelne Datanodes nicht angesprochen werden müssen, eigenen sich sowohl Deployments als auch StatefulSets für diese Aufgabe. Aus Gründen der Einheitlichkeit wurde auch hierfür ein StatefulSet verwendet. Der Replikationsfaktor kann je nach Anzahl der benötigten Datanode-Instanzen variiert werden.

Verbinden von Namenode und Datanodes

Damit das HDFS als Ganzes funktioniert, müssen Namenode und Datanodes miteinander kommunizieren können. Datanodes müssen den Namenode des HDFS-Clusters kennen und ansprechen können. Da die IP-Adresse eines Pods sich jedes Mal ändern kann, bietet es sich an, den Namenode anhand des Hostnamens, der durch das StatefulSet eindeutig vergeben wurde, anzusprechen (dies wäre bei einem Deployment nicht möglich gewesen). Dieser kann über die Umgebungsvariable `CORE_CONF_fs_defaultFS` im Datanode angegeben werden:

```
1 # Datanode-Konfiguration wie StatefulSet des Namenodes
2 # ...
3         env:
4         - name: CORE_CONF_fs_defaultFS
5           value: "hdfs://hdfs-namenode-0.hdfs-namenode-service
6                   .default.svc.cluster.local:8020"
```

Services bieten die Funktion der DNS-Namensauflösung. Somit eignen sie sich, wenn Pods über ihre Hostnamen und nicht ihre IP-Adressen angesprochen werden sollen. Damit nun Datanodes Nachrichten an den Namenode senden können, muss sich dieser innerhalb eines Services befinden. In diesem Fall soll der Service allerdings nicht direkt angesprochen werden, wodurch er keine eigene IP-Adresse benötigt. Ein *Headless-Service* aus Kapitel 2.2.1 erfüllt eben diese Anforderungen. Dieser Service ist von dem Typ *ClusterIP* und wird headless, wenn die Eigenschaft `spec.clusterIP` auf `None` gesetzt, also keine IP-Adresse zugewiesen wird. Damit die Nachrichten auch empfangen werden, müssen zusätzlich folgende Ports geöffnet werden:

Port	Beschreibung
50070	Namenode Web-Interface
8020	Dateisystem-Metadaten-Operationen

Tabelle 3.2. Namenode-Ports und ihre Beschreibung

In einem Service werden diese durch das Setzen der Portnummer bei `spec.ports.port` und des Portnamens bei `spec.ports.name` spezifiziert. Die entsprechende Konfigurationsdatei des Services könnte so aussehen:

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: hdfs-namenode-service
5      labels:
6        app: hdfs-namenode
7    spec:
8      ports:
9      - port: 50070
10       name: web
11     - port: 8020
12       name: fs
13     clusterIP: None
14     selector:
15       app: hdfs-namenode
```

Zwar sind nun die Ports im Service offen, jedoch nicht in dem Pod des Namenodes. StatefulSets besitzen die Eigenschaft, unter `spec.template.spec.containers.ports` Ports für dessen Pods freizugeben:

```
1  # Bisherige Namenode-Konfiguration
2  # ...
3          ports:
4          - name: web
5            containerPort: 50070
6          - name: fs
7            containerPort: 8020
```

Zu diesem Zeitpunkt kann der Namenode jedoch keine Antwort zurück an die Datanodes senden, da die Namensauflösung nur von Datanodes zu Namenode funktioniert. Um dies auch für die andere Richtung zu ermöglichen, müssen die Datanodes ebenfalls in einem Headless-Service gruppiert und die passenden Ports geöffnet werden. Die freizulegenden Ports lauten wie folgt:

Port	**Beschreibung**
50075	DataNode Web-Interface
50010	Datentransfer
50020	Metadaten-Operationen

Tabelle 3.3. Datanode-Ports und ihre Beschreibung

Bis auf die angegebenen Ports und ihre Namen sind die Konfigurationsdateien des Datanodes identisch zu denen des Namenodes.

Nun sind sowohl Namenode als auch Datanodes in ihren eigenen Services und können mit Hilfe von Hostname und DNS-Namensauflösung innerhalb des Kubernetes-Clusters miteinander kommunizieren. Das HDFS ist nun betriebsbereit, jedoch kann noch nicht von außerhalb des Clusters darauf zugegriffen werden.

Netzwerkzugriff von außen

Um nun mit dem HDFS-Cluster arbeiten zu können, sollte der Zugriff von außerhalb des Kubernetes-Clusters möglich sein. Um dies zu erreichen, wurden folgende Herangehensweisen in Betracht bezogen:

- Aufschalten per SSH
- Öffnen von Ports im Kubernetes-Cluster

Im Gegensatz zu den obigen Abschnitten schließen sich die Möglichkeiten hier nicht aus, sondern werden für unterschiedliche Aufgaben verwendet.

Kubernetes-Knoten sind reale oder virtuelle Maschinen, die eine IP-Adresse besitzen. Sofern diese einen SSH-Server installiert haben, kann sich per SSH aufgeschaltet werden, um Zugriff auf den Knoten zu erhalten. Über das auf dem Server installierte `kubectl` kann nun auf den Kubernetes-Cluster zugegriffen und Operationen darauf ausgeführt werden. Diese Methode wird dazu verwendet, um beispielsweise Dateien auf das HDFS zu schreiben oder zu manipulieren. Der Befehl, um alle Dateien im Home-Verzeichnis des HDFS auf dem ersten Datanode anzeigen zu lassen, könnte so aussehen:

```
kubectl exec datanode-0 -c hadoop fs -ls
```

Damit wird der Befehlt `hadoop fs -ls` direkt auf dem Pod `datanode-0` ausgeführt.

Namenode und Datanode besitzen ein Web-Interface, welches Informationen über den gesamten Cluster bereitstellt. Es kann jedoch nicht von außerhalb des Kubernetes-Cluster erreicht werden. Dafür muss dieser selbst den entsprechenden Port für einen angegebenen Pod freigeben. So kann das Web-Interface direkt über die IP-Adresse des Kubernetes-Clusters aufgerufen werden. Ein Service von dem Typ `NodePort` besitzt die Fähigkeit, Netzwerkanfragen von außerhalb des Clusters an bestimmte Pods zu leiten. Über die Eigenschaft `spec.ports.port` kann ein Port spezifiziert werden und über `spec.selector.app` der Label-Selector, um die Pods auszuwählen, für die sie geöffnet werden sollen:

```
1 # Service-Konfiguration wie ClusterIP-Service des Namenodes
2 # ...
3 spec:
4   type: NodePort
5   ports:
6   - port: 50070
7   selector:
8     app: hdfs-namenode
```

Dauerhafte Speicherung der Daten

Das Ziel von HDFS ist nicht nur eine große Menge von Daten kurzfristig zu halten, sondern auch dauerhaft zu speichern. Docker und Kubernetes bieten ohne weitere Konfiguration keinen solchen Mechanismus. Sollte ein Pod also ausfallen, würden somit alle Daten verloren gehen. Jedoch existieren unterschiedliche Konfigurationsmöglichkeiten, um diesen Mechanismus zu erhalten.

Eine der Anforderungen an ein Speicherungsmechanismus für HDFS ist eine variable Anzahl an Volumes, die sich an die Menge der Datanode-Instanzen anpasst. Der Grund dafür liegt darin, dass ein Datanode die Daten immer in den gleichen Ordner schreibt und diesen durch einen Lock in Form einer Datei ablegt, der es anderen Datanodes, die eventuell auf das gleiche Volume schreiben wollen, den Zugriff verbietet. Des Weiteren sollten Pods die abgestürzt sind, wieder ihre ursprünglichen Volumes bekommen, da es sonst zu Inkonsistenzen des Datensatzes führen könnte. Es wurden die folgenden Mechanismen in Betracht gezogen und untersucht:

- `hostPath`
- Persistent-Volume und Persistent-Volume-Claim

Ein Volume von dem Typ `hostPath` aus Kapitel 2.2.1 beschreibt eine Datei oder einen Ordner auf dem Dateisystem des Knotens, die/der in den Pod eingehängt (engl. *mounted*) wird [Aut18k]. Das bedeutet, dass das effektive Volume das reale Speichermedium des Knotens ist. Das Problem bei diesem Mechanismus tritt auf, wenn mehrere Datanode-Pods auf dem selben Knoten laufen und alle auf den gleichen eingehangenen Ordner zugreifen, was die erste Anforderung verletzen würde. Ein erster Lösungsansatz könnte sein, dass jeder generierte Pod einen exklusiven `hostPath` zugewiesen bekommt. Um dies umzusetzen kann jedoch kein StatefulSet oder Deployment verwendet werden, da der `hostPath` in deren Konfigurationsdatei für alle daraus generierten Pods fest spezifiziert ist. Die einzige Möglichkeit wäre das manuelle Erstellen von Einzel-Pods mit individuellen Konfigurationsdateien, was aus den Gründen, die in Abschnitt 3.2.1 vorgestellt wurden, nicht praktikabel ist.

Ein *Persistent-Volume* (im Folgenden *PV* genannt) ist ein Stück Speicher und eine Ressource des Clusters, ähnlich wie ein Knoten [Aut18d]. Im Zusammenspiel dazu ist ein *Persistent-Volume-Claim* (im Folgenden *PVC* genannt) die Einforderung eines PVs [Aut18d]. Wie in Kapitel 2.2.1 beschrieben, wird ein PV durch ein PVC gebunden, somit bleibt ein PVC auch nach dem Absturz oder Löschen eines Pods bei StatefulSets erhalten und kann erneut von einem Pod der gleichen ID beansprucht werden. Dadurch wird sichergestellt, dass die gespeicherten Daten nicht verloren gehen und auch wieder ihren ursprünglichen Pods zugewiesen werden. Der Standardfall ist eine statische Bereitstellung der PVs. Dafür muss der Cluster-Administrator im Vorhinein eine bestimmte Anzahl an PVs erstellen [Aut18d]. Da dies im Sinne der Skalierbarkeit schnell an seine Grenzen stößt, müssen PVs dynamisch entlang der PVCs erzeugt werden. Um genau dies zu erreichen, besitzen PVs einen dynamischen Bereitstellungsmechanismus [Aut18d].

Sollte zu einem PVC kein passendes PV existieren, versucht Kubernetes das eingeforderte PV dynamisch zu erzeugen [Aut18d]. Diese Funktion wird mit Hilfe einer `StorageClasses` realisiert, die durch eine Konfigurationsdatei spezifiziert wie ein PV erzeugt werden soll [Aut18d][Aut18b]:

```
1  apiVersion: storage.k8s.io/v1
2  kind: StorageClass
3  metadata:
4    name: slow
5  provisioner: kubernetes.io/gce-pd
6  parameters:
7    type: pd-standard
```

Diese `StorageClass` muss bei einem PVC angegeben werden, um eine dynamische Bereitstellung der PVs zu ermöglichen [Aut18d]. Jeder Pod, der ein PV einfordert, benötigt ein eigenes PVC. Bei einer dynamischen Pod-Generierung mit StatefulSets, sind ebenso dynamisch generierte PVCs erforderlich. Aus diesem Grund bieten sich `volumeClaimTemplates` an um vorzugeben, wie PVCs dynamisch erstellt werden können, ähnlich wie bei `StorageClasses` und PVs. Dafür muss unter `spec.template.spec.containers.volumeMounts` in der Konfiguration des StatefulSets der Pfad angegeben werden, unter dem das Volume eingehängt wird. Das PVC-Template ist unter `spec.volumeClaimTemplates` anzugeben [Aut18h]:

```
1  # Bisherige Datanode-Konfiguration
2  # ...
3          volumeMounts:
4          - name: hadoop
5            mountPath: "/hadoop"
6    volumeClaimTemplates:
7    - metadata:
8        name: hadoop
9      spec:
10       accessModes: [ "ReadWriteOnce" ]
11       storageClassName:
12       resources:
13         requests:
14           storage: 100Gi
```

Mit diesem Mechanismus kann also eine variable Anzahl an Volumes erzeugt werden, je nach Menge der Datanodes. Außerdem erhalten alle abgestürzten oder gelöschten Pods wieder ihre ursprünglichen Volumes, sobald sie erneut erzeugt werden. Somit werden alle oben genannten Anforderungen an ein Speicherungsmechanismus abgedeckt und das HDFS kann schnell und dynamisch skaliert werden, ohne einen Datenverlust zu erleiden. Diese Anforderungen beziehen sich jedoch nur auf die Funktionalität von HDFS. Um aber auch die Performanz und den hohen Datendurchsatz zu gewährleisten, entsteht die Anforderung der Datenlokalität, also dass auf den Knoten gerechnet wird, wo auch die Daten liegen. Dies vermeidet unnötigen Datentransfer zwischen den Knoten. Da jedoch keine Annahmen darüber gemacht werden dürfen, wo sich ein Persistent-Volume im Cluster befindet, kann die Datenlokalität nicht sichergestellt und somit die Anforderung nicht

erfüllt werden. Dies bedeutet nun, dass eine dauerhafte Speicherung für HDFS zwar funktioniert, unter Umständen jedoch unperformant sein kann.

Die in diesem Kapitel vorgestellte Lösung wurde nicht praktisch getestet und ist nur theoretischer Natur. Sie basiert ausschließlich auf der Dokumentation von Kubernetes und wurde auf den hier vorgestellten Cluster angepasst.

3.2.2 Aufsetzen von YARN

YARN (*Yet Another Resource Negotiator*) ist ein Ressourcen-Verwaltungssystem, das für die Zuweisung und Überwachung von beispielsweise Rechenleistung an alle Jobs zuständig ist [Fou16]. Dieses System wurde dafür entworfen, einen Hadoop-Cluster und das damit verbundene HDFS für andere Big-Data-Systeme nutzbar zu machen, und bildet zusammen mit dem HDFS eine Grundlage für verteilte Rechensysteme [Mur18]. YARN ist ähnlich wie HDFS nach dem Master-Slave-Prinzip aufgebaut und besitzt einen Resourcemanager (Master) und eine beliebige Anzahl an Nodemanagern (Slaves).

Bereitstellung eines Resourcemanagers

Der Resourcemanager ist die Master-Instanz für YARN und beinhaltet zwei Komponenten: den *Scheduler* und *ApplicationsManager* [Fou16]. Der Scheduler ist für die Beschaffung von Ressourcen für die einzelnen Applikationen verantwortlich [Fou16]. Er ist ein reiner Scheduler, somit übernimmt er keine Überwachung der Applikationen und reagiert auch nicht auf Ausfälle und Fehler. Dies übernimmt jedoch der ApplicationsManager. Er ist für das Akzeptieren von Job-Anfragen, das initiale Starten der Applikationen und deren Neustart bei Fehlern zuständig [Fou16].

Ein YARN-Cluster benötigt wie ein HDFS-Cluster nur eine Master-Instanz, also nur *einen* ResourceManager. Dieser wird auch hier als StatefulSet repräsentiert und mit einem Replikationsfaktor von 1 in dessen Pod-Anzahl limitiert, wie in Kapitel 3.2.1 erläutert.

Die Kommunikation zwischen YARN und HDFS wird über die Angabe des Namenodes mit Hilfe der Umgebungsvariable `CORE_CONF_fs_defaultFS` realisiert. Damit würde der Resourcemanager bereits funktionieren und könnte Daten auf das HDFS schreiben und davon lesen. Um nun MapReduce auch unter YARN in dem Hadoop-Cluster zu unterstützen, muss ein *Shuffle Service* in der Umgebungsvariable `YARN_CONF_yarn_nodemanager_aux___services` angegeben werden [MDT18]:

```
1 # Resourcemanager-Konfiguration wie im StatefulSet-Geruest
2 # ...
3         env:
4         - name: CORE_CONF_fs_defaultFS
5           value: "hdfs://hdfs-namenode-0.hdfs-namenode-service
6                   .default.svc.cluster.local:8020"
7         - name: YARN_CONF_yarn_nodemanager_aux___services
8           value: "mapreduce_shuffle"
```

Bereitstellung und Integration von Nodemanagern

Ein Nodemanager ist der ausführende Dienst in YARN [VKV13]. Er übernimmt einen Teil der Containerverwaltung, überwacht diese und meldet Ressourceninformationen über den Knoten an den Resourcemanager [VKV13].

Jeder Kubernetes-Knoten, der keinen Resourcemanager oder Namenode enthält, besitzt einen Datanode, da möglichst viele Knoten die Schreib- und Leselast übernehmen können. Datanodes interagieren jedoch nur mit ihrem Namenode und nicht mit YARN. Um die Ressourcenverteilung über YARN dennoch zu ermöglichen, muss es eine Instanz geben, die Ressourceninformationen über jeden Knoten an den Resourcemanager sendet. Somit bietet es sich an, einen Nodemanager pro Datanode im Cluster zu betreiben. Es ist vorzuziehen, den Nodemanager auf demselben Knoten laufen zu lassen wie den Datanode, da neben der Verteilung der Knoteninformationen auch die Datenlokalität berücksichtigt werden sollte, um eine effizientere Datenverarbeitung zu gewährleisten.

Damit eine Nodemanager-Instanz pro Datanode auf dem selben Kubernetes-Knoten läuft, muss ein Nodemanager-Container im selben Pod erzeugt werden wie der Datanode-Container. In Kubernetes wird dies umgesetzt, indem die Container-Konfiguration des Nodemanagers in dieselbe Konfigurationsdatei unter `spec.template.spec.containers` geschrieben wird wie die für den Datanode:

```
1  # Datanode-/Nodemanager-Konfiguration wie im StatefulSet-Geruest
2  # ...
3        containers:
4        # Datanode-Container-Konfiguration
5        # ...
6        - name: yarn-nodemanager
7          image: fronepi/yarn-nodemanager
8          imagePullPolicy: Always
9          env:
10         - name: CORE_CONF_fs_defaultFS
11           value: "hdfs://hdfs-namenode-0.hdfs-namenode-service
12                    .default.svc.cluster.local:8020"
13         - name: YARN_CONF_yarn_nodemanager_aux___services
14           value: "mapreduce_shuffle"
```

Es ist zu erkennen, dass auch hier wieder die gleichen Umgebungsvariablen gesetzt werden wie bei dem Resourcemanager, nämlich `CORE_CONF_fs_defaultFS` und `YARN_CONF_yarn_nodemanager_aux___services`, um den Zugriff auf das HDFS und die Kompatibilität mit MapReduce zu gewährleisten.

Einführen eines History-Servers

Im Verlauf des Projektes wurde festgestellt, dass Apache MapReduce mit den verwendeten Containern nur funktionieren kann, wenn auch ein History-Server für die Jobs existiert. Für den History-Server kann, wie bekannt, ein StatefulSet mit Headless-Service erstellt werden, welcher den Port der Tabelle 3.4 freigeben muss. Wie auch bei dem HDFS-Namenode wird der Replikationsfaktor auf 1 gesetzt, da zu jedem gegebenen Zeitpunkt nur ein History-Server existieren soll.

Port	Beschreibung
10020	General communication

Tabelle 3.4. Der vom History-Server verwendete Port

Dem History-Server müssen die folgenden Umgebungsvariablen mitgegeben werden:

```
1  # History-Server Aufbau wie im StatefulSet Geruest
2  # ...
3            env:
4            - name: CORE_CONF_fs_defaultFS
5              value: "hdfs://hdfs-namenode-0.hdfs-namenode-service
6                      .default.svc.cluster.local:8020"
```

Zuletzt muss dem YARN-Resourcemanager und -Nodemanagern der History-Server über folgende Umgebungsvariablen bekannt gemacht werden:

```
1  # YARN-Resourcemanager bzw. -Nodemanager Aufbau wie im StatefulSet
2  # Geruest
3  # ...
4            env:
5            # ...
6            - name: MAPRED_CONF_mapreduce_jobhistory_address
7              value: "history-server-0.history-server-service.default
8                      .svc.cluster.local:10020"
```

Verbinden von Resource- und Nodemanager

Eine Fragestellung, die noch nicht beantwortet wurde ist, wie der Resourcemanager nun mit den Nodemanagern kommunizieren kann und andersherum. Wie auch bei HDFS wird jeweils für den Resourcemanager und den Nodemanagern ein Headless-Service wie bekannt definiert, wobei die Ports der Tabelle 3.5 für den Resourcemanager freigeschaltet werden müssen.

Port	Beschreibung
8030	Scheduler
8031	Job Tracker
8032	General communication
8033	Admin interface
8088	Web interface

Tabelle 3.5. Die vom YARN-Resourcemanager verwendeten Ports

Optional könnte bei Bedarf zusätzlich ein externer `NodePort` Service definiert werden, welcher den Zugriff auf das WebUI über den Port 8088 ermöglicht. Der Nodemanager verwendet die Ports der Tabelle 3.6.

Port	Beschreibung
8041	General communication
8042	Web interface

Tabelle 3.6. Die vom YARN-Nodemanager verwendeten Ports

Wie auch bei dem HDFS-Datanode muss nun auch dem YARN-Nodemanager über eine Umgebungsvariable (`YARN_CONF_yarn_resourcemanager_hostname`) mitgeteilt werden, unter welcher IP-Adresse bzw. welchem Hostnamen der Resourcemanager erreichbar ist. Aufgrund dessen, dass sich der Resourcemanager in einem StatefulSet und Headless-Service befindet, ist dies wie folgt leicht zu bewerkstelligen:

```
1 # YARN-Nodemanager und HDFS-Datanode Aufbau wie im StatefulSet Geruest
2 # ...
3           env:
4           - name: YARN_CONF_yarn_resourcemanager_hostname
5             value: "yarn-resourcemanager-0.yarn-resourcemanager-service
6                     .default.svc.cluster.local"
```

4

Zusammenfassung und Ausblick

In Kapitel 2.1 und 2.2 wurden die Anwendungen Docker und Kubernetes und ihre wesentlichen Eigenschaften und Funktionen erläutert.

Docker ist ein Container-Management-System, dessen Container im Rahmen dieser Ausarbeitung zusammen mit Kubernetes verwendet werden. Es wurde aufgezeigt, dass Docker das Problem löst, welches aufgrund abweichender Abhängigkeiten und Entwicklungswerkzeugen auf verschiedenen Systemen die Auslieferung von Anwendungen erschwert wird. Die Lösung ist, dass jeder Container ein gekapseltes virtuelles System darstellt, in dem alle benötigten Abhängigkeiten und Einstellungen inne leben. Ein Vergleich zwischen der Architektur von herkömmlichen Hypervisors und Docker zeigte, dass Dockers Leichtgewichtigkeit darauf basiert, dass es keine Hardware emuliert und auch keinen eigenen, sondern den Kernel des Wirtbetriebssystems verwendet. Abschließend wurden die für diese Arbeit wichtigsten Konfigurationsbausteine Dockers erklärt.

Kubernetes ist ein System zur Orchestrierung von containerisierten Applikationen und dient im Rahmen der Ausarbeitung der Verteilung, Überwachung, Steuerung, sowie Persistenz der auf Docker laufenden Big-Data-Technologien. Insbesondere wurde hierbei auf den Aufbau Kubernetes eingegangen und erklärt, wie die einzelnen Komponenten eines Kubernetescluster funktionieren. Dabei lag der Fokus vor allem auf dem Zusammenspiel zwischen Volumes, StatefulSets und Services, die in der Arbeit verwendet wurden, um zustandsbehaftete Big-Data-Technologien im Cluster zu verteilen. Abschließend wurde anhand einer Abbildung gezeigt, wie die Topologie eines einfachen Kubernetesclusters aussehen könnte.

In Kapitel 3.1 wurde gezeigt, wie das Datenbankmanagementsystem Cassandra auf einem Kubernetescluster verteilt werden kann. Dazu wurde zunächst die Frage, wie Cassandra in einen Dockercontainer installiert werden kann, geklärt. Als Basisimage wurde Ubuntu verwendet, um über den Paketmanager die offiziellen Pakete herunterzuladen. Aufgrund der ungünstigen Voreinstellungen Cassandras existierte das Problem, dass die Knoten von außen trotz freigeschalteter Ports nicht ansprechbar waren und sich in ihrem eigenen abgegrenzten Cluster befanden. Gelöst wurde dieses dadurch, dass über eine Shellskriptdatei die Cassandrakonfiguration entsprechend beim Start des Container angepasst wurde, sodass die IP-Adresse eines Seedknotens als Kommandozeilenargument übergeben werden kann. Beim Ansatz, den Container über Kubernetes zu orchestrieren, wurde offensicht-

lich, dass ein Deployment nicht ausreicht, sondern ein StatefulSet benötigt wird, damit der Seedknoten stets über denselben Hostnamen ansprechbar ist. Zusätzlich wurde ein externer Service definiert, damit das Datenbanksystem auch von außen ansprechbar ist.

In Kapitel 3.2 wurde gezeigt, wie die Big-Data-Infrastruktur Apache Hadoop mit Fokus auf MapReduce auf einem Kubernetescluster verteilt werden kann. Dazu wurde zunächst untersucht, wie das verteilte Dateisystem HDFS aufgesetzt werden kann. Von den dargestellten Möglichkeiten, den Namenode bereitzustellen, erwies sich das StatefulSet als am tauglichsten, da dieses die Verfügbarkeit und Einzigartigkeit sicherstellen kann. Verglichen mit dem Namenode kann es beliebig viele Datanodes geben und obwohl diese nicht eindeutig identifiziert werden müssen, wurde sich hier aus Gründen der Einheitlichkeit ebenfalls für ein StatefulSet entschlossen. Auch wurde beschrieben, wie die Dockercontainer über Umgebungsvariablen zu konfigurieren sind und welche Ports und Services eingerichtet werden müssen, damit die Datanodes mit dem Namenode und vice versa kommunizieren können. Abschließend wurden die Volumetypen hostPath und Persistent Volumes nach dem Kriterium Persistenz untersucht, welches sich am besten dafür eignen würde, die Daten der Knoten persistent, selbst nach Ausfall eines Pods, zu halten. Das Ergebnis dieser Untersuchung zeigte, dass Persistent Volumes dieses Kriterium erfüllen, hostPath jedoch nicht. Dies resultierte daraus, dass ein `hostPath` einem Pod nach einem Neustart nicht wieder zuordenbar ist, das Persistent Volume jedoch fest gebunden wird.

Das Unterkapitel 3.2.2 befasst sich damit, wie YARN nun im Zusammenhang mit dem bereits auf Kubernetes laufendem HDFS ebenfalls auf dem Cluster verteilt werden kann. Dazu wurde zunächst gezeigt, wie ein YARN-Resourcemanager als StatefulSet auf Kubernetes aufgesetzt und mit dem, bereits darauf laufendem, HDFS über eine Umgebungsvariable verbunden werden kann. Folgend wurde beschrieben, wie die YARN-Nodemanager, ebenfalls als StatefulSet, so konfiguriert werden können, dass diese jeweils auf denselben Maschinen laufen, wie auch die HDFS-Datanodes. Zu diesem Zwecke wurden die bereits existierende Podkonfiguration der Datanodes um Container mit den Nodemanagern erweitert. Weiterhin wurde der History-Server eingeführt und konfiguriert, welchen die in dieser Ausarbeitung verwendeten Container benötigen, um MapReduce Jobs auf YARN ausführen zu können. Sein Hostname musste ebenfalls über eine Umgebungsvariable dem Resourcemanager und Nodemanagern mitgeteilt werden. Abschließend wurde dargestellt, wie der Resourcemanager und die Nodemanager über das Netzwerk verbunden werden können. Dazu mussten zunächst die korrekten TCP Ports und eine Umgebungsvariable, welche die Adresse des Resourcemanagers festlegt, für die Nodemanager eingerichtet werden.

Die Ergebnisse der Arbeit zeigen, dass sich Docker und Kubernetes durchaus dazu eignen, auch zustandsbehaftete Big-Data-Technologien, wie zum Beispiel verteilte Datenbankmanagement- oder Dateisysteme, bereitzustellen. Die Konfiguration der Systeme und der Umgang mit Kubernetes erwies sich anfangs als sehr komplex und unübersichtlich. Dies wird jedoch dadurch ausgeglichen, dass, sobald die Anwendungen aufgesetzt wurden, der Wartungs- und Verwaltungsaufwand sehr

gering ausfallen. Es benötigt lediglich nur zwei Befehle, um die Anwendungen zu starten oder zu skalieren.

In dieser Arbeit wurden zwar Persistent-Volumes als Lösung für das Persistenzproblem zustandsbehafteter Anwendungen genannt, jedoch wurden diese nicht ausführlich in der Praxis getestet. Eine fortführende Arbeit könnte aufbauend darauf auch behandeln, wie zum Beispiel Anwendungen, welche aus vielen kleinen Programmen bestehen, in denen jeder Container oder Pod verschieden konfiguriert wird, bereitgestellt werden könnten. Dies ging über das Ziel dieser Arbeit hinaus, insbesondere da hier ausschließlich Technologien untersucht wurden, die bereits vom Aufbau her für eine leichte Verteilung gestaltet sind.

Literaturverzeichnis

Aut18a. Authors, The Kubernetes: *Deployments - Creating a Deployment.* [Online; Stand 28. Mai 2018; https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment], 2018.

Aut18b. Authors, The Kubernetes: *Dynamic Volume Provisioning.* [Online; Stand 16.07.2018; https://kubernetes.io/docs/concepts/storage/dynamic-provisioning], 2018.

Aut18c. Authors, The Kubernetes: *Images.* [Online; Stand 17. Juli 2018; https://kubernetes.io/docs/concepts/containers/images/], Juni 2018.

Aut18d. Authors, The Kubernetes: *Persistent Volumes.* [Online; Stand 15.05.2018; https://kubernetes.io/docs/concepts/storage/persistent-volumes/], 2018.

Aut18e. Authors, The Kubernetes: *Pod Overview - Pod Templates.* [Online; Stand 28. Mai 2018; https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pod-templates], 2018.

Aut18f. Authors, The Kubernetes: *Production-Grade Container Orchestration.* [Online; Stand 23. Mai 2018; https://kubernetes.io/], Mai 2018.

Aut18g. Authors, The Kubernetes: *Services.* [Online; Stand 29. Mai 2018; https://kubernetes.io/docs/concepts/services-networking/service/], 2018.

Aut18h. Authors, The Kubernetes: *StatefulSets.* [Online; Stand 30. Mai 2018; https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/], 2018.

Aut18i. Authors, The Kubernetes: *Using a Service to Expose Your App.* [Online; Stand 05. Juni 2018; https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/], 2018.

Aut18j. Authors, The Kubernetes: *Using kubectl to Create a Deployment.* [Online; Stand 28. Mai 2018; https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/], 2018.

Aut18k. Authors, The Kubernetes: *Volumes.* [Online; Stand 30. Mai 2018; https://kubernetes.io/docs/concepts/storage/volumes/], 2018.

Bai17. Baier, Jonathan: *Getting Started with Kubernetes, Second Edition*. Packt Publishing, 2017.

Dat18. DataStax: *What is Apache Cassandra™?* [Online; Stand 18. Juli 2018; https://academy.datastax.com/planet-cassandra/what-is-apache-cassandra], Juli 2018.

Fou13. Foundation, The Apache Software: *HDFS Architecture Guide*. [Online; 07.06.2018; https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html], 2013.

Fou16. Foundation, The Apache Software: *Apache Hadoop YARN*. [Online; 16.07.2018; https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.htmll], 2016.

Fou17. Foundation, The Linux: *Cloud Native Computing Foundation*. [Online; Stand 23. Mai 2018; https://www.cncf.io/], 2017.

Fou18a. Foundation, Cloud Native Computing: *GitHub: Kubernetes: Production-Grade Container Scheduling and Management*. [Online; Stand 23. Mai 2018; https://github.com/kubernetes/kubernetes], Mai 2018.

Fou18b. Foundation, The Apache Software: *Installation from Debian packages*. [Online; Stand 18. Juli 2018; https://cassandra.apache.org/download/], Juli 2018.

Fou18c. Foundation, The Apache Software: *What Is Apache Hadoop?* [Online; Stand 24.05.2018; https://hadoop.apache.org/], 2018.

IBM17. IBM: *What Is MapReduce?* [Online; 05.06.2018; https://www.ibm.com/analytics/hadoop/mapreduce], 2017.

IBM18. IBM: *What Is HDFS?* [Online; 05.06.2018; https://www.ibm.com/analytics/hadoop/hdfs], 2018.

Inc18a. Inc., Docker: *Definition of: layer*. [Online; 05.06.2018; https://docs.docker.com/glossary/?term=layer], 2018.

Inc18b. Inc., Docker: *Dockerfile reference*. [Online; 31.05.2018; https://docs.docker.com/engine/reference/builder/], 2018.

Inc18c. Inc., Docker: *What is Docker*. [Online; Stand 23. Mai 2018; https://www.docker.com/what-docker], 2018.

Kro17. Krochmalski, Jaroslaw: *Docker and Kubernetes for Java Developers: Scale, deploy, and monitor multi-container applications*. Packt Publishing - ebooks Account, 2017.

MDT18. MapR Data Technologies, Inc.: *yarn-site.xml*. [Online; Stand 17.07.2018; https://maprdocs.mapr.com/52/ReferenceGuide/yarn-site.xml.html], 2018.

Met15. Metz, Cade: *Google Made Its Secret Blueprint Public to Boost Its Cloud*. [Online; Stand 23. Mai 2018; https://www.wired.com/2015/06/google-kubernetes-says-future-cloud-computing/], Juni 2015.

MG17. McKendrick, Russ und Scott Gallagher: *Mastering Docker - Second Edition: Master this widely used containerization tool*. Packt Publishing - ebooks Account, 2017.

Mic18. Microsoft: *The Big Bang: How the Big Data Explo- sion Is Changing the World.* [Online; Stand 24.05.2018; https://news.microsoft.com/2013/02/11/the-big-bang-how-the-big-data-explosion-is-changing-the-world/], 2018.

Mur18. Murthy, Arun: *Introducing Apache Hadoop YARN.* [Online; Stand 16.07.2018; https://hortonworks.com/blog/introducing-apache-hadoop-yarn], 2018.

oLS18. Labor Statistics, U.S. Bureau of: *43-9011 Computer Operators.* [Online; Stand 29.05.2018; https://www.bls.gov/oes/current/oes439011.htm], 2018.

Say17. Sayfan, Gigi: *Mastering Kubernetes.* PACKT PUB, 2017.

VKV13. Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas et al.: *Apache Hadoop YARN: Yet Another Resource Negotiator.* [Online; Stand 17.07.2018; http://dx.doi.org/10.1145/2523616.2523633], 2013.