Universität Augsburg
Fakultät für Angewandte Informatik

Modellbasierte Testautomatisierung eines verteilten, adaptiven Load-Balancing-Systems

Masterarbeit

im Studiengang Informatik

zur Erlangung des akademischen Grades

Master of Science

von

Gerald Siegert

Mat.-Nr.:	1450117
Datum:	18. März 2018
Betreuer:	M.Sc. Benedikt Eberhardinger
1. Prüfer:	Prof. Dr. X
2. Prüfer:	Prof. Dr. Y

Zusammenfassung

Abstract

Inhaltsverzeichnis

Verzeichnisse

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Abkürzungsverzeichnis

AM ApplicationManager

AppMstr ApplicationMaster

HDFS Hadoop Distributed File System

MARP `maximum-am-resource-percent`

MC Model Checking

NM NodeManager

RM ResourceManager

SWIM	Statistical Workload Injector for Mapreduce
TLS	Timeline-Server
dfs-w	`TestDFSIO -write`
rtw	`randomtextwriter`
tgen	`teragen`
dfs-r	`TestDFSIO -read`
wc	`wordcount`
rw	`randomwriter`
sort	`sort`
tsort	`terasort`
pi	`pi`
pent	`pentomino`
tstsort	`testmapredsort`
tval	`teravalidate`

1. Einleitung

Im Bereich der Softwaretests wird heutzutage sehr viel mit automatisierten Testverfahren gearbeitet. Dies ist insofern logisch, als dass diese Testautomatisierung einerseits Aufwand und damit andererseits direkt Kosten einer Software einspart. Daher gibt es vor allem im Bereich der Komponententests zahlreiche Frameworks, mit denen Tests einfach und automatisiert erstellt bzw. ausgeführt werden können. Ein Beispiel für ein solches Testframework wäre das *xUnit*-Framework, zu dem u. A. JUnit[1] für Java und NUnit[2] für .NET zählen. Dabei werden zunächst einzelne Testfälle erstellt und können im Anschluss mit der jeweils aktuellen Codebasis jederzeit ausgeführt werden. Automatisierte Tests können auch dazu genutzt werden, um einen einzelnen Test mit verschiedenen Eingaben durchzuführen. Dadurch können verschiedene Eingabeklassen (wie negative oder positive Ganzzahlen) mit sehr geringem Aufwand in einem Test genutzt werden und somit verschiedene Testfälle direkt ausgeführt werden, wodurch eine massive Kosteneinsparung einhergeht [1].

Es gibt aber nicht nur Frameworks für Komponententests, sondern auch für modellbasierte Testverfahren wie z. B. dem Model Checking (MC). Beim MC wird ein Modell mithilfe eines entsprechenden Frameworks automatisiert auf seine Spezifikation getestet und geprüft, unter welchen Umständen diese verletzt wird [2, 3].

In dieser Masterarbeit soll daher nun ein verteiltes, adaptives Load-Balancing-System getestet werden. Hauptziel ist es, zu ermitteln, wie ein modellbasierter Testansatz auf ein komplexes Beispiel übertragen werden kann. Dafür wird zunächst ein reales System als vereinfachtes Modell nachgebildet und anschließend mithilfe eines MC getestet. Es soll dabei auch ermittelt werden, wie ein reales System in das Modell eingebunden werden kann und wie bei Problemen mit asynchronen Prozessen innerhalb des verteilten Systems umgegangen werden muss.

[1] `https://junit.org`
[2] `https://nunit.org/`

2. Aufbau der Fallstudie

In der Fallstudie im Rahmen dieser Masterarbeit wird **Apache™Hadoop®**[1] mithilfe eines modellbasierten Tests getestet. Da Hadoop normalerweise keine adaptive Komponente besitzt, wurde Hadoop mit der von Zhang u. a. entwickelten selbst-adaptiven Komponente erweitert und ein Cluster mithilfe der ebenfalls von Zhang u. a. entwickelten Plattform Hadoop-Benchmark erstellt.

2.1. Apache Hadoop

Apache Hadoop ist ein Open-Source-Software-Projekt, mit dessen Hilfe ermöglicht wird, Programme zur Datenverarbeitung mit großen Ressourcenbedarf auf verteilten System auszuführen. Hadoop wird von der *Apache Foundation* entwickelt und bietet verschiedene Komponenten an, welche vollständig skalierbar sind, von einer einfachen Installation auf einem PC bis hin zu einer Installation über mehrere Server in einem Serverzentrum. Hadoop besteht hauptsächlich aus folgenden Kernmodulen [5]:

Hadoop Common Gemeinsam genutzte Kernkomponenten

Hadoop YARN Framework zur Verteilung und Ausführung von Anwendungen und das dazugehörige Ressourcen-Management

Hadoop Distributed File System Kurz HDFS, Verteiltes Dateisystem

Hadoop MapReduce YARN-Basiertes System zum Verarbeiten von großen Datenmengen

Hadoop ermöglicht es dadurch, sehr einfach mit Anwendungen umzugehen, welche große Datenmengen verarbeiten. Da es für Hadoop nicht relevant ist, auf wie vielen Servern es läuft, kann es beliebig skaliert werden, wodurch entsprechend viele Ressourcen zur Bearbeitung und Speicherung von großen Datenmengen zur Verfügung stehen können.

Die Kernidee der Architektur von **YARN** ist die Trennung vom Ressourcenmanagement und Scheduling. Dazu besitzt der Master bzw. *Controller* den ResourceManager (RM), welcher für das gesamte System zuständig ist und die Anwendungen im System verteilt und überwacht und somit auch als *Load-Balancer* agiert. Er besteht aus zwei Kernkomponenten, dem ApplicationManager (AM) und dem *Scheduler*. Der AM ist für die Annahme und Ausführung von einzelnen Anwendungen zuständig, denen der Scheduler die dafür notwendigen Ressourcen im Cluster zuteilt.

[1] `https://hadoop.apache.org/`

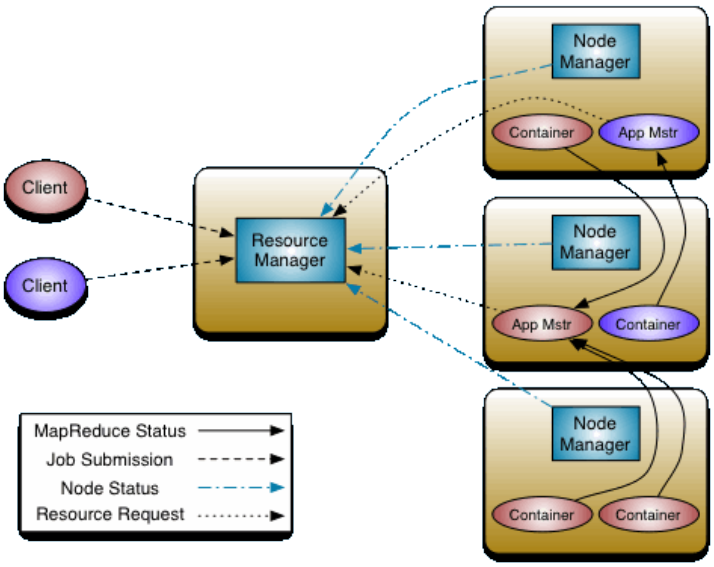

Abbildung 2.1.: Architektur von YARN (entnommen aus [6])

Jeder Slave-*Node* im Hadoop-Cluster besitzt einen NodeManager (NM), welcher für die Überwachung der Ressourcen des Nodes und der darauf ausgeführten Anwendungs-Container zuständig ist und diese dem RM mitteilt.

Jede YARN-Anwendung bzw. Job besteht aus einem oder mehreren Ausführungsversuchen, genannt *Attempts*, denen wiederum mehrere *Container* zugeordnet sind. Container können auf einem beliebigen Node ausgeführt werden und repräsentieren die Ausführung eines Tasks der Anwendung. Ein besonderer Container bildet dabei der ApplicationMaster (AppMstr), welcher innerhalb seines Attempts für das anwendungsbezogene Monitoring und die Kommunikation mit dem RM und NM zuständig ist und die dazu notwendigen Informationen bereit stellt [6].

Hadoop enthält zudem einen sog. Timeline-Server (TLS). Er ist speziell dafür entwickelt, die Metadaten und Logs der YARN-Anwendungen zu speichern und jederzeit, also auch als Anwendungshistorie, auszugeben [7].

Das **HDFS** basiert auf der gleichen Architektur wie YARN und besitzt ebenfalls einen Master und mehrere Slaves, welches in der Regel die gleichen Nodes sind wie bei YARN sind. Der *NameNode* ist als Master für die Verwaltung des Dateisystems zuständig und reguliert den Zugriff auf die darauf gespeicherten Daten. Die Daten selbst werden in mehrere Blöcke aufgeteilt auf den *DataNodes* gespeichert. Um den Zugriff auf die Daten im Falle eines Node-Ausfalls zu gewährleisten, wird jeder Block auf anderen Nodes repliziert. Dateioperationen (wie Öffnen oder Schließen) werden direkt auf den DataNodes ausgeführt, sie sind darüber hinaus auch dafür verantwortlich, dass Clients die Daten lesen oder beschreiben können [8].

MapReduce bietet analog zu YARN die Möglichkeit, Anwendungen mit einem großen Ressourcenbedarf, welche große Datenmengen verarbeiten, auf einem gesamten Cluster auszuführen. Dazu werden bei einem MapReduce-Job die Eingabedaten aufge-

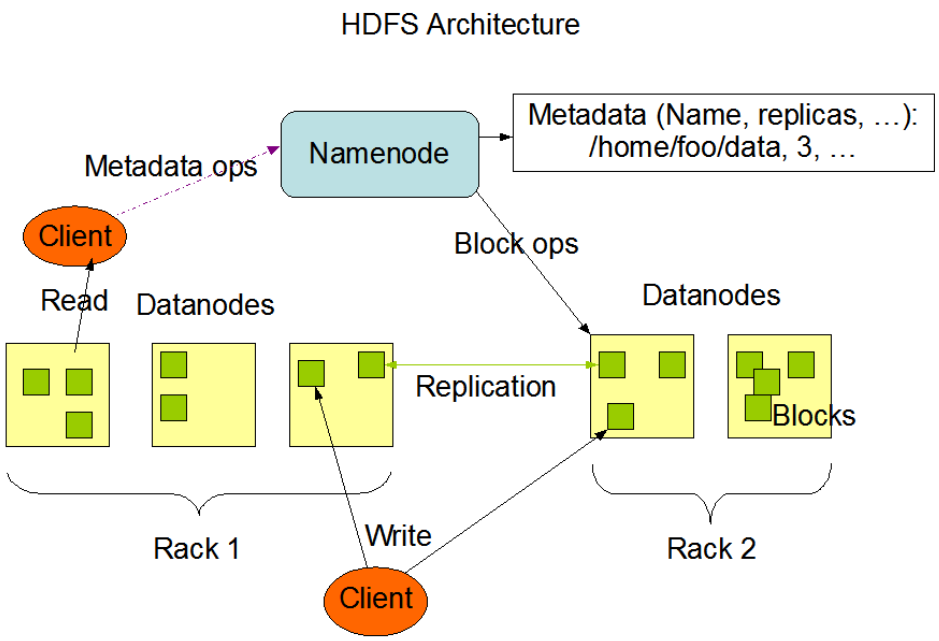

Abbildung 2.2.: Architektur des HDFS (entnommen aus [8])

teilt, anschließend von den sog. *Map Tasks* verarbeitet und deren Ausgaben von den sog. *Reduce Tasks* geordnet. Für die Ein- und Ausgabe der Daten wird in der Regel das HDFS, für die Ausführung der einzelnen Tasks YARN genutzt [9]. MapReduce kann auch als Vorgänger von YARN angesehen werden, da YARN auch als *MapReduce Next Gen* bzw. *MRv2* bezeichnet wird und aufgrund der API-Kompatibilität von YARN jede MapReduce-Anwendung in der Regel auch auf YARN ausgeführt werden kann [6, 10].

2.2. Adaptive Komponente in Hadoop

Eine normale Hadoop-Installation besitzt keine adaptive Komponente, sondern rein statische Einstellungen. Um damit Hadoop zu optimieren, müssen die Einstellungen immer manuell auf den jeweils benötigten Anwendungstyp angepasst werden. Dazu gibt es auch bereits verschiedene Scheduler, den *Fair Scheduler*, welcher alle Anwendungen ausführt und ihnen gleich viele Ressourcen zuteilt, und den *Capacity Scheduler*. Letzterer sorgt dafür, dass nur eine bestimmte Anzahl an Anwendungen pro Benutzter gleichzeitig ausgeführt wird und teilt ihnen so viele Ressourcen zu, wie benötigt werden bzw. der Benutzter nutzen darf. Entwickelt wurde der Capacity Scheduler vor allem für Cluster, die von mehreren Organisationen gemeinsam verwendet werden und sicherstellen soll, dass jede Organisation eine Mindestmenge an Ressourcen zur Verfügung hat [11].

Je nach Bedarf besitzt der Capacity Scheduler entsprechende Einstellungen, um z. B. den verfügbaren Speicher pro Container festzulegen. Eine weitere Einstellung des Schedulers ist `maximum-am-resource-percent`, auch MARP gennant, der angibt, wie viele Prozent der gesamten Ressourcen durch AppMstr-Container genutzt werden dürfen [11]. Damit bewirkt diese Einstellung indirekt auch die maximale Anzahl an

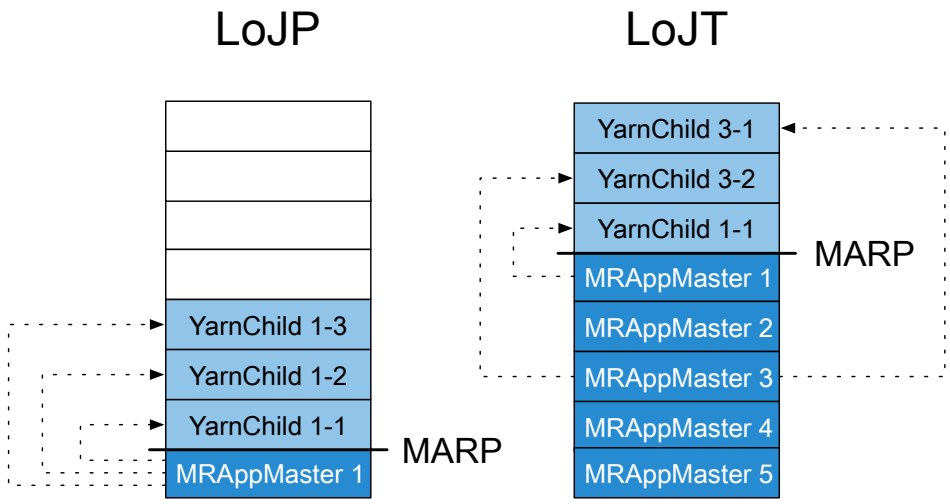

Abbildung 2.3.: LoJP und LoJT in Hadoop (entnommen aus [4])

Anwendungen, die gleichzeitig ausgeführt werden dürfen. Da der MARP-Wert jedoch nicht während der Laufzeit dynamisch angepasst werden kann, haben Zhang u. a. in [4] einen Ansatz zur dynamischen Anpassung des MARP-Wertes zur Laufzeit von Hadoop vorgestellt. Dadurch wird der MARP-Wert abhängig von den ausgeführten Anwendungen adaptiv zur Laufzeit angepasst, sodass immer möglichst viele Anwendungen gleichzeitig ausgeführt werden können. Dadurch werden Anwendungen im Schnitt um bis zu 40 % schneller ausgeführt [4].

Der Hintergrund dieser *Selfbalancing-Komponente* ist der, dass durch den MARP-Wert der für die Anwendungen verfügbare Speicher in zwei Teile aufgeteilt wird. Im einen Teil befinden sich alle derzeit ausgeführten AppMstr, im anderen Teil die von den Anwendungen benötigten weiteren Container. Wie groß der Teil für die AppMstr ist, wird nun durch den MARP-Wert bestimmt. Ist der MARP-Wert zu klein, können nur wenige AppMstr (und damit Anwendungen) gleichzeitig ausgeführt werden (*Loss of Jobs Parallelism*, LoJP). Ist der MARP-Wert jedoch zu groß, können für die ausgeführten Anwendungen nur wenige Container bereitgestellt werden, wodurch sich die Ausführung für eine Anwendung wesentlich verlangsamt (*Loss of Job Throughput*, LoJT)[4]. Abbildung 2.3 illustriert beide Situationen, wodurch einerseits viel Speicher für weitere Anwendungscontainer ungenutzt bleiben kann, andererseits aber zahlreiche AppMstr ohne laufende Anwendungscontainer Speicher unnötig belegen können.

Die Selfbalancing-Komponente passt daher den MARP-Wert abhängig von der Speicherauslastung dynamisch zur Laufzeit an. So wird der MARP-Wert verringert, wenn die Speicherauslastung sehr hoch ist, und erhöht, wenn die Speicherauslastung sehr niedrig ist [4]. Dadurch wird es ermöglicht, dass die maximal mögliche Anzahl an Anwendungen ausgeführt werden kann. Die Evaluation von Zhang u. a. ergab zudem, dass die dynamische Anpassung des MARP-Wertes zudem auch effizienter ist als eine manuelle, statische Optimierung.

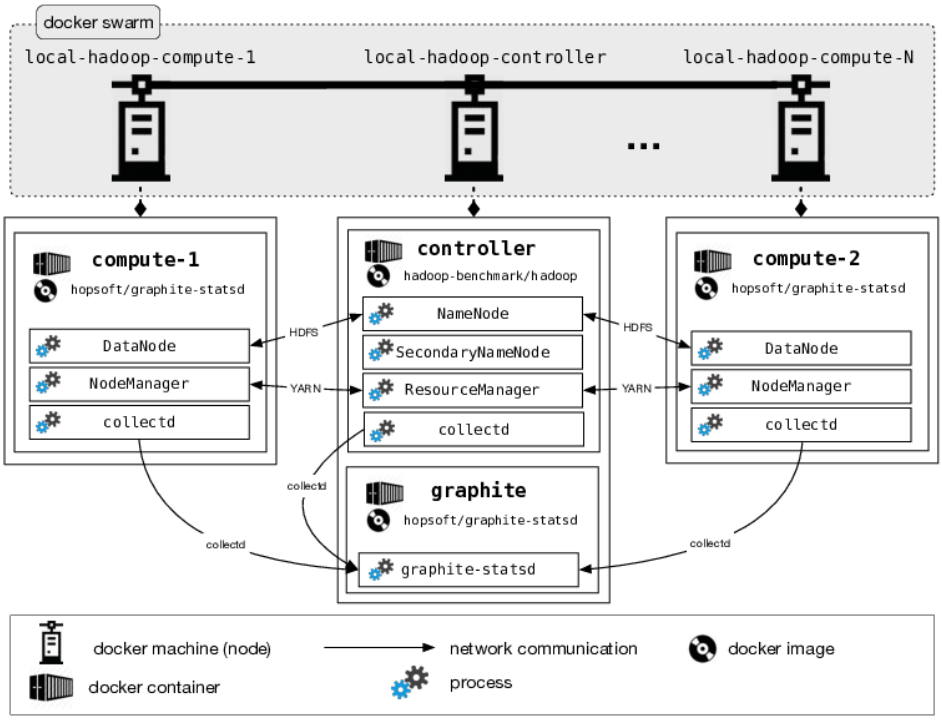

Abbildung 2.4.: High-Level-Architektur von Hadoop-Benchmark [12]

2.3. Umsetzung des realen Clusters

Zhang u. a. haben im Rahmen ihrer gesamten Forschungsarbeit die Open-Source-Plattform Hadoop-Benchmark entwickelt und auf Github zur Verfügung gestellt.[2] Sie wurde speziell zum Einsatz in der Forschung erstellt und kann jederzeit an die eigenen Bedürfnisse angepasst werden. Zur Umsetzung des realen Clusters im Rahmen dieser Masterarbeit wurde daher eine speziell angepasste Version der Plattform eingesetzt.

2.3.1. Plattform Hadoop-Benchmark

Die Plattform ist in mehrere Szenarien unterteilt, darunter ein Hadoop in der Version 2.7.1 ohne Änderungen und ein darauf basierendes Szenario mit der Selfbalancing-Komponente. Hadoop-Benchmark basiert auf der Software *Docker*[3] und dem dazugehörigen Tool *Docker Machine*, um damit einfach und schnell ein Hadoop-Cluster aufbauen zu können. Mit *Graphite*[4] ist zudem ein Monitoring-Tool enthalten, mit dem die Performance des Clusters überwacht und analysiert werden kann.

Abbildung 2.4 zeigt die grundlegende Architektur der Plattform, die mithilfe eines Docker-Swarms auf mehreren *Docker Machines* (für den Einsatz von Docker eingerichtete virtuelle Maschinen) ein Cluster erstellt, auf denen dann in den Docker-Containern das eigentliche Hadoop-Cluster ausgeführt wird. Jeder Hadoop-Container enthält zudem

[2] https://github.com/Spirals-Team/hadoop-benchmark
[3] https://www.docker.com/
[4] https://graphiteapp.org/

	Cluster-PC	**VM-PC**	**Windows-VM**
CPU	Intel Core i5-4570 @ 3,2 GHz x 4		4 CPU Cores
RAM	16 GB	16 GB	8 GB
SSD	512 GB	512 GB	$\leq$ 100 GB VHD
OS	Ubuntu 16.04 LTS	Ubuntu 16.04 LTS	Windows 10 1709 Edu.

Tabelle 2.1.: Spezifikationen der verwendeten PCs und Windows-VM

das Tool *collectd*[5], was das Monitoring des Containers auf Systemebene übernimmt und die Daten an den Graphite-Container auf der Controller-Machine übermittelt. Es ist dabei möglich, eine beliebige Anzahl an Nodes zu nutzen. Auch ist es möglich, den Docker Machines einen beliebig großen Arbeitsspeicher zur Verfügung zu stellen.

Die Plattform Hadoop-Benchmark enthält zudem einige Benchmark-Anwendungen:

- Hadoop Mapreduce Examples
- Intel HiBench[6]
- Statistical Workload Injector for Mapreduce (SWIM) [7]

Eine Besonderheit bildet der SWIM-Benchmark, welcher sehr Ressourcenintensiv ist und daher auf einem *Single Node Cluster*, also einem kompletten Hadoop-Cluster auf nur einem Computer, sehr zeitintensiv sein kann. Der Intel HiBench-Benchmark besteht aus Kategorien wie *Machine Learning* oder Graphen, welche wiederum aus einen oder mehreren *Workloads* bestehen, welche entsprechende Anwendungen bzw. Algorithmen auf dem Hadoop-Cluster ausführen. Einige der Hibench-Workloads basieren auf den Mapreduce Examples, welche wiederum voneinander unabhängige Beispielanwendungen für Hadoop darstellen.

2.3.2. Anpassungen und Setup

Da mithilfe der Plattform Hadoop-Benchmark die Erstellung eines Hadoop-Clusters massiv vereinfacht wird, kommt die Plattform auch in dieser Masterarbeit zum Einsatz. Da Docker und Hadoop aber vor allem für den Einsatz in einer Linux-Umgebung entwickelt wurden, wird dazu ein eigener PC mit Ubuntu 16.04 LTS genutzt. Da S# das .NET-Framework, und damit Windows, benötigt, wird dafür ebenfalls ein eigener PC verwendet. Im konkreten Versuchsaufbau wird für Windows eine VM genutzt, welche auf einem anderen PC als das Cluster ausgeführt wird. Die genauen Spezifikationen der PCs und der Windows-VM sind in Tabelle 2.1 aufgelistet. Die Windows-VM und der Cluster-PC werden mithilfe von SSH-Verbindungen miteinander verbunden.

Auf dem Cluster-PC nutzt Docker-Machine zur Erstellung, Verwaltung und Ausführung der VMs die Treiber von VirtualBox 5.2[8], zum Abrufen der Daten der REST-API

[5] `https://collectd.org/`
[6] `https://github.com/intel-hadoop/HiBench`
[7] `https://github.com/SWIMProjectUCB/SWIM`
[8] `https://www.virtualbox.org/`

über die SSH-Verbindung wird *curl*[9] genutzt. Für das Cluster werden 4 Nodes, der Controller sowie eine Consul-VM zur internen Verwaltung der Netzwerkverbindungen zwischen den VMs und Docker-Containern erstellt. Der Controller erhält 4 GB RAM, jeder der vier Nodes jeweils 2 GB, für den Consul sind 512 MB ausreichend. Für die Windows-VM wird ebenfalls VirtualBox 5.2 eingesetzt.

In keinem Szenario der Plattform Hadoop-Benchmark wird standardmäßig der TLS von Hadoop gestartet. Daher wurde für diese Fallstudie basierend auf dem Selfbalancing-Szenario ein neues Szenario erstellt, bei dem der TLS gestartet wird. Dadurch ist einerseits die Selfbalancing-Komponente von Zhang u. a. aktiv und andererseits besteht die Möglichkeit, für das Monitoring zusätzlich den TLS zu nutzen.

Um die in dieser Fallstudie benötigten Befehle einfach ausführen zu können, wurden zwei eigene Scripte erstellt, welche, sofern möglich, auf den bestehenden Scripten der Plattform aufbauen. Das Setup-Script dient für folgende clusterbezogene Zwecke:

- Starten, Beenden und Löschen des kompletten Clusters bzw. Hadoops
- Starten und Beenden des Docker-Containers eines Hadoop-Nodes
- Hinzufügen und Entfernen der Netzwerkverbindung des Docker-Containers eines Hadoop-Nodes
- Ausführen von eigenen Befehlen auf dem Docker-Container des Controllers

Das zweite erstellte Script dient ausschließlich zum Starten der Benchmarks.

Für die Befehle, die das gesamte Cluster betreffen, wird vom Setup-Script meist auf das in Hadoop-Benchmark enthaltene Start-Script zugegriffen. Die Befehle, welche die Docker-Container der Nodes betreffen, sowie das Ausführen von Befehlen im Controller-Container, werden vom Setup-Script direkt ausgeführt. Für das Starten der Benchmarks werden dagegen die in Hadoop-Benchmark enthaltenen Ausführungs-Scripte der Benchmarks gestartet.

[9] `https://curl.haxx.se/`

3. Aufbau des Modells

Die grundlegende Architektur des gesamten Aufbaus besteht aus den drei rechts abgebildeten Schichten. Die oberste Schicht bildet das S#-Modell von Hadoop YARN, welches die relevanten YARN-Komponenten und Komponentenfehler abbildet. Das reale Pendant dazu bildet das reale Hadoop-Cluster auf einem eigenen PC als unterste Schicht. Die Verbindung zwischen Modell und realem Cluster bildet der Treiber als eigenständige Schicht. Der Treiber besteht aus folgenden Komponenten:

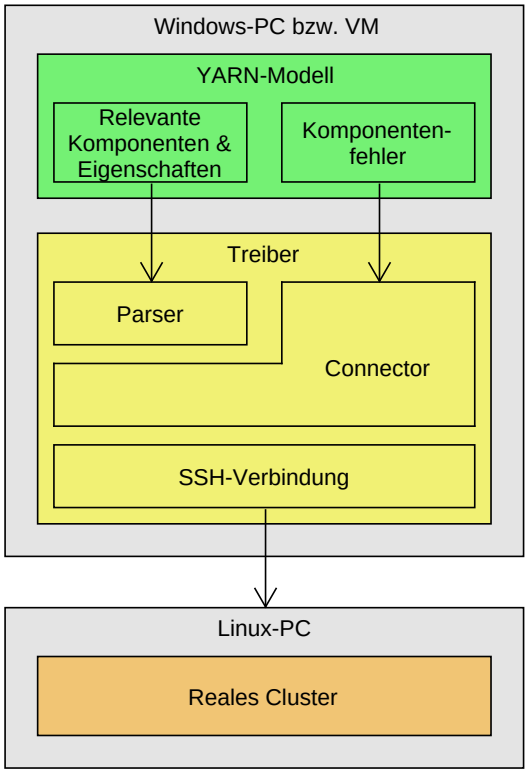

Abbildung 3.1.: Grundlegende Architektur des Gesamtmodells

Parser Verarbeitet die Monitoring-Ausgaben vom realen Cluster und konvertiert diese für die Nutzung im Modell

Connector Abstrahierung der SSH-Verbindung mit den auszuführenden Befehlen

SSH-Verbindung Verbindung zum Cluster-PC

Auf den Treiber bzw. das reale System wird meist mithilfe des Parsers zugegriffen. Lediglich zum Starten von Anwendungen, zum Aktivieren bzw. Deaktivieren von Komponentenfehlern u. Ä. auf dem realen Cluster wird direkt der Connector genutzt.

3.1. YARN-Modell

Abbildung 3.2 beschreibt im Grunde bereits das gesamte von S# verwendete YARN-Modell. Enthalten sind alle hier relevanten Komponenten sowie deren Eigenschaften. Als Eigenschaften wurden die Daten aufgenommen, welche mithilfe von Shell-Kommandos bzw. mithilfe der REST-API von YARN ermittelt werden können.

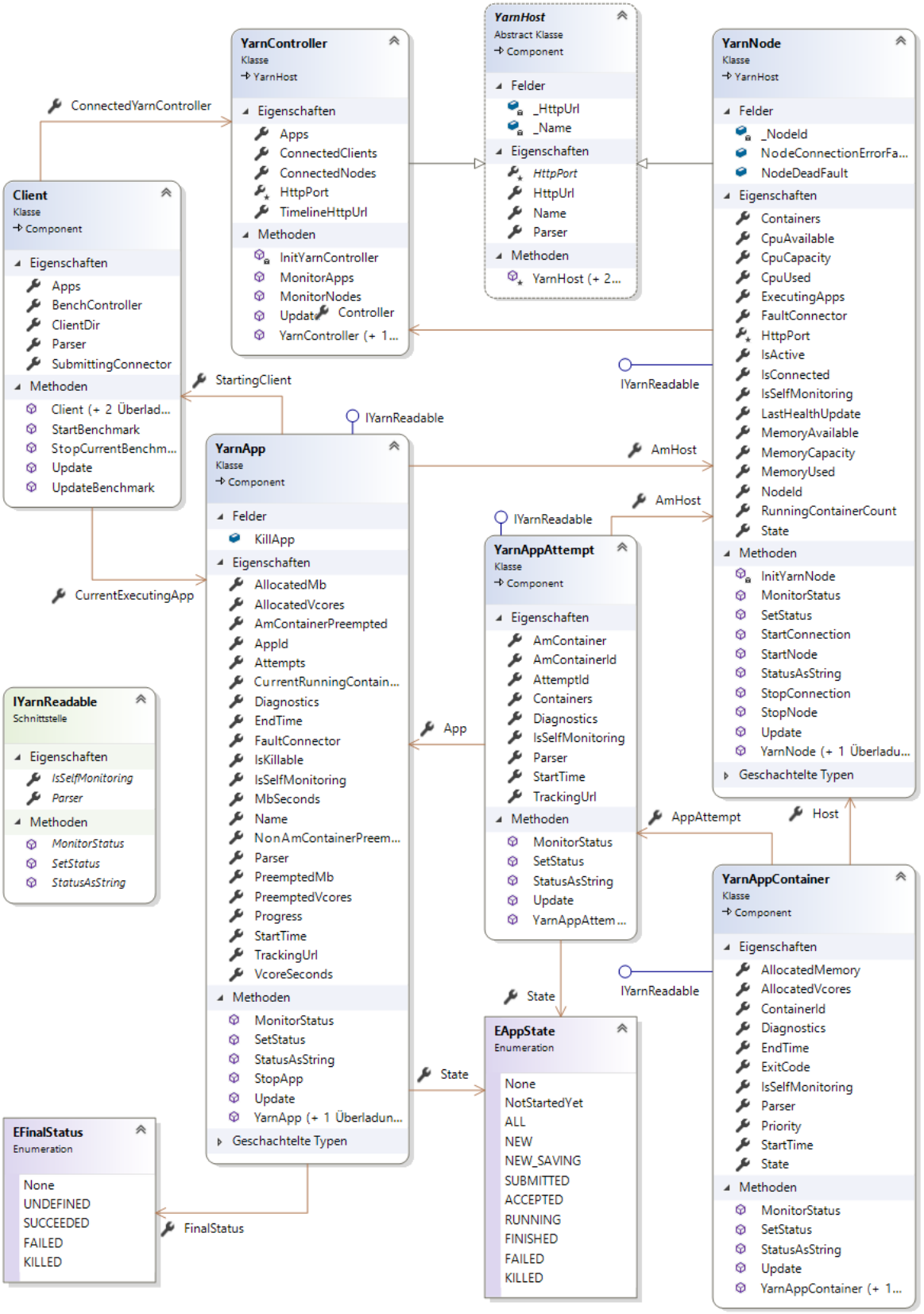

Abbildung 3.2.: Aufbau des YARN-Modells. Das Modell wurde mithilfe des Klassendiagramm-Designers in Visual Studio 2017 visualisiert. Daher werden Assoziationen mit höherer Multiplizität als 1, die daher mithilfe von `List<T>` umgesetzt wurden (z. B. `YarnApp.Attempts`) im Diagramm nicht als Assoziationen zwischen den Klassen angezeigt.

Die abstrakte Basisklasse `YarnHost` stellt die Basis für alle Hosts des Clusters dar, also dem `YarnController` mit dem RM, und dem `YarnNode`, was einen Node darstellt, auf dem die Anwendungen bzw. deren Container ausgeführt werden. Die abstrakte Eigenschaft `YarnHost.HttpPort` dient als Hilfs-Eigenschaft, da Controller und Nodes unterschiedliche Ports für die Weboberfläche nutzen, deren URL mit Port in der Eigenschaft `YarnHost.HttpUrl` abrufbar ist. Sie wird daher vom Controller bzw. Node mit dem entsprechenden Port versehen. Die Felder `YarnNode.NodeConnectionError` und `YarnNode.NodeDead` bilden die Komponentenfehler, wenn ein Node seine Netzwerkverbindung verliert bzw. beendet wird. Die Effekte der Komponentenfehler werden über entsprechende innere Klassen realisiert.

Die mithilfe von `YarnApp` dargestellten Anwendungen werden mithilfe des `BenchControllers` (vgl. Abschnitt 4.3) eines Clients (entsprechend repräsentiert durch die gleichnamige Klasse) gestartet. Jeder Client kann nur eine Anwendung ausführen, daher gibt es die Möglichkeit, mehrere Clients zum Starten von mehreren gleichzeitig ausgeführten Anwendungen zu nutzen. Die Anwendungen selbst enthalten neben grundlegenden Daten wie z. B. den Namen auch einige Daten zum Ressourcenbedarf (Speicher und CPU). Zwar gibt Hadoop nicht direkt die zu der Anwendung gehörigen Job-Ausführungen an, allerdings können diese mithilfe der `YarnApp.AppId` sehr einfach ermittelt werden und dann in der Liste `YarnApp.Attempts` gespeichert werden. Das Feld `YarnApp.IsKillable` gibt an, ob die Ausführung der Anwendung mit den aktuellen Daten im Modell durch den Komponentenfehler `YarnApp.KillApp` abgebrochen werden kann. Abhängig ist das durch `YarnApp.FinalStatus`, was angibt, ob eine Anwendung erfolgreich oder nicht erfolgreich ausgeführt wurde oder die Ausführung noch nicht abgeschlossen ist (durch `EFinalStatus.UNDEFINED`). Um die Komponentenfehler zu aktivieren bzw. bei Bedarf auch wieder zu deaktivieren, besitzen `YarnNode` und `YarnApp` jeweils die Eigenschaft `FaultConnector`, mit der auf den benötigten Connector zugegriffen werden kann.

Jede Ausführung `YarnAppAttempt` hat eine eigene ID und kann einer Anwendung zugeordnet werden. Genau wie bei den Anwendungen selber wird hier direkt der Node gespeichert, auf welchem der AppMstr ausgeführt wird und einen eigenen Container bildet, dessen ID direkt gespeichert wird. Container (dargestellt durch `YarnAppContainer`) existieren in Hadoop nur während der Laufzeit eines Programmes und enthalten nur wenige Daten, darunter ihr ausführender Node. Jede Anwendung, deren Ausführungen und deren Container enthalten zudem den derzeitigen Status, ob die Komponente noch initialisiert wird, bereits ausgeführt wird oder beendet ist. `EAppState.NotStartedYet` dient als Status, den es nur im Modell gibt und angibt, dass die Anwendung im späteren Verlauf der Testausführung gestartet wird.

Alle vier YARN-Kernkomponenten implementieren das Interface `IYarnReadable`, was angibt, dass die Komponente ihren Status aus Hadoop ermitteln kann. Entsprechend wird in allen Komponenten die Methode `ReadStatus()` implementiert, in welchem

mithilfe des angegebenen Parsers auf den SSH-Treiber zugegriffen werden kann und die Komponenten im Modell so ihre eigenen Daten aus dem realen Cluster ermitteln können. Da die REST-API ermöglicht, alle Daten auch über die reinen Listen zu erhalten anstatt ausschließlich über die Detailausgabe, besteht auch im Modell mithilfe der Eigenschaft `IsRequireDetailsParsing` das Ermitteln der Daten so einzustellen, dass die übergeordnete YARN-Komponente bereits alle Daten ermittelt und der Untergeordneten zum Speichern (mittels `SetStatus()`) übergibt. Als Basis dazu dient der `YarnController`, der dafür die Daten aller Anwendungen ausliest, die wiederum die Daten ihrer Ausführungen auslesen, welche dann die Daten ihrer Container auslesen und den Komponenten zum Speichern übergeben.

3.2. SSH-Treiber

Im Einführungstext zu diesem Kaptiel wurde bereits auf den grundlegenden Aufbau des Treibers eingegangen, der aus den drei einzelnen Komponenten Parser, Connector und der eigentlichen SSH-Verbindung besteht. Der Parser selbst besteht neben dem eigentlichen Parser zudem aus Datenhaltungs-Klassen für die relevanten YARN-Komponenten. Sie sind außerdem so aufgebaut, dass sie für beide hier implementierten Parser bzw. Connectoren für die Kommandozeilen-Befehle und die REST-API genutzt werden können.

3.2.1. Integration im Modell

Hadoop besitzt zwei primäre Wege, um die Daten vom RM bzw. dem TLS ausgeben zu können. Dies ist zum einen die Kommandozeile, mithilfe der die Daten vom RM und vom TLS kombiniert ausgegeben werden, und die REST-API. Die benötigten Befehle für die Kommandozeile und deren Ausgaben sind in Anhang A, die für die REST-API benötigten URLs und deren Rückgaben in Anhang B gelistet. Auf beiden Wegen können u. A. die Daten zu folgenden Komponenten ausgegeben werden [7, 13–15]:

Anwendungen als nach dem Status gefilterte Liste oder der Report einer Anwendung

Ausführungen als Liste aller Ausführungen einer Anwendung oder der Report einer Ausführung

Container als Liste aller Container einer Ausführung oder der Report eines Containers

Nodes als Liste aller Nodes oder der Report eines Nodes

Zur Integration des Treibers wurden daher entsprechende Interfaces entwickelt, über die das Modell auf den eigentlichen Treiber zugreifen kann.

Die vier Interfaces `IApplicationResult`, `IAppAttemptResult`, `IContainerResult` und `INodeResult` dienen der Übergabe der geparsten Daten der einzelnen Komponenten an die korrespondierenden Komponenten im S#-Modell. Sie enthalten jeweils alle relevanten Daten, die von Hadoop über die Kommandozeile oder die REST-API ausgegeben werden. Alle vier Interfaces implementieren zudem `IParsedComponent`, welches wiederum als Basis für die Übergabe der ausgelesenen Daten an `IYarnReadable.SetStatus()` im Modell dient.

Das Interface `IHadoopParser` dient als Einbindung des Parsers im Modell mithilfe von `IYarnReadable.Parser` und enthält für jede der acht relevanten Ausgaben von Hadoop entsprechende Methodendefinitionen.

Beim Interface `IHadoopConnector`, das im Modell den Connector über die `Fault-Connector`-Eigenschaften von `YarnApp` und `YarnNode` einbindet, besitzt ebenfalls für jede der acht Datenrückgaben entsprechende Deklarationen, für Ausführungen und Container dabei jeweils vom RM (NM für Container) und vom TLS. Auf die Nutzung des TLS zum Ermitteln der Daten zu Anwendungen wird verzichtet. Dies liegt darin begründet, dass bei Nutzung der REST-API des RM neben den vom TLS bereitgestellten Daten einige weitere Informationen zu den Anwendungen ausgegeben werden [7, 14]. Das Connector-Interface enthält darüber hinaus Deklarationen, um die im Modell implementierten Komponentenfehler im realen Cluster zu steuern und Anwendungen starten zu können. Architektonisch ist der Treiber zudem so aufgebaut, dass das Modell keine Kontrolle über den vom Parser benötigten Connector besitzt und die SSH-Verbindung ausschließlich vom Connector gesteuert werden kann.

3.2.2. Implementierte Parser

Da die Daten für die relevanten Komponenten auf zwei Arten ermittelt werden können und unterschiedliche Ausgaben erzeugen, wurden auch für beide Arten ein Parser (`CmdParser` und `RestParser`) entwickelt. Da der Parser von außerhalb keinerlei weitere Informationen erhält außer der ID der zu parsenden YARN-Komponente, ist der Parser selbst dafür verantwortlich, die Daten von einem korrespondierenden Connector zu erhalten. Daher muss zur Initialisierung eines Parsers zunächst der korrespondierende Connector initialisiert werden. Da für die Nutzung der REST-API zum Teil die IDs der übergeordneten YARN-Komponenten ebenfalls nötig sind, ist der `RestParser` zudem auch dafür verantwortlich, die entsprechenden IDs zu ermitteln, bei der Nutzung der Kommandozeile reichen aufgrund der Befehlsstruktur die IDs der Komponenten selbst.

Die konkreten Implementierungen der auf `IParsedComponent` basierenden Übergabe-Interfaces können ebenfalls als Bestandteil des Parsers angesehen werden. Sie wurden zudem so implementiert, dass sie für beide entwickelten Parser genutzt werden können.

Der grundlegende Ablauf ist bei jedem Parsing-Vorgang gleich. Zunächst werden, sofern benötigt, die benötigten YARN-Komponenten-IDs ermittelt und die Rohdaten

mithilfe des Connectors von Hadoop abgefragt. Auch vom Parser wird dabei analog zum Modell das Abrufen der Daten ausschließlich mithilfe des Interfaces `IHadoopConnector` durchgeführt. Anschließend findet das eigentliche Parsing der Ausgabe von Hadoop statt, deren Daten direkt in der für die YARN-Komponente vorgesehene `IParsedComponent`-Implementierung gespeichert werden. Da Hadoop über die Kommandozeile die Daten in keinem standardisierten Format zurückgibt, wurde das Parsing der Rohdaten von Hadoop beim `CmdParser` in eigenem Code mithilfe von *Regular Expressions* realisiert. Bei der Nutzung der REST-API werden die Daten dagegen im JSON-Format zurückgegeben [7, 14, 15], wodurch diese mithilfe des *Json.NET*-Frameworks[1] deserialisiert und direkt als die entsprechende `IParsedComponent`-Implementierung gespeichert werden. Da RM und TLS verschiedene Daten einer YARN-Komponente ausgeben, werden, sofern nötig, RM und TLS abgefragt und die dabei ermittelten Daten zusammengeführt.

Eine erste Besonderheit bildet zudem das Abrufen und Parsen der Report-Daten mittels REST-API. Da die Listen hierbei als Array der einzelnen Reports zurückgegeben werden [7, 14, 15], wird beim Parsen eines Ausführungs- oder Container-Reports die komplette Liste abgerufen und geparst. Anschließend wird in dieser Liste basierend auf der ID die benötigte Komponente herausgefiltert.

Die zweite Besonderheit bei der Nutzung der REST-API liegt darin, dass die Daten zu derzeit ausgeführten Container ausschließlich vom NM, auf dem der Container ausgeführt wird, zurückgegeben werden können [14, 15]. Daher werden zur Ermittlung der Container-Listen alle Nodes abgefragt und anschließend die benötigten Container gefiltert.

Die geparsten Daten werden abschließend als das für die YARN-Komponente vorgesehene Interface zurückgegeben, was anschließend im Modell zum Speichern der Daten genutzt werden kann.

3.2.3. Implementierte Connectoren

Für die beiden Parser wurden die beiden korrespondieren Connectoren `CmdConnector` und `RestConnector` entwickelt. Während der Connector für die REST-API nur über eine SSH-Verbindung verfügt, besteht beim Connector für die Kommandozeile die Möglichkeit, mehrere einzelne SSH-Verbindungen zu nutzen. Dies ist damit begründet, dass zum Steuern der Komponentenfehler, was nur über die Kommandozeile möglich ist, eine eigene SSH-Verbindung genutzt wird. Zum Starten von Anwendungen besteht zudem die Möglichkeit, eine beliebige Anzahl an einzelnen SSH-Verbindungen aufzubauen, damit mehrere Anwendungen parallel gestartet werden können. Da die Daten der einzelnen YARN-Komponenten in der Fallstudie bevorzugt mithilfe der REST-API ermittelt werden, kann die dafür vorgesehene SSH-Verbindung des `CmdConnector` deaktiviert werden.

[1] `https://www.newtonsoft.com/json`

Da über die Kommandozeile die Befehle für die Daten vom TLS die gleichen wie für die Daten vom RM sind [7, 13], sind beim `CmdConnector` die TLS-Methoden von geringer Bedeutung und nutzen daher ebenfalls die RM-Methoden.

Der Connector ist beim Abrufen der Daten dafür zuständig, die dafür notwendigen Befehle auszuführen. Während dies für die Kommandozeilen-Befehle die entsprechenden Hadoop-Befehle sind, wird dies zum Abrufen der Daten über die REST-API mithilfe des Tools *curl* durchgeführt. Die dabei zurückgegebenen Daten werden vom Connector ohne Verarbeitung zurückgegeben und können dann vom Parser verarbeitet werden.

Beim Steuern der Komponentenfehler wird vom Connector das für die Fallstudie entwickelte Start-Script verwendet. Nach dem eigentlichen Start bzw. Aufheben eines Komponentenfehlers wird vom Connector zudem überprüft, ob die Injizierung bzw. Aufhebung erfolgreich war. Während der Datenabruf sowie die Steuerung der Komponentenfehler synchron stattfindet, findet das Starten der Anwendungen asynchron und mithilfe des Benchmark-Scriptes statt. Da eine Ausführung einer YARN-Anwendung längere Zeit in Anspruch nehmen kann, wird dadurch die Ausführung von S# nicht behindert und es können mehrere Anwendungen parallel ausgeführt werden.

3.2.4. SSH-Verbindung

Die SSH-Verbindung selbst ist der einzige Bestandteil des Treibers, welches kein entsprechendes Interface benötigt, die SSH-Verbindung wird ausschließlich vom Connector genutzt. Realisiert wird die Verbindung mithilfe des Frameworks SSH.NET,[2] weshalb die SSH-Verbindung im Treiber nur entsprechende Funktionen zum Aufbauen, Nutzen und Beenden der Verbindung enthält.

Um die Verbindung mit dem Cluster-PC aufzubauen, ist zudem ein dort installierter SSH-Key nötig. Ein Kommando auf dem Cluster-PC kann mithilfe der Treiberkomponente synchron und asynchron ausgeführt werden.

[2] `https://github.com/sshnet/SSH.NET`

4. Implementierung der Benchmarks

Neben dem YARN-Modell selbst sind auch die während der Testausführung genutzten Anwendungen ein wichtiger Bestandteil des gesamten Testmodells. Da Hadoop selbst sowie die Plattform Hadoop-Benchmark bereits einige Anwendungen und Benchmarks enthalten, konnten diese auch im Modell genutzt werden. Dazu wurde eine Auswahl an Anwendungen in einer Markow-Kette miteinander verbunden, mit dem die Ausführungsreihenfolge der einzelnen Anwendungen basierend auf Wahrscheinlichkeiten bestimmt wird.

4.1. Übersicht möglicher Anwendungen

Hadoop-Benchmark enthält bereits die Möglichkeit, unterschiedliche Benchmarks zu starten. Wie in Unterabschnitt 2.3.1 erwähnt, sind folgende Benchmarks in der Plattform integriert:

- Hadoop Mapreduce Examples
- Intel HiBench
- SWIM

Jeder Benchmark enthält zum Starten ein jeweiliges Start-Script, mit dem ein neuer Docker-Container auf der Controller-VM gestartet wird, mit dem die Anwendungen des Benchmarks an das Cluster übergeben werden. Dass dafür jeweils eigene Docker-Container genutzt werden liegt daran, dass es in Docker-Umgebungen *best practice* ist, einen Docker-Container für nur einen Einsatzzweck zu erstellen bzw. zu nutzen. Die Hauptgründe dafür sind, dass dadurch die Skalierbarkeit erhöht und die Wiederverwendbarkeit gesteigert wird [16]. Daher wurden im Rahmen dieser Arbeit die bestehenden Startscripte der Plattform für die Benchmarks so angepasst, dass die jeweiligen Benchmarks mehrfach gleichzeitig gestartet werden können.

Die **Hadoop Mapreduce Examples** sind unterschiedliche und meist voneinander unabhängige Anwendungen, die beispielhaft für die meisten Anwendungsfälle in einem produktiv genutzten Cluster sind. Die Examples sind Teil von Hadoop und daher bei jeder Hadoop-Installation enthalten. Einige der Anwendungen der Examples sind:

- Generatoren für Text und Binärdaten, z. B. `randomtextwriter`
- Analysieren von Daten, z. B. `wordcount`
- Sortieren von Daten, z. B. `sort`
- Ausführen von komplexen Berechnungen, z. B. *Bailey-Borwein-Plouffe-Formel* zur Berechnung einzelner Stellen von π

Intel HiBench ist eine von Intel entwickelte Benchmark-Suite mit *Workloads* zu verschiedenen Anwendungszwecken mit jeweils unterschiedlichen einzelnen Anwendungen. Da in Hadoop-Benchmark noch die HiBench-Version 2.2 verwendet wird, wurde der Docker-Container von HiBench zunächst auf Version 7 aktualisiert, die einige neue Workloads und Anwendungen enthält. HiBench enthält damit folgende Workloads mit einer unterschiedlichen Anzahl an möglichen Anwendungen:

- Micro-Benchmarks (basieret auf den Mapreduce-Examples und den Jobclient-Tests)
- Maschinelles Lernen
- SQL/Datenbanken
- Websuche
- Graphen
- Streaming

SWIM ist eine Benchmark-Suite, die aus 50 verschiedenen Workloads besteht. Das besondere dabei ist, dass die dabei verwendeten Mapreduce-Jobs anhand mehrerer tausend Jobs erstellt wurden und im Vergleich zu anderen Benchmarks eine größere Vielfalt an Anwendungen und somit ein größerer Testumfang gewährleistet wird [17]. Bei der Ausführung auf dem in dieser Arbeit verwendete Cluster wurden jedoch nicht alle Workloads fehlerfrei ausgeführt. Zudem wird in [18] explizit erwähnt, dass es bei der Ausführung auf einem Cluster auf einem einzelnen PC bzw. Laptop Probleme geben kann. SWIM ist außerdem für Benchmarks eines Clusters mit mehreren physischen Nodes ausgelegt, weshalb die Ausführung in dieser Fallstudie extrem viel Zeit benötigten würde. Daher wurde die Nutzung des SWIM-Benchmarks nicht weiter verfolgt.

Ebenfalls im Installationsumfang von Hadoop enthalten sind die sog. **Jobclient-Tests**. Hauptbestandteil dieser Tests sind vor allem weitere, den Examples ergänzende, Benchmarks, welche das gesamte Cluster oder einzelne Nodes testen. Der Fokus der Jobclient-Tests liegt im Gegensatz zu den Examples nicht auf dem MapReduce- bzw. YARN-Framework, sondern beim HDFS. Da die Jobclient-Tests kein Teil von Hadoop-Benchmark sind, wurde zur Ausführung der Jobclient-Test zunächst ein eigenes Start-Script analog zur Ausführung der Mapreduce-Examples erstellt, damit hierfür ebenfalls ein eigener Docker-Container gestartet wird. Die Jobclient-Tests enthalten u. A. folgende Arten an Anwendungen:

- HDFS-Systemtests, z. B. `SilveTest`
- Reine Lastgeneratoren, z. B. `NNloadGenerator`
- Eingabe/Ausgabe-Durchsatz-Tests, z. B. `TestDFSIO`
- Dummy-Anwendungen `sleep` (blockiert Ressourcen, führt aber nichts aus) und `fail` (Anwendung schlägt immer fehl)

4.2. Auswahl der verwendeten Anwendungen

Damit die Fallstudie die Realität abbilden kann, wurden von allen verfügbaren Anwendungen einige ausgewählt und in ein Transitionssystem in Form einer Markow-Kette überführt. Diese Kette definiert die Ausführungsreihenfolge zwischen den einzelnen Anwendungen. Eine zufallsbasierte Markow-Kette wurde aus dem Grund verwendet, dass auch in der Realität Anwendungen nicht immer in der gleichen Reihenfolge ausgeführt werden und daher auch in der Fallstudie eine unterschiedliche Ausführungsreihenfolge der Anwendungen gewährleistet werden soll. Mithilfe der Festlegung eines bestimmten Seeds für den in der Fallstudie benötigten Pseudo-Zufallsgenerator besteht bei Bedarf dennoch die Möglichkeit, einen Test mit den gleichen Anwendungen wiederholen zu können.

Da alle verfügbaren Anwendungen zunächst je nach Anwendungsart in verschiedene Kategorien eingeteilt wurden, wurden bei der Erstellung des Transitionssystems einige dieser Kategorien exemplarisch für mögliche, typische Anwendungsfälle auf einem produktiv genutzten Cluster verwendet. Folgende Kategorien und Anwendungen der Mapreduce-Examples und Jobclient-Tests wurden daher letztlich ausgewählt:

- Generatoren für
  - Textdateien: `randomtextwriter` (rtw) und `TestDFSIO -write` (dfs-w)
  - Binärdateien: `randomwriter` (rw) und `teragen` (tgen)
- Datenverarbeitung in Form von:
  - Auslesen: `wordcount` (wc) und `TestDFSIO -read` (dfs-r)
  - Sortieren: `sort` für Textdaten und `terasort` (tsort) für Binärdaten
  - Validieren: `testmapredsort` (tstsort) und `teravalidate` (tval) für die jeweiligen Sortier-Anwendungen
- Ausführen von Berechnungen:
  - `pi`: Quasi-Monte-Carlo-Methode zur einfachen Berechnung von π
  - `pentomino` (pent): Berechnung von Pentomino-Problemen
- Dummy-Anwendungen: `sleep` und `fail`

Der Grund für die Berücksichtigung von mehreren gleichen bzw. ähnlichen Anwendungen für einige Kategorien liegt darin, dass eine Anwendung für eine eher kurze Ausführung, die andere für eine umfangreiche ausgewählt wurde. So stehen die beiden `TestDFSIO`-Varianten für eine umfangreichere Datennutzung, während die jeweils anderen Anwendungen einen kleineren Umfang repräsentieren. Ähnlich verhält es sich bei den beiden Berechnungs-Anwendungen, bei denen die `pentomino`-Anwendung die deutlich umfangreicheren Berechnungen durchführt. `TestDFSIO` enthält zudem die Möglichkeit, Daten zu genieren und zu lesen, weshalb diese Anwendung in zwei Kategorien verwendet

	dfs-w	rtw	tgen	dfs-r	wc	rw	sort	tsort	pi	pent	tstsort	tval
dfs-w		20	0	40	0	0	0	0	20	20	0	0
rtw	10		0	0	40	10	30	0	10	0	0	0
tgen	0	10		0	0	0	0	70	0	20	0	0
dfs-r	0	20	0		0	10	0	0	40	30	0	0
wc	20	30	0	0		0	20	0	20	10	0	0
rw	0	20	20	0	0		0	0	30	30	0	0
sort	0	20	10	0	20	10		0	20	0	20	0
tsort	0	0	0	0	0	0	0		30	20	0	50
pi	40	30	0	0	0	0	0	0		30	0	0
pent	30	30	0	0	0	20	0	0	20		0	0
tstsort	0	40	0	0	0	20	0	0	10	30		0
tval	20	30	0	0	0	0	0	0	30	20	0	

Tabelle 4.1.: Verwendete Markov-Kette für die Anwendungs-Übergänge in Tabellenform, ohne Selbst-Transitionen und Dummy-Anwendungen, Werte in Prozent

wurde. Haupteinsatzzweck der Anwendung liegt vor allem darin, den Datendurchsatz des HDFS zu testen.

Eine Besonderheit bilden die beiden Dummy-Anwendungen. Beide werden in dieser Fallstudie dafür genutzt, um zu simulieren, wenn eine Anwendung auf externe Daten warten muss bzw. ein unerwarteter Fehler während der Ausführung auftaucht. Daher können beide Anwendungen unabhängig von der derzeit ausgeführten Anwendung als nachfolgende Anwendung ausgewählt werden und haben ihrerseits als nachfolgende Anwendung immer die zuvor ausgeführte Anwendung. Beide Anwendungen sind aus diesem Grund zwar im implementierten Transitionssystem enthalten, jedoch nicht in der darauf zugrundeliegenden Markow-Kette.

Für die Markow-Kette der Übergänge zwischen den Anwendungen wurde berücksichtigt, welche Anwendungen bestimmte Eingabedaten benötigen, sodass einige Anwendungen nur dann Ausgeführt werden können, sobald eine andere Anwendung direkt zuvor die benötigten Eingabedaten bereitgestellt hat. Andere Anwendungen können dagegen fast jederzeit ausgeführt werden. In der in Tabelle 4.1 dargestellten Markow-Kette wird zudem ersichtlich, dass alle Selbst-Transitionen sowie die Transitionen für die beiden Dummy-Anwendungen fehlen. Dies liegt darin, dass Selbst-Transitionen für jede Anwendung pauschal auf 60 Prozent festgelegt wurden, sodass das hier dargestellte Transitionssystem nur in 40 Prozent der möglichen Zustandswechsel zum Einsatz kommt. Ebenso sind die Übergänge in die beiden Dummy-Anwendungen mit jeweils fünf Prozent definiert, weshalb es im Falle eines Anwendungswechsels eine gesamte Wahrscheinlichkeit von 110 Prozent gibt, die in der Implementierung jedoch auf 100 Prozent normalisiert ist. Zudem beschränken sich die Zustandsübergänge ausgehend von den Dummy-Anwendungen jeweils auf eine Selbst-Transition sowie auf die Rückkehr zur zuvor ausgeführten Anwendung mit jeweils 50 Prozent.

4.3. Implementierung der Anwendungen im Modell

Die Verwaltung der auszuführenden Benchmarks wurde komplett vom restlichen YARN-Modell getrennt. Verbunden sind beide durch die Eigenschaft `Client.BenchController`, das den vom Client verwendeten `BenchmarkController` enthält, der zur Verwaltung der auszuführenden Anwendung dient. Der Controller besteht aus zwei wesentlichen Teilen, einem statischen und einem dynamischen.

Der **statische Teil** des Controllers definiert die möglichen Anwendungen sowie das im Abschnitt zuvor definierte und in Tabelle 4.1 dargestellte Transitionssystem. Die einzelnen Anwendungen werden mithilfe der Klasse `Benchmark` repräsentiert, in der die benötigten Informationen wie z. B. der Befehl zum Starten der Anwendung definiert werden. Da mehrere Clients unabhängig voneinander agieren können müssen, erhält jeder Client zudem ein eigenes Unterverzeichnis im HDFS, in dem sich die Ein- und Ausgabeverzeichnisse für die von ihm gestarteten Anwendungen befinden. Das muss auch bei der Definition der Startbefehle der Anwendungen berücksichtigt werden, weshalb in Listing 4.1 entsprechende Platzhalter vorhanden sind. Aus diesem Grund muss vor dem Start der Anwendung mithilfe der Methode `GetStartCmd()` der Startbefehl generiert werden, indem der zu startende Client das in `Client.ClientDir` gespeicherte Client-Basisverzeichnis übergibt. Da einige Anwendungen zudem voraussetzen, dass das genutzte Ausgabe-Verzeichnis noch nicht im HDFS existiert, muss das Verzeichnis vor dem Anwendungsstart gelöscht werden.

Jede Anwendung erhält zudem eine eigene ID, die mit ihrem Index im Array `BenchmarkController.Benchmarks` übereinstimmt. Diese wird bei der in Listing 4.2 dargestellte Auswahl der nachfolgenden Anwendung benötigt, um innerhalb des gesamten Transitionssystems in `BenchmarkController.BenchTransitions` die Wahrscheinlichkeiten für die Wechsel von der derzeitigen Anwendung zu anderen Anwendungen auszuwählen.

Der **dynamische Teil** des Controllers ist für die Auswahl der auszuführenden Anwendung zuständig, was auch die Auswahl der initial auszuführenden Anwendung einschließt. Als initiale Anwendung wird zufällig, und mit jeweils gleicher Wahrscheinlichkeit, eine der sechs Anwendungen ausgeführt, die keine Eingabedaten benötigen bzw. diese für andere Anwendungen generieren:

- `TestDFSIO -write`
- `randomtextwriter`
- `teragen`
- `randomwriter`
- `pi`
- `pentomino`

```
1  public class Benchmark
2  {
3    public const string BaseDirHolder = "$DIR";
4    public const string OutDirHolder = "$OUT";
5    public const string InDirHolder = "$IN";
6
7    public Benchmark(int id, string name, string startCmd, string
         outputDir, string inputDir)
8    {
9      _StartCmd = startCmd;
10     _InDir = inputDir;
11     HasInputDir = true;
12   }
13
14   public string GetStartCmd(string clientDir = "")
15   {
16     var result = _StartCmd.Replace(OutDirHolder, GetOutputDir(
           clientDir)).Replace(InDirHolder, GetInputDir(clientDir));
17     if(result.Contains(BaseDirHolder))
18       result = ReplaceClientDir(result, clientDir);
19     return result;
20   }
21 }
22
23 using static Benchmark;
24 public class BenchmarkController
25 {
26
27   public static Benchmark[] Benchmarks { get; } // benchmarks
28   public static int[][] BenchTransitions { get; } // transitions
29
30   static BenchmarkController()
31   {
32     Benchmarks = new[]
33     {
34       new Benchmark(04, "wordcount", $"example␣wordcount␣{InDirHolder}
             ␣{OutDirHolder}", $"{BaseDirHolder}/wcout", $"{BaseDirHolder
             }/rantw"),
35     };
36   }
37 }
38
39 public class Client : Component
40 {
41   public string StartBenchmark(Benchmark benchmark)
42   {
43     if(benchmark.HasOutputDir)
44       SubmittingConnector.RemoveHdfsDir(benchmark.GetOutputDir(
             ClientDir));
45     var appId = SubmittingConnector.StartApplicationAsync(benchmark.
           GetStartCmd(ClientDir));
46   }
47 }
```

Listing 4.1: Definition und Start einer Anwendung (gekürztes Beispiel). Die Generierung des komplettes Startbefehls mit Nutzung des Benchmark-Scriptes führt der vom Client verwendete Connector durch, weshalb hier nur definiert werden muss, dass das Example-Programm `wordcount` gestartet wird.

```
// get probabilities from current benchmark
var transitions = BenchTransitions[CurrentBenchmark.Id];

var ranNumber = RandomGen.Next(_TransitionCumulatedProbability);
var cumulative = 0;
for(int i = 0; i < transitions.Length; i++)
{
  cumulative += transitions[i];
  if(ranNumber >= cumulative)
    continue;

  // save benchmarks
  PreviousBenchmark = CurrentBenchmark;
  CurrentBenchmark = Benchmarks[i];
}
```

Listing 4.2: Normalisierung und Auswahl der nachfolgenden Anwendung (gekürzt)

Das im vorherigen Abschnitt definierte und im statischen Teil implementierte Transitionssystem kommt immer dann zum Einsatz, wenn Entschieden werden muss, welche Anwendung der derzeit ausgeführten Anwendung folgt. Jeder Client bzw. sein `BenchmarkController` entscheidet unabhängig von anderen Clients einmal pro S#-Takt, welche Anwendung ausgeführt wird. Dabei gibt es folgende drei Fälle, die zum Teil im vorherigen Abschnitt bereits erläutert wurden:

Dummy-Anwendungen Wenn derzeit eine Dummy-Anwendung ausgeführt wird, wird mit einer Wahrscheinlichkeit von 50 Prozent die zuvor ausgeführte Anwendung ausgewählt, andernfalls wird die Dummy-Anwendung erneut ausgeführt.

Selbst-Transitionen Wenn derzeit keine Dummy-Anwendung ausgeführt wird, wird die Selbst-Transition mit einer Wahrscheinlichkeit von 60 Prozent ausgewählt, sodass die Anwendung erneut ausgeführt wird.

Transition zwischen Anwendungen Trifft keiner der anderen Fälle zu, wird basierend auf dem Transitionssystem die nachfolgende Anwendung ausgewählt.

Das im dritten Fall zum Einsatz kommende Transitionssystem stellt sich zum einen aus der Markow-Kette in Tabelle 4.1, sowie möglicher Übergänge zu den beiden Dummy-Anwendungen mit einem Wert von jeweils 5 zusammen. Listing 4.2 zeigt, wie bei der Auswahl des Benchmarks die Werte normalisiert und anschließend die nachfolgende Anwendung ausgewählt wird. Die vom Zufallsgenerator ausgewählte Zahl liegt zwischen 0 und `_TransitionCumulatedProbability`, das hier entsprechend den Wert 110 hat.

Wenn eine neue Anwendung ausgewählt wurde, muss zunächst sichergestellt werden, dass die bisher ausgeführte Anwendung beendet ist. Dafür wird der in Anhang A dargestellte Befehl von Hadoop zum Abbruch von Anwendungen ausgeführt. Im Anschluss kann das von der neuen Anwendung benötigte HDFS-Ausgabeverzeichnis gelöscht werden, bevor die Anwendung selbst gestartet wird.

Eine Anwendung wird wie in Listing 4.1 gezeigt zwar asynchron gestartet, allerdings wird zunächst noch synchron auf die Ausgabe der `applicationId` gewartet. Die gesamte Ausgabe einer zu startenden Anwendung ist ebenfalls in Anhang A zu finden. Die ID wird vom Cluster im Rahmen der Übergabe und Initialisierung der Anwendung vergeben. Erst nachdem diese bekannt ist, wird die restliche Ausführung der Anwendung asynchron durchgeführt. Benötigt wird die ID damit der zu startende Client die Anwendung im Falle eines Anwendungswechsels in den folgenden Takten beenden kann. Ohne die direkte Speicherung der ID wäre es sonst nicht möglich, klar entscheiden zu können, welchem Client die Anwendung zugeordnet ist. Dies ist auch der Grund, weshalb kein HiBench-Workload in das Transitionssystem aufgenommen wurde, da hier die `applicationId` gemeinsam mit der gesamten Ausgabe der einzelnen HiBench-Anwendungen nach Abschluss der Ausführung ausgegeben wird. Gespeichert wird die ID zunächst in einer noch verfügbaren `YarnApp`-Instanz, welche anschließend selbst in `Client.CurrentExecutingApp` gespeichert wird.

Literatur

[1] M. Polo u. a. „Test Automation“. In: *IEEE Software* 30.1 (Jan. 2013), S. 84–89. ISSN: 0740-7459. DOI: 10.1109/MS.2013.15.

[2] Orna Grumberg, EM Clarke und DA Peled. „Model checking“. In: (1999).

[3] A. Habermaier u. a. „Runtime Model-Based Safety Analysis of Self-Organizing Systems with S#“. In: *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. Sep. 2015, S. 128–133. DOI: 10.1109/SASOW.2015.26.

[4] Bo Zhang u. a. „Self-Balancing Job Parallelism and Throughput in Hadoop“. In: *16th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS)*. Hrsg. von Márk Jelasity und Evangelia Kalyvianaki. Bd. LNCS-9687. Distributed Applications and Interoperable Systems. Heraklion, Crete, Greece: Springer, Juni 2016, S. 129–143. DOI: 10.1007/978-3-319-39577-7_11. URL: https://hal.inria.fr/hal-01294834.

[5] Apache Software Foundation. *Welcome to Apache™ Hadoop®!* 18. Dez. 2017. URL: https://hadoop.apache.org/ (besucht am 27. 12. 2017).

[6] Apache Software Foundation. *Apache Hadoop NextGen MapReduce (YARN)*. 29. Juni 2015. URL: https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html (besucht am 27. 12. 2017).

[7] Apache Software Foundation. *The YARN Timeline Server*. 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/TimelineServer.html (besucht am 27. 01. 2018).

[8] Apache Software Foundation. *HDFS Architecture*. 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html (besucht am 27. 12. 2017).

[9] Apache Software Foundation. *MapReduce Tutorial*. 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html (besucht am 02. 01. 2018).

[10] Apache Software Foundation. *MapReduce NextGen aka YARN aka MRv2*. 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/index.html (besucht am 02. 01. 2018).

[11] Apache Software Foundation. *Hadoop: Capacity Scheduler*. 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html (besucht am 21. 01. 2018).

[12] Filip Krikava. *Architecture*. 23. Jan. 2017. URL: https://github.com/Spirals-Team/hadoop-benchmark/blob/b32711e3a724e7183e4f52ba76e34f2e587a523a/README.md (besucht am 22. 01. 2018).

[13] Apache Software Foundation. *YARN Commands*. 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YarnCommands.html (besucht am 08. 02. 2018).

[14] Apache Software Foundation. *ResourceManager REST API's.* 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/ResourceManagerRest.html (besucht am 08. 02. 2018).

[15] Apache Software Foundation. *NodeManager REST API's.* 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/NodeManagerRest.html (besucht am 08. 02. 2018).

[16] Docker Inc. *Best practices for writing Dockerfiles.* URL: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (besucht am 09. 03. 2018).

[17] Yanpei Chen; Sara Alspaugh; Archana Ganapathi; Rean Griffith; Randy Katz. *SWIM Wiki: Home.* 12. Juni 2016. URL: https://github.com/SWIMProjectUCB/SWIM/wiki (besucht am 10. 03. 2018).

[18] Bo Zhang. *Tutorial.* 10. März 2017. URL: https://github.com/Spirals-Team/hadoop-benchmark/wiki/Tutorial (besucht am 21. 11. 2017).

A. Kommandozeilen-Befehle von Hadoop

Für jede der vier relevanten YARN-Komponenten können die Daten jeweils als Liste oder als ausführlicher Report ausgegeben werden. Im Folgenden sind Beispielhaft die dafür notwendigen Befehle für Anwendungen aufgelistet, für Ausführungen, Container und Nodes sind analoge Befehle verfügbar. Neben den Monitoring-Befehlen sind auch einige weitere für diese Arbeit relevante Befehle mit ihren Ausgaben aufgelistet. Die Ausgaben zu den Befehlen sind hier zudem auf das wesentliche gekürzt, u. A. da Hadoop bei einigen Befehlen ausgibt, über welche Services (in Listing A.1 z. B. TLS, RM und *Application History Server*) die Daten ermittelt werden. Weiterführende Informationen zu den einzelnen Befehlen sind in der dazugehörigen Dokumentation in [13] zu finden.

Listing A.1: CMD-Ausgabe der Anwendungsliste. Anwendungen können mithilfe der Optionen `--appTypes` und `--appStates` gefiltert werden.

```
$ yarn application --list --appStates ALL
18/02/08 15:37:51 INFO impl.TimelineClientImpl: Timeline service
   address: http://0.0.0.0:8188/ws/v1/timeline/
18/02/08 15:37:51 INFO client.RMProxy: Connecting to ResourceManager
   at controller/10.0.0.3:8032
18/02/08 15:37:51 INFO client.AHSProxy: Connecting to Application
   History server at /0.0.0.0:10200
Total number of applications (application-types: [] and states: [NEW,
   NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED
   ]):1
Application-Id  Application-Name    Application-Type    User    Queue
      State   Final-State Progress    Tracking-URL
application_1518100641776_0001  QuasiMonteCarlo MAPREDUCE   root
   default FINISHED    SUCCEEDED   100%    http://controller:19888/
   jobhistory/job/job_1518100641776_0001
```

Listing A.2: CMD-Ausgabe des Reports einer Anwendung

```
$ yarn application --status application_1518100641776_0001
[...]
Application Report :
    Application-Id : application_1518100641776_0001
    Application-Name : QuasiMonteCarlo
    Application-Type : MAPREDUCE
    User : root
    Queue : default
    Start-Time : 1518103712160
    Finish-Time : 1518103799743
    Progress : 100%
    State : FINISHED
    Final-State : SUCCEEDED
    Tracking-URL : http://controller:19888/jobhistory/job/
       job_1518100641776_0001
    RPC Port : 41309
    AM Host : compute-1
    Aggregate Resource Allocation : 1075936 MB-seconds, 942 vcore-
       seconds
    Diagnostics :
```

Listing A.3: Starten einer Anwendung in HadoopBenchmark. Hier mit dem Mapreduce Example `pi` und dem Abbruch der Anwendung durch den in Listing A.4 gezeigten Befehl. Die `applicationId` ist hier in Zeile 13 enthalten.

```
1  $ hadoop-benchmark/benchmarks/hadoop-mapreduce-examples/run.sh pi 20
       1000
2  Number of Maps  = 20
3  Samples per Map = 1000
4  Wrote input for Map #0
5  [...]
6  Starting Job
7  18/03/14 13:06:26 INFO impl.TimelineClientImpl: Timeline service
       address: http://0.0.0.0:8188/ws/v1/timeline/
8  18/03/14 13:06:27 INFO client.RMProxy: Connecting to ResourceManager
       at controller/10.0.0.3:8032
9  18/03/14 13:06:27 INFO client.AHSProxy: Connecting to Application
       History server at /0.0.0.0:10200
10 18/03/14 13:06:27 INFO input.FileInputFormat: Total input paths to
       process : 20
11 18/03/14 13:06:27 INFO mapreduce.JobSubmitter: number of splits:20
12 18/03/14 13:06:27 INFO mapreduce.JobSubmitter: Submitting tokens for
       job: job_1520342317799_0002
13 18/03/14 13:06:28 INFO impl.YarnClientImpl: Submitted application
       application_1520342317799_0002
14 18/03/14 13:06:28 INFO mapreduce.Job: The url to track the job: http
       ://controller:8088/proxy/application_1520342317799_0002/
15 18/03/14 13:06:28 INFO mapreduce.Job: Running job:
       job_1520342317799_0002
16 18/03/14 13:06:34 INFO mapreduce.Job: Job job_1520342317799_0002
       running in uber mode : false
17 18/03/14 13:06:34 INFO mapreduce.Job:  map 0% reduce 0%
18 18/03/14 13:06:58 INFO mapreduce.Job:  map 20% reduce 0%
19 18/03/14 13:06:59 INFO mapreduce.Job:  map 60% reduce 0%
20 18/03/14 13:07:03 INFO mapreduce.Job:  map 0% reduce 0%
21 18/03/14 13:07:03 INFO mapreduce.Job: Job job_1520342317799_0002
       failed with state KILLED due to: Application killed by user.
22 18/03/14 13:07:03 INFO mapreduce.Job: Counters: 0
23 Job Finished in 37.53 seconds
```

Listing A.4: Vorzeitiges Beenden einer Anwendung. Hier wird die in Listing A.3 gestartete Anwendung vorzeitig beendet.

```
1  $ yarn application -kill application_1520342317799_0002
2  [...]
3  Killing application application_1520342317799_0002
4  18/03/14 13:07:02 INFO impl.YarnClientImpl: Killed application
       application_1520342317799_0002
```

B. REST-API von Hadoop

Wie bei der Ausgabe der Daten der YARN-Komponenten über die Kommandozeile können auch bei der Ausgabe mithilfe der REST-API die Daten als Liste oder als einzelner Report ausgegeben werden. Der Unterschied zur Kommandozeile liegt jedoch darin, dass die Listenausgaben einem Array der einzelnen Reports entsprechen. Neben der hier verwendeten Ausgabe im JSON-Format unterstützt Hadoop auch eine Ausgabe im XML-Format. Im Folgenden sind daher Beispielhaft die Ausgaben im JSON-Format für die Anwendungsliste vom RM und für Ausführungen vom TLS aufgeführt. Im Rahmen dieser Masterarbeit relevant waren vom RM die Rückgaben der Listen für Anwendungen, Ausführungen, Container (jedoch vom NM) und der Nodes. Vom TLS relevant waren die Listen für Ausführungen und Container. Weitere Informationen zu den hier verwendeten Nutzungsmöglichkeiten sind in der dazugehörigen Dokumentation in [7, 14, 15] zu finden.

Listing B.1: REST-Ausgabe aller Anwendungen vom RM. Die Liste kann mithilfe verschiedener Query-Parameter gefiltert werden.
URL: `http://<rmhttpaddress:port>/ws/v1/cluster/apps`


```
{
  "apps": {
    "app": [
      {
        "id": "application_1518429920717_0001",
        "user": "root",
        "name": "QuasiMonteCarlo",
        "queue": "default",
        "state": "FINISHED",
        "finalStatus": "SUCCEEDED",
        "progress": 100,
        "trackingUI": "History",
        "trackingUrl": "http://controller:8088/proxy/
          application_1518429920717_0001/",
        "diagnostics": "",
        "clusterId": 1518429920717,
        "applicationType": "MAPREDUCE",
        "applicationTags": "",
        "startedTime": 1518430260179,
        "finishedTime": 1518430404123,
        "elapsedTime": 143944,
        "amContainerLogs": "http://compute-2:8042/node/
          containerlogs/
          container_1518429920717_0001_01_000001/root",
        "amHostHttpAddress": "compute-2:8042",
        "allocatedMB": -1,
        "allocatedVCores": -1,
        "runningContainers": -1,
        "memorySeconds": 1756786,
        "vcoreSeconds": 1546,
        "preemptedResourceMB": 0,
        "preemptedResourceVCores": 0,
        "numNonAMContainerPreempted": 0,
        "numAMContainerPreempted": 0
      }
    ]
  }
}
```

Listing B.2: REST-Ausgabe aller Ausführungen einer Anwendung vom RM.
URL: `http://<rmhttpaddress:port>/ws/v1/cluster/apps/{appid}/appattempts`


```
{
  "appAttempts": {
    "appAttempt": [
      {
        "id": 1,
        "startTime": 1518430260521,
        "containerId": "
          container_1518429920717_0001_01_000001",
        "nodeHttpAddress": "compute-2:8042",
        "nodeId": "compute-2:45454",
        "logsLink": "//compute-2:8042/node/containerlogs/
          container_1518429920717_0001_01_000001/root"
      }
    ]
  }
}
```