

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: 12. März 2018

Betreuer: M.Sc. Benedikt Eberhardinger

1. Prüfer: Prof. Dr. X

2. Prüfer: Prof. Dr. Y

Zusammenfassung

Abstract

Inhaltsverzeichnis

Zusammenfassung	I
Abstract	II
Verzeichnisse	IV
Abbildungsverzeichnis	IV
Listings	IV
Tabellenverzeichnis	IV
Abkürzungsverzeichnis	IV
1. Einleitung	1
2. Aufbau der Fallstudie	2
2.1. Apache Hadoop	2
2.2. Adaptive Komponente in Hadoop	4
2.3. Umsetzung des realen Clusters	6
2.3.1. Plattform Hadoop-Benchmark	6
2.3.2. Anpassungen und Setup	7
3. Aufbau des Modells	9
3.1. YARN-Modell	9
3.2. SSH-Treiber	12
3.2.1. Integration im Modell	12
3.2.2. Implementierte Parser	13
3.2.3. Implementierte Connectoren	14
3.2.4. SSH-Verbindung	15
4. Implementierung der Benchmarks	16
4.1. Übersicht möglicher Anwendungen	16
4.2. Auswahl der verwendeten Anwendungen	18
4.3. Implementierung der Anwendungen im YARN-Modell	19
Literatur	20
A. Kommandozeilen-Befehle von Hadoop	22
B. REST-API von Hadoop	24

Verzeichnisse

Abbildungsverzeichnis

2.1. Architektur von YARN	3
2.2. Architektur des HDFS	4
2.3. LoJP und LoJT in Hadoop	5
2.4. High-Level-Architektur von Hadoop-Benchmark	6
3.1. Grundlegende Architektur des Gesamtmodells	9
3.2. Aufbau des YARN-Modells	10

Listings

A.1. CMD-Ausgabe der Anwendungsliste	23
A.2. CMD-Ausgabe des Reports einer Anwendung	23
B.1. REST-Ausgabe aller Anwendungen vom RM	25
B.2. REST-Ausgabe aller Ausführungen einer Anwendung vom RM	26

Tabellenverzeichnis

2.1. Spezifikationen der verwendeten PCs und Windows-VM	7
4.1. Verwendete Markov-Kette für die Anwendungs-Übergänge in Tabellenform	19

Abkürzungsverzeichnis

AM	ApplicationManager
AppMstr	ApplicationMaster
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. JUnit¹ für Java und NUnit² 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 Benutzer gleichzeitig ausgeführt wird und teilt ihnen so viele Ressourcen zu, wie benötigt werden bzw. der Benutzer 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 genannt, 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

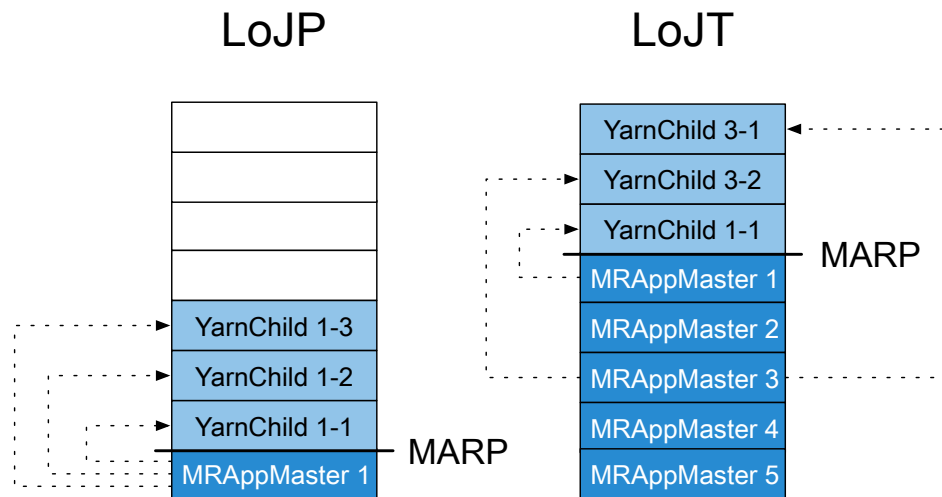


Abbildung 2.3.: LoJP und LoJT in Hadoop (entnommen aus [4])

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. In 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.

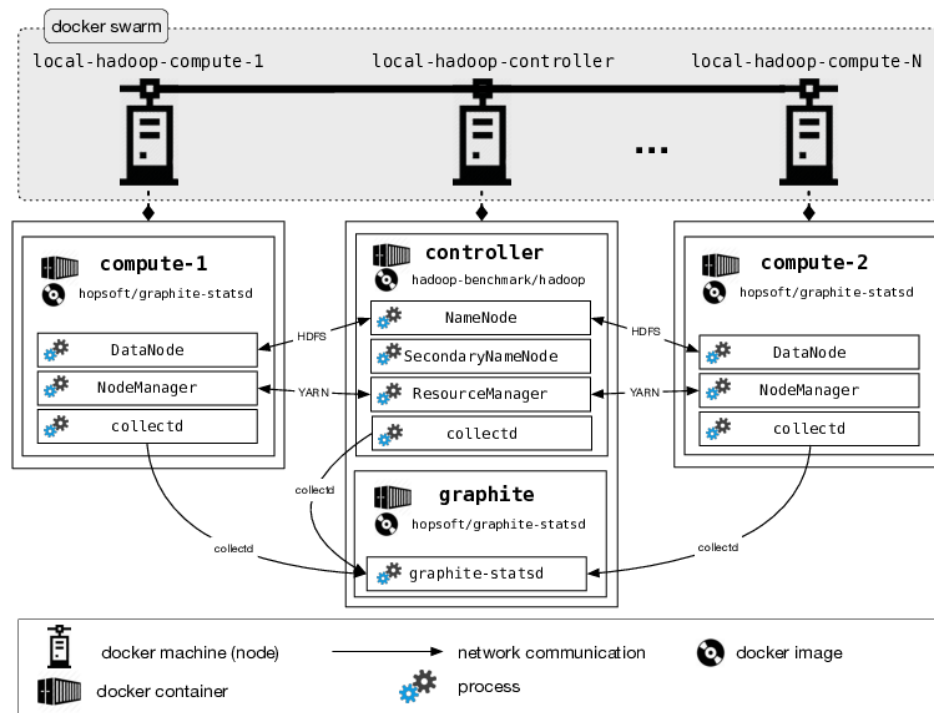


Abbildung 2.4.: High-Level-Architektur von Hadoop-Benchmark [12]

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	≤ 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 HiBench⁶
- 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 5.2⁸, 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:

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

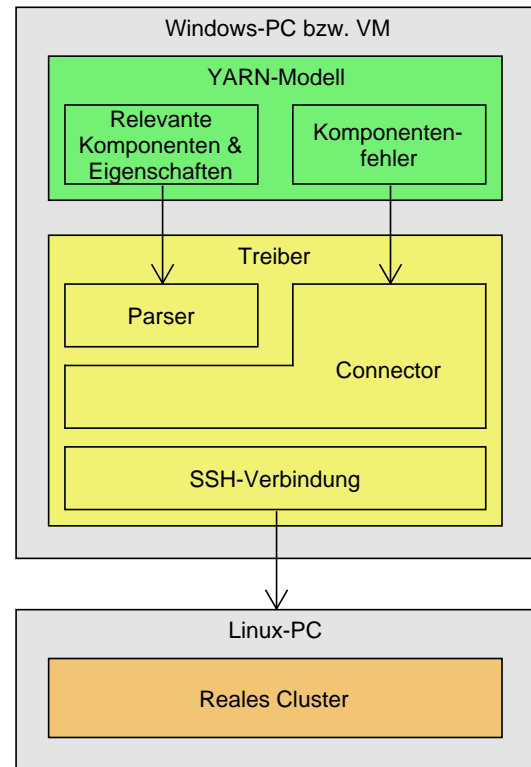


Abbildung 3.1.: Grundlegende Architektur des Gesamtmodells

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.

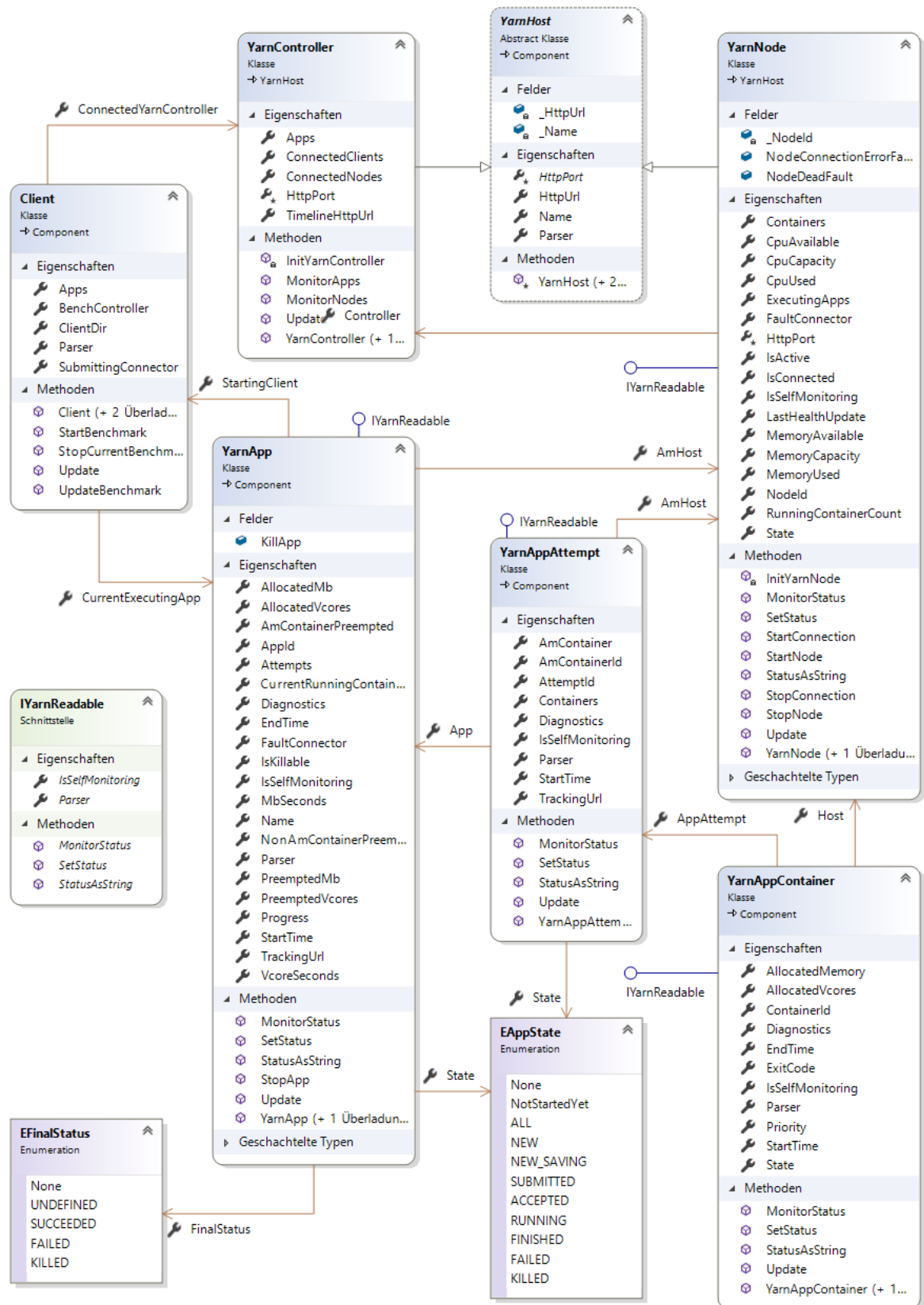


Abbildung 3.2.: Aufbau des YARN-Modells. Das Modell wurde mithilfe des Klassendiagramm-Designers in Visual Studio 2017 visualisiert. Daher werden Assoziationen mit höherer Multiplizität als 1, die daher mithilfe von `List<T>` umgesetzt wurden (z. B. `YarnApp.Attempts`) im Diagramm nicht als Assoziationen 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 Anwendungen, welche mit **YarnApp** dargestellt werden, werden mithilfe des **Client** gestartet, was den zu startenden Client darstellt. 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 er-

halten 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*-Frameworks¹ 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 geparkt. 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 geparkten 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 korrespondierenden 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.

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.

¹<https://www.newtonsoft.com/json>

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

Ebenfalls ein wichtiger Bestandteil des gesamten Modells sind die während der Testausführung genutzten Anwendungen und Benchmarks. 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 der Benchmark auf dem Cluster gestartet wird. Dass dafür ein eigener Docker-Container genutzt wird liegt daran, dass es in Docker-Umgebungen *best practice* ist, einen Docker-Container für nur einen Einsatzzweck zu erstellen und zu nutzen. Die Hauptgründe dafür sind, dass dadurch die Skalierbarkeit erhöht und die Wiederverwendbarkeit gesteigert werden [16]. Daher wurden auch die Startscripte für die Benchmarks so angepasst, dass die jeweiligen Benchmarks mehrfach gestartet werden können.

Die **Hadoop Mapreduce Examples** sind unterschiedliche und meist voneinander unabhängige Anwendungen, die beispielhaft für die meisten Anwendungsfällen 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 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 neuen 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. Zudem ist SWIM für Benchmarks eines Clusters mit mehreren physischen Nodes ausgelegt ist und daher würde die Ausführung in dieser Fallstudie extrem viel Zeit benötigen. 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 Benchmarks, welche das gesamte Cluster oder einzelne Nodes testen, wobei die meisten Anwendungen das HDFS betreffen. 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**
- 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, die gleichen Anwendungen erneut ausführen 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` (ttsort) und `teravalidate` (tval) für die jeweiligen Sortierungs-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 steht `TestDFSIO` für eine umfangreichere Datennutzung, während die jeweils anderen Anwendungen einen kleineren Umfang repräsentieren. Ähnlich verhält es sich bei den beiden Berechnungs-Anwendungen. `TestDFSIO` enthält zudem die Möglichkeit, Daten zu generieren und zu lesen, weshalb diese Anwendung in zwei Kategorien verwendet wurde.

Eine Besonderheit bilden die beiden Dummy-Anwendungen. Beide wurden dafür genutzt, wenn eine Anwendung auf externe Daten warten muss bzw. ein unerwarteter

	dfs-w	rtw	tgen	dfs-r	wc	rw	sort	tsort	pi	pent	ttsort	tval
dfs-w	0	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
ttsort	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

Fehler auftaucht, weshalb sie unabhängig von der derzeit ausgeführten Anwendung als nachfolgende Anwendung ausgewählt werden können. Daher sind die beiden 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 skaliert 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 YARN-Modell

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 ApacheTMHadoop®!* 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. Die Ausgaben zu den Befehlen sind hier zudem auf das wesentliche gekürzt, u. A. da Hadoop bei jedem Befehl zunächst 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 [13] zu finden.

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

```

1 $ yarn application --list --appStates ALL
2 18/02/08 15:37:51 INFO impl.TimelineClientImpl: Timeline service
   address: http://0.0.0.0:8188/ws/v1/timeline/
3 18/02/08 15:37:51 INFO client.RMProxy: Connecting to ResourceManager
   at controller/10.0.0.3:8032
4 18/02/08 15:37:51 INFO client.AHSPProxy: Connecting to Application
   History server at /0.0.0.0:10200
5 Total number of applications (application-types: [] and states: [NEW,
   NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED
   ]):1
6 Application-Id  Application-Name      Application-Type    User      Queue
   State      Final-State Progress      Tracking-URL
7 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

```

1 $ yarn application --status application_1518100641776_0001
2 [...]
3 Application Report :
4   Application-Id : application_1518100641776_0001
5   Application-Name : QuasiMonteCarlo
6   Application-Type : MAPREDUCE
7   User : root
8   Queue : default
9   Start-Time : 1518103712160
10  Finish-Time : 1518103799743
11  Progress : 100%
12  State : FINISHED
13  Final-State : SUCCEEDED
14  Tracking-URL : http://controller:19888/jobhistory/job/
   job_1518100641776_0001
15  RPC Port : 41309
16  AM Host : compute-1
17  Aggregate Resource Allocation : 1075936 MB-seconds, 942 vcore-
   seconds
18  Diagnostics :

```

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 [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`

```
1 {
2   "apps": {
3     "app": [
4       {
5         "id": "application_1518429920717_0001",
6         "user": "root",
7         "name": "QuasiMonteCarlo",
8         "queue": "default",
9         "state": "FINISHED",
10        "finalStatus": "SUCCEEDED",
11        "progress": 100,
12        "trackingUI": "History",
13        "trackingUrl": "http://controller:8088/proxy/
14          application_1518429920717_0001/",
15        "diagnostics": "",
16        "clusterId": 1518429920717,
17        "applicationType": "MAPREDUCE",
18        "applicationTags": "",
19        "startedTime": 1518430260179,
20        "finishedTime": 1518430404123,
21        "elapsedTime": 143944,
22        "amContainerLogs": "http://compute-2:8042/node/
23          containerlogs/
24          container_1518429920717_0001_01_000001/root",
25        "amHostHttpAddress": "compute-2:8042",
26        "allocatedMB": -1,
27        "allocatedVCores": -1,
28        "runningContainers": -1,
29        "memorySeconds": 1756786,
30        "vcoreSeconds": 1546,
31        "preemptedResourceMB": 0,
32        "preemptedResourceVCores": 0,
33        "numNonAMContainerPreempted": 0,
34        "numAMContainerPreempted": 0
35      }
36    ]
37  }
38 }
```

Listing B.2: REST-Ausgabe aller Ausführungen einer Anwendung vom RM.

URL: `http://<rmhttpaddress:port>/ws/v1/cluster/apps/{appid}/appattempts`

```
1 {
2   "appAttempts": {
3     "appAttempt": [
4       {
5         "id": 1,
6         "startTime": 1518430260521,
7         "containerId": "
            container_1518429920717_0001_01_000001",
8         "nodeHttpAddress": "compute-2:8042",
9         "nodeId": "compute-2:45454",
10        "logsLink": "//compute-2:8042/node/containerlogs/
            container_1518429920717_0001_01_000001/root"
11      }
12    ]
13  }
14 }
```