

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: 21. Mai 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. In dieser Fallstudie verwendetes Setup	7
3. Aufbau des Modells	10
3.1. YARN-Modell	10
3.1.1. Modellierte YARN-Komponenten	12
3.1.2. Implementierung der Komponentenfehler	13
3.1.3. Fehlerüberprüfung	13
3.2. SSH-Treiber	16
3.2.1. Integration im Modell	16
3.2.2. Implementierte Parser	17
3.2.3. Implementierte Connectoren	18
3.2.4. SSH-Verbindung	19
4. Implementierung der Benchmarks	20
4.1. Übersicht möglicher Anwendungen	20
4.2. Auswahl der verwendeten Anwendungen	22
4.3. Implementierung der Anwendungen im Modell	24
5. Ausführung der Testfälle	28
5.1. Implementierung der Simulation	28
5.1.1. Grundlegender Aufbau	28
5.1.2. Initialisierung des Modells	30
5.1.3. Ablauf eines Simulations-Schrittes	32
5.1.4. Weitere Möglichkeiten der Simulation	34
Literatur	35
A. Kommandozeilen-Befehle von Hadoop	37
B. REST-API von Hadoop	40

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	7
2.5. In der Fallstudie verwendetes Cluster-Setup	8
3.1. Grundlegende Architektur des Gesamtmodells	10
3.2. Aufbau des YARN-Modells	11

Listings

3.1. Injizierung eines Komponentenfehlers	14
3.2. Definition der Constraints in YarnApp	15
4.1. Definition und Start einer Anwendung	26
4.2. Normalisierung und Auswahl der nachfolgenden Anwendung	27
5.1. Simulation in dieser Fallstudie	28
5.2. Initialisierung des Modells für die Simulation	30
A.1. CMD-Ausgabe der Anwendungsliste	37
A.2. CMD-Ausgabe des Reports einer Anwendung	37
A.3. Starten einer Anwendung in Hadoop-Benchmark	38
A.4. Vorzeitiges Beenden einer Anwendung	39
B.1. REST--Ausgabe aller Anwendungen vom RM	40
B.2. REST-Ausgabe aller Ausführungen einer Anwendung vom TLS	41

Tabellenverzeichnis

4.1. Verwendete Markov-Kette für die Anwendungs-Übergänge in Tabellenform.	24
--	----

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
dfw	TestDFSIO -write
rtw	randomtextwriter
tg	teragen
dfr	TestDFSIO -read
wc	wordcount
rw	randomwriter
so	sort
tsr	terasort
pi	pi
pt	pentomino
tms	testmapredsort
tv1	teravalidate
sl	sleep
fl	fail

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 innerhalb 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. Unterstützt wird der *NameNode* dabei vom *Secondary NameNode*, der einige der internen Verwaltungsaufgaben des HDFS übernimmt [9]. 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].



Abbildung 2.2.: Architektur des HDFS (entnommen aus [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 aufgeteilt, 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 [10]. 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, 11].

Evtl. noch ein paar Infos zur Node-Erkennung und zeitlichen abläufen

(funktionale) Anforderungen einfügen

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,

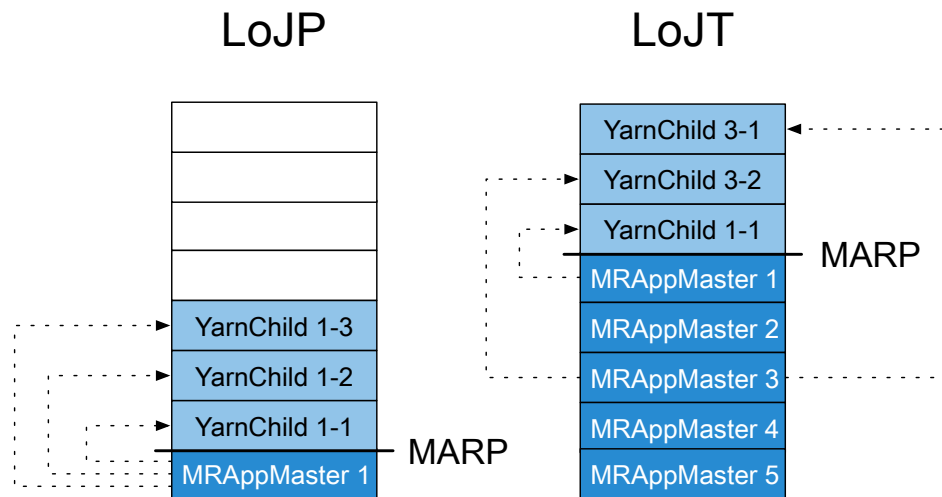


Abbildung 2.3.: LoJP und LoJT in Hadoop (entnommen aus [4])

die von mehreren Organisationen gemeinsam verwendet werden und sicherstellen soll, dass jede Organisation eine Mindestmenge an Ressourcen zur Verfügung hat [12].

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 [12]. Damit bewirkt diese Einstellung indirekt auch die maximale Anzahl an 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 darüber hinaus auch effizienter ist als eine manuelle, statische Optimierung.

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. Auf Basis dieser Plattform und der enthaltenen Benchmarks wurde das reale Cluster für diese Masterarbeit aufgebaut.

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 mit wenigen Befehlen ein Hadoop-Cluster aufbauen zu können. Mit *Graphite*⁴ ist zudem ein Monitoring-Tool enthalten, mit dem die Systemwerte wie CPU- oder Speicher-Auslastung 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* ein Cluster erstellt, auf denen dann in den Docker-Containern das eigentliche Hadoop-Cluster ausgeführt wird. In Hadoop-Benchmark werden mithilfe von Docker-Machine und VirtualBox⁵ virtuelle Maschinen erstellt, die mit dem Betriebssystem *Boot2Docker* ausgestattet sind. Boot2Docker ist eine leichtgewichtige Linux-Distribution, auf der Docker bereits vorinstalliert ist [14]. Jeder Hadoop-Container enthält zudem das Tool *collectd*⁶, was das Monitoring des Containers auf Systemebene übernimmt und die Daten an den Graphite-Container übermittelt. Dadurch wird es möglich, eine beliebige Anzahl an voneinander unabhängigen Nodes auf einem physischen Computer ausführen zu können. Auch ist es möglich, den Docker-Machines einen beliebig großen Arbeitsspeicher zur Verfügung zu stellen.

²<https://github.com/Spirals-Team/hadoop-benchmark>

³<https://www.docker.com/>

⁴<https://graphiteapp.org/>

⁵<https://www.virtualbox.org/>

⁶<https://collectd.org/>

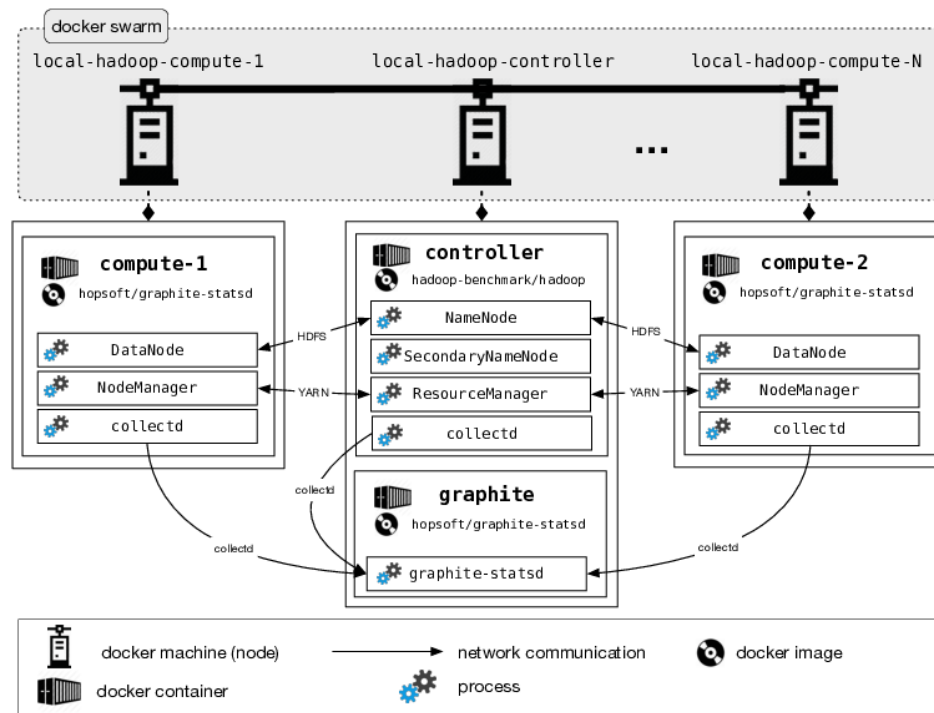


Abbildung 2.4.: High-Level-Architektur von Hadoop-Benchmark. Entnommen aus [13].

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

- Hadoop Mapreduce Examples
- Intel HiBench⁷
- Statistical Workload Injector for Mapreduce (SWIM)⁸

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. In dieser Fallstudie verwendetes Setup

Da die Plattform Hadoop-Benchmark mithilfe von Docker auf einem physischen PC sehr einfach ein komplettes Hadoop-Cluster ausführen kann, wurde die Plattform für diese Fallstudie als Basis genutzt. Da Docker und Hadoop vor allem für den Einsatz in einer Linux-Umgebung entwickelt wurden, werden für die Fallstudie zwei Computer genutzt, auf denen das Cluster wahlweise auf einem oder auf beiden Hosts ausgeführt

⁷<https://github.com/intel-hadoop/HiBench>

⁸<https://github.com/SWIMProjectUCB/SWIM>

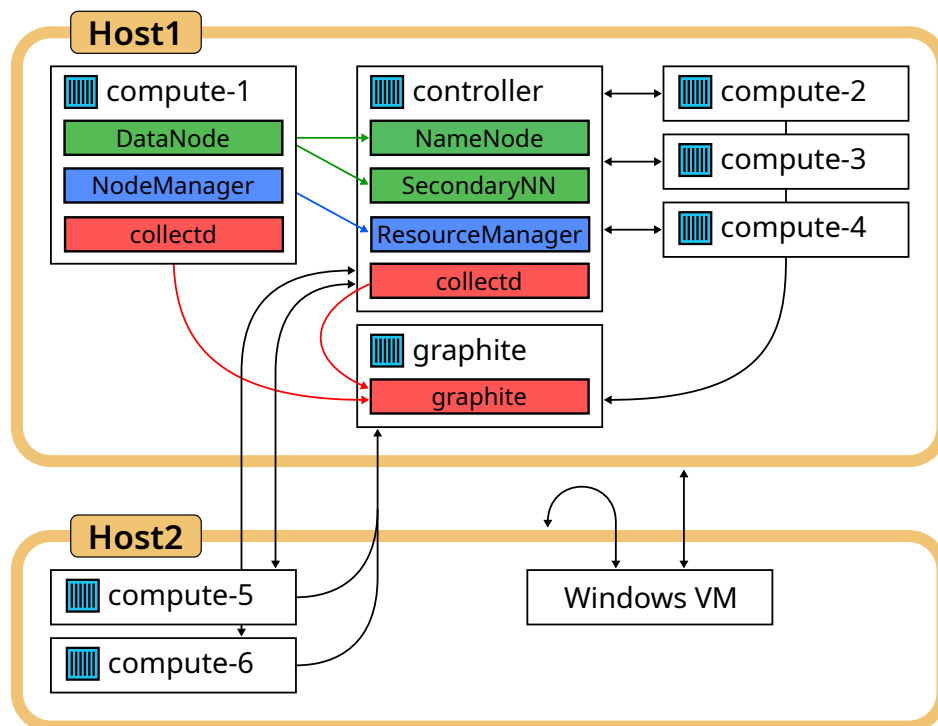


Abbildung 2.5.: In der Fallstudie verwendetes Cluster-Setup. Grün: HDFS, Blau: YARN, Rot: Graphite.

werden kann. Zudem wird auf einem Host eine VM mit Windows 10 ausgeführt, das zum Ausführen des .NET-Frameworks bzw. S# benötigt wird. Beide zum Einsatz kommenden Hosts sind jeweils mit einem Intel Core i5-4570 @ 3,2 GHz x 4, 16 GB Arbeitsspeicher sowie einer SSD ausgestattet, auf der Ubuntu 16.04 LTS installiert ist. Die Verbindung von Windows zu Linux auf beiden Hosts wird mithilfe von SSH-Verbindungen umgesetzt.

Die beiden Abbildungen Abbildung 2.4 und Abbildung 2.5 zeigen bereits den großen Hauptunterschied zwischen der Plattform und dem hier verwendeten Cluster-Setup. Da durch die Nutzung von virtuellen Maschinen ein zusätzlicher Ressourcenbedarf entsteht, wird im hier verwendeten Setup darauf verzichtet. Durch die Ausführung der Docker-Container des Hadoop-Clusters direkt auf dem Host stehen dem Cluster mehr Ressourcen zur Verfügung. Zudem wird es mithilfe von *Docker Swarm* so ermöglicht, das Hadoop-Cluster auf beiden Hosts auszuführen. Im konkreten Setup werden dabei Graphite, der Hadoop-Controller sowie vier Hadoop-Nodes auf dem Host1, sowie zwei weitere, optionale Nodes auf Host2 ausgeführt. Weitere Anpassungen des verwendeten Setups bestehen u. A. darin, dass der TLS von Hadoop ebenfalls gestartet wird. Zudem wurden einige Einstellungen von Hadoop so angepasst, dass defekte Nodes schneller erkannt werden.

Zum Ausführen der Windows-VM auf Host2 wird VirtualBox 5.2 verwendet. Zum Abrufen von Daten mithilfe der REST-API von Hadoop über die SSH-Verbindungen

wird *curl*⁹ genutzt. Zum Ausführen des Hadoop-Clusters wird Docker in der Version 18.03 CE genutzt.

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

- Starten und Beenden des Clusters
- Starten und Beenden einzelner Hadoop-Nodes
- Hinzufügen und Entfernen der Netzwerkverbindung des Docker-Containers eines Hadoop-Nodes
- Ausführen von eigenen Befehlen auf dem Docker-Container des Hadoop-Controllers
- Erstellen des Hadoop-Docker-Images

Das zweite erstellte Script dient ausschließlich zum Starten der Benchmarks. Dazu werden die in der Plattform bereits enthaltenen Start-Scripte aufgerufen, die für das konkrete Setup angepasst wurden.

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

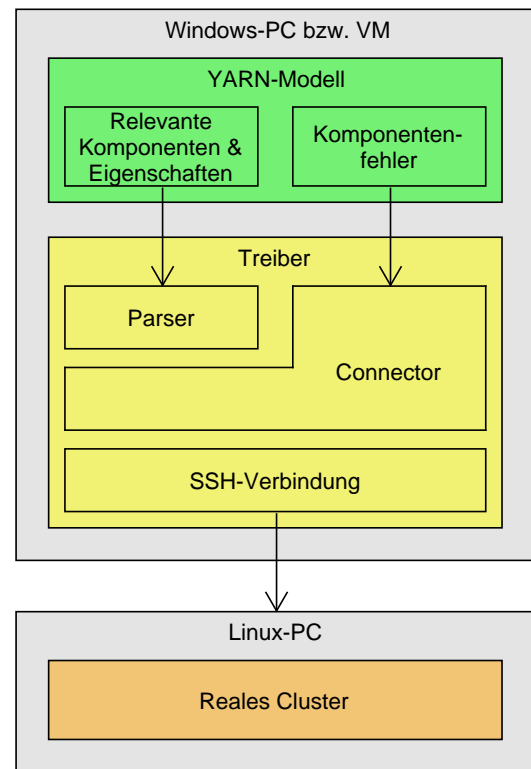


Abbildung 3.1.: Grundlegende Architektur des Gesamtmodells

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.

Multihost-Mode irgendwo erklären

3.1. YARN-Modell

Komplett neu strukturieren, am besten (inkl. Bild) in einzelne Bestandteile aufteilen, zB Controller, Nodes, Anwendungsmodell, Client, implementierte Komponentenfehler immer direkt mit rein, Fehlerprüfung wohl im Rahmen vom Controller

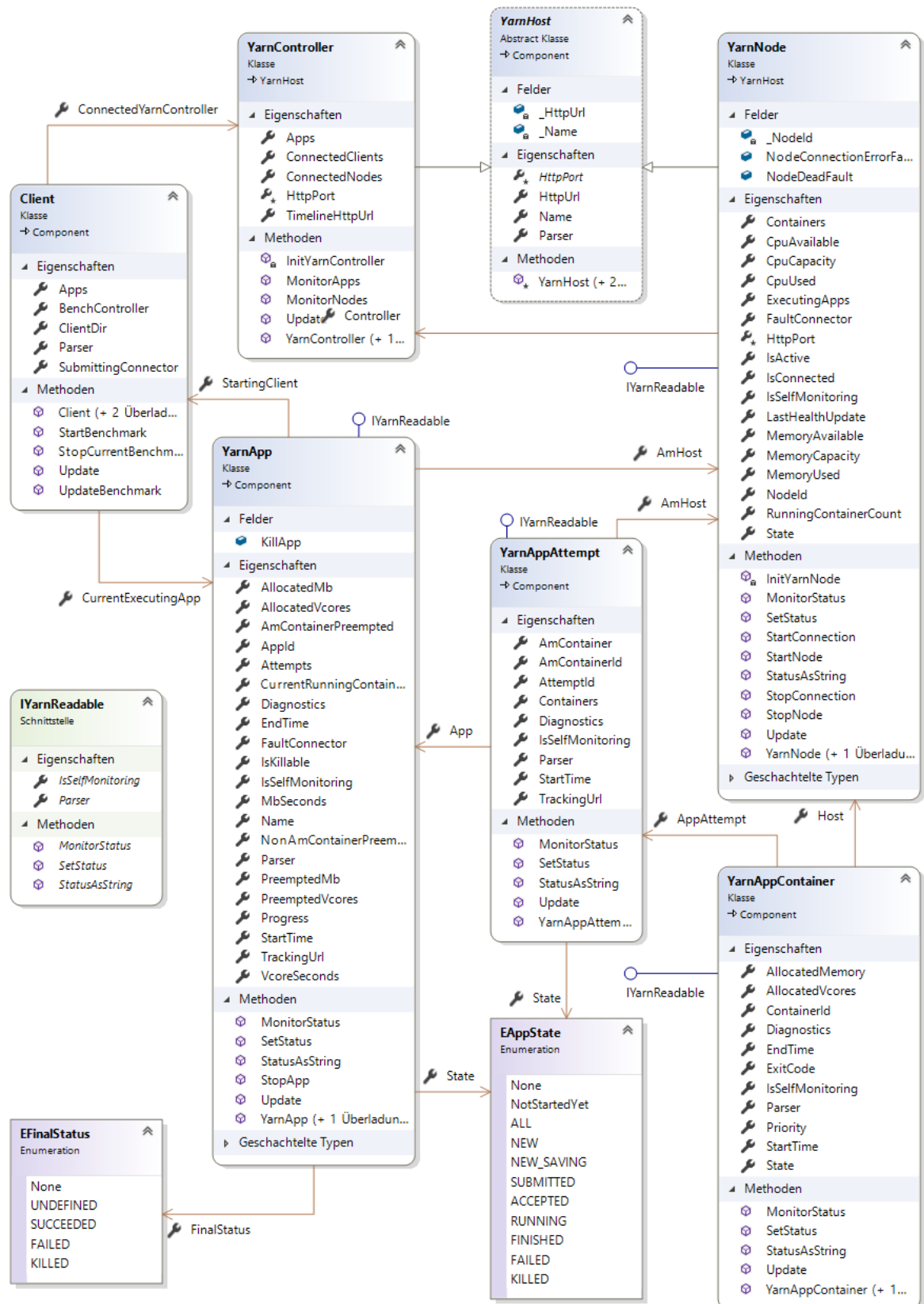


Abbildung 3.2.: Aufbau des YARN-Modells. Das Modell wurde mithilfe des Klassendiagramm-Designers in Visual Studio 2017 visualisiert. Daher werden Assoziationen mit höherer Multiplizität als 1, die daher mithilfe von `List<T>` umgesetzt wurden (z. B. `YarnApp.Attempts`) im Diagramm nicht als Assoziationen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

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.

3.1.1. Modellierte YARN-Komponenten

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 mithilfe von `YarnApp` dargestellten Anwendungen werden mithilfe des `Bench-Controllers` (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.

3.1.2. Implementierung der Komponentenfehler

Die Felder `YarnNode.NodeConnectionError` und `YarnNode.NodeDead` definieren die Komponentenfehler, wenn ein Node seine Netzwerkverbindung verliert bzw. beendet wird. Die aus den Komponentenfehlern resultierenden Effekte werden in den dafür implementierten geschachtelten Klassen definiert. Listing 3.1 zeigt beispielhaft die Implementierung und Injizierung des `NodeDead`-Komponentenfehlers mithilfe des für den Node verwendeten `CmdConnector` (vgl. Unterabschnitt 3.2.3). Die Injizierung des `NodeConnectionError`-Komponentenfehlers und die Aufhebung beider Komponentenfehler sind analog implementiert.

3.1.3. Fehlerüberprüfung

Um zu prüfen, ob sich das reale Cluster nach der Aktivierung bzw. Deaktivierung eines Komponentenfehlers korrekt rekonfiguriert, werden *Constraints* genutzt.

Verweis zu Anforderungen

Diese richten sich nach den funktionalen Anforderungen des Systems und prüfen, ob diese weiterhin eingehalten werden. Da die funktionalen Anforderungen bei jeder YARN-Komponente unterschiedlich sind, wurden diese mithilfe der Eigenschaft `IYarnReadable.Constraints` für jede Komponente einzeln definiert. Listing 3.2 zeigt die Definition der Constraints für `YarnApp`, bei der die Anforderungen 1 und 3 eine Rolle spielen. In jeder Komponente sind nur die funktionalen Anforderungen als Constraints implementiert, die für diese Komponente auch relevant sind. Daher finden sich die beiden Anforderungen 2 und 4 nicht in der Klasse `YarnApp` wieder, letztere dafür aber z. B. in `YarnNode`.

```
1 public class YarnNode : YarnHost, IYarnReadable
2 {
3     public readonly Fault NodeDeadFault = new TransientFault();
4     public IHadoopConnector FaultConnector { get; set; }
5
6     public bool StopNode(bool retry = true)
7     {
8         if(IsActive)
9         {
10             var isStopped = FaultConnector.StopNode(Name);
11             if(isStopped)
12                 IsActive = false;
13             else if(retry)
14                 StopNode(false); // try again once
15         }
16         return !IsActive;
17     }
18
19     [FaultEffect(Fault = nameof(NodeDeadFault))]
20     public class NodeDeadEffect : YarnNode
21     {
22         public override void Update()
23         {
24             StopNode();
25         }
26     }
27 }
28
29 public class CmdConnector : IHadoopConnector
30 {
31     private SshConnection Faulting { get; }
32
33     public bool StopNode(string nodeName)
34     {
35         var id = DriverUtilities.ParseInt(nodeName);
36         Faulting.Run($"{Model.HadoopSetupScript} hadoop stop {id}",
37             IsConsoleOut);
38         return !CheckNodeRunning(id);
39     }
40 }
```

Listing 3.1: Injizierung eines Komponentenfehlers (gekürzt). Sollte der Node nicht beendet werden, wird die Injizierung einmalig erneut versucht. `CmdConnector.Faulting` ist der für Komponentenfehler verwendete Connector.

```

1 public Func<bool>[] Constraints => new Func<bool>[]
2 {
3     () =>
4     {
5         if(FinalStatus != EFinalStatus.FAILED) return true;
6         if(!String.IsNullOrEmpty(Name) && Name.ToLower().Contains("
7             fail job")) return true;
8         return false;
9     },
10    () =>
11    {
12        if(State == EAppState.RUNNING)
13            return AmHost?.State == ENodeState.RUNNING;
14        return true;
15    },
16 };

```

Listing 3.2: Definition der Constraints in YarnApp

Geprüft werden die Constraints im Anschluss an das Monitoring der einzelnen YARN-Komponenten. Wenn dabei die Bedingungen einer funktionalen Anforderungen nicht erfüllt werden, wird von den Constraints `false` zurückgegeben und so erkannt, dass bei dieser Komponente ein Fehler von Hadoop nicht selbst korrigiert wurde. Die ID der Komponente wird daher entsprechend ausgegeben bzw. in der Logdatei gespeichert. Zwar wäre es hier auch möglich gewesen, ähnlich wie in den Modellen der anderen Fallstudien, die mit dem S $\#$ -Framework entwickelt wurden, eine Exception zu werfen, jedoch wurde hier darauf verzichtet, damit immer die Daten aller Komponenten geprüft werden können. Dadurch kann erkannt werden, wenn mehrere Komponenten nicht den funktionalen Anforderungen entsprechen.

Nach der Überprüfung der Constraints wird abschließend geprüft, ob es dem Cluster möglich ist, sich überhaupt rekonfigurieren zu können. Dies wird dadurch realisiert, dass geprüft wird, ob mindestens ein Node noch aktiv ist. Dabei wird jedoch nicht der interne Fehlerstatus in `YarnNode.IsActive` oder `YarnNode.IsConnected` geprüft, sondern der beim Monitoring vom Cluster zurückgegebene `YarnNode.State`. Nur wenn dieser den Wert `ENodeState.Running` hat, ist der Node aktiv und kann Anwendungen ausführen. Das reale Hadoop-Cluster kann sich somit nicht mehr rekonfigurieren und neue Container allokalieren bzw. in der Ausführung befindliche Anwendungen und ihre Komponenten umverteilen, wenn kein Node den Wert `ENodeState.Running` hat.

Verweis auf Abschnitt, wo implementierung der Simulation genauer erklärt wird

Kommt es zu diesem Fall, wird dies analog zu den Constraints ebenfalls ausgegeben und in der Logdatei vermerkt und die Ausführung des Simulationsschrittes fortgeführt, da die Daten aller Yarn-Komponenten erst nach Abschluss der Simulation eines Schrittes ausgegeben werden. Somit kann im Fehlerfall einfacher ermittelt werden, wie der Systemzustand zum Zeitpunkt des Fehlers war.

3.2. SSH-Treiber

Im Einführungstext zu diesem Kapitel 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, 15–17]:

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 geparsen 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 `FaultConnector`-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, 16]. 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, 16, 17], 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.

¹<https://www.newtonsoft.com/json>

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, 16, 17], 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 [16, 17]. 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, 15], 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 Rahmen dieser Fallstudie 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 [18]. 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. Der anfangs nur wenige Anwendungen enthaltene Benchmark [19] wurde stetig mit neuen Anwendungsarten und Workloads erweitert. Das zeigt sich auch darin, dass in in Hadoop-Benchmark noch die HiBench-Version 2.2 verwendet wird, die einen noch deutlich geringeren Umfang an Workloads und Anwendungen besitzt, als die aktuelle Version 7. Daher wurde der der Docker-Container von HiBench zunächst auf die aktuelle Version 7 aktualisiert. HiBench enthält damit folgende Workloads mit einer unterschiedlichen Anzahl an möglichen Anwendungen:

- Micro-Benchmarks (basierend 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 [20]. Bei der Ausführung auf dem in dieser Fallstudie verwendeten Cluster wurden jedoch nicht alle Workloads fehlerfrei ausgeführt. Zudem wird in [21] 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 hier aufgrund ihres Dateinamens als **Jobclient-Tests** bezeichneten Anwendungen. 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.

Einige der in Abschnitt 4.1 erwähnten Mapreduce Examples werden häufig als Benchmark verwendet. Einige Beispiele dafür sind die Anwendungen `sort` und `grep` (ermittelt Anzahl von Regex-Übereinstimmungen), die bereits im Referenzpapier zum MapReduce-Algorithmus als Benchmarks verwendet wurden [22]. `terasort` ist ebenfalls ein weit verbreiteter Benchmark, der die Hadoop-Implementierung der standardisierten *Sort Benchmarks*¹ darstellt [23]. Ebenfalls als guter Benchmark dient die Anwendung `wordcount`, mit der ein großer Datensatz stark verkleinert bzw. zusammengefasst wird und dient daher als gute Repräsentation für Anwendungsarten, bei denen Daten extrahiert werden [19, 24].

Da in dieser Fallstudie ein realistisches Abbild der ausgeführten Anwendungen ausgeführt werden soll, ist es nicht sehr hilfreich, die einzelnen Übergangswahrscheinlichkeiten im Transitionssystem anzugleichen oder rein zufällig zu verteilen. Einen realistischen Einblick, welche Anwendungs- und Datentypen in produktiv genutzten Hadoop-Clustern genutzt werden, geben u. A. [24] und [25]. Auffällig ist hierbei, dass die meisten Anwendungen in einem Hadoop-Cluster innerhalb weniger Sekunden oder Minuten abgeschlossen sind und/oder Datensätze im Größenbereich von wenigen Kilobyte bis hin zu wenigen Megabyte verarbeiten. Zu einem ähnlichen Ergebnis kamen auch Ren u. a. in [26] und folgerten daher, dass für kleine Jobs evtl. einfachere Frameworks abseits von Hadoop besser geeignet wären. Die Autoren der Studie in [25] bezeichneten Hadoop aufgrund ihrer Ergebnisse als „potentielle Technologie zum Verarbeiten aller Arten von Daten“, stellten aber eine ähnliche Vermutung an wie Ren u. a., dass Hadoop primär Daten nutze, die auch mit „traditionellen Plattformen“ verarbeitet werden könnten.

¹<https://sortbenchmark.org/>

Basierend auf den Ergebnissen der Studien und der in den anderen Publikationen verwendeten Benchmark-Anwendungen, wurden folgende Anwendungen der Mapreduce-Examples und Jobclient-Tests in das Transitionssystem übernommen:

- Generieren von Eingabedaten für andere Anwendungen:
 - Textdateien: `randomtextwriter` (rtw) und `TestDFSIO -write` (dfw)
 - Binärdateien: `randomwriter` (rw) und `teragen` (tg)
- Verarbeitung von Eingabedaten:
 - Auslesen bzw. Zusammenfassen: `wordcount` (wc) und `TestDFSIO -read` (dfr)
 - Transformieren: `sort` (so) für Textdaten und `terasort` (tsr) für Binärdaten
 - Validierung: `testmapredsort` (tms) und `teravalidate` (tv1) für die jeweiligen Sortier-Anwendungen
- Ausführen von Berechnungen:
 - `pi`: Quasi-Monte-Carlo-Methode zur einfachen Berechnung von π
 - `pentomino` (pt): Berechnung von Pentomino-Problemen
- Dummy-Anwendungen: `sleep` (sl) und `fail` (fl)

Der Grund für die Berücksichtigung von mehreren gleichen bzw. ähnlichen Anwendungen für einige Kategorien liegt darin, dass die unterschiedlichen Anwendungen eine unterschiedliche Ausführungsdauer bzw. Datenrepräsentation (Text und Binär) repräsentieren. 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 generieren und zu lesen, weshalb diese Anwendung in zwei Kategorien verwendet 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 auf dem Cluster z. B. derzeit nichts ausgeführt wird, oder ein unerwarteter Fehler während der Ausführung auftaucht. Daher können beide Anwendungen unabhängig von der derzeit ausgeführten Anwendung als nachfolgende Anwendung ausgewählt werden. Als nachfolgende Anwendungen für die Dummy-Anwendungen kommen nur Anwendungen in Betracht, welche ihrerseits keine Eingabedaten benötigen. Dies sind:

- `TestDFSIO -write`
- `randomtextwriter`
- `teragen`

	<i>dfw</i>	<i>rtw</i>	<i>tg</i>	<i>dfr</i>	<i>wc</i>	<i>rw</i>	<i>so</i>	<i>tsr</i>	<i>pi</i>	<i>pt</i>	<i>tms</i>	<i>ttl</i>	<i>sl</i>	<i>fl</i>
<i>dfw</i>	.600	.073	0	.145	0	0	0	0	.073	.073	0	0	.018	.018
<i>rtw</i>	.036	.600	0	0	.145	.036	.109	0	.036	0	0	0	.019	.019
<i>tg</i>	0	.036	.600	0	0	0	0	.255	0	.073	0	0	.018	.018
<i>dfr</i>	0	.073	0	.600	0	.036	0	0	.145	.109	0	0	.018	.019
<i>wc</i>	.073	.109	0	0	.600	0	.073	0	.073	.036	0	0	.018	.018
<i>rw</i>	0	.073	.073	0	0	.600	0	0	.109	.109	0	0	.018	.018
<i>so</i>	0	.073	.036	0	.073	.036	.600	0	.073	0	.073	0	.018	.018
<i>tsr</i>	0	0	0	0	0	0	0	.600	.109	.073	0	.182	.018	.018
<i>pi</i>	.145	.109	0	0	0	0	0	0	.600	.109	0	0	.018	.019
<i>pt</i>	.109	.109	0	0	0	.073	0	0	.073	.600	0	0	.018	.018
<i>tms</i>	0	.145	0	0	0	.073	0	0	.036	.109	.600	0	.018	.019
<i>ttl</i>	.073	.109	0	0	0	0	0	0	.109	.073	0	.600	.018	.018
<i>sl</i>	.167	.167	.167	0	0	.167	0	0	.167	.167	0	0	0	0
<i>fl</i>	.167	.167	.167	0	0	.167	0	0	.167	.167	0	0	0	0

Tabelle 4.1.: Verwendete Markov-Kette für die Anwendungs-Übergänge in Tabellenform.

- `randomwriter`
- `pi`
- `pentomino`

Für die in Tabelle 4.1 dargestellte Markov-Kette der Übergänge zwischen den Anwendungen wurde neben den Ergebnissen aus den Studien zudem berücksichtigt, welche Anwendungen bestimmte Eingabedaten benötigen. Dadurch wird sichergestellt, dass die für einige Anwendungen benötigten Eingabedaten immer vorhanden sind, da diese ebenfalls im Rahmen der Ausführung der Benchmarks generiert werden. Anwendungen ohne Eingabedaten können dagegen fast jederzeit ausgeführt werden.

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. Zur Auswahl der initialen Anwendung wird basierend auf der `sleep`-Anwendung das Transitionssystem genutzt und so eine Anwendung ausgewählt, die keine Eingabedaten benötigt bzw. diese für andere Anwendungen generiert.

Das im vorherigen Abschnitt definierte und im statischen Teil implementierte Transitionssystem kommt auch 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.

Nachdem eine neue Anwendung ausgewählt wurde, muss zunächst sichergestellt werden, dass die bisher ausgeführte Anwendung beendet ist. Dafür wird der in Listing A.4 dargestellte Befehl von Hadoop zum Abbruch von Anwendungen ausgeführt, wodurch die derzeit ausgeführte Anwendung beendet wird, sollte sie noch nicht abgeschlossen sein. 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 in Listing A.3 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 erst nach Abschluss der Ausführung ausgegeben wird. Gespeichert wird

```

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.


```
1 // get probabilities from current benchmark
2 var transitions = BenchTransitions[CurrentBenchmark.Id];
3
4 var ranNumber = RandomGen.NextDouble();
5 var cumulative = 0D;
6 for(int i = 0; i < transitions.Length; i++)
7 {
8     cumulative += transitions[i];
9     if(ranNumber >= cumulative)
10         continue;
11
12     // save benchmarks
13     PreviousBenchmark = CurrentBenchmark;
14     CurrentBenchmark = Benchmarks[i];
15 }
```

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

die ID zunächst in einer noch verfügbaren **YarnApp**-Instanz, welche anschließend selbst in `Client.CurrentExecutingApp` gespeichert wird.

5. Ausführung der Testfälle

Um nun Testfälle ausführen zu können, wurde zunächst eine Simulation erstellt, mit der einzelne Testfälle ohne die Aktivierung von Komponentenfehlern ausgeführt werden können. Die Simulation dient vor allem als Vergleichswert für die Evaluation der gesamten Fallstudie. Neben der Simulation wurde aber auch ein Analysetest erstellt, bei dem das S#-Framework die implementierten Komponentenfehler aktiviert und so ermittelt, ob sich das reale Cluster so verhält, wie es erwartet wird.

5.1. Implementierung der Simulation

Wie bereits erwähnt, wurden zwei grundlegende Tests implementiert. Das ist zum einen die Simulation, bei der ein Testfall ohne die Aktivierung von Komponentenfehlern ausgeführt wird, sowie der Analysetest, bei dem Komponentenfehler aktiviert werden.

5.1.1. Grundlegender Aufbau

Anpassen wenn Stand der Technik mit S#-Einführung geschrieben ist

Die Simulation ist die einfache Ausführung eines Testfalls ohne die Aktivierung der implementierten Komponentenfehlern oder der Erzeugung von weiteren Fehlern im realen Cluster. Der S#-Simulator unterstützt eine Simulation in einzelnen oder mehreren Schritten, zwischen denen in reinen Modellen beliebig gewechselt werden kann. Da hier jedoch ein reales System getestet wird, wird jeder Schritt einzeln ausgeführt.

Da im realen Cluster Hadoop kontinuierlich Anpassungen durchführt und Tests in S# mit diskreten Schritten durchgeführt werden, muss beachtet werden, dass die Werte, die beim Test ermittelt werden, immer nur Momentaufnahmen darstellen. Ebenso muss beachtet werden, dass bei der Deaktivierung von einzelnen Nodes bzw. deren Netzwerkverbindungen diese nicht in Echtzeit, sondern um einige Zeit verzögert erkannt werden und erst nach einer gewissen Zeit aus der Konfiguration des Clusters entfernt werden. Genauso verhält es sich, wenn ein Node bzw. seine Verbindung wieder aktiviert wird, da dieser zunächst selbst starten muss und sich mit dem YARN-Controller verbinden muss. Außerdem werden die für die auf dem Cluster ausgeführten Anwendungen benötigten AppMstr und YARN-Container aufgrund der komplexen internen Prozesse von Hadoop nicht innerhalb weniger Millisekunden allokiert, sondern benötigen ebenfalls eine gewisse Zeit. Aus diesen Gründen darf ein Schritt nicht zu schnell vorüber sein.

```
1 public void SimulateHadoop()  
2 {  
3     ModelSettings.FaultActivationProbability = 0.0;
```

```
4   ModelSettings.FaultRepairProbability = 1.0;
5
6   ExecuteSimulation();
7 }
8
9 private void ExecuteSimulation(bool isWithFaults)
10 {
11     var origModel = InitModel();
12
13     var wasFatalError = false;
14     try
15     {
16         var simulator = new SafetySharpSimulator(origModel);
17         var simModel = (Model)simulator.Model;
18         var faults = CollectYarnNodeFaults(simModel);
19
20         SimulateBenchmarks();
21
22         for(var i = 0; i < _StepCount; i++)
23         {
24             OutputUtilities.PrintStepStart();
25             var stepStartTime = DateTime.Now;
26
27             if(isWithFaults)
28                 HandleFaults(faults);
29             simulator.SimulateStep();
30
31             var stepTime = DateTime.Now - stepStartTime;
32             OutputUtilities.PrintStepTime(stepTime);
33             if(stepTime < ModelSettings.MinStepTime)
34                 Thread.Sleep(ModelSettings.MinStepTime - stepTime);
35
36             OutputUtilities.PrintFullTrace(simModel.Controller);
37         }
38
39         OutputUtilities.PrintExecutionFinish();
40     }
41     // catch/finally
42 }
```

Listing 5.1: Simulation in dieser Fallstudie (gekürzt).

Listing 5.1 zeigt den Ablauf einer Hadoop-Simulation. Da der Ablauf der Simulation unabhängig von der Aktivierung der Komponentenfehler der gleiche ist, ist hier nur die Variante ohne deren Aktivierung aufgezeigt. Im Falle einer Aktivierung der Komponentenfehler unterscheiden sich beide Simulationsvarianten nur durch die Angabe der Wahrscheinlichkeiten zum Aktivieren und Deaktivieren der Komponentenfehler sowie des Übergabeparameters an `ExecuteSimulation()`. Da die einzelnen Schritte

einer Simulation eine gewisse Mindestdauer haben, wird nach jedem Schritt geprüft, wie viel Zeit für die Ausführung des Schrittes benötigt wurde. Liegt die Zeit unterhalb der Mindestdauer für einen Schritt, wird die Ausführung des nächsten Schrittes solange hinausgezögert, bis die Mindestdauer des Schrittes erreicht wurde.

Wenn während der Simulation eine im Modell nicht behandelte `Exception` auftritt, dann wird diese außerhalb der Simulation abgefangen und geloggt. Dadurch wird zudem die Simulation beim aktuellen Stand abgebrochen und unabhängig von aufgetretenen `Exceptions` Nodes mit injizierten Komponentenfehlern neu gestartet.

5.1.2. Initialisierung des Modells

Bevor das Modell im Simulator ausgeführt werden kann, muss es initialisiert werden. Das folgende Listing 5.2 zeigt die Definition der Variablen zur Modellinitialisierung sowie die entsprechenden Methoden, die in Listing 5.1 zur Initialisierung aufgerufen werden:

```

1 private static TimeSpan _MinStepTime = new TimeSpan(0, 0, 0, 20);
2 private static int _BenchmarkSeed = Environment.TickCount;
3 private static int _StepCount = 3;
4 private static bool _PrecreatedInputs = true;
5 private static double _FaultActivationProbability = 0.4;
6 private static double _FaultRepairProbability = 0.5;
7 private static int _HostsCount = 1;
8 private static int _NodeBaseCount = 4;
9 private static int _ClientCount = 1;
10
11 private Model InitModel()
12 {
13     ModelSettings.HostMode = ModelSettings.EHostMode.Multihost;
14     ModelSettings.HostsCount = _HostsCount;
15     ModelSettings.NodeBaseCount = _NodeBaseCount;
16     ModelSettings.IsPrecreateBenchInputs = _PrecreatedInputs;
17     ModelSettings.RandomBaseSeed = _BenchmarkSeed;
18
19     var model = Model.Instance;
20     model.InitModel(appCount: _StepCount, clientCount: _ClientCount);
21     model.Faults.SuppressActivations();
22
23     return model;
24 }
25
26 private FaultTuple[] CollectYarnNodeFaults(Model model)
27 {
28     return (from node in model.Nodes
29
30         from faultField in node.GetType().GetFields()

```

```
31     where typeof(Fault).IsAssignableFrom(faultField.FieldType)
32
33     let attr = faultField.GetCustomAttribute<NodeFaultAttribute>()
34     where attr != null
35
36     let fault = (Fault)faultField.GetValue(node)
37
38     select Tuple.Create(fault, attr, node)
39 ).ToArray();
40 }
```

Listing 5.2: Initialisierung des Modells für die Simulation

Die einzelnen Variablen für die Simulation werden statisch definiert und beim Initialisieren des Modells in den `ModelSettings` gespeichert. Die dort gespeicherten Werte werden anschließend beim Initialisieren der Modell-Instanz bzw. während der Ausführung der Simulation genutzt.

Einige Variablen haben lediglich einen Zweck, während andere umfangreichere Auswirkungen besitzen. Die einfachen Variablen sind:

`_MinStepTime` Definiert die Mindestdauer eines Schrittes.

`_BenchmarkSeed` Gibt den Seed an, mit dem die Zufallsgeneratoren in den Klassen `BenchmarkController` und `NodeFaultAttribute` initialisiert werden. Dadurch wird es ermöglicht, einzelne Testfälle erneut ausführen zu können.

`_StepCount` Definiert die Anzahl der ausgeführten Schritte.

`_FaultActivationProbability` Definiert die generelle Häufigkeit zum Aktivieren von Komponentengehlern. Ist dieser Wert 0,0, werden grundsätzlich keine Komponentengehler aktiviert, bei einem Wert von 1,0 werden Komponentengehler dagegen immer aktiviert.

`_FaultRepariProbability` Definiert die generelle Häufigkeit zum Deaktivieren von Komponentengehlern. Die hier definierte Wahrscheinlichkeit verhält sich analog zu `_FaultActivationProbability`. Bei einem Wert von 0,0 werden Komponentengehler niemals deaktiviert, während sie bei einem Wert von 1,0 im nachfolgenden Schritt immer deaktiviert werden.

`_HostsCount` Definiert die Anzahl der in der Simulation verwendeten Hosts. Benötigt wird dieser Wert, damit zu jedem verwendeten Host eine SSH-Verbindung aufgebaut werden kann.

`_NodeBaseCount` Definiert die Anzahl der Nodes auf Host1. Auf Host2 wird die Hälfte der Nodes verwendet. Benötigt wird dieser Wert, um mithilfe der REST-API auf

die Hadoop-Nodes zugreifen zu können, um die Daten der YARN-Container zu ermitteln.

_ClientCount Definiert die Anzahl der zu simulierenden Clients. Da jeder Client gleichzeitig nur eine Anwendung startet, wird dadurch gleichzeitig definiert, wie viele Anwendungen gleichzeitig auf dem Cluster ausgeführt werden sollen.

Eine Besonderheit bildet die Variable **_PrecreatedInputs**. Sie definiert, ob die ausgeführten Anwendungen auf dem Cluster vorab generierte Eingabedaten nutzen oder alle Eingabedaten während der Ausführung selbst generieren. Der Unterschied zwischen beiden Varianten liegt darin, dass vorab generierte Eingabedaten in einem anderen Verzeichnis im HDFS gespeichert sind und während der Simulation die Eingabedaten aus diesem Verzeichnis gelesen werden. Wenn keine Eingabedaten vorab generiert werden, werden als Eingabeverzeichnis für die Anwendungen die Ausgabeverzeichnis der entsprechenden Benchmarks genutzt, die die dafür benötigten Daten generieren. Der genaue Ablauf der Bereitstellung der Eingabedaten wird in

Vorabgenerierung der Eingabedaten irgendwo schreiben und hier drauf verweisen

beschrieben.

Die Auswirkungen der in `InitModel()` definierten Einstellung `ModelSettings.HostMode` wird in

`ModelSettings.HostMode` beschrieben und hier verweisen

beschrieben.

Die direkt im Anschluss an die Initialisierung des Simulators ausgerufene Methode `CollectYarnNodeFaults()` ermittelt alle im Modell enthaltenen Komponentenfeler, die mit dem `NodeFaultAttribute` markiert sind. Die gefundenen Komponentenfeler werden als Array aus Tupel, bestehend aus dem Komponentenfeler selbst, dem Attribut sowie dem dazugehörigen Node zurückgegeben. Die jeweiligen Instanzen der Attribute und Nodes werden für die in Abschnitt 5.1.3 beschriebene Aktivierung der dazugehörigen Komponentenfeler benötigt.

5.1.3. Ablauf eines Simulations-Schrittes

Der Ablauf eines Schrittes lässt sich in die folgenden fünf Abschnitte einteilen. Während die Aktivierung und Deaktivierung der Komponentenfeler komplett außerhalb des Modells und durch den Simulator erfolgt, werden die anderen Abschnitte durch den `YarnController` innerhalb des Modells während der Ausführung eines Simulations-Schrittes durchgeführt. Die Ausgaben während eines Schrittes werden direkt von den jeweils betroffenen Komponenten im Rahmen des normalen Loggings durchgeführt.

Aktivierung und Deaktivierung der Komponentenfehler

Die Aktivierung der Komponentenfehler läuft in jeweils drei Schritten ab. Der erste Schritt ist die Prüfung, ob der Fehler bereits aktiviert wurde. Bei einem derzeit nicht injizierten Komponentenfehler, wird im zweiten Schritt geprüft, ob der Fehler aktiviert werden soll bevor er im dritten Schritt aktiviert und damit zur Injizierung im realen Cluster durch den S# -Simulator freigegeben wird.

Zur Entscheidung, ob ein Komponentenfehler aktiviert wird, hängt von folgenden Parametern ab:

- Von der Auslastung des Nodes im vorhergehenden Simulationsschritt,
- von der in `ModelSettings.FaultActivationProbability` definierten generellen Wahrscheinlichkeit zur Fehleraktivierung,
- sowie von einer Zufallszahl.

Zur Entscheidung, ob ein Komponentenfehler aktiviert wird, werden diese Parameter miteinander verrechnet und somit Entschieden, ob er aktiviert wird:

Algorithmus 5.1 Entscheidung zur Aktivierung von Komponentenfehlern

```

1: nodeUsage  $\leftarrow$  (memUsage + cpuUsage)/2
2: if nodeUsage < 0.1 then
3:   nodeUsage  $\leftarrow$  0.1
4: end if
5: if nodeUsage > 0.9 then
6:   nodeUsage  $\leftarrow$  0.9
7: end if
8: faultProbability  $\leftarrow$  1 - (nodeUsage * ActivationProbability * 2)
9: randomValue  $\leftarrow$  GetRandomDouble()
10: if faultProbability < randomValue then
11:   return true{Activate}
12: else
13:   return false{No activation}
14: end if

```

Die Entscheidung zur Deaktivierung eines Komponentenfehlers verhält sich analog. Anstatt der generellen Aktivierungswahrscheinlichkeit in `ModelSettings.FaultActivationProbability` wird die generelle Wahrscheinlichkeit zur Deaktivierung in `ModelSettings.FaultRepairProbability` genutzt. Außerdem spielt bei der Deaktivierung die Auslastung des Nodes zum Zeitpunkt der Aktivierung eine Rolle. Der grundlegende Algorithmus zur Entscheidung ist jedoch gleich.

Ausführung Benchmarks

Monitoring der ausgeführten Anwendungen

Prüfungen durch das Oracle

Ausgaben während eines Schrittes

5.1.4. Weitere Möglichkeiten der Simulation

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 ApacheTM 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. *HDFS Users Guide*. 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html#Secondary_NameNode (besucht am 27.03.2018).
- [10] 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).
- [11] 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).
- [12] 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).
- [13] Filip Krikava. *Architecture*. 23. Jan. 2017. URL: <https://github.com/Spirals-Team/hadoop-benchmark/blob/b32711e3a724e7183e4f52ba76e34f2e587a523a/README.md> (besucht am 22.01.2018).
- [14] Docker Inc. *Get started with Docker Machine and a local VM*. URL: <https://docs.docker.com/machine/get-started/> (besucht am 19.05.2018).

- [15] 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).
- [16] 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).
- [17] 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).
- [18] Docker Inc. *Best practices for writing Dockerfiles*. URL: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (besucht am 09.03.2018).
- [19] S. Huang u. a. „The HiBench benchmark suite: Characterization of the MapReduce-based data analysis“. In: *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. März 2010, S. 41–51. DOI: 10.1109/ICDEW.2010.5452747.
- [20] 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).
- [21] Bo Zhang. *Tutorial*. 10. März 2017. URL: <https://github.com/Spirals-Team/hadoop-benchmark/wiki/Tutorial> (besucht am 21.11.2017).
- [22] Jeffrey Dean und Sanjay Ghemawat. „MapReduce: Simplified Data Processing on Large Clusters“. In: *Commun. ACM* 51.1 (Jan. 2008), S. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: <http://doi.acm.org/10.1145/1327452.1327492>.
- [23] Thomas Graves. *GraySort and MinuteSort at Yahoo on Hadoop 0.23*. 2013. URL: <http://sortbenchmark.org/Yahoo2013Sort.pdf>.
- [24] Yanpei Chen, Sara Alspaugh und Randy Katz. „Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads“. In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), S. 1802–1813. ISSN: 2150-8097. DOI: 10.14778/2367502.2367519. URL: <http://dx.doi.org/10.14778/2367502.2367519>.
- [25] BARC GmbH. *Transactional Data is the Most Commonly Used Data Type in Hadoop*. URL: <https://bi-survey.com/hadoop-data-types> (besucht am 11.04.2018).
- [26] Kai Ren u. a. „Hadoop's Adolescence: An Analysis of Hadoop Usage in Scientific Workloads“. In: *Proc. VLDB Endow.* 6.10 (Aug. 2013), S. 853–864. ISSN: 2150-8097. DOI: 10.14778/2536206.2536213. URL: <http://dx.doi.org/10.14778/2536206.2536213>.

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 [15] 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 :

```

Listing A.3: Starten einer Anwendung in Hadoop-Benchmark. 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, 16, 17] zu finden.

Listing B.1: REST-Ausgabe aller Anwendungen vom RM. Die Liste kann mithilfe verschiedener Query-Parameter gefiltert werden.

URL: `http://addr: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/containerlogs/
23                          container_1518429920717_0001_01_000001/root",
24        "amHostHttpAddress": "compute-2:8042",
```

```
23     "allocatedMB": -1,
24     "allocatedVCores": -1,
25     "runningContainers": -1,
26     "memorySeconds": 1756786,
27     "vcoreSeconds": 1546,
28     "preemptedResourceMB": 0,
29     "preemptedResourceVCores": 0,
30     "numNonAMContainerPreempted": 0,
31     "numAMContainerPreempted": 0
32   }
33 ]
34 }
35 }
```

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

URL: `http://addr:port/ws/v1/applicationhistory/apps/{appid}/appattempts`

```
1 {
2   "appAttempt": [
3     {
4       "appAttemptId": "appattempt_1518429920717_0001_000001",
5       "host": "compute-2",
6       "rpcPort": 46481,
7       "trackingUrl": "http://controller:8088/proxy/
8         application_1518429920717_0001/",
9       "originalTrackingUrl": "http://controller:19888/jobhistory/job/
10         job_1518429920717_0001",
11       "diagnosticsInfo": "",
12       "appAttemptState": "FINISHED",
13       "amContainerId": "container_1518429920717_0001_01_000001"
14     }
15   ]
16 }
```