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:	3. Juni 2018
Betreuer:	M.Sc. Benedikt Eberhardinger
1. Prüfer:	Prof. Dr. X
2. Prüfer:	Prof. Dr. Y

Zusammenfassung

Abstract

Inhaltsverzeichnis

Verzeichnisse

Abbildungen

Listings

Tabellen

Abkürzungen

In dieser Masterarbeit wurden folgende Abkürzungen und Akronyme verwendet:

AM ApplicationManager

AppMstr ApplicationMaster

DCCA Deductive Cause-Consequence Analysis

HDFS Hadoop Distributed File System

MARP `maximum-am-resource-percent`

MC Model Checking

MC Model Checker

NM NodeManager

RM ResourceManager

S# Safety Sharp

SWIM Statistical Workload Injector for Mapreduce

TLS Timeline-Server

Für die genutzten Benchmarks (vgl. Kapitel 5):

dfw `TestDFSIO -write`

rtw `randomtextwriter`

tg `teragen`

dfr `TestDFSIO -read`

wc `wordcount`

rw `randomwriter`

so `sort`

tsr `terasort`

pi `pi`

pt `pentomino`

tms `testmapredsort`

tvl `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. JUnit[1] für Java und NUnit[2] für .NET zählen. Dabei werden zunächst einzelne Testfälle erstellt und können im Anschluss mit der jeweils aktuellen Codebasis jederzeit ausgeführt werden. Automatisierte Tests können auch dazu genutzt werden, um einen einzelnen Test mit verschiedenen Eingaben durchzuführen. Dadurch können verschiedene Eingabeklassen (wie negative oder positive Ganzzahlen) mit sehr geringem Aufwand in einem Test genutzt werden und somit verschiedene Testfälle direkt ausgeführt werden, wodurch eine massive Kosteneinsparung einhergeht [1].

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

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

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

2. Grundlagen und Stand der Technik

2.1. Safety Sharp

Safety Sharp (S#) ist ein am Institute for Software & Systems Engineering der Universität Augsburg entwickeltes Framework zum Testen und Verifizieren von Systemen und Modellen. Da es in C# entwickelt wurde und C# auch zum Entwickeln von Modellen und dazugehörigen Testszenarien genutzt wird, können zahlreiche Features des .NET-Frameworks bzw. der Sprache C# im Speziellen genutzt werden. S# vereint dabei die Simulation, die Visualisierung, modellbasierte Tests sowie die Verifizierung der Modelle durch einen Model Checker (MC) [3, 4]. Dadurch können alle Schritte einer vollständigen Analyse inkl. Modellierung direkt im Visual Studio ausgeführt werden und somit auch alle Features der IDE und .NET, wie z. B. die Debugging-Werkzeuge, genutzt werden. Um entsprechende Analysen zu gewährleisten, hat das Framework jedoch auch einige Einschränkungen, wodurch z. B. Schleifen und Rekursionen nur eingeschränkt bzw. nicht möglich sind. Eine der größten Einschränkungen ist allerdings, dass während der Laufzeit keine neuen Objektinstanzen innerhalb des zu testenden Modells erzeugt werden können, sodass alle benötigten Instanzen bereits während der Initialisierung des Modells erzeugt werden müssen [3].

Um nun ein System testen zu können, muss dieses zunächst mithilfe von C#-Klassen und -Instanzen modelliert werden. Die dafür verwendeten Modelle sind meist stark vereinfacht und bilden nur die wesentlichen Aspekte der realen Systeme ab. Für einen korrekten Test ist es jedoch wichtig, dass das Modell des Systems vergleichbar mit dem echten System ist.

Folgendes Beispiel zeigt den typischen, grundlegenden Aufbau einer S#-Komponente:

```
1  public class YarnNode : Component
2  {
3    // fault definition, also possible: new PermanentFault()
4    public readonly Fault NodeConnectionError = new TransientFault();
5
6    // interaction logic (Fields, Properties, Methods...)
7
8    // fault effect
9    [FaultEffect(Fault = nameof(NodeConnectionError))]
10   internal class NodeConnectionErrorEffect : YarnNode
11   {
12     // fault effect logic
13   }
14 }
```

Listing 2.1: Grundlegender Aufbau einer S#-Komponente

Jede Komponente des Modells muss von `Component` erben, um als S#-Komponente definiert zu sein. Jede Komponente kann nun temporäre (`TransientFault`) oder dauerhafte (`PermanentFault`) Komponentenfehler enthalten, welche zunächst innerhalb der Komponente als Felder definiert werden. Der Effekt eines Komponentenfehlers wird anschließend in der entsprechenden Effekt-Klasse definiert, welche von der Hauptklasse (hier `YarnNode`) erbt und mithilfe des Attributs `FaultEffectAttribute` dem dazugehörigen Komponentenfehler zugeordnet wird [4].

Um die Modelle zu testen, kommen in S# verschiedene Werkzeuge zum Einsatz. Eines davon ist eine reine Simulation, bei der das Framework nur einen Ausführungspfad ausführt und dabei keine Komponentenfehler aktiviert bzw. die Aktivierung *manuell* gesteuert werden kann. Ein weiterer Nutzen liegt in der Möglichkeit, im ausgeführten Ausführungspfad zeitliche Abläufe zu berücksichtigen, da hier das Modell Schritt für Schritt ausgeführt wird. Hierbei wird für jede im Modell genutzte Komponente pro Schritt einmal die jeweilige Methode `Update()` aufgerufen, in der die jeweiligen Komponenten ihre Aktivitäten durchführen [4].

Ein anderes wichtiges Werkzeug von S# ist die Deductive Cause-Consequence Analysis (DCCA), welche eine vollautomatische und MC-basierte Sicherheitsanalyse ermöglicht. Dabei wird selbstständig die Menge der aktivierten Komponentenfehler ermittelt, mit denen das Gesamtsystem nicht mehr rekonfiguriert werden kann und somit ausfällt. Je nach Konfiguration können dazu auch Heuristiken genutzt werden, welche die Analyse beschleunigen und genauer machen können [5]. Dabei werden die verschiedenen aktivierten Komponentenfehler während der Analyse in tolerierbare und nicht-tolerierbare Fehler unterschieden. Tolerierbare Komponentenfehler werden dazu genutzt, die Grenzen der Selbstkonfiguration des Systems zu ermitteln. Dabei wird für jeden Systemzustand nach einer Rekonfiguration durch die DCCA eine neue Fehlermenge ermittelt, mit der das System gerade noch lauffähig ist. Das Auftreten eines tolerierbaren Komponentenfehler ist also gleichbedeutend mit einem einfachen Fehler im System, welcher die gesamte Funktionsweise des Systems nicht massiv einschränkt und eine Rekonfiguration noch ermöglicht. Sobald jedoch ein Fehler auftritt, durch den eine Rekonfiguration des Systems nicht mehr möglich ist, wurde ein nicht-tolerierbarer Fehler gefunden, durch den das System nicht mehr funktionsfähig ist [3].

Das dritte Werkzeug zur Ausführung von Modellen in S# ist der MC selbst. Hierbei kann der in S# bereits enthaltene, oder alternativ *LTSmin*[1] genutzt werden [6]. Beim MC werden in einem *brute-force*-ähnlichem Verfahren alle möglichen Zustände und Ausführungspfade in einem Modell mit einer endlichen Anzahl an Zuständen getestet.

[1] `http://ltsmin.utwente.nl/`

Dadurch wird es ermöglicht, verschiedene Eigenschaften eines System zu testen und Fehler (z. B. Deadlocks) zu erkennen [2].

2.2. Apache Hadoop

Apache™Hadoop®[2] ist ein Open-Source-Software-Projekt, welches die Verarbeitung von großen Datenmengen auf einem verteilten System ermglicht. Hadoop wird von der *Apache Foundation* entwickelt und enthält verschiedene vollständig skalierbare Komponenten. Es ist daher möglich, ein Hadoop-Cluster auf nur einem einzelnen PC, aber auch verteilt auf ein ganzen Serverzentrum auszuführen. Hadoop besteht aus folgenden Kernmodulen [7]:

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 Anwendungen auszuführen, um große Datenmengen zu verarbeiten. Die für das Cluster verfügbaren Ressourcen beschränken sich lediglich auf die Summe der verfügbaren Ressourcen aller Hosts, auf denen das Cluster ausgeführt wird.

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. Dem gegenüber stehen die Slave-*Nodes*, auf denen die Anwendungen ausgeführt werden:

Der RM 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.

Der NodeManager (NM) eines jeden Nodes überwacht die Ressourcen seines jeweiligen Nodes sowie der auf dem Node ausgeführten Anwendungs-Container und übermittelt diese dem RM.

Jede YARN-Anwendung bzw. Job besteht aus einer oder mehreren Ausführungsinstanzen, genannt *Attempts*. Die eigentliche Ausführung einer Anwendung findet in den bereits erwähnten *Containern* statt, die jeweils einem Attempt zugeordnet sind.

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

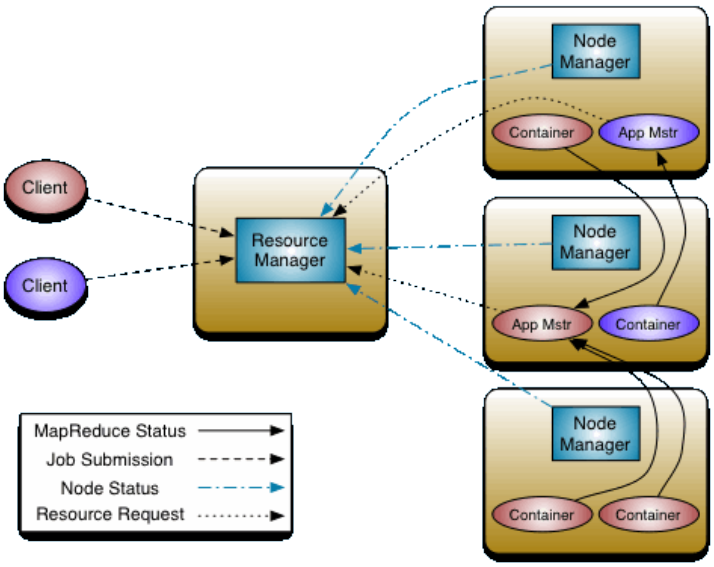

Abbildung 2.1.: Architektur von YARN (entnommen aus [8])

Container können auf einem beliebigen Node ausgeführt werden und repräsentieren die Ausführung eines Tasks innerhalb der Anwendung. Ein besonderer Container bildet hierbei der ApplicationMaster (AppMstr). Er übernimmt das Monitoring der Anwendung und die Kommunikation mit dem RM und NM und stellt die dafür benötigten Informationen bereit [8].

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

Ein weiterer Bestandteil von Hadoop bzw. YARN ist der Timeline-Server (TLS). Er ist speziell dafür entwickelt, die Metadaten und Logs der YARN-Anwendungen zu speichern und jederzeit, also als Anwendungshistorie, auszugeben [9].

Das **HDFS** basiert auf der gleichen Architektur wie YARN und besitzt ebenfalls einen Master und mehrere Slaves:

Der *NameNode* dient als Master für die Verwaltung des Dateisystems und reguliert den Zugriff auf die darauf gespeicherten Daten. Unterstützt wird der NameNode vom *Secondary NameNode*, der Teile der internen Datenverwaltung des HDFS durchführt [11]]. 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 die gespeicherten Daten gelesen und beschrieben werden können [10].

MapReduce bietet analog zu YARN die Möglichkeit, Anwendungen mit einem großen Ressourcenbedarf 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.

Literatur für weitere Infos zu MR?

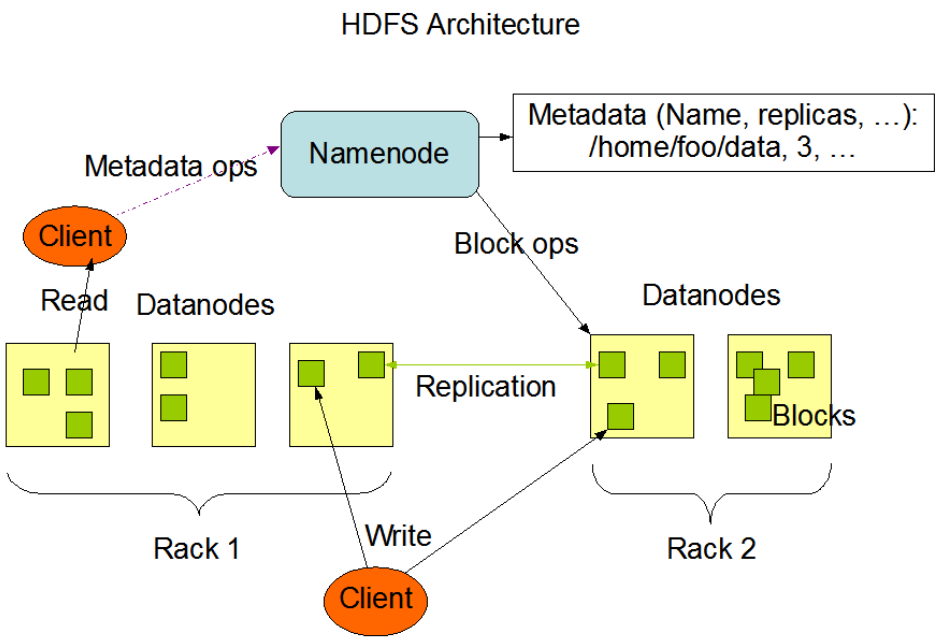

Abbildung 2.2.: Architektur des HDFS (entnommen aus [10])

Für die Ein- und Ausgabe der Daten wird in der Regel das HDFS, für die Ausführung der einzelnen Tasks YARN genutzt [12]. Die MapReduce-Komponente von Hadoop 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 [8, 13].

2.3. Adaptive Komponente in Hadoop

Text besser an neue unterabschnitte anpassen

Eine normale Hadoop-Installation besitzt keine adaptive Komponente, sondern rein statische Einstellungen. Um damit Hadoop zu optimieren, müssen die Einstellungen immer manuell auf den jeweils benötigten Anwendungstyp angepasst werden. Dazu gibt es auch bereits verschiedene Scheduler, den *Fair Scheduler*, welcher alle Anwendungen ausführt und ihnen gleich viele Ressourcen zuteilt, und den *Capacity Scheduler*. Letzterer sorgt dafür, dass nur eine bestimmte Anzahl an Anwendungen pro Benutzter gleichzeitig ausgeführt wird und teilt ihnen so viele Ressourcen zu, wie benötigt werden bzw. der Benutzter nutzen darf. Entwickelt wurde der Capacity Scheduler vor allem für Cluster, die von mehreren Organisationen gemeinsam verwendet werden und sicherstellen soll, dass jede Organisation eine Mindestmenge an Ressourcen zur Verfügung hat [14].

2.3.1. MARP-Werte

Je nach Bedarf besitzt der Capacity Scheduler entsprechende Einstellungen, um z. B. den verfügbaren Speicher pro Container festzulegen. Eine weitere Einstellung des

Schedulers ist `maximum-am-resource-percent`, auch MARP gennant, der angibt, wie viele Prozent der gesamten Ressourcen durch AppMstr-Container genutzt werden dürfen [14]. 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 [15] 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 können Anwendungen im Schnitt um bis zu 40 % schneller ausgeführt [15] werden.

Der Hintergrund dieser *Selfbalancing-Komponente* ist der, dass durch den MARP-Wert der für die Anwendungen verfügbare Speicher in zwei Teile aufgeteilt wird. Im einen Teil befinden sich alle derzeit ausgeführten AppMstr, im anderen Teil die von den Anwendungen benötigten weiteren Container. Wie groß der Teil für die AppMstr ist, wird nun durch den MARP-Wert bestimmt. Ist der MARP-Wert zu klein, können nur wenige AppMstr (und damit Anwendungen) gleichzeitig ausgeführt werden (*Loss of Jobs Parallelism*, LoJP). Ist der MARP-Wert jedoch zu groß, können für die ausgeführten Anwendungen nur wenige Container bereitgestellt werden, wodurch sich die Ausführung für eine Anwendung wesentlich verlangsamt (*Loss of Job Throughput*, LoJT)[15]:

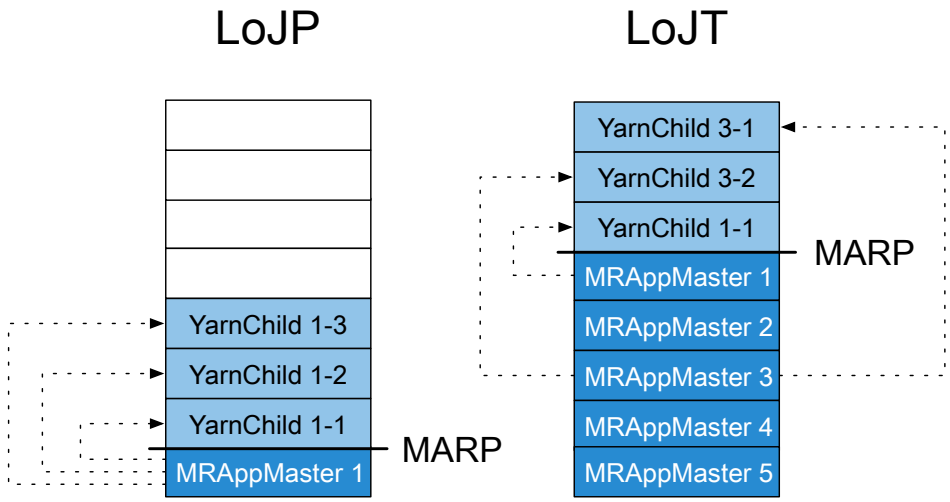

Abbildung 2.3.: LoJP und LoJT in Hadoop (entnommen aus [15]). Während beim LoJP sehr viel Speicher für Anwendungscontainer ungenutzt bleibt, können beim LoJT nicht genügend Anwendungscontainer allokiert werden, um die Anwendungen auszuführen.

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 [15]. 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.

2.3.2. Analyse der Selfbalancing-Komponente

Da in dieser Fallstudie auch Mutationstests zum Einsatz kommen, bei denen die Selfbalancing-Komponente entsprechend verändert wird (vgl. Abschnitt 6.2), wurde die Komponente zunächst analyisiert. Sie besteht aus folgenden vier Java-Klassen und drei Shell-Scripten:

- Java-Klassen:
  - `controller.Controller`
  - `effectuator.Effectuator`
  - `monitor.ControlNodeMonitor`
  - `monitor.MemUtilization`
- Shell-Scripte:
  - `selfTuning-CapacityScheduler.sh`
  - `selfTuning-controlNode.sh`
  - `selfTuning-mem-controlNode.sh`

Um Messfehler auszugleichen bzw. zu reduzieren nutzt die Komponente zudem einen Kalman-Filter, welcher in Form der Open-Source-Bibliotek JKalman[3] eingebunden ist.
literatur zum kalman?

Die drei Shell-Scripte dienen zur Interaktion zwischen dem Controller der Selfbalancing-Komponente und Hadoop selbst. Sie werden von den beiden Monitor-Klassen sekündlich gestartet und ermitteln basierend auf den Logs die Auslastung des Clusters. Mithilfe von `selfTuning-controlNode.sh`, das von `ControlNodeMonitor` gestartet wird, wird die Anzahl an aktiven und wartenden YARN-Jobs ermittelt und anschließend in die `controlNodeLog`-Datei geschrieben. Durch die Ausführung von `selfTuning-mem-controlNode.sh` (gestartet durch `MemUtilization`) wird dagegen die Auslastung des Arbeitsspeichers ermittelt und in die `memLog`-Datei geschrieben.

Die in den beiden Dateien enthaltenen Werten im Anschluss wiederum sekündlich vom `Controller` ausgelesen und mithilfe des Kalman-Filters bereinigt. Anschließend werden die bereits in [15] vorgestellten Algorithmen zum Ermitteln des neuen MARP-Wertes ausgeführt, damit dieser entsprechend erhöht bzw. verringert wird.

Um den dadurch neu ermittelten MARP-Wert anzuwenden, wird abschließend mithilfe des `Effectuators` das dritte Shell-Script `selfTuning-CapacityScheduler.sh` ausgeführt. Mithilfe dieses Shell-Scriptes wird der neue MARP-Wert in der Konfiguration des *Capacity Schedulers* gespeichert.

[3] `https://jkalman.sourceforge.io/`

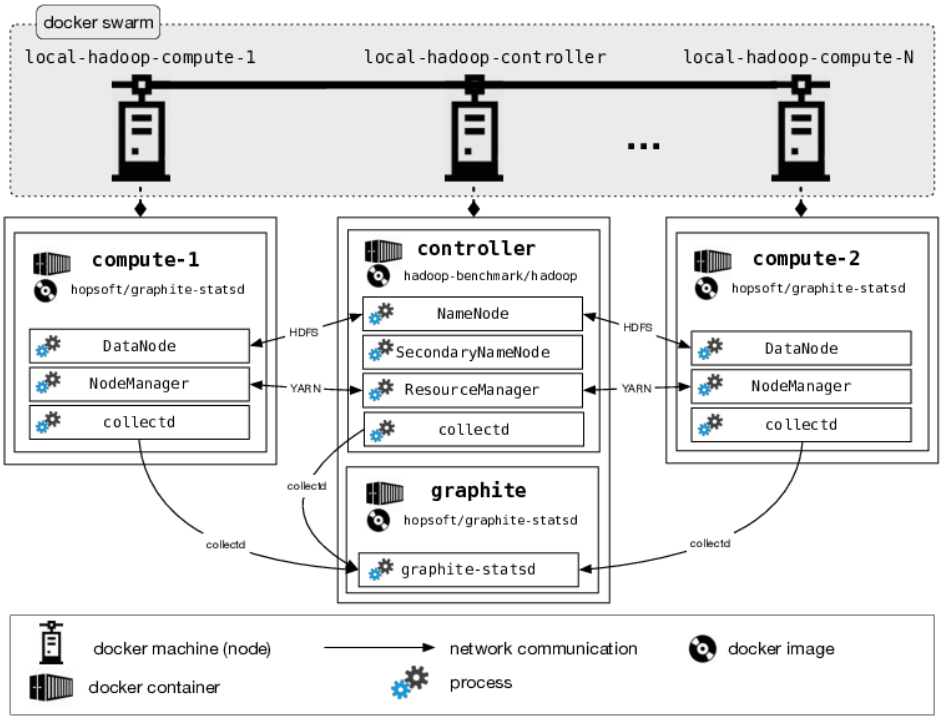

Abbildung 2.4.: High-Level-Architektur von Hadoop-Benchmark. Entnommen aus [16].

2.4. Plattform Hadoop-Benchmark

Zhang u. a. haben im Rahmen ihrer gesamten Forschungsarbeit an der Selfbalancing-Komponente die Open-Source-Plattform Hadoop-Benchmark entwickelt[4]. Sie wurde speziell zum Einsatz in der Forschung erstellt und kann jederzeit an die eigenen Bedürfnisse angepasst werden.

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*[5] und dem dazugehörigen Tool *Docker Machine*, um damit mit wenigen Befehlen ein Hadoop-Cluster aufbauen zu können. Mit *Graphite*[6] 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 VirtualBox[7] virtuelle Maschinen erstellt, die mit dem Betriebssystem *Boot2Docker* ausgestattet sind. Boot2Docker ist eine leichtgewichtige Linux-Distribution, auf der Docker bereits vorinstalliert ist [17]. Jeder

[4] https://github.com/Spirals-Team/hadoop-benchmark
[5] https://www.docker.com/
[6] https://graphiteapp.org/
[7] https://www.virtualbox.org/

Hadoop-Container enthält zudem das Tool *collectd*[8], 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.

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

- Hadoop Mapreduce Examples
- Intel HiBench[9]
- Statistical Workload Injector for Mapreduce (SWIM) [10]

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.

[8] `https://collectd.org/`
[9] `https://github.com/intel-hadoop/HiBench`
[10] `https://github.com/SWIMProjectUCB/SWIM`

3. Aufbau der Fallstudie

Im Rahmen dieser Masterarbeit soll nun mithilfe von Hadoop und der Selfbalancing-Komponente eine Fallstudie durchgeführt werden, durch die ermittelt wird, unter welchen Umständen eine Testautoamtisierung möglich ist. Dazu werden zunächst Anforderungen an das Hadoop-System selbst gestellt, die es im Rahmen dieser Tests erfüllen soll. Neben diesen funktionalen Anforderungen werden aber auch Anforderungen an das Testsystem selbst gestellt, die durch die Tests selbst erfüllt sein sollen. Zur Realisierung dieser Tests wird für das reale Cluster die von Zhang u. a. entwickelte Plattform Hadoop-Benchmark genutzt.

3.1. Funktionale Anforderungen an das Cluster

3.2. Anforderungen an das Testsystem

3.2.1. Behauptungen und Variablen

3.2.2. Generierung der Testfälle

3.2.3. Organisation der Daten

3.3. Umsetzung des realen Clusters

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 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.

Irgendwo erwähnen, wie Docker-Container aufeinander aufbauen

Die beiden Abbildungen Abbildung 2.4 und Abbildung 3.1 zeigen bereits den großen Hauptunterschied zwischen der Plattform und dem hier verwendeten Cluster-Setup. Da durch die Nutzung von virtuellen Maschinen ein zusätzlicher Ressourcenbedarf

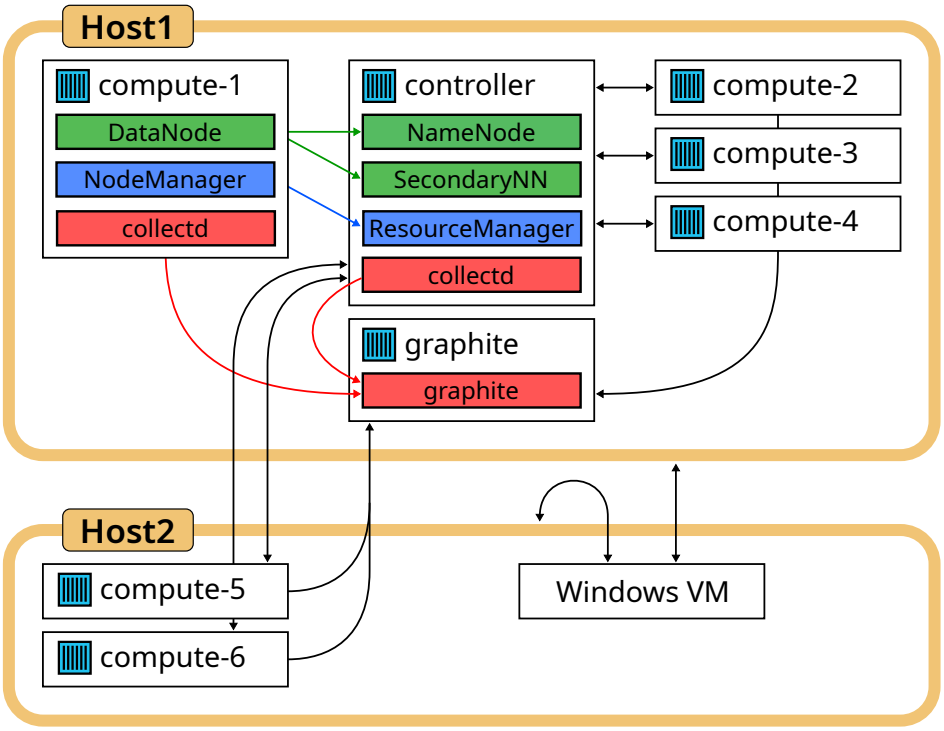

Abbildung 3.1.: In der Fallstudie verwendetes Cluster-Setup. Grün: HDFS, Blau: YARN, Rot: Graphite.

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 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*[1] 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

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

- 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.

4. Aufbau des Modells

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

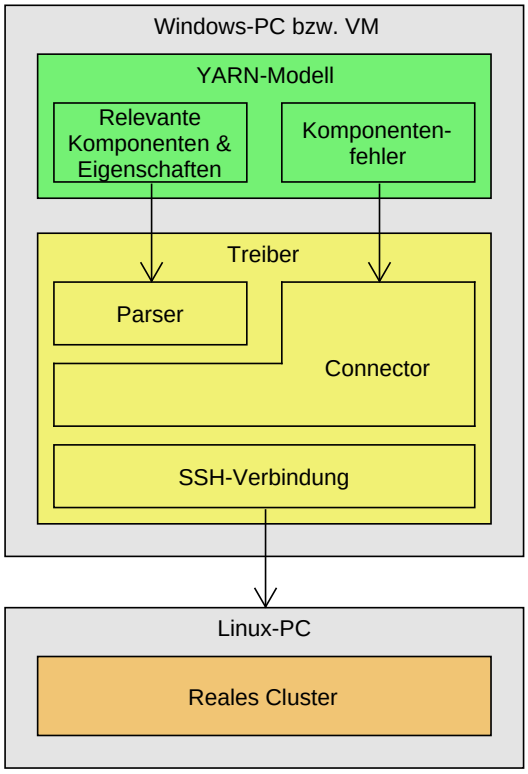

Abbildung 4.1.: Grundlegende Architektur des Gesamtmodells

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

Connector Abstrahierung der SSH-Verbindung mit den auszuführenden Befehlen

SSH-Verbindung Verbindung zum Cluster-PC

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

Multihost-Mode irgendwo erklären

4.1. 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

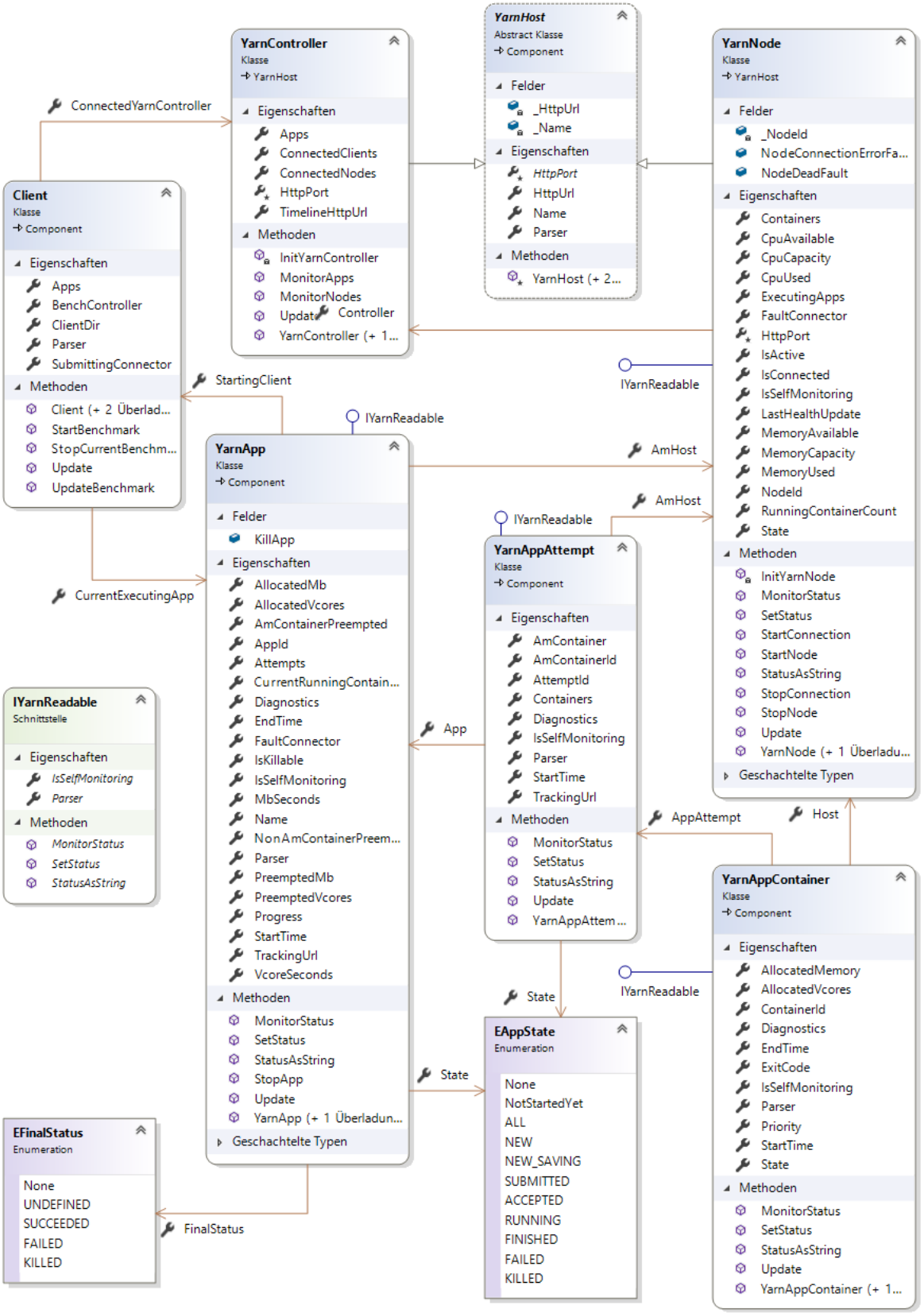

Abbildung 4.2.: Aufbau des YARN-Modells. Das Modell wurde mithilfe des Klassendiagramm-Designers in Visual Studio 2017 visualisiert. Daher werden Assoziationen mit höherer Multiplizität als 1, die daher mithilfe von `List<T>` umgesetzt wurden (z. B. `YarnApp.Attempts`) im Diagramm nicht als Assoziationen zwischen den Klassen angezeigt.

Abbildung 4.2 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.

4.1.1. 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 `BenchControllers` (vgl. Abschnitt 5.3) 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.

4.1.2. 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 4.1 zeigt beispielhaft die Implementierung und Injizierung des `NodeDead`-Komponentenfehlers mithilfe des für den Node verwendeten `CmdConnector` (vgl. Unterabschnitt 4.2.3). Die Injizierung des `NodeConnectionError`-Komponentenfehlers und die Aufhebung beider Komponentenfehler sind analog implementiert.

4.1.3. 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 4.2 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`.

```
public class YarnNode : YarnHost, IYarnReadable
{
  public readonly Fault NodeDeadFault = new TransientFault();
  public IHadoopConnector FaultConnector { get; set; }

  public bool StopNode(bool retry = true)
  {
    if(IsActive)
    {
      var isStopped = FaultConnector.StopNode(Name);
      if(isStopped)
        IsActive = false;
      else if(retry)
        StopNode(false); // try again once
    }
    return !IsActive;
  }

  [FaultEffect(Fault = nameof(NodeDeadFault))]
  public class NodeDeadEffect : YarnNode
  {
    public override void Update()
    {
      StopNode();
    }
  }
}

public class CmdConnector : IHadoopConnector
{
  private SshConnection Faulting { get; }

  public bool StopNode(string nodeName)
  {
    var id = DriverUtilities.ParseInt(nodeName);
    Faulting.Run($"{Model.HadoopSetupScript} hadoop stop {id}",
        IsConsoleOut);
    return !CheckNodeRunning(id);
  }
}
```

Listing 4.1: Injizierung eines Komponentenfehlers (gekürzt). Sollte der Node nicht beendet werden, wird die Injizierung einmalig erneut versucht. `CmdConnector.Faulting` ist der für Komponentenfehler verwendete Connector.

```
public Func<bool>[] Constraints => new Func<bool>[]
{
  () =>
  {
    if(FinalStatus != EFinalStatus.FAILED) return true;
    if(!String.IsNullOrWhiteSpace(Name) && Name.ToLower().Contains("
       fail job")) return true;
    return false;
  },
  () =>
  {
    if(State == EAppState.RUNNING)
      return AmHost?.State == ENodeState.RUNNING;
    return true;
  },
};
```

Listing 4.2: 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 allokieren 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.

4.2. 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.

4.2.1. 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 [9, 18–20]:

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 `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 [9, 19]. 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.

4.2.2. 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 [9, 19, 20], wodurch diese mithilfe des *Json.NET*-Frameworks[1] deserialisiert und direkt als die entsprechende `IParsedComponent`-Implementierung gespeichert werden. Da RM und TLS verschiedene Daten einer YARN-Komponente ausgeben, werden, sofern nötig, RM und TLS abgefragt und die dabei ermittelten Daten zusammengeführt.

[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 [9, 19, 20], wird beim Parsen eines Ausführungs- oder Container-Reports die komplette Liste abgerufen und geparst. Anschließend wird in dieser Liste basierend auf der ID die benötigte Komponente herausgefiltert.

Die zweite Besonderheit bei der Nutzung der REST-API liegt darin, dass die Daten zu derzeit ausgeführten Container ausschließlich vom NM, auf dem der Container ausgeführt wird, zurückgegeben werden können [19, 20]. Daher werden zur Ermittlung der Container-Listen alle Nodes abgefragt und anschließend die benötigten Container gefiltert.

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

4.2.3. Implementierte Connectoren

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

Da über die Kommandozeile die Befehle für die Daten vom TLS die gleichen wie für die Daten vom RM sind [9, 18], 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.

4.2.4. 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`

5. 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.

5.1. Übersicht möglicher Anwendungen

Hadoop-Benchmark enthält bereits die Möglichkeit, unterschiedliche Benchmarks zu starten. Wie in Abschnitt 2.4 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 [21]. 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 [22] 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 [23]. Bei der Ausführung auf dem in dieser Fallstudie verwendetem Cluster wurden jedoch nicht alle Workloads fehlerfrei ausgeführt. Zudem wird in [24] 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)

5.2. 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 5.1 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 [25]. `terasort` ist ebenfalls ein weit verbreiteter Benchmark, der die Hadoop-Implementierung der standardisierten *Sort Benchmarks*[1] darstellt [26]. 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 [22, 27].

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. [27] und [28]. 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 [29] und folgerten daher, dass für kleine Jobs evtl. einfachere Frameworks abseits von Hadoop besser geeignet wären. Die Autoren der Studie in [28] 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“ verarbeiten 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` (tvl) 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 genieren 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`

	dfw	*rtw*	*tg*	*dfr*	*wc*	*rw*	*so*	*tsr*	*pi*	*pt*	*tms*	*tvl*	*sl*	*fl*
dfw	.600	.073	0	.145	0	0	0	0	.073	.073	0	0	.018	.018
rtw	.036	.600	0	0	.145	.036	.109	0	.036	0	0	0	.019	.019
tg	0	.036	.600	0	0	0	0	.255	0	.073	0	0	.018	.018
dfr	0	.073	0	.600	0	.036	0	0	.145	.109	0	0	.018	.019
wc	.073	.109	0	0	.600	0	.073	0	.073	.036	0	0	.018	.018
rw	0	.073	.073	0	0	.600	0	0	.109	.109	0	0	.018	.018
so	0	.073	.036	0	.073	.036	.600	0	.073	0	.073	0	.018	.018
tsr	0	0	0	0	0	0	0	.600	.109	.073	0	.182	.018	.018
pi	.145	.109	0	0	0	0	0	0	.600	.109	0	0	.018	.019
pt	.109	.109	0	0	0	.073	0	0	.073	.600	0	0	.018	.018
tms	0	.145	0	0	0	.073	0	0	.036	.109	.600	0	.018	.019
tvl	.073	.109	0	0	0	0	0	0	.109	.073	0	.600	.018	.018
sl	.167	.167	.167	0	0	.167	0	0	.167	.167	0	0	0	0
fl	.167	.167	.167	0	0	.167	0	0	.167	.167	0	0	0	0

Tabelle 5.1.: Verwendete Markov-Kette für die Anwendungs-Übergänge in Tabellenform.

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

Für die in Tabelle 5.1 dargestellte Markow-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.

5.3. 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 5.1 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 5.1 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 5.2 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 5.1 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

```
public class Benchmark
{
  public const string BaseDirHolder = "$DIR";
  public const string OutDirHolder = "$OUT";
  public const string InDirHolder = "$IN";

  public Benchmark(int id, string name, string startCmd, string
      outputDir, string inputDir)
  {
    _StartCmd = startCmd;
    _InDir = inputDir;
    HasInputDir = true;
  }

  public string GetStartCmd(string clientDir = "")
  {
    var result = _StartCmd.Replace(OutDirHolder, GetOutputDir(
        clientDir)).Replace(InDirHolder, GetInputDir(clientDir));
    if(result.Contains(BaseDirHolder))
      result = ReplaceClientDir(result, clientDir);
    return result;
  }
}

using static Benchmark;
public class BenchmarkController
{

  public static Benchmark[] Benchmarks { get; } // benchmarks
  public static int[][] BenchTransitions { get; } // transitions

  static BenchmarkController()
  {
    Benchmarks = new[]
    {
      new Benchmark(04, "wordcount", $"example wordcount {InDirHolder}
          {OutDirHolder}", $"{BaseDirHolder}/wcout", $"{BaseDirHolder
         }/rantw"),
    };
  }
}

public class Client : Component
{
  public string StartBenchmark(Benchmark benchmark)
  {
    if(benchmark.HasOutputDir)
      SubmittingConnector.RemoveHdfsDir(benchmark.GetOutputDir(
          ClientDir));
    var appId = SubmittingConnector.StartApplicationAsync(benchmark.
        GetStartCmd(ClientDir));
  }
}
```

Listing 5.1: Definition und Start einer Anwendung (gekürztes Beispiel). Die Generierung des komplettes Startbefehls mit Nutzung des Benchmark-Scriptes führt der vom Client verwendete Connector durch, weshalb hier nur definiert werden muss, dass das Example-Programm `wordcount` gestartet wird.

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

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

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

Listing 5.2: Normalisierung und Auswahl der nachfolgenden Anwendung (gekürzt)

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

6. Implementierung und Ausführung der Tests

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.

6.1. 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 Komponentenfehler ausgeführt wird, sowie der Analysetest, bei dem Komponentenfehler aktiviert werden.

6.1.1. 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.

```
public void SimulateHadoop()
{
  ModelSettings.FaultActivationProbability = 0.0;
  ModelSettings.FaultRepairProbability = 1.0;

  ExecuteSimulation();
}

private void ExecuteSimulation(bool isWithFaults)
{
  var origModel = InitModel();

  var wasFatalError = false;
  try
  {
    var simulator = new SafetySharpSimulator(origModel);
    var simModel = (Model)simulator.Model;
    var faults = CollectYarnNodeFaults(simModel);

    SimulateBenchmarks();

    for(var i = 0; i < _StepCount; i++)
    {
      OutputUtilities.PrintStepStart();
      var stepStartTime = DateTime.Now;

      if(isWithFaults)
        HandleFaults(faults);
      simulator.SimulateStep();

      var stepTime = DateTime.Now - stepStartTime;
      OutputUtilities.PrintSteptTime(stepTime);
      if(stepTime < ModelSettings.MinStepTime)
        Thread.Sleep(ModelSettings.MinStepTime - stepTime);

      OutputUtilities.PrintFullTrace(simModel.Controller);
    }

    OutputUtilities.PrintExecutionFinish();
  }
  // catch/finally
}
```

Listing 6.1: Simulation in dieser Fallstudie (gekürzt).

Listing 6.1 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 `Exception`s Nodes mit injizierten Komponentenfehlern neu gestartet.

6.1.2. Initialisierung des Modells

Bevor das Modell im Simulator ausgeführt werden kann, muss es initialisiert werden. Das folgende Listing 6.2 zeigt die Definition der Felder zur Modellinitialisierung sowie die entsprechenden Methoden, die in Listing 6.1 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 6.2: Initialisierung des Modells für die Simulation

Die einzelnen Felder 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 Felder haben lediglich einen Zweck, während andere umfangreichere Auswirkungen besitzen. Die einfachen Felder 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 Komponentenfehlern. Ist dieser Wert 0,0, werden grundsätzlich keine Komponentenfehler aktiviert, bei einem Wert von 1,0 werden Komponentenfehler dagegen immer aktiviert.

_FaultRepariProbability Definiert die generelle Häufigkeit zum Deaktivieren von Komponentenfehlern. Die hier definierte Wahrscheinlichkeit verhält sich analog zu `_FaultActivationProbability`. Bei einem Wert von 0,0 werden Komponentenfehler 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 Eingabeverzeichnisse für die Anwendungen die Ausgabeverzeichnisse 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 Komponentenfehler, die mit dem `NodeFaultAttribute` markiert sind. Die gefundenen Komponentenfehler werden als Array aus Tupel, bestehend aus dem Komponentenfehler selbst, dem Attribut sowie dem dazugehörigen Node zurückgegeben. Die jeweiligen Instanzen der Attribute und Nodes werden für die in Abschnitt 6.1.3 beschriebene Aktivierung der dazugehörigen Komponentenfehler benötigt.

6.1.3. Ablauf eines Simulations-Schrittes

Der Ablauf eines Schrittes lässt sich in die folgenden fünf Abschnitte einteilen. Während die Aktivierung und Deaktivierung der Komponentenfehler komplett außerhalb des Modells und durch den Simulator erfolgt (durch die in Listing 6.1 aufgerufene `HandleFaults()`-Methode), werden die anderen Abschnitte durch die `Update()`-Methode des `YarnControllers` innerhalb des Modells während der Ausführung eines Simulations-Schrittes ausgeführt. Die Ausgaben während eines Schrittes werden dagegen gemischt 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.

Ob ein Komponentenfehler aktiviert wird, wird folgendermaßen anhand dieser Parameter berechnet:

```
1  var node = Nodes.First(n => n.Name == nodeName);
2  var nodeUsage = (node.MemoryUsage + node.CpuUsage) / 2;
3
4  if(nodeUsage < 0.1) nodeUsage = 0.1;
5  else if(nodeUsage > 0.9) nodeUsage = 0.9;
6
7  NodeUsageOnActivation = nodeUsage; // for using on repairing
8
9  var faultUsage = nodeUsage * ActivationProbability * 2;
10
11 var probability = 1 - faultUsage;
12 var randomValue = RandomGen.NextDouble();
13 Logger.Info($"Activation probability: {probability} < {randomValue}");
14 return probability < randomValue;
```

Listing 6.3: Berechnung der Aktivierung von Komponentenfehlern (zusammengefasst).

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, welche hierzu in Zeile 7 in Listing 6.3 entsprechend gespeichert wird. Der grundlegende Algorithmus zur Entscheidung ist jedoch gleich.

Ausführung Benchmarks

Damit die Ausführung der Benchmarks vor dem Monitoring der Anwendungen sowie dem Auswerten der Constraints durch das Oracle stattfindet, wird die Ausführung

der Benchmarks ebenfalls durch den `YarnController` gestartet. Dazu wird für jeden Client die entsprechende Methode aufgerufen, welche ihrerseits den in Abschnitt 5.3 erläuterten `BenchmarkController` nutzt, um den folgenden Benchmark zu bestimmen und im Falle eines Wechsels des Benchmarks diesen zu starten:

```
1  public void UpdateBenchmark()
2  {
3    var benchChanged = BenchController.ChangeBenchmark();
4
5    if(benchChanged)
6    {
7      StopCurrentBenchmark();
8      StartBenchmark(BenchController.CurrentBenchmark);
9    }
10 }
```

Listing 6.4: Auswahl und Start des nachfolgenden Benchmarks (gekürzt). Die Methode `BenchmarkController.ChangeBenchmark()` ist in Listing 5.2 zu sehen.

Da ein Client auf dem Cluster nur eine Anwendung gleichzeitig ausführt, wird zunächst der zuvor ausgeführte Benchmark abgebrochen. Bevor der neue Benchmark im Anschluss auf dem Cluster gestartet werden kann, wird zunächst geprüft, ob das Ausgabeverzeichnis der Anwendung im HDFS vorhanden ist und gelöscht, da die Anwendung auf dem Cluster andernfalls nicht gestartet werden kann. Beim Starten der zum Benchmark zugehörigen Anwendung wird zunächst solange gewartet, bis der Anwendung vom RM eine *Application ID* zugewiesen wurde, da diese in einer `YarnApp`-Instanz sowie in `Client.CurrentExecutingAppId` gespeichert wird. Sollte keine `YarnApp`-Instanz mehr verfügbar sein, wird stattdessen eine `OutOfMemoryException` ausgelöst, da während der Simulation keine neuen Instanzen erzeugt werden dürfen.

Monitoring der ausgeführten Anwendungen

Bevor das Monitoring der Anwendungen durchgeführt wird, wird zunächst drei Sekunden gewartet, bis der AppMstr sowie weitere Container der Anwendung gestartet wurden. Die Wartezeit ist prinzipiell optional, wird hier jedoch genutzt, damit die Auslastung des Clusters besser ermittelt werden kann.

Beim Monitoring werden zunächst die Daten der Nodes, danach die der Anwendungen, ihrer Attempts und zum Abschluss deren Container ermittelt. Für das Monitoring selbst gib es zwei Ausführungsvarianten. Die eine Variante liegt darin, dass jede `IYarnComponent` (also Nodes, Anwendungen, Attempts und Container) jeweils ihre eigenen Daten ermittelt. Entwickelt wurde diese Variante vor allem für das Monitoring durch die entsprechenden Kommandozeilen-Befehle. Die zweite Variante, welche optimal zur Nutzung der REST-API von Hadoop ist, liegt darin, dass die jeweils übergeordnete Komponente alle Daten für all ihre jeweils untergeordneten Komponenten ermittelt und

zur Speicherung übergibt. Unterschieden werden die beiden Variante durch die Variable `IYarnComponent.IsSelfMonitoring`:

```
public void MonitorStatus()
{
  if(IsSelfMonitoring)
  {
    var parsed = Parser.ParseAppDetails(AppId);
    if(parsed != null)
      SetStatus(parsed);
  }

  var parsedAttempts = Parser.ParseAppAttemptList(AppId);
  foreach(var parsed in parsedAttempts)
  {
    var attempt = // get existing or empty attempt
    if(attempt == null)
      // throw OutOfMemoryException

    attempt.AppId = AppId;
    attempt.IsSelfMonitoring = IsSelfMonitoring;
    if(IsSelfMonitoring)
      attempt.AttemptId = parsed.AttemptId;
    else
    {
      attempt.SetStatus(parsed);
      attempt.MonitorStatus();
    }
  }
}
```

Listing 6.5: Monitoring der Anwendungen (gekürzt). Wenn `IsSelfMonitoring` auf `false` gesetzt ist, werden die Daten der Anwendung selbst bereits vom `YarnController` ermittelt und mithilfe von `YarnApp.SetStatus` gespeichert, analog zu den Attempts, deren Status hier bereits gespeichert wird.

Die `OutOfMemoryException` im vorangegangenen Listing 6.5 ist analog zur gleichen Ausnahme beim Starten der Anwendung und wird dann ausgelöst, wenn bereits alle `YarnAppAttempt`-Instanzen für diese Anwendung belegt sind.

Das Monitoring der Container bietet eine Besonderheit. Während bei Anwendungen und Attempts auch die Daten von beendeten Anwendungen ermittelt und gespeichert werden, ist dies bei beendeten Containern nicht der Fall. Das Monitoring für Container wird nur für derzeit aktive Container durchgeführt. Dies schließt die Nutzung des TLS von Hadoop ein. Während bei den Anwendungen und Attempts auch solche, deren Daten ausschließlich beim TLS gespeichert sind, ermittelt werden, werden die Daten des TLS bei Containern nur als Ergänzung der Daten von derzeit ausgeführten Containern vom

RM genutzt. Zudem werden bei jedem Schritt die zuvor ausgeführten Container gelöscht bzw. deren Instanzen geleert, bevor die Daten der derzeit ausgeführten Container ermittelt werden.

Prüfungen durch das Oracle

Im Anschluss an das Monitoring erfolgt die Prüfung der Constraints durch das Oracle. Das Oracle validiert hierbei analog zum Monitoring zunächst die Nodes und danach die Anwendungen, Attempts und Container auf ihre Constraints. Hierbei wird zunächst überprüft, ob die in Abschnitt 3.1 beschrieben funktionalen Anforderungen an Hadoop in Form der Constraints für die jeweiligen Komponenten noch erfüllt werden können. Ist das nicht der Fall, wird de geloggt und die weiteren Komponenten geprüft.

Das Oracle überprüft auch, ob für das Cluster eine weitere Rekonfiguration möglich ist. Dies ist dann der Fall, wenn noch mindestens ein Node vorhanden ist, der keine Fehler aufweist und damit den *State* `Running` hat:

```
1  public bool IsReconfPossible()
2  {
3    Logger.Debug("Checking if reconfiguration is possible");
4
5    var isReconfPossible = ConnectedNodes.Any(n => n.State == ENodeState
         .RUNNING);
6    if(!isReconfPossible)
7    {
8      Logger.Error("No reconfiguration possible!");
9      throw new Exception("No reconfiguration possible!");
10   }
11
12   return true;
13 }
```

Listing 6.6: Prüfung nach der Möglichkeit weiterer Rekonfigurationen

Ist eine Rekonfiguration nicht mehr möglich, wird durch die hierbei ausgelöste `Exception` die gesamte Simulation abgebrochen.

Zum Abschluss eines Schrittes werden die in Unterabschnitt 3.2.1 beschriebenen Behauptungen an das Testverfahren selbst validiert. Hierbei können jedoch nicht alle Behauptungen in Form von Constraints durch das Oracle automatisch während der Ausführung validiert werden. Von den implementierten Constraints können zudem nicht alle direkt innerhalb des Modells während der Ausführung eines Simulations-Schrittes validiert werden, weshalb außerhalb der Simulation ebenfalls Constraints definiert sind, die zum Abschluss der Simulation geprüft werden.

Ausgaben während eines Schrittes

Während eines Simulationsschrittes werden an verschiedenen Stellen bereits einige Daten ausgegeben. Darunter zählt z. B. die Ausgabe mithilfe von `Logger.Info()` in Listing 6.3. Auch im Falle einer Aktivierung bzw. Deaktivierung eines Fehlers wird dies im Rahmen eines üblichen Programmlogs direkt in den entsprechenden Methoden ausgegeben. Einige weitere Parameter, die direkt während der Ausführung eines Schrittes ausgegeben werden sind:

- Der vom `BenchmarkController` ausgewählte Benchmark
- Die Komponenten von ungültigen Constraints
- Die Information, wenn eine Rekonfiguration nicht möglich ist (vgl. Listing 6.6)

Die beim Monitoring ermittelten Daten werden nach der Ausführung eines Schrittes ausgegeben. Die in Unterabschnitt 3.2.3 definierten Ausgaben der Eigenschaften der jeweiligen Komponenten werden im folgenden Format ausgegeben:

- Für jeden Node der `YarnNode`-Klasse:
  - `NodeId`
  - `State`
  - `IsActive`
  - `IsConnected`
  - `RunningContainerCount`
  - `MemoryUsed`, `MemoryAvailable`, `MemoryUsage`
  - `CpuUsed`, `CpuUsed`, `CpuUsed`
- Für jeden Client der `Client`-Klasse:
  - `ClientId`
  - `BenchController.CurrentBenchmark.Name`
  - `CurrentExecutingAppId`
- Für jede Anwendung eines Clients der `YarnApp`-Klasse:
  - `AppId`
  - `Name`
  - `State`
  - `FinalStatus`
  - `AmHostId`, `AmHost.State`
- Für jeden Attempt einer Anwendung der `YarnAppAttempt`-Klasse:
  - `AttemptId`
  - `State`
  - `AmContainerId`
  - `AmHostId`, `AmHost.State`

- Für jeden Container eines Attempts der `YarnAppContainer`-Klasse:
  - `ContainerId`
  - `HostId`
  - `State`

6.1.4. Weitere mit der Simulation zusammenhängende Methoden

Neben der Ausführung der Simulation mit und ohne der Möglichkeit zur Aktivierung der Komponentenfehler gibt es noch einige weitere Methoden, die mit der Simulation zusammenhängen. Darüber besteht die Möglichkeit, die vorab generierten Eingabedaten für die Simulation ohne die Simulation selbst auszuführen, zu generieren. Da die Generierung der Eingabedaten nur dann durchgeführt wird, wenn die Verzeichnisse im HDFS noch nicht vorhanden sind (und somit auch die Daten selbst nicht), besteht auch die Möglichkeit, die bestehenden Eingabedaten zu löschen und anschließend neu zu geniereren

vgl. abschnitt benchcontroller damit und dann evtl. neu formulieren

. Zudem kann die Simulation der durch den `BenchmarkController` ausgewählten Benchmarks direkt und ohne die Ausführung der gesamten Simulation durchgeführt werden:

```
1  public void SimulateBenchmarks()
2  {
3    for(int i = 1; i <= _ClientCount; i++)
4    {
5      var seed = _BenchmarkSeed + i;
6      var benchController = new BenchmarkController(seed);
7      Logger.Info($"Simulating Benchmarks for Client {i} with Seed {seed
           }:");
8      for(int j = 0; j < _StepCount; j++)
9      {
10       benchController.ChangeBenchmark();
11       Logger.Info($"Step {j}: {benchController.CurrentBenchmark.Name}"
             );
12     }
13   }
14 }
```

Listing 6.7: Simulation der auszuführenden Benchmarks

6.2. Implementierungen der Mutationstests

Da in dieser Arbeit nicht nur Hadoop bzw. die Selfbalancing-Komponente getestet werden soll, sondern vor allem das in den vorherigen Abschnitten und Kapiteln beschrie-

bene Modell und Simulation zum Testen, wurden auch einige Mutationstests entwickelt. Implementiert wurden die Mutationstests in der bereits in Abschnitt 2.3 beschriebenen, Selfbalancing-Komponente, wodurch einzelne Bestandteile fehlerhaft gemacht wurden und daher nicht mehr korrekt funktionieren. Hierfür wurden drei Mutationsszenarien entwickelt:

mut1: Einschränkungen beim Monitoring
: Hierfür wurden die Monitoring-Möglichkeiten der Selfbalancing-Komponente eingeschränkt. Es wird dabei verhindert, dass die Anzahl der aktiven und wartenden Jobs in die *controlLog*-Datei geschrieben werden, aus welcher der Controller der Komponente die Daten ermittelt. Ebenso wurde verhindert, dass der Anteil des von Anwendungscontainer benötigten Arbeitsspeichers des Clusters in der *memLog*-Datei gespeichert wird. Dadurch erhält der Controller der Selfbalancing-Komponente nicht mehr alle benötigten Daten, um den MARP-Wert auf einen optimalen Wert anpassen zu können und führt daher keinerlei Anpassungen mehr durch.

mut2: Fehler beim Berechnen der Änderung des MARP-Wertes
: Für das zweite Mutationsszenario wurde die Berechnung des MARP-Wertes verändert. Konkret wird hierbei der Wert, um den der MARP="Wert erhöht bzw. verringert wird, mit dem Faktor 1,5 multipliziert. Die dadurch deutlich stärker ausfallende Veränderung wird anschließend in die Konfiguration von Hadoop eingetragen.

mut3: Neuer MARP-Wert kann nicht gespeichert werden
: Beim dritten Mutationsszenario wird verhindert, dass der von der Selfbalancing-Komponente ermittelte, neue MARP-Wert in die Konfiguration von Hadoop übernommen wird. Dazu wird unterbunden, dass der neue Wert in der `capacity-scheduler.xml` eingetragen und das anschließende Aktualisieren der Konfiguration durchgeführt wird.

Für jedes der drei Szenarien wurde ein entsprechendes Szenario im Rahmen der Plattform Hadoop-Benchmark entwickelt. Dadurch ist es möglich, die einzelnen Szenarien genauso zu starten, wie ein Testfall ohne Mutationstest.

6.3. Implementierung der Testfälle

Die im in
Evaluationsplan Testfälle
beschriebenen Testfälle werden mithilfe verschiedener Variablen implementiert. Relevant für die Ausführung eines Testfalles sind folgende, bereits in Listing 6.2 gezeigte, Felder:

```
1 private static int _BenchmarkSeed = Environment.TickCount;
2 private static double _FaultActivationProbability = 0.4;
3 private static double _FaultRepairProbability = 0.5;
4 private static int _HostsCount = 1;
5 private static int _NodeBaseCount = 4;
6 private static int _ClientCount = 1;
```

Listing 6.8: Zur Definition eines Testfalls relevante Felder

Da die jeweiligen Auswirkungen der Felder bereits in Unterabschnitt 6.1.2 erläutert wurden, wird an dieser Stelle hierauf verweisen.

6.4. Durchführung der Tests

7. Evaluation der Ergebnisse

8. Reflexion und Ausblick

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] Axel Habermaier, Johannes Leupolz und Wolfgang Reif. „Unified Simulation, Visualization, and Formal Analysis of Safety-Critical Systems with S#“. In: *Critical Systems: Formal Methods and Automated Verification: Joint 21st International Workshop on Formal Methods for Industrial Critical Systems and 16th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2016, Pisa, Italy, September 26-28, 2016, Proceedings*. Hrsg. von Maurice H. ter Beek, Stefania Gnesi und Alexander Knapp. Cham: Springer International Publishing, 2016, S. 150–167. ISBN: 978-3-319-45943-1. DOI: 10.1007/978-3-319-45943-1_11. URL: https://doi.org/10.1007/978-3-319-45943-1_11.

[5] Benedikt Eberhardinger u. a. „Back-to-Back Testing of Self-organization Mechanisms“. In: *Testing Software and Systems: 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*. Hrsg. von Franz Wotawa, Mihai Nica und Natalia Kushik. Cham: Springer International Publishing, 2016, S. 18–35. ISBN: 978-3-319-47443-4. DOI: 10.1007/978-3-319-47443-4_2. URL: https://doi.org/10.1007/978-3-319-47443-4_2.

[6] Axel Habermaier. *Model Checking*. URL: https://github.com/isse-augsburg/ssharp/wiki/Model-Checking (besucht am 30. 05. 2018).

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

[8] 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).

[9] 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).

[10] 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).

[11] 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).

[12] 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).

[13] 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).

[14] 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).

[15] 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.

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

[17] Docker Inc. *Get started with Docker Machine and a local VM*. URL: https://docs.docker.com/machine/get-started/ (besucht am 19. 05. 2018).

[18] 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).

[19] 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).

[20] 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).

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

[22] 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.

[23] 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).

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

[25] 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.

[26] Thomas Graves. *GraySort and MinuteSort at Yahoo on Hadoop 0.23*. 2013. URL: http://sortbenchmark.org/Yahoo2013Sort.pdf.

[27] 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.

[28] 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).

[29] 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 [18] 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.AHSProxy: 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.
```

```
18/03/14 13:07:03 INFO mapreduce.Job: Counters: 0
Job Finished in 37.53 seconds
```

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

```
$ yarn application -kill application_1520342317799_0002
[...]
Killing application application_1520342317799_0002
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 [9, 19, 20] 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`


```
{
  "apps": {
    "app": [
      {
        "id": "application_1518429920717_0001",
        "user": "root",
        "name": "QuasiMonteCarlo",
        "queue": "default",
        "state": "FINISHED",
        "finalStatus": "SUCCEEDED",
        "progress": 100,
        "trackingUI": "History",
        "trackingUrl": "http://controller:8088/proxy/
           application_1518429920717_0001/",
        "diagnostics": "",
        "clusterId": 1518429920717,
        "applicationType": "MAPREDUCE",
        "applicationTags": "",
        "startedTime": 1518430260179,
        "finishedTime": 1518430404123,
        "elapsedTime": 143944,
        "amContainerLogs": "http://compute-2:8042/node/containerlogs/
           container_1518429920717_0001_01_000001/root",
        "amHostHttpAddress": "compute-2:8042",
```

```
        "allocatedMB": -1,
        "allocatedVCores": -1,
        "runningContainers": -1,
        "memorySeconds": 1756786,
        "vcoreSeconds": 1546,
        "preemptedResourceMB": 0,
        "preemptedResourceVCores": 0,
        "numNonAMContainerPreempted": 0,
        "numAMContainerPreempted": 0
      }
    ]
  }
}
```

Listing B.2: REST-Ausgabe aller Ausführungen einer Anwendung vom TLS.
URL: `http://addr:port/ws/v1/applicationhistory/apps/{appid}/appattempts`

```
{
  "appAttempt": [
    {
      "appAttemptId": "appattempt_1518429920717_0001_000001",
      "host": "compute-2",
      "rpcPort": 46481,
      "trackingUrl": "http://controller:8088/proxy/
         application_1518429920717_0001/",
      "originalTrackingUrl": "http://controller:19888/jobhistory/job/
         job_1518429920717_0001",
      "diagnosticsInfo": "",
      "appAttemptState": "FINISHED",
      "amContainerId": "container_1518429920717_0001_01_000001"
    }
  ]
}
```