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:	2. Juli 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

MR MapReduce

NM NodeManager

RM ResourceManager

S# Safety Sharp

SuT System under Test

SWIM Statistical Workload Injector for Mapreduce

TLS Timeline-Server

YARN Yet Another Resource Negotiator

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.

vielleicht Testing MapReduce-Based Systems einbringen?

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

2. Grundlagen und Stand der Technik

2.1. Safety Sharp

Testen unter SS allgemein genauer erklären

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 ermöglicht. Hadoop wird von der *Apache Foundation* entwickelt und stellt und enthält verschiedene vollständig skalierbare Komponenten. Es ist daher möglich, ein Hadoop-Cluster auf nur einem einzelnen PC, aber auch verteilt auf zahlreichen Servern auszuführen. Hadoop ermöglicht es dadurch, sehr einfach Anwendungen auszuführen, um große Datenmengen zu verarbeiten. Die für das Cluster, und damit den Anwendungen, verfügbaren Ressourcen beschränken sich lediglich auf die Summe der verfügbaren Ressourcen aller Hosts, auf denen das Cluster ausgeführt wird.

Hadoop besteht aus folgenden Kernmodulen [7]:

Hadoop Common
: Gemeinsam genutzte Kernkomponenten

Hadoop YARN (Yet Another Resource Negotiator)
: Framework zur Verteilung und Ausführung von Anwendungen und das dazugehörige Ressourcen-Management

Hadoop Distributed File System
: Kurz HDFS, verteiltes Dateisystem

Hadoop MapReduce
: Kurz MR, Implementierung des MR-Ansatzes zum Verarbeiten von großen Datenmengen, nutzt YARN zur Ausführung der Anwendungen

Aufgrund seiner Verbreitung stellt Hadoop eine der wichtigsten Implementierungen des MR-Ansatzes dar [8]. Die Eingabe- und Ausgabedaten sind hierbei als *Key-Value*-Paare definiert, die mithilfe des MR-Frameworks verarbeitet werden. Hierbei werden zunächst die eingelesenen Eingabedaten in kleine und dadurch einfach zu verarbeitende Datenmengen aufgeteilt. Die geteilten Daten werden dann in mehreren, parallel ausgeführten Map-Tasks verarbeitet und zwischengespeichert. Die zwischengespeicherten Daten werden anschließend von einem oder mehreren Reduce-Tasks zusammengeführt und in die Ausgabedateien geschrieben. Das Framework ist hierbei sehr Fehlertolerant, da ein fehlerhafter Task jederzeit neu gestartet werden kann [9, 10]. Zur Speicherung der Ein-, Zwischen- und Ausgabedaten wird im Falle von Hadoop das HDFS genutzt [11].

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

Für weitere Details zum MR-Framework sei auf [9] und [10] verwiesen, eine ausführliche Betrachtung des MR-Frameworks mit Vorteilen und Problemen lässt sich in [12] finden.

Da das MR-Framework bzw. seine Implementierung in Hadoop nicht perfekt ist und in der Praxis zum Teil auch zweckentfremdet wurde, wurde in [13] das **YARN**-Framework vorgestellt. Die Kernidee ist hierbei die Trennung von Ressourcenmanagement und Scheduling vom eigentlichen Programm. In diesem Kontext bildet der MR-Ansatz eine mögliche Anwendung, die mithilfe des YARN-Frameworks ausgeführt wird [13].

Ein Hadoop-Cluster mit dem YARN-Framework besteht aus zwei wesentlichen Komponenten, dem *Controller* mit ResourceManager (RM) und den angeschlossenen *Nodes*:

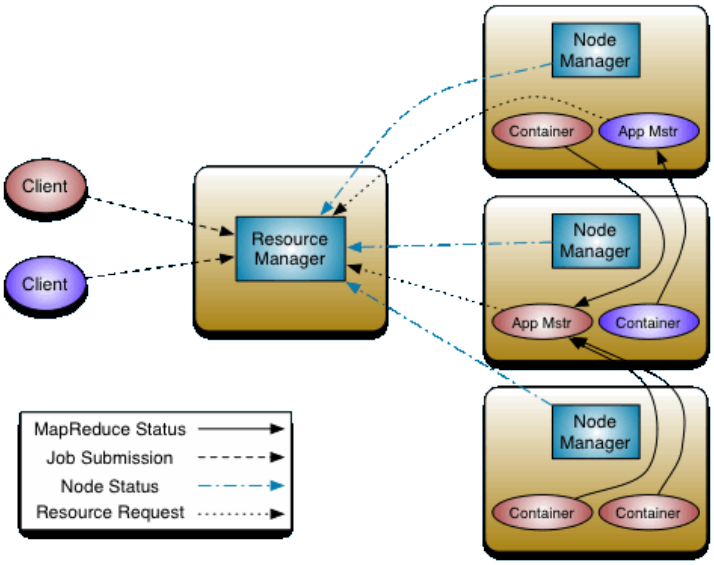

Abbildung 2.1.: Architektur von YARN (entnommen aus [14])

Der RM dient hierbei als *Load-Balancer* für das gesamte Cluster und besteht aus dem ApplicationManager (AM) und dem *Scheduler*, die eigentliche Ausführung der Anwendungen findet auf den Nodes statt. Der AM ist für die Annahme und Ausführung von einzelnen Anwendungen zuständig, denen der Scheduler die dafür notwendigen Ressourcen im Cluster zuteilt. Jeder Node besitzt einen NodeManager (NM), der für die Überwachung der Ressourcen auf dem jeweiligen Node sowie der auf dem Node ausgeführten Anwendungs-Container zuständig ist und diese Daten dem RM übermittelt.

Jede YARN-Anwendung bzw. Job besteht aus einer oder mehreren Ausführungsinstanzen, genannt *Attempts*. Jeder Attempt besitzt einen eigenen ApplicationMaster (AppMstr), welcher das Monitoring der Anwendung und die Kommunikation mit dem RM und NM übernimmt und die dafür benötigten Informationen bereitstellt [14]. Die eigentliche Ausführung einer Anwendung findet in den bereits erwähnten *Containern* statt, die jeweils einem Attempt zugeordnet sind. Container können auf einem beliebigen Node ausgeführt werden und repräsentieren die Ausführung eines Tasks innerhalb der Anwendung.

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 [15].

Das **HDFS** basiert auf einer ähnlichen Architektur wie YARN und besitzt ebenfalls einen Controller und mehrere Nodes:

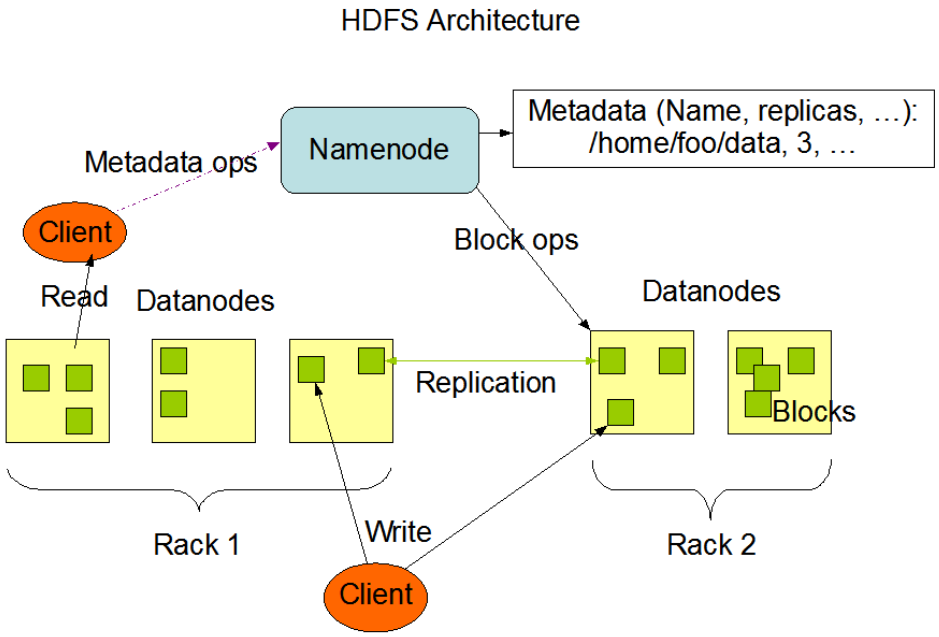

Abbildung 2.2.: Architektur des HDFS (entnommen aus [16])

Der *NameNode* dient als Controller 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 [17]. Die Daten selbst werden in mehreren 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 [16].

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 [18].

2.3.1. MARP-Wert

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 [18]. 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 [19] 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 gemäß [19] Anwendungen im Schnitt um bis zu 40 % schneller ausgeführt 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)[19]:

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 [19]. 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

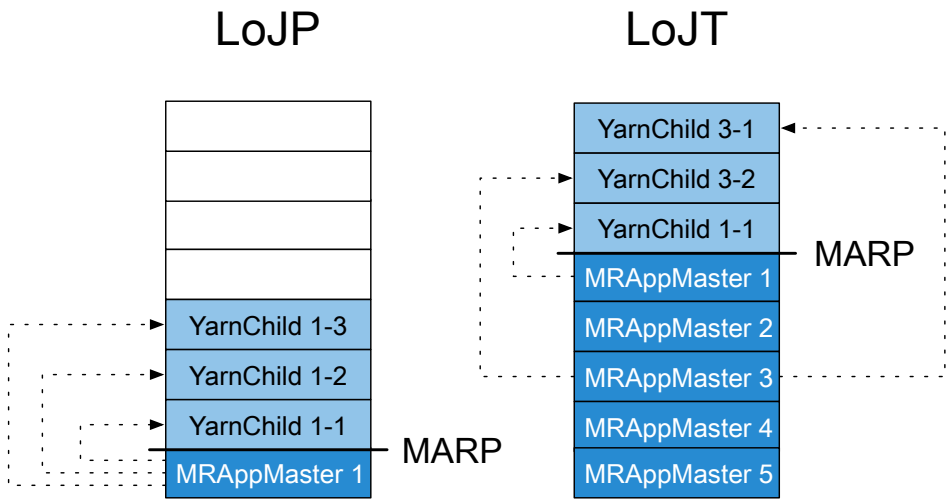

Abbildung 2.3.: LoJP und LoJT in Hadoop (entnommen aus [19]). Während beim LoJP sehr viel Speicher für Anwendungscontainer ungenutzt bleibt, können beim LoJT nicht genügend Anwendungscontainer allokiert werden, um die Anwendungen auszuführen.

Komponente zunächst analyisiert. Sie besteht aus folgenden vier Java-Klassen, welche den Kern der Komponente darstellen, und drei Shell-Scripten, die als Verbindung zum Hadoop-Cluster dienen:

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

Um den Zustand von Hadoop korrekt zu ermitteln, wird ein Kalman-Filter in Form der Open-Source-Bibliotek JKalman[3] genutzt. Der Kalman-Filter wurde von Kálmán erstmals in [20] beschrieben und dient, um „aus verrauschten und teils redundanten Messungen die Zustände und Parameter des Systems zu schätzen“ und lässt sich aufgrund seines Aufbaus auch für Echtzeitanwendungen nutzen [21]. Als einfaches Anwendungsbeispiel hierfür ist in [21] die Apollo-Mondlandefähre genannt, Strukov nutzte ihn in [22] aber auch zur Reduktion der Komplexität im Controlling. Für weitere Informationen zum Kalman-Filter wie seinen Aufbau, Funktionsweise und Anwendung sei hier auf entsprechende Fachliteratur wie z. B. [23–25] verwiesen.

Die drei Shell-Scripte dienen zur Interaktion zwischen dem Controller der Selfbalancing-Komponente und Hadoop selbst. Die beiden zuletzt genannten Scripte werden

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

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 [19] 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 `Effectuator`s das dritte Shell-Script `selfTuning-CapacityScheduler.sh` ausgeführt. Mithilfe dieses Shell-Scriptes wird der neue MARP-Wert in der Konfiguration des *Capacity Schedulers* gespeichert.

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 [27]. Jeder Hadoop-Container enthält zudem das Tool *collectd*[8], was das Monitoring des Containers

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

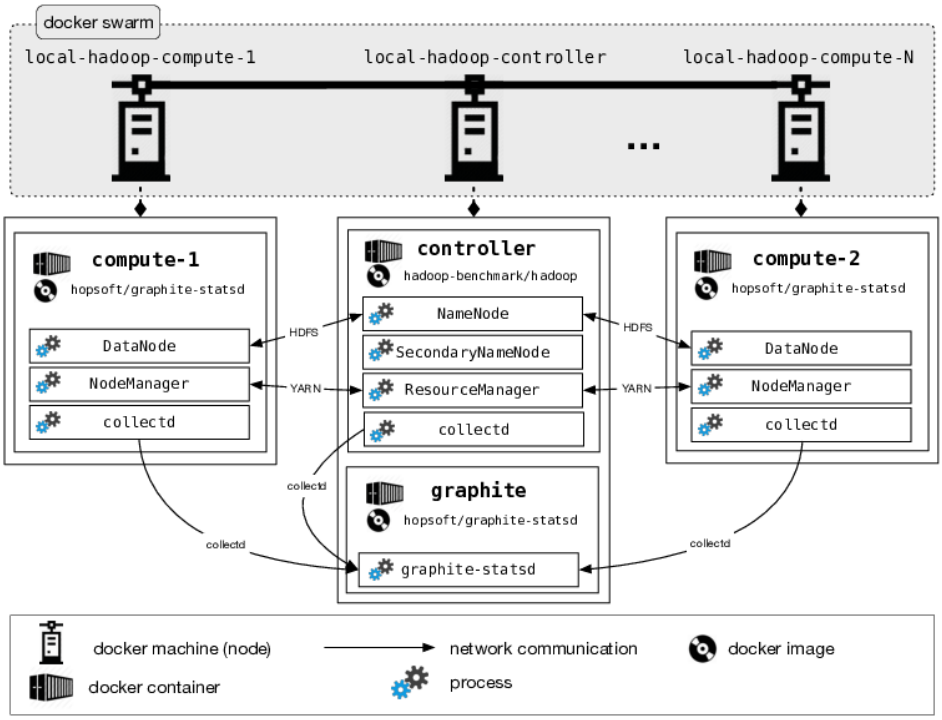

Abbildung 2.4.: High-Level-Architektur von Hadoop-Benchmark. Entnommen aus [26].

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.

[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 Testautomatisierung möglich ist.

Was für arten von Behauptungen gibts alles und welche werden wo behandelt? Wie wird unterschieden?

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 mithilfe des S#-Frameworks ein vereinfachtes Modell entwickelt und dieses mit einem realen Cluster verbunden. Um das reale Cluster auszuführen, wird eine für diese Fallstudie angepasste Version der von Zhang u. a. entwickelten Plattform Hadoop-Benchmark genutzt.

Teile der Beiträge und Inhalte dieses Kapitels wurden bereits in [28] publiziert.

3.1. Funktionale Anforderungen an das Cluster

Obwohl in dieser Masterarbeit der Fokus auf Testautomatisierung und Validieren eines Testsystems liegt, müssen auch die funktionalen Anforderungen an das zu testende System berücksichtigt werden. Folgende Anforderungen wurden hierfür bereits in [28] spezifiziert:

1. Ein Task wird vollständig ausgeführt, sofern er nicht abgebrochen wird
2. Kein Task oder Anwendung wird an inaktive, defekte oder nicht verbundene Nodes gesendet
3. Die Konfiguration wird aktualisiert, sobald eine entsprechende Regel erfüllt ist
4. Defekte oder Verbindungsabbrüche werden erkannt

generelles wie Benchmarks oder ss hier erklören

Da die Auswahl der ausgeführten Anwendungen nicht manuell bestimmt werden soll, wird hierfür ein Transitionssystem verwendet. Mithilfe dieses Transitionssystems, in dem die Wahrscheinlichkeiten von Wechsel zwischen zwei Anwendungen definiert sind, soll während der Ausführung eines Testfalls zufällig eine nachfolgende Anwendung ausgewählt werden. Da zum Testen des Clusters der S#-Simulator eingesetzt wird, hängt die Anzahl der Anwendungen primär von der Anzahl der ausgeführten Simulations-Schritte ab. Ein weiterer Faktor zur Anzahl der Anwendungen ist die Anzahl an

simulierten Clients, da auch getestet werden soll, wie sich das Cluster bei der Ausführung von mehreren parallel gestarteten Anwendungen verhält.

Diese funktionalen Anforderungen sind ebenso wie die Anforderungen an das Testsystem selbst (vgl. Unterabschnitt 3.2.1) im Modell zu implementieren.

3.2. Anforderungen an das Testsystem

Um das Testsystem zu validieren, wurde zunächst ein Evaluationsplan aufgestellt. In diesem ist festgehalten, was getestet wird, wie die Testfälle aussehen und wie die bei der Ausführung gewonnen Daten organisiert werden.

3.2.1. Behauptungen und Variablen

Neben den in Abschnitt 3.1 genannten funktionalen Anforderungen, bestehen an das gesamte Testsystem weitere Anforderungen. Zur Evaluation des Testsystems wurden hierbei folgende, weiteren Anforderungen bzw. Behauptungen aufgestellt:

1. Der MARP-Wert ändert sich basierend auf den derzeit ausgeführten Anwendungen
2. Der jeweils aktuelle Status des Clusters wird erkannt und im Modell gespeichert
3. Defekte Nodes und Verbindungsabbrüche werden erkannt
4. Im Modell implementierte Komponentenfehler werden im realen Cluster injiziert und repariert
5. Wenn alle Nodes defekt sind, wird erkannt, dass sich das Cluster nicht mehr rekonfigurieren kann
6. Ein Test kann vollautomatisch ausgeführt werden
7. Das Cluster kann ohne Auswirkungen auf die Funktionsweise auf einem oder beiden Hosts ausgeführt werden
8. Es können mehrere Benchmark-Anwendungen gleichzeitig gestartet und ausgeführt werden
9. Ein Testfall kann zeitlich unabhängig und mehrmals ausgeführt werden

Möglichst viele Anforderungen sollen, sofern möglich, im Testsystem implementiert werden, sodass sie in Form von *Constraints* während der Ausführung der Tests bereits automatisch validiert werden können (vgl.
abschnitt constraints impl
).

Basierend auf den Anforderungen wurden folgende unabhängigen Variablen zur Evaluation ermittelt:

- Anzahl der Hosts und Nodes
- Anzahl der Clients

- Anzahl der Simulations-Schritte
- *Seed* für Zufallsgeneratoren
- Generelle Wahrscheinlichkeit zur Aktivierung und Deaktivierung der Komponentenfehler

Basierend auf den unabhängigen Variablen wurden folgende abhängigen Variablen ermittelt:

- Verhältnis zwischen AppMstr und normalen Anwendungs-Containern
- Art und Anzahl der aktivierten und deaktivierten Komponentenfehler
- Anzahl erfolgreich bzw. nicht erfolgreich beendeter Anwendungen
- Anzahl abgebrochener Anwendungen

Diese Variablen werden dazu genutzt, um einerseits einzelne Testfälle zu definieren, andererseits auch als primäre Kennzahlen im Rahmen der Evaluation in Kapitel 7.

3.2.2. Generierung der Testfälle

Die Testfälle werden anhand der im vorherigen Unterabschnitt 3.2.1 genannten unabhängigen Variablen definiert. Ein Testfall ist somit zum einen Abhängig von der Größe des Clusters, also ob das Cluster auf einem oder beiden Hosts ausgeführt wird und aus wie vielen Nodes das Cluster besteht. Mithilfe des Seeds für die Zufallsgeneratoren werden die ausgeführten Anwendungen anhand des Transitionssystems ermittelt. Die Anzahl der insgesamt ausgeführten Anwendungen wird primär von der Anzahl der Simulations-Schritte sowie der Anzahl der Clients bestimmt. Die zu injizierenden und zu reparierenden Komponentenfehler, mit denen defekte Nodes und Verbindungsausfälle simuliert werden, werden zum Teil mithilfe des ausgewählt. Zudem sollen neben den Tests mit einem normalen Cluster zusätzliche Mutationstest mit einer veränderten Selfbalancing-Komponente ausgeführt werden.

Der in dieser Fallstudie verwendete Zufallsgenerator des .NET-Frameworks ist hierbei kein *echter* Zufallsgenerator, sondern ein Pseudo-Zufallsgenerator, der mithilfe mathematischer Formeln anhand eines Start-Seeds Zufallszahlen ermittelt, die anschließend zur Auswahl der Anwendungen und Komponentenfehlern genutzt werden. Da standardmäßig ein zeitbasierter Seed zur Initialisierung des Zufallsgenerators genutzt wird, muss zur Wiederausführbarkeit einzelner Testfälle der Seed der Zufallsgeneratoren manuell festgelegt werden können.

Wo wird Seed wie genutzt? am besten in implementierung davon!

Die Auswahl der konkreten Testfälle ist in Abschnitt 6.3 beschrieben, die Details zu deren Implementierung in Abschnitt 6.4.

3.2.3. Organisation und Ausgabe der Daten

Damit die bei der Ausführung gewonnenen Daten auch zur Evaluation genutzt werden können, wurde hierzu festgelegt, welche Daten während der Ausführung ausgegeben werden. Alle relevante Daten werden hierzu während der Ausführung der Testfälle in einer Log-Datei gespeichert. Zur Unterscheidung von einzelnen Ausführungen werden die Daten klar strukturiert.

Beim Start eines Testfalls sollen daher zunächst einige generelle Daten ausgegeben werden:

- Basis-Seed für die Zufallsgeneratoren
- Wahrscheinlichkeit für Aktivierung und Deaktivierung der Komponentenfehler
- Anzahl genutzter Hosts, Nodes und Clients
- Anzahl der ausgeführten Simulations-Schritte
- Angabe, ob ein normaler Test oder ein Mutationstest ausgeführt wird

Im Rahmen der Simulation können weitere Daten ausgegeben werden, wie z. B.:

- Angabe, ob vorab generierte Eingabedaten genutzt werden oder diese während der Ausführung eines Testfalls generiert werden
- Mindestdauer für einen Simulations-Schritt
- Auszuführende Benchmarks pro Client

Die Ausgabe der Daten der YARN-Komponenten wird bei jedem Simulations-Schritt durchgeführt, damit das Verhalten des Clusters berücksichtigt werden kann. Ausgegeben werde hierbei:

- Für jeden Node:
  - ID bzw. Name des Nodes
  - Aktueller Status
  - Informationen zur Fehleraktivierung
  - Anzahl ausgeführter Container auf dem Node
  - Angaben zur Speicherauslastung
  - Angaben zur CPU-Auslastung
- Für jeden Client:
  - ID bzw. Name des Clints
  - Aktuell ausgeführter Benchmark
  - ID der aktuell ausgeführten Anwendung auf dem Cluster
- Für jede Anwendung:
  - ID der Anwendung
  - Bezeichnung der Anwendung

  - Aktueller und finaler Status der Anwendung
  - ID bzw. Name des Nodes, auf dem der AppMstr ausgeführt wird
- Für jeden Attempt:
  - ID des Attempts
  - Aktueller Status des Attempts
  - ID des AppMstr-Containers
  - ID bzw. Name des Nodes, auf dem der AppMstr ausgeführt wird
- Für jeden Container:
  - ID des Containers
  - ID bzw. Name des auszuführenden Nodes
  - Aktueller Status des Containers

Die Details zur Implementierung und dem Ausgabeformat während der Ausführung der Simulation sind in Abschnitt 6.1.3 erläutert, die zur Ausgabe der generellen Testfalldaten in Abschnitt 6.4. Ein Beispiel einer möglichen Ausgabe für einen Testfall findet sich in Anhang C.

4. Aufbau des Testmodells

EInführung in Kapitel

4.1. Grundlegende Architektur des Modells

Um Hadoop mit der Selfbalancing-Komponente mit den in Unterabschnitt 3.2.1 beschriebenen Behauptungen prüfen zu können, wird mithilfe des S#-Frameworks ein vereinfachtes Modell entwickelt.

Die rechts abgebildete Abbildung 4.1 beschreibt die grundlegende Architektur des Modells für diese Fallstudie. Die Fallstudie besteht aus den drei Schichten des YARN-Modells selbst, dem Treiber zum Herstellen der Verbindung zwischen dem Modell und dem realen Cluster sowie dem realen Cluster selbst.

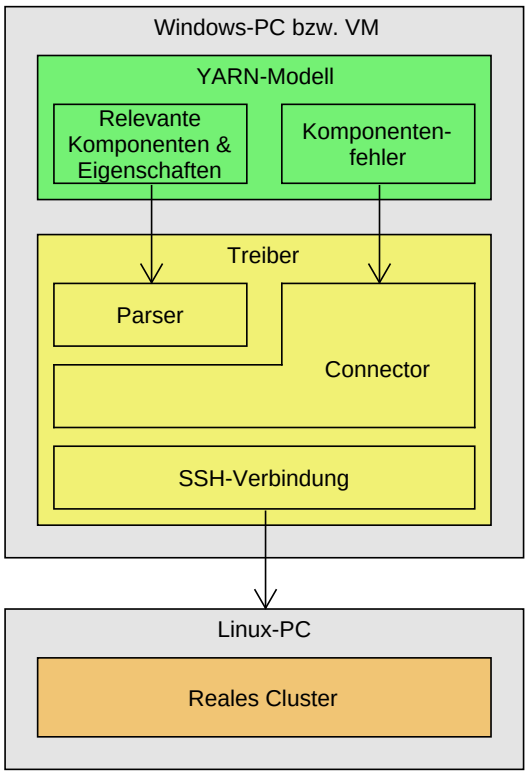

Abbildung 4.1.: Grundlegende Architektur des Gesamtmodells

Das eigentliche YARN-Modell bildet die grundlegende Architektur von YARN bzw. YARN-Anwendungen stark vereinfacht ab. Es enthält daher nur die für diese Fallstudie relevanten Komponenten und Eigenschaften sowie die Definitionen der Komponentenfehler.

Die Verbindung zwischen dem Modell und dem realem Cluster bildet der Treiber als eigenständige Schicht. Der Treiber besteht aus folgenden Komponenten:

Parser

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

Connector

Abstrahiert die Verbindung zum realen Cluster und die dabei auszuführenden Befehle.

SSH-Verbindung

Stellt die Verbindung zum realen Cluster her.

Der Zugriff mithilfe des Treibers auf das reale Cluster im Rahmen des Monitoring wird mithilfe des Parsers durchgeführt, welcher wiederum den Connector nutzt. Bei anderen Zugriffen wie z. B. die Injizierung und das Reparieren von Komponentenfehlern oder dem Starten von Anwendungen wird vom YARN-Modell mithilfe des Connectors auf das reale Cluster zugegriffen.

Die Umsetzung des realen Clusters wird im folgenden Abschnitt 4.4 beschrieben. Die Umsetzung des YARN-Modells und des Treibers wird im folgenden Kapitel 4 beschrieben.

Multihost-Mode irgendwo erklären

4.2. 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 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.

4.2.1. Modellierte YARN-Komponenten

Die abstrakte Basisklasse `YarnHost` stellt die Basis für alle Hosts des Clusters dar, also dem `YarnController` mit dem RM, und dem `YarnNode`, was einen Node darstellt, auf dem die Anwendungen bzw. deren Container ausgeführt werden. Die abstrakte Eigenschaft `YarnHost.HttpPort` dient als Hilfs-Eigenschaft, da Controller und Nodes unterschiedliche Ports für die Weboberfläche nutzen, deren URL mit Port in der Eigenschaft `YarnHost.HttpUrl` abrufbar ist. Sie wird daher vom Controller bzw. Node mit dem entsprechenden Port versehen.

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

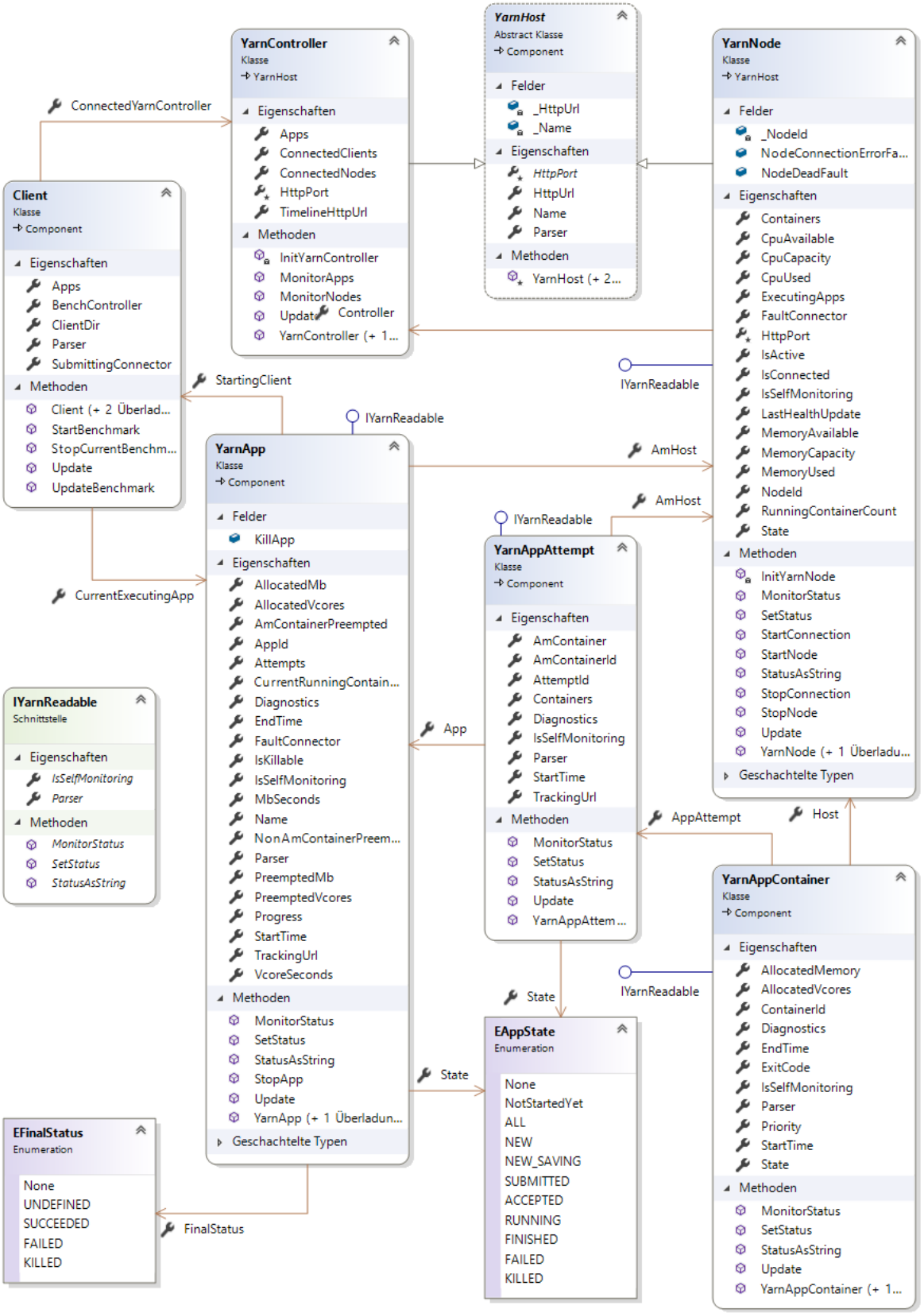

Abbildung 4.2.: Aufbau des YARN-Modells. Das Modell wurde mithilfe des Klassendiagramm-Designers in Visual Studio 2017 visualisiert. Daher werden Assoziationen mit höherer Multiplizität als 1, die daher mithilfe von `List<T>` umgesetzt wurden (z. B. `YarnApp.Attempts`) im Diagramm nicht als Assoziationen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

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.

4.2.2. 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 4.3.3). Die Injizierung des `NodeConnectionError`-Komponentenfehlers und die Aufhebung beider Komponentenfehler sind analog implementiert.

4.2.3. 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`.

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

```
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`

ausgegeben werden. Somit kann im Fehlerfall einfacher ermittelt werden, wie der Systemzustand zum Zeitpunkt des Fehlers war.

4.3. 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.

4.3.1. 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 [15, 29–31]:

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 [15, 30]. 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.

4.3.2. 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 [15, 30, 31], wodurch diese mithilfe des *Json.NET*-Frameworks[1] deserialisiert und direkt als die entsprechende `IParsedComponent`-Implementierung gespeichert werden. Da RM und TLS verschiedene Daten einer YARN-Komponente ausgeben, werden, sofern nötig, RM und TLS abgefragt und die dabei ermittelten Daten zusammengeführt.

Eine erste Besonderheit bildet zudem das Abrufen und Parsen der Report-Daten mittels REST-API. Da die Listen hierbei als Array der einzelnen Reports zurückgegeben werden [15, 30, 31], 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 [30, 31]. 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.

4.3.3. 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

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

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 [15, 29], 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.

4.3.4. 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.

4.4. Umsetzung des realen Clusters

Da mithilfe der Plattform Hadoop-Benchmark ein komplettes Hadoop-Cluster auf einem PC ausgeführt werden kann und dieses speziell für die Forschung entwickelt wurde, 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

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

Fallstudie zwei physische Hosts 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. Um den Ressourcenbedarf durch die von docker-machine erzeugten VMs zu reduzieren, werden die Docker-Container direkt auf den physischen Hosts ausgeführt:

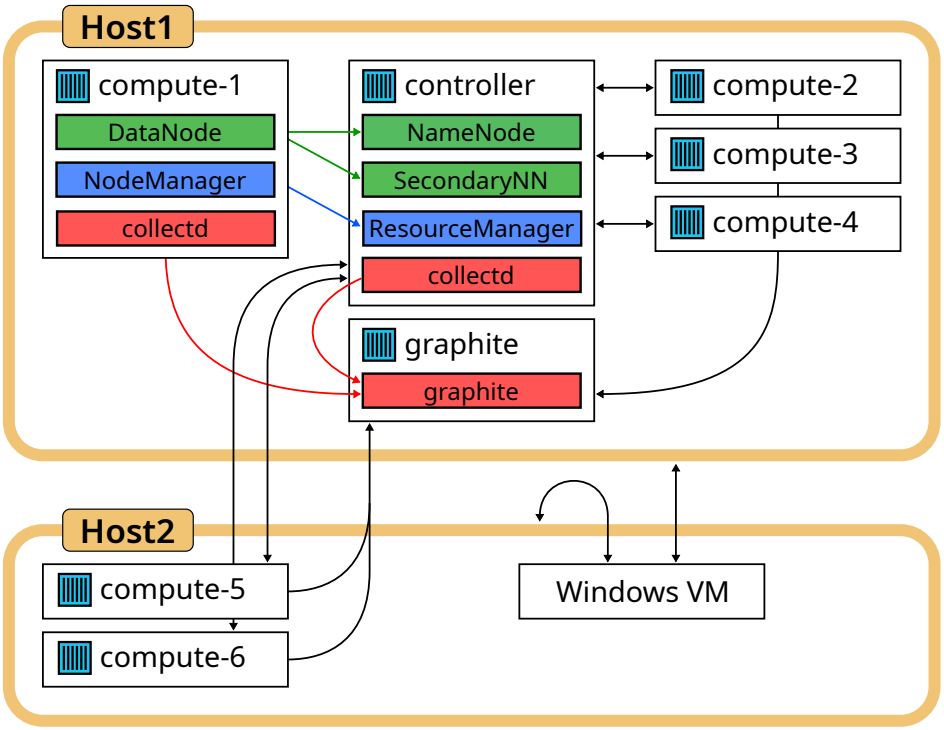

Abbildung 4.3.: In der Fallstudie verwendetes Cluster-Setup. Grün: HDFS, Blau: YARN, Rot: Graphite.

Irgendwo erwähnen, wie Docker-Container aufeinander aufbauen

Durch die leicht veränderte Architektur stehen dem Cluster daher mehr Ressourcen zur Verfügung. Zudem wird es mithilfe von von *Docker Swarm* so ermöglicht, das Hadoop-Cluster auf beiden Hosts auszuführen. Neben der auf Host1 ausgeführten Basis des Clusters besteht die Möglichkeit, die auf Host2 ausgeführten Hadoop-Nodes optional zum Cluster hinzuzufügen.

Basisanzahl Nodes erläutern

Ein weiterer Unterschied zur originalen Plattform besteht darin, dass der Hadoop-TLS ebenfalls gestartet wird. Zudem wurden einige Einstellungen von Hadoop so angepasst, dass defekte Nodes schneller erkannt werden.

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

wird *curl*[3] 7.47 genutzt. Zum Ausführen des Hadoop-Clusters wird Docker in der Version 18.03 CE genutzt.

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

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

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

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

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 [32]. 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 [33] 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 [34]. Bei der Ausführung auf dem in dieser Fallstudie verwendetem Cluster wurden jedoch nicht alle Workloads fehlerfrei ausgeführt. Zudem wird in [35] 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 [9]. `terasort` ist ebenfalls ein weit verbreiteter Benchmark, der die Hadoop-Implementierung der standardisierten *Sort Benchmarks*[1] darstellt [36]. 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 [33, 37].

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. [37] und [38]. 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 [39] und folgerten daher, dass für kleine Jobs evtl. einfachere Frameworks abseits von Hadoop besser geeignet wären. Die Autoren der Studie in [38] 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

Zur Ausführung der Tests wurde zunächst eine Simulation entwickelt, welche mithilfe des S#-Simulators ausgeführt werden kann. Alle hierfür relevanten Methoden wurden in der Klasse `SimulationTests` zusammengefasst, welche wiederum als Basis für die Ausführung der einzelnen Testkonfigurationen dient. Die Implementierung und Ausführung der Testkonfigurationen wird mithilfe der hierfür entwickelten Klasse `CaseStudyTests` durchgeführt.

6.1. Implementierung der Simulation

Für die Ausführung der Simulation wurden zwei grundlegende Tests implementiert. Das ist zum einen eine reine Simulation ohne die Aktivierung von Komponentenfehlern, sowie ein weiterer Test, bei dem Komponentenfehler aktiviert werden können. Ausgeführt werden können die Tests mithilfe des NUnit-Frameworks.

6.1.1. Grundlegender Aufbau

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 gestartet und die Verbindung mit den YARN-Controller wiederhergestellt werden 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 muss ein Simulations-Schritt um eine gewisse Zeit verzögert werden, sodass alle Aktivitäten innerhalb von Hadoop genügend Zeit zur Ausführung erhalten.

Der grundlegende Ablauf einer Simulation sieht wie folgt aus:

```
1 [Test]
2 public void SimulateHadoop()
3 {
```

```
4     ModelSettings.FaultActivationProbability = 0.0;
5     ModelSettings.FaultRepairProbability = 1.0;
6
7     var execRes = ExecuteSimulation();
8     Assert.IsTrue(execRes, "fatal error occured, see log for details");
9   }
10
11  private bool ExecuteSimulation()
12  {
13    var model = InitModel();
14    var isWithFaults = FaultActivationProbability > 0.000001; // prevent
          inaccuracy
15
16    var wasFatalError = false;
17    try
18    {
19      // init simulation
20      var simulator = new SafetySharpSimulator(model);
21      var simModel = (Model)simulator.Model;
22      var faults = CollectYarnNodeFaults(simModel);
23
24      SimulateBenchmarks();
25
26      // do simuluation
27      for(var i = 0; i < StepCount; i++)
28      {
29        OutputUtilities.PrintStepStart();
30        var stepStartTime = DateTime.Now;
31
32        if(isWithFaults)
33          HandleFaults(faults);
34        simulator.SimulateStep();
35
36        var stepTime = DateTime.Now - stepStartTime;
37        OutputUtilities.PrintDuration(stepTime);
38        if(stepTime < ModelSettings.MinStepTime)
39          Thread.Sleep(ModelSettings.MinStepTime - stepTime);
40
41        OutputUtilities.PrintFullTrace(simModel.Controller);
42      }
43
44      // collect fault counts and check constraint
45    }
46    // catch/finally
47
48    return !wasFatalError;
49  }
```

Listing 6.1: Simulation in dieser Fallstudie (gekürzt).

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 generellen Wahrscheinlichkeiten zum Aktivieren und Deaktivieren der Komponentenfehler. 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. Weitere zeitliche Verzögerungen während der Ausführung eines Simulations-Schrittes sind in Unterabschnitt 6.1.3 beschrieben.

Wenn während der Simulation eine im Modell nicht behandelte `Exception` auftritt, wird diese außerhalb der Simulation abgefangen und entsprechend geloggt. Dadurch wird zudem die Simulation beim aktuellen Stand abgebrochen. Nach Abschluss der Simulation werden immer alle zu dem Zeitpunkt mit Komponentenfehlern injizierten Nodes 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  public TimeSpan MinStepTime { get; set; } = new TimeSpan(0, 0, 0, 25);
2  public int BenchmarkSeed { get; set; } = Environment.TickCount;
3  public int StepCount { get; set; } = 3;
4  public bool PrecreatedInputs { get; set; } = true;
5  public bool RecreatePreInputs { get; set; } = false;
6  public double FaultActivationProbability { get; set; } = 0.25;
7  public double FaultRepairProbability { get; set; } = 0.5;
8  public int HostsCount { get; set; } = 1;
9  public int NodeBaseCount { get; set; } = 4;
10 public int ClientCount { get; set; } = 2;
11
12 private Model InitModel()
13 {
14   ModelSettings.HostMode = ModelSettings.EHostMode.Multihost;
15   ModelSettings.HostsCount = HostsCount;
16   ModelSettings.NodeBaseCount = NodeBaseCount;
17   ModelSettings.IsPrecreateBenchInputsRecreate = RecreatePreInputs;
18   ModelSettings.IsPrecreateBenchInputs = PrecreatedInputs;
19   ModelSettings.RandomBaseSeed = BenchmarkSeed;
20
```

```
21    var model = Model.Instance;
22    model.InitModel(appCount: StepCount, clientCount: ClientCount);
23    model.Faults.SuppressActivations();
24
25    return model;
26  }
```

Listing 6.2: Initialisierung des Modells für die Simulation

Die einzelnen Eigenschaften für die Simulation werden vor dem Initialisieren des Modells in den `ModelSettings` gespeichert. Die dort gespeicherten Werte werden wiederum zum Initialisieren der Modell-Instanz bzw. während der Ausführung der Simulation genutzt.

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

MinStepTime

Definiert die Mindestdauer eines Schrittes.

BenchmarkSeed

Gibt den Seed an, mit dem die Zufallsgeneratoren in den Klassen `Benchmark Controller` 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 `_FaultActivation Probability`. 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 Eigenschaft `PrecreatedInputs`. Es 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. Die Eigenschaft `RecreatePreInputs` definiert hierfür, ob bereits bestehende Eingabedaten neu generiert werden, was standardmäßig nicht der Fall ist bzw. dieses Feld auf `false` gesetzt ist. 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 bereits in

ModelSettings.HostMode beschrieben und hier verweisen

beschrieben.

Die direkt im Anschluss an die Initialisierung des Simulators ausgerufene Methode `CollectYarnNodeFaults()` ermittelt alle im initialisierten Modell enthaltenen Komponentenfehler, die mit dem `NodeFaultAttribute` markiert sind:

```
1  private FaultTuple[] CollectYarnNodeFaults(Model model)
2  {
3    return (from node in model.Nodes
4        from faultField in node.GetType().GetFields()
5        where typeof(Fault).IsAssignableFrom(faultField.FieldType)
6
7        let attribute = faultField.GetCustomAttribute<NodeFaultAttribute
             >()
8        where attribute != null
9
10       let fault = (Fault)faultField.GetValue(node)
11
```

```
12          select Tuple.Create(fault, attribute, node, new IntWrapper(0),
                new IntWrapper(0))
13      ).ToArray();
14  }
```

Listing 6.3: Ermitteln der Komponentenfehler mit dem `NodeFaultAttribute`

Die gefundenen Komponentenfehler werden als Array aus Tupel, bestehend aus dem Komponentenfehler selbst, dem Attribut sowie dem dazugehörigen Node zurückgegeben. Zur Speicherung hierfür dient der Typ `FaultTuple`, welcher ein Alias für das hierfür genutzte `Tupel<T>` darstellt. Die jeweiligen Instanzen der Attribute und Nodes werden für die in Abschnitt 6.1.3 beschriebene Aktivierung der dazugehörigen Komponentenfehler benötigt. Die beiden im Tupel gespeicherten Instanzen des `IntWrapper` dienen zur Speicherung der Anzahl der Aktivierungen bzw. Deaktivierungen der Komponentenfehler. Da der Wert einer Struktur wie `int` nicht direkt in einem Tupel geändert werden kann, dient die Klasse `IntWrapper` hierfür als Adapter.

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 ausgeführten Modells 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

Zur Aktivierung der Komponentenfehler gibt es drei Einzelschritte. 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 im realen Cluster injiziert 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 6.4: Berechnung der Aktivierung von Komponentenfehlern (zusammengefasst).

Die Entscheidung zur Deaktivierung eines Komponentenfehlers verhält sich analog. Anstatt der generellen Aktivierungswahrscheinlichkeit in `ModelSettings.FaultActivationProbability` wird hierbei 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 6.4 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` initiiert. Dazu wird vom `YarnController` aus 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 6.5: 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 (vgl. Abschnitt 2.1).

Monitoring der ausgeführten Anwendungen

Bevor das Monitoring der Anwendungen durchgeführt wird, wird zunächst fünf Sekunden gewartet, bis der AppMstr sowie weitere Container der Anwendung allokiert bzw. gestartet wurden. Diese Wartezeit ist prinzipiell optional, wird hier jedoch genutzt, damit die Auslastung des Clusters besser ermittelt werden kann. Die Wartezeit vor dem Monitoring ist bereits in der inUnterabschnitt 6.1.1 beschriebenen Mindestdauer eines Schrittes enthalten.

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`:

```
1  public void MonitorStatus()
2  {
3    if(IsSelfMonitoring)
4    {
5      var parsed = Parser.ParseAppDetails(AppId);
6      if(parsed != null)
7        SetStatus(parsed);
8    }
9
10   var parsedAttempts = Parser.ParseAppAttemptList(AppId);
11   foreach(var parsed in parsedAttempts)
12   {
13     var attempt = // get existing or empty attempt instance
14     if(attempt == null)
```

```
15        // throw OutOfMemoryException
16
17      attempt.AppId = AppId;
18      attempt.IsSelfMonitoring = IsSelfMonitoring;
19      if(IsSelfMonitoring)
20        attempt.AttemptId = parsed.AttemptId;
21      else
22      {
23        attempt.SetStatus(parsed);
24        attempt.MonitorStatus();
25      }
26    }
27  }
```

Listing 6.6: 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 6.6 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 zum Zeitpunkt des Monitoring aktive bzw. allokierte Container durchgeführt. 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. Da Container nur während der der Laufzeit von Anwendungen bzw. Attempts zu deren Ausführung existieren, werden die beim vorherigen Schritt ermittelten Container-Daten gelöscht, bevor die aktuellen Daten der Container eines Attempts ermittelt werden.

Validierung durch das Oracle

Im direkten Anschluss an das Monitoring erfolgt die Validierung 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 in

constraint implementierung

implementierten Constraints für die jeweiligen Komponenten noch erfüllt werden können. Ist das nicht der Fall, wird dies 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   return true;
12 }
```

Listing 6.7: 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 (vgl.
constraint implementierung
).

Ausgaben während eines Schrittes

Die wesentlichen Ausgaben während eines Tests wurden bereits in Unterabschnitt 3.2.3 definiert und beschrieben. Neben diesen Daten werden im Programmlog weitere Daten gespeichert, damit die Ausführung eines Testfalles besser nachvollzogen werden kann. Zudem werden alle Ein- und Ausgabedaten der SSH-Verbindungen zwischen dem Modell und dem realen Cluster in einer eigenen Log-Datei gespeichert. Dieses SSH-Log dient dazu, die Ursache von unerwarteten Fehlern herauszufinden.

Neben den bereits beschriebenen Daten werden im Programmlog folgende Daten gespeichert:

- Verbundene SSH-Verbindungen mit ihrer ID zur besseren Zuordnung im SSH-Log
- Ausführung der Erstellung von vorab generierten Eingabedaten

- Vollständiger Pfad des Setup-Scriptes (vgl. Abschnitt 4.4)
- URL des Controllers zur Nutzung der REST-API
- Vorschau auf bzw. derzeit vom `BenchmarkController` ausgewählte Benchmarks
- Ausführung von Komponentenfehlern
- Diagnostik-Daten der YARN-Komponenten
- Welche Constraints bei welchen Komponenten verletzt wurden
- Die Information, wenn eine Rekonfiguration nicht möglich ist (vgl. Listing 6.7)

Nach Abschluss der Simulation wird ein erneutes Monitoring des gesamten Clusters durchgeführt und der hierbei ermittelte Status als finaler Clusterstatus ausgegeben. Zudem werden einige statistische Kenndaten zur Simulation ausgegeben:

- Gesamtdauer der Simulation
- Anzahl erfolgreicher Schritte
- Anzahl der maximal möglichen, aktivierbaren Komponentenfehler
- Anzahl aktivierter und deaktivierter Komponentenfehler
- Letzter ermittelter MARP-Wert
- Anzahl aller ausgeführten, erfolgreicher, nicht erfolgreicher sowie abgebrochener Anwendungen
- Anzahl aller ausgeführten Attempts
- Anzahl aller während der Ausführung erkannten Container
- Anzahl aller validierten Constraints und fehlerhaften Constraints, getrennt nach System under Test (SuT)- und Testsystem-Constraints

Ein möglicher Programmlog sowie das exakte Ausgabeformat für eine Ausführung eines Testfalls findet sich in Anhang C.

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:

```
public void SimulateBenchmarks()
{
  for(int i = 1; i <= _ClientCount; i++)
  {
    var seed = _BenchmarkSeed + i;
    var benchController = new BenchmarkController(seed);
    Logger.Info($"Simulating Benchmarks for Client {i} with Seed {seed
        }:");
    for(int j = 0; j < _StepCount; j++)
    {
      benchController.ChangeBenchmark();
      Logger.Info($"Step {j}: {benchController.CurrentBenchmark.Name}"
          );
    }
  }
}
```

Listing 6.8: Simulation der auszuführenden Benchmarks

6.2. Generierung der Mutanten

Zur Entwicklung der für die Mutationstests verwendeten Mutanten wurde der von Groce u. a. in [40] vorgestellte **Universalmutator**[1] genutzt. Der Universalmutator ist ein Tool, das den vorhandenen Quellcode eines Programmes verändert, sodass damit Mutationstests durchgeführt werden können. Diese werden vor allem in der Forschung eingesetzt, um Testsysteme zu verifizieren, in dem das SuT verändert wird. Ziel hierbei ist es, mithilfe des Testsystems zu erkennen, dass das SuT verändert wurde bzw. nicht korrekt funktioniert.

Allgemeines zu Mutationstests in Grundlagen?

Der Universalmutator kann zum Entwickeln von Mutationstests hierbei nicht nur innerhalb einer bestimmten Umgebung bzw. Programmiersprache, sondern prinzipiell für alle Programmiersprachen eingesetzt werden. Dies wird dadurch ermöglicht, dass die vom Universalmutator generierten Mutanten basierend auf einem oder mehreren Regelsätzen durchgeführt werden und damit der Quellcode verändert wird. So kann vom Universalmutator Quellcode u. A. in den Sprachen Python, Java, C/C++ oder Swift mutiert werden [40].

Da bei der Ausführung des Universalmutators auch zahlreiche Mutanten erzeugt werden, die nicht kompiliert bzw. ausgeführt werden können, nutzt das Tool die Compiler der jeweiligen Sprache zur Validierung der generierten Mutationen. Ein validierter Mutant zeichnet sich hierbei dadurch aus, dass dieser durch den Original-Compiler

[1] https://github.com/agroce/universalmutator

der jeweiligen Sprache kompiliert werden kann und die generierten Objektdateien bzw. Bytecode nicht dem nicht-mutierten Original oder anderen bereits generierten Mutationen entsprechen [40]. Diese Validierung kann mithilfe von entsprechenden Startparametern durch ein benutzerdefiniertes Programm durchgeführt werden oder alternativ nicht durchgeführt werden [40, 41].

Da in dieser Fallstudie nicht nur Hadoop bzw. die Selfbalancing-Komponente getestet werden soll, sondern vor allem das in den vorherigen Abschnitten und Kapiteln beschriebene Testsystem, wurden auch Mutationstests erstellt. Hierbei wurden mithilfe des Universalmutators insgesamt 431 valide Mutationen aus dem Quellcode der Selfbalancing-Komponente generiert. Von allen validen Mutationen wurden anschließend für jede der vier Klassen der Selfbalancing-Komponente jeweils ein Mutant zufällig ausgewählt, welche als Basis für die in dieser Fallstudie verwendeten Mutationstests dienen:

1. Zur Ermittlung der Veränderung des MARP-Wertes muss zunächst der jeweils aktuelle Arbeitsspeicher-Verbrauch im Cluster eingelesen werden. Dies geschieht im `Controller` mithilfe einer `for`-Schleife, mit der der Speicherverbrauch im Cluster nahezu sekündlich aus der `memLog`-Datei eingelesen wird. Der Mutant verändert die Schleifenbedingung, damit die Schleife kein einziges mal ausgeführt wird. Dadurch wird verhindert, dass der Speicherverbrauch des Cluster vom `Controller` eingelesen und verwendet werden kann. Dadurch ist der Speicherverbrauch des Clusters auch nicht für den in [19] vorgestellten Algorithmus der Selfbalancing-Komponente verfügbar.
2. Der `Effectuator` dient dazu, um die Veränderung des MARP-Wertes im Cluster zu speichern. Dazu wird das entsprechende Shell-Script mithilfe der *Bash*-Shell ausgeführt. Der Mutant sorgt dafür, dass anstatt des korrekten Dateipfades der Bash (`/bin/bash`) ein ungültiger Dateipfad (hier `%bin/bash`) aufgerufen wird und somit der neue MARP-Wert nicht in das Cluster übertragen werden kann.
3. Mithilfe des `ControlNodeMonitor` wird das Schell-Script zum Ermitteln der Anzahl der aktiven YARN-Jobs ausgeführt. Dies geschieht in einem eigenen Thread, der mithilfe einer `while`-Schleife, die solange aktiv ist, solange der Thread aktiv ist. Hierbei wird das Script rund einmal pro Sekunde aufgerufen und danach ermittelt, ob bei der Ausführung des Shell-Scriptes Fehler aufgetreten sind. Dazu wird ein entsprechender `BufferedReader` geöffnet und der Error-Stream des Scriptes eingelesen und anschließend von der Selfbalancing-Komponente ausgegeben. Umgesetzt wird das mithilfe einer `while`-Schleife, die solange den Fehler ausgibt, solange die mithilfe von `BufferedReader.readLine()` ausgelesene Fehlermeldung nicht `null` ist, also noch weiteren Text enthält. Der Mutant ändert die Schleifenbedingung nun so ab, dass die Schleife durchlaufen wird, solange die Fehlermeldung keinen Text enthält, `readLine()` also `null` zurück gibt. Dadurch wird der `ControlNodeMonitor` in einer Dauerschleife gefangen und die Anzahl

der aktiven YARN-Jobs wird einmalig direkt nach dem Start der Selfbalancing-Komponente ermittelt.

4. Die Klasse `MemUtilization` funktioniert analog wie die `ControlNodeMonitor`-Klasse, führt jedoch das Script zum Auslesen des Arbeitsspeicher-Verbrauches aus dem Cluster aus. Dieser Mutant verhindert hierbei die komplette Ausführung des entsprechenden Threads, indem die Bedingung der Schleife für den gesamten Thread so verändert wurde, dass diese nur dann ausgeführt wird, wenn der Thread nicht aktiv ist. Dadurch wird verhindert, dass das entsprechende Shell-Script überhaupt ausgeführt wird und der Speicherverbrauch somit nicht ausgelesen wird.

Aufgrund der verwendeten Mutationen erhält die Selfbalancing-Komponente bei jedem einzelnen Mutanten bereits nicht alle benötigten Informationen zur Anpassung des MARP-Wertes bzw. kann die Änderung des Wertes nicht in der Konfiguration des Clusters speichern.

Für jede Mutation wurde ein Mutationsszenario im Rahmen der Plattform Hadoop-Benchmark entwickelt, bei dem keine weitere Mutationen enthalten sind (vgl. Abschnitt 4.4). Zudem wurde ein weiteres Mutationsszenario entwickelt, bei dem alle vier Mutationen enthalten sind.

6.3. Auswahl der Testkonfigurationen

Die im in Unterabschnitt 3.2.2 beschriebenen Testkonfigurationen werden mithilfe verschiedener Variablen implementiert. Anhand dieser Konfiguration wurden dynamisch zur Laufzeit die entsprechenden Testfälle generiert, wobei jeder Simulations-Schritt des S#-Simulators einem Testfall entspricht.

Relevant für die Ausführung einer Konfigurationen sind folgende, bereits in Listing 6.2 gezeigte, Eigenschaften:

```
1 public int BenchmarkSeed { get; set; } = Environment.TickCount;
2 public double FaultActivationProbability { get; set; } = 0.25;
3 public double FaultRepairProbability { get; set; } = 0.5;
4 public int HostsCount { get; set; } = 1;
5 public int NodeBaseCount { get; set; } = 4;
6 public int ClientCount { get; set; } = 2;
```

Listing 6.9: Zur Definition einer Testkonfiguration relevante Felder

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

Zur Festlegung dieser Variablen und damit der Testkonfigurationen wurde zunächst eine Systematik entwickelt, nach welcher die Testfälle durchgeführt werden. Hierfür wurden mithilfe des folgenden Programmcodes zunächst zwei Seeds ermittelt:

```
public void GenerateCaseStudyBenchSeeds()
{
  var ticks = Environment.TickCount;
  var ran = new Random(ticks);
  var s1 = ran.Next(0, int.MaxValue);
  var s2 = ran.Next(0, int.MaxValue);
  Console.WriteLine($"Ticks: 0x{ticks:X}");
  Console.WriteLine($"s1: 0x{s1:X} | s2: 0x{s2:X}");
  // Specific output for generating test case seeds:
  // Ticks: 0x5829F2
  // s1: 0xAB4FEDD | s2: 0x11399D3
}
```

Listing 6.10: Ermittlung der für die Testkonfigurationen genutzten Basisseeds

Die beiden ermittelten Seeds 0xAB4FEDD und 0x11399D3 wurden jeweils bei jeder Konfiguration zwischen den anderen Variablen genutzt.

Zur Festlegung der Werte zur generellen Wahrscheinlichkeiten zur Aktivierung bzw. Deaktivierung von Komponentenfehlern wurden zunächst über 20.000 mögliche Aktivierungen und Deaktivierungen mit verschiedenen generellen Wahrscheinlichkeiten und Auslastungsgraden der Nodes simuliert. Der dabei für alle Konfigurationen ausgewählte Wert von 0,3 stellt hierbei eine ausgewogene Aktivierung bzw. Deaktivierung der Komponentenfehler bei unterschiedlichen Auslastungsgraden der Nodes dar.

Die Anzahl der Hosts wurde bei einigen Konfigurationen auf 1 festgelegt, bei den meisten liegt diese jedoch bei 2. Die Node-Basisanzahl wurde auf 4 festgelegt, da hierbei das Cluster eine ausreichende Größe besitzt und jedem Node ausreichend Ressourcen zur Verfügung stehen, um Anwendungen auszuführen. Bei einer zu hohen Basisanzahl erhält jeder einzelne Node geringere Ressourcen, was vor allem die Ausführung bei ressourcenintensiven Anwendungen wie z. B. `pentomino` behindert, während bei einer zu geringen das Cluster sehr klein ist und daher keine ausreichende Evaluationsbasis bietet. Die Anzahl der Simulations-Schritte und damit der ausgeführten Testfälle selbst wurde variiert, wodurch in einigen Konfigurationen 5 und in anderen 10 Testfälle ausgeführt werden. Ebenso variiert wurde die Anzahl der simulierten Clients, , die im Bereich von 2, 4 oder 6 Clients liegt.

Alle Konfigurationen wurden mindestens einmal jeweils mit der Selfbalancing-Komponente ohne Mutationen sowie in einem der in Abschnitt 6.2 erläuterten Mutationsszenarien ausgeführt. Von den hiermit möglichen 48 Testkonfigurationen wurden die möglichen Konfigurationen mit einem Host und sechs simulierten Clients sowie die möglichen Konfigurationen mit zwei simulierten Clients und zehn Testfällen nicht ausgeführt. Das ergibt für die Evaluation somit eine Datenbasis von 32 grundlegenden Testkonfigurationen. Eine Übersicht der genutzten Testkonfigurationen und der Dauer der jeweiligen Tests ist in Anhang D zu finden.

Bei der Ausführung der Tests zur Evaluation wurden Eingabedaten nicht vorab generiert, sondern während der Ausführung von den Anwendungen direkt generiert. Daher wurde die Mindestdauer für einen Simulations-Schritt in allen Fällen auf 25 Sekunden festgelegt, da hierbei ein Großteil der ausgeführten Anwendungen auf dem Cluster erfolgreich beendet werden können. Eine Ausreichende Mindestdauer ist vor allem für die Generierung der Eingabedaten für nachfolgende Anwendungen wichtig, da nicht vollständig generierte Daten von abgebrochenen Anwendungen nicht von nachfolgenden Anwendungen genutzt werden können. Zudem stellt dies eine ausreichende Zeitspanne zur Rekonfiguration von Hadoop dar.

6.4. Implementierung der Tests

Genauso wie die Simulation wurde zur Implementierung das NUnit-Framework sowie zur Ausführung der *ReSharper Unit Test Runner*[2] genutzt. Alle zur Ausführung der Testfälle der Fallstudie relevanten Methoden wurden zudem in der Klasse `CaseStudyTests` zusammengefasst, welche die bereits in Abschnitt 6.1 beschriebene Simulation nutzt. Zur Ausführung der Tests wurde folgende Methode entwickelt, bei der mithilfe von NUnit die Testfälle ermittelt werden:

```
1  [Test]
2  [TestCaseSource(nameof(GetTestCases))]
3  public void ExecuteCaseStudy(int benchmarkSeed, double
       faultProbability, int hostsCount, int clientCount, int stepCount,
       bool isMutated)
4  {
5    // write test case parameter to log
6
7    InitInstances();
8    var isFailed = false;
9    try
10   {
11     // Setup
12     StartCluster(hostsCount, isMutated);
13     Thread.Sleep(5000); // wait for startup
14
15     var simTest = new SimulationTests();
16     // save test case parameter to simTest
17
18     // Execution
19     simTest.SimulateHadoopFaults();
20   }
21   // catch exceptions and set isFailed=true
22   finally
```

[2] https://www.jetbrains.com/resharper/

```
23    {
24      // Teardown
25      StopCluster();
26      MoveCaseStudyLogs(/* test case parameter */);
27    }
28    Assert.False(isFailed);
29  }
```

Listing 6.11: Methode zur Ausführung der Testfälle (gekürzt)

Das Starten und Beenden des jeweiligen Cluster dient der automatisierten Ausführung aller Tests inkl. denen mit der mutierten Selfbalancing-Komponente. Dadurch ist es möglich, das Cluster neben dem normalen Szenario auch in einem Mutationsszenario zu starten. Durch das Beenden des Clusters im Finally-Block ist es möglich, bei einer abgebrochenen Simulation andere Testfälle regulär auszuführen, da dadurch das Cluster regulär beendet wird und die Daten des abgebrochenen Testfalls wie bei einem erfolgreichen Test gespeichert werden.

Da die verwendeten Connectoren bzw. SSH-Verbindungen prinzipiell nur einmal initialisiert werden müssen und anschließend für alle auszuführenden Testfälle verwendet werden können, werden diese einmalig in `InitInstances()` initialisiert und anschließend bei jedem weiteren ausgeführten Test wiederverwendet. Eine möglicherweise bereits zuvor verwendete Modell-Instanz wird hier jedoch in jedem Fall genauso wie einige statische Zählvariablen gelöscht bzw. zurückgesetzt.

Mithilfe der im `TestCaseSourceAttribute` referenzierten Methode `GetTestCases()` werden die implementierten Testkonfigurationen ermittelt:

```
1  public string MutationConfig { get; set; } = "mut1234";
2
3  public IEnumerable GetTestCases()
4  {
5    return from seed in GetSeeds()
6           from prob in GetFaultProbabilities()
7           from hosts in GetHostCounts()
8           from clients in GetClientCounts()
9           from steps in GetStepCounts()
10          from isMut in GetIsMutated()
11
12          where !(hosts == 1 && clients >= 6)
13          where !(clients <= 2 && steps >= 10)
14          select new TestCaseData(seed, prob, hosts, clients, steps,
                isMut);
15 }
16
17 private IEnumerable<int> GetSeeds()
18 {
19   yield return 0xAB4FEDD;
```

```
20    yield return 0x11399D3;
21 }
```

Listing 6.12: Implementierung der Testfälle (gekürzt). Die hier nicht gezeigten Methoden zur Rückgabe der implementierten Werte wie `GetFaultProbabilities()` sind nach dem gleichen Schema aufgebaut wie `GetSeeds()`. Mithilfe der Eigenschaft `MutationConfig` erfolgt die Auswahl des zu verwendeten Mutationsszenarios.

Hierbei werden nur Konfigurationen generiert, auf denen die in Abschnitt 6.3 genannten Bedingungen zutreffen, womit anstatt den möglichen 48 Testfällen nur die gewählten 32 generiert werden.

Damit die bei der Ausführung der Tests generierten Logs einfacher zur Evaluation genutzt werden können, werden die angefallenen Logdateien nach jeder Ausführung in ein entsprechendes Verzeichnis verschoben. Hierbei werden die Logdateien gemäß der Parameter der Testkonfiguration wie folgt umbenannt:

```
1 var todayStrShort = DateTime.Today.ToString("yyMMdd");
2 var mutated = isMutated ? "MT" : "MF";
3 var faultProbStr = faultProbability.ToString(CultureInfo.
     InvariantCulture);
4 var baseFileName = $"0x{benchmarkSeed:X8}-{faultProbStr}F-{hostsCount:
     D1}H-{clientCount:D1}C-{stepCount:D2}S-{mutated}-{todayStrShort}";
```

Listing 6.13: Bestimmung des Dateinamens zur Umbenennung der Logdateien

Da beim Monitoring immer die Daten aller auf dem Cluster ausgeführten Anwendungen übertragen und im SSH-Log gespeichert werden, hat das Neustarten des Clusters bei jedem Testfall zudem den Nebeneffekt, dass im SSH-Log keine Daten von ausgeführten Anwendungen eines anderen Testfalls enthalten sind.

7. Evaluation der Ergebnisse

Einleitungstext, vielleicht die Übersicht als Einleitung?

7.1. Übersicht der ausgeführten Tests

In Abschnitt 6.3 wurden 32 Testkonfigurationen ermittelt, die mithilfe des in dieser Fallstudie entwickelten Testansatzes insgesamt 42 mal ausgeführt wurden. Die meisten Konfigurationen wurden hierbei jeweils einmal ausgeführt, 6 Konfigurationen wurden auch mehrmals ausgeführt. Die Gründe für eine mehrfache Ausführung einzelner Konfigurationen sind in im Rahmen der entsprechenden Auffälligkeiten bzw. Fehler beschrieben. Eine Übersicht aller genutzten Testkonfigurationen und deren Ausführungen findet sich in Anhang D.

Bei der Auswertung der Programmlogs der einzelnen Tests musste zudem beachtet werden, dass die jeweiligen Monitoring-Informationen nur Momentaufnahmen bilden. Vor allem bei längeren Schritten werden vom RM sehr viele Anpassungen vorgenommen, die aufgrund der Struktur eines Simulations-Schrittes (vgl. Unterabschnitt 6.1.3) nicht erkannt werden können.

Zur Auswertung der Evaluation dienten die in

Constraintabschnitt

implementierten und nicht implementierten Constraints der in Abschnitt 3.1 und Unterabschnitt 3.2.1 definierten Anforderungen.

In diesem Kapitel wurden folgende Begriffe mit folgenden Bedeutungen genutzt:

Bessere Lastverteilung
: Die gesamte Last im Cluster ist für den jeweiligen Kontext besser verteilt. Meist bedeutet dies, dass die Last auf allen Nodes gleichmäßig verteilt ist anstatt einer vollständigen Auslastung einzelner Nodes.

Gefailte Anwendung
: Nicht vollständig abgeschlossene Anwendung, die aufgrund eines Fehler vorzeitig beendet wurde.

Testkonfiguration
: Eine Konfiguration bestehend aus mehreren Parametern, die einen Test definieren. Die Nummerierung der in Abschnitt 6.3 definierten Konfigurationen erfolgte fortlaufend.

Testfall

Ein ausgeführter Simulations-Schritt. Ein Testfall wird während der Laufzeit basierend auf einer zugrundeliegenden Testkonfiguration sowie den Ereignissen und Ergebnissen zuvor ausgeführter Testfälle der zugrundeliegenden Konfiguration generiert. In einem Testfall können daher unterschiedliche Komponentenfehler aktiviert und deaktiviert sowie unterschiedliche Anwendungen gestartet werden, auch wenn sie durch die gleiche Testkonfiguration generiert wurden.

Test

Eine Ausführung einer Testkonfiguration. Um mehrmalige Ausführungen einer Testkonfiguration zu kennzeichnen, wurde der jeweiligen Konfiguration eine weitere Ziffer angehängt.

7.2. Statistische Kenndaten

Die Dauer aller Simulationen betrug 4:06:03 Stunden, die gesamte Ausführungsdauer inkl. Starten und Beenden des Clusters bei jeder Konfiguration betrug 5:07:47 Stunden. Von den 280 Testfällen, die ausgeführt hätten sollen, wurden nur 215 Testfälle (77 %) ausgeführt. Der Grund für den Abbruch von 13 Tests liegt im in Abschnitt 6.1.3 beschriebenen Abbruch der Simulation, wenn keine Rekonfiguration des Clusters mehr möglich ist, also bei allen Nodes des Clusters ein Komponentenfehler injiziert und dies beim Monitoring erkannt wurde.

Insgesamt wurde bei allen Tests 419 Komponentenfehler aktiviert (14 % von 2980 möglichen), von denen jedoch nicht alle injiziert wurden, da bei einigen Testfällen beide Komponentenfehler der Nodes gleichzeitig aktiviert wurden. In diesen Fällen überwog die Aktivierung es Komponentenfehlers, der den Node komplett beendet. Von allen aktivierten Komponentenfehlern wurden während der Simulationen 251 Komponentenfehler deaktiviert bzw. repariert, was eine Quote von 60 % ergibt. In 4 der ausgeführten Testfällen wurde jedoch kein einziger Komponentenfehler deaktiviert, weshalb die Tests der Konfigurationen 4, 5.1, 5.2 und 6 entsprechend frühzeitig abgebrochen wurden.

Bei den 42 Tests wurden 393 Anwendungen im Cluster gestartet, von denen 200 (51 %) erfolgreich und damit vollständig ausgeführt wurden, aufgrund eines Fehlers vorzeitig beendet bzw. gefailt waren 102 Anwendungen (26 %). Vorzeitig abgebrochen wurden bei allen Tests 49 Anwendungen (12 %), was 42 Anwendungen macht, die zum Ende der Simulationen noch ausgeführt wurden. Nicht eingerechnet sind hier 25 nicht gestartete Anwendungen, die Gründe hierfür sind in Unterabschnitt 7.3.8 erläutert. Für die gestarteten Anwendungen wurden 532 Attempts mit einem AppMstr zugewiesen, was im Schnitt 1,35 Attempts pro Anwendung ergibt. Auffällig ist hierbei, dass mit 200 Attempts ein Attempt mehr aufgrund eines AppMstr-Timeouts abgebrochen wurde, als während der Simulation erfolgreich beendet wurden (199 Attempts). 30 weitere Attempts

wurden aufgrund eines Fehlers im Map-Task abgebrochen, 12 weitere terminierten mit dem Exitcode -100, was ebenfalls auf Fehler hindeutet. Das ergibt dadurch eine Quote von 45,5 % aller gestarteten Attempts, die nicht erfolgreich abgeschlossen werden konnten. Beim Monitoring wurden 3039 Anwendungs-Container erkannt, was im Schnitt 7,73 Container pro Anwendung bzw. 5,71 pro Attempt ergibt. Da bei den zu startenden Anwendungen einige kleine und einige sehr ressourcenintensive Anwendungen enthalten sind (vgl. Abschnitt 5.2), kann sich die Anzahl der Container zwischen den einzelnen Anwendungen sehr unterscheiden.

Vom Oracle wurden bei allen Tests zusammengezählt 74.051 Constraints validiert, von denen 541 falsifiziert wurden (0,73 %). Die meisten falsifizierten Constraints hatten hierbei die Tests der Konfigurationen 31 und 32 mit 40 bzw. 42 Constraints (von jeweils 5140 geprüften), die höchste Quote Konfiguration 8 mit 1,97 % der Constraints (13 von 661). Der Hauptgrund für die teilweise sehr hohe Anzahl an falsifizierten Constraints vor allem liegt darin, dass die Constraints für fehlerhaften Anwendungen auch in nachfolgenden Testfällen innerhalb einer Ausführung einer Testkonfiguration entsprechend erkannt werden. Dies resultiert in bis zu 34 ungültigen Constraints für fehlerhafte Anwendungen bei den einzelnen Tests.

7.3. Erkenntnisse der Evaluation

Zusammenfassung, hier erwähnen, dass MUT schneller und weniger failapps hatte?

Mehr auf Anforderungen verweisen

das heißt defekte nodes!!11

7.3.1. Betrachtung der MARP-Werte

Bei der Betrachtung der MARP-Werte lässt sich generell sagen, dass die Selfbalancing-Komponente den MARP-Wert entsprechend der Auslastung des Clusters anpasst. Während bei allen Testkonfigurationen, bei denen alle 4 Mutationen aktiv waren, der MARP-Wert unverändert blieb, wurde er bei 16 von 20 Ausführungen der 16 Konfigurationen ohne Mutationen verändert:

Konf.	1.1	1.2	3	5.1	5.2	7.1	7.2	9.1	9.2	11
Wert	0,100	0,100	0,474	0,242	0,100	0,100	1,000	0,269	0,175	0,539

Konf.	13	15	17	19	21	23	25	27	29	31
Wert	0,356	0,368	0,731	0,430	0,335	0,498	0,521	,0819	0,273	0,333

Tabelle 7.1.: Finale MARP-Werte der Testkonfigurationen ohne Mutanten

Da er in den Konfigurationen 1 und 7 bei der jeweils ersten Ausführung nicht verändert wurde, wurden beide Konfigurationen erneut ausgeführt, wobei der MARP-Wert bei

letzterem mehrmals erhöht wurde, bevor er im finalen Clusterstatus auf 1 gesetzt wurde. Bei der Konfiguration 1 wurde der Wert dagegen bei keiner der beiden Ausführungen verändert.

Die nicht durchgeführte Änderung des MARP-Wertes in Konfiguration 1 liegt sehr wahrscheinlich daran, dass in hier nur im ersten der fünf Testfälle zwei Anwendungen gleichzeitig gestartet werden. Dadurch wurden in allen 10 Testfällen zusammen nur 8 Anwendungen gestartet, die Hälfte davon jeweils beim ersten Testfall. Da zudem 4 der 8 Anwendungen nur kleine Anwendungen (`randomtextwriter` und `pi`) sind und diese entsprechend schnell beendet werden können, steht den wesentlich ressourcenintensiveren Anwendungen `TestDFSIO -write` und `TestDFSIO -read` das gesamte Cluster nahezu exklusiv zur Verfügung. Daher stehen den Anwendungen ausreichend Ressourcen zur Verfügung, was eine Anpassung des MARP-Wertes unnötig erscheinen lässt und daher durch die Selfbalancing-Komponente nicht durchgeführt wird.

Bei der Testkonfiguration 7 ist dies ähnlich, wobei die gesamte Last auf mehr Nodes verteilt werden kann. Der Test 7.2 und der hier deutlich veränderte MARP-Wert im Vergleich zu Test 7.1 ohne Anpassung zeigt jedoch auch, dass es stark abhängig davon ist, wie die Last im Cluster verteilt wird. Bestätigt wird dies durch die Tests 9.1 und 9.2, da bei letzterem weniger Komponentenfehler injiziert wurden und sich die Last entsprechend besser verteilen konnte. Dadurch war im Test 9.2 ein um rund 0,1 niedrigerer MARP-Wert als im Test 9.1 nötig.

Auffällig war zudem, dass der MARP-Wert in den Testausführungen 7.2, 9.1 und 23 nicht direkt im ersten Testfall verändert wurde, sondern erst bei der Ausführung von Testfällen im späteren Verlauf der jeweiligen Tests. Als Resultat wurde daher in 9 der 15 Testfälle zunächst das hierfür genutzte Constraint als ungültig validiert.

7.3.2. Erkennung der Mutanten

Da die Selfbalancing-Komponente den MARP-Wert basierend auf der aktuellen Auslastung des Clusters anpasst, konnte anhand der Betrachtung der MARP-Werte auch geprüft werden, ob die implementierten Mutationen vom Testsystem erkannt wurden. Um das zu bewerkstelligen wurden von den 16 Testkonfigurationen mit einem Mutationsszenario 22 Testausführungen durchgeführt (vgl. Anhang D).

Zunächst wurde das Mutationsszenario genutzt, in dem alle vier Mutationen enthalten sind (vgl. Abschnitt 6.2). Bei jeder der 17 Testausführungen mit allen Mutanten wurden diese basierend auf dem Constraint zur Erkennung des MARP-Wertes erkannt. Eine Besonderheit bildet hier jedoch der Test 2, der den korrespondierenden Mutationstest zur Konfiguration 1 darstellt, bei der bei beiden Ausführungen der MARP-Wert nicht verändert wurde. Bei Test 2 kann daher nicht eindeutig festgestellt werden, ob der Mutant erkannt wurde, oder ob aufgrund der gestarteten Anwendungen der MARP-Wert nicht verändert wurde (vgl. Vermutungen zu Testkonfiguration 1 in Tabelle 7.1).

Anders ist dies im Vergleich der Ausführungen der Testkonfigurationen 7 und 8. Durch die massive Veränderung des MARP-Wertes im Test 7.2 auf den finalen Wert von 1 kann davon ausgegangen werden, dass die Mutanten der Konfiguration 8 erkannt wurden. Dies wird dadurch gestützt, dass bei Konfiguration 8 im Gegensatz zur korrespondieren Testkonfiguration 3 der 4 gestarteten Anwendungen gefailt sind. Zudem stellt das ein Indiz dafür dar, dass der Mutant in Konfiguration 2 erkannt worden sein könnte.

Während bei jeder Konfiguration ein Mutationsszenario mit jeweils allen vier Mutanten genutzt wurde, wurde die Testkonfiguration 10 zusätzlich mit jeweils einem Mutanten ausgeführt. Ziel hierbei war es zu validieren, ob einzelne Mutanten ebenfalls vom Testsystem erkannt werden oder zur Erkennung der Mutanten vom Testsystem eine Kombination aus mehreren Mutaten nötig ist. Hierzu wurde die Testkonfiguration 10 mit unterschiedlichen Mutationsszenarien der Plattform Hadoop-Benchmark ausgeführt, bei denen jeweils ein Mutant aktiv ist. Die Auswahl dieser Testkonfiguration hierfür liegt darin begründet, dass hier das Cluster auf beiden Hosts mit sechs Nodes gestartet wird, auf denen bis zu vier Anwendungen gleichzeitig gestartet werden. Zudem wurde beim Test 9.2 festgestellt, dass sich der MARP-Wert auch direkt im ersten ausgeführten Testfall ändern kann und es gab bei den Ausführungen der Konfiguration 9 keine weiteren ungültigen Constraints als fehlerhafte Anwendungen.

Einige Ergebnisse der hierfür 5 ausgeführten Tests sind sehr unterschiedlich. So variiert die Anzahl der aktivierten und deaktivierten Komponentenfehler zwischen 7 und 11 bzw. 5 und 9, sowie die Anzahl der gefailten Anwendungen zwischen 1 und 3. Gemein haben alle Tests jedoch, dass der MARP-Wert bei allen 5 Tests nicht verändert wurde, womit alle Mutationen erkannt wurden. Damit kann festgestellt werden, dass jeder der vier in Abschnitt 6.2 beschriebenen Mutanten durch das entwickelte Testsystem erkannt wird.

7.3.3. Aktivierung und Deaktivierung der Komponentenfehler

Die Aktivierung und Deaktivierung der Komponentenfehler in einem Testfall hängt neben dem zur Berechnung benötigtem Seed vor allem von den zuvor ausgeführten Testfällen einer Testkonfiguration ab (vgl. Abschnitt 6.1.3). Daher werden abhängig von der Lastverteilung im Cluster auch bei einer mehrmaligen Ausführung der gleichen Konfiguration u. U. unterschiedliche Komponentenfehler aktiviert. Unterschieden werden muss hierbei zudem zwischen aktivierten und injizierten Komponentenfehlern. Während beide implementierten Komponentenfehler für einen Node in einem Testfall auch gleichzeitig aktiviert werden konnten, wurde gemäß

Abschnitt mit Komponentenfehler im Node

in so einem Fall jedoch nur der `NodeDead`-Fehler im Cluster injiziert. Die Deaktivierung bzw. das Reparieren der Komponentenfehler verhält sich analog hierzu.

Im Vergleich zwischen korrespondierenden Konfigurationen, die sich nur in der Nutzung des Mutationsszenarios unterschieden, wurde nur 5 mal die gleiche Anzahl an Komponentenfehler aktiviert, bei der Deaktivierung der Komponentenfehler besitzen nur 3 korrespondierende Konfigurationen die gleiche Anzahl. Die Anzahl der aktivierten und deaktivierten Komponentenfehler unterschied sich dagegen in 8 bzw. 7 korrespondierenden Testkonfigurationen um jeweils einen Komponentenfehler. In allen anderen Ausführungen von korrespondierenden Konfigurationen unterschied sich die Anzahl um jeweils mehr als einen Komponentenfehler. Mit 20 aktivierten Komponentenfehlern wurden bei der Ausführung der Testkonfiguration 32 die meisten aktiviert, die meisten Komponentenfehler deaktiviert wurden bei den Konfigurationen 11 und 12 mit jeweils 15 Stück. Nur im Test zur Konfiguration 2 wurden mit 3 Stück alle aktivierten Komponentenfehler während der Simulation auch wieder deaktiviert. In den Tests 4, 5.1, 5.2 und 6 wurden jeweils 6 oder 7 Komponentenfehler aktiviert, jedoch keine deaktiviert, weshalb diese Tests bereits beim 3. ausgeführten Testfall gemäß Abschnitt 6.1.3 abgebrochen wurden.

Im Vergleich zwischen den Tests von korrespondierenden Testkonfigurationen sind die Tests der Konfigurationen 1 und 2 auffällig. Während beim Test 1.1 mit 5 Komponentenfehlern bzw. beim Test 1.2 mit 7 Komponentenfehlern jeweils rund jeder achte mögliche Komponentenfehler aktiviert wurde, wurden beim Test 2 lediglich 3 Komponentenfehler für 4 Nodes in 5 Testfällen (insgesamt also 40 mögliche Komponentenfehler) aktiviert. Eine geringere Quote weist lediglich Test 9.2 auf, bei dem mit 4 von 60 möglichen Komponentenfehler nur 7 % aktiviert wurden. Die Testkonfiguration 9 ist auch deshalb auffällig, da im Test 9.1 fast dreimal so viele Komponentenfehler, also 11 Stück, aktiviert wurden. Auch in den korrespondierenden Tests der Konfiguration 10 liegt die Anzahl der aktivierten Komponentenfehler mit 7 bis 11 jeweils mehr als doppelt so hoch wie in Test 9.2.

Auffällig ist zudem, dass bei korrespondierenden Testkonfigurationen mit unterschiedlicher Anzahl an aktivierten Komponentenfehlern die niedrigere Anzahl meist die Mutationstests aufweisen. Nur bei den Tests 27 und 28.1 sowie bei den Tests 31 und 32 weisen die Tests ohne Mutationsszenario die niedrigere Anzahl auf.

auf diskussion der selfbalancing komponenten verweisen als grund

7.3.4. Nicht vollständig abgeschlossene Anwendungen

Wie bereits erwähnt, sind rund ein viertel aller gestarteten Anwendungen gefailt, was im Schnitt 2,4 Anwendungen pro ausgeführten Test ergibt. Die meisten gefailten Anwendungen sind hierbei mit 9 bzw. 8 bei den Tests 31 und 32 zu finden. Auffällig ist zudem der Vergleich zwischen den Tests 19 und 20. Während bei der Ausführung der Testkonfiguration 19 ganze 5 Anwendungen gefailt sind, ist bei der Ausführung des Tests 20 keine einzige gefailt. Auffallend ist zudem, dass bei Konfigurationen mit

Mutationsszenario fast immer weniger oder gleich viele Anwendungen gefailt sind als bei den korrespondierenden Konfigurationen ohne Mutationsszenario. Eine Ausnahme bildet der Test 8, bei dem 3 Anwendungen gefailt sind, während bei den Tests 7.1 und 7.2 jeweils keine Anwendung gefailt ist. Eine weitere Ausnahme bildet der Test 9.2, bei dem eine Anwendung mehr gefailt ist als im Test 10.3, die restlichen Tests der Konfigurationen 9 und 10 verhalten sich jedoch wie andere korrespondierende Testkonfigurationen.

Bei der Betrachtung der Constraints, welche die in Abschnitt 3.1 definierte Anforderung umsetzen, dass Anwendungen vollständig ausgeführt werden, solange sie nicht manuell bzw. durch das Testsystem vorzeitig abgebrochen werden, fällt auf, dass die Anzahl der ungültigen Validierungen durch das Oracle mit kumuliert 317 ungültigen Constraints mehr als die Hälfte aller ungültigen Constraints ausmacht (58,6 %). Im Schnitt ergibt das somit 7,5 ungültige Constraints pro Test bzw. 1,4 ungültige Constraints pro durch das Oracle überprüften Testfall. Die auf den ersten Blick sehr hohe Anzahl an ungültigen Constraints resultiert daraus, dass eine gefailte Anwendung bei jedem nachfolgenden Testfall bei einer Testausführung erneut durch das Oracle entsprechend validiert wurde. Aussagekräftiger ist daher die Anzahl von 102 nicht vollständig abgeschlossenen Anwendungen.

Anhand der Datenbasis lassen sich vier Ursachen für nicht vollständig ausgeführte Anwendungen ausmachen:

- Der AppMstr ist nicht mehr erreichbar, da der auszuführende Node aufgrund eines Komponentenfehlers nicht mehr erreichbar ist und der AppMstr nach einiger Zeit mit dem Fehler *AppMstr-Timeout* als abgebrochen markiert wird.
- Die den AppMstr zugewiesenen Nodes sind vollständig ausgelastet, wodurch der AppMstr selbst nicht ausgeführt werden kann und dieser nach einiger Zeit mit dem Fehler *AppMstr-Timeout* abgebrochen wird. Dies beinhaltet auch Timeouts, wenn einem AppMstr keine Ressourcen in Form eines ausführenden Nodes zugewiesen werden können.
- Während der Ausführung einer MR-Anwendung wird ein Fehler im Map-Task festgestellt, der dazu führt, dass der Task abgebrochen wird. Dieser Fehler kam bei den hier ausgeführten Tests lediglich bei der Anwendung `TestDFSIO -read` vor, wenn die zuvor generierten Eingabedaten für diesen Benchmark aufgrund aktivierter Komponentenfehler nicht mehr im Cluster vorhanden waren. Zwar werden Dateien im HDFS immer auf mehr als einem Node gespeichert (vgl. Abschnitt 2.2), jedoch ist es möglich, dass die für die Anwendung benötigten Daten auf Nodes repliziert wurden, die alle beendet wurden. Dies führte dazu, dass die benötigten Daten nicht gefunden werden können bzw. bereits im HDFS als fehlerhaft markiert sind. Dadurch wird im Map-Task ein Fehler ausgelöst, der die gesamte Anwendung vorzeitig beendet.

- Der AppMstr eines Attempts wird mit dem Exitcode -100 beendet. Dieser Fehler kommt dann vor, wenn versucht wird, einen Anwendungs-Container für den jeweilige Anwendung bzw. Attempt auf einem defekten Node zu starten. Dieser Fehler trat nur dann auf, wenn im ausführende Node des betroffenen AppMstr im gleichen Testfall ein Komponentenfehler injiziert wurde und der Node dadurch aufgrund des Defektes ausfiel. Aufgrund der mit dem Fehler verbundene Fehlermeldung „Container released on a *lost* node“ liegt die Vermutung nahe, dass Anwendungs-Container, hier wahrscheinlich der AppMstr, zum Zeitpunkt der Fehlerinjizierung bereits abgeschlossen waren und das Cluster bereits die benötigten Ressourcen wieder freigegeben hat. Da dies jedoch nicht möglich war, wurde der AppMstr mit dem entsprechenden Fehler beendet.

Hierbei werden aufgrund eines AppMstr-Timeouts zunächst nur die Attempts mit dem entsprechenden Fehler abgebrochen, nicht jedoch die Anwendung selbst. Die Anwendung selbst wird in so einem Fall erst dann als gefailt abgebrochen, sobald zwei Attempts aufgrund eines Timeouts abgebrochen werden mussten. Wenn ein Attempt mit dem Exitcode -100 terminiert, wird unabhängig von zuvor ausgeführten Attempts ein erneuter Attempt mit entsprechendem AppMstr gestartet.

Bei einigen der AppMstr-Timeouts aufgrund der Aktivierung von Komponentenfehler lässt sich zudem ein spezielles Muster erkennen. Hierbei wurde in einem zuvor ausgeführten Testfall regulär auf einem Node ein AppMstr einer Anwendung allokiert. Nun kann es passieren, dass für diesen Node ein Komponentenfehler injiziert wird, was dazu führt, dass der Node nicht mehr erreichbar ist und der AppMstr aufgrund eines Timeouts als beendet markiert wird. Hierbei wird direkt im Anschluss ein neuer AppMstr allokiert, was auch dazu führt, dass die Anwendung nun einen zweiten Attempt besitzt, nachdem der erste aufgrund des Timeouts abgebrochen wurde. Nun kann es dabei passieren, dass dies noch während der Aktivierung von Komponentenfehlern innerhalb eines Simulations-Schrittes geschieht, wodurch es möglich ist, dass der auszuführende Node des zweiten AppMstr ebenfalls aufgrund eines im gleichen Testfall injizierten Komponentenfehlers nicht mehr erreichbar ist. Dadurch wird der zweite AppMstr bzw. Attempt aufgrund des Timeouts vorzeitig als abgebrochen markiert und die gesamte Anwendung dadurch abgebrochen.

7.3.5. Rekonfiguration des Clusters nicht möglich

Cluster nicht mehr rekonfigurieren auch in Anforderungen

Insgesamt 13 der 42 ausgeführten Tests wurden vorzeitig abgebrochen, da eine Rekonfiguration des Clusters nicht möglich war. Dies entspricht dem in Abschnitt 6.1.3 definierten Verhalten und in Unterabschnitt 3.2.1 gefordertem Verhalten, wenn alle Node im Cluster defekt sind. Im Folgenden wird für die betroffenen Testkonfigurationen und Tests betrachtet, weshalb es dazu kam.

Testkonfigurationen 3 bis 6

Erstmalig ist ein Abbruch im Test 4 aufgetreten, auch die weiteren korrespondierende Tests der Konfigurationen 5 und 6 wurden abgebrochen. Hier waren bereits beim 3. Testfall alle verfügbaren Nodes beendet, was auffällig ist, vor allem da damit die Hälfte der Tests mit dem ersten Seed und dem Cluster auf einem Host vorzeitig abgebrochen wurden. Dies liegt einerseits daran, dass im Gegensatz zu den beiden Konfigurationen mit nur zwei Clients hier bis zu vier Anwendungen gleichzeitig gestartet werden, was die Last auf den Nodes deutlich erhöht. In Test 3, welcher somit theoretisch ebenfalls abgebrochen werden hätte müssen, wurden 11 Anwendungen im Cluster gestartet. Dies liegt an der geringeren Auslastung eines einzelnen Nodes im Gegensatz zu den anderen Tests. In den abgebrochenen Tests hatte Node 4 im ersten Testfall eine hohe bzw. sehr hohe Auslastung, im Test 3 jedoch nur eine mittlere. Diese mittlere Auslastung reichte jedoch aus, um den Node im ausgeführten 3. Testfall gemäß

deaktivieren von komponentenfehler

wieder zu aktivieren, während die anderen noch aktiven Nodes spätestens in diesem Testfall einen Komponentenfehler injiziert bekamen. Durch diesen einen nun weiterhin ausgeführten Node ist es dem Cluster daher möglich gewesen, sich im Test 3 zu rekonfigurieren.

Testkonfigurationen 15 und 16

Die Ausführung der Tests 13 bzw. 14 und 15 bzw. 16 unterscheidet sich nur in der Anzahl der Testfälle der jeweiligen Testkonfiguration. Dementsprechend wurden die äquivalenten Tests 13 und 14 im Gegensatz zu den beiden anderen vollständig ausgeführt, da der Abbruch der Tests 15 und 16 im sechsten ausgeführten Testfall stattfand. Die Nodes hatten in den vier Tests folgende Auslastung:

Test	13	14	15	16
Fehlerhafte Nodes	2	2	3	1
Auslastung in Prozent	47	97	96	98

Tabelle 7.2.: Status der Nodes im fünften Testfall der Tests 13 bis 16. Der Wert der Auslastung ist die kumulierte Auslastung aller noch aktiven Nodes.

Bei den beiden betroffenen Tests 15 und 16 sehr hohe Auslastung der noch aktven Nodes im 5. Testfall führte im darauf folgenden Testfall dazu, dass bei allen noch aktiven Nodes ein Komponentenfehler aktiviert wurde. Daher wurden die beiden Tests im jeweils 6. ausgeführten Testfall abgebrochen.

Testkonfigurationen 19 bis 22

Bei den Konfigurationen 19 bis 22 verhält es sich ähnlich wie bei den Konfigurationen 3 bis 6. Analog dazu wurde auch Test 19 nicht vorzeitig abgebrochen, die Tests 20 bis 22 im vierten Testfall dagegen schon.

Alle vier Tests haben gemeinsam, dass im jeweils dritten Testfall lediglich Node 1 inaktiv ist. Bei den beiden Tests ohne Mutationsszenario wurde hierbei jeweils die Verbindung zum Node im Testfall zuvor getrennt, bei den Mutationstests wurde der Node durch einen Komponentenfehler beendet. Dies liegt in der Historie des Nodes innerhalb des Tests begründet:

Test	19	20	21	22
Testfall 1	93 %	0 %	100 %	0 %
Testfall 2	Verbindung getrennt	100 %	Verbindung getrennt	93 %
Testfall 3	-	Node beendet	-	Node beendet
Testfall 4	Verbunden 93 %	-	*Verbunden*	-

Tabelle 7.3.: Auslastungen und injizierte Komponentenfehler in Node 1 in den Tests 19 bis 22

Die Aktivierung und Deaktivierung der Komponentenfehler in den anderen Nodes ist in allen Tests gleich und daher zur Ermittlung der Gründe des Abbruchs der Testausführung nicht relevant. Durch die unterschiedliche Auslastungen in den Testfällen 1 der Tests zwischen Tests ohne Mutationen (19 und 21) und mit Mutationen (20 und 22) wurden unterschiedliche Komponentenfehler aktiviert. Dies führte dazu, dass der relevante Node 1 bei den Mutationsstests nicht gestartet wurde, während die anderen Nodes beendet wurden wie in den Tests 19 und 21.

Eine Besonderheit bildet hier zudem Test 21, bei dem der Komponentenfehler vom Testsystem deaktiviert wurde, jedoch nicht repariert werden konnte. Dies liegt darin, dass der Docker-Container nicht mit dem Docker-Netzwerk verbunden werden konnte. Aus diesem Grund wurde vom Oracle bei der Prüfung der Rekonfigurierbarkeit des Clusters der Test entsprechend beendet, da der Node nicht verbunden war. Zwar wurde der Fehler von Docker nicht absichtlich oder durch das Testsystem herbeigeführt, jedoch zeigt dies auch, dass der Fehler ebenfalls erkannt werden konnte.

Testkonfigurationen 27 und 28

In den beiden Tests 28.1 und 28.2 wird der Test im 8. Testfall abgebrochen, während Test 27 nach allen 10 Testfällen regulär beendet wird. Das liegt daran, dass im 8. Testfall bei den beiden Mutationstests in fünf der sechs Nodes ein Komponentenfehler injiziert wird, von Node 1 wird die Verbindung getrennt, die Nodes 3 bis 6 komplett

beendet. Im Test 27 ohne Mutationsszenario wird dagegen zwar auch die Verbindung von Node 1 getrennt, aber zusätzlich nur Node 3 beendet, sodass die Nodes 4 bis 6 weiterhin aktiv sind. Node 2 wird in allen drei Tests bereits im 3. Testfall beendet, da die Auslastung des Nodes im 2. Testfall bei jeweils über 90 Prozent liegt. Die übrigen der 19 bzw. 20 aktivierten und zwischen 10 und 13 wieder deaktivierten Komponentenfehlern unterschieden sich in den drei Tests bis auf einzelne, hier nicht relevante, Ausnahmen nicht.

Der Grund für die Injizierung von Komponentenfehlern bei noch allen aktiven Nodes im 8. Testfall liegt in der Auslastung der Nodes im 7. Testfall. Diese beträgt im Test 27 ohne Mutationen bei den beiden betroffenen Nodes bei jeweils 100 Prozent, bei den übrigen Nodes ist jedoch keine bzw. eine geringe Auslastung vorhanden. In den beiden Tests der Konfiguration 28 ist das Cluster jeweils vollständig Ausgelastet, wodurch die Wahrscheinlichkeit zur Aktivierung der Komponentenfehler im folgenden Testfall stark ansteigt. Dadurch war es möglich, dass alle noch aktiven Nodes vom Cluster getrennt bzw. beendet wurden und der Test aufgrund fehlender Rekonfigurationsmöglichkeiten abgebrochen wurde.

Testkonfigurationen 31 und 32

Die beiden Tests 31 und 32 verliefen beide ähnlich zueinander. Die hohe Anzahl der 19 bzw. 20 aktivierten Komponentenfehler reichten bei jeweils 11 wieder deaktivierten Fehlern aus, um die beiden Tests im 8. Testfall zu beenden.

Besonders war hier, dass es bei jeweils mehreren Testfällen vorgekommen ist, dass mehr als 3 Komponentenfehler aktiviert bzw. deaktiviert wurden. So kam es vor, dass z. B. in 3. Testfall bereits eine Rekonfiguration nur deshalb möglich war, weil der zuvor vom Cluster getrennte Node 1 wieder mit dem Cluster verbunden wurde, während die Nodes 2 und 4 bis 6 anderen Nodes getrennt oder beendet wurden, Node 3 wurde bereits im Testfall zuvor beendet. Dies trifft auch auf die beiden korrespondierenden Konfigurationen 29 und 30 zu, bei denen nur fünf Testfälle ausgeführt wurden.

Bis auf den beendeten Node 2 wurden spätestens im 6. ausgeführten Testfall die im 3. Testfall injizierten Komponentenfehler wieder repariert. Zwar wurde im 7. Testfall je ein Komponentenfehler repariert, jedoch im Test ohne Mutationen auch ein weiterer injiziert. In Kombination mit den drei bzw. vier aktivierten Komponentenfehlern in Testfall 8 führte das daher dazu, dass kein aktiver Node im Cluster mehr vorhanden war und der Test entsprechend abgebrochen wurde.

7.3.6. Nicht erkannte oder gespeicherte Daten des Clusters

Einige Daten des Clusters wurden nicht im Testsystem gespeichert bzw. im Programmlog ausgegeben. Dies verstößt damit gegen die in Unterabschnitt 3.2.1 definierte Anforderung an das Testsystem, dass der jeweils aktuelle Status des Clusters erkannt und im Modell

gespeichert werden muss. Vorgekommen ist das auf zwei Arten, die im folgenden erläutert werden.

Nicht erkannte Nodes auf Host 2

Einer der beiden Fälle ist, dass ausführende Nodes von Anwendungen bzw. Attempts nicht erkannt bzw. ausgegeben wurden. Hier geht es jedoch nicht um Anwendungen bzw. Attempts, die zwar bereits gestartet wurden, für die aber noch kein AppMstr allokiert werden konnte. In diesen Fällen ist es daher das normale Verhalten von Hadoop, keinen ausführenden Node anzugeben, da keiner vorhanden ist. Wenn dieser Status zu lange anhält, wurden die Attempts bzw. AppMstr durch Hadoop mit einem Timeout beendet.

Anders sieht das jedoch in den sechs Tests 7.1, 8 und 23 bis 26 aus. In diesen Tests wurden zwar regulär die Daten der Nodes ermittelt und auch in den Logdateien ausgegeben, jedoch nicht alle ausführenden Nodes von Anwendungen und Attempts. Konkret betrifft das hier alle Nodes der betroffenen AppMstr auf Host 2, da in den betroffenen sechs Tests die Nodes erkannt und auch ausgegeben wurden, wenn sich diese auf Host 1 befunden haben, also Node 1 bis Node 4. Während die betroffenen Nodes gemäß des SSH-Logs fast immer von Hadoop Übertragen wurden, wurden die Nodes auf Host 2 in einigen Tests jedoch nicht gespeichert. Zwar tritt hierbei ein gewisses Muster auf (pro Seed die jeweils zuerst ausgeführten Tests mit Nodes auf beiden Hosts), allerdings konnte dieser Fehler nicht gezielt reproduziert werden. Bei der erneuten Ausführung der Testkonfiguration 7 (Test 7.2) wurden alle Nodes korrekt erkannt und vom Testsystem im Programmlog gespeichert. Zum gegenwärtigen Zeitpunkt kann daher nicht gesagt werden, weshalb die ausführenden Nodes in den betroffenen Testfällen nicht immer gespeichert wurden. Es kann nur vermutet werden, dass während dem Parsen der übertragenen Daten mit diesen Daten die betroffenen Nodes im Modell nicht gefunden werden konnten (vgl.

Parser/Node-Speicherung, da erklären, dass Objekt vom Node gespeichert wird und nicht ID

). Die Gründe dafür sind jedoch unklar.

Diagnostic-Daten von Anwendungen

Was bei allen Tests aufgefallen ist, dass die Diagnostic-Daten von Anwendungen nicht im Programmlog enthalten sind. Auch hier wurden die entsprechenden Diagnostic-Daten von Hadoop an das Testsystem übertragen, dort jedoch nicht gespeichert im Gegensatz zu den Diagnostic-Daten der Attempts. Zur Auswertung der Daten im Rahmen der Evaluation ist dies zwar irrelevant, da dies auch aufgrund der Daten der Attempts geschehen kann, allerdings wird die entsprechende Anforderung an das Testsystem nicht erfüllt, wonach die Daten gespeichert werden müssen.

Eine Analyse ergab, dass die Diagnostic-Daten der Anwendungen aufgrund eines falsch gesetzten Attributs in der `ApplicationResult`-Klasse des Parsers nicht im Testsystem gespeichert werden konnten.

Reflexion: Hätte bei vorabtests erkannt werden müssen!

Da die Diagnostic-Daten der Anwendungen eine Zusammenfassung der gesamten Anwendung darstellen, und alle Diagnostic-Daten bereits durch die der Attempts vorhanden waren, wurde hier auf erneute Testausführungen verzichtet.

7.3.7. Nicht erkannte, injizierte bzw. reparierte Komponentenfehler

7.3.8. Nicht gestartete Anwendungen

8. Reflexion und Abschluss

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*. 10. Mai 2016. 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] zuletzt bearbeitet von XingWang. *Powered by Apache Hadoop*. 5. Apr. 2018. URL: https://wiki.apache.org/hadoop/PoweredBy (besucht am 24. 06. 2018).

[9] 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.

[10] Jeffrey Dean und Sanjay Ghemawat. „MapReduce: A Flexible Data Processing Tool“. In: *Commun. ACM* 53.1 (Jan. 2010), S. 72–77. ISSN: 0001-0782. DOI: 10.1145/1629175.1629198. URL: http://doi.acm.org/10.1145/1629175.1629198.

[11] 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).

[12] Kyong-Ha Lee u. a. „Parallel Data Processing with MapReduce: A Survey“. In: *SIGMOD Rec.* 40.4 (Jan. 2012), S. 11–20. ISSN: 0163-5808. DOI: 10.1145/2094114.2094118. URL: http://doi.acm.org/10.1145/2094114.2094118.

[13] Vinod Kumar Vavilapalli u. a. „Apache Hadoop YARN: Yet Another Resource Negotiator“. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633. URL: http://doi.acm.org/10.1145/2523616.2523633.

[14] 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).

[15] 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).

[16] 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).

[17] 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).

[18] 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).

[19] 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.

[20] Rudolph Emil Kálmán. „A new approach to linear filtering and prediction problems“. In: *Journal of basic Engineering* 82.1 (1960), S. 35–45.

[21] Reiner Marchthaler und Sebastian Dingler. *Kalman-Filter*. Wiesbaden: Springer Vieweg, 2017. ISBN: 9783658167271.

[22] Urs Strukov. „Anwendung des Kalman-Filters zur Komplexitätsreduktion im Controlling“. Diss. 2001.

[23] Phil Kim. *Kalman-Filter für Einsteiger*. 1. Auflage. Wrocław: Amazon Fulfillment Poland Sp. z o.o, 2016. ISBN: 9781502723789.

[24] Dan Simon. *Optimal state estimation*. Hoboken, NJ: Wiley-Interscience, 2006. ISBN: 9780471708582.

[25] Lakhdar Aggoun und Robert J. Elliott. *Measure theory and filtering*. 1. publ. Cambridge series in statistical and probabilistic mathematics. Cambridge u.a.: Cambridge Univ. Press, 2004. ISBN: 9780521838030.

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

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

[28] Benedikt Eberhardinger u. a. *Case Study: Adaptive Test Automation for Testing an Adaptive Hadoop Resource Manager*. Institute for Software & Systems Engineering, University of Augsburg, Apr. 2018.

[29] 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).

[30] 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).

[31] 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).

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

[33] 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.

[34] 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).

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

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

[37] 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.

[38] 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).

[39] 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.

[40] Alex Groce u. a. „An Extensible, Regular-Expression-Based Tool for Multi-Language Mutant Generation". In: (Mai 2018).

[41] Charlie Poole und Rob Prouse. *universalmutator/genmutants.py*. 26. Mai 2018. URL: https://github.com/agroce/universalmutator/blob/R0.8.13/universalmutator/genmutants.py (besucht am 09. 06. 2018).

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 Dokumentation von Hadoop in [29] zu finden.

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

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

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

```
$ yarn application --status application_1518100641776_0001
[...]
Application Report :
    Application-Id : application_1518100641776_0001
    Application-Name : QuasiMonteCarlo
    Application-Type : MAPREDUCE
    User : root
    Queue : default
    Start-Time : 1518103712160
```

```
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 gezeigten und auch in der Fallstudie genutzten 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 sind die Rückgaben für Listen von Anwendungen, Attempts, Container und der Nodes vom RM und bzw. NM (Container) sowie des TLS (Attempts und Container)relevant. Weitere Informationen zur REST-API sind in der Dokumentation in [15, 30, 31] 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,
```

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

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

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

C. Ausgabeformat des Programmlogs

Die in Unterabschnitt 3.2.3 und Abschnitt 6.1.3 beschriebenen Ausgaben werden im nachfolgend dargestellten Format gespeichert. Es handelt sich hierbei um den gekürzten Programmlog der Ausführung des Testfalls #1 (vgl. Anhang D).

Listing C.1: Ausgaben einer Simulation im Programmlog (gekürzt)

```
Starting Case Study test
Parameter:
  benchmarkSeed=    0x0AB4FEDD (179633885)
  faultProbability= 0,3
  hostsCount=       1
  clientCount=      2
  stepCount=        5
  isMutated=        False
Start cluster on 1 hosts (mutated: False)
Is cluster started: True
Setting up test case
=================  START  ====================================
Starting Simulation test
Base benchmark seed: 179633885
Min Step time:       00:00:25
Step count:          5
Fault probability:   0,3
Fault repair prob.:  0,3
Inputs precreated:   False
Host mode:           Multihost
Hosts count:         1
Node base count:     4
Full node count:     4
Setup script path:   ~/hadoop-benchmark/multihost.sh -q
Controller url:      http://localhost:8088
Simulating Benchmarks for Client 1 with Seed 179633886:
Step 0: dfsiowrite
Step 1: dfsiowrite
Step 2: dfsioread
Step 3: dfsioread
Step 4: dfsioread
Simulating Benchmarks for Client 2 with Seed 179633887:
Step 0: randomwriter
Step 1: randomwriter
Step 2: randomwriter
Step 3: pi
Step 4: pi
================  Step: 0  ====================================
```

```
Fault NodeConnectionErrorFault@compute-1
Activation probability: 0,94 < 0,536382840264767
Fault NodeDeadFault@compute-1
Activation probability: 0,94 < 0,0263571413356611
Fault NodeConnectionErrorFault@compute-2
Activation probability: 0,94 < 0,0658538276636292
Fault NodeDeadFault@compute-2
Activation probability: 0,94 < 0,405010061992803
Fault NodeConnectionErrorFault@compute-3
Activation probability: 0,94 < 0,662199801608082
Fault NodeDeadFault@compute-3
Activation probability: 0,94 < 0,666254943546958
Fault NodeConnectionErrorFault@compute-4
Activation probability: 0,94 < 0,194108738188682
Fault NodeDeadFault@compute-4
Activation probability: 0,94 < 0,763726766111202
Selected Benchmark client1: dfsiowrite
Selected Benchmark client2: randomwriter
Checking SuT constraints.
Checking test constraints
 YARN component not valid: Constraint 0 in Controller
Step Duration: 00:00:42.5186656
=== Controller ===
MARP Value on start: 0,1
MARP value on end:   0,1
=== Node compute-1:45454 ===
    State:          RUNNING
    IsActive:       True
    IsConnected:    True
    Container Cnt: 4
    Mem used/free: 4096/4096 (0,500)
    CPU used/free: 4/4 (0,500)
    Health Report:
=== Node compute-2:45454 ===
    State:          RUNNING
    IsActive:       True
    IsConnected:    True
    Container Cnt: 8
    Mem used/free: 8192/0 (1,000)
    CPU used/free: 8/0 (1,000)
    Health Report:
=== Node compute-3:45454 ===
    State:          RUNNING
    IsActive:       True
    IsConnected:    True
    Container Cnt: 2
    Mem used/free: 4096/4096 (0,500)
    CPU used/free: 2/6 (0,250)
```

```
    Health Report:
=== Node compute-4:45454 ===
    State:         RUNNING
    IsActive:      True
    IsConnected:   True
    Container Cnt: 0
    Mem used/free: 0/8192 (0,000)
    CPU used/free: 0/8 (0,000)
    Health Report:
=== Client client1 ===
    Current executing bench:  dfsiowrite
    Current executing app id: application_1529401644907_0001
    === App application_1529401644907_0001 ===
        Name:        hadoop-mapreduce-client-jobclient-2.7.1-tests.jar
        State:       RUNNING
        FinalStatus: UNDEFINED
        IsKillable:  True
        AM Host:     compute-3:45454 (RUNNING)
        Diagnostics:
        === Attempt appattempt_1529401644907_0001_000001 ===
            State:        None
            AM Container: container_1529401644907_0001_01_000001
            AM Host:      compute-3:45454 (RUNNING)
            Cont. Count:  13
            Detected Cnt: 13
            Diagnostics:
=== Client client2 ===
    Current executing bench:  randomwriter
    Current executing app id: application_1529401644907_0002
    === App application_1529401644907_0002 ===
        Name:        random-writer
        State:       RUNNING
        FinalStatus: UNDEFINED
        IsKillable:  True
        AM Host:     compute-3:45454 (RUNNING)
        Diagnostics:
        === Attempt appattempt_1529401644907_0002_000001 ===
            State:        FINISHED
            AM Container: container_1529401644907_0002_01_000001
            AM Host:      compute-3:45454 (RUNNING)
            Cont. Count:  2
            Detected Cnt: 2
            Diagnostics:
...
================  Step: 2  ====================================
...
Fault NodeConnectionErrorFault@compute-3
Activation probability: 0,8875 < 0,799780692346292
```

```
135 Fault NodeDeadFault@compute-3
136 Activation probability: 0,8875 < 0,962228187807942
137 ...
138 Stop node compute-3
139 Selected Benchmark client1: dfsioread
140 Selected Benchmark client2: randomwriter
141 Checking SuT constraints.
142 Checking test constraints
143  YARN component not valid: Constraint 0 in Controller
144 Step Duration: 00:00:42.7485612
145 ...
146 ================  Finish  ====================================
147 Final status of the cluster:
148 ...
149 Finishing test.
150 Simulation Duration: 00:02:44.3497896
151 Successfull Steps:    5
152 Activated Faults:     5/40
153 Repaired Faults:      3
154 Last detected MARP:   0,1
155 Executed apps:        4
156 Successed apps:       3
157 Failed apps:          1
158 Killed apps:          0
159 Executed attempts:    5
160 Detected containers:  36
161 Checked Constraints:  270 SuT / 261 Test
162 Failed Constraints:   3 SuT / 6 Test
163 Killing running apps.
164 Stop cluster
165 Is cluster stopped: True
```

D. Übersicht ausgeführter Testkonfigurationen

Die folgende Tabelle gibt eine Übersicht über die für diese Fallstudie ausgeführten Testkonfigurationen. Die Auswahl der Konfigurationen ist in Abschnitt 6.3 beschrieben.

Die Spalte *Mutanten* gibt an, welche der in Abschnitt 6.2 beschriebenen Mutanten in der Selfbalancing-Komponente genutzt wurden. In den Spalten *Ausgeführte Testfälle* und *Dauer* ist angegeben, wie viele Testfälle bzw. Simulations-Schritte jeweils ausgeführt wurden, bzw. wie lang die jeweiligen Simulationen in Minuten und Sekunden gedauert haben. Wenn nicht alle möglichen Testfälle ausgeführt wurden, war im darauf folgenden Testfall eine Rekonfiguration des Clusters nicht mehr möglich und die Simulation wurde abgebrochen (vgl. Abschnitt 6.1.3).

Die Nummerierung der Konfigurationen bzw. Ausführungen erfolgte basierend auf den grundlegenden Testkonfigurationen bestehend aus Seed, Anzahl der Hosts, Clients und ausgeführten Testfällen sowie der Angabe, ob ein Mutationsszenario verwendet wurde. Bei mehrmals ausgeführten Testkonfigurationen ist der Konfiguration eine entsprechende Ziffer angehängt, um die jeweilige Ausführung zu Kennzeichnen. Eine Besonderheit bildet hierbei die Testkonfiguration 10 mit insgesamt 6 Ausführungen, da diese Konfiguration mit verschiedenen Mutanten durchgeführt wurde.

#	Seed	Hosts	Clients	Testfälle	Mutanten	Ausgeführte Testfälle	Dauer
1.1	0xAB4FEDD	1	2	5	keine	5	2:44
1.2						5	2:56
2	0xAB4FEDD	1	2	5	1,2,3,4	5	2:34
3	0xAB4FEDD	1	4	5	keine	5	5:52
4	0xAB4FEDD	1	4	5	1,2,3,4	2	3:13
6	0xAB4FEDD	1	4	10	1,2,3,4	2	3:14
5.1	0xAB4FEDD	1	4	10	keine	2	3:35
5.2						2	3:23
7.1	0xAB4FEDD	2	2	5	keine	5	2:49
7.2						5	2:56
8	0xAB4FEDD	2	2	5	1,2,3,4	5	2:23
9.1	0xAB4FEDD	2	4	5	keine	5	07:13
9.2						5	4:49
10.1	0xAB4FEDD	2	4	5	1,2,3,4	5	7:42
10.2					1	5	6:17
10.3					2	5	6:04
10.4					3	5	6:37
10.5					3	5	6:21
10.6					4	5	6:26
11	0xAB4FEDD	2	4	10	keine	10	12:16
12	0xAB4FEDD	2	4	10	1,2,3,4	10	11:36
13	0xAB4FEDD	2	6	5	keine	5	8:02
14	0xAB4FEDD	2	6	5	1,2,3,4	5	6:24
15	0xAB4FEDD	2	6	10	keine	5	8:41
16	0xAB4FEDD	2	6	10	1,2,3,4	5	9:26
17	0x11399D3	1	2	5	keine	5	3:07
18	0x11399D3	1	2	5	1,2,3,4	5	3:02
19	0x11399D3	1	4	5	keine	5	5:25
20	0x11399D3	1	4	5	1,2,3,4	3	3:22
21	0x11399D3	1	4	10	keine	3	4:17
22	0x11399D3	1	4	10	1,2,3,4	3	2:50
23	0x11399D3	2	2	5	keine	5	4:25
24	0x11399D3	2	2	5	1,2,3,4	5	4:22
25	0x11399D3	2	4	5	keine	5	4:53
26	0x11399D3	2	4	5	1,2,3,4	5	5:47
27	0x11399D3	2	4	10	keine	10	10:30
28.1	0x11399D3	2	4	10	1,2,3,4	7	8:17
28.2						7	7:37
29	0x11399D3	2	6	5	keine	5	7:03
30	0x11399D3	2	6	5	1,2,3,4	5	6:02
31	0x11399D3	2	6	10	keine	7	10:21
32	0x11399D3	2	6	10	1,2,3,4	7	11:08

Tabelle D.1.: Übersicht der ausgeführten Testkonfigurationen