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:	10. Juli 2018
Betreuer:	M.Sc. Benedikt Eberhardinger
1. Gutachter:	Prof. Dr. Wolfgang Reif
2. Gutachter:	Prof. Dr. Bernhard Bauer

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

Einleitungstext

2.1. Safety Sharp

Testen unter SS allgemein genauer erklären und an neue Abschnitte anpassen

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].

2.1.1. Aufbau eines Modells

Hier Oracle, Modell selbst und so rein

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 MyCOmponent : Component
2 {
3   // fault definition, also possible: new PermanentFault()
4   public readonly Fault MyFault = new TransientFault();
5
6   // interaction logic (Fields, Properties, Methods...)
```

```
7
8   // fault effect
9   [FaultEffect(Fault = nameof(MyFault))]
10  internal class MyFaultEffect : MyCOmponent
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].

2.1.2. Ausführung eines Modells mit S#

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. 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

[1] `http://ltsmin.utwente.nl/`
[2] `https://hadoop.apache.org/`

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]. 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*:

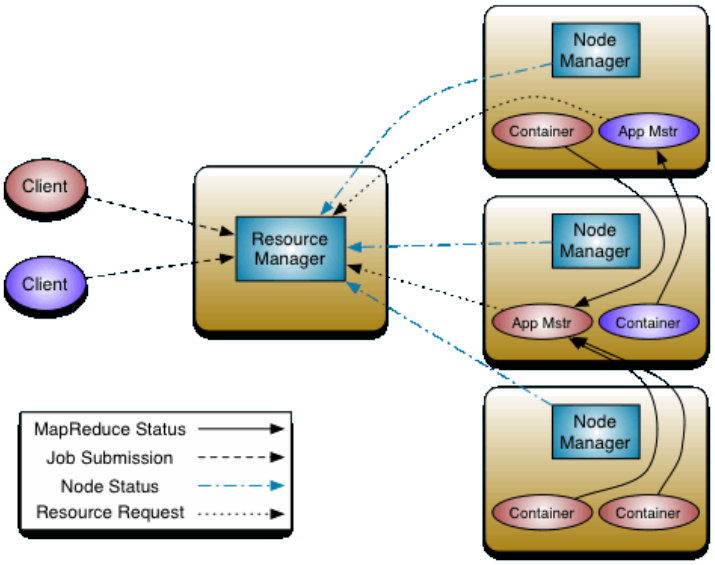

Abbildung 2.1.: Architektur des YARN-Frameworks (entnommen aus [14])

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.

Zu erwähnen ist hier zudem, dass die Kommunikation zwischen den einzelnen Komponenten nicht in allen Fällen in Echtzeit statt findet. Vor allem das Prüfen des generellen Node-Zustandes durch den RM wird bei einer Standard-Konfiguration in periodischen Abständen von jeweils mehreren Minuten durchgeführt. Sollte der NM bei solchen Status-Abfragen zunächst nicht reagieren, wird mehrere Minuten gewartet, bis der Node als defekt erkannt wird. Ähnlich verhält es sich bei Zustandsabfragen an den AppMstr [15].

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 [16].

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

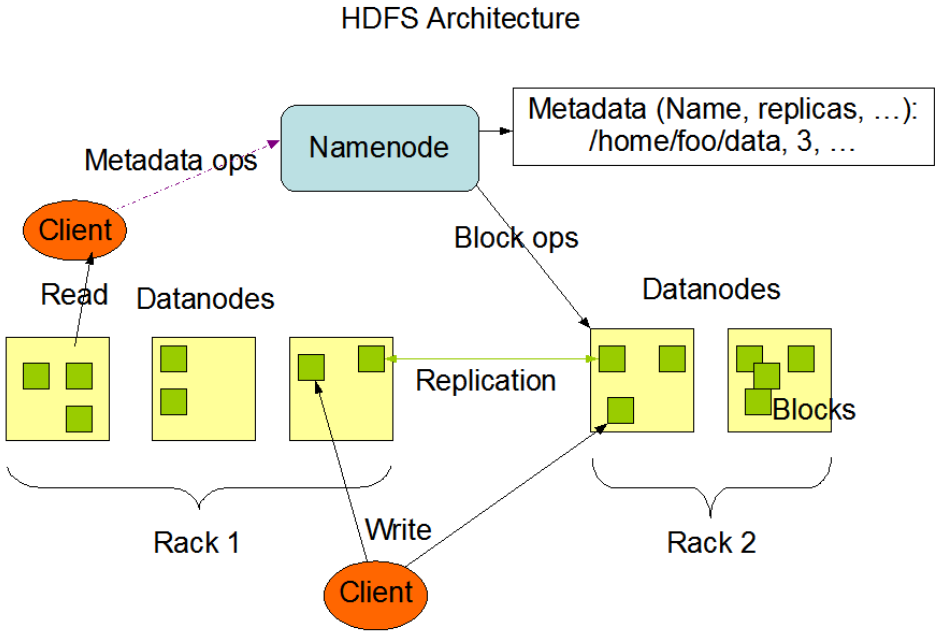

Abbildung 2.2.: Architektur des HDFS (entnommen aus [17])

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 [18]. 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 [17].

Das Überprüfen der DataNodes durch den NameNode erfolgt genauso wie bei den entsprechenden YARN-Komponenten periodisch im Abstand von mehreren Minuten. Auch hier dauert es bei einer Standard-Konfiguration daher mehrere Minuten, bis erkannt wird, wenn ein DataNode nicht mehr verfügbar ist [19].

2.3. Adaptive Komponente in Hadoop

Eine normale Hadoop-Installation besitzt keine adaptive Komponente, sondern rein statische Einstellungen. Um damit Hadoop zu optimieren, müssen die Einstellungen daher immer manuell auf den jeweils benötigten Anwendungstyp angepasst werden. Dazu gibt es 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 [20]. Für diesen Scheduler wurde in [21] ein selbst-adaptiver Ansatz vorgestellt, welcher im Folgenden genauer erläutert wird.

2.3.1. MARP-Wert

Der Capacity Scheduler verschiedene Einstellungen, um ihn für das konkrete Cluster anzupassen. So besteht z. B. die Möglichkeit, den verfügbaren Speicher pro Anwendungs-Container festzulegen oder wie viel Ressourcen durch AppMstr-Container beansprucht werden darf. Vor allem letztere Einstellungsmöglichkeit namens `maximum-am-resource-percent` (MARP) ist sehr wichtig, wenn mehrere Anwendungen gleichzeitig ausgeführt werden sollen. Der in der Konfiguration des Schedulers definierte MARP-Wert gibt an, wie viel Prozent des verfügbaren Speichers durch AppMstr-Container genutzt werden darf [20]. Der gesamte, für Anwendungen verfügbare Speicher wird durch den MARP-Wert in zwei Teile aufgeteilt. Während der einen Teil des Speichers nur durch AppMstr-Container beansprucht werden darf, wird der andere Teil des Speichers durch alle anderen Anwendungs-Container genutzt. Wird durch den MARP-Wert der erste, für für AppMstr-reservierten Teil zu klein gehalten, können daher weniger AppMstr allokiert werden und somit auch weniger Anwendungen gestartet werden (*Loss of Jobs Parallelism*, LoJP). Ist der MARP-Wert dagegen zu groß, wird der verfügbare Speicher zu entsprechend großen Teilen für mögliche AppMstr reserviert. Dadurch ist der Anteil des Speichers für Anwendungs-Container entsprechend klein und es können dadurch

weniger Container gestartet werden, um eine Anwendung auszuführen, womit sich die Ausführungsgeschwindigkeit der Anwendungen verringert (*Loss of Job Throughput*, LoJT) [21]:

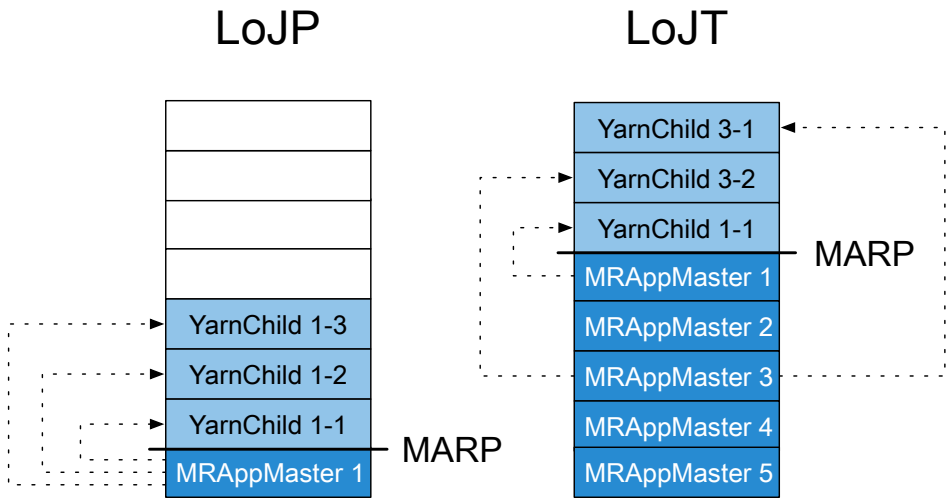

Abbildung 2.3.: LoJP und LoJT in Hadoop (entnommen aus [21]). Während beim LoJP sehr viel Speicher für Anwendungs-Container ungenutzt bleibt, können beim LoJT nicht genügend Anwendungs-Container allokiert werden, um die Anwendungen auszuführen.

Damit bestimmt der MARP-Wert indirekt auch die maximale Anzahl an Anwendungen, die gleichzeitig ausgeführt werden können. Da der MARP-Wert jedoch nicht während der Laufzeit dynamisch angepasst werden kann, haben Zhang u. a. in [21] einen Ansatz zur dynamischen Anpassung des MARP-Wertes zur Laufzeit von Hadoop vorgestellt. Die entwickelte **Selfbalancing-Komponente** passt den MARP-Wert abhängig von der Speicherauslastung der ausgeführten Anwendungen dynamisch zur Laufzeit an. So wird der MARP-Wert verringert, wenn die Speicherauslastung sehr hoch ist, und erhöht, wenn die Speicherauslastung sehr niedrig ist. Die Selfbalancing-Komponente ermöglicht daher, dass immer die maximal mögliche Anzahl an Anwendungen ausgeführt werden kann. Die Evaluation von Zhang u. a. ergab zudem, dass Anwendungen dadurch im Schnitt um bis zu 40 Prozent schneller ausgeführt werden können. Zudem kann die dynamische Anpassung auch effizienter sein, als eine manuelle, statische Optimierung [21].

2.3.2. Analyse der Selfbalancing-Komponente

Da in dieser Fallstudie auch Mutationstests eingesetzt werden, bei denen die Selfbalancing-Komponente entsprechend verändert wird (Implementierung in Abschnitt 6.2), wurde die 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 [22] beschrieben und wird genutzt, um „aus verrauschten und teils redundanten Messungen die Zustände und Parameter des Systems zu schätzen“ [23]. Der Filter lässt sich aufgrund seines Aufbaus zudem auch für Echtzeitanwendungen nutzen [23]. Als einfaches Anwendungsbeispiel hierfür ist in [23] die Apollo-Mondlandefähre genannt, Strukov nutzte ihn in [24] 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. [25–27] verwiesen.

Die drei Shell-Scripte der Selfbalacing-Komponente dienen zur Interaktion zwischen der Komponente und dem Hadoop-Cluster. Die beiden zuletzt genannten Scripte werden von den beiden Monitor-Klassen sekündlich gestartet und ermitteln basierend auf den Logs von Hadoop 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 der `controlNodeLog`-Datei gespeichert. Durch die Ausführung von `selfTuning-mem-controlNode.sh` (gestartet durch `MemUtilization`) wird dagegen die Auslastung des Speichers des Clusters ermittelt und in der `memLog`-Datei notiert.

Die in den beiden Dateien enthaltenen Werten werden im Anschluss wiederum sekündlich vom `Controller` der Selfbalancing-Komponente ausgelesen und mithilfe des Kalman-Filters bereinigt. Anschließend werden die bereits in [21] 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.

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

2.4. Plattform Hadoop-Benchmark

Zhang u. a. haben im Rahmen ihrer gesamten Forschungsarbeit an der Selfbalancing-Komponente darüber hinaus auch die Open-Source-Plattform **Hadoop-Benchmark** entwickelt[4]. Sie dient zur einfachen und schnellen Ausführung eines Hadoop-Clusters und wurde speziell zum Einsatz in der Forschung erstellt. Dadurch kann sie auch mit geringem Aufwand an eigene Bedürfnisse angepasst werden.

Zur Ausführung des Clusters wird die Virtualisierungs-Software Docker[5] und das dazugehörige *Docker Machine* genutzt. Durch die Virtualisierung wird für jeden Hadoop-Node eine Docker-Machine gestartet, auf der der Hadoop-Node wiederum in einem Docker-Container ausgeführt wird. Verbunden werden die Nodes dabei mithilfe eines Docker-Swarms:

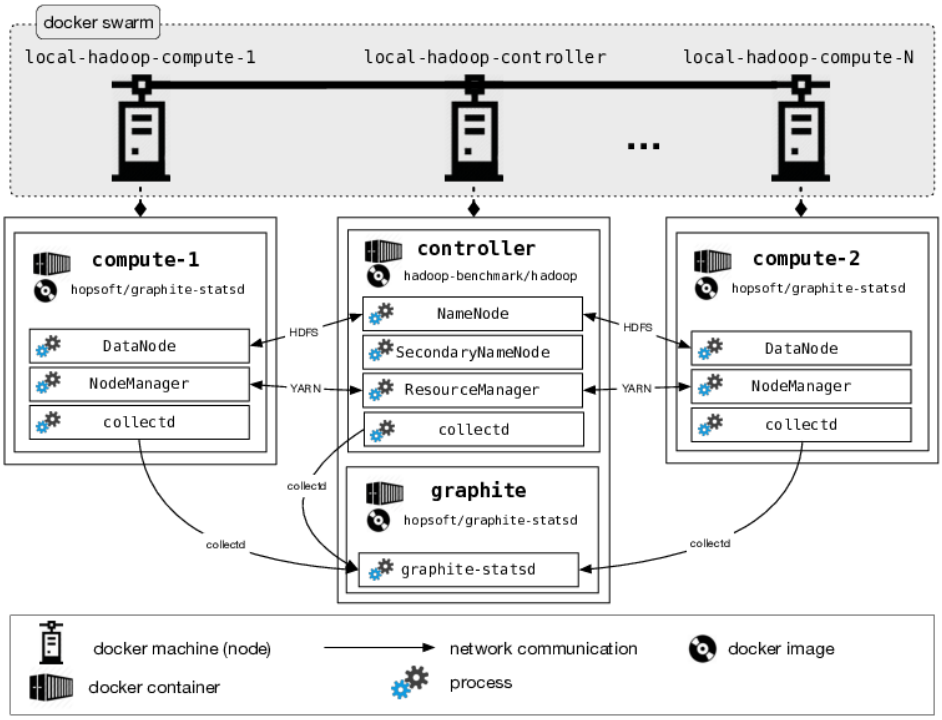

Abbildung 2.4.: High-Level-Architektur von Hadoop-Benchmark (entnommen aus [28])

Docker-Machine ist keine komplette Virtualisierungs-Software, sondern nutzt hier VirtualBox[6], um virtuelle Maschinen zu starten, die mit dem Betriebssystem *Boot2Docker* ausgestattet sind. Boot2Docker ist eine leichtgewichtige Linux-Distribution, auf der Docker bereits vorinstalliert ist [29].

Mit *Graphite*[7] ist zudem ein Monitoring-Tool enthalten, mit dem die Systemwerte wie CPU- oder Speicher-Auslastung des Clusters überwacht und analysiert werden kann. Jeder Hadoop-Container enthält dazu das Tool *collectd*[8], was das Monitoring

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

des Containers auf Systemebene übernimmt und die Daten an den Graphite-Container übermittelt.

Da mithilfe der Plattform auch unterschiedliche Hadoop-Konfigurationen ausgeführt werden können, ist die Plattform in mehrere Szenarien unterteilt. Jedes Szenario stellt eine Hadoop-Konfiguration dar, die vollständig angepasst werden kann. Jedes Szenario enthält daher eine *Dockerfile*, aus der die Docker-Images und -Container erstellt werden, weitere für Hadoop benötigte Daten und Einstellungen, sowie dazugehörige generelle Einstellungen des Szenarios. Die Plattform enthält bereits mehrere Szenarien, u. A. Hadoop in der Version 2.7.1 ohne Anpassungen sowie ein darauf basierendes Szenario mit der Selfbalancing-Komponente. Aufgrund eines der Kernkonzepte von Docker, wonach Docker-Images auf einem passenden, bereits vorhandenen Images aufbauen können bzw. sollten [30], ist es möglich, neue Szenarien basierend auf bereits vorhandenen zu entwickeln.

Die Plattform Hadoop-Benchmark enthält auch 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 physischem Host, sehr zeitintensiv sein kann. Beim Intel HiBench-Benchmark sind die verfügbaren Anwendungen dagegen in verschiedene *Worklaods* wie *Machine Learning* unterteilt. Hierbei können die enthaltenen Anwendungen einzeln oder als ganze Workloads auf dem Cluster ausgeführt werden. Einige der Hibench-Workloads bzw. Anwendungen basieren auf den Mapreduce Examples, welche wiederum voneinander unabhängige Beispielanwendungen für Hadoop darstellen.

Genauere Informationen zu den drei in der Plattform enthaltenen Benchmarks sind in Abschnitt 5.1 erläutert.

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

3. Aufbau und Ablauf 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.

Ergebnis zur Frage zur Testautomatistierung in Reflexion

Hierfür werden mehrere Anforderungen an das durch den grundlegenden Versuchsaufbau definierte Modell gestellt. Dieses Modell wird zur Realisierung der Tests mithilfe des S#-Frameworks als ein vereinfachtes Modell von Hadoop entwickelt und mit einem realen Cluster verbunden.

Teile der Beiträge und Inhalte dieses Kapitels wurden bereits in [31] publiziert.

3.1. Grundlegender Versuchsaufbau

Neben den Anforderungen an Hadoop und das gesamte Testsystem muss auch der grundlegende Versuchsaufbau und das System under Test (SuT) in definiert werden. Im Grunde wird, wie bereits in Kapitel 1 erwähnt, Hadoop mithilfe des S#-Frameworks nachgebildet und dieses Modell mit einem realen Cluster verbunden.

dort erwähnen

In diesem Cluster sollen anhand des Modells unterschiedliche Komponentenfehler injiziert und repariert werden, als auch unterschiedliche Benchmarks gestartet werden. Hierbei soll nicht nur das Verhalten von Hadoop selbst analysiert werden, sondern auch das der von Zhang u. a. entwickelten Selfbalancing-Komponente. Anhand dieses Verhaltens und dem des kompletten Testsystems soll schließlich ermittelt werden, ob eine Testautomatisierung in diesem Versuchsaufbau erfolgreich war. Es ist hierbei der gleiche Versuchsaufbau wie in [31], da dafür das gleiche Testsystem genutzt wurde wie für diese Fallstudie, in deren Rahmen es entwickelt wurde.

Bei der Entwicklung des Modells liegt der Fokus auf dem grundlegenden Aufbau von YARN. Dazu gehören die Anwendungen und ihre Attempts, sowie zum Teil auch ihre Container. Daneben muss das Modell auch die Nodes des Clusters und zum Ausführen der Benchmarks auch simulierte Clients enthalten. Da bei den Tests auch Ausfälle von Nodes eine Rolle spielen, müssen hierfür entsprechende Komponentenfehler implementiert werden, die mithilfe von S# aktiviert und deaktiviert werden können.

Da die Auswahl der ausgeführten Benchmarks eines jeden Clients nicht bei jedem Test manuell bestimmt werden soll, wird hierfür ein Transitionssystem verwendet. Mithilfe dieses Transitionssystems, in dem die Wahrscheinlichkeiten von Wechseln zwischen zwei

Anwendungen definiert sind, soll während der Ausführung eines Testfalls zufällig eine nachfolgende Anwendung ausgewählt werden.

Die Verbindung zwischen dem Modell und dem realen Hadoop-Cluster wird mithilfe eines dafür entwickelten Treibers durchgeführt. Der Treiber ist dafür verantwortlich, Komponentenfehler und Anwendungen an das reale Cluster zu senden. Zudem dient er dazu, um den Status des Clusters jederzeit ermitteln und an das Modell zur dortigen Speicherung übergeben zu können. Er kann daher nicht nur aus der Verbindung zum Cluster selbst bestehen, sondern muss auch die Kommunikation zwischen Modell und Cluster sicherstellen und übermittelte Daten entsprechend umwandeln.

Das SuT selbst stellt das reale Cluster dar, das mithilfe der von Zhang u. a. entwickelten Plattform Hadoop-Benchmark umgesetzt werden soll. Hierfür sollen basierend auf dem in der Plattform enthaltenen Szenario mit der Selfbalancing-Komponente für diese Fallstudie angepasste Szenarien genutzt werden. Zudem soll auch mithilfe von Mutationstests, bei denen einer oder mehrere Mutanten in der Selfbalancing-Komponente implementiert werden, das Testsystem geprüft werden.

Dieser Versuchsaufbau soll zudem mithilfe eines dafür entwickelten *Oracles* geprüft werden. Das Oracle dient zur Validierung der in Abschnitt 3.2 definierten Anforderungen an das Cluster und das Testsystem. Hierfür werden, sofern möglich, die Anforderungen als *Constraints* im Modell implementiert und bei jedem Test automatisch geprüft. Die Implementierung der Constraints erfolgt hierbei nicht zentral mithilfe des Oracles, sondern dezentral in den jeweiligen Komponenten des Modells. Für jede implementierte Komponente werden somit nur die jeweils relevanten Bestandteile der Anforderungen als Constraints implementiert und durch das Oracle validiert.

Die Implementierung des eben beschriebenen Modells und Oracles ist im Kapitel 4 beschrieben. Die Auswahl der verwendeten Benchmarks und deren Implementierung mit dem Transitionssystem findet sich in Kapitel 5.

3.2. Anforderungen an das Cluster und Testsystem

Zur Überprüfung des Clusters und des Testssystems selbst werden hierfür jeweils mehrere Anforderungen gestellt. Unterschieden wird hierbei zwischen funktionalen Anforderungen an das SuT und Anforderungen an das Testsystem. Während die funktionalen Anforderungen ausschließlich vom Hadoop-Cluster als SuT erfüllt werden müssen, müssen die Test-Anforderungen vom gesamten Testsystem erfüllt werden.

Mithilfe der im Folgenden definierten Anforderungen soll bereits automatisiert geprüft werden können, inwieweit eine Testautomatisierung möglich ist. Hierfür werden die Anforderungen, sofern möglich, in Form Form von Constraints ebenfalls im Modell implementiert. Mithilfe dieser Constraints können die Anforderungen somit ebenfalls automatisiert und bereits während der Testausführung durch das Oracle validiert werden.

3.2.1. 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 SuT, also das Hadoop-Cluster selbst, berücksichtigt werden. Da im Rahmen der Publikation [31] ebenfalls der in Abschnitt 3.1 beschriebene und in dieser Fallstudie genutzte Versuchsaufbau genutzt wurde, wurden im Rahmen dieser Fallstudie auch funktionale Anforderungen an das Cluster selbst durch das Oracle geprüft. Dies betrifft konkret folgende, bereits in [31] definierte, Anforderungen an das SuT:

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

3.2.2. Anforderungen an das Testsystem

Neben den funktionalen Anforderungen, gibt es weitere Anforderungen an das gesamte Testsystem. Diese Anforderungen betreffen das Hadoop-Cluster, die Selfbalancing-Komponente, das entwickelte S#-Modell sowie der Treiber zur Kommunikation zwischen Modell und Cluster. Konkret sind dies folgende Anforderungen an das Testsystem:

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 seine Funktionsweise auf einem oder mehreren Hosts ausgeführt werden
8. Es können mehrere Benchmark-Anwendungen gleichzeitig gestartet und ausgeführt werden
9. Tests und Testfälle können zeitlich unabhängig und mehrmals ausgeführt werden

Die funktionalen Anforderungen dienen zudem ebenfalls als Anforderungen an das Testsystem und erweitern somit die hier genannten Anforderungen.

Eine Besonderheit bildet zudem die fünfte Anforderung, wonach erkannt werden muss, dass im Cluster keine weitere Rekonfiguration möglich ist. Wird diese Anforderung verletzt soll der ausgeführte Test abgebrochen werden, während bei den anderen, auch

den funktionalen, Anforderungen dies nur durch das Oracle vermerkt werden soll, die Ausführung aber nicht weiter durch das Oracle beeinträchtigt werden soll.

3.3. Planung der Tests und der Evaluation

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

3.3.1. Behauptungen und Variablen

Als Basis für die Behauptungen und Variablen dienten die in Abschnitt 3.2 definierten Anforderungen an das Cluster und das Testsystem. Sie gehen damit auch einher mit den Behauptungen, welche für die Evaluation aufgestellt werden. Basierend auf den Anforderungen wurden daher folgende unabhängigen Variablen zur Evaluation ermittelt:

- Anzahl der Hosts und Nodes
- Anzahl der Clients
- Anzahl der Testfälle pro Test
- *Seed* für Zufallsgeneratoren
- Generelle Wahrscheinlichkeit zur Aktivierung und Deaktivierung der Komponentenfehler

Basierend auf den unabhängigen Variablen wurden u. A. folgende abhängigen Variablen ermittelt:

- Anzahl der tatsächlich ausgeführten Testfälle
- Aktivierten und deaktivierte Komponentenfehler
- Anzahl ausgeführter Anwendungen
- Anzahl und Gründe für evtl. nicht vollständig ausgeführte Anwendungen

Während die unabhängigen Variablen dazu genutzt werden, um Testkonfigurationen zu definieren (vgl. Abschnitt 3.3.2), dienen die abhängigen Variablen als wichtige Kennzahlen im Rahmen der Evaluation in Kapitel 7.

3.3.2. Generierung der Testkonfigurationen

Um nun anhand der in Abschnitt 3.3.1 definierten Variablen die in Abschnitt 3.2 definierten Anforderungen zu prüfen, sind mehrere Tests nötig. Als Basis zur Definition einer Testkonfiguration dienen die in Abschnitt 3.3.1 definierten unabhängigen Variablen, die durch weitere Angaben ergänzt werden:

- Anzahl genutzter Hosts
- Basisanzahl der Nodes
- Anzahl simulierter Clients
- Verwendeter Seed für Zufallsgeneratoren
- Anzahl auszuführender Testfälle
- Mindestdauer für einen Testfall
- Nutzung einer oder mehreren Mutationen
- Generelle Wahrscheinlichkeit zur Aktivierung von Komponentenfehlern
- Generelle Wahrscheinlichkeit zur Deaktivierung von Komponentenfehlern
- Verwendung von vorab generierten Eingabedaten

Die Auswahl der ausgeführten Anwendungen erfolgt während der Testausführung anhand des Seeds der Testkonfiguration. Zwar werden durch das Transitionssystem wahrscheinlichkeitsbasierend zufällig Anwendungen ausgewählt, jedoch kann dies durch den Charakter des im .NET-Framework vorhandenen Zufallsgenerators gesteuert werden. Der in .NET implementierte Zufallsgenerator ist nämlich kein *echter*, sondern ein Pseudo-Zufallsgenerator, der mithilfe von mathematischen Formeln und anhand eines Seeds Zufallszahlen berechnet. Zwar wird standardmäßig ein zeitbasierter Seed zur Initialisierung des Zufallsgenerators genutzt, durch die Angabe eines spezifischen Seeds kann jedoch die Wiederausführbarkeit der Tests sichergestellt werden.

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

Auch die Aktivierung und Deaktivierung von Komponentenfehlern selbst soll während der Ausführung eines Testfalls festgelegt werden, wodurch die Wahrscheinlichkeit für deren Aktivierung bzw. Deaktivierung einen maßgeblichen Einfluss besitzt. Da Anwendungen, die Eingabedaten für andere Anwendungen generieren, u. U. nicht erfolgreich beendet werden können, kann eine Testkonfiguration mit vorab generierten und im HDFS gespeicherten Daten durchgeführt werden. Dadurch wird es ermöglicht, dass in solchen Tests spätere Anwendungen nicht vom Erfolg der zuvor ausgeführten Anwendungen zur Generierung der Eingabedaten abhängig sind. Da einige Tests zudem als Mutationstests durchgeführt werden sollen, muss für eine Testkonfiguration dies entsprechend definiert werden.

Die Auswahl der konkreten Testkonfigurationen in Abschnitt 6.3 erfolgte im Rahmen der in Abschnitt 6.4 beschriebenen Implementierung der Tests.

3.3.3. 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 eine Log-Datei ausgegeben und 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 erfolgreichen Testfall durchgeführt, damit das Verhalten des Clusters berücksichtigt werden kann. Bei nicht erfolgreichen Testfällen wird die Simulation dagegen beendet. Für solche Fälle werden nach Abschluss der Simulation erneut die Daten des Clusters ausgelesen und ausgegeben. Es können hierbei alle Daten ausgegeben werden, welche erkannt werden können, mindestens jedoch:

- 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
- Anzahl der derzeit ausgeführten Container

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. Entwicklung des Testmodells

Um die in Kapitel 3 beschriebene Fallstudie durchführen zu können, wurde zunächst das Testmodell mithilfe des S#-Frameworks entwickelt. Das dabei entwickelte Modell bildet vereinfacht die für die Fallstudie relevanten YARN-Komponenten ab und besteht aus den drei im Folgenden beschriebenen, architektonischen Schichten.

4.1. Grundlegende Architektur des Testmodells

Um Hadoop mit der Selfbalancing-Komponente mit den in Abschnitt 3.2 beschriebenen Anforderungen prüfen zu können, wird mithilfe des S#-Frameworks ein vereinfachtes Modell der relevanten YARN-Komponenten entwickelt. Dieses YARN-Modell wird mithilfe des Treibers mit dem realen Cluster verbunden, was durch hierfür entwickelte Scripte gesteuert wird. Daraus resultiert folgende Drei-Schichten-Architektur für das gesamte Testmodell:

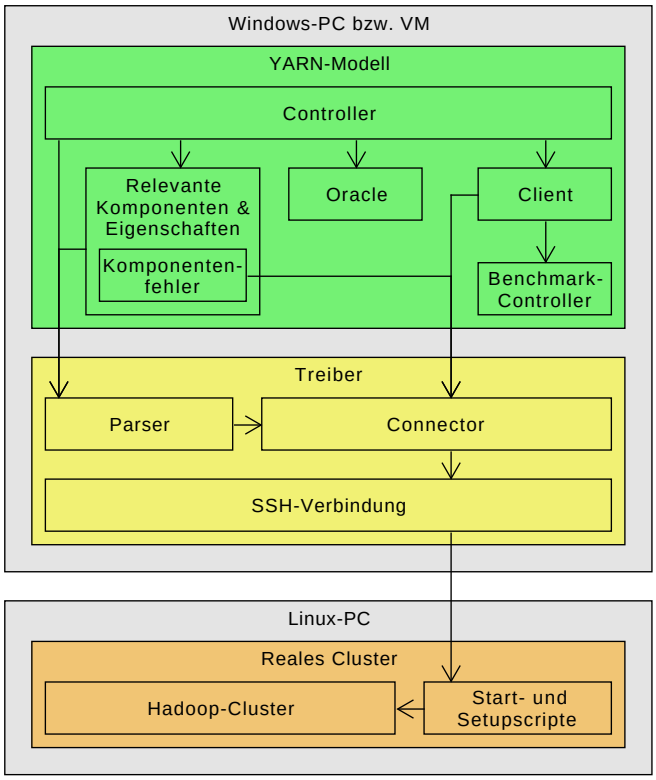

Abbildung 4.1.: Grundlegende Architektur des Testsystems

Das YARN-Modell stellt die oberste Schicht des Testmodells dar. Es bildet das Kernstück dieser Fallstudie, da dieses Modell mit den hierin abgebildeten, für diese

Fallstudie relevanten YARN-Komponenten und implementierten Komponentenfehlern, dem Controller und dem Oracle direkt im Rahmen des modellbasierten Testens mit S# interagiert. Folgende Komponenten sind im YARN-Modell enthalten:

Controller
: Steuert den Ablauf einer Testausführung und das Zusammenspiel zwischen den Komponenten des YARN-Modells.

Relevante YARN-Komponenten und Eigenschaften
: Bilden die grundlegende Architektur von Hadoop YARN ab. Implementiert wurden in dieser Fallstudie die Nodes, Anwendungen, Attempts und Container mit den jeweils relevanten Eigenschaften zur Durchführung der Fallstudie.

Komponentenfehler der YARN-Komponenten
: Bilden die bei den Tests zu injizierenden Komponentenfehler der jeweiligen YARN-Komponenten.

Oracle
: Validiert die in Form von Constraints in den jeweiligen YARN-Komponenten implementierten Anforderungen.

Client
: Dient zum starten und beenden von Benchmarks im Cluster.

Benchmark-Controller
: Enthält das Transitionssystem zur Auswahl der Benchmarks und steuert diese.

Die Verbindung zwischen dem YARN-Modell und dem realem Cluster bildet der Treiber. Er 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 Parser wird hierbei nur zur Durchführung des Monitoring benötigt und nutzt wiederum den Connector zum abrufen der Daten. Andere Befehle und Zugriffe auf das reale Cluster, wie z. B. das Injizieren von Komponentenfehlern, werden direkt mithilfe des Connectors durchgeführt.

Die Implementierung des YARN-Modells wird in Abschnitt 4.2 beschrieben, die Implementierung des Treibers in Abschnitt 4.3. Die Umsetzung des realen Clusters wird in Abschnitt 4.4 beschrieben.

4.2. Implementierung des YARN-Modells

Das implementierte YARN-Modell besteht, wie bereits in Abschnitt 4.1 gezeigt, aus fünf Komponenten und den Komponentenfehlern der hier relevanten YARN-Komponenten. Die vier implementierten YARN-Komponenten sind die Anwendungen, ihre Attempts und Container, sowie die Nodes. Zudem wurde eine Klasse implementiert, die zur Repräsentation des RM dient, und als Controller im Rahmen des Testens mit S# dient. Einen Überblick über den Aufbau des implementierten YARN-Modells gibt folgendes Klassendiagramm:

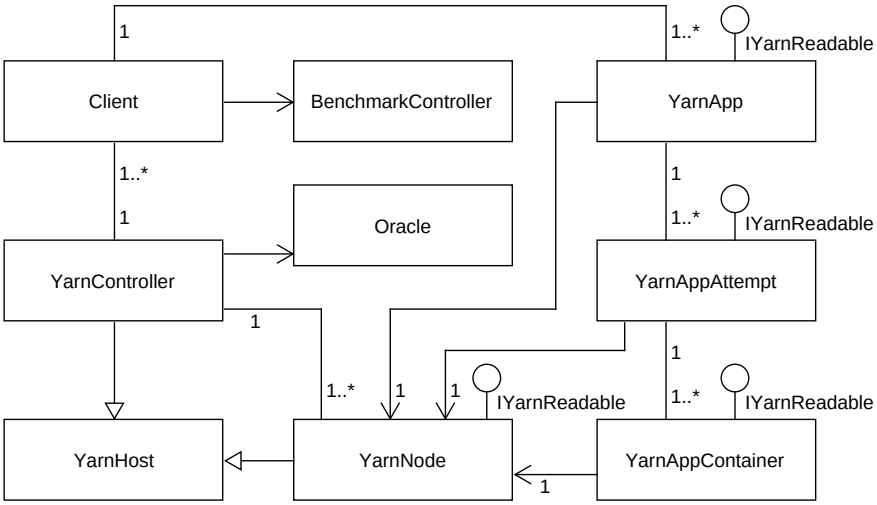

Abbildung 4.2.: Grundlegender Aufbau des YARN-Modells. Assoziationen und weitere Verbindungen zum Treiber und S# sind hier aus Gründen der Übersichtlichkeit nicht dargestellt.

Im Klassendiagramm wird zudem ersichtlich, dass jedem Client mehrere Anwendungen in Form der Klasse `YarnApp` zugeordnet sind, jeder Anwendung mehrere Attempts (`YarnAppAttempt`) und jedem Attempt mehrere Anwendungs-Container (`YarnAppContainer`). Der Client stellt somit auch die den relevanten YARN-Komponenten übergeordnete Komponente dar.

Zunächst wird im Folgenden zunächst die dem Modell übergeordnete zentrale Klasse `Model` erläutert, danach die relevanten YARN-Komponenten mit ihren Komponentenfehlern, anschließend die anderen Komponenten des YARN-Modells.

4.2.1. Die Klassen Model und ModelSettings

Die Klasse `Model` stellt die zentrale Schnittstelle des Testsystems mit S# dar, während die Klasse `ModelSettings` alle generellen, variablen Einstellungen und Konstanten des YARN-Modells bereitstellt. Da die `Model`-Klasse auch das gesamte Modell repräsentiert, wird sie zur Verwaltung des von S# auszuführenden Modells genutzt. Hierfür werden alle Komponenten wie Clients oder Anwendungen zusätzlich zur Struktur innerhalb des Modells auch direkt in entsprechenden Eigenschaften dieser Klasse gespeichert, welche zur Interaktion mit S# entsprechend markiert sind (vgl. Abschnitt 2.1.1).

Daneben dient die Klasse auch zur Initialisierung des gesamten Modells. Hierbei werden zunächst der Controller, der Treiber in Form der benötigten Parser und Connectoren, sowie die Nodes des Clusters initialisiert. Anschließend wird für jeden Client eine gewisse Anzahl an `YarnApp`-Instanzen zum Speichern der Daten von Anwendungen, für jede Anwendung eine gewisse Anzahl an `YarnAppAttempt`-Instanzen für Attempts, sowie für jeden Attempt eine gewisse Anzahl an `YarnAppContainer` für die Daten der Anwendungs-Container initialisiert. Die genaue Anzahl der erzeugten Instanzen kann hierbei für jede YARN-Komponente und auch für Clients nach Bedarf angepasst werden, wodurch auch einzelne Komponenten wie z. B. in der Fallstudie die Anwendungs-Container, deaktiviert werden können.

Alle initialisierten YARN-Komponenten werden durch die `Model`-Klasse auch innerhalb des Modells zur Verfügung gestellt, sofern Komponenten diese benötigen. Darunter zählen auch die bei der Ausführung des Modells benötigten Parser und Connectoren. Aufgrund der Einschränkungen von S# im Umgang mit dem Treiber (vgl. Abschnitt 4.3.1), sowie zur einfacheren Bereitstellung von benötigten Komponenten und Funktionen im Modell, ist die `Model`-Klasse als Singleton realisiert.

Die `ModelSettings`-Klasse dient zum Speichern der variablen Einstellungen sowie zum Bereitstellen von generellen, Modell- oder Testsystemweiten Konstanten. Gespeichert und bereitgestellt werden hier Daten wie z. B. die Zugangsdaten der SSH-Verbindungen oder den genutzten `HostMode` (vgl. Abschnitte 4.3.3 und 4.4) im YARN-Modell und dem Treiber transparent zu machen.

4.2.2. Relevante YARN-Komponenten

Die vier implementierten, relevanten YARN-Komponenten sind die Anwendungen, ihre Attempts und Container sowie die Nodes des Clusters. Obwohl die die Anwendungs-Container in dieser Fallstudie nicht benötigt werden, waren sie für die in [31] beschriebene Fallstudie notwendig, welche ebenfalls mit dem hier beschriebenen Modell durchgeführt wurden.

Übersicht der implementierten Komponenten

Eine Übersicht über die Implementierung der YARN-Komponenten gibt folgendes Klassendiagramm:

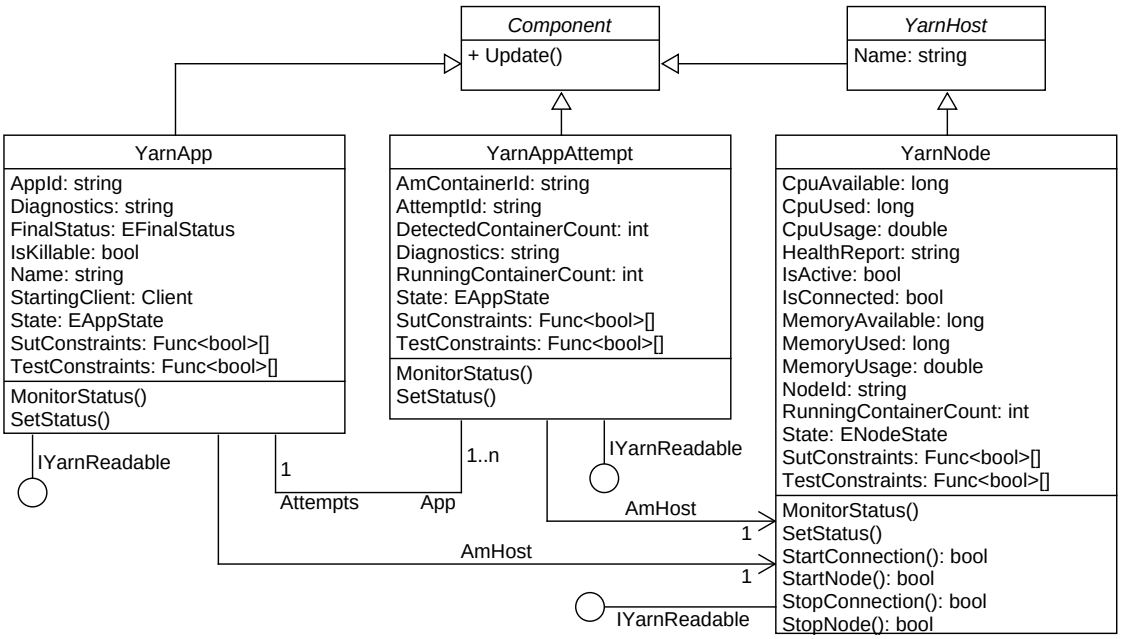

Abbildung 4.3.: Für die Fallstudie relevante, implementierte YARN-Komponenten, hier dargestellt mit den wichtigsten Eigenschaften und Methoden. Dies sind alle für die spätere Durchführung und zur Ausgabe des Zustandes (vgl. Abschnitt 3.3.3) wichtigen Eigenschaften und Methoden. Aus Gründen der Übersichtlichkeit sind die implementierten Komponentenfehler, einige der `IYarnReadable` bereitgestellten, relevanten Eigenschaften und Methoden, sowie die Klasse `YarnAppContainer` nicht aufgeführt.

Die Eigenschaft `AmHost` speichert den ausführenden Node des AppMstr der Anwendung bzw. des Attempts, die Eigenschaft `IsKillable` gibt an, ob eine Anwendung derzeit ausgeführt ist und somit vorzeitig abgebrochen werden kann. Die Eigenschaften `Diagnostics` bzw. `HealthReport` stellen von Hadoop zur Verfügung gestellte weitere Informationen bei Fehlern dar. Die `State`-Eigenschaften bzw. `FinalState` speichern die von Hadoop angegebenen Zustände der YARN-Komponenten. Hierfür wurden entsprechende Auflistungen basierend auf den in [32] angegebenen möglichen Werten für die jeweiligen Zustände im Modell implementiert.

Da für diese Fallstudie die Daten der Container selbst nicht relevant sind, wird im hier verwendeten Modell nur gespeichert, wie viele Anwendungs-Container beim Monitoring im aktuellen Testfall (`RunningContainerCount`) bzw. für alle bisher ausgeführten Testfälle kumuliert (`DetectedContainerCount`) erkannt wurden. Für die Tests in [31] werden die Daten der Container genauso wie die der anderen YARN-Komponenten gespeichert. Hierfür enthalten die Klassen `YarnAppAttempt` und `YarnAppContainer` entsprechende Eigenschaften zur jeweiligen Zuordnung analog zu denen zwischen `YarnApp`

und `YarnAppAttempt`. Analoge Zuordnungen bestehen auch zu den Clients bzw. den Anwendungs-Containern, sofern benötigtt. Die Eigenschaft `AmContainerId` speichert zudem die Container-ID des AppMstr-Containers.

Die Eigenschaften mit dem Präfix `CPU` bzw. `Memory` in `YarnNode` dienen zur Speicherung der derzeitigen Auslastung der CPU-Kerne bzw. des Speichers eines Nodes. Benötigt werden diese Werte zur in Abschnitt 6.1.3 beschriebenen Berechnung, ob ein Komponentenfehler aktiviert oder deaktiviert wird.

Die Eigenschaften `SutConstraints` und `TestConstraints` sowie die Methoden `MonitorStatus()` und `SetStatus()` werden von `IYarnReadable` bereitgestellt und speichern die Constraints bzw. dienen zur Durchführung des Monitorings der implementierten Komponenten. Die Start- und Stop-Methoden sowie die Eigenschaften `IsActive` und `IsConnected` in `YarnNode` dienen zur Identifikation und Injizierung bzw. Reparieren der Komponentenfehlern. Diese Eigenschaften und Methoden werden daher auf den folgenden Seiten näher beschrieben.

Neben den dargestellten, relevanten Eigenschaften und Methoden wurden zahlreiche weitere implementiert, welche einerseits zur Vollständigkeit der implementierten Komponenten für die Tests in [31], andererseits auch zur Ausführung mithilfe des S#-Frameworks benötigt werden. Letztere sind z. B. zur Speicherung von Strings notwendig, welche aufgrund der Einschränkungen von S# (vgl. Abschnitt 2.1) nicht jederzeit frei genutzt werden können. Strings sind im YARN-Modell daher zur Speicherung immer als `char`-Arrays implementiert, werden jedoch zur einfacheren Nutzung im Modell in Strings konvertiert:

```
1 public char[] AppIdActual { get; }
2
3 [NonSerializable]
4 public string AppId
5 {
6   get { return ModelUtilities.GetCharArrayAsString(AppIdActual); }
7   set { ModelUtilities.SetCharArrayOnString(AppIdActual, value); }
8 }
```

Listing 4.1: Implementierung der Eigenschaft `AppId`. Die beiden Methoden `GetCharArrayAsString` und `SetCharArrayOnString` führen die Konvertierung in den `char`-Array bzw. des `char`-Arrays in einen String durch.

Die `Update()`-Methoden starten bei allen vier implementierten YARN-Komponenten die jeweiligen Monitoring-Funktionen, bei `YarnNode` werden zudem `StartNode()` und `StartConnection()` ausgeführt um mögliche zuvor injizierte Komponentenfehler im realen Cluster zu reparieren.

Da bei der Ausführung des Modells durch S# keine neuen Instanzen erzeugt werden können (vgl. Abschnitt 2.1), dienen die jeweiligen IDs der Komponenten auch als Indikator, ob die jeweilige Komponenten-Instanz derzeit benötigt wird und somit Daten

gespeichert werden sollen. Daher wird das Monitoring auch nur dann ausgeführt, wenn der Instanz eine nicht leere ID z. B. in Form der `AppId` zugewiesen wurde.

Implementierung der Komponentenfehler

Die im YARN-Modell implementierten Komponentenfehler wurden direkt in den entsprechenden YARN-Komponenten implementiert. Implementiert wurden hierbei mit `NodeDeadFault` und `NodeConnectionErrorFault` zwei jeweils als `TransientFault` definierte Komponentenfehler für die durch `YarnNode` repräsentierten Nodes. Während durch `NodeDeadFault` der komplette Node beendet wird, trennt `NodeConnectionErrorFault` nur die Verbindung des Nodes zum Cluster. Die dazugehörigen Effekt-Klassen der Komponentenfehler sind jeweils als innere Klassen in `YarnNode` implementiert.

Die Injizierung und das Reparieren der beiden Fehler geschieht mithilfe der vier Stop- bzw. Start-Methoden wie z. B. `StopNode()`, die bereits im Klassendiagramm in Abb. 4.3 zu sehen sind. Wenn ein Komponentenfehler durch das Framework aktiviert wird, wird durch die Effekt-Klasse die jeweilige Stop-Methode aufgerufen und so der Fehler injiziert:

```
1  public class YarnNode : YarnHost, IYarnReadable
2  {
3    [NodeFault]
4    public readonly Fault NodeDeadFault = new TransientFault();
5    public IHadoopConnector FaultConnector { get; set; }
6
7    public bool StopNode(bool retry = true)
8    {
9      if(IsActive)
10     {
11       var isStopped = FaultConnector.StopNode(Name);
12       if(isStopped)
13         IsActive = false;
14       else if(retry)
15         StopNode(false); // try again once
16     }
17     return !IsActive;
18   }
19
20   [FaultEffect(Fault = nameof(NodeDeadFault))]
21   public class NodeDeadEffect : YarnNode
22   {
23     public override void Update()
24     {
25       StopNode();
26     }
27   }
28 }
```

```
29
30 public class CmdConnector : IHadoopConnector
31 {
32   private SshConnection Faulting { get; }
33
34   public bool StopNode(string nodeName)
35   {
36     var id = DriverUtilities.ParseInt(nodeName);
37     Faulting.Run($"{Model.HadoopSetupScript} hadoop stop {id}",
         IsConsoleOut);
38     return !CheckNodeRunning(id);
39   }
40 }
```

Listing 4.2: Injizierung des Komponentenfehlers `NodeDeadFault` (gekürzt). Sollte der Node nicht beendet werden, wird die Injizierung einmalig erneut versucht. `CmdConnector.Faulting` stellt die zur Injizierung verwendete SSH-Verbindung dar.

Das Reparieren von Komponentenfehlern geschieht analog hierzu. Hierfür werden in der `Update()`-Methode die beiden Start-Methoden aufgerufen, die einen Fehlerhaften Node reparieren. Eine Besonderheit bildet hierbei die Reparatur des Komponentenfehlers `NodeConnectionErrorFault`, bei der der Node komplett neu gestartet wird, da es sonst passieren kann, dass der wieder ans Netzwerk angebundene Node sich nicht mit dem RM verbindet.

Ob ein Komponentenfehler in einem Testfall aktiviert wird, entscheidet sich anhand der Auslastung des Nodes. Hierfür sind beide Komponentenfehler mit dem Attribut `NodeFault` versehen, mit dessen Hilfe entschieden wird, ob ein Komponentenfehler aktiviert oder deaktiviert und somit injiziert bzw. repariert wird. Der Entscheidungsprozess hierfür ist in Abschnitt 6.1.3 beschrieben.

Da beide Fehler zudem auch gleichzeitig aktiviert werden können, wurde `NodeDeadFault` mithilfe entsprechender S#-Funktionen eine höhere Priorität vergeben, wodurch dieser Vorrang vor dem anderen Komponentenfehler erhält. Dadurch wird in solchen Fällen der Node beendet und nicht zunächst noch vom Netzwerk getrennt.

Zur einfachen Identifikation der aktiven Komponentenfehler dienen die beiden Eigenschaften `IsActive` und `IsConnected`. In Listing 4.2 wird bereits die Auswirkung der beiden Eigenschaft gezeigt, indem verhindert wird, dass ein möglicherweise bereits injizierter bzw. reparierter Komponentenfehler erneut injiziert bzw. repariert wird. Sie dienen aber auch zur Validierung der Constraints, bei denen mithilfe der beiden Eigenschaften geprüft wird, ob ein Node korrekt als aktiv bzw. defekt erkannt wurde.

Interface IYarnReadable und Monitoring

Das Interface `IYarnReadable` ist das zentrale Erkennungsmerkmal der im Modell abgebildeten und implementierten YARN-Komponenten. Es dient zum einen zur Identifikati-

on aller implementierten YARN-Komponenten, andererseits stellt es auch Eigenschaften und Methoden bereit, welche einerseits dem Testen in S# dienen, primär aber dem Ermitteln der Daten aus dem realen Cluster:

- `GetId()`
- `StatusAsString()`
- `Parser`
- `IsSelfMonitoring`
- `MonitorStatus()`
- `SetStatus()`
- `PreviousParsedComponent`
- `CurrentParsedComponent`
- `SutConstraints`
- `TestConstraints`

Die beiden erstgenannten Methoden dienen primär zu Debugging-Zwecken und zur Rückgabe der ID beim Zugriff auf die jeweiligen Komponenten mithilfe das Interfaces bzw. der Werte aller Eigenschaften der Komponente als ein String.

Die nachfolgenden vier Eigenschaften und Methoden dienen zum Monitoring der entsprechenden YARN-Komponenten. Während die Eigenschaft `Parser` den zu verwendenden Parser (vgl. Abschnitt 4.3.2) speichert, dient die Eigenschaft `IsSelfMonitoring` zur Unterscheidung, ob die Daten einer Komponente von dieser selbst ermittelt werden oder dies die übergeordnete Komponente durchführt. Diese Unterscheidung ist nötig, da YARN zwei unterschiedliche Möglichkeiten zur Ermittlung der Daten bietet, die Rückgabe der Daten durch die Kommandozeile oder mithilfe der REST-API. Ausführlichere Informationen zu den beiden Varianten sind in Abschnitt 4.3 zu finden. Bei der Nutzung der Kommandozeile zur Ermittlung der Daten eignet sich daher die Selbstermittlung der Daten besser, während bei der Nutzung der Rest-API die Ermittlung der Daten durch die übergeordnete Komponente geeigneter ist. Aus diesem Grund ist auch die Methode `SetStatus()` definiert, da hier unabhängig von der Datenermittlung der aktuelle Status der Komponente abgespeichert werden kann. Die Durchführung des Monitoring findet in beiden Fällen jedoch mithilfe der Methode `MonitorStatus()` statt:

```
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      // search attempt with this id or an free attempt
14      attempt.IsSelfMonitoring = IsSelfMonitoring;
15      if(IsSelfMonitoring)
16        attempt.AttemptId = parsed.AttemptId;
17      else
18      {
19        attempt.SetStatus(parsed);
20        attempt.MonitorStatus();
21      }
22    }
23  }
```

Listing 4.3: Implementierung der Methode `MonitorStatus()` in der Klasse `YarnApp` (gekürzt). Das Monitoring der anderen Komponenten erfolgt analog hierzu.

Beim Monitoring der untergeordneten Komponenten wird zunächst immer geprüft, ob bereits eine Instanz mit der ID der Komponente vorhanden ist. Wenn dies der Fall ist, werden die vom realen Cluster ermittelten Daten weiterhin in der bereits bestehenden Instanz gespeichert, ansonsten wird eine leere Instanz genutzt um die Daten der Subkomponente zu speichern. Wenn keine freie Instanz mehr nötig ist, wird eine entsprechende `OutOfMemoryException` ausgelöst, damit die Ausführung des Modells abgebrochen werden kann.

Die vier restlichen Eigenschaften und Methoden des Interfaces dienen zur Auswertung der Komponente durch S#. Die beiden Eigenschaften `SutConstraints` und `TestConstraints` dienen zur Implementierung der in Abschnitt 3.2 definierten Anforderungen in Form von Constraints.

Constraints der YARN-Komponenten

Einige der in Abschnitt 3.2 definierten Anforderungen an das SuT und gesamte Testsystem sind auch für die YARN-Komponenten relevant. Die relevanten Bestandteile der Anforderungen für die jeweiligen Komponenten sind mithilfe der beiden in `IYarnReadable` definierten Eigenschaften `SutConstraints` und `TestConstraints` implementiert. Realisiert sind die beiden Eigenschaften für die Constraints jeweils als `Func<bool>[]`:

```
1  public Func<bool>[] SutConstraints => new Func<bool>[]
2  {
3    // task will be completed if not canceled
4    () =>
5    {
6      if(FinalStatus != EFinalStatus.FAILED)
7        return true;
8      if(!String.IsNullOrWhiteSpace(Name) &&
```

```
        Name.ToLower().Contains("fail job"))
      return true;
    return false;
  },
  // configuration will be updated
};

public Func<bool>[] TestConstraints => new Func<bool>[]
{
  // current state is detected and saved
  () =>
  {
    var prev = PreviousParsedComponent as IApplicationResult;
    var curr = CurrentParsedComponent as IApplicationResult;
    // compare prev, curr and this and return the result
    // or otherwise
    return false;
  },
};
```

Listing 4.4: Definition der Constraints in `YarnApp` (gekürzt)

Die Constraints werden im Anschluss an das Monitoring vom Oracle validiert, was in Abschnitt 4.2.5 beschrieben wird. Die Constraints sind so definiert, dass im Falle einer erfolgreichen Validierung `true` zurückgegeben wird, in allen anderen Fällen die Rückgabe `false` dagegen eine nicht erfolgreiche Validierung anzeigt.

Wenn die ID der Komponenten-Instanzen leer ist, und die jeweilige Instanz somit derzeit nicht im Modell zum Speichern der Daten des realen Clusters benötigt wird, werden die Constraints immer erfolgreich validiert.

4.2.3. Implementierung des Clients

Der Client im YARN-Modell simuliert einen Client und dient somit zum Starten der Benchmarks in dieser Fallstudie. Die Auswahl des zu startenden Benchmarks erfolgt durch den Benchmark Controller, welcher in Abschnitt 5.3 erläutert wird. Jeder Client besitzt hierzu einen eigenen Benchmark Controller, womit die auszuführenden Benchmarks für jeden Client unabhängig von anderen Clients ausgewählt werden. Ein Client kann jedoch nur eine Anwendung gleichzeitig starten und muss eine zuvor gestartete Anwendung beenden, bevor er eine neue starten kann. Dies wird jedoch nur durchgeführt, wenn sich der auszuführende Benchmark geändert hat:

```
public void UpdateBenchmark()
{
  var benchChanged = BenchController.ChangeBenchmark();

  if(benchChanged)
```

```
6   {
7     StopCurrentBenchmark();
8     StartBenchmark(BenchController.CurrentBenchmark);
9   }
10 }
```

Listing 4.5: Auswahl und Start des nachfolgenden Benchmarks (gekürzt). Der Benchmark Controller und seine Methode `BenchmarkController.ChangeBenchmark()` wird in Abschnitt 5.3 erläutert.

Da die auf dem Cluster ausgeführten Anwendungen u. U. nicht gestartet werden, wenn im HDFS das Ausgabeverzeichnis bereits vorhanden ist, muss dieses beim Starten eines Benchmarks ebenfalls zunächst gelöscht werden. Anschließend kann die Anwendung gestartet werden. Die weitere Ausführung des Clients wird dabei solange unterbrochen, bis der gestarteten Anwendung eine *Application ID* zugewiesen wurde, die wiederum vom Client abgespeichert wird. Hierbei wird auch eine noch leere `YarnApp`-Instanz ermittelt, um diese analog zu den anderen YARN-Komponenten zum Speichern der Daten zu nutzen. Der Unterschied hierbei ist jedoch, dass immer eine neue Instanz genutzt wird, da eine neue Anwendung automatisch immer eine ID erhält, welche im Cluster noch nicht existiert. Wenn keine leere `YarnApp`-Instanz mehr verfügbar ist, wird analog zum Monitoring ebenfalls eine `OutOfMemoryException` ausgelöst.

4.2.4. Implementierung des Controllers

Der Controller repräsentiert zum einen den RM des Hadoop-Clusters, weshalb er auch von `YarnHost` erbt, genauso wie die Klasse der Nodes des YARN-Modells. Seine Hauptaufgabe besteht jedoch darin, als Controller zum Testen mit S# zu dienen (vgl. Abschnitt 2.1.1).

Der Controller steuert einen Großteil der Ausführung eines einzelnen Testfalls und ist die einzige Komponente des YARN-Modells, welche direkt mit dem Oracle interagiert:

```
1  public override void Update()
2  {
3    MonitorMarp();
4
5    foreach(var client in ConnectedClients)
6      client.UpdateBenchmark();
7
8    // optional, to allocate at least the AM container
9    ModelUtilities.Sleep(5);
10
11   MonitorAll();
12
13   Oracle.ValidateConstraints(EConstraintType.Sut);
14   Oracle.IsReconfPossible();
```

```
15    Oracle.ValidateConstraints(EConstraintType.Test);
16  }
```

Listing 4.6: `Update()`-Methode des Controllers (gekürzt). Eine ausführliche Beschreibung des Ablaufs der Ausführung eines Testfalls findet sich in Abschnitt 6.1.3.

Nach einem ersten Monitoring des MARP-Wertes (vgl.
MARP-Monitoring!
) wird durch den Controller sichergestellt, dass jeder simulierte Client eine Anwendung startet, sofern vom Benchmark Controller hierbei eine neue Anwendung ausgewählt wird (vgl. Abschnitte 4.2.3 und 5.3).

Vor dem Monitoring wird zunächst fünf Sekunden gewartet, damit die gestarteten Anwendungen auf dem Cluster die benötigten Ressourcen erhalten können, wodurch die Auslastung des Clusters besser ermittelt werden kann. Zum Monitoring nutzt der Controller die von `IYarnReadable` bereitgestellte Eigenschaft um das Selbstmonitoring der einzelnen YARN-Komponenten zu deaktivieren und führt dabei das Monitoring der Nodes und Anwendungen aus. Er startet zudem bei jeder Anwendung auch das Monitoring der jeweiligen Attempts bzw. dadurch auch das der Anwendungs-Container, sofern benötigt. Abschließend startet der Controller die Validierung der Constraints durch das Oracle sowie die Prüfung, ob eine weitere Rekonfiguration des Clusters möglich ist.

Für den Controller selbst sind ebenfalls einige der in Abschnitt 3.2 definierten Anforderungen relevant. Die hierbei relevanten Anforderungen sind ebenfalls direkt im Controller implementiert und werden durch das Oracle validiert.

4.2.5. Implementierung des Oracles

Das Oracle dient zur Validierung der in Abschnitt 3.2 definierten Anforderungen automatisiert durch das Testsystem. Hierzu validiert es die Constraints aller YARN-Komponenten und des Controllers, jeweils getrennt nach Constraints für das SuT und das gesamte Testsystem, und prüft, ob eine Rekonfiguration des Clusters möglich ist. Da die beiden von `IYarnReadable` bereitgestellten Eigenschaften zum Speichern der Constraints vom Typ `Func<bool>[]` sind, können so jeweils alle implementierten Constraints nacheinander validiert werden:

```
1  public static bool ValidateConstraints(string componentId,
2     Func<bool>[] constraints, EConstraintType constraintType)
3  {
4    var isCompontenValid = true;
5    for(var i = 0; i < constraints.Length; i++)
6    {
7      var constraint = constraints[i];
8      bool isValid;
9      try
```

```
10      {
11        isValid = constraint();
12      }
13      catch
14      {
15        isValid = false;
16      }
17
18      CountCheck(constraintType, isValid);
19      if(!isValid)
20      {
21        Logger.Error($"YARN component not valid: " +
22           "Constraint {i} in {componentId}");
23        if(isCompontenValid)
24          isCompontenValid = false;
25      }
26    }
27
28    return isCompontenValid;
29  }
```

Listing 4.7: Validieren der Constraints durch das Oracle. Die zu validierenden Constraints werden im Parameter `constraints` übergeben, der Parameter `constraintType` dient zu statistischen Zwecken in `CountCheck()`.

Das Oracle prüft auch ob eine weitere Rekonfiguration möglich ist. Sind dagegen alle Nodes im Cluster defekt bzw. wurde dies beim Monitoring des realen Clusters erkannt, wird die Ausführung gemäß Abschnitt 3.2.2 abgebrochen. Dazu wird eine `Exception` ausgelöst, welche zum Abbruch der Ausführung dient:

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

Listing 4.8: Prüfung nach der Möglichkeit weiterer Rekonfigurationen

Die bereits in Listing 4.7 genutzte Methode `CountCheck()` dient zu statistischen Zwecken. Hierbei wird getrennt nach Constraint-Typ (für das SuT oder für das Testsystem) gezählt, wie viele Constraints insgesamt validiert wurden und wie viele verletzt

wurden. Die jeweiligen Werte können beim Abschluss einer Ausführung des Modells entsprechend ausgegeben werden.

4.3. Entwicklung des Treibers

Multihost-Mode irgendwo erklären

In Abschnitt 4.1 wurde bereits aufgezeigt, dass der Treiber zur Verbindung des YARN-Modells mit dem realen Cluster aus den drei Komponenten Parser, Connector und der SSH-Verbindung selbst besteht. Der Treiber ist im YARN-Modell mithilfe verschiedener Interfaces zur Nutzung des Parsers und Connectors eingebunden. Da YARN mithilfe von Befehlen für die Kommandozeile und einer REST-API zwei unterschiedliche Schnittstellen zum Auslesen der Daten der YARN-Komponenten für das Monitoring bereitstellt, wurden jeweils zwei entsprechende Parser und Connectoren hierfür entwickelt. Andere Befehle wie z. B. HDFS-Befehle können ebenfalls mithilfe des entwickelten Kommadozeilen-Connectors ausgeführt werden, da Connectoren mithilfe von SSH-Verbindungen mit den Cluster-Hosts verbunden sind.

4.3.1. Grundlegender Aufbau und Integration im YARN-Modell

Zur Integration des Treibers im YARN-Modell stellt dieser mehrere Interfaces bereit. Dadurch sind einerseits der Treiber und das YARN-Modell strikt getrennt, andererseits wird es dadurch auch ermöglicht, in Zukunft andere Möglichkeiten als die hier Entwickelten zur Interaktion mit dem realen Cluster zu entwickeln und zu nutzen.

Zur Interaktion des YARN-Modells mit dem Treiber werden dem Modell folgende Interfaces zur Verfügung gestellt:

- `IHadoopParser` für Parser
- `IHadoopConnector` für Connectoren
- Von `IParsedComponent` abgeleitete Interfaces für geparste YARN-Komponenten:
  - `IApplicationResult` für Anwendungen
  - `IAppAttemptResult` für Attempts
  - `IContainerResult` für Anwendungs-Container
  - `INodeResult` für Nodes

Das Monitoring der Daten des realen Clusters wird mithilfe des Parsers durchgeführt. Das Interface `IHadoopParser` stellt hierfür entsprechende Parsing-Methoden für die vier implementierten YARN-Komponenten sowie der Übersichtslisten aller einer YARN-Komponente untergeordneten Subkomponenten. Zudem stellt das Parser-Interface eine Methode zum Auslesen des aktuellen MARP-Wertes des Schedulers bereit. Beim Monitoring werden immer die entsprechenden von `IParsedComponent` abgeleiteten

Interfaces zur Rückgabe der ermittelten Daten genutzt. Hierfür stellen diese Interfaces entsprechende Eigenschaften bereit, um alle mithilfe der Kommandozeile oder der REST-API auslesbaren Daten an das YARN-Modell übergeben zu können.

Das Connector-Interface `IHadoopConnector` stellt alle zum Abrufen der Daten oder weiteren Interaktion wie das Injizieren von Komponentenfehlern oder Starten von Anwendungen benötigten Methoden und Befehle bereit. Hierbei wird für das Monitoring unterschieden, ob die Daten vom TLS oder vom RM von Hadoop abgerufen werden. Dies ist vor allem bei der Nutzung der REST-API wichtig, da sich hier die Adressen und Pfade unterscheiden, während bei der Benutzung der Kommandozeile die Befehle gleich sind . Der TLS wird zum Abrufen der Daten vor allem aus dem Grund genutzt, da hierbei zusätzliche Daten ermittelt werden können, die bei der reinen Nutzung der Schnittstellen des RM nicht zurückgegeben werden würden. Ausgenommen sind hiervon Anwendungen, bei denen die Nutzung des TLS keine weiteren Daten von Hadoop zurückgegeben werden[16, 32–34]. Aus diesem Grund ist die Nutzung des TLS zum Monitoring von Anwendungen mithilfe des Connector-Interfaces nicht möglich.

Die Implementierten Parser und Connectoren sind jeweils als Singleton realisiert und sind von der Serialisierung des YARN-Modells durch S# ausgenommen. Dies liegt vor allem darin Begründet, dass dadurch Speicher eingespart wird, der dadurch für andere YARN-Komponenten zur Verfügung steht. Aber auch weitere Einschränkungen durch S# spielten eine Rolle (vgl. Abschnitt 2.1). Ein weiterer Vorteil liegt zudem darin, dass für Unterschiedliche Einsatzzwecke der gleiche Connector genutzt werden kann, und somit auch die einzelnen vom Connector benötigten SSH-Verbindungen für unterschiedliche Einsatzzwecke wiederverwendet werden können. Die Initialisierung der Parser und Connectoren erfolgt jeweils beim ersten Aufruf der Singletons. Dies geschieht innerhalb des Testsystems üblicherweise entweder beim Initialisieren des Tests (beschrieben in Abschnitt 6.4) oder beim Initialisieren des YARN-Modells selbst durch die Klasse `Model`. Die Initialisierung des Modells stellt zudem den einzigen Zeitpunkt dar, bei dem im YARN-Modell direkt mit den implementierten Parsern und Connectoren interagiert wird, jede andere Interaktion findet stattdessen gekapselt mithilfe der Interfaces statt.

Die drei Komponenten des Treibers sind zudem untereinander voneinander gekapselt. Bei der Ausführung des Parsers wird daher analog nur zur Initialisierung mit dem konkreten Connector interagiert, während das Connector-Interface zum Monitoring als einzige Schnittstelle dient. Da für die SSH-Verbindung kein eigenes Interface zur Kapselung existiert, ist die Kapselung der Connectoren und der bereitstellenden Klasse der SSH-Verbindung nicht so streng wie zwischen anderen Komponenten. Dennoch werden Befehle auf den Cluster-Hosts ausschließlich mithilfe des Connectors durchgeführt.

4.3.2. Entwicklung der Parser

Implementierung des CmdParsers

Implementierung des RestParsers

4.3.3. Entwicklung der Connectoren

Implementierung des CmdConnectors

Implementierung des RestConnectors

4.3.4. Implementierung der SSH-Verbindung

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 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:

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

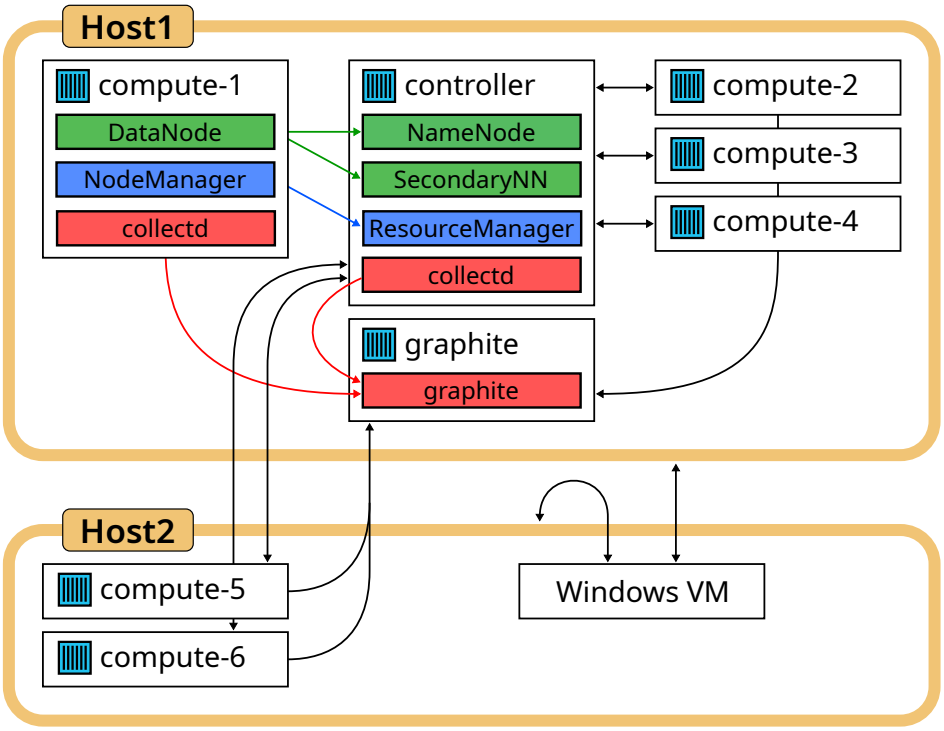

Abbildung 4.4.: In der Fallstudie verwendetes Cluster-Setup. Grün: HDFS, Blau: YARN, Rot: Graphite.

wird *curl*[1] 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.

[1] `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 [35]. 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 [36] 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 [37]. Bei der Ausführung auf dem in dieser Fallstudie verwendetem Cluster wurden jedoch nicht alle Workloads fehlerfrei ausgeführt. Zudem wird in [38] 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 [39]. 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 [36, 40].

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. [40] und [41]. 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 [42] und folgerten daher, dass für kleine Jobs evtl. einfachere Frameworks abseits von Hadoop besser geeignet wären. Die Autoren der Studie in [41] 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.

```
1  public class Benchmark
2  {
3    public const string BaseDirHolder = "$DIR";
4    public const string OutDirHolder = "$OUT";
5    public const string InDirHolder = "$IN";
6
7    public Benchmark(int id, string name, string startCmd, string
         outputDir, string inputDir)
8    {
9      _StartCmd = startCmd;
10     _InDir = inputDir;
11     HasInputDir = true;
12   }
13
14   public string GetStartCmd(string clientDir = "")
15   {
16     var result = _StartCmd.Replace(OutDirHolder, GetOutputDir(
           clientDir)).Replace(InDirHolder, GetInputDir(clientDir));
17     if(result.Contains(BaseDirHolder))
18     result = ReplaceClientDir(result, clientDir);
19     return result;
20   }
21 }
22
23 using static Benchmark;
24 public class BenchmarkController
25 {
26
27   public static Benchmark[] Benchmarks { get; } // benchmarks
28   public static int[][] BenchTransitions { get; } // transitions
29
30   static BenchmarkController()
31   {
32     Benchmarks = new[]
```

```
33      {
34        new Benchmark(04, "wordcount", $"example wordcount {InDirHolder}
              {OutDirHolder}", $"{BaseDirHolder}/wcout", $"{BaseDirHolder
            }/rantw"),
35      };
36    }
37  }
38
39  public class Client : Component
40  {
41      public string StartBenchmark(Benchmark benchmark)
42      {
43          if(benchmark.HasOutputDir)
44          SubmittingConnector.RemoveHdfsDir(benchmark.GetOutputDir(
                ClientDir));
45          var appId = SubmittingConnector.StartApplicationAsync(
                benchmark.GetStartCmd(ClientDir));
46      }
47  }
```

Listing 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.

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.

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

```
12    // save benchmarks
13    PreviousBenchmark = CurrentBenchmark;
14    CurrentBenchmark = Benchmarks[i];
15  }
```

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

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 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 Abschnitt 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:

```
var node = Nodes.First(n => n.Name == nodeName);
var nodeUsage = (node.MemoryUsage + node.CpuUsage) / 2;

if(nodeUsage < 0.1) nodeUsage = 0.1;
else if(nodeUsage > 0.9) nodeUsage = 0.9;

NodeUsageOnActivation = nodeUsage; // for using on repairing

var faultUsage = nodeUsage * ActivationProbability * 2;

var probability = 1 - faultUsage;
var randomValue = RandomGen.NextDouble();
Logger.Info($"Activation probability: {probability} < {randomValue}");
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:

```
public void UpdateBenchmark()
{
    var benchChanged = BenchController.ChangeBenchmark();

    if(benchChanged)
    {
        StopCurrentBenchmark();
        StartBenchmark(BenchController.CurrentBenchmark);
    }
}
```

Listing 6.5: Auswahl und Start des nachfolgenden Benchmarks (gekürzt). Die Methode `BenchmarkController.ChangeBenchmark()` ist bereits in Listing 5.2 aufgeführt.

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

besser mit modellkapiutel koordinieren

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 inAbschnitt 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

besser mit modellkapiutel koordinieren

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 3.2.1 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 Abschnitt 3.2.2 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 Abschnitt 3.3.3 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 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 [43] 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 [43].

Da bei der Ausführung des Universalmutators auch zahlreiche Mutanten erzeugt werden, die nicht kompiliert bzw. ausgeführt werden können, nutzt das Tool die Compiler

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

der jeweiligen Sprache zur Validierung der generierten Mutationen. Ein validierter Mutant zeichnet sich hierbei dadurch aus, dass dieser durch den Original-Compiler der jeweiligen Sprache kompiliert werden kann und die generierten Objektdateien bzw. Bytecode nicht dem nicht-mutierten Original oder anderen bereits generierten Mutationen entsprechen [43]. Diese Validierung kann mithilfe von entsprechenden Startparametern durch ein benutzerdefiniertes Programm durchgeführt werden oder alternativ nicht durchgeführt werden [43, 44].

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 [21] 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 Abschnitt 3.3.2 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 Abschnitt 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   }
```

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

```
21    // catch exceptions and set isFailed=true
22    finally
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()
```

```
{
  yield return 0xAB4FEDD;
  yield return 0x11399D3;
}
```

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:

```
var todayStrShort = DateTime.Today.ToString("yyMMdd");
var mutated = isMutated ? "MT" : "MF";
var faultProbStr = faultProbability.ToString(CultureInfo.
    InvariantCulture);
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

In Abschnitt 6.3 wurden 32 Testkonfigurationen ermittelt, die mithilfe des in dieser Fallstudie entwickelten Testansatzes insgesamt 43 mal ausgeführt wurden. Die meisten Konfigurationen wurden hierbei jeweils einmal ausgeführt, 7 Konfigurationen wurden auch mehrmals ausgeführt. Die Gründe für eine mehrfache Ausführung einzelner Konfigurationen sind in diesem Kapitel 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 in Abschnitt 6.1.3 implementierten Struktur eines Simulations-Schrittes bzw. Testfalls nicht erkannt werden können.

Zur Auswertung der Evaluation dienten vor allem die in
Constraintabschnitt
implementierten Constraints, die sich aus den in Abschnitt 3.2 definierten Anforderungen ergeben.

In diesem Kapitel wurden folgende Begriffe mit folgenden Bedeutungen genutzt:

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.

7.1. Statistische Kenndaten

Die Dauer aller Simulationen betrug ca. 4:16:44 Stunden, die gesamte Ausführungsdauer inkl. Starten und Beenden des Clusters bei jeder Konfiguration betrug ca. 5:19:57 Stunden. Von den 290 Testfällen, die ausgeführt werden hätten sollen, wurden nur 222 Testfälle (77 %) ausgeführt. Der Grund für den Abbruch von 14 Tests liegt zum Großteil 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. Ein Test wurde aufgrund der zu geringen Anzahl an Submittern abgebrochen, was in Abschnitt 7.7.3 genauer erläutert wird.

Insgesamt wurde bei allen Tests 439 Komponentenfehler aktiviert (14 % von 3100 möglichen), von denen jedoch nicht alle injiziert wurden, da bei einigen Testfällen beide Komponentenfehler der Nodes gleichzeitig aktiviert wurden. In diesen Fällen überwog gemäß

Abschnitt mit Komponentenfehler im Node

die Aktivierung es Komponentenfehlers, der den Node komplett beendet. Von allen aktivierten Komponentenfehlern wurden während der Simulationen 262 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 (vgl. Abschnitt 7.5.1 und Abschnitt 7.6.1).

Bei den 43 Tests wurden 408 Anwendungen im Cluster gestartet, von denen mit 204 rund die Hälfte erfolgreich und damit vollständig ausgeführt wurden, aufgrund eines Fehlers vorzeitig beendet bzw. gefailt waren mit 110 etwas mehr als ein Viertel der gestarteten Anwendungen. Vorzeitig abgebrochen wurden bei den Tests 52 Anwendungen (13 %), was 42 Anwendungen macht, die zum Ende der Simulationen noch ausgeführt wurden. Nicht eingerechnet sind hier 29 nicht gestartete Anwendungen, die Gründe hierfür sind in Abschnitt 7.7.2 erläutert. Für die gestarteten Anwendungen wurden 555 Attempts gestartet, was im Schnitt 1,36 Attempts pro Anwendung ergibt. Auffällig ist hierbei, dass mit 214 Attempts 9 Attempts mehr aufgrund eines AppMstr-Timeouts abgebrochen wurde, als während der Simulation erfolgreich beendet wurden (203 Attempts). 32 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 46,5 % aller gestarteten Attempts, die nicht erfolgreich abgeschlossen werden konnten. Beim Monitoring wurden 3150 Anwendungs-Container

erkannt, was im Schnitt 7,72 Container pro Anwendung bzw. 5,68 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 78.825 Constraints validiert, von denen 573 vom Oracle als ungültig validiert wurden (0,73 %). Die meisten ungültigen Constraints hatten hierbei die Tests 31.2 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 ungültigen Constraints vor allem liegt darin, dass die Constraints für fehlerhaften Anwendungen auch in nachfolgenden Testfällen innerhalb einer Ausführung einer Testkonfiguration entsprechend erkannt werden (vgl. Abschnitt 7.7.1). Dies resultiert in bis zu 34 ungültigen Constraints für fehlerhafte Anwendungen bei den einzelnen Tests.

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

7.2. Zusammenfassung der Ergebnisse

Zusammenfassend basierend auf der vorhandenen Datenbasis der 43 ausgeführten Tests lässt sich sagen, dass sich das entwickelte Testsystem und das Hadoop-Cluster im Großen und Ganzen so verhält, wie es erwartet werden konnte. Dennoch wurden nicht alle der in Abschnitt 3.2.1 definierten funktionalen Anforderungen an das Cluster selbst und der in Abschnitt 3.2.2 definierten Anforderungen an das gesamte Testsystem vollständig erfüllt. Die meisten dieser Anforderungen wurden mithilfe der in

Constraints

definierten Constraints implementiert, wodurch die Anforderungen bei jedem Testfall automatisch validiert werden konnten.

Vor allem die Cluster-Anforderung, wonach alle Tasks vollständig ausgeführt werden, sofern sie nicht abgebrochen werden, wurde bei den 110 nicht erfolgreichen Anwendungen nicht erfüllt. Aber auch bei einigen vollständig ausgeführten Anwendungen wurde diese nicht komplett erfüllt, was die mit dem Exitcode -100 beendeten Attempts zeigen (vgl. Abschnitt 7.7.1). Bei Attempts mit dem Exitcode -100 wird zudem die Anforderung, dass kein Task an defekte Nodes gesendet wird, verletzt.

Bei der Validierung der Constraints ist es zudem vorgekommen, dass diejenigen für die Anforderungen, dass die Konfiguration aktualisiert sowie der Status des Clusters vom Testsystem erkannt und im Testmodell gespeichert wird, als ungültig validiert wurden. Dies waren nach einer genaueren Betrachtung der Gründe hierfür in Abschnitt 7.8.1 zum Teil jedoch falscher Alarm, wodurch diese Anforderungen zu großen Teilen als erfüllt angesehen werden können. Genauso verhält es sich bei den in Abschnitt 7.5.2 beschriebenen, nicht erkannten, injizierten und reparierten Komponentenfehlern, wonach

die Anforderungen, dass defekte Nodes und Verbindungsabbrüche erkannt werden, bei der Betrachtung eines einzelnen Testfalls in 19 Fällen zwar nicht erfüllt, bei der Betrachtung der gesamten Tests jedoch als erfüllt angesehen werden können.

Auch die Anforderung, dass sich der MARP-Wert anhand der ausgeführten Anwendungen verändert, wurde nicht immer erfüllt. Die Betrachtung der Werte bei den einzelnen Tests in Abschnitt 7.3 ergab, dass es durchaus möglich ist, dass die bei einem Test ausgeführten Anwendungen nicht ausreichen, damit sich der Wert verändert. Dennoch war es anhand dieser Anforderung möglich, die in Abschnitt 6.2 implementierten Mutanten der Selfbalancing-Komponente, bis auf einige in Abschnitt 7.4 erläuterte Besonderheiten, in den ausgeführten Tests zu erkennen. Auch die Anforderung, dass die in

impl komponentenfehler

implementierten Komponentenfehler im realen Cluster injiziert und repariert werden, konnte aufgrund des in Abschnitt 7.6.3 erläuterten Falls nicht immer erfüllt werden. In diesem Kontext zeigte sich aber, dass immer erkannt werden konnte, wenn keine weitere Rekonfiguration des Clusters möglich ist und diese Anforderung somit erfüllt wird (vgl. Abschnitt 7.6). Zudem konnte in einigen der in Abschnitt 7.7.2 erläuterten Fällen die Anforderung, dass mehrere Benchmark-Anwendungen gleichzeitig gestartet und ausgeführt werden können, nicht erfüllt werden.

Zu einem großen Teil erfüllt werden konnte jedoch die Anforderung, dass ein Test vollautomatisch ausgeführt werden kann. Lediglich bei der Ausführung von mehreren Testfällen direkt hintereinander mithilfe der in Abschnitt 6.4 implementierten `CaseStudyTests`-Klasse kam es vor, dass genauso wie in Abschnitt 7.7.3 die vom Connector bereitgestellten Submitter zum Starten von Anwendungen nicht ausgereicht haben.

Die genauen Gründe für die verletzten Anforderungen und Constraints sind in den bereits verwiesenen, nachfolgenden Abschnitten erläutert.

Die 43 ausgeführten Tests haben aber auch gezeigt, dass das Cluster ohne Auswirkung auf seine Funktionsweise auf einem oder mehreren Hosts ausgeführt werden kann. Auch zeigte sich bei den 7 Testkonfigurationen mit mehrmaligen Ausführungen, dass die Tests und seine Testfälle im Grunde mehrmals ausgeführt werden können. Die einzigen Unterschiede bei den jeweiligen Ausführungen waren ausschließlich durch die Verteilung der Last innerhalb des Clusters bedingt, was sich vor allem in direkten Vergleichen zwischen korrespondierenden Tests wie z. B. in Abschnitt 7.6.3 zeigt.

7.3. 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. Wäh-

rend bei allen Testkonfigurationen, bei denen Mutationen aktiv waren, der MARP-Wert unverändert blieb (vgl. Abschnitt 7.4), wurde er bei 17 von 21 Ausführungen der 16 Konfigurationen ohne Mutationen verändert:

Konf.	1.1	1.2	3	5.1	5.2	7.1	7.2
Wert	0,100	0,100	0,474	0,242	0,100	0,100	1,000

Konf.	9.1	9.2	11	13	15	17	19
Wert	0,269	0,175	0,539	0,356	0,368	0,731	0,430

Konf.	21	23	25	27	29	31.1	31.2
Wert	0,335	0,498	0,521	,0819	0,273	0,488	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 in diesen Tests allen 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 auf mehr aktive Nodes 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 der drei Testausführungen zunächst das hierfür genutzte Constraint als ungültig validiert.

7.4. 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 Abschnitt 7.3).

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 einer der in Abschnitt 6.2 definierten Mutanten aktiv ist. Die Auswahl dieser Testkonfiguration hierfür liegt darin begründet, dass hier das Cluster auf beiden Hosts mit zusammen sechs Nodes gestartet wird, auf denen bis zu vier Anwendungen gleichzeitig gestartet werden. Zudem wurde bei den Tests 9.1 und 9.2 festgestellt, dass sich der MARP-Wert nicht direkt im ersten, sondern auch in später ausgeführten Testfällen ändern kann, er aber während der Ausführung wirklich geändert wird (vgl. Abschnitt 7.3).

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.

7.5. Betrachtung 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 bzw. der Lastverteilung bei den zuvor ausgeführten Testfällen eines Tests ab (vgl. Abschnitt 6.1.3). Daher wurden abhängig von der Lastverteilung im Cluster auch bei einer mehrmaligen Ausführung der gleichen Konfiguration bei einigen Testausführungen 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 Folgenden wird nun ein Überblick über die bei den Tests aktivierten bzw. deaktivierten und nicht injizierten Komponentenfehler bzw. erkannten Injektionen und Reperaturen der Komponentenfehler gegeben.

7.5.1. Aktivierte und deaktivierte Komponentenfehler

Die Aktivierung und Deaktivierung der Komponentenfehler hing manchmal stark, manchmal weniger stark von der ausgeführten Testkonfiguration ab. Im Vergleich zwischen korrespondierenden Konfigurationen, die sich nur in der Nutzung des Mutationsszenarios unterschieden, wurde nur 5 mal bei allen Tests die gleiche Anzahl an Komponentenfehler aktiviert, bei der Deaktivierung der Komponentenfehler besitzen nur 4 korrespondierende Konfigurationen die gleiche Anzahl bei allen Tests. Die Anzahl der aktivierten und deaktivierten Komponentenfehler unterschied sich dagegen in 8 bzw. 7 korrespondierenden Testkonfigurationen um einen einen Komponentenfehler in allen Testausführungen. Bei den anderen korrespondierenden Konfigurationen unterschied sich die Anzahl bei allen jeweiligen Tests um mehr als einen Komponentenfehler. Mit jeweils 20 aktivierten Komponentenfehlern wurden bei den Tests 28.1, 31.1 und 32 die meisten aktiviert, die meisten Komponentenfehler deaktiviert wurden bei den Tests der 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 (vgl. Abschnitt 7.6.1).

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 darüber hinaus 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 diejenigen mit Mutationen aufweisen. Nur bei den Konfigurationen 9 und 10, 13 und 14, 27 und 28 und 31 und 32 weisen einige Tests ohne Mutationen eine geringere Anzahl an aktivierten Komponentenfehler auf als Tests mit Mutationen. Dies liegt wohl auch darin begründet, dass durch den veränderten MARP-Wert die verfügbaren Ressourcen besser an die Anwendungen verteilt werden konnten und bestätigt damit die Funktionalität der von Zhang u. a. entwickelten Komponente.

Weitere Auffälligkeiten ergeben sich zudem beim Vergleich der Ausführungszeiten der Simulationen. Die Tests 9.2, 15, 31.1 sowie 31.2 stellen die einzigen Test ohne Mutationen dar, bei denen die Simulation schneller abgeschlossen wurde als in den korrespondierenden Tests mit Mutationsszenario. Da sich das mit der generellen Aussage beim Vergleich der aktivierten Komponentenfehler deckt, kann davon ausgegangen werden, dass die geringere Anzahl an Komponentenfehler zudem die Auswirkung hat, dass Anwendungen schneller gestartet werden können. Der Grund hierfür könnte darin liegen, dass bei weniger injizierten Komponentenfehler auch entsprechend weniger Verwaltungsaufwand für bereits ausgeführte Anwendungen nötig ist, wodurch neu gestartete Anwendungen ebenfalls schneller durch das Cluster verarbeitet werden können. Um hier jedoch eine fundierte Aussage treffen zu können, wären weitere vergleichende Tests nötig (vgl.

auf Diskussion dazu, evtl. auch mit diskussion zu selfbalancing-komponente verbinden

)

7.5.2. Nicht erkannte, injizierte bzw. reparierte Komponentenfehler

Bei 18 aller ausgeführten Tests kam es vor, dass ein injizierter bzw. reparierter Komponentenfehler zunächst nicht vom Testsystem erkannt wurde. Das betraf konkret

drei mal das Injizieren eines Komponentenfehlers sowie 16 mal das Reparieren eines Komponentenfehlers:

Test	Testfall	Art	Node
1.1	5	Node beenden	4
2	5	Node starten	2
7.1	2	Node beenden	5
7.1	5	Node starten	5
7.2	5	Node starten	5
11	6	Node trennen	6
17-28.2	2	Node verbinden	4

Tabelle 7.2.: Übersicht der nicht erkannten, injizierten bzw. reparierten Komponentenfehler

Bei den aufgetretenen als ungültig validierten Constraints fällt auf, dass die betroffenen Nodes im jeweils nachfolgenden Testfall mit ihrem jeweils korrekten Status erkannt wurden. Die 19 als ungültig markierten Constraints zu den Anforderungen aus Abschnitt 3.2, dass defekte Nodes und Verbindungsabbrüche erkannt werden, wurden somit korrekt, als auch inkorrekt, als ungültig validiert. Dies liegt daran, dass das Cluster bei defekten Nodes erst einige Zeit benötigt, um zu erkennen, dass ein Node ausgefallen ist. Auch wenn ein Node nicht mehr defekt ist, benötigt dieser bzw. der RM erst einige Zeit, bis erkannt wird, dass der Node wieder aktiv ist. Dies liegt einerseits daran, dass Hadoop bzw. der RM nicht kontinuierlich, sondern periodisch nach einer bestimmten Zeitspanne den Status der Nodes prüft und bei nicht erreichbaren Nodes zunächst solange wartet, bis die Abfrage durch einen *Timeout* beendet wird. Zwar wurden beide Zeitspannen im getesteten Cluster auf jeweils 10 Sekunden festgelegt, jedoch reichte diese Zeitspanne wohl nicht immer aus, um den Status rechtzeitig zu erkennen. Beim Starten bzw. Wiederverbinden eines Nodes sieht dies analog dazu aus, wobei Hadoop auf dem jeweiligen Node hier zunächst gestartet werden muss, bevor es sich dann selbstständig mit dem RM verbindet, was ebenfalls eine gewisse Zeit benötigt. Dies wird auch dadurch bestätigt, dass für die betroffenen Nodes in den jeweils nachfolgenden Testfällen bzw. dem finalen Clusterstatus oder in korrespondierenden Tests der Status korrekt erkannt wurde, den die Nodes gemäß aufgrund der Komponentenfehler besitzen sollten.

Eine Besonderheit bilden hierbei zunächst die beiden Tests zur Konfiguration 7. Im Test 7.1 wurde der gleiche Node zunächst nicht als defekt erkannt bevor er im letzten Testfall noch als defekt erkannt wurde, obwohl er bereits wieder gestartet wurde. Dazwischen wurde der betroffene Node 5 zunächst im Testfall 3 wieder gestartet, bevor er im Testfall 4 wieder beendet wurde, von denen beide Aktionen korrekt erkannt wurden. Im Unterschied zum ersten Test der Konfiguration wurde im Test 7.2 nur das Starten des Nodes nicht korrekt erkannt, während das beenden des Nodes 5 im 2.

Testfall erkannt wurde. Im finalen Clusterstatus ist der Node jedoch wie im Test 7.1 korrekt als aktiv markiert.

Die zweite Besonderheit bilden die Tests der Konfigurationen 17 bis 28. Hier wurde in allen Tests der Node 4 im jeweils ersten Testfall direkt vom Cluster getrennt, was noch korrekt erkannt wurde. In den Testkonfigurationen 25 bis 28 wurde im nachfolgenden dritten Testfall jedoch der Node direkt wieder vom Netzwerk getrennt, weshalb hier nur vermutet werden kann, dass der Node im zweiten Testfall korrekt mit dem Cluster verbunden wurde. Davon kann jedoch ausgegangen werden, da in den sechs Tests auf einem Host (Tests 17 bis 22) der Node im nachfolgenden, dritten Testfall nicht verändert wird und auch als aktiv erkannt wurde. Auch in den Tests 29 bis 32 wurde alles korrekt erkannt, da hier der Node im ersten Testfall ebenfalls getrennt, im zweiten wieder verbunden, und im dritten Testfall zudem erneut vom Cluster getrennt wird. In den Tests 23 bis 28.2 wird der Node dagegen zwar ebenfalls im dritten Testfall wieder vom Cluster getrennt, jedoch wird dies hier auch wieder korrekt erkannt.

7.6. Rekonfiguration des Clusters nicht möglich

Insgesamt 13 der 42 ausgeführten Tests wurden vorzeitig abgebrochen, da eine Rekonfiguration des Clusters nicht möglich war. Dies entspricht dem in Abschnitt 3.2.2 gefordertem und in Abschnitt 6.1.3 implementierten Verhalten, wenn alle Node im Cluster defekt sind. Im Folgenden wird für die betroffenen Testkonfigurationen und Tests betrachtet, weshalb es dazu kam.

7.6.1. 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. Das 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 aufgrund der hohen Last einen Komponentenfehler injiziert bekamen. Durch

diesen einen nun weiterhin ausgeführten Node ist es dem Cluster daher möglich gewesen, sich im Test 3 zu rekonfigurieren.

7.6.2. 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 im fünften Testfall der vier Tests folgende Auslastung:

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

Tabelle 7.3.: 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 fünften Testfall führte im darauf folgenden Testfall dazu, dass bei allen noch aktiven Nodes ein Komponentenfehler aktiviert wurde. Daher wurden die beiden Tests im jeweils sechsten ausgeführten Testfall abgebrochen. Es ist auch davon auszugehen, dass der Test 14 aufgrund der ebenfalls sehr hohen Auslastung im sechsten Testfall wahrscheinlich ebenfalls abgebrochen worden wäre.

7.6.3. 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:

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 im ersten Testfall 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.

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

Tabelle 7.4.: Auslastungen und Komponentenfehler in Node 1 der Tests 19 bis 22

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, hat jedoch eine positive, als auch eine negative Seite. So wurde auch ein externer, nicht direkt durch die Anforderungen in Abschnitt 3.2.2 abgedeckter Fehler erkannt, jedoch auf Kosten der Anforderung, dass im Modell implementierte Komponentenfehler im realen Cluster repariert werden.

7.6.4. 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 dritten Testfall beendet, da die Auslastung des Nodes im zweiten 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.

kürzen und die genaue erklärung auf entsprechende abschnitte verweisen

Der Grund für die Injizierung von Komponentenfehlern bei noch allen aktiven Nodes im achten Testfall liegt in der Auslastung der Nodes im siebten Testfall. Diese beträgt im Test 27 ohne Mutationen bei den beiden betroffenen Nodes 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.

7.6.5. Testkonfigurationen 31 und 32

Die Tests der Konfigurationen 31 und 32 verliefen ähnlich zueinander. Die hohe Anzahl der 19 bzw. 20 aktivierten Komponentenfehler reichten bei jeweils 11 wieder deaktivierten Fehlern aus, um die Tests 31.2 und 32 im achten Testfall abzubrechen. Der Test 31.1 verlief zwar ebenfalls ähnlich zu den beiden anderen Tests, wurde jedoch aufgrund fehlender, verfügbaren Submitter des Connectors beendet. Die Gründe dafür sind in Abschnitt 7.7.2 erläutert, weshalb der Test 31.1 hier nicht genauer betrachtet wird.

Bei den beiden Tests 31.2 und 32 fällt auf, 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 dritten ausgeführten 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, während Node 3 bereits im Testfall zuvor beendet wurde. Ebenso verlief der dritte Testfall auch in den beiden Tests der Konfigurationen 29 und 30, bei denen nur fünf Testfälle ausgeführt wurden.

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

7.7. Betrachtung der Anwendungen

Bei der Betrachtung der Anwendungen sind vor allem zwei Punkte aufgefallen: Viele Anwendungen wurden aufgrund von Fehlern beendet und einige Anwendungen, vor allem beim zweiten Seed, konnten nicht gestartet werden. Dies widerspricht zum Teil den in Abschnitt 3.2 definierten Anforderungen, hat aber mehrere Gründe, die im Folgenden erläutert werden.

7.7.1. Aufgrund von Fehlern abgebrochene Anwendungen

Wie bereits erwähnt, sind etwas mehr als ein viertel aller gestarteten Anwendungen gefailt, was im Schnitt 2,6 gefailte Anwendungen pro ausgeführten Test ergibt. Die meisten gefailten Anwendungen sind hierbei mit 9 bzw. 8 bei den Tests der Konfigurationen 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. Ebenfalls auffallend ist, 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 3.2.1 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 343 ungültigen Constraints mehr als die Hälfte aller ungültigen Constraints ausmacht (59,9 %). Im Schnitt ergibt das somit rund 8 ungültige Constraints pro Test bzw. ca. 1,8 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. Dadurch sind ein Großteil der als ungültig validierten Constraints ein falscher Alarm, da die entsprechende Anforderung pro Anwendung nur einmal nicht erfüllt werden kann. Aussagekräftiger ist daher die Anzahl von 110 nicht vollständig abgeschlossenen bzw. aufgrund eines Fehlers abgebrochenen 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. Dadurch wird der AppMstr nach einiger Zeit mit dem Fehler *AppMstr-Timeout* als abgebrochen markiert.
- Die den AppMstr zugewiesenen Nodes sind vollständig ausgelastet, wodurch dem AppMstr selbst die benötigten Ressourcen nicht allokiert bzw. der AppMstr nicht ausgeführt werden kann. Nach einiger Zeit wird der AppMstr daher mit dem Fehler *AppMstr-Timeout* abgebrochen. Das beinhaltet auch Timeouts, wenn einem AppMstr nicht einmal ein ausführender Node zugewiesen werden kann.
- 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 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. Aufgrund eines Fehlers im Map-Task wird auch die `fail`-Anwendung beendet, jedoch ist das in diesem Fall das gewünschte Verhalten der Anwendung und zählt daher nicht als Fehler.

- Der AppMstr eines Attempts wird mit dem Exitcode -100 beendet. Dieser Fehler kommt dann vor, wenn versucht wird, einen Task eines Anwendungs-Containers der jeweiligen Anwendung bzw. Attempt auf einem defekten Node auszuführen und widerspricht somit zusätzlich der in Abschnitt 3.2.1 Anforderung, dass kein Task oder Anwendung an defekte Nodes gesendet wird. Dieser Fehler trat nur dann auf, wenn im ausführende Node des betroffenen AppMstr im gleichen Testfall ein Komponentenfehler injiziert wurde und der Node dadurch 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 die benötigten Ressourcen zu dem Zeitpunkt 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, wodurch hier die Anforderung, dass ein Task vollständig ausgeführt werden muss, teilweise erfüllt werden kann.

Bei einigen der AppMstr-Timeouts aufgrund der Aktivierung von Komponentenfehler lässt sich zudem ein spezielles Muster erkennen. Hierbei wurde in einem zuvor ausgeführten Testfall auf einem Node ein AppMstr einer Anwendung ohne Fehler 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. Dabei

ist es nun möglich, dass dies noch während der Aktivierung von Komponentenfehlern innerhalb des Testfalls geschieht (vgl. Abschnitt 6.1.3), 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.

7.7.2. Nicht gestartete Anwendungen

Bei den Tests 19, 25 und 27 bis 32 kam es vor, dass insgesamt 29 Anwendungen nicht gestartet werden konnten. Meistens war die Anwendung `terasort` davon betroffen, einige male die Anwendung `teravalidate`. Ursächlich dafür ist die jeweils hohe Auslastung des Clusters in den Testfällen zuvor, bei denen den benötigten AppMstr der `teragen`-Anwendungen keine Ressourcen auf den ausführenden Nodes allokiert werden konnte und diese daher mit einem AppMstr-Timeout beendet wurden (vgl. Abschnitt 7.7.1). Da in Abschnitt 6.3 definiert wurde, dass benötigte Eingabedaten für Anwendungen während der Ausführung der Tests generiert werden, konnten so die benötigten Eingabedaten für die Anwendung `terasort` nicht generiert werden (vgl.
details zu tsort
). Aufgrund der fehlenden Daten wurde daher die Anwendung direkt wieder abgebrochen, wodurch in 42 Testfällen nicht jeder Client eine Anwendung ausgeführt hat. In diesen Fällen wurde als Resultat zudem das Constraint der Anforderung aus Abschnitt 3.2.2, wonach mehrere Benchmark-Anwendungen gleichzeitig gestartet und ausgeführt werden können, aufgrund der Implementation aus
Constraint-Impl
durch das Oracle als ungültig validiert. Analog dazu verhält es sich bei der Anwendung `teravalidate`, welche wiederum die `terasort`-Ausgabedaten als Eingabedaten benötigt (vgl.
details zu tvalidate
).

7.7.3. Nicht ausreichend Submitter

Ein unerwarteter Fehler trat bei der Ausführung des Testfalls 31.1 auf. Hierbei kam es vor, dass die für die anderen Tests genutzten acht Submitter des Connectors zum Starten von Anwendungen (vgl. Abschnitt 4.3.3) nicht ausreichend waren. Der Test wurde hierbei im achten Testfall abgebrochen, weil keine weiteren freien Submitter zur Verfügung standen. Daher wurde der Test zur Konfiguration 31 mit zehn Submittern erneut ausgeführt, wodurch dieser wie in Abschnitt 7.6.5 erläutert im achten Testfall aufgrund fehlender Rekonfigurierbarkeit abgebrochen wurde. Die Gründe für den Abbruch des Tests 31.1

liegen darin, dass die Docker-Container der Benchmarks (vgl. Abschnitt 4.4) nicht korrekt beendet wurden und die Submitter daher auf weitere Ausgaben der gestarteten Anwendungen gewartet haben.

Warten der submitter (auch im Treiber) nochmal genauer erklären

7.8. 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 Abschnitt 3.2.2 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.

7.8.1. Nicht erkannte Nodes auf Host 2

Einer der beiden Fälle ist, dass ausführende Nodes von Anwendungen bzw. Attempts nicht erkannt bzw. ausgegeben wurden, wodurch vom Oracle auch Verletzungen gegen die in Abschnitt 3.2 definierten Anforderungen erkannt wurden, wonach die Konfiguration des Clusters aktualisiert, und der aktuelle Status im Cluster erkannt und im Testmodell gespeichert werden muss. 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 (vgl. Abschnitt 7.7.1).

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 die beiden auf Host 2 ausgeführten Nodes der betroffenen AppMstr. In allen sechs betroffenen Tests wurden nur die vier auf Host 1 ausgeführten Nodes als ausführende Nodes der Attempts bzw. Anwendungen erkannt und auch in den Logdateien ausgegeben. Die auf Host 2 ausgeführten Nodes wurden gemäß des SSH-Logs allerdingsn ebenfalls übertragen, sofern den Attempts bzw. Anwendungen ein Node zugewiesen wurde, jedoch wurden diese nicht im Programmlog ausgegeben. 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

). Dennoch lässt sich sagen, dass die beiden verletzten Anforderungen nach einer genaueren Begutachtung der Gründe dafür ein falscher Alarm des Oracles war.

7.8.2. Diagnostic-Daten von Anwendungen

Bei allen Tests ist zudem aufgefallen, dass die Diagnostic-Daten von Anwendungen nicht im Programmlog enthalten sind. Genauso wie bei den nicht erkannten Nodes auf Host 2 (vgl. Abschnitt 7.8.1) wurden alle Diagnostic-Daten von Hadoop an das Testsystem übertragen, die der Anwendungen im Gegensatz zu denen der Attempts jedoch nicht gespeichert. Zur Auswertung der Daten im Rahmen der Evaluation ist dies zwar irrelevant, da dies auch aufgrund der Daten der Attempts geschehen konnte, allerdings wird dadurch die in Abschnitt 3.2.2 definierte Anforderung an das Testsystem nur teilweise erfüllt, wonach der jeweils aktuelle Status des Clusters erkannt und gespeichert wird.

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.

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. *yarn-default.xml*. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-common/yarn-default.xml (besucht am 04. 07. 2018).

[16] 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).

[17] 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).

[18] 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).

[19] Apache Software Foundation. *hdfs-default.xml*. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml (besucht am 04. 07. 2018).

[20] 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).

[21] 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.

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

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

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

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

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

[27] 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.

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

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

[30] Docker Inc. *Docker development best practices*. URL: https://docs.docker.com/develop/dev-best-practices/ (besucht am 04. 07. 2018).

[31] 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.

[32] 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).

[33] 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).

[34] 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).

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

[36] 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.

[37] 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).

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

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

[40] 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.

[41] 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).

[42] 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.

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

[44] 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 [33] zu finden.

```
1  $ yarn application --list --appStates ALL
2  18/02/08 15:37:51 INFO impl.TimelineClientImpl: Timeline service
       address: http://0.0.0.0:8188/ws/v1/timeline/
3  18/02/08 15:37:51 INFO client.RMProxy: Connecting to ResourceManager
       at controller/10.0.0.3:8032
4  18/02/08 15:37:51 INFO client.AHSProxy: Connecting to Application
       History server at /0.0.0.0:10200
5  Total number of applications (application-types: [] and states: [NEW,
       NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED
       ]):1
6  Application-Id  Application-Name  Application-Type  User  Queue State
       Final-State Progress  Tracking-URL
7  application_1518100641776_0001  QuasiMonteCarlo MAPREDUCE root
       default FINISHED  SUCCEEDED 100%  http://controller:19888/
       jobhistory/job/job_1518100641776_0001
```

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

```
1  $ yarn application --status application_1518100641776_0001
2  [...]
3  Application Report :
4      Application-Id : application_1518100641776_0001
5      Application-Name : QuasiMonteCarlo
6      Application-Type : MAPREDUCE
7      User : root
8      Queue : default
9      Start-Time : 1518103712160
10     Finish-Time : 1518103799743
11     Progress : 100%
```

```
12     State : FINISHED
13     Final-State : SUCCEEDED
14     Tracking-URL : http://controller:19888/jobhistory/job/
           job_1518100641776_0001
15     RPC Port : 41309
16     AM Host : compute-1
17     Aggregate Resource Allocation : 1075936 MB-seconds, 942 vcore-
           seconds
18     Diagnostics :
```

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

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

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

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

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 [16, 32, 34] zu finden.


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

```
29          "preemptedResourceVCores": 0,
30          "numNonAMContainerPreempted": 0,
31          "numAMContainerPreempted": 0
32        }
33      ]
34    }
35  }
```

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

```
1  {
2    "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 }
```

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

C. Ausgabeformat des Programmlogs

Die in Abschnitt 3.3.3 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 vollständig und erfolgreich 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, wie in Abschnitt 6.1.3 beschrieben, abgebrochen.

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
31.1	0x11399D3	2	6	10	keine	7	10:41
31.2						7	10:21
32	0x11399D3	2	6	10	1,2,3,4	7	11:08

Tabelle D.1.: Übersicht der ausgeführten Testkonfigurationen