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

Matrikel-Nr.:	1450117
Datum:	30. Juli 2018
Betreuer:	M.Sc. Benedikt Eberhardinger
Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Bernhard Bauer

Zusammenfassung

Durch eine Automatisierung von Tests lassen sich im Bereich der Softwareentwicklung hohe Kosten einsparen. Daher wurden zahlreiche Test-Frameworks und Möglichkeiten zum Testen von Systemen und ihrer Software entwickelt. Ein solches Framework ist S# (Safety Sharp), mit dem mithilfe eines modellbasierten Ansatzes Systeme getestet werden können. Mithilfe des S#-Frameworks soll nun ein Testsystem entwickelt werden, um hiermit automatisiert ein verteiltes, adaptives Load-Balancing-System zu testen. Hierfür wurde Apache Hadoop ausgewählt, welches mit einer selbstadaptiven Komponente ergänzt wird. Diese selbstadaptive Komponente verändert dynamisch, und basierend auf den derzeit auf dem Hadoop-Cluster ausgeführten Anwendungen, einige der sonst statischen Einstellungen von Hadoop, womit die verfügbaren Ressourcen des Clusters optimaler genutzt werden können.

Um Hadoop testen zu können, wurde zunächst mithilfe von S# ein Modell entwickelt, welches die wesentlichen Komponenten des YARN-Frameworks von Hadoop abbildet. Dieses Modell wurde wiederum mithilfe eines hierfür entwickelten Treibers mit einem realen Hadoop-Cluster verbunden. Dadurch wurde es ermöglicht, durch die Testausführung mit S# unterschiedliche Anwendungen auf dem realen Cluster auszuführen und die Daten der Anwendungen und des Clusters im Modell zu nutzen. Um zu testen, ob sich das entwickelte Testsystem zur Testautomatisierung eines verteilten, adaptiven Load-Balancing-Systems eignet, wurde hierfür eine Fallstudie durchgeführt.

In dieser Masterarbeit werden der Aufbau und Ablauf der durchgeführten Fallstudie sowie die Entwicklung und Implementierung des hierfür genutzten Testsystems erläutert. Es wird gezeigt, welche Besonderheiten bei der Durchführung und Auswertung der Fallstudie aufgetreten sind, und inwiefern sich das entwickelte, modellbasierte Testsystem zur Testautomatisierung eines verteilten, adaptiven Load-Balancing-Systems eignet.

Abstract

By automating tests, high costs can be saved in software development. Therefore, numerous test frameworks and ways to test systems and their software have been developed. One such framework is S# (Safety Sharp), which uses a model-based approach to test systems. By using the S# framework, a test system will be developed to automatically test a distributed, adaptive load-balancing system. For this, Apache Hadoop was chosen, which is equipped with an adaptive resource manager. The manager detect the current usage of the cluster and modify some of Hadoop's otherwise static settings to make a better use of the available resources.

To test hadoop, a S# model was developed, which contains the essential components of the Hadoop YARN framework. To connect the model to a real Hadoop cluster, a driver was developed for this purpose. This allows S# to run different applications on the real cluster and detect the state of the running applications and the cluster. To determine the developed test system is suitable for the test automation of a distributed, adaptive load-balancing system, a case study was performed.

This master thesis explains the structure and processes of the case study, as well as the development and implementation of the test system used for this purpose. It shows the won experiences by performing the case study and shows how the developed, model-based test system is suitable for test automation of a distributed, adaptive load-balancing system.

Inhaltsverzeichnis

Verzeichnisse

Abbildungen

Listings

Tabellen

Glossar und Abkürzungen

AM ApplicationManager

Anwendung Ein auf dem Hadoop-Cluster ausgeführtes Programm.

AppMstr ApplicationMaster

Attempt Ausführungsinstanz einer Anwendung auf dem Hadoop-Cluster.

CLI Kommandozeile

Container 1. Ausführungsinstanz eines Tasks einer YARN-Anwendung auf dem Hadoop-Cluster. 2. Ausgeführte Instanz eines Docker-Images.

DCCA Deductive Cause-Consequence Analysis

HDFS Hadoop Distributed File System

MARP `maximum-am-resourcepercent`

MC Model Checking

MC Model Checker

MR MapReduce

Mutationstest Test, bei denen das zu testende Programm verändert wird. Ziel hierbei ist es, Fehler im Programm oder Testsystem zu finden (vgl. Abschnitt 6.2).

NM NodeManager

Regex Regular Expression

REST Abkürzung für *Representational State Transfer*. Programmierparadigma um maschinenlesbare Schnittstellen, z. B. für Webservices, bereitzustellen.

RM ResourceManager

S# Safety Sharp

SuT System under Test

SWIM Statistical Workload Injector for Mapreduce

System under Test Das mit einem Test zu testende System selbst.

Test Eine Ausführung einer Testkonfiguration mit mehreren Testfällen. Um mehrmalige Ausführungen einer Testkonfiguration zu kennzeichnen, wurde der jeweiligen Konfiguration eine weitere Ziffer angehängt. Alle ausgeführten Test sind in Anhang F zu finden.

Testfall Ein ausgeführter Schritt der Simulation. 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 (vgl. Unterabschnitt 6.1.4).

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

TLS Timeline Server

YARN Yet Another Resource Negotiator

Abkürzungen der Benchmarks (vgl. Unterabschnitt 5.2.1)

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 Gegenstand und Thema der Arbeit

Im Bereich der Softwaretests wird heutzutage meist mit automatisierten Testverfahren gearbeitet. Dies ist insofern logisch, als dass diese Testautomatisierung einerseits Aufwand und damit andererseits auch 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. Einige Beispiele für solche Test-Frameworks sind die *xUnit*-Frameworks, wie JUnit[1] für Java-Programme und NUnit[2] für .NET-Programme. Dabei werden zunächst einzelne Testfälle erstellt, die im Anschluss mit der jeweils aktuellen Codebasis jederzeit ausgeführt werden können. Automatisierte Tests können auch dazu genutzt werden, um einzelne Tests mit unterschiedlichen Eingabedaten durchzuführen. Dadurch können verschiedene Eingabeklassen (z. B. negative oder positive Ganzzahlen) mit sehr geringem Aufwand in einem Test genutzt werden und somit verschiedene Testfälle ausgeführt werden, wodurch eine massive Kosteneinsparung einhergeht [1].

Es gibt aber nicht nur Frameworks für Komponententests, sondern auch für andere Testverfahren, z. B. für modellbasierte Tests wie dem Model Checking (MC). Beim MC wird ein Modell automatisiert auf seine Spezifikation getestet und geprüft, unter welchen Umständen diese verletzt wird [2, 3]. Ein solches Framework, das zudem weitere Möglichkeiten zum Testen von Systemen bietet, ist Safety Sharp (S#). Mithilfe von S# können, meist selbstorganisierende oder sicherheitskritische, Systeme mit einem modellbasierten Ansatz vollautomatisch getestet werden.

Das Framework S# ist jedoch nicht nur auf den Einsatz mit selbstorganisierenden und sicherheitskritischen System beschränkt. So wurde bereits in [4] mit der von Cheng entwickelten ZNN.com-Fallstudie [5] ein Load-Balancing-System als S#-Modell implementiert und entsprechende Tests durchgeführt.

In dieser Masterarbeit soll nun eine ähnliche Fallstudie durchgeführt werden. Konkret soll hier mit Apache Hadoop[3] ein reales, und in der Forschung und Praxis eingesetztes [6], verteiltes Load-Balancing-System als zu testendes System (System under Test, kurz SuT) dienen. Hadoop soll jedoch nicht in seiner Standardversion getestet werden, sondern gemeinsam mit der von Zhang et al. entwickelten, selbstadaptiven Komponente [7]. Diese Komponente sorgt dafür, dass einige sonst statische Einstellungen von Hadoop während der Laufzeit, abhängig von der Auslastung des Hadoop-Clusters, dynamisch verändert werden [7]. Mithilfe des S#-Frameworks soll hierfür ein modellbasierter Ansatz entwickelt und ausgeführt werden. Damit einhergehend soll auch analysiert

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

werden, wie ein reales System in einem zum Testen entwickelten Modell eingebunden werden kann. Ziel ist es in dieser Masterarbeit nicht Hadoop selbst, sondern den hierfür entwickelten Testansatz zu testen. Aus diesem Grund sollen hierfür auch einige Mutationstests entwickelt werden, bei denen die selbstadaptive Komponente von Zhang et al. entsprechende Mutationen erhält, welche bei der Testausführung vom Testsystem erkannt werden müssen.

Da der im Rahmen dieser Masterarbeit entwickelte Testansatz auch alle wesentlichen Komponenten von Hadoop beschreibt, konnte er auch zum Testen von Hadoop selbst genutzt werden. Die hierfür durchgeführte Fallstudie und ihre Ergebnisse wurden bereits in [8] publiziert.

Hadoop wurde für diese Fallstudie vor allem aus dem Grund ausgewählt, da es bereits in der Praxis sehr häufig eingesetzt wird [6] und die auf einem Hadoop-Cluster ausgeführten Anwendungen dynamisch auf dem Cluster allokiert werden. Die von Zhang et al. entwickelte Komponente ergänzt Hadoop, sodass Anwendungen schneller und optimaler ausgeführt werden können [7]. Dadurch bildet Hadoop ein verteiltes, selbstadaptives Load-Balancing-System, das mithilfe von S# getestet werden soll. Das für die Tests genutzte reale Hadoop-Cluster soll hierbei in einer Docker-Umgebung[4] ausgeführt werden, um so ein verteiltes Cluster zu simulieren.

Zunächst werden in dieser Masterarbeit im Kapitel 2 weitere Informationen zu S#, Hadoop und der hierfür entwickelten selbstadaptiven Komponente erläutert. In Kapitel 3 folgt der grundlegende Aufbau dieser Fallstudie, deren Implementierung in Kapitel 4 erläutert wird. Da für die Testausführung die auf dem Hadoop-Cluster auszuführenden Anwendungen dynamisch, und während der Laufzeit, ausgewählt werden sollen, wird die Entwicklung des hierfür benötigten Transitionssystems in Kapitel 5 beschrieben. Im darauf folgenden Kapitel 6 werden die zur Ausführung benötigten Implementierungen beschrieben. Die Evaluation der in dieser Fallstudie ausgeführten Tests folgt in Kapitel 7, bevor abschließend in Kapitel 8 die Ergebnisse der Fallstudie zusammenfassend diskutiert werden.

[4] `https://www.docker.com/`

2 Grundlagen und Stand der Technik

Im folgenden werden zunächst die genutzten Frameworks und Tools erläutert, die zur Durchführung der Fallstudie benötigt wurden. Dies sind neben Hadoop selbst auch die hierfür entwickelte, und in dieser Fallstudie genutzte, selbstadaptive Komponente für Hadoop, sowie die Plattform Hadoop-Benchmark, die zur einfachen Ausführung eines Hadoop-Clusters dient. Zunächst wird jedoch das Framework S# vorgestellt, das zur Durchführung der Fallstudie genutzt wurde.

2.1 Safety Sharp

S# ist ein am Institute for Software & Systems Engineering der Universität Augsburg entwickeltes Framework zum modellbasierten Testen und Verifizieren von Systemen. Entwickelt wurde das Framework mithilfe des .NET-Frameworks und der Sprache C#, die auch zum Entwickeln der zu testenden Modelle genutzt werden. Dadurch können zahlreiche Funktionen 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, 9]. Dadurch können alle Schritte einer vollständigen Analyse inkl. Modellierung auch direkt im Visual Studio ausgeführt werden und somit auch die Funktionen der IDE und .NET, wie z. B. die Test- und Debugging-Werkzeuge, genutzt werden. Um korrekte Analysen zu gewährleisten, hat S# 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 eines Tests durch S# 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

Um ein System testen zu können, muss dieses zunächst 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. Da S# mithilfe von .NET und der Sprache C# entwickelt wurde, können entsprechend zahlreiche Funktionen hiervon zum Entwickeln des Modells genutzt werden. Die Modelle sind dadurch einerseits ganz normale, ausführbare C#-Programme, stellen andererseits aber auch Modelle von realen, oftmals sicherheitskritischen, Systemen dar [9].

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

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

Listing 2.1: Grundlegender Aufbau einer S#-Komponente.

Jede Komponente des Modells muss von `Component` erben, um als S#-Komponente definiert zu sein. Sie kann dabei einen oder mehrere Komponentenfehler enthalten, die bei dieser Komponente auftreten können, im Beispiel in Listing 2.1 ist dies `NodeDeadFault`, der den Ausfall eines Nodes herbeiführt. Ein Komponentenfehler kann dabei temporär (`TransientFault`) oder dauerhaft (`PermanentFault`) sein. Der Effekt eines Komponentenfehlers wird in der entsprechenden Effekt-Klasse definiert, welche von der Hauptklasse (hier `YarnNode`) erbt und mithilfe des Attributs `FaultEffectAttribute` dem dazugehörigen Komponentenfehler zugeordnet wird [3, 9]. Neben den eigentlichen Komponenten enthält jedes Modell zudem eine zentrale `Model`-Klasse, die von `ModelBase` erbt. Sie stellt die zentrale Schnittstelle zwischen S# und dem Modell selbst dar und wird zur Ausführung bzw. der Analyse des Systemmodells durch S# benötigt [10].

Das Systemmodell kann nach der Modellierung vom S#-Compiler kompiliert werden und bindet sich mithilfe der S#-Laufzeitumgebung nach folgendem Schema in die grundlegende Architektur des Testens mit S# ein:

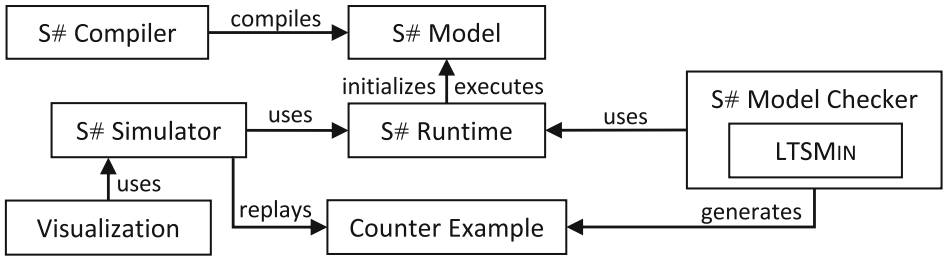

Abbildung 2.1: Grundlegende Architektur eines Tests mit S# (entnommen aus [9])

Zur Ausführung eines S#-Modells gibt es neben der Simulation und dem MC zusätzlich noch die Möglichkeit der Ausführung mithilfe der Deductive Cause-Consequence Analysis (DCCA) (vgl. Unterabschnitt 2.1.2). Alle drei Ausführungswerkzeuge führen auf jeweils ihre Art das entwickelte Modell aus. Dabei werden die Gegenbeispiele dazu genutzt, um

mit deren Hilfe die benötigte Fehlermenge im Modell zu ermitteln, bei der das gesamte System ausfällt [9].

Ein S#-Modell enthält jedoch auch noch weitere Bestandteile, was sich vor allem beim Ansatz des Back-to-Back-Testings mithilfe der DCCA zeigt:

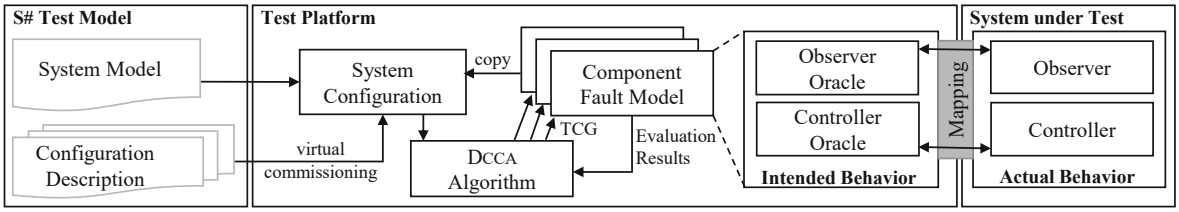

Abbildung 2.2: Ansatz des Back-to-Back-Testings mit S# (entnommen aus [11])

Um solche Tests durchzuführen, benötigt das Modell weitere Komponenten. Der *Controller* dient vor allem dazu, um das System zu steuern, während der *Observer* das System überwacht und dem Controller die benötigten Daten bereitstellt. Mithilfe eines *Oracle*s wird geprüft, ob sich das System im Verlauf des Tests so verhält, wie es erwartet wird. Hierfür werden im Modell *Constraints* definiert, welche die Implementierung der Anforderungen an das SuT im Modell darstellen und bei der Ausführung durch das Oracle validiert werden. Dadurch kann automatisiert ermittelt werden, ob sich das SuT wie erwartet verhält [3, 11].

Für weitere Informationen zum Aufbau eines S#-Modells sei an dieser Stelle auf entsprechende Literatur [3, 9, 11], sowie auf die Dokumentation von S# [12] verwiesen.

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* durch *eigenen* Code im entwickelten Modell gesteuert werden kann. Ein weiterer Nutzen liegt in der Möglichkeit, im ausgeführten Ausführungspfad zeitliche Abläufe zu berücksichtigen, da hier das Modell schrittweise 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 [9].

Das zweite Werkzeug zur Ausführung von Modellen in S# ist der MC. Hierbei kann der in S# enthaltene, oder alternativ *LTSmin*[1] genutzt werden [9, 13]. 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].

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

Ein weiteres, wichtiges Werkzeug von S# ist die DCCA, welche eine vollautomatische und MC-basierte Sicherheitsanalyse ermöglicht (Back-To-Back-Testing). Dabei wird durch die DCCA die Menge der aktivierten Komponentenfehler ermittelt, mit denen das SuT 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 [11]. 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. Hierbei 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 Komponentenfehlers ist also gleichbedeutend mit einem einfachen Fehler im System, welcher die gesamte Funktionsweise des Systems nicht massiv einschränkt und eine weitere 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].

2.2 Apache Hadoop

Apache™Hadoop®ist ein in Java entwickeltes Open-Source-Software-Projekt, welches die Verarbeitung von großen Datenmengen auf einem verteilten System ermöglicht. Hadoop wird federführend von der *Apache Foundation* entwickelt und enthält verschiedene vollständig skalierbare Komponenten [14]. Dadurch wird ermöglicht, ein Hadoop-Cluster auf nur einem einzelnen PC, aber auch verteilt auf zahlreichen Servern auszuführen, auf dem wiederum Anwendungen zum Verarbeiten von großen Datenmengen ausgeführt werden können. Die dem Cluster und den Anwendungen verfügbaren Ressourcen beschränken sich lediglich auf die Summe der verfügbaren Ressourcen aller Hosts, auf denen das Cluster ausgeführt wird. Hadoop besitzt hierzu folgende Kernmodule [14]:

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 (HDFS)
: Verteiltes Dateisystem zum Speichern von Daten auf dem Cluster

Hadoop MapReduce (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 [6]. Die Ein- und Ausgabedaten sind hierbei als *Key-Value*-Paare definiert, die mithilfe des MR-Frameworks verarbeitet werden. Hierbei werden zunächst die eingelesenen Eingabedaten in kleine und dadurch einfach zu verarbeitende Datenmengen aufgeteilt. Die geteilten Daten werden dann in mehreren, parallel ausgeführten Map-Tasks verarbeitet und zwischengespeichert. Die zwischengespeicherten Daten werden anschließend von einem oder mehreren Reduce-Tasks zusammengeführt und in die Ausgabedateien geschrieben. Das Framework besitzt eine hohe Fehlertoleranz, da ein fehlerhafter Task jederzeit neu gestartet werden kann [15, 16]. Zur Speicherung der Ein-, Zwischen- und Ausgabedaten wird im Falle von Hadoop das HDFS genutzt [17]. Für eine ausführliche Beschreibung des MR-Frameworks sei hier auf [15] und [16] verwiesen; eine generelle Betrachtung des MR-Frameworks mit Vorteilen und möglichen Problemen lässt sich in [18] finden.

Da das MR-Framework bzw. seine Implementierung in Hadoop nicht immer optimal genutzt werden kann und in der Praxis daher zum Teil auch zweckentfremdet wurde, wurde in [19] 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 [19].

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

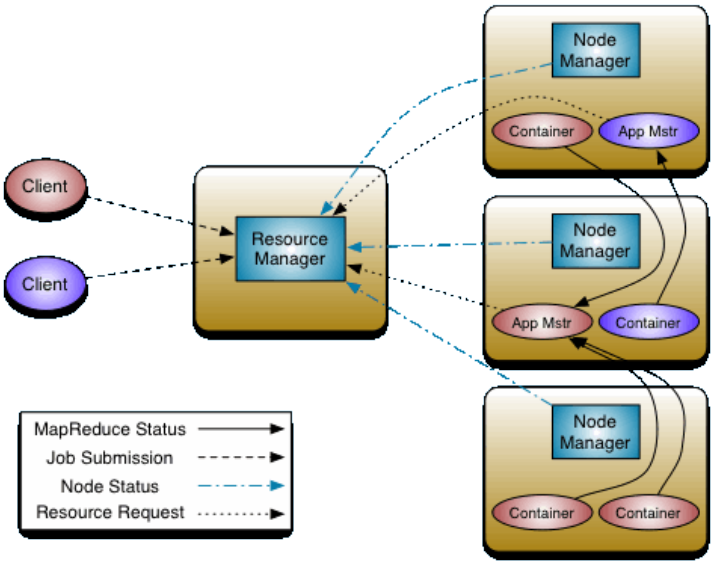

Abbildung 2.3: Architektur des YARN-Frameworks (entnommen aus [20])

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 YARN-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), der das Monitoring der Anwendung und die Kommunikation mit dem RM und NM durchführt und die dafür benötigten Daten bereitstellt [20]. Die eigentliche Ausführung eines Tasks einer Anwendung findet in den bereits erwähnten *Containern* statt, die jeweils einem Attempt zugeordnet sind. Container können auf einem beliebigen Node ausgeführt werden und dienen der Ausführung eines Tasks der Anwendung. Der AppMstr wird hierbei ebenfalls in einem YARN-Container ausgeführt.

Für die Fallstudie in dieser Masterarbeit ist zudem nicht unwesentlich, dass die Kommunikation zwischen den einzelnen Komponenten nicht in allen Fällen in Echtzeit stattfindet. Vor allem das Prüfen des generellen Node-Zustandes durch den RM wird bei einer Standard-Konfiguration in periodischen Abständen von jeweils mehreren Minuten durchgeführt. Erst nachdem ein NM nach mehreren Minuten keine Rückmeldung an den RM sendet, wird der Node als defekt markiert. Ähnlich verhält es sich bei Zustandsabfragen an die AppMstr der Anwendungen [21].

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 als Anwendungshistorie bereitzustellen [22].

Zum Steuern des Clusters bzw. dem Monitoring der mithilfe von YARN ausgeführten Anwendungen stellt Hadoop drei Schnittstellen zur Verfügung. Dies sind eine graphische Weboberfläche, was zugleich auch die wichtigste Schnittstelle darstellt, entsprechende Befehle für die Kommandozeile (engl. *Command-line interface*, kurz CLI) sowie eine REST-API. Während sich die Weboberfläche zur menschlichen Interaktion oder zur Fehlersuche eignet, dienen die CLI-Befehle vor allem zum Steuern des Clusters und die REST-API zur automatisierten Rückgabe der Daten des Clusters zur Nutzung in anderen Programmen [23–26].

Abb. 2.4 zeigt die dem YARN-Framework sehr ähnliche Architektur des HDFS, welche ebenfalls aus einem Controller und mehreren Nodes besteht.

Der *NameNode* dient als Controller für die Verwaltung des Dateisystems und reguliert den Zugriff auf die darauf gespeicherten Daten. Unterstützt wird er vom *Secondary NameNode*, der Teile der internen Datenverwaltung des HDFS durchführt [28]. Die Daten selbst werden in mehrere Blöcke aufgeteilt auf den *DataNodes* gespeichert. Um den Zugriff auf die Daten im Falle eines Node-Ausfalls zu gewährleisten, wird jeder Block auf anderen Nodes repliziert. Dateioperationen (wie Öffnen oder Schließen) werden direkt auf den DataNodes ausgeführt. Sie sind darüber hinaus auch dafür verantwortlich, dass die gespeicherten Daten gelesen und beschrieben werden können [27, 29].

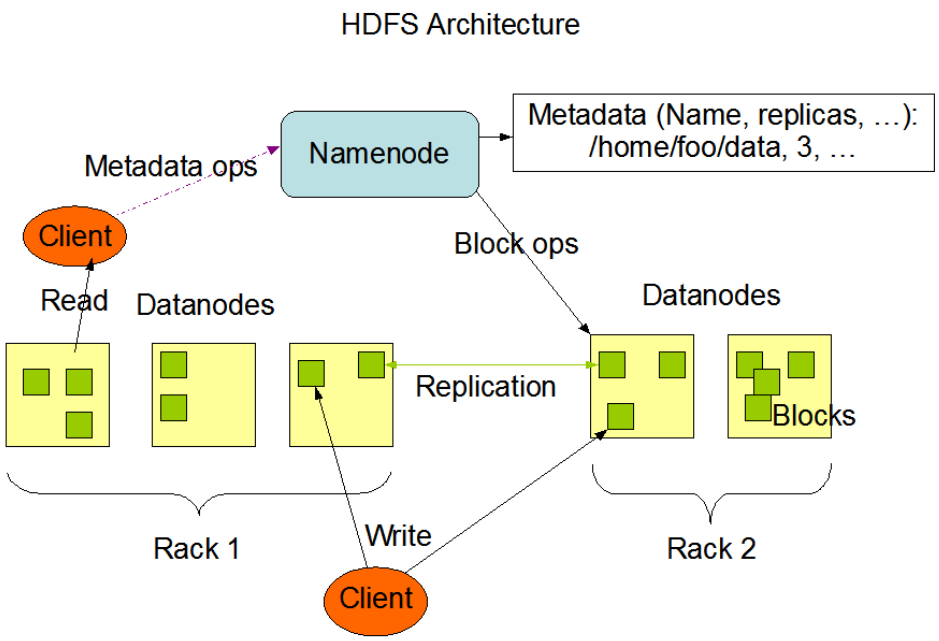

Abbildung 2.4: Architektur des HDFS (entnommen aus [27])

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 ein DataNode als defekt markiert wird [30].

2.3 Adaptive Komponente in Hadoop

Eine normale Hadoop-Installation besitzt keine Komponente zu dynamischen Anpassungen der Einstellungen des Clusters. Um damit Hadoop zu optimieren, müssen die Einstellungen daher immer manuell auf den jeweils benötigten Anwendungstyp angepasst werden. Dazu gibt es u. a. verschiedene Scheduler, den *Fair Scheduler*, welcher alle Anwendungen ausführt und ihnen gleich viele Ressourcen zuteilt, und den *Capacity Scheduler*. Letzterer sorgt dafür, dass nur eine bestimmte Anzahl an Anwendungen pro Benutzer gleichzeitig ausgeführt wird. Dieser teilt ihnen so viele Ressourcen zu, wie benötigt werden bzw. dem Benutzer zur Verfügung stehen. Entwickelt wurde der Capacity Scheduler vor allem für Cluster, die von mehreren Organisationen gemeinsam verwendet werden. Er dient in diesem Kontext vor allem dazu, jeder Organisation eine Mindestmenge an Ressourcen zur Verfügung zu stellen [31].

Für diesen Scheduler wurde von Zhang et al. [7] ein selbstadaptiver Ansatz vorgestellt, welcher im Folgenden genauer erläutert und im weiteren Verlauf dieser Masterarbeit als **Selfbalancing-Komponente** bezeichnet wird.

2.3.1 MARP-Wert

Der Capacity Scheduler besitzt verschiedene Einstellungen, um ihn für das konkrete Cluster anzupassen. So besteht z. B. die Möglichkeit, den verfügbaren Speicher pro YARN-Container festzulegen oder welcher Anteil der verfügbaren Ressourcen durch AppMstr-Container beansprucht werden darf. Vor allem letztere Einstellungsmöglichkeit Namens `maximum-am-resourcepercent` (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 dürfen [31]. Der gesamte, für Anwendungen verfügbare Speicher wird durch den MARP-Wert in zwei Teile aufgeteilt. Während ein Teil des Speichers nur durch AppMstr-Container beansprucht werden darf, wird der andere Teil des Speichers durch alle anderen YARN-Container genutzt. Wird durch den MARP-Wert der erste, der für AppMstr-reservierte 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 YARN-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) [7]:

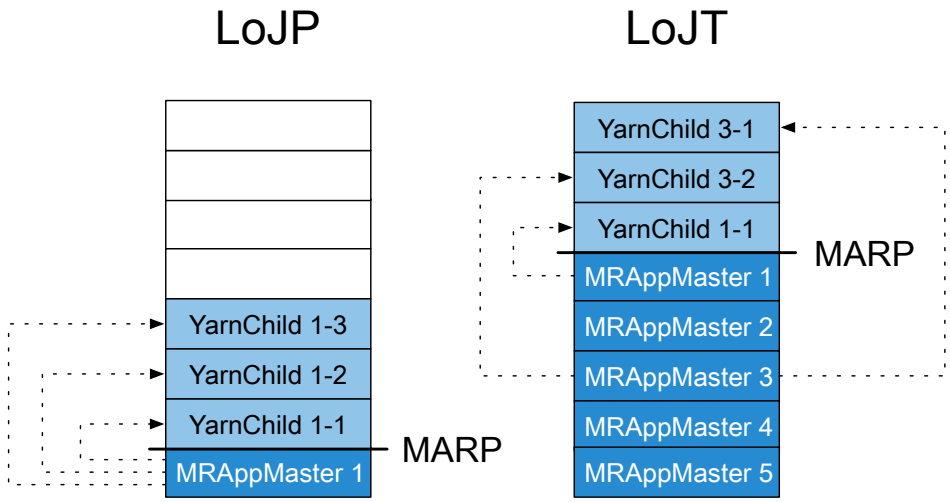

Abbildung 2.5: LoJP und LoJT in Hadoop (entnommen aus [7]). Während beim LoJP sehr viel Speicher für YARN-Container ungenutzt bleibt, können beim LoJT nicht genügend YARN-Container allokiert werden, um die Anwendungen auszuführen.

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 et al. einen Ansatz zur dynamischen Anpassung des MARP-Wertes zur Laufzeit von Hadoop vorgestellt [7]. Die entwickelte Selfbalancing-Komponente passt den MARP-Wert, abhängig

von der Speicherauslastung der ausgeführten Anwendungen, dynamisch zur Laufzeit an. So wird der MARP-Wert, und damit auch die Anzahl der ausführbaren AppMstr, 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 et al. 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 [7].

2.3.2 Analyse der Selfbalancing-Komponente

Da in dieser Fallstudie auch Mutationstests eingesetzt werden, bei denen die Selfbalancing-Komponente entsprechend verändert wird (vgl. Abschnitte 3.1 und 6.2), wurde die Komponente zunächst analysiert. 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[2] genutzt. Der Kalman-Filter wurde von Kálmán erstmals in [32] beschrieben und wird genutzt, um „aus verrauschten und teils redundanten Messungen die Zustände und Parameter des Systems zu schätzen“ [33]. Der Filter lässt sich aufgrund seines Aufbaus zudem auch für Echtzeitanwendungen nutzen [33]. Als einfaches Anwendungsbeispiel hierfür ist in [33] die Apollo-Mondlandefähre genannt, Strukov nutzte ihn in [34] 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. [35–37], verwiesen.

Die drei Shell-Scripte der Selfbalacing-Komponente dienen zur Interaktion zwischen der Komponente und dem Cluster. Die beiden zuletzt genannten Scripte werden von den beiden Monitor-Klassen sekündlich gestartet und ermitteln, basierend auf den

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

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 Werte werden im Anschluss wiederum sekündlich vom `Controller` der Selfbalancing-Komponente ausgelesen und mithilfe des Kalman-Filters bereinigt. Anschließend werden die Algorithmen [7] zum Berechnen des neuen MARP-Wertes ausgeführt.

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

2.4 Plattform Hadoop-Benchmark

Zhang et al. haben im Rahmen ihrer gesamten Forschungsarbeit an der Selfbalancing-Komponente darüber hinaus auch die Open-Source-Plattform Hadoop-Benchmark[3] entwickelt. 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 werden die Virtualisierungs-Software Docker und *Docker Machine* genutzt. Docker-Machine startet hierbei mithilfe von VirtualBox[4] eine VM, auf der mit *Boot2Docker* eine einfache Linux-Distribution zum Ausführen von Docker-Containern installiert ist [38]. Auf dieser VM wird wiederum der Hadoop-Node in einem Docker-Container ausgeführt. Abb. 2.6 zeigt den grundlegenden Aufbau der Plattform, bei der die einzelnen Nodes mithilfe eines *Docker Swarms*[5] verbunden werden.

Mit *Graphite*[6] ist zudem ein Monitoring-Tool enthalten, mit dem die Systemwerte, wie CPU- oder Speicher-Auslastung des Clusters, überwacht werden können. Jeder Hadoop-Container enthält dazu das Tool *collectd*[7], was das Monitoring des Containers auf Systemebene übernimmt und die Daten an Graphite übermittelt.

Da mithilfe der Plattform auch unterschiedliche Hadoop-Konfigurationen ausgeführt werden können, ist die Plattform in mehrere Szenarien unterteilt. Jedes Szenario stellt eine vollständig anpassbare Hadoop-Konfiguration dar. Ein Szenario enthält dafür eine *Dockerfile*, aus der die Docker-Images und -Container erstellt werden, weitere für

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

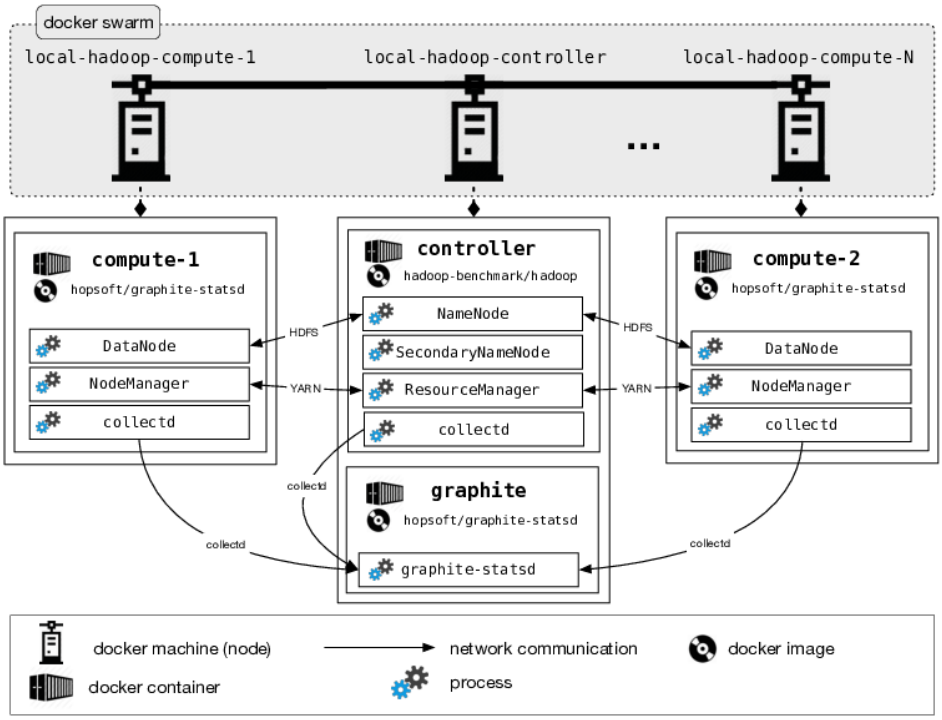

Abbildung 2.6: High-Level-Architektur von Hadoop-Benchmark (entnommen aus [39])

Hadoop benötigte Daten und Einstellungen sowie dazugehörige generelle Einstellungen des Szenarios. Die Plattform enthält bereits mehrere Szenarien, u. a. Hadoop in einer Standardinstallation und ein darauf basierendes Szenario mit der Selfbalancing-Komponente. Aufgrund eines der Kernkonzepte von Docker, wonach Docker-Images auf einem passenden, bereits vorhandenen Image aufbauen können bzw. sollten [40], können neue Szenarien, basierend auf bereits vorhandenen, entwickelt werden.

Zum Starten des Clusters ist zudem ein Script enthalten, welches, basierend auf dem zu nutzenden Szenario das Cluster, in der entsprechenden Konfiguration startet.

In der Plattform Hadoop-Benchmark sind auch bereit folgende Benchmarks integriert:

- Hadoop-Mapreduce-Examples
- Intel HiBench[8]
- Statistical Workload Injector for Mapreduce (SWIM) [9]

Die Benchmarks werden ebenfalls mithilfe der in der Plattform enthaltenen Scripte gestartet. Hierbei besitzt jede Benchmark ein eigenes Start-Script, das die Benchmark in einem Docker-Container startet und die entsprechende Anwendung so dem Cluster zur Ausführung übergibt. Die Ausführung einer Anwendung mithilfe des entsprechenden Start-Scriptes sowie das Beenden einer Anwendung ist in Anhang A beispielhaft aufgezeigt.

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

[8] https://github.com/intel-hadoop/HiBench/
[9] 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. 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 [8] publiziert.

3.1 Grundlegender Versuchsaufbau

Neben den Anforderungen an Hadoop und das gesamte Testsystem muss auch der grundlegende Versuchsaufbau und das SuT definiert werden. Im Grunde wird, wie bereits in der Einleitung erwähnt, Hadoop mithilfe des S#-Frameworks nachgebildet und dieses Modell mit einem realen Cluster verbunden. In diesem Cluster sollen anhand des Modells unterschiedliche Komponentenfehler injiziert und repariert, sowie unterschiedliche Benchmarks gestartet werden. Hierbei soll nicht nur das Verhalten von Hadoop selbst analysiert werden, sondern auch das der von Zhang et al. 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 [8], da dafür das gleiche Testsystem genutzt wurde wie für diese Fallstudie, in deren Rahmen es entwickelt wurde. Zur Ausführung der Tests soll der von S# bereitgestellte Simulator genutzt werden, da hierbei nur ein Ausführungspfad mit der Berücksichtigung von zeitlichen Abfolgen ausgeführt werden kann und Komponentenfehler gemäß der im Modell implementierten Definitionen aktiviert und deaktiviert werden können.

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. Das Modell soll dabei immer den aktuellen Status des realen Clusters repräsentieren, weshalb regelmäßig alle benötigten Daten des realen Clusters ausgelesen und im Modell gespeichert werden müssen.

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 der Anwendungswechsel definiert sind, soll während der Ausführung eines Testfalls zufällig eine nachfolgende Anwendung ausgewählt werden. Hierbei soll berücksichtigt werden, dass einige Anwendungen die Eingabedaten für andere Anwendungen generieren können.

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 konvertieren.

Das SuT selbst stellt das reale Hadoop-Cluster dar, das mithilfe 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 ein oder mehrere Mutanten in der Selfbalancing-Komponente implementiert werden, das Testsystem geprüft werden.

Dieser Versuchsaufbau soll mithilfe eines dafür entwickelten *Oracles* bereits während der Ausführung automatisiert 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 dezentral in den jeweils betroffenen 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 Testmodells mit seinen Komponenten ist im Kapitel 4 beschrieben. Die Auswahl der im Testsystem verwendeten Benchmarks und deren Implementierung 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, inwieweit eine Testautomatisierung möglich ist. Hierfür werden die Anforderun-

gen, sofern möglich, in Form von Constraints ebenfalls im Modell implementiert und während der Ausführung durch das Oracle validiert.

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 [8] 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 Anforderungen an das SuT [8]:

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 den 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 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, die Ausführung aber nicht weiter 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 gewonnenen 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 unabhängige 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 abhängige Variablen ermittelt:

- Anzahl der tatsächlich ausgeführten Testfälle
- Aktivierte 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. Unterabschnitt 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 Unterabschnitt 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 Unterabschnitt 3.3.1 definierten unabhängigen Variablen, die durch weitere Angaben ergänzt werden:

- Anzahl genutzter Hosts
- Basisanzahl der Nodes
- Anzahl simulierter Clients
- Basisseed für Zufallsgeneratoren
- Anzahl auszuführender Testfälle
- Mindestdauer für einen Testfall
- Nutzung einer oder mehrerer 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 Basisseeds der Testkonfiguration. Zwar werden durch das Transitionssystem, basierend auf den dort definierten Wahrscheinlichkeiten, 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 eines mathematischen Algorithmus Zufallszahlen berechnet [41]. Obwohl standardmäßig ein zeitbasierter Seed als Startwert für den Algorithmus genutzt wird [41], kann durch die Angabe eines spezifischen Seeds die Wiederausführbarkeit der Tests sichergestellt werden.

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. fehlschlagen 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 von zuvor ausgeführten Anwendungen zur Generierung der Eingabedaten abhängig sind. Ebenfalls definiert werden müssen die Tests, die als Mutationstests durchgeführt werden sollen. Hierbei müssen auch die Mutationen selbst definiert werden, die vom Testsystem erkannt werden sollen.

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 gespeichert werden. Alle relevanten Daten werden hierzu während der Ausführung der Testfälle in

einer Log-Datei gespeichert. Zur Unterscheidung von einzelnen Ausführungen werden die Daten klar strukturiert. Beim Start eines Tests sollen daher zunächst einige generelle Daten ausgegeben werden:

- Basis-Seed für die Zufallsgeneratoren
- Wahrscheinlichkeiten 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 oder diese während der Ausführung eines Tests generiert werden
- Mindestdauer für einen Testfall
- 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 Clients
  - 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 der Ausgaben sind in Unterabschnitt 6.1.4 erläutert, die zur Ausgabe der generellen Testdaten in Abschnitt 6.4. Ein Beispiel für den Inhalt einer Logdatei eines Tests findet sich in Anhang E.

4 Entwicklung des Testmodells

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

Da für die in [8] durchgeführte Fallstudie das in diesem Kapitel beschriebene Testmodell genutzt wurde, wurden entsprechende Teile der Beiträge dieses Kapitels dort bereits publiziert.

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:

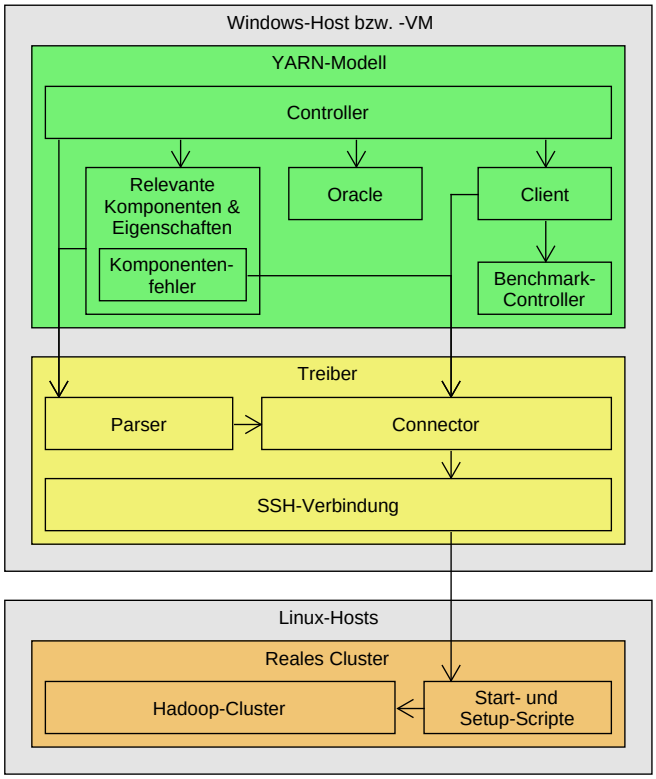

Abbildung 4.1: Grundlegende Architektur des Testmodells

Das YARN-Modell stellt die oberste Schicht des Testmodells dar. Es bildet das Kernstück dieser Fallstudie, da dieses Modell, mit den hierin abgebildeten YARN-Komponenten, den 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
: Stellen die bei den Tests zu injizierenden Komponentenfehler der jeweiligen YARN-Komponenten dar.

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 Monitorings 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.

Alle Informationen des Monitorings, der Validierung der Constraints oder Debug-Informationen werden während der Ausführung im Programmlog gespeichert (vgl. Unterabschnitte 4.2.5 und 4.2.9). Zu Analysezwecken im Fehlerfall wird zudem jede Kommunikation der SSH-Verbindungen in einem eigenen Log, dem SSH-Log, abgespeichert (vgl. Unterabschnitt 4.3.4). Durchgeführt wird dies mithilfe des Frameworks log4net[1], mit denen an den entsprechenden Stellen im Programm das Logging durchgeführt wird.

Die Implementierung des YARN-Modells wird in den Abschnitten 4.2 und 5.3 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 und als Controller im Rahmen des Testens mit S# dient. Einen Überblick über den Aufbau des implementierten YARN-Modells gibt das in Abb. 4.2 dargestellte Klassendiagramm.

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 YARN-Container (`YarnAppContainer`). Der Client stellt somit auch die den relevanten YARN-Komponenten übergeordnete Komponente dar.

Zunächst wird im Folgenden die dem Modell übergeordnete zentrale Klasse `Model` erläutert, danach die relevanten YARN-Komponenten mit ihren Komponentenfehlern, anschließend die anderen Komponenten des YARN-Modells. Der Aufbau des Benchmark-Controllers wird in Abschnitt 5.3 erläutert.

4.2.1 Die Klassen Model und ModelSettings

Die Klasse `Model` stellt die zentrale Schnittstelle des Testsystems mit S# dar (vgl. Unterabschnitt 2.1.1), 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

[1] `https://logging.apache.org/log4net/`

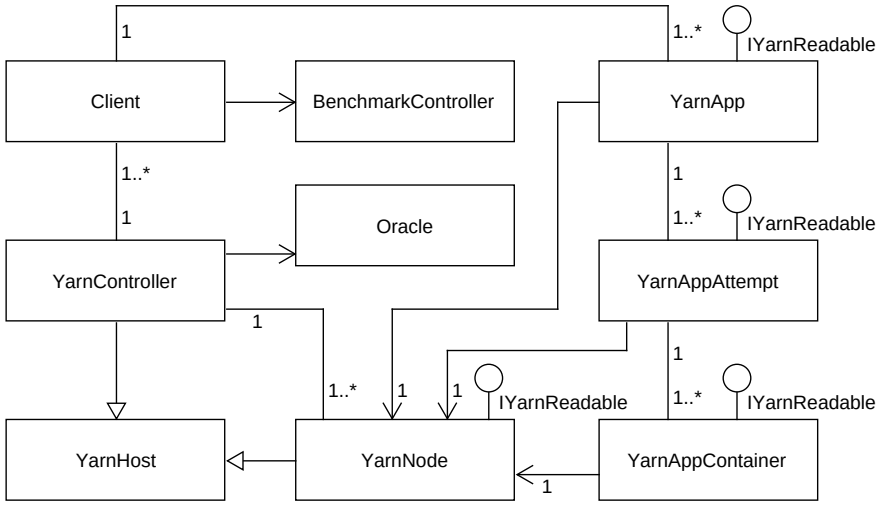

Abbildung 4.2: Grundlegender Aufbau des YARN-Modells. Assoziationen und weitere Verbindungen zum Treiber und S# sind hier aus Gründen der Übersichtlichkeit nicht dargestellt.

Modells genutzt. Hierfür werden alle Komponenten, wie Clients oder Anwendungen, zusätzlich zur Struktur innerhalb des Modells auch in `Model` gespeichert, welche zur Interaktion mit S# entsprechenden Attributen versehen sind (vgl. Unterabschnitt 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 YARN-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 YARN-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 des Treibers. Aufgrund der Einschränkungen von S# im Umgang mit dem Treiber (vgl. Unterabschnitt 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 Konstanten für das Modell bzw. Testsystem. Gespeichert und bereitgestellt werden hier Daten, wie z. B. die Zugangsdaten der SSH-Verbindungen oder der genutzte `HostMode` (vgl. Unterabschnitte 4.3.3 und 4.4.2). Die

`ModelSettings`-Klasse ist dabei zur Verwaltung des `HostMode`s zuständig und dafür verantwortlich, die vom `HostMode` abhängigen Pfade und Adressen dem Treiber bereit zu stellen. Dies betrifft z. B. den Pfad zum benötigten Setup-Script (vgl. Unterabschnitt 4.4.3) oder die zur Nutzung der REST-API benötigten Adressen. Eine Besonderheit bildet hierbei die Eigenschaft `HttpUrl` der `YarnNode`s, welche beim Initialisieren des Modells durch die `ModelSettings` die zum `HostMode` passende Adresse erhält (vgl. Unterabschnitte 4.2.2 und 4.4.2).

4.2.2 Relevante YARN-Komponenten

Die vier implementierten, relevanten YARN-Komponenten sind die Anwendungen, ihre Attempts und Container sowie die Nodes des Clusters. Diese vier implementierten YARN-Komponenten stellen das zum Testen mit S# benötigte Modell des SuT dar (vgl. Unterabschnitt 2.1.1). Obwohl die YARN-Container in dieser Fallstudie nicht benötigt werden, waren sie für die Fallstudie [8] notwendig, welche ebenfalls mit dem hier beschriebenen Modell durchgeführt wurde.

Eine Übersicht über die Implementierung der für diese Fallstudie relevanten YARN-Komponenten gibt folgendes Klassendiagramm:

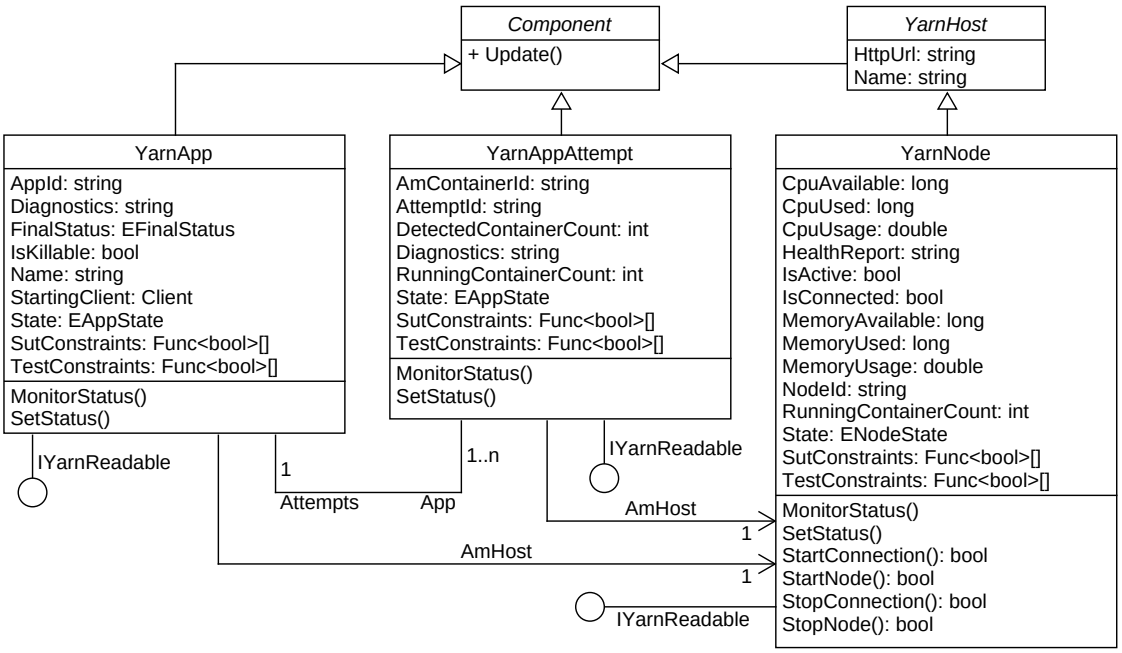

Abbildung 4.3: Für die Fallstudie relevante, implementierte YARN-Komponenten, hier dargestellt mit den wichtigsten Eigenschaften und Methoden. Dies sind alle für die spätere Durchführung und zur Ausgabe des Zustandes (vgl. Unterabschnitt 3.3.3) wichtigen Eigenschaften und Methoden. Aus Gründen der Übersichtlichkeit 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 wird und somit vorzeitig abgebrochen werden kann. Die Eigenschaften `Diagnostics` bzw. `HealthReport` stellen von Hadoop zur Verfügung gestellte, weitere Diagnostik-Informationen bei Fehlern dar. Die `State`-Eigenschaften bzw. `FinalState` speichern die von Hadoop angegebenen Zustände der YARN-Komponenten. Hierfür wurden entsprechende Enumerationen, basierend auf den in [25] 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 YARN-Container beim Monitoring im aktuellen Testfall (`RunningContainerCount`) bzw. für alle bisher ausgeführten Testfälle kumuliert (`DetectedContainerCount`) erkannt wurden. Für die Tests [8] 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 YARN-Containern, sofern benötigt. 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 Berechnung, ob ein Komponentenfehler aktiviert oder deaktiviert wird. Die durch die Basisklasse `YarnHost` bereitgestellte Eigenschaft `HttpUrl` beschreibt die für die Nutzung der REST-API benötigte Adresse des Nodes (vgl. Unterabschnitt 4.3.3). Da die Adresse vom `HostMode` abhängig ist, wird sie mithilfe der `ModelSettings` beim initialisieren des Modells entsprechend festgelegt (vgl. Unterabschnitte 4.2.1 und 4.4.2).

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 (vgl. Unterabschnitt 4.2.5). Die Start- und Stop-Methoden sowie die Eigenschaften `IsActive` und `IsConnected` in `YarnNode` dienen zur Identifikation und Injizierung bzw. Reparieren der Komponentenfehlern (vgl. Unterabschnitt 4.2.3)

Neben den dargestellten, relevanten Eigenschaften und Methoden wurden zahlreiche weitere implementiert, welche einerseits zur Vollständigkeit der implementierten Komponenten für die Tests [8], 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) u. U. 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:

```
public char[] AppIdActual { get; }

[NonSerializable]
public string AppId
{
  get { return ModelUtilities.GetCharArrayAsString(AppIdActual); }
  set { ModelUtilities.SetCharArrayOnString(AppIdActual, value); }
}
```

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. Das Attribut `NonSerializable` definiert, dass die Eigenschaft bei der Ausführung der Simulation von S# ignoriert wird.

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.

4.2.3 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 aktiviert wird, wird durch die Effekt-Klasse die jeweilige Stop-Methode ausgeführt und so der Fehler injiziert:

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

```

```
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
30public 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}",
38       IsConsoleOut);
39    return !CheckNodeRunning(id);
40  }
41}
```

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, um einen defekten Node zu reparieren. Eine Besonderheit bildet hierbei die Reparatur des Komponentenfehlers `NodeConnectionErrorFault`, welche den Node komplett neu startet, da sich der Node ansonsten möglicherweise nicht wieder mit dem RM verbinden kann.

Da beide Fehler zudem auch gleichzeitig aktiviert werden können, wurde dem Komponentenfehler `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 zusätzlich auch vom Netzwerk getrennt.

Zur einfachen Identifikation der aktiven Komponentenfehler innerhalb des YARN-Modells dienen die beiden Eigenschaften `IsActive` und `IsConnected`. In Listing 4.2 wird bereits die Auswirkung der beiden Eigenschaften 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.

4.2.4 Aktivierung von Komponentenfehlern

Die Aktivierung von Komponentenfehlern ist von folgenden Parametern abhängig:

- Auslastung des Nodes im vorhergehenden Testfall
- Generelle Wahrscheinlichkeit zur Fehleraktivierung
- Einer Zufallszahl

Der hierfür benötigte Zufallsgenerator wird mithilfe des spezifizierten Basisseeds initialisiert (vgl. Unterabschnitt 3.3.2). Da der Zufallsgenerator ein statischer Generator ist, werden über einen Test bei der gleichen Anzahl an möglichen Komponentenfehlern immer die gleichen Werte zurückgegeben, nicht jedoch zwingend für einen bestimmten Komponentenfehler eines Nodes. Die Komponentenfehler werden damit nicht automatisiert durch S# aktiviert bzw. injiziert (vgl. Unterabschnitt 2.1.2), sondern mithilfe von speziell hierfür entwickelten Klassen und Methoden.

Um bei der Ausführung eines Testfalls bestimmen zu können, ob ein Komponentenfehler aktiviert wird, sind alle Komponentenfehler mit dem Attribut `NodeFault` markiert. Dieses Attribut enthält die für den Komponentenfehler benötigten Berechnungen, um über die Aktivierung des Komponentenfehlers zu entscheiden:

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

```
14 return probability < randomValue;
```

Listing 4.3: Berechnung der Aktivierung von Komponentenfehlern (zusammengefasst).

Die Entscheidung zur Deaktivierung eines Komponentenfehlers verhält sich analog. Anstatt der generellen Aktivierungswahrscheinlichkeit wird hierbei die generelle Wahrscheinlichkeit zur Deaktivierung der Komponentenfehler genutzt. Außerdem spielt bei der Deaktivierung die Auslastung des Nodes zum Zeitpunkt der Aktivierung eine Rolle, welche hierzu in der Eigenschaft `NodeUsageOnActivation` des Attributs entsprechend gespeichert und zur Deaktivierung genutzt wird.

4.2.5 Interface IYarnReadable und Monitoring

Das Interface `IYarnReadable` ist das zentrale Erkennungsmerkmal der im Modell abgebildeten und implementierten YARN-Komponenten. Es dient zum einen zur Identifikation 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. Unterabschnitt 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 hierfür zwei unterschiedliche Schnittstellen bietet: Die Rückgabe der Daten durch die CLI oder mithilfe der REST-API. Ausführlichere Informationen in diesem Kontext sind in Abschnitt 4.3 zu finden. Bei der Nutzung der CLI 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 4.4: 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.

Eine Besonderheit bildet das Monitoring der YARN-Container. Da eine Anwendung sehr schnell eine sehr hohe Anzahl an Containern allokiert und jede Container-Instanz im Modell Speicherplatz benötigt, werden nur die Daten der während des Monitoring ausgeführten Container ermittelt. Bei Anwendungen und Attempts werden dagegen auch die Daten von bereits beendeten Anwendungen bzw. Attempts gespeichert.

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.

4.2.6 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 und werden in Form von Constraints im Modell implementiert (vgl. Unterabschnitt 2.1.1 und Abschnitte 3.1 und 3.2). 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>[]`:

```
public Func<bool>[] SutConstraints => new Func<bool>[]
{
  // task will be completed if not canceled
  () =>
  {
    if(FinalStatus != EFinalStatus.FAILED)
      return true;
    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 4.5: Definition der Constraints in `YarnApp` (gekürzt)

Die Constraints werden im Anschluss an das Monitoring vom Oracle validiert (vgl. Unterabschnitt 4.2.9). 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.

4.2.7 Implementierung des Clients

Der Client im YARN-Modell simuliert einen Client und dient somit zum Starten der Benchmarks. 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:

```
1  public void UpdateBenchmark()
2  {
3    var benchChanged = BenchController.ChangeBenchmark();
4
5    if(benchChanged)
6    {
7      StopCurrentBenchmark();
8      StartBenchmark(BenchController.CurrentBenchmark);
9    }
10 }
11
12 public void StartBenchmark(Benchmark benchmark)
13 {
14   if(benchmark.HasOutputDir)
15     SubmittingConnector.RemoveHdfsDir(benchmark.GetOutputDir(ClientDir
         ));
16   var appId = SubmittingConnector.StartApplicationAsyncTillId(
       benchmark.GetStartCmd(ClientDir));
17   // save application id
18 }
```

Listing 4.6: Auswahl und Start des nachfolgenden Benchmarks (gekürzt). Die Methode `ChangeBenchmark()` des Benchmark-Controllers wird in Unterabschnitt 5.3.2 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 zunächst gelöscht werden. Um mehrere Clients unabhängig voneinander nutzen zu können, besitzt jeder Client zudem ein spezifisches Verzeichnis, das seiner

eigenen Client-ID entspricht. Die Ein- und Ausgabedaten werden dadurch nur im Verzeichnis des jeweiligen Clients gespeichert.

Wenn eine Anwendung gestartet wurde, wird diese zunächst synchron ausgeführt, bis von der gestarteten Anwendung ihre vom Cluster zugewiesene Anwendungs-ID zurückgegeben wird. Diese ID wird vom Client abgespeichert und wird dafür benötigt, um in späteren Testfällen die Anwendung wieder beenden zu können. Zudem wird auch eine noch leere `YarnApp`-Instanz ermittelt und dieser die ID ebenfalls zugewiesen, um diese analog zu den anderen YARN-Komponenten zum Speichern der Daten der Anwendung 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.

4.2.8 Implementierung des Controllers

Der Controller repräsentiert zum Einen den RM des Hadoop-Clusters, weshalb er genauso wie die modellierten Nodes von `YarnHost` erbt. Primär dient er jedoch als Controller zum Testen mit S# (vgl. Unterabschnitt 2.1.1). Er steuert daher den wesentlichen Ablauf eines Testfalls und ist die einzige Komponente des YARN-Modells, die 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 4.7: `Update()`-Methode des Controllers (gekürzt). Eine ausführliche Beschreibung des Ablaufs der Ausführung eines Testfalls findet sich in Unterabschnitt 6.1.4.

Nach einem ersten Monitoring des MARP-Wertes wird durch den Controller sichergestellt, dass jeder simulierte Client eine Anwendung startet, sofern vom Benchmark-

Controller hierbei eine neue Anwendung ausgewählt wird (vgl. Unterabschnitt 4.2.7 und Abschnitt 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 YARN-Container, sofern benötigt. Dabei wird der MARP-Wert hier erneut ausgelesen, da er sich abhängig von den ausgeführten Anwendungen jederzeit verändern kann. 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.

4.2.9 Implementierung des Oracles

Das Oracle dient zur automatisierten Validierung der in Abschnitt 3.2 definierten Anforderungen durch das Testsystem (vgl. Unterabschnitt 2.1.1). Hierzu werden die Constraints aller YARN-Komponenten und des Controllers validiert, jeweils getrennt nach Constraints für das SuT und das gesamte Testsystem, und geprüft, 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 4.8: Validieren der Constraints durch das Oracle. Die zu validierenden Constraints werden im Parameter `constraints` übergeben, der Parameter `constraintType` dient zu statistischen Zwecken in `CountCheck()`.

Bei der Prüfung, ob eine weitere Rekonfiguration möglich ist, wird geprüft, ob alle Nodes im Cluster defekt sind bzw. dies beim Monitoring des realen Clusters erkannt wurde. Wenn dies der Fall ist, wird die weitere Testausführung gemäß Unterabschnitt 3.2.2 abgebrochen, indem eine `Exception` ausgelöst wird:

```
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 4.9: Prüfung nach der Möglichkeit weiterer Rekonfigurationen

Die bereits in Listing 4.8 gezeigte Methode `CountCheck()` dient zu statistischen Zwecken. Hierbei wird getrennt nach Constraint-Typ (der funktionalen Anforderungen an das SuT oder aller Anforderungen an das Testsystem) gezählt, wie viele Constraints insgesamt validiert, und wie viele davon verletzt wurden. Die jeweiligen Werte können beim Abschluss einer Testausführung entsprechend ausgegeben werden.

4.3 Entwicklung des Treibers

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 CLI 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 CLI-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 YARN-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 CLI 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 CLI 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 ausgegeben werden würden. Ausgenommen sind hiervon Anwendungen, bei denen die Nutzung des TLS keine weiteren Daten von Hadoop zurückgegeben werden[22, 24–26]. 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 werden bei der Ausführung des YARN-Modells durch den S#-Simulator weitestgehend ignoriert. Dies ist 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 (vgl. 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 ebenfalls nur zur Initialisierung mit dem konkreten Connector interagiert, während zum Monitoring das Connector-Interface 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

Der Parser dient dazu, die von Hadoop zurückgegebenen Daten der YARN-Komponenten zu konvertieren, sodass sie im Modell gespeichert werden können. Zur Datenhaltung innerhalb der Parser-Komponente des Treibers wurden hierfür entsprechende Klassen entwickelt, welche die von `IParsedComponent` abgeleiteten Interfaces implementieren. Dadurch sind die Datenhaltungs-Klassen dafür geeignet, die Daten an das Modell zu übergeben. Eine Besonderheit bilden hierbei die ausführenden Nodes der Container bzw. AppMstr-Container. Während bei den anderen YARN-Komponenten lediglich die jeweiligen IDs zurückgegeben werden, werden bei den ausführenden Nodes direkt ihre korrespondierenden Instanzen im Modell zurückgegeben.

Der grundlegende Ablauf der Initialisierung und dem Abrufen und Konvertieren der Daten ist bei beiden implementierten Parsern gleich. Beim Initialisieren des Parsers

wird zunächst der benötigte, passende Connector initialisiert, sofern dieser noch nicht initialisiert wurde. Beim Abrufen und Konvertieren der Daten werden zunächst die je nach Komponente benötigten IDs von weiteren YARN-Komponenten ermittelt, bevor die Rohdaten durch den Connector ermittelt werden. Die Rohdaten werden anschließend konvertiert und in der entsprechenden Datenhaltungs-Klasse gespeichert, welche mithilfe der entsprechenden von `IParsedComponent` abgeleiteten Interfaces an das Modell zurückgegeben werden.

Obwohl in dieser Fallstudie das Monitoring ausschließlich mithilfe der wesentlich schnelleren REST-API durchgeführt wird, wurde auch ein Parser zur Nutzung der CLI-Schnittstelle entwickelt. Dies liegt darin begründet, dass Hadoop diese beiden Schnittstellen zum Monitoring der Anwendungen zur Verfügung stellt.

Implementierung des CmdParsers

Der erste der beiden entwickelten Parser ist der `CmdParser` zum Monitoring mithilfe von CLI-Befehlen. Zum Auslesen der Daten selbst nutzt der `CmdParser` den dazugehörigen `CmdConnector` (vgl. Unterabschnitt 4.3.3), mit dem zum Auslesen der Daten die entsprechenden CLI-Befehle ausgeführt werden. Die hierbei zurückgegebenen Daten sind im Vergleich zu den von der REST-API zurückgegebenen im Umfang deutlich reduziert, in den jeweiligen Übersichten der Subkomponenten auf das notwendigste beschränkt. Im Gegenzug zur REST-API werden hier die Daten des RM und TLS kombiniert ausgegeben. Eine kurze Übersicht über einige Befehle und deren Ausgaben ist im Anhang A zu finden.

Ausgewertet werden die von Hadoop zurückgegebenen Daten mithilfe von Regular Expressions (Regex). Da das Ausgabeformat jeweils in Listenform oder als ausführlicher Report immer das gleiche Format aufweist, wurden hierfür zwei generische Regex-Pattern entwickelt, welche zur Auswertung fast aller Daten ausreichend sind:

```
1 Regex _GenericListRegex = new Regex(@"\s*([^\t]+)");
2 Regex _GenericDetailsRegex =
3    new Regex(@"\t(.+)\s:\s([^\t]*)[\n\r]", RegexOptions.Multiline);
```

Listing 4.10: Implementierte Regex-Pattern des `CmdParser`s

Bei zurückgegebenen Listen müssen diese zur Auswertung zunächst zeilenweise getrennt werden, was bei einzelnen Reports nicht nötig ist. Dies wurde aus dem Grund so umgesetzt, da in Listen jede zurückgegebene Zeile eine eigene Komponente darstellt, wogegen ein Report aus mehreren Zeilen besteht. Danach können durch die Reihenfolge der jeweiligen Regex-Matchgruppen die jeweiligen Daten der Komponente den entsprechenden Eigenschaften der Datenhaltungsklassen zugeordnet werden.

Eine Besonderheit bildet das Konvertieren der Zeitstempel. Diese werden von Hadoop meist in Form der Millisekunden seit dem 1. Januar 1970 00:00:00 Uhr, der Java-Epoche, zurückgegeben [22, 25, 26, 42]. In der Liste der ausgeführten Container eines Att-

empts wird der Zeitstempel stattdessen im Format `ddd MMM dd HH:mm:ss zz00 yyyy` ausgegeben, was z. B. dem Zeitstempel `Fri Jan 05 11:08:16 +0000 2018` entspricht. Weiterführende Informationen zur Formatierung von Zeitstempeln in .NET finden sich in [43], zur Zeitberechnung in Java u. a. in [42]. Der von Hadoop zurückgegebene Zeitstempel muss in beiden Fällen zunächst in ein .NET-kompatibles Format umgewandelt werden. Dies geschieht mithilfe der Methode `ParseJavaTimestamp()`, für die mehrere Überladungen implementiert wurden:

```
1  public static DateTime ParseJavaTimestamp(long javaMillis)
2  {
3    if(javaMillis < 1)
4    return DateTime.MinValue;
5    var javaTimeUtc = new DateTime(1970, 1, 1, 0, 0, 0,
6       DateTimeKind.Utc).AddMilliseconds(javaMillis);
7    return javaTimeUtc.ToLocalTime();
8  }
9
10 public static DateTime ParseJavaTimestamp(string value,
11    string format, CultureInfo culture = null)
12 {
13   culture = culture ?? new CultureInfo("en-US");
14   DateTime time;
15   DateTime.TryParseExact(value, format, culture,
16      DateTimeStyles.AssumeUniversal, out time);
17   return time;
18 }
```

Listing 4.11: Überladungen der Methode `ParseJavaTimestamp()`. Es steht zudem eine weitere Überladung zur Verfügung, um den Timestamp in Form der Millisekunden seit 1970 als `string` zu übergeben. Dabei wird der `string` in einen `long` konvertiert und anschließend die erste hier gezeigte Überladung aufgerufen.

Die hierbei zurückgegebenen `DateTime`-Instanzen werden anschließend zum Speichern der Zeitstempel genutzt.

Die Speicherung und Übergabe der ausführenden Nodes des AppMstr oder der Container an das Modell geschieht direkt als entsprechende Node-Instanz innerhalb des Modells. Hierfür wird die ID bzw. die URL des Nodes genutzt, um die korrespondierende Instanz im YARN-Modell zu ermitteln und zu speichern.

Implementierung des RestParsers

Der zweite entwickelte Parser ist der `RestParser`. Er dient dazu, um die mithilfe der REST-API zurückgegebenen Daten auszuwerten und an das YARN-Modell zu übergeben. Zum Auslesen der Daten aus dem Cluster wurde hierfür der `RestConnector` entwickelt (vgl. Unterabschnitt 4.3.3). Die REST-API besitzt, auch im Vergleich zur

CLI-Schnittstelle, einige Besonderheiten, auf die bei der Implementierung geachtet werden musste [22, 24–26]:

- Die zurückgegebenen Daten sind deutlich umfangreicher als bei der CLI-Schnittstelle
- Die Daten können im XML- oder JSON-Format zurückgegebenen werden
- Attempts können nur als Liste aller Attempts einer Anwendung zurückgegeben werden
- Daten zu Containern können nur durch die NM der ausführenden Nodes ermittelt werden
- Die Adressen und Pfade von RM, NM und TLS unterscheiden sich
- Die Daten von RM und NM sind immer umfangreicher als die des TLS
- Der TLS enthält für Attempts und YARN-Container jedoch zusätzliche Daten
- Es werden bei Listen und Reports immer die gleichen Objekte der YARN-Komponenten zurückgegebenen

Da die REST-API zwei Ausgabeformate unterstützt, wurde der `RestParser` aufgrund der kleineren Datenmengen und des übersichtlicheren Datenformats zur Nutzung des JSON-Formats entwickelt. Einige Beispiele für die entsprechenden Pfade der REST-API sowie deren Ausgaben im JSON-Format sind in Anhang B zu finden.

Die Auswertung der Daten wird beim `RestParser` nicht mit Regex, sondern mithilfe des *Json.NET*-Frameworks[2] durchgeführt. Hierfür wurden neben den bestehenden Datenhaltungsklassen noch weitere Hilfsklassen entwickelt, welche das Ausgabeformat der REST-API nachbilden. Mit deren Hilfe ist es möglich, die Daten mithilfe von Json.NET konvertieren zu können.

Analog zum `CmdParser` müssen auch bei der Nutzung der REST-API die Zeitstempel gesondert betrachtet werden. Damit die Konvertierung der Java-Zeitstempel in Millisekunden seit dem 1. Januar 1970 00:00:00 Uhr gemeinsam mit den anderen Daten durch das Json.NET-Framework durchgeführt werden kann, musste hierfür ein gesonderter Konverter entwickelt werden. Der Konverter nutzt ebenfalls die in Listing 4.11 gezeigte `ParseJavaTimestamp()` zum Konvertieren der Daten:

```
1  public class JsonJavaEpochConverter : DateTimeConverterBase
2  {
3    private static readonly DateTime _Epoch =
4       new DateTime(1970, 1, 1, 0, 0, 0, DateTimeKind.Utc);
5
6    public override void WriteJson(JsonWriter writer,
7       object value, JsonSerializer serializer)
8    {
9      writer.WriteRawValue(
10        ((DateTime)value - _Epoch).TotalMilliseconds.ToString());
```

[2] https://www.newtonsoft.com/json/

```
11    }
12
13    public override object ReadJson(JsonReader reader, Type objectType,
14        object existingValue, JsonSerializer serializer)
15    {
16      return DriverUtilities.ParseJavaTimestamp((long)reader.Value);
17    }
18  }
```

Listing 4.12: Entwickelter Konverter für Java-Zeitstempel zur Nutzung mit Json.NET. Dieser erbt dafür von `DateTimeConverterBase` des Json.NET-Frameworks, damit der `JsonJavaEpochConverter` auch zur Konvertierung genutzt werden kann.

Da der TLS zusätzliche Informationen zu Attempts und Containern enthält, werden hier nicht nur die Daten mithilfe des RM bzw. der NM der ausführenden Nodes, sondern auch mithilfe des TLS ermittelt. Hierzu werden zunächst die Daten des RM bzw. der NM ermittelt und mit den zusätzlichen Informationen des TLS ergänzt, sofern hier Daten verfügbar sind. Aufgrund der Besonderheiten der REST-API in Bezug auf Attempts und Container, werden bei diesen beiden YARN-Komponenten zunächst immer die Daten aller Attempts einer Anwendung bzw. aller Container eines Attempts ermittelt und konvertiert. Sollten jedoch nur die Daten jeweils eines Attempts bzw. Containers benötigt werden, werden diese Listen entsprechend gefiltert:

```
1 public IContainerResult ParseContainerDetails(string containerId)
2 {
3   var attemptId = DriverUtilities.ConvertId(containerId, EConvertType.
        Attempt);
4   var allContainers = ParseContainerList(attemptId);
5   return allContainers.FirstOrDefault(c => c.ContainerId ==
        containerId);
6 }
```

Listing 4.13: Konvertierung und Rückgabe eines Containers durch den `RestParser`. Hierbei muss für den hier gezeigten, einzelnen Container zunächst die ID des übergeordneten Attempts ermittelt werden, bevor aus der Liste aller Container die Daten des gesuchten Containers zurückgegeben werden können. Bei Attempts ist dieses Vorgehen analog.

Aufgrund dieser Besonderheiten eignet sich auch das in Unterabschnitt 4.2.5 beschriebene Monitoring durch die übergeordnete Komponente bei der Nutzung der REST-API besser als das Selbstmonitoring.

4.3.3 Entwicklung der Connectoren

Der Connector dient zur Abstrahierung der SSH-Verbindung (vgl. Unterabschnitt 4.3.4), damit diese in höheren Schichten einfach genutzt werden kann. Dazu beinhaltet der Connector die jeweiligen Befehle, die auf dem Cluster-Host ausgeführt werden können:

- Monitoring aller YARN-Komponenten vom RM, NM oder TLS
- Starten und Beenden von Nodes bzw. derer Netzwerkverbindungen
- Starten und Beenden von Anwendungen
- Prüfen und Löschen von Daten auf dem HDFS
- Starten und Beenden des gesamten Clusters
- Monitoring des MARP-Wertes

Die implementierten Connectoren stellen hierzu eine oder mehrere SSH-Verbindungen her, von denen jede Verbindung nur für einen bestimmten Typ an Befehlen genutzt wird. Je nach Fähigkeiten des Connectors werden dadurch einzelne Verbindungen zum Monitoring, zum Injizieren und Reparieren von Komponentenfehlern oder zum Starten von Anwendungen aufgebaut. Wenn das Cluster auf mehreren Hosts ausgeführt wird, werden zu jedem Host die entsprechend benötigten Verbindungen aufgebaut.

Das Initialisieren und Ausführen von Befehlen auf dem Cluster-Host erfolgt ähnlich wie bei den Parsern immer nach dem gleichen Schema. Zunächst werden beim Initialisieren des Connectors alle für seine Aufgaben benötigten SSH-Verbindungen initialisiert und aufgebaut. Die hierfür notwendigen Zugangsdaten werden durch den Connector aus den `ModelSettings` den Verbindungen zur Verfügung gestellt (vgl. Unterabschnitte 4.2.1 und 4.3.4). Bevor ein Befehl auf dem Cluster-Host ausgeführt wird, wird zunächst geprüft, ob der Connector diesen Befehl unterstützt bzw. die dafür benötigten Verbindungen initialisiert wurden. Wenn der Connector den Befehl nicht unterstützt, wird stattdessen eine `PlatformNotSupportedException` ausgelöst, wenn die benötigten Verbindungen nicht initialisiert wurden eine `InvalidOperationException`. Anschließend werden die benötigten Parameter des auszuführenden Befehls ermittelt, zu denen auch die Auswahl des Hosts dazu gehört, auf dem der Befehl ausgeführt werden soll. Nach der im Anschluss folgenden Ausführung des Befehls auf dem Host des Clusters werden die zurückgegebenen Daten bei Monitoring-Befehlen im Rohformat an die anfragende Komponente weitergeleitet bzw. bei anderen Befehlen zunächst einer einfachen Auswertung unterzogen, damit das Ergebnis des Befehls durch die anfragende Komponente verarbeitet werden kann. Aus diesem Grund werden die meisten Befehle synchron ausgeführt, wodurch die weitere Ausführung eines Testfalls solange unterbrochen wird, solange der Befehl auf dem Host ausgeführt wird. Eine Ausnahme bildet hierbei das Starten von Anwendungen, was auch *teilsynchron* und vollständig asynchron durchgeführt werden kann. Teilsynchron bedeutet hier, dass die auf dem Cluster zu startende Anwendung solange synchron ausgeführt wird, bis der Anwendung eine ID zugewiesen wurde, die vom Connector zurückgegeben wird, bevor sie

anschließend asynchron ausgeführt wird. Diese Funktion wird daher zum Starten der Anwendungen durch den Client genutzt, da dieser die Anwendungs-ID abspeichert (vgl. Unterabschnitt 4.2.7).

Da die beiden implementierten Parser unterschiedliche Befehle zum Abrufen der Daten benötigen, wurden hierfür zwei entsprechende Connectoren entwickelt. Die Connectoren führen dabei nicht nur Befehle zum Monitoring aus, sondern alle die für ihre Schnittstellen verfügbaren und benötigten Befehle. Dies wirkt sich vor allem auf den `CmdConnector` aus, da dieser in dieser Fallstudie bis auf das Monitoring jede Aktion mit dem Cluster durchführt.

Implementierung des CmdConnectors

Der erste der beiden implementierten Connectoren ist der `CmdConnector`. Er dient zum Ausführen von allen CLI-Befehlen, die im Rahmen der Fallstudie benötigt werden. Dabei wird nicht immer direkt mit Hadoop oder den Anwendungen interagiert, sondern immer mithilfe eines entsprechenden Setup- oder Startscriptes (vgl. Unterabschnitt 4.4.3), welches vom Connector hierzu aufgerufen wird. Das konkret genutzte Setup-Script ist hierbei abhängig vom `HostMode`. Da die Verwaltung des genutzten `HostMode`s vollständig durch die `ModelSettings` durchgeführt wird, kann der Connector unabhängig hiervon genutzt werden (vgl. Unterabschnitt 4.2.1). Notwendig ist hierfür jedoch, dass die Scripte immer die in Anhang C gezeigten Befehle enthalten.

Da das Starten der Anwendungen die einzige asynchrone Operation darstellt, die der Connector durchführen muss, werden hierfür die meisten SSH-Verbindung aufgebaut. Wenn eine Anwendung gestartet werden soll, wird immer eine freie SSH-Verbindung ausgewählt und die Anwendung mithilfe dieser gestartet. Die Verbindung wird dabei solange zum Starten von anderen Anwendungen gesperrt, solange der ausführende Befehl, also die gestartete Anwendung, nicht beendet wurde (vgl. Abschnitt 4.3). Sind dadurch alle Submitter belegt und eine weitere Anwendung soll gestartet werden, wird nach einer gewissen Zeit eine `TimeoutException` ausgelöst. Die Befehlsparameter zum Starten der Anwendungen werden von den jeweiligen Benchmarks zur Verfügung gestellt, sodass der Connector nur das Benchmark-Script mit den bereitgestellten Befehlsparametern ausführen muss (vgl. Unterabschnitte 4.4.3 und 5.3.1).

Implementierung des RestConnectors

Der zweite implementierte Connector, der `RestConnector`, dient vor allem dem Monitoring mithilfe der REST-API. Aus diesem Grund lösen alle Methoden, die nicht dem Monitoring durch die REST-API dienen, eine entsprechende `PlatformNotSupported Exception` aus.

Zum Abrufen der Daten dienen ebenfalls SSH-Verbindungen, jedoch ist hier eine pro Host, auf dem das Cluster ausgeführt wird, ausreichend. Auf dem Host wird zum

Abrufen der Daten das Tool curl[3] genutzt, wobei die Daten immer explizit im JSON-Format angefragt werden.

Da die Daten von YARN-Containern immer durch die NM der ausführenden Nodes zurückgegeben werden, kann es hier passieren, dass der entsprechende Node aufgrund eines Komponentenfehlers beendet oder nicht erreichbar ist. Wenn daher statt den Container-Daten eine Fehlermeldung zurückgegeben wird, werden in so einem Fall keine Rohdaten vom Connector zurückgegeben, sondern `String.Empty`.

Auch die grundlegenden Funktionen des `RestConnector`s sind unabhängig vom genutzten `HostMode` (vgl. Unterabschnitt 4.4.2). Die Verwaltung der Adressen von RM und TLS erfolgt analog zu der des Setup-Scripts beim `CmdConnector` durch die `ModelSettings`-Klasse. Ein Unterschied besteht jedoch bei den Adressen der NM zum Monitoring der Container. Da die Adressen der jeweiligen Nodes in `YarnNode.HttpUrl` gespeichert sind, werden diese Adressen auch vom Connector genutzt, um mithilfe der REST-API die Daten der auf einem Node ausgeführten Container zu abzurufen. Daher wird die vom `HostMode` abhängige Adresse der Nodes bereits beim Initialisieren des Modells entsprechend im Modell gespeichert (vgl. Unterabschnitte 4.2.1 und 4.2.2).

4.3.4 Implementierung der SSH-Verbindung

Die SSH-Verbindung zum Host des realen Clusters wird mithilfe des Frameworks SSH.NET[4] hergestellt. Verwaltet werden die Verbindungen mithilfe der Klasse `SshConnection`. Die Verbindung ist zudem der einzige Bestandteil des Treibers, zu dem kein entsprechendes Interface existiert, da die SSH-Verbindungen ausschließlich durch die implementierten Connectoren genutzt werden.

Die in `ModelSettings` gespeicherten Zugangsdaten zum Aufbau der Verbindung benötigten Zugangsdaten müssen beim Initialisieren vom Connector übergeben werden Unterabschnitt 4.2.1). Zwar kann hierzu mithilfe der Verbindungsklasse auch ein Passwort genutzt werden, jedoch werden in `ModelSettings` nur die SSH-Schlüssel gespeichert. Zudem bietet die implementierte `SshConnection`-Klasse bei der Nutzung eines SSH-Schlüssels die Möglichkeit, die Verbindung jederzeit zu trennen und erneut aufzubauen, was im implementierten Modell jedoch nicht durchgeführt wird.

Die Verbindungsklasse ist auch dafür zuständig, die vom Connector erhaltenen auszuführenden Befehle synchron an den Host zur Ausführung zu senden. Hierbei wird zudem jede Kommunikation zwischen dem Treiber und dem Cluster-Host in einer eigenen, vom Testsystem unabhängigen, Logdatei gespeichert, dem SSH-Log. Die Verbindungsklasse ist auch dafür zuständig, dass beim teilsynchronen Starten einer Anwendung zunächst die Anwendungs-ID ermittelt wird, bevor die Anwendung asynchron ausgeführt wird (vgl. Unterabschnitte 4.2.7 und 4.3.3).

[3] `https://curl.haxx.se/`
[4] `https://github.com/sshnet/SSH.NET`

Je SSH-Verbindung kann nur ein Befehl gleichzeitig auf dem Host ausgeführt werden. Daher muss die Verbindungsklasse sicherstellen, dass die Verbindung auch bei asynchron ausgeführten Befehlen solange als belegt markiert ist, solange der Befehl nicht beendet ist. Dies geschieht mithilfe einer entsprechenden Eigenschaft `InUse`, mit der freie und belegte Verbindungen voneinander getrennt werden können.

4.4 Umsetzung des realen Clusters

Das reale Cluster wurde mithilfe der Plattform Hadoop-Benchmark umgesetzt. Hierzu wurden speziell für diese Fallstudie verschiedene Szenarien entwickelt, mit denen das Cluster mit den benötigten Einstellungen bzw. den zu testenden Mutanten gestartet wird. Um die Verwaltung des Clusters zu vereinfachen, wurden zudem entsprechende Setup- und Startscripte entwickelt, die auch von den implementierten Connectoren genutzt werden.

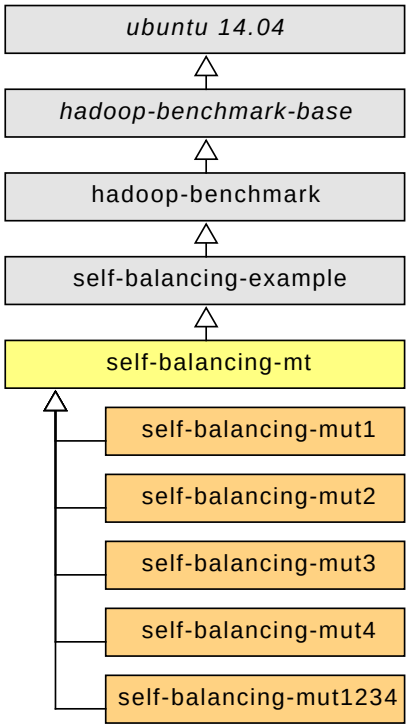

Abbildung 4.4: Vererbungshierarchie der Docker-Images in Hadoop-Benchmark. Die kursiv markierten Images können nicht mithilfe von Hadoop-Benchmark gestartet werden, die grau unterlegten Images sind bereits in Hadoop-Benchmark enthalten bzw. benötigt.

4.4.1 Grundlegender Aufbau

Da Docker und Hadoop vor allem zum Einsatz in Linux-Umgebungen entwickelt wurden, werden für das reale Cluster bis zu zwei physische Linux-Hosts genutzt. Auf einem dieser beiden Hosts wird mithilfe von VirtualBox eine VM mit Windows 10 ausgeführt, um die Tests mithilfe von S# auszuführen. Dies ist nötig, da S# mithilfe des .NET-Frameworks entwickelt wurde (vgl. Abschnitt 2.1), welches für Linux in Form von .NET Core zudem nur mit einem verringertem Funktionsumfang verfügbar ist [44].

Die beiden Hosts sind jeweils mit einem Intel Core i5-4570 @ 3,2 GHz x 4, 16 GB Arbeitsspeicher sowie einer SSD ausgestattet, auf der Ubuntu als Betriebssystem installiert ist.

Insgesamt wurden für diese Fallstudie die 6 in Abb. 4.4 gelb bzw. orange hinterlegten Szenarien mit den entsprechenden Docker-Images entwickelt. Das zentrale Image in dieser Fallstudie ist das gelb hinterlegte `self-balancing-mt`, in dem alle benötigten angepassten Einstellungen enthalten sind. Dies betrifft z. B. die Anpassung der Zeitabstände zur Erkennung von defekten Nodes oder das Starten des TLS, welcher in den Standard-Szenarien der Plattform nicht gestartet wird. Die orange hinterlegten Images

dienen zur Ausführung der Mutationstests. Die Images enthalten daher nur die mutierte Selfbalancing-Komponente, welche dadurch die in `self-balancing-example` enthaltene überschreiben. Das Namenssuffix der Images bzw. Szenarien in Form der Nummern geben an, welche der in Abschnitt 6.2 generierten Mutationen enthalten sind.

4.4.2 HostMode des Clusters

Der bereits mehrfach erwähnte `HostMode` beschreibt den prinzipiellen Aufbau des gestarteten Clusters. Das Cluster kann hierbei mithilfe von Docker-Machine auf entsprechenden VMs gestartet werden, auf denen das Cluster in Docker-Containern ausgeführt wird (vgl. Abschnitt 2.4). Der größte Nachteil dieses `DockerMachine`-Modes ist, dass das Cluster nicht bzw. nur umständlich auf mehreren Hosts ausgeführt werden kann.

Um das Cluster auf beiden für diese Fallstudie genutzten Hosts auszuführen, wurde daher der `Multihost`-Mode entwickelt, bei dem die Docker-Container des Clusters direkt auf den jeweiligen Hosts ohne Docker Machine gestartet werden:

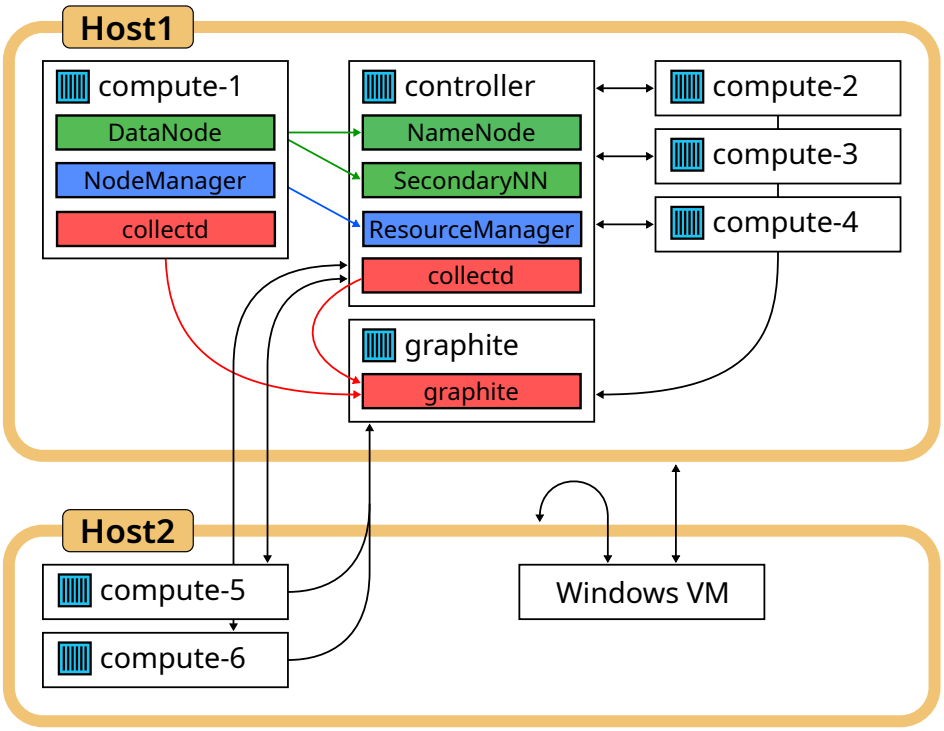

Abbildung 4.5: Cluster-Setup bei der Nutzung des `Multihost`-Modes. Hier ist das konkrete, in dieser Fallstudie genutzte Setup gezeigt. Grün: HDFS, Blau: YARN, Rot: Graphite.

Die genaue Anzahl der zu nutzenden Hosts und der Hadoop-Nodes ist in beiden `HostModes` variabel, wodurch es auch möglich ist, das Cluster nur auf Host1 zu starten. Damit das Cluster jedoch auf beiden Hosts gestartet werden kann, ist es nötig, zuvor beide Hosts als Docker-Swarm miteinander zu verbinden. Außerdem können mithilfe des `Multihost`-Modes dem Hadoop-Cluster weitere, sonst für die Ausführung der VMs benötigte Ressourcen, zur Verfügung gestellt werden, weshalb in dieser Fallstudie der

`Multihost`-Mode genutzt wurde. Beim `Mutihost`-Mode werden zudem nur auf Host1 die volle Anzahl an definierten Nodes pro Host gestartet, auf allen anderen Hosts wird jeweils die Hälfte gestartet und ausgeführt.

Je nach `HostMode` unterscheiden sich einzelne Pfade oder Adressen, z. B. die der REST-API. Auch aus diesem Grund wurde die bereits in Unterabschnitt 4.2.1 erläuterte Klasse `ModelSettings` entwickelt, welche die Verwaltung der entsprechenden Pfade und Adressen übernimmt.

4.4.3 Setup- und Startscripte

Die Plattform Hadoop-Benchmark enthält bereits zum Starten des Clusters ein Setup-Script und zum Ausführen der Benchmarks entsprechende Start-Scripte (vgl. Abschnitt 2.4). Um die Interaktion aufgrund der beiden unterschiedlichen `HostMode`s zu vereinfachen, wurde für jeden `HostMode` ein Setup-Script entwickelt, um eine einheitliche und vereinfachte Befehlssyntax bereitzustellen. Die beiden entwickelten Setup-Scripte werden vom `CmdConnector` genutzt, um die benötigten Aktionen auf dem Cluster auszuführen (vgl. Unterabschnitt 4.3.3). Die beiden Setup-Scripte abstrahieren somit die vom `HostMode` abhängigen, benötigten Befehle und sorgen dafür, dass der genutzte `HostMode` transparent ist, benötigen hierfür jedoch die gleiche, in Anhang C gezeigte, Befehlssyntax. Sie beinhalten entsprechende Befehle für folgende Aktionen:

- Starten und Beenden des gesamten Clusters
- Injizieren und Reparieren von Komponentenfehlern
- Ausführen von CLI-Befehlen von Hadoop

Zudem wird beim Starten des Clusters immer geprüft, ob sich die Dockerfiles geändert haben und entsprechend die Docker-Images aktualisiert, aus denen das Cluster gestartet wird. Die Setup-Scripte enthalten außerdem weitere, jeweils spezifische Befehle zum Umgang mit dem Cluster im entsprechenden `HostMode`.

Zum zentralen Starten der Benchmarks wurde ebenfalls ein zentrales Benchmark-Startscript entwickelt, welches die jeweiligen Start-Scripte der einzelnen Benchmarks ausführt. Dadurch kann analog zum Setup-Script unabhängig vom zu startenden Benchmark vom `CmdConnector` immer die gleiche Syntax genutzt werden. Das zentrale Benchmark-Startscript ermittelt, basierend auf dem zu startenden Benchmark, das jeweils benötigte Start-Script und übergibt diesem die benötigten Parameter. Die vom `HostMode` abhängigen Parameter zum Starten der Benchmark-Container werden von den hierfür angepassten Start-Scripten der Benchmarks selbst ermittelt. Die jeweiligen Startparameter der Anwendungen selbst werden vom Benchmark-Controller bereitgestellt (vgl. Unterabschnitt 5.3.1).

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 einem Transitionssystem in Form einer Markow-Kette miteinander verbunden, mit dem die Ausführungsreihenfolge der einzelnen Anwendungen, basierend auf Wahrscheinlichkeiten, bestimmt wird. Verwaltet werden die implementierten Benchmarks mithilfe des Benchmark-Controllers.

Da für die durchgeführte Fallstudie [8] das für diese Fallstudie entwickelte Testmodell genutzt wurde, enthält es auch das hierfür entwickelte Transitionssystem. Daher wurden Teile der Beiträge dieses Kapitels dort bereits publiziert.

5.1 Übersicht möglicher Anwendungen

Hadoop-Benchmark enthält bereits die Möglichkeit, unterschiedliche Benchmarks zu starten. Folgende Benchmarks sind bereits in der Plattform integriert (vgl. Abschnitt 2.4):

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

Jede Benchmark enthält ein spezifisches Start-Script, um die jeweiligen Benchmarks in einem Docker-Container zu starten. Dieser Container wird, abhängig vom `HostMode` (vgl. Unterabschnitt 4.4.2), auf der Docker-Machine des RM oder direkt auf dem Host gestartet. Da es in Docker-Umgebungen *best practice* ist, für jeden Einsatzzweck ein eigenes Image zu erstellen bzw. Container zu starten, werden für jede der drei Benchmark-Suites eigene Container ausgeführt. Um mehrere Benchmarks gleichzeitig ausführen zu können, wurden die Startscripte der Benchmarks daher entsprechend angepasst.

5.1.1 Mapreduce-Examples

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 der Hadoop-Installation und daher standardmäßig in jedem Hadoop-Cluster verfügbar. 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. die Ausführung der *Bailey-Borwein-Plouffe-Formel* [45] zur Berechnung einzelner Stellen von π

5.1.2 Intel HiBench

Intel HiBench[1] ist eine von Intel entwickelte Benchmark-Suite mit *Workloads* zu verschiedenen Anwendungszwecken mit jeweils unterschiedlichen einzelnen Anwendungen. Die Suite enthielt anfangs nur wenige Anwendungen [46], wurde im Laufe der Zeit jedoch stetig um neue Anwendungen und auch Workloads erweitert. Das zeigt sich auch darin, dass in Hadoop-Benchmark die HiBench-Version 2.2 integriert ist, die einen noch deutlich geringeren Umfang an Workloads und Anwendungen besitzt, wie z. B. die aktuellere Version 7. Aus diesem Grund wurde vor der Analyse der Anwendungen der HiBench-Suite das Docker-Image entsprechend auf Version 7 aktualisiert, um hier entsprechend alle in der Zwischenzeit hinzugefügten Workloads und Anwendungen der Suite nutzen zu können. HiBench enthält damit folgende Workloads mit einer unterschiedlichen Anzahl an möglichen Anwendungen:

- Micro-Benchmarks (basierend auf den Examples und den Jobclient-Tests)
- Maschinelles Lernen
- SQL/Datenbanken
- Websuche
- Graphen
- Streaming

5.1.3 SWIM

SWIM[2] ist eine aus 50 einzelnen Workloads bestehende Benchmark-Suite. Das besondere an SWIM ist, dass die Suite im Rahmen der Studie [47] entwickelt wurde, und dadurch anhand mehrerer tausend real ausgeführter MR-Jobs entwickelt wurde. Die dabei enthaltenen Workloads stellen damit eine größere Vielfalt an ausgeführten Anwendungen und damit einen größeren Testumfang dar als vergleichbare Benchmarks [48].

Bei der Ausführung auf dem in dieser Fallstudie verwendeten Cluster wurden jedoch nicht alle Workloads fehlerfrei ausgeführt. Zudem wird in [49] explizit erwähnt, dass die Ausführung auf einem Cluster auf einem Host sehr zeitintensiv ist, sofern die Workloads überhaupt ausgeführt werden können. SWIM ist außerdem für Benchmarks eines Clusters mit mehreren physischen Nodes ausgelegt, weshalb die Ausführung in

[1] `https://github.com/intel-hadoop/HiBench/`
[2] `https://github.com/SWIMProjectUCB/SWIM/`

dieser Fallstudie extrem viel Zeit benötigten würde. Daher wurde die Nutzung des SWIM-Benchmarks nicht weiter verfolgt.

5.1.4 Jobclient-Tests

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 Mapreduce-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 MR- 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 diese ebenfalls im Rahmen der Plattform Hadoop-Benchmark gestartet werden können. 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`
- DummyAnwendungen `sleep` und `fail`

5.2 Entwicklung des Transitionssystems

Damit die Fallstudie die Realität abbilden kann, wurden von den 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 Praxis Anwendungen nicht immer in der gleichen Reihenfolge ausgeführt werden und daher auch in dieser Fallstudie eine unterschiedliche Ausführungsreihenfolge der Anwendungen ermöglicht werden soll. Mithilfe der Festlegung eines bestimmten Seeds für den hierfür benötigten Pseudo-Zufallsgenerator besteht dennoch die Möglichkeit, einen Test mit den gleichen Anwendungen wiederholen zu können.

5.2.1 Auswahl der Benchmarks

Einige der in Unterabschnitt 5.1.1 erwähnten Mapreduce-Examples werden häufig als Benchmark verwendet. Einige Beispiele dafür sind die Anwendungen `sort` und `grep`, die bereits im Referenzpapier [15] des MR-Frameworks zum Testen genutzt wurden. Zum Testen des HDFS dient in [29] die DFSIO-Benchmark, um den Durchsatz beim Lesen und Schreiben einer großen Datenmenge auf dem HDFS zu messen. `terasort` und die beiden dazugehörigen Anwendungen `teragen` und `teravalidate` bilden ebenfalls

eine weit verbreitete Benchmark, der die Hadoop-Implementierung der standardisierten *Sort Benchmarks*[3] darstellt [50]. Ebenfalls als gute Benchmark dient die Anwendung `wordcount`, mit der ein großer Datensatz stark verkleinert bzw. zusammengefasst wird, und daher als gute Repräsentation für Anwendungsarten dient, bei denen Daten extrahiert werden [46, 47].

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 in genutzte Anwendungs- und Datentypen in produktiv genutzten Hadoop-Clustern geben u. a. die Studien [47, 51]. Auffällig ist hierbei, dass die meisten Anwendungen in einem Hadoop-Cluster innerhalb weniger Sekunden oder Minuten abgeschlossen sind und bzw. oder Datensätze im Größenbereich von wenigen Kilobyte bis hin zu wenigen Megabyte verarbeiten. Zu einem ähnlichen Ergebnis kam auch die Studie [52], in der gefolgert wird, dass für kleinere Jobs einfachere Frameworks abseits von Hadoop besser geeignet wären. Die Autoren der Studie [51] bezeichneten Hadoop aufgrund ihrer Ergebnisse als „potentielle Technologie zum Verarbeiten aller Arten von Daten“, folgerten aber ähnlich wie Ren et al. in [52], dass mit Hadoop meist Daten verarbeitet werden, die auch mit „traditionellen Plattformen“ verarbeitet werden könnten.

Basierend auf den Ergebnissen der Studien in [46, 47, 51, 52] und der in den Publikationen [15, 29, 50] verwendeten Benchmarks, wurden folgende Anwendungen der Mapreduce-Examples und Jobclient-Tests in das Transitionssystem übernommen:

- Generieren von Eingabedaten für andere Anwendungen:
  - Textdateien:
    - `randomtextwriter` (rtw): Generierung von zufälligen Zeichenfolgen
    - `TestDFSIO -write` (dfw): Schreiben von großen Datenmengen auf dem HDFS
  - Binärdateien:
    - `randomwriter` (rw): Generierung von zufälligen Binärdaten
    - `teragen` (tg): Generierung von Eingabedaten für `terasort`
- Verarbeitung von Eingabedaten:
  - Auslesen bzw. Zusammenfassen:
    - `wordcount` (wc): Auslesen von Textdaten und Ermitteln der Anzahl der darin enthaltenen Wörter
    - `TestDFSIO -read` (dfr): Auslesen von großen Datenmengen auf dem HDFS
  - Transformieren:

[3] `https://sortbenchmark.org/`

        * `sort` (so): Sortieren von Daten, wird in dieser Fallstudie zum Sortieren von Textdaten genutzt
        * `terasort` (tsr): Sortieren von großen Binärdatenmengen
    - Validierung der Transformationen:
        * `testmapredsort` (tms): Validierung der von `sort` transformierten Daten
        * `teravalidate` (tvl): Validierung der von `terasort` transformierten Binärdaten
- Ausführen von Berechnungen:
    - `pi`: Einfache Berechnung von π mithilfe der Quasi-Monte-Carlo-Methode. Die Monte-Carlo-Methode und die darauf basierende Quasi-Monte-Carlo-Methode sind stochastische Verfahren, um komplexe Probleme numerisch lösen zu können. Für weitere Informationen hierzu sei auf entsprechende Literatur, wie z. B. [53, 54], verweisen.
    - `pentomino` (pt): Berechnung einer Lösung von Pentomino-Problemen. Hierbei soll eine Fläche aus 64 Quadraten mithilfe von zwölf *Bausteinen* bedeckt werden, wobei jeder Baustein aus fünf Quadraten besteht und nur einmal genutzt werden darf. Weitere Informationen hierzu sind in entsprechender Literatur, wie z. B. in [55], zu finden.
- Dummy-Anwendungen:
    - `sleep` (sl): Blockieren von Ressourcen
    - `fail` (fl): Ausführung eines fehlschlagenden Map-Tasks

Der Grund für die Berücksichtigung von mehreren gleichen bzw. ähnlichen Anwendungen für einige Kategorien liegt darin, dass die unterschiedlichen Anwendungen einen unterschiedlichen Umfang bzw. unterschiedliche Datenrepräsentation (Text- und Binärdaten) repräsentieren. So stehen die beiden `TestDFSIO`-Varianten für eine umfangreichere Datennutzung, während die jeweils anderen Anwendungen einen kleineren Umfang repräsentieren. Ähnlich verhält es sich bei den beiden Berechnungs-Anwendungen, bei denen die `pentomino`-Anwendung die deutlich umfangreicheren Berechnungen durchführt. `TestDFSIO` enthält zudem die Möglichkeit, Daten zu generieren und zu lesen, weshalb dieser Benchmark in zwei Kategorien als Anwendung genutzt wird.

Eine Besonderheit bilden die beiden Dummy-Anwendnugen. Beide werden bei der Ausführung dieser Fallstudie dafür genutzt, um zu simulieren, wenn nichts ausgeführt werden soll oder bei der Ausführung der Anwendungen ein Fehler auftritt.

Auf die Implementierung einer Anwendung der HiBench-Suite wurde verzichtet. Da beim Starten einer Anwendung durch den Client die vom Cluster zugewiesene Anwendungs-ID benötigt wird, um die Anwendung in späteren Testfällen beenden zu können, kann der HiBench nicht sinnvoll im Testmodell genutzt werden. Der Grund hierfür ist, dass beim Starten einer Anwendung der HiBench-Suite die ID erst nach Abschluss

	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: Entwickelte Markov-Kette für die Anwendungs-Übergänge in Tabellenform

der gesamten Anwendung zurückgegeben wird, womit eine asynchrone Ausführung der Anwendung nicht mehr möglich wäre.

5.2.2 Entwicklung der Markow-Kette

Basierend auf den ausgewählten Anwendungen und der in den Studien genannten Anwendungstypen, wurde das Transitionssystem in Form einer Markov-Kette entwickelt. Die Markov-Kette definiert die Wahrscheinlichkeiten, mit denen die ausführenden Anwendungen bei der Ausführung eines Testfalls gewechselt werden. Damit die Übergänge nicht bei jedem Testfall stattfinden, sondern Anwendungen auch mehrere Testfälle lang ausgeführt werden können, wurden Selbst-Transitionen mit einer Wahrscheinlichkeit von 60 Prozent definiert.

Für die beiden Dummy-Anwendungen gelten einige Besonderheiten. Sie können beide unabhängig von der derzeit ausgeführten Anwendung mit einer sehr geringen Wahrscheinlichkeit als nachfolgende Anwendung ausgewählt werden. Zudem wurden als den Dummy-Anwendungen nachfolgende Anwendungen nur solche definiert, die ihrerseits keine Eingabedaten benötigten bzw. diese für andere Anwendungen bereitstellen:

- `TestDFSIO -write`
- `randomtextwriter`
- `teragen`
- `randomwriter`
- `pi`
- `pentomino`

Bei der Entwicklung der Markow-Kette des Transitionssystems wurde zudem berücksichtigt, welche Anwendungen welche Art von Eingabedaten benötigen. Dadurch wird sichergestellt, dass benötigte Eingabedaten immer vorhanden sind, da diese ebenfalls im

Rahmen der Ausführung der Benchmarks generiert werden können (vgl. Abschnitt 3.1 und Unterabschnitt 3.3.2). Anwendungen ohne Eingabedaten können dagegen fast jederzeit ausgeführt werden, wie die entwickelte Markov-Kette in Tabelle 5.1 zeigt.

5.3 Entwicklung des Benchmark-Controllers

Die im YARN-Modell (vgl. Abschnitt 4.2) implementierten Benchmarks und das zur Auswahl der Benchmarks entwickelte Transitionssystem bilden zusammen den Benchmark-Controller. Der Benchmark-Controller wurde als eigene Klasse `BenchmarkController` im Modell implementiert und wird von den Clients genutzt, um die Auswahl der Benchmarks vorzunehmen. Der Benchmark-Controller verwaltet die implementierten Benchmarks und stellt diese den Clients zum Starten zur Verfügung. Damit die Clients unabhängig voneinander sind, wird für jeden Client ein eigener Benchmark-Controller instanziiert (vgl. Unterabschnitt 4.2.7).

5.3.1 Implementierung von Benchmarks und Transitionssystem

Die Benchmarks sind mithilfe der Klasse `Benchmark` implementiert. Sie enthält alle zur Ausführung der Benchmarks benötigten Informationen und stellt diese der `BenchmarkController`-Klasse bzw. dem Client bereit, um die Anwendung des Benchmarks zu starten. Da mehrere Clients unabhängig voneinander agieren können müssen, erhält jeder Client ein spezifisches 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 hierfür entsprechende Platzhalter ersetzt werden müssen, wenn mithilfe der Methode `GetStartCmd()` der Start-Befehl des Benchmarks zurückgegeben wird:

```
1  public class Benchmark
2  {
3    public Benchmark(int id, string name, string startCmd,
4       string outputDir, string inputDir)
5    {
6      _StartCmd = startCmd;
7      _InDir = inputDir;
8      HasInputDir = true;
9    }
10
11   public string GetStartCmd(string clientDir = "")
12   {
13     var result = _StartCmd
14        .Replace(OutDirHolder, GetOutputDir(clientDir))
15        .Replace(InDirHolder, GetInputDir(clientDir));
16     if(result.Contains(BaseDirHolder))
17       result = ReplaceClientDir(result, clientDir);
```

```
18      return result;
19    }
20  }
```

Listing 5.1: Wesentliche Methoden der Klasse `Benchmark`

Die im Modell enthaltenen Benchmarks sind als Array in `BenchmarkController` gespeichert. Hierbei werden mithilfe der Holder-Variablen, die in `GetStartCmd()` ersetzt werden, die Startbefehle der Anwendungen definiert:

```
public Benchmark[] Benchmarks => new[]
{
  new Benchmark(04, "wordcount",
     $"example wordcount {InDirHolder} {OutDirHolder}",
     $"{BaseDirHolder}/wcout"),
};
```

Listing 5.2: Definition der verfügbaren Benchmarks im `BenchmarkController` (gekürzt)

Der hierbei definierte und durch `GetStartCmd()` zurückgegebene vollständige Startbefehl wird beim Starten der Anwendungen vom Connector als Befehlsparameter des Benchmark-Script genutzt (vgl. Unterabschnitt 4.3.3). Damit kann durch das Benchmark-Script die zu startende Anwendung identifiziert und das jeweilige Start-Script ausgeführt werden (vgl. Unterabschnitt 4.4.3).

Das Transitionssystem selbst wurde im `BenchmarkController` als zweidimensionaler Array implementiert, auf das mithilfe von `Benchmark.ID` zugegriffen werden kann. Für die spätere Auswahl der Benchmarks ist es hierbei wichtig, dass die Reihenfolge der Werte im Transitionssystem mit den IDs der Benchmarks übereinstimmt:

```
public double[][] BenchTransitions => new[]
{
  /* from / to ->    00    01    02    ...*/
  /* from / to ->   dfw   rtw   tg    ...*/
  new[] /* 00 */ { .600, .073,  000, ... ,
  new[] /* 01 */ { .036, .600,  000, ... ,
  new[] /* 02 */ {  000, .036, .600, ... ,
};
```

Listing 5.3: Implementierung des Transitionssystems im `BenchmarkController` (gekürzt)

5.3.2 Auswahl der nachfolgenden Benchmarks

Zur Auswahl der nachfolgenden Anwendung dient die Methode `ChangeBenchmark()` des `BenchmarkControllers`. Hier wird mithilfe des Transitionssystems und unabhängig von anderen Clients die nachfolgende Benchmark bestimmt, welche dabei in `CurrentBenchmark` gespeichert wird:

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

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

  // prevent saving current benchmark as previous
  if(CurrentBenchmark == _BenchmarksInstance[i])
    break;

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

Listing 5.4: Auswahl des nachfolgenden Benchmarks (gekürzt). Dies stellt einen Ausschnitt der Methode `ChangeBenchmark()` dar, welche vom Client zur Bestimmung des nachfolgenden Benchmarks aufgerufen wird (vgl. Unterabschnitt 4.2.7).

Bevor auf die Daten des implementierten Transitionssystems zugegriffen wird, wird außerdem zunächst geprüft, ob die Markow-Kette alle möglichen Übergänge für den aktuellen Benchmark enthält. Ist das nicht der Fall, wird eine `InvalidOperationException` ausgelöst. Wenn die Auswahl des Benchmarks dagegen erfolgreich war, wird an den aufrufenden Client `true` zurückgegeben und der ausgewählte Benchmark kann gestartet werden (vgl. Unterabschnitt 4.2.7).

Der zur Auswahl der nachfolgenden Benchmarks benötigte Zufallsgenerator `RandomGen` wird bei der Initialisierung des Benchmark-Controllers, basierend auf dem Basisseed (vgl. Unterabschnitt 3.3.2), initialisiert. Damit die Benchmark-Controller aller Clients nicht die gleichen Benchmarks auswählen, wird zum Basisseed die numerische ID des zum Benchmark-Controler zugehörigen Clients addiert und der Zufallsgenerator mit diesem Wert initialisiert. Als initiale Anwendung wird immer der `sleep`-Benchmark genutzt, sodass als erste Anwendung immer eine Anwendung ohne benötigte Eingabedaten gestartet wird (vgl. Unterabschnitt 5.2.2).

5.3.3 Vorabgenerierung von Eingabedaten

Neben der Generierung der für einige Anwendungen benötigten Eingabedaten während der Testausführung, gibt es auch die Möglichkeit, die Eingabedaten vorab zu generieren und diese bei der Testausführung zu nutzen (vgl. Abschnitt 3.1 und Unterabschnitte 3.3.2 und 6.1.2). Hierfür muss bei der Initialisierung des Tests die entsprechende Eigenschaft `ModelSettings.IsPrecreateBenchInputs` auf `true` gesetzt werden, wodurch auch direkt die entsprechenden Eingabedaten auf dem Cluster generiert werden.

Die Vorabgenerierung der Daten startet folgende Anwendungen, die Eingabedaten für andere Anwendungen generieren und speichert diese in einem nur hierfür genutzten Verzeichnis im HDFS:

- `TestDFSIO -write`
- `randomtextwriter`
- `teragen`
- `sort`
- `terasort`

Gestartet werden kann die Vorabgenerierung mithilfe des Benchmark-Controllers. Hierbei werden die Anwendungen, sofern möglich, gleichzeitig ausgeführt und anschließend gewartet, bis alle fünf Anwendungen beendet sind. Hierbei wird standardmäßig die Generierung der Eingabedaten einer Anwendung übersprungen, wenn das entsprechende Ausgabeverzeichnis der Anwendung bereits existiert.

Es ist jedoch auch möglich, vorhandene Verzeichnisse zu löschen, um somit die Daten vollständig neu zu generieren. Hierbei wird zudem ein HDFS-Filecheck ausgeführt, um fehlerhafte Daten zu finden und zu löschen. Fehlerhafte Daten können im HDFS z. B. dadurch entstehen, dass alle Nodes defekt sind, auf denen ein Block repliziert wurde, sich die Datei jedoch noch im Index des HDFS befindet (vgl. Abschnitt 2.2). Die vollständige Neugenerierung der Eingabedaten kann mithilfe der Eigenschaft `ModelSettings.IsPrecreateBenchInputsRecreate` gesteuert werden (vgl. Unterabschnitt 6.1.2).

Um beim Starten der Anwendungen die vorab generierten Eingabedaten zu nutzen, wird beim Starten der Anwendungen das Verzeichnis der vorab generierten Daten als entsprechendes Client-Verzeichnis genutzt (vgl. Unterabschnitte 4.2.7 und 5.3.1).

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

Alle zur Ausführung der Simulation relevanten Methoden sind in der Klasse `Simulation Tests` zusammengefasst. Hierbei gibt es neben der Simulation mit Aktivierung der Komponentenfehler auch eine Möglichkeit zur Simulation ohne die Aktivierung von Komponentenfehlern sowie weitere, mit der Simulation zusammenhängende Methoden.

6.1.1 Grundlegender Aufbau

Da im realen Cluster Hadoop kontinuierlich Anpassungen und Tests mithilfe von S# mit diskreten Schritten durchgeführt werden, muss beachtet werden, dass die bei den Tests ermittelten Werte 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 und erst nach einer gewissen Zeit aus der Konfiguration des Clusters entfernt werden (vgl. Abschnitt 2.2). Genauso verhält es sich, wenn ein Node bzw. seine Verbindung wieder aktiviert wird, da dieser zunächst gestartet und anschließend die Verbindung mit den RM hergestellt 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 vom Simulator ausgeführter Testfall um eine gewisse Zeit verzögert werden, sodass alle Aktivitäten zur Verwaltung des Hadoop-Clusters genügend Zeit zur Ausführung erhalten. Dadurch erhält jeder Testfall eine gewisse Mindestdauer, die erreicht sein muss, bevor der nachfolgende Testfall ausgeführt werden kann.

Beim grundlegenden Ablauf der Simulation wird dieser Umstand daher bereits berücksichtigt:

```
private bool ExecuteSimulation()
{
  var model = InitModel();
  // 0.000001 to prevent inaccuracy
  var isWithFaults = FaultActivationProbability > 0.000001;

  var wasFatalError = false;
  try
  {
    // init simulation
    var simulator = new SafetySharpSimulator(model);
    var simModel = (Model)simulator.Model;
    var faults = CollectYarnNodeFaults(simModel);

    SimulateBenchmarks();

    // do simuluation
    for(var i = 0; i < StepCount; i++)
    {
      OutputUtilities.PrintStepStart();
      var stepStartTime = DateTime.Now;

      if(isWithFaults)
      HandleFaults(faults);
      simulator.SimulateStep(); // execute test case

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

      OutputUtilities.PrintFullTrace(simModel.Controller);
    }
    // collect fault counts and check constraint
  }
  // catch/finally

  return !wasFatalError;
}
```

Listing 6.1: Ausführung der Simulation (gekürzt).

Hierbei gibt es zwei Variationen zum Ausführen der Simulation, welche abhängig von der Aktivierung der Komponentenfehler ist. Sollen keine Komponentenfehler aktiviert bzw. deaktiviert werden, werden die entsprechenden Variablen zur Festlegung der generellen Wahrscheinlichkeiten (vgl. Unterabschnitt 6.1.2) entsprechend gesetzt und die Simulation ausgeführt. Nach Abschluss eines Testfalls durch `SimulateStep()` wird

geprüft, ob die Ausführung des Testfalls die Mindestdauer unterschritten hat und die weitere Ausführung in dem Fall entsprechend verzögert.

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 alle noch ausgeführten Anwendungen beendet und defekte Nodes neu gestartet, sofern nötig.

Für die Simulation selbst sind zudem ebenfalls Constraints definiert. Dies ist dadurch nötig, da Anforderung, dass Komponentenfehler injiziert bzw. repariert werden (vgl. Unterabschnitt 3.2.2), nicht immer innerhalb eines Testfalls validiert werden kann. Aus diesem Grund wird diese Anforderung in Form von Constraints nach Abschluss einer Simulation durch das Oracle geprüft.

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   // save fields in ModelSettings
16
17   var model = Model.Instance;
18   model.InitModel(appCount: StepCount, clientCount: ClientCount);
19   model.Faults.SuppressActivations();
20
21   return model;
22 }
```

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 Testfälle der Simulation.

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 (vgl. Unterabschnitt 4.3.3). Diese Eigenschaft hat jedoch nur einen Einfluss, wenn der `Mutlihost`-Mode genutzt wird (vgl. Unterabschnitt 4.4.2).

NodeBaseCount

Definiert die Anzahl der Nodes auf Host1. Auf einem möglichen Host2 wird die Hälfte der Nodes verwendet (vgl. Unterabschnitt 4.4.2). Benötigt wird dieser Wert, um mithilfe der REST-API auf die Hadoop-Nodes zugreifen zu können, um die Daten der YARN-Container zu ermitteln (vgl. Unterabschnitte 4.2.1 und 4.3.3).

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 (vgl. Abschnitt 3.1 und Unterabschnitte 3.3.2 und 5.3.3). Die Eigenschaft `RecreatePreInputs` definiert hierfür, ob bereits bestehende Eingabedaten neu generiert werden, was standardmäßig nicht der Fall ist bzw. die Eigenschaft auf `false` gesetzt ist (vgl. Unterabschnitt 5.3.3). Die Werte der beiden Eigenschaften werden daher entsprechend in ihren korrespondierenden Eigenschaften in `ModelSettings` gespeichert.

Die direkt im Anschluss an die Initialisierung des Simulators aufgerufene Methode `CollectYarnNodeFaults()` ermittelt alle im initialisierten Modell enthaltenen, mit `NodeFaultAttribute` markierten Komponentenfehler (vgl. Unterabschnitt 4.2.3):

```
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 Unterabschnitt 4.2.4 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. Benötigt werden diese Werte zur Validierung der Constraints der Simulation durch das Oracle (vgl. Unterabschnitt 6.1.1).

6.1.3 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. So kann z. B. die Vorabgenerierung der Eingabedaten durch eine entsprechende Methode durchgeführt werden (vgl. Unterabschnitt 5.3.3). Es ist aber auch möglich, nur die Simulation der durch den Benchmark-Controller ausgewählten Benchmarks durchzuführen. Hierzu kann die bei der Ausführung der Simulation aufgerufene Methode `SimulateBenchmark()` als eigener Test durchgeführt werden:

```
1  [Test]
2  public void SimulateBenchmarks()
3  {
4    for(int i = 1; i <= _ClientCount; i++)
5    {
6      var seed = _BenchmarkSeed + i;
7      var benchController = new BenchmarkController(seed);
8      Logger.Info($"Simulating Benchmarks for Client {i} with Seed {seed
           }:");
9      for(int j = 0; j < _StepCount; j++)
10     {
11       benchController.ChangeBenchmark();
12       Logger.Info($"Step {j}: {benchController.CurrentBenchmark.Name}"
             );
13     }
14   }
15 }
```

Listing 6.4: Simulation der auszuführenden Benchmarks

6.1.4 Ablauf eines Tests und der Testfälle

Zu Beginn eines Tests werden zunächst die in Unterabschnitt 3.3.3 definierten, grundlegenden Daten des Tests im Programmlog gespeichert. Daneben werden aber auch noch weitere, auch den `HostMode` (vgl. Unterabschnitt 4.4.2) betreffende Daten im Programmlog abgespeichert:

- Verbundene SSH-Verbindungen mit einer ID zur besseren Zuordnung im SSH-Log
- Ausführung der Erstellung von vorab generierten Eingabedaten
- Vollständiger Pfad des verwendeten Setup-Scriptes
- Adresse des Controllers zur Nutzung der REST-API
- Simulation der auszuführenden Benchmarks (vgl. Unterabschnitt 6.1.3)

Im Anschluss werden das auszuführende YARN-Modell und der zur Ausführung des Tests genutzte Simulator initialisiert, bevor die Testfälle selbst ausgeführt werden.

Der Ablauf eines Testfalls lässt sich in mehrere Abschnitte einteilen. Zunächst wird vom Simulator mithilfe der Attribute der Komponentenfehler entschieden, ob ein Komponentenfehler aktiviert und im Cluster injiziert wird (vgl. Unterabschnitte 4.2.3 und 4.2.4). Anschließend führt der Simulator den Testfall aus, indem für alle Komponenten des Modells die `Update()`-Methoden ausgeführt werden. Hierdurch werden deaktivierte Komponentenfehler auch im Cluster wieder repariert, da `YarnNode.Update()` die entsprechenden Start-Methoden aufruft (vgl. Unterabschnitt 4.2.3).

Anschließend wird die in Unterabschnitt 4.2.8 erläuterte Routine des Controllers ausgeführt. Dabei wird zunächst der MARP-Wert aus dem Cluster ausgelesen, bevor die Benchmark-Anwendungen des Testfalls gestartet werden. Jeder Client nutzt dafür die in Unterabschnitt 4.2.7 beschriebe Routine, um zunächst durch den Benchmark-Controller die zu startende Anwendung auszuwählen (vgl. Unterabschnitt 5.3.2) und zu starten. Die dabei vom Cluster zugewiesene Anwendungs-ID wird vom Client gespeichert, um die Anwendung in nachfolgenden Testfällen bei Bedarf abbrechen zu können, wenn eine neue Anwendung gestartet werden soll (vgl. Unterabschnitt 4.2.7).

Sobald alle Anwendungen gestartet sind, wird vom Controller das Monitoring aller Nodes, Anwendungen und ihrer Attempts durchgeführt. Dabei wird der MARP-Wert erneut ausgelesen, da sich der Wert durch die Selfbalancing-Komponente regelmäßig ändern kann (vgl. Unterabschnitt 4.2.8 und Abschnitt 2.3).

Den Abschluss eines Testfalls bildet die Validierung der Werte des Clusters, die beim Monitoring im YARN-Modell gespeichert wurden. Dazu werden für jede Komponente im Modell die jeweiligen auf den Anforderungen basierenden Constraints durch das Oracle geprüft (vgl. Abschnitt 3.2 und Unterabschnitte 4.2.6 und 4.2.8). Wenn ein Constraint nicht erfolgreich validiert wurde, wird dies jeweils im Programmlog gespeichert bzw. die Ausführung der Simulation abgebrochen, wenn sich das Cluster nicht mehr rekonfigurieren kann (vgl. Unterabschnitt 4.2.9).

Nach Abschluss eines Testfalls werden durch den Simulator die in Unterabschnitt 3.3.3 geforderten Daten im Programmlog abgespeichert, die beim Monitoring erkannt wurden. Daneben werden bei der Ausführung eines Testfalls aber auch weitere Daten im Programmlog gespeichert, wie z. B.:

- Ausführung von Komponentenfehlern
- Diagnostik-Daten der YARN-Komponenten
- Verletzte Constraints und die betroffenen YARN-Komponenten
- Die Information, wenn eine Rekonfiguration nicht möglich ist

Zum Abschluss der Simulation werden zudem die für die Simulation als gesamtes betreffenden Constraints validiert, welche nicht im YARN-Modell selbst implementiert wurden (vgl. Unterabschnitt 6.1.1).

Nach Abschluss der Simulation wird ein erneutes Monitoring des gesamten Clusters durchgeführt. Der hierbei ermittelte Status wird im Programmlog als finaler Clusterstatus und gemeinsam mit einigen statistischen Kenndaten gespeichert:

- Gesamtdauer der Simulation
- Anzahl erfolgreicher Schritte
- Anzahl der maximal möglichen, aktivierbaren Komponentenfehler
- Anzahl aktivierter und deaktivierter Komponentenfehler
- Letzter ermittelter MARP-Wert
- Anzahl aller ausgeführten, erfolgreichen, fehlgeschlagenen sowie abgebrochenen Anwendungen
- Anzahl aller ausgeführten Attempts
- Anzahl aller während der Ausführung erkannten Container
- Anzahl aller validierten Constraints und verletzten Constraints, getrennt nach SuT- und Testsystem-Constraints

Der auszugsweise Programmlog eines Tests findet sich in Anhang E.

6.2 Generierung der Mutanten

Um das entwickelte Testsystem selbst zu validieren, wurden einige Mutationstests entwickelt. Mutationstests werden vor allem in der Forschung eingesetzt, um Fehler im SuT oder dem Testsystem selbst zu finden, in dem das SuT verändert wird. Ziel ist es, die im SuT implementierten Mutanten zu finden, und dabei möglichst weitere Fehlerquellen zu ermitteln [56–59].

Da sich Mutationstests zu einem großen Forschungszweig entwickelt haben, gibt es hierfür entsprechend viele Tools, um solche Tests zu entwickeln [58, 59]. Einige Beispiele hierfür sind PIT[1] und Judy[2] für Java- bzw. Milu[3] für C-Programme [60–62].

Ein weiteres, sehr einfach zu nutzendes Tool zum Entwickeln von Mutationstests ist der Universalmutator[4] [59]. Er kann zum Entwickeln von Mutationstests nicht nur innerhalb einer bestimmten Umgebung bzw. Programmiersprache, sondern prinzipiell für alle Programmiersprachen eingesetzt werden. Dies wird dadurch ermöglicht, da der Universalmutator den Quellcode der Programme verändert, und hierbei einen oder mehrere Regex-basierte Regelsätze anwendet. So kann vom Universalmutator Quellcode u. a. in den Sprachen Python, Java, C/C++ oder Swift mutiert werden [59].

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

[1] `http://pitest.org/`
[2] `http://mutationtesting.org/`
[3] `https://github.com/yuejia/Milu/`
[4] `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 [59]. Diese Validierung kann mithilfe von entsprechenden Startparametern durch ein benutzerdefiniertes Programm durchgeführt oder alternativ nicht durchgeführt werden [59, 63].

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 (vgl. Abschnitt 3.1). 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 (vgl. Unterabschnitt 2.3.2):

1. Zur Ermittlung der Veränderung des MARP-Wertes muss zunächst der jeweils aktuelle Arbeitsspeicher-Bedarf des Clusters eingelesen werden. Dies geschieht im `Controller` mithilfe einer `for`-Schleife, mit deren Hilfe der Speicherverbrauch im Cluster nahezu sekündlich aus der `memLog`-Datei eingelesen wird. Der Mutant 1 verändert hierbei die Schleifenbedingung, damit die Schleife nicht ausgeführt wird. Dadurch wird verhindert, dass der Speicherverbrauch des Clusters vom `Controller` der Selfbalancing-Komponente eingelesen und verwendet werden kann. Dadurch ist der Speicherverbrauch des Clusters auch nicht für den Algorithmus [7] 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 2 sorgt dafür, dass anstatt des korrekten Dateipfades der Bash-Shell (`/bin/bash`) ein ungültiger Dateipfad (hier `%bin/bash`) aufgerufen wird, womit das zur Übertragung des neuen Wertes benötigte Script nicht ausgeführt 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 Feh-

lermeldung nicht `null` ist, also noch weiteren Text enthält. Der Mutant 3 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. Dadurch steht die Anzahl der YARN-Jobs nicht zur Berechnung des MARP-Wertes zur Verfügung.

4. Die Klasse `MemUtilization` dient analog zur `ControlNodeMonitor`-Klasse zum Auslesen des Arbeitsspeicher-Bedarfs des Clusters. Der Mutant 4 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, der Speicherverbrauch somit nicht ausgelesen, und somit auch nicht zur Berechnung des neuen MARP-Wertes zur Verfügung steht.

Als Resultat der Mutanten 1, 3 und 4 kann somit der neue MARP-Wert nicht berechnet werden bzw. im Falle von Mutant 2 der korrekt berechnete MARP-Wert nicht an das Cluster übertragen werden.

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

6.3 Auswahl der Testkonfigurationen

Die im in Unterabschnitt 3.3.2 beschriebenen Testkonfigurationen werden mithilfe verschiedener Variablen implementiert. Anhand dieser Konfiguration wurden dynamisch zur Laufzeit die entsprechenden Testfälle generiert.

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 6.5: Zur Definition einer Testkonfiguration relevante Felder

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

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

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

Listing 6.6: Ermittlung der für die Testkonfigurationen genutzten Basisseeds

Die beiden ermittelten Seeds 0xAB4FEDD und 0x11399D3 stellen somit die erste Variable einer Testkonfiguration dar.

Zur Festlegung der Werte zur generellen Wahrscheinlichkeiten zur Aktivierung bzw. Deaktivierung von Komponentenfehlern wurden über 20.000 mögliche Aktivierungen und Deaktivierungen mit verschiedenen generellen Wahrscheinlichkeiten und Auslastungsgraden der Nodes simuliert. Der dabei für alle Testkonfigurationen 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 bei allen Konfigurationen auf 4 festgelegt, da hierbei das Cluster eine ausreichende Größe (4 oder 6 Nodes, vgl. Unterabschnitte 4.4.2 und 6.1.2) 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 auszuführenden Testfälle wurde variiert, wodurch in einigen Konfigurationen 5 und in anderen 10 Testfälle auszuführende Testfälle definiert sind. Ebenso variiert wurde die Anzahl der simulierten Clients, die auf 2, 4 oder 6 Clients festgelegt wurde.

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 aller genutzten Testkonfigurationen, der ausgeführten Tests mit ihrer benötigten Ausführungszeit, sowie der Anzahl der dabei tatsächlich ausgeführten Testfälle, ist in Anhang F zu finden.

Bei der Ausführung der Tests zur Evaluation wurden Eingabedaten nicht vorab generiert, sondern während der Ausführung von den Anwendungen direkt generiert (vgl. Abschnitt 3.1 und Unterabschnitte 3.3.2, 5.3.3 und 6.1.2). Dies liegt darin begründet, da durch die Vorabgenerierung der MARP-Wert manipuliert werden würde, wodurch die Tests an Aussagekraft verlieren.

Dazu wurde die Mindestdauer für einen Testfall bei allen Konfigurationen auf 25 Sekunden festgelegt, da in diesem Zeitraum die meisten *kleinen* Anwendungen erfolgreich durchgeführt werden können, wenn sonst keine Anwendung aktiv ist. 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*[5] genutzt. Alle zur Ausführung der Testfälle der Fallstudie relevanten Methoden wurden zudem in der Klasse `CaseStudyTests` zusammengefasst, welche die Klasse `SimulationTests` zum Ausführen der 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,
4     double faultProbability, int hostsCount, int clientCount,
5     int stepCount, bool isMutated)
6  {
7    // write test case parameter to log
8
9    InitInstances();
10   var isFailed = false;
11   try
12   {
13     // Setup
14     StartCluster(hostsCount, isMutated);
15     Thread.Sleep(5000); // wait for startup
16
```

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

```
17      var simTest = new SimulationTests();
18      // save test case parameter to simTest
19
20      // Execution
21      simTest.SimulateHadoopFaults();
22    }
23    // catch exceptions and set isFailed=true
24    finally
25    {
26      // Teardown
27      StopCluster();
28      MoveCaseStudyLogs(/* test case parameter */);
29    }
30    Assert.False(isFailed);
31  }
```

Listing 6.7: Methode zur Ausführung der Tests der Fallstudie (gekürzt)

Das Starten und Beenden des Clusters 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(
15              seed, prob, hosts, clients, steps, isMut);
16  }
17
18  private IEnumerable<int> GetSeeds()
19  {
20    yield return 0xAB4FEDD;
21    yield return 0x11399D3;
22  }
```

Listing 6.8: Implementierung der Testkonfigurationen (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 bei Mutationstests (vgl. Unterabschnitt 4.4.1 und Abschnitt 6.2).

Hierbei werden nur Konfigurationen generiert, auf denen die in Abschnitt 6.3 genannten Bedingungen zutreffen, womit anstatt den möglichen 48 Testkonfigurationen 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 Testausführung, gemäß der Parameter der Testkonfiguration, wie folgt umbenannt:

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

Listing 6.9: 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 Test zudem den Nebeneffekt, dass im SSH-Log keine Daten von ausgeführten Anwendungen eines anderen Tests enthalten sind.

7 Evaluation der Ergebnisse

In Abschnitt 6.3 wurden 32 Testkonfigurationen ermittelt, welche die Basis für die 43 ausgeführten Tests bilden. Bei den meisten Konfigurationen wurde hierbei jeweils ein Test, bei sieben Konfigurationen mehrere Tests ausgeführt. Die Gründe für die mehrfache Testausführung einzelner Konfigurationen sind in diesem Kapitel im Rahmen der entsprechenden Auffälligkeiten bzw. Fehler beschrieben. Eine Übersicht aller Testkonfigurationen und der ausgeführten Tests findet sich in Anhang F.

Bei der Auswertung der Programmlogs der einzelnen Tests musste zudem beachtet werden, dass die jeweiligen Monitoring-Informationen nur Momentaufnahmen bilden. Vor allem bei umfangreicheren Testfällen werden vom RM viele Anpassungen vorgenommen, die aufgrund der Struktur eines Testfalls nicht erkannt werden können.

Zur Auswertung der Evaluation dienten vor allem die im YARN-Modell implementierten Constraints, die sich aus den in Abschnitt 3.2 definierten Anforderungen ergeben (vgl. Unterabschnitte 4.2.6, 4.2.8 und 6.1.1).

7.1 Statistische Kenndaten

Die Dauer aller Simulationen betrug ca. 4:17 Stunden, die gesamte Ausführungsdauer inkl. Starten und Beenden des Clusters bei den Tests betrug ca. 5:20 Stunden. Auffällig ist hierbei, dass die Mutationstests meist schneller abgeschlossen wurden als die korrespondierenden Tests ohne Mutationen (vgl. Unterabschnitt 7.5.1). Von den 290 zur Ausfürhung vorgesehenen Testfällen wurden nur 222 Testfälle (77 %) tatsächlich auch ausgeführt. Der Grund für den Abbruch von 14 Tests liegt zum Großteil im 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 (vgl. Unterabschnitt 4.2.9). Ein Test wurde aufgrund der zu geringen Anzahl an Submittern abgebrochen, was in Unterabschnitt 7.7.3 genauer erläutert wird.

Insgesamt wurden 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 die Aktivierung des Komponentenfehlers, der den Node komplett beendet (vgl. Unterabschnitt 4.2.3). Von allen aktivierten Komponentenfehlern wurden während der Simulationen 262 Komponentenfehler deaktiviert bzw. repariert, was eine Quote von 60 % ergibt. In 4 der ausgeführten Tests wurde jedoch kein einziger Komponentenfehler deaktiviert, weshalb die Tests 4, 5.1, 5.2 und 6 entsprechend frühzeitig abgebrochen wurden (vgl. Unterabschnitte 7.5.1 und 7.6.1).

Bei den 43 ausgeführten Tests wurden 408 Anwendungen im Cluster gestartet, von denen mit 204 rund die Hälfte erfolgreich, und damit vollständig, ausgeführt wurden, fehlgeschlagen 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 Unterabschnitt 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, 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 (vgl. Abschnitt 7.7). Das ergibt dadurch eine Quote von 46,5 % aller gestarteten Attempts, die nicht erfolgreich abgeschlossen werden konnten. Beim Monitoring wurden 3150 YARN-Container erkannt, was im Schnitt 7,72 Container je Anwendung bzw. 5,68 je Attempt ergibt. Da bei den zu startenden Anwendungen einige kleine und einige sehr ressourcenintensive Anwendungen enthalten sind (vgl. Unterabschnitt 5.2.1), kann sich die Anzahl der Container zwischen einzelnen Anwendungen jedoch sehr unterscheiden.

Vom Oracle wurden bei allen Tests 78.825 Constraints validiert, von denen 573 verletzt bzw. als ungültig validiert wurden (0,73 %). Die meisten verletzten 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 liegt vor allem darin, dass die Constraints für fehlgeschlagene Anwendungen auch in nachfolgenden Testfällen einer Testausführung entsprechend validiert wurden (vgl. Unterabschnitt 7.7.1). Dies resultiert in bis zu 34 ungültigen Constraints für fehlgeschlagene Anwendungen bei einzelnen Tests.

7.2 Zusammenfassung der Ergebnisse

Zusammenfassend lässt sich, basierend auf der vorhandenen Datenbasis der 43 ausgeführten Tests, 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 Unterabschnitt 3.2.1 definierten funktionalen Anforderungen an das Cluster selbst und der in Unterabschnitt 3.2.2 definierten Anforderungen an das gesamte Testsystem vollständig erfüllt. Die meisten dieser Anforderungen wurden mithilfe der definierten Constraints implementiert (vgl. Unterabschnitte 4.2.6, 4.2.8 und 6.1.1), wodurch diese Anforderungen bei jedem Testfall automatisch validiert werden konnten.

Vor allem die funktionale Anforderung, wonach alle Tasks vollständig ausgeführt, 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. Bei Attempts mit dem Exitcode -100 wird zudem die Anforderung, dass kein Task an defekte Nodes gesendet wird, verletzt (vgl. Unterabschnitt 7.7.1).

Bei der Validierung der Constraints ist es zudem vorgekommen, dass die Constraints der 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 zum Teil jedoch falscher Alarm, wodurch diese Anforderungen zu großen Teilen als erfüllt angesehen werden können (vgl. Unterabschnitt 7.8.1). Genauso verhält es sich bei den 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 (vgl. Unterabschnitt 7.5.2).

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 ergab, dass es durchaus möglich ist, dass die bei einem Test ausgeführten Anwendungen nicht ausreichen, damit sich der Wert verändert (vgl. Abschnitt 7.3). Dennoch war es anhand dieser Anforderung möglich, die in Abschnitt 6.2 implementierten Mutanten der Selfbalancing-Komponente, bis auf einige Besonderheiten, in den ausgeführten Tests zu erkennen (vgl. Abschnitt 7.4). Auch die Anforderung, dass die in Unterabschnitt 4.2.3 implementierten Komponentenfehler im realen Cluster injiziert und repariert werden, konnte nicht immer erfüllt werden (vgl. Unterabschnitt 7.6.3). In diesem Kontext zeigte sich aber, dass immer erkannt werden konnte, wenn keine weitere Rekonfiguration des Clusters möglich ist, womit diese Anforderung vollständig erfüllt wird (vgl. Abschnitt 7.6). Zudem konnte in einigen Fällen die Anforderung, dass mehrere Benchmark-Anwendungen gleichzeitig gestartet und ausgeführt werden können, nicht erfüllt werden (vgl. Unterabschnitt 7.7.2).

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 die vom Connector bereitgestellten Submitter zum Starten von Anwendungen nicht ausgereicht haben (vgl. Unterabschnitt 7.7.3).

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 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. Während 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 (auf drei Nachkommastellen gerundet). Eine Übersicht aller Tests findet sich in Anhang F.

Da der MARP-Wert in den Konfigurationen 1 und 7 bei der jeweils ersten Testausführung nicht verändert wurde, wurden beide Konfigurationen erneut ausgeführt. Hierbei wurde der MARP-Wert beim Test 7.2 mehrmals erhöht, bevor er im finalen Clusterstatus auf 1 gesetzt war. Bei der Konfiguration 1 wurde der Wert dagegen bei keiner der beiden Ausführungen verändert.

Die nicht durchgeführte Änderung des MARP-Wertes in Konfiguration 1 rührt sehr wahrscheinlich daher, dass nur im ersten der fünf Testfälle zwei Anwendungen gleichzeitig gestartet werden. Dadurch wurden in allen zehn ausgeführten Testfällen zusammengezählt nur acht Anwendungen gestartet, die Hälfte davon jeweils beim ersten Testfall. Da zudem vier der acht Anwendungen nur kleine Anwendungen (`randomtextwriter` und `pi`) sind, und diese entsprechend schnell abgeschlossen werden können, steht den wesentlich umfangreicheren 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 auch nicht durchgeführt wird.

Bei der Testkonfiguration 7 ist dies ähnlich, wobei die gesamte Last auf mehr Nodes verteilt werden kann. Der beim Test 7.2 deutlich veränderte MARP-Wert im Vergleich zum 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 das entsprechende Constraint verletzt.

7.4 Erkennung der Mutanten

Da die Selfbalancing-Komponente den MARP-Wert auf Basis 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 F).

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, 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 drei der vier Anwendungen fehlgeschlagen sind. Zudem stellt das ein Indiz dafür dar, dass der Mutant im Test 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 den unterschiedlichen Mutationsszenarien der Plattform Hadoop-Benchmark ausgeführt, bei denen jeweils einer der in Abschnitt 6.2 definierten Mutanten aktiv ist (vgl. Unterabschnitt 4.4.1). 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 auf jeden Fall geändert wird (vgl. Abschnitt 7.3).

Einige Ergebnisse der hierfür fünf 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 fehlgeschlagenen Anwendungen zwischen 1 und 3. Gemein haben alle Tests jedoch, dass der MARP-Wert bei allen fünf Tests nicht verändert wurde, womit alle Mutationen erkannt wurden. Damit kann festgestellt werden, dass jeder der vier in Abschnitt 6.2 definierten Mutanten durch das Testsystem erkannt wird.

7.5 Betrachtung der Komponentenfehler

Die Aktivierung und Deaktivierung der Komponentenfehler in einem Testfall hängt neben dem zur Berechnung benötigtem Basisseed vor allem von den zuvor ausgeführten Testfällen bzw. der Lastverteilung bei den zuvor ausgeführten Testfällen eines Tests ab (vgl. Unterabschnitt 4.2.4). 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 in so einem Fall jedoch nur der `NodeDead`-Fehler im Cluster injiziert (vgl. Unterabschnitt 4.2.3). 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 von der ausgeführten Testkonfiguration ab (eine Übersicht aller Testkonfigurationen und Tests findet sich in Anhang F). Im Vergleich zwischen korrespondierenden Konfigurationen, die sich nur in der Nutzung des Mutationsszenarios unterschieden, wurde nur bei 5 korrespondierenden Testkonfigurationen die gleiche Anzahl an Komponentenfehlern 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 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 Fehlern 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 dritten ausgeführten Testfall abgebrochen wurden (vgl. Unterabschnitte 4.2.9 und 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 Komponentenfehlern nur 7 Prozent 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 Komponentenfehlern 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. Dadurch wird auch die Funktionalität der Selfbalancing-Komponente von Zhang et al. bestätigt.

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 Komponentenfehlern 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. Abschnitt 8.1).

7.5.2 Nicht erkannte, injizierte bzw. reparierte Komponentenfehler

Bei 18 aller ausgeführten Tests ist aufgetreten, dass ein injizierter bzw. reparierter Komponentenfehler zunächst nicht vom Testsystem erkannt wurde. Das betraf konkret in drei Testfällen das Injizieren eines Komponentenfehlers sowie in 16 Testfällen 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. Eine Übersicht aller Tests findet sich in Anhang F.

Bei den aufgetretenen, verletzten 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, dass defekte Nodes und Verbindungsabbrüche erkannt werden (vgl. Abschnitt 3.2), wurden somit korrekt, als auch inkorrekt, als ungültig validiert. Dies liegt daran, dass das Cluster bei defekten Nodes erst einige Zeit benötigt, um den Ausfall eines Nodes zu erkennen. 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 (vgl. Abschnitt 2.2). Zwar wurden beide Zeitspannen in den genutzten Szenarien der Plattform Hadoop-Benchmark auf jeweils 10 Sekunden festgelegt (vgl. Unterabschnitt 4.4.1), jedoch reichte diese Zeitspanne wohl nicht immer aus, um den Status rechtzeitig zu erkennen. Beim Starten bzw. Wiederverbinden eines Nodes verhält es sich analog dazu, 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 weitere 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 erneut vom Cluster getrennt wird. In den Tests 23 bis 28.2 wird der Node ebenfalls im dritten Testfall wieder vom Cluster getrennt, was hier auch korrekt erkannt wird.

7.6 Analyse der Testabbrüche

Insgesamt 13 der 42 ausgeführten Tests wurden vorzeitig abgebrochen, da eine Rekonfiguration des Clusters nicht möglich war. Dies entspricht dem in Unterabschnitt 3.2.2 geforderten und in Unterabschnitt 4.2.9 implementierten Verhalten, wenn alle Nodes im Cluster aufgrund eines Komponentenfehlers defekt sind. Im Folgenden werden daher die Ursachen für die abgebrochenen Tests betrachtet (vgl. Anhang F).

7.6.1 Testkonfigurationen 3 bis 6

Erstmalig ist ein Abbruch im Test 4 aufgetreten, auch die weiteren korrespondierenden Tests der Konfigurationen 5 und 6 wurden abgebrochen. Hier waren bereits beim dritten ausgeführten Testfall alle verfügbaren Nodes beendet, was auffällig ist, da somit auch die Hälfte aller Tests mit dem ersten Seed und dem Cluster auf einem Host vorzeitig abgebrochen wurden. Das liegt primär darin begründet, 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 dritten ausgeführten

Testfall wieder zu aktivieren, während bei den anderen noch aktiven Nodes spätestens in diesem Testfall aufgrund der hohen Last ein Komponentenfehler injiziert wurde (vgl. Unterabschnitt 4.2.4). 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. Eine Übersicht aller Tests findet sich in Anhang F.

Bei den beiden betroffenen Tests 15 und 16 führte die sehr hohe Auslastung der noch aktiven Nodes im fünften Testfall daher im darauf folgenden Testfall dazu, dass bei allen noch aktiven Nodes ein Komponentenfehler injiziert 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 in Tabelle 7.4 gezeigt wird.

Die Aktivierung und Deaktivierung der Komponentenfehler bei den anderen Nodes ist in allen Tests gleich und daher zur Ermittlung der Gründe des Abbruchs der Testausführung nicht relevant. Durch die unterschiedliche Auslastung 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

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: Auslastung und Komponentenfehler in Node 1 der Tests 19 bis 22. Eine Übersicht aller Tests findet sich in Anhang F.

relevante Node 1 bei den Mutationsstests nicht gestartet wurde, während die anderen Nodes beendet wurden wie in den Tests 19 und 21.

Eine Besonderheit bildet hier zudem Test 21, bei dem der Komponentenfehler vom Testsystem deaktiviert wurde, jedoch nicht repariert werden konnte. Dies liegt darin, dass der Docker-Container nicht mit dem Docker-Netzwerk verbunden werden konnte. Aus diesem Grund wurde vom Oracle bei der Prüfung der Rekonfigurierbarkeit des Clusters der Test entsprechend beendet, da der Node nicht verbunden war. Zwar wurde der Fehler von Docker nicht absichtlich bzw. durch das Testsystem herbeigeführt, hat jedoch eine positive, als auch eine negative Seite. So wurde auch ein externer, nicht spezifizierter Fehler erkannt, jedoch auf Kosten der Anforderung, dass im Modell implementierte Komponentenfehler im realen Cluster repariert werden (vgl. Abschnitt 3.2).

7.6.4 Testkonfigurationen 27 und 28

In den beiden Tests 28.1 und 28.2 wird der Test im achten Testfall abgebrochen, während Test 27 nach allen 10 Testfällen regulär beendet wird. Das liegt daran, dass im achten 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 werden 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 Komponentenfehler unterschieden sich in den drei Tests bis auf einzelne, hier nicht relevante, Ausnahmen nicht.

Der Grund für die Injizierung von Komponentenfehlern bei noch allen aktiven Nodes im 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 (vgl. Unterabschnitt 4.2.4). Dadurch war es möglich, dass alle noch aktiven Nodes vom Cluster getrennt bzw. beendet wurden und der Test aufgrund fehlender Rekonfigurationsmöglichkeiten abgebrochen wurde (vgl. Unterabschnitt 4.2.9).

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, verfügbarer Submitter des Connectors beendet. Die Gründe dafür sind in Unterabschnitt 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 bei jeweils mehreren Testfällen 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 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 auf dem Cluster auszuführenden Anwendungen sind vor allem zwei Punkte aufgefallen: Viele Anwendungen wurden aufgrund von Fehlern beendet und einige Anwendungen, vor allem bei den Tests der Konfigurationen 17 bis 32 (vgl. Anhang F), 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 fehlgeschlagen, was im Schnitt 2,6 fehlgeschlagene Anwendungen pro ausgeführtem Test ergibt. Die meisten fehlgeschlagenen 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 fehlgeschlagen sind, ist dies bei der Ausführung des Tests 20 keine einzige. Ebenfalls auffallend ist, dass Konfigurationen mit Mutationsszenario meist weniger oder gleich viele fehlgeschlagene Anwendungen aufweisen als die korrespondierenden Konfigurationen ohne Mutationen. Eine Ausnahme bildet der Test 8, bei dem 3 Anwendungen fehlgeschlagen sind, während die Tests 7.1 und 7.2 keine fehlgeschlagene Anwendung aufweisen. Eine weitere Ausnahme bildet der Test 9.2, der eine fehlgeschlagene Anwendung mehr aufweist als Test 10.3. Die anderen Tests der Konfigurationen 9 und 10 verhalten sich jedoch wie andere korrespondierende Testkonfigurationen.

Bei der Betrachtung der Constraints, welche die in Unterabschnitt 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 verletzten Constraints mit 343 mehr als die Hälfte aller verletzten Constraints ausmacht (59,9 %). Im Schnitt ergibt das somit rund 8 ungültige Constraints pro Test bzw. ca. 1,8 verletzte Constraints pro durch das Oracle überprüften Testfall. Die auf den ersten Blick sehr hohe Anzahl an verletzten Constraints resultiert daraus, dass eine fehlgeschlagene Anwendung bei jedem nachfolgenden Testfall bei einer Testausführung erneut durch das Oracle entsprechend validiert wurde. Dadurch sind ein Großteil der verletzten Constraints ein falscher Alarm, da die entsprechende Anforderung pro Anwendung logischerweise 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 bzw. Attempts ausmachen:

- Der AppMstr ist nicht mehr erreichbar, da der auszuführende Node aufgrund eines injizierten Komponentenfehlers nicht mehr erreichbar ist. Dadurch wird der AppMstr nach einiger Zeit mit einem *AppMstr Timeout* als fehlgeschlagen 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 ebenfalls aufgrund eines *AppMstr Timeout*s abgebrochen. Das beinhaltet auch Timeouts, wenn einem AppMstr kein 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 trat bei den hier ausgeführten Tests mehrmals bei der Anwendung `TestDFSIO -read` auf, wenn die zuvor generierten Eingabedaten für diesen Benchmark aufgrund injizierter 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, bei denen ein Komponentenfehler injiziert wurde. Dies führte dazu, dass die benötigten Daten nicht gefunden werden können bzw. 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 tritt dann auf, wenn versucht wird, einen Task eines YARN-Containers auf einem defekten Node auszuführen, was somit auch der Anforderung widerspricht, dass kein Task an defekte Nodes gesendet wird (vgl. Unterabschnitt 3.2.1). Dieser Fehler trat nur dann auf, wenn im ausführenden Node des betroffenen AppMstr im gleichen Testfall ein Komponentenfehler injiziert wurde und der Node dadurch ausfiel. Aufgrund der mit dem Fehler verbundenen Fehlermeldung *„Container released on a *lost* node“* liegt die Vermutung nahe, dass die Tasks des YARN-Containers, hier wahrscheinlich des 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 fehlgeschlagen abgebrochen, sobald zwei Attempts aufgrund eines Timeouts abgebrochen wurden. Wenn ein Attempt mit dem Exitcode -100 terminiert, wird unabhängig von zuvor ausgeführten Attempts ein weiterer Attempt ausgeführt. Dadurch wird hier die Anforderung, dass ein Task vollständig ausgeführt werden muss, teilweise erfüllt.

Bei einigen der AppMstr-Timeouts aufgrund der Injizierung von Komponentenfehlern 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 abgebrochen 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 noch während der Aktivierung von Komponentenfehlern innerhalb des Testfalls auf dem ausführenden Node des neuen AppMstr ein Komponentenfehler injiziert wird. Dadurch wird der zweite AppMstr ebenfalls aufgrund eines Timeouts abgebrochen, wodurch die gesamte Anwendung als fehlgeschlagen gilt.

7.7.2 Nicht gestartete Anwendungen

Bei den Tests 19, 25 und 27 bis 32 (vgl. Anhang F) konnten insgesamt 29 Anwendungen nicht gestartet werden. Meistens war hiervon die Anwendung `terasort` davon betroffen, einige Male auch die Anwendung `teravalidate`. Ursächlich dafür ist die jeweils hohe Auslastung des Clusters in den Testfällen zuvor, bei denen den AppMstr-Containern der `teragen`-Anwendungen keine Ressourcen auf den ausführenden Nodes allokiert werden konnten und diese daher aufgrund eines AppMstr-Timeouts beendet wurden (vgl. Unterabschnitt 7.7.1). Da für die ausgeführten Tests 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. Unterabschnitt 5.2.1 und Abschnitt 6.3). 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, wonach mehrere Benchmark-Anwendungen gleichzeitig gestartet und ausgeführt werden können, entsprechend verletzt, da hier geprüft wurde, ob alle Clients eine derzeit ausgeführte Anwendung aufweisen (vgl. Unterabschnitte 3.2.2 und 4.2.3). Analog dazu verhält es sich bei der Anwendung `teravalidate`, welche wiederum die `terasort`-Ausgabedaten als Eingabedaten benötigt (vgl. Unterabschnitt 5.2.1).

7.7.3 Nicht ausreichend Submitter

Ein unerwarteter Fehler trat bei der Ausführung des Tests 31.1 auf. Hierbei waren die für die anderen Tests genutzten acht Submitter des Connectors zum Starten von Anwendungen nicht ausreichend. Dadurch wurde der Test im achten Testfall abgebrochen, da keine freien Submitter mehr verfügbar waren (vgl. Unterabschnitt 4.3.3). Daher wurde der Test zur Konfiguration 31 mit zehn Submittern erneut ausgeführt, wodurch dieser hierbei aufgrund fehlender Möglichkeiten zur weiteren Rekonfiguration abgebrochen wurde (vgl. Unterabschnitt 7.6.5). 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 noch belegt waren.

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 Unterabschnitt 3.2.2 definierte Anforderung an das Testsystem, dass der jeweils aktuelle Status des Clusters erkannt und im Modell gespeichert werden muss. Hierbei muss zwischen zwei unterschiedlichen Fällen unterschieden werden, 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, der aktuelle Status im Cluster erkannt, und dieser im Testmodell gespeichert werden muss. Hierbei muss jedoch unterschieden werden, dass für gestartete Anwendungen und Attempts, deren AppMstr jedoch noch kein ausführender Node zugeteilt wurde, es das normale und logische Verhalten von Hadoop ist, hier keine Nodes auszugeben. Wenn dieser Status zu lange anhält, wurden die Attempts bzw. AppMstr durch Hadoop mit einem Timeout beendet (vgl. Unterabschnitt 7.7.1).

Anders sieht das jedoch in den sechs Tests 7.1, 8 und 23 bis 26 aus (vgl. Anhang F). In diesen Tests wurden zwar regulär die Daten der Nodes ermittelt und auch in den Logdateien gespeichert, jedoch nicht alle ausführenden Nodes der Anwendungen und Attempts. Konkret betrifft das hier die beiden auf Host 2 ausgeführten Nodes. 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 ebenfalls übertragen, sofern den Attempts bzw. Anwendungen ein Node zugewiesen wurde, jedoch wurden diese nicht im Programmlog gespeichert. Zwar tritt hierbei ein gewisses Muster auf (pro Seed die jeweils zuerst ausgeführten Tests mit Nodes auf beiden Hosts), allerdings konnte dieser Fehler nicht gezielt reproduziert werden. Bei der erneuten Ausführung der Testkonfiguration 7 in Form des Tests 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. Unterabschnitt 4.3.2). 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 Fehlende Diagnostik-Daten von Anwendungen

Bei allen Tests ist zudem aufgefallen, dass die Diagnostik-Daten von Anwendungen nicht im Programmlog enthalten sind. Genauso wie bei den nicht erkannten Nodes auf Host 2 (vgl. Unterabschnitt 7.8.1) wurden alle Diagnostik-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 Anforderung an das Testsystem nur teilweise erfüllt, wonach der jeweils aktuelle Status des Clusters erkannt und gespeichert wird (vgl. Unterabschnitt 3.2.2).

Eine Analyse ergab, dass die Diagnostik-Daten der Anwendungen aufgrund eines falsch gesetzten Attributs in der `ApplicationResult`-Klasse des Parsers nicht im Testsystem gespeichert werden konnten. Dadurch konnten die Daten nicht mithilfe von Json.NET dem korrekten Attribut zugeordnet werden, wodurch die Diagnostik-Daten entsprechend verworfen wurden (vgl. Unterabschnitt 4.3.2). Da die Diagnostik-Daten der Anwendungen eine Zusammenfassung der gesamten Anwendung darstellen, und alle Diagnostik-Daten bereits durch die der Attempts vorhanden waren, wurde hier auf erneute Testausführungen verzichtet.

8 Reflexion und Abschluss

Die Evaluation der ausgeführten Tests hat gezeigt, dass sich das entwickelte Testsystem und das Hadoop-Cluster im Großen und Ganzen so verhalten haben, wie es erwartet wurde (vgl. Abschnitt 7.2). Dennoch wurden auffällig viele Anwendungen aufgrund verschiedener Fehler beendet (vgl. Abschnitt 7.7), was auf Probleme im Testsystem oder der Auswahl der Testkonfiguration hinweist. Es hat sich aber dennoch gezeigt, dass das entwickelte Testsystem mit entsprechenden Tests durchaus in der Lage ist, ein selbstadaptives Load-Balancing-System automatisiert testen zu können.

8.1 Diskussion der Ergebnisse der Fallstudie

Über die bei der Evaluation ermittelten Ergebnisse der ausgeführten Tests lässt sich natürlich nun diskutieren, wie aussagekräftig diese sind. Dies liegt vor allem aufgrund der hohen Anzahl an fehlgeschlagenen Anwendungen, für die sich folgende vier Ursachen herausgestellt haben (vgl. Unterabschnitt 7.7.1):

- Der Node ist nicht erreichbar, was zu einem AppMstr-Timeout führt
- Der den AppMstr ausführende Node ist zur sehr ausgelastet, was ebenfalls zu einem AppMstr-Timeout führt
- Benötigte Dateien sind nicht im HDFS verfügbar, was in einem Map-Task-Fehler resultiert
- Es wird versucht, Tasks von YARN-Containern auf einem defekten Node auszuführen, was im Exitcode -100 resultiert

Es ist erkennbar, dass drei der vier Ursachen im Zusammenhang mit der Injizierung von Komponentenfehlern im realen Cluster zu finden sind. Dies zeigt sich vor allem an der häufigen Zahl an AppMstr-Timeouts, welche bei einem injiziertem Komponentenfehler die direkte Folge davon ist, wenn auf dem Node ein AppMstr ausgeführt wurde. Es hat sich daher gezeigt, dass die in Abschnitt 6.3 ausgewählte generelle Wahrscheinlichkeit zur Aktivierung und Deaktivierung der Komponentenfehler von 0,3 vor allem für hohe Auslastungsgrade bei konkreten Testausführungen eine nicht sehr ausgewogene Mischung ergibt. Während bei niedrigen Auslastungsgraden nur wenige Komponentenfehler injiziert wurden, wurden bei hohen Auslastungsgraden häufig welche injiziert, was sich auch in der Anzahl der 14 abgebrochenen Tests zeigt (vgl. Abschnitt 7.6).

Mit 14 Prozent aller möglichen Komponentenfehler wurden zwar nicht viele Fehler aktiviert. Jedoch muss man bei diesem Wert beachten, dass für jeden Node pro Testfall bis zu zwei Komponentenfehler aktivierbar sind. Aus diesem Grund wurden nicht

14 Prozent der möglichen Defekte ausgelöst, sondern rund ein Viertel. Das ergibt umgerechnet pro ausgeführtem Testfall im Schnitt mindestens einen Node, bei dem ein Komponentenfehler injiziert und somit ein Defekt ausgelöst wurde. Die Quote von rund einem Viertel defekter Nodes dürfte im Praxisbetrieb jedoch eher unwahrscheinlich sein. Da viele in der Praxis eingesetzten Cluster eine zwei-, drei- oder zum Teil auch eine vierstellige Zahl an Nodes aufweisen [6], müssten hier für eine ähnliche Quote bis zu mehrere hundert Nodes gleichzeitig ausfallen. Ein Beispiel mit den Daten des Musik-Streaming-Dienstes Spotify[1] verdeutlicht hierbei die Tragweite eines solchen Ausfalls. Das von Spotify genutzte Cluster zur Analyse der Hörgewohnheiten und für Musikempfehlungen für seine Benutzer und Kunden weist folgende Kenndaten auf [6]:

- 1.650 Nodes
- 43.000 virtualisierte Kerne
- 70 TB Arbeitsspeicher
- 65 PB Speicherplatz
- mehr als 20.000 täglich auf dem Cluster ausgeführte Jobs

Es stellt damit auch eines der größten Hadoop-Cluster dar [6]. Es ist schnell ersichtlich, dass beim Ausfall von über 400 Nodes wohl ein Großteil der über 20.000 ausgeführten Anwendungen nicht mehr ausgeführt werden könnten und zahlreiche Blöcke von möglicherweise benötigten Daten im HDFS nicht mehr verfügbar wären (vgl. Abschnitt 2.2). Entsprechende Auswirkungen hätte das auch auf die Kunden von Spotify, da das Cluster u. a. zur Ermittlung von Musikempfehlungen genutzt wird, welche dann nur noch eingeschränkt verfügbar wären. Es lässt sich daher auch sagen, dass der Ausfall eines Viertels aller Nodes lediglich bei sehr kleinen Clustern im ein- bzw. niedrigen zweistelligen Bereich realitätsnah sein dürfte.

Daher lässt sich die hohe Anzahl an fehlgeschlagenen Anwendungen wohl vor allem an der generellen Wahrscheinlichkeit zur Aktivierung der Komponentenfehlern festmachen. Es wäre daher durchaus sinnvoll gewesen, einige der ausgeführten Tests auch ohne Aktivierung der Komponentenfehler durchzuführen. So hätte besser festgestellt werden können, ob die hohe Anzahl an fehlgeschlagenen Anwendungen an zu vielen injizierten Komponentenfehlern liegt oder ob der Grund im Testsystem selbst liegt. Da diese Tests im Rahmen dieser Fallstudie nicht durchgeführt wurden, wären hierfür entsprechend weitere Tests nötig.

Jedoch hätten selbst dadurch u. U. nicht alle fehlgeschlagenen Anwendungen unterbunden werden können. Unabhängig von der Anzahl der Nodes gab es zahlreiche Testfälle, bei denen das Cluster vollständig ausgelastet war und keine weiteren AppMstr ausgeführt werden konnten. In solchen Fällen wurden einzelne AppMstr häufig mit einem Timeout beendet, da die benötigten Ressourcen nicht allokiert werden konnten.

[1] `https://www.spotify.com/`

Da es mit `TestDFSIO -write`, `TestDFSIO -read` und `pentomino` zudem einige Benchmarks gab, welche bereits bei den ausgeführten Tests unabhängig von der Anzahl der aktiven Nodes das komplette Cluster auslasteten, hätte es hier wahrscheinlich dennoch entsprechende Timeouts gegeben. Jedoch hätten sich die AppMstr-Timeouts darauf beschränkt, dass keine Ressourcen allokiert werden konnten, was eine deutlich geringere Zahl gewesen wäre.

Durch eine Testausführung ohne Aktivierung der Komponentenfehler hätten vermutlich auch viele der 29 nicht gestarteten Anwendungen (vgl. Unterabschnitt 7.7.2) gestartet werden können. Mit 7 Prozent ist der Anteil der nicht gestarteten Anwendungen im Verhältnis zu allen gestarteten Anwendungen zwar kein signifikanter Anteil, wobei dieser aber auch nicht zu vernachlässigen ist. Zwar hätten die nicht gestarteten Anwendungen keine signifikant höhere Auslastung des Clusters bedeutet, aber dennoch wurde hiermit auch eine in Abschnitt 3.2 nicht erwähnte Anforderung verletzt, wonach alle Anwendungen der ausgewählten Benchmarks auch gestartet werden müssten. Da hierbei immer zu `terasort` zugehörige Anwendungen betroffen waren, gab es hierfür auch zwei Ursachen. Die erste war eine zu hohe Auslastung des Clusters, wodurch die benötigten Daten von der `teragen`-Anwendung nicht generiert werden konnten, da die Anwendung keine Ressourcen zur Ausführung des AppMstr erhalten hat oder die Anwendung auch deshalb vorab abgebrochen wurde. Die zweite Ursache liegt in der Injizierung der Komponentenfehler, wodurch erfolgreich generierte Daten für die nachfolgenden Anwendungen nicht mehr verfügbar waren. Es ist daher stark davon auszugehen, dass sich die Anzahl der nicht gestarteten Anwendungen bei einer Testausführung ohne Aktivierungen von Komponentenfehlern signifikant reduzieren würde.

Die Ausführungsdauer der einzelnen Tests ist zudem ebenfalls nicht zu vernachlässigen. Aufgrund der Ergebnisse lässt sich sagen, dass mehr injizierte Komponentenfehler auch die Ausführungsdauer erhöht haben. Da die Ursache für mehr injizierte Komponentenfehler eine höhere Clusterauslastung ist, lässt sich auch ein Zusammenhang mit der Selfbalancing-Komponente [7] herstellen. Dies liegt daran, dass Mutationstests im Vergleich meist die Tests waren, bei denen weniger Komponentenfehler aktiviert wurden (vgl.Unterabschnitt 7.5.1). Die Ursache hierfür liegt in der geringeren Auslastung des Clusters, die Auswirkung hiervon ist durch weniger injizierte Komponentenfehler jedoch auch eine schnellere Ausführung der Tests gewesen. Zudem reduziert sich durch weniger injizierte Komponentenfehler der Verwaltungsaufwand durch ausfallende Nodes. Um diese Vermutung zu bestätigen, wären jedoch noch weitere Tests nötig. Dennoch lässt sich auch schlussfolgern, dass mithilfe der Selfbalancing-Komponente das Cluster besser ausgelastet werden kann als mit den Standardeinstellungen. Dadurch bestätigt diese Fallstudie auch die Evaluation [7] zur Performance eines Clusters, bei dem die Selfbalancing-Komponente genutzt wird.

8.2 Bewertung und Ausblick

Im Rahmen dieser Masterarbeit sollte vor allem ermittelt werden, ob eine Testautomatisierung eines selbstadaptiven Load-Balancing-Systems mithilfe eines modellbasierten Ansatzes möglich ist (vgl. Kapitel 3). Basierend auf den Ergebnissen dieser Fallstudie lässt sich sagen, dass dies mit dem entwickelten Testsystem und dem Testframework S# möglich ist. Alle in Abschnitt 3.2 definierten Anforderungen, die automatisiert geprüft werden konnten, wurden entsprechend implementiert, wodurch auch die Auswertung der Testfälle zu großen Teilen automatisiert verlief.

Jedoch konnten nicht alle Anforderungen vollständig automatisiert werden, wie z. B. die Anforderung, dass ein Test vollautomatisiert durchgeführt werden kann. Diese Anforderung musste manuell validiert werden, womit es auch an das Halteproblem [64, 65] erinnert, wonach durch einen Algorithmus nicht entschieden werden kann, ob ein anderer Algorithmus terminiert. Die Anforderung konnte hierbei insofern validiert werden, als dass die Durchführung eines Tests vollautomatisch durchgeführt werden konnte. Sollten jedoch mehrere Tests direkt hintereinander ausgeführt werden, traten bei der Ausführung der Tests in einigen Fällen Problemen auf, wenn der zuvor ausgeführte Test aufgrund der fehlenden Möglichkeit zur weiteren Rekonfiguration regulär abgebrochen, dabei jedoch nicht die noch auf dem Cluster ausgeführten Anwendungen abgebrochen wurden (vgl. Unterabschnitt 6.1.1 und Abschnitt 7.2). Dadurch bestand die Möglichkeit, dass bei nachfolgenden Tests einige Submitter des Connectors aufgrund der noch ausgeführten Anwendungen belegt waren (vgl. Unterabschnitt 4.3.3), und der Test letztlich aufgrund fehlender, noch freier Submitter abgebrochen wurde (vgl. Abschnitt 7.2 und Unterabschnitt 7.7.3). Um dies zu verhindern, mussten die Docker-Container der noch ausgeführten Benchmark-Anwendungen manuell beendet werden. Dadurch wurde die automatische Ausführung aller Tests ermöglicht, sofern für die Ausführung eines Tests ausreichend Submitter zur Verfügung standen (vgl. Unterabschnitt 7.7.3).

Ähnlich verhält es sich mit der Anforderung, dass Tests und Testfälle zeitlich unabhängig und mehrmals ausgeführt werden können. Für diese Anforderungen kann aufgrund der Evaluation jedoch entschieden werden, dass diese Anforderungen insofern erfüllt werden, als dass ein Test zeitlich unabhängig unter den gleichen Bedingungen gestartet werden kann, die einzelnen Testfälle jedoch unterschiedlich sein können. Dies liegt daran, dass die ausgeführten Testfälle während der Testausführung dynamisch und basierend auf der Testkonfiguration, den zuvor ausgeführten Testfällen, sowie dem Verhalten des Clusters generiert werden (vgl. Abschnitte 6.3 und 7.2).

Neben den Problemen, dass einige Anwendungen nicht immer beendet wurden, wenn ein Test abgebrochen wurde, gibt es aber auch noch weitere Punkte, bei denen das entwickelte Testsystem nach den Erfahrungen der ausgeführten Tests optimiert werden kann. Eine Möglichkeit wäre z. B. die vermehrte Nutzung von mehreren Threads für

einzelne Abläufe innerhalb der Simulation bzw. des Treibers, wie beim Zugriff auf das Cluster. Problematisch könnte hierbei jedoch sein, dass Multithreading von S# nicht unterstützt wird (vgl. Abschnitt 2.1). Hierfür müsste daher geprüft werden, an welcher Stelle entsprechende Optimierungen möglich wären.

Ebenso könnte die SSH-Verbindung selbst analog zu den Parsern und Connectoren (vgl. Unterabschnitt 4.3.1) besser gekapselt werden, z. B. in Form eines hierfür definierten Interfaces. Auch könnte die Initialisierung des Treibers anstatt über den erstmaligen Aufruf der jeweiligen Singletons möglicherweise mithilfe des Factory-Patterns durchgeführt werden.

Was einen wichtigen, aber diskutablen, Punkt für mögliche Optimierungen im Testsystem darstellt, ist das Starten von Anwendungen. Hierfür werden im entwickelten Testsystem vom Connector mehrere gemeinsam genutzte SSH-Verbindungen bereit gestellt (vgl. Unterabschnitt 4.3.3). Da bei der Ausführung der Tests jedoch Probleme aufgrund einiger nicht korrekt beendeter Anwendungen aufgetreten sind, waren nicht in jedem Test ausreichend Submitter vorhanden bzw. deren Anzahl musste erhöht werden (vgl. Unterabschnitt 7.7.3). Eine Lösungsmöglichkeit hierfür wäre, jedem Client einen eigenen Submitter bereitzustellen. Damit einhergehend müsste jeder Client auch selbst dafür sorgen, dass Anwendungen beendet werden, wenn eine andere Anwendung gestartet werden soll (vgl. Unterabschnitt 4.2.7). Jedoch hätte das auch den Nachteil, dass ein Client durch eine nicht zu beendende Anwendung keine weiteren Anwendungen ausführen könnte, was mit der derzeitigen Architektur des Modells jedoch möglich ist, sofern ausreichend Submitter zur Verfügung stehen.

Weitere Tests könnten zudem dazu durchgeführt werden, um die Gründe für die nicht erkannten Nodes auf Host 2 zu ermitteln und die Ursache hierfür zu beheben. Dies wäre vor allem dadurch nötig, da das Testsystem bei der Validierung der Constraints auch einen falschen Alarm ausgelöst hat, und dadurch die entsprechenden Constraints verletzt wurden, obwohl die Daten vom Cluster an das Testsystem übertragen wurden. Daher kann derzeit nur vermutet werden, dass die betroffenen Nodes im Modell aufgrund der Architektur der Konvertierung der Nodes in den Parsern nicht ermittelt werden konnten (vgl. Unterabschnitt 4.3.2). Als problematisch könnte sich hierbei jedoch erweisen, dass dieser Fehler bei weiteren Tests in dieser Fallstudie nicht reproduziert werden konnte (vgl. Unterabschnitt 7.8.1).

Ein spezieller Fall bildet das Fehlen der Diagnostik-Daten von ausgeführten Anwendungen im Programmlog. Hierfür hat sich nach einer Analyse herausgestellt, dass nicht fehlerhafte Daten des Clusters ursächlich waren, sondern ein falsch gesetztes Attribut im Parser (vgl. Unterabschnitt 7.8.2). Dies hätte bereits bei Vorabtests, wie den durchgeführten Unit-Tests der beiden implementierten Parser, festgestellt und entsprechend korrigiert werden müssen.

Erwähnenswert ist zudem, dass bei der Recherche für diese Masterarbeit mehrmals aufgefallen ist, dass Hadoop oftmals für unpassende Aufgaben genutzt wird [19, 52]. Das

bestätigt auch die Studie [51], wonach viele der mithilfe von Hadoop durchgeführten Aufgaben auch von „traditionellen Plattformen“ durchgeführt werden könnten. Dadurch sei auch unklar, weshalb Hadoop in vielen Fällen überhaupt genutzt werde. Als logische Folge erwähnt Apache selbst ebenfalls, dass Hadoop nicht „die Silberkugel [ist,] um alle Anwendungs- oder Datencenter-Probleme zu lösen“, sondern entwickelt wurde, um einige spezifische Probleme einiger Firmen und Organisationen zu lösen [66].

Abschließend lässt sich sagen, dass das entwickelte Testsystem für weitere Tests mit Hadoop als konkretem Load-Balancing-System als geeignet erscheint. Das ist nicht nur auf weiterführende Tests für diese Fallstudie bezogen (vgl. Abschnitt 8.1 und Unterabschnitt 7.5.1), sondern auch auf weitere Tests mit Bezug auf Hadoop, die unabhängig von der durchgeführten Fallstudie sind. Abhängig von der Art der Tests müsste das Testsystem hierfür angepasst werden. Dafür gibt es mit den Einstellungsmöglichkeiten für das Modell bzw. die Simulation selbst (vgl. Unterabschnitt 6.1.2) oder der Plattform Hadoop-Benchmark (vgl. Abschnitte 2.4 und 4.4) bereits einige Möglichkeiten, ohne das Testsystem selbst verändern zu müssen, was jedoch auch möglich wäre.

Literatur

[1] M. Polo et al. „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 et al. „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] B. Eberhardinger, A. Habermaier und W. Reif. „Toward Adaptive, Self-Aware Test Automation“. In: *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*. Mai 2017, S. 34–37. DOI: 10.1109/AST.2017.1.

[5] Shang-Wen Cheng. „Rainbow: Cost-effective Software Architecture-based Self-adaptation“. AAI3305807. Diss. Pittsburgh, PA, USA: Carnegie Mellon University, 2008. ISBN: 978-0-549-52525-7.

[6] zuletzt bearbeitet von XingWang. *Powered by Apache Hadoop*. 5. Apr. 2018. URL: https://wiki.apache.org/hadoop/PoweredBy (besucht am 18. 07. 2018).

[7] Bo Zhang et al. „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.

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

[9] 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.

[10] Axel Habermaier. *S# Wiki - Home*. 13. Mai 2016. URL: https://github.com/isse-augsburg/ssharp/wiki/Models (besucht am 17. 07. 2018).

[11] Benedikt Eberhardinger et al. „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.

[12] Axel Habermaier und Johannes Leupolz. *S# Wiki - Home*. 15. Mai 2017. URL: https://github.com/isse-augsburg/ssharp/wiki (besucht am 17. 07. 2018).

[13] Axel Habermaier. *S# Wiki - Model Checking*. 10. Mai 2016. URL: https://github.com/isse-augsburg/ssharp/wiki/Model-Checking (besucht am 30. 05. 2018).

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

[15] Jeffrey Dean und Sanjay Ghemawat. „MapReduce: Simplified Data Processing on Large Clusters“. In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, S. 137–150. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[16] 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.

[17] 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).

[18] Kyong-Ha Lee et al. „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.

[19] Vinod Kumar Vavilapalli et al. „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.

[20] 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).

[21] 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).

[22] 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).

[23] Apache Software Foundation. *Hadoop Cluster Setup*. 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-common/ClusterSetup.html (besucht am 10. 07. 2018).

[24] 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).

[25] 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).

[26] 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).

[27] 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).

[28] 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).

[29] K. Shvachko et al. „The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. Mai 2010, S. 1–10. DOI: 10.1109/MSST.2010.5496972.

[30] 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).

[31] 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).

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

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

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

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

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

[37] 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.

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

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

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

[41] Ron Petrusha, olprod und Saisang Cai. *Random Class*. URL: https://docs.microsoft.com/de-de/dotnet/api/system.random?view=netframework-4.7.2 (besucht am 13. 07. 2018).

[42] *Class Instant*. URL: https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html (besucht am 21. 07. 2018).

[43] Ron Petrusha, olprod und Saisang Cai. *Benutzerdefinierte Formatzeichenfolgen für Datum und Uhrzeit*. 30. März 2017. URL: https://docs.microsoft.com/de-de/dotnet/standard/base-types/custom-date-and-time-format-strings (besucht am 10. 07. 2018).

[44] Holger Schwichtenberg. *Kommentar: .NET Core 2.0 kann zwar mehr, aber es gibt immer noch gravierende Lücken*. 16. Aug. 2017. URL: https://heise.de/-3803583 (besucht am 12. 07. 2018).

[45] David Bailey, Peter Borwein und Simon Plouffe. „On the rapid computation of various polylogarithmic constants“. In: *Mathematics of Computation of the American Mathematical Society* 66.218 (1997), S. 903–913. DOI: https://doi.org/10.1090/S0025-5718-97-00856-9.

[46] S. Huang et al. „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.

[47] 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.

[48] 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).

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

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

[51] 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).

[52] Kai Ren et al. „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.

[53] Ralf Korn. *Monte Carlo methods and models in finance and insurance*. Hrsg. von Elke Korn und Gerald Kroisandt. Chapman & Hall/CRC financial mathematics series. A Chapman & Hall book. Boca Raton [u.a.]: CRC Press, 2010, : graph. Darst. ISBN: 9781420076189.

[54] Christiane Lemieux. *Monte Carlo and Quasi-Monte Carlo Sampling*. 1. Aufl. New York, NY: Springer-Verlag New York, 2009. ISBN: 9780387781655.

[55] Solomon W. Golomb. *Polyominoes*. Rev. and expanded 2. ed. Princeton science library. Princeton, N. J.: Princeton Univ. Press, 1995. ISBN: 9780691024448.

[56] R. A. DeMillo, R. J. Lipton und F. G. Sayward. „Hints on Test Data Selection: Help for the Practicing Programmer“. In: *Computer* 11.4 (Apr. 1978), S. 34–41. ISSN: 0018-9162. DOI: 10.1109/C-M.1978.218136.

[57] R. G. Hamlet. „Testing Programs with the Aid of a Compiler“. In: *IEEE Transactions on Software Engineering* SE-3.4 (Juli 1977), S. 279–290. ISSN: 0098-5589. DOI: 10.1109/TSE.1977.231145.

[58] Y. Jia und M. Harman. „An Analysis and Survey of the Development of Mutation Testing“. In: *IEEE Transactions on Software Engineering* 37.5 (Sep. 2011), S. 649–678. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.62.

[59] Alex Groce et al. „An Extensible, Regular-expression-based Tool for Multi-language Mutant Generation“. In: ICSE '18 (2018), S. 25–28. DOI: 10.1145/3183440.3183485. URL: http://doi.acm.org/10.1145/3183440.3183485.

[60] Henry Coles et al. „PIT: A Practical Mutation Testing Tool for Java (Demo)“. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: ACM, 2016, S. 449–452. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2948707. URL: http://doi.acm.org/10.1145/2931037.2948707.

[61] L. Madeyski und N. Radyk. „Judy - a mutation testing tool for java“. In: *IET Software* 4.1 (Feb. 2010), S. 32–42. ISSN: 1751-8806. DOI: 10.1049/iet-sen.2008.0038.

[62] Y. Jia und M. Harman. „MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language“. In: *Testing: Academic Industrial Conference - Practice and Research Techniques (taic part 2008)*. Aug. 2008, S. 94–98. DOI: 10.1109/TAIC-PART.2008.18.

[63] 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).

[64] A. M. Turing. „On Computable Numbers, with an Application to the Entscheidungsproblem“. In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), S. 230–265. DOI: 10.1112/plms/s2-42.1.230. URL: http://dx.doi.org/10.1112/plms/s2-42.1.230.

[65] A. M. Turing. „On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction“. In: *Proceedings of the London Mathematical Society* s2-43.1 (1938), S. 544–546. DOI: 10.1112/plms/s2-43.6.544. URL: http://dx.doi.org/10.1112/plms/s2-43.6.544.

[66] zuletzt bearbeitet von SteveLoughran. *What Hadoop is Not*. 10. Juli 2013. URL: https://wiki.apache.org/hadoop/HadoopIsNot (besucht am 21. 07. 2018).

A CLI-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 Attempts, 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 hier aufgeführten Befehlen sowie die vollständige Befehlsreferenz sind in der Dokumentation von Hadoop [24] zu finden.

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

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

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

```
$ yarn application --status application_1518100641776_0001
...
Application Report :
    Application-Id : application_1518100641776_0001
    Application-Name : QuasiMonteCarlo
    Application-Type : MAPREDUCE
    User : root
    Queue : default
    Start-Time : 1518103712160
    Finish-Time : 1518103799743
```

```
11     Progress : 100%
12     State : FINISHED
13     Final-State : SUCCEEDED
14     Tracking-URL : http://controller:19888/jobhistory/job/
          job_1518100641776_0001
15     RPC Port : 41309
16     AM Host : compute-1
17     Aggregate Resource Allocation : 1075936 MB-seconds, 942 vcore-
          seconds
18     Diagnostics :
```

Listing A.3: Starten einer Anwendung in Hadoop-Benchmark. Hier mit dem Mapreduce Example `pi` und dem Abbruch der Anwendung durch den in Listing A.4 gezeigten Befehl. Die Anwendungs-ID `application_1520342317799_0002` ist hier in Zeile 13 enthalten.

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

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

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

```
$ yarn application -kill application_1520342317799_0002
...
Killing application application_1520342317799_0002
18/03/14 13:07:02 INFO impl.YarnClientImpl: Killed application
   application_1520342317799_0002
```

B REST-API von Hadoop

Wie bei der Ausgabe der Daten der YARN-Komponenten mithilfe der CLI können auch bei der Ausgabe mithilfe der REST-API die Daten als Liste oder als einzelner Report ausgegeben werden. Der Unterschied zur CLI liegt jedoch darin, dass in Listenform und als einzelner Report immer die vollständigen Objekte der Komponenten zurückgegeben werden. 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 sowie hier nicht gezeigte Pfade für die YARN-Komponenten sind in der Dokumentation [22, 25, 26] zu finden.

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


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

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

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

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

C Benötigte Befehle des Setup-Scriptes

Das Setup-Script dient einerseits zur Trennung des genutzten `HostMode`s, aber auch zur Vereinfachung der benötigten Befehle zur Steuerung des Clusters (vgl. Abschnitt 4.4). Zur Nutzung der implementierten Connectoren muss das genutzte Setup-Script mindestens folgende Befehle beinhalten:

Listing C.1: Benötigte Befehle eines Setup-Scriptes. Das Setup-Script des `Multihost`-Modes bietet zum Teil andere Befehle an, besitzt jedoch entsprechende Befehle zur vollständigen Kompatibilität.

```
1  hadoop start [node-id]  starting hadoop or the given node
2  hadoop stop [node-id]   stopping hadoop or the given node
3  hadoop restart [node]   restarts hadoop or the given node
4  hadoop destroy          destroys hadoop
5  hadoop info [id] [form]
6                      list running containers or node container details
7                            and can use --format string
8
9  net start <node-id> enables networking interfaces on the given node
10 net stop <node-id>  disables networking interfaces on the given node
11
12 cmd <cmd>           executes the given command on hadoop controller
13 hdfs <cmd>          executes the hdfs command and prints the exit code
14 marp                Gets the current MARP value from hadoop config
```

D Genutzte Tools und Frameworks

Im Folgenden werden alle für die Entwicklung des Testsystems und der Durchführung der Fallstudie benötigten und relevanten Programme, Tools und Frameworks aufgelistet:

Tool/Framework	Version	Homepage
Apache Hadoop	2.7.1	https://hadoop.apache.org/
curl	7.47	https://curl.haxx.se/
Docker	18.03 CE	https://www.docker.com/
Hadoop-Benchmark	vom 22.05.2017	https://github.com/Spirals-Team/hadoop-benchmark/
Json.NET	11.0.2	https://www.newtonsoft.com/json/
log4net	2.0.8	https://logging.apache.org/log4net/
.NET	4.7.2	https://www.microsoft.com/net/
nUnit Test Adapter	2.1.1	http://nunit.org/
ReSharper	2018.1	https://www.jetbrains.com/resharper/
Safety Sharp	1.2.3 vom 11.12.2017	http://safetysharp.isse.de/
Selfbalancing-Komponente	vom 26.10.2015	https://github.com/zhangbbo/Self-tunning-MARP/
SSH.NET	2016.1.0	https://github.com/sshnet/SSH.NET/
Ubuntu	16.04 LTS	https://www.ubuntu.com/
Universalmutator	0.8.13	https://github.com/agroce/universalmutator/
Windows 10	Education Version 1803	https://www.microsoft.com/de-de/windows/
VirtualBox	5.2.12	https://www.virtualbox.org/
Visual Studio	Enterprise 2017 15.5.6	https://visualstudio.microsoft.com/

Tabelle D.1: Relevante, genutzte Tools und Frameworks

E Ausgabeformat des Programmlogs

Die in Unterabschnitt 3.3.3 und u. a. in Unterabschnitt 6.1.4 beschriebenen Ausgaben werden im nachfolgend dargestellten Format gespeichert. Es handelt sich hierbei um den gekürzten Programmlog der Ausführung des Testfalls 1 (vgl. Anhang F).

Listing E.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
```

F Übersicht der ausgeführten Tests

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

Die Spalte *Mutanten* gibt an, welche der in Abschnitt 6.2 definierten Mutanten 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 viel Zeit die jeweiligen Simulationen in Minuten und Sekunden benötigten. Wenn nicht alle möglichen Testfälle ausgeführt wurden, war im darauf folgenden Testfall eine Rekonfiguration des Clusters nicht mehr möglich und die Simulation wurde abgebrochen (vgl. Unterabschnitt 4.2.9).

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 F.1: Übersicht der ausgeführten Testkonfigurationen