Universität Augsburg
Fakultät für Angewandte Informatik

Modellbasierte Testautomatisierung eines verteilten, adaptiven Load-Balancing-Systems

Masterarbeit

im Studiengang Informatik

zur Erlangung des akademischen Grades

Master of Science

von

Gerald Siegert

Mat.-Nr.: 1450117

Datum: 7. März 2018

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

Zusammenfassung

Abstract

Inhaltsverzeichnis

Verzeichnisse

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Abkürzungsverzeichnis

AM ApplicationManager

AppMstr ApplicationMaster

DCCA Deductive Cause-Consequence Analysis

MARP `maximum-am-resource-percent`

MC Model Checking

MC Model Checker

NM NodeManager

RM ResourceManager

TLS Timeline-Server

1. Einleitung

Im Bereich der Softwaretests wird heutzutage sehr viel mit automatisierten Testverfahren gearbeitet. Dies ist insofern logisch, als dass diese Testautomatisierung einerseits Aufwand und damit andererseits direkt Kosten einer Software einspart. Daher gibt es vor allem im Bereich der Komponententests zahlreiche Frameworks, mit denen Tests einfach und automatisiert erstellt bzw. ausgeführt werden können. Ein Beispiel für ein solches Testframework wäre das *xUnit*-Framework, zu dem u. A. JUnit[1] für Java und NUnit[2] für .NET zählen. Dabei werden zunächst einzelne Testfälle erstellt und können im Anschluss mit der jeweils aktuellen Codebasis jederzeit ausgeführt werden. Automatisierte Tests können auch dazu genutzt werden, um einen einzelnen Test mit verschiedenen Eingaben durchzuführen. Dadurch können verschiedene Eingabeklassen (wie negative oder positive Ganzzahlen) mit sehr geringem Aufwand in einem Test genutzt werden und somit verschiedene Testfälle direkt ausgeführt werden, wodurch eine massive Kosteneinsparung einhergeht [1].

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

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

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

2. Relevante Methodiken

2.1. Model Checking

MC ist eine Möglichkeit, um Systeme zu testen und zu verifizieren. Dazu werden vom Model Checker (MC) alle möglichen Systemzustände in einem *brute-force*-ähnlichem Vorgehen getestet und somit alle möglichen Szenarien getestet. Die Anzahl der Zustände kann sehr schnell 10^{120} oder mehr betragen [2, 4].

Ein MC nutzt, wie der Name schon sagt, ein Modell des Systems, um das System zu testen. Wie bei jeder anderen modellbasierten Technik ist daher die Qualität des MC nur so gut wie das darauf zugrunde liegende Modell. Ein Modell kann auch als endlicher Automat angesehen werden, da ein Modell ebenfalls eine endliche Anzahl an möglichen Zuständen und dazugehörige Übergänge besitzt. Für jede Eigenschaft eines Zustandes muss zudem mithilfe einer sog. *temporalen Logik*, also mathematisch bzw. formal, festgelegt werden, was gültige Werte dieser Eigenschaft sind. Die dazu benötigten Informationen werden aus den Anforderungen des Systems ermittelt und dem MC übergeben. So können später verschiedene Eigenschaften des gesamten Systems (z. B. die formale Korrektheit, die Ausführbarkeit ohne Deadlocks oder die Einhaltung von Sicherheitsvorgaben) geprüft werden.

Zur Ausführung wird das gesamte Modell zunächst initialisiert und dann automatisch und systematisch vom MC auf Fehler und ungültige Zustände geprüft. In der Regel ist aber auch eine Ausführung als reine Simulation des Systems möglich, ohne explizit nach Fehlern zu suchen.

Wenn alle Zustände und deren Eigenschaften die Anforderungen erfüllen, erfüllt auch das Modell die Spezifikation. Wenn ein Zustand bzw. Eigenschaft die Anforderungen nicht erfüllt, prüft der MC anhand eines Gegenbeispiels den Ausführungspfad zum Fehler. Dadurch kann ermittelt werden, wo die Fehlerursache liegt. Einige der wesentlichen Fehlertypen und Ursachen sind:

Modelling Error Der Fehler liegt im Modell, welches korrigiert werden muss.

Design Error Der Fehler liegt in den formellen oder informellen Anforderungen, dadurch muss das Modell und/oder die temporale Logik korrigiert werden.

Property Error Der Fehler ist wirklich ein Fehler im System, welcher gefunden werden soll.

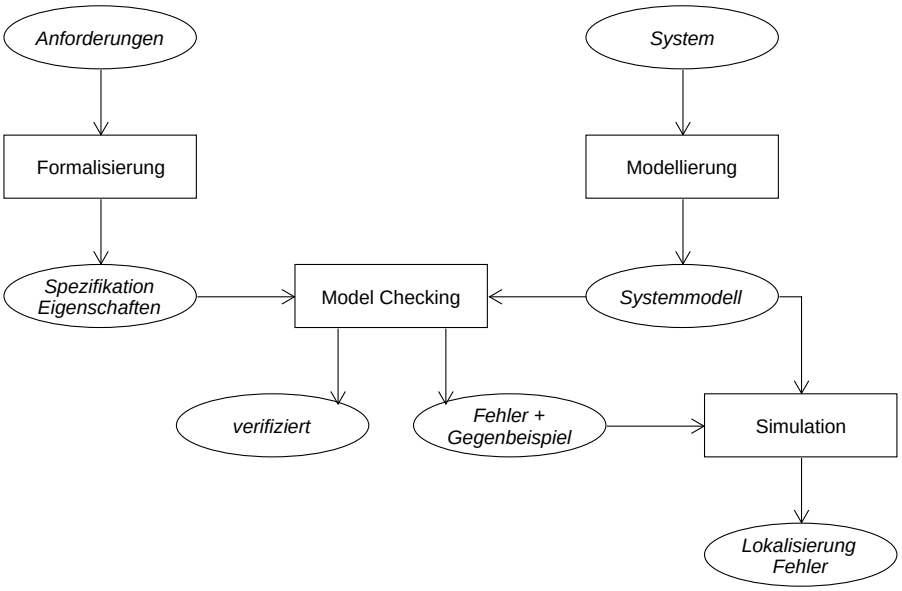

Abbildung 2.1.: Schematischer Aufbau beim MC, nach [4]

Möglich ist aber auch, dass die Ressourcen nicht ausreichen, um alle Zustände zu prüfen. In so einem Fall gibt es mehrere Möglichkeiten, damit umzugehen, z. B. können Heuristiken oder Abstraktionen vom Modell genutzt werden [4, 5].

MC besitzt durch seine Charakteristik einige Vorteile, u. A. [4]:

- MC ist universell nutzbar, z. B. für Software, Hardware oder eingebettete Systeme
- Partielle Verifikation ist möglich ohne das gesamte System testen zu müssen
- Vollständig automatisierbar und benötigt kaum Benutzerinteraktion oder hohe Expertise

Natürlich gibt es aber auch einige Nachteile, u. A. [4]:

- Mit MC wird nur ein Systemmodell und nicht das eigentliche System getestet, was weitere Fehler nicht ausschließt
- Hauptsächlich für steuerungsbasierte Anwendungen und nicht für datenbasierte Anwendungen geeignet
- Anzahl der möglichen Zustände kann zu hoch sein, um alle zu testen

Es gibt zahlreiche MC-Frameworks, die bereits erwähnten *LTSmin* und *S#* sind nur zwei davon.

2.2. S#

S# ist ein am Institute for Software & Systems Engineering der Universität Augsburg entwickeltes Testframework, das auch einen MC beinhält. Da es in C# entwickelt wurde und C# auch zum Entwickeln von Modellen und dazugehörigen Testszenarien genutzt wird, können zahlreiche Features des .NET-Frameworks bzw. der Sprache C# im

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

Listing 2.1: Grundlegender Aufbau einer S#-Komponente

Speziellen genutzt werden. S# vereint dabei die Simulation, die Visualisierung, modellbasierte Tests sowie das MC der Modelle [3, 6]. Dadurch können alle Schritte einer vollständigen Analyse inkl. Modellierung direkt im Visual Studio ausgeführt werden und somit auch alle Features der IDE und von .NET, wie z. B. die Debugging-Werkzeuge, genutzt werden. Um den MC zu nutzen, hat S# jedoch einige Einschränkungen, u. A. sind Schleifen und Rekursionen nur eingeschränkt bzw. nicht möglich. Eine der größten Einschränkungen ist allerdings, dass während der Laufzeit keine neuen Objektinstanzen innerhalb des zu testenden Modells erzeugt werden können, sodass alle benötigten Instanzen bereits während der Initialisierung des Modells erzeugt werden müssen [3].

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

Listing 2.1 zeigt den typischen, grundlegenden Aufbau einer S#-Komponente. Jede Komponente des Modells muss von `Component` erben, um als S#-Komponente definiert zu sein. Jede Komponente kann nun temporäre (`TransientFault`) oder dauerhafte (`PermanentFault`) Komponentenfehler enthalten, welche zunächst innerhalb der Komponente als Felder definiert werden. Der Effekt eines Komponentenfehlers wird anschließend in der entsprechenden inneren Klasse definiert, welche von der Hauptklasse (hier `YarnNode`) erbt und mithilfe des Attributs `FaultEffect` dem dazugehörigen Komponentenfehler zugeordnet wird [6].

Um die Modelle zu testen, kommt in S# die Deductive Cause-Consequence Analysis (DCCA) zum Einsatz. Die DCCA ermöglicht eine vollautomatisch und MC-basierte Sicherheitsanalyse, wodurch selbstständig die Menge der aktivierten Komponentenfehler ermittelt wird, mit denen sich das Gesamtsystem nicht mehr rekonfigurieren kann und somit ausfällt. Je nach Konfiguration können dazu auch Heuristiken genutzt wer-

den, welche die Analyse beschleunigen und genauer machen können [5]. Dabei werden die verschiedenen aktivierten Komponentenfehler während der Analyse in tolerierbare und nicht-tolerierbare Fehler unterschieden. Tolerierbare Komponentenfehler werden dazu genutzt, die Grenzen der Selbstkonfiguration des Systems zu ermitteln. Dabei wird für jeden Systemzustand nach einer Rekonfiguration durch die DCCA eine neue Fehlermenge ermittelt, mit der das System gerade noch so lauffähig ist. Das Auftreten eines tolerierbaren Komponentenfehler ist also gleichbedeutend mit einem einfachen Fehler im System, welcher die gesamte Funktionsweise des Systems nicht massiv einschränkt und es sich noch selbst rekonfigurieren kann. Sobald jedoch ein Fehler auftritt, durch den es dem System nicht mehr möglich ist, sich selbst zu rekonfigurieren, wurde ein nicht-tolerierbarer Fehler gefunden, durch den das System nicht mehr funktionsfähig ist [3].

3. Aufbau der Fallstudie

In der Fallstudie im Rahmen dieser Masterarbeit wird Apache™Hadoop®[1] mithilfe eines modellbasierten Tests getestet. Da Hadoop normalerweise keine adaptive Komponente besitzt, wurde Hadoop mit der von Zhang u. a. entwickelten selbst-adaptiven Komponente erweitert und ein Cluster mithilfe der ebenfalls von Zhang u. a. entwickelten Plattform Hadoop-Benchmark erstellt.

3.1. Apache Hadoop

Apache Hadoop ist ein Open-Source-Software-Projekt, mit dessen Hilfe ermöglicht wird, Programme zur Datenverarbeitung mit großen Ressourcenbedarf auf verteilten System auszuführen. Hadoop wird von der *Apache Foundation* entwickelt und bietet verschiedene Komponenten an, welche vollständig skalierbar sind, von einer einfachen Installation auf einem PC bis hin zu einer Installation über mehrere Server in einem Serverzentrum. Hadoop besteht hauptsächlich aus folgenden Kernmodulen [8]:

Hadoop Common Gemeinsam genutzte Kernkomponenten

Hadoop YARN Framework zur Verteilung und Ausführung von Anwendungen und das dazugehörige Ressourcen-Management

Hadoop Distributed File System Kurz HDFS, Verteiltes Dateisystem

Hadoop MapReduce YARN-Basiertes System zum Verarbeiten von großen Datenmengen

Hadoop ermöglicht es dadurch, sehr einfach mit Anwendungen umzugehen, welche große Datenmengen verarbeiten. Da es für Hadoop nicht relevant ist, auf wie vielen Servern es läuft, kann es beliebig skaliert werden, wodurch entsprechend viele Ressourcen zur Bearbeitung und Speicherung von großen Datenmengen zur Verfügung stehen können.

Die Kernidee der Architektur von **YARN** ist die Trennung vom Ressourcenmanagement und Scheduling. Dazu besitzt der Master bzw. *Controller* den ResourceManager (RM), welcher für das gesamte System zuständig ist und die Anwendungen im

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

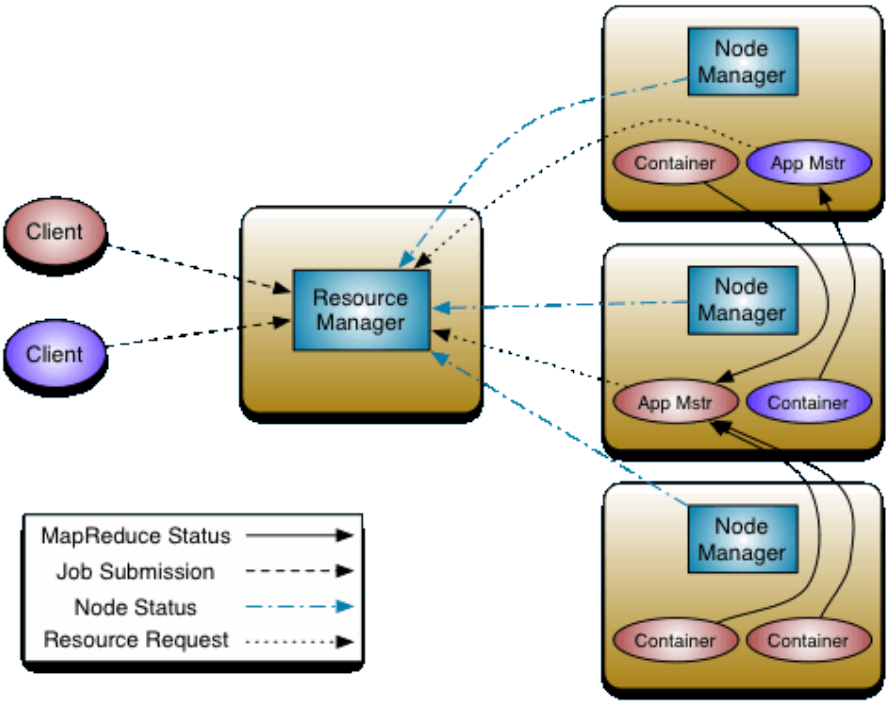

Abbildung 3.1.: Architektur von YARN (entnommen aus [9])

System verteilt und überwacht und somit auch als *Load-Balancer* agiert. Er besteht aus zwei Kernkomponenten, dem ApplicationManager (AM) und dem *Scheduler*. Der AM ist für die Annahme und Ausführung von einzelnen Anwendungen zuständig, denen der Scheduler die dafür notwendigen Ressourcen im Cluster zuteilt.

Jeder Slave-*Node* im Hadoop-Cluster besitzt einen NodeManager (NM), welcher für die Überwachung der Ressourcen des Nodes und der darauf ausgeführten Anwendungs-Container zuständig ist und diese dem RM mitteilt.

Jede YARN-Anwendung bzw. Job besteht aus einem oder mehreren Ausführungsversuchen, genannt *Attempts*, denen wiederum mehrere *Container* zugeordnet sind. Container können auf einem beliebigen Node ausgeführt werden und repräsentieren die Ausführung eines Tasks der Anwendung. Ein besonderer Container bildet dabei der ApplicationMaster (AppMstr), welcher innerhalb seines Attempts für das anwendungsbezogene Monitoring und die Kommunikation mit dem RM und NM zuständig ist und die dazu notwendigen Informationen bereit stellt [9].

Hadoop enthält zudem einen sog. Timeline-Server (TLS). Er ist speziell dafür entwickelt, die Metadaten und Logs der YARN-Anwendungen zu speichern und jederzeit, also auch als Anwendungshistorie, auszugeben [10].

Das **HDFS** basiert auf der gleichen Architektur wie YARN und besitzt ebenfalls einen Master und mehrere Slaves, welches in der Regel die gleichen Nodes sind wie bei YARN sind. Der *NameNode* ist als Master für die Verwaltung des Dateisystems zuständig und reguliert den Zugriff auf die darauf gespeicherten Daten. Die Daten selbst werden in mehrere Blöcke aufgeteilt auf den *DataNodes* gespeichert. Um den Zugriff auf

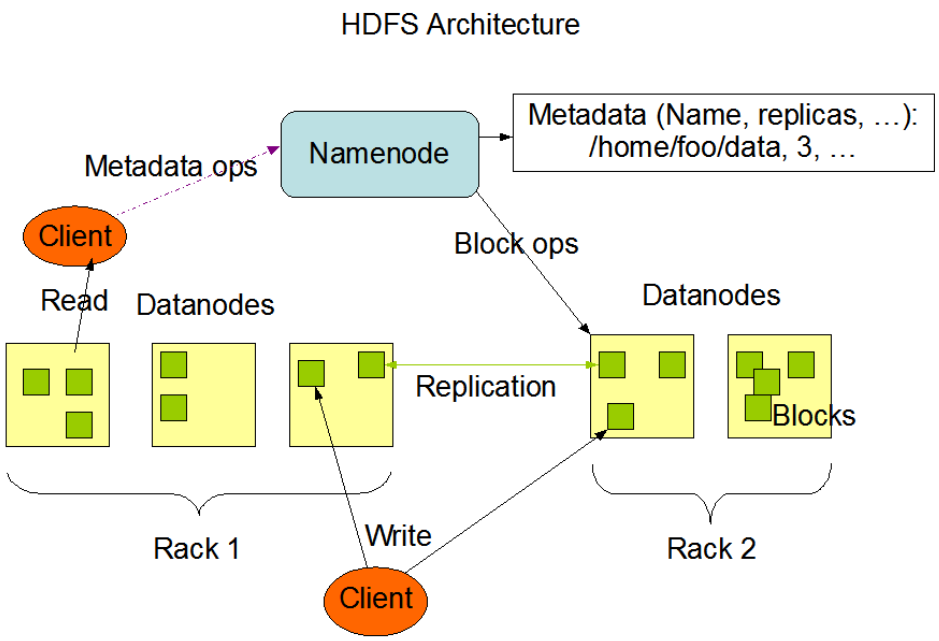

Abbildung 3.2.: Architektur des HDFS (entnommen aus [11])

die Daten im Falle eines Node-Ausfalls zu gewährleisten, wird jeder Block auf anderen Nodes repliziert. Dateioperationen (wie Öffnen oder Schließen) werden direkt auf den DataNodes ausgeführt, sie sind darüber hinaus auch dafür verantwortlich, dass Clients die Daten lesen oder beschreiben können [11].

MapReduce bietet analog zu YARN die Möglichkeit, Anwendungen mit einem großen Ressourcenbedarf, welche große Datenmengen verarbeiten, auf einem gesamten Cluster auszuführen. Dazu werden bei einem MapReduce-Job die Eingabedaten aufgeteilt, anschließend von den sog. *Map Tasks* verarbeitet und deren Ausgaben von den sog. *Reduce Tasks* geordnet. Für die Ein- und Ausgabe der Daten wird in der Regel das HDFS, für die Ausführung der einzelnen Tasks YARN genutzt [12]. MapReduce kann auch als Vorgänger von YARN angesehen werden, da YARN auch als *MapReduce Next Gen* bzw. *MRv2* bezeichnet wird und aufgrund der API-Kompatibilität von YARN jede MapReduce-Anwendung in der Regel auch auf YARN ausgeführt werden kann [9, 13].

3.2. Adaptive Komponente in Hadoop

Eine normale Hadoop-Installation besitzt keine adaptive Komponente, sondern rein statische Einstellungen. Um damit Hadoop zu optimieren, müssen die Einstellungen immer manuell auf den jeweils benötigten Anwendungstyp angepasst werden. Dazu gibt es auch bereits verschiedene Scheduler, den *Fair Scheduler*, welcher alle Anwendungen ausführt und ihnen gleich viele Ressourcen zuteilt, und den *Capacity Scheduler*. Letzterer sorgt dafür, dass nur eine bestimmte Anzahl an Anwendungen pro Benutzter gleichzeitig ausgeführt wird und teilt ihnen so viele Ressourcen zu, wie benötigt werden

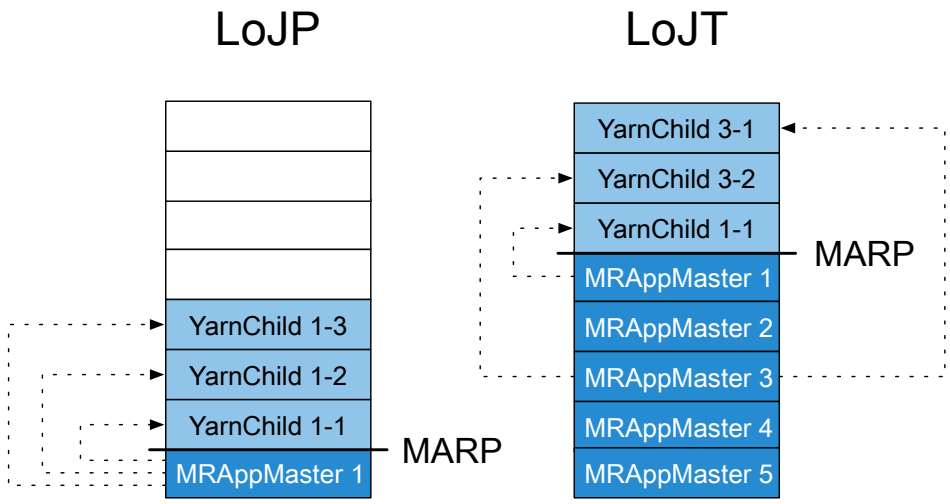

Abbildung 3.3.: LoJP und LoJT in Hadoop (entnommen aus [7])

bzw. der Benutzter nutzen darf. Entwickelt wurde der Capacity Scheduler vor allem für Cluster, die von mehreren Organisationen gemeinsam verwendet werden und sicherstellen soll, dass jede Organisation eine Mindestmenge an Ressourcen zur Verfügung hat [14].

Je nach Bedarf besitzt der Capacity Scheduler entsprechende Einstellungen, um z. B. den verfügbaren Speicher pro Container festzulegen. Eine weitere Einstellung des Schedulers ist `maximum-am-resource-percent`, auch MARP gennant, der angibt, wie viele Prozent der gesamten Ressourcen durch AppMstr-Container genutzt werden dürfen [14]. Damit bewirkt diese Einstellung indirekt auch die maximale Anzahl an Anwendungen, die gleichzeitig ausgeführt werden dürfen. Da der MARP-Wert jedoch nicht während der Laufzeit dynamisch angepasst werden kann, haben Zhang u. a. in [7] einen Ansatz zur dynamischen Anpassung des MARP-Wertes zur Laufzeit von Hadoop vorgestellt. Dadurch wird der MARP-Wert abhängig von den ausgeführten Anwendungen adaptiv zur Laufzeit angepasst, sodass immer möglichst viele Anwendungen gleichzeitig ausgeführt werden können. Dadurch werden Anwendungen im Schnitt um bis zu 40 % schneller ausgeführt [7].

Der Hintergrund dieser *Selfbalancing-Komponente* ist der, dass durch den MARP-Wert der für die Anwendungen verfügbare Speicher in zwei Teile aufgeteilt wird. Im einen Teil befinden sich alle derzeit ausgeführten AppMstr, im anderen Teil die von den Anwendungen benötigten weiteren Container. Wie groß der Teil für die AppMstr ist, wird nun durch den MARP-Wert bestimmt. Ist der MARP-Wert zu klein, können nur wenige AppMstr (und damit Anwendungen) gleichzeitig ausgeführt werden (*Loss of Jobs Parallelism*, LoJP). Ist der MARP-Wert jedoch zu groß, können für die ausgeführten Anwendungen nur wenige Container bereitgestellt werden, wodurch sich die Ausführung für eine Anwendung wesentlich verlangsamt (*Loss of Job Throughput*, LoJT)[7]. Abbildung 3.3 illustriert beide Situationen, wodurch einerseits viel Speicher

für weitere Anwendungscontainer ungenutzt bleiben kann, andererseits aber zahlreiche AppMstr ohne laufende Anwendungscontainer Speicher unnötig belegen können.

Die Selfbalancing-Komponente passt daher den MARP-Wert abhängig von der Speicherauslastung dynamisch zur Laufzeit an. So wird der MARP-Wert verringert, wenn die Speicherauslastung sehr hoch ist, und erhöht, wenn die Speicherauslastung sehr niedrig ist [7]. Dadurch wird es ermöglicht, dass die maximal mögliche Anzahl an Anwendungen ausgeführt werden kann. Die Evaluation von Zhang u. a. ergab zudem, dass die dynamische Anpassung des MARP-Wertes zudem auch effizienter ist als eine manuelle, statische Optimierung.

3.3. Umsetzung des realen Clusters

Zhang u. a. haben im Rahmen ihrer gesamten Forschungsarbeit die Open-Source-Plattform Hadoop-Benchmark entwickelt und auf Github zur Verfügung gestellt.[2] Sie wurde speziell zum Einsatz in der Forschung erstellt und kann jederzeit an die eigenen Bedürfnisse angepasst werden. Zur Umsetzung des realen Clusters im Rahmen dieser Masterarbeit wurde daher eine speziell angepasste Version der Plattform eingesetzt.

3.3.1. Plattform Hadoop-Benchmark

Die Plattform ist in mehrere Szenarien unterteilt, darunter ein Hadoop in der Version 2.7.1 ohne Änderungen und ein darauf basierendes Szenario mit der Selfbalancing-Komponente. Hadoop-Benchmark basiert auf der Software *Docker*[3] und dem dazugehörigen Tool *Docker Machine*, um damit einfach und schnell ein Hadoop-Cluster aufbauen zu können. Mit *Graphite*[4] ist zudem ein Monitoring-Tool enthalten, mit dem die Performance des Clusters überwacht und analysiert werden kann.

Abbildung 3.4 zeigt die grundlegende Architektur der Plattform, die mithilfe eines Docker-Swarms auf mehreren *Docker Machines* (für den Einsatz von Docker eingerichtete virtuelle Maschinen) ein Cluster erstellt, auf denen dann in den Docker-Containern das eigentliche Hadoop-Cluster ausgeführt wird. Jeder Hadoop-Container enthält zudem das Tool *collectd*[5], was das Monitoring des Containers auf Systemebene übernimmt und die Daten an den Graphite-Container auf der Controller-Machine übermittelt. Es ist dabei möglich, eine beliebige Anzahl an Nodes zu nutzen. Auch ist es möglich, den Docker Machines einen beliebig großen Arbeitsspeicher zur Verfügung zu stellen.

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

- Hadoop Mapreduce Examples

[2] `https://github.com/Spirals-Team/hadoop-benchmark`
[3] `https://www.docker.com/`
[4] `https://graphiteapp.org/`
[5] `https://collectd.org/`

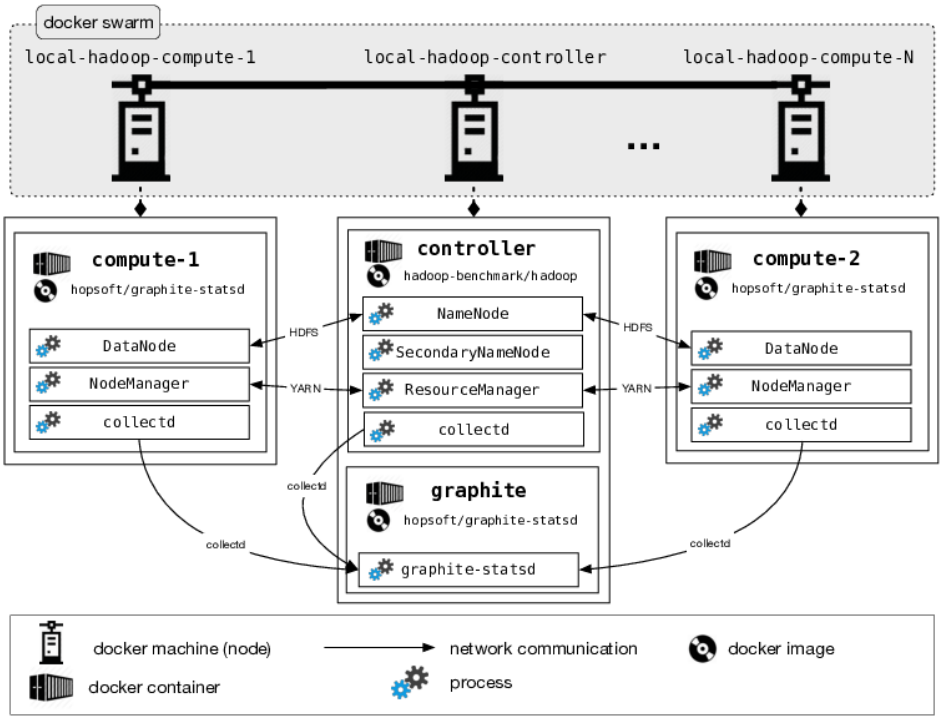

Abbildung 3.4.: High-Level-Architektur von Hadoop-Benchmark [15]

- Intel HiBench[6]
- SWIM (Statistical Workload Injector for Mapreduce)[7]

Eine Besonderheit bildet der SWIM-Benchmark, welcher sehr Ressourcenintensiv ist und daher auf einem *Single Node Cluster*, also einem kompletten Hadoop-Cluster auf nur einem Computer, sehr zeitintensiv sein kann. Der Intel HiBench-Benchmark besteht aus Kategorien wie *Machine Learning* oder Graphen, welche wiederum aus einen oder mehreren *Workloads* bestehen, welche entsprechende Anwendungen bzw. Algorithmen auf dem Hadoop-Cluster ausführen. Einige der Hibench-Workloads basieren auf den Mapreduce Examples, welche wiederum voneinander unabhängige Beispielanwendungen für Hadoop darstellen.

3.3.2. Anpassungen und Setup

Da mithilfe der Plattform Hadoop-Benchmark die Erstellung eines Hadoop-Clusters massiv vereinfacht wird, kommt die Plattform auch in dieser Masterarbeit zum Einsatz. Da Docker und Hadoop aber vor allem für den Einsatz in einer Linux-Umgebung entwickelt wurden, wird dazu ein eigener PC mit Ubuntu 16.04 LTS genutzt. Da S# das .NET-Framework, und damit Windows, benötigt, wird dafür ebenfalls ein eigener PC verwendet. Im konkreten Versuchsaufbau wird für Windows eine VM genutzt, welche auf einem anderen PC als das Cluster ausgeführt wird. Die genauen Spezifikationen

[6] https://github.com/intel-hadoop/HiBench
[7] https://github.com/SWIMProjectUCB/SWIM

	Cluster-PC	**VM-PC**	**Windows-VM**
CPU	Intel Core i5-4570 @ 3,2 GHz x 4		4 CPU Cores
RAM	16 GB		8 GB
SSD	512 GB	512 GB	$\leq$ 100 GB VHD
OS	Ubuntu 16.04 LTS	Ubuntu 16.04 LTS	Windows 10 1709 Edu.

Tabelle 3.1.: Spezifikationen der verwendeten PCs und Windows-VM

der PCs und der Windows-VM sind in Tabelle 3.1 aufgelistet. Die Windows-VM und der Cluster-PC werden mithilfe von SSH-Verbindungen miteinander verbunden.

Auf dem Cluster-PC nutzt Docker-Machine zur Erstellung, Verwaltung und Ausführung der VMs die Treiber von VirtualBox 5.2[8], zum Abrufen der Daten der REST-API über die SSH-Verbindung wird *curl*[9] genutzt. Für das Cluster werden 4 Nodes, der Controller sowie eine Consul-VM zur internen Verwaltung der Netzwerkverbindungen zwischen den VMs und Docker-Containern erstellt. Der Controller erhält 4 GB RAM, jeder der vier Nodes jeweils 2 GB, für den Consul sind 512 MB ausreichend. Für die Windows-VM wird ebenfalls VirtualBox 5.2 eingesetzt.

In keinem Szenario der Plattform Hadoop-Benchmark wird standardmäßig der TLS von Hadoop gestartet. Daher wurde für diese Fallstudie basierend auf dem Selfbalancing-Szenario ein neues Szenario erstellt, bei dem der TLS gestartet wird. Dadurch ist einerseits die Selfbalancing-Komponente von Zhang u. a. aktiv und andererseits besteht die Möglichkeit, für das Monitoring zusätzlich den TLS zu nutzen.

Um viele standardmäßige Aufgaben und Möglichkeiten zu vereinfachen, wurde zudem ein eigenes Setup-Script erstellt. Es vereinfacht folgende Aufgaben:

- Starten, Beenden und Löschen des kompletten Clusters mit Hadoop
- Starten und Beenden des Docker-Containers eines Nodes
- Hinzufügen und Entfernen der Netzwerkverbindung des Docker-Containers eines Nodes
- Ausführen der verwendeten Benchmarks
- Ausführen von eigenen Befehlen auf dem Docker-Container des Controllers

Für die Befehle, die das gesamte Cluster betreffen, wird vom Setup-Script meist auf das in Hadoop-Benchmark enthaltene Start-Script zugegriffen. Die Befehle, welche die Docker-Container der Nodes betreffen, sowie das Ausführen von Befehlen im Controller-Container, werden vom Setup-Script direkt ausgeführt. Für das Starten der Benchmarks werden dagegen die in Hadoop-Benchmark enthaltenen Ausführungs-Scripte der Benchmarks gestartet.

[8] `https://www.virtualbox.org/`
[9] `https://curl.haxx.se/`

4. Aufbau des Modells

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

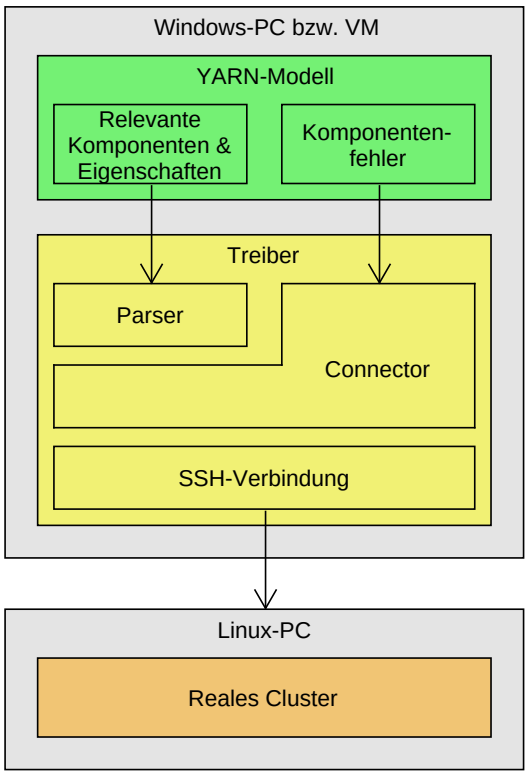

Abbildung 4.1.: Grundlegende Architektur des Gesamtmodells

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

Connector Abstrahiert die SSH-Verbindung

SSH-Verbindung Eigentliche Verbindung zum Cluster-PC

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

4.1. YARN-Modell

Abbildung 4.2 beschreibt im Grunde bereits das gesamte von S# verwendete YARN-Modell. Enthalten sind alle hier relevanten Komponenten sowie deren Eigenschaften. Als Eigenschaften wurden die Daten aufgenommen, welche mithilfe von Shell-Kommandos bzw. mithilfe der REST-API von YARN ermittelt werden können.

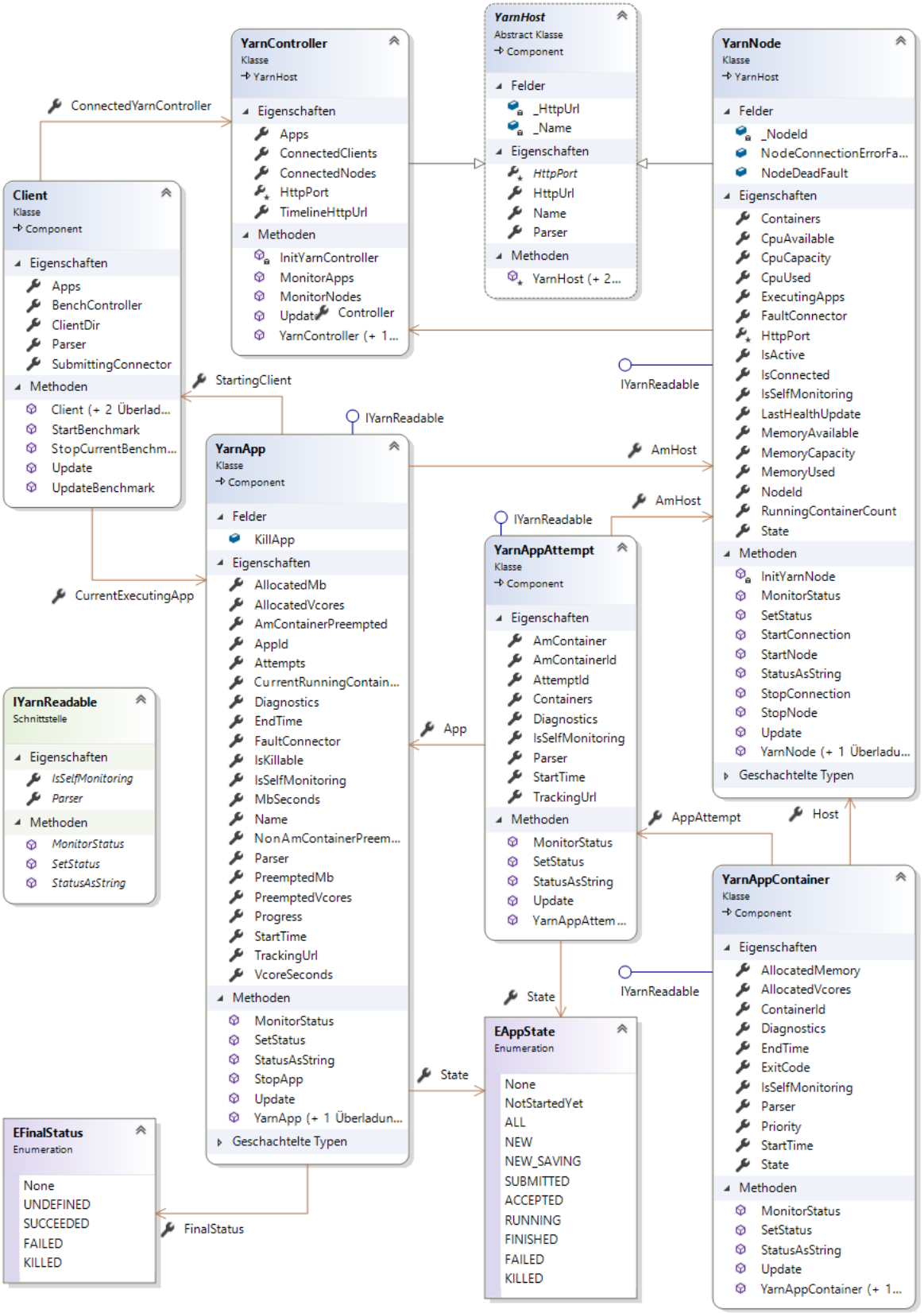

Abbildung 4.2.: Aufbau des YARN-Modells. Das Modell wurde mithilfe des Klassendiagramm-Designers in Visual Studio 2017 visualisiert. Daher werden Assoziationen mit höherer Multiplizität als 1 (wie `YarnApp.Attempts`, die mithilfe von `List<T>` umgesetzt wurden) im Diagramm nicht als Assoziationen zwischen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

Die abstrakte Basisklasse `YarnHost` stellt die Basis für alle Hosts des Clusters dar, also dem `YarnController` mit dem RM, und dem `YarnNode`, was einen Node darstellt, auf dem die Anwendungen bzw. deren Container ausgeführt werden. Die abstrakte Eigenschaft `YarnHost.HttpPort` dient als Hilfs-Eigenschaft, da Controller und Nodes unterschiedliche Ports für die Weboberfläche nutzen, deren URL mit Port in der Eigenschaft `YarnHost.HttpUrl` abrufbar ist. Sie wird daher vom Controller bzw. Node mit dem entsprechenden Port versehen. Die Felder `YarnNode.NodeConnectionError` und `YarnNode.NodeDead` bilden die Komponentenfehler, wenn ein Node seine Netzwerkverbindung verliert bzw. beendet wird. Die Effekte der Komponentenfehler werden über entsprechende innere Klassen realisiert.

Die Anwendungen, welche mit `YarnApp` dargestellt werden, werden mithilfe des `Client` gestartet, was den zu startenden Client darstellt. Die Anwendungen selbst enthalten neben grundlegenden Daten wie z. B. den Namen auch einige Daten zum Ressourcenbedarf (Speicher und CPU). Zwar gibt Hadoop nicht direkt die zu der Anwendung gehörigen Job-Ausführungen an, allerdings können diese mithilfe der `YarnApp.AppId` sehr einfach ermittelt werden und dann in der Liste `YarnApp.Attempts` gespeichert werden. Das Feld `YarnApp.IsKillable` gibt an, ob die Ausführung der Anwendung mit den aktuellen Daten im Modell durch den Komponentenfehler `YarnApp.KillApp` abgebrochen werden kann. Abhängig ist das durch `YarnApp.FinalStatus`, was angibt, ob eine Anwendung erfolgreich oder nicht erfolgreich ausgeführt wurde oder die Ausführung noch nicht abgeschlossen ist (durch `EFinalStatus.UNDEFINED`). Um die Komponentenfehler zu aktivieren bzw. bei Bedarf auch wieder zu deaktivieren, besitzen `YarnNode` und `YarnApp` jeweils die Eigenschaft `FaultConnector`, mit der auf den benötigten Connector zugegriffen werden kann.

Jede Ausführung `YarnAppAttempt` hat eine eigene ID und kann einer Anwendung zugeordnet werden. Genau wie bei den Anwendungen selber wird hier direkt der Node gespeichert, auf welchem der AppMstr ausgeführt wird und einen eigenen Container bildet, dessen ID direkt gespeichert wird. Container (dargestellt durch `YarnAppContainer`) existieren in Hadoop nur während der Laufzeit eines Programmes und enthalten nur wenige Daten, darunter ihr ausführender Node. Jede Anwendung, deren Ausführungen und deren Container enthalten zudem den derzeitigen Status, ob die Komponente noch initialisiert wird, bereits ausgeführt wird oder beendet ist. `EAppState.NotStartedYet` dient als Status, den es nur im Modell gibt und angibt, dass die Anwendung im späteren Verlauf der Testausführung gestartet wird.

Alle vier YARN-Kernkomponenten implementieren das Interface `IYarnReadable`, was angibt, dass die Komponente ihren Status aus Hadoop ermitteln kann. Entsprechend wird in allen Komponenten die Methode `ReadStatus()` implementiert, in welchem mithilfe des angegebenen Parsers auf den SSH-Treiber zugegriffen werden kann und die Komponenten im Modell so ihre eigenen Daten aus dem realen Cluster ermitteln können. Da die REST-API ermöglicht, alle Daten auch über die reinen Listen zu

erhalten anstatt ausschließlich über die Detailausgabe, besteht auch im Modell mithilfe der Eigenschaft `IsRequireDetailsParsing` das Ermitteln der Daten so einzustellen, dass die übergeordnete YARN-Komponente bereits alle Daten ermittelt und der Untergeordneten zum Speichern (mittels `SetStatus()`) übergibt. Als Basis dazu dient der `YarnController`, der dafür die Daten aller Anwendungen ausliest, die wiederum die Daten ihrer Ausführungen auslesen, welche dann die Daten ihrer Container auslesen und den Komponenten zum Speichern übergeben.

4.2. SSH-Treiber

Im Einführungstext zu diesem Kaptiel wurde bereits auf den grundlegenden Aufbau des Treibers eingegangen, der aus den drei einzelnen Komponenten Parser, Connector und der eigentlichen SSH-Verbindung besteht. Der Parser selbst besteht neben dem eigentlichen Parser zudem aus Datenhaltungs-Klassen für die relevanten YARN-Komponenten. Sie sind außerdem so aufgebaut, dass sie für beide hier implementierten Parser bzw. Connectoren für die Kommandozeilen-Befehle und die REST-API genutzt werden können.

4.2.1. Integration im Modell

Hadoop besitzt zwei primäre Wege, um die Daten vom RM bzw. dem TLS ausgeben zu können. Dies ist zum einen die Kommandozeile, mithilfe der die Daten vom RM und vom TLS kombiniert ausgegeben werden, und die REST-API. Die benötigten Befehle für die Kommandozeile und deren Ausgaben sind in Anhang A, die für die REST-API benötigten URLs und deren Rückgaben in Anhang B gelistet. Auf beiden Wegen können u. A. die Daten zu folgenden Komponenten ausgegeben werden [10, 16–18]:

Anwendungen als nach dem Status gefilterte Liste oder der Report einer Anwendung
Ausführungen als Liste aller Ausführungen einer Anwendung oder der Report einer Ausführung
Container als Liste aller Container einer Ausführung oder der Report eines Containers
Nodes als Liste aller Nodes oder der Report eines Nodes

Zur Integration des Treibers wurden daher entsprechende Interfaces entwickelt, über die das Modell auf den eigentlichen Treiber zugreifen kann.

Die vier Interfaces `IApplicationResult`, `IAppAttemptResult`, `IContainerResult` und `INodeResult` dienen der Übergabe der geparsten Daten der einzelnen Komponenten an die korrespondierenden Komponenten im S#-Modell. Sie enthalten jeweils alle relevanten Daten, die von Hadoop über die Kommandozeile oder die REST-API ausgegeben werden. Alle vier Interfaces implementieren zudem `IParsedComponent`, wel-

ches wiederum als Basis für die Übergabe der ausgelesenen Daten an `IYarnReadable.SetStatus()` im Modell dient.

Das Interface `IHadoopParser` dient als Einbindung des Parsers im Modell mithilfe von `IYarnReadable.Parser` und enthält für jede der acht relevanten Ausgaben von Hadoop entsprechende Methodendefinitionen.

Beim Interface `IHadoopConnector`, das im Modell den Connector über die `FaultConnector`-Eigenschaften von `YarnApp` und `YarnNode` einbindet, besitzt ebenfalls für jede der acht Datenrückgaben entsprechende Deklarationen, für Ausführungen und Container dabei jeweils vom RM (NM für Container) und vom TLS. Auf die Nutzung des TLS zum Ermitteln der Daten zu Anwendungen wird verzichtet. Dies liegt darin begründet, dass bei Nutzung der REST-API des RM neben den vom TLS bereitgestellten Daten einige weitere Informationen zu den Anwendungen ausgegeben werden [10, 17]. Das Connector-Interface enthält darüber hinaus Deklarationen, um die im Modell implementierten Komponentenfehler im realen Cluster zu steuern und Anwendungen starten zu können. Architektonisch ist der Treiber zudem so aufgebaut, dass das Modell keine Kontrolle über den vom Parser benötigten Connector besitzt und die SSH-Verbindung ausschließlich vom Connector gesteuert werden kann.

4.2.2. Implementierte Parser

Da die Daten für die relevanten Komponenten auf zwei Arten ermittelt werden können und unterschiedliche Ausgaben erzeugen, wurden auch für beide Arten ein Parser (`CmdParser` und `RestParser`) entwickelt. Da der Parser von außerhalb keinerlei weitere Informationen erhält außer der ID der zu parsenden YARN-Komponente, ist der Parser selbst dafür verantwortlich, die Daten von einem korrespondierenden Connector zu erhalten. Daher muss zur Initialisierung eines Parsers zunächst der korrespondierende Connector initialisiert werden. Da für die Nutzung der REST-API zum Teil die IDs der übergeordneten YARN-Komponenten ebenfalls nötig sind, ist der `RestParser` zudem auch dafür verantwortlich, die entsprechenden IDs zu ermitteln, bei der Nutzung der Kommandozeile reichen aufgrund der Befehlsstruktur die IDs der Komponenten selbst.

Die konkreten Implementierungen der auf `IParsedComponent` basierenden Übergabe-Interfaces können ebenfalls als Bestandteil des Parsers angesehen werden. Sie wurden zudem so implementiert, dass sie für beide entwickelten Parser genutzt werden können.

Der grundlegende Ablauf ist bei jedem Parsing-Vorgang gleich. Zunächst werden, sofern benötigt, die benötigten YARN-Komponenten-IDs ermittelt und die Rohdaten mithilfe des Connectors von Hadoop abgefragt. Auch vom Parser wird dabei analog zum Modell das Abrufen der Daten ausschließlich mithilfe des Interfaces `IHadoopConnector` durchgeführt. Anschließend findet das eigentliche Parsing der Ausgabe von Hadoop statt, deren Daten direkt in der für die YARN-Komponente vorgesehene `IParsedCompo-`

`nent`-Implementierung gespeichert werden. Da Hadoop über die Kommandozeile die Daten in keinem standardisierten Format zurückgibt, wurde das Parsing der Rohdaten von Hadoop beim `CmdParser` in eigenem Code mithilfe von *Regular Expressions* realisiert. Bei der Nutzung der REST-API werden die Daten dagegen im JSON-Format zurückgegeben [10, 17, 18], wodurch diese mithilfe des *Json.NET*-Frameworks[1] deserialisiert und direkt als die entsprechende `IParsedComponent`-Implementierung gespeichert werden. Da RM und TLS verschiedene Daten einer YARN-Komponente ausgeben, werden, sofern nötig, RM und TLS abgefragt und die dabei ermittelten Daten zusammengeführt.

Eine erste Besonderheit bildet zudem das Abrufen und Parsen der Report-Daten mittels REST-API. Da die Listen hierbei als Array der einzelnen Reports zurückgegeben werden [10, 17, 18], wird beim Parsen eines Ausführungs- oder Container-Reports die komplette Liste abgerufen und geparst. Anschließend wird in dieser Liste basierend auf der ID die benötigte Komponente herausgefiltert.

Die zweite Besonderheit bei der Nutzung der REST-API liegt darin, dass die Daten zu derzeit ausgeführten Container ausschließlich vom NM, auf dem der Container ausgeführt wird, zurückgegeben werden können [17, 18]. Daher werden zur Ermittlung der Container-Listen alle Nodes abgefragt und anschließend die benötigten Container gefiltert.

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

4.2.3. Implementierte Connectoren

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

Da über die Kommandozeile die Befehle für die Daten vom TLS die gleichen wie für die Daten vom RM sind [10, 16], sind beim `CmdConnector` die TLS-Methoden von geringer Bedeutung und nutzen daher ebenfalls die RM-Methoden.

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

Der Connector ist beim Abrufen der Daten dafür zuständig, die dafür notwendigen Befehle auszuführen. Während dies für die Kommandozeilen-Befehle die entsprechenden Hadoop-Befehle sind, wird dies zum Abrufen der Daten über die REST-API mithilfe des Tools *curl* durchgeführt. Die dabei zurückgegebenen Daten werden vom Connector ohne Verarbeitung zurückgegeben und können dann vom Parser verarbeitet werden.

Beim Steuern der Komponentenfehler wird vom Connector das für die Fallstudie entwickelte Start-Script verwendet. Nach dem eigentlichen Start bzw. Aufheben eines Komponentenfehlers wird vom Connector zudem überprüft, ob die Injizierung bzw. Aufhebung erfolgreich war. Während der Datenabruf sowie die Steuerung der Komponentenfehler synchron stattfindet, findet das Starten der Anwendungen asynchron und mithilfe des Benchmark-Scriptes statt. Da eine Ausführung einer YARN-Anwendung längere Zeit in Anspruch nehmen kann, wird dadurch die Ausführung von S# nicht behindert und es können mehrere Anwendungen parallel ausgeführt werden.

4.2.4. SSH-Verbindung

Die SSH-Verbindung selbst ist der einzige Bestandteil des Treibers, welches kein entsprechendes Interface benötigt, die SSH-Verbindung wird ausschließlich vom Connector genutzt. Realisiert wird die Verbindung mithilfe des Frameworks SSH.NET,[2] weshalb die SSH-Verbindung im Treiber nur entsprechende Funktionen zum Aufbauen, Nutzen und Beenden der Verbindung enthält.

Um die Verbindung mit dem Cluster-PC aufzubauen, ist zudem ein dort installierter SSH-Key nötig. Ein Kommando auf dem Cluster-PC kann mithilfe der Treiberkomponente synchron und asynchron ausgeführt werden.

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

Literatur

[1] M. Polo u. a. „Test Automation“. In: *IEEE Software* 30.1 (Jan. 2013), S. 84–89. ISSN: 0740-7459. DOI: 10.1109/MS.2013.15.

[2] Orna Grumberg, EM Clarke und DA Peled. „Model checking“. In: (1999).

[3] A. Habermaier u. a. „Runtime Model-Based Safety Analysis of Self-Organizing Systems with S#“. In: *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. Sep. 2015, S. 128–133. DOI: 10.1109/SASOW.2015.26.

[4] Joost-Pieter Baier Christel; Katoen. *Principles of model checking*. Cambridge, MA: MIT Press, 2008. ISBN: 978-0-262-026499.

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

[6] Axel Habermaier, 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.

[7] Bo Zhang u. a. „Self-Balancing Job Parallelism and Throughput in Hadoop“. In: *16th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS)*. Hrsg. von Márk Jelasity und Evangelia Kalyvianaki. Bd. LNCS-9687. Distributed Applications and Interoperable Systems. Heraklion, Crete, Greece: Springer, Juni 2016, S. 129–143. DOI: 10.1007/978-3-319-39577-7_11. URL: https://hal.inria.fr/hal-01294834.

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

[9] 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).

[10] 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).

[11] 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).

[12] Apache Software Foundation. *MapReduce Tutorial*. 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html (besucht am 02. 01. 2018).

[13] Apache Software Foundation. *MapReduce NextGen aka YARN aka MRv2*. 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/index.html (besucht am 02. 01. 2018).

[14] Apache Software Foundation. *Hadoop: Capacity Scheduler*. 29. Juni 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html (besucht am 21. 01. 2018).

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

[16] 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).

[17] 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).

[18] 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).

A. Kommandozeilen-Befehle von Hadoop

Für jede der vier relevanten YARN-Komponenten können die Daten jeweils als Liste oder als ausführlicher Report ausgegeben werden. Im Folgenden sind Beispielhaft die dafür notwendigen Befehle für Anwendungen aufgelistet, für Ausführungen, Container und Nodes sind analoge Befehle verfügbar. Die Ausgaben zu den Befehlen sind hier zudem auf das wesentliche gekürzt, u. A. da Hadoop bei jedem Befehl zunächst ausgibt, über welche Services (in Listing A.1 z. B. TLS, RM und *Application History Server*) die Daten ermittelt werden. Weiterführende Informationen zu den einzelnen Befehlen sind in [16] zu finden.

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

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

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

```
$ yarn application --status application_1518100641776_0001
[...]
Application Report :
    Application-Id : application_1518100641776_0001
    Application-Name : QuasiMonteCarlo
    Application-Type : MAPREDUCE
    User : root
    Queue : default
    Start-Time : 1518103712160
    Finish-Time : 1518103799743
    Progress : 100%
    State : FINISHED
    Final-State : SUCCEEDED
    Tracking-URL : http://controller:19888/jobhistory/job/
       job_1518100641776_0001
    RPC Port : 41309
    AM Host : compute-1
    Aggregate Resource Allocation : 1075936 MB-seconds, 942 vcore-
       seconds
    Diagnostics :
```

B. REST-API von Hadoop

Wie bei der Ausgabe der Daten der YARN-Komponenten über die Kommandozeile können auch bei der Ausgabe mithilfe der REST-API die Daten als Liste oder als einzelner Report ausgegeben werden. Der Unterschied zur Kommandozeile liegt jedoch darin, dass die Listenausgaben einem Array der einzelnen Reports entsprechen. Neben der hier verwendeten Ausgabe im JSON-Format unterstützt Hadoop auch eine Ausgabe im XML-Format. Im Folgenden sind daher Beispielhaft die Ausgaben im JSON-Format für die Anwendungsliste vom RM und für Ausführungen vom TLS aufgeführt. Im Rahmen dieser Masterarbeit relevant waren vom RM die Rückgaben der Listen für Anwendungen, Ausführungen, Container (jedoch vom NM) und der Nodes. Vom TLS relevant waren die Listen für Ausführungen und Container. Weitere Informationen zu den hier verwendeten Nutzungsmöglichkeiten sind in [10, 17, 18] zu finden.

Listing B.1: REST-Ausgabe aller Anwendungen vom RM. Die Liste kann mithilfe verschiedener Query-Parameter gefiltert werden.
URL: `http://<rmhttpaddress: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 RM.
URL: `http://<rmhttpaddress:port>/ws/v1/cluster/apps/{appid}/appattempts`

```
{
  "appAttempts": {
    "appAttempt": [
      {
        "id": 1,
        "startTime": 1518430260521,
        "containerId": "
          container_1518429920717_0001_01_000001",
        "nodeHttpAddress": "compute-2:8042",
        "nodeId": "compute-2:45454",
        "logsLink": "//compute-2:8042/node/containerlogs/
          container_1518429920717_0001_01_000001/root"
      }
    ]
  }
}
```