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: 21. Januar 2018

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

Zusammenfassung

Abstract

Inhaltsverzeichnis

Abbildungsverzeichnis **IV**

Listingverzeichnis **V**

Abkürzungsverzeichnis **VI**

1 Einleitung **1**

2 Relevante Methodiken **2**
2.1 Model Checking 2
2.2 S# 4

3 Ablauf der Fallstudie **6**
3.1 Ablauf der Fallstudie 6
3.1.1 Aufbau des Modells 6
3.1.2 Mapping zum realen System 7
3.1.3 Erstellung der Lastprofile 7
3.1.4 Erstellen und Ausführen der Tests 8
3.1.5 Evaluierung der Ergebnisse 8
3.2 Apache Hadoop 8
3.3 Adaptive Komponente in Hadoop 11
3.4 Aufbau des realen Clusters 11

4 Aufbau des Modells **12**
4.1 Grundlegende Architektur 12
4.2 YARN-Modell 13
4.3 SSH-Treiber 15
4.3.1 Kommandozeilen-Parser 15
4.3.2 REST-API-Parser 15
4.3.3 Connector 15
4.3.4 SSH-Verbindung 15

Abbildungsverzeichnis

2.1 Schematischer Aufbau beim MC 3

3.1 Architektur von YARN 9
3.2 Architektur des HDFS 10

4.1 Grundlegende Architektur des Gesamtmodells 12
4.2 Aufbau des YARN-Modells 14

Listingverzeichnis

2.1 Grundlegender Aufbau einer S#-Komponente 4

Abkürzungsverzeichnis

AM	ApplicationManager
AppMstr	ApplicationMaster
DCCA	Deductive Cause-Consequence Analysis
MC	Model Checking
MC	Model Checker
NM	NodeManager
RM	ResourceManager

Kapitel 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 [**Polo2013**].

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 [**Grumberg1999**, **Habermaier2015**].

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/`

Kapitel 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 [**Grumberg1999**, **Baier2008**].

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.

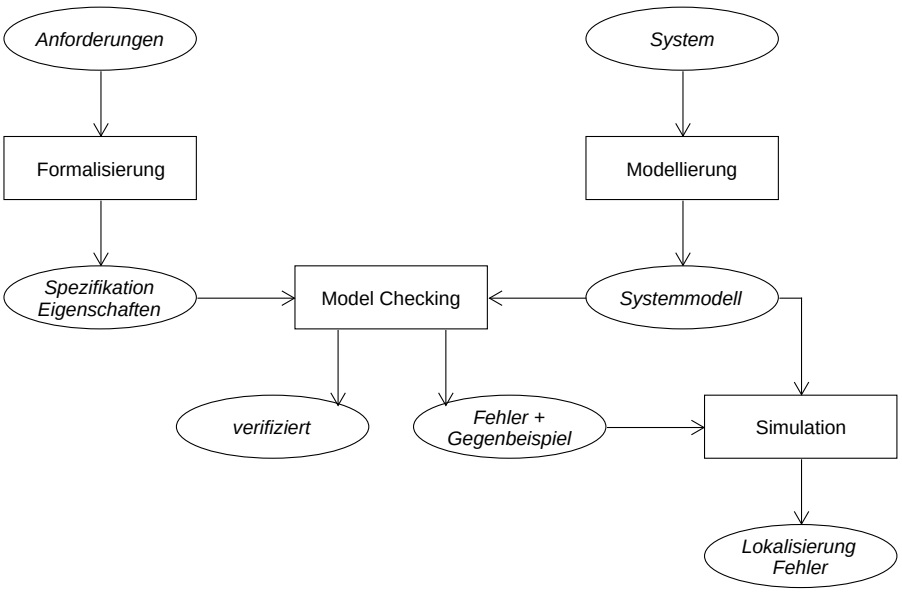

Abbildung 2.1: Schematischer Aufbau beim MC, nach [**Baier2008**]

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.

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 [**Baier2008**, **Eberhardinger2016**].

MC besitzt durch seine Charakteristik einige Vorteile, u. A. [**Baier2008**]:

- 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. [**Baier2008**]:

- 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.

```
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

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 Speziellen genutzt werden. S# vereint dabei die Simulation, die Visualisierung, modellbasierte Tests sowie das MC der Modelle [**Habermaier2015**, **Habermaier2016**]. 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 [**Habermaier2015**].

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 Hauptklas-

se (hier `YarnNode`) erbt und mithilfe des Attributs `FaultEffect` dem dazugehörigen Komponentenfehler zugeordnet wird [**Habermaier2016**].

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 werden, welche die Analyse beschleunigen und genauer machen können [**Eberhardinger2016**]. 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 [**Habermaier2015**].

Kapitel 3

Ablauf der Fallstudie

Da die Testautomatisierung bei einem verteilten, adaptiven System sehr aufwändig werden kann, soll nun ein modellbasierter Ansatz genutzt werden. Als reales System wurde Apache™Hadoop®[1] ausgewählt, welches mit einer adaptiven Komponente erweitert wurde. Von Hadoop sollen nun einige relevante Komponenten als Modell nachgebildet werden und anschließend mit einem realen Hadoop-Cluster verbunden werden. Dadurch soll der Ansatz des modellbasierten Testen auf ein reales System übertragen werden. Ebenfalls eine Rolle spielen einige Hauptprobleme wie z. B. die Einbindung des realen Systems in das Modell oder der Umgang mit asynchronen Prozessen bei verteilten Systemen.

3.1 Ablauf der Fallstudie

3.1.1 Aufbau des Modells

Zunächst muss natürlich erst einmal der Versuchsaufbau selbst in S# modelliert werden. Ein Modell beinhaltet in S# zunächst einmal die Komponenten des Systems und deren Zusammenhänge, also wie die Komponenten miteinander agieren. Wichtig sind in einem S#-Modell aber auch mögliche Komponentenfehler, welche bekannt sind und jederzeit auftreten können. Komponentenfehler werden bereits in der Designphase eines Modells eingearbeitet und können bei der späteren Ausführung flexibel aktiviert und deaktiviert werden, um die Probleme des zu testenden Systems zu ermitteln (vgl. Abschnitt 2.2).

Um nun Hadoop in S# zu modellieren wird zunächst ein Konzept erstellt, in dem ausgearbeitet wird, welche Komponenten und Komponentenfehler relevant sind. Anschließend müssen deren Zusammenhänge und wesentlichen Eigenschaften ausgearbeitet werden und in das Konzept übernommen werden. Sobald das Konzept fertig ausgearbeitet ist, kann das Modell in S# implementiert werden.

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

3.1.2 Mapping zum realen System

Nachdem die Basis des Modells steht, kann die Funktionalität entwickelt werden. Dazu werden in S# nur Basisfunktionen eingebaut, um mit dem realen System kommunizieren zu können. Um für das reale System möglichst viele Ressourcen zur Verfügung zu stellen, wird das reale System auf einem eigenen PC installiert. Die Verbindung zwischen Modell und realem System wird mit einer Art Treiber realisiert, welcher mithilfe von mehreren SSH-Verbindungen mit dem realen Hadoop-Cluster kommuniziert und so das Mapping zwischen Modell und realem System übernimmt. Jede der SSH-Verbindungen ist dabei nur für einen Einsatzzweck gedacht, sodass es Verbindungen für u. A. folgende Einsatzzwecke gibt:

- Starten von Benchmark-Anwendungen
- Monitoring des realen Cluster
- Injektion von Komponentenfehler

Der Vorteil von mehreren Verbindungen liegt darin, dass jede Verbindung unabhängig ist und nicht auf die Antwort des zuvor ausgeführten Befehls einer anderen Verbindung warten muss. So ist es möglich, mithilfe mehrerer Verbindungen mehrere Anwendungen parallel zu starten und jede Rückgabe auszuwerten und währenddessen verschiedene Komponentenfehler zu aktivieren.

3.1.3 Erstellung der Lastprofile

Sobald das Grundmodell steht, können die Testfälle selbst entwickelt werden. Als Testfälle dienen dazu unterschiedliche Lastprofile, um verschiedene Auslastungsgrade und Nutzungsszenarien zu simulieren. Dazu sollen die Lastprofile verschiedene Benchmarks beinhalten, deren einzelne Anwendungen kombiniert oder alleine auf dem realen System ausgeführt werden:

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

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 basiert auf einzelnen Bestandteilen der Mapreduce Examples. Die Examples wiederum sind zahlreiche, voneinander unabhängige Beispielanwendungen für Hadoop. Dadurch besteht die Möglichkeit, abhängig davon, welche Anwendungen einzeln oder parallel gestartet werden, unterschiedliche Profile zu simulieren. Daher muss zunächst auch geprüft werden, welcher Benchmark welche Möglichkeiten bietet, um die benötigten Testfälle bzw. Lastprofile zu erstellen und so den dynamischen Teil des zu testenden Modells zu erstellen.

3.1.4 Erstellen und Ausführen der Tests

Sobald Modell und Testfälle stehen, kann mit der Erstellung der Tests fortgefahren werden. Die Tests müssen nun so erstellt werden, dass sie sich einerseits auf veränderte Bedingungen des realen Clusters anpassen, aber auch automatisiert die einzelnen Anwendungen der Lastprofile aktivieren und ausführen. Dies schließt auch unterschiedliche Profile für die Aktivierung der Komponentenfehler ein. Zum einen kann nur eine Simulation ohne Fehler gestartet werden, zum anderen aber auch unterschiedliche Komponentenfehler aktiviert werden. Der MC von S# besitzt dazu auch Möglichkeiten, um Komponentenfehler kombiniert auszuführen. Dazu werden basierend auf zuvor definierte *Constraints* Komponentenfehler aktiviert, um so typische Probleme des realen Systems zu simulieren. Basierend darauf wird dann ermittelt, welche Fehler nun im realen System auftauchen.

3.1.5 Evaluierung der Ergebnisse

Je nachdem welche *Constraints* bei der Ausführung genutzt werden, sind nun unterschiedliche Fehler und Daten im realen System ermittelt worden, welche zum Abschluss evaluiert werden müssen. Einige Erwartungen sind da natürlich bereits im Voraus klar: Sollte es zu einem Netzwerk- oder Serverausfall eines Hadoop-Nodes kommen, muss das System dies selbstständig erkennen und die Anwendung an einen anderen Node abgeben. Dabei sollte das System auch erkennen, welche anderen Nodes bereits beschäftigt sind und entsprechend auf dem von Hadoop genutzten *Load Balancer* einen Node auswählen. Neben einer Fehleranalyse können aber auch die Laufzeiten unter bestimmten Bedingungen analysiert werden.

3.2 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 [**HadoopHomePage**]:

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

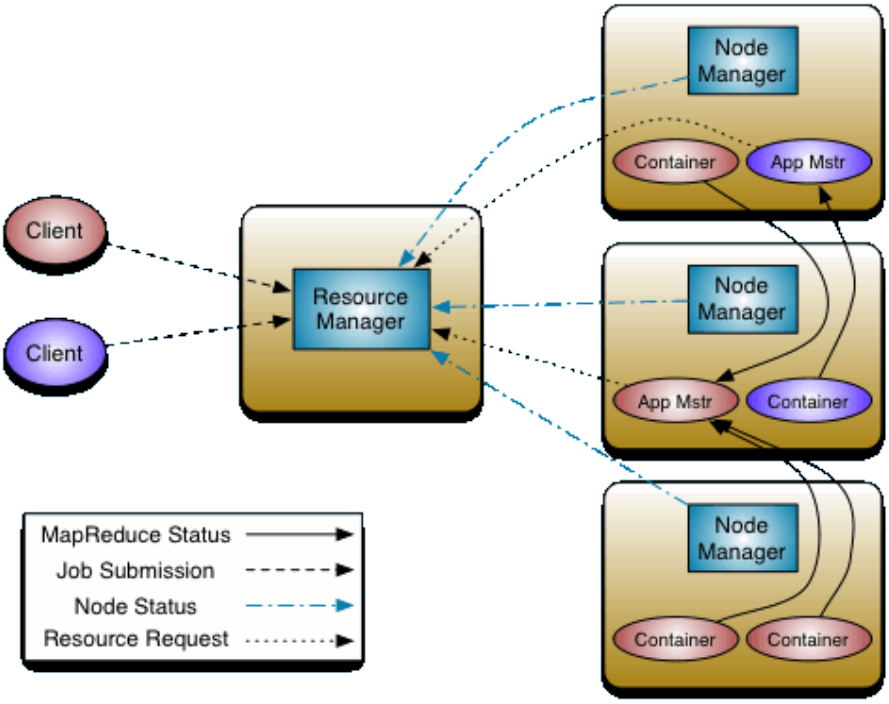

Abbildung 3.1: Architektur von YARN [**HadoopYarnArch271**]

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 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 zuständig ist und diese dem RM mitteilt. Jede Anwendung besitzt jeweils einen eigenen ApplicationMaster (AppMstr), welcher für das anwendungsbezogene Monitoring und die Kommunikation mit dem RM und NM zuständig ist und die dazu notwendigen Informationen bereit stellt. Jede YARN-Anwendung bzw. Job besteht aus mehreren *Containern*, in denen die einzelnen Tasks der Anwendung auf einem beliebigen Node ausgeführt werden. Auch der AppMstr wird dadurch in einem eigenen Container ausgeführt. Das Monitoring der einzelnen

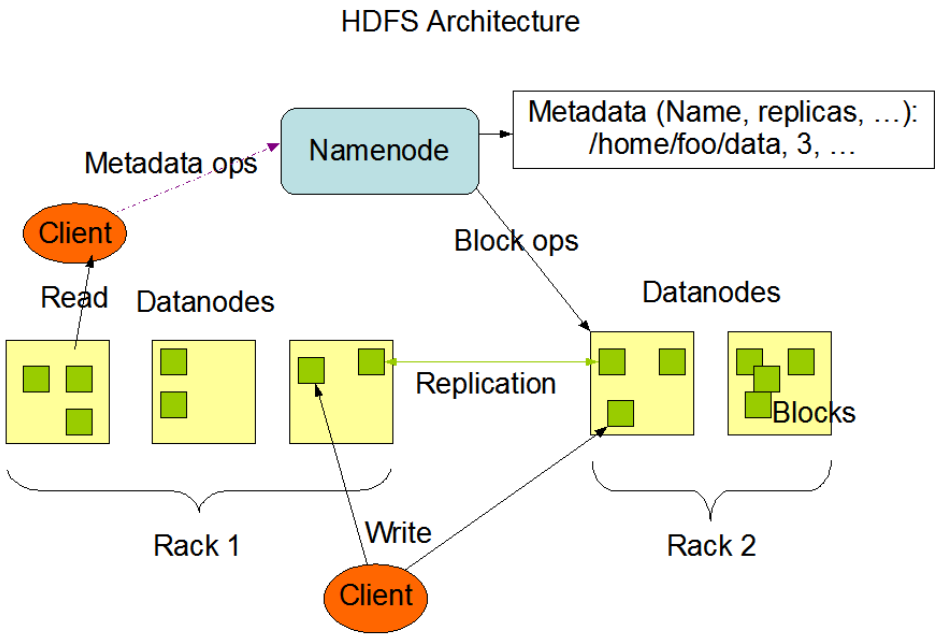

Abbildung 3.2: Architektur des HDFS [**HadoopHdfsDesc271**]

Container wird geteilt durchgeführt. Der NM übernimmt das Monitoring der genutzten Ressourcen, der jeweilige AppMstr die anwendungsbezogenen Monitoring-Aufgaben [**HadoopYarnArch271**].

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 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 [**HadoopHdfsDesc271**].

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 [**HadoopMapRedTutorial271**]. MapReduce kann man auch als Vorgänger von YARN ansehen, 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 [**HadoopYarnArch271**, **HadoopYarnOverview271**].

3.3 Adaptive Komponente in Hadoop

Ein normales Hadoop besitzt von sich aus keine adaptive Komponente, sondern rein statische Einstellungen. Um damit Hadoop zu optimieren, müssen die Einstellungen immer manuell auf den jeweils benötigten Anwendungstyp angepasst werden. Dazu gibt es auch bereits verschiedene Scheduler, den *Fair Scheduler*, welcher alle Anwendungen ausführt und ihnen gleich viele Ressourcen zuteilt, und den *Capacity Scheduler*. Letzterer sorgt dafür, dass nur eine bestimmte Anzahl an Anwendungen pro Benutzter gleichzeitig ausgeführt wird und teilt ihnen so viele Ressourcen zu, wie benötigt werden bzw. der Benutzter nutzen darf. Entwickelt wurde der Capacity Scheduler vor allem für Cluster, die von mehreren Organisationen gemeinsam verwendet werden und sicherstellen soll, dass jede Organisation eine Mindestmenge an Ressourcen zur Verfügung hat [**HadoopCapScheduler271**].

Je nach Bedarf besitzt der Capacity Scheduler entsprechende Einstellungen, um z. B. den verfügbaren Speicher pro Container festzulegen, auch *MARP* genannt. Da diese Einstellung direkt beeinflusst, wie viele Container und damit Anwendungen gleichzeitig ausgeführt werden können, ist das gesamte Cluster je nach Wert eher für kleine oder eher große Anwendungen effizient. Da der MARP-Wert jedoch nicht während der Laufzeit dynamisch angepasst werden kann, haben **zhang2016** in [**zhang2016**] 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.

, welche genutzt werden soll. Im Vergleich zur Standard-Einstellung von Hadoop benötigt diese mit einer selbstadaptiven Komponente ausgestatte Modifikation im Schnitt um bis zu 40 Prozent weniger Zeit zur Ausführung eines Tasks. Dazu wird der zur Verfügung stehende Arbeitsspeicher zur Laufzeit so eingeteilt, damit immer die maximal mögliche Anzahl an Tasks ausgeführt werden können.

3.4 Aufbau des realen Clusters

Kapitel 4

Aufbau des Modells

4.1 Grundlegende Architektur

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:

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 PC mit dem realen Cluster

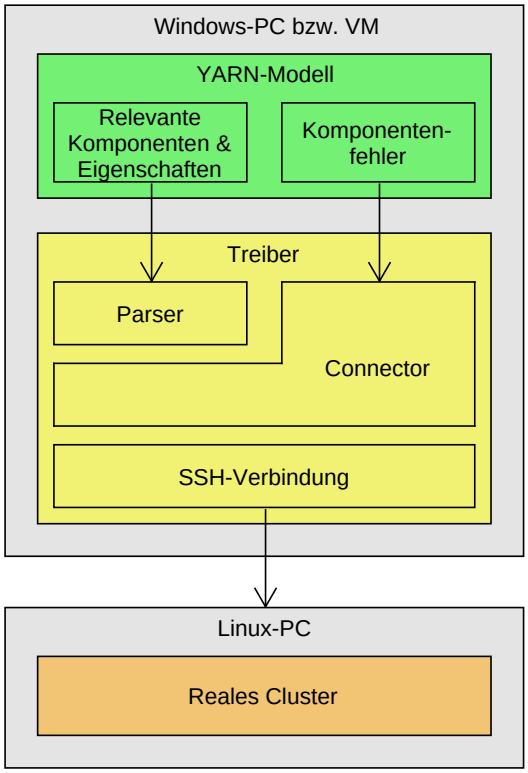

Abbildung 4.1: Grundlegende Architektur des Gesamtmodells

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

4.2 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 jeweils alle Daten aufgenommen, welche mithilfe von Shell-Kommandos bzw. mithilfe der REST-API von YARN ermittelt werden kann.

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 ausgeführt wurde oder die Ausführung noch nicht abgeschlossen ist (durch `EFinalStatus.UNDEFINED`).

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 eingereicht 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 `GetStatus()` implementiert, in welchem

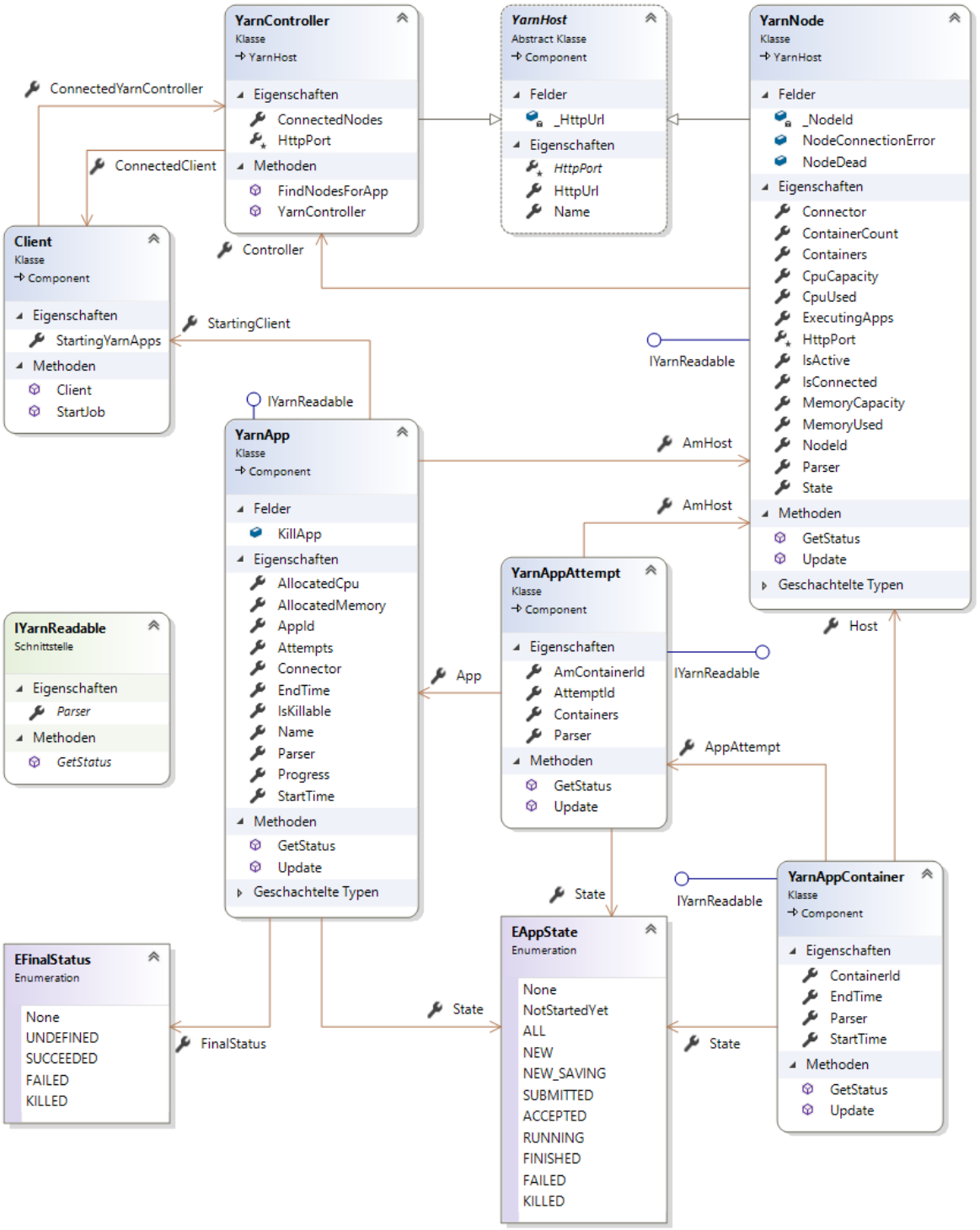

Abbildung 4.2: Aufbau des YARN-Modells. Das Modell wurde mithilfe des Klassendiagramm-Designers in Visual Studio 2017 erstellt. Daher werden Assoziationen aus Listen (wie `YarnApp.Attempts`) im Diagramm nicht angezeigt.

mithilfe des angegebenen Parsers auf den SSH-Treiber zugegriffen werden kann und die Komponenten im Modell so ihre Daten aus dem realen Cluster ermitteln können.

4.3 SSH-Treiber

In Abschnitt 4.1 wurde bereits auf den grundlegenden Aufbau des Treibers eingegangen. Der SSH-Treiber besteht aus drei einzelnen Komponenten, welche mithilfe von Interfaces im YARN-Modell eingebunden sind. Dadurch ist es möglich, unterschiedliche Parser bzw. auch Verbindungen für unterschiedliche Komponenten zu nutzen. Da es zwei Möglichkeiten gibt, die Daten des realen Clusters ausgeben zu lassen, wurde dies auch genutzt und so ein Parser für das Monitoring mittels Kommandozeilen-Befehle, und ein Parser für die Nutzung der REST-API von Hadoop erstellt.

4.3.1 Kommandozeilen-Parser

4.3.2 REST-API-Parser

4.3.3 Connector

4.3.4 SSH-Verbindung